Freie Universität Berlin

Master's Thesis at the Department for Informatics and Mathematics

Implementation and test of an infrastructure for automatic evaluation of electrocardiograms

Thierry Meurers
Matriculation Number: 4663215
thierry.meurers@fu-berlin.de

First Examiner: Prof. Dr. Tim Landgraf
Second Examiner: Barry Linnert
Supervisor: Dr. Peter Liebisch

Berlin, July 8th, 2019

Abstract

Artificial pacemakers and implantable cardioverter-defibrillators (ICDs) provide life-sustaining therapies for those affected by cardiac diseases. With the aim of supporting follow-up care modern implants transmit diagnostic data to the manufacturer. The data includes an electrocardiogram (ECG) and is made available to the attending physician. An additional monitoring by means of machine learning methods is largely unresearched but holds great potential considering the recent advances in medical data science.
In the first part of the thesis an infrastructure that enables research on the transmitted ECGs was developed. It allows the data to be accessed language-independent and approximately 120 times faster compared to a formerly utilized approach. Secondly, two machine learning methods originally designed for surface ECGs were tested on intracardiac signals. The first classifier was based on manual feature engineering. The second employed a convolutional neural network (CNN). Both were used to distinguish between supraventricular tachycardia (SVT) and ventricular tachycardia (VT) addressing current issues in ICD therapy. Compared to conventional SVT/VT detection algorithms they received a significantly smaller part of the ECG. With an accuracy of 94.9 % the CNN outperformed the manual feature engineering (86.0 %) and equaled the performance of the classifier used to annotate the training data.
The tested classifiers indicate not only that research on surface ECGs is transferable to intracardiac signals but also showed the importance of the new infrastructure.

Eidesstattliche Erklärung
Ich versichere hiermit an Eides statt, dass diese Arbeit von niemand anderem als meiner Person verfasst worden ist. Alle verwendeten Hilfsmittel wie Berichte, Bücher, Internetseiten oder ähnliches sind im Literaturverzeichnis angegeben, Zitate aus fremden Arbeiten sind als solche kenntlich gemacht. Die Arbeit wurde bisher in gleicher oder ähnlicher Form keiner anderen Prüfungskommission vorgelegt und auch nicht veröffentlicht.

Berlin, July 8th, 2019
Thierry Meurers

Contents

Introduction

Motivation

Implantable pacemakers and defibrillators provide effective therapies for potentially life-threatening cardiac diseases. While pacemakers are used for treating abnormal slow heart rates, defibrillators are additionally capable of terminating morbidly fast or unsynchronized contractions of the heart muscle. To improve therapy modern implants use telemetry and transmit diagnostic data to the manufacturer at regular intervals. After a certain time or following an arrhythmia an electrocardiogram (ECG) is transmitted. The collected data is made available to the attending physician to facilitate follow-up care.
Additionally, the manufacturer could use the data for computer-aided diagnostics or early detection of device related problems. This is especially interesting as the evaluation of ECGs by means of machine learning techniques developed rapidly within the last decade. However, research is almost exclusively done on surface ECGs. To what extent the existing methods can be used to evaluate the signals recorded by implants is hardly investigated.

Scope

The first scope of this work is to build an infrastructure that enables research on the collected electrocardiograms. It will specifically be designed for scientific purposes and should facilitate the test of new evaluation techniques. The following aspects will be considered: Support of multiple programming languages, fast, network-independent access, anonymization, scalability and maintainability.
Using the newly developed infrastructure the second scope is to test two machine learning approaches on intracardiac signals. The first approach is based on manual feature engineering. The second employs a convolutional neural network. Both were originally designed for surface ECGs and will be used to distinguish between supraventricular and ventricular tachycardia. Distinguishing these rhythms can be difficult as they have similar ECGs but is crucial because only the latter requires therapy.

Outline

The thesis is separated into four parts. Firstly, a brief introduction into the human heart's physiology, different arrhythmias, electrocardiography and the therapy provided by cardiac implants is given (1). The second part focuses on the infrastructure and its development. Special attention is paid to explaining the requirements and considerations addressed during the infrastructure's design (2). The third part begins by reviewing related work previously done on comparable classification tasks. Subsequently, two of the reviewed methods are selected, explained in more detail and tested on intracardiac ECGs (3). The findings obtained during testing the infrastructure and the classifiers are discussed in part four. Thereafter, the thesis is assessed as a whole by summing up the findings and providing an outlook (4).

Chapter 1

Origin, treatment and monitoring of cardiac arrhythmias

1.1 Cardiovascular system

The cardiovascular system of the human body is separated into the systemic circulation and the pulmonary circulation. The systemic circulation is responsible for providing oxygen and nutrients to organs and tissues. The pulmonary circulation is used to carry oxygen-poor blood to the lungs where it can get oxygenated again. Both circulatory systems are connected to each other by the heart.
The heart is a hollow muscle and is separated into two sides (left side and right side). Each side has an upper chamber (atrium) and a lower chamber (ventricle). The blood flow between atrium and ventricle is controlled by the tricuspid valve (on the right side) and the mitral valve (on the left side).

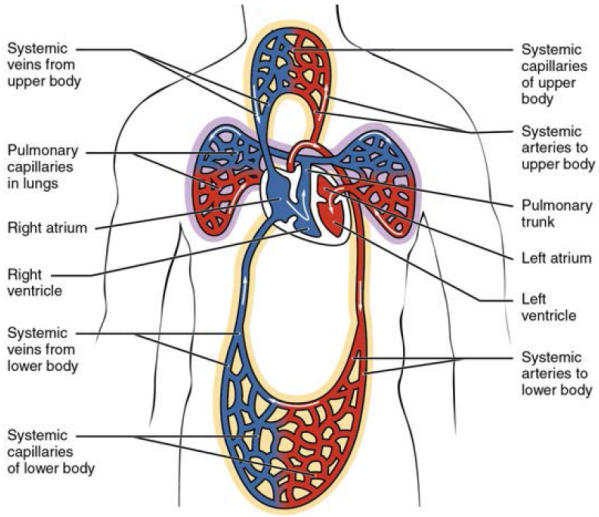


Figure 1.1: Simplified illustration of the cardiovascular system of the human body [7]

1.2 Cardiac cycle

The heart performs the following steps to propel blood through the body:

1. Filling: The cardiac cycle starts when atria and ventricles are relaxed (diastole). Oxygen-poor blood from the systemic circulation is flowing to the right atrium and oxygen-rich blood from the pulmonary circulation is flowing to the left atrium. The two valves between the atria and ventricles are open, allowing the blood to flow into the ventricles.

2. Atrial Systole: The contraction of the heart (systole) starts in the upper chambers. Both atria contract to push blood from the upper chambers to the lower chambers in order to finalize the filling of the ventricles.

3. Ventricular Systole: After a short delay the contraction of the lower chambers starts. The valves between atria and ventricles close on account of the rising pressure of the ventricles. Once the ventricles' pressure exceeds the pressure of the aortas the blood is ejected back to the circulatory systems.

1.3 Electrical conduction

Adult humans have a resting heart rate from 60 to 100 beats per minute (bpm). In the heart of a healthy person the cardiac rhythm is initiated by the sinoatrial (sinus) node located in the right atrium. As the natural pacemaker of the heart, the sinus node can adjust the heart rate depending on physical activity or mental stress. The impulses generated by the sinus node are spread to the rest of the heart using two mechanisms:

1. The cells of the cardiac muscles are capable of triggering each other. Once a cell is triggered (depolarized), it will activate its neighbor cells resulting in a contraction of the whole muscle. A depolarized cell has to be repolarized before it can be triggered again. The time needed for repolarization is called refractory period and is important for ensuring a controlled contraction.
Upper and lower chambers of the heart are insulated from each other, meaning the simple cell-to-cell pathway cannot be used to trigger the contraction of the ventricles.

2. In order to transmit the impulse generated in the right atrium to the lower chambers an internodal pathway, starting at the atriventricular (AV) node, is used. The AV node is connected to the His bundle which divides into the right and left bundle branches. Finally the signal reaches the Purkinje fibers which trigger the contraction of the heart muscle cells in the ventricles.
The AV node delays the transmission of the impulse by approximately 100 ms. This entails the pause between atrial an ventricular systole (described in 1.2). Furthermore, the AV node serves as a frequency filter as it can only transmit impulses up to 220 bpm.

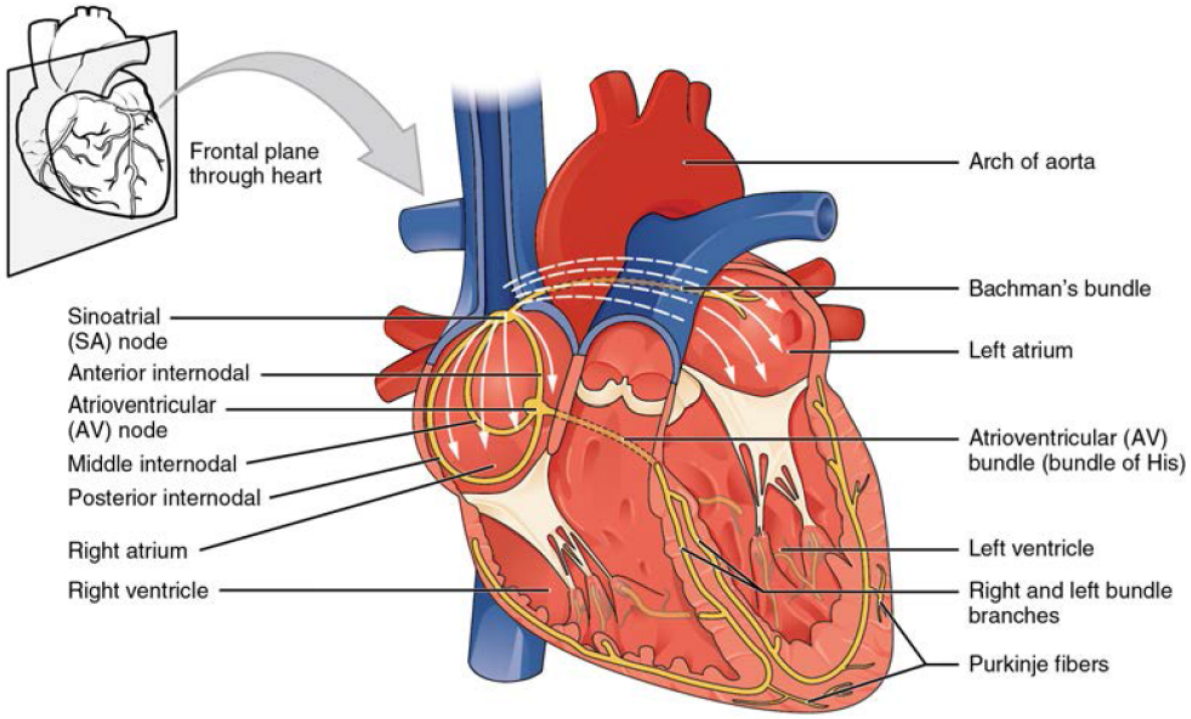


Figure 1.2: Conduction system of the human heart [7]

1.4 Electrocardiography

Cardiac activity can be visualized by measuring the electrical discharges arising from depolarization and repolarization of cardiac cells. The process of measuring is called electrocardiography and generates an output called electrocardiogram (ECG). For examination electrodes are attached to the body. The placement of electrodes depends on the aim of the examination. Most configurations include at least one electrode attached to each arm. A varying number of additional electrodes can be attached to the chest or to lower parts of the body.

The heart of a healthy human creates a repeating pattern composed of the following parts:

- The **P wave** represents the depolarization and contraction of the atria.
- The **QRS complex** represents the depolarization of the ventricles. The peak (**R wave**) is the point of ventricular contraction. As the ventricles are larger in size than the atria, they generate a stronger electrical signal. The delay between atrial and ventricular contraction is called **PR interval**.
- The **T wave** represents the repolarization of the ventricles. Once they are repolarized they can get triggered again. The atrial repolarization cannot be seen because it occures during the QRS complex.

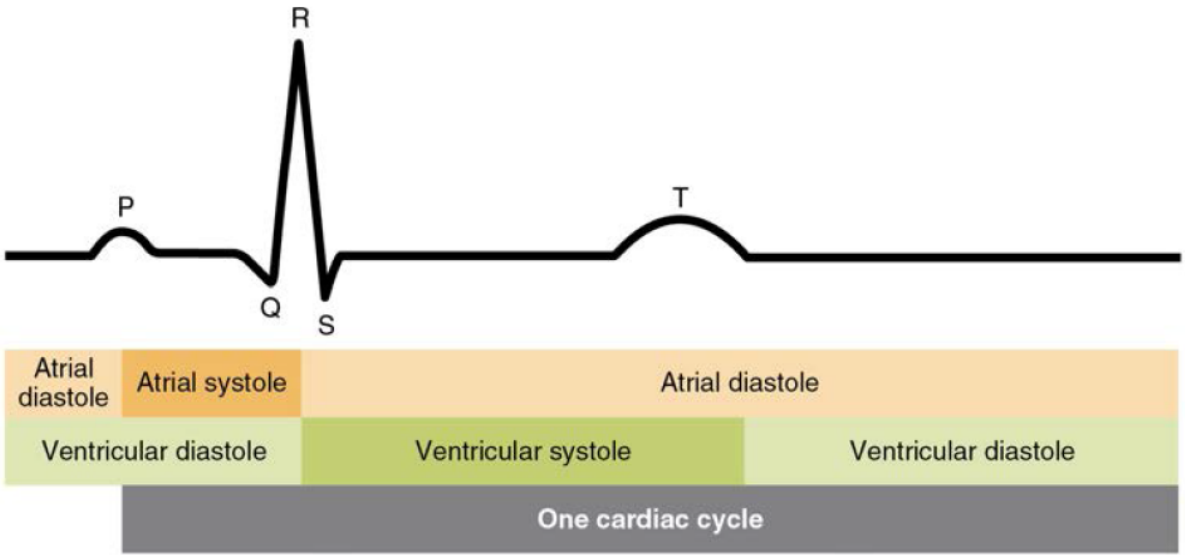


Figure 1.3: Relationship between cardiac cycle and ECG [7]

1.5 Cardiac arrhytmias

Disorders of the electrical conduction system can cause an abnormal heart rhythm (Arrhythmia). Abnormal rhythms include the heart beating too slow (Bradycardia) or too fast (Tachycardia).

1.5.1 Bradycardia

A resting heart rate under 60 bpm is called bradycardia. Possible symptoms (especially with a heart rate under 50 bpm) include weakness, dizziness, fainting and respiratory distress. If no symptoms occur the bradycardia is not considered clinically significant.
Common reasons for bradycardia are problems with the sinus node or the AV node. Especially in older adults these components tend to fail making age a risk factor for bradycardia. A dysfunction of the sinus node results in a slower and irregular stimulation of the whole heart. The dysfunction of the atrioventricular node could impede the transmission of impulses to the ventricles (AV block). This means not all impulses are transmitted to the lower chambers or that the transmission is delayed for too long.

1.5.2 Tachycardia and fibrillation

A resting heart rate over 100 bpm in adult humans is called tachycardia. Possible symptoms include dizziness, shortness of breath, heart palpitations, chest pain and fainting. It is important to note that tachycardia refers to a fast but organized contraction of the lower chambers. The heart, therefore, still maintains blood flow.
The fast and disorganized contraction of chambers is called fibrillation. Twitching chambers lose their ability to propel blood. Without treatment ventricular fibrillation (VF) results in loss of consciousness followed by death within seconds. Fibrillation of the upper chambers is unpleasent but not fatal as the contraction of the atria is not vital.

For this thesis it is important to understand that tachycadias can have two different origins.

Tachycardias arising from the ventricles are called ventricular tachycadias (VT). They can be caused by an abnormal conduction of electrical signals across the muscle tissue. Usually they occur in individuals that previously suffered from a heart disease (e.g. coronary heart disease, cardiomyopathy or heart attack). VTs are considered an emergency because they can lead to VF and cardiac arrest.

If the cause for a tachycadia is located in the upper part of the heart it is called a supraventricular tachycardia (SVT). SVTs can have non-pathological and pathological causes. An SVT induced by the sinus node (Sinus tachycardia) can be an appropriate response to physical activity or mental stress. Pathological causes include atrial fibrillation (AF) and reentrant trachycadias. The latter refers to uncontrolled circulation of electrical signals between atria and ventricles. In both cases (non-pathological and pathological) the increased electrical activity of the atria is transmitted to the lower chambers forcing the heart rate to go up. However, it is highly unlikely that an SVT will lead to VF because morbidly high impulse frequencies are filtered by the AV node (described in 1.3). Pathological SVTs can be unpleasant but are not considered an emergency (once they are clearly identified as an SVT and not confused with a VT).

A visualization of different tachycardia origins can be found in the appendix (figure 5.1).

1.6 Cardiac implants

Arrhythmia can be treated or prevented with medication and surgery. The latter includes implantation of an artificial pacemaker or implantable cardioverter-defibrillator, hereafter referred to simply as implants.

1.6.1 Artificial cardiac pacemaker

Artificial cardiac pacemakers are implantable medical devices used for bradycardia treatment. Conventional pacemaker systems consist of a generator and one to three electrode leads. The generator is placed under the skin of the upper chest and contains a battery and a small processing unit. The leads are connected to the generator and to different locations inside the heart. Leads can be used to measure electric activity (sensing) or to deliver electrical stimuli (pacing). Pacing is used for artificially triggering the contraction of the heart.
Systems with only one lead are called single-chamber pacemakers. They are often used in patients suffering from a sinus node dysfunction. The lead is placed in the right atrium or the right ventricle. When the intrinsic heart rhythm is too slow, irregular or comes to a halt, the pacemaker starts pacing. Furthermore, single-chamber pacemakers are capable of adjusting their pacing frequency to physical or mental activity by utilizing different parameters measured by thermal, mechanical and chemical sensors.
Dual-chamber pacemakers have two leads - one running to the right atrium and the second to the right ventricle. They provide an effective therapy for patients with AV block. By sensing intrinsic signals in the atrium and transmitting them to the ventricle they serve as an artificial AV node.
Biventricular pacemakers use a third lead to resynchronize the ventricles. A detailed description is omitted at this point. (For further information see [5], chapter 'Cardiac resynchronization' p. 405 ff.)

1.6.2 Implantable cardioverter-defibrillator

An implantable cardioverter-defibrillator (ICD) is similar in design to a pacemaker and supports the same set of features. Additionally, ICDs are capable of terminating a ventricular tachycardia or ventricular fibrillation by using either or both of the following techniques:

The first and preferred approach is antitachycarida pacing (ATP). During ATP the ICD paces the heart faster than its intrinsic rate. By the premature depolarization of cells the undesired circus movement of electrical impulses can be disrupted and terminated. ATP therapy is successful in up to 90 % of VTs and is painless for the patient. However, the fast pacing can also lead to an acceleration of the VT and cannot be applied if the tachycardia already caused a fibrillation. [49]

If ATP was not successful or cannot be applied the implant deliveres a high energey shock (30-40 J / 800 V) to the patient's heart. Through this shock a larger amount of heart muscle cells is simultaneously depolarzied leading to a 'reset' of any intrinsic impulses. A capacitor contained in the generator builds up the necessary energy which then is delivered to the muscle tissue by means of shock coils attached to the leads.[47]

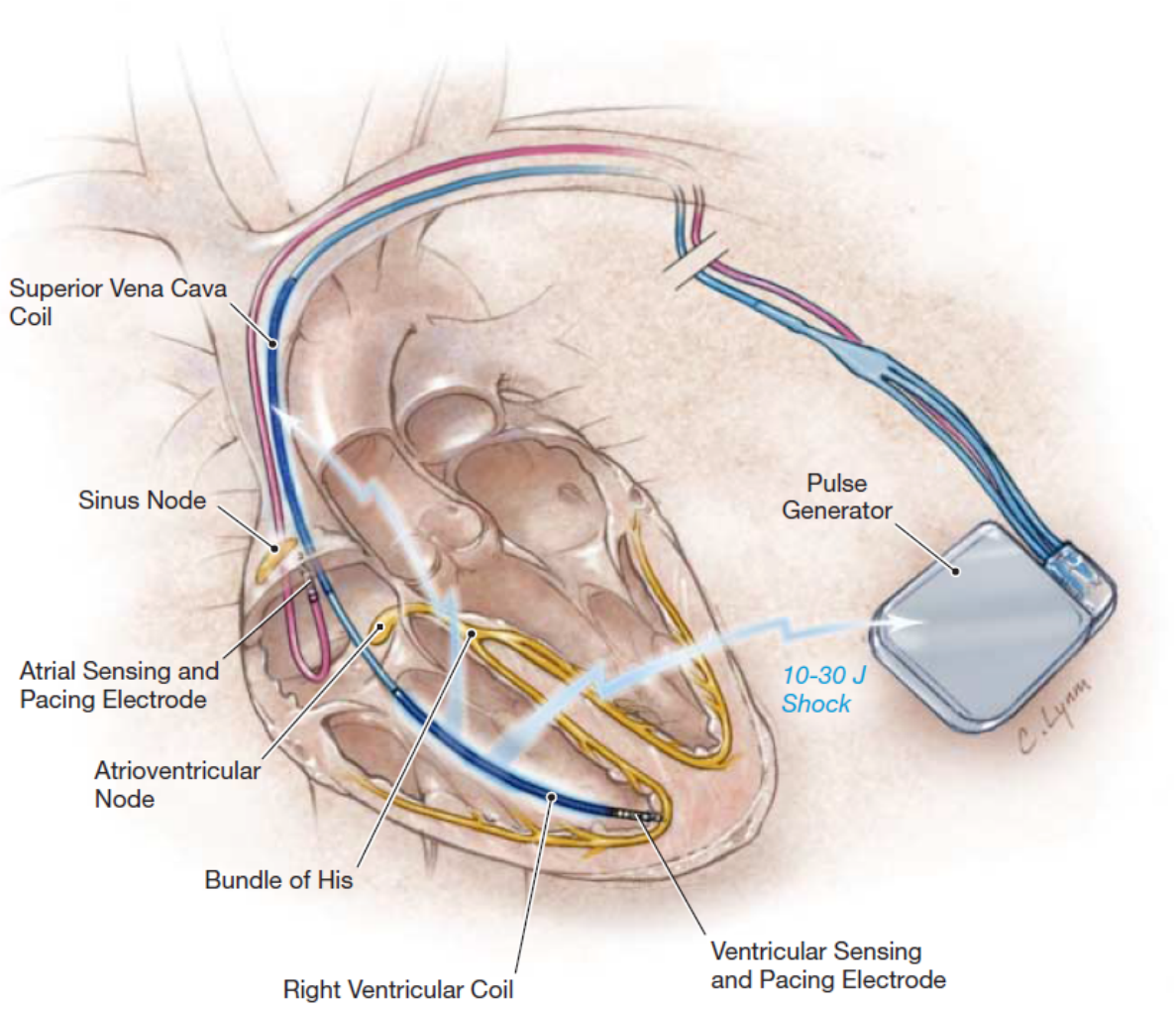


Figure 1.4: Illustration of a dual-chamber ICD system [22]

1.6.3 Intracardiac electography

The sensing ability of implants can be used to record an intracardiac electrocardiogram. The employed principle is the same as for externally measured ECGs (described in 1.4). Instead of using electrodes attached to the patient's skin the implants measure the electrical activity from inside the heart. The set of available measurement vectors depends on the implant's lead configuration.

The approaches of ECG classification implemented in this thesis are focused on dual-chamber ICDs. They usually provide at least the following measurement locations [50]:

- Two local measurements - one in the right atrium (RA channel), one in the right ventricle (RV channel). For the local measurement the voltage between electrode ring and electrode tip of the corresponding lead is measured (see figure 1.6).
- One far-field measurement (FF channel). This channel refers to the voltage measured between the ventricular shock coil and the generator can.

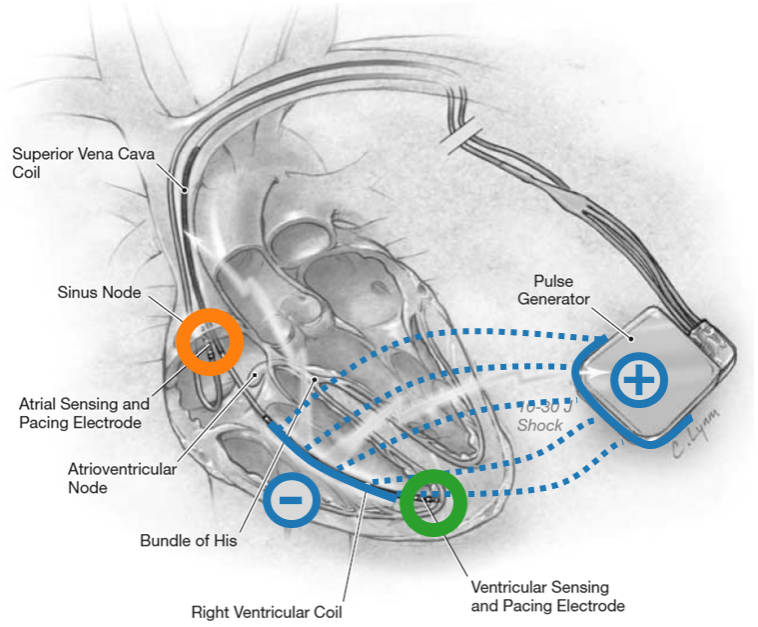


Figure 1.5: Illustrtation of different ECG channels provided by a dual-chamber ICD: RA channel (orange), RV channel (green) and FF channel (blue). [22]

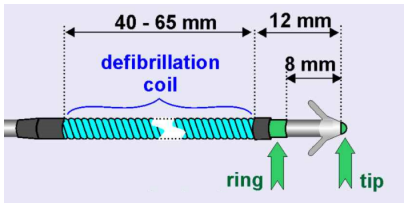


Figure 1.6: Schematic illustration of the distal end of an ICD lead [47]

Figure 1.7: Picture of the distal end of a lead [howtopace.com/basics-of-pacing-leads]

Hereafter the abbreviation ECG refers to an intracadiac electrogram. An example of an ECG recorded by a dual-chamber ICD is shown in chapter 2.2.2.

1.6.4 Remote patient monitoring

Patients with cardiac implants have to be followed-up on a regular basis to ensure a successful therapy. With the aim of complementing and supporting the follow-up care, modern pacemakers and ICDs can be used to remotely monitor the patients heart and the implant's function itself. [20]

Although the exact implementation of remote patient monitoring may vary depending on the manufacturer, all approaches share the same principle. Within a close radius ($\sim$ 3 m) the implant communicates wirelessly with a transmitter. The transmitter is a small device given to the patient which is capable of receiving data from the implant and uploading it to the manufacturer. The manufacturer provides the data to the attending physician using a secure website.

Figure 1.8: Illustration of the workflow of remote patient monitoring. (1) Patient with implant; (2) Transmitter; (3) Internet / manufacturer's server; (4) Device used by the physician to access the manufacturer's website; (5) Physician reviewing the available data. [10]

A transmission automatically follows a predefined schedule (e.g. every 24 hours) or a major cardiac or technical event (e.g. an arrhythmia or a device failure). Some systems also allow for a manually transmission triggered by the patient.
The content of a transmission (hereafter referred to as message) can vary. Periodic messages (triggered by schedule) often only contain a limited set of parameters used to sum up the status of the patient and the implant. Messages following a cardiac event or triggered by the patient also contain an ECG. The ECG can be useful to the physician for further diagnostics. [24]

Chapter 2

Developing the Infrastructure

2.1 Objective

Techniques for automatic evaluation of medical data evolved rapidly within the last two decades. Using these latest advances is of particular interest for manufactures of cardiac implants. New algorithms could support the classification and prediction of different arrhythmias or the recognition of technical difficulties and failures at an early stage.

As mentioned in 1.6.4 the messages recorded by the implants are directly sent to the manufactures. Nevertheless, the presence of the data does not necessarily imply its easy accessibility. This becomes evident when looking at the technical infrastructure used by the manufacturer which supports this thesis. This manufacture's infrastructure proved to be reliable and convenient for the purpose it was primarily designed for. However, it was not refined for being flexible and simple to use. Conduct research on the stored messages is not easily possible. Changes made to the infrastructure are critical as they can affect the follow-up care of thousands of patients. Redesigning it in order to meet new - and noncrucial - requirements is not an option.

Within this chapter a solution for the limitation mentioned above is documented. Instead of modifying the existing infrastructure it was decided to build a second one. This new infrastructure's requirements and design are focused on research.

Note: In the interest of the manufacturer some figures and explanations in this chapter were intentionally formulated vaguely or have been omitted. However, this is unlikely to impede the understanding of the thesis.

2.2 Existing Infrastructure

Before discussing the new infrastructure a brief insight into the existing one is provided.

2.2.1 Accessing messages

For transmission and storage the messages are serialized and compressed using a proprietary data format. The data format opts for compression, reliability and compatibility across multiple generations of implants. The compression technique is optimized for ECGs and provides a space saving way to transmit and persistently store messages. The messages are stored as a binary large object (BLOB) in a relational database. Within the database messages are labeled and addressed with a unique message ID.

A service (only accessible through the intranet of the manufacturer) can be used to decode a message and access its content. The service used to access the data is called diagnostic data interpreter (DDI) and is written in Java. The DDI gets the content of a BLOB and a model identifier (referring to the product family). Given this two information the DDI outputs java-objects containing the diagnostic data.

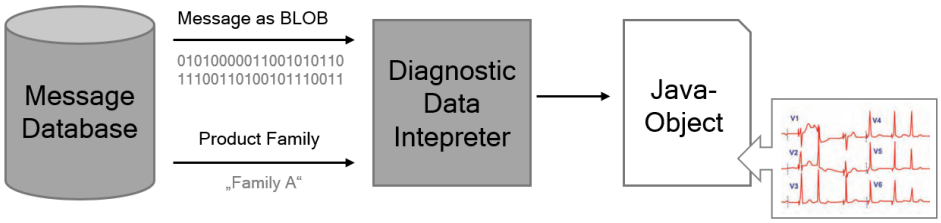


Figure 2.1: Visualization of the API provided by the existing infrastructure

2.2.2 Message content

Each message can contain several episodes. The episodes are numbered and can succeed each other closely in time. Bundling the episodes to a message is done for technical reasons. From a physiological point of view the episodes can be evaluated independently from each other.

The DDI outputs an object for every episode contained in the message. Each episode contains the following information:

- Parameters to identify the used implant, the model of the implant and its configuration
- The reason the episode was transmitted (e.g. triggered by schedule, by patient or by cardiac event)
- Several counters summing up cardiac events and performed therapies
- Multiple timestamps (e.g. referring to the point of recording and transmission)
- And, optionally, an ECG

The transmitted ECGs have the following attributes:

- They contain a varying number of channels (described in 1.6.3). Most ICDs have at least three of them.
- A good sampling frequency for ECGs is considered to be 250 Hz to 500 Hz [14] with a resolution in a range between 8 and 24 bit [16]. The ECGs recorded by the implants are in the same magnitude.
- The length of an ECG is variable. The most recordings have a length between 20 and 40 seconds.
- Using an additional marker channel the implant annotates the ECGs. The markers can be useful to understand the device's functionality. [28]

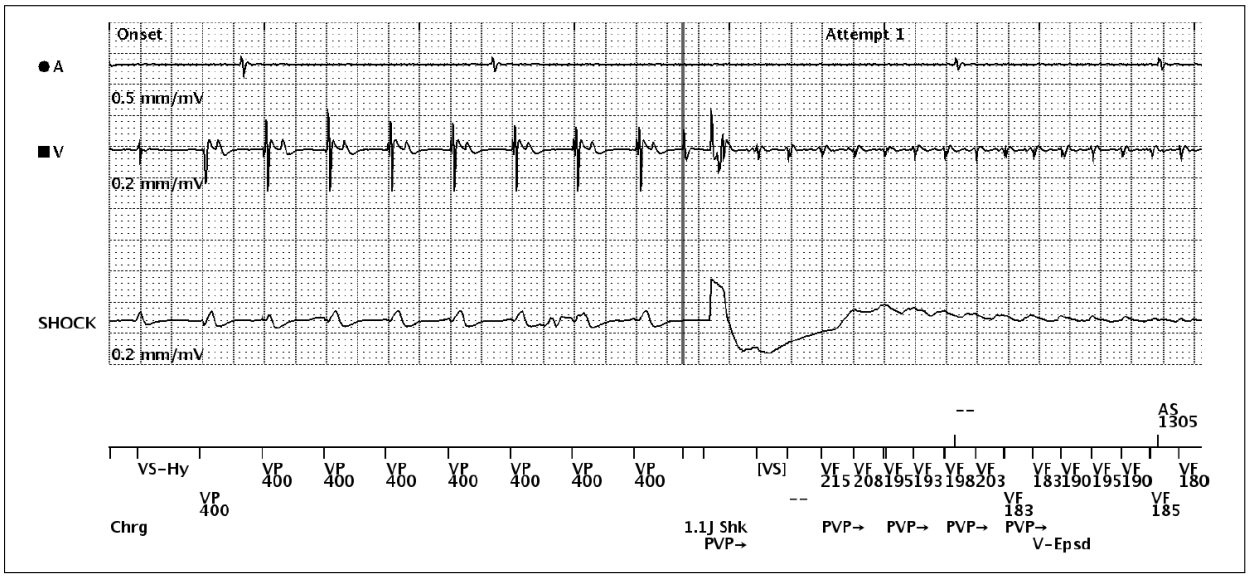


Figure 2.2: Example of an ECG recorded by a dual-chamber ICD. The three measurement vectors are labeled A (left atrium), V (right ventricle) and SHOCK (far-field). This example shows a fibrillation induced by delivering a shock during the ventricles' repolarization. This is done under supervision of a physician to test the implant's function.
Before the fibrillation is induced the implant paces the heart on a high rate (150 bpm). The individual paces are labeled 'VP 400' (ventricular pacing - 400 ms) on the marker channel. After shocking the implant detects an abnormally high frequency of impulses. These impulses are labeled 'VF' (ventricular fibrillation) and occur every 180 to 215 ms. The marker channel also reveals when the capacitor's charging starts ('Chrg') and when the shock is delivered ('1.1J Shk'). [28]

2.3 Requirements

The new infrastructure should enable the test of new algorithms and is focused on providing easy and quick access to the ECGs contained in the messages. Live monitoring of patients is currently not planned. Once a new algorithm is proven to be useful the manufacturer could implement it elsewhere in the existing telemonitoring infrastructure.

To meet its intended purpose the new infrastructure should meet the following requirements.

- **Support for multiple programming languages**
  Due to the DDI's design it is not possible to access the diagnostic data with any other programming language than Java. Although Java is a powerful programming language with a large community, it is comparatively uncommon to be used in the field of artificial intelligence - especially in the context of data science.
  The new infrastructure should at least provide support for the languages Python, R and C. Python offers support for several machine learning libraries such as TensorFlow, PyTorch and ScikitLearn. R is a powerful toolkit for statistical evaluations and is already used by the manufacturer. C can be used for low-level programming on embedded systems (e.g. artificial pacemakers and ICDs).

- **Access time**
  Another important aspect is the time needed to access the data. Due to the fact the current format was never intended to be used for a quick and repeated access the decoding is comparatively slow. Accessing one million messages can take several days - which is insufficient for fast and agile testing of new algorithms.
  To read one million messages in less than one hour is considered to be a desirable result for the improvement.

- **Local processing of data**
  All messages are exclusively stored in a database and can only be decoded using the DDI. This results in two disadvantages: The data can only be obtained and processed within the manufacturer's intranet. The transfer of data to external researchers is not possible. Furthermore, the network and the access times to the databases and the DDI can become a bottleneck.
  For fast, agile tests it can be useful to store subsets of the messages on a local machine and be able to access this messages without being dependent on any network resources.

- **Anonymization**
  For the manufacturer it is very important that individual messages can no longer be linked to a patient. Currently the messages contain information which could disclose the patient's identity.
  Most of these information can simply be removed as they are not important for any statistical evaluation - except for the implant's serial number. The serial number is crucial because it is the only way of grouping messages sent from the same implant. This is important for the realization of time series analysis or for avoiding biases in training and test data sets.

- **Scalability**
  As telemedicine continues to be established the manufacturer holds hundreds of millions of messages. Many of them contain an ECG.
  The new infrastructure should be able to process and store higher amounts of data.

2.4 Concept

The new infrastructure will be based on converting the data contained in the messages to another file format. Once the data is converted it is persistently stored in a cache to enhance the access time.

Following this approach the infrastructure is separated into three major parts:

- The **Episode Cache** which stores the data once it has been converted. As the name suggests the cache will hold a file for every individual episode. For serialization two different libraries are used: Protocol Buffers and Flatbuffers. Both techniques are combined with Gzip. Finally, the files can be stored to a database or simply to the filesystem of a local machine.
- The **Message Converter** which is used to initially and continuously decode and cache data. It is implemented in Java and connected to the existing infrastructure. Particular effort has been made to ensure that large amounts of data can be processed. Many parts of the processing were parallelized. To easily configure and control the program a GUI was designed.
- An **API for Python** demonstrating how to access and work with the cache. Python was chosen as it is the programming language used in the second part of this thesis.

The concept and the implementation of each item is be addressed in the following chapters.

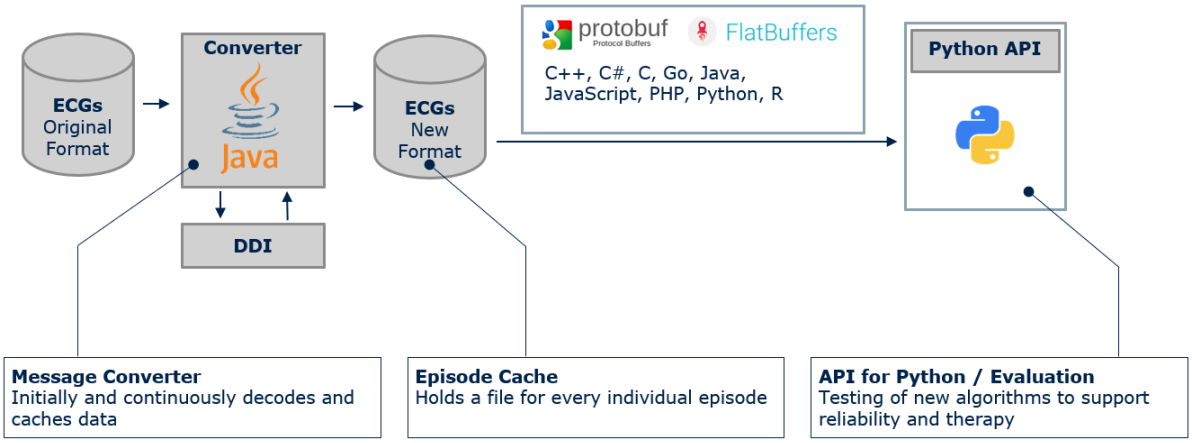


Figure 2.3: Visualization of the infrastructure's concept

2.5 Episode Cache

2.5.1 Considerations

Before deciding on the cache's design and implementation the following considerations had to be taken into account:

- **Addressing level**
  As mentioned in 2.2.2 messages contain multiple episodes which can be evaluated independently from each other. The ECGs contained in episodes are likely to be used as a whole or in larger parts. Depending on the subject of research different sets of episodes will be put together. For creating these sets some parameters contained in the episodes (e.g. the implant's model and configuration) are likely to be necessary.
  Instead of grouping episodes to messages the cache should provide access to individual episodes. A more precise addressing of data is not needed except for certain parameters used to filter the episodes.
- **Characterization of stored data**
  Episodes contain several hundreds of diagnostic and technical parameters. For the currently planed application approximately only 40 are considered important. These parameters are represented as string, integer (32/64 bit), floating-point value (32/64 bit) or boolean. Some of them are optional. Episode carrying an ECG will contain additionally 10,000 to 15,000 floating-point values (32 bit) and 100 to 200 markers (each consisting of a string and one to three numerical values).[1]

Furthermore, the following considerations are derived from the requirements listed in 2.3:

- **Support for multiple programming languages**
  The cache should be accessible with Python, R, C and Java. The latter is important as the Message Converter will be written in Java.
- **Access time**
  Compared to the currently employed format the cache should feature a significantly faster access to the data.
- **Local processing of data**
  Utilization of network resources should be avoided or kept optional.

2.5.2 Design

Taking these considerations into account it was possible to decide on the cache's design.

The cache will be based on repacking individual episodes into another file-based format. Initially the generated files will be stored to a relational database. Subsequently, subsets of the data can be copied to local machines.

Using suited databases (e.g. NoSQL, OLAP) it would have been possible to store the information contained in the messages without explicitly repacking it. However, the infrastructure is focused on processing data on local machines. Potential bottlenecks and locational restrictions arising form a network should be avoided. Migrating these databases to local machines would be possible but also rather impractical and increase the machines workload. Choosing a file-based approach allows for an easy transfer of data.
By generating a file for every single episode they stay individually accessible. For addressing the files the original message ID will be combined with the episode's number (mentioned in 2.2.2).
Initially the Message Converter will store the generated files to a relational database. Parameters that are required for filtering the episodes will be stored twice: They are contained in the episode's file and will additionally be stored directly to the database. The redundant storage of these parameters is not optimal yet justifiable given the small number of needed parameters.
For further processing specific episodes can be copied to a local machine.

[1] Most implants record two or three channels. All channels have the same length and are recorded simultaneously. Depending on the sampling rate and the length (both mentioned in 2.2.2) every channel carries approximately 5,000 data points on average. Each data point equates to a 32 bit floating-point number referring to the measured voltage. Due to the fixed time interval between measuring points the temporal information can be calculated using a timestamp and the sampling rate.
Furthermore, the ECG comes with an additional channel for annotations (marker-channel) carrying between 100 to 200 entries. Each entry consists of a timestamp, a short description (given as string) and up to two additional numerical values.

2.5.3 Data format

The potential of the proposed design significantly relies on the employed serialization format. To meet the requirements the format should provide fast access and support multiple programming languages. The data which has to be stored is variable, meaning the content of individual episodes could vary. Furthermore, it is quite likely that new parameters will become requisite in the future.
To implement a customized file format which is fast, provides flexibility, enables evolution and supports different programming languages is costly. Therefore, it was decided to use an existing format.

Initially the following formats have been considered: Apache Avro, Apache Thrift, BSON, Cap'n Proto, FlatBuffers, MessagePack, and Protocol Buffers. This selection was based on different papers comparing different serialization techniques [37, 42, 53] and online sources [36, 23, 43]. All listed formats support binary encoding. Text-based formats (e.g. JSON, XML) were excluded as human-readability is not needed and binary-based formats are likely to provide a faster deserialization [37].
The considered formats are listed in table 2.1 and will be compared using different properties.

Table 2.1: Comparison of different serialization standards/libraries

Name	Maintainer	Language Support				Number of Implementations	IDL based	Zero copy
		C	Java	Python	R			
Apache Avro	ASF	✓	✓	✓	-	8	no	n/a
Apache Thrift	ASF	✓	✓	✓	-	27	yes	n/a
BSON	MongoDB	(✓)	(✓)	(✓)	(✓)	50	no	n/a
Cap'n Proto	Kenton Varda	✓	(✓)	(✓)	-	17	yes	yes
FlatBuffers	Google	✓	✓	✓	-	12	yes	yes
MessagePack	Sadayuki Furuhashi	✓	✓	✓	(✓)	100+	no	yes
Protocol Buffers	Google	✓	✓	✓	(✓)	100+	yes	n/a

✓ supported by maintainer; (✓) supported by third-party implementation; - not supported.
References: Apache Avro: [48], Apache Thrift: [4], BSON: [13] Cap'n Proto: [54], FlatBuffers: [21], MessagePack: [38], Protocol Buffers: [52].

Interface Definition Language
Formats based on an Interface Definition Language (IDL) use a language-independent interface to specify the stored or transmitted data. Once an interface is defined it can be used to automatically generate code for one or multiple of the supported programming languages. This is done with a format specific compiler. The generated code enables the programmer to read and write data based on the given format and interface.
IDL-based formats are particularly useful in supporting multiple programming languages as a large part of the required code is generated by the compiler. However, they are less flexible when changes become necessary. Each time the interface changes new code has to be compiled.
Both support for multiple programming languages and evolution are required for this application. However, it is unlikely that changes will happen on a regular basis. Therefore, IDL-based languages are favorable.

Number of Implementations
The Number of Implementations was determined by counting the official and third-party implementations listed on the maintainers' websites. It was used to estimate the size of the community that maintained and utilized a format. The individual implementations did not have to meet any special requirements (many of them offer support for an already supported language or only implement a subset of the format's features).
Assuming that formats with a higher number of implementations are used and supported by a larger community and, assuming further, that a larger community results in higher numbers of examples, tutorials and discussions it follows that formats with a higher number of implementations are preferable.

Access time
Predicting the access time of the different formats is hard. It has to be assumed that a format's performance is not only linked to the format itself but also to the implementation in use and the stored data. Finding suitable benchmarks was therefore not possible.
All listed formats advertise to be fast. However, three of the formats - Cap'n Proto, FlatBuffers and MessagePack - stand out as they use a technique referred to as in-memory representation or zero-copy. These formats avoid

to allocate additional memory when handling the data. This technique goes without extra steps for encoding or decoding. Once the data is loaded to a system's memory it can directly be accessed.

Compression
As ECGs often contain repeating patterns and have zero lines in their waveform it is likely that they compress well. The majority of the listed formats do not provide any kind of compression. Therefore, the files generated by the selected format will separately be compressed using Gzip.
Gzip was chosen as it offers support for many programming languages (including C, Python, Java and R). Furthermore, it is known to offer a quick decompression of floating point values [45]. In comparison to the serialization format the decompression performance is assumed not to differ significantly with respect to the target programming language. The required interface for compression and decompression is small allowing for an easier optimization and for calls to external libraries. Phyton's zlib-library for example decompresses gzip by calling code written in C [62].

Conclusion
Only three of the seven formats offer libraries for R: BSON, MessagePack and Protocol Buffers. All three formats are likely to be well established as they have a high number of implementations. Protocol Buffers is IDL-based which simplifies the support of multiple programming languages. MessagePack uses zero-copy and is presumably faster. BSON is excluded as it does not use an IDL or zero-copy. The decision between the remaining two formats was based on an advantage of Protocol Buffers which was not yet mentioned: Protocol Buffer interfaces can also be processed by the FlatBuffers's IDL compiler. This makes it easy to develop and maintain both formats simultaneously.
Therefore, it was decided to use Protocol Buffers and additionally add support for the zero-copy format FlatBuffers. Whether it is justifiable to store the data twice is discussed in 2.5.5.

All involved licenses, BSD (Protocol Buffers), Apache 2 (FlatBuffers) and GNU GPL (Gzip), allow the free use of the corresponding software.

2.5.4 Implementation

2.5.4.1 Protocol Buffers and FlatBuffers - IDL and encoding

Both formats, Protocol Buffers and FlatBuffers, are maintained by Google, use a similar IDL[2] and support evolution. The schema created for the episodes consists of approximately 100 lines of code. It utilizes nested interfaces to reproduce the object-based structure orginially used by the DDI to provide the data.

The following example explains the encoding and the IDL of the two formats. The same example is used in 2.6.2.3 to demonstrate the serialization in Java and in 2.7.2.3 to show the deserialization using Python.

Example
The example 'ECGMessage' carries the patient's name (string), the patient's age (integer, 32 bit), the ECG's sampling rate (integer, 32 bit) and an array of measured ECG values (floating-point numbers, 32 bit). The Protocol Buffers schema can be seen in listing 2.1. The same schema is used for the FlatBuffers compiler.

```
message ECGMessage {
  //(modifier) type name = ID;
  string patient_name = 1;
  int32 patient_age = 2;
  int32 sampling_rate = 3;
  repeated float ecg_data = 4;
}
```

Listing 2.1: ECG message schema

[2] As already mentioned FlatBuffers can also process a Protocol Buffers schema. However, FlatBuffers also have its own IDL which is similar to that of Protocol Buffers.

For discussing the encoding an example containing the following values was created:
- Patient's name: **John_Doe**
- Patient's age:
- Sampling rate: **1337**
- ECG data: **[22.22, 22.22]**
The patient's age is omitted on purpose.

Protocol Buffers
Using Protocol Buffers the example message generates the following byte-array as output:

```
10, 8, 74, 111, 104, 110, 95, 68, 111, 101, 24, 185, 10, 34, 8, 143, 194,
177, 65, 143, 194, 177, 65
```

Reverse engineering the message reveals the following observations:

- Interpretation of the message must partly be done on bit-level.
- Each parameter is labeled with its ID (see 2.1) and its type. For labeling a single byte is used. The first 5 bits represent the id and the last 3 bits refer to the type.[3]
- Numerical values are not represented by a fixed number of bytes. Instead, each byte used to store a part of the value uses its first bit as a flag to indicate whether or not the succeeding byte is also part of the number.
- Omitted parameters do not leave any footprint.

```
10,  ->[00001|010]    // int5|int3: id 1 (patients name) & type 2 (list)
8,   ->[0|0001000]    // bool|int7: flag & length of list (8 bytes)
74, 111, 104, 110,    // 4x char: J o h n
95, 68, 111, 101,     // 4x char: _ D o e

24,  ->[00011|000]    // int5|int3: id 3 (sampling rate) & type 0 (int)
185, ->[1|0111001]    // bool|int7: flag & 2nd part of sampling rate
10,  ->[0|0001010]    // bool|int7: flag & 1st part of sampling rate
                      // 0001010 0111001 --> 1337

34,  ->[00100|010]    // int5|int3: id 4 (ecg data) & type 2 (list)
8,   ->[0|0001000]    // bool|int7: flag & length of list (8 bytes)
143, 194, 177, 65,    // float32: 22.22
143, 194, 177, 65,    // float32: 22.22
```

Listing 2.2: Interpretation of the example message serialized by Protocol Buffers

Avoiding overhead Protocol Buffers stores the data in a space saving way. Individual parameters have a variable length (or can be omitted) and are only labeled prior their appearance. Therefore, the position of parameters cannot be determined without completely deserializing the message making a zero-copy usage impossible.

FlatBuffers
Using FlatBuffers the example message generates the following byte-array as output:

```
16, 0, 0, 0, 12, 0, 16, 0, 12, 0, 0, 0, 8, 0, 4, 0, 12, 0, 0, 0, 12, 0,
0, 0, 0, 5, 0, 0, 16, 0, 0, 0, 2, 0, 0, 0, 143, 194, 177, 65, 143, 194,
177, 65, 8, 0, 0, 0, 74, 111, 104, 110, 95, 68, 111, 101
```

By examining FlatBuffers's source code and reverse engineering the message the following observations were made:

- FlatBuffers uses little endian - making the first byte the least significant one.

[3] As 5 bits would result in a restriction to 32 IDs it can be assumed that the third observation (numerical values are not represented by a fixed number of bytes) also applies to the IDs.

- The first 4 byte (int32) represent the position of the root interface. This is important as a message can consist of multiple subinterfaces.
- Each interface consists of three parts which have been named metadata-table, data-table and redirection-table.[4]
- The metadata-table is used to store the sizes of the metadata-table and the data-table. Furthermore, it stores the position of the individual parameters within the data-table.
- The position of omitted parameters is set to 0 in the metadata-table.
- The data-table stores the values of parameters with a fixed length. For parameters with a variable length a redirection to the redirection-table is made.

```
16, 0, 0, 0,    // int32: position of (root) data-table (16 bytes ahead)
*** begin metadata-table ***
12, 0,          // int16: size of the metadata-table (12 byte)
20, 0,          // int16: size of the data-table (20 byte)
12, 0,          // int16: position of the patients name in the data-table
0,  0,          // int16: position of the patients age (discarded)
8,  0,          // int16: position of the sampling rate in the data-table
4,  0,          // int16: position of the ecg data in the data-table
*** end metadata-table ***

*** start data-table ***
12, 0, 0, 0,    // int32: position of the metadata-table (12 bytes back)
12, 0, 0, 0,    // int32: redirection to the ecg data (12 bytes ahead)
57, 5, 0, 0,    // int32: sampling rate: 1337 (57*1 + 5*256)
16, 0, 0, 0,    // int32: redirection to patients name (16 bytes ahead)
*** end data-table ***

*** start redirection-table ***
2, 0, 0, 0,         // int32: length of ecg data array (2)
143, 194, 177, 65,  // float32: 22.22
143, 194, 177, 65,  // float32: 22.22
8, 0, 0, 0,         // int32: length of string (8)
74, 111, 104, 110,  // 4x char: J o h n
95, 68, 111, 101,   // 4x char: _ D o e
*** end redirection-table ***
```

Listing 2.3: Interpretation of the example message serialized by FlatBuffers

FlatBuffers initially lists the position of all parameters and does not shorten the representation of numerical values. This results in a big overhead but allows individual parameters and their values to be accessed without previously decoding the whole message.

2.5.4.2 Database

Once they have been converted the episodes are stored to a table in a relational database. The table's structure can be seen in figure 2.4.
The MESSAGE_ID and EPISODE_NUMBER are used as a composite primary key for individual episodes. The converted files are stored as a BLOB. The information required to filter the data is stored alongside the episodes.

[4]This may only apply to the current example. More complex schemata using a bigger variety of data types may be more complex.

[redacted]._[redacted]_CACHE_COMPLETE_V6

* MSG_ID	VARCHAR2 [redacted]
* EPISODE_NUMBER	NUMBER
DECODE_INFO	VARCHAR2 (20 BYTE)
EPISODE_TYPE	VARCHAR2 (40 BYTE)
HAS_IEGM	VARCHAR2 (1 BYTE)
EPISODE_BEGIN_TIME	DATE
[redacted]	VARCHAR2 [redacted]
[redacted]	VARCHAR2 [redacted]
[redacted]	NUMBER
PROTOBUF_PAYLOAD	BLOB
PROTOBUF_SIZE	NUMBER
FLATBUFFERS_PAYLOAD	BLOB
FLATBUFFERS_SIZE	NUMBER

Figure 2.4: Structure of the table used to store the episodes

2.5.5 Storage consumption

1,000 messages containing ECGs were randomly selected and converted. Subsequently, the size of all generated files was averaged to estimate the size of converted messages. This has been done separately for Protocol Buffers and FlatBuffers before and after compression.

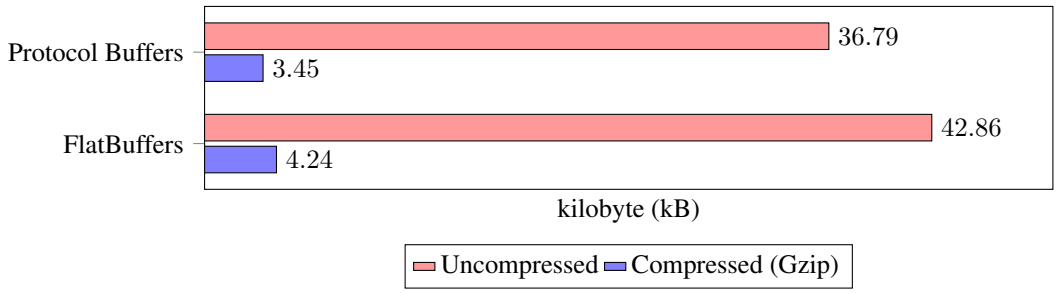


Figure 2.5: Average size required for one message in dependence on the format (mixed physiological content)

As expected (considering the observations made in 2.5.4.1) Protocol Buffers is the more space efficient protocol. On average FlatBuffers uses 16.5 % more space. Using Gzip both file formats can be compressed to under 10 % of their original size.
Storing both formats approximately requires 7.7 kB per message. Therefore, less than 10 GB are sufficient for storing one million messages. Considering the current and also upcoming amount of messages containing an ECG and the capacity provided by the machine hosting the database it is justifiable to store the information twice.

Out of interest a second test was performed. This test compares the size of messages carrying a non-pathological ECG to those carrying an arrhythmia. The expectation is that the latter take up more space as they are more likely to carry multiple episodes. Furthermore, healthy ECGs traces should compress better as the zero segments of the recordings should be longer.
For testing 1,000 messages containing healthy rhythms and 1,000 messages containing tachycardia or fibrillation where randomly selected. The test is exclusively performed with the Protocol Buffers.

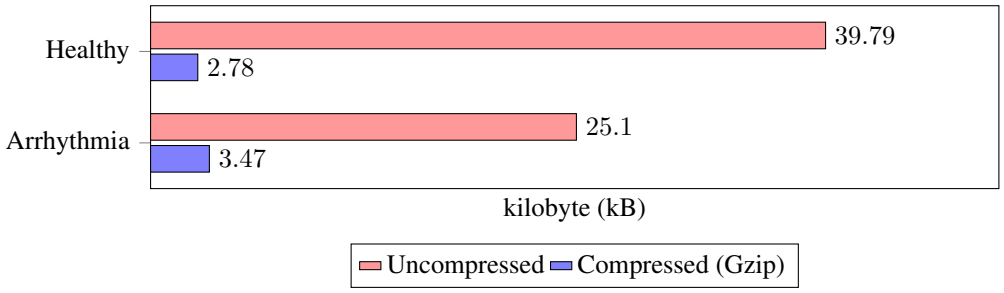


Figure 2.6: Average size required for one message in dependence on its content (based on Protocol Buffers)

The assumptions regarding the compression proved to be true. While non-pathological data can be compressed to 7.0 % of its original size, pathological rhythms can only be compressed to 13.8 %. However, without compression the healthy ECGs take up more space. Assuming this ist caused by longer recordings it stands to reason that the transmitter is already taking the compression into account and adjusts or cuts the length of the transmitted ECGs.

2.6 Message Converter

2.6.1 Considerations

Before implementing the Message Converter the following considerations have been made:

- **Usage of the existing infrastructure**
  The Converter has to use the API of the DDI for processing the messages. Furthermore, it should directly be connected to the database which is used to initially store the received messages.
- **Maintainability**
  The new infrastructure should be used for a longer period of time. Therefore, maintenance of the Converter must be easy. The implementation should enable the change and exchange of individual functions and parts of the software.
- **Configurability and usability**
  The converter should easily be configurable. Settings regarding the database, DDI, anonymization or processing itself should not be hardcoded.
- **Continuous converting**
  The manufacturer receives messages on a daily basis. The Converter should provide a mechanism to automatically add new messages to the cache.

Furthermore, the following considerations are derived from the requirements listed in 2.3:

- **Anonymization**
  The Converter should allow to discard or obfuscate information which could be used to identify the patient.
- **Scalability**
  Initially the Converter will have to convert several million messages. Therefore, the program should be able to process big amounts of data.

2.6.2 Implementation

2.6.2.1 Tools and dependencies

The Message Converter is implemented in Java. This is necessary as the DDI is written in Java and returns decoded messages as objects.
The employed Java version is Java 1.8.0 update 201. The Eclipse IDE (Photon Release 4.8.0) was used for development. The GUI was designed using SceneBuilder version 8.5.0. For connecting to the used databases the Oracle Database Driver version 11.2.0.4.0 is used. Serialization is done with Google Protocol Buffers version 3.6.1 and FlatBuffers version 1.10.0.

2.6.2.2 Processing steps

The Converter's main purpose is to transfer messages from the existing infrastructure to the new Episode Cache. This process is completely automated. Instead of providing an detailed description of the Converter's implementation it was decided to give a brief overview of its functionality and discuss noteworthy features in chapter 2.6.3 without focusing on their technical implementation.

The Converter's functionality can be described as follows:

Initialization
Using the GUI (described in 2.6.3.2) or the CLI (Command Line Interface, described in 2.6.3.4) the program is started. During initialization a configuration file is loaded (described in 2.6.3.3). The file is used to initiate an Options-object containing relevant settings. Thereafter, the Manager is started.

Processing
The processing is supervised by the Manager and starts with the Feeder. The Feeder continuously loads message IDs form a database or a local file and uses them to create Jobs. Jobs contain an adjustable number of IDs and are processed by DecoderThreads. DecoderThreads are connected to the existing infrastructure. Using the IDs they read the transmitted messages from a database and use the DDI to access the contained diagnostic data. Once the data is loaded it is serialized with Protcol Buffers and/or FlatBuffers and compressed with Gzip. Subsequently, they are copied back to the Job. Once a Job is finished the DecoderThread passes it to the Deliverer and fetches a new Job. The Deliver writes the converted files back to a database and/or the local machine.

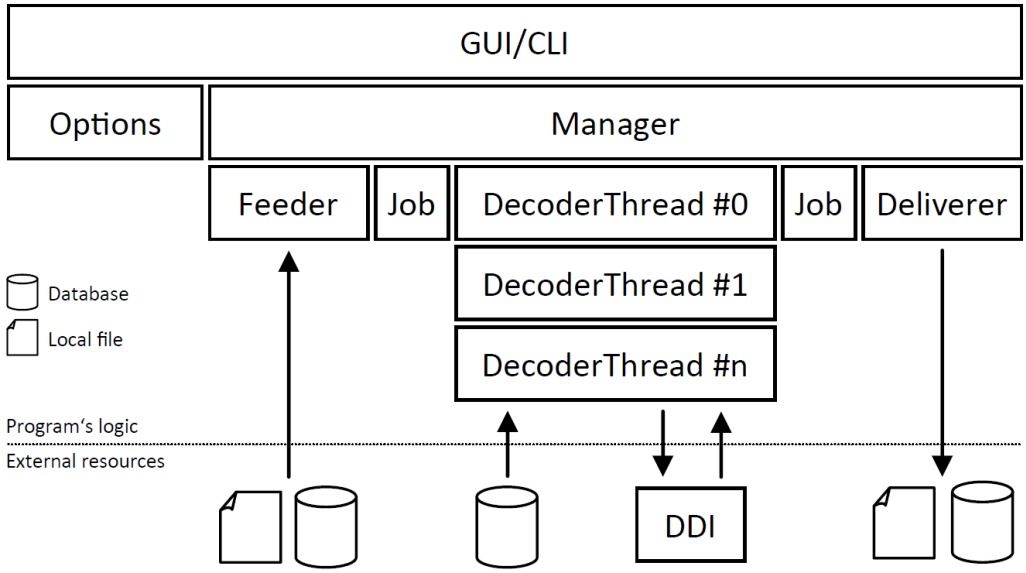


Figure 2.7: Simplified illustration of the Converter's structure

2.6.2.3 Protocol Buffers and FlatBuffers - Serialization

The ECGMesssage described in 2.5.4.1 is used for demonstrating the serialization process.

Protocol Buffers
Listing 2.4 shows how an ECGMessage is created based on the code generated by the Protcol Buffers compiler. The format provides high abstraction and is easy to use as the individual parameters can simply be passed via set-methods.

```
public static byte[] generateProto(String patientName, int patientAge,
        int samplingRate, float[] ecgData) {

    ECGMessage.Builder builder = ECGMessage.newBuilder();

    builder.setPatientName(patientName);
    builder.setPatientAge(patientAge);
    builder.setSamplingRate(samplingRate);
    builder.addAllEcgData(arrayToList(ecgData));

    return builder.build().toByteArray();
}
```

Listing 2.4: Method used to generate an ECGMessage based on Protocol Buffers

FlatBuffers
Listing 2.5 shows how an ECGMessage based on FlatBuffers is created. First, all parameters with a variable length are passed to the builder (patient's name and ECG data). The builder serializes these parameters and returns an offset for each of them. Next, the ECGMessage is build. Parameters with a fixed length (patient's age and sampling rate) are passed directly to the builder. For the remaining parameters the previously calculated offset is passed. The nested creation of segments with a variable length is not allowed. Therefore, the creation of the ECGMessage cannot be started before serializing the patient's name and the ECG data. This is due to the technique used to store the data and can be challenging when handling more complex schemata.

```
public static byte[] generateFlat(String patientName, int patientAge,
        int samplingRate, float[] ecgData) {

    ECGMessage.Builder builder = ECGMessage.newBuilder();

    //Step 1: serialize content of ECGMessage which has a variable length
    int patientNameOffset = builder.createString(patientName);
    int ecgDataOffset = ECGMessage.createEcgDataVector(builder, ecgData);

    //Step 2: create ECGMessage
    ECGMessage.startECGMessage(builder);
    ECGMessage.addPatientName(builder, patientNameOffset);
    ECGMessage.addPatientAge(builder, patientAge);
    ECGMessage.addSamplingRate(builder, recordingDate);
    ECGMessage.addEcgData(builder, ecgDataOffset);

    int ecgMessageOffset = ECGMessage.endECGMessage(builder);
    builder.finish(ecgMessageOffset);

    return byteBufferToByteArray(builder.dataBuffer());
}
```

Listing 2.5: Method used to generate an ECGMessage based on FlatBuffers

2.6.2.4 Package structure

Without including the code generated by the Protocol Buffers and FlatBuffers compiler the Converter is composed of 2,670 lines of code arranged in 13 packages. The classes and interfaces contained in each package are briefly described in the appendix (table 5.1).

All components and functionalities of the Converter which are likely to be modified or replaced in the future are implemented in individual classes or packages to keep the program maintainable. For example the routines for accessing the DDI or the databases can easily be changed or a new mechanism to store the converted episodes can be added.

2.6.3 Features

2.6.3.1 Obfuscation of serial numbers

As mentioned in the requirements (2.3) messages contain information which could impede the patient's privacy. Except for the serial number these information can simply be discarded. The serial number is useful as it allows to group messages coming from the same implant.

The Message Converter offers three options to handle the serial number:

- **Discard** - copying the serial number is omitted
- **Copy** - the serial number is copied to the cache
- **Obfuscate** - using a hash function the serial number is obfuscated before being copied to the cache

Hash functions are deterministic and will generate the same output for a serial number processed multiple times. Only given the hash function's output it is infeasible - or at least costly - to recover the original serial number. However, serial numbers are unlikely to be assigned randomly. They are generated using predictable patterns. Therefore, it is important to combine the serial numbers with a undisclosed salt value before hashing them.

Using the SHA-1 algorithm the following example illustrates how a 16 bit serial number is obfuscated. The actual implementation of the Message Converter is based on the same principle but uses a different hash function and different input values.

```
// 1. Geneate a byte array based on the salt value ('sALTValUE')
[115, 65, 76, 116, 86, 97, 108, 85, 69]
// 2. Extend the array by the size of the original serial number (4 byte)
[115, 65, 76, 116, 86, 97, 108, 85, 69, _ , _ , _ , _ ]

// 3. Copy the serial (1337) to the empty positions
[115, 65, 76, 116, 86, 97, 85, 69, 0, 0, 5, 39]
// 4. Calculate the hash
[194, 115, 125, 38, 136, 142, 160, 64, 180, 208, 217, 254, 116, 249, ... ]
// 5. Copy the first 4 bytes
[194, 115, 125, 38]
// 6. Generate a numerical serial number based on the selected bytes
3264141648

// Step 1 and 2 are only processed once.
// Steps 3 to 6 are repeated for every serial number / message.
```

2.6.3.2 GUI

A graphical user interface (GUI) was implemented to improve the Converter's usability. The GUI was programmed using JavaFx and designed using the corresponding ScenceBuilder tool. In total the GUI consists of 56 elements and offers easy control and configuration of the Converter.
Particularly noteworthy is the possibility to easily save and load different configurations. Furthermore, the GUI is continuously refreshed while the Converter is running showing the progress and also an estimation of the remaining processing time. The latter has proven to be useful and accurate when processing larger amounts of data. Screenshots of the GUI can be seen in the appendix (figure 5.3).

2.6.3.3 Persistently stored settings

Using the GUI in total 40 different parameters can be set. The ability to save these settings is an important aspect of the program's usability as it avoids the Converter's configuration from being discarded each time it is restarted. Saving the settings is done using a human-readable, XML-based file-format. As already mentioned it is possible

to create and switch different configuration files. The converter will always try to load the most recently used settings. If the last configuration cannot be found or its file is corrupted a set of hard-coded default-settings is loaded.

2.6.3.4 Command-line interface

A command-line interface (CLI) was added to the Converter to simplify the successive caching of new messages. By starting the program with the command-line argument '-start' the GUI will be suppressed and the converter will immediately start its execution based on the last known configuration. This process can easily be automated by a task scheduler.
To determine and input the ids of new messages one solution is to use a database view which subtracts ids already contained in the cache from all available message ids. This function was intentionally not implemented by the converter itself as databases are optimized for these kind of operations. A screenshots of the CLI can be found in the appendix (figure 5.2).

2.6.3.5 Concurrency

Parts of the Converter are executed concurrently to optimize its performance. The individual threads are listed below.

JavaFX Application Thread / RefreshUIThread
Depending on the chosen interface mode the JavaFX Application Thread (GUI) or the RefreshUIThread (CLI) is started. Both are responsible for refreshing the user interface and update the displayed progress. The JavaFX Application Thread is also required for keeping the GUI responsive.

ManagerThread
The manager-object organizes the required processing steps and ressources. Its execution extends over the entire runtime of the Converter. To avoid the manager from blocking other parts of the program it is wrapped by a thread - the MangerThread.

FeederThread
The FeederThread is responsible for creating new Jobs. Each Job contains an adjustable number of message IDs which subsequently are processed by the DecoderThread(s). Especially during the Cache's initialization several million messages have to be decoded. Loading all IDs at once could delay other processing steps and cause memory problems. Therefore, the feeder-object maintains a queue of jobs and refills the queue on demand by loading new ids. The queue's size can also be adjusted by the user.

DecoderThread
The DecoderThread can be started multiple times. It is responsible for handling the following steps:

1. Get IDs by accessing the queue maintained by the feeder-object
2. Load the original message files from the source database
3. Communicate with the DDI to decode the messages
4. Generate the Protocol Buffers and FlatBuffers file
5. Compress the files using Gzip
6. Push the converted data to the deliverers

2.6.3.6 Optimized database queries

The Converter has to read and write data from and to databases for processing messages. To prevent these steps from becoming a bottleneck a lot of attention was paid to optimize the used queries. All queries executed repeatedly are implemented as 'Prepared Statement'. By using prepared statements a query is compiled once and can then be reused without being recompiled again. Furthermore, the messages/episodes are not transferred individually. Instead, the queries are designed for reading and writing data in sets. The size of a set is determined by the number of messages contained in each Job.

2.6.4 Test

2.6.4.1 Performance

Two parameters can be varied by the user to optimize the Converter's performance: The number of concurrently running DecoderThreads (thread-count) and the number of messages contained in a Job (job-size).

Increasing the thread-count is likely to have a positive impact on the program's performance as steps executed locally (e.g. generating and compressing files) can benefit from nowadays employed multi-core processors. Furthermore, calling the DDI (which is expected to be a bottleneck) can be done simultaneously by multiple threads. By increasing the job-size the number of rows transferred with each database query is raised. Therefore, the number of required queries is reduced. Reducing the number of individual queries is expected to have a positive impact on the program's performance. Furthermore, a reasonable job-size also prevents the DecoderThreads from frequently fetching new jobs which could result in the threads impairing each other.

Test set-up
A fixed set of 100,000 randomly selected messages containing approximately 23,000 ECGs was repeatedly processed to test these variable's impact on the Converter's performance. During processing the duration of the following steps was measured: Reading transmitted messages from the database (Read), deserializing them using the DDI (Deserialize), serializing them with Protocol Buffers (Protobuf), serializing them with FlatBuffers (FlatBuffers), compressing them with Gzip (Gzip) and writing them back to the Cache's database (Write). Using Java's System.currentTimeMillis() method each DecoderThread measured how long the individual steps lasted. Once the processing was completed the measured timings of all threads were summed up and divided by the number of threads.
Furthermore, the overall execution time was measured in order to calculate an overhead. The overhead equates to the difference between the overall execution time and the summed up timings of the individual steps.
The system used to run the Converter had the following configuration: Intel Core i5-8500 (3 GHz, 6 cores); 8 GB RAM (2.66 Ghz); SSD (SATA); Windows 10 (64 bit). It was connected to the manufacturer's network using 100 MBit/s Ethernet.

Results
The impact of varying the thread-count on the overall execution time can be seen in figure 2.8. Quadrupling the thread-count from one thread to four threads results in an almost four times faster overall processing time. Furthermore, it was observed that the required time largely depended on the time required to deserialize the data. Using more than six threads does not have a positive impact on the execution time as adding additional threads does not result in a more effective use of the DDI. (One can consider that the DDI's configuration or host computer is limited to six threads.)
The effect on the locally executed steps (Protobuf, FlatBuffers, Gzip) can be seen in figure 2.9. Even when exceeding the number of processing cores provided by the CPU adding additional threads has a positive impact on the required time. Considering how the measurement is obtained and that a majority of the threads will always spend their time waiting for the DDI instead of utilizing the local machine's CPU this behaviour was to be expected.
The impact of the job-size on reading and writing from and too a database can be seen in figure 2.10. Especially reading is less costly when fetching the messages in larger sets. This is plausible as it lowers the number of individual search requests. However, combined with six threads a job-size of 250 is sufficient to fully utilize the DDI's performance. Increasing it further will not have an impact on the overall processing time but will take up the local machine's memory.

2.6.4.2 Reliability

To test the Converter's ability to process high amounts of data 3.86 million messages containing approximatly 278,000 ECGs were converted and stored to the Cache. During processing a thread-count of 6 combined with a job-size of 250 was used. The process was repeated five times without difficulty and took 9 hours on average. Thus, the Converter is considered reliable.

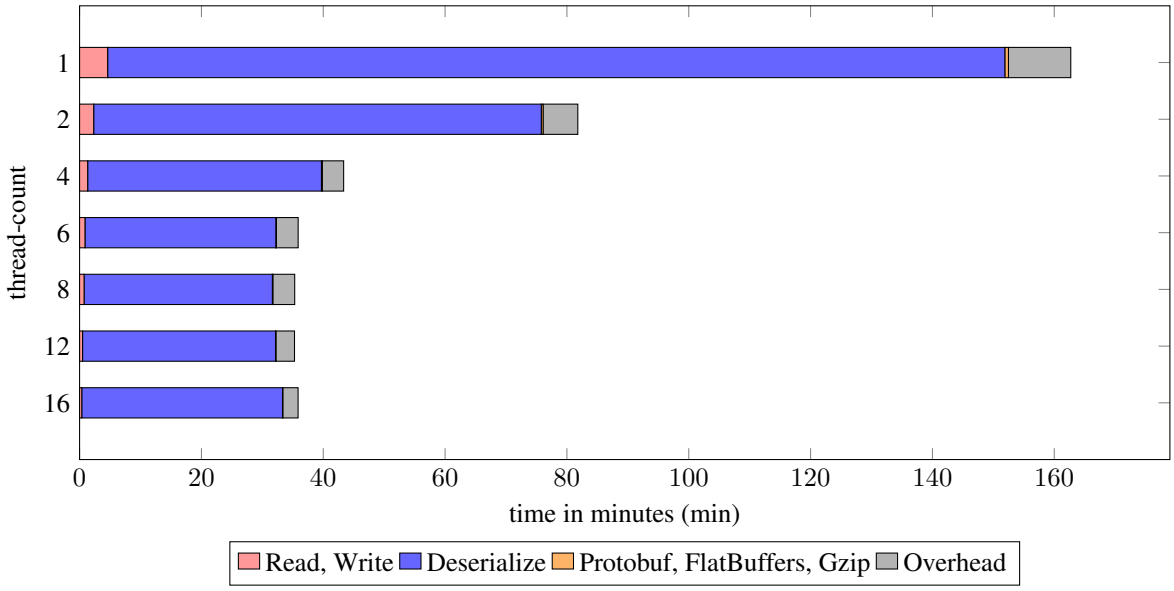


Figure 2.8: Overall time required for converting 100,000 messages (job-size: 250)

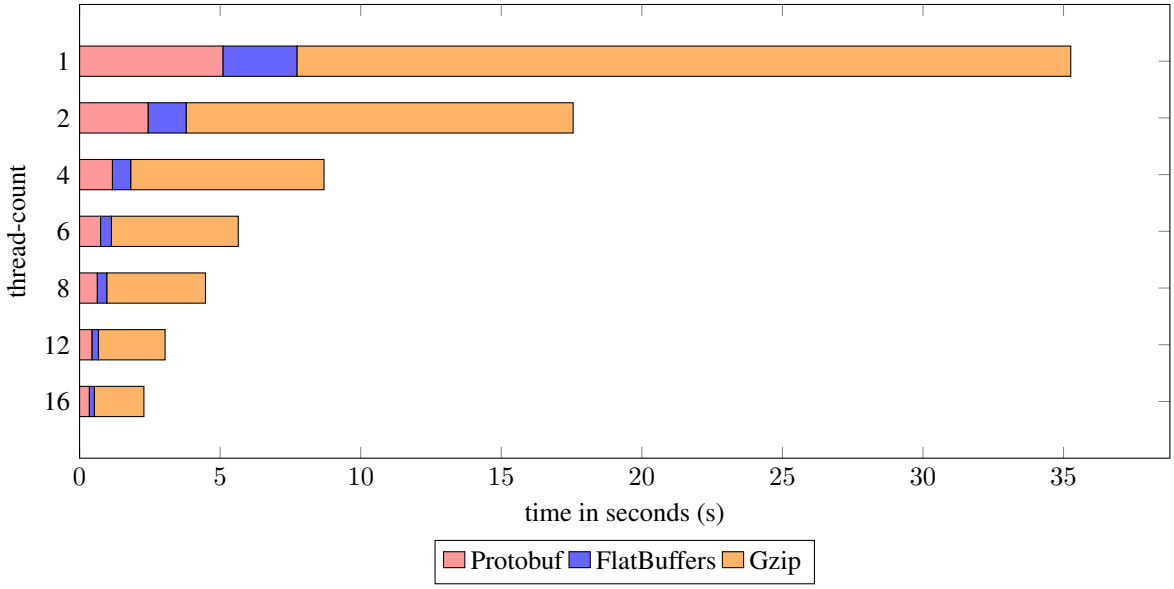


Figure 2.9: Time required for locally executed steps for converting 100,000 messages (job-size: 250)

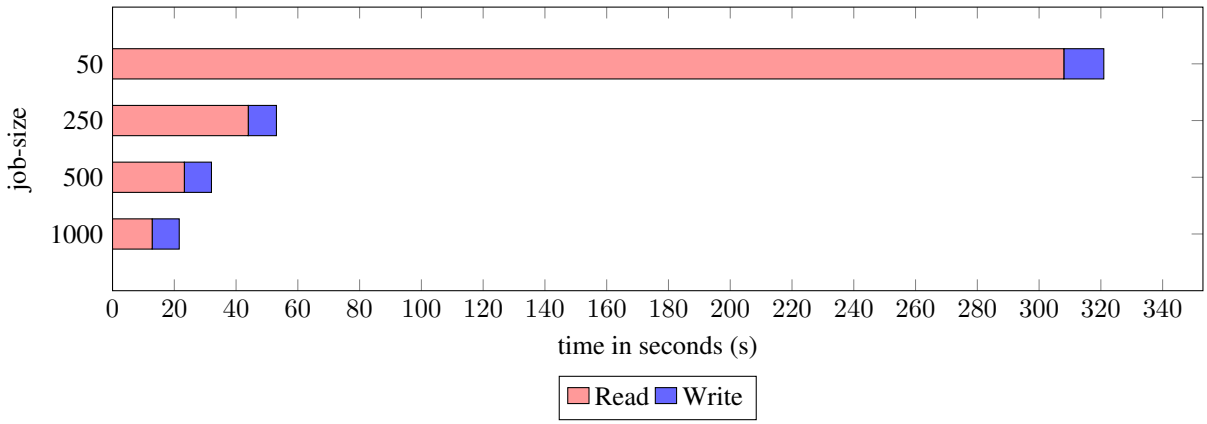


Figure 2.10: Time required for reading and writing 100,000 messages (thread-count: 6)

2.7 API for Python

2.7.1 Considerations

- **Abstraction - Deserialization**
  The code generated by the Protocol Buffers respectively the FlatBuffers compiler enables the user to deserialize the data stored in the Episode Cache. However, the usage of this code requires knowledge of technical decisions made at the stage of defining the used interface.
  The API should provide further abstraction to the process of decoding data in order to hide technical aspects and make accessing the episodes' content as easy as possible.
- **Abstraction - Fetching data**
  Episodes can be stored on a local machine or in a database. Depending on the use case they will be loaded individually or combined to sets of an arbitrary size.
  The API should provide mechanisms to directly fetch and deserialize episodes stored locally or in a database. These mechanisms should be able to load individual episodes or sets of episodes with a limited size. The limitation is caused by the decision that all requested episodes will be loaded at once to keep the usability of the API simple. Bigger amounts of data still can be handled as using the fetching mechanisms will be optional allowing the user to define own fetching and processing methods.
- **Supported data formats**
  The Python API serves as a test of the infrastructure.
  Both formats, Protocol Buffers and FlatBuffers, should be supported.

2.7.2 Implementation

2.7.2.1 Tools and dependencies

For developing the IDE PyCharm 2018.3.2 Community Edition was used. During the API's implementation Python version 3.6 was used. Once Google added Python 3.7 support for its TensorFlow API the Python version was upgraded to 3.7. Deserialization is done using Google Protocol Buffers 3.7.1 and FlatBuffers version 1.10. For accessing databases the package cx-Oracle 7.0.0 is used.

2.7.2.2 Structure

The API is implemented in a module called 'ecg_util' and defines three classes: Episode, FileHandle and DatabaseHandle.

The Episode class describes how episode files are deserialized, parsed and subsequently accessed by the user. For every single episode a new instance of the Episode class is created. Several inner classes are used to structure the episode's content in a coherent and easy to use way.
Depending on the input the method parse_from_protobuf or parse_from_flatbuffers is used to deserialize the passed file and parse its content to the object.

The classes FileHandle and DatabaseHandle are similar in design. Both are initialized with a mode and a location. The mode parameter indicates whether the used data format will be Protocol Buffers (mode 0), FlatBuffers (mode 1) or FlatBuffers but without parsing of the markers (mode 2)[5]. The second parameter refers to the location of the Episode Cache. A path on the local machine is passed when used in context of the FileHandle. When using the DatabaseHandle the location refers to the name of the table storing the episodes. In addition, the DatabaseHandle's init method also gets a connection_url which is required to initially connect to the server hosting the database.
Both classes provide the methods pull_episode and pull_episodes which can be used to load individual episodes or sets of episodes. Given an ID or, respectively, a list of IDs they will return an Episode-object or a list of Episode-objects.

[5] For many applications the markers will not be required. Omitting the markers is expected to have a positive impact on the time required to deserialize the episodes. (Discussed further in 2.7.3.2)

```
class Episode:

    class EcgChannel
    class EcgAnnotation
    class ImplantModel
    [...]

    def parse_from_protobuf(raw_input)

    def parse_from_flatbuffers(raw_input, parse_markers)

class FileHandle:

    def __init__(mode, location)

    def pull_episode(id)

    def pull_episodes(ids)

class DatabaseHandle:

    def __init__(mode, connection_url, location)

    def disconnect()

    def pull_episode(id)

    def pull_episodes(ids)
```

Listing 2.6: Simplified overview of the ecg_util's classes and methods

2.7.2.3 Protocol Buffers and FlatBuffers - Deserialization

The previously used example 'ECGMessage' described in 2.5.4.1 is used in listing 2.7 to demonstrate how encoded data can be accessed in Python.

Once a Protocol Buffers message is deserialized its content can simply be used by accessing variables. In contrast, accessing the content of a FlattBuffers message is done by calling methods corresponding to individual variables. This is due to FlatBuffers exploiting its zero-copy design by only deserializing variables when demanded.

The output generated by executing the code can be seen in listing 2.8. Both formats return a zero for the patient's age which was omitted when initially creating the message.

```
import serialization.protobuf.ECG_schema_pb2 as proto
import serialization.flatbuffers.ECGMessage  as flat
import numpy as np

class ECGMessage:

    def parse_from_protobuf(self, msg_raw):
        #decode whole message
        msg_proto = proto.ECGMessage()
        msg_proto.ParseFromString(msg_raw)

        #assign values to new variables
        self.name = msg_proto.patient_name
```

```
        self.age = msg_proto.patient_age
        self.sampling_rate = msg_proto.sampling_rate
        self.ecg = np.array(msg_proto.ecg_data)

    def parse_from_flatbuffers(self, msg_raw):
        #prepare message to be decoded
        msg_flat = flat.ECGMessage.GetRootAsECGMessage(msg_raw, 0)

        #decode values and assign them to new variables
        self.name = msg_flat.PatientName().decode("utf-8")
        self.age = msg_flat.PatientAge()
        self.rate = msg_flat.SamplingRate()
        self.ecg = msg_flat.EcgDataAsNumpy()

    def __str__(self):
        line1 = "Name: %s; Age: %d \n" % (self.name, self.age)
        line2 = "ECG (%d Hz): %s" % (self.rate, self.ecg)
        return line1 + line2

#open file on local machine
msg_raw = open("example.flat", "rb").read()

#parse content of file to new ECGMessage object
message = ECGMessage()
message.parse_from_flatbuffers(msg_raw)

print(message)
```

Listing 2.7: Python code to access the example message described in 2.5.4.1

```
Name: John_Doe; Age: 0
ECG (1337 Hz): [22.22 22.22]
```

Listing 2.8: Output generated by executing the code in 2.7

2.7.3 Test

2.7.3.1 Example usage

Listing 2.9 shows a minimal working example using the API to visualize an ECG. As the new infrastructure was used to develop and test different classification algorithms a more comprehensive test of the API was done implicitly in chapter 3 and will be discussed in chapter 4.

```
import ecg_util6 as iu
import matplotlib.pyplot as plt

def show_ecg(episode):
    #configure figures appearance
    fig, axs = plt.subplots(nrows=3, figsize=(18, 8), dpi=120, sharex=True)
    fig.subplots_adjust(hspace=0)
    fig.text(0.5, 0.06, "Time (s)", ha="center", va="center")

    #set title
    model = episode.device_info.device_family
    chambers = episode.device_info.number_of_chambers
    fig.suptitle("%s (%d chambers)" % (model, chambers))
```

```
    #draw 3-channel ecg
    tags = ["ra", "rv", "ff"]
    names = ["Right Atrium (mV)", "Right Ventricle (mV)", "Far-Field (mV)"]
    colors = ["C1", "C2", "C0"]
    i = 0
    for tag, name, color in zip(tags, names, colors):
        ch_data = episode.s0.get_channel(tag)
        axs[i].plot(ch_data.x, ch_data.y, label=name, color=color)
        axs[i].legend(loc='upper left')
        i += 1

    plt.show()

#prepare FileHandle
location = "/ecgs_proto/example/"
file_handle = iu.FileHandle(location, mode=0)

#show episode
episode_id = input("Episode id: ") #1147191_2756
episode = file_handle.pull_episode(episode_id)
```

Listing 2.9: Example usage of the ecg_util module

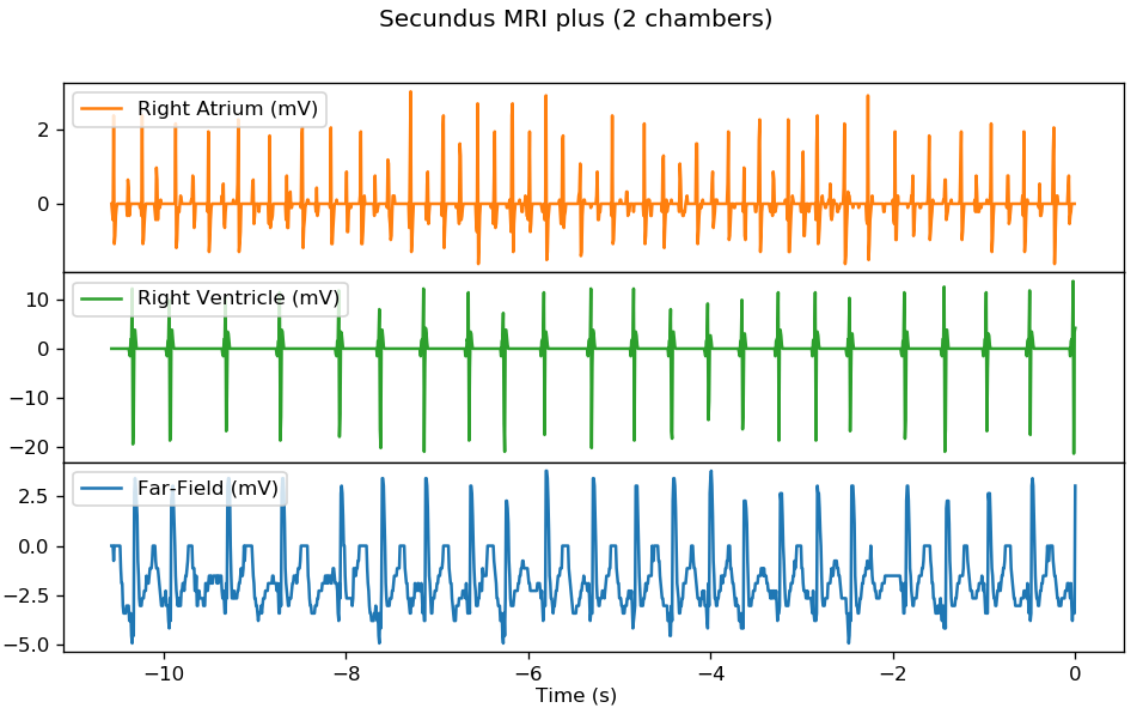


Figure 2.11: Output of the Python code in listing 2.9

2.7.3.2 Performance

Test set-up
To test the API's performance 10,000 episodes containing ECGs were randomly selected and their Protocol Buffers and FlatBuffers files were copied to the local machine. Subsequently, the episodes were repeatedly loaded to Python in order to compare the performance of the different modes described in 2.7.2.2 (0 - Protocol Buffers, FlatBuffers (complete), FlatBuffers (without markers)).
Three kinds of periods were measured: The periods of reading the files from the disk (Read), the periods of decompressing them using the zlib-module (Zlib) and the periods of deserializing them with Protocol Buffer or, respectively, FlatBuffers (Deserialize).

Averaging the results the test was repeated three times to obtain a significant measurement. To prevent the results from being affected by caching mechanism a new set of episodes was used for each iteration.[6] Nevertheless, the measured time should only be considered a guiding value. The true execution duration is expected to strongly depend on the used machine and its current workload.
The system that ran the test had the following configuration: Intel Core i5-8500 (3 GHz, 6 cores); 8 GB RAM (2.66 Ghz); SSD (SATA); Windows 10 (64 bit). All tests were performed single-threaded.

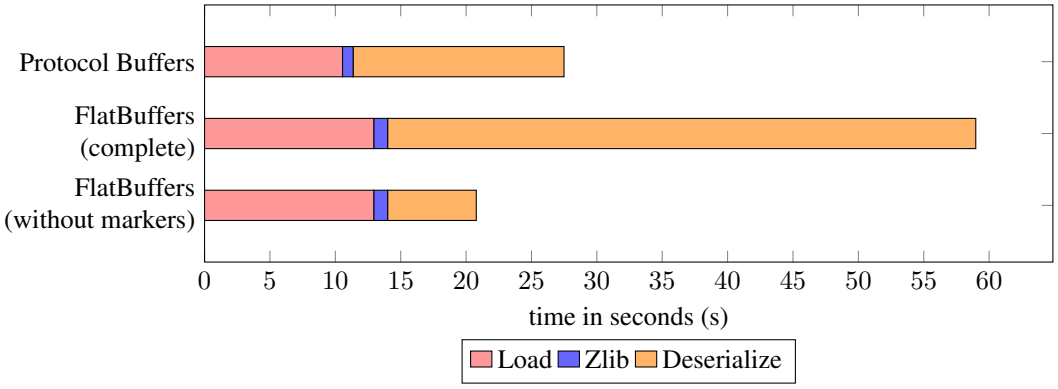


Figure 2.12: Time required for loading 10,000 episodes

Results
How the different modes performed can be seen in figure 2.12. Loading and decompressing is slightly faster using files based on Protocol Buffers. This is presumably caused by Protocol Buffers's smaller file size. Deserializing the entire file Protocol Buffers clearly outperforms FlatBuffers. However, using mode 2 and reading the FlatBuffers files but omitting deserialization of the markers is the fastest way to access the data.
The negative impact of processing the markers was expected and can be explained as follows:
Episodes mainly consist of ECG data and markers. The ECG data is represented as a list of floating point numbers with a known size. Using FlatBuffers this list can be processed by calling Python's numpy-library which heavily relies on external code written in C to improve its performance. When parsing a list of markers (containing strings with an unknown size) this mechanism cannot be used.
However, considering the time Protocol Buffers requires to completely parse the file the markers' effect on FlatBuffers's performance was not expected to be this significant.

[6] Mode 1 and mode 2 use the same FlatBuffers files. Therefore, the Loading and Zlib duration for these modes where only measured once and used for both modes.

Chapter 3

Classification of tachycardias utilizing different machine learning techniques

3.1 Objective

Implantable cardioverter-defibrillators provide live-saving therapies. They prevent patients from suffering a sudden cardiac death using ATP or high-energy shocks (described in 1.6.2).
However, heart diseases do not only impact the patients' physiology but also affect their mental health. Approximately half of ICD patients suffer from depression or anxiety [8]. The physiological distress is not exclusively caused by the ICD. Nevertheless, studies suggest that the severity of symptoms is linked to the number of shocks received during therapy [8, 46]. Both appropriate and inappropriate shocks can be painful and can cause patients to develop shock anxiety. Affected patients tend to limit leisure activities which impedes their quality of life [55].
ICD manufactures put much effort in developing algorithms that distinguish between shockable and non-shockable rhythms [59, 12, 58, 11]. The employed detection algorithms mostly rely on the recognition of single cardiac events (e.g. atrial and ventricular depolarization) and the interpretation of the sequence in which they appear. One of these algorithm is briefly explained in the appendix (figure 5.4).
A major difficulty during detection is to differentiate between SVT and VT (described in 1.5.2). The implant's ability to measure the electrical activity of atrium and ventricle independently of one another improves the implant's ability to differentiate between the two rhythms. Nevertheless, misclassification of SVTs is reported to be the main reason for inappropriate shock therapy [59, 11, 15].

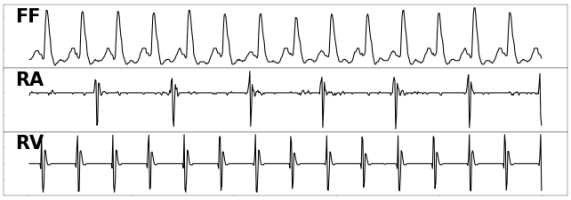
Automated classification of ECGs is important not only for implants but also for non-invasive devices (e.g. external heart monitors or automated external defibrillators (AEDs)). Unlike ICDs, external devices increasingly rely on machine learning techniques for rhythm classification. Ranging from approaches based on manual feature engineering using cardiological knowledge to novel methods employing convolutional neural networks a variety of solutions had been proposed.

Within this chapter a brief overview of existing methods is provided. Thereafter, two of them are implemented to test to what extent they are capable of classifying intracardiac ECG signals. The methods are used to distinguish between SVT and VT addressing current challenges mentioned above.

3.2 Related work

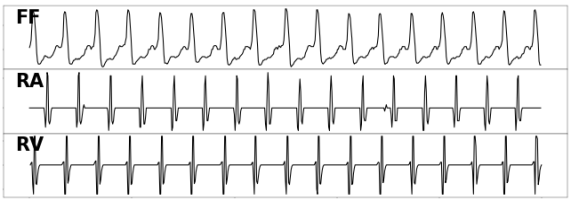
Within the last three decades automated ECG evaluation has been continuously researched and improved. Solutions specifically designed to distinguish between SVT and VT are uncommon as their application is mostly limited to ICD therapy. Related problems of a wider interest are the improvement of automated external defibrillators and the detection of atrial fibrillation.

SVT: The atria flutter. Every second to third impulse is conducted to the ventricles.

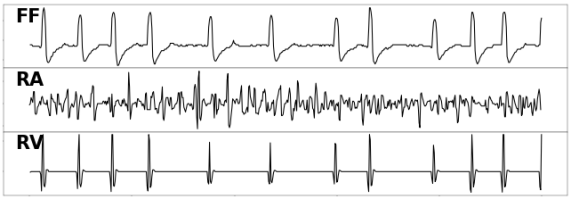


VT: The atrial frequency is normal and exceeded by the ventricular frequency.

SVT: The atrial frequency is increased and conducted (1:1) to the ventricles.



VT: The ventricular frequency is increased and conducted (1:1) to the atria.



SVT: The atria lost an organized contraction pattern. Sporadically impulses are conducted to the ventricles.

VT+SVT: Both atria and ventricle contract rapidly but with a slightly different frequency.

Figure 3.1: ECG examples showing SVT (left side) and VT (right side) episodes. (FF: far-field, RA: right atrium, RV right ventricle, X-axis: Time (5 seconds), Y-axis: Voltage)

The majority of early classification approaches were designed to be used in AEDs and focused on binary classification between shockable rhythms (VF, fast VT[1]) and non-shockable rhythms (e.g. slow VT, SVT).
These approaches often relied on calculating hand-crafted features and comparing them to fixed thresholds. Features directly derived from the signal's time-domain can use, for instance, the auto-correlation function [17], phase space reconstruction [1] or different filter techniques to analyze and quantify certain components of the signal [31, 33]. Other approaches convert the ECG to the frequency-domain to perform simple spectral analysis [6, 39] or to a binary representation to calculate its complexity [60]. In reviving these methods it was shown that combining several features e.g. by means of discriminant analysis [30] or support vector machines (SVMs) [35] could further improve classification.

Novel approaches and their application are not limited to be used in scenarios where computer-aided classification is inevitable. Instead, they focus on complementing therapy and supporting physicians. One application is the detection of atrial fibrillation. AF is one of the most common abnormal heart rhythms and can cause SVT.
Within the PhysioNet Challenge 2017 a variety of known and novel AF detection methods had been implemented and tested under uniform conditions [18]. For the challenge the classifiers had to distinguish between normal rhythms, atrial fibrillation and other anomaly. Top performing approaches relied on hand-engineered features, convolutional neural networks (CNNs), recurrent neural networks (RNNs) or the use of multiple classifiers at once. All approaches using manual feature engineering analyzed the signal not only globally but on a beat-to-beat basis. This necessitated the inclusion of detecting QRS comlexes (see 1.4) as a preprocessing step. While some of the features are comparable to those mentioned above the majority of metrics are new as the per-beat analysis provides new possibilities (e.g. measuring PR and QT intervals or the heart rate variability). Extracted features were interpreted, for instance, by means of Random forests [57], AdaBoost [9] or multilayered peceptrons (MLPs) [3]. Methods that relied on CNNs classified the signal directly using 1d convolution [56, 3] or processed a spectrogram of the ECG which can be calculated using short-time Fourier transform (STFT) [56, 61]. Using RNNs

[1] 'Fast VT' is not uniformly defined. Common thresholds are 150 or 180 bpm.

some solutions analyzed the signal or the features derived from it in sequences [51]. For enhancing performance or combining multiple classifiers some approaches employed boosting algorithms such as XGBoost [27]. Noteworthy solutions found beyond the PhysioNet Challenge include the use of 2d CNNs combined with wavelet transform [26] or simply an image of the ECG. The latter can be classified using well-known CNN models like AlexNet or VGGNet [32].

Table 3.1: Comparison of different methods proposed for ECG classification. (Selected methods marked blue.)

Author	Techniques involved	Rhythms	Data	Performance	Split
Chent et al. 1987 [17] †	Autocorrelation	Shockable, Non-shockable	AHA, MIT-BIH	Se: 78 % Sp: 32 %	-
Barro et al. 1989 [6] †	Fourier transform, Spectral analysis	Shockable, Non-shockable	AHA, MIT-BIH	Se: 79 % Sp: 93 %	-
Zhang et al. 1999 [60] †	Complexity of binary representation	Shockable, Non-shockable	AHA, MIT-BIH	Se: 66 % Sp: 75 %	-
Jekova et al. 2004 [31]	band-pass filter, auxiliary counts	Shockable, Non-shockable	AHA, MIT-BIH	Ac: 94.7 % Se: 95.9% Sp: 94.4 %	-
Amann et al. 2005 [2]	Complexity of binary representation	Shockable, Non-shockable	AHA, MIT-BIH	Se: 66 % Sp: 75 %	-
Jekova 2007 [30]	Feature engineering Discriminant analysis	Shockable, Non-shockable	AHA, CUDB, VFDB	Se: 94.1 % Sp: 93.8 %	-
Li et al. 2013 [35]	Feature engineering, SVM	Shockable, Non-shockable	AHA, CUDB, VFDB	Ac: 96.3 % Se: 96.2% Sp: 96.2 %	✓
Xiong et al. 2017 [56]	STFT, 2d CNN	AF, NSR, Other	PhysioNet DB	F_1: 0.780	(✓)
	1d CNN	AF, NSR, Other	PhysioNet DB	F_1: 0.817	(✓)
Bin et al. 2017 [9]	Feature engineering, AdaBoost	AF, NSR, Other	PhysioNet DB	F_1: 0.821	(✓)
Zihlmann et al. 2017 [61]	STFT, 2d CRNN	AF, NSR, Other	PhysioNet DB	F_1: 0.821	(✓)
Hong et al. 2017 [27]	Feature engineering, 1d CNN, XGBoost	AF, NSR, Other	PhysioNet DB	F_1: 0.825	(✓)
Zabihi et al. 2017 [57]	Feature engineering, Random Forest	AF, NSR, Other	PhysioNet DB	F_1: 0.826	(✓)
Teijeriro et al. 2017 [51]	Feature engineering, RNN, XGBoost	AF, NSR, Other	PhysioNet DB*	F_1: 0.831	(✓)
Andreotti et al. 2017 [3]	Feature engineering, MLP	AF, NSR, Other	PhysioNet DB	F_1: 0.791	(✓)
	1d CNN	AF, NSR, Other	PhysioNet DB*	F_1: 0.830	(✓)
Jun et al. 2017 [32]	2d CNN (AlexNet)	6 classes (including fast VT)	MIT-BIH	Ac: 98.5 % Se: 96.8 % Sp: 99.7 %	-
	2d CNN (VGGNet)	6 classes (including fast VT)	MIT-BIH	Ac: 98.4 % Se: 97.3 % Sp: 99.4 %	-
He et al. 2018 [26]	Wavelet transform, 1d CNN	AF, Other	MIT-BIH	Ac: 99.2 % Se: 99.4 % Sp: 98.9 %	-

✓ patient-split mentioned; (✓) patient-split very likely; - patient-split not mentioned; * Database augmented or manually relabeled; † performance reported by Jekova [29]; Ac - (averaged) Accuracy; Se - (averaged) Sensitivity; Sp - (averaged) Specificity; F_1 - averaged F_1-Score; ECG-Databases: American Heart Association (AHA), Massachusetts Institute of Technology Beth Israel Hospital (MIT-BIH), PhysioNet (see [18])

When comparing the proposed solutions it is important to consider the technique used for creating training and test sets. ECGs can be used for biometric recognition [41]. Furthermore, individual patients have individual probabilities of suffering from certain arrhythmias. Thus, it is advisable to split training and test data not only by samples but by patients when evaluating ECG classifiers. Without performing a 'patient-split' it must be considered

a classifier does not recognizes different rhythms but patients or is only capable of distinguishing rhythms of already known patients.
Although applications with models fitted to individual patients exist (e.g. long-term inpatient monitoring) this is not feasible for ICD therapy. Implants have to recognize treatable rhythms with their first appearance in a patient.

3.2.1 Method selection

Two of the methods listed in table 3.1 were chosen to be tested on intracadiac ECGs.

The first method is the one proposed by Li et al. in 2013. Using an SVM Li combines several metrics previously published by other authors. Therefore, this approach was considered a good representative for research done in the field of AED improvement.

The second one is the 1d CNN submitted by Anderotti in the PhysioNet challenge 2017. This CNN was originally designed by Hannun et al. in 2017 [44] and further reviewed in 2019 [25]. Although other competitors achieved similar results without augmenting the training data they did rely on QRS detection and feature extraction. In contrast, the CNN did not require any preprocessing and, therefore, substantially differs from the first method chosen.

3.2.2 Method 1: Li et al.

Using surface ECGs from 67 patients Li focuses on distinction between shockable rhythms (including ventricular flutter, VT and VF) and non-shockable rhythms (including, among others, normal rhythms, AF and SVT). The ECGs were obtained from different databases and had an overall length of 29 hours. For classification the signals were segmented testing different window sizes ranging from 1 to 10 seconds. Individual segments were annotated using cardiological knowledge. Reviewing other studies aimed at AED improvement Li addresses the following limitations: Lag of out-of-sample testing (previously referred to as patient-split), relying on single features or a single type of features and the use of fixed thresholds for classification.
To overcome these limitations 14 features previously proposed by other authors were implemented and combined using an SVM. For optimization different subsets of features were tested by means of a genetic algorithm. Testing was performed using five-fold cross-validation with distinct patients between folds.
The final model used a window length of 5 seconds, a combination of only two features (Count2 and Leakage, will be described in 3.4.1) and achieved an accuracy of 96.3 ± 3.4 %. Shorter windows lowered the accuracy but still provided decent performance (e.g. 2 s - Ac: 95.2 ± 3.3). Larger windows could not further improve the performance. Using more features resulted in a higher training accuracy but failed to improve the model's performance during testing. Li suspects the high standard deviation during cross-validation to be caused by the characteristic diversity of rhythm types between patients. Addressing the importance of out-of-sample testing the final model's performance was reported to be 3 % higher without patient-split. [35]

3.2.3 Method 2: Hannun et al.

The CNN model proposed by Hannun is designed to distinguish ECGs into 14 different classes (including AF, SVT and VT). Training is done using 64,121 ECGs with a length of 30 seconds from 29,163 patients. All signals had been captured by a single-lead wearable heart monitor. For classification the samples were segmented with a window size of 1.28 seconds[2] and annotated by clinical experts. During implementation Hannun experimented with different CNN models varying the number of convolutional layers and the size and number of filters. Furthermore, residual (short-cut) connections as well as recurrent layers were tested. The model's performance was measured using 336 ECGs annotated by a committee of experts discussing the samples as a group. The same set of samples was also given to individual experts in order to estimate the performance of individual cardiologists.
The final CNN consists of 33 convolutional layers, residual connections and will be described further in 3.5.1. Recurrent layers are not used as they did not improve the performance but substantially increased the runtime. The

[2]The singal is sampled with 200 Hz. For classification 256 sample points are used.

average F_1 score (calculated using the harmonic mean) of the CNN was 0.776 and exceeded the score achieved by individual cardiologists which was 0.719. [44]

In 2019 the model was reviewed using a larger set of ECGs (92,232) from a larger group of patients (53,549). This time 9 (instead of 6) experts labeled the test data. The findings from 2017 were conformed. The CNN's F_1 score was 0.807. Individual experts achieved 0.753. [25]

3.3 Data

Collection
The data used in this thesis was collected by over 4,000 two-chamber ICDs of the same product family. Altogether the implants transmitted approximately 20,000 tachycardia episodes during a period of 4 years. Further analysis of underlying diseases, comorbidities or demographic aspects was not possible as all episodes were fully anonymized and separated from patient-specific information.

Channel
The ECGs contained within the episodes provide three channels: RA, RV, FF (described in 1.6.3). To train and test the selected classification methods only the FF channel is used. The FF signal is similar to an externally recorded signal and, thus, comparable to the data originally used by Li and Hannun. Unlike the near-field channels (RA, RV) the FF channel is expected to contain both atrial and ventricular activity. This is considered important for distinguishing between SVT and VT. Furthermore, conventional ICD detection algorithms mainly rely on the near-field channels. Therefore, omitting these channels and relying purely on the FF channel can be an interesting alternative.

Segmentation
The ECGs transmitted by an ICD have a length of 20 to 40 seconds and are separated into two phases: Detection and, if required, therapy. During the detection phase the implant decides whether or not medical intervention is appropriated. Thereafter, any kind of treatment is recorded within the therapy phase.
Following Li the classification will be done on non-overlapping segments with a length of 5 seconds. Each episode is only used to extract one segment which refers to the last 5 seconds of the detection phase. Early parts of the detection phase are not guaranteed to contain any abnormal rhythm. Using parts of the therapy phase could bias the classification to the implant's intervention.
The features used for the SVM are calculated using the complete segment. CNN classification is done using a randomly chosen 1 to 4 second snippet of the segment. This is done to exclude the possibility of the cut-off point (chosen by the implant) becoming an unintended feature.

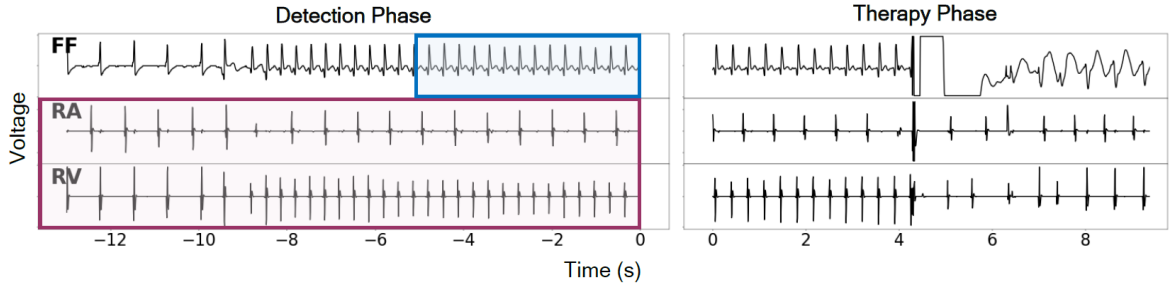


Figure 3.2: ECG transmitted by an ICD before and during successfully delivering a shock. The part marked blue referrers to the far-field segment used for classification. The area marked purple referrers to the data originally used by the implant and the trainer to distinguish between SVT and VT. (FF: far-field, RA: right atrium, RV right ventricle)

Splitting
Using the ICDs' hashed serial numbers the implants/patients were randomly but evenly assigned to 6 distinct groups. Subsequently, each group was used to create a data set $(s1, s2, ..s6)$. The sets were filled by adding up to 10 SVT episodes and 10 VT episodes for each implant included in the corresponding group. Except for a few outliers the majority of ICDs will stay within the limitation of 10+10 episodes per implant. Finally, the SVT and VT episodes inside each set were balanced. As the number of both do not vary significantly this was simply done

by randomly removing episodes.
Following this procedure an implant/patient cannot contribute to more than one data set, a bias on implants sending more frequently than others is avoided and problems arising from unbalanced data are prevented.
How the sets are used to train and test the models will be explained in 3.6.

Annotation
The segments were annotated SVT or VT with a classifier that provides an accuracy of approximately 94 % with a high focus on sensitivity[3], from now on referred to as trainer. The accuracy of the trainer was estimated by two experts who relabeled a randomly chosen subset of the samples contained in $s6$. The relabeled subset is referred to as ground truth (GT) and will be used to test the models.
Although the annotation provided by the trainer is not ideal it was deliberately avoided to use data available in public databases. The annotation accuracy of such databases is presumably higher. However, they rarely contain ECGs recorded by implants. Therefore, using external data would miss the point of testing the models on intracardiac signals. Furthermore, it would fail to test the infrastructure as a whole.

Table 3.2: Number of samples and contributing patients per set

Set	$s1$	$s2$	$s3$	$s4$	$s5$	$s6$	GT
Number of samples	2252	2230	2176	2294	2018	2372	375 (212 SVT + 162 VT)
Number of patients	608	617	601	612	585	608	227

[3]Meaning in case of doubt or ambiguity it would classify an episode as VT. (Sensitivity is explained further in 3.6)

3.4 Methodology 1: Support vector machine

The methodology of the first approach is based on the work of Li et al. [35] and is implemented using the scikit-learn library 0.20.3.

3.4.1 Feature extraction

Before calculating the features Li originally passes the signal through different filters to eliminate noise and artifacts (e.g. baseline drifts, power line interferences and high-ferquency noises). Non of these effects are present in ICDs or are already considered by the implant. Therefore, the preprocssing step is skipped.
The 15 extracted features can be categorized in three different types as follows: 6 time-domain features, 4 binary features and 5 frequency-domain features. Except for one self-proposed frequency-domain feature (FunFreq) they correspond to the metrics used by Li. All are calculated using the whole segment (5 seconds).

Binary features

Before calculating the features Complexity, AreaBin, FreqBin and CovarBin, the analyzed ECG segment is transformed to a 0/1 binary string. This is done by comparing the data points of the segment to a suited threshold.
For a segment containing n data points $x_1...x_n$ the binary representation is calculated as follows: First, the mean value x_m of the segment is calculated. Thereafter, x_m is subtracted from each data point. Then the negative peak value V_n and the positive peak value V_p are determined. Next, the values C_p and C_n are calculated with $C_n = |\{x_i \mid 0.1V_n < x_i < 0\}|$ and $C_p = |\{x_i \mid 0 < x_i < 0.1V_p\}|$. Finally, the threshold Td is set: $Td = 0$ if $(C_n + C_p) < 0.4n$; else if $C_n > C_p$ then $Td = 0.2V_p$; else $Td = 0.2V_n$. Given Td the binary representation $s_1...s_n$ with $s_i = 0$ if $x_i < Td$ else $s_i = 1$ is calculated. [60]
The procedure's Python implementation is shown in appendix (listing 5.1).

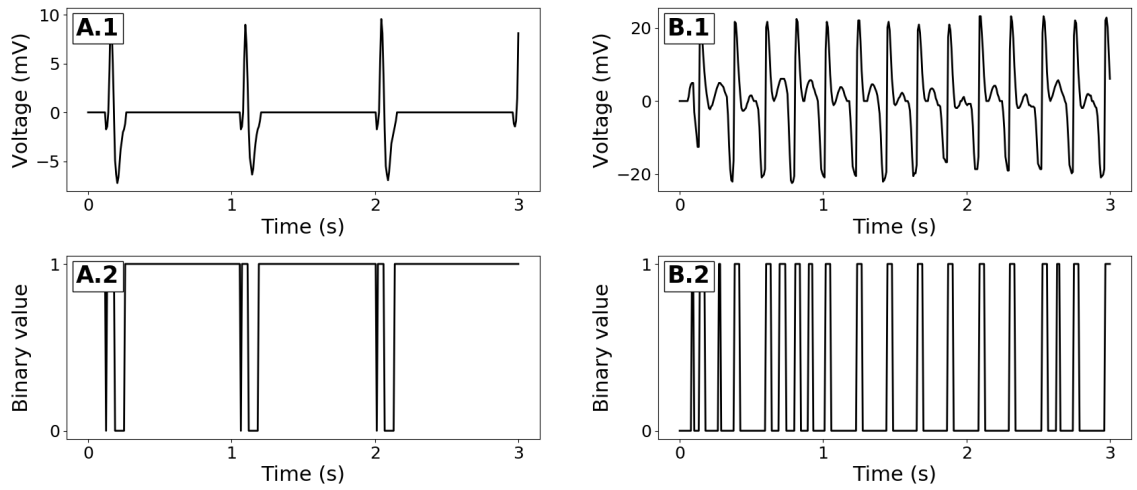


Figure 3.3: A.1: ECG of a normal sinus rhythm; B.1: ECG of a ventricular fibrillation; A.2: Binary representation of A.1 with $Td = -1.42$; B.2: Binary representation of B.1 with $Td = 4.57$.

Complexity - Zhang et al. 1999 [60]

Given the binary string $B = s_1...s_n$ the string's Complexity c is calculated using the following procedure: Let S and Q represent strings, SQ the strings' concatenation and $SQ\pi$ the strings' concatenation without the last character. At the beginning S and Q are empty and c is set to 0. Starting with s_1 the characters contained in B are appended to Q. For each appended character the algorithm checks if Q is a substring of $SQ\pi$. If $Q \in SQ\pi$, the next character from B is added to Q. S and c remain unchanged. Otherwise, if $Q \notin SQ\pi$, the content of Q is appended to S, c is incremented by 1 and Q is emptied. The procedure is repeated until the last character in B was copied to Q.

Zhang describes the asymptotic behavior a of a binary string's complexity with length n as $a = n/\log_2 n$. Therefore, the complexity c of a string with length n can be normalized with $C = c/a$.
The corresponding Python source code is shown in the appendix (listing 5.2).

AreaBin, FreqBin, CovarBin - Jekova et al. 2005 [31]

AreaBin equals the maximum between the numbers of zeros and the number of ones contained in the binary string. FreqBin counts the transitions between zeros and ones. CovarBin refers to the variance of the string.[4]

Time domain features

Timedelay - Amann et al. 2007 [1]

The Timedelay is calculated by plotting the signal's points $x_1...x_n$ to a 40×40 grid. The size a of each square in the grid equals $a = (max(x_1...x_n) - min(x_1...x_n))/40$ in order to stretch the grid from the minimum to the maximum of the analyzed ECG signal. The points are plotted to the grid with the x-axis corresponding to x_i and the y-axis corresponding to x_{i+N}. N depends on the sampling frequency and represents the number of points contained in 0.5 s. For measuring the Timedelay the number of squares containing at least one data point is divided by the total number of squares.

$$Timedelay = \frac{\text{number of visited squares}}{1600}$$

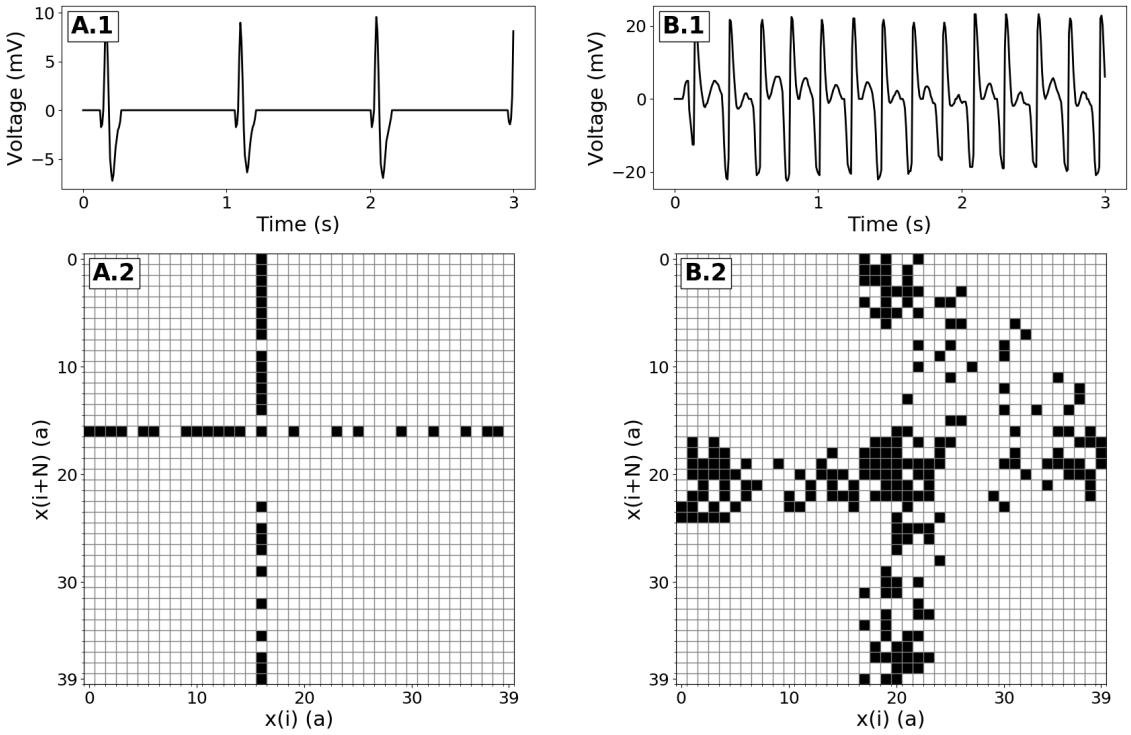


Figure 3.4: A.1: ECG of a normal sinus rhythm; B.1: ECG of a ventricular fibrillation; A.2: Timedelay grid of A.1 (45 visited squares); B.2: Timedelay grid of B.1 (204 visited squares).

Leakage - Kuo et al. 1978 [33]

Leakage is based on the idea of applying a narrow band-stop filter to the ECG. The filtered frequency equates to the mean frequency of the ECG signal.
The ECG is added pointwise to a shifted copy of itself. The shift coincides with half a period of the signal. For a segment containing n data points $x_1...x_n$ the mean number of points N contained in half a period is estimated with

$$N = \lfloor \pi \left(\sum_{i=1}^{n} |x_i| \right) \left(\sum_{i=1}^{n} |x_i - x_{i-1}| \right)^{-1} + \frac{1}{2} \rfloor. \tag{3.1}$$

[4]The variance of a binary string is expected to have a direct dependence on the feature AreaBin. Nevertheless, both were implemented to comply with the original paper.

Given N the leakage is calculated as

$$Leakage = \left(\sum_{i=1}^{n} |x_i + x_{i-N}|\right)\left(\sum_{i=1}^{n} (|x_i| + |x_{i-N}|)\right)^{-1}.$$

Count1, Count2, Count3 - Jekova et al. 2004 [31]

The auxiliary count calculation starts with applying a band-pass filter to the ECG signal which only allows frequencies in the range from 13 to 16.5 Hz to pass. The frequency range is based on previously made observations [39, 40] and is used to focus on the non-shockable rhythm components of the signal. Jekova implemented this step using an integer coefficient recursive filter. However, the filter was designed for signals recorded with a sampling frequency of 250 Hz. As the implants use a different sampling frequency preprocessing was done employing a fifth order butterworth filter.
Using the absolute values of the filter's output FS each auxiliary count corresponds to the number of sample points that have an amplitude within a certain range:

Count1 = $|\{x \in FS : 0.5 \cdot FS_{max} < x < FS_{max}\}|$
Count2 = $|\{x \in FS : FS_{mean} < x < FS_{max}\}|$
Count3 = $|\{x \in FS : FS_{mean} - FS_{MD} < x < FS_{mean} + FS_{MD}\}|$

The max value FS_{max}, mean value FS_{mean} and mean deviation FS_{MD} are calculated for every 1 s time interval of FS.

Kurtosis - Li et al. 2007 [34]

After mean, variance and skewness the kurtosis is the fourth standardized moment of a distribution. It describes to what degree a distribution is peaked. For an ECG segment X with mean μ and standard deviation σ the signals Kurtosis is calculated as

$$K(X) = E\left[\left(\frac{X - \mu}{\sigma}\right)^4\right] \tag{3.2}$$

with $E[\,]$ refering to the mathematical expectation operator.

Frequency domain features

For frequency analysis the ECG segments are multiplied by a Hamming window and transformed into the frequency domain using the Fast Fourier Transform. For further processing the modulus $|z|$ of the resulting complex numbers $(x + iy)$ is calculated using $|z| = \sqrt{x^2 + y^2}$.[5] Values in the range from 0 to 0.5 Hz are rejected from the analysis as they are unlikely to contain diagnostic-relevant data but can contain baseline drifts.[6]

FunFreq - self-proposed

Following a self-proposed approach FunFreq is used to estimate the fundamental frequency (f_0) of the ECG which usually coincides with the heart rate. For normal sinus rhythms f_0 is mostly represented by the first peak of the spectrum and has up to 20 harmonics[7] represented by narrow peaks. In contrast, SVT and VT rhythms tend to have a fundamental frequency matching the maximum peak and contain fewer but broader harmonic peaks. However, in both cases f_0 is not necessarily represented by the maximum peak nor by the first peak of the spectrum. Thus, it is calculated as follows:

1. Based on the observation f_0 is smaller or equal to the peak frequency all peaks for frequencies smaller or equal to the peak frequency are considered to coincide with the potential f_0. They are labeled f_0-candidates (f_C).

[5] The authors of the spectral features (Barro et al. [6]) originally approximated the modulus simply by summing the real and the imaginary part ($|z| = x + y$). However, Barro does not justify this approach. As the paper was released in 1989 it is conceivable it was done in respect to the computational complexity of a root function.

[6] Although baseline drifts are not present in the used data this step was implemented.

[7] Whole number multiples of f_0

2. For each f_C a list of corresponding harmonics is calculated. This is done by multiplying f_C with whole numbers and searching for peaks close to the expected locations. Allowing the algorithm to search for peaks within a small margin is important because f_C can be estimated only roughly given the low resolution of the spectrum[8].
3. Based on the average distance between the potential harmonics f_C is recalculated to receive a more accurate value.
4. For each f_C a cosine wave is created with the wave's period equivalent to f_C. The cosine wave's amplitudes linearly decent and equals zero after 20 oscillations.
5. The cosine wave is multiplied by the spectrum. The integral of the resulting output is used as a score for f_C. The candidate with the highest score is considered to be f_0.

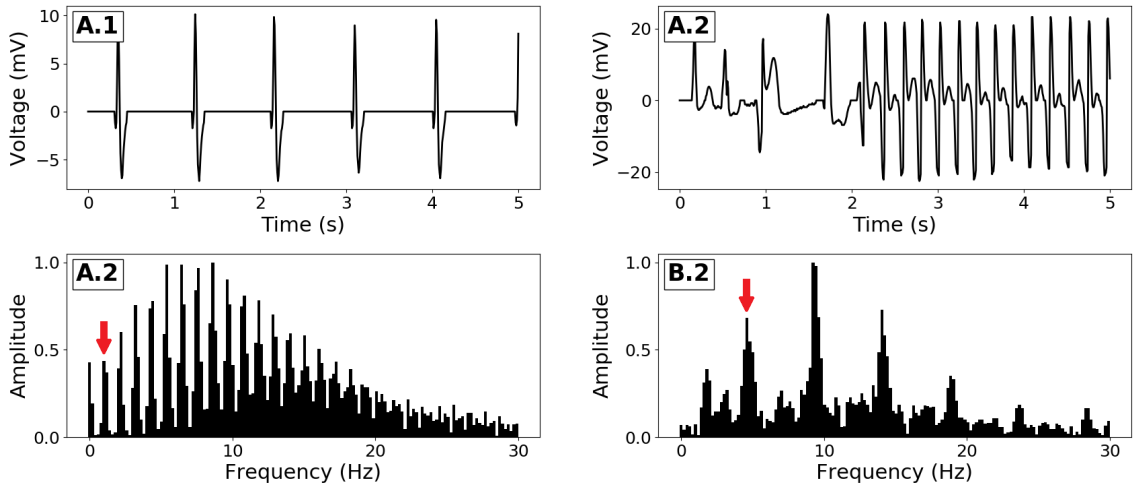


Figure 3.5: A.1: ECG of a normal sinus rhythm; B.1: ECG of a ventricular fibrillation; A.2: Spectrum of A.1; B.2: Spectrum of B.1. (Fundamental frequency marked by red arrow.)

FSMN, A1, A2, A3 - Barro et al. 1978 [6]

Based on the ECG's spectrum and a reference frequency F four metrics are calculated. Barro described F as follows: For rhythms with rapid VF F coincides with the maximum peak of the spectrum; For rhythms with identifiable heart beats F refers to the heart rate. The first category of rhythms is unlikely to be present in the data as an ICD will not allow the heart to change into this dangerous condition. For the second category Barro does not specify how the heart rate is estimated using the spectrum. Thus, F is set using the FunFreq metric described previously.

With $f_1 ... f_n$ being the frequencies and $a_1 ... a_n$ the associated amplitudes the spectrum's total area (T) is calculated as follows:

$$T = \sum a_i \quad i \in \{i : 0.5 < f(i) < 20 \cdot F\}$$

Subsequently, the four features are calculated.

1. Spectrum's center of gravity (FMSN):

$$FSMN = \frac{1}{F \cdot T} \sum_i a_i \cdot f_i \quad i \in \{i : 0.5 < f_i < 20 \cdot F\}$$

2. Ratio between 0.5 Hz to $0.5F$ and the total area (A1):

$$A1 = \frac{1}{T} \sum_i a_i \quad i \in \{i : 0.5 < f_i < 0.5 \cdot F\}$$

[8]The spectrum's resolution depends on the used window length.

3. Ratio between $0.7F$ to $1.4F$ and the total area (A2):

$$A2 = \frac{1}{T}\sum_i a_i \quad i \in \{i : a(i) : 0.7 \cdot F < f_i < 1.4 \cdot F\}$$

4. Ratio between amplitudes surrounding the 2nd to 8th harmonics in $0.6F$ bands and the total area (A3):

$$A3 = \frac{1}{T}\sum_i a_i \quad i \in \{i : F \cdot k - 0.6 \cdot F < f_i < F \cdot k + 0.6 \cdot F\} \quad k \in \{2, 3..8\}$$

3.4.2 Classification

Before being used for classification all features are normalized by the StandardScaler provided by the scikit-learn library. For normalization the mean μ and standard deviation σ of each feature is calculated. Thereafter, every value (x) of the feature is normalized as follows:

$$x' = \frac{x - \mu}{\sigma}$$

The SVM is configured to a Gaussian radial basis function kernel defined by

$$K(x_n, x_m) = exp(-\gamma |x_n - x_m|^2)$$

where x_n and x_m refer to feature vectors expressed in the initial input space. The kernel coefficient γ is not further described by Li and is set to the implementation's default which is the reciprocal value of the number of used features. All other parameters (including the penalty parameter C) also remain unchanged. Their default values can be found in the scikit-learn documentation [19].

3.4.3 Feature selection

It must be assumed that using all features at once does not necessarily improve the performance. For selecting a suited subset of features Li originally used a genetic algorithm which explores possible combinations in a semi-random manner. However, for reasons of simplicity the implementation of a genetic algorithm was avoided in this thesis. Instead, the feature selection is performed using brute-force search. During search all possible combinations of up to 15 features are systematically tested. For 15 features this results in 32,767 permutations.

3.5 Methodology 2: Convolutional neural network

The second methodology is based on the work of Hannun et al. [44], the source code submitted by Andreotti and is implemented using Google's TensorFlow API version 1.13.0.

3.5.1 Network architecture

The models architecture can be seen in figure 3.6. Its input is a raw ECG segment passed as series of 256 measuring points. The CNN consists of 16 residual blocks. Each block contains two convolutional layers and one dropout layer. Following the convolutional layers batch normalization and ReLU activation are performed. Additionally, max pooling is applied every second residual block on both the main branch and the shortcut connection. Main branch and shortcut connection are merged using elemet-wise addition. The fully connected layer at the end of the network outputs the sample's probability being an SVT or, respectively, a VT. Thanks to the residual blocks' repeating pattern the larger part of the network can be implemented using a loop.
All convolutional layers use a filter length of 16 with a stride of 1 and are initialized using the He normalization. The number of filters per layer starts with 64 but is doubled every forth time the loop is passed. The dropout rate is set to 0.5 and the pooling layers subsample the signal by a factor of 2.
The model is trained using the Adam optimizer (with its default settings) and the categorical cross-entropy loss function.

3.5.2 Optimization

In context of this thesis the available ECG classification models should be tested on intracardiac signals. Therefore, it was avoided to fundamentally change the network's structure or the used training procedure. However, four parameters are varied as they are easy to manipulate and likely to have a positive impact when adjusted properly.
First, the number of residual blocks. The number of blocks can be controlled easily by changing the number of iterations (n) of the loop contained in the network. Originally n is set to 15. As the number of available training samples is considerably smaller compared to the number used by Hannun it can be reasonable to reduce the network's complexity.
Secondly, the number of measuring points used as an input. This is important as the sampling frequency of the implants does not equal the sampling frequency of the external heart monitors. Thus, the same number of points does not correspond to the same period. Furthermore, the ECG period used by Hannun (1.28 seconds) is comparatively short. This is comprehensible as he wants to distinguish the signal into 14 types with some of them being able to quickly evolve into others. However, this is not the case for SVT/VT classification. Larger segments corresponding to longer periods could enhance classification.
Furthermore, the mini-batch size and the number of epochs will be optimized.

Using grid search all combinations of considered parameters will be tested in an exhaustive manner.

Table 3.3: Hyperparameters and their tested values

	Length of input (s)	**Size controlled by n**	**Batch size**	**Number of epochs**
Tested in this thesis	1, 2, 4	5, 10, 15	16, 32, 64, 128	up to 50
Used by Hannun	1.28	15	32	up to 100

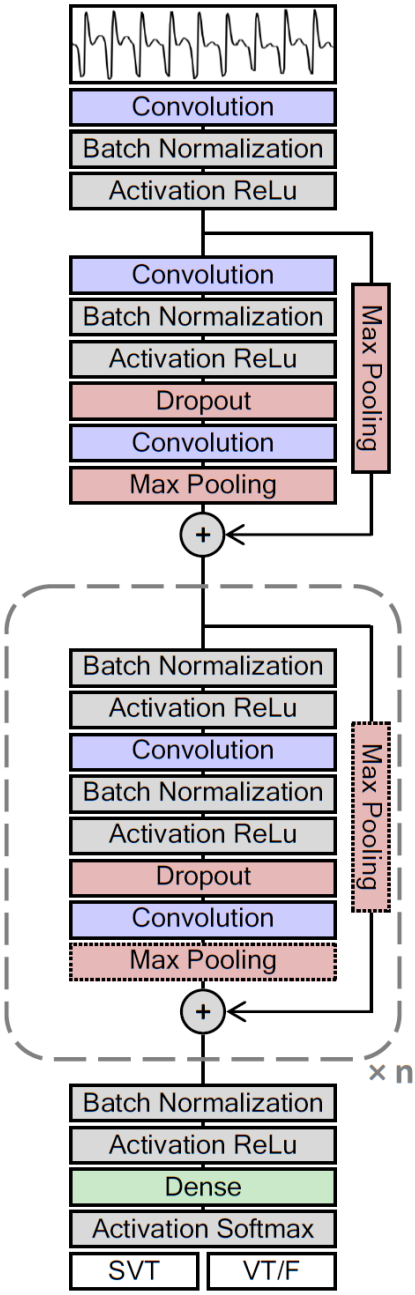


Figure 3.6: CNN architecture

3.6 Validation and test

5 of 7 sets ($s1, s2, ..s5$, see 3.3) are used to optimize the models by means of five-fold cross-validation. Subsequently, the optimized models are tested using $s6$ and the ground truth.

For measuring a model's performance its accuracy (Ac), sensitivity (Se) and specificity (Sp) are calculated. In context of binary classification tasks the accuracy simply referrers to the proportion of correctly predicted samples among the total number of samples. The sensitivity measures the proportion of correctly identified positives (VT episodes) and the specificity is the proportion of correctly identified negatives (SVT episodes).
As the relabeled ground truth is imbalanced its balanced accuracy (Ac_B) is obtained by averaging sensitivity and specificity.

$$Ac = \frac{TP + TN}{TP + TN + FP + FN} \quad Se = \frac{TP}{TP + FN} \quad Sp = \frac{TN}{TN + FP} \quad Ac_B = \frac{Se + Sp}{2}$$

(TP = True Positives, TN = True Negatives, FP = False Positives, FN = False Negatives)

Validation
During validation the models are tuned by varying, for instance, the SVM's feature selection or the CNN's size (described in 3.4.3 and 3.5.2) to maximize the respective model's performance. Employing cross-validation all tested configurations are evaluated as follows: 4 of 5 sets are used to train the model while the remaining set is used for testing it. This process is repeated five times with a different test set in each iteration. Thereafter, the results of all five runs are averaged to estimate the performance.
Following this procedure every sample is used for training and testing. This helps to overcome the possibility that a single set (more specifically, the group of patients used to create it) is not representative of the whole population.

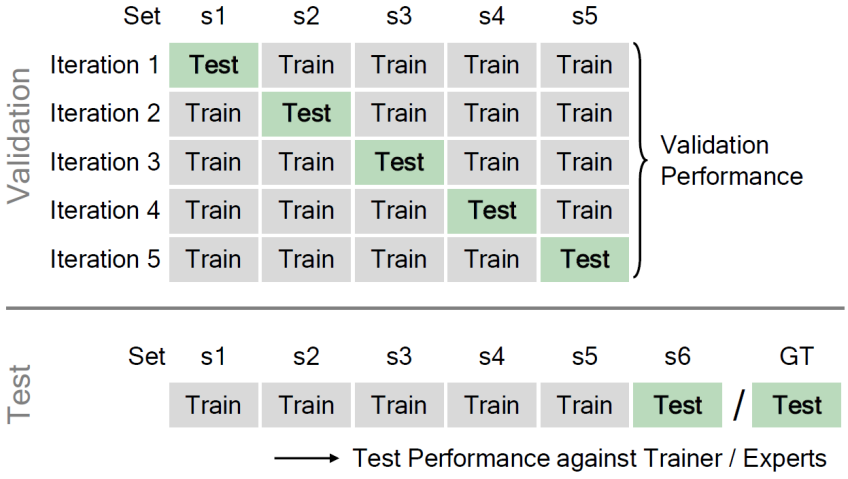


Figure 3.7: Validation and test strategy

Test
The final performance of the models is evaluated using the samples contained in the 6th set as test data. The remaining sets ($s1, s2, ..s5$) are used for training. First, the models are tested against $s6$ to report how well they imitate the original trainer. Secondly, the models are tested against the expert annotations contained in the ground truth to measure their actual performance.
As the CNN is initialized using random weights its test is repeated 10 times. Thereafter, the test result is reported twice using two different approaches: First, the performance of individual test runs is averaged and reported with a standard deviation. Secondly, the 10 repetitive runs are combined to a single model by averaging their probabilities.

3.7 Results

The optimization results of SVM and CNN are listed separately from each other. Afterwards, a consolidated visualization of the results is provided.

3.7.1 Support vector machine

The SVM's performance in dependence of the number of used features can be seen in table 4.1. The best accuracy during validation was 86.2±0.6 % and was achieved using 8 features. Testing the same configuration against trainer and experts accuracies of 86.2 % or, respectively, 86.1 % were obtained. Using fewer features does not necessarily impact the results during validation but is particularly noticeable in a decrease in test performance. More than 8 features did not significantly change validation or test results (figure 3.8).
Regardless of the number of features the model showed an imbalance between sensitivity and specificity with the latter being considerably higher. As the ground truth contains more SVT than VT episodes the balanced accuracy during testing is slightly lower (Ac_B 84.9 %).

The importance of individual features can been seen in table 3.5. Their importance was estimated by averaging the accuracies of all 16,384 combinations in which the respective feature did appear in during optimization. The most important feature is FunFreq. It achieved an average accuracy of 83.9 % and was selected in all of the top performing feature combinations. Overall features calculated in the frequency-domain performed best and provided 3 of the top 5 features.

Table 3.4: Achieved SVM performance using the best performing feature combination of each size. (Best combination marked blue.)

Number of Features	Validation			Test against Trainer ($s6$)			Test against Experts (GT)			
	Ac (%)	Se (%)	Sp (%)	Ac (%)	Se (%)	Sp (%)	Ac (%)	Se (%)	Sp (%)	Ac_B (%)
1	78.9±0.9	73.0±1.6	84.8±1.0	77.7	70.7	84.7	77.6	68.1	84.9	75.4
2	82.5±0.7	80.3±1.7	84.6±0.8	80.7	78.0	83.4	80.3	75.5	84.0	79.2
3	83.7±1.1	79.5±2.3	88.0±1.3	80.8	74.3	87.4	83.5	76.1	89.2	81.8
4	84.8±1.0	81.3±2.2	88.2±1.1	83.7	79.2	88.3	81.9	74.8	87.3	80.2
5	85.3±0.8	82.6±1.5	88.0±1.8	83.9	79.5	88.4	82.9	76.1	88.2	81.3
6	85.9±0.9	83.4±1.9	88.3±1.6	84.8	80.7	89.0	84.5	77.3	90.1	82.9
7	85.8±0.7	82.7±1.1	88.8±1.6	85.8	81.4	90.3	86.4	81.0	90.6	85.1
8	**86.2±0.6**	83.6±1.8	88.8±1.7	**86.2**	82.5	89.8	86.1	81.0	90.1	84.9
9	86.1±0.6	83.5±1.6	88.7±1.5	86.0	82.4	89.5	**86.4**	79.8	91.5	84.9
10	86.1±0.5	83.5±1.5	88.6±1.5	85.8	82.1	89.5	86.1	80.4	90.6	84.8
11	86.0±0.5	83.4±1.6	88.7±1.3	85.9	82.4	89.5	85.1	79.1	89.6	83.7
12	85.9±0.6	83.7±1.3	88.1±1.6	85.9	82.8	89.0	85.6	81.6	88.7	84.7
13	85.9±0.5	83.5±1.7	88.3±1.7	85.7	82.5	88.8	**86.4**	82.8	89.2	85.6
14	86.0±0.4	83.9±1.3	88.2±1.4	85.8	82.8	88.8	84.5	80.4	87.7	83.6
15	85.9±0.4	83.6±1.2	88.2±1.4	85.9	82.8	89.0	85.9	79.8	90.6	84.5

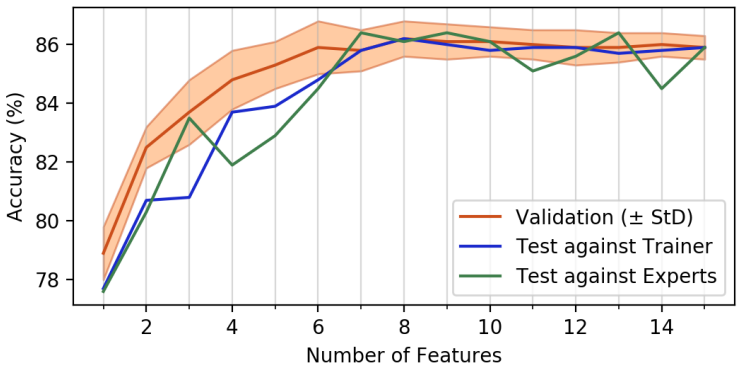
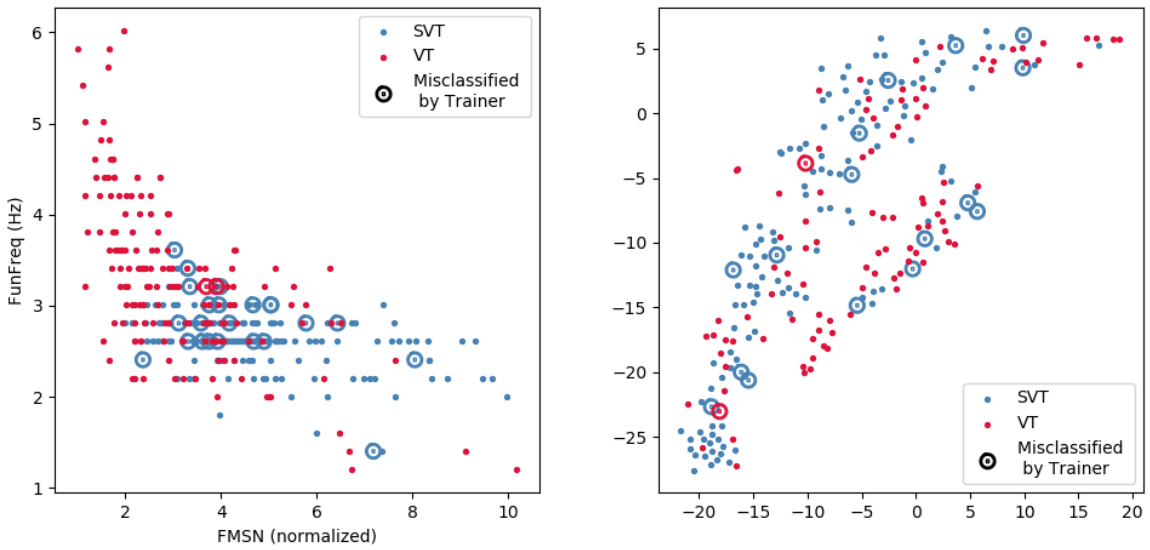


Figure 3.8: Best achievable accuracy in dependence on the number of features

Table 3.5: Importance of individual features

Feature	Avg. Val. Ac when selected (%)	Selected in best performing combination of size ...														
		1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
FunFreq	83.92	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓
Kurtosis	82.42				✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓
A3	82.17				✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓
FSMN	82.14		✓	✓		✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓
Count2	81.94					✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓
A2	81.89									✓	✓	✓		✓	✓	✓
FreqBin	81.88				✓		✓		✓	✓	✓	✓	✓	✓	✓	✓
Count3	81.80								✓	✓	✓	✓	✓	✓	✓	✓
AreaBin	81.78			✓				✓		✓	✓	✓	✓	✓	✓	✓
Complexity	81.76							✓			✓	✓	✓	✓	✓	✓
CovarBin	81.74								✓					✓	✓	✓
Count1	81.73											✓	✓	✓	✓	✓
Leakage	81.70															✓
Timedelay	81.69												✓		✓	✓
A1	81.68												✓	✓	✓	✓

Plotting the samples contained in the ground truth by using FunFreq and FSMN (best combination of two features) a rough differentiation between both classes is possible (figure 3.9a). However, a large part of the samples (including the majority of samples misclassified by the trainer) are contained within an overlapping area.
Using 8 features and t-distributed Stochastic Neighbor Embedding (t-SNE) to visualize the ground truth a clear differentiation remains difficult (figure 3.9b). Furthermore, it can been seen that a group of samples misclassified by the trainer is also likely to be misclassified according to the features.



(a) 2 features

(b) 8 featues and t-SNE

Figure 3.9: Visualization of samples contained in the ground truth. **(a)** Samples plotted using the best performing features in the feature combination of size 2, **(b)** Samples visualized using t-SNE and the features contained in the best performing feature combination of size 8.

3.7.2 Convolutional neural network

Table 3.6: Performance of CNN in dependence on the input length and the network's size. (Best combination marked blue.)

(a) Test performance obtained by averaging the performance of 10 repetitive runs.

Configuration		**Validation**			**Test against Trainer** ($s6$)			**Test against Experts** (GT)			
Input (s)	**Size (n)**	**Ac (%)**	**Se (%)**	**Sp (%)**	**Ac (%)**	**Se (%)**	**Sp (%)**	**Ac (%)**	**Se (%)**	**Sp (%)**	**Ac$_B$ (%)**
1	5	88.5±0.7	86.9±3.7	90.2±3.4	88.7±1.2	89.7±2.1	87.6±3.8	91.6±1.5	95.3±2.5	88.7±3.8	92.5±1.2
	10	89.8±0.8	88.1±3.2	91.5±3.0	89.0±0.5	89.1±4.2	88.9±4.1	91.7±1.6	94.7±2.5	89.4±4.5	92.4±0.8
	15 †										
2	5	89.5±1.0	90.7±1.8	88.4±2.9	89.5±1.2	89.8±2.1	89.1±3.7	91.6±2.1	94.2±1.9	89.5±4.6	92.2±1.4
	10	90.6±1.0	87.0±3.5	94.2±1.6	89.6±1.6	90.5±2.3	88.7±5.0	91.4±2.6	94.0±2.7	89.4±5.7	92.0±1.9
	15	89.7±2.4	89.5±2.9	89.8±6.2	90.4±0.6	87.6±2.7	93.2±3.0	93.0±1.3	92.0±2.1	93.8±2.7	92.8±1.1
4	5	89.0±1.6	90.1±3.1	88.0±3.8	**90.9±0.7**	90.9±1.8	90.9±2.5	92.2±1.5	94.4±1.6	90.5±2.6	92.7±1.2
	10	**91.1±1.1**	89.3±2.6	93.0±1.7	90.4±1.1	88.9±2.4	91.9±3.2	**93.1±1.3**	93.3±2.1	93.1±2.2	93.2±1.0
	15	90.5±0.9	88.7±2.3	92.3±1.8	89.5±1.9	88.3±3.2	90.6±6.2	91.4±3.3	93.4±2.4	89.8±6.7	91.8±2.4

(b) Test performance obtained by combining 10 repetitive test runs to an ensemble by averaging their probabilities.

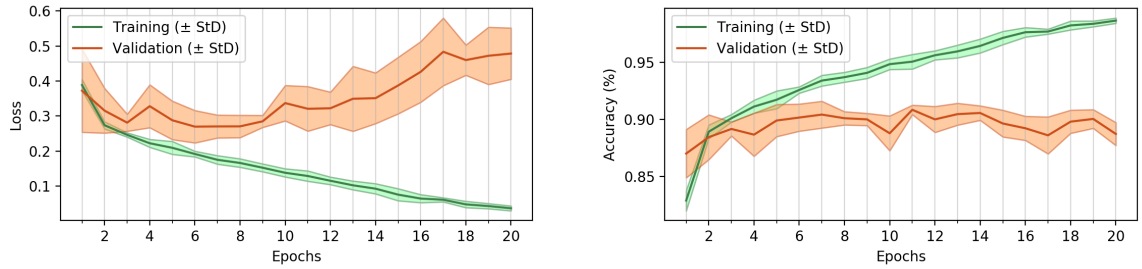
Configuration		**Validation**			**Test against Trainer** ($s6$)			**Test against Experts** (GT)			
Input (s)	**Size (n)**	**Ac (%)**	**Se (%)**	**Sp (%)**	**Ac (%)**	**Se (%)**	**Sp (%)**	**Ac (%)**	**Se (%)**	**Sp (%)**	**Ac$_B$ (%)**
1	5	88.5±0.7	86.9±3.7	90.2±3.4	90.3	90.7	89.9	92.3	95.7	89.6	92.7
	10	89.8±0.8	88.1±3.2	91.5±3.0	90.3	90.2	90.5	93.9	96.9	91.5	94.2
	15 †										
2	5	89.5±1.0	90.7±1.8	88.4±2.9	90.7	90.8	90.6	92.8	95.1	91.0	93.1
	10	90.6±1.0	87.0±3.5	94.2±1.6	91.2	90.7	91.7	93.3	95.1	92.0	93.5
	15	89.7±2.4	89.5±2.9	89.8±6.2	91.6	88.1	95.0	94.7	92.6	96.2	94.4
4	5	89.0±1.6	90.1±3.1	88.0±3.8	92.1	91.7	92.4	92.8	95.7	90.6	93.1
	10	**91.1±1.1**	89.3±2.6	93.0±1.7	91.9	89.5	94.4	**94.9**	93.9	95.8	94.8
	15	90.5±0.9	88.7±2.3	92.3±1.8	91.3	89.1	93.4	93.9	95.7	92.5	94.1

† Due to the pooling layers the input is required to have a length from at least 256 measuring points (not contained in 1 second) to be processed by the full-sized network.

The performance of the CNN in dependence to its size and received input is shown in table 3.6a and 3.6b. Table 3.6a shows the average test performance measured over 10 test runs of the respective CNN. For table 3.6b the test runs were combined to a new model by averaging their predictions.
The top performing CNN configuration used an input window of 4 seconds, had n set to 10 (23 convolutional layers in 11 residual blocks) and achieved a validation accuracy of 91.1±1.1 %. The average test run of this configuration obtained an accuracy of 90.4±1.1 % when being tested against its trainer and 93.1±1.3 % when being tested against the experts. By combining all test runs in an ensemble the accuracy was further improved to 91.9 % (trainer) and 94.9 % (experts).
When given a shorter input of only 1 second the CNN ensemble still managed to obtain a test performance of up to 93.9 %.[9]
During validation and when being tested against the trainer an imbalance between sensitivity and specificity was observed with the latter being slightly higher.

All CNN configurations based on a 1 or 2 second input performed best when trained using a batch size of 32. CNNs using 4 seconds performed best with a batch size of 64. Most of the configurations obtained the best validation loss between 7 and 12 epochs. Non of the models required more than 17 epochs.

[9] As explained in 3.3 the CNNs receive a randomly chosen snippet of each sample. In this context it is important to mention that all repeated test runs of a CNN always receive the same (randomly chosen) snippet.



(a) Loss

(b) Accuracy

Figure 3.10: Loss and accuracy through epochs during training of the best performing CNN (4 s input, size 10)

3.7.3 Consolidated

The terms SVM and CNN will from now on refer to the previously described, best performing version of the respective classifier.

According to the ground truth the trainer obtains an accuracy of 93.3 % with 98.8 % sensitivity and 89.2 % specificity. Figure 3.11 shows that approximately 40 % of the samples wrongly classified by the trainer were also misclassified by each of the models. Furthermore, only 6 of the 19 samples misclassified by the CNN were labeled differently by SVM.
Figure 3.12 visualizes the samples contained in the ground truth using the probabilities calculated by the SVM or, respectively, the CNN. It shows that the two positive samples missed by the trainer were also not easy to classify for the models. None of the models managed to classify both correctly.

The receiver operating characteristic curve (ROC curve) in figure 3.13 visualizes the trade-off between sensitivity and specificity. It shows that the CNN's decision threshold could be adjusted to equal the trainier's sensitivity while maintaining a decent specificity. For instance, a sensitivity of 97.5 % with 88.2 % specificity and an overall accuracy of 92.9 % could be achieved by lowering the threshold. In contrast, adjusting the SVM's threshold to exceed a sensitivity of 95 % would lower its specificity to approximately 50 %.

Reviewing individual samples showed that the majority of wrongly classified ECGs contained constant atrial and ventricular activity with 1:1 conduction (3.14a). All classifiers (including the trainer) frequently misclassified this type of ECG. Although the CNN was best in distinguishing these rhythms they accounted for the majority of its misclassifications. Many samples solely misclassified by the models showed constant atrial and ventricular activity but without 1:1 conduction (3.14b). ECGs solely misclassified by the trainer often contained a weak RA signal.

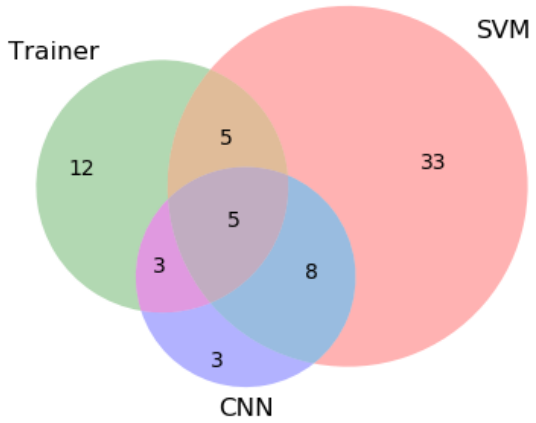


Figure 3.11: Number of misclassified samples in the ground truth in dependence on the used classifier (trainer, SVM, CNN). (Total number of samples: 375)

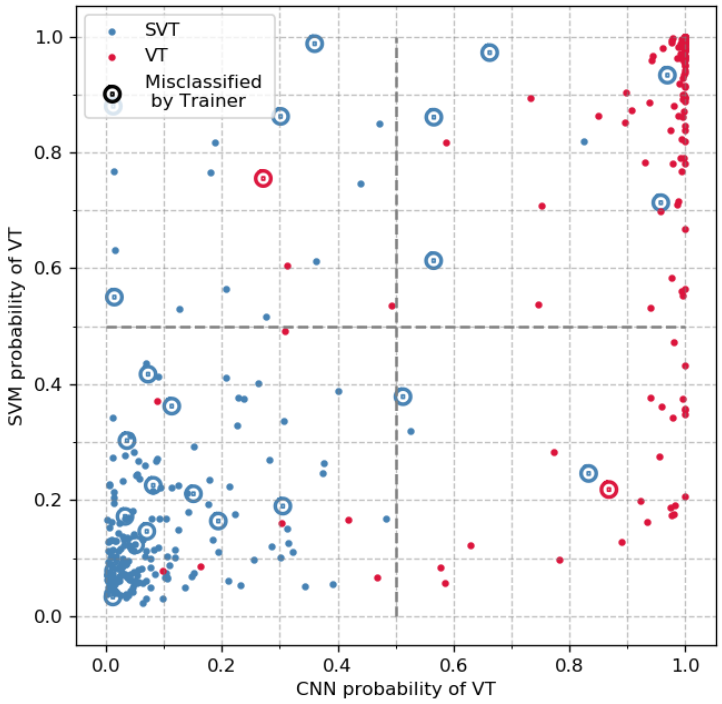


Figure 3.12: Visualization of samples contained in the ground truth by their estimated probability for being VT. Both SVM and CNN use a classification threshold of 0.5 (dotted line).

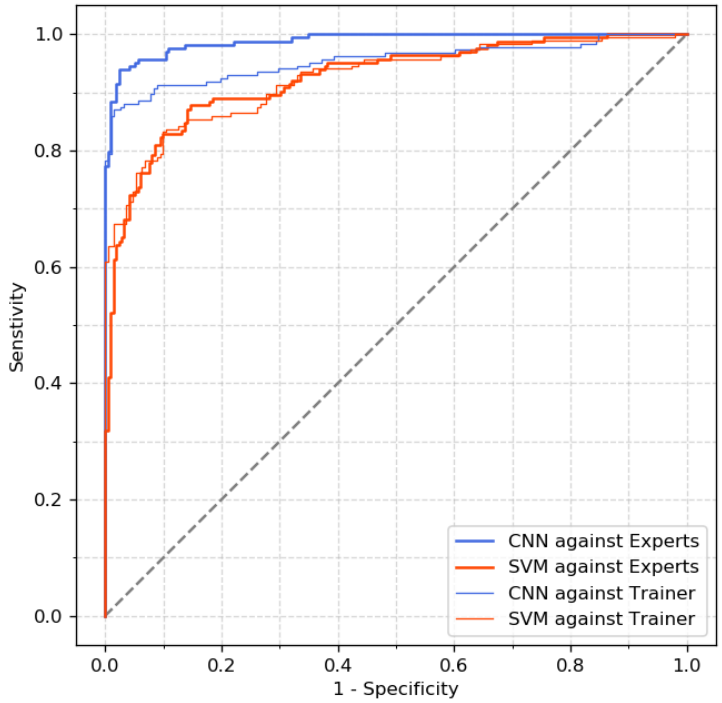
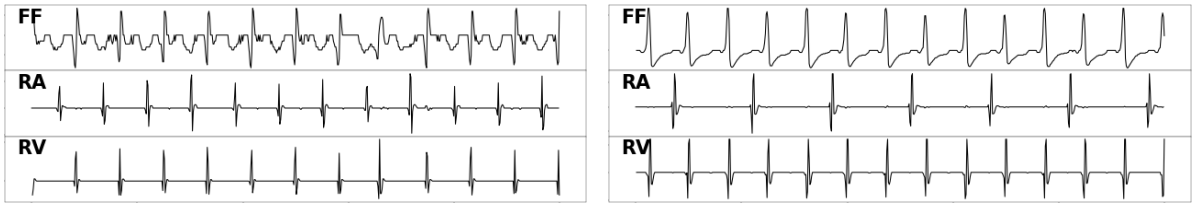


Figure 3.13: ROC curve of classifiers based on the samples in the ground truth



(a) **(b)**

Figure 3.14: Misclassified ECGs: **(a)** SVT misclassified as VT by trainer and models, **(b)** VT misclassified as SVT by the CNN. (FF: far-field, RA: right atrium, RV right ventricle, X-axis: Time (5 seconds), Y-axis: Voltage)

Chapter 4

Discussion

4.1 Infrastructure

The developed infrastructure proved to be reliable, fast and functional during the synthetic tests and, more importantly, throughout the usage within the second part of the thesis.

The synthetic tests showed that the data can be stored in a storage efficient, quickly accessible manner. 10,000 locally stored ECGs were loaded in under 30 seconds and took up less than 1 Gb of disk space. Assuming the deserialization process scales linearly the desired throughput of one million messages per hour is met.
When implementing the classification models the designed API proved to be convenient. This was particularly noticeable when implementing the CNN. Larger parts of the CNN's source-code were found online and running first tests on it was possible within a day. Furthermore, the access time's importance became clear as both models required several hundred test runs when being implemented and optimized.

Comparing the new infrastructure to the existing one highlights the different use cases each of them addressed. The existing infrastructure is heavily focused on reliability and is certified for medical usage. However, its API is rather impractical and comparatively slow. In contrast, the new infrastructure is solely intended to be used for research purposes. Using it the data can be processed in a vast range of programming languages and approximately 120 times faster without any locational restrictions. However, it utilizes third-party software components (e.g. Protocol Buffers or Gzip) and, therefore, is presumably harder to validate for medical use.

Maintaining the Converter is expected to be the greatest difficulty during a long-term use of the infrastructure. Many parts of the program (e.g. database configurations) can easily be configured using the GUI. Moreover, attention has been paid to meaningfully structure the program. However, exchanging parts of the Converter or adding new parameters to the files requires understanding of the existing infrastructure, Java and the employed serialization formats. To improve the maintainability it should be considered to discontinue supporting FlatBuffers. Protocol Buffers offers a similar (partly faster) performance and is easier to handle especially when implementing the serialization.

Apart from discontinuing FlatBuffers further improvements should address the file's structure and the API's features. Testing revealed that decoding ECG-markers could significantly increase the access time. To overcome this shortcoming the markers' contents should be stored using enumerated data types instead of strings. The Python API could be improved further by adding parallelization, as well as a mechanism that automatically combines or synchronizes locally and remotely stored files.

4.2 Classification

Table 4.1: Performance overview of classifiers

Classifier	**Test against Trainer** (2,372 samples)			**Test against Experts** (375 samples)			
	Ac (%)	**Se (%)**	**Sp (%)**	**Ac (%)**	**Se (%)**	**Sp (%)**	**Ac$_B$ (%)**
SVM + 8 features	86.2	82.5	89.8	86.1	81.0	90.1	84.9
CNN ensemble + 1 s input	90.3	90.2	90.5	93.9	96.9	91.5	94.2
CNN ensemble + 4 s input	91.9	89.5	94.4	94.9	93.9	95.8	94.8
Trainer	100.0	100.0	100.0	93.3	98.8	89.2	94.0

Both SVM and CNN obtained satisfactory results and suggest that classification methods proposed for surface ECGs can be applied to intracardiac signals. The CNN even demonstrated that distinguishing SVT and VT rhythms is possible using a shorter window of the ECG and less channels than conventional distinction algorithms.

During optimization of the SVM a combination containing 8 of the 15 features were found to achieve the best results. It was expected that using all features would not necessarily increase the performance. The metrics were gathered from multiple sources and it was highly likely that some of them would carry redundant information. The most important feature was the self-proposed FunFreq metric. It provided a reliable estimate of the heart frequency which is a good first indicator when distinguishing the rhythms. The imbalance between sensitivity and specificity when testing the SVM against the trainer was expected considering the trainer's own imbalance. However, the imbalance remained when testing the model against experts. The analysis of misclassified samples revealed that many of them contained constant, synchronous atrial and ventricular activity. In most cases this type of ECG will correspond to an SVT. Therefore, one possible explanation for the poor sensitivity could simply be the SVM's inability to distinguishing these type of rhythms and, thus, labeling them as SVT.

The CNN received 4 seconds of the raw ECG signal and slightly exceeded the accuracy of its trainer. This is surprising as the trainer had the advantage of receiving atrial and ventricular activity separately from one another and further used a longer segment of the signal. Even when given only 1 second of the signal the CNN equaled the accuracy of its trainer which is remarkable as conventional SVT/VT detection algorithms heavily rely on analyzing the chronology of the ECG. Both configurations (1 second and 4 seconds) could greatly benefit from averaging the predictions of repetitive test runs. Considering the high variance between individual test runs this is plausible.

When comparing SVM and CNN to one another the CNN clearly is superior in terms of classification performance. However, the SVM has the advantage of being easier to interpret. Individual features have predictable outcomes for different rhythm types. Understanding an algorithm's decision-making is not only important when validating medical applications but also helps to develope new insights into the data.
Comparing both approaches to the papers they were originally based on is only possible to a limited extent. Li's work (SVM) was aimed at improving AEDs and obtained an accuracy of 96.3$\pm$3.4 %. However, AEDs are applied to individuals who already went unconscious. To decide whether or not a rhythm is responsible for making someone pass out can be considered an easier task compared to early SVT/VT differentiation. In contrast, Hannun (CNN) distinguished the ECGs into 14 different classes with some of them quite similar to one another and achieved an average sensitivity of 0.78 %. Considering these differences the results achieved by the models in this thesis meet the expectations.
Another important comparison is the one to the trainer. At this moment in time the trainer's performance corresponds to the performance of a state of the art classifier designed to differentiate between SVT and VT. Therefore, equaling its performance while requiring a smaller part of the ECG can be considered a success.

One major limitation when discussing the models and their results is the annotation accuracy of the training data. Some ECGs were misclassified by the trainer as well as the models. For this samples it remains unclear whether the models' failure was caused by the flawed trainer or by the ECG's characteristics itself. Furthermore, some of the samples solely misclassified by the models also contained patterns similar to those in ECGs frequently misclassified by the trainer. Hence, the actual impact of faulty training samples cannot be determined.
Another limitation is the size of the ground truth. Although its samples were chosen randomly and there is no reason to conclude that it is not representative of the whole population it should be enlarged. This is especially interesting given the almost equal performance of the trainer and the CNN.

Further improvements should primarily address the annotation accuracy of the training data. Another promising strategy is the combination of both classifiers or of different SVM and CNN configurations. This step could further be refined by varying the training set of each model in the ensemble or by testing more sophisticated ways of combining their outputs. Moreover, some hyperparameters including the CNN's learning rate or the SVM's kernel coefficient γ and penalty parameter C have not been optimized yet.

4.3 Conclusion

The first aim of this thesis was to implement an infrastructure that enables research on the data received from cardiac implants. Secondly, it was examined to what extent classifiers developed for surface ECGs can be used to evaluate intracardiac signals.

The newly implemented infrastructure fulfilled all its requirements. It is fast, network-independent and scalable. Moreover, it supports a vast range of programming languages and anonymization. It demonstrated the potential and the importance of fitting the design of a system to its intended use-case. Although the new infrastructure is no substitution for the existing one it surely is an useful addition.

The two classifiers tested in the second part of the thesis were neither designed for intracadiac ECGs nor for SVT/VT distinction. Both remained largely unchanged and managed to achieve satisfactory results. Unexpectedly the CNN even equaled the performance of its trainer despite the fact it used a shorter window of the ECG and less channels. However, the approach based on feature engineering has the advantage of being easier to interpret which is important when validating medical applications. One major limitation when testing the models was the suboptimal annotation accuracy of the training data.

The tested classifiers showed not only that research done on surface ECGs is transferable to intracardiac signals but also demonstrated the importance of the new infrastructure which made testing them possible in the first place.

4.4 Outlook

Is it possible to distinguish SVT and VT rhythms by using only the far-field signal? Yes, it is - in 19 of 20 cases. Although the employed methods might not be the final or most practical way of classifying the rhythms they certainly answered the question. A question that is relevant for new devices with designs that avoid the insertion of leads into the patient's heart like subcutaneous ICDs or implantable heart monitors.

Moreover, the question addressed in this thesis was only one of many. Other topics of interest that could benefit from machine learning include the detection of lead fracture, noise, other rhythms like atrial fibrillation or the early diagnosis of inflammation and lead dislodgement. Finding new answers to these issues holds great potential in further improving the patients' quality of life.

Abbreviations

AED	Automated External Defibrillator
AF	Atrial Fibrillation
API	Application Programming Interface
ATP	Antitachycardia Pacing
AV	Atrioventricular
BLOB	Binary Large Object
bpm	Beats Per Minute (heart rate)
CLI	Command-Line Interface
CNN	Convolutional Neural Network
DDI	Diagnostic Data Interpreter
ECG	Electrocardiogram
FF	Far-Field
GT	Ground Truth
GUI	Graphical User Interface
ICD	Implantable Cardioverter-Defibrillator
IDL	Interface Definition Language
MLP	Multilayer Perceptron
RA	Right Atrium
RNN	Recurrent Neural Network
RV	Right Ventricle
STFT	Short-Time Fourier Transform
SVM	Support Vector Machine
SVT	Supraventricular Tachycardia
VF	Ventricular Fibrillation
VT	Ventricular Tachycardia

List of Figures

List of Tables

References

[1] Anton Amann, Robert Tratnig, and Karl Unterkofler. "Detecting ventricular fibrillation by time-delay methods". In: *IEEE Transactions on Biomedical Engineering* 54.1 (2007), pp. 174–177.

[2] Anton Amann, Robert Tratnig, and Karl Unterkofler. "Reliability of old and new ventricular fibrillation detection algorithms for automated external defibrillators". In: *Biomedical engineering online* 4.1 (2005), p. 60.

[3] Fernando Andreotti et al. "Comparing feature-based classifiers and convolutional neural networks to detect arrhythmia from short segments of ECG". In: *2017 Computing in Cardiology (CinC)*. IEEE. 2017, pp. 1–4.

[4] *Apache Thrift Language Support*. URL: `https://thrift.apache.org/docs/Languages` (visited on 03/03/2019).

[5] S.S. Barold, R.X. Stroobandt, and A.F. Sinnaeve. *Cardiac Pacemakers and Resynchronization Step by Step: An Illustrated Guide*. Wiley, 2010. ISBN: 9781444396164.

[6] S Barro et al. "Algorithmic sequential decision-making in the frequency domain for life threatening ventricular arrhythmias and imitative artefacts: a diagnostic system". In: *Journal of biomedical engineering* 11.4 (1989), pp. 320–328.

[7] J.G. Betts et al. *Anatomy & Physiology*. Open Textbooks. OpenStax College, Rice University, 2013. ISBN: 9781938168130.

[8] Ahmet Kaya Bilge et al. "Depression and anxiety status of patients with implantable cardioverter defibrillator and precipitating factors". In: *Pacing and Clinical Electrophysiology* 29.6 (2006), pp. 619–626.

[9] Guangyu Bin et al. "Detection of atrial fibrillation using decision tree ensemble". In: *2017 Computing in Cardiology (CinC)*. IEEE. 2017, pp. 1–4.

[10] BIOTRONIK SE & Co. KG. "CardioMessenger Smart - Technical Manual (revision H)". In: (2017).

[11] Mark L Brown and Charles D Swerdlow. "Sensing and detection in Medtronic implantable cardioverter defibrillators". In: *Herzschrittmachertherapie+ Elektrophysiologie* 27.3 (2016), pp. 193–212.

[12] Thomas Brüggemann et al. "Tachycardia detection in modern implantable cardioverter–defibrillators". In: *Herzschrittmachertherapie+ Elektrophysiologie* 27.3 (2016), pp. 171–185.

[13] *BSON Libraries*. URL: `http://bsonspec.org/implementations.html` (visited on 03/03/2019).

[14] AJMM Camm et al. "Heart rate variability: standards of measurement, physiological interpretation and clinical use. Task Force of the European Society of Cardiology and the North American Society of Pacing and Electrophysiology". In: *Circulation* 93.5 (1996), pp. 1043–1065.

[15] Rhanderson N Cardoso et al. "ICD discrimination of SVT versus VT with 1: 1 VA conduction: A review of the literature". In: *indian pacing and electrophysiology journal* 15.5 (2015), pp. 236–244.

[16] Federica Censi et al. "On the resolution of ECG acquisition systems for the reliable analysis of the P-wave". In: *Physiological measurement* 33.2 (2012), N11.

[17] S Chen, Nitish V Thakor, and Morton Maimon Mower. "Ventricular fibrillation detection by a regression test on the autocorrelation function". In: *Medical and Biological Engineering and Computing* 25.3 (1987), pp. 241–249.

[18] Gari D Clifford et al. "AF Classification from a short single lead ECG recording: the PhysioNet/Computing in Cardiology Challenge 2017". In: *2017 Computing in Cardiology (CinC)*. IEEE. 2017, pp. 1–4.

[19] *C-Support Vector Classification*. URL: `https://scikit-learn.org/stable/modules/generated/sklearn.svm.SVC.html` (visited on 06/11/2019).

[20] Jean-Claude Daubert et al. “A randomized trial of long-term remote monitoring of pacemaker recipients (The COMPAS trial)”. In: *European Heart Journal* 33.9 (Nov. 2011), pp. 1105–1111. ISSN: 0195-668X.
[21] FPL. *Platform / Language / Feature support*. URL: https://google.github.io/flatbuffers/flatbuffers_support.html (visited on 03/03/2019).
[22] Anil K Gehi, Davendra Mehta, and J Anthony Gomes. “Evaluation and management of patients after implantable cardioverter-defibrillator shock”. In: *JAMA* 296.23 (2006), pp. 2839–2847.
[23] lya Grigorik. *Protocol Buffers, Avro, Thrift MessagePack*. URL: https://www.igvita.com/2011/08/01/protocol-buffers-avro-thrift-messagepack/ (visited on 03/04/2019).
[24] Milton E Guevara-Valdivia and Pedro Iturralde Torres. “Remote monitoring of implantable pacemaker, cardioverter defibrillator, and cardiac resynchronizer”. In: *Modern Pacemakers-Present and Future*. InTech, 2011.
[25] Awni Y Hannun et al. “Cardiologist-level arrhythmia detection and classification in ambulatory electrocardiograms using a deep neural network”. In: *Nature medicine* 25.1 (2019), p. 65.
[26] Runnan He et al. “Automatic detection of atrial fibrillation based on continuous wavelet transform and 2d convolutional neural networks”. In: *Frontiers in physiology* 9 (2018), p. 1206.
[27] Shenda Hong et al. “ENCASE: An ENsemble ClASsifiEr for ECG classification using expert features and deep neural networks”. In: *2017 Computing in Cardiology (CinC)*. IEEE. 2017, pp. 1–4.
[28] *Induction of VF by T-wave shock*. URL: https://www.cardiocases.com/en/pacingdefibril%20lation/traces/icd/boston-scientific/induction-vf-t-wave-shock (visited on 02/20/2019).
[29] Irena Jekova. “Comparison of five algorithms for the detection of ventricular fibrillation from the surface ECG”. In: *Physiological measurement* 21.4 (2000), p. 429.
[30] Irena Jekova. “Shock advisory tool: Detection of life-threatening cardiac arrhythmias and shock success prediction by means of a common parameter set”. In: *Biomedical Signal Processing and Control* 2.1 (2007), pp. 25–33.
[31] Irena Jekova and Vessela Krasteva. “Real time detection of ventricular fibrillation and tachycardia”. In: *Physiological measurement* 25.5 (2004), p. 1167.
[32] Tae Joon Jun et al. “ECG arrhythmia classification using a 2-D convolutional neural network”. In: *arXiv preprint arXiv:1804.06812* (2018).
[33] S Kuo. “Computer detection of ventricular fibrillation”. In: *Proc. of Computers in Cardiology, IEEE Comupter Society* (1978), pp. 347–349.
[34] Qiao Li, Roger G Mark, and Gari D Clifford. “Robust heart rate estimation from multiple asynchronous noisy sources using signal quality indices and a Kalman filter”. In: *Physiological measurement* 29.1 (2007), p. 15.
[35] Qiao Li, Cadathur Rajagopalan, and Gari D Clifford. “Ventricular fibrillation and tachycardia classification using a machine learning approach”. In: *IEEE Transactions on Biomedical Engineering* 61.6 (2013), pp. 1607–1613.
[36] Yuhui Lin. *Data Serialization: JSON, BSON, MessengePack, Protocol Buffer, Thrift, Avro, Cap’n Proto, FlatBuffers*. URL: https://yuhui-lin.github.io/blog/2017/08/01/serialization (visited on 03/04/2019).
[37] Kazuaki Maeda. “Performance evaluation of object serialization libraries in XML, JSON and binary formats”. In: (2012), pp. 177–182.
[38] *Message Pack*. URL: https://msgpack.org/ (visited on 03/03/2019).
[39] Kei-ichiro Minami, Hiroshi Nakajima, and Takeshi Toyoshima. “Real-time discrimination of ventricular tachyarrhythmia with Fourier-transform neural network”. In: *IEEE transactions on Biomedical Engineering* 46.2 (1999), pp. 179–185.
[40] Alan Murray, Ronald WF Campbell, and Desmond G Julian. “Characteristics of the ventricular fibrillation waveform”. In: *Proc. Computers in Cardiology* (1985), pp. 275–278.
[41] Ikenna Odinaka et al. “ECG biometric recognition: A comparative analysis”. In: *IEEE Transactions on Information Forensics and Security* 7.6 (2012), pp. 1812–1824.
[42] Bo Søborg Petersen et al. “Smart grid serialization comparison”. In: (2017).
[43] Jim Pivarski. *Overview of Serialization Technologies*. URL: https://indico.cern.ch/event/658060/contributions/2898569/attachments/1622526/2582399/pivarski%20-serialization.pdf (visited on 03/04/2019).
[44] Pranav Rajpurkar et al. “Cardiologist-level arrhythmia detection with convolutional neural networks”. In: *arXiv preprint arXiv:1707.01836* (2017).

[45] Paruj Ratanaworabhan, Jian Ke, and Martin Burtscher. “Fast lossless compression of scientific floating-point data”. In: (2006), pp. 133–142.
[46] Samuel F Sears, Melissa Matchett, and Jamie B Conti. “Effective management of ICD patient psychosocial issues and patient critical events”. In: *Journal of cardiovascular electrophysiology* 20.11 (2009), pp. 1297–1304.
[47] R.X. Stroobandt, S.S. Barold, and A.F. Sinnaeve. *Implantable Cardioverter - Defibrillators Step by Step: An Illustrated Guide*. Wiley, 2011. ISBN: 9781444359060.
[48] *Supported Languages*. URL: https://cwiki.apache.org/confluence/display/AVRO/Supported+Languages (visited on 03/03/2019).
[49] Michael O Sweeney. “Antitachycardia pacing for ventricular tachycardia using implantable cardioverter defibrillators: substrates, methods, and clinical experience”. In: *Pacing and clinical electrophysiology* 27.9 (2004), pp. 1292–1305.
[50] Charles D Swerdlow et al. “Troubleshooting implanted cardioverter defibrillator sensing problems I”. In: *Circulation: Arrhythmia and Electrophysiology* 7.6 (2014), pp. 1237–1261.
[51] Tomás Teijeiro et al. “Arrhythmia classification from the abductive interpretation of short single-lead ECG records”. In: *2017 Computing in Cardiology (CinC)*. IEEE. 2017, pp. 1–4.
[52] *Third-Party Add-ons for Protocol Buffers*. URL: https://github.com/protocolbuffers/protobuf/blob/master/docs/third_party.md (visited on 03/03/2019).
[53] Jan Vanura and Pavel Kriz. “Perfomance Evaluation of Java, JavaScript and PHP Serialization Libraries for XML, JSON and Binary Formats”. In: (2018), pp. 166–175.
[54] Kenton Varda. *Other Languages*. URL: https://capnproto.org/otherlang.html (visited on 03/03/2019).
[55] Lauren D Vazquez et al. “Sexual health for patients with an implantable cardioverter defibrillator”. In: *Circulation* 122.13 (2010), e465–e467.
[56] Zhaohan Xiong, Martin K Stiles, and Jichao Zhao. “Robust ECG signal classification for detection of atrial fibrillation using a novel neural network”. In: *2017 Computing in Cardiology (CinC)*. IEEE. 2017, pp. 1–4.
[57] Morteza Zabihi et al. “Detection of atrial fibrillation in ECG hand-held devices using a random forest classifier”. In: *2017 Computing in Cardiology (CinC)*. IEEE. 2017, pp. 1–4.
[58] Norbert Zanker et al. “Tachycardia detection in ICDs by Boston Scientific”. In: *Herzschrittmachertherapie+ Elektrophysiologie* 27.3 (2016), pp. 186–192.
[59] Jan Zdarek and Carsten W Israel. “Detection and discrimination of tachycardia in ICDs manufactured by St. Jude Medical”. In: *Herzschrittmachertherapie+ Elektrophysiologie* 27.3 (2016), pp. 226–239.
[60] Xu-Sheng Zhang et al. “Detecting ventricular tachycardia and fibrillation by complexity measure”. In: *IEEE Transactions on biomedical engineering* 46.5 (1999), pp. 548–555.
[61] Martin Zihlmann, Dmytro Perekrestenko, and Michael Tschannen. “Convolutional recurrent neural networks for electrocardiogram classification”. In: *2017 Computing in Cardiology (CinC)*. IEEE. 2017, pp. 1–4.
[62] *zlib - Compression compatible with gzip*. URL: https://docs.python.org/3/library/zlib.html#zlib.decompress (visited on 03/06/2019).

Acknowledgements

I would like to thank all those who supported this thesis. Furthermore, I want to express my particular gratitude towards:

Prof. Dr. Tim Landgraf. Without hesitation he took his time and supervised this work. Furthermore, he provided me with suggestions that proved to be very valuable for the results of this thesis.

Barry Linnert. Since my Bachelor's thesis he supported me and had an open door whenever I encountered problems. Moreover, he proofread my theses and helped me to improve my academic writing.

Dr. Peter Liebisch. Without him offering his support and guidance this thesis would never been possible in the first place. He was always available for questions and valuable discussions.

Jens Knüppel. Over several hours he introduced me into the existing, historically grown infrastructure. Furthermore, he granted me insights into implants and their technical characteristics. ...and insights into Oracle databases and their technical characteristics.

Dr. Michael Hofmann, *El Patrón*. Since my first day at The-Company-Which-Must-Not-Be-Named he always encouraged me to follow my interests in cardiac physiology and therapy. He generously funded this thesis.

René Fischer. He provided me with data which was highly important for the test set.

Ercan Dietzfelbinger. He proofread this work and helped me to create a thesis worthy of a Shakespeare.[1] Even more importantly he is one of my closest friends and learning from him was indeed a delight.

Julian Petrasch. He not only supported this thesis but was without any doubt my most important companion during school days and studies. Let's see what comes up next, my great friend!

Kain Gontarska and Mehmed Halilovic. My friends and fellow sufferers in the times of writing a Master's thesis that are full of privations.

My friends, colleagues and neighbours (including Hannes). They helped me to relax every now and then and did a great job in cheering me up.

My family - from whom I always receive unconditional support and encouragement. And food.

[1] Observant readers may have noticed a touch of irony and exaggeration. A touch.

Appendix

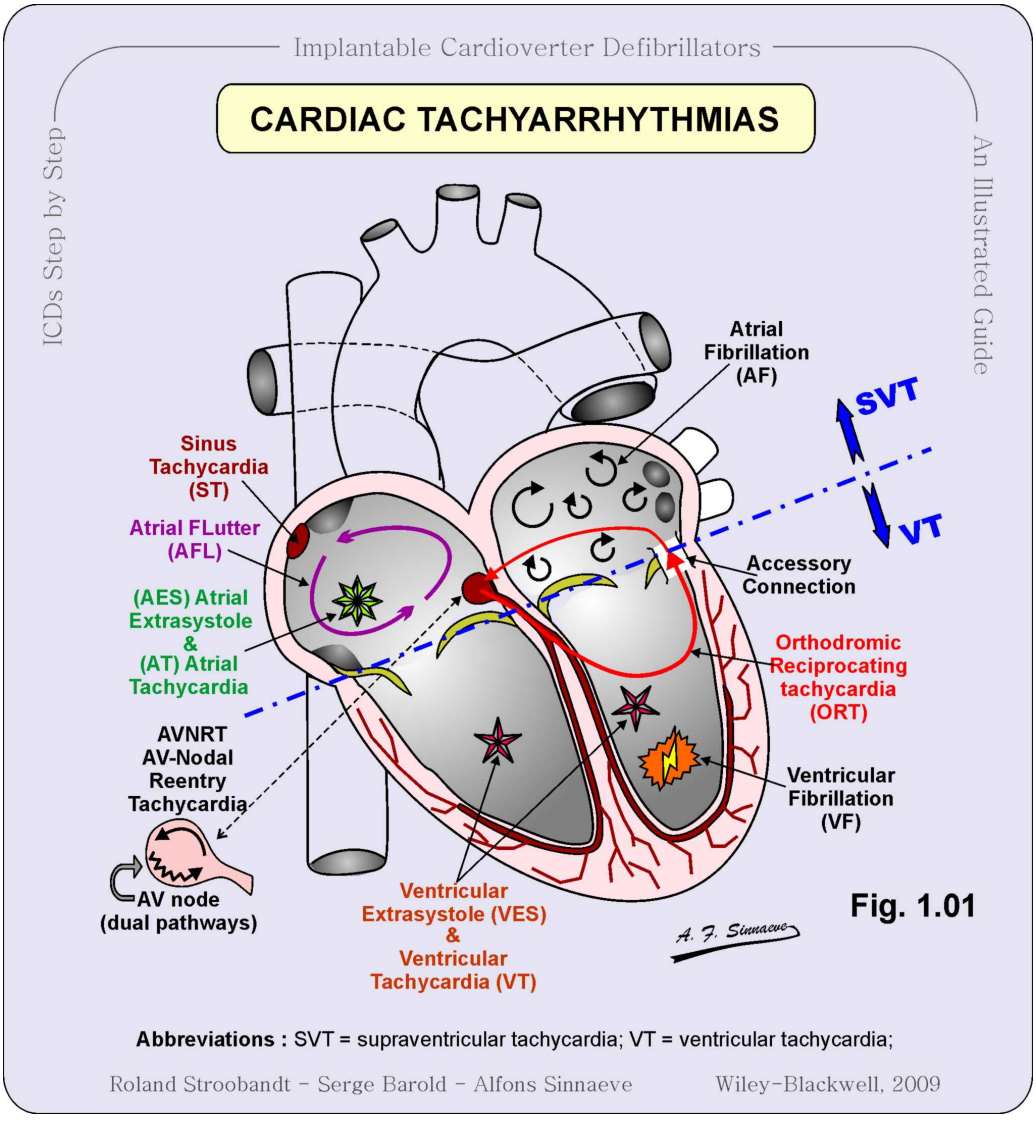


Figure 5.1: Visualization of different tachycardia origins [47]

Table 5.1: Components of the Converter's implementation

package → name (type)	**Lines of Code**
Short description	
main → Main (class)	7
Entry point of the converter	
ui.cli → SimpleCLI (class)	67
Defines the CLI and initializes the DynamicOptions and the Manager	
ui.fx → MainFX (class)	32
Initializes the GUI, the DynamicOptions and the Manager	
ui.fx → CacheManagerControllerFX (class)	461
Links the GUI to the program's logic / Manager	
ui.fx → CacheMangerGUI (fxml)	274
Defines the GUI's appearance (created with SceneBuilder)	
options → StaticOptions (class)	33
Holds several static constants	
options → DynamicOptions (class)	307
Reads and writes the settings defined by the user	
serialization.protobuf → EcgProtoBuilder (class)	209
Converts episode-objects to a Protocol Buffers file	
serialization.flatbuffers → EcgFlatbuffersBuilder (class)	250
Converts episode-objects to a FlatBuffers file	
serialization.util → SerialObfuscator (class)	31
Obfuscates serial numbers by hashing	
serialization.util → Gzip (class)	36
Compresses files with Gzip	
manager → Manager (interface)	24
Defines the communication between the Manager (program's logic) and the CLI/GUI	
manager.nsmanager → NotSequentialManager (class)	89
Manages the Feeder, the DecoderTheads(s) and the Delivere(s)	
manager.nsmanager → Job (class)	19
Contains a set of message ids respectively converted messages	
manager.nsmanager → DecoderThread (class)	221
Processes a job by fetching messages and converting their content	
manager.nsmanager → DecodedMessage (class)	106
Contains the converted episodes	
manager.nsmanager → StaticticsSetImpl (class)	104
Used to update the CLI/GUI with the program's progress	

manager.nsmanager.database → DatabaseHandle (abstract class)	20
Provides basic functionalities needed to access a database	

manager.nsmanager.decoder → DecoderClientHandle (class)	21
Provides access to the DDI	

manager.nsmanager.feeder → Feeder (interface)	10
Defines the functionalities of a Feeder-object (used to create new jobs)	
manager.nsmanager.feeder → DatabaseFeeder (class)	93
Reads ids from a database table to create new jobs	
manager.nsmanager.feeder → FileFeeder (class)	106
Reads ids from a file on the local machine to create new jobs	

manager.nsmanager.delivery → Deliverer (interface)	10
Defines the functionalities of Deliverer-objects (responsible for storing converted episodes)	
manager.nsmanager.delivery → DatabaseDeliverer (class)	95
Writes converted episodes to a database	
manager.nsmanager.delivery → FileDeliverer (class)	45
Writes converted episodes to the local machine	

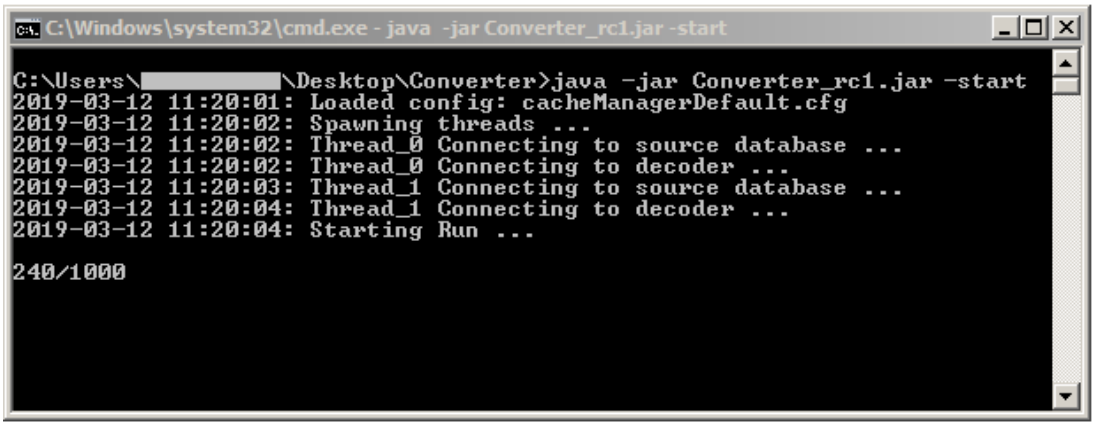


Figure 5.2: Screenshot of the command-line interface

```
def calculate_binary_string(data_window):
    # get mean
    dw_mean = np.mean(data_window)
    # substract mean from every element
    substracted_data = data_window - dw_mean
    # get min and max
    vn, vp = np.amin(substracted_data), np.amax(substracted_data)

    # calculate cn and pn
    cn = len([True for x in substracted_data if (0.1 * vn) < x < 0])
    cp = len([True for x in substracted_data if 0 < x < (0.1 * vp)])

    # calculate threshold
    if (cn + cp) < 0.4 * data_window.size:
        td = 0.0
    elif cn > cp:
        td = 0.2 * vp
    else:
        td = 0.2 * vn

    # generate binary string by comparing values to threshold
    binary_string = ''.join(list(map(lambda x: '0' if x < td else '1',
        substracted_data)))
    return binary_string
```

Listing 5.1: Python code to calculate the ECG's binary representation

```
def complexity(binary_string):
    S, Q = "", ""
    complexity = 0
    for s in binary_string:
        Q += s
        if not Q in S+Q[:-1]:
            S, Q = S+Q, ""
            complexity += 1

    asym_behavior = len(binary_string) / math.log2(len(binary_string))
    return complexity / asym_behavior
```

Listing 5.2: Python code to calculate the binary string's normalized complexity

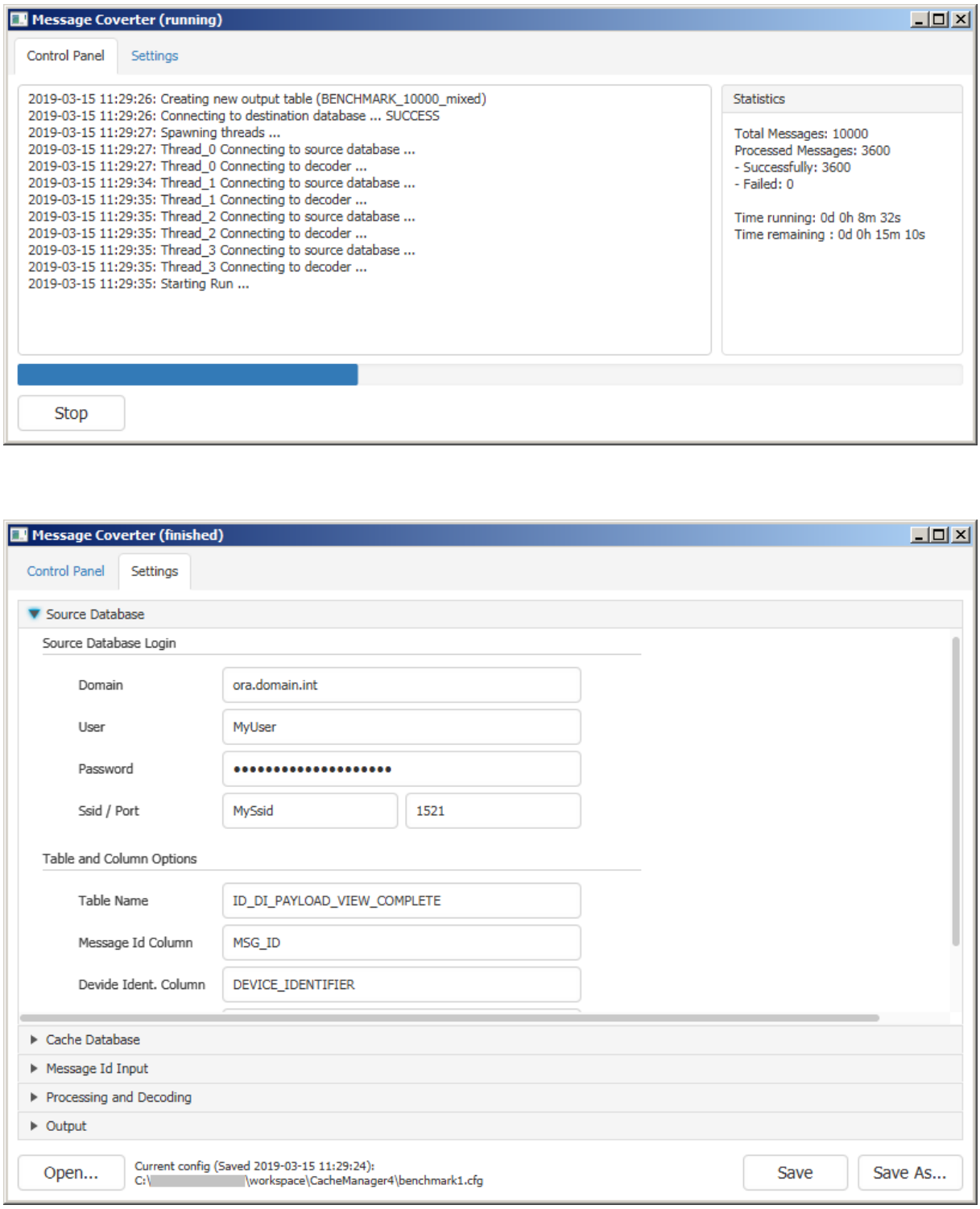


Figure 5.3: Screenshots of the Converter's GUI.

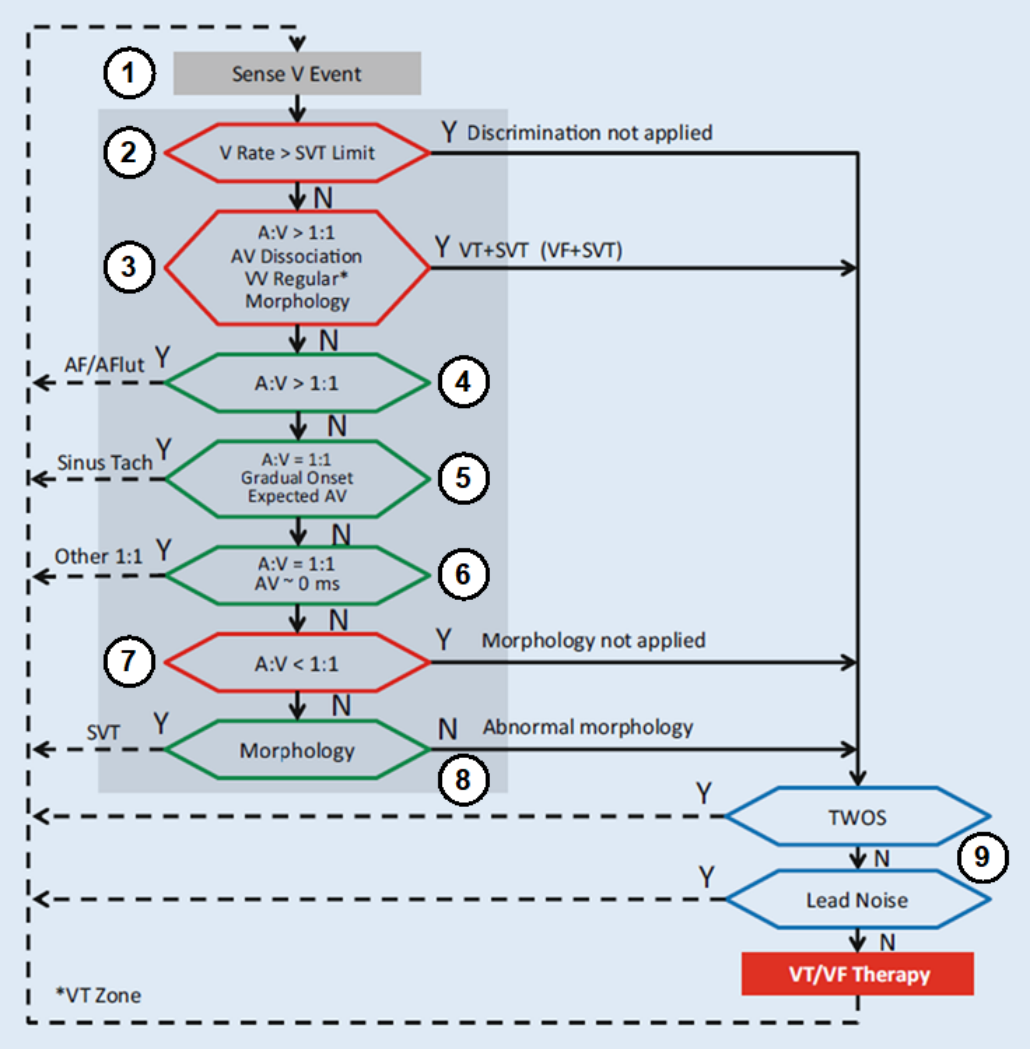


Figure 5.4: Simplified flowchart of an SVT/VT detection algorithm employed by a leading ICD manufacturer. The algorithm primarily relies on sensing the atrial (A) and ventricular (V) activity and, additionally, on comparing the signal to previously recorded healthy rhythms of the patient (Morphology).
1. SVT/VT discrimination is started once the ventricular rate exceeds a predefined threshold. **2.** If the ventricles contract faster than a predefined 'SVT Limit' the rhythm is classified as VT. As explained in 1.3 the AV node transmits signals only up to a limited frequency. Thus, particularly fast rhythms are unlikely to be caused by an SVT. **3.** The algorithm checks for double tachycardia (SVT+VT). A double tachycardia is considered if the atrial rate exceeds the ventricular rate (A:V $>$1:1), atria and ventricle beat independently of each other (AV Dissociation), ventricular rate is stable (VV Regular) and the ECG fits a certain morphology. **4.** If the rhythm is not classified as double tachycardia but the atrial rate exceeds the ventricular rate it is considered SVT caused by atrial fibrillation. **5.** Sinus tachycardia (non-pathological SVT) is diagnosed if atria and ventricle beat synchronously with stable PR intervals (Expected AV) and without sudden changes of the heart rate (Gradual Onset). **6.** If atria and ventricle beat synchronous and the PR interval equals zero (AV ˜0 ms) the rhythm is classified as SVT (e.g. reentrant tachycardia). **7.** Whenever the ventricular rate exceeds the atrial rate the rhythm is diagnosed as VT. **8.** If atria and ventricle beat synchronously and none of the previous criteria applied to the rhythm's morphology is used to classify the arrhythmia. **9.** Before applying therapy the algorithm checks for oversensing and device malfunctions. The TWOS feature is designed to prevent T-wave oversensing. This is important to ensure the ventricles' repolarization (T-wave) was not confused with their depolarization (P-wave). The Lead Noise detection checks whether or not a defective electrode lead triggered the algorithm by causing signal artifacts. [11]