# Tool Comparison: Optuna vs Hyperopt

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

## Introduction

In the field of machine learning, **hyperparameter tuning** is the process of selecting an optimal set of **hyperparameters** for a learning algorithm.

Hyperparameters are user-selected parameters applied to learning algorithms that affect how they are implemented–for example, constraints, weights, learning rates, number of algorithm layers/complexity, etc.

Depending on which set of hyperparameters is selected, the results and performance of a particular machine learning algorithm can vary dramatically. The possible decision space, however, is enormous. In many cases, there are essentially an infinite set of possible combinations for all of the tuning parameters in a model. It is *impossible* to try every single one of them and find the absolute best option–instead, users must select a *good* option.

Typically, in hyperparameter tuning, users will pre-define a list of combinations to try. For example, if they must provide two parameters, $\beta$ and $\lambda$, they might give the following potential values for these two parameters and ask the algorithm to try every possible combination within:

- $\beta = \{3, 5, 7, 9\}$
- $\lambda = \{0.001, 0.01, 0.1, 1.0\}$

This technique is known as a **grid search**.

It can sometimes produce decent results, but has several downsides.

1. The user must themselves define which lists of values to try. Coming up with this list introduces a high amount of subjectivity.
2. It is very computationally intensive (and slow) to try every single combination, one at a time. As the number of parameters increases, the number of combinations grows exponentially.
3. There is no easy way to compare results without a lot of additional coding on top of constructing the algorithm.

This is where hyperparameter tuning tools come into play. Several tools have been developed in recent years to help simplify this process and produce better results. Some use parallel computing to test a parameters simultaneously and speed up the training process, Many have UI implementations that can help make it easier for users to compare performance across models. Additionally, modern tuning software often provides advanced techniques to help the user select the best hyperparameters, such as intelligently moving the tuning in a direction that appears to be associated with increased performance scores scores.

**Hyperopt** (released in 2011) and **Optuna** (released in 2020) and are two industry standard parameter optimization tools designed to integrate with Python.

### Hyperopt

Hyperopt is a popular Python library for hyperparameter optimization that has been around–and been considered industry standard–for more than a decade.

It is based on Bayesian optimization techniques, specifically Sequential Model-Based Optimization (SMBO). In SMBO, trials are run one after another, with each trial updating its hyperparameters by updating a probability model and applying Bayesian reasoning. Its goal is to maximize some "score" by modifying a configuration, and it does this by working to narrow the search space and use advanced algorithms for finding which combination of parameters can maximize the probability model.

Within SMBO, it implements several advanced hyperparameter tuning methods including:

- Tree-Structured Parzen Estimator (TPE)
- Adaptive Tree of Parzen Estimators (ATPE)
- Gaussian Processes (GP)

Hyperopt does not have a full UI implementation, but it does have several built in visualization functions for comparing models.

Hyperopt's code is publicly available on GitHub:

- https://github.com/hyperopt/hyperopt

It also has a basic documentation website:

- http://hyperopt.github.io/hyperopt/

### Optuna

Optuna was released into the hyperparameter tuning market at a time when there were already quite a few established and trusted tools available (including the second tool we will discuss– Hyperopt).

Its creators chose to introduce Optuna after they discovered limitations in many other existing tuning algorithms, including that they:

1. Had instability in some environments
2. Were not utilizing the newest technology/advancements in hyperparameter optimization
3. Did not provide a way to specify which hyperparameters should be tuned directly within the Python code (instead, requiring the user to write special code just for the optimizer)

The creators of Optuna attempted to fill those gaps, advertising the following features:

1. **Define-By-Run style API** – An API style that is beginning to become industry standard in deep learning, but had not yet been applied to hyperparameter tuning prior to Optuna. Compared to an older system Define-And-Run. Using Define-By-Run allows users to write more modular code and access more complex hyperparameter spaces.
2. **Use of learning curves to prune trials** – Optuna uses deep learning and gradient boosting algorithms to predict the end result of a training trial before it is over. This can help it quit unpromising trials before they complete, improving efficiency.
3. **Parallel distributed optimization** – Optuna supports distributed optimization, simultaneously running multiple trials. This can dramatically speed up the tuning process and allow the user to increase the scale of how many parameters they can try.
4. **Visualization dashboard** - Optuna provides a UI dashboard where users can watch the optimization process and easily compare results from optimization experiments.

Like Hyperopt, the tool uses several advanced algorithms (including Bayesian methods) for tuning that go beyond standard grid search:

- Tree-Structured Parzen Estimator (TPE)
- Gaussian Processes (GP)
- Covariance Matrix Adaptation (CMA)
- Asynchronous Successive Halving Algorithm (ASHA)

These algorithms are similar to those used in Hyperopt, though Hyperopt is missing CMA and ASHA. The inclusion of ASHA in Optuna is particularly of note, as this is a key feature for allowing parallel optimization.

Optuna is framework agnostic, meaning it can easily be integrated with any of Python's machine learning/deep learning frameworks: Scikit-Learn, PyTorch, Tensorflow, etc.

The code for Optuna is publicly available on GitHub:

- https://github.com/optuna/optuna

There is a second GitHub page supporting their UI dashboard component, which was released separately:

- https://github.com/optuna/optuna-dashboard

Optuna also has a custom website with tutorials, examples, and links to documentation:

- https://optuna.org

---

In this project, we compare these two tools. We start with a code review of both tools based on the code available in their public repositories. After this, 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.

## Code Review: Hyperopt

Repository Link: GitHub

### Repository Overview

In this section, we cover several basic characteristics of the GitHub repository that are not code specific. For each category, we rate the repository on a scale of 1-10.

**Note:** All of the information in this section is as of March 2023, and may have changed since then.

#### Popularity/Use

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.

`Score: 8/10`

### Community

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.

`Score: 7/10`

### Maintenance

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.

`Score: 4/10`

### Dependencies

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.

`Score: 8/10`

### Security

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.

`Score: 9/10`

## Non-Functional Requirements

In this section, we will look at several non-functional requirements and consider how well the Hyperopt repository achieves them. Functional requirements will be considered later, in the *Experimental Comparison* section, along with a look at the UI features of the tool.

### Understandability

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:

- 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 mentionthe 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 for a project so math-heavy, requiring the user to define search spaces and limiting functions.

There is also some slightly contradictory information. For example, on a page titled "Getting started with Hyperopt", one sentence says:

> Parallel search is possible when replacing the Trials database with a MongoTrials one; there is another wiki page on the subject of using mongodb for parallel search.

This implies that `mongodb` is an optional addition that you can use if you want to do parallel search. However, if you go to the Installation page on the menu, the page is all about installing `mongodb`. Since this is the main **installation** page of the wiki, most users would probably assume you need to take this step. The page starts with:

> Hyperopt requires mongodb (sometimes "mongo" for short) to perform parallel search.

The way this is organized is needlessly confusing, especially when there is a whole separate tab on the documentation page about scaling out with `mongodb`. Why aren't the `mongodb` instructions included there, and the Installation instructions simplified down to what is necessary with a note about optional additions?

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 autodocumentation 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 autodocumentation 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.

**Score:5/10**

### Testability/Debuggability

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.

For example `test_webpage.py` seems to be testing a function that returns certain values. However, the code just runs this function and prints the result, then does nothing else. The expected values for these functions appear to be written in a comment under print statements, so why don't the authors just add an assert statement to confirm that the values being returned match what is expected? This particular test is also quite strange as it just appears to be a standard function and not written with `unittest`, like the rest of the functions.

```
    def test_landing_screen():
```

```
    # define an objective function
    def objective(args):
        case, val = args
        if case == "case 1":
            return val
        else:
            return val ** 2

    # 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)),
        ],
    )

    # minimize the objective over the space
    import hyperopt

    best = hyperopt.fmin(objective, space,
    algo=hyperopt.tpe.suggest, max_evals=100)

    print(best)
    # -> {'a': 1, 'c2': 0.01420615366247227}

    print(hyperopt.space_eval(space, best))
    # -> ('case 2', 0.01420615366247227}
```

`test_plotting.py` also has some questionable behavior. This test 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 based on a comment left in the code, it should be predictable, and thus 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`
    """
    # TODO: consider insisting on sorted idxs
    # TODO: use np.searchsorted instead of dct
    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.

**Score: 7/10**

#### Extensibility/Scalability

## Code Review: Optuna

Repository Link: GitHub

## Repository Overview

In this section, we cover several basic characteristics of the GitHub repository that are not code specific. For each category, we rate the repository on a scale of 1-10.

**Note:** All of the information in this section is as of March 2023, and may have changed since then.

### Popularity/Use

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 for over a decade, while Optuna has only been around for a few years.

`Score: 8/10`

### Community

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.

`Score: 9/10`

### Maintenance

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. There does not appear to be any `dev` branch used for testing, which is slightly strange, but most of the pull requests appear to be coming from *forks* of the repository (by other contributing users than the official Optuna team), which use different branch structures.

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.

`Score: 9/10`

### Dependencies

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.

`Score: 5/10`

### Security

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.

`Score: 9/10`

## Non-Functional Requirements

In this section, we will look at several non-functional requirements and consider how well the Optuna repository achieves them. Functional requirements will be considered later, in the *Experimental Comparison* section, along with a look at the UI features of the tool.

#### Understandability

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.

https://github.com/optuna/optuna-examples

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. For example, the benchmarks folder opens to this message:

> ## Benchmarks for Optuna
>
> Interested in measuring Optuna's performance? You are very perceptive. Under this directory, you will find scripts that we have prepared to measure Optuna's performance.

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, e.g:

```
def json_to_distribution(json_str: str) -> BaseDistribution:

    """Deserialize a distribution in JSON format.
    Args:
        json_str: A JSON-serialized distribution.
    Returns:
        A deserialized distribution.
    """
```

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. For example, the following code from the same function above just has one small comment:

```
json_dict = json.loads(json_str)

    if "name" in json_dict:
        if json_dict["name"] == CategoricalDistribution.__name__:
            json_dict["attributes"]["choices"] =
            tuple(json_dict["attributes"]["choices"])

        for cls in DISTRIBUTION_CLASSES:
            if json_dict["name"] == cls.__name__:
                return cls(**json_dict["attributes"])

        raise ValueError("Unknown distribution class: {}".format(
        json_dict["name"]))

    else:
        # Deserialize a distribution from an abbreviated format.
```

```
        if json_dict["type"] == "categorical":
            return CategoricalDistribution(json_dict["choices"])
```

`Score: 9/10`

### Testability/Debuggability

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, though `pytest` is known to have an advantage in understandability. This is because `unittest` requires a class-based formatting for tests, which requires a more complex structure, while `pytest` just uses functions to run tests.

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. This set of tests appears to help make sure the examples page that Optuna provides is up to date. There is an integration test confirming that each integration with a machine learning library (as provided in the examples) works properly.

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 1500 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 below:

https://app.codecov.io/gh/optuna/optuna

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:
        """Exit from the current optimization loop after the running
        trials finish. This method lets the running :meth:
        `~optuna.study.Study.optimize` method return immediately after
        all trials which the :meth: `~optuna.study.Study.optimize`
        method spawned finishes. This method does not affect any
        behaviors of parallel or successive study processes. This
        method only works when it is called inside an objective
        function or callback.

        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.

**Score: 9/10**

### Extensibility/Scalability

## Experimental Comparison

## Conclusion