Python (DFFML): Implementing AutoML

About Me

Name: Edison Siow (online pseudonym: seraphimstreets)

University: National University of Singapore

Degree: Computer Science (1st year)

Matriculation year: 2021

Expected graduation year: 2025

Phone Number: [redacted]

Email: [redacted]

Location: Singapore

Timezone: UTC+8

Linkedin: https://www.linkedin.com/in/edisonsiow/

Kaggle: https://www.kaggle.com/seraphimstreets

Proficient Programming Languages: Python, Javascript, Java, C++

Contributions

No.	PR Description	Link	Status
#1359	Created a H2O AutoML plugin for the DFFML library, capable of performing classification and regression. During the coding process, I familiarized myself greatly with DFFML's modelling, documentation and unit testing structures, which I believe will greatly accelerate the development process for this project.	https://github.com/intel/dffml/pull/1359	Awaiting approval.
#1367	Added missing GSOC rubric to documentation, in relation to issue #1343.	https://github.com/intel/dffml/pull/1367	Awaiting approval.

Project Information

Sub-organization name:

DFFML

Project Abstract:

Implement hyperparameter tuners, best model selection via grid search and or other means, automated feature engineering by modifying dataflows.

Detailed description (Issue #968):

AutoML or Automated Machine Learning as the name suggests automates the process of solving problems with Machine Learning. AutoML is generally helpful for people who aren't either familiar with Machine Learning or the involved programming. AutoML aims to improve the efficiency of any task involving Machine Learning.

The primary objective we are trying to achieve is to create a model that takes as a property of its config a set of models to used for hyperparameter tuning. Another property of its config is the set of models which we should attempt to tune (via the first set). Default values for these results in using all installed models to try to tune all installed model plugins.

To start, we should define a reduced set of models (not all the ones we have). We'll implement AutoML supporting only this reduced set. The first phase of this project will be to make sure that one model can be used to tune hyperparameters of another model.

The next phase will be to tune two models using the same tuning model. This followed by tuning two models, using two models which amounts to doing the previous task twice, with a different tuning model the second time.

The following phase will be to go through each model in each model plugin we have and see which ones have issues being tuned using the approach taken in the previous phase. This phase will help us determine which properties or methods we may need to add to models to help them self identify and thereby indicate their requirements for hyperparameter tuning, or maybe their inherent lack of support for it.

The final phase will be to implement hyperparameter tuning for N by N models, after implementing what we found to be gaps in the previous phase.

Due to the shortened GSoC cycle, we may end up not doing all of these phases. Which one we go to will be decided as we approach the selection process.

Approach Outline

Here, I will give a broad overview to the different aspects of my approach in this project. Note that this does not reflect the actual chronological order in which I will engage in these aspects, which I will describe in the Proposed Timeline section.

In this project, we are tasked to employ a set of different techniques to perform hyperparameter tuning over DFFML models. There have a been a broad range of different techniques utilized for hyperparameter tuning[1]. In the interest of keeping the project scope feasible, I will defer to the expertise of the authors of the paper *AutoML: A Survey of the State-of-the-Art* [2], who describe the most popular approaches to hyperparameter tuning for machine learning (ML) models.

Hyperparameter Optimization

In general, I will be using the dffml.tuner.Tuner class as the base for all the hyperparameter optimization classes. For this section, I will split it into two subsections: optimization techniques that work for all models, and those that only work for neural networks.

All Models

- Random search

Random search refers to the random sampling of hyperparameter combinations in the search space, and selecting the best one. For this, I will create a RandomSearch tuner in the dffml.tuner module. I will be using scikit-learn's RandomizedSearchCV() function, or do manual random sampling, at the request of my mentors.

- Grid search

Grid search refers to the systematic sampling of possibilities on a "grid" of hyperparameters. (may or may not be complete) For this, a parameter_grid tuner has already been written in the dffml library by my mentor, so I may not need to modify it. If I did, I could use scikit-learn's GridSearchCV().

- Bayesian optimization

Bayesian optimization (BO) is an approach to optimizing black-box functions with expensive-to-evaluate functions. It is well-suited for our purposes, since it works on general functions and training a model can be quite expensive. BO works by building a surrogate model to optimize the objective function, and typical choices for the surrogate model are Gaussian processes (GP), tree-structured Parzen estimators (TPE), random forest regression, and Bayesian neural networks.

Bayesian Optimization with GP is the most popular and is empirically more efficient than its counterparts for most cases [3], so I will prioritise implementing it. For this, I will create a BayesOptGP tuner in the dffml.tuner module, and will be using the BayesianOptimization (https://github.com/fmfn/BayesianOptimization) Python library.

However, it has been shown that Bayesian Optimization with GP scales poorly compared to BO with random forest regressors. (RF) [2] So, I will also implement BO with random forest regression using the SMAC library (https://github.com/automl/SMAC3) in the BayesOptRF tuner.

Using Bayesian networks (BN) as the surrogate model in BO has also shown results comparable to Gaussian processes[3] , and an open-source implementation is available in the Neural Network

Intelligence library (https://nni.readthedocs.io/en/latest/reference/hpo.html#dngo-tuner). I will implement this in a BayesOptBN tuner.

- Evolutionary algorithms

From what I've seen, evolutionary algorithms are not typically SOTA for most models [2], so this is lower priority. At the very least, I will implement the naïve evolutionary algorithm from Neural Network Intelligence since it is simple to do so. If there is time, I may implement Scipy's differential evolution (https://docs.scipy.org/doc/scipy/reference/generated/scipy.optimize.differential_evolution.html) using the approach described here (https://towardsdatascience.com/genetic-algorithm-to-optimize-machine-learning-hyperparameters-72bd6e2596fc)

Neural Networks Only (Optional)

- Architecture search

As deep learning exploded in popularity, so too did methods designed specifically to optimize them. For instance, one-shot methods such as DARTS [4] and ENAS [5] exploited the weight-sharing property of subgraphs to perform efficient search of neural architectures on the computational supergraph. Fortunately for us, the library Neural Network Intelligence (https://github.com/microsoft/nni) provides the requisite implementations of these neural-network specific techniques. They also have support for the major deep learning libraries (Pytorch, Tensorflow, Sklearn etc.) which is excellent for this project.

However, since these algorithms were designed for large neural networks that run on multiple GPUs, I foresee that there may be potential problems with GPU support. Therefore, at the request of my mentors, I have decided to make this a very low-priority add-on, which we will only consider if I am able to produce proof-of-concept code of it working with Tensorflow/Pytorch in the DFFML library during the pre-GSOC period.

If possible, though, I would be interested in implementing DARTS and ENAS tuners for DFFML, which will only be available for neural network Models.

ENAS: A reinforcement learning (RL) one-shot approach where a controller searches for an optimal subgraph within the computational supergraph.

DARTS: Addresses scalability by formulating the search space in a differentiable manner, allowing for search via gradient descent.

Additional Considerations

- LinearRegressionModel will not be supported by any tuners, since it has no hyperparameters.
- Autosklearn will not be supported by any tuners, since it is an autoML module in of itself.

Overview of tuners:

- RandomSearch
- GridSearch
- BayesOptGP
- BayesOptRF
- BayesOptBN

- DARTS (maybe, NN only)
- ENAS (maybe, NN only)

Overview of supported Models:

- All models except Linear regression and autosklearn

Benchmarking

Benchmarking is important for the third phase in the project description, where we will determine whether there are any issues with the above tuners for any of the DFFML models and exclude them as necessary. It will also be useful for documentation, where we may wish to display results to users and provide recommendations for usage. I will be benchmarking based on time and performance (accuracy/MLE etc.). Thus, I will test on one small/medium dataset for regression/classification. The tentative common datasets to be used are:

Classification (iris dataset,[1] Higgs Data set 10K points[2])

Regression (UCI ML housing dataset[3], Bike sharing dataset[4])

Image Classification (CIFAR-10[5])

I understand that DFFML is looking to expanding to time-series models. If needed, I can expand benchmarking to that as well.

Modifications to support tunable fields (hyperparameters)

In order to facilitate the Tuner class, we will need to modify the fields of the model config classes so that the tuner knows which are tunable. Currently, the construction of the dffml.base.field() method allows for the creation of a mutable field with the mutable=True argument. However, to allow for more flexibility, we should allow a field to be mutable but optionally tunable, depending on the needs of the user/internal application. Therefore, a simple solution is to add a tunable=True/False argument to the field, as a signal to the Tuner that it is a viable hyperparameter to tune. In terms of implementation, it would be similar to the implementation of mutable property in https://github.com/intel/dffml/pull/1122, where tunable will be False by default, and can only be made True by the specifications of the model implementer.

Users should be able to decide which fields they want to make tunable, to avoid the overhead of unnecessary training. Thus, we also add another property to all model configs, *tunable_hyperparameters*, which is a set of strings of tunable hyperparameters. Tunable hyperparameters defaults to a set of strings defined by the model-creator, but the user may provide a different set in the CLI command. If so, we will compare the two sets, and get their intersection (this is to avoid invalid user parameters). Then, during initialization of the config in *dffml.base._config*, all hyperparameter names in the resultant set will have their tunable property set to True. This will allow maximum flexibility for both end-users and model-creators.

---

[1] https://www.kaggle.com/datasets/uciml/iris
[2] https://archive.ics.uci.edu/ml/datasets/HIGGS
[3] https://archive.ics.uci.edu/ml/machine-learning-databases/housing/
[4] https://archive.ics.uci.edu/ml/datasets/Bike+Sharing+Dataset
[5] https://www.cs.toronto.edu/~kriz/cifar.html

For each tunable field, there will also be an additional (optional) field {*field_name}-range*, which will indicate to the Tuner what range of values to search for. The value of these field can either be a tuple of three numbers indicating (start, end. interval) for numeric fields, or a list of values (any type). During initialization of the config, a dictionary called *tunable_ranges* will be created with parameters and their respective ranges, which will be accessible by all Tuners via the model configuration. To avoid forcing the user to have to initialize values for every tunable hyperparameter, I will also be creating reasonable default values for these range_fields. These default values are to be determined by research and experimentation.

Finally, to allow users to access the functionality of tuners, I will be adding a high level CLI command (eg. "tune") which allows the user to tune their model with a given Tuner. Example usage is as follows:

```
python tune -model xgboost -tuner bayesGP -tunable-hyperparameters max-
depth max-depth-range 3 4 5 6 7 -model-features FEATURES -model-target
TARGET etc.
```

In terms of code for the tune CLI command, it will likely be similar to predict CLI command, but with the additional tuning step using the specified tuner.

Feature engineering by modifying Dataflows

It is generally accepted that data and features determine the upper bound of ML, and that models and algorithms can only approximate this limit. [2] Feature engineering is thus extremely important for optimizing the performance and efficiency of AutoML applications. Feature engineering can be split into 3-subtopics: selection, construction and extraction.

Feature Selection

Feature selection refers to the removal of redundant or irrelevant features. Here, we can utilize filter-based feature selection by performing statistical analysis and eliminating features with low correlation with the target value.

Based on whether the predictor/target variable is numerical/categorical, we will perform different statistical tests. The associated scikit-learn functions are listed in brackets.

Numerical input, numerical output: Pearson's Correlation Coefficient (f_regression())

Numerical input, categorical output: ANOVA (f_classif())

Categorical input, numerical output: ANOVA reversed (f_classif())

Categorical input, categorical output: Chi-squared test (chi2())

After performing the appropriate statistical test given the dataset, we will eliminate a subset of the features that has the lowest correlation. This can be done by keeping only the top K number of features, or alternatively by keeping the top k percentile of features. For simplicity's sake, we will only keep the top k percentile of features, where k is a user-defined parameter. We can do so using scikit-learn's SelectPercentile wrapper function, which takes in a statistical test and will output the reduced datatset once fit_transform() is called on it.

eg. data = SelectPercentile(chi2, percentile=70).fit_transform(X, y)

In terms of implementation, I will create a general Operation named featureSelection that allows for users to perform filter-based selection on a given dataset. The method signature may look something like this:

```
featureSelection(data, input_type, output_type, target_name,
select_percentile=”70”):
      """
      data: List[List[]] dataset
      input_type: str, either “numerical” or “categorical”,
      output_type: str, either “numerical” or “categorical”,
      target_name: str, name of the target variable,
      select_percentile int, The top % of these features are
      kept after statistical test.

      """
```

However, the assumption is that all the variables are of one type (categorical/numerical). We can create another operation getInputAndOutputType that checks whether the predictor variables can be cast into a homogenous type.

Additionally, we also have an operation called getInputAndOutputType that takes in a dataset with defined predictor and target variables, and returns the typing (categorical/numerical) for both. This will be converted to input_type and output_type in the featureSelection operation.

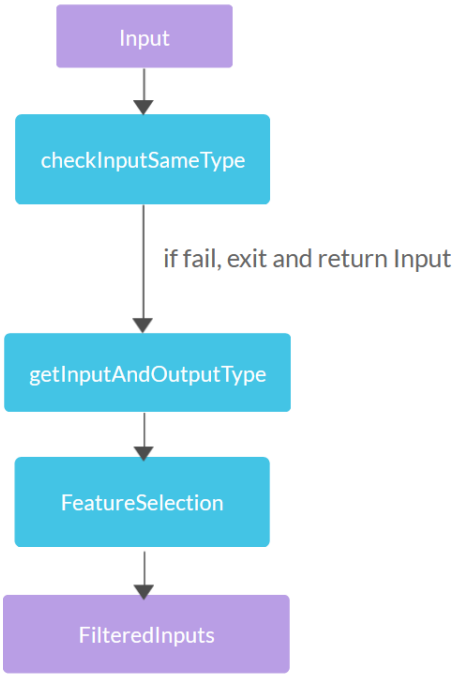


Figure 1 Dataflow for feature selection

Feature Construction

For feature construction, there exist standard techniques for generating new features (eg. min,max of feature, finding conjunctions and disjunctions of binary variables.) However, in my experience, these automated features are typically not very helpful and may even damage the models’ efficiency and

performance. The best new features are typically constructed by domain experts, so I do not foresee myself implementing operations for feature construction, although I am open to doing so if requested by my mentors.

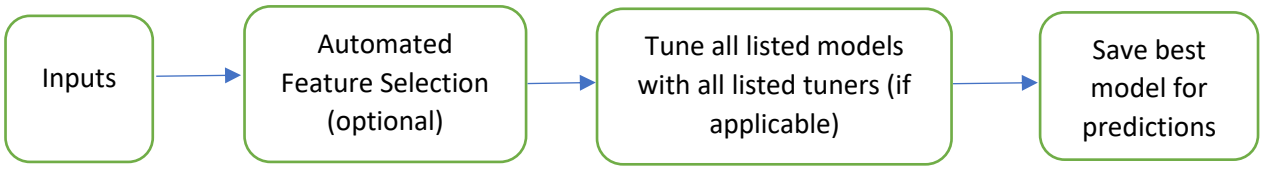
Feature Extraction

Feature extraction is a dimensionality reduction process which also alters the input data. On this front, the dffml_operations_data library already contains the principal_components_analysis operation, which is one of the most popular feature extraction techniques. Thus, I do not foresee the need to create any new operations, but again am open to implementing operations suggested by my mentors.

Creation of AutoML Model class

Since the final objective of the project is to integrate AutoML into the DFFML library, we must naturally create an AutoML Model. This model will be similarly structured to all the other DFFML models (being able to call train, predict and accuracy). The main difference will be the train() function, where we first perform automated data cleaning and feature selection for the user, and then perform the N by M tuning process for all N models and all M tuners as specified by the user, whenever applicable. (Default is all models/all tuners) The best model is then saved, and whenever the user calls the 'predict' or 'accuracy' function with automl as model, that model is loaded and used for predictions.

Here will be the general flow of the automl command:



Here is example usage of -model automl for train:

```
python train -model automl -features FEATURES -target TARGET -sources SOURCE -
source-filename FILENAME -scorer SCORER -models MODELS -tuners TUNERS -
feature_selection TRUE etc.
```

Adding documentation and unit tests

Once we have finished benchmarking, we will have a defined specification about which tuners support which models/plugins. Then, we will have to add additional unit tests for the "tune" and command to all DFFML models/plugins to ensure they integrate well with the new tuning modules..

Throughout the coding process, it is also important that I include internal documentation (with associated docstring tests) for the new modifications to dffml.base, the new AutoML Model, the Tuners, and the tuning command, so that it is more maintainable for future contributors/maintainers. I will do so in a modular fashion; ensuring that the documentation and tests are complete for a single function before moving on to the next.

I also plan to add example usages of the ‘tune’ CLI command in the Command Line/Tutorials section of the public documentation and usage of the AutoML model in the ‘Examples’ section. (or whichever sections my mentors deem most appropriate)

Post-hoc ensembles (Optional)

Ensembling in AutoML systems have shown to regularly outperform pipelines without ensembles [6]. Thus, it would be natural for us to consider incorporating ensemble models into the AutoML pipeline. These ensembles are to be done post-hoc, after all models have been trained/tuned using the MxN tuning method. The ensembles will also follow a greedy strategy, selecting models which are expected to increase the validation set accuracy when added. (in line with autosklearn’s implementation [6]) This should be simple to complete, given there is already a defined stacking pipeline for DFFML.
(https://intel.github.io/dffml/master/examples/notebooks/ensemble_by_stacking.html).

The main concern would be how to optimize for time/performance in the ensemble feature selection and training process, given the already lengthy autoML process. However, given my previous experiences with stacking, I am optimistic that I will be able to implement some form of ensemble modelling by the end of the project, which should outperform singular models. However, in the interest of keeping the project highly feasible, I will keep this as an optional add-on if time permits.

Proposed Timeline

Commitments

My summer vacation starts at the start of May 2022 and ends at 13th August 2022. I have no other commitments during this time and expect to work about 45 hours a week from 13th June (start of GSOC) to 9th Aug 2022. This is assuming 9 hours a day for 5 days a week, which was the norm during my previous internship at Defence Science Organization. This period of 8 weeks adds up to a cumulative total of 45*8 = 360 hours, which exceeds the expected time given in the DFFML projects index page.

Proposed Timeline

- Overview

(Phase 0) Till 20 June: Pre-GsoC Period	i. Familiarizing myself with DFFML codebase and relevant technologies ii. Contributing to DFFML
May 20 – June 12: Community Bonding Period	i. Plotting a roadmap with guidance of mentors ii. Alleviating any doubts iii. Preliminary coding to get a head start
June 13 – July 28: Coding Period 1	i. Implement AutoML with at least 3 general tuners and support for at least 5 DFFML models. ii. Implement the “tune” CLI method. iii. Implement feature engineering operations.
July 29	Phase 1 Evaluations
July 30 – Sept 4: Coding Period 2	i. Extend to all tuners and DFFML models (where applicable) ii. Implement the AutoML model iii. Finalizing documentation
Sept 5 – Sept 12 : Finalising code	Submission of final work product and final evaluation
Sept 13 – Sept 19	Mentors submit final evaluation
Sept 20	End of GSOC

- Detailed Timeline

Phase 0: Pre-GsoC Period (1 May – 19 May)

During this period, I will be familiarizing myself with the DFFML codebase via consistent contributions to DFFML and communication with my mentors via Gitter. I will also be participating in 2 Kaggle competitions with a friend, thus I will be sharpening my knowledge of feature engineering/ML frameworks during this period.

Phase 1: Community Bonding Period (20 May – 12 June)

With the guidance of my mentors, I will construct a roadmap for the project, and clarify any doubts. This is of course, in addition to bonding, which is what the period is for. I will also be studying the present work in DFFML and formulating a concrete plan to integrate this project into the codebase. I

also additionally hope to begin some preliminary coding at this step, to get a head-start on the project. Specifically, I hope to be able to tune a single model with a Tuner by the end of this period. (see below for details)

Phase 2: Coding Period 1 (13 June – 28 July)

1.5 weeks (13 June – 21 June)

I will finalise the functionality to tune a single model with a single Tuner, if not done so. The model will be XGBoost tentatively and the Tuner will be grid search. I will also implement the 'tune' high-level command to perform tuning.

2.5 weeks (22 June – 8 July)

I will create 2 additional tuners (random search, Bayesian optimization with GP) and extend support to Tensorflow, Pytorch, Scikit models. Scikit support may be time-consuming, since there are many models to come up with default values for. (I will also work on this during the pre-GSOC period.) While creating the tuners, I will simultaneously perform benchmarking for future documentation.

1 week (9 July – 15 July)

I will be adding the feature engineering operations, as described in the above section. Ideally, I will have added the operation as a contribution during the pre-GSOC/bonding period, allowing this week to act as buffer time.

2 weeks (16 July – 28 July)

I will be adding internal documentation and continuous integration tests for all available Tuners and supported plugins. I will also add an example usage of the 'tune' command in the Tutorials and Command Line section of the documentation.

Phase 3: GSOC Phase 1 Evaluation (29 July)

Phase 4: Coding Period 2 (July 30 – Sept 4)

2 weeks (30 July – 12 Aug)

I will extend the set of general tuners to support all other applicable plugins. If I have spare time, I will also attempt incorporating level 1 ensemble models into the AutoML pipeline.

1 week (13 Aug – 19 Aug)

I will finish implementing AutoML model, using the feature engineering/tuning modules currently defined.

2 weeks (20 Aug – Sept 4)

I will be finalizing documentation and continuous integration tests for all Tuners and supported plugins. I will also add an example usage of the AutoML model in the Examples section of the documentation. This is also buffer time to iron out bugs and perform code review.

Phase 5: Finalising Code (Sept 5 – Sept 12)

At this stage, my code should be complete and merged to master branch. This again effectively acts as a buffer week for any contingencies and for mentors' final evaluations.

Why Me

I am deeply passionate about programming, particularly in the area of machine learning/artificial intelligence. I have participated in numerous competitions on Kaggle, a popular ML competition site, attaining top 10% in the 2019 NFL Big Data Bowl and winning the Best Dataset prize worth US$3000 for the CDP Analytics Competition. On Coursera, I have finished several deep learning courses (Andrew Ng's deep learning specialization, Alberta University's RL specialization etc.). Additionally, I have completed many self-directed ML projects, including a stock price predictor, audio classifier and text classifier/generator. With 3 years of experience, I am very familiar with common machine learning frameworks (Tensorflow, Pytorch, Spacy etc.) and hyperparameter optimization techniques such as grid search and Bayesian optimization.

Beyond that, I also have a broad base of knowledge and experience in working on medium-sized projects. During my internship at Singapore's Defence Science Organization (DSO), I helped to develop a web-based application for pointcloud labelling (Django/ThreeJS) which incorporated an image segmentation model (Detectron) and pointcloud classification model (Rangenet). I also helped them to implement a novel paper on loop closure and benchmarked against current methods. Additionally, I had attained the AWS Certified Solutions Architect certification in 2020, and an Azure Data Scientist Associate certification last year, which included utilizing Azure's AutoML functionalities. In my first semester in NUS, I have performed well, averaging 4.5/5 GPA.

With my passion and experience, I believe I am well-suited to tackle this project within the allocated timeframe. I hope to be an asset for this project, DFFML, and the Python organization as a whole.

References

1. https://link.springer.com/content/pdf/10.1007%2F978-3-030-05318-5_1.pdf
2. https://arxiv.org/pdf/1908.00709.pdf
3. https://proceedings.neurips.cc/paper/2019/file/0668e20b3c9e9185b04b3d2a9dc8fa2d-Paper.pdf
4. https://arxiv.org/abs/1806.09055
5. https://arxiv.org/abs/1802.03268
6. https://arxiv.org/pdf/2007.04074.pdf