Seascape: A Due-Diligence Framework For Algorithm Acquisition

Christopher Pitts[a] and Forest Danford and Emily Moore and William Marchetto and Henry Qiu and Leon Ross and Todd Pitts

[a] Sandia National Laboratories, 1515 Eubank Blvd SE, Albuquerque, New Mexico, United States, cwpitts@sandia.gov;

ABSTRACT

Any program tasked with the evaluation and acquisition of algorithms for use in deployed scenarios must have an impartial, repeatable, and auditable means of benchmarking both candidate and fielded algorithms. Success in this endeavor requires a body of representative sensor data, data labels indicating the proper algorithmic response to the data as adjudicated by subject matter experts, a means of executing algorithms under review against the data, and the ability to automatically score and report algorithm performance. Each of these capabilities should be constructed in support of program and mission goals. By curating and maintaining data, labels, tests, and scoring methodology, a program can understand and continually improve the relationship between benchmarked and fielded performance of acquired algorithms. A system supporting these program needs, deployed in an environment with sufficient computational power and necessary security controls is a powerful tool for ensuring due diligence in evaluation and acquisition of mission critical algorithms. This paper describes the Seascape system and its place in such a process.

Keywords: Machine learning, algorithm assurance, algorithm acquisition, artificial intelligence, algorithm assessment

1. INTRODUCTION

All programs that acquire algorithms have a need to accumulate, curate, label, preserve, and disseminate valuable data for algorithm development. Additionally, there is a strong need to reliably, impartially, and repeatably benchmark algorithms under consideration, comparing them to competitors as well as earlier versions of themselves.

The Seascape system uses off-the-shelf open-source software to create a system for the management of test and evaluation data, automated benchmarking of algorithms, and standardized reports across all algorithms under evaluation. To manage data, the Nexus artifact storage server was used to provide easily indexable storage for both evaluation data and final reports, with Apache Solr and Banana being used to provide more in-depth analytics and indexing capability. GitLab provides version control and integrated Continuous Integration (CI) capability supporting automated evaluation and result generation. Automated schedules configured in GitLab CI are used to trigger evaluation and report generation stages, with Python shim code determining which algorithms to assess.

Seascape is designed to use abstract interfaces to algorithms, driven by shell scripts and packaged dependencies in algorithm repositories. Both bare-metal and containerized (via Docker, Podman, or other Open Container Initiative (OCI)-compliant container image) versions of algorithms are supported. A containerized algorithm also offers the ability to run an algorithm under an environment that replicates actual deployment environments, enabling a "test-what-you-fly" methodology that provides an additional layer of algorithm assurance.

In this paper we first outline a set of principles for due diligence in algorithm acquisition for the purpose of discussing the Seascape system. We then proceed to discussing the design of Seascape and show how the application of the principles produces a system that directly addresses the needs of an algorithm acquisitions process at the program level. We conclude with an analysis of the deployment of Seascape at Sandia in support of sponsor-directed algorithm acquisition, and discuss how Seascape provides a robust, streamlined, and transparent evaluation framework.

2. BACKGROUND AND RELATED WORK

The term *technical debt* was coined by Ward Cunningham to refer to the trade-off between speed and quality.[1] The precise meaning of the term has fluctuated, and with the rise of formal software program management has come to mean something akin to "risk arising from code". Beyond risk stemming from code, risk to the success of a program can come from factors external to the actual development of an algorithm. Sculley *et al.*[2] discussed the sources of risk that arise in machine learning systems that have multiple component parts, and these conclusions are generally true for any system involving multiple software components.

For a program that is in the business of funding and acquiring algorithms, the technical debt is small to nonexistent, as the development work is done by third parties, either directly funded by a sponsor or opportunistically presenting their algorithms for acquisition. The system-level debt, however, can be enormous, as a flawed algorithms acquisitions process can lead to the deployment of algorithms that are not effective in real-world usage, are not the most effective algorithm for the problem the sponsor wishes to solve, or are not usable by non-experts. The goal of an algorithm acquisitions program must therefore be to manage the risk of acquiring an ineffective or incomplete algorithm, and ensuring that any acquisition decisions will deliver tested, validated, and trustworthy algorithm solutions, regardless of the source.

It is appropriate to consider the relationship of the Seascape algorithm evaluation framework to other environments focusing on managing development. Many of these other frameworks such as Domino Data Lab, Anaconda Enterprise Edition, and MLflow, have arisen recently in the machine learning (ML)/Artificial Intelligence (AI) communities in support of Machine Learning Operations (MLOps). They work with ML and data science frameworks such as Jupyter, RStudio, SAS, MATLAB, Spark, etc. to assist developers in managing model and algorithm development. In contrast to these frameworks Seascape helps a program manage the validation and verification process regardless of the frameworks or tools used to develop the algorithms, or in the specific case of ML, the models. It supports due diligence in the acquisition process by managing data stores, test definitions, released algorithm versions, and benchmark results. It can also provide the ability to manage pipelines ensuring requirements compliance.

3. PRINCIPLES OF DUE DILIGENCE IN ALGORITHM ACQUISITION

The dynamic nature of research and development work, particularly in algorithm development, prevents a universal definition of what constitutes "due diligence" in algorithm acquisition. For the purposes of this paper, we will be using the following principles:

1. Automation
2. Data management
3. Reproducibility
4. Analysis
5. Scalability

We note at the outset that these principles are not independent, but rather support one another. For example, reproducibility is not really possible without automation and data management. Also, we assert that a good tool for program level management of the acquisition process should be agnostic to the tool sets chosen by the algorithm developers. This encourages use of the right tool for each development process instead of shoehorning teams into a preselected framework.

3.1 Automation

All algorithm evaluation processes should be automated. Automated evaluation ensures that each algorithm is properly tested in a repeatable fashion. It also significantly reduces the cost of testing multiple times. Testing should happen each time an algorithm is changed. Manual processes are expensive and can be tedious for large tests. They also lead to significant program risk of error, omission, lack of repeatability, and extended time lines.

3.2 Data management

In an objective evaluation framework, data must be available in a format that is easy to index and search. The format must have a simple interface, and be straightforward to parse in a wide array of programming languages. Using a Python pickle format, for example, would be a poor choice, as it would force all algorithms to be implemented in Python (or at least include Python as a dependency). Leaving the choice of implementation language up to development teams widens the field of potential contributors, and allows the framework to take an agnostic high-level approach to how it executes the algorithm against test data.

In addition to the storage and access requirements, the data should be curated, labeled, and verified post-ingest by subject matter experts. *Curation* refers to the process of ensuring the data are stored in a consistent, complete, accurate, and machine readable, format. For example, longitude could be specified as a value in degrees in the range $[0, 360)$, or in the range $[-180, 180)$. It could also be specified as a value between 180W and 180E. However it is done, it should be done consistently. Curation also includes the correction of errors in the data (for example image coordinates given with longitude in the range $[0, 360)$ and labels given with coordinates in the range $[-180, 180)$ or image labels in row-column format when column-row is intended. *Labeling* refers to the process of indicating the proper response for an algorithm when shown the data. For time series signals it may be selection of a segment of the signature or a high level label indicating it is a signature of a particular class. For imagery, it may be a region of pixels containing objects to be detected and classified together with the class of the object. How labels are applied depends in large measure on the type of data and intended application. The datasets should accurately represent both in- and out-of-class targets. Such a practice also helps to mitigate the effects of an adversarial attempt to poison the evaluation dataset by introducing false training samples. Curation and labeling is generally time consuming and expensive. However, if properly done, it represents Non-recurring engineering (NRE) capturing the evaluations of subject matter experts that can be used by all developers in perpetuity.

3.3 Reproducibility

Any sort of evaluation should be reproducible; that is, given the same algorithm implementation and inputs, the same outputs should be produced, and the same assessment generated. This ensures that assessments are reliable and consistent across evaluation runs. It is also one of the primary goals of test automation. Crucial to the notion of reproducibility is the idea of smart bookkeeping that enables the tracking of algorithm version and their association with correlated benchmark results. The association of a released version of an algorithm with its corresponding performance benchmark should be inherent in the testing process.

3.4 Analysis

Any algorithm evaluation process must include a summary analysis that compiles results and expresses them using a set of metrics that are meaningful to program experts skilled in evaluating deployed performance. These metrics should be determined and controlled by the acquisition program, not the vendor. Specific deployment environments may lead to favoring a low probability of false alarm, or a particular scenario may necessitate identifying which specific classes of false positive are especially egregious. These analyses will be highly specific to the program acquiring the algorithms, and are not necessarily implemented in the algorithm itself. Thus, a viable framework must provide capability to generate standardized assessments that can be configured for different program needs.

3.5 Scalability

As the number of algorithms (and their versions) under evaluation grows, the demand on infrastructure correspondingly increases. So, any algorithm evaluation framework should be able to efficiently scale with the number of algorithms currently under evaluation. This is especially true when algorithms that require a significant amount of time to execute are being evaluated; having one algorithm consume all of the available computing power would be a significant risk to the success of an acquisitions program. Scalability is also critical in ensuring adequate testing can be accomplished for any single algorithm. Performance testing sufficiently thorough to ensure meaningful correlation with deployed performance is often computationally burdensome even for a single

algorithm. Use of compute clusters and parallel processing in the evaluation framework can allow faster-than-real-time benchmarking and significantly improve evaluation quality while maintaining a practical time line. Estimating false positive rates over long periods of time can only realistically be done with parallel processing. This will almost always occur when computing false positive rate while attempting to detect low probability events.

4. SEASCAPE DESIGN

Seascape is designed around a software stack composed primarily of broadly used open source components. It is implemented as a lightweight overlay capable of distributing tests to multiple computers, gathering results, producing reports, and handling the logistics and accounting of correctly associating them with algorithm release versions. It is agnostic towards development and language frameworks placing no related requirements on the development teams. Support for containerization technologies (such as Docker) reduces the time required to insert an algorithm into the testing harness. Containerization allows the development team to effectively integrate at their own site in their own environment and deliver the final container to the Seascape instance without the need for burdensome coordination or instruction from the procurement team. If desired, containerization also allows the test framework to mimic deployed conditions.

These are discussed in more detail in the sections below. Section 4.1 discusses how an algorithm release version is created and, together with a Git release tag, used to control testing. The Seascape data store is simply the corpus of data used for validation and verification by the program. Its structure is reviewed in Section 4.2. The use of multiple computers to accelerate testing as well as the collation of test results is outlined in Section 4.3. Finally, Section 4.4 discusses the generation and customization of evaluation metrics and reports. A general overview of the Seascape system is given in Figure 1.

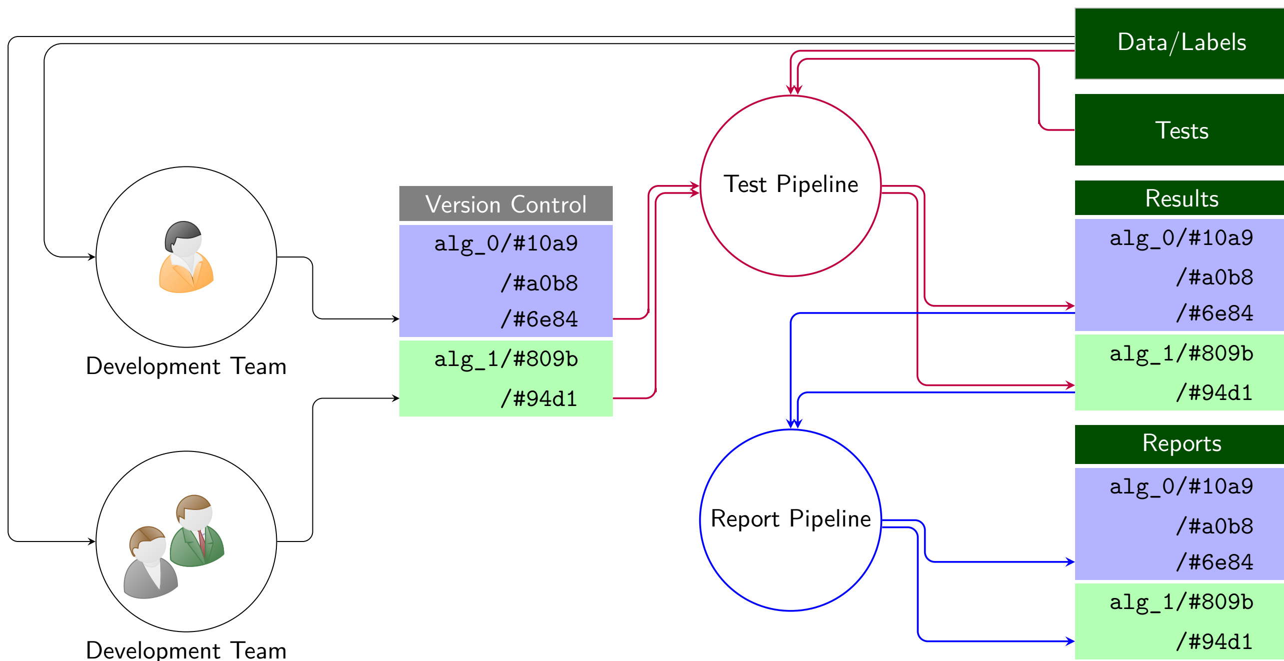

Figure 1: An overview of Seascape. Development teams use labeled data from the data store to develop algorithms. They then submit release versions by checking them in to a version controlled repository, implemented in Seascape using GitLab. A unique number (hash) is automatically associated with each checked in algorithm. Seascape then uses additional data and labels to drive test pipelines and generate results, storing them under the hash in the artifact repository. These results are then used by the report generation pipelines to create and store performance evaluations.

4.1 Version Control

Seascape uses the Git version control system to manage algorithms under test. A team that submits an algorithm to the program checks various artifacts into a Git repository, including the algorithm, any trained models or parameters that are used by the algorithm, and a shell script used to load input data, execute the algorithm, and output results. This is a key part of the design of Seascape, as Git commit hashes provide a means to associate algorithm versions with results and assessments. A *commit hash* is a Secure Hash Algorithm (SHA)-1 hash of the contents of the repository (that is, all files being tracked by Git) at the moment the snapshot was taken.[3] Having an immutable reference to the state of a repository makes it possible to directly link a particular version of an algorithm with the results produced by the algorithm, and the final assessment of performance based on test results. This supports reproducibility of results (principle 3), because it is easy to checkout a particular version of the algorithm and run the assessment again.

To determine which commits should be treated as new versions to test, Git *tags* are used to mark particular commits. A Git tag is a named reference to a specific commit, used primarily to mark commits with particular significance (version releases, new features, etc.). Rather than treat each new commit as a new version, Seascape's evaluation and report generation stages search for tagged commits in the repository. Tags names matching a specified format are treated as new versions of an algorithm. This avoids the generation of results and reports for commits where stylistic changes, documentation updates, or other non-algorithm changes are added to the repository.

4.2 Data Store

Three types of data are stored in Seascape: datasets (with labels), test instance results (detailed in Section 4.3), and reports (detailed in Section 4.4).

Datasets are stored in a directory tree format similar to what is often found on disk. Each data instance is stored in a separate directory under the top-level dataset directory, along with a JavaScript Object Notation (JSON) label file and, if necessary, a JSON metadata file. An example of this structure using a fictional "penguins" dataset with Tagged Image File Format (TIFF) images is shown in Figure 2.

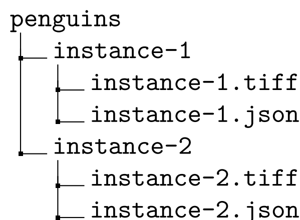

Figure 2: Each instance is stored in a separate directory, along with all related information. This example is for explanatory purposes only, Seascape can be used with data of any type (time series, image, etc.).

The JSON file contains any information necessary for the assessment stage to determine whether or not the algorithm made the correct decision. The label file contents may be as shown in Figure 3.

Having expert-adjudicated data in a clear format that is accessible to all members of the program (and access that can be extended to teams submitting algorithms) ensures that research will be done on an agreed-upon dataset, and that results will be computed and reports generated for the same dataset every time (principle 2).

4.3 Job Dispatch and Result Collation

Seascape uses GitLab CI to distribute batches of test data or *instances* to machines configured to accept jobs. An *instance* is a single training or testing unit. For example, it may be a single radio frequency time series in which we wish to detect a target signature or an image in which we wish to locate and classify objects. Because each instance is processed separately, the dispatch problem is embarrassingly parallelizable and eminently scalable (addressing point 5 in our definition of due diligence in section 3), and the speedup is directly proportional to the number of machines on hand and the batch size.

```
[
  {
    "class": "adelie",
    "imageID": "instance-1.tiff",
    "polygonPxl": [
      [10, 10],
      [11, 11],
      [10, 12],
      [9, 11],
      [10, 10]
    ]
  },
    {
    "class": "emperor",
    "imageID": "instance-1.tiff",
    "polygonPxl": [
      [100, 110],
      [110, 110],
      [110, 100],
      [100, 100],
      [100, 110]
    ]
  },
]
```


Figure 3: Keys for the class label, the source image, and the bounding box of the target

Algorithm responses are automatically adjudicated by the system using the expert provided labels and a program defined scoring metric (expressible in Seascape though a plugin architecture). As jobs are completed, results for each instance in the batch are stored under a key built from the Git commit hash of the algorithm and the test instance ID. Such a composite key structure imparts an idempotent characteristic to Seascape; a particular test can be re-run and skip the computation of results, because an algorithm and model that have not changed would produce the same result on the same input data instance. Additionally, a *new* test composed of both data instances with results and instances not yet used in a test only needs to process results for the instances not yet seen, because the results are tied to the algorithm commit and the instance ID, reducing the runtime for new tests. Given a new test N and an existing test P, such that $N \subset P$ and all instances in P have been processed, no new jobs need to be run at all, and Seascape will move directly to the report generation stage. Naturally, a program user does not need to manage this process at all. It is inherently part of the way Seascape works.

The dispatch of jobs *can* be manually triggered by a member of the program team, but the primary method for starting an evaluation job is a schedule, configurable using the built-in GitLab scheduling method. This needs to be done only once per algorithm and consists of entering a periodic time to launch any necessary jobs for evaluating the new algorithm. Algorithm changes will be automatically detected and adjudicated after check in at the next scheduled test time. This permits regular automated assessments of all changes in algorithms (principle 1), and relieves groups submitting algorithms of the need to manually generate assessments for sponsors.

4.4 Report Generation

Along with evaluation jobs, Seascape also schedules jobs for generating reports. A report generation job examines all tagged algorithm versions, and checks the data store for each test definition. When an algorithm version has results for all images in a test definition, a new report is generated from the results. As with results and test definitions, final reports are stored under a key composed of a dataset name and algorithm commit hash.

Report generation is designed to use a plugin architecture, enabling sponsors or programs to design customized

assessments of algorithm performance, addressing due-diligence principle 4. Reports can be generated as HTML documents, LaTeX Beamer slides, or any format that can be generated programmatically from a Python script.

5. EMPIRICAL RESULTS

Here we discuss a specific example involving two algorithms: RetinaNet and DetectNet. We might consider these algorithms to be the product of the two development teams shown in Figure 1 by labeling them `alg_0` and `alg_1` respectively. Each algorithm has been trained for the identification of aircraft in overhead imagery. In order to test the performance of each, we are using a data set we'll call `RAVE_planes`. These data correspond to the `Data/Labels` block (dark green) in the upper right hand corner of Figure 1. The test data set consists of 1390 images. Each image has labels for a number of object types. Here we'll concern ourselves with three of these labels: PassengerCargoPlane, SmallAircraft, and FighterJet. Over the 1390 images in the test data set we find, respectively, 2029, 2228, and 735 examples of each class (see Table 1).

Algorithm name (hash) Test Date	**retinanet (6e84)** 2022-03-25	**detectnet (94d1)** 2022-03-28
FighterJet (735)		
True positive	368	525
False negative	367	210
False positive	46	54
Precision	0.50	0.71
Recall	0.89	0.91
F1	0.64	0.80
PassengerCargoPlane (2029)		
True positive	1542	1873
False negative	488	156
False positive	150	1049
Precision	0.76	0.92
Recall	0.91	0.64
F1	0.83	0.76
SmallAircraft (2228)		
True positive	1524	706
False negative	704	1527
False positive	149	101
Precision	0.68	0.32
Recall	0.91	0.87
F1	0.78	0.46

Table 1: Seascape-generated evaluation of two algorithms against the same dataset. Three targets are considered. The number of labels for each class (support) in the test data set is given after the label in parentheses. Seascape allows custom report generation via a Python language module, tailored to the needs of a program. In this case, we use the a brief set of six statistics shown in the table to compare the algorithms.

The associated GitLab instantiation (represented by the `Version Control` block (light gray) in the center of Figure 1) automatically computed a unique hash for the version of each algorithm under test. The first four digits of each hash are shown in the Table 1 immediately following the algorithm release name. In the web-based report the full hash is given. Using GitLab we may apply a tag to each hash. We use this built-in facility of Git (accessible via the GitLab web interface) to indicate which algorithm versions should be run against a test. Affixing a tag to an algorithm hash ending in `_full` or `_baseline` indicates that Seascape should generate results and reports for baseline (simple) and full test data sets respectively. In our example, we use a full test for each algorithm. After tagging, Seascape automatically ran each algorithm on all 1390 images using GitLab runner over a small cluster of machines. This is shown in Figure 1 as the red component labeled `Test Pipeline`. The results for each image were stored under the corresponding algorithm and hash. After completion of the

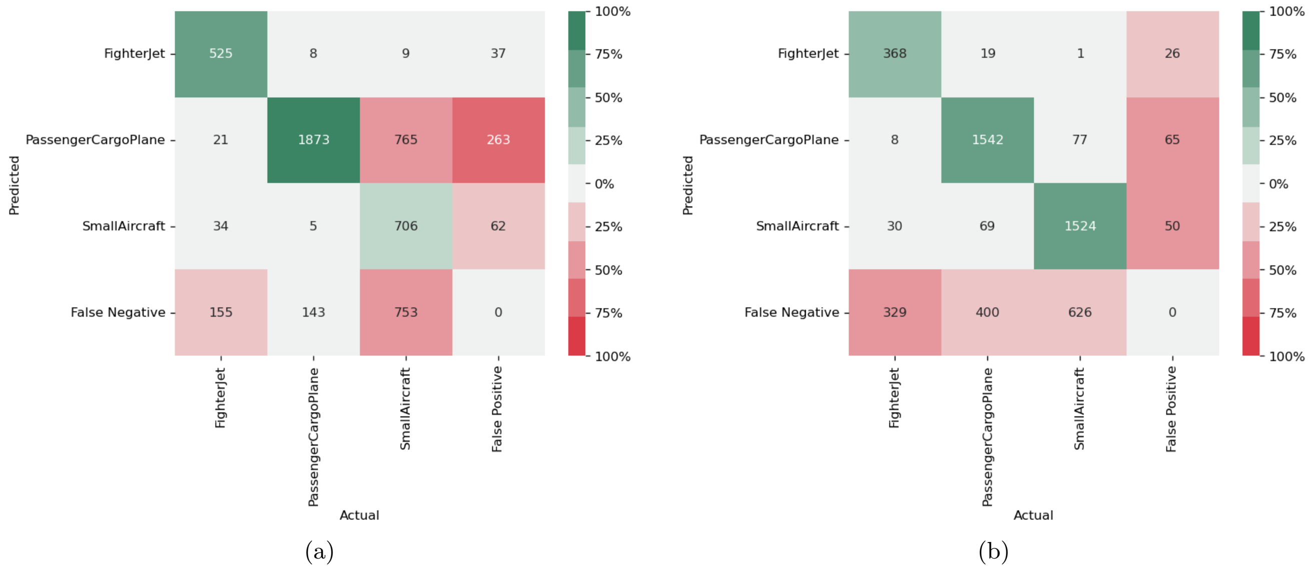

Figure 4: (a) Confusion matrix for the RetinaNet algorithm. (b) Confusion matrix for the DetectNet algorithm. The sum of each column gives the support (total number) of test elements in each category. For this data set we have 735, 2029, and 2228 respectively for labels FighterJets, PassengerCargoPlanes, and SmallAircraft.

entire test set a report was automatically generated and stored under the algorithm name and hash (see the blue component labeled `Report Pipeline` in Figure 1). The results are summarized in Table 1 and Figure 4.

Table 1 compares the two algorithms based on six simple statistics: the numbers of true positives, false negatives, false positives, precision*, recall†, and F1 score‡. It is clear from the data that DetectNet performs better under the F1 score than RetinaNet on the FighterJet class, while the converse is true for the PassengerCargoPlane and SmallAircraft labels.

Figure 4 shows the confusion matrices for each algorithm from the corresponding Seascape reports. The columns and rows of a confusion matrix correspond, respectively, to the actual and predicted class labels. The final row of the matrix gives false negatives (not detected at all). The final column shows false positives (erroneous detections). The final square is always zero. The diagonal of each confusion matrix shows the number of objects correctly detected and classified. The remainder of the entries show mislabeled detections.

Whenever a Git tag is updated to point to a newer version of either algorithm, a new corresponding set of results and report are automatically generated.

6. SUMMARY

This report describes the Seascape system and its ability to provide support for due diligence in program level algorithm development, acquisition, and deployment. Seascape is a modular composition of a small number of open-source components (GitLab, GitLab Runner, Nexus, Solr, Banana), together with some orchestration code written in Python. Any language that can run in CI is supported for algorithms. All data types (time series, imagery, etc.) are supported. The primary purpose of the system is to gather, label, and curate program relevant data sets, provide the ability to explore the data, define representative bodies of data to serve as tests, and to automate the running of tests, storage of intermediate results, and generation of reports on a per algorithm per release basis.

---

* **Precision** is defined as the ratio of true positives to the sum of true positives and false positives (total number algorithm generated labels for the given class).

† **Recall** is defined as the ratio of true positives to the sum of true positives and false negatives (the total number of in-class labels in the test data set).

‡ **F1** is defined as the harmonic mean of precision and recall.

From a technical burden standpoint, development teams only have to know how to check code in for evaluation in CI or use a Docker repository (very common knowledge). Currently, on the program side the set up of GitLab schedules and use of JSON to define tests is required. All subsequent processing is automatic after set up. It is also possible to define a program specific test adjudication metric and report format via Python.

Among the advantages of using such a system as part of ensuring diligence in acquisition are: impartial, repeatable, explainable adjudication, significant reduction in integration cost/effort through the use of containerization (Apptainer, Podman, Docker), more frequent assessment, up-to-date statistics, reduction of benchmark errors and repeatability via automation. Integrated use of version control also enables understanding of algorithms deployed in the field. The application of open-source technology reduces cost and increases the number of individuals who are likely to have preexisting experience with system components already. Finally, it facilitates the collection, organization, curation, labeling, and understanding of the data, which is central to establishing a benchmark expectation for field performance.

Acknowledgements

This work was conducted with funding from the Navy RSCD (Remote Sensing Capability Development) program operating in PMW-120 (Battlespace Awareness and Information Operations Program Office) under PEO C4I (Program Executive Office Command, Control, Communications, Computers, Intelligence and Space Systems), contract number N0003922IP00013. Some Seascape capabilities were developed with funding from the Air Force Air Combat Command under contract number F3QCAZ0161G101. Approved for unclassified unlimited release (UUR) by Sandia National Laboratories, SAND Number SAND2022-10391 C.

REFERENCES

1. W. Cunningham, "The WyCash Portfolio Management System," *SIGPLAN OOPS Mess.* **4**, p. 2930, dec 1992.
2. D. Sculley, G. Holt, D. Golovin, E. Davydov, T. Phillips, D. Ebner, V. Chaudhary, M. Young, J.-F. Crespo, and D. Dennison, "Hidden Technical Debt in Machine Learning Systems," in *Proceedings of the 28th International Conference on Neural Information Processing Systems - Volume 2, NIPS'15*, p. 25032511, MIT Press, (Cambridge, MA, USA), 2015.
3. S. Chacon and B. Straub, *Pro Git/1.3 Getting Started - What is Git?* Apress, 2 ed., 2014.