Multimedia Tools and Applications (2024) 83:9785–9815
https://doi.org/10.1007/s11042-023-15249-5

metaFERA: a meta-framework for creating emotion recognition frameworks for physiological signals

João Oliveira[1] · Soraia M. Alarcão[1] · Teresa Chambel[1] · Manuel J. Fonseca[1]

Received: 5 July 2021 / Revised: 9 December 2022 / Accepted: 6 April 2023 /
Published online: 26 June 2023

Abstract
Recognizing emotions from physiological signals has proven to be important in various scenarios. To assist in developing emotion recognizers, software frameworks and toolboxes have emerged, offering ready-to-use components. However,these have limitations regarding the type of physiological signals supported, the recognition steps covered, or the acquisition of multiple physiological signals. This paper presents metaFERA, an architectural meta-framework for creating software frameworks for end-to-end emotion recognition from physiological signals. The modularity and flexibility of the meta-framework and the resulting frameworks allow the fast prototyping of emotion recognition systems and experiments to test and validate new algorithms. To that end, metaFERA offers: (i) a set of pre-configured blocks to which we can add behavior to create framework components; (ii) an easy way to add behavior to the pre-configured blocks; (iii) a channel-based communication mechanism that transparently and efficiently supports the exchange of information between components; (iv) a simple and easy way to use and link components from a resulting framework to create applications. Additionally, we provide a set of Web services, already configured, to make the resulting recognition systems available as a service. To validate metaFERA, we created a framework for Electrodermal Activity, an emotion recognizer to identify high/low arousal using the aforementioned framework, and a layer to offer the recognizer as a service.

Keywords Emotion recognition · Physiological signals · Software as a service · Software framework

✉ Manuel J. Fonseca
mjfonseca@ciencias.ulisboa.pt

João Oliveira
joliveira@lasige.di.fc.ul.pt

Soraia M. Alarcão
smalarcao@ciencias.ulisboa.pt

Teresa Chambel
mtchambel@ciencias.ulisboa.pt

1 LASIGE, Faculdade de Ciências, Universidade de Lisboa, Lisbon, Portugal

1 Introduction

In recent years, recognizing emotions from physiological signals has become an important area of research. Being able to accurately identify the emotion a person is feeling has proven useful in various scenarios. For example, to improve the effectiveness of psychological therapy sessions [21], gather feedback on how users perceive ads [23] or their reactions when interacting with user interfaces [15], and even to dynamically adjust the content of a video game to the player's emotional reactions [26]. Therefore, there is a need for efficient solutions that can identify people's emotions from physiological signals.

To help develop these solutions, a few frameworks and toolboxes have been created [4–6, 8, 9, 16, 18, 20, 24, 25]. For example, EEGLab [4] and OpenBCI [16] are two domain-specific frameworks for EEG signals, providing tools for signal pre-processing and functionalities for signal analysis and visualization. Matlab [9] and Weka [6], on the other hand, are two examples of more generic and extensible tools that allow the use of various types of physiological signals while providing algorithms for the majority of the emotional recognition stages at the expense of a potentially complex programming task for the developers. There is also iMotions, a complete commercial biometric research platform aimed at conducting studies of human behavior and not at supporting the development and testing of techniques and solutions for affective computing.

Although these frameworks help in the creation of new emotion recognition systems, they still have limitations, for instance, their specificity for only one type (or very few types) of signals; their suitability for only one of the stages of the emotional recognition process (e.g., pre-processing); their inability to deal with the online acquisition of physiological signals, being able to process only already pre-computed features; their difficulty of use, which increases the development effort involved in the creation of new applications; or the high monetary cost of using it.

To overcome these issues, we propose metaFERA (Meta-**F**ramework for **E**motion **R**ecognition **A**pplications), a software meta-framework that offers a backbone structure, a flexible and optimized connection mechanism, and a set of pre-configured building blocks. metaFERA can be seen as a set of configurable molds that allow the creation of "Lego" blocks, which can later be used to build emotion recognition applications. Since the configurable molds are pre-configured and are designed to produce blocks that fit well together, the researcher only needs to be concerned about configuring the molds (behavior of the components). Moreover, the blocks can be easily reused, added, removed, or switched. These blocks can be used to create specialized frameworks for each type of physiological signal (domain-specific frameworks), which can then be used to develop end-to-end emotion recognition systems with minimal effort for programmers.

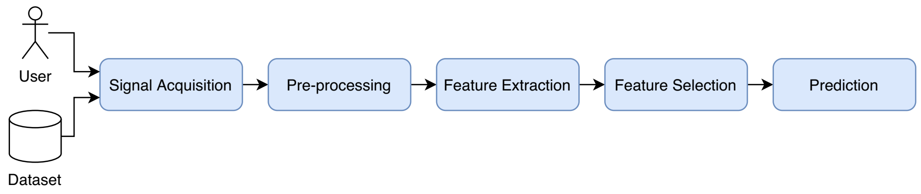

Fig. 1 Steps of a traditional emotion recognition system from physiological signals acquired from sensors or annotated datasets

Although most recent emotion recognition solutions use deep learning techniques, we designed metaFERA to support the traditional emotion recognition pipeline (Fig. 1). We made this choice because there are several situations where deep learning cannot be applied, such as when there is not enough data to train deep learning models or when we need to perform learning and prediction in an online setting while collecting data in real-time from human subjects. Moreover, deep learning models currently lack interpretability. Our open-source meta-framework was conceived to be easy to use for creating framework components (e.g., pre-processing, feature extraction, prediction algorithms, etc.), and for the resulting frameworks to also be easy to use for developing complete emotion recognition applications. metaFERA was designed as a set of interconnected pre-configured blocks encompassing all the main components of a traditional emotion recognition system from physiological signals. This way, its modular architecture offers developers not only the common structure of an emotion recognition system but also the abstraction needed to add new behaviors whenever they need to.

In this paper, we describe the meta-framework infrastructure and provide details on its modules, connection mechanism, recognition as a service concept, and how everything can be used to create domain-specific frameworks and emotion recognition applications. To validate metaFERA, we applied it to a specific scenario, by (i) developing an emotion recognition framework for Electrodermal Activity (EDA), exemplifying the usage of its building blocks and how to add behavior to them; (ii) creating a recognition application for identifying high/low arousal from EDA, showing the use of a specific framework created with metaFERA; (iii) using the recognition as a service concept, showing how an emotion recognizer can be made available for use by multiple thin clients (e.g., Web applications or mobile applications).

The rest of the paper is organized as follows: Section 2 gives an overview of the related frameworks for creating emotion recognizers. In Section 3, we present the building blocks of the meta-framework, as well as how it works and how it can be used to create recognition frameworks. Section 4 describes the Web services platform to offer recognition as a service. In Section 5, we describe the use of metaFERA, as a test case, for developing a domain-specific framework, which is then used to create a recognition system, and the correspondent layer to offer it as a service. We conclude the paper by presenting a discussion and main conclusions in Section 6.

2 Related work

As illustrated in Fig. 1, emotion recognition systems from physiological signals typically consist of a number of steps, namely: (i) acquisition of physiological signals from the user (through sensors) or from datasets; (ii) pre-processing the signals to remove noise and artifacts; (iii) extraction of relevant features from the signals; (iv) feature selection to identify the most discriminant features; and (v) machine learning methods for emotion prediction. To make the creation of this type of system easier, a few tools and frameworks have emerged seeking to offer blocks that correspond to each of the aforementioned steps.

Although there are a few tools and frameworks that can be used to build emotion recognition systems, many of them focus on a limited number of steps, like, for example, the toolboxes ANSLAB [2], Biosig [27], BrainStorm [25], CarTool [3], EEGLab [4], PhysioLab [18], or the Python library MNE [8], which focus mainly on signal pre-processing and visualization. These types of tools and frameworks have some disadvantages. First, they frequently can only read physiological data from datasets. An exception is the OpenBCI [16] and iMotions [12], which can read data from sensors. Second, they are typically specialized only in one type of signal, as is the case of EEGLab, BrainStorm, MNE NeuroTechX MOABB, and OpenBCI, which were created only for Electroencephalography (EEG) signal analysis. Finally, they can only pre-process the data and save the results in a file, which is not suitable for online emotion recognition systems, where signals are being collected in real-time from users and results are being produced.

On the other hand, there are more general tools, like the Matlab [9] platform, which can load datasets with physiological data and has an extensive library of functions for pre-processing and feature extraction. Additionally, its powerful scripting language allows further extensions to its capabilities. TEAP [24] is an example of a toolbox that expands Matlab's pre-processing library. One of the advantages of TEAP over other pre-processing tools is its generalization when it comes to signals, being able to process EEG, galvanic skin response (GSR), electrocardiography (ECG), blood volume pulse (BVP), electromyography (EMG), among others. Other extensions to Matlab add missing features, such as PRTools [5] and FieldTrip [20]. PRTools is a toolbox for Matlab that adds feature extraction, feature selection, and classification, showing that Matlab can be an effective tool for building emotion recognition algorithms; Fieldtrip has the advantage of supporting online processing of physiological signals by retrieving sensor data from a server. However, the scripting required to build an online emotion recognition system can be considerably complex. Moreover, adding or removing algorithms to the recognition pipeline is not trivial, making the experimentation of different methods slow and ineffective.

Another popular software for performing feature selection and classification/regression is Weka [6]. However, it can only deal with datasets of pre-computed features, not supporting the extraction of features from physiological signals nor the acquisition of signals directly from sensors. Although its API allows the creation of independent applications, this process has some complexity and does not allow fast experimentation of different methods.

Python [19] is a full-fledged programming language with a vast range of libraries available and widely popular within the scientific community. Among these libraries we have SciPy [28] and Scikit-learn [22], which provide extensive pre-processing and machine learning capabilities, making Python one of the most powerful tools for machine learning tasks. However, like the previously mentioned tools, it does not have a simple and efficient way to create online emotion recognition systems, where signals are directly collected from sensors in real-time, nor a fast and easy way to test and validate new methods or algorithms while developing traditional end-to-end emotion recognition applications.

iMotions is a complete software solution for biometric research, supporting a wide range of biosensors (e.g., eye tracking, facial expression analysis, EEG, GSR, and ECG) [12]. It offers an open API to import and export data from many data sources, stimuli presentation, built-in surveys, live visualization of data streams, complete study analysis, etc. Although iMotions provides the most comprehensive, easy-to-use, and scalable biometric research platform to conduct human behavior studies, it is primarily an expensive commercial closed research platform, making it difficult to create new methods.

Outside the realm of physiological signals and emotion recognition, we have RapidMiner [11], a popular enterprise solution for machine learning, primarily used to make predictive models for database analysis in economics and human behavior. An interesting feature of RapidMiner is the mechanism to create the prediction pipeline. It allows the connection of algorithms together to build the system, and all the communication between them is handled internally by RapidMiner. Although RapidMiner cannot be used to create emotion recognition systems nor to use physiological signals, we believe it will be possible

Table 1 Summary of the available libraries and tools that support the development of emotion recognition systems

Tool	Signal Acquisition	Pre-Processing	Feature Extraction	Feature Selection	Classification/ Regression
ANSLAB [2]	Dataset	✓	✓		
Biosig [27]	Dataset	✓	✓		Classification
BrainStorm [25]	Dataset	✓	✓		
CarTool [3]	Dataset	✓			
EEGLab [4]	Dataset	✓	✓	✓	
FieldTrip [20]	Dataset	✓	✓		
g.BSanalyze [7]	Dataset	✓	✓		Classification
iMotions [12]	Stream				
MNE [8]	Dataset	✓	✓		
NeuroTechX MOABB [13]	Dataset	✓	✓	✓	Classification
OpenBCI [16]	Stream	✓			
PhysioLab [18]	Dataset	✓	✓		
PRTools [5]	Dataset			✓	Classification
RapidMiner [11]	Dataset				✓
SMILE [14]	Dataset			✓	✓
TEAP [24]	Dataset	✓	✓		
Weka [6]	Dataset			✓	✓

to load a database with feature vectors and use its machine learning algorithms similarly to Weka. Thus, having the same drawbacks as Weka.

Despite these efforts over the years, as we can see in Table 1, most of the existing libraries and tools support a limited number of steps of a recognition system. Therefore, to create an emotion recognition application, we must combine a few frameworks and tools and make additional code, which is not always easy and fast. Thus, there is a need for an open-source meta-framework to support the development of domain-specific frameworks, which can then be used to easily create emotion recognition applications and the testing of new research ideas with minimal effort.

3 metaFERA

metaFERA[1] is an architectural meta-framework that provides generic pre-configured building blocks for creating domain-specific software frameworks for physiological signals. Such frameworks can then be used to develop end-to-end emotion recognition systems. Moreover, by being modular, it makes it simple to test new pre-processing techniques, feature extraction methods, or prediction algorithms, since the developer can easily replace components of an emotion recognition application without having to change the rest of the pipeline. Its architecture can be seen as a set of interlinked independent and pre-configured blocks, which

[1] metaFERA is developed in Java, and its source code and documentation can be found at https://git.lasige.di.fc.ul.pt/explore?name=metafera

define the main components of an end-to-end emotion recognition system from physiological signals and their connections, but not their behavior. The behavior is added during the creation of the domain-specific frameworks. For example, metaFERA offers a pre-configured block for feature extraction. This means, that developers who want to create, for example, a component to extract the Power Spectral Density (PSD) for their frameworks, only need to implement the algorithm needed to compute the PSD. They do not need to be concerned about the way the signal is received or the feature vector is sent since metaFERA takes care of that.

3.1 Architecture

Figure 2 presents an example of the multiple layers for application development using domain-specific frameworks created using metaFERA. Our meta-framework provides developers with structure, pre-configured building blocks, and the connection between them, so they can create new domain-specific frameworks (e.g., a framework for dealing with EDA). To construct these frameworks, developers must add behavior to the building blocks offered by metaFERA, like for example a pre-processing algorithm (e.g., band-pass filter), or a feature extraction algorithm (e.g., Hjorth parameters [10]) to create the components of the framework. Finally, these components from the domain-specific frameworks can be used to develop different emotion recognition applications. Notice that metaFERA is not directly used to develop applications, but rather to build frameworks that can later be used to create them.

metaFERA is organized as a set of independent, interrelated, and reusable modules with a well-defined API, which can be combined to create domain-specific frameworks. It offers

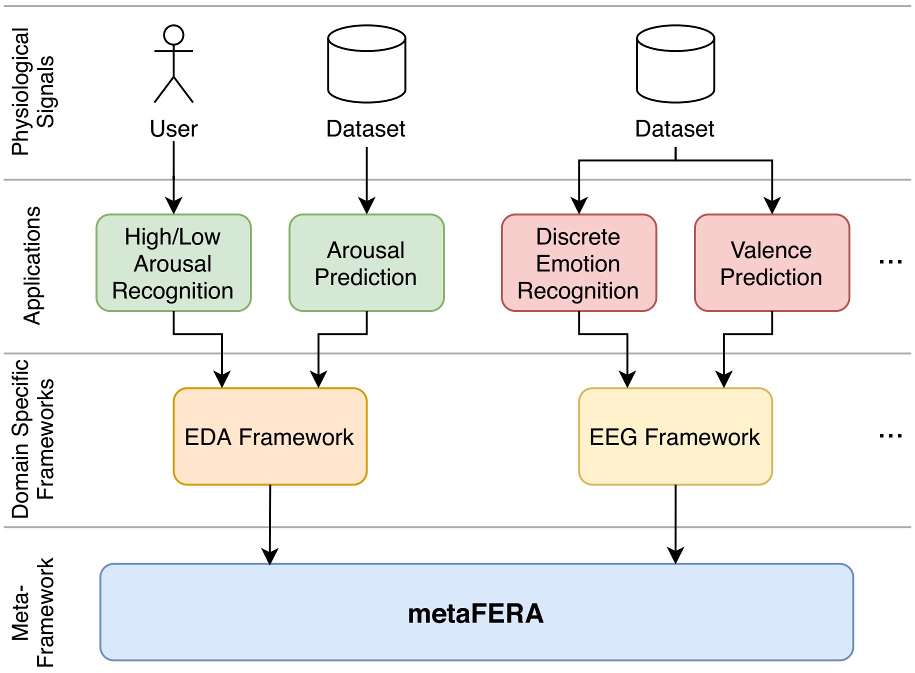

Fig. 2 Layers diagram for an example of use of the metaFERA meta-framework in the creation of domain-specific frameworks, which can then be used for developing emotion recognition applications

five main building blocks, corresponding each to the steps of a typical emotion recognition system from physiological signals, namely, signal acquisition, pre-processing, feature extraction, feature selection, and prediction (Figure 1); and five auxiliary modules to support feature scaling, feature aggregation, dimensionality reduction, results aggregation and results summarization.

Note that both types of modules only provide structure, and therefore it is necessary to add behavior to each of the pre-configured building blocks to create domain-specific frameworks. For instance, filters in the pre-processing module, or a machine learning method in the prediction module. To that end, developers can use already existing libraries, in Java, or wrappers for other programming languages, such as python and C++, that implement the desired behavior. As an example, if we want to create a band-pass filter component for a framework, we can import an existing Java library and use it to add behavior to a metaFERA Pre-processing block.

3.2 Communication mechanism

metaFERA provides a channel-based communication mechanism, which allows a simple and fast connection between blocks. Both types of blocks can have input channels, to receive data from other modules, and output channels to send data to other modules. The only exceptions are the Signal Acquisition which does not have any input channels and Results Summarization which does not have any output channels, since they are at the beginning and end of the recognition pipeline, respectively. An input channel can only receive data from one component, while an output channel can be connected to several modules. In such a case, all of them will receive the same data.

metaFERA has methods for sending a value (lines 2 and 3 in Listing 1), or a list of values (lines 4 and 5 in Listing 1) to the next components via the specified channel, and for receiving values from a given channel (line 7 in Listing 1). Each of the components' input channels has an input queue where the values are stored until it retrieves them.

Listing 1 Methods for sending and receiving data from channels.

```
1 //sending methods
2 public void send(double value)
3 public void send(double value, int channel)
4 public void send(double[] values)
5 public void send(double[] values, int channel)
6 //receiving methods
7 public Double receive(int channel)
```

The first method in Listing 2 checks if an output channel is open. This may be useful if for instance a module is computing several values (e.g., a Pre-processing module computing the delta, theta, alpha, beta, and gamma bands from an EEG) and sending each to a different output channel. If the output channel associated with one of the values is not open (i.e. it is not connected to another module), then that value does not need to be calculated, making the overall computation more efficient. The second method checks if the input channel is open and the third if it has more values available. If temporarily there are no values available, then the module can pause its execution for a short period of time using the method `rest()`.

Listing 2 Methods for checking the status of input and output channels.

```
public boolean isOutputChannelOpen(int channel)
public boolean isInputChannelOpen(int channel)
public boolean isInputChannelEmpty(int channel)
```

3.3 Main building blocks

In this subsection, we provide an overview of each of the metaFERA main building blocks, namely Signal Acquisition, Pre-processing, Feature Extraction, Feature Selection, and Prediction. We highlight what needs to be filled to add behavior to them and present an example for each.

In the description, and when it applies to the block, we consider two modes: i) building the model (learning) - where segments of signal (epochs) from several users are processed, feature vectors for each of these epochs are computed, and stored together with a ground-truth, and then used to build a model; and ii) using the model (prediction) - where the pipeline receives a signal epoch, computes a feature vector and produces a result for that epoch, using an existing model.

3.3.1 Signal acquisition

This block holds the API for acquiring physiological signals online from sensors or offline from datasets. In the latter case, it can also collect additional information, like for instance labels or self-assessment information for training or the gender of the users if we want to create different models for each gender.

Listing 3 is an excerpt of a block of this kind. It connects to a BITalino device [1] to acquire EDA signal samples and sends them to the next blocks. To create an instance of the Signal Acquisition block, the developer needs to extend the `SignalAcquisition` class, define its constructor, and implement its `run()` method. In the constructor, we need to call the `super()` method to call the constructor of the superclass, and pass the name of the component and the number of output channels. The `run()` method will contain the behavior of the block. In this example, it reads samples from a BITalino sensor and sends them to the output channel. Signal Acquisition blocks can be used to acquire signals from any type of biosensor, as long as we make the code to connect to the sensor.

3.3.2 Pre-processing

This block abstracts pre-processing algorithms, such as applying filters or splitting the signal into epochs. Instances of this block receive data from Signal Acquisition blocks and prepare them for feature extraction.

Listing 4 presents the excerpt of a band-pass filter created using this type of block. Here, the same two methods (constructor and `run()`) need to be defined. In the `run()` method, the while loop is performed until the block is closed. This is done by the module itself when it has no more data to be processed in the input channels. As long as there is data in the input channel, the module receives the data, applies the band-pass filter to the epoch, and sends the result to the output channel.

Listing 3 Example of a Signal Acquisition definition.

```
public class BitalinoSignalAcquisition extends SignalAcquisition {

  public BitalinoSignalAcquisition(String name) {
    //one output channel
    super(name, 1);
  }
  @Override
  public void run() {
    //connect to BITalino device
    ...
    while (bitalinoConnected) {
      //acquires sample from BITalino
      sample = device.read();
      //sends the acquired sample to the output channel
      send(sample, 0);
      ...
    }
  }
}
```

Listing 4 Example of a Pre-processing definition.

```
public class BandPass extends Preprocessing {

  public BandPass(String name, double signalFrequency, double centerfrequency, double width–
    frequency, int order) {
    //one input and one output channel
    super(name, 1, 1);
    ...
  }

  @Override
  public void run() {
    //create bandpass filter
    ...
    while (!closed()) {
      Double r;
      if ((r = receive(0)) != null) {
        //applies the bandpass filter and sends the result to the output channel
        send(bandpass.filter(r), 0);
      }
      else {
        rest(500);
      }
    }
  }
}
```

3.3.3 Feature extraction

This block holds the API for feature extraction methods. It can receive data directly from Signal Acquisition blocks or from Pre-processing blocks.

Listing 5 is an excerpt of a Feature Extraction block for computing the Hjorth parameters. Like the previous components, the constructor and the `run()` method are required. The workflow of the latter is similar to the one presented in Listing 4; the main difference is in the logic inside the while loop. In this case, we compute the three Hjorth parameters, join them in a feature vector and send it to the output channel.

Listing 5 Example of a Feature Extraction definition.

```
public class Hjorth extends FeatureExtraction {

  public Hjorth(String name) {
    //one input channel
    super(name, 1);
  }

  @Override
  public void run() {
    List<Double> values = new ArrayList<>();
    while (!closed()) {
      Double v;
      //acquires the epoch signal from the input channel and puts it in values
      if((v = receive(0)) != null) {
        values.add(v);
        //if the epoch is complete compute the features
        if(values.size() == EPOCH_SIZE) {
          ArrayList<Double> features = new ArrayList<>();

          features.add(activity(values));
          features.add(mobility(values));
          features.add(complexity(values));

          //sends the feature vector
          send(features);
          values.clear();
        }
      }
      else {
        rest(500);
      }
    }
  }
  //functions that calculate activity, mobility and complexity
  ...
}
```

The behavior added to the `run()` method is used for both the learning and prediction modes, since it deals with the different modes internally, making the task of adding behavior easier for the developer. While in prediction mode, the feature vector sent to the output channel is received directly by the next components, in learning mode, the feature vectors sent to the output channel are stored in a list together with the ground-truth labels (creating a temporary dataset). Only after the component closes itself (i.e. finishes processing all signals) the dataset is passed on to the following components.

3.3.4 Feature selection

This block exposes the feature selection API, allowing the use of different techniques to retain the most relevant and discriminant features. Unlike Signal Acquisition, Pre-processing, and Feature Extraction, this block receives a list of feature vectors in learning mode and a single feature vector in prediction mode. In the former, it uses the input dataset to identify the most discriminant features. As output, it produces a new dataset of feature vectors containing only the selected features and the ground-truth labels. In prediction mode, it uses the information about the features selected during the learning mode, to determine which features should be discarded from the input feature vector.

Listing 6 Example of a Feature Selection definition.

```
public class RecursiveFeatureSelection extends FeatureSelection {

  public RecursiveFeatureSelection(String name) {
    super(name);
  }

  //——— methods to use in learning mode ———
  //executes the feature selection algorithm
  public void performFeatureSelection() {
    Dataset dataset = getInputDataset();
    ...
    setOutputDataset(best_features_dataset);
  }

  //saves information about the selected features
  public void dump() throws IOException {
    super.dump();
  }

  //——— methods to use in prediction mode ———
  //loads information about the selected features
  public void restore() throws IOException {
    super.restore();
  }

  //selects the discriminant features from the feature vector
  public List<Double> selectFeatures(List<Double> features) {
    ...
  }
}
```

Listing 6 is an excerpt of a Feature Selection block, using the Recursive Feature Selection technique. The constructor of this type of block only needs to define its name, since by definition it only has one input and one output channel. The method `performFeatureSelection()` is where the feature selection algorithm is defined, while method `dump()` saves information about the selected features. They can be loaded later in prediction mode by the `restore()` method, and used in the `selectFeatures()` method to modify the input feature vectors. The `run()` method is pre-programmed and does not need to be implemented by the developer. In learning mode, it calls the `performFeatureSelection()` as soon as the previous block is closed, and in prediction mode, it receives feature vectors from the input channel and calls the `selectFeatures()` method before sending the resulting feature vector to the next components.

3.3.5 Prediction

This block holds the API for training machine learning models and making predictions using them. In learning mode, it receives a list of feature vectors and the corresponding ground-truth labels and builds the model. In prediction mode, it receives a feature vector and predicts a class or a value, depending if we are using a classification or a regression model, and sends the result to the next components.

Listing 7 is an excerpt of a Prediction block, that implements a RandomForest classifier. Prediction blocks, like Feature Selection ones, have a pre-programmed `run()` method, which in learning mode calls the `train()` method, and in prediction mode calls the `predict()` method. In these blocks, it is necessary to define the constructor, and the methods `train()` to create the model, `dump()` to save the created model, and `restore()` to load it. There is also a `validate()` method, for the evaluation of the model's performance, using for example cross-validation.

3.4 Auxiliary blocks

Besides the main building blocks, metaFERA also provides a set of auxiliary blocks, namely Feature Scaling, Feature Aggregation, Dimensionality Reduction, Results Aggregation, and Results Summarization. These blocks can be used to create more complex and complete recognition systems involving the manipulation of features or predicted results.

3.4.1 Feature scaling

This block defines the API for performing feature normalization/scaling. Similarly to the Feature Selection blocks, this block receives a dataset in learning mode and a feature vector in prediction mode. In learning mode, it uses the input dataset to compute the scaling parameters for each feature and produces as output a new dataset with the feature vectors normalized. In prediction mode, it uses the normalization information identified during the learning mode, to produce a normalized feature vector.

Listing 8 presents an excerpt of a Z-score normalization using the Feature Scaling block. The method `performNormalization()` is where the normalization algorithm is specified, while method `dump()` saves information about the normalization parameters. These can then be loaded in prediction mode by the `restore()` method, and used in the `normalize()` method to normalize the input feature vectors. Similarly to the Feature Selection block, the `run()` method is pre-programmed. In learning mode, it calls the `performNormalization()` method and in prediction mode calls the `normalize()` method.

Listing 7 Example of a Prediction definition.

```
public class RandomForestClassifier extends Prediction {
  private RandomForest rf = null;

  public RandomForestClassifier(String name) {
    super(name);
  }

  //——— methods to use in learning mode ———
  //builds a model
  @Override
  public void train() {
    //preparation of the feature vectors and the ground–truth for creating the model
    ...
    //RandomForest with 500 trees
    rf = new RandomForest(training_data, ground_truth_labels, 500);
  }

  //saves the trained model
  @Override
  public void dump() throws IOException {
    ...
  }

  //——— methods to use in prediction mode ———
  //loads the trained model
  @Override
  public void restore() throws IOException {
    ...
  }

  //predicts a class for the input feature vector
  @Override
  public void predict(List<Double> vector) {
    if (rf != null) {
      double r = rf.predict(vector);
      send(r);
    }
    else {
      // error message
      ...
    }
  }

  //method for training and testing a model
  @Override
  public void validate() {
    //validates a model using some validation method, like Cross–Validation.
    ...
    //sends results to a Results Summarization that can calculate metrics like Accuracy,
      F–Score, etc.
    ...
  }
}
```

Listing 8 Example of a Feature Scaling definition.

```
public class ZScoreNormalization extends FeatureScaling {

  public ZScoreNormalization(String name) {
    super(name);
  }

  //——— methods to use in learning mode——
  //computes the scaling parameters for each feature and applies them to all feature vectors
  public void performNormalization() {
    Dataset dataset = getInputDataset();
    ...
    setOutputDataset(normalized_dataset);
  }

  //saves information about feature scaling
  public void dump() throws IOException {
    //saves the standard deviation and mean for each feature
    ...
  }

  //——— methods to use in prediction mode——
  //loads information about feature scaling
  public void restore() throws IOException {
    ...
  }

  //normalizes the feature vector
  public List<Double> normalize(List<Double> features) {
    ...
  }
}
```

3.4.2 Feature aggregation

This block defines the API for joining several features using early fusion. It can receive data from more than one input channel, and combine them using an early fusion strategy. The Feature Aggregation block already has pre-programmed behavior. It constructs the feature vectors by joining the input vectors by the order they were added to this block. However, developers can overwrite the default behavior and define a new merging strategy, by redefining the `merge()` method (see Listing 9).

3.4.3 Dimensionality reduction

This block defines the API for performing dimensionality reduction. It has a structure very similar to Feature Selection and Feature Scaling blocks, with a learning mode where the criteria for dimensionality reduction are identified and a prediction mode, where these criteria are used to reduce the dimension of the feature vectors. Listing 10 illustrates the code for implementing a Principal Component Analysis (PCA) with metaFERA.

Listing 9 Example of a Feature Aggregation definition.

```
public class SpecialMerge extends FeatureAggregation {
  public SpecialMerge(String name) {
    super(name);
  }

  @Override
  public List<Double> merge(List<List<Double>> vectors) {
    //code for the special merge of feature vectors
  }
}
```

Listing 10 Example of a Dimensionality Reduction definition.

```
public class PrincipalComponentAnalysis extends DimensionalityReduction {

  public PrincipalComponentAnalysis(String name) {
    super(name);
  }

  //——— methods to use in learning mode ———
  //computes the criteria for the resulting features and applies them to all feature vectors
  public void performReduction() {
    Dataset dataset = getInputDataset();
    ...
    setOutputDataset(reduced_dataset);
  }

  //saves information about the reduction criteria
  public void dump() throws IOException {
    ...
  }

  //——— methods to use in prediction mode ———
  //loads information about the reduction criteria
  public void restore() throws IOException {
    ...
  }

  //applies the reduction to a feature vector
  public List<Double> reduce(List<Double> features) {
    ...
  }
}
```

3.4.4 Results aggregation

This block abstracts algorithms for aggregating several results using late fusion. This fusion can be done, for example, with a Voting algorithm. It already has the code needed to read the results received in the input channels, and send them to the `apply()` method. Thus, the developer only needs to implement the aggregation algorithm in the `apply()` method (see Listing 11).

Listing 11 Example of a Results Aggregation definition.

```
public class FusionWithClassifier extends ResultsAggregation {

  public FusionWithClassifier(String name) {
    super(name);
  }

  //receives a list of results and applies the aggregation
  public List<Double> apply(List<Double> results) {
    ...
  }
}
```

Listing 12 Example of a Results Summarization definition.

```
public class SimpleResults extends ResultsSummarization {

  public SimpleResults(String name) {
    super(name,1);
  }

  //returns a list with all the results from the session
  public List<Double> getAllResults() {
    ...
  }

  //saves the list with all the results from the session into a file
  public void saveAllResults() {
    ...
  }

  //returns the latest predicted result
  public Double getLatestResult() {
    ...
  }

  public void run() {
    while (!closed()) {
      //gathers the results from the input channel and adds them to a list
      ...
    }
  }
}
```

3.4.5 Results summarization

This block stores the results produced during a prediction session, exposing an API for accessing them. Developers can add behavior for creating for example a block just to provide access to the last value or to all the values of the session (see Listing 12). It can also be used to compute metrics, like precision, or recall, when using the `validate()` method of the Prediction block, since the Results Summarization block, has access to the ground-truth.

3.5 Connecting the blocks

By adding behavior to each of the main and auxiliary blocks, we can create components for domain-specific frameworks. Next, we can use these frameworks to develop systems for recognizing emotions from physiological signals. To that end, we need to define which components from the created framework are going to be used and how they will be connected.

The `AffectiveApplication` class is the main element for building a recognition system. It offers methods to create and connect components in sequence, from Signal Acquisition (first block) to Results Summarization (last block). Listing 13 shows a simple emotion recognizer built with a framework created with metaFERA. We start by creating an `AffectiveApplication` and then we connect all the components, by linking the output channels to the input channels. For example, to connect the output channel zero of the signal acquisition to the input channel zero of the band-pass filter, we execute the `app.addPreProcessing(feed, bandpass, 0, 0)` method.

Listing 13 A simple emotion recognition system created using a framework built with metaFERA.

```
public class EmoRecognizer {
  private static AffectiveApplication app;
  private static ResultsSummarization results;
  private static SignalAcquisition feed;

  //constants definition (omitted for readability reasons)
  ...

  public static void main(String[] args) {
    app = new AffectiveApplication("EmoRecognizer", ApplicationMode.PREDICT);
    feed = new BitalinoSignalAcquisition("Bitalino");
    app.addSignalAcquisition(feed);

    BandPass bandpass = new BandPass("Bandpass",  SIGNAL_FREQ, BANDPASS_CENTER_FREQ, BANDPASS
   _WIDTH_FREQ,BANDPASS_ORDER);
    app.addPreProcessing(feed, bandpass, 0, 0);

    Hjorth hjorth = new Hjorth("Hjorth");
    app.addFeatureExtraction(bandpass, hjorth, 0, 0);

    RandomForestClassifier rf = new RandomForestClassifier("RF");
    app.addPrediction(hjorth, rf);

    results = new ResultsCompiler("RESULTS");
    app.addResultsSummarization(rf, results, 0, 0);

    app.restore(); //restore the pre-saved models to use in prediction
    app.startAllComponents();

    while (!app.finished()) {
      Thread.sleep(SLEEP_TIME); //for showing the results only on every SLEEP_TIME millise-
      conds
      System.out.println(results.getLatestResult()); //prints the last prediction
    }
    app.stopAllComponents();
  }
}
```

In this simple emotion recognizer, we are creating a feed from a device, connecting it to a band-pass filter, and then to a Feature Extraction component (to compute the Hjorth parameters). This is connected to a Prediction component (Random Forest), and the latter, finally is connected to a Results Summarization block. As we can see, the creation and connection of components from a framework built with metaFERA is easy and simple.

If we want to create a multimodal recognizer, where we use more than one physiological signal (e.g., EDA, ECG, and EEG), it will be enough to use several Signal Acquisition components (implemented in a single framework or on different frameworks), one for each signal type, and then create the necessary connections to define the behavior of the emotion recognizer.

A more complete emotion recognition system is explained further in Section 5.2 and shown in Listings 19 and 21. Although this system has at least one component for each type of block offered by metaFERA, that is not required. We can use only a subset of the blocks, as long as Signal Acquisition (to input the signal data), Feature Extraction (to generate the features), Prediction (to predict the values), and Results Summarization (to obtain the final values) are included since they are mandatory blocks in an emotion recognition system. There is also some flexibility in the order the components that receive and send feature vectors can connect to each other. Feature Selection, Feature Scaling, Dimensionality Reduction, and Feature Aggregation components can be connected in any order, as long as they are included after Feature Extraction and before Prediction.

As we mentioned before, an affective application can work in two modes: learning and prediction. In learning mode, all components that generate features will create an internal list of feature vectors, known as dataset, along with their expected class (ground-truth). When this list is fully generated, that is, when the previous components have finished their task and there is no more data to process, the components apply their respective modifications to the dataset and pass it to the next component. This continues until the dataset reaches a Prediction component, where it is used to train the prediction model. In prediction mode, the components receive a feature vector, apply the required modifications according to the previously obtained data in training mode and send it to the next components. Finally, the Prediction component uses the received feature vector to calculate a result. All this processing is managed automatically and transparently by metaFERA, in accordance with the mode in which the application is running.

When developers connect the framework components to build an application, they need to call two mandatory methods. In learning mode, they must call the `dump()` method after all processing is finished to save the models and other data necessary in the future. Similarly, in prediction mode, they have to call the `restore()` method before starting the application to retrieve all the previously stored data.

4 Emotion recognition as a service

One of our goals while developing metaFERA, was to ensure that emotion recognition could be performed online, that is, get the signals directly from sensors in real-time and recognize the emotions, and be able to do it efficiently. However, although the framework was designed for this, the devices where the recognition will be performed may not have enough computational power to guarantee an efficient response.

metaFERA offers a Web service platform in order for emotion recognition to be used on a wider range of devices with different computing capabilities and in a wider variety of applications. This way, all the structures and essential functionalities needed to create a recognition system as a service are already developed and ready to use. This mechanism isolates the client applications from the services offered, making it possible to change the functionalities of the services (e.g., improve the algorithms available) without the need to modify the clients.

In the following subsections, we detail the services provided, how the communication between client and server is done, and the two client APIs (Java and JavaScript) to make using the Web services simpler.

4.1 Services provided

The platform provides the services summarized in Table 2, which can be organized in two groups of services: One for account management and another for using the affective applications.

4.1.1 Account services

The Account Services are used to register and unregister the client's account. When a client registers an account the server generates a token and returns it to the client. This token acts as an identifier and is required to use the affective applications.

Unregistering an account deletes the account and all its data and makes the token invalid. Clients that remain inactive for more than two hours are automatically unregistered. This way, we reduce the number of active accounts that are not in use, allowing us to optimize the use of the server's resources.

If a client was using the baseline service (for normalization purposes), when the account is unregistered, this information about the baseline is lost, and the client needs to restart the process again.

4.1.2 Affective application services

The Affective Application Services offer all the tools needed to setup and perform real-time prediction. The client can list the affective applications available using `getapplication-prdelist`, and check their details using `getapplicationspec`. These two endpoints, unlike the others, do not require the client to be registered.

The `loadmodel` endpoint instructs the server to load the desired affective application, which can be used to perform predictions from physiological signals, using the endpoint `predict`. The `addbaseline` endpoint receives signal data and uses it to identify the normalization parameters. These are used to normalize the signal sent when the client calls the `predict` endpoint.

4.2 Communication

The communication between client and server happens through HTTPS requests and responses. The type of request and response depends on the endpoint used. As shown in Table 2, services use either a GET or a POST request. For each endpoint it is also specified what kind of data the client should send to the server and what kind of data it will receive from the server.

Table 2 Summary of the services provided

Request URL	HTTPS	Description	Input	Output
Account services: `/emoservices/account/`				
`register`	GET	Registers an account in the server	–	*token* (account identifier)
`unregister/ token`	GET	Removes the account and all its data	*token*	–
Affective Application services: `/emoservices/applications/`				
`getapplicationlist`	GET	Returns a list of the available affective applications	–	*applicationid* (identifier) *name*
`getapplicationspec/ applicationid`	GET	Get a description of the affective application	*applicationid*	*description*
`loadmodel/ token/ applicationid`	GET	Ensures the affective application is loaded in the server	*token applicationid*	–
`predict/ token/ applicationid`	POST	Estimates using the affective application and signals in the epoch JSON. Calculates the emotional value and returns it to the client	*tokenapplicationid*	predicted values
`addbaseline/ token/applicationid`	POST	Adds the signal epochs to be used as baseline for whatever baseline process the affective application uses	*token applicationid*	success message

Account services allows the creation and deletion of accounts. Affective Application Services allows the users to get a list of applications and select which they want to use

Every server response have the same format, which is composed by a code and a message. This format, `Message JSON`, is presented in Listing 14.

Listing 14 Message JSON format.

```
{
  "code": messagecode,
  "message": message
}
```

The `code` tag represents the kind of response provided by the server, while the `message` tag contains the server response. The responses can be of the following types:

– ERROR - An error occurred in the server and it returned an error message;
– STRING_MESSAGE - The call to the end point was successful and the server returned a message with a success response;
– INFO_JSON - The server returned a list of generic data.
– PREDICTION_JSON - The server returned the predicted values and the feature vector used for prediction;

A response `code` ERROR or STRING_MESSAGE, will imply a `message` tag containing a String message, while a INFO_JSON `code`, will contain a JSON with the format shown in Listing 15.

Listing 15 Info JSON format.

```
{
  "info1":value1,
  "info2":value2,
  ...
}
```

Finally, when the `code` is PREDICTION_JSON, the `message` tag will contain a JSON with the format shown in Listing 16. Here, `expected` are the predicted values, and `vector` represents the several features extracted from the signal and used for prediction.

Listing 16 Prediction JSON format.

```
{
  "epochkeyword1":{
    "expected":[value1, value2, ..., valueM],
    "vector":[feature1, feature2, ..., featureN],
  },
  ...
}
```

4.3 Client API

To facilitate the development of applications that use emotion recognition as a service, we provide a pre-programmed layer that handles all communications with the endpoints. This layer provides a simplified API for the client application, so it does not have to deal with data serialization, Web service calls, or error handling. Currently, we offer a Java and a JavaScript version of this layer.

The Java layer API offers five classes with functionalities to help build and serialize JSON messages, deal with epochs, create feature vectors, and access the endpoints. The five classes are:

– JsonSerializationHelper - Contains methods to serialize and deserialize Epoch and PredictionOutput JSONs;
– Epoch - Corresponds to a single signal epoch;
– EpochSplitter - Allows the client to split an epoch into smaller epochs, by defining the size of the window and how many samples to slide after every window;
– PredictionOutput - Contains the prediction for an epoch, and the corresponding computed feature vector;
– EmoServicesWrapper - Allows the client to call the endpoints using a simple syntax.

For creating their clients, developers only need to use the `Epoch`, `PredictionOutput` and `EmoServicesWrapper` classes.

Listing 17 Example of the client-side code using the Java layer API.

```
//creates the wrapper and registers the account
EmoServicesWrapper wrapper = new EmoServicesWrapper();
wrapper.registerAccount();

//loads application with id 1 from the server
int appID = 1;
wrapper.loadModel(appID);

Dictionary<String, Epoch> epoch_list = new Hashtable<>();
List<Double> signal = new ArrayList<>();

boolean stop = false;
while(!stop) {
  //gather the signal
  signal.add(/*get data from sensor*/);

  if(signal.size() == NUMBER_SAMPLES) {
    //creates a signal epoch, adds it to the list of epochs and sends to server for
    prediction
    Epoch e = new Epoch();
    e.addSignal("EDA", signal);
    epoch_list.add("epoch", e);
    System.out.println(wrapper.predict(appID, epoch_list).get("epoch").getExpected()[0]));
  }

  //check some conditions to stop execution
  ...
}
```

Listing 18 Example of the client-side code using the Javascript layer API.

```
import {EmoServicesWrapper} from 'EmoServicesWrapper.js';

let appID = 1;
//creates the wrapper, registers the account and loads application 1 from the server
window.wrapper = new EmoServicesWrapper();
window.wrapper.registerAccount().then(() =>
  window.wrapper.loadModel(appID));

let signal = [];

//setup sensor connection and callbacks
...

//callback function to get sensor data
function receiveValues(values) {
  //gather the signal
  signal.push(values);

  if(signal.length >= NUMBER_SAMPLES) {
    //gets the first NUMBER_SAMPLES samples and builds a JSON
    let valuestosend = signal.slice(0, NUMBER_SAMPLES);
    let jsondata = {"epoch":{"EDA":valuestosend}};

    //sends the data and uses the predicted value
    window.wrapper.predict(appID, JSON.stringify(jsondata)).then((response) => { /* uses the
    predicted value returned in response.epoch.expected */ });

    //removes the samples used
    signal.splice(0, NUMBER_SAMPLES);
  }
}
```

The JavaScript layer API is much simpler, due to the language's built-in capabilities in building and manipulating JSON structures. This API has only one class, the `EmoServicesWrapper`, which provides methods for calling the endpoints.

Listings 17 and 18 show the Java and JavaScript code, respectively, needed to create an epoch of an EDA signal, send it to the server for prediction, and use the predicted result. As we can see, the developers do not need to be concerned with low-level operations, such as serialization, calls to the endpoints, or even error handling. All of this is offered transparently by the client's API.

5 Test case

In the previous sections, we described metaFERA, its main and auxiliary blocks, and how to add behavior to these pre-configured blocks to create domain-specific framework components. We have also seen how to use recognition as a service via the Web service platform and the client APIs.

In this section, we show how to put all the pieces together to create a set of components of the domain-specific framework for EDA, and how to build an end-to-end affective application for identifying high and low arousal from EDA using this framework (Fig. 3). We also describe

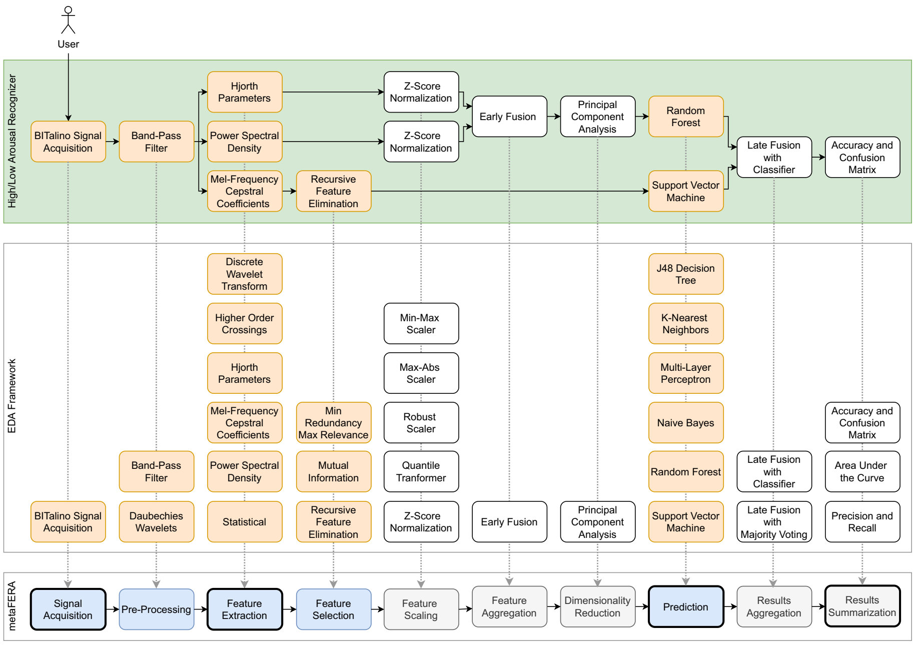

Fig. 3 Example of the use of metaFERA for creating a domain-specific framework, and the use of this to develop an emotion recognizer. (bottom) the main (in blue) and auxiliary (in gray) blocks of metaFERA; (middle) set of possible components for an EDA framework, created by adding behavior to the metaFERA blocks; and (top) an high/low arousal recognizer built using some of the components from the EDA framework. The bold border in the metaFERA blocks means these are mandatory in the creation of an emotion recognition system

how to offer this arousal recognizer as a service, and how to use it in a Java or JavaScript client. We conclude the section with a short performance evaluation of the resulting emotion recognition system, to measure the influence of the meta-framework in its overall efficiency.

5.1 Domain-specific framework

According to the scenario described above, we must first develop the components of the domain-specific framework for EDA that will allow us to create emotion recognizers using EDA signals. This consists in filling the main and auxiliary building blocks offered by metaFERA with specific behavior for each one (Fig. 3 - middle).

In Sections 3.3 and 3.4 we described how to add behavior to each of the main and auxiliary blocks to create the components of a domain-specific framework. Although we have only described the creation of some of the components of the EDA framework shown in Figure 3, the creation of the remaining ones would be very similar, so we do not describe it here. In practical terms, we can create as many components as we wish for the domain-specific frameworks, using the building blocks offered by metaFERA, and then use these components to create several recognition systems. Indeed, we can use the same component in different frameworks, as well as use components from different frameworks in the same recognition system.

5.2 Emotion recognition system

After creating the various components of the domain-specific framework, the next step is to create an affective application. To accomplish this, we combine the components of the framework to get the desired behavior.

According to our scenario, we are going to create a recognizer to identify high/low arousal from EDA (Fig. 3 - top). This recognizer collects EDA signals using a BITalino device and filters them using a band-pass filter. For feature extraction, it uses Hjorth parameters, Power Spectral Density (PSD), and Mel-Frequency Cepstral Coefficients (MFCC). The first two are normalized and combined using early fusion, and then the dimension is reduced

Listing 19 Building a (standalone) high/low arousal recognizer.

```
public class ArousalRecognizer {
  private static AffectiveApplication app;
  private static ResultsSummarization results;
  private static SignalAcquisition feed;

  //constants definition (omitted for readability reasons)
  ...

  public static void main(String[] args) {
    ApplicationMode mode = args[0] //learn/predict
    createPipeline(mode);

    app.startAllComponents();
    if(mode == ApplicationMode.LEARN) {
      app.stopAllComponents();
      app.dump(); //dumps all information for later use
    } else {
      while (!app.finished()) {
        Thread.sleep(SLEEP_TIME); //for showing the results only on every SLEEP_TIME millise—
        conds
        System.out.println(results.getLatestResult()); //prints the last prediction
      }
      app.stopAllComponents();
    }
  }

  private static void createPipeline(ApplicationMode mode) {
    app = new AffectiveApplication("ArousalRecognizer", mode);

    if (mode == ApplicationMode.LEARN)
      feed = new ExampleDatasetAcquisition("Dataset");
    else
      feed = new BitalinoSignalAcquisition("Bitalino");

    app.addSignalAcquisition(feed);
    createComponents(mode);
  }

  private static void createComponents(ApplicationMode mode) {
    ...
  }
}
```

using Principal Component Analysis (PCA). Several MFCC features are selected using the Recursive Feature Elimination feature selection method. Two machine learning methods for classification are used and their results are combined using a late fusion approach with a classifier, to produce the final result. Finally, a Results Summarization module is used to compute accuracy and the confusion matrix. Listings 19 and 21 show the creation of this affective application.

5.3 Emotion recognition system as a service

To provide an emotion recognizer as a service, developers only need to extend the `ServiceAffectiveApplication` class. It already provides all the necessary mechanisms to link the recognizer with the endpoints, without the need for additional programming. Developers just have to program their emotion recognizer as they would for a standalone application. The main differences to consider are: (i) the need to pause and resume components in order to optimize the server resources; and (ii) the signal acquisition method, which in this case uses an epoch provided by the client application rather than receiving it directly from a signal feed.

As we can see in Listing 20 the way the recognizer is created is the same as in the standalone application (Listing 19). Indeed, the code for creating the components (Listing 21) is the same for both. Thus, it is easy to migrate or adapt a standalone recognizer to be offered as a service.

5.4 Performance

We conducted an experiment to analyze the performance of the resulting emotion recognition system, using a PC Intel Core i7-3630QM CPU@2.40GHz, with 16GB of RAM and running Windows 8.1 64-bit.

In the experiment, we used as input the EDA signals present in the AMIGOS dataset [17]. This dataset contains several physiological signals collected from 40 participants while they were alone watching 16 short videos (2 to 5 minutes). The EDA signals were recorded using the Shimmer 2R platform with an EDA module board (128 Hz, 12-bit resolution).

Our goal was to simulate the acquisition of physiological signals from a real user (directly from a sensor). To do this, we used the signals from the first participant present in the dataset, and simulated online signal acquisition by applying a 1/128 seconds pause between samples to simulate the sampling frequency of the device used to collect the EDA. The signal was split into epochs of 5 seconds, with no overlap.

We considered the time required to obtain a prediction as the time elapsed from when the Signal Acquisition block finishes sending the epoch until the Results Summarization component receives the prediction result. We did not consider the time needed to collect the epoch samples, because it will always be fixed, and dependent on the chosen epoch duration.

The average prediction time per epoch, considering 100 epochs, was 2.84 ms (SD=1.79), with a minimum of 1 ms and a maximum of 8 ms. In practical terms, and considering the epoch of 5 s, this means that in a real situation, we would have a prediction every 5 s. So, the time to get a prediction is determined by the duration of the epoch, and not by the meta-framework performance. It is however worth noting that the performance of the algorithms to include in the framework components can influence the performance of the overall system. Therefore, programmers should favor the use of more efficient algorithms in their components.

Listing 20 Building the high/low arousal recognizer to be used as a service.

```
public class ServerArousalRecognizer extends ServiceAffectiveApplication {
  protected static AffectiveApplication app;
  protected static ResultsSummarization results;
  protected static SignalAcquisition feed;

  //constants definition (omitted for readability reasons)
  ...

  public ServerArousalRecognizer() {
    //creates the pipeline, restores the models and tells the pipeline to wait until it is
    needed
    createPipeline();
    app.startAllComponents();
    app.restUntilResumed();
  }

  public List<Double> predict(Epoch epoch) {
    feed.addEpoch(epoch);
    app.resumeComponents(); //unpauses the components
    List<Double> result = null;
    while ((result = results.getLastResult()) == null) {
      Thread.sleep(SLEEP_TIME);
    }
    app.restUntilResumed(); //pauses the components
    results.resetResults();

    return result;
  }

  private static void createPipeline() {
    app = new AffectiveApplication("ServerArousalRecognizer", ApplicationMode.PREDICT);
    feed = new EpochFeed("Epoch");
    app.addSignalAcquisition(feed);
    createComponents();
  }

  private static void createComponents() {
    ...
  }
}
```

Taking all this into account, we can conclude that: (i) in a scenario where we have an emotion recognizer with a relatively high complexity, which uses blocks of all types and several blocks of the same type (Fig. 3), the execution time is almost insignificant; (ii) the weight of the meta-framework on the performance of the final solution is negligible. Thus, we can consider that metaFERA does not affect the overall performance of the recognition system created using frameworks developed with it.

6 Conclusions

In this paper, we described metaFERA, an open-source architectural meta-framework for creating domain-specific software frameworks, which enables developers to build end-to-end

Listing 21 Method for creating the components of the application.

```
private static void createComponents(ApplicationMode mode) {
  BandPass bandpass = new BandPass("Bandpass", SIGNAL_FREQ, BANDPASS_CENTER_FREQ,
  BANDPASS_WIDTH_FREQ,BANDPASS_ORDER);
  app.addPreProcessing(feed, bandpass, 0, 0);

  Hjorth hjorth = new Hjorth("Hjorth");
  app.addFeatureExtraction(bandpass, hjorth, 0, 0);
  PowerSpectralDensity psd = new PowerSpectralDensity("PSD");
  app.addFeatureExtraction(bandpass, psd, 0, 0);
  MelFrequencyCepstralCoefficients mfcc = new MelFrequencyCepstralCoefficients("MFCC",
  MFCC_NCOEFFS);
  app.addFeatureExtraction(bandpass, mfcc, 0, 0);

  RecursiveFeatureSelection rfselect = new RecursiveFeatureSelection("RFS");
  app.addFeatureVectorModifier(mfcc, rfs);

  FeatureAggregation join = new FeatureAggregation("JOIN");
  app.addFeatureAggregation(hjorth, join);
  app.addFeatureAggregation(psd, join);

  ZScoreNormalization zscore = new ZScoreNormalization("ZSCORE");
  app.addFeatureVectorModifier(join, zscore);

  PrincipalComponentAnalysis pca = new PrincipalComponentAnalysis("PCA");
  app.addFeatureVectorModifier(zscore, pca);

  RandomForestClassifier rf = new RandomForestClassifier("RF");
  app.addPrediction(pca, rf);
  SupportVectorMachineClassifier svm = new SupportVectorMachineClassifier("SVM");
  app.addPrediction(rfselect, svm);

  FusionWithClassifier mv = new FusionWithClassifier("MV");
  app.addResultsAggregation(rf, mv);
  app.addResultsAggregation(svm, mv);

  results = new ResultsCompiler("RESULTS");
  app.addResultsSummarization(mv, results, 0, 0);

  //=== code for standalone application only ===
  if (mode == ApplicationMode.LEARN) {
    //connect class labels (ground-truth) for validation purposes
    app.connectExpectedClassLabelInput(feed, hjorth, 1);
    app.connectExpectedClassLabelInput(feed, mfcc, 1);
    app.connectExpectedClassLabelInput(rf, mv, 1);
    app.connectExpectedClassLabelInput(mv, results, 1);
  }
  else {
    app.restore(); //restore the pre-saved models to use in prediction
  }

  //=== code for using the recognizer as a service only ===
  app.restore(); //restore the pre-saved models to use in prediction
}
```

emotion recognition systems based on physiological signals without requiring a thorough development process. The main building blocks for creating frameworks are aligned along the traditional processing pipeline commonly used in emotion recognition systems from physiological signals. It also offers a layer to provide emotion recognition as a service. This allows developers to create frameworks for developing emotion recognizers with minimal effort. To do this, they simply need to add behavior to the pre-configured blocks through specific implementations and combine them by connecting the output channels to the input ones.

Although metaFERA requires developers to define behavior for its blocks (to create the framework's components), they can use existing libraries in various programming languages. Nevertheless, we intend in the near future to provide a set of components with their behavior implemented to make it even easier and faster for developers to create new frameworks and emotion recognizers. We plan to start by offering components that are independent of the type of physiological signals, such as feature selection or prediction algorithms, so that they can be used as components on a wider variety of domain-specific frameworks. Our ultimate goal is to allow the creation of an ecosystem of several frameworks and components (created by different researchers) to which the community can contribute and use the existing frameworks and components to develop new frameworks and emotion recognition applications.

We evaluated and validated the potential of metaFERA for building domain-specific frameworks and using these frameworks to build emotion recognition systems by using a practical and complete example. Throughout the paper, we created a framework for dealing with EDA signals and then used it to construct a standalone application for identifying high/low arousal. We also illustrated how to provide this arousal recognizer as a service, showing the minor differences from developing a standalone application.

In summary, metaFERA allows developers to quickly and easily develop new frameworks for building affective applications, as well as to test and compare new algorithms for coping with physiological signals. It supports all steps of a traditional emotion recognition pipeline and can process various types of physiological signals, either individually or simultaneously.

Acknowledgements This work was supported by FCT through project AWESOME, ref. PTDC/CCI-INF/29234/2017, and the LASIGE Research Unit, ref. UIDB/00408/2020 and ref. UIDP/00408/2020. Soraia M. Alarcão is funded by an FCT grant, ref. SFRH/BD/138263/2018.

Funding Open access funding provided by FCT|FCCN (b-on).

Data Availability Data sharing not applicable to this article as no datasets were generated or analyzed during the current study.

Declarations

Conflicts of interest The authors declare no conflict of interest.

References

1. Alves, A.P., Silva, H., Lourenço A. L., Fred, A.L.: BITalino: A Biosignal Acquisition System based on the Arduino. In: Proceeding of the International Conference on Biomedical Electronics and Devices, pp. 261–264 (2013)
2. Blechert J, Peyk P, Liedlgruber M, Wilhelm FH (2016) Anslab: Integrated multichannel peripheral biosignal processing in psychophysiological science. Behavior Research Methods 48(4):1528–1545
3. Brunet, D., Murray, M.M., Michel, C.M.: Spatiotemporal analysis of multichannel eeg: Cartool. Computational intelligence and neuroscience **2011** (2011)
4. Delorme A, Makeig S (2004) Eeglab: an open source toolbox for analysis of single-trial eeg dynamics including independent component analysis. Journal of neuroscience methods 134(1):9–21
5. Duin, R.P.W.: Prtools version 3.0: A matlab toolbox for pattern recognition. In: Proceedings of SPIE, p. 1331 (2000)
6. Frank E, Hall M, Trigg L, Holmes G, Witten IH (2004) Data mining in bioinformatics using weka. Bioinformatics 20(15):2479–2481
7. g.BSANALYZE: OFFLINE BIOSIGNAL ANALYSIS FOR MATLAB. https://www.gtec.at/product/gbsanalyze/. [Online; Accessed 02 December 2022]
8. Gramfort A, Luessi M, Larson E, Engemann DA, Strohmeier D, Brodbeck C, Parkkonen L, Hämäläinen MS (2014) Mne software for processing meg and eeg data. Neuroimage 86:446–460
9. Higham, D.J., Higham, N.J.: MATLAB guide. SIAM (2016)
10. Hjorth B (1970) EEG Analysis Based on Time Domain Properties. Electroencephalography and Clinical Neurophysiology 29(3):306–310. https://doi.org/10.1016/0013-4694(70)90143-4
11. Hofmann, M., Klinkenberg, R.: RapidMiner: Data mining use cases and business analytics applications. CRC Press (2016)
12. IMotions: Biometric Research Platform (SW Version). https://imotions.com/ (2001). [Online; accessed 02 December 2022]
13. Jayaram, V., Barachant, A.: Moabb: trustworthy algorithm benchmarking for bcis. Journal of neural engineering **15**(6) (2018)
14. Li, H.: Smile-statistical machine intelligence & learning engine (2016)
15. Liapis, A., Katsanos, C., Karousos, N., Xenos, M., Orphanoudakis, T.: User experience evaluation: A validation study of a tool-based approach for automatic stress detection using physiological signals. International Journal of Human-computer Interaction pp. 1–14 (2020)
16. Michalska M (2009) Openbci: Framework for brain-computer interfaces. University of Warsaw Faculty of Mathematics, Informatics and Mechanics
17. Miranda Correa, J.A., Abadi, M.K., Sebe, N., Patras, I.: AMIGOS: A Dataset for Affect, Personality and Mood Research on Individuals and Groups. IEEE Transactions on Affective Computing pp. 1–14 (2018). 10.1109/TAFFC.2018.2884461
18. Muñoz, J.E., Gouveia, E.R., Cameirão, M.S., i Badia, S.B.: Physiolab-a multivariate physiological computing toolbox for ecg, emg and eda signals: a case of study of cardiorespiratory fitness assessment in the elderly population. Multimedia Tools and Applications **77**(9), 11521–11546 (2018)
19. Oliphant TE (2007) Python for scientific computing. Computing in Science & Engineering 9(3): 10–20
20. Oostenveld, R., Fries, P., Maris, E., Schoffelen, J.M.: Fieldtrip: open source software for advanced analysis of meg, eeg, and invasive electrophysiological data. Computational intelligence and neuroscience **2011** (2011)
21. Palestra G, Pino O (2020) Detecting Emotions During a Memory Training Assisted by a Social Robot for Individuals with Mild Cognitive Impairment (MCI). Multimedia Tools and Applications 79(47):35829–35844
22. Pedregosa, F., Varoquaux, G., Gramfort, A., Michel, V., Thirion, B., Grisel, O., Blondel, M., Müller, A., Nothman, J., Louppe, G., Prettenhofer, P., Weiss, R., Dubourg, V., Vanderplas, J., Passos, A., Cournapeau, D., Brucher, M., Perrot, M., Èdouard Duchesnay: Scikit-learn: Machine learning in python (2018)
23. Pham MT, Geuens M, De Pelsmacker P (2013) The influence of ad-evoked feelings on brand evaluations: Empirical generalizations from consumer responses to more than 1000 tv commercials. International Journal of Research in Marketing 30(4):383–394
24. Soleymani M, Villaro-Dixon F, Pun T, Chanel G (2017) Toolbox for emotional feature extraction from physiological signals (teap). Frontiers in ICT 4:1
25. Tadel, F., Baillet, S., Mosher, J.C., Pantazis, D., Leahy, R.M.: Brainstorm: a user-friendly application for meg/eeg analysis. Computational intelligence and neuroscience **2011** (2011)

26. Tijs, T.J.W., Brokken, D., IJsselsteijn, W.A.: Dynamic game balancing by recognizing affect. In: P. Markopoulos, B. de Ruyter, W. IJsselsteijn, D. Rowland (eds.) Fun and Games, pp. 88–93. Springer Berlin Heidelberg, Berlin, Heidelberg (2008)
27. Vidaurre, C., Sander, T.H., Schlögl, A.: Biosig: the free and open source software library for biomedical signal processing. Computational intelligence and neuroscience **2011** (2011)
28. Virtanen, P., Gommers, R., Oliphant, T.E., Haberland, M., Reddy, T., Cournapeau, D., Burovski, E., Peterson, P., Weckesser, W., Bright, J., van der Walt, S.J., Brett, M., Wilson, J., Millman, K.J., Mayorov, N., Nelson, A.R.J., Jones, E., Kern, R., Larson, E., Carey, C.J., Polat, İ., Feng, Y., Moore, E.W., VanderPlas, J., Laxalde, D., Perktold, J., Cimrman, R., Henriksen, I., Quintero, E.A., Harris, C.R., Archibald, A.M., Ribeiro, A.H., Pedregosa, F., van Mulbregt, P., SciPy 1.0 Contributors: SciPy 1.0: Fundamental Algorithms for Scientific Computing in Python. Nature Methods **17**, 261–272 (2020). 10.1038/s41592-019-0686-2