Test report

Outlr.

A web application for analyzing subspace-outliers on high dimensional datasets

Bennet Alexander Hörmann, Salomo Hummel,
Simeon Hendrik Schrape, Erik Bowen Wu, Udo Ian Zucker

17.03.2023

Contents

1 Fixed bugs

The accuracy calculation after running an ODM is flawed. #1

Symptom After viewing the results of an Experiment the accuracy is always displayed incorrectly as 0.

Reason When converting the ground truth CSV to a numpy array, the header was included, even though the ground truth does not have one. This led to a data loss of one row. Then the accuracy calculation was done using our own method, which compares two numpy arrays in a single statement. However, when comparing two arrays of different size, it will always return 0, thus leading to this bug.

Fix Now, we fixed the conversion and use the accuracy method from scikit-learn to ensure accurate results.

The user is not redirected from routes requiring authentication when authentication "suddenly" is not valid anymore, even after a single page reload #2

Symptom The token expire event is not handled, when a page requiring authentication is already loaded. This way, the user is seemingly able to communicate with the backend although not being logged in, that is the user seeminly has a valid access token. Since no request is responded to with useful information the user might reload the page, which however, also does not directly redirect to the login page. This way, the user is in a state where nothing seems to work as the user is not redirected to login page nor is able to send requests to the backend.

Reason The token expire event is not handled correctly as the `isAuthenticated` state is not set when the token expires. Even if this state was properly set according to token validity, no action would take place since the authentication state has to be listened to.

Fix Now, the user is redirected when the user triggers a component sending a http request to the backend. Furthermore, a single page reload now redirects to the login page when a route requiring authentication is trying to be accessed. This was handling the token expiration event correctly on frontend side by setting the Vuex state according to the token's validity when http requests are sent as well as listening to the Vuex state change (similarly to observer pattern) by subscribing to the state change.

Running experiments can fail #3

Symptom Running an experiment sometimes raises an exception from SQLAlchemy.

Reason Every API request uses the same SQLAlchemy session. The objects that are created and modified during the execution are still in the session while the experiment is running. (Objects with relationships to objects that are already in the session will automatically be added to the session.) When the session is automatically flushed or when it is flushed by another API request while the experiment is still running, SQLAlchemy needs to work with these objects. But they can be in a state that is not accepted by the database. For example, the `execution_time` of `ExperimentResult` must not be `None`, but can only be defined after the execution has been completed. In this case, SQLAlchemy raises an exception.

Fix Now each API Request has its own session as it is intended by the SQLAlchemy documentation. Additionally, the API request that starts the execution now closes the session before starting the execution and uses a second session to write the changes back to the database once the execution has finished.

Exceptions from ODMs too unspecific #4

Symptom Exceptions raised by ODMs should be instances of `ODMFailureError`. Instead, they are instances of more general exception classes.

Reason Exceptions from ODMs are not wrapped in the `ODMFailureError` exception class.

Fix Now, exception handling is added to the `PyODM` class to raise `UnknownODMError`s and `ODMFailureError`s.

Exceptions from ODMs not caught #5

Symptom Exceptions raised by PyOD ODMs are not caught when using the `ExecutorODMScheduler` with a `ProcessPoolExecutor`. This causes the execution to crash instead of finish with an error. It can also lead to a broken process pool with the following error message: *BrokenProcessPool: A process in the process pool was terminated abruptly while the future was running or pending.*

Reason The exceptions that are raised by `PyODM.run_odm` are constructed using keyword arguments. When constructing them by passing the same arguments without a keyword, the bug does not occur. It is unclear why this happens.

Fix The given exceptions are now constructed using positional arguments.

kwargs #6

Symptom 'kwargs' parameters are recognized as mandatory but are optional.

Reason They do not have a default value.

Fix Set parameters named 'kwargs' to optional.

2 Unfixed bugs

Executor shuts down (not reproducible) #7

Symptom When scheduling an experiment sometimes one of the following exceptions is raised

- *RuntimeError: cannot schedule new futures after interpreter shutdown*
- *RuntimeError: cannot schedule new futures after shutdown*

Reason Unknown. The bug occurs rarely and cannot be reproduced reliably. It could be caused by modifying a python file while the server is running.

Supervised methods are displayed but not runnable #8

Symptom Supervised methods are not supported but displayed on the create Page.

Reason When scraping the PyOD module the methods are not checked on whether they are supervised or unsupervised.

3 Improvements

Remove Dataset Class #I1

Previously The `Dataset` class bundles the dataset name and the pandas `DataFrame`. But since the dataset name is stored in the database there must be a single attribute for it. The `Dataset.name` is therefore never used.

Improved Instead of a `Dataset` class pandas `DataFrame`s are used directly.

Improve Execution #I2

Previously The experiment execution is built around the asyncio library.

Improved The experiment execution is now built around the concurrent.futures library. This library is better suited for CPU-bound tasks while asyncio is better suited for IO-bound tasks. However, the benefits of asyncio are still used, since the two libraries are compatible. In particular, experiments are now executed as coroutines in an asyncio event loop that runs on a single background thread. There, the subspace logic evaluation, metrics calculations, and the waiting for the individual subspaces is done. The individual subspaces however, are run in a concurrent.futures `ProcessPoolExecutor`, which holds a pool of processes to distribute individual ODM calls.

Better Subspace logic Parser #I3

Previously Multiple operators are possible, but they can be applied only right associatively. The simple case of `([0] or [1]) and ([2] or [3])` is not possible. Additionally, many incorrect inputs are accepted.

Improved Now, subspace logic can be nested using parenthesis. Many cases of incorrect inputs are caught and not parsed.

Dashboard Table Metadata #I4

Previously Table sorting was accomplished by comparing the string values of each cell.

Improved Each cell now contains a tuple with both metadata and string representation. This allows for more accurate sorting, as the rows can be sorted based on the metadata of each cell rather than simply comparing string values.

Dashboard Table Sorting Options #I5

Previously Table sorting was reliant on the header strings to indicate which cell the table should be sorted by.

Improved Each table header now contains a tuple consisting of a Sort Enum entry and the header string. This Enum makes the sorting selection process easier and more efficient.

Dashboard Table Date Shown #I6

Previously The date on the dashboard table was previously displayed in a standard date format.

Improved Now, when an experiment is created less than a week ago, the time is shown as [time] ago.

Show Subspace logic on Experiment Result Page #I7

Previously Subspace logic is never displayed after an experiment is created.

Improved The subspace logic of an experiment is displayed on the experiment result page.

Subspace Logic Indicator #I8

Previously Incorrect subspace logic was indicated by an unusable button.

Improved Now, incorrect subspace logic is directly indicated on the text area field with a red border. The button is still unusable with incorrect subspace logic.

Enhanced Password Storage Security #I9

Previously Password were stored in the database after being hashed with the SHA-256 hashing algorithm.

Improved Passwords now are hashed using bcrypt, which is specifically designed for password hashing. Bcrypt incorporates a cost parameter that makes the hashing process deliberately slower. This means that each password hash computation takes a longer time to complete, making it much more difficult for attackers to perform brute force attacks. It also includes salting as part of its hashing process, further increasing the security of the password hash.

Minor Improvement to User Experience when Registering #I10

Previously The green color used to mark valid input was quite bright.

Improved The color to mark valid input was changed to a more subtle green.

AUC and ROC Curve #I11

Previously The AUC and ROC Curve was neither stored in the database nor displayed to the user.

Improved The AUC and ROC Curve are stored in the database. They get displayed to the user on the experiment result page.

4 Unit tests

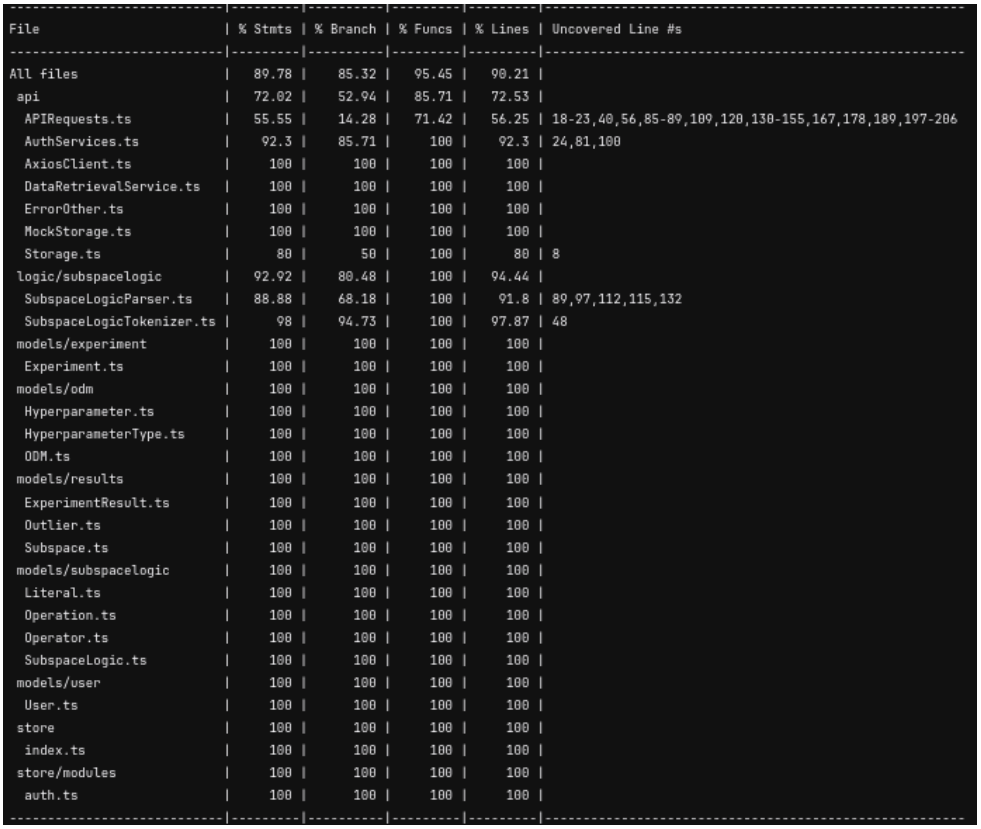
We have utilized the unittest package for Python to ensure good quality in our backend. By creating a comprehensive suite of unit tests, we are able to systematically test individual units of code in isolation, ensuring that each function and method works as expected.

To ensure that our unit tests cover as much of the codebase as possible, we have also utilized the coverage package for Python. This package allows us to measure how much our unit tests exercise the codebase. By analyzing the results of code coverage reports, we can identify areas of the codebase that are not being tested and prioritize efforts to improve test coverage.

On the frontend side, we utilized the Jest testing framework as our unit test package for TypeScript. Jest is a popular testing framework used for testing Vue applications and is highly regarded for its ease of use and powerful features. Since Jest already includes coverage we didn't need another framework in the frontend.

Overall we achieved $\geq 90\%$ code coverage on both front and backend as documented in the following pictures.

```
Name                                                                                    Stmts   Miss  Cover   Missing
---------------------------------------------------------------------------------------------------------------------
src/config.py                                                                             6      0   100%
src/database/database_access.py                                                          55      3    95%   61, 66-67
src/execution/execution_error/__init__.py                                                10      0   100%
src/execution/execution_error/odm_failure_error.py                                        7      0   100%
src/execution/execution_error/subspace_error.py                                           6      0   100%
src/execution/execution_error/unknown_odm_error.py                                        7      0   100%
src/execution/experiment_scheduler/__init__.py                                           69      3    96%   39, 106, 111
src/execution/experiment_scheduler/background_thread_event_loop_experiment_scheduler.py  15      0   100%
src/execution/experiment_scheduler/event_loop_experiment_scheduler.py                    31      2    94%   53-54
src/execution/odm_scheduler/__init__.py                                                  10      1    90%   29
src/execution/odm_scheduler/executor_odm_scheduler.py                                    12      0   100%
src/execution/odm_scheduler/sequential_odm_scheduler.py                                  19      2    89%   28-29
src/models/base.py                                                                        2      0   100%
src/models/experiment.py                                                                104      5    95%   278-279, 301, 322, 338
src/models/json_error.py                                                                  5      0   100%
src/models/odm/__init__.py                                                                3      0   100%
src/models/odm/hyper_parameter.py                                                        12      0   100%
src/models/odm/odm.py                                                                    22      4    82%   45-47, 51
src/models/odm/pyodm.py                                                                  22      2    91%   35-36
src/models/subspacelogic/__init__.py                                                     43      7    84%   20, 33, 46, 58, 80, 88, 96
src/models/subspacelogic/literal.py                                                      35      1    97%   67
src/models/subspacelogic/operation.py                                                    39      1    97%   97
src/models/subspacelogic/operator.py                                                     21      2    90%   15, 20
src/models/user.py                                                                       11      0   100%
src/odmprovider/odm_provider.py                                                           7      1    86%   17
src/odmprovider/pyod_scraper.py                                                          35      0   100%
src/util/data.py                                                                         39     25    36%   29-31, 43-47, 60-66, 70-79, 83-86
tests/test_PyODM.py                                                                      26      3    88%   39-49
tests/test_config.py                                                                      6      0   100%
tests/test_execution.py                                                                  79      0   100%
tests/test_execution_errors.py                                                           21      0   100%
tests/test_models.py                                                                     42      0   100%
tests/test_models_result.py                                                              64      0   100%
tests/test_subspace_logic.py                                                            105      0   100%
tests/test_using_db.py                                                                  105      3    97%   87-89
tests/test_util_data.py                                                                  27      6    78%   33-35, 40-42
---------------------------------------------------------------------------------------------------------------------
TOTAL                                                                                  1122     71    94%
```



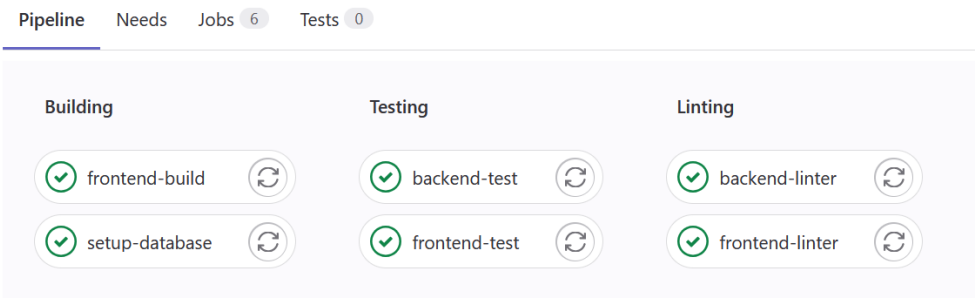
File	% Stmts	% Branch	% Funcs	% Lines	Uncovered Line #
All files	89.78	85.32	95.45	90.21	
api	72.02	52.94	85.71	72.53	
APIRequests.ts	55.55	14.28	71.42	56.25	18-23,40,56,85-89,109,120,130-155,167,178,189,197-206
AuthServices.ts	92.3	85.71	100	92.3	24,81,100
AxiosClient.ts	100	100	100	100	
DataRetrievalService.ts	100	100	100	100	
ErrorOther.ts	100	100	100	100	
MockStorage.ts	100	100	100	100	
Storage.ts	80	50	100	80	8
logic/subspacelogic	92.92	80.48	100	94.44	
SubspaceLogicParser.ts	88.88	68.18	100	91.8	89,97,112,115,132
SubspaceLogicTokenizer.ts	98	94.73	100	97.87	48
models/experiment	100	100	100	100	
Experiment.ts	100	100	100	100	
models/odm	100	100	100	100	
Hyperparameter.ts	100	100	100	100	
HyperparameterType.ts	100	100	100	100	
ODM.ts	100	100	100	100	
models/results	100	100	100	100	
ExperimentResult.ts	100	100	100	100	
Outlier.ts	100	100	100	100	
Subspace.ts	100	100	100	100	
models/subspacelogic	100	100	100	100	
Literal.ts	100	100	100	100	
Operation.ts	100	100	100	100	
Operator.ts	100	100	100	100	
SubspaceLogic.ts	100	100	100	100	
models/user	100	100	100	100	
User.ts	100	100	100	100	
store	100	100	100	100	
index.ts	100	100	100	100	
store/modules	100	100	100	100	
auth.ts	100	100	100	100	

5 Code Quality

Static code analysis is an effective method for detecting defects and vulnerabilities in software code. It involves the use of automated tools to analyze code without actually executing it. One of the most common types of static analysis is linting, which checks code for syntax errors, style violations, and other potential issues. Our Code now has 0 lint issues.

6 CI/CD Pipeline

We maintained and extended our Gitlab CI/CD Pipeline to a three-step process. The first one builds the web application and prepares our PostgreSQL database. In the second step, we execute our backend and frontend unit tests. And lastly, we check for code smells and style with our linter.



This three-step process in our Gitlab CI/CD pipeline has been extremely beneficial for our testing phase.

Firstly, by building the web application and preparing the PostgreSQL database in the initial step, we ensure that all necessary dependencies are installed and the environment is properly set up. This helps us avoid any errors or issues that may arise from missing dependencies or configuration problems.

Secondly, executing backend and frontend unit tests in the second step ensures that all changes made to the codebase do not break existing functionality. The unit tests act as a safety net that helps us catch any issues or bugs before they make it to the production environment. By running the tests in an automated way through the CI/CD pipeline, we can ensure that the tests are always run consistently, and we can easily detect issues as soon as they arise.

Finally, the third step in our pipeline checks for code smells and style with our linter. This helps us maintain a high level of code quality and consistency throughout the project. By catching potential issues with code style or best practices early on, we can avoid issues and ensure that the code is easy to maintain and extend in the future.

Overall, by maintaining and extending our Gitlab CI/CD pipeline to a three-step process, we have been able to improve the quality and reliability of our web application. The pipeline provides us with a consistent and automated way to build and test our code, catch issues early on, and ensure that our codebase adheres to best practices and style guidelines.

Glossary

asyncio Python library for writing concurrent code. It can be very useful for IO-bound tasks. See documentation. 7

concurrent.futures Python library for writing concurrent code. It provides thread pools and process pools. See documentation. 7

Experiment A configurable execution of outlier detection methods on subspaces of a dataset. 3, 8, 14

numpy Fundamental package for array computing in Python. 3

Object-Relational Mapping ORM (Object-Relational Mapping) is a programming technique that maps relational databases to objects in an object-oriented language. It provides an abstraction layer between the database and the application, and provides a set of tools to interact with databases using an object-oriented API.. 14

pandas Python library used for data analysis. 7

PyOD Python library providing methods for outlier detection. 4

Python Programming language commonly used in the field of machine learning and outlier detection. 14

SQLAlchemy Python library for working with relational databases. It provides a set of high-level APIs for interacting with databases in an object-oriented way, as well as a powerful ORM (Object-Relational Mapping) system for mapping Python objects to database tables.. 3, 4

Subspace A subspace of the complete dataset. 7, 14

Subspace logic A logic that specifies in which subspaces a point of the dataset must be identified as an outlier in order to be contained in the experiment result. 7, 8

Acronyms

CSV Comma-separated values, a file format commonly used for storing datasets. 3

ODM Outlier detection method. 3, 4, 7