

# **NAML : An Easy-to-Deploy Application for Time Series Analysis**

A Major Qualifying Project  
Submitted to the Faculty of Worcester Polytechnic Institute  
In partial fulfillment of the requirements for the  
Degree in Bachelor of Science  
In Computer Science  
By

---

---

Patrick Houlihan

Date : 4/28/2022

Project Advisors:

---

Professor Rodica Neamtu  
Professor Erin Solovey

This report represents the work of one or more WPI undergraduate students submitted to the faculty as evidence of completion of a degree requirement. WPI routinely publishes these reports on the web without editorial or peer review. For more information about the projects program at WPI, please see <https://www.wpi.edu/academics/undergraduate/major-qualifying-project>.

# Table of Contents

<b>Table of Contents</b>	<b>1</b>
<b>Abstract</b>	<b>3</b>
<b>Introduction</b>	<b>4</b>
<b>Background</b>	<b>6</b>
2. 1. Neuroimaging Techniques	6
2. 2. sktime	8
2. 3. Describing NAML	8
2. 3. 1. Initial State of NAML	9
2. 3. 2. NAML Documentation	11
2. 3. 3. NAML Dataset and Configurations	12
2. 3. 4. User Interface	14
2. 4. Continuous Integration	15
2. 5. Docker and Containers	16
<b>Methodology</b>	<b>18</b>
3. 1. Continuous Integration	18
3. 2. Documentation	19
3. 3. Refactoring and Cleaning	21
<b>Results</b>	<b>23</b>
4. 1. Installation Requirements	23
4. 2. Running Remotely	23
4. 3. Continuous Integration with sktime	24
4. 4. Classifier Removal	25
4. 5. Documentation	25
4. 6. User Interface	26
4. 7. Tooltips	27
4. 8. Helper Text	28
<b>Case Study</b>	<b>30</b>
5. 1. Dataset Choice and Conversion	30
5. 2. The Test Configurations	32
5. 3. Results	34
5. 4. Summary	39
<b>Future Work</b>	<b>40</b>
<b>Conclusion</b>	<b>41</b>

<b>Works Cited</b>	<b>43</b>
<b>Appendix</b>	<b>45</b>
Appendix. A. Classifier Table in sktime vs NAML	45
Appendix. B. NAML User Interface Guide	48
Section A. Batch Job Name and Data	48
A1. Job Name	48
A2. Target Column	48
Section B. Upload CSV Data	48
B1. CSV Upload	49
Section C. Configure Your NAML Batch Job	49
C1. Name Configuration	49
C2. Test Config Name	49
C3. Logging Enabled	50
Section D. Batch Job Classifiers	50
D1. Oversampling	50
D2. Classifier	50
D3. Plus Button	51
D4. Run Batch Classifier Job	51
D5. Save Config to Database	51
Section E. Results	52
E1. Results Port	52
E2. Save Test to Database	53
E3. Download Result	53
Section F. Header	53
F1. Help	53
F2. About	54
Appendix. C. NAML Installation Guide	55
NAML Installation	55
Windows	55
Step 1 : Windows Docker Prerequisites	55
Step 2: Installing Docker	59
Step 3: Installing NAML	59
Step 4: Running NAML	60
Linux	61
Step 1: Installing Docker	61
Step 2: Installing NAML	61
Step 3: Running NAML	62

## **Abstract**

This project expanded the capabilities of NirsAutoML (NAML), an application developed at WPI dedicated to using the *sktime* library for multivariate time series classification of fNIRS data. NAML was used solely as a command line interface. It was not synced with the *sktime* library, resulting in the classifiers used in NAML lacking improvements as *sktime* matured. The application now exposes a remotely hosted frontend for access by researchers, improved code quality for future maintenance, documentation for installation, use, and development, and a Docker container that streamlines the installation and deployment process. A case study for NAML comparing classifier efficacy by channel feature type has been performed on the “fNIRS to Mental Workload” dataset provided by Tufts University [1].

## Introduction

The goal of this project was to expand upon the work of prior Major Qualifying Projects (MQP) at Worcester Polytechnic Institute (WPI) that worked on NirsAutoML (NAML), a program made to classify time series data. This project extended the features and capabilities of NAML to provide a user interface, a Docker container, documentation associated with changes, and improvements to NAML from the installation, developer, and user perspectives; it also included continuous integration with the open source Python library *sktime*. This library provides a toolbox of machine learning classifiers with tunable parameters to run on time series data [2].

Researchers at WPI conduct studies on human brain activity using functional near-infrared spectroscopy (fNIRS) data. A typical experiment will have a participant use an fNIRS montage, a configuration of optodes placed upon the head measuring change in concentration of micromolar hemoglobin within the brain over a selected time period. The participant completes tasks while wearing the fNIRS montage on a cap; the data produced can then be analyzed to detect and isolate patterns of brain activity during the task. This data is multivariate time series data; at every interval of measurement, the changes in oxygenation of hemoglobin from many different sections of the brain are measured by each channel. Each channel's measurement must be distinct as different regions of the brain will respond differently to the task. For every experiment, there are periods of time where the participant is reacting to some stimuli, coded in the dataset as an event.

In order to process this data, WPI students have created NirsAutoML, also referred to as NAML. The purpose of NAML is to act as a user-friendly framework for the running of classifiers from the powerful python library *sktime* [2]. The classifiers in *sktime* are made for time series data, but are not explicitly made for fNIRS data, thus, the data must be preprocessed

to bring it to a format *sktime* will accept, necessitating a tool like NAML. In NAML, researchers can define a job to perform on a provided dataset, using a subset of available *sktime* classifiers, specify parameters for these jobs, and receive accuracy data on the effectiveness of the classification model created. NAML is remotely hosted on a server for easy access by researchers. The user experience, onboarding process, and overall functionality have been greatly improved through user interface improvements and addition of new features. The success of this project was measured both by the completion of deliverables and a case study using a fresh installation and testing of the application using a dataset generated outside of the WPI Human Computer Interaction (HCI) lab. Long term success can be measured by faster maintenance turnarounds and integration as *sktime* updates. NAML is currently used in research in the WPI HCI lab, so the improvements to maintainability and user experience will have tangible impacts on active research [3][4].

## Background

### 2. 1. Neuroimaging Techniques

Functional Near Infrared Spectroscopy (fNIRS) is a non-invasive method of measuring brain activity by observing and measuring the changes in concentration of micromolar hemoglobin in the brain. The measurements are taken by an fNIRS montage affixed to a cap on the participant's head, consisting of fiber optic channels that shine non ionizing, near infrared light into the head [5]. A picture of an fNIRS montage can be seen in Figure 1.



Figure 1: A picture of an fNIRS montage on an fNIRS cap. The montage is the series of red and blue optodes on top of the helmet/skullcap which is the fNIRS cap.

An fNIRS montage consists of many channels, each of which consist of a detector and emitter. Light is carried to the participant's head by fiber optic cable in the emitter, shined into the participant's head, then the light absorbed by hemoglobin while refracting and scattering off of other brain tissue which exits the head. The difference in what was deflected back to the detector acts as an estimator of brain activity. The difference in exiting light intensity and

entering light intensity is then proportional to brain activity. A picture of fNIRS data can be seen below [5].

Signal form of channel Channel-1 HbO Channel-4 HbO Channel-7 HbO

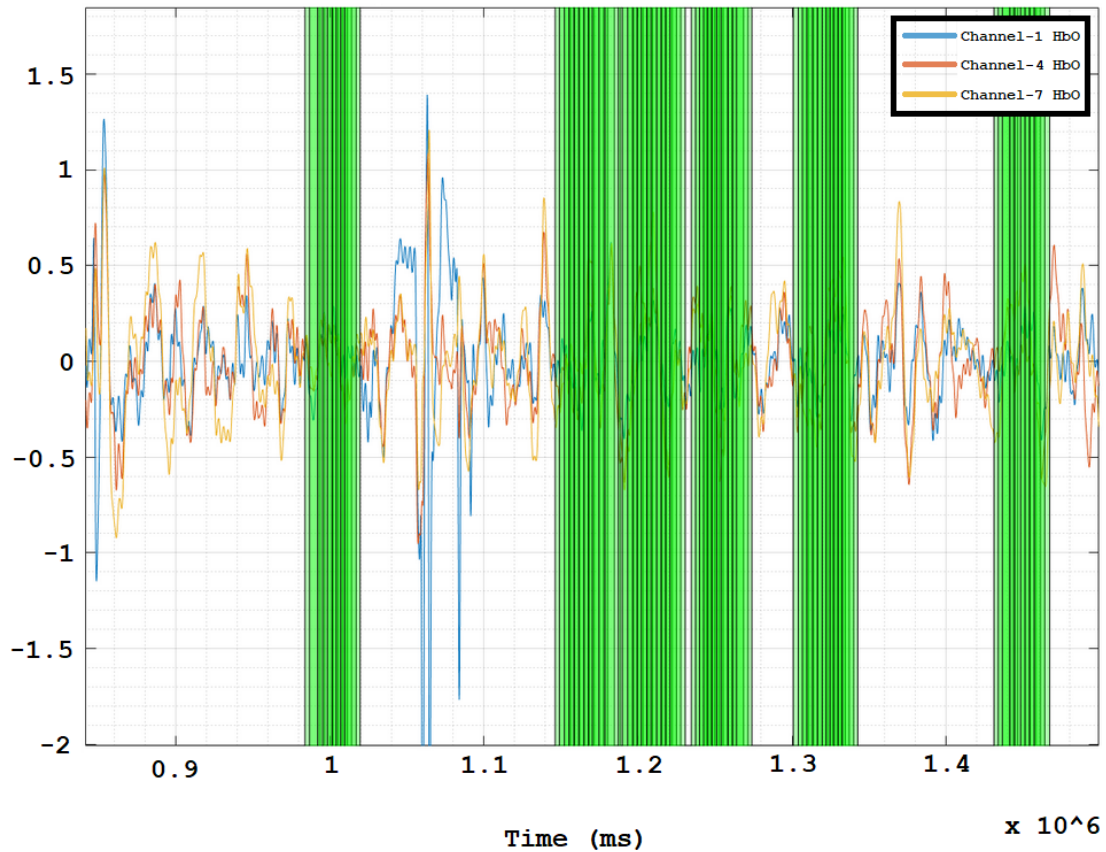


Figure 2: The three colored lines are time series measurements of changes in blood oxygenation observed in three channels. The green highlighted bars indicate a five second period before and after an event occurred. Due to the test structure, there is overlap between the event periods, identified by the areas of higher opacity green shading.

Functional Magnetic Resonance Imaging is another option for measuring the same oxygenation changes in the brain, however the fMRI machine is stationary, giving the fNIRS cap more versatility in measuring different tasks. Another option that can be combined with fNIRS measurement is Electroencephalogram (EEG), which measures electric charges in the brain [6]. While fNIRS measurements need four to seven seconds for the signal to reach peak effectiveness



to measure changes in micromolar hemoglobin, EEG is instantaneous. EEG, unlike fNIRS can not detect precise locations of brain activity.

## **2. 2. *sktime***

fNIRS produces multivariate time series data. At every time interval, the measurements from many channels measuring different regions of the brain are recorded. Typical machine learning resources are not well suited for fNIRS data because data points within a channel must retain both sequence order and order of values within each sequence. This complexity of the data necessitates a tool like *sktime*. The classification methods NAML runs on data produced from brain data experiments come from the *sktime* python library. The *sktime* library provides a toolbox that adapts existing machine learning capabilities to function with both univariate and multivariate time series data. The *sktime* mission statement is to provide a single API that can process any machine learning time series needed and allow extensive parameter customization for deeper control of data analysis [2]. *sktime* provides many features like forecasting and regression for time series data; however, NAML uses classification and transformation of time series primarily. A machine learning classifier takes features of the dataset as input in order to determine what class a target feature belongs to. Classifiers are separated into packages based on their type composition, dictionary, distance, feature, hybrid, interval, kernel, and shapelet.

## **2. 3. Describing NAML**

NirsAutoML (NAML) is a machine learning framework developed at WPI to work with fNIRS time series data. It is written in Python and Javascript, with a Django backend, a React frontend, and a Postgres database associated with it for remote access of datasets. NAML is

designed to make use of the *sktime* python library more efficiently and provide a lower barrier to entry for researchers, allowing them more advanced tools for understanding and interpreting experiment data. The *sktime* library features 27 classifiers, of which 14 are implemented in NAML. In order to work with *sktime*, the data must be reformatted. fNIRS data is multivariate; for every instance of time, there are many channels associated with the event. In order to make effective use of the classifiers provided by *sktime*, NAML was created to logically order a dataset into the structure defined in [Background 2.3.3](#). This allows for classification to be run on a channel to channel basis via `ColumnEnsembleClassifier` or to be run by concatenating each channel together to run the sequence values as univariate data by row. Ensemble methods are preferred because they maintain the data structure closer to the original format in the provided CSV file, applying univariate methods to singular channels rather than treating multiple channels as a single variable through concatenation [3][4].

### **2.3.1. Initial State of NAML**

The initial installation process of NAML required the user to enter multiple terminal commands as it lacked startup scripts to install the prerequisite programs automatically. In order to install, a user would have had to meet the following prerequisites

1. A user should be running the Ubuntu 16.04 LTS operating system.
2. A user must have access to the environment variables required for the project.
3. A user must be added to the NAML private GitHub repository.

After meeting these prerequisites, a user would then install NAML on their local machine to process configuration files in a command line application by following the steps outlined below.

1. Clone the NAML repository by following steps listed on the NAML GitHub repository.

2. Enter the cloned NAML repository root on the local installation.
3. Install Python 3.8.9 on the system.
4. Find the path associated with the Python installation from the previous step.
5. Run ``pipenv --python <path to python>`` to create a virtual Python environment.
6. Run ``pipenv shell`` to enter the virtual environment.
7. Run ``sudo apt-get install python3.8-dev``
8. Run ``pipenv install -r requirements.txt`` in project root.
9. Change directories to `naml_backend/naml_django/naml`.
10. You can now run `naml.py` as a command line application in the form ``python naml.py <path to configuration file>``

The requirements outlined in the `requirements.txt` file, used by Python's Pip package manager, were outdated, and specified *sktime* version 0.4.2 rather than 0.7.0, the version the program was written to use. The NAML application did not run on a remote server, and it must have been run locally by a user that has access to the GitHub repository. NAML featured a UI, however it contained incomplete documentation on how to run the Django backend necessary for the UI to function. *sktime* had undergone a major version update to 0.10.1 which impacted some implemented classifiers [3][4].

## 2.3.2. NAML Documentation

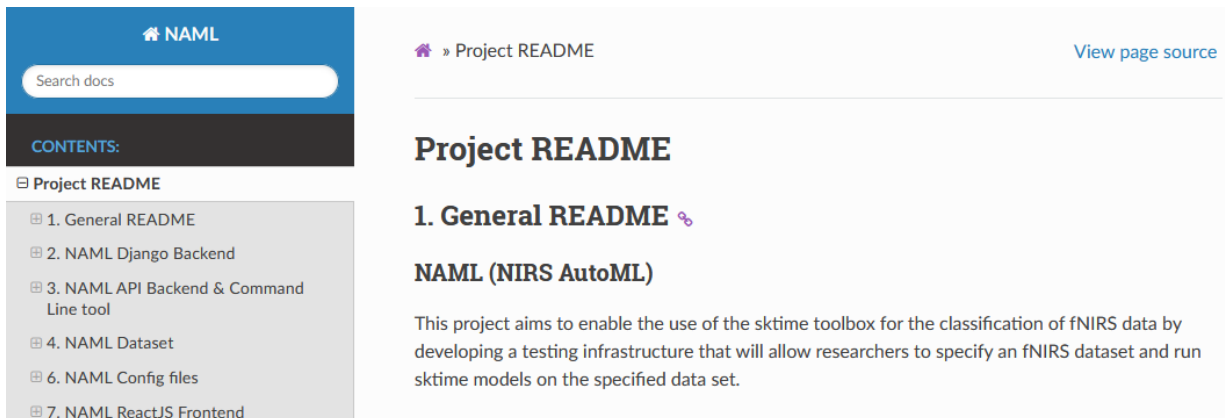


Figure 3: A snapshot of a section of the NAML Project README page generated with Sphinx, bringing users to a description of the project with a sidebar indicating sections of the document.

NAML had a wide variety of documentation available in the repository. There was a seven part README for the project that covers basic background about the project structure and information about the tools used to create it such as React, Django, *sktime*, and Postgres. The README for NAML is generated with Sphinx, a python library which uses reStructured Text to generate documentation for projects. There was initially a section describing how to write a job configuration file for processing; this is how NAML is invoked on a specified dataset. A comparison between algorithms supported by NAML and the algorithms on offer from *sktime* as of major version 0.7.0 is provided in Appendix A.1. The function documentation format for the project is Google Style Doctoring, which uses a string of format description, arguments, returns, and errors [7]. An installation guide was provided which is suitable for a first time user with the exception of the version incompatibility from the requirements file. Using this documentation, a user should be able to install the application with minor troubleshooting after figuring out the versions required for the requirements.txt file. The user will only be able to run the command line application by following the guide [3]. Further documentation was provided in the form of

external documents in Google Docs, Slides, and PDF documents. These governed sample workflows, lists of supported algorithms at the time of writing, and examples of configuration files.

### 2.3.3. NAML Dataset and Configurations

name	event	channel	start time	end time	1	2	3	4	5	6
sub_1		0 AB_I_O	0	1	-0.19829	-0.19565	-0.19279	-0.18994	-0.1874	-0.18548
sub_1		0 AB_PHI_O	0	1	0.335677	0.29721	0.275041	0.271573	0.288043	0.32442
sub_1		0 AB_I_DO	0	1	-0.56079	-0.58071	-0.60045	-0.61975	-0.63838	-0.65609
sub_1		0 AB_PHI_D	0	1	-0.05604	-0.03217	-0.01237	0.002192	0.010818	0.013304
sub_1		0 CD_I_O	0	1	0.460231	0.524553	0.585149	0.641092	0.691434	0.735207
sub_1		0 CD_PHI_O	0	1	0.271119	0.208284	0.162781	0.133816	0.119983	0.119547
sub_1		0 CD_I_DO	0	1	-0.64589	-0.68137	-0.71663	-0.751	-0.78375	-0.81415
sub_1		0 CD_PHI_D	0	1	0.198804	0.202136	0.194758	0.176938	0.149429	0.113429
sub_1		0 AB_I_O	1	2	-0.18994	-0.1874	-0.18548	-0.1845	-0.18472	-0.18633
sub_1		0 AB_PHI_O	1	2	0.271573	0.288043	0.32442	0.37938	0.450381	0.533808

Figure 4: An example dataset used with NAML demonstrating the required feature columns and a subset of value columns. This dataset is from the “fNIRS to Mental Workload” datasets from Tufts University [1].

NAML requires datasets containing five feature columns and at least one value column. The feature columns in order are name, event, channel, start time, and end time. The value columns are some amount of columns that are ordered chronologically indicating the measurement of changes in micromolar hemoglobin in that region at that time. The headings for these value columns are sequential, starting from one until the maximum sequence length. Channels may also be different measurement types in the same region, though they should be distinct from each other. NAML processes these datasets by creating a dataframe where the participant name and start time identify rows while the channel identifies the column. The cells contain lists of readings for a given sequence, which is the time series values recorded for a

given channel starting at a given time for a given participant. The values must not be zero for any value column.

```
{
  "filepath": "~/NAML/naml_backend/naml_django/naml/data/SART.csv",
  "logging": true,
  "target_col": "event",
  "n_splits": 2,
  "jobs": [
    {
      "classifier": "TimeSeriesForestClassifier",
      "parameters": [
        "n_estimators",
        15,
        "min_samples_split",
        3,
        "min_samples_leaf",
        2
      ],
      "oversample": true
    }
  ]
}
```

Figure 5: This image shows a sample configuration file written in JSON, featuring examples of properties for the TimeSeriesForestClassifier.

The configuration files for NAML must be written in JSON. The top level JSON properties that all configuration files required were: filepath or csvdata, logging, target\_col, and jobs. Filepath identifies the file location on the local machine that NAML code is being executed on. This could be replaced by csvdata which provides the file data representing a dataset in the form of a string to be processed by NAML as a csv file. Logging indicated to NAML whether a logfile will be created. A logfile would, on successful completion of a NAML job, contain information regarding the result accuracy of the classifier, the classifier parameters, and the time it had taken to complete the classifier, for each classifier in the jobs JSON property. Target\_col indicated to NAML what column would be used as the event column. Lastly, jobs was an array of JSON objects representing classifiers to be run in succession. A classifier was represented by

a JSON object with the required property classifier, and the optional properties: parameters, oversampling, and ensemble\_info. The classifier property identified the classifier being used from the *sktime* library. The parameters property identified an array of properties the user sought to override compared to the properties specified in the backend of the form [property\_name, property\_value...]. The oversampling property indicated whether underrepresented classes in the dataset would be resampled to improve dataset balancing. Lastly, ensemble\_info was a JSON array used for ColumnEnsembleMethod and stored classifier JSON objects inside it.

#### **2.3.4. User Interface**

The user interface for NAML was written in React as a single page application (Figure 3). It provided an interface to add and run tests. The app was not usable due to a lack of environment variables connecting it to the Django backend. A goal of this MQP was to link the user interface to the backend and ensure it was hosted on the server for easy access. The user interface featured large blocky sections outlining the actions by category. The three categories are Test Information, Test Configuration, and Results. The Test Information section highlighted the Test Name field as a required field regardless of user input. The Test Information and Test Configuration sections featured large blocks surrounding the buttons and text fields associated with the section. We removed these to limit unnecessary clutter on the interface. The Test Configuration section had many classifiers that had no exposed, tunable parameters, a major feature of the *sktime* toolbox. Without tunable parameters, researchers were limited to the default parameters set on the backend. It also featured the ColumnEnsemble classifier, which had no default parameters on the backend, and thus always errored out.

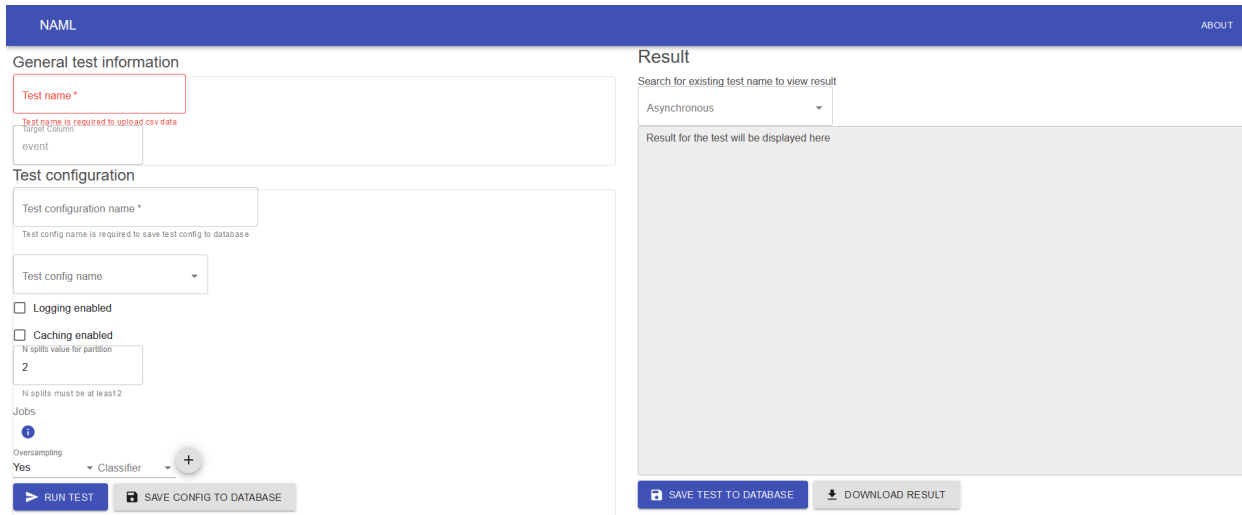


Figure 6: The initial state of the NAML User Interface. This interface was non functional nor hosted remotely before this project began.

## 2. 4. Continuous Integration

*sktime* is an actively developed python library where a stated goal is to add new meta-estimators. This will increase the customization of jobs as more classifiers and parameters are added to the NAML suite. Currently, in the NAML application, there is no infrastructure to identify when there are new *sktime* updates. In order to accomplish this, a Continuous Integration pipeline should be considered. Continuous Integration is the process by which changes can be incorporated from other code sources in an automated capacity [8]. NAML currently uses GitHub workflows to accomplish a test suite on new pull requests. GitHub workflows can run automated procedures based on events in the repository. Another common tool used in software development and continuous integration is Docker. Docker is a software that creates containers for applications: a standalone image of an operating system separate from the host operating



system. By moving NAML to a container, it will be more portable to other platforms that may have other programs with similar, conflicting stacks [9].

## **2. 5. Docker and Containers**

Containers are lightweight, executable software packages that are designed with specific softwares in mind. Application maintenance is more consistent as Docker containers are designed to outsource as much as possible to the instruction file from which the container was built. This is beneficial because it takes much of the installation process out of the end user's control. NAML is hosted on a server with another tool named BrainEx [10]. NAML and BrainEx both have separate managers for their python environments: NAML uses Pipenv and BrainEx uses Conda. NAML will be moved to Conda as a consistent standard to make maintenance easier; containers allow for very fine grained management of the environment specific to that container. The similar tech stack and environment managers allow for easy adaptation between Dockerfiles as the two applications will share layers. In the case of NAML and BrainEx, both use React, Conda, and Django, meaning the Dockerfile instructions to install these prerequisites can be repeated.

While other container systems like Singularity exist, Docker has a litany of tutorials and instructions on common tasks for the tool due to its wider support. Docker has a strong support ecosystem featuring 2.9 million installations of the tool [11] with many available first and third party resources for tutorials [12]. Using the most supported tool is beneficial to decrease time spent on development and increase scope. A Docker container is built from a Dockerfile. This Dockerfile can pull existing Docker images and iterate upon them to create a more specific container or build a new container from scratch using shell scripts and internal package managers

to the Operating System being run. We used Ubuntu 16.04 as the internal operating system in the Docker Container, as that was the original version NAML was designed for.

These managers create a virtual environment where python and python dependencies for a program can be managed and controlled to ensure standard versions and simple addition of new dependencies. A benefit to standardizing the environment manager was that large portions of the Dockerfile could be reused, allowing for starter code for future applications that may be hosted on the remote server.

## Methodology

We used a Docker container to implement NAML on a remote server. This allows NAML to maintain a separate environment so there is no conflict with other tools or services hosted on the same server. We created installation scripts for NAML to streamline the installation process through the use of shell scripts inside of the Docker container. These scripts download the correct dependencies for a stable version of *sktime*, version 0.10.1, and the latest version of *sktime*.

### 3. 1. Continuous Integration

Continuous Integration was introduced to the application at two levels: 1) the GitHub repository to the server level, and 2) the *sktime* library to the application level. Changes pushed to the GitHub repository are deployed to the server. This Continuous Integration method was already implemented in the NAML application structure, however it did not work due to a dead SSH key. This was fixed by reimplementing the CI/CD pipeline with a fresh SSH key for deploying changes on the server and converting the existing actions to work with the Docker build [13].

The second Continuous Integration method was *sktime* to the server. This was a novel introduction to NAML. It allowed for updates to *sktime* to be reflected in NAML by automatically attempting a new build of NAML when an *sktime* update is detected. If the build fails, it will roll back to a previous stable version, alert the user that the tool needs to be updated, and keep a log of the errors encountered. This allows NAML to quickly receive the benefits of *sktime* updates as the library improves over time while also avoiding any updates that would break the tool. From the last update of the installation to the time of this report, there have been four major version changes. Historically, major version changes have affected the library

package structure, thus breaking NAML's usability upon update. These types of changes require manual intervention to fix the advanced branch of NAML.

### **3. 2. Documentation**

The documentation contained no information about the environment variables needed to run the application and many links were nonfunctional. The varied formats of documentation was a challenge, as developers needed to view multiple sources to view comprehensive NAML documentation. Moving as much of this documentation as possible to one unified source was an important goal of this project. The original documentation was laid out with the following sections:

For users:

1. General README, describing the rationale behind NAML.
2. NAML Django Backend, describing the tech stack used with an example of a python test.
3. NAML API Backend & Command Line Tool, this section links to *sktime* information and describes how to run the command line application.
4. NAML Dataset, describes the nature of NAML data with an explanation of packaged datasets.
5. NAML Config Files, this section describes how to write many types of NAML jobs.
6. NAML ReactJS Frontend, this section describes React and the techniques used.

For developers:

1. A section linking to the NAML documentation sections for users.

2. A section containing instructions on how to expand documentation using Sphinx.
3. A section describing the installation process.
4. A section describing the existing GitHub workflow Continuous Integration pipeline.
5. A section linking back to section users:3.
6. A section describing the steps NAML takes to process a dataset.
7. A section describing potential installation issues.

The documentation was pruned to remove extraneous sections like developers:1 and developers:5. This streamlined the documentation for easier user interaction. The users section and developer section should not have been distinct. The section users:2 was more appropriate for a backend maintainer of the tool, while the sections developers:3 and developers:7 needed to be accounted for in the general users section since the tool may need to be installed by both users and developers. In order to fix this, the [documentation](#) was separated into the following sections:

### [NAML Background](#)

1. General README, describing the rationale behind NAML.
2. The NAML Tech Stack, a brief working description of
3. NAML Datasets, describing the type of data used and the requirements.
4. NAML Config Files, what needs to be written to run a job.
5. Previous Work, linking to prior MQP work.

### [NAML Installation](#)

1. Prerequisites, a description of the needed prerequisites for installing NAML
2. Installing Docker, a description of the install process for Docker on Windows, MacOS and Linux.

3. Installing NAML using Docker, the required instructions needed to be entered to install.
4. Accessing NAML Remotely, the server to access to run NAML
5. Configuring Docker, a section on changing resource allocation to Docker to increase processing power.
6. Potential Installation Issues, the errors that may occur during the installation process.

### NAML Developer's Guide

1. Dockerfile, a section expanding upon the Dockerfile with annotated sections describing how it works.
2. GitHub Continuous Integration, a section describing the GitHub Workflow used to manage testing and reflecting changes on the server.
3. *sktime* Continuous Integration, describing the script used to manage *sktime* updates.
4. Environment Variables, a description of the variables needed to run the project and where to get them.
5. Django Backend and React Frontend, talking about how the two connect to each other.

### 3.3. Refactoring and Cleaning

The classifier mapping worked previously by identifying the classifier type, setting the default parameters for that classifier, overwriting the parameters specified by the job configuration, then running the classifier. This code was repeated for every classifier, adding an overhead of 6 lines of code per classifier to the NAML.py file. By creating a dictionary of key classifier names and values of a tuple containing the function call for the classifier and the default parameters for the classifier, the lines of code are significantly reduced.

The code was also refactored to improve error handling and error feedback. The datasets used by NAML must fit a strict format, and must have a consistent sequence length for all

channels and sequences. If the dataset does not have this, obtuse numpy errors from deep within the *sktime* library get thrown, making the issues very difficult to track. NAML was changed to ensure that the error type is more clear and the user is provided with advice on what to change in the dataset in order to make it work with the NAML structure.

## Results

### 4. 1. Installation Requirements

There are now three prerequisites required to run NAML as a command line application: 1) a deploy key must be provided by a NAML administrator, authorizing cloning of the repository, 2) the environment variables file must be provided by a system administrator to complete the Docker build, and 3) Docker must be installed on the user machine. Documentation governing the installation of Docker on Windows and Linux is provided in [Appendix 3](#). These instructions should leave users with a working Docker container, in which they can run `naml.py` with existing test configuration files, or create their own. Two HCI lab members tested installing Docker and NAML using the instruction files, one with Windows 10 and one with Ubuntu 16.04, and both were able to run a simple test configuration file. If a user would like to use the NAML application via web browser, instructions for launching the application's frontend and backend have also been included in the documentation provided at <http://brainex.wpi.edu:8090/docs>.

### 4. 2. Running Remotely

By connecting to the WPI HCI lab server via SSH, a user can run a working command line version of NAML within the Docker container. The steps to run the container are identical to that of the local container, however the Docker installation is already completed on the server, and the Docker container is already built. Instructions are provided to update the Docker container with a simple two-step process. This allows end users to skip all the installation setup, simply run the container, activate the Conda environment, and start processing datasets. Information on moving datasets into the container is provided in the documentation in [Appendix Item].



The Docker container on the server can also be accessed via web browser at the address <http://brainex.wpi.edu:3030>. This link brings users to an updated web application built using React and Django to run NAML commands on server hardware as opposed to a user's local hardware. More information on changes and improvements made to the user interface can be found in section 4.6. End users can store configuration files on the server, create custom jobs with varied parameters, and access *sktime* documentation. The user interface is designed to be accessible to a researcher that may be uncomfortable working with command line applications and seeks to keep users informed of options provided in creating a custom NAML batch job.

#### 4.3. Continuous Integration with *sktime*

On creation of the Docker container, a fresh installation of NAML is run using **install\_script.sh** in the root of the repository. This script creates two Conda environments. NAML\_dev is an advanced version of *sktime* using a Conda managed *sktime* module, where the dependencies are automatically updated when the script is run. NAML\_stable is a stable version which copies the advanced environment and reinstalls *sktime* at the static version 0.10.1, with the dependencies specific to that version. This allows users to attempt to use *sktime*'s improvements as the library updates, while also providing a safe fallback to ensure the tool can still be used. If there are any complications in the updating process, it's a one line command to switch between versions, ``conda activate NAML_stable``. If the entire installation breaks during usage, the Docker container can always be rebuilt as shown in the installation guide.

#### 4. 4. Classifier Removal

As *sktime* has updated, it has dropped support for various algorithms. MrSEQL is one such algorithm that was removed during the update cycle of *sktime*. MrSEQL was deprecated in the *sktime* version 0.9.0 that removed Cython from the project, replacing its use with Numba, a different python library focused on converting python into machine code for performance increases. The justification for this change was that Cython caused a buggy installation when used with *sktime*, and the distance measured and other previously implemented methods in Cython were replaced with Numba versions [14]. Other removed algorithms are proximity based algorithms such as ProximityForest, ProximityTree and ProximityStump all of which have issues in their implementation, making them inconsistent. During testing of the algorithms, errors on existing datasets while using default parameters were recorded, suggesting that removal would be prudent to ensure all implemented classifiers offered in NAML are functional.

#### 4. 5. Documentation

The documentation material for NAML uses Sphinx to build easily readable HTML documentation out of both Markdown and reStructured Text files. The documentation structure for NAML has been expanded to include a formal installation instructions section. All documentation referring to removed features has also been removed. A list of supported classifiers has been included. The goal of the documentation improvements is to move as much as possible to the documentation packaged within the NAML repository, ensuring that future maintainers will not have to search far for necessary documentation. By moving more documentation into the repository, it is more likely that the documentation will change as NAML

matures further, allowing maintainers both a place to read documentation about existing features but also to add documentation about features in the future during further work on the project.

#### **4. 6. User Interface**

The NAML user interface has been improved and extended as the application has been made available on the url <http://brainex.wpi.edu:3030>. The application UI has been improved to extend exposure of information to the user, featuring dynamic tooltips generated for each classifier on implementation, and helper text provided on each classifier parameter. Classifiers and their parameter configuration fields now stack vertically, allowing use of the user interface on a half-width browser window. The CSV reader area of the interface has been replaced with an explicit upload icon and solid border to improve readability.

## 4. 7. Tooltips

### NAML


#### Batch Job Name and Data

Job Name \* i **Name your Batch Job Results**

Target Column i

event

#### Upload CSV Data



#### Configure your NAML Batch Job

Name Configuration \* i

Test config name i **Search for Existing Configuration**

Logging enabled i

Caching enabled i

#### Batch Job Classifiers i

Oversampling

Yes i

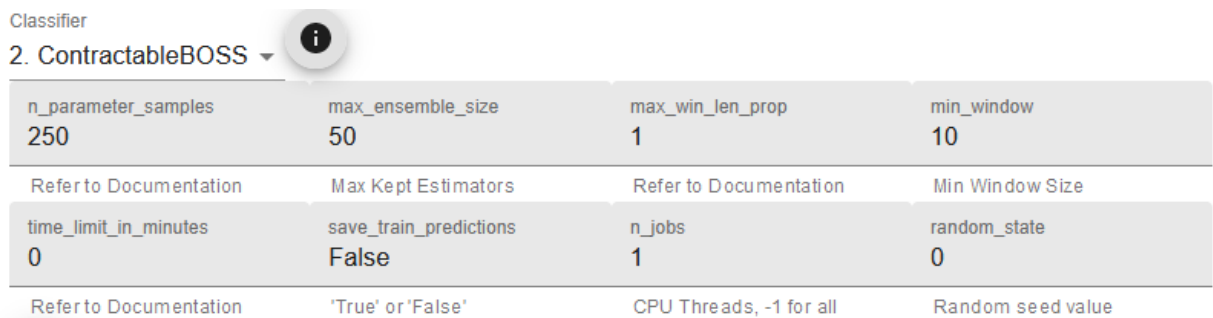
Classifier

2. ContractableBOSS i **Get more Information on Classifier by Clicking**

Figure 7: Sample hoverable tooltips are shown, providing further context to the button or text field.

Tooltips are hoverable icons that provide a user with extra information when using a tip. They have been implemented in the NAML user interface using Material UI InfoIcons. By interacting with an InfoIcon, a user can hover for more information about the adjacent user interface element. This allows for optional information display based on user need. Consistent iconography allows users to identify where they can get more information on their tasks while not cluttering the interface with unnecessary static text. The color of the icon indicates the type of information the icon provides. Blue InfoIcons indicate an external link that provides more information on the adjacent element, while gray InfoIcons identify hoverable text information. This distinction is important because simple text is sometimes not enough to clarify the purpose of the element. External tooltips are used to link to the API reference page for selected classifiers comprising the NAML Batch Job [15]. When InfoIcons are unsuitable for describing the adjacent element, as in the classifier parameter configuration, helper text can be used to provide a user with more context to their actions.

#### 4. 8. Helper Text



The screenshot shows a dropdown menu for the classifier set to "2. ContractableBOSS". Below it is a table of tunable parameters. A tooltip icon is visible over the "max\_ensemble\_size" parameter. The table has the following content:

n_parameter_samples	max_ensemble_size	max_win_len_prop	min_window
250	50	1	10
Refer to Documentation	Max Kept Estimators	Refer to Documentation	Min Window Size
time_limit_in_minutes	save_train_predictions	n_jobs	random_state
0	False	1	0
Refer to Documentation	'True' or 'False'	CPU Threads, -1 for all	Random seed value

Figure 8: Helper text is provided under each tunable parameter TextField. For max\_ensemble\_size, a short description was provided while for max\_win\_len\_prop, “Refer to Documentation” was shown instead. For save\_train\_predictions, the available options were listed below to hint to users what valid input is.

The API reference page on the *sktime* website provides brief descriptors for each classifier tunable parameter. These descriptions have been made more clear when able in helper text provided below the parameter entry field. This helper text is generated from a parameters dictionary provided in the classifier frontend implementation, where parameters can either have a helptext value provided or not, in which case the helper text defaults to “Refer to the Documentation”. This “Refer to the Documentation” is used for some descriptions that would either be too long or are too complex to condense without further context. A user may click on the InfoIcon provided next to the classifier name selector to open a new tab leading to the API reference page for that named classifier. Examples of classifier parameters that have been reworded in helper text are “CPU Threads, -1 for all” for the `n_jobs` field. By changing the CPU thread amount, it enables parallel jobs resulting in faster runtimes for classifier jobs. This helper text is important for providing users with example options for fields that do not have obvious parameters such as words or negative numbers in an integer field. By displaying the helper text when able, a user does not always need to refer to another webpage, and can instead potentially get necessary information without disrupting their NAML-based workflow.

## Case Study

### 5. 1. Dataset Choice and Conversion

In order to demonstrate the opportunity afforded to WPI researchers by the easy to access and use NAML tool, a case study has been performed on a dataset from the “fNIRS to Mental Workload” dataset provided by Tufts University [1]. This dataset is composed of multivariate time series data of a different format than the WPI datasets packaged and prepared for NAML. This dataset requires conversion into a NAML-friendly state. This dataset serves as a very large, openly available fNIRS dataset intended for researchers looking to benchmark classifiers. The dataset also provides for consideration of changes by demographics, as the dataset comes with rich demographic information about the participants involved in the experiments. The brain data of 68 participants is provided in this dataset of varying sequence lengths and fidelity, and the case study for this is on a specific subject at a set level of fidelity. By showing the effectiveness of NAML on a dataset with this level of user demographics information, size, and multiple channel measurement types, researchers can ask a variety of questions. In this case study, the effectiveness of classifying between oxygenated and deoxygenated hemoglobin differently was asked, but researchers could easily use NAML to compare classifier ensembles trained using one subject and those trained on other subjects. The chosen file for this test is sub\_1.csv, the readings for the first subject in the trial, with a sequence time of five seconds and sequence length of 25 measurements. This provides 2143 evenly distributed events across four labels, 0, 1, 2, 3, which is good for machine learning while also ensuring fast results for comparison between other classifiers and parameters.

In order to convert the dataset to a NAML-friendly version, the datasets that NAML uses and the dataset provided must be compared. A NAML dataset must have five feature columns: name, event, channel, start time, end time, and at least one value column, indicating the time

series measurement at that moment. The chosen dataset for the case study featured ten columns, eight of which were permutations of the brain region, the measurement type, and the blood oxygenation concentration of interest. These are equivalent to different channels in the existing NAML datasets. The next two columns are chunk and label. Label corresponds to the event column of a NAML dataset. Chunk indicates a sequence, and the sequence length is 25. For each chunk of 25 rows, the readings for each of the channels are appended as a new column, the chunk number leads to the start and end times being generated. NAML is only concerned with differing start times when identifying rows, so as long as the chunks start at unique times, this will ensure that the dataset will be properly nested. Finally, a column will be added identifying the name of the subject as “sub\_1”.

By converting the dataset to something that NAML can process into an *sktime* compliant format, the classifiers can be run on a channel by channel basis, allowing researchers to compare the effectiveness of algorithms if channels are included or excluded, or compare many different combinations of classifiers. An example chosen for this case study was comparing a base case of running only one classifier on all channels independently, to running one classifier type on oxygenated hemoglobin and another classifier type on deoxygenated hemoglobin. This would allow researchers to take the provided dataset and examine the strength of individual features for prediction. Classifiers were run with set random states to ensure results were reproducible. Classifier jobs were run within the Docker container for NAML using the command line interface for the application. Jobs were multithreaded when able using the `n_jobs` parameter. To limit the amount of testing, the default parameters were used whenever multiple classifiers were being compared. The tests were run on a Ryzen 7 5800X processor with 32 GB RAM inside of a Docker container with a WSLconfig file allocating 16 GB and 8 processors to the container.



## 5. 2. The Test Configurations

In order to run the tests on the dataset for benchmarking accuracy values and amount of time spent during the training of the classifier, a few test configurations were created. For certain algorithms, the observed runtimes during the development and testing process of NAML necessitated running overnight as the jobs would take hours to complete. These tests were separated by expected runtime into two files. The benefit of separating the two allows for indicating which classifiers may be best to tune parameters while the other classifiers may require using multiple instantiations of NAML on separate hardware to obtain results promptly. The configuration files used the default parameters with three exceptions, the `n_jobs` field was changed to two, the `random_state` field was changed to one, and the number of splits was set to three. By setting `n_jobs` to two, *sktime* classifiers were able to use multiple CPU cores, decreasing runtime at the cost of higher resource usage. Instability was observed when running sequential jobs with `n_jobs` set to negative one, which indicates usage of all cores. By setting `random_state` to one, it allows for reproducible results. This will allow researchers wishing to confirm the results to do so. The increased number of `n_splits` results in more test/train splits which increases the precision of the accuracy values if the random state is changed, giving a better idea of the true performance of the algorithm on the benchmark dataset.

## Accuracy and Time of Classifier Combinations

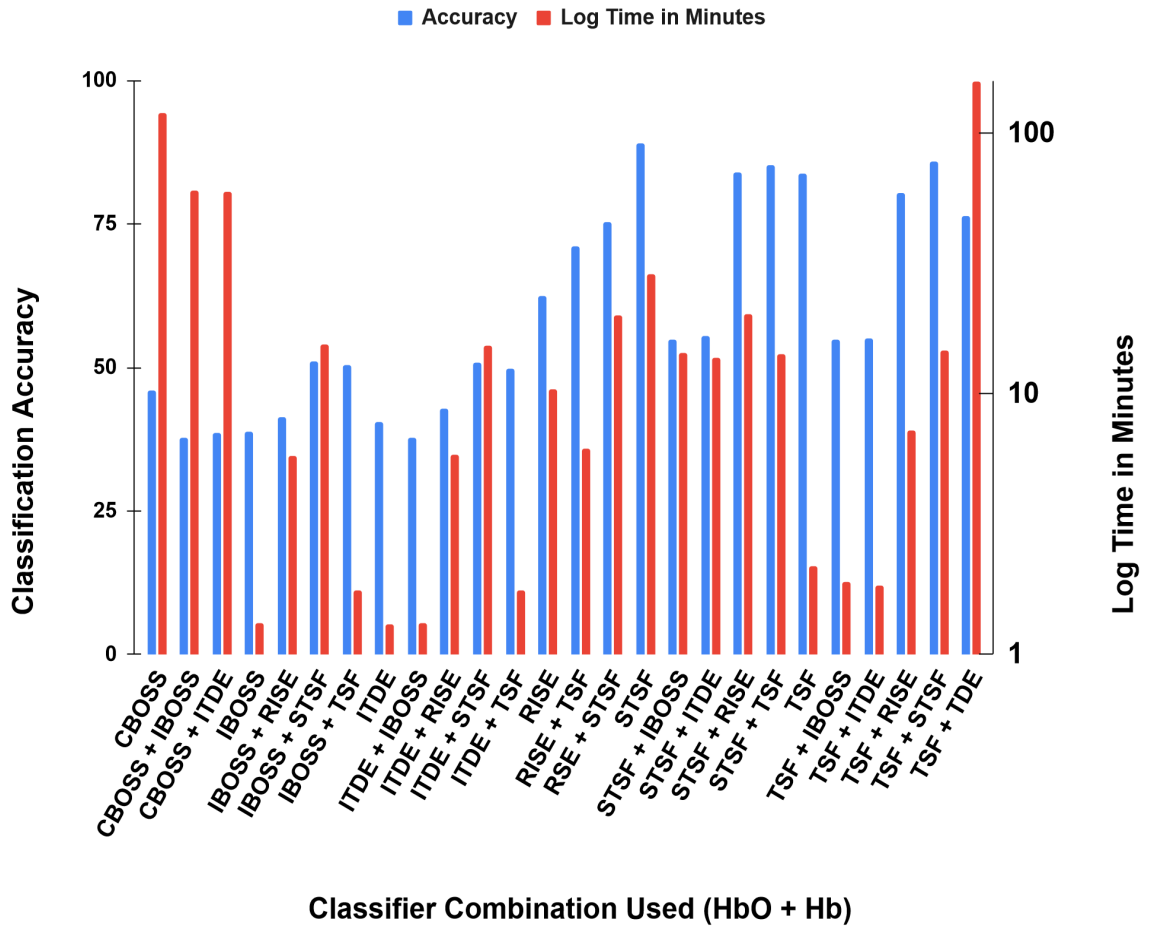


Figure 9: This chart shows the results of a large group of Classifier Ensembles using the ColumnEnsembleClassifier. The Classifier combination indicates that the first classifier was used on HbO data while the second was on Hb data.

Classifier	Accuracy	Time	Time in Minutes
STSF	89.08	1736.94	28.95
TSF + STSF	85.81	884.57	14.74
STSF + TSF	85.21	852.98	14.22
STSF + RISE	83.99	1215.3	20.26

TSF	83.86	130.54	2.18
TSF + RISE	80.45	435.86	7.26
TSF + TDE	76.34	9544.19	159.07
RSE + STSF	75.41	1198.05	19.97
RISE + TSF	71.16	369.47	6.16
RISE	62.44	624.06	10.4
STSF + ITDE	55.53	831.38	13.86
TSF + ITDE	55.16	110.17	1.84
STSF + IBOSS	54.92	861.21	14.35
TSF + IBOSS	54.88	113.94	1.9
IBOSS + STSF	51	929.14	15.49
ITDE + STSF	50.86	922.3	15.37
IBOSS + TSF	50.4	105.61	1.76
ITDE + TSF	49.7	105.46	1.76
CBOSS	45.96	7159.08	119.32
ITDE + RISE	42.79	350.34	5.84
IBOSS + RISE	41.39	348.22	5.8
ITDE	40.46	78.69	1.31
IBOSS	38.73	78.96	1.32
CBOSS + ITDE	38.59	3603.2	60.05
CBOSS + IBOSS	37.84	3607.61	60.13
ITDE + IBOSS	37.84	79.61	1.33

Figure 7: This table indicates the raw results of the ColumnEnsembleClassifiers ensembles run in Figure 6.

### 5. 3. Results

In total, twenty-six classifier combinations were run on the dataset, representing seven of the total fourteen classifiers implemented in NAML, those being ContractableBOSS (CBOSS), IndividualBOSS (IBOSS), IndividualTDE (ITDE), RandomIntervalSpectralEnsemle (RISE), SupervisedTimeSeriesForest (STSF), TemporalDictionaryEnsemble (TDE), and

TimeSeriesForest (TSF). These classifiers represent most of the interval based classifiers and dictionary based classifiers, allowing for comparisons in the efficacy between the two categories as a whole while also provided finer grained information about performance within the dataset when applying different classifiers to the oxygenated and deoxygenated hemoglobin data. In Figure 6, the different accuracy values observed for the test and the runtimes associated with them are shown. The first classifier shown on the horizontal axis on lines with two classifiers joined by a plus sign indicates that that classifier was used on oxygenated hemoglobin data while the second classifier was used on deoxygenated hemoglobin data. The classifiers are sorted by accuracy.

The most successful classifiers observed were SupervisedTimeSeriesForest, and ensembles of TimeSeriesForest when applied to oxygenated or deoxygenated hemoglobin and SupervisedTimeSeriesForest applied to the opposite. The accuracies of these classifier combinations were 83.86% and 85.81% respectively with runtimes of 2.17 minutes and 14 minutes respectively. This is an important part of NAML, as it was shown that using TSF on each channel scored within 2.3% of TSF + STSF with a 6.77 times shorter runtime. This allows for researchers to better allocate time on creating a more accurate model by choosing to change TSF parameters to observe changes in the accuracy values as a first step since it has a far cheaper runtime cost associated with it, allowing more jobs with varied parameters to be run in the same time as it would take to run fewer STSF jobs. An extreme example of this is in the difference between TSF + TDE and TSF + RISE. By switching the second classifier, similar accuracy values were achieved at the cost of over two hours of runtime as opposed to under eight minutes of runtime. Since writing the configuration files for this test is so simple, researchers can do an investigative batch job, return at a later time and then determine which classifiers may be worth

investigating more.

The interval based classifiers tended to outperform the dictionary based classifiers, with the exception of TemporalDictionaryEnsemble showing promise, but at extreme cost in runtime. As it has similar accuracy when paired with TSF as RISE, which shows strong results when run alone or in tandem with other classifiers, it was not pursued for further comparison due to the significant time cost associated with it. After running these tests, the two most promising classifiers, STSF and TSF were paired with other classifiers to identify differences in effectiveness between the two across a large range of combinations to identify which classifier improved the model accuracy of the ensemble more. This comparison type is possible due to the reproducibility of results afforded by setting `random_state`. The four channels used with a classifier such as IndividualBOSS will fit the same and the four channels fit by TSF and STSF will be different because they are different classifiers.

## Accuracy and Time of Classifier Combinations

Comparing STSF and TSF Performance in Various Ensembles

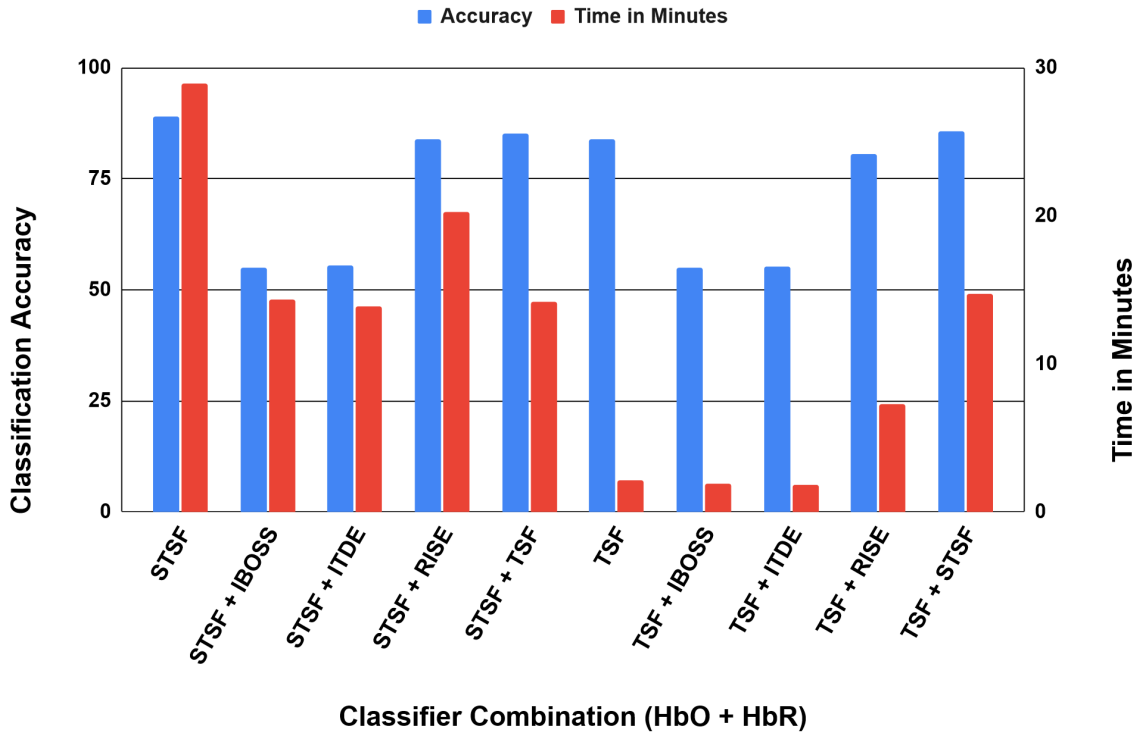


Figure 10: STSF and TSF classifiers used with ensembles of IBOSS, ITDE, and RISE.

STSF beat TSF by slim margins in most comparisons. The highest increase in accuracy came from using entirely STSF over using entirely TSF on all channels, however this only yielded a six percent increase in classification accuracy. The increase in time spent running the classifier was very noticeable, with STSF taking 13.27 times longer to run when used on all channels. Since TSF is significantly faster to run than STSF, a researcher may decide that this time cost is not worth it for tuning parameters if they are hardware bottlenecked, and they may choose to pursue tuning TSF to try and get STSF accuracy without the STSF runtime. Replacing TSF with STSF when using RISE on deoxygenated hemoglobin channels yielded a 4.4%

accuracy increase while replacing the two in other classifiers yielded a difference of fewer than a percent.

**Accuracy and Time of Classifier Combinations**  
Comparing Accuracy on Oxygenated and Deoxygenated Hemoglobin Channels

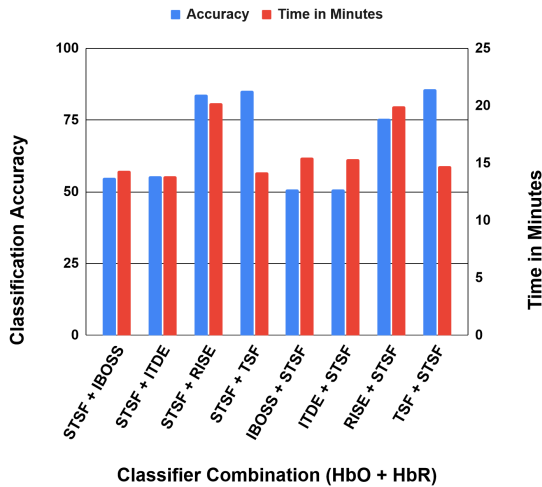


Figure 11.A: This chart shows the difference in observed classification accuracies and runtimes when the classifiers are run on HbO + HbR data then swapped. The classifier of interest in this chart was STSF.

When comparing the performance of classifiers when applying the same classifiers to oxygenated or deoxygenated data, differences are observed. STSF and TSF were used to compare the positions because they were the most effective classifiers, implying that they may have a bigger impact if either channel group was more significant when determining the class with the algorithm. This was shown to be the case, with STSF scoring between 7 and 11 percent better classification accuracy when run on oxygenated data as opposed to deoxygenated data with the exception of TSF. TSF also scored similarly with improvements on non STSF classifiers between 8 and 13 percent. TSF outscored STSF when run on oxygenated micromolar hemoglobin data by less than a percent, which is not a significant change.

**Accuracy and Time of Classifier Combinations**  
Comparing Accuracy on Oxygenated and Deoxygenated Hemoglobin Channels

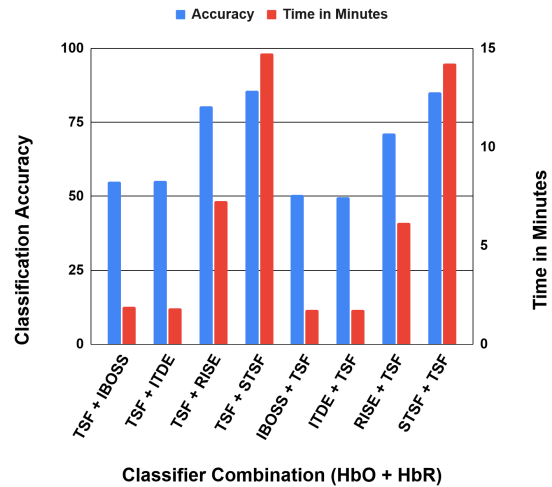


Figure 11.B: This chart shows the difference in observed classification accuracies and runtimes when the classifiers are run on HbO + HbR data then swapped. The classifier of interest in this chart was TSF.

## 5. 4. Summary

Using the NAML application, the performance of *sktime* classifiers on a high quality, publicly available benchmarking dataset was observed in a convenient and easy to write way. The dataset was converted into a proper NAML format following the process detailed in the Dataset Choice and Conversion [Section 5.1](#). Though the dataset at first glance looks very different from a dataset packaged with NAML, the 2013e dataset, the process was simple and easily repeatable by a researcher that understands the differences in formatting. Twenty five tests were run on this dataset using the `ColumnEnsembleClassifier` feature of *sktime* that NAML assembles using JSON configuration files. The `ColumnEnsembleClassifier` is ideal for this dataset because it features a multitude of different channels and it allows for classifiers to be run on a per channel basis as opposed to the results for all channels at a specific time being concatenated. These tests featured multithreading when able to improve runtime and were run sequentially on one computer. The tests also feature set random states to allow for reproducibility, and three validation splits to increase precision in the accuracy results.



## Future Work

Further improvements to NAML can be pursued. Allowing saving of models after the fit step in the NAML pipeline would allow models trained on one participant to then be applied to other participants in an experiment to see how effective models work from user to user. It would also save time, as a computationally expensive model to fit on a large dataset with many participants would not have to be retrained every time a new participant is added. Expansion of the output from NAML to include a confusion matrix would show which events tend to be misclassified by the algorithm, which may have value to researchers. Integration with other HCI lab tools would increase potential for NAML, as a dataset could be mined with BrainEx then examined in NAML. The ability to generate tests with varied parameters would allow researchers to avoid having to click and add each parameter in the User Interface or manually type each classifier out in a configuration file. The *sktime* library allows researchers to change parameters for the algorithm very easily, and this ease is reflected in NAML, but giving users a simple workflow where many different configurations of a chosen classifier can be tested and examined may be very helpful to push researchers towards specific parameters over others.

## Conclusion

Through the work on this MQP, the NAML application has been improved in multiple ways. The application has been made very simple to install, run as a command line interface, or deploy as a webapp. As the application is now in a Docker container, it is more portable and will work on systems where Docker works. The installation process creates an advanced Conda environment that features dependencies for the latest version of *sktime* and a stable Conda environment that has been tested throughout this MQP and used in the case study presented in this report. The user interface has been improved to expose more information to users within the page, while also providing users easy access to external information such as the *sktime* API reference page. The classifiers implemented have been curated to ensure that users only work with functional classifiers. A case study has been performed showing the process a researcher might take to select a dataset, convert it, and process classifiers on it during the use of the NAML tool. The potential of the tool was demonstrated, showing the effect different classifier combinations had on classification accuracy, as well as demonstrating the effect of running classifiers on different features.



## Works Cited

1. Huang, Z., Wang, L., Blanley, G., Slaughter, C., McKeon, D., Zhou, Z., Jacob, R., & Hughes, M. C. (2021). The Tufts fNIRS Mental Workload Dataset & Benchmark for Brain-Computer Interfaces that Generalize. *Neural Information Processing System Track on Datasets and Benchmarks*. <https://openreview.net/pdf?id=OzNHE7QHhut>
2. Löning, M., Bagnall, A., Ganesh, S., Kazakov, V., Lines, J., & Király, F. J. (2019). *sktime: a unified interface for Machine Learning with Time Series*. LearningSys. [https://learningsys.org/neurips19/assets/papers/sktime\\_ml\\_systems\\_neurips2019.pdf](https://learningsys.org/neurips19/assets/papers/sktime_ml_systems_neurips2019.pdf)
3. Ikram, F. (2019). *NirsAutoML: Building an automated classification platform for fNIRS data*. : Worcester Polytechnic Institute.
4. Buntel, E. (2020). *Brain Wave Analysis*. : Worcester Polytechnic Institute.
5. Naseer, N., & Hong, K. (2015). Fnirs-based brain-computer interfaces: A review. *Frontiers in Human Neuroscience*, 9. <https://doi.org/10.3389/fnhum.2015.00003>
6. John Hopkins Medicine. (2021). *Electroencephalogram (EEG)*. <https://www.hopkinsmedicine.org/health/treatment-tests-and-therapies/electroencephalogram-eeeg>
7. Google. (n.d.). *Pyguide*. Google Styleguide. <https://google.github.io/styleguide/pyguide.html>
8. Atlassian. (n.d.). *What is continuous integration*. <https://www.atlassian.com/continuous-delivery/continuous-integration>
9. Docker. (2021, October 5). *Dockerfile reference*. Docker Documentation. <https://docs.docker.com/engine/reference/builder/>

Bagnall, T. (2021). *sktime*. Time Series Classification Website.

<https://www.timeseriesclassification.com/sktime.php>

10. Clements, M., Powell, J., & Nolan, A. (2021). *BrainEx: Visual Exploration and Discovery in Time Series Data*. : Worcester Polytechnic Institute.

11. Kriesa, J. (2020, July 30). *Docker index: Dramatic growth in docker usage affirms the continued rising power of developers*. Docker Blog.

<https://www.docker.com/blog/docker-index-dramatic-growth-in-docker-usage-affirms-the-continued-rising-power-of-developers/>

12. Tutorialspoint. (2021). *Docker tutorial*. Biggest Online Tutorials Library.

<https://www.tutorialspoint.com/docker/index.htm>

13. GitHub. (2022). *Generating a new SSH key and adding it to the ssh-agent*. GitHub Docs.

<https://docs.github.com/en/authentication/connecting-to-github-with-ssh/generating-a-new-ssh-key-and-adding-it-to-the-ssh-agent>

14. Alan Turing Institute. (2021, October 18). *[ENH] remove all use of cython, use numba instead* · Issue #1538 · alan-Turing-institute/sktime. GitHub.

<https://github.com/alan-turing-institute/sktime/issues/1538>

15. sktime. (2021). *Multivariate time series classification with sktime — sktime documentation*. Welcome to sktime — sktime documentation. Retrieved October 6, 2021, from

[https://www.sktime.org/en/stable/examples/03\\_classification\\_multivariate.html](https://www.sktime.org/en/stable/examples/03_classification_multivariate.html)

## Appendix

### Appendix. A. Classifier Table in *sktime* vs NAML

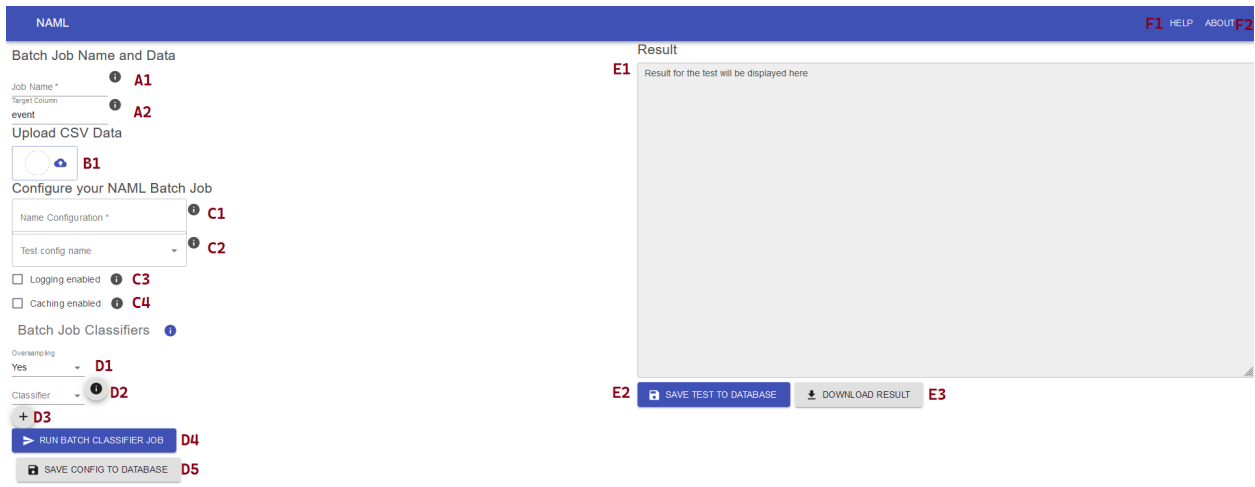
Classifier	Classifier Family	Implemented in NAML?
ColumnEnsembleClassifier (CEC)	Composition	Yes
IndividualBOSS	Dictionary-based	Yes
BOSSEnsemble	Dictionary-based	Yes
ContractableBOSS	Dictionary-based	Yes
WEASEL	Dictionary-based	Yes
MUSE	Dictionary-based	Yes
IndividualTDE	Dictionary-based	Yes
TemporalDictionaryEnsemble	Dictionary-based	Yes
KNeighborsTimeSeriesClassifier	Distance-based	No, There are issues established with the move away from Cython towards Numba. Distance measurements used in this method are not converted yet.
ElasticEnsemble	Distance-based	Yes
ProximityForest	Distance-based	No, There are issues established with the move away from Cython towards Numba. Distance measurements used in this method are not converted yet.
ProximityTree	Distance-based	No, There are issues established with the move away from Cython towards Numba. Distance measurements used in this method are not converted

		yet.
ProximityStump	Distance-based	No, There are issues established with the move away from Cython towards Numba. Distance measurements used in this method are not converted yet.
HIVECOTEV1	Hybrid	Yes
Catch22Classifier	Feature-based	No, This was never implemented in NAML prior to this MQP and it did not fall into the scope of the project.
MatrixProfileClassifier	Feature-based	No, This was never implemented in NAML prior to this MQP and it did not fall into the scope of the project.
TFreshClassifier	Feature-based	No, This was never implemented in NAML prior to this MQP and it did not fall into the scope of the project.
SignatureClassifier	Feature-based	No, This was never implemented in NAML prior to this MQP and it did not fall into the scope of the project.
TimeSeriesForestClassifier	Interval-based	Yes
RandomIntervalSpectralForest	Interval-based	Yes
SupervisedTimeSeriesForest	Interval-based	Yes
CanonicalIntervalForest	Interval-based	No, This was never implemented in NAML prior to this MQP and it did not fall into the scope of the project.

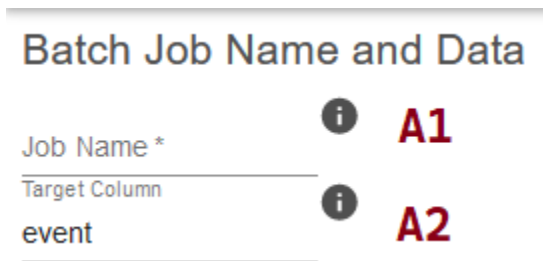
DrCIF	Interval-based	No, This was never implemented in NAML prior to this MQP and it did not fall into the scope of the project.
ROCKETClassifier	Kernel-based	No, This was never implemented in NAML prior to this MQP and it did not fall into the scope of the project.
Arsenal	Kernel-based	No, This was never implemented in NAML prior to this MQP and it did not fall into the scope of the project.
ShapeletTransformClassifier	Shapelet-based	Yes



## Appendix. B. NAML User Interface Guide



### Section A. Batch Job Name and Data



#### A1. Job Name

This text field allows the user to name the test results that can get saved onto the server. Every batch job will output some results on the results port detailed in [Appendix B. E. E1](#). This required field associates a name with those results should a user choose to save the batch job output.

#### A2. Target Column

This field allows a user to select the column being predicted by the NAML models. Currently this should not be changed from event, as the backend can not support it.

### Section B. Upload CSV Data

## Upload CSV Data



### B1. CSV Upload

This section allows you to click to initiate an upload dialogue for a CSV file. By dragging and dropping a file into the outlined field, a user may upload that file to the server. The files should be CSV files that have the required columns detailed in the associated MQP paper linked in [Background 2. 3. 3](#). These columns are name, event, channel, start time, and end time. These columns may then be followed by separated time series values that fall between the start and end time in any number of additional columns.

## Section C. Configure Your NAML Batch Job

### Configure your NAML Batch Job

A screenshot of a web form titled "Configure your NAML Batch Job". The form contains three main elements: a text input field labeled "Name Configuration \*" with a red callout "C1" and an information icon; a dropdown menu labeled "Test config name" with a red callout "C2" and an information icon; and a checkbox labeled "Logging enabled" with a red callout "C3" and an information icon.

### C1. Name Configuration

This textfield allows a user to name the configuration settings of the Batch Job. In Section Appendix A the available classifiers are detailed. This name field allows a user to load an existing configuration or save a new configuration as a configuration of the given name for later retrieval.

### C2. Test Config Name

This textfield allows a user to search for an existing named configuration and load it into the NAML UI application fields in [Appendix B. D. D1](#) and [Appendix B. D. D2](#)

### C3. Logging Enabled

This checkbox makes the server log the data from the NAML core process and save a log file for later examination.

### Section D. Batch Job Classifiers



#### D1. Oversampling

This yes or no dropdown menu allows a user to select if oversampling will be done for a given method. Oversampling is the process by which underrepresented classes of results in the test subset of the dataset are repeatedly sampled in order to balance the appearance of each class for training an accurate model. This is a good option if you have a large number of one class and a small number of another class of event.

#### D2. Classifier

This dropdown box opens a menu of all supported classifiers of NAML. These classifiers will then allow a user to change parameters depending on the algorithm. BOSSensembles for example is pictured below.

## Batch Job Classifiers i

Oversampling

Yes v

Classifier

1. BOSSensembles i

n_parameter_samples	threshold	max_ensemble_size	max_win_len_prop	time_limit
250	0.92	500	1	0
min_window	max_window	random_state	n_jobs	
3	10	0	1	





Oversampling

Yes v

Classifier i



 RUN BATCH CLASSIFIER JOB

 SAVE CONFIG TO DATABASE

Changing these parameters will send them to the NAML Core backend where they will change how the model is trained. For a detailed list of parameters, see the InfoIcon next to the Classifier Name, which will link to the sktime documentation for that classifier.

### D3. Plus Button

NAML Batch Jobs may consist of many classifiers that run in sequence. The Plus Button allows a user to add another classifier to their batch job. To remove a classifier, a user may press the Minus button that appears upon click of the Plus Button. The Minus Button appears at the bottom of the classifier parameter list.

### D4. Run Batch Classifier Job

This button allows a user to send their batch job configuration to the server and run it using NAML Core. The output of the results can be seen in [Appendix B. E. E1](#).

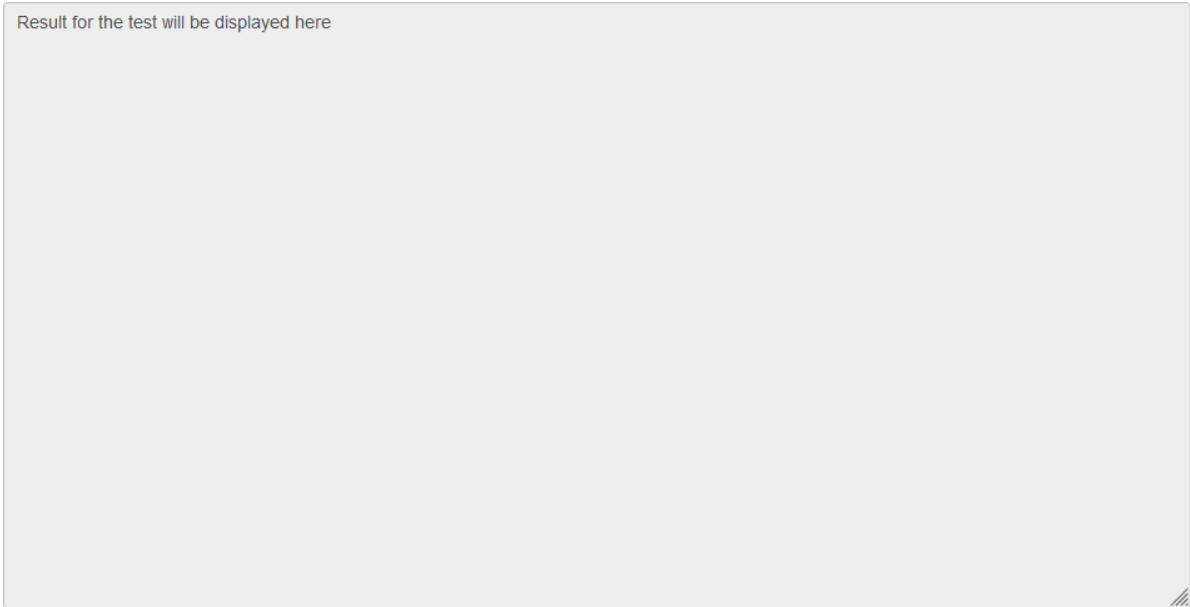
### D5. Save Config to Database



This button allows a user to save their configuration to the database under the name specified in . This is useful if a user has created tuned parameters to observe changes as it can reload them much faster than individually setting each.

## Section E. Results

Result

**E1** Result for the test will be displayed here



**E2**  SAVE TEST TO DATABASE  DOWNLOAD RESULT **E3**


### E1. Results Port

This is the main output area for Batch Classifier Jobs, for each classifier, it will report on the Classifier Name, Oversampling value, a list of parameters as set in the Classifier Parameters section shown in [Appendix B. D. D2](#), an accuracy value showing what percentage of events were properly categorized, and total time in seconds. A delimiter line separates classifiers.

## Result

```
Job: BOSSEnsemble
  oversampling: true
  parameters:
{"id":0,"classifier":"BOSSEnsemble","n_parameter_samples":250,"threshold":0.92,"max_ensemble_size":500,"max_win_len_prop":1,"time_limit":0,"min_window":3,"max_window":10,"random_state":0,"n_jobs":1}
Accuracy: 33.33
Total time: 1.71
-----
```

 SAVE TEST TO DATABASE

 DOWNLOAD RESULT

### E2. Save Test to Database

This button will allow a user to save their result in the database for later retrieval or processing. This would be useful if a user is tuning parameters and needs to compare a set of tests.

### E3. Download Result

This button allows a user to download the results to a text file on their computer. The textfile is of the same format shown in [Appendix B. E. E1](#)

## Section F. Header

### F1. Help

This button links a user to this document to review NAML UI functionality.

## **F2. About**

This button allows a user to view the contributors to the project as well as a brief description of the NAML project.

## Appendix. C. NAML Installation Guide

### NAML Installation

#### Windows

##### Step 1 : Windows Docker Prerequisites

Official Docker Windows Install Instructions : <https://docs.docker.com/desktop/windows/install/>.

Docker Desktop Install:

<https://desktop.docker.com/win/main/amd64/Docker%20Desktop%20Installer.exe> (link copied from link above).

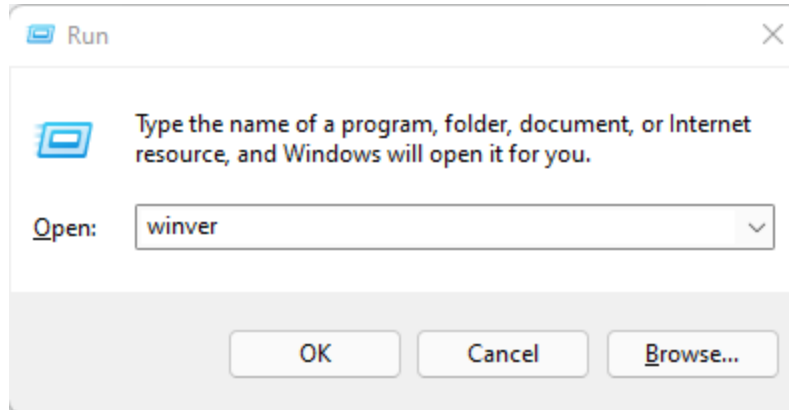
Prerequisites:

- Windows 11 64-bit: Home or Pro version 21H2 or higher, or Enterprise or Education version 21H2 or higher.
- Windows 10 64-bit: Home or Pro 2004 (build 19041) or higher, or Enterprise or Education 1909 (build 18363) or higher.
- Enable the WSL 2 feature on Windows. For detailed instructions, refer to the [Microsoft documentation](#).
- The following hardware prerequisites are required to successfully run WSL 2 on Windows 10 or Windows 11:
  - 64-bit processor with [Second Level Address Translation \(SLAT\)](#)
  - 4GB system RAM
  - BIOS-level hardware virtualization support must be enabled in the BIOS settings. For more information, see [Virtualization](#).
- Download and install the [Linux kernel update package](#).

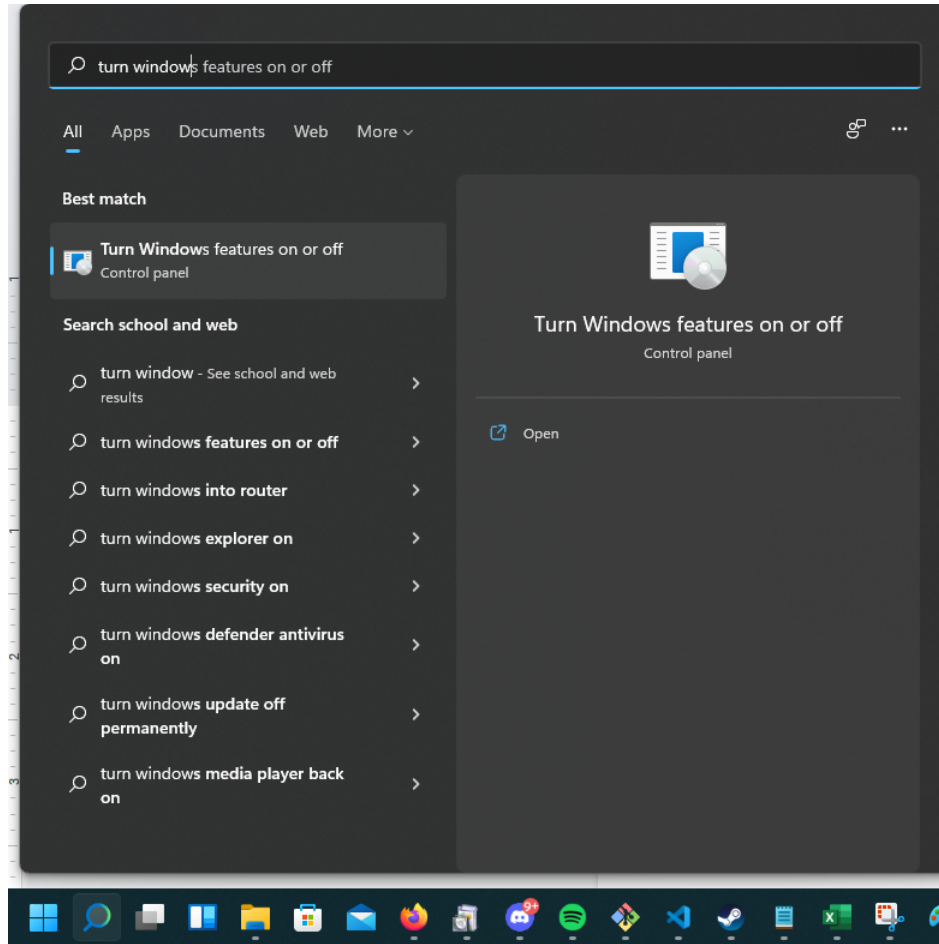
Prerequisites explained

1. Check your windows version by typing run in the windows search dialog, opened by clicking the bottom left corner windows icon.
2. Type in winver in the run dialog box and press enter.

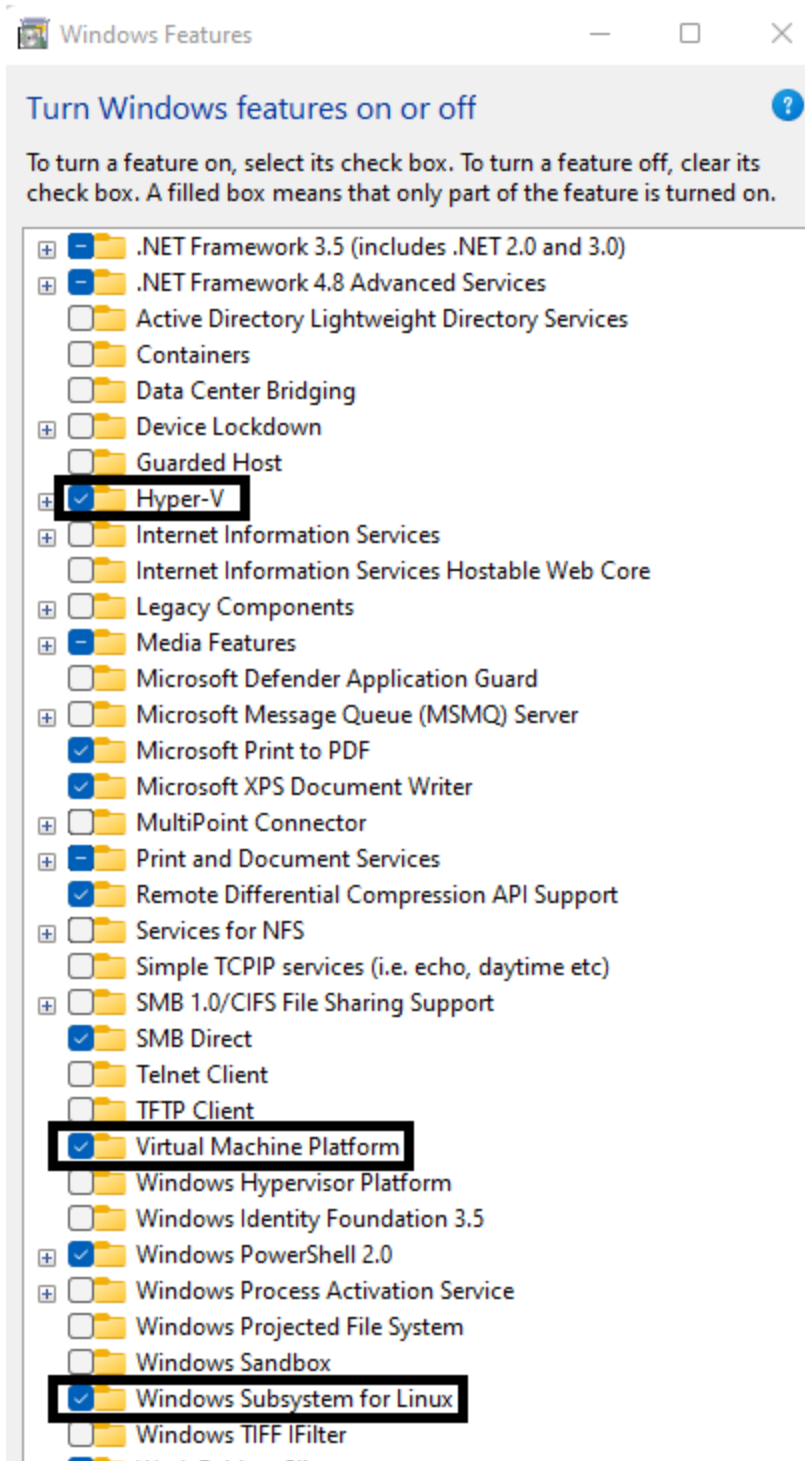




- a. If your version exceeds the required versions then you can continue with installation.
  - b. If your version does not exceed the required versions, you must do a windows update. Do this by typing in the Windows search bar Windows Update and following the dialogue from that link.
3. To enable the WSL2 Feature on Windows, follow the instructions in this link <https://docs.microsoft.com/en-us/windows/wsl/install-win10>.
4. To enable BIOS visualization, you must enter your BIOS menu on computer startup.
  - a. To enter the BIOS menu, you must enter a key on computer startup. This key is different for different motherboard vendors, however F2 is a common one. On computer startup it should display this key so look for it on the splash screen when you restart your computer.
    - i. Look for a section in the bios entitled Virtualization Technology or something to that effect (it is vendor specific) and turn it on.
5. You must make sure to enter the Turn Windows Features On or Off dialog menu.



6. Turn on Windows Subsystem for Linux, Hyper - V and Virtual Machine Platform.



7. In elevated windows command prompt, run the command

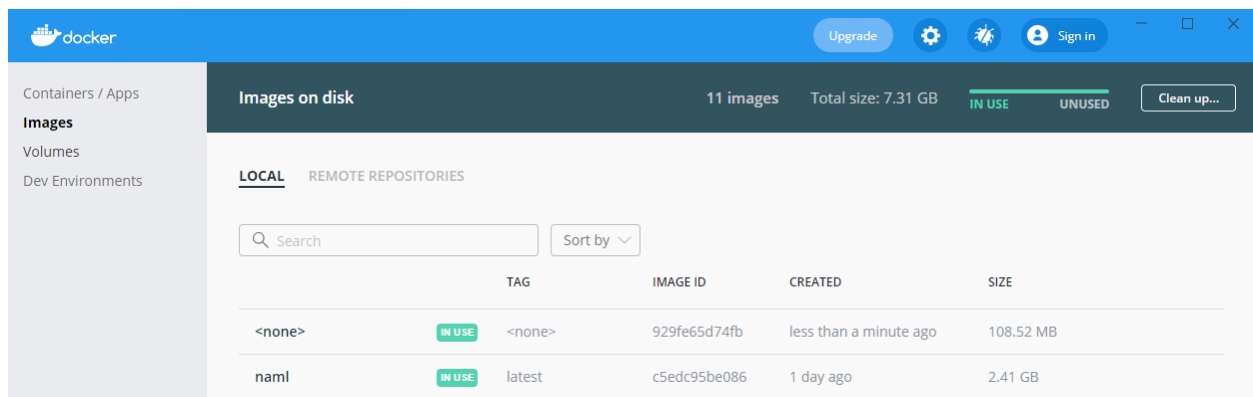
bcdedit /set hypervisorlaunchtype auto

## Step 2: Installing Docker

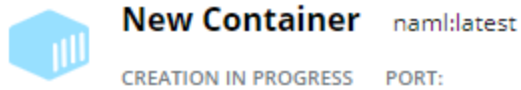
1. Download the Docker Installation executable and run it after ensuring the prerequisites above are completed.

## Step 3: Installing NAML

1. Clone the NAML repository from <https://github.com/WPIHCILab>.
2. Navigate to the root of the cloned directory.
3. Run the command : “ git checkout docker-integration “
4. Open the Docker Desktop Application.
5. Copy the NAML\_ssh file included in the email and place it in the root directory of the cloned repository.
6. Open Powershell and navigate to the root of the cloned directory.
7. Run the command “ docker build -t naml . ” in the root.
8. Navigate to the images tab of the Docker Desktop application.



9. Select the image named naml and click run.
10. In the optional settings, add the name naml and click run.



Optional Settings ^

**Container Name**

**Ports**

Local Host	Container Port	
<input type="text"/>	22/tcp	<span>+</span>

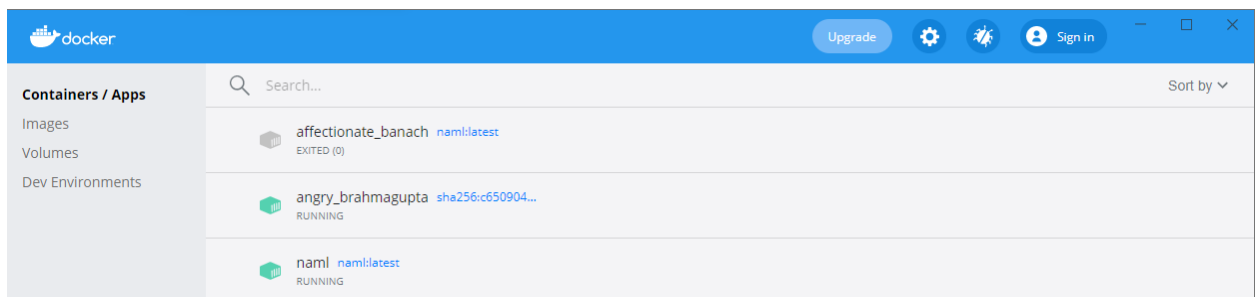
**Volumes**

Host Path	Container Path	
<input type="text"/> ...	<input type="text"/>	<span>+</span>

Cancel Run

#### Step 4: Running NAML

1. Enter the docker container named naml by hovering over the container and clicking the terminal icon.



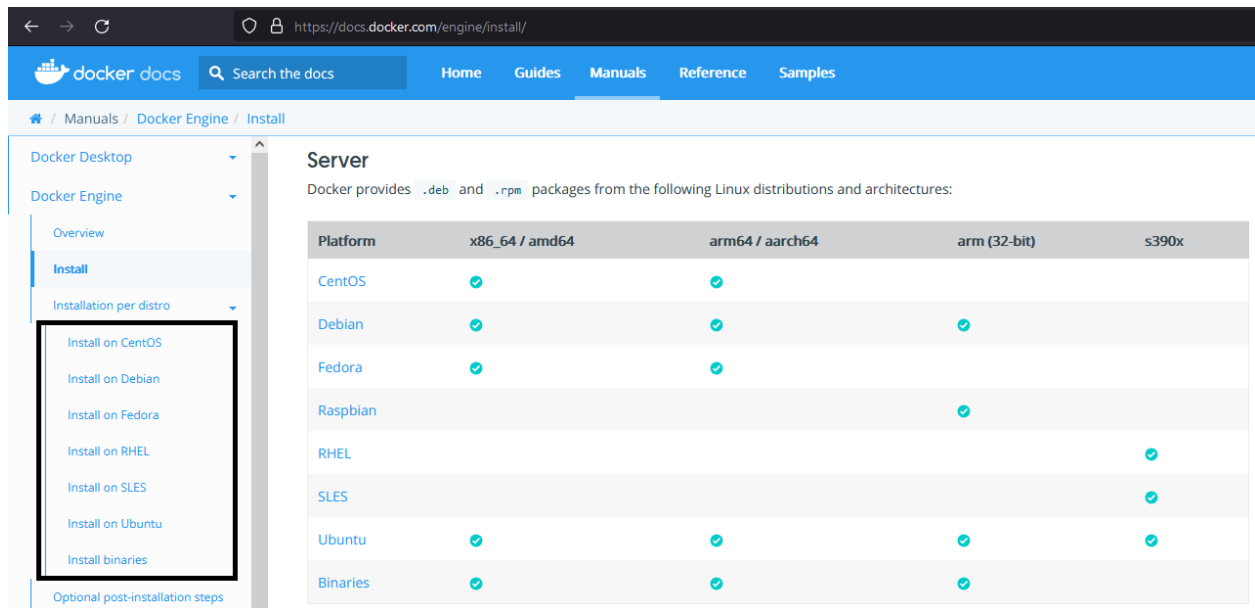
2. When in the CLI, type the following commands
  - a. `/bin/bash`

- b. `conda activate NAML`
  - c. `cd ~/NAML/naml_backend/naml_django/naml`
  - d. `python naml.py configFiles/column_ensemble_example.json`
3. Congratulations! You have successfully executed a NAML job.

## Linux

### Step 1: Installing Docker

1. Installing for linux has instructions provided by Docker that are version dependent. Navigate to this link <https://docs.docker.com/engine/install/> and click on the side bar where it says installation per distro and follow those instructions according to your distro.



### Step 2: Installing NAML

1. Clone the NAML repository from : <https://github.com/WPIHCILab>.
  - a. If you do not have git follow this : <https://git-scm.com/book/en/v2/Getting-Started-Installing-Git>
2. Navigate to the root of the cloned directory.
3. Run the command : “ `git checkout docker-integration` “ in the root of the cloned directory
4. Run the command : “ `sudo systemctl status docker` “
  - a. If active, continue.
  - b. If not active, run : “ `sudo systemctl docker start` “

5. Copy the NAML\_ssh file included in the email and put it in the root directory of the cloned NAML repository.
6. Run the command : “ docker build -t naml . ” in the root.
7. Run the command : “ docker run -t naml “ while still in the root directory

### **Step 3: Running NAML**

1. When in the naml Docker container terminal after step 2.7, type the following commands
  - a. /bin/bash
  - b. conda activate NAML
  - c. cd ~/NAML/naml\_backend/naml\_django/naml
  - d. python naml.py configFiles/column\_ensemble\_example.json
2. Congratulations! You have successfully executed a NAML job.