Enabling MLOps in the SPIRA Training Pipeline

Daniel Angelo Esteves Lawand

Capstone Project
presented to the discipline
MAC0499

Supervisors:
Prof. Dr. Alfredo Goldman vel Lejbman
MSc. Renato Cordeiro Ferreira

São Paulo, December of 2023
[1]

Abstract

LAWAND, D. A. E. **Enabling MLOps in the SPIRA Training Pipeline**. 2023. 33 p. Thesis (BSc.) - Instituto de Matemática e Estatística, Universidade de São Paulo, São Paulo, 2023.

SPIRA is a ML-enabled system to pre-diagnosis insufficiency respiratory based on speech analysis. The training pipeline created during its initial modelling led to an accidental architecture, which lacked good software quality attributes. The goal of this research is to design a continuous training pipeline to enable MLOps for SPIRA. Based on the issues found in the experimental implementation, this research proposes a new architecture for the system. It combines design principles and design patterns with the hexagonal architecture pattern, supporting several quality attributes: maintainability, extensibility, reusability, robustness, efficiency, understandability, and readability. With them, the training pipeline can be easily modified and evolved alongside its three axes of change: code, model, and data. Furthermore, this research also improves the infrastructure of the system, to ensure it is easy to run. This is essential to continuously deliver new versions of the SPIRA model. Together, these changes enable MLOps practices in the project.

Keywords: SPIRA, MLOps, Machine Learning Engineering, Intelligent Systems, Design Patterns, Hexagonal Architecture.

Resumo

LAWAND, D. A. E. **Habilitando MLOps na Pipeline de Treinamento do SPIRA**. 2023. 33 f. Tese (Bacharelado) - Instituto de Matemática e Estatística, Universidade de São Paulo, São Paulo, 2023.

O SPIRA é um sistema inteligente para pré-diagnóstico de insuficiência respiratória com base na análise da fala. A pipeline de treinamento criada durante a fase de modelagem levou a uma arquitetura acidental que carecia de bons atributos de qualidade de software. O objetivo desta pesquisa é projetar uma pipeline de treinamento contínuo para habilitar MLOps no SPIRA. Com base nos problemas encontrados na implementação experimental, esta pesquisa propõe uma nova arquitetura para o sistema que combina princípios e padrões de design com o padrão de arquitetura hexagonal, apoiando vários atributos de qualidade: manutenabilidade, extensibilidade, reutilização, robustez, eficiência, compreensibilidade e legibilidade. Baseado neles, a pipeline de treinamento pode ser facilmente modificada e evoluída ao longo dos seus três eixos de mudança: código, modelo, e dados. Além disso, esta pesquisa também aprimora a infraestrutura do sistema para que seja fácil executá-lo. Isso é essencial para entregar continuamente novas versões do modelo SPIRA. Em conjunto, essas mudanças habilitam práticas de MLOps no projeto.

Palavras-chave: SPIRA, MLOps, Engenharia de Machine Learning, Sistemas Inteligentes, Padrões de Design, Arquitetura Hexagonal.

Contents

Chapter 1

Introduction

SPIRA is a multidisciplinary project created during the COVID-19 pandemic for pre-diagnosis of insufficiency respiratory based on speech analysis. As this problem is inherently difficult, a solution based on Convolutional Neural Networks (CNN) was proposed (Casanova et al., 2021). The code produced during this endeavour was experimental in nature. This gave flexibility for the data scientists to do modeling, but hindered the long-term maintainability of the code.

The behavior of a ML-enabled system can change in three different axes: code, model, and data (Sato et al., 2019). In 2024, the SPIRA project will start a new phase of data collection. As a result, it is expected that the SPIRA model will evolve in these three axes. As soon as new data gets acquired, it may be incorporated into the training dataset. The model then can be retrained to improve its performance. This generates insights that may change the modeling. These changes have to be supported in the training pipeline. Notwithstanding, given the codebase is hard to evolve and the execution of the system is manual, producing new models is a slow and error-prone process.

The goal of this research is to design a continuous training pipeline to enable MLOps for SPIRA. Enabling MLOps, or Machine Learning Operations, means enabling the continuous delivery of Machine Learning models. This research aims to reimplement the existing training pipeline with focus on improving different quality attributes: maintainability, efficiency, extensibility, understandability, reusability, robustness, and readability. Nonetheless, to better support this goal, this redesign will start by analyzing the code quality issues present in the experimental architecture.

This monograph is structured as follows. Chapter 2 gives an overview of the different researches performed in the SPIRA system. Chapter 3 lists the quality attributes and the consequence of the lack of them. Chapter 4 explores the lack of quality attributes in the experimental training architecture. Chapter 5 details the new architecture focusing on the practices applied to ensure good code quality. Chapter 6 compares the changes in the infrastructure between the two architectures. Chapter 7 analyzes how the new architecture achieved the desired quality attributes. Chapter 8 summarizes the main results and lists future work. Lastly, Appendix A introduces some fundamentals about convolutional neural networks, which are the basis for the SPIRA model.

Chapter 2

SPIRA Overview

This chapter gives an overview of the technical side of the SPIRA project and explains where this research fits into the SPIRA ecosystem. Figure 2.1 gives the seven researches made in the technical side of SPIRA: System Architecture, CNN Model, Revision of SPIRA's *v1* model, Training Data Collection App, Model Server and Inference System, Highly Availability with Kubernetes, and this research.

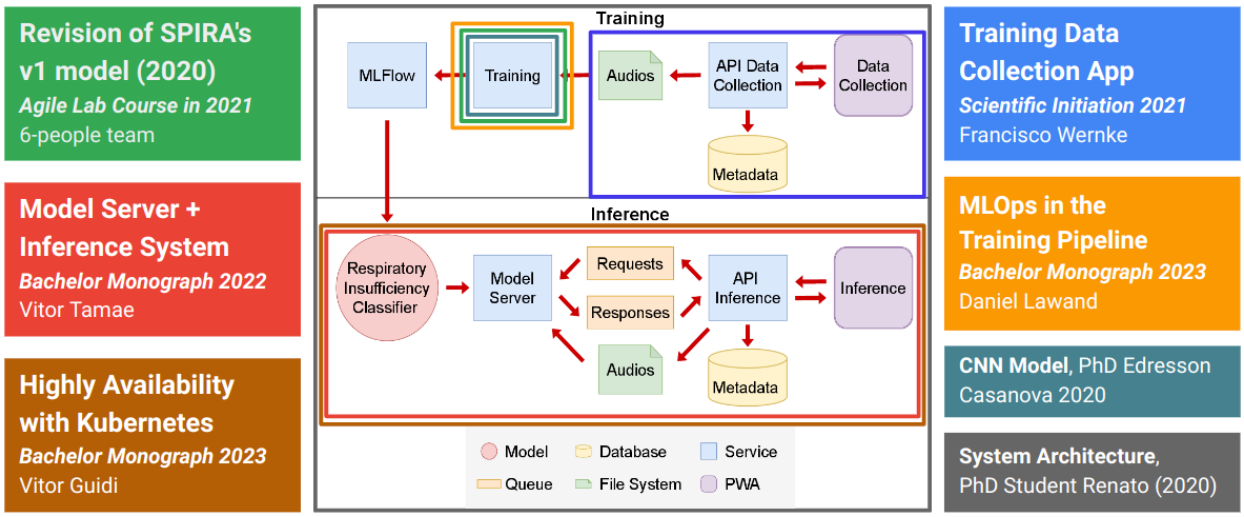

Figure 2.1: ***SPIRA Overview.***

- **System Architecture:** That research designed the SPIRA architecture to follow the architectural style of microservices (Ferreira et al., 2022).
- **CNN Model:** That research investigated whether it is possible to detect respiratory insufficiency by analyzing COVID-19 patient's speech samples (Casanova et al., 2021).
- **Revision of SPIRA's *v1* model:** That research goal was to "produce an improved model: trained with data collected from hospital patients, and integrated into an app for pre-diagnosis of respiratory insufficiency" (Ferreira et al., 2022).
- **Training Data Collection App:** That research developed a Progressive Web App that collects audios from patients and sends them to a Back-end; a Back-End in Java Quarkus that implements a Restful API and stores the data sent in a MongoDB; and a website that allows the visualization of the data collected (Ferreira et al., 2022).

- **Model Server and Inference System:** The goal of that research was to implement and deploy an "intelligent distributed inference system that allows medical personnel to perform a respiratory insufficiency pre-diagnosis using the models created by SPIRA" (Tamae, 2022).
- **Highly Availability with Kubernetes:** The goal of that research was to "enable high-availability for SPIRA by migrating all of its business and infrastructure microservices to Kubernetes" (Guidi, 2023).
- **MLOps in the Training Pipeline:** It refers to this research, the goal of which is to design a continuous training pipeline to enable MLOps for SPIRA. This research proposes a new architecture for the system combining several quality attributes. With them, the training pipeline can be easily modified and evolved alongside its three axes of change: code, model, and data. Furthermore, this research also improves the infrastructure of the system, to ensure it is easy to run. This is essential to continuously deliver new versions of the SPIRA model. Together, these changes enable MLOps practices in the project.

Chapter 3

Code Quality

This chapter lists attributes that characterize good quality code. These attributes are called **Quality Attributes** (Suryanarayana et al., 2014). Also, this chapter lists the consequences of the lack of these attributes.

3.1 Quality Attributes

- **Readability:** refers to how easily the code can be read and understood by a human without necessarily understanding the entire functionality of the code. It primarily concerns the visual presentation of the code, emphasizing aspects such as formatting, indentation, and the use of meaningful names. Readable code facilitates quick comprehension during casual inspection, making it easier for developers to navigate, troubleshoot and debug the code (Martin, 2009).

- **Understandability:** refers to the ease with which a design fragment can be understood by stakeholders (Suryanarayana et al., 2014). Understandability enables a deeper understanding of the system's logic, behavior, and structure. It aims to establish clear interfaces between modules, allowing for a more straightforward understanding of the system's design. A system that is easily understandable is more likely to be maintainable and adaptable over its lifecycle.

- **Efficiency:** efficiency in algorithms directly impacts system performance. Optimized algorithms contribute to faster execution times, reduced resource consumption, and an overall more responsive system. Improving efficiency is essential for meeting performance requirements and enhancing user experience.

- **Robustness:** code should be robust and resilient to errors. Designing algorithms with error-resistant features ensures that the system can gracefully handle unexpected situations, preventing potential disruptions and improving overall reliability.

- **Extensibility:** is the ability of a design fragment to accommodate new features, functionalities, or changes with minimal effort and impact on the existing code (Suryanarayana et al., 2014). A system is considered extensible when it can be easily extended or modified without causing unintended side effects or breaking existing functionality. Extensibility is an important quality attribute that contributes to the maintainability and long-term viability of a software project.

- **Reusability:** refers to "the ease with which a design fragment can be used in a problem context other than the one for which the design fragment was originally developed" (Suryanarayana et al., 2014). A reusable fragment is designed in a way that promotes flexibility and adaptability, allowing it to be applied in various contexts without significant modification. Reusability is a crucial aspect of software design, contributing to ease of development, reducing redundancy, and enhancing maintainability.

- **Maintainability:** refers to the ease with which a software system can be maintained, modified and updated during its life cycle. It incorporates principles of modularity, encapsulation, and abstraction to facilitate updates, bug fixes, enhancements without causing undue disruptions, and adaptations to changing requirements The goal of designing for maintainability is to minimize the effort, time, and cost required to keep the software in a good and reliable state.

3.2 Lack of Quality

3.2.1 Cyclomatic Complexity

One way to measure these quality attributes is the **cyclomatic complexity** (Mccabe, 1976): a software metric that counts the number of independent paths through a program's source code. High cyclomatic complexity indicates intricate and convoluted control flow structures. This can lead to difficulties in understanding, testing, and maintaining the code. This metric is often used as an indicator of potential software defects, and it can be helpful in estimating the testing effort required for a program.

Cyclomatic complexity identifies the number of **decision points** within a program and calculates a score based on those elements. A decision point refers to a location in the source code where a decision is made, based on a condition, that creates different paths through the algorithm.

These decision points are typically associated with control flow structures that alter the flow of program execution (such as if statements, loops, switch statements and exception handling). As the number of decision points increases, the number of independent paths increases, and cyclomatic complexity goes up.

Reducing cyclomatic complexity is desirable as it can lead to more maintainable, efficient, and readable code. To achieve this goal, this research apply techniques such as **design patterns** (Gamma et al., 1996).

3.2.2 Bad Smells

In the early 1990s, Kent Beck and Martin Fowler (Beck and Fowler, 1999) introduced the term code smell in the context of refactoring. The terms bad smells and code smells describe certain characteristics in the source code that may indicate potential issues or areas for improvement. The idea is that just as a bad smell in the physical world can indicate a potential problem, a code smell in software can suggest areas of the code that might need attention or improvement. The bad smells affect the readability by making it harder to comprehend and maintain the code. Improving modularization involves addressing these bad smells to enhance readability, maintainability and extensibility.

Let's explore the consequences of some specific code smells:

- **Functions Descend More than One Level of Abstraction:**
  - **Decreased readability:** Code becomes harder to understand as it mixes different levels of abstraction.
  - **Maintenance challenges:** Changes in one part of the function might inadvertently affect unrelated parts, leading to errors.
- **Misplaced Responsibility:**
  - **Lack of cohesion:** Code becomes less modular, making it difficult to understand and maintain.
  - **Increased coupling:** Classes or modules become tightly coupled, making it challenging to change one without affecting the other.
- **Defensive Programming:**
  - **Decreased readability:** Excessive defensive code can clutter the main logic, making it harder to follow.
  - **Redundancy:** Unnecessary checks may lead to duplicated code and increased maintenance efforts.
  - **Processing Cost:** Later checks can invalidate an invariant that could be checked in the beginning of the code avoiding processing cost.
- **Functions with too many arguments:**
  - **Reduced readability:** It becomes challenging to understand the purpose of each argument and their order.
  - **Maintenance difficulties:** Changes to the function signature may require updates in multiple places.
- **Commented Code:**
  - **Outdated comments:** Comments can become obsolete and misleading if the code is changed but the comments are not updated.
  - **Decreased maintainability:** Commented-out code can confuse developers and clutter the code.
- **Nested Statements:**
  - **Reduced readability:** Deeply nested `if` or `for` statements can be hard to follow and understand.
  - **Increased complexity:** Nested conditions make it harder to reason about the code and may introduce logical errors.

Identifying and addressing code smells is essential for make the code readable, understandable, and maintainable. Regular code reviews, refactoring and applying **Design Patterns** can help mitigate the consequences of these code smells.

3.2.3 Big Ball of Mud

The term BIG BALL OF MUD (Yorder and Foote, 1999) refers to an anti-pattern where the architectural design lacks clear structure, modularity, and organization. This anti-pattern is characterized by a system that has evolved over time without a cohesive architectural plan, resulting in a tangled and monolithic code.

A BIG BALL OF MUD can include the following characteristics:

- **Lack of Modularity:** The system lacks well-defined modules or components. The code responsible for different functionalities is often intertwined, making it difficult to isolate and understand specific parts of the system.
- **Tight Coupling:** The components of the system are tightly coupled, which means they are highly interdependent. Changes in one part of the system can have unintended consequences in other areas, leading to a lack of flexibility and difficulty in maintenance.
- **No Clear Separation of Concerns:** The separation of concerns (Dijkstra, 1982) is often ignored. Business logic, user interface code and data access can be mixed up, making it difficult to discern the responsibilities of the different parts of the system.
- **Uncontrolled Growth:** The system tends to grow organically without a well-defined plan. New features or modifications are added without taking architectural principles into account, contributing to increased complexity.
- **Difficulty in Understanding:** Due to the lack of structure and organization, the overall system becomes difficult to understand. Developers can find it difficult to navigate the code base and understand the interactions between the different components.

To address the consequences of the BIG BALL OF MUD, architectural refactoring and the adoption of modular design principles are essential to bring understandability, and maintainability to the system.

Chapter 4

Experimental Training Architecture

To ease the communication this chapter will refer to the SPIRA architecture made by Edresson Casanova (Casanova et al., 2021) as SPIRA *v1*, whereas the architecture proposed by this research will be called SPIRA *v2*. This chapter presents the SPIRA *v1* architecture and analyzes its issues related to code quality using concepts from Chapter 3.

In order to visualize the SPIRA *v1* system, the first idea was to create a diagram of the architecture. However, the system lacks structure, preventing a useful representation. In fact, the experimental modeling process of SPIRA *v1* accidentally created a system that follow the architectural anti-pattern Big Ball of Mud, previously described in Chapter 3.

The lack of quality attributes is manifest throughout the SPIRA *v1* system starting with the organization of the file system. Figure 4.1 highlights how there is no clear structure between the directories. The `scripts` directory does not contain all the `scripts` nor the `train` folder contain the `train script`. This makes it very difficult to understand the responsibilities of each directory.

The following sections show the lack of quality attributes and the presence of bad smells:

4.1 Functions descend more than one level of abstraction

The `Dataset` class should work with the abstraction of the data, which is `audio`. The `audio` is an abstraction of WAV (Waveform Audio File) format, while the `wav` is an abstraction of `tensor`, a multi-dimensional array of data implemented by PyTorch. Code 4.1 illustrates this smell.

```
elif self.c.dataset["temporal_control"] == "one_window":
    # print( "one_window")
    # choise a random part of audio
    step = self.ap.sample_rate * self.c.dataset["window_len"]
    # print(wav.shape, step)
    idx = random.randint(0, wav.size(1) - (step + 1))
    feature = self.ap.get_feature_from_audio(
        wav[:, idx : idx + step]
    ).transpose(1, 2)
    target = torch.FloatTensor([class_name])
```

Code 4.1: ***Functions descend more than one level of abstraction:*** *This code snippet shows part of the method `__getitem__` in class `Dataset`. It descend four levels of abstractions: dataset, audio, wav and tensor.*

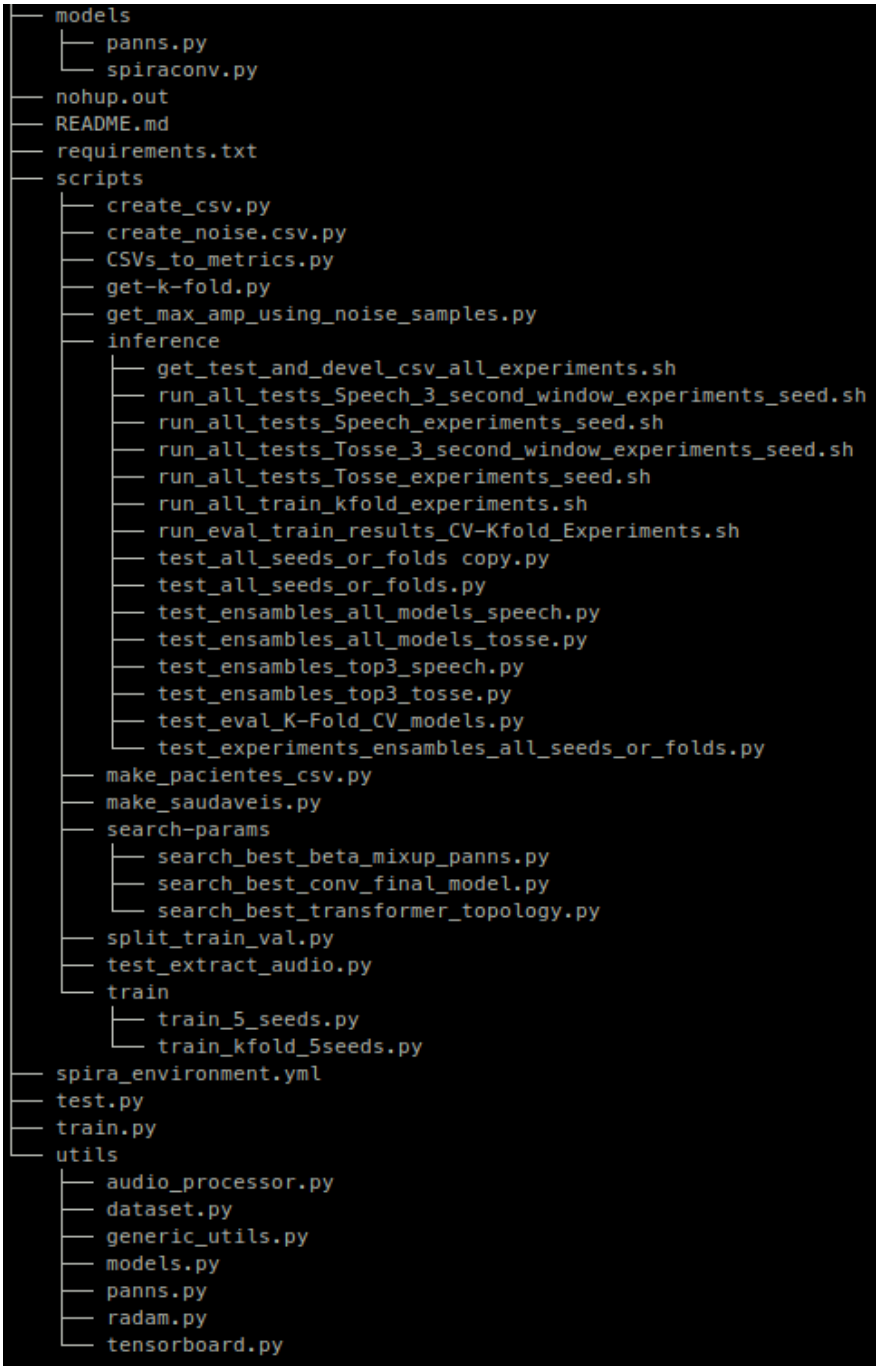

Figure 4.1: ***File system tree:*** *Shows the unclear structure between the directories and files.*

4.2 Misplaced Responsibility

This bad smell is present in multiple places. Code 4.2 shows random number generation, data loading, and assertion in a single method.

4.3 Defensive Programming

Exceptions are raised arbitrarily in the middle of the code, even when they are related to programming errors that can be checked earlier. Code 4.3 shows the occurrence of this smell.

```
# set random seed
random.seed(c.train_config["seed"])
torch.manual_seed(c.train_config["seed"])
torch.cuda.manual_seed(c.train_config["seed"])
np.random.seed(c.train_config["seed"])
torch.backends.cudnn.deterministic = True
torch.backends.cudnn.benchmark = False
self.c = c
self.ap = ap
self.train = train
self.test = test
self.test_insert_noise = test_insert_noise
self.num_test_additive_noise = num_test_additive_noise
self.num_test_specaug = num_test_specaug

self.dataset_csv = c.dataset["train_csv"] if train else c.dataset["eval_csv"]

assert os.path.isfile(
    self.dataset_csv
), "Test or Train CSV file don't exists! Fix it in config.json"

accepted_temporal_control = ['overlapping', 'padding', 'avgpool',
                                    'speech_t', 'one_window']
assert (self.c.dataset['temporal_control'] in accepted_temporal_control),"You cannot \
                    use the padding_with_max_length option in conjunction with the \
                    split_wav_using_overlapping option, disable one of them !!"

self.control_class = c.dataset['control_class']
self.patient_class = c.dataset['patient_class']

# read csvs
self.dataset_list = pd.read_csv(self.dataset_csv, sep=',').replace(
                    {'?': -1}).replace(
                    {'negative': self.control_class}, regex=True).replace(
                    {'positive': self.patient_class}, regex=True).values
```

Code 4.2: ***Misplaced responsibility:*** *This code snippet shows part of the constructor of the class* `Dataset`*. Lines two to seven handle random number generation, lines eight to sixteen assigning values to attributes, lines eighteen to twenty and twenty-four to twenty-seven handle assertions, and line thirty-two handles data loading from a* `.csv`*.*

```
if not self.num_test_additive_noise and not self.num_test_specaug:
    raise RuntimeError(
        "ERROR: when  test_insert_noise is True, num_test_additive_noise\
         or num_test_specaug need to be > 0"
    )
if (
    self.c.dataset["temporal_control"] == "overlapping"
    or self.c.dataset["temporal_control"] == "speech_t"
):
    raise RuntimeError(
        "ERROR: Noise insertion in  'temporal_control' overlapping and\
          speech_t is not supported !! You need implement it !!"
    )
```

Code 4.3: ***Defensive programming:*** *This code snippet shows part of the constructor of the class* `Dataset`*. Errors are raised in case the data augmentation (noise insertion) configuration has an issue.*

4.4 Too many arguments

Code 4.4 shows the occurrence of this smell. A function with too many arguments leverages questions of why it needs such amount of arguments and what the importance of each is.

```
def __init__(
    self,
    c,
    ap,
    train=True,
    max_seq_len=None,
    test=False,
    test_insert_noise=False,
    num_test_additive_noise=0,
    num_test_specaug=0,
):
```

Code 4.4: ***Functions with too many arguments:*** *This code snippet shows part of the constructor of the class* `Dataset`*. It has eight arguments, (some with meaningless names), six of which are optional.*

4.5 Commented Code

This bad smell occur throughout the codebase. Many methods have dead code snippets, a legacy from the modelling experiments. Code 4.5 shows the occurrence of this smell.

```
# append zeros before features
feature = torch.cat([feature, zeros], 1)
target = torch.FloatTensor([class_name])
# print("ERROR: Some sample in your dataset is less than %d seconds!
# Change the size of the overleapping
#           window"%self.c.dataset['window_len'])
# raise RuntimeError("ERROR: Some sample in your \
#       dataset is less than {} seconds!
# Change the size of the overleapping window
# (CONFIG.dataset['window_len'])".format(
#               self.c.dataset['window_len']))
```

Code 4.5: ***Commented Code:*** *This code snippet shows part of the method* `__getitem__` *in class* `Dataset`*. Lines four to six are part of a* `print`*, and lines seven to eleven are part of a raise an exception.*

4.6 Nested Statements

All methods have high cyclomatic complexity deriving from convoluted logic to cover different experimental path. Code 4.6 shows occurence of nested statements bad smell.

```
if (
    self.test_insert_noise
    and not self.c.dataset["temporal_control"] == "overlapping"
    and not self.c.dataset["temporal_control"] == "speech_t"
):
    features = []
    targets = []
    feature = feature.unsqueeze(0)
    target = target.unsqueeze(0)
    features.append(feature)
    targets.append(target)
    for _ in range(self.num_test_additive_noise):
        wav_noise = self.augment_wav.additive_noise(self.ap, wav)
        feature = self.ap.get_feature_from_audio(wav_noise)
        # transpose for (Batch_size, timestamp, n_features)
        feature = feature.transpose(1, 2)
        if self.c.dataset["temporal_control"] == "padding":
            # padding for max sequence
            zeros = torch.zeros(
                feature.size(0),
                self.max_seq_len - feature.size(1),
                feature.size(2),
            )
```

Code 4.6: ***Nested Statements:*** *This code snippet shows part of the method* `__getitem__` *in class* `Dataset`*. Lines twelve and seventeen are nested inside the statement in line one.*

Chapter 5

Production Architecture

This chapter presents the SPIRA *v2* architecture following practices that deal with the issues presented in Chapter 4. The next sections describe the SPIRA *v2* system using the **C4 model** (Brown, 2018). Four layers of abstractions describe the structure of the system, from lower to higher level of detail: **context**, **containers**, **components** and **code**. Afterward, Chapter 6 will compare the legacy and the new infrastructure, and Chapter 7 will discuss about the consequences of the suggested changes.

5.1 Context

The system context is the highest level of abstraction. It shows the system in the context of other systems or tools. The SPIRA system aims to be a tool for detecting respiratory insufficiency in hospitals, used by health professionals. The system is currently under construction, but professionals from different areas are working to deliver it into production. Since this is a consumer centered product that does not rely on external systems, the context diagram is trivial.

5.2 Containers

According to Brown, the system containers give a "high-level shape of the software architecture and how responsibilities are distributed across it" (Brown, 2018). Diagram 5.1 shows the system containers, which has five major services:

- **Data Collection**: collects patient data and store it in SPIRA's data lake (Housley and Reis, 2022; Kleppmann, 2017).
- **Training**: produces a neural network model from the input data.
- **Model Repository**: stores model artifacts and manages their versions (Lakshmanan, 2021).
- **Model Server**: uses a model artifact stored in the model repository to make inferences (Lakshmanan, 2021).
- **Inference App**: receives data, communicates with the model server, asks it to make an inference using a message-driven request-response communication protocol (Boner, 2016).

This research focus on the training container.

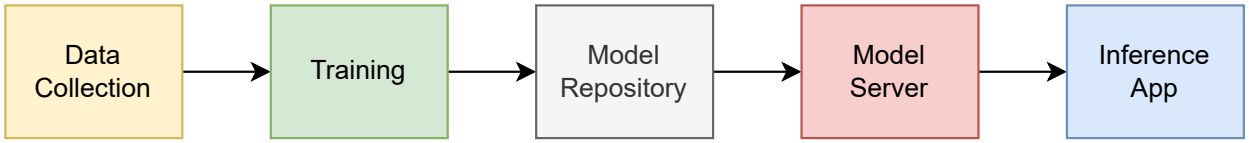

Diagram 5.1: ***SPIRA Container.***

5.3 Components

The system components give more detailed information of a container. The HEXAGONAL ARCHITECTURE pattern (Cockburn, 2005) was chosen to organize the training software. Figure 5.1 shows the file tree developed during this research, whose structure follows the HEXAGONAL ARCHITECTURE pattern.

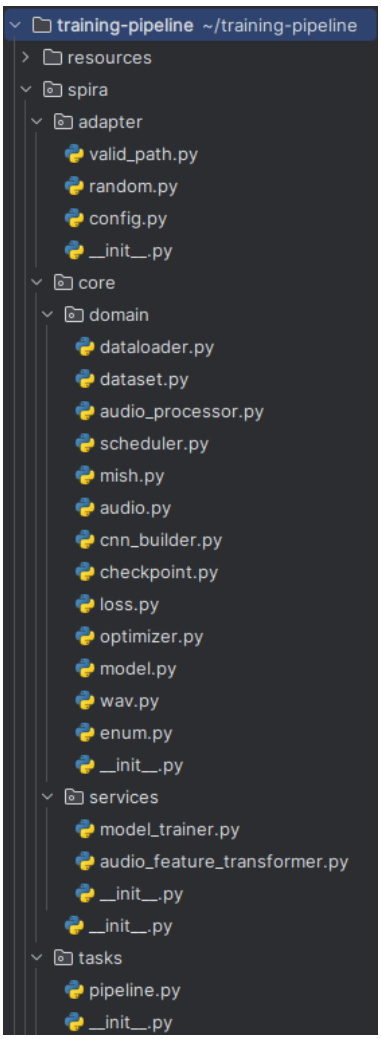

Figure 5.1: ***Hexagonal Architecture***

5.3.1 Core

The organization of the core follows the TRAINING PIPELINE pattern. Diagram 5.2 shows the components of the training pipeline, including:

- `Setup`: loads user defined configurations for the training pipeline.

- `Data Loading`: loads the raw data obtained by the data collection app.
- `Feature Engineering`: applies transformations (such as data processing and data augmentation) in the raw data to produce features.
- `Dataset Generation`: receives a set of features and labels, and generates train and test datasets.
- `Training`: uses the train dataset to learn the parameters of the convolutional neural network and uses the test dataset to validates the model performance.
- `Model Storage`: receives a model and stores it.

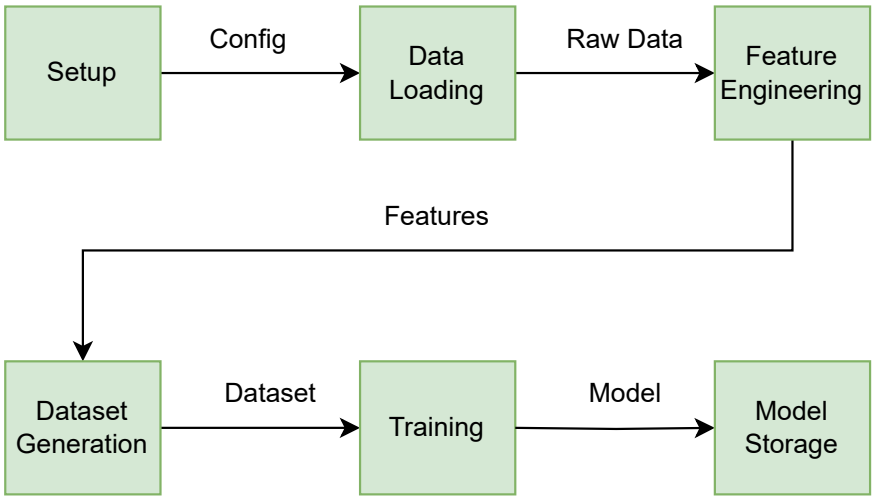

Diagram 5.2: ***Training Pipeline Components.***

5.3.2 Adapter

In this research, adapters perform auxiliary tasks, such as path validation, random number generation, and environment variable configuration.

5.3.3 Ports

Since the adapters in this research only performs auxiliary tasks, it was not necessary to create explicit classes or objects to represent ports, but rather use Python's dynamic duck typing to define these interfaces.

5.4 Code

The system code focus on the class-module design and implementation details. For sake of brevity, this section does not cover all details of each component above. The next subsections show key pieces of code to illustrate the adopted patterns and their respective principles.

5.4.1 `Pipeline (src/core/pipeline.py)`

The Code 5.1 exemplifies the adoption of the following pattern and principles:

- Mediator pattern: the training pipeline was broken into steps with one component (pipeline module) orchestrating their implementation.
- Dependency Inversion principle: the training pipeline doesn't depend on low-level implementation or details, but relies on high-level interfaces.
- Interface Segregation principle: the training pipeline is composed of several smaller specialized components.

5.4.2 `Random (src/adapters/random.py)`

The Code 5.2 exemplifies the adoption of the following patterns and principles:

- Strategy pattern: the class `Random` defines the strategy behavior, and the concrete classes `TrainRandom` and `TestRandom` have the alternative implementations of this behavior.
- Adapter pattern: the class `Random` is a wrapper around methods from different Python libraries, such as `random`, `torch`, and `numpy`.
- Factory Method pattern: The function `initialize_random` builds `Random` objects for clients of the module.
- Single Responsibility principle: the class `Random` encapsulates only the randomness functionality.
- Open-Closed principle: the class `Random` can be extended to create new behaviors while keeping the existing ones.
- Dependency Inversion principle: the class `Random` doesn't depend on low-level implementation or details, but relies on high-level interfaces.
- Liskov Substitution principle: objects of a superclass `Random` are replaced by objects of the subclasses `TrainRandom` and `TestRandom` without affecting the program's behavior.
- Design by Contract pattern: the function `initialize_random` assumes as invariant the argument `operation_mode` is valid, otherwise it would raise an error.

5.4.3 `Config (src/adapter/config.py)`

The Code 5.3 exemplifies the following patterns and principles:

- Singleton pattern: the function `load_config` guarantees the existence of only one instance of the `Config` class.
- Single Responsibility principle: the function `load_config` handles only the creation of a Config object.

```python
# Setup
################################################################################

config_path = ValidPath.from_str("/app/spira/spira.json")
config = load_config(config_path)

operation_mode = OperationMode.TRAIN
randomizer = initialize_random(config, operation_mode)

# Data Loading
################################################################################

patients_paths = read_valid_paths_from_csv(config.parameters.dataset.patients_csv)
controls_paths = read_valid_paths_from_csv(config.parameters.dataset.controls_csv)
noises_paths = read_valid_paths_from_csv(config.parameters.dataset.noises_csv)

patients_inputs = Audios.load(
    patients_paths,
    config.parameters.audio.hop_length,
    config.parameters.dataset.normalize,
)
controls_inputs = Audios.load(
    controls_paths,
    config.parameters.audio.hop_length,
    config.parameters.dataset.normalize,
)
noises = Audios.load(
    noises_paths,
    config.parameters.audio.hop_length,
    config.parameters.dataset.normalize,
)

# Feature Engineering
################################################################################

audio_processor = create_audio_processor(config.parameters.audio)

patient_feature_transformer = create_audio_feature_transformer(
    randomizer,
    audio_processor,
    config.options.feature_engineering,
    config.parameters.feature_engineering,
    config.parameters.feature_engineering.noisy_audio.num_noise_control,
    noises,
)

control_feature_transformer = create_audio_feature_transformer(
    randomizer,
    audio_processor,
    config.options.feature_engineering,
    config.parameters.feature_engineering,
    config.parameters.feature_engineering.noisy_audio.num_noise_control,
    noises,
)
```

Code 5.1: ***Training:*** *This code snippet shows part of the training pipeline that follows* Mediator *pattern, and* Dependency Inversion *and* Interface Segregation *principles.*

```python
class Random(ABC):
    def __init__(self, seed: int):
        self.seed = seed
        self.random_state = np.random.RandomState(self.seed)

    @abstractmethod
    def create_random(self, seed) -> Self:
        pass

    def apply_random_seed(self):
        random.seed(self.seed)
        torch.manual_seed(self.seed)
        torch.cuda.manual_seed(self.seed)
        np.random.seed(self.seed)

    @staticmethod
    def get_randint_in_interval(first: int, second: int) -> int:
        return random.randint(first, second)  # type: ignore

    @staticmethod
    def get_random_float_in_interval(first: float, second: float) -> float:
        return random.uniform(first, second)  # type: ignore

    @staticmethod
    def choose_n_elements(elements: list[Any], num_elements: int) -> list[Any]:
        return random.sample(elements, num_elements)  # type: ignore

    def get_probability(self, alpha: float, beta: float) -> float:
        return self.random_state.beta(alpha, beta)


class TrainRandom(Random):
    def create_random(self, seed) -> Random:  # type: ignore
        return cast(Random, self)


class TestRandom(Random):
    def create_random(self, seed) -> Random:  # type: ignore
        new_seed = self.seed * seed
        return cast(Random, TestRandom(seed=new_seed))


def initialize_random(config, operation_mode) -> Random:
    match operation_mode:
        case OperationMode.TRAIN:
            return TrainRandom(config.seed)
        case OperationMode.TEST:
            return TestRandom(config.seed)
        case _:
            raise RuntimeError("You must configure the operation mode to train or test")
```

Code 5.2: ***Random:*** *This code snippet shows the* `Random` *class that follows* Strategy, Adapter, Factory Method *and* Design By Contract *patterns, and* Single Responsibility, Open–Closed, Dependency Inversion *and* Liskov Substitution *principles.*

```python
def load_config(config_path: ValidPath) -> Config:
    config = read_config(config_path)
    validate_feature_engineering_options_or_raise(config.options.feature_engineering)
    validate_feature_type(config.parameters)
    return config
```

Code 5.3: ***Config:*** *This code snippet shows the method* `load_config` *that follows* Singleton *pattern, and* Single Responsibility *principle.*

5.4.4 AudioProcessor (src/core/domain/audio_processor.py)

The Code 5.4 exemplifies the adoption of three patterns:

- Template Method pattern: the method `create_transformer` is an abstract method defined by the superclass and implemented by the subclasses (including `MFCCAudioProcessor`, `SpectrogramAudioProcessor` and `MelspectrogramAudioProcessor`) create the attribute `transformer` in the superclass (`AudioProcessor`).
- Factory Method pattern: the method `create_audio_processor` builds objects of the class `AudioProcessor`.
- Strategy pattern: the class `AudioProcessor` defines the strategy behavior, and the three concrete subclasses `MFCCAudioProcessor`, `SpectrogramAudioProcessor`, and `MelspectrogramAudioProcessor` have the alternative implementations of this behavior.
- Single Responsibility principle: the class `AudioProcessor` encapsulates only the audio processing functionality.
- Open-Closed principle: the class `AudioProcessor` can be extended to create new behaviors while keeping the existing ones.
- Liskov Substitution principle: objects of the superclass `AudioProcessor` are replaced by objects of the subclasses `MFCCAudioProcessor`, `SpectrogramAudioProcessor` and `MelspectrogramAudioProcessor` without affecting the program's behavior.
- Dependency Inversion principle: the class `Random` doesn't depend on low-level implementation or details, but relies on high-level interfaces.

5.4.5 ValidPath (src/adapter/valid_path.py)

The Code 5.5 shows:

- Design by Contract: the function `create_valid_path` assumes as invariant the given `string` argument is a valid path, otherwise it would raise an error.
- Single Responsibility Principle: the function `create_valid_path` creates one instance of the `Path` class. The function `check_file_exists_or_raise` raise an error if the given path does not exist. The function `check_file_exists` returns a `boolean` if a given path exists.

```python
class AudioProcessor(object):
    def __init__(self, hop_length: int):
        self.hop_length = hop_length
        self.transformer = self.create_transformer()

    @abstractmethod
    def create_transformer(self):
        pass


class MFCCAudioProcessor(AudioProcessor):
    def __init__(self, config: MFCCAudioProcessorConfig, hop_length: int):
        super().__init__(hop_length)
        self.config = config

    def create_transformer(self):
        return MFCC(
            sample_rate=self.config.sample_rate,
            n_mfcc=self.config.num_mfcc,
            log_mels=self.config.log_mels,
            melkwargs={
                "n_fft": self.config.n_fft,
                "win_length": self.config.win_length,
                "hop_length": self.hop_length,
                "n_mels": self.config.num_mels,
            },
        )


class SpectrogramAudioProcessor(AudioProcessor):
    def __init__(self, config: SpectrogramAudioProcessorConfig, hop_length: int):
        super().__init__(hop_length)
        self.config = config

    def create_transformer(self):
        pass


class MelspectrogramAudioProcessor(AudioProcessor):
    def __init__(self, config: MelspectrogramAudioProcessorConfig, hop_length: int):
        super().__init__(hop_length)
        self.config = config

    def create_transformer(self):
        pass


def create_audio_processor(parameters: AudioProcessorParametersConfig):
    match parameters.feature_type:
        case AudioProcessorType.MFCC:
            return MFCCAudioProcessor(
                parameters.mfcc,
                parameters.hop_length,
            )
        case AudioProcessorType.SPECTROGRAM:
            return SpectrogramAudioProcessor(
                parameters.spectrogram,
                parameters.hop_length,
            )
        case AudioProcessorType.MELSPECTROGRAM:
            return MelspectrogramAudioProcessor(
                parameters.melspectrogram,
                parameters.hop_length,
            )
```

Code 5.4: ***Audio Processor:*** *This code snippet shows the* `AudioProcessor` *class that follows* Strategy, Template Method *and* Factory Method *patterns, and* Single Responsibility, Open–Closed, Dependency Inversion *and* Liskov Substitution *principles.*

```python
def check_file_exists(path: Path) -> bool:
    return os.path.isfile(path)


def check_file_exists_or_raise(path: Path):
    if not check_file_exists(path):
        raise FileExistsError(f"File {path} does not exist.")


def create_valid_path(unvalidated_path: str) -> Path:
    path = Path(unvalidated_path)
    check_file_exists_or_raise(path)
    return path
```

Code 5.5: ***Validation Path:*** *This code snippet shows functions that follow the* Single Responsibility *principle. And the* `create_valid_path` *function that follows* Design By Contract *pattern.*

Chapter 6

Infrastructure

This chapter presents the SPIRA *v1* and SPIRA *v2* infrastructure. This research did not focus on the infrastructure, so there is not much to discuss. However, the proposed changes to the infrastructure were essential for the development of the SPIRA *v2* architecture.

6.1 Legacy Infrastructure

The experimental training infrastructure relies on a Python runtime called Anaconda[1] and Bash scripts to test the model. The dependency management is handled by Anaconda via a `.txt` file and a `.yaml` file. Unfortunately, Anaconda does not resolve the versions of the subdependencies, which can lead to inconsistencies.

6.2 New Infrastructure

The proposed training infrastructure aims to solve the consistency issues from the legacy infrastructure, using tools such as:

- **Docker:** a platform for developing, shipping, and running applications in containers.
- **Docker Compose:** a tool for defining and running multi-container Docker applications.
- **Poetry:** a dependency management and packaging tool for Python projects.
- **Makefile:** a script that defines and executes a series of tasks or commands needed to build and manage a project.

This infrastructure achieves the desired consistency. Docker and docker compose guarantees environment reproducibility. Poetry solves the issue of subdependencies with `lockfile`. Makefile modernizes automation to simplify the use both tools in development.

[1] https://anaconda.org/

Chapter 7

Results

This chapter explains how the proposed changes in Chapter 5 added quality attributes to SPIRA's system, solving the lack of code quality in SPIRA *v1*.

7.1 Readability

The code is cleaner and more readable when BAD SMELLS are removed. Commented codes have been removed throughout the code. The adoption of the SINGLE RESPONSIBILITY PRINCIPLE in all the classes and functions allows the removal of the following BAD SMELLS:

- Functions descend more than one level of abstraction
- Misplaced responsibility
- Functions with too many arguments

The adoption of the FACTORY METHOD pattern showed in subsection 5.4.2 and 5.4.4, and the TEMPLATE METHOD pattern showed in subsection 5.4.4 allowed the removal of nested statements, improving the code readability.

7.2 Understandability

The adoption of the MEDIATOR pattern showed in subsection 5.4.1 and the STRATEGY pattern in subsection 5.4.2 and 5.4.4 allowed developers and data scientists to understand the logic, behavior, and structure of each module.

The HEXAGONAL ARCHITECTURE pattern isolates the business logic from external dependencies, expressing it in a clearer way. For example, the `Config` class (in subsection 5.4.3) became an adapter, removing code related to reading environment variables from the middle of the training pipeline.

7.3 Efficiency

The adoption of the FACTORY METHOD pattern showed in subsection 5.4.2 and 5.4.4, and the TEMPLATE METHOD pattern showed in subsection 5.4.4 allowed the removal of nested (`if` and `for`) statements reducing the cyclomatic complexity and improving the system efficiency.

7.4 Robustness

The Design by Contract guarantees invariants at the beginning of the code, unlike defensive programming, where the invariant is guaranteed in the middle of the code. The adoption of the Design by Contract showed in subsection 5.4.5 and in 5.4.2 allowed the removal of defensive programming, decreasing the reprocessing cost.

The adoption of the Singleton pattern in subsection 5.4.3 guarantees one instance of the training configuration. Guaranteeing the immutability of the configuration. This means that there will be no disruption if more than one configuration file is loaded throughout the program's life cycle.

The new infrastructure in section 6.2 provides a consistent and isolated environment, ensuring that the SPIRA *v2* runs consistently through different environments. With a streamlined dependency management, there is less risk of compatibility issues caused by Python dependencies.

7.5 Reusability

The proper use of object-oriented programming and further modularization enabled reusability, as seen in the training pipeline in subsection 5.4.1, where `patients_paths`, `controls_path` and `noises_path` reuse the same function.

In addition, the use of inheritance allowed the `Random` (subsection 5.4.2) and `AudioProcessor` (subsection 5.4.4) subclasses to reuse certain features already implemented.

7.6 Extensibility

The adoption of the Strategy pattern and the Factory Method the in classes `Random` (subsection 5.4.2) and `AudioProcessor` (subsection 5.4.4) allowed the creation of subclasses extend specific pieces of functionality.

The adoption of the Mediator pattern in the training pipeline allowed changing the workflow of the pipeline regardless of the implementation of the steps. The adoption of the Adapter pattern in subsection 5.4.2 made it possible to create new methods that make it easier to use the libraries wrapped up in the `Random` class. Overall, these patterns embraced the use of Open Closed Principle allowing developers to create extensions without modifying the existing code.

The use of the Template Method pattern in subsection 5.4.4 made the sub classes of the `AudioProcessor` class to customize specific steps of the feature engineering. This allows for variations in behavior while maintaining a consistent framework.

In the Hexagonal Architecture pattern, ports are the contract that adapters need to follow to interact with the core, which allows the interaction with external dependencies to be changed without affecting the business logic. For example, the `Random` port naturally becomes a candidate for implementing the Strategy pattern. This makes it easier to create automated unit tests for the core independently of the external dependencies wrapped by adapters.

7.7 Maintainability

Adopting the Strategy and Factory Method patterns minimized the effort, time and cost required to evolve the system according to changing requirements over time. In addition, unit testing is simplified because clients can depend on abstract factories, which allows for easier substitution of mock objects during testing. The same reasoning can be applied to subsection 5.4.4.

The use of the Mediator pattern in subsection 5.4.1 reduces the dependencies between components, making the system more modular. With that, it is easier to test and maintain each module. With centralized control logic, debugging becomes more straightforward. Issues can be traced more easily within the mediator, simplifying the identification and resolution of problems.

The new infrastructure presented in section 6.2 enables the of use the same Docker image in development, testing, and production. This ensures that the development environment closely matches the production environment, making delivery easier. Developers and data scientists can experiment safely in a containerized environment that closely mirrors the production setup. This ensures that deployments are reproducible and can be rolled back in case of problems.

Chapter 8

Conclusions

SPIRA is a multidisciplinary project created during the COVID-19 pandemic for pre-diagnosis of insufficiency respiratory based on speech analysis. The implementation created during the modeling phase of SPIRA (Casanova et al., 2021) led to an accidental architecture that lacked good software quality attributes, converging to a Big Ball of Mud anti-pattern. This research analyzed how bad smells and high cyclomatic complexity impacted this first version of SPIRA, justifying the need for a reimplementation.

The goal of this research is to design a continuous training pipeline to enable MLOps for SPIRA. Based on the issues found in the experimental implementation, this research proposed a new architecture for the system. It combined design principles and design patterns with the hexagonal architecture pattern to support several quality attributes: maintainability, extensibility, reusability, robustness, efficiency, understandability, and readability. With these quality attributes, the training pipeline can be easily modified and evolved alongside its three axes of change: code, model, and data. Furthermore, this research also improved the infrastructure to ensure the system is easy to run. This is essential to continuously deliver new versions of the SPIRA model. Together, these changes enable MLOps practices in the project.

For future work, there are three steps required to deliver the implementation of the new continuous training pipeline. First, the codebase requires automated tests. Given the complexity of the experimental architecture, the code produced during this research was made without automated tests. This allowed flexibility for the incremental redesign of all the components present in the legacy codebase. Now that the new design is stable, automated tests will improve maintainability and robustness of the system.

Second, the system has to be integrated with the model repository MLFlow[1]. This tool will work as the intermediary between the training pipeline and the model server. It will store model artifacts as well as metadata about the training process. This integration also did not make sense for the current state of the project, since the feature engineering and training had to be redesign before an integration with this third-party tool.

Lastly, the system needs to be integrated with task scheduler such as Airflow[2]. This tool will provide a simple interface to run the training pipeline. However, this integration makes sense only when the system is production ready, that is, after the implementation of automated tests.

[1] https://mlflow.org/
[2] https://airflow.apache.org/

Appendix A

Neural Networks

A.1 Definition

Neural networks are a computational technique with the goal of learning patterns from data. Artificial neural networks came from the first works that attempted to model networks of neurons in the brain (McCulloch and Pitts, 1943). As any other machine learning model, a neural network receives a set of input data, processes it, and generates a set of outputs — which can be numbers or categories. Figure A.1 shows a schematic representation of an artificial neural network.

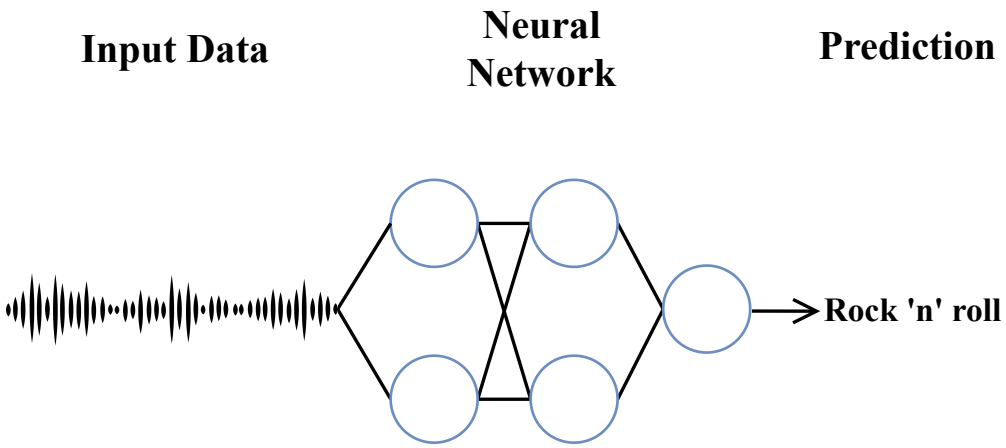

Figure A.1: ***Schematic representation of an artificial neural network.*** *In this example, it receives a sound wave sampled from a song and classifies its genre in a predefined category.*

A.1.1 Unit

Neural networks are constituted by a set of nodes. Each node is called a **unit** or a **neuron**. Each unit has at least one connection to another unit. A neuron is a function that takes a set of real valued numbers as input, performs some computation on them, and produces an output.

This computation is a weighted sum of its inputs with one additional term known as the **bias term**. It allows the result to be nonzero even when the inputs are zero. Afterwards, the unit applies an often non-differentiable **activation function** – also known as a **step function** – resulting finally in its outputs. The data flows through the network from the initial inputs to the final outputs.

To illustrate how the computation in an unit is calculated, take the example of the **perceptron** shown in Figure A.2, which is a single-unit neural network for linear classification that results in a

binary output.

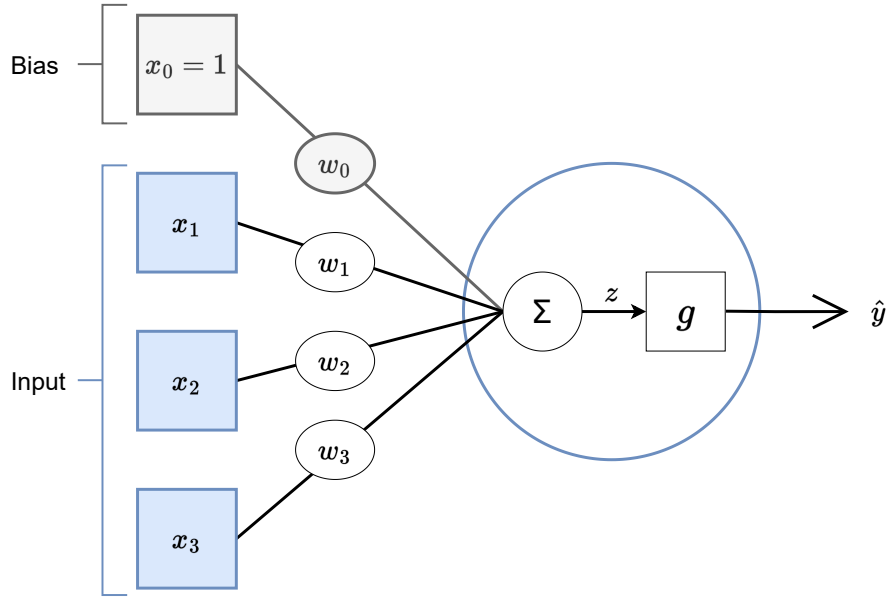

Figure A.2: ***Perceptron:*** *The neuron unit takes three inputs x_1, x_2 and x_3, and the bias term $x_0 = 1$. For each input, it has corresponding weights w_1, w_2 and w_3, as well a weight w_0 for the bias term. The output of the weighted sum is z, which is fed into the activation function g then producing the output $\hat{y}$.*

The weighted sum can be represented as:

$$z = \sum_{i=0}^{n} x_i \cdot w_i. \tag{A.1}$$

After the calculation, the neuron applies its activation function g to the result z. For the perceptron, the output of the neuron is also the final output of the neural network, denoted by $\hat{y}$. Therefore,

$$\hat{y} = g(z), \tag{A.2}$$

and the perceptron can be summarized as:

$$\hat{y} = g\left(\sum_{i=0}^{n} x_i \cdot w_i\right). \tag{A.3}$$

A.1.2 Layers

Neural networks can have more than one unit, which can be organized into **layers**. A **feed-forward neural network** (FNN) is a multi-layer network. It has connections only in one direction, i.e., the outputs from one layer become inputs to the successor layer. Therefore, there is no feedback to predecessor layers and the network's architecture forms a directed acyclic graph.

A feed-forward neural network is also called a **multi-layer perceptrons**. However, units in a FNN are not perceptrons, since the units in a perceptron have differentiable activation functions whereas the units in a FNN do not.

Commonly, a FNN is also a **fully connected network**, meaning that inputs of a unit in the layer k come from the outputs of all the units in the predecessor layer $k-1$. Therefore, every pair of units from two adjacent layers is connected.

There are three kinds of layers:

- **Input layer**: It is the set of **input units**, denoted by $\boldsymbol{x}$. Each unit j in this layer has a scalar value x_j that represents a **feature** of the input data. All the input units pass their data to the units in the next layer.
- **Hidden layer**: It is a set of **hidden units**, denoted by $\boldsymbol{h_i}$, where $1 \leq i \leq n$, and n is the number of hidden layers. Each unit j in a hidden layer i is represented as $h_{i,j}$. Together, the hidden units compute the intermediate outputs of an artificial neural network.
- **Output layer**: It is the set of **output units**, denoted by $\hat{\boldsymbol{y}}$. Each unit j in this layer computes the final output $\hat{y}_j$, also known as a **prediction**.

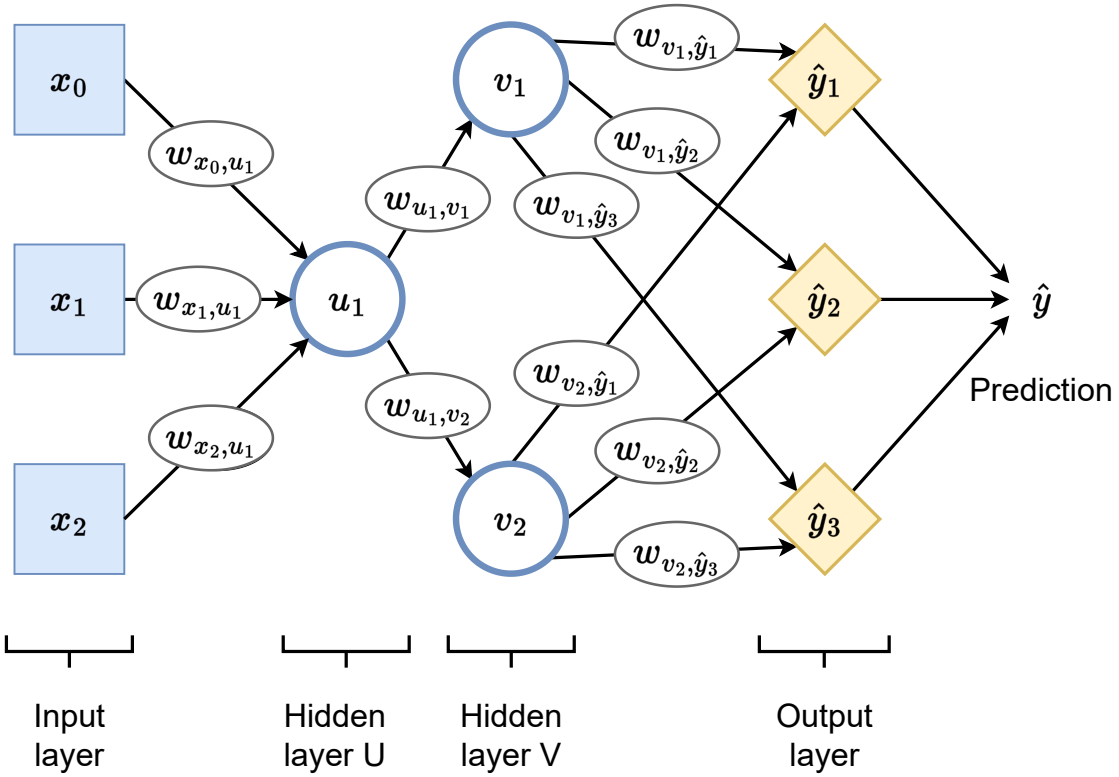

Figure A.3: ***A three layer FNN*** *with one input layer, two hidden layers, and one output layer. The input layer is usually not counted when enumerating layers. The hidden layers were represented as* $\boldsymbol{u}$ *and* $\boldsymbol{v}$ *instead of* $\boldsymbol{h_1}$ *and* $\boldsymbol{h_2}$ *to simplify the image.*

Units in the input layer are represented in a different format due to their different behavior from other units in the feed-forward neural network. Input units don't do any computation. They only store scalar values derived from the raw data (features).

Both hidden and output units make a weighted sum of their inputs and apply an activation function, resulting in the output. The outputs of a hidden unit are the input for other units. These associations are shown in Figure A.3.

Consider now a neural network similar to Figure A.4. Let's generalize the computation made for hidden and output units:

- **First hidden layer computation**: Let $a_{1,j}$ denote the output of the hidden unit $h_{1,j}$ in the first hidden layer. Let $w_{x_k,h_{1,j}}$ be the weight between units x_k and $h_{1,j}$. Let $g_{1,j}$ denote the

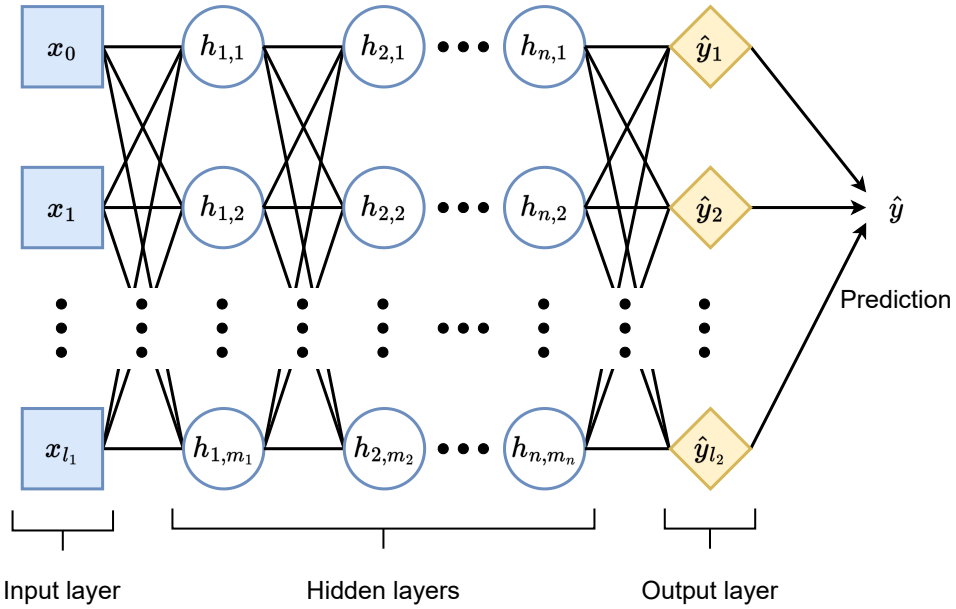

Figure A.4: ***A generic FNN*** *with one input layer, n hidden layers, and one output layer. The weights were hidden in the image to simplify it.*

activation function associated with the unit $h_{1,j}$. Then we have:

$$a_{1,j} = g_{1,j}\left(\sum_k x_k \cdot w_{x_k,h_{1,j}}\right) \equiv g_{1,j} \cdot z_{1,j}, \tag{A.4}$$

where $0 \leq k \leq l_1$ indicates each input unit, and $1 \leq j \leq m_i$ indicates each unit in the hidden layer $i = 1$.

- **Other hidden layers computation**: Let $a_{i,j}$ denote the output of the hidden unit $h_{i,j}$. Let $w_{h_{i-1,d},h_{i,j}}$ be the weight between units $h_{i-1,d}$ and $h_{i,j}$. Let $g_{i,j}$ be the activation function associated with the unit $h_{i,j}$. Then we have:

$$a_{i,j} = g_{i,j}\left(\sum_i a_{i-1,d} \cdot w_{h_{i-1,d},h_{i,j}}\right) \equiv g_{i,j} \cdot z_{i,j}, \tag{A.5}$$

where $2 \leq i \leq n$ indicates which hidden layer that unit belongs, $1 \leq j \leq m_i$ indicates each unit in the hidden layer i. And, $1 \leq d \leq m_{i-1}$ indicates each unit in the hidden layer $i-1$.

- **Output layer computation**: The outputs of the last hidden layer $\boldsymbol{h_n}$ are inputs for the units in the output layer.

  Let $\hat{y}_k$ denote the output of the $k-$th unit in the output layer. Let $h_{n,j}$ denote a hidden unit in the last hidden layer $\boldsymbol{h_n}$. Let $a_{n,j}$ denote the output of unit $h_{n,j}$. Let $w_{h_{n,j},k}$ denote the weight between the $h_{n,j}$ and the $k-$th unit in the output layer. Let g_k denote the activation function associated with the kth unit in the output layer. Then we have:

$$\hat{y}_k = g_k\left(\sum_j a_{n,j} \cdot w_{h_{n,j},k}\right) \equiv g_k \cdot z_k, \tag{A.6}$$

where $1 \leq k \leq l_2$ indicates each output unit, $h_{n,j}$ is a unit in the last hidden layer $\boldsymbol{h_n}$, and $1 \leq j \leq m_n$ indicates each unit in the hidden layer n.

The arrangement of neurons into layers allows a neural network to derive features from its input, thus enabling it to identify patterns in its training data. Neural networks have the ability to approximate complex functions and classify data across an array of domains, such as image classification, natural language processing, speech recognition, and others.

When a neural network has many hidden layers, it is known as **deep neural networks**. The field of **deep learning** is dedicated to study and apply these neural networks.

A.1.3 Training and optimization process

In **supervised learning**, a neural network is trained using the **back-propagation** algorithm. This algorithm adjusts the values of the neural network's weights. These weights are **parameters** used to make predictions. For that, it relies on **labeled** data. The algorithm has the following steps:

1. **Weights initialization**: The weights in a neural network are initialized randomly.
2. **Forward**: Input data is fed forward through the network, from the input layer to the output layer.
3. **Loss computation**: Compare the network's predictions to the ground truth labels, calculating a value known as **loss**. The choice of loss function depends on the problem, as detailed below.
4. **Backward**: For each layer, starting from the output layer to the input layer (in the reverse direction of the forward step), compute the gradient of the loss with respect to the weights of that layer. Update the weights using an optimization algorithm.
5. **Repeat**: Repeat steps 1 to 4 for a specified number of iterations (**epochs**), or until the loss converges.

Figure A.5 shows the training steps of a neural network following the back-propagation algorithm.

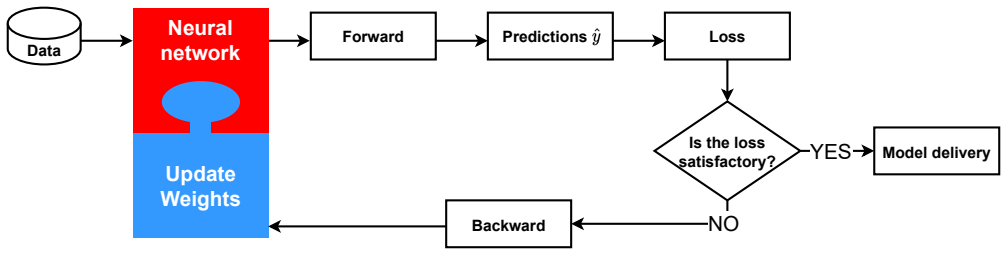

Figure A.5: ***Training steps of a neural network.***

During training, the goal is to minimize the loss function, which improves the neural network's overall predictive **accuracy**. The **loss function** is a mathematical function that indicates the magnitude of the error that the neural network has made in its prediction (Raff, 2022; Russell and Norvig, 2021; Trochim, 2017). There are different techniques for reducing the loss function. One of them is the gradient descent. Furthermore, there are different types of loss functions.

The one used in the SPIRA project is known as **cross-entropy** (Casanova et al., 2021). It goes beyond the scope of this research to cover the details of minimizing the loss function. For more information, refer to Russell and Norvig.

A.2 Convolutional Neural Networks

Convolutional Neural Networks (CNNs) are a specialized type of neural network that captures spatial hierarchies and local patterns within the input data. As the name suggests the key operation is the **convolution**.

For a given input vector $\mathbf{x}$ of size n and a kernel $\mathbf{k}$ of size l, the convolution operation represented as $*$ is defined as:

$$(\mathbf{x} * \mathbf{k})_i = \sum_{j=1}^{l} k_j x_{j+i-(l+1)/2}. \tag{A.7}$$

For each position i, the operation represents the dot product between the kernel $\mathbf{k}$ and a segment of $\mathbf{x}$ centered on x_i with width l.

In a CNN, there are **convolutional layers** that apply the convolutional operation on their input to calculate their output. **Downsampling** is a process that can be used in convolutional layers to reduce the sampling rate of a data sequence.

The SPIRA model is a CNN that uses convolutional layers to downsample audio signals recorded from voice speech to identify respiratory insufficiency. Discussing all the details of the SPIRA model goes beyond the scope of this research. For more information, check the paper by Casanova et al..

Bibliography

Kent Beck and Martin Fowler. Bad smells in code. **Refactoring: Improving the design of existing code**, 2018-Janua:75–88, 1999. 5

Jonas Boner. **Reactive Microservices Architecture Design Principles for Distributed Systems**. 2016. 13

Simon Brown. The C4 model for visualising software architecture. **Infoq.Com**, 2018. 13

Edresson Casanova, Lucas Gris, Augusto Camargo, Daniel da Silva, Murilo Gazzola, Ester Sabino, Anna S. Levin, Arnaldo Candido, Sandra Aluisio, and Marcelo Finger. Deep Learning against COVID-19: Respiratory Insufficiency Detection in Brazilian Portuguese Speech. In **Findings of the Association for Computational Linguistics: ACL-IJCNLP 2021**, 2021. doi: 10.18653/v1/2021.findings-acl.55. 1, 2, 8, 26, 32

Alistair Cockburn. Hexagonal Architecture, 2005. URL https://alistair.cockburn.us/hexagonal-architecture/. 14

Edsger W. Dijkstra. On the Role of Scientific Thought. In **Selected Writings on Computing: A personal Perspective**. 1982. doi: 10.1007/978-1-4612-5695-3{_}12. 7

Renato Cordeiro Ferreira, Dayanne Gomes, Vitor Tamae, Francisco Wernke, Alfredo Goldman, and Marcelo Finger. SPIRA: Building an Intelligent System for Respiratory Insufficiency Detection. 2022. doi: 10.5753/ise.2022.227048. 2

E Gamma, R Helm, R Johnson, and J Vlissides. **Design Patterns: Elements of Reusable Software**. 1996. 5

Vitor Guidi. **Aprendizagem de Máquina Nativa à Nuvem: Provendo Alta Disponibilidade ao Sistema SPIRA com Kubernetes**. PhD thesis, Instituto de Matemática e Estatística da Universidade de São Paulo, 2023. 3

Matt Housley and Joe Reis. **Fundamentals of Data Engineering**. O'Reilly Media, 6 2022. 13

Martin Kleppmann. **Designing Data-Intensive Applications: The Big Ideas behind Reliable, Scalable, and Maintainable Systems**. 2017. 13

Valliappa Lakshmanan. **Machine learning design patterns : solutions to common challenges in data preparation, model building, and MLOps / Valliappa Lakshmanan, Sara Robinson, and Michael Munn.** 2021. 13

Robert C. Martin. Clean Code: A Handbook of Agile Software Craftsmanship. **Kybernetes**, 38 (6), 2009. ISSN 0368-492X. doi: 10.1108/03684920910973252. 4

J Mccabe. THOMAS J. McCABE. **IEEE Transactions on Software Engineering**, SE-2(4): 308–320, 1976. 5

Warren S. McCulloch and Walter Pitts. A logical calculus of the ideas immanent in nervous activity. **The Bulletin of Mathematical Biophysics**, 5(4), 1943. ISSN 00074985. doi: 10.1007/BF02478259. 27

Edward Raff. **Inside Deep Learning**. 2022. 31

Stuart Russell and Peter Norvig. **Artificial Intelligence: A Modern Approach (Global Edition)**. 2021. 31, 32

Danilo Sato, Arif Wider, and Christoph Windheuser. Continuous Delivery for Machine Learning. **Martin Fowler**, 2019. 1

Girish Suryanarayana, Ganesh Samarthyam, and Tushar Sharma. **Refactoring for Software Design Smells: Managing Technical Debt**. Elsevier Inc., 11 2014. ISBN 9780128016466. doi: 10.1016/C2013-0-23413-9. 4, 5

Vitor Tamae. **Building an Inteligent System to Detect Respiratory Insufficiency**. PhD thesis, Instituto de Matemática e Estatística da Universidade de São Paulo, 2022. 3

Piotr Trochim. What is a loss function in simple words?, 2017. URL https://stackoverflow.com/questions/42877989/what-is-a-loss-function-in-simple-words?rq=4. 31

Joseph Yorder and Brian Foote. Big Ball of Mud. **ACM SIGPLAN Notices**, 24(11), 11 1999. 7