ATM Documentation

Release 0.0.1

Thomas Swearingen, Kalyan Veeramachaneni, Bennett Cyphers

Apr 25, 2019

Contents

Contents:

CHAPTER 1

ATM: Scalable model selection and tuning

Auto Tune Models (ATM) is an AutoML system designed with ease of use in mind. In short, you give ATM a classification problem and a dataset as a CSV file, and ATM will try to build the best model it can. ATM is based on a paper of the same name, and the project is part of the Human-Data Interaction (HDI) Project at MIT.

To download ATM and get started quickly, head over to the setup section.

1.1 Background

AutoML systems attempt to automate part or all of the machine learning pipeline, from data cleaning to feature extraction to model selection and tuning. ATM focuses on the last part of the machine-learning pipeline: model selection and hyperparameter tuning.

Machine learning algorithms typically have a number of parameters (called *hyperparameters*) that must be chosen in order to define their behavior. ATM performs an intelligent search over the space of classification algorithms and hyperparameters in order to find the best model for a given prediction problem. Essentially, you provide a dataset with features and labels, and ATM does the rest.

1.2 Our goal: flexibility and power

Nearly every part of ATM is configurable. For example, you can specify which machine-learning algorithms ATM should try, which metrics it computes (such as F1 score and ROC/AUC), and which method it uses to search through the space of hyperparameters (using another HDI Project library, BTB). You can also constrain ATM to find the best model within a limited amount of time or by training a limited amount of total models.

ATM can be used locally or on a cloud-computing cluster with AWS. Currently, ATM only works with classification problems, but the project is under active development. If you like the project and would like to help out, check out our guide to contributing!

CHAPTER 2

Setup

This page will guide you though downloading and installing ATM.

2.1 0. Requirements

Currently, ATM is only compatible with Python 2.7, 3.5 and 3.6 and *NIX systems.

We also recommend using virtualenv, which you can install as follows.:

```
$ sudo apt-get install python-pip
$ sudo pip install virtualenv
```

For development, also git is required in order to download and update the software.

2.2 1. Install ATM

2.2.1 Install using pip

The recommended way to install ATM is using pip inside a dedicated virtualenv:

```
$ virtualenv atm-env
$ . atm-env/bin/activate
(atm-env) $ pip install atm
```

2.2.2 Install from source

Alternatively, and for development, you can clone the repository and install it from source by running `make install`:

```
$ git clone https://github.com/hdi-project/atm.git
$ cd atm
$ virtualenv atm-env
$ . atm-env/bin/activate
(atm-env) $ make install
```

For development, replace the last command with `make install-develop` command in order to also install all the required dependencies for testing and linting.

Note: You will need to execute the command `. atm-env/bin/activate` to activate the virtualenv again every time you want to start working on ATM. You will know that your virtualenv has been activated if you can see the **(atm-env)** prefix on your prompt. If you do not, activate it again!

2.3 2. Install a database

ATM requires a SQL-like database to store information about datasets, dataruns, and classifiers. It's currently compatible with the SQLite3 and MySQL dialects. For first-time and casual users, we recommend installing SQLite:

```
$ sudo apt-get install sqlite3
```

If you're planning on running large, distributed, or performance-intensive jobs, you might prefer using MySQL. Run:

```
$ sudo apt-get install mysql-server mysql-client
```

and following the instructions.

No matter which you choose, you'll need to install the mysql client developer library in order for SQLAlchemy to work correctly:

```
$ sudo apt-get install libmysqlclient-dev
```

2.4 3. Start using ATM!

You're all set. Head over to the quick-start section to create and execute your first job with ATM.

CHAPTER 3

Quick-start guide

This page is a quick tutorial to help you get ATM up and running for the first time. We'll use a featurized dataset for a binary classification problem, already saved in `atm/data/test/pollution_1.csv`. This is one of the datasets available on openml.org. More information about the data can be found here.

Our goal is predict mortality using the metrics associated with the air pollution. Below we show a snapshot of the csv file. The dataset has 15 features, all numeric, and and a binary label column called "class".

```
PREC    JANT    JULT    OVR65   POPN    EDUC    HOUS    DENS    NONW    WWDRK   POOR
↪  HC      NOX     SO@     HUMID   class
35      23      72      11.1    3.14    11      78.8    4281    3.5     50.7    14.4
↪  8       10      39      57      1
44      29      74      10.4    3.21    9.8     81.6    4260    0.8     39.4    12.4
↪  6       6       33      54      1
47      45      79      6.5     3.41    11.1    77.5    3125    27.1    50.2    20.6
↪  18      8       24      56      1
43      35      77      7.6     3.44    9.6     84.6    6441    24.4    43.7    14.3
↪  43      38      206     55      1
53      45      80      7.7     3.45    10.2    66.8    3325    38.5    43.1    25.5
↪  30      32      72      54      1
43      30      74      10.9    3.23    12.1    83.9    4679    3.5     49.2    11.3
↪  21      32      62      56      0
45      30      73      9.3     3.29    10.6    86      2140    5.3     40.4    10.5
↪  6       4       4       56      0
..      ..      ..      ...     ....    ....    ...     ....    ..      ....    ....
↪  ..      ..      ..      ..      .
..      ..      ..      ...     ....    ....    ...     ....    ..      ....    ....
↪  ..      ..      ..      ..      .
..      ..      ..      ...     ....    ....    ...     ....    ..      ....    ....
↪  ..      ..      ..      ..      .
37      31      75      8       3.26    11.9    78.4    4259    13.1    49.6    13.9
↪  23      9       15      58      1
35      46      85      7.1     3.22    11.8    79.9    1441    14.8    51.2    16.1
↪  1       1       1       54      0
```

3.1 Create a datarun

Before we can train any classifiers, we need to create a datarun. In ATM, a datarun is a single logical machine learning task. The `enter_data.py` script will set up everything you need.:

```
(atm-env) $ atm enter_data
```

The first time you run it, the above command will create a ModelHub database, a dataset, and a datarun. If you run it without any arguments, it will load configuration from the default values defined in `atm/config.py`. By default, it will create a new SQLite3 database at ./atm.db, create a new dataset instance which refers to the data at `atm/data/test/pollution_1.csv`, and create a datarun instance which points to that dataset.

The command should produce output that looks something like this::

```
method logreg has 6 hyperpartitions
method dt has 2 hyperpartitions
method knn has 24 hyperpartitions
Data entry complete. Summary:
                                Dataset ID: 1
                                Training data: /home/bcyphers/work/fl/atm/atm/data/
↪test/pollution_1.csv
                                Test data: None
                                Datarun ID: 1
                                Hyperpartition selection strategy: uniform
                                Parameter tuning strategy: uniform
                                Budget: 100 (classifier)
```

The datarun you just created will train classifiers using the "logreg" (logistic regression), "dt" (decision tree), and "knn" (k nearest neighbors) methods. It is using the "uniform" strategy for both hyperpartition selection and parameter tuning, meaning it will choose parameters uniformly at random. It has a budget of 100 classifiers, meaning it will train and test 100 models before completing. More info about what is stored in the database, and what the fields of the datarun control, can be found here.

The most important piece of information is the datarun ID. You'll need to reference that when you want to actually compute on the datarun.

3.2 Execute the datarun

An ATM *worker* is a process that connects to a ModelHub, asks it what dataruns need to be worked on, and trains and tests classifiers until all the work is done. To run one, use the following command:

```
(atm-env) $ atm worker.py
```

This will start a process that builds classifiers, tests them, and saves them to the ./models/ directory. As it runs, it should print output indicating which hyperparameters are being tested, the performance of each classifier it builds, and the best overall performance so far. One round of training looks like this:

```
Computing on datarun 1
Selector: <class 'btb.selection.uniform.Uniform'>
Tuner: <class 'btb.tuning.uniform.Uniform'>
Chose parameters for method "knn":
        _scale = True
        algorithm = brute
        metric = euclidean
```

(continues on next page)

(continued from previous page)

```
        n_neighbors = 8
        weights = distance
        Judgment metric (f1, cv): 0.813 +- 0.081
New best score! Previous best (classifier 24): 0.807 +- 0.284
Saving model in: models/pollution_1-62233d75.model
Saving metrics in: metrics/pollution_1-62233d75.metric
Saved classifier 63.
```

And that's it! You're executing your first datarun, traversing the vast space of hyperparameters to find the absolute best model for your problem. You can break out of the worker with Ctrl+C and restart it with the same command; it will pick up right where it left off. You can also run the command simultaneously in different terminals to parallelize the work – all workers will refer to the same ModelHub database.

Occassionally, a worker will encounter an error in the process of building and testing a classifier. Don't worry: when this happens, the worker will print error data to the terminal, log the error in the database, and move on to the next classifier.

When all 100 classifiers in your budget have been built, the datarun is finished! All workers will exit gracefully.

```
Classifier budget has run out!
Datarun 1 has ended.
No dataruns found. Exiting.
```

You can then load the best classifier from the datarun and use it to make predictions on new datapoints.

```
>>> from atm.database import Database
>>> db = Database(dialect='sqlite', database='atm.db')
>>> model = db.load_model(classifier_id=110)
>>> import pandas as pd
>>> data = pd.read_csv('atm/data/test/pollution_1.csv')
>>> model.predict(data[0])
```

CHAPTER 4

Guide to the ModelHub database

The ModelHub database is what ATM uses to save state about ongoing jobs, datasets, and previously-generated models. It allows multiple workers on multiple machines to collaborate on a single task, regardless of failures or interruptions. The ideas behind ModelHub are described in the corresponding paper, although the structure described there does not match up one-to-one with the ModelHub implemented in `atm/database.py`. This page gives a brief overview of the structure of the ModelHub database as implemented and how it compares to the version in the paper.

4.1 Datasets

A Dataset represents a single set of data which can be used to train and test models by ATM. The table stores information about the location of the data as well as metadata to help with analysis.

- `dataset_id` (Int): Unique identifier for the dataset.
- `name` (String): Identifier string for a classification technique.
- **`description` (String): Human-readable description of the dataset.**
  - not described in the paper
- `train_path` (String): Location of the dataset train file.
- `test_path` (String): Location of the dataset test file.
- `class_column` (String): Name of the class label column.

The metadata fields below are not described in the paper.

- `n_examples` (Int): Number of samples (rows) in the dataset.
- `k_classes` (Int): Number of classes in the dataset.
- `d_features` (Int): Number of features in the dataset.
- `majority` (Number): Ratio of the number of samples in the largest class to the number of samples in all other classes.
- `size_kb` (Int): Approximate size of the dataset in KB.

4.2 Dataruns

A Datarun is a single logical job for ATM to complete. The Dataruns table contains a reference to a dataset, configuration for ATM and BTB, and state information.

- `datarun_id` (Int): Unique identifier for the datarun.
- `dataset_id` (Int): ID of the dataset associated with this datarun.
- **`description` (String): Human-readable description of the datarun.**
  - not in the paper

BTB configuration:

- **`selector` (String): Selection technique for hyperpartitions.**
  - called "hyperpartition_selection_scheme" in the paper
- `k_window` (Int): The number of previous classifiers the selector will consider, for selection techniques that set a limit of the number of historical runs to use.
  - called "t_s" in the paper
- `tuner` (String): The technique that BTB will use to choose new continuous hyperparameters.
  - called "hyperparameters_tuning_scheme" in the paper
- `r_minimum` (Int): The number of random runs that must be performed in each hyperpartition before allowing Bayesian optimization to select parameters.
- `gridding` (Int): If this value is set to a positive integer, each numeric hyperparameter will be chosen from a set of `gridding` discrete, evenly-spaced values. If set to 0 or NULL, values will be chosen from the full, continuous space of possibilities.
  - not in the paper

ATM configuration:

- `priority` (Int): Run priority for the datarun. If multiple unfinished dataruns are in the ModelHub at once, workers will process higher-priority runs first.
- `budget_type` (Enum): One of ["learner", "walltime"]. If this is "learner", only `budget` classifiers will be trained; if "walltime", classifiers will only be trained for `budget` minutes total.
- `budget` (Int): The maximum number of classifiers to build, or the maximum amount of time to train classifiers (in minutes).
  - called "budget_amount" in the paper
- `deadline` (DateTime): If provided, and if `budget_type` is set to "walltime", the datarun will run until this absolute time. This overrides the `budget` column.
  - not in the paper
- `metric` (String): The metric by which to score each classifier for comparison purposes. Can be one of ["accuracy", "cohen_kappa", "f1", "roc_auc", "ap", "mcc"] for binary problems, or ["accuracy", "rank_accuracy", "cohen_kappa", "f1_micro", "f1_macro", "roc_auc_micro", "roc_auc_macro"] for multiclass problems
  - not in the paper
- `score_target` (Enum): One of ["cv", "test", "mu_sigma"]. Determines how the final comparative metric (the *judgment metric*) is calculated.
  - "cv" (cross-validation): the judgment metric is the average of a 5-fold cross-validation test.

  - "test": the judgment metric is computed on the test data.
  - "mu_sigma": the judgment metric is the lower error bound on the mean CV score.
  - not in the paper

State information:

- `start_time` (DateTime): Time the DataRun began.
- `end_time` (DateTime): Time the DataRun was completed.
- `status` (Enum): Indicates whether the run is pending, in progress, or has been finished. One of ["pending", "running", "complete"].
  - not in the paper

4.3 Hyperpartitions

A Hyperpartition is a fixed set of categorical hyperparameters which defines a space of numeric hyperparameters that can be explored by a tuner. ATM uses BTB selectors to choose among hyperpartitions during a run. Each hyperpartition instance must be associated with a single datarun; the performance of a hyperpartition in a previous datarun is assumed to have no bearing on its performance in the future.

- `hyperparition_id` (Int): Unique identifier for the hyperparition.
- `datarun_id` (Int): ID of the datarun associated with this hyperpartition.
- `method` (String): Code for, or path to a JSON file describing, this hyperpartition's classification method (e.g. "svm", "knn").
- `categoricals` (Base64-encoded object): List of categorical hyperparameters whose values are fixed to define this hyperpartition.
  - called "partition_hyperparameter_values" in the paper
- `tunables` (Base64-encoded object): List of continuous hyperparameters which are free; their values must be selected by a Tuner.
  - called "conditional_hyperparameters" in the paper
- `constants` (Base64-encoded object): List of categorical or continuous parameters whose values are always fixed. These do not define the hyperpartition, but their values must be passed to the classification method to fully parameterize it.
  - not in the paper
- `status` (Enum): Indicates whether the hyperpartition has caused too many classifiers to error, or whether the grid for this partition has been fully explored. One of ["incomplete", "gridding_done", "errored"].
  - not in the paper

4.4 Classifiers

A Classifier represents a single train/test run using a method and a set of hyperparameters with a particular dataset.

- `classifier_id` (Int): Unique identifier for the classifier.
- `datarun_id` (Int): ID of the datarun associated with this classifier.
- `hyperpartition_id` (Int): ID of the hyperpartition associated with this classifier.

- `host` (String): IP address or name of the host machine where the classifier was tested.
  - not in the paper
- `model_location` (String): Path to the serialized model object for this classifier.
- `metrics_location` (String): Path to the full set of metrics computed during testing.
- `cv_judgment_metric` (Number): Mean of the judgement metrics from the cross-validated training data.
- `cv_judgment_metric_stdev` (Number): Standard deviation of the cross-validation test.
- `test_judgment_metric` (Number): Judgment metric computed on the test data.
- `hyperparameters_values` (Base64-encoded object): The full set of hyperparameter values used to create this classifier.
- `start_time` (DateTime): Time that a worker started working on the classifier.
- `end_time` (DateTime): Time that a worker finished working on the classifier.
- `status` (Enum): One of [“running”, “errored”, “complete”].
- `error_message` (String): If this classifier encountered an error, this is the Python stack trace from the caught exception.

CHAPTER 5

Contributing to ATM and BTB

5.1 Ways to contribute

ATM is a research project under active development, and there's a *ton* of work to do. To get started helping out, you can browse the issues page on Github and look for issues tagged with "help wanted" or "good first issue." An easy first pull request might flesh out the documentation for a confusing feature or just fix a typo. You can also file an issue to report a bug, suggest a feature, or ask a question.

If you're looking to make a more in-depth contribution, check out our guides on adding a classification method and adding a BTB Tuner or Selector.

5.2 Requirements

If you'd like to contribute code or documentation, to have installed the project in development mode.

5.3 Style

We try to stick to the Google style guide where possible. We also use flake8 (for Python best practices) and isort (for organizing imports) to enforce general consistency.

To check if your code passes a style sanity check, run `make lint` from the main directory.

5.4 Tests

We currently have a limited (for now!) suite of unit tests that ensure at least most of ATM is working correctly. You can run the tests locally with `pytest` (which will use your local python environment) or `tox` (which will create a new one from scratch); All tests should pass for every commit on master – this means you'll have to update the code

in `atm/tests/unit_tests` if you modify the way anything works. In addition, you should create new tests for any new features or functionalities you add. See the pytest documentation and the existing tests for more information.

All unit and integration tests are run automatically for each pull request and each commit on master with CircleCI. We won't merge anything that doesn't pass all the tests and style checks.

5.5 Docs

All documentation source files are in the `docs/source/` directory. To build the docs after you've made a change, run `make html` from the `docs/` directory; the compiled HTML files will be in `docs/build/`.

CHAPTER 6

Adding a classification method

ATM includes several classification methods out of the box, but it's possible to add custom ones too.

From 10,000 feet, a "method" in ATM comprises the following:

1. A Python class which defines a fit-predict interface;
2. A set of *hyperparameters* that are (or may be) passed to the class's constructor, and the range of values that each hyperparameter may take;
3. A *conditional parameter tree* that defines how hyperparameters depend on one another; and
4. A JSON file in `atm/methods/` that describes all of the above.

6.1 1. Valid method classes

Every method must be implemented by a python class that has the following instance methods:

1) `fit`: accepts training data and labels (X and y) and trains a predictive model.
2) `predict`: accepts a matrix of unlabeled feature vectors (X) and returns predictions for the corresponding labels (y).

This follows the convention used by scikit-learn, and most of the classifier methods already included with ATM are `sklearn` classes. However, any custom python class that implements the fit/predict interface can be used with ATM.

Once you have a class, you need to configure the relevant hyperparameters and tell ATM about your class.

6.2 2. Creating the JSON file

All configuration for a classification method must be described in a json file with the following format:

```
{
    "name": "bnb",
    "class": "sklearn.naive_bayes.BernoulliNB",
    "hyperparameters": {...},
    "root_hyperparameters": [...],
    "conditions": {...}
}
```

- "name" is a short string (or "code") which ATM uses to refer to the method.
- "class" is an import path to the class which Python can interpret.
- "hyperparameters" is a list of hyperparameters which ATM will attempt to tune.

6.2.1 Defining hyperparameters

Most parameter definitions have two fields: "type" and either "range" or "values". The "type" is one of ["float", "float_exp", "float_cat", "int", "int_exp", "int_cat", "string", "bool"]. Types ending in "_cat" are categorical types, and those ending in "_exp" are exponential types.

- If the type is ordinal or continuous (e.g. "int" or "float"), "range" defines the upper and lower bound on possible values for the parameter. Ranges are inclusive: [0.0, 1.0] includes both 0.0 and 1.0.
- If the type is categorical (e.g. "string" or "float_cat"), "values" defines the list of all possible values for the parameter.

Example categorical types:

```
"nu": {
    "type": "float_cat",
    "values": [0.5, 1.5, 3.5]  // will select one of the listed values
}

"kernel": {
    "type": "string",
    "values": ["constant", "rbf", "matern"]  // will select one of the listed strings
}
```

Example (uniform) numeric type:

```
"max_depth": {
    "type": "int",
    "range": [2, 10]   // will select integer values uniformly at random between 2␣
↪and 10, inclusive
}
```

Example exponential numeric type:

```
"length_scale": {
    "type": "float_exp",
    "range": [1e-5, 1e5]  // will select floating-point values from an exponential␣
↪distribution between 10^-5 and 10^5, inclusive
}
```

6.2.2 Defining the Conditional Parameter Tree

There are two kinds of hyperparameters: *root hyperparameters* (also referred to as "method hyperparameters" in the paper) and *conditional parameters*. Root parameters must be passed to the method class's constructor no matter what, and conditional parameters are only passed if specific values for other parameters are set. For example, the GaussianProcessClassifier configuration has a single root parameter: `kernel`. This must be set no matter what. Depending on how it's set, other parameters might need to be set as well. The format for conditions is as follows:

```
{
    "root_parameter_name": {
        "value1": ["conditional_parameter_name", ...],
        "value2": ["other_conditional_parameter_name", ...]
    }
}
```

In `gaussian_process.json`, there are three sets of parameters which are conditioned on the value of the root parameter `kernel`:

```
"root_parameters": ["kernel"],

"conditions": {
    "kernel": {
        "matern": ["nu"],
        "rational_quadratic": ["length_scale", "alpha"],
        "exp_sine_squared": ["length_scale", "periodicity"]
    }
}
```

If `kernel` is set to "matern", it means `nu` must also be set. If it's set to "rational_quadratic" instead, `length_scale` and `alpha` must be set instead. Conditions can overlap – for instance, `length_scale` must be set if kernel is either "rational_quadratic" or "exp_sine_squared", so it's included in both conditional lists. The only constraint is that any parameter which is set as a result of a condition (i.e. a conditional parameter) must not be listed in "root_parameters".

The example above defines a conditional parameter tree that looks something like this:

```
kernel-----------------------
|        \                    \
matern    rational_quadratic  exp_sine_squared
|         |           |       |              |
nu      length_scale  alpha   length_scale  periodicity
```

6.3 3. (Optional) Adding a new method to the ATM library

We are always looking for new methods to add to ATM's core! If your method is implemented as part of a publicly-available Python library which is compatible with ATM's other dependencies, you can submit it for permanent inclusion in the library.

Save a copy of your configuration json in the `atm/methods/` directory. Then, in in the `METHODS_MAP` dictionary in `atm/constants.py`, enter a mapping from a short string representing your method's name to the name of its json file. For example, `'dt': 'decision_tree.json'`. If necessary, add the library where your method lives to `requirements.txt`.

Test out your method with `python scripts/test_method.py --method <your_method_code>`. If all hyperpartitions run error-free, you're probably good to go. Commit your changes to a separate branch, then open up a pull request in the main repository. Explain why your method is a useful addition to ATM, and we'll merge it in if we agree!

CHAPTER 7

Adding a BTB Selector or Tuner

BTB is the metamodeling library and framework at the core of ATM. It defines two general abstractions:

1. A *selector* chooses one of a discrete set of possibilities based on historical performance data for each choice. ATM uses a selector before training each classifier to choose which hyperpartition to try next.
2. A *tuner* generates a metamodel which tries to predict the score that a set of numeric hyperparameters will achieve, and can generate a set of hyperparameters which are likely to do well based on that model. After ATM has chosen a hyperpartition, it uses a tuner to choose a new set of hyperparameters within the hyperpartition's scope.

Like with methods, ATM allows domain experts and tinkerers to build their own selectors and tuners. At a high level, you just need to define a subclass of `btb.Selector` or `btb.Tuner` in a new python file and create a new datarun with the 'selector' or 'tuner' set to "path/to/your_file.py:YourClassName".

More to come... stay tuned!