EPiC Series in Computing

Volume 64, 2019, Pages 194–203

Proceedings of 28th International Conference
on Software Engineering and Data Engineering

Creational and Structural Patterns in a Flexible Machine Learning Framework for Medical Ultrasound Diagnostics

Corey M. Thibeault[1]

Neural Analytics, Inc., Los Angeles, CA
corey@neuralanalytics.com

Abstract

The impact of machine learning in medicine has arguably lagged behind its commercial counterparts. This may be attributable to the generally slower pace and higher costs associated with clinical applications, but also present are the conflicting constraints and requirements of learning from data in a highly regulated industry that introduce levels of complexity unique to the medical space. Because of this, the balance between innovation and controlled development is challenging. Adding to this are the multiple modalities found in most clinical applications where applying traditional machine learning preprocessing and cross-validation techniques can be precarious. This work presents the novel use of creational and structural design patterns in a generalized software framework intended to alleviate some of those difficulties. Designed to be a configurable pipeline to not only support the experimentation and development of diagnostic machine learning algorithms, but also to support the transition of those algorithms into production level systems in a composed manner. The resulting framework provides the foundation for developing unique tools by both novice and expert data scientists.

1 Introduction

Design patterns are an important method for communicating and reusing architectural knowledge in software systems. Although the implementations are not inherently reusable, the formalized definitions provide concepts that increase quality, comprehensibility, maintainability, and testability [7]. However, amongst developers the use of patterns can be divisive – as some patterns can result in an increase in complexity and a reduction in understandability [19]. This is partly due to the cognition involved in appropriately conceptualizing a particular pattern [8], as well as the contextual mismatch between software domain and pattern definitions that can result in poor design choices and a further reduction in maintainability [2]. This is particularly true when dealing with novice developers [3]. Regardless, patterns are an important aspect of any large-scale software project.

Presented here is the application of several classic design patterns in a generic machine learning framework – named Atlas. The codebase was created to assist data scientists developing machine learning algorithms for medical diagnostics. This unique combination of complex data analysis and varying levels of developer experience – data scientists do not always have

extensive software engineering exposure – presents a difficult set of constraints when designing a generalized framework. This paper outlines what makes Atlas unique and how some of these classic patterns – creational and structural – were applied in novel ways.

Creational patterns are a basic form of dependency inversion. Rather than relying on a concrete class definition, the instantiating class can defer the specific implementation to a subclass at runtime. This creates a mechanism for decoupling code in an application – resulting in a codebase that can be easier to extend and more importantly, easier to test. Similarly, structural patterns rely on composition to alter the functionality of an object dynamically. A particularly useful structural design element is the decorator pattern[5]. This provides a mechanism for adding functionality to a class or instantiated object dynamically.

Atlas is the realization of several different patterns. Developed in Python [12], the framework is constructed around a dynamic class import system – allowing the instantiation of data processing blocks at runtime based on user defined configurations. The unique aspect of this is that new building blocks can be added into the processing pipeline by changing the text based configuration. This provides a dynamic development framework for data scientists and facilitates experimentation without consequences to the overarching production level system – the first motivating factor behind developing a proprietary platform, as opposed to using off-the-shelf solutions.

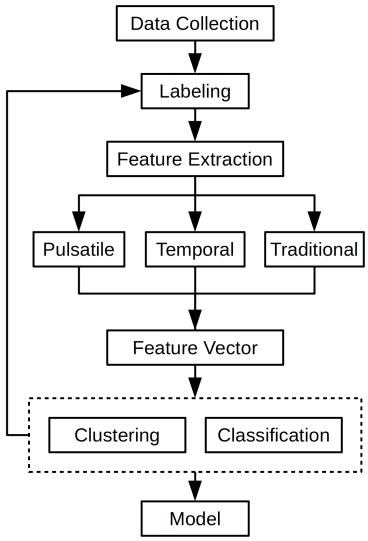
Although Atlas is solving a unique problem, there are several libraries that share similarities. For example, NiftyNet [6], Nuts-ML [9], and DeepNeuro [1] were all developed to handle the preprocessing of diagnostic images exclusively for use in deep learning applications. Similarly, the Deep Learning Toolkit [10] was developed specifically for prototyping deep learning models and modules. Whereas libraries such as NifTK [4] focus on compatibility and interoperability of transferring imaging data. Unfortunately, there were no general purpose machine learning packages available for medical applications at the time Atlas development began.

The remainder of this paper presents the requirements that motivated the development of Atlas along with the configuration system that is the core of the framework. This is followed by an explanation of the creational and structural patterns that are employed. These are accompanied by a motivating example of how an Atlas experiment is invoked. Finally, the existing uses of Atlas and the concluding remarks close out the discussion.

2 Software Requirements

There are three model stages in the workflow Atlas supports – experimentation (selection), evaluation, and production (distribution). These have motivated a system that uniquely fulfills the non-functional requirements of usability, extensibility, and flexibility. While many organizations have departmental separation of these states – where data scientists develop prototype models and software engineers create structured implementations of them – a finite resource pool drove the need for a consistent interface between these stages. Unfortunately, these stages can often introduce requirements in conflict with one another. Atlas was conceived as a way to mitigate those competing requirements while allowing data scientists to write flexible experiments that could be more readily refined for production.

The experiment stage of model development necessitates flexibility and extensibility. The generalized flow of development, outlined in Figure 1, includes preprocessing, cross validation, and evaluation. However, once experiments are fully developed they need to be transferred to a publishable state. The evaluation phase is the initial model release state and provides a framework for reevaluating them and their corresponding experiments as new data is collected. This provides a way for data scientists to continually update models, as well as evaluate their



Listing 1: Basic configuration syntax

```
module_list = module1, module2

[general]
general_bool = False
general_float = 0.65

[module1]
    [[SpecificModule1Implementation]]
    implementation_specific_int = 10

[module2]
module2_variable = String
    [[SpecificModule2Implementation]]
    implementation_specific_int = 125,
    implementation_specific_float = 10.2
```

Figure 1: General experiment flow.

utility, but it requires a consistent shared interface with the experimental framework. The evaluation framework however, requires tested, peer-reviewed, and documented code. Finally, the production system introduces regulatory requirements. This increases the testing and documentation burden of any project. Here, the framework and included models have to be stable and documented through a well-defined product development process. The design elements described in this paper cover the shared core of these three states but the focus and examples are on the initial evaluation stage. Each of these include a different compromise in the balance between flexibility and determinism.

Although there are a myriad of machine learning frameworks available, the development of Atlas began out of the necessity to meet the requirements of the specific three stage model process. One of the major difficulties in providing a system for developing and validating ML models, was in the preprocessing of the data. This is certainly not unique in data science – where formatting and sanitizing data is a significant step in any machine learning application [20]. However, in the case of diagnostic algorithm development, a different set of obstacles emerges. First, are the multiple modalities present in most of the clinical studies generating data. The primary data source in our case is transcranial Doppler (TCD) ultrasound. But most studies include other biological signals, such as end-tidal CO_2 (the amount of carbon dioxide exhaled), absolute blood pressure, EKG, or intracranial pressure. The difficulty in incorporating all of these signals into a ML model is defining a unique uncorrelated feature set. The second source of difficulty arises during cross-validation. In most cases multiple datasets may be generated from a single subject. This is often the case for traumatic brain injuries (TBIs), where a subject is tracked along their recovery. During cross-validation, a simple leave-one-out turns into a leave-one-subject-out – making most off-the-shelf ML packages out of the question.

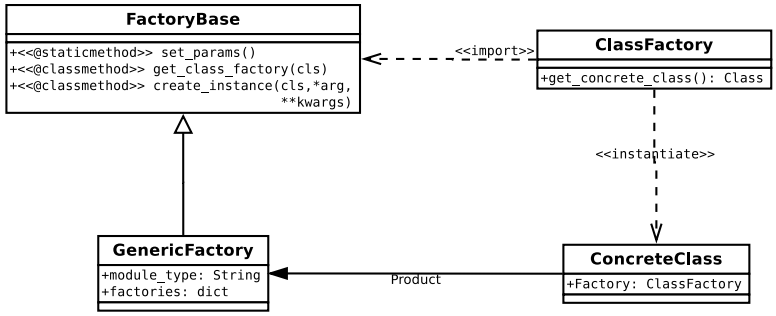


Figure 2: Atlas factory pattern class diagram.

3 Software Design

3.1 Configuration System

The configuration system, AtlasConfig, is built around the ConfigObj Python library. Employing a singleton pattern, AtlasConfig stores the requested processing blocks and parameters that can be accessed from within the namespace. The configuration files take the general form of Listing 1. These are dependent on the experiment and the particular processing elements, but as these are developed they become a significant part of the experiment documentation.

The configuration begins with a list of requested modules. These are core elements specific to Atlas that delineate a functional separation or data encapsulation along the processing pipeline. Not only can this list be used by the validation system for ensuring the config meets the formatting specification, but it also provides a mechanism for breaking from the traditional flow of Figure 1 without modifying the core code. The list corresponds to module definitions, further down in the configuration file, that specify the requested implementation and any subsections or variables required, as illustrated in Listing 1. These are fixed and have corresponding factories associated with them, but the specific implementation can be from outside of the core codebase – detailed further below.

3.2 Atlas BaseFactory

The factory method [5], is a creational pattern for abstracting away the instantiation of a concrete class to another class or subclass. The pattern defers the decision of which implementation to invoke to the factory at run-time. Unique to this work, is the utilization of the Python import framework to completely remove any prior awareness of the instantiable classes. Using this with the configuration system, data scientists can replace specific blocks of the experiment pipeline with their own implementations, without ever changing the Atlas core. This provides flexibility and encourages exploration, without risking untested or immature code diluting the shared codebase. As components mature and evolve, they can be incorporated into the common framework through the traditional development/test lifecycle. In Atlas, the factory class diagrams take the form of Figure 2. The `BaseFactory` class is inherited by all `ModuleFactories`. This contains the specific import mechanisms unique to Atlas.

An example of how this pattern is implemented is presented in Listing 2. This function takes advantage of the `@classmethod` decorator provided by Python and allows it to modify the instantiating class definition, rather than an instance. When a specific module instance is requested, this method will search for the corresponding implementation factory. If a file path

Listing 2: Example of a dynamic import function in Python.

```
@classmethod
def get_class_factory(cls):
    # Get classname and path from params (AtlasConfig Object).
    classname, module_path = FactoryBase.__params.get_module_type(cls.module_type)
    # If the class has already been registered.
    if classname in cls.factories:
        return cls.factories[classname]
    # Was a path to the implementation given?
    if module_path is None:
        if not cls.initialized:
            # Search the sub-directories of the ModuleFactory for implementations.
            cls._get_module_paths()
            cls.initialized = True
        modules = cls.modulePaths
    else:
        # Split path on extension.
        filename, _ = os.path.splitext(module_path)
        module_name = os.path.basename(filename)
        modules = [(module_path, module_name)]
    # Search through the list of possible modules.
    for import_path, module_name in modules:
        try:
            # Try to import the module.
            mod = imp.load_source(module_name, import_path)
            if hasattr(mod, classname):
                # Get the implementation factory.
                module_class = getattr(mod, classname)
                module_class_factory = module_class.Factory()
                # Register the factory so future searches are not required.
                cls.factories[classname] = module_class_factory
                # Return the factory.
                return module_class_factory
        except ImportError:
            print("Error importing {} from {}".format(module_name, import_path))
    # All of modules were searched and none returned the requested class.
    raise Exception("Module:{}, class:{}".format(module_path, classname))
```

is included in the configuration file, then that module will be imported and the corresponding implementation will be returned. If a path is not specified, then the function will search the associated directory structure of the `ModuleFactory` object. This pattern is utilized throughout the Atlas pipeline.

3.3 Atlas Exam Decorators

The motivation for employing a decorator pattern in an already complex framework was driven by the `Exam` classes. The different clinical studies result in a surprising number of analysis configurations. This is best exemplified in the different beat processing methods used in the creation of feature vectors. Physiologically, TCD beats correspond to the pulsatile blood flow measured in the vasculature. In Atlas feature generation, this includes everything from raw beats truncated to a common size, normalized beats, or beats averaged over each exam segment. The decorator pattern provides a mechanism for adding components and functionality at runtime – meaning that different beat processing mechanisms can be swapped into an exam by changing the configuration.

The implementation in Atlas utilizes the built-in Python `Metaclass` abstraction. Additionally, the `six` package is incorporated to allow cross-functionality between Python 2 and 3. The use of the somewhat controversial – at least in the Python community – `Metaclass` as

Listing 3: Metaclass used for decorating inheriting classes with defined functionalities.

```
from atlas.config import AtlasConfig

class MetaDecorator(type):
    def __new__(mcs, name, bases, dct):
        params = AtlasConfig()

        functionalities = params.config["exam"]["MetaExam"].sections

        for mod_name in functionalities:
            mod_path = params.config["exam"]["MetaExam"][mod_name].get(
                "module_path", None)
            module = mcs.import_module(mod_name, mod_path)
            explicit_decorations = getattr(module, "DECORATIONS", None)

            if explicit_decorations is not None:
                for func_name in explicit_decorations:
                    func = getattr(module, func_name)
                    dct[func.__name__] = func
            else:
                func = getattr(module, "get_{}".format(mod_name))
                dct[func.__name__] = func

            init_method = getattr(module, "init_{}".format(mod_name), None)

            if init_method is not None:
                dct["__"+init_method.__name__] = init_method
                dct["registered_inits"].append("__"+init_method.__name__)
            else:
                print("init_method {} is None:".format(mod_name))

        return super(MetaDecorator, mcs).__new__(mcs, name, bases, dct)

    @staticmethod
    def import_module(mod_name, mod_path):

        if mod_path is None:
            curr_dir = dirname(abspath(__file__))
            mod_path = join(curr_dir, "functionalities", "{}.py"
                            .format(mod_name))

        module = imp.load_source(mod_name, mod_path)
        return module
```

Listing 4: Example implementation of a dynamic functionality.

```
from atlas.experiment_framework.preprocessing.beats import BeatsFactory

def init_beats(exam, dec_params):
    beats = {}
    segments = ["baseline"]
    # loop through all of the segments and create the beats.
    # Normally this would include a larger list of exam segments.
    for segment in segments:
        beats[segment] = BeatsFactory.create_instance(dec_params, segment)
    # add the beats as an exam attribute
    exam.beats = beats

def get_beats(exam, segment):
    return exam.beats.get(segment)
```

Listing 5: Example exam base class implementing the Metaclass.

```
import six
from atlas.config import AtlasConfig

class MetaExam(six.with_metaclass(MetaDecorator, object)):
    """ This will be populated by the MetaDecorator. """
    registered_inits = []

    class Factory(object):
        """Used by the ExamFactory to generate MetaExam instances."""
        def create(self, *args, **kwargs):
            """Return an instance of MetaExam"""
            return MetaExam(*args, **kwargs)

    def __init__(self, params, subid, examid_string, sensor, is_kit):
        """Initialize the object since this does not happen at instantiation."""

        """
                [Raw data and other basic processing may occur here.]
        """

        # Run initialization for decorated functionalities
        for init_method in MetaExam.registered_inits:
            dec_params = self.params.config["exam"]["MetaExam"].get(
                init_method.split('_')[-1], {})

            try:
                dec_params.pop("module_path")
            except KeyError:
                pass

            getattr(self, init_method)(dec_params)
```

opposed to the built-in `@decorator` keyword, came from the need for adding functionalities dynamically at run-time. Because of this, similar to the factory methods, the use of the decorator pattern in Atlas relies on the `AtlasConfig` class to define which functionalities will be attached.

Listing 3 presents the super class implementation for decorating `exam` classes with functionalities – comparable to the classic `Component` object. The requested functionalities are first pulled from the `AtlasConfig` object and imported with the static method `import_module`. The functionality definitions are added to the concrete exam class definition and the init functions are added to the `registered_inits` list. None of the imported functions are called at this point, only the class definition is modified. An example functionality is included in Listing 4. Only two functions are implemented here, `init_beats`, which is added to the registered initializers, and `get_beats`, which is automatically added by the `MetaDecorator` class. All of this is then employed in the inheriting class illustrated in Listing 5 – this is comparable to the classical `ConcreteComponent`. In addition to loading the basic raw data, the `MetaExam` class loops through all of the registered functionalities and initializes them at runtime. This pattern allows for a component or abstract-oriented approach to the data containers. It has encouraged reuse of the core functionalities and added the necessary flexibility to the machine learning experiments.

3.4 Motivating Example

Running an Atlas experiment generally involves 2 steps. The first is constructing the configuration file – as illustrated in Listing 6. The second step is the basic Python code to instantiate

Listing 6: Motivating Example.

```
module_list =
    experiment, exam_list,
    beats, exam,
    ml_framework, data_set_gen,
    subspace_decomp, classifier
[general]
search_path = /data/
output_path = ~/test
extract_peaks = True
[experiment]
    [[SimpleExperiment]]
[exam_list]
    [[ExamList]]
    module_path = ~/example_examlist.py
[exam]
    [[MetaExam]]
        extract_peaks=True
        [[[beats]]]
        [[[events]]]
[beats]
    [[RawBeats]]
[ml_framework]
    [[LeaveOneOutml]]
[data_set_gen]
    [[AverageDataSetGen]]
    train_segments = baseline,
    test_segments = baseline,
[subspace_decomp]
    [[SciKitPCADecomp]]
    subspace_n_components = 5
    random_subspace = False
    random_subspace_samples = 10
[classifier]
    [[SciKitSVMClassifier]]
    theta0 = 1e-2
    svm_C = 100
```

Listing 7: Running the Experiment.

```
params = AtlasConfig("./example.cfg")
FactoryBase.set_params(params)
exp = ExperimentFactory.create_instance(
    params)
exp.initialize(params)
exp.run_experiment()
```

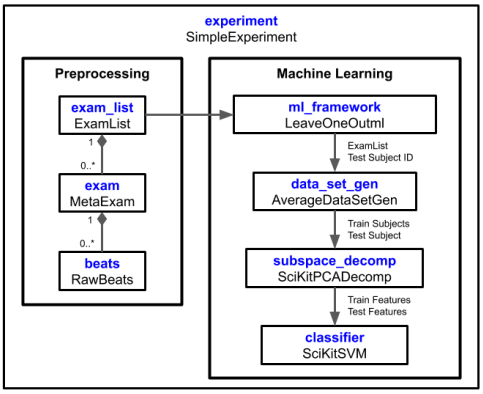


Figure 3: Motivating Example.

the `Experiment` object and run the ML experiment – given in Listing 7.

After the `[general]` section in the configuration file, the requested modules are defined. Figure 3 illustrates the relationships between these modules. In the Preprocessing blocks the instantiated modules have a compositional relationship, with `exam` containing multiple `beats` and the `exam_list` organizing all of the `exam` objects. In the Machine Learning block however, the diagram illustrates a behavioral or data dependency interaction – with the output from a module feeding the subsequent module. These are contained within a single `experiment` object that controls the instantiation and execution of the experiment.

In this example a `SimpleExperiment` is requested with a `LeaveOneOutml` cross validation. The experiment first collects the requested exams – Preprocessing block. A single subject is then removed from `exam_list` – using `AverageDataSetGen`. A subspace decomposition is then constructed using the training data with a PCA from SciKit-learn [11] – `SciKitPCADecomp`. A classifier is then constructed using an SVM model, also from SciKitLearn. The classifier is then used on the subject that was removed for testing. The experiment continues until the cross-validation has completed. The ability to wrap other packages – SciKitLearn in this example – is another benefit of the creational pattern. This provides an important mechanism for encouraging the inclusion of external libraries in the pipeline.

4 Discussion

The ultimate question of any generic framework is in how useful it actually is. Atlas has supported a number of different studies, ranging from mild traumatic brain injuries [15, 14, 16], to stroke [18, 13, 17]. However, what's most interesting about those publications, is that most are based more traditional analysis using pooled statistics or statistical models, rather than machine learning. The functional decoupling designed into Atlas has supported use cases that fall outside of the initial software requirements. This is probably the best illustration of the true extensibility that the architecture provides.

The decision to incorporate the design patterns presented here was not made without hesitation. As discussed in the introduction, including this level of extensibility and flexibility comes at a cost of reduced code readability and a level of architectural complexity that can be unnecessary at times. This framework has been through several design iterations before arriving at its current state and it could be argued that there is a reduction in the understanding of the codebase. This is particularly true for the creational aspects of Atlas. The use of decorator pattern has not generated as much confusion – this has been the case even when the developer had no prior exposure to patterns. This observation is consistent with more rigorous studies of design patterns – where the decorator in particular has also been shown to have generally positive effects on extensibility and developer comprehension [19]. There is a learning curve to fully utilizing Atlas for novice developers. However, this only seems to hold true when trying to extend the framework, rather than during its general use. Regardless, the benefits it offers more than outweigh the developer start-up costs.

The aspect-oriented approach to exam composition has allowed for efficient changes in feature vector extraction and the module design has supported use outside of its original design. However, one important concept not discussed by these choices is performance. These design considerations can in many cases result in a significant performance drop. In this instance, the patterns are applied in places where either performance is not a concern, or other processing dominates the total computational costs – for example, the actual machine learning training and testing. If performance becomes a concern in the future, the modularized design of Atlas will allow for precise profiling and identification of bottlenecks. Furthermore, these modules can be more readily replaced with optimized implementations.

This work describes the use of creational and structural patterns in the Atlas framework as a way to mitigate the conflicting requirements of machine learning in the medical space. Outside the scope of this work are the model exploration and production systems that employ the resulting models from the experimental framework. In addition, the report generation and resulting experiment database deserve mention as well. All of these have resulted in a stable and usable system for machine learning model development.

Funding: This work was supported by the National Institutes of Health award number 2R44NS092209-02. The content is solely the responsibility of the author and does not necessarily represent the official views of the National Institutes of Health.

References

[1] Andrew Beers, James Brown, Ken Chang, Katharina Hoebel, Elizabeth Gerstner, Bruce Rosen, and Jayashree Kalpathy-Cramer. Deepneuro: an open-source deep learning toolbox for neuroimaging. *arXiv preprint arXiv:1808.04589*, 2018.

[2] Marc Boyer and Vojislav B Mišić. Generic patterns: Bridging the contextual divide. *GRACE TECHNICAL REPORTS*, page 20, 2009.

[3] Alexander Chatzigeorgiou, Nikolaos Tsantalis, and Ignatios Deligiannis. An empirical study on students' ability to comprehend design patterns. *Computers & Education*, 51(3):1007–1016, 2008.

[4] Matthew J. Clarkson, Gergely Zombori, Steve Thompson, Johannes Totz, Yi Song, Miklos Espak, Stian Johnsen, David Hawkes, and Sébastien Ourselin. The niftk software platform for image-guided interventions: platform overview and niftylink messaging. *International Journal of Computer Assisted Radiology and Surgery*, 10(3):301–316, Mar 2015.

[5] Erich Gamma. *Design patterns: elements of reusable object-oriented software*. Pearson Education India, 1995.

[6] Eli Gibson, Wenqi Li, Carole Sudre, Lucas Fidon, Dzhoshkun I Shakir, Guotai Wang, Zach Eaton-Rosen, Robert Gray, Tom Doel, Yipeng Hu, et al. Niftynet: a deep-learning platform for medical imaging. *Computer methods and programs in biomedicine*, 158:113–122, 2018.

[7] Günter Kniesel, Tobias Rho, and Stefan Hanenberg. Evolvable pattern implementations need generic aspects. *RAM-SE*, 4:111–126, 2004.

[8] Christian Kohls and Katharina Scheiter. The relation between design patterns and schema theory. In *Proceedings of the 15th Conference on Pattern Languages of Programs*, page 15. ACM, 2008.

[9] Stefan Maetschke, R Tennakoon, Christian Vecchiola, and Rahil Garnavi. nuts-flow/ml: data pre-processing for deep learning. *arXiv preprint arXiv:1708.06046*, 2017.

[10] Nick Pawlowski, S. Ira Ktena, Matthew C.H. Lee, Bernhard Kainz, Daniel Rueckert, Ben Glocker, and Martin Rajchl. Dltk: State of the art reference implementations for deep learning on medical images. *arXiv preprint arXiv:1711.06853*, 2017.

[11] Fabian Pedregosa, Gaël Varoquaux, Alexandre Gramfort, Vincent Michel, Bertrand Thirion, Olivier Grisel, Mathieu Blondel, Peter Prettenhofer, Ron Weiss, Vincent Dubourg, et al. Scikit-learn: Machine learning in python. *Journal of machine learning research*, 12(Oct):2825–2830, 2011.

[12] Guido Rossum. Python reference manual. 1995.

[13] C Thibeault, S Thorpe, S Wilk, T Devlin, and R Hamilton. E-071 using transcranial doppler ultrasound for the objective evaluation and prediction of endovascular treatment outcomes, 2018.

[14] Corey M Thibeault, Samuel Thorpe, Nicolas Canac, Michael J O'Brien, Mina Ranjbaran, Seth J Wilk, and Robert B Hamilton. A model of longitudinal hemodynamic alterations after mild traumatic brain injury in adolescents. *Journal of Concussion*, 3:2059700219838654, 2019.

[15] Corey M Thibeault, Samuel Thorpe, Michael J O'Brien, Nicolas Canac, Mina Ranjbaran, Ilyas Patanam, Artin Sarraf, James LeVangie, Fabien Scalzo, Seth J Wilk, et al. A cross-sectional study on cerebral hemodynamics after mild traumatic brain injury in a pediatric population. *Frontiers in neurology*, 9:200, 2018.

[16] Corey Michael Thibeault, Samuel Garrett Thorpe, Nicolas Canac, Seth J Wilk, and Robert Benjamin Hamilton. Sex-based differences in transcranial doppler ultrasound and self-reported symptoms after mild traumatic brain injury. *Frontiers in Neurology*, 10:590, 2019.

[17] Samuel G Thorpe, Corey M Thibeault, Nicolas Canac, Seth J Wilk, Thomas Devlin, and Robert B Hamilton. Decision criteria for large vessel occlusion using transcranial doppler waveform morphology. *Frontiers in neurology*, 9:847, 2018.

[18] Samuel G Thorpe, Corey M Thibeault, Seth J Wilk, Michael O'Brien, Nicolas Canac, Mina Ranjbaran, Christian Devlin, Thomas Devlin, and Robert B Hamilton. Velocity curvature index: a novel diagnostic biomarker for large vessel occlusion. *Translational stroke research*, pages 1–10, 2018.

[19] Marek Vokáč, Walter Tichy, Dag IK Sjøberg, Erik Arisholm, and Magne Aldrin. A controlled experiment comparing the maintainability of programs designed with and without design patterns—a replication in a real programming environment. *Empirical Software Engineering*, 9(3):149–195, 2004.

[20] Ian H Witten, Eibe Frank, Mark A Hall, and Christopher J Pal. *Data Mining: Practical machine learning tools and techniques*. Morgan Kaufmann, 2016.