Tool Comparison: Optuna vs Hyperopt

Audrey Bertin, Nasko Apostolov, Nelson Evbarunegbe, & Raymond Li

Introduction

In this project, we compare three different options for the step of hyperparameter tuning within a Python-based machine learning pipeline: 1) Grid Search (which is a standard tool built into machine learning modules in Python, and which represents our control condition), and two external libraries, 2) Hyperopt, and 3) Optuna, which users might apply to a project if they want more control over their tuning process or want to try out a more advanced or efficient algorithm for hyperparamter tuning.

For our comparison, we start with a code review of both external tools based on the code available in their public repositories. We then conduct an experimental comparison where we train and tune a machine learning model using Scikit-Learn in combination with both Hyperopt and Optuna and compare their performance to a standard grid search. As part of this experiment, we generate the UI components provided by Hyperopt and Optuna, and provide a UI review.

Code Review:

Hyperopt (GitHub)

In this section, we cover several characteristics of the GitHub repository that are not code specific, as well as non-functional requirements (functional requirements and UI review will be discussed in the experimental comparison). For each category, we rate the repository on a scale of 1-10. Note that all of the information in this section is as of March 2023, and may have changed since then.

Popularity/Use `(Score: 8/10)`

Hyperopt has over 6.6K stars, 1K forks, and 9.2K active GitHub users. 90% of repositories have < 5K stars, putting Hyperopt in the top 10%. It clearly is having significant impact and influence in the field of hyperparameter tuning.

Community `(Score: 7/10)`

Hyperopt has a reasonably active community with 90 separate contributors. There are 16 open pull requests from 13 separate users. This indicates that users *other* than those who originally created the project are helping maintain and improve the code, which can be helpful in keeping long-term projects updated.

Maintenance `(Score: 4/10)`

The last commit on the main branch is from Nov 29, 2021, nearly 1.5 years ago. The latest official release is from around the same time period (Nov 17, 2021). Contrary to best practice, the `dev` branch is *not* updated compared to the `main` branch. It is 75 commits behind, when standard practice would be for the `dev` branch to be *ahead*(for testing out new features).

There are several pull requests that have been sitting in the repository for a long time, including one as old as August 2020. Similarly, there are 375 open issues with a few dozen being **more than a decade old**!! This indicates a somewhat poor level of maintenance.

Issues are getting buried. Pull requests are left opened instead of closed (if not going to be put into use/merged). Additionally, the longer that packages go without an update, the higher the likelihood for bugs to be introduced as dependencies and Python versions update and change. In the world of software, 1.5 years is a long time to not update.

Dependencies `(Score: 8/10)`

Hyperopt has **8** direct dependencies, which is a relatively small number, especially for the scale of the project. The low number of dependencies helps keep the project stable by reducing the chance that a dependency could have vulnerabilities or that an update to a could introduce unexpected functionality-breaking bugs.

Security `(Score: 9/10)`

A Snyk analysis found 0 direct vulnerabilities in the code and 3 indirect vulnerabilities, all associated with software dependencies. These indirect vulnerabilities were classified as **Low** severity. Overall, this indicates that Hyperopt is quite secure.

Understandability `(Score:5/10)`

Hyperopt contains a basic README. It provides setup instructions, a simple example, and instructions for those interested in contributing to the software.

However, this README is certainly not perfect. Consider the example code provided below. There is a comment describing this code chunk, but it is not necessarily obvious to a new user *what* a search space is. There is no definition provided along with this code sample (and no direct link to the documentation page explaining it), so just reading the sample for the first time is a bit confusing.

```
# define a search space
from hyperopt import hp
space = hp.choice('a',
    [
        ('case 1', 1 + hp.lognormal('c1', 0, 1)),
        ('case 2', hp.uniform('c2', -10, 10))
    ])
```

The organization of the documentation is somewhat confusing as well. It appears to be hosted in three separate places: a GitHub wiki, an official separate documentation website, and several Jupyter notebooks written into the repository under a folder called `Tutorial`. This leads to a very confusing system. Several links on the README point to functional wiki pages (e.g. Basic tutorial, Installation notes, Using mongodb). However, other links on the README point to pages that are now deprecated, e.g. one part of the README says "See projects using hyperopt on the wiki" but this does not link to the current documentation for this information. This indicates that the README is not fully up to date. The README also completely fails to mention the existence of the Jupyter notebooks, and does not explain what the different folders in the repository store either (which can be helpful to those wanting to understand the code).

The official documentation website is better than the README for providing key information. It has several pages that expand significantly on terms like "search space" and "minimizing function" that are discussed in the README with full-length walkthrough pages for each. However, it is still somewhat limited. There is only one basic example of how to use the code, which is very minimal.

On the code side of things, understandability is not much better. The organization for the repository is not particularly good. Everything is stored in a folder called `hyperopt` , which has dozens of files stacked together in the same folder, rather than being separated out by purpose. For example, we see some of the following files:

```
graph_viz.py, plotting.py, pyll_utils.py, utils.py
```

These are all in the same folder, but seem to have different purposes. Some are basic utilities that help with modeling, while others have to do solely with visualization. Separating these into separate folders by purpose would make it more clear which files go together.

Function documentation is okay, but definitely not perfect. There are several classes in the code and these seem to be documented in detail using an auto-documentation tool (see the docstring in the example method below):

```
def shrinking(self, label):
        """Return fraction of original search width
        Parameters
        ----------
        label: string
            the name of a hyperparameter
        """
        T = len(self.node_vals[label])
        return old_div(1.0, (1.0 + T * self.shrink_coef))
```

This auto-documentation does not actually appear to be running though, as there is no indication on the README about a site where each method is documented, even though the code is already written in the format to support that.

Additionally, some functions are not documented at all. For example, the following, from `utils.py`:

```
def get_most_recent_inds(obj):
    data = numpy.rec.array(
        [(x["_id"], int(x["version"])) for x in obj], names=["_id", "version"]
    )
    s = data.argsort(order=["_id", "version"])
    data = data[s]
    recent = (data["_id"][1:] != data["_id"][:-1]).nonzero()[0]
    recent = numpy.append(recent, [len(data) - 1])
    return s[recent]
```

Every function should have at least some amount of documentation, which is certainly not the case here.

Overall, minimal documentation exists but leaves a lot to be desired.

Testability/Debuggability `(Score: 7/10)`

In its GitHub repository, `hyperopt` has a `/tests` folder which contains both **integration** and **unit** tests. These tests all appear to be written using Python's built in `unittest` library, which is the most popular testing framework in Python. Some (but not all) of the tests use `setUp`/`tearDown` methods to configure the environment for testing, which is considered best practice when applicable.

There is a clear lack of documentation across the tests. Most have *no* comments whatsoever describing what they are testing.

The integration tests cover the two main extensions that `hyperopt` describes on their documentation site: `mongodb` and `spark`. These tests appear to be quite extensive and cover a variety of different conditions. For example, consider the following list of tests written for the `spark` integration: `test_timeout_without_job_cancellation, test_timeout_without_job_cancellation_fmin_timeout, test_timeout_with_job_cancellation, test_invalid_timeout.`

These are four separate tests all looking for the same general behavior (timeout), but there is a version custom-generated for each possible kind of timeout situation. This indicates that the test authors wanted to ensure that no cases were missed.

For the unit tests, there are 17 test files which each test a different component of the tool. This clear separation helps make clear which tests are available for which features.

Most of these tests look pretty standard and effective. However, there are some strange features in a few of the tests. First, some of the tests don't actually include any assert statements to check their behavior. This makes some sense for tests such as those that generate plots (it is hard to test an image to ensure that it is showing the correct thing), but it does not make sense for all tests.

There also appear to be some tests that get *skipped*. `test_plotting.py` has a *try* statement in the imports, and skips the test if there is an issue with `matplotlib`. Having a built in feature to skip a test is not good practice. If the authors know why this `matplotlib` error happens, which thy appear to, it should be possible to write the test in such a way as to get around it.

```
try:
    import matplotlib
    matplotlib.use("svg")  # -- prevents trying to connect to X server
except ImportError:
    import nose
    raise nose.SkipTest()
```

Zooming out and looking at the code itself, rather than just the tests, overall the code appears to be written in a way that promotes testability. Many of the functions in `hyperopt` use `try/except` pairs to raise errors for a variety of behaviors. Some of the functions even include `assert` statements to check that something is true *within* the original function (and not just in the corresponding test):

```
def idxs_take(idxs, vals, which):
    """
    Return `vals[which]` where `which` is a subset of `idxs`
    """
    assert len(idxs) == len(vals)
    table = dict(list(zip(idxs, vals)))
    return np.asarray([table[w] for w in which])
```

These are sometimes self explanatory but sometimes not, and the code appears to be mostly lacking any sort of comments explaining when these exceptions are expected to appear, which makes testing more difficult for unfamiliar users.

On a more positive note, the code is highly modular. Different components are separated and most functions are shorter than around 20 lines. Very short, separated functions makes it significantly easier to debug and trace where an error came from–particularly combined with the frequent exception statements.

Extensibility/Scalability `(Score: 5/10)`

Etensibility and scalability are associated with how easy it is to expand a tool–either to more users or to a more production type environment (rather than a testing/development one), or by adding in new features.

Hyperopt's major limitation in terms of scalability is the fact that parallel processing is not built in. This means the tool can not handle large scale problems out of the box. If the user wanted to test 100 different combinations of parameters, Hyperopt would have to run them one by one, which is a *very* slow process. In order to be used at scale, a hyperparameter tuning tool needs this feature. Although not built in, it *is* possible to integrate with Spark to do this. This can be limiting for some production uses, though, as it would require a Spark cluster to be available and installed in the environment, which could potentially be time consuming and expensive to set up.

For use on a production scale, Hyperopt's built-in visualization features are lacking as well. In the tool itself, there are just three functions that the user can call to visualize results, rather than a full dashboard that integrates with the tool.

Once again, there is an integration that can help with this. Hyperopt results can be displayed with MLFlow, a platform that provides a dashboard/UI to support end-to-end machine learning workflows. MLFlow tools can be put into production through deployments to MLFlow's server or to another external Cloud tool. Again, though, this UI is not built in and requires the user to be familiar with the MLFlow system.

In terms of extensibility, the first major feature promoting this is of course that the code is hosted on GitHub. This makes it much easier to extend, as anyone interested in creating their own version of the tool could fork off the main repository and add on new features. The tool also uses classes throughout in order to manage its different algorithms. This makes the code easier to extend–users need only add methods to the corresponding class.

However, the lack of documentation and clear organization makes extending the tool more difficult. It is sometimes hard to tell what each function is doing and what features are really available, which is necessary in order to understand what could potentially be added.

Optuna (GitHub)

Popularity/Use `(Score: 8/10)`

Optuna has over 7.8K stars, 813 forks, and 6.6K active GitHub users. 90% of repositories have < 5K stars, putting Optuna in the top 10%. It clearly is having significant impact and influence in the field of hyperparameter tuning. This growth is particularly impressive compared with Hyperopt when considering that usage is relatively similar between the two tools but Hyperopt has been around much longer.

Community `(Score: 9/10)`

Optuna has a very active community with 199 separate contributors, more than double that of Hyperopt. There are 19 open pull requests from 14 separate users, and nearly 3000 pull requests already closed. This indicates that users *other* than those who originally created the project are regularly helping maintain and improve the code, which can be helpful in keeping long-term projects updated. The high number of contributors also means many different perspectives are being considered in the code, which often leads to a better product.

Maintenance (Score: 9/10)

The last commit on the main branch was made *within the last day.* The latest official release was put out around two months prior to writing this (Jan 18, 2023).

There is a separate branch for each major release, making it simple to find the corresponding code. The oldest pull request is less than 6 months old, with quite a few being very new (opened within the last day or two). There are 99 issues open, with the oldest from August 2020. Over 1200 issues are closed. Overall this indicates a very high level of maintenance.

Issues tend to be getting closed in a reasonable amount of time in most cases, though 99 open is still a semi-large number to keep track of and can lead to old issues getting buried. Pull requests seem to mostly be processed quickly and have lots of engagement in the comments. Commits are being created regularly, as are new releases. This helps keep the code up to date and reduces the number of bugs.

Dependencies (Score: 5/10)

Optuna has **28** direct dependencies, more than three times the number of Hyperopt. This is quite a few dependencies, which increases the risk of potential security vulnerabilities or function-breaking bugs.

Security (Score: 9/10)

A Snyk analysis found 0 direct vulnerabilities in the code and 3 indirect vulnerabilities, all associated with software dependencies. These indirect vulnerabilities were classified as **Low** severity. Overall, this indicates that even though it has many dependencies, Optuna is quite secure, just like Hyperopt.

Understandability (Score: 9/10)

Like Hyperopt, Optuna's repository has a basic README with a quick code sample, information on installation, and additional features that can be integrated. The code sample provided for Optuna is well documented. It defines terms that are used in the framework ("study" and "trial"), and the example code even has some step by step comments.

```
study = optuna.create_study()  # Create a new study.
study.optimize(objective, n_trials=100)  # Invoke optimization of the
# objective function.
```

Examples like this were one area Hyperopt fell quite short. The official documentation had only one example, and a link to additional examples in the README directed to a deprecated page. Optuna, on the other hand, has an **entire repository** dedicated to examples, in addition to the example on the README. This repository is linked in the README.

It contains full examples written out in pretty much *every* Python machine learning library out there, including `keras`, `pytorch`, `tensorflow`, and `sklearn`. Each example is well documented and contains a unique set of requirements telling the user which software versions they need installed to run the example. This is about as well documented of an example guide as you can create!

Optuna's official documentation site is also excellent. It includes a link to a YouTube overview of the tool and a walkthrough of key features of Optuna compared with other hyperparamter optimizers. It also has a full API reference guide with documentation of different classes and functions. Finally, it provides an extensive FAQ page providing detailed answers (including code snippets) to common questions.

The documentation is very easy to find and well organized. All sites linked in the README are also up to date.

The organization of the repository is also quite clear. Each subfolder (that is not obvious in purpose solely by its name, such as "tests") contains its *own* readme, describing its purpose.

Folders are also relatively clearly named and there is a high degree of separation. Folders are broken down into subfolders where similar pieces of code are stored together. For example, in the benchmarks folder, there are separate folders for each benchmark technique used for measuring performance.

There are also other nice features. For example, the authors have a full guide for contributing to the project and they also provide a dockerfile that can be used in projects wanting to run Optuna using docker.

Function documentation is also quite good. Most functions have a full docstring, particularly if they are user facing.

However, not all functions have one. Some back end functions that aren't provided to the user are missing docstrings. This could make it more difficult for open source contributors to understand these functions. Additionally, some functions have smaller line-by-line type comments, but these are relatively minimal.

Testability/Debuggability (Score: 9/10)

Similar to hyperopt, optuna contains both integration tests and unit tests. The tests in Optuna are written using `pytest` instead of `unittest`. Both frameworks are excellent and effective.

The tests are separated out into separate folders in the repository, with each category of unit test getting its own folder (one for visualization, one for trials, one for search spaces, etc.). There are also a few additional tests that cover things that hyperopt did not cover, such as a set of tests to check whether importing all the requirements for optuna works properly and another that checks whether some built in logging functionality is working properly.

The `integration_tests/` folder is very extensive and contains a test for essentially every possible Python machine learning library that users could use Optuna with: `keras`, `mlflow`, `pytorch`, `sklearn`, `tensorflow`, `xgboost`, etc.

The unit tests also appear extensive. One file (`study_tests/test_study.py`), which is just one of dozens of test files, testing *one small aspect of the code,* contains over 1,500 lines of test code. This code appears to test every possible state of a trial.

This wide level of code coverage is actually verifiable by the user, since Optuna uses a `codecov` plugin on the repository. `codecov` is a tool that checks test coverage *every time* a new commit is added to the repository. Optuna's `codecov` page is linked here.

This tool measures in terms of line coverage. Optuna has **90.79%** coverage, and that coverage appears to be trending upward, having increased by 1% in the last 3 months. We can actually see how much *each* individual file and folder in the code is covered. The file with the least coverage appears to be `cli.py` (testing the command line interface), which makes sense. It is entirely possible that some code in the command line interface is simply not testable via an automated test checking system, as this requires someone executing commands via the command line (rather than solely running python). On the other end, some files and folders have 100% coverage as well.

Similar to Hyperopt, the code for Optuna is written in a way that promotes testability and debuggability. Functions tend to be very short and modular. Many contain assertions and error raises.

In addition, one excellent feature provided by the authors of Optuna is that some of the more complex functions in the tool actually have *examples* written in their doc strings, which give users a piece of code that should work for that function, helping make the expected use of each function much easier to understand.

```
def stop(self) -> None:
        """
        ... {rest of docstring above}

        Example:
            .. testcode::
                import optuna
                def objective(trial):
                    if trial.number == 4:
                        trial.study.stop()
                    x = trial.suggest_float("x", 0, 10)
                    return x**2
                study = optuna.create_study()
                study.optimize(objective, n_trials=10)
                assert len(study.trials) == 5
        """
```

Overall the test suite appears to be well written and extensive, though we only see metrics on line coverage (and not branch coverage, which typically requires more tests). It is possible that some improvements could still be made.

Extensibility/Scalability `(Score: 9/10)`

Optuna solves some of the issues with scalability faced by Hyperopt. For one, parallel computing is built into Optuna. Anyone who wants to use it can apply parallel processing out of the box, without the need for integration with other tools. This means it is already designed to run well at scale and handle large processing requests.

Optuna also has a built in UI dashboard. The repository storing its code is linked here. This dashboard is super simple to use. The user need only define a location to store the results of the study/parameter tuning process, run `pip install optuna-dashboard` and then run a single line of code to get the dashboard running. It appears like you can set which IP address the dashboard gets hosted on, allowing for customization.

The option to specify a storage location lets users set up storage integration with production-ready platforms, such as SQL databases.

One additional convenient feature for production deployments is the inclusion of docker images. The creators of Optuna have made official docker images for the tool and its UI dashboard, which users can use to install and run the tool in a standardized format. For example, users can install the dashboard image as follows:

```
$ docker pull ghcr.io/optuna/optuna-dashboard:latest
```

The code is also written with a high level of extensibility. Just like Hyperopt, all of the code is publicly available on GitHub–both for the original tool itself and for additional features like the dashboard–making it easy to contribute.

In addition, Optuna provides lots of extra resources to make contribution easier. For example, there is a full Developer's Guide provided on GitHub for **both** the original Optuna repository **and** the dashboard repository. For example, this is how the guide on the original repository starts:

Contribution Guidelines

It's an honor to have you on board!

We are proud of this project and have been working to make it great since day one. We believe you will love it, and we know there's room for improvement. We want to

- implement features that make what you want to do possible and/or easy.
- write more tutorials and examples that help you get familiar with Optuna.
- make issues and pull requests on GitHub fruitful.
- have more conversations and discussions on GitHub Discussions.

We need your help and everything about Optuna you have in your mind pushes this project forward. Join Us!

If you feel like giving a hand, here are some ways:

The authors already note which areas particularly need more work (e.g. extending tutorials and examples, improving documentation) and then later in the document provide an extensive overview of the steps for contributing to the code. This is excellent and more detailed than the contribution

pages on most other repositories. For those wanting to extend the package, it provides an excellent overview to get them started and helps maintain a consistent process and workflow.

Similar to Hyperopt, Optuna also uses a lot of classes throughout its code, which again can simplify the process of adding new features (one need only add a method to the class that the new feature corresponds to).

Optuna's excellent documentation helps make this process easy as well. Almost all functions and classes contain extensive details about their inputs, outputs, methods, and expected behavior. This makes it easy for developers to see which features are present and which ones may be missing.

Some functions can be a little long, however, and some utility functions are not fully documented, so there is still room for a little bit of improvement on the extensibiility side.

Summary

In the table below, we summarize the results of the code review in terms of each tool's scores by category.

Table 1: Code Review Scores by Category

Metric	Hyperopt	Optuna
Popularity	8	8
Community	7	9
Maintenance	4	9
Dependencies	8	5
Security	9	9
Understandability	5	9
Testability/Debuggability	7	9
Extensibility/Scalability	5	9
TOTAL (/80)	53	67

As we can see, Optuna excelled overall, with 67 points out of a possible 80, while Hyperopt trailed behind. Optuna's strengths primarily came in terms of its active community and regular maintenance, and high degree of understandability, testability/debuggability and extensibility/scalability. Hyperopt and Optuna both performed excellently in terms of security, with no direct vulnerabilities and only a couple weak, low severity ones. Hyperopt did out-score Optuna in one category as well: dependencies. Hyperopt has *less than half* the number of dependencies Optuna has, which helps protect it from bugs.

Optuna's high level of performance in this code review is not particularly surprising. It is currently considered one of the top hyperparameter tuning tools, and is often praised for its easy of use, understandability, and advanced features. Hyperopt *used* to be the gold standard, but its peak era was a decade ago, so it appears to be later in its lifecycle, and is potentially receiving less care and attention as it once did as a result.

Experimental Comparison

In addition to comparing the code, we conducted an experiment to compare Hyperopt and Optuna – both to each other and to the more basic parameter tuning method *Grid Search*. For each of our three conditions (Grid Search, Hyoperopt, Optuna), we ran two separate machine learning pipelines, one for a classification problem and one for a regression problem.

In order to keep the results consistent, we maintained the following across our trials:

- **The train/test split** – We use a random seed to split the data into training and testing sets *exactly* the same way for each condition so that we are comparing 1:1.
- **Predictors** – We used the exact same set of predictor columns to predict our outcome variable in each dataset (all columns of the data other than the target). Since the train test split was identical, this means an exact data copy/set of predictor and target values across all conditions.
- **Hyperparameters.** We tuned the same set of three hyperparameters in all of our tests: `n_estimators`, `max_depth`, `max_features`.
- **Hyperparameter ranges.** We used the same range of hyperparameters for each tool as well. For grid search, we provided the following grid: `max_depth`: [5, 7, 10, 15], `max_features`: [3, 5, 8, 10], `n_estimators`: [50, 150, 200, 300]. For Hyperopt and Optuna, rather than pre-selecting values, you provide a range. We provided the following integer ranges, so that they exactly overlapped: `max_depth`: 5-15, `max_features`: 3-10, `n_estimators`: 50-300.
- **Metrics.** Within the classification and regression conditions, we used the same metrics to score each tool. Hyperopt and Optuna also require us to select a specific metric to maximize or minimize. We used the same metric for each tool.
- **Computer.** To keep computing power consistent and make tuning time meaningful, each experiment was executed on the same laptop with the same settings.

For each trial, we completed the same steps:

1. Read in data and split into train and test.
2. Run the specified tuning algorithm/tool, calculating tuning time in the process.
3. If there is a UI component of the tuning tool, run the code needed to create this.
4. Take the optimal parameters returned by the tuning step and retrain the model using these parameters.
5. Use this model to return final performance metrics (described in the sections below).

Classification

For the classification problem, we used data on hospital readmissions, predicting whether someone will be readmitted to the hospital based on data from their last stay. To do this, we used the `RandomForestClassifier` model from `scikit-learn`. A snapshot of some of this data can be seen below:

time_in_hospital	n_lab_procedures	n_procedures	n_medications	n_outpatient	n_inpatient	n_emergency	change	diabetes_med	readmitted	age_decade	high_glucose	high_A1C
8	72	1	18	2	0	0	0	1	0	70	0	0
3	34	2	13	0	0	0	0	1	0	70	0	0

In order to compare the three tools, we used four different metrics:

1. **Accuracy** – How often the model makes the right prediction overall
2. **Precision** – For positive predictions, how many were *actually* positive? (catches false positives)
3. **Recall** – How many true positives we capture through labeling them positive? (catches false negatives)
4. **Tuning Time** – How long the algorithm takes to run and pick the best parameters

Accuracy, precision, and recall are measured as decimal values between 0 and 1. *Higher* values (closer to 1) mean better performance. Tuning time is measured in minutes and seconds, and *lower* values mean a more efficient algorithm/better performance.

For our three tools, we saw the following results for these metrics. Note that Hyperopt and Optuna were both defined to maximize *Accuracy*.

Table 2: Classification Experiment Metrics By Tuning Tool

Metric	Grid Search	Hyperopt	Optuna
Accuracy	0.628	0.627	0.629
Precision	0.625	0.630	0.632
Recall	0.497	0.479	0.482
Tuning Time	9 min, 24 sec	3 min, 35 sec	3 min, 25 sec

Grid search was by far the slowest tool, nearly 3 times slower than Hyperopt and Optuna, which were both approximately the same speed. All tools performed approximately the same in terms of accuracy (the goal metric), Hyperopt and Optuna excelled at precision, and Grid Search was the best for recall.

Regression

For the regression problem, we used data on CO2 emissions from vehicles, predicting the CO2 output from a vehicle based on its engine size, fuel type, vehicle type, etc. To do this, we used the `RandomForestRegressor` model from `scikit-learn`. A snapshot of this data can be seen below:

engine_size	cyl	fuel_consump_city	fuel_consump_hwy	fuel_consump_comb	co2_emissions	fuel_diesel	fuel_ethanol	fuel_natgas	fuel_regular	fuel_premium	class_SUV_minivan	class_compact	class_other	class_sedan_or_similar	class_truck
2.0	4	9.9	6.7	8.5	196	0	0	0	0	1	0	1	0	0	0
2.4	4	11.2	7.7	9.6	221	0	0	0	0	1	0	1	0	0	0

In order to compare the three tools, we used four different metrics:

1. **Mean Absolute Error (MAE)** – Average distance between predicted values and true values (in units of problem)
2. **Mean Absolute Percentage Error (MAPE)** – Average distance between predicted values and true values (in *percentage* difference)
3. **Mean Squared Error (MSE)** – Average *squared* distance between predicted values and true values; punishes outliers more than MAE
4. **Tuning Time** – How long the algorithm takes to run and pick the best parameters

MAE, MAPE, and MSE are all positive decimals with no upper limit. *Smaller* values (as close to zero as possible) mean better performance. Again, tuning time is measured in minutes and seconds, and *lower* values mean a more efficient algorithm/better performance.

For our three tools, we saw the following results for these metrics. Note that Hyperopt and Optuna were both defined to minimize *MSE*.

Table 3: Regression Experiment Metrics By Tuning Tool

Metric	Grid Search	Hyperopt	Optuna
MAE	1.692	1.686	1.691
MAPE	0.007	0.007	0.007
MSE	10.143	9.835	9.665
Tuning Time	2 min, 47 sec	0 min, 59 sec	1 min, 5 sec

Similar to the classification experiment, Grid Search was much slower than Hyperopt and Optuna, taking more than twice as long. Note that the overall decreased tuning time for this experiment is likely the result of the dataset for regression being much smaller (~1/4th the size) of the dataset for classification.

Hyperopt and Optuna both scored better than Grid Search in our goal metric (MSE), with Optuna scoring the best overall. All tools were relatively similar in terms of MAE and MAPE.

Additional Notes

In terms of ease to code, both Hyperopt and Optuna took relatively similar amount of time to figure out how to use. However, this was mostly because of similar examples being available online. Based purely on the official examples provided by Hyperopt, it was not clear how to conduct the experiment as desired, and external resources were required. With Optuna, we only needed to look at the official documentation.

However, using both Hyperopt and Optuna required more code and thinking to use than doing Grid Search, as users needed to write additional definitions for what to optimize and how to create an tuning study.

Both Hyperopt and Optuna share the nice feature of being able to specify exactly which metric you want to maximize or minimize. This is a great option for situations where users may care about one metric more than others. For example, if working with medical data, false negatives may be dangerous and users might want to prioritize *recall* in their modeling order to minimize those.

Based on this experiment, Optuna's version of this metric customization is much more straightforward to implement. Hyperopt uses an unintuitive system where you can only *minimize* a function. This means if you are trying to maximize something (such as accuracy), you need to return the negative version of this metric. This is easy to miss, providing extra opportunity for having a mistake and accidentally tuning the model the opposite direction of what is desired. In Optuna, you can manually select minimization or maximization, making this much less error-prone.

UI Review

Hyperopt

Optuna

Conclusion