EECR: Energy-efficient context recognition

Release 0.1

Vito Janko

Feb 17, 2020

CONTENTS:

1 Overview **1**

2 Code documentation **3**
2.1 Setting up the EnergyOptimizer object 3
2.2 Summarizing its properties 4
2.3 Searching for and evaluating configurations 5
2.4 Saving and loading data 10
2.5 Automatic settings generators 11
2.6 Visualizing solutions 14
2.7 Alternative search methods 15
2.8 Cost-sensitive decision trees 16

3 Method list **19**

4 Code examples **21**
4.1 Base case 21
4.2 Test on real-life SHL dataset 25
4.3 Cost-sensitive decision trees 27
4.4 Automatic settings generation 30

5 Indices and tables **31**

Python Module Index **33**

CHAPTER ONE

OVERVIEW

Widespread accessibility of wearable sensing devices opens many possibilities for tracking the users who wear them. Possible applications range from measuring their exercise patterns and checking on their health, to determining their location. In this documentation we will use the term context-recognition for all these tasks.

A common problem when using such context-recognition systems is their impact on the battery life of the sensing device. It is easy to imagine that an application that monitors users' habits using all the sensors in a smartphone (accelerometer, GPS, Wi-Fi etc.) will quickly drain the phone's battery, making it useless in practice.

While many methods for reducing the energy-consumption of a context-recognition system already exist, most of them are specialized. They work either in a specific domain or can only optimize the energy consumption of specific sensors. Adapting these methods to another domain can be laborious and may require a lot of expert knowledge and experimentation.

We developed three novel methods for generating good energy-efficient solutions that are independent of the domain and can easily be used to optimize a wide range of context-recognition tasks. The first method, ""Setting-to-Context Assignment (SCA)" changes the sensing settings – which sensors to use, with what frequency, what duty cycles to use etc. – depending on what context was last detected. To do so, we developed a mathematical model that can predict the performance of any given setting-to-context assignment. The SCA method then uses the NSGA-II algorithm to search the space of possible assignments, finding the best ones. The second method, "Duty-Cycle-Assignment (DCA)" works in a similar way to the SCA method, but is specialized in optimizing only duty-cycling, i.e., periodically turning the sensors on and off. This specialization can be justified by the fact that duty-cycling is useful in basically all context-recognition tasks.

Finally, the "Cost-Sensitive Decision-Tree (CS-DT)" method adapts sensing settings directly to the sensor data. It works by using a cost-sensitive decision tree that was adapted for context-recognition tasks. All three methods were then combined in three different ways in order to join their individual strengths. These combined methods can adapt to both the current context and to the sensor data.

Instead of returning only one solution, all of our methods can find different trade-offs between the classification quality and the energy consumption. Returning these trade-offs can help the system designer to pick one suitable for their system.

The methods are explained and in depth in PhD work <Insert a link>. The methodology was also tested on four different real-life datasets, and on a family of artificial datasets. Doing so, we proved that it works under many different conditions, with different possible settings, sensors and problem domains. We also showed that our methods compare favourably against other state-of-the-art methods. For each of the tested datasets we found many energy-efficient solutions. For example, for the Commodity12 dataset, we reduced the energy consumption from 123 mA to 29 mA in exchange for less than 1 percentage point of accuracy. For another example, we were able to use only 5% of the available data in the SHL dataset (by using a lower frequency, duty-cycling and a subset of sensors) and in exchange sacrifice only 5% of the accuracy.

CHAPTER

TWO

CODE DOCUMENTATION

2.1 Setting up the EnergyOptimizer object

The first step is to create the EnergyOptimizer object and update it with the properties of the desired dataset. All these properties can be set either in the constructor or one-by-one with setter functions. Warning: do not manually change any method variables without using the appropriate functions.

class `eecr.eeoptimizer.`**`EnergyOptimizer`**(*sequence=None, contexts=None, setting_to_energy=None, setting_to_sequence=None, quality_metric=None, sensors_off=None, sensors_on=None, path=None*)

`__init__`(*sequence=None, contexts=None, setting_to_energy=None, setting_to_sequence=None, quality_metric=None, sensors_off=None, sensors_on=None, path=None*)

Parameters

- **`sequence`** – a sequence of contexts to be used for testing the energy-efficiency of settings. Their ordering and proportions should be representative for the domain
- **`contexts`** – the list of contexts in the domain. If not provided, they will be inferred as all unique elements of the `sequence` parameter in the alphabetical order
- **`setting_to_energy`** – a dictionary, where the keys are the possible settings and the values are the energy costs (per time unit) when using that setting
- **`setting_to_sequence`** – a dictionary, where the keys are the possible settings and the values are lists of contexts. Each list represents the original sequence (see `set_sequence()`), classified using the corresponding setting
- **`quality_metric`** – a function that maps a confusion matrix to a quality indicator. Accuracy score is used by default
- **`sensors_off`** – the energy cost per time unit when the system is sleeping
- **`sensors_on`** – the energy cost per time unit when the system is working
- **`path`** – the root of the path used for saving and loading objects (e.g. with `save_data()` or `load_data()`)

`set_dca_costs`(*cost_off, cost_on*)

Sets the base energy costs for duty-cycling.

This costs will be used for all DCA methods (e.g. `dca_model()`, `dca_real()`, `find_dca_tradeoffs()` etc.). This overrides the default costs of 1 when the system is working and 0 when the system is sleeping.

Parameters

- **`cost_off`** – the energy cost per time unit when the system is sleeping
- **`cost_on`** – the energy cost per time unit when the system is working. If a setting is specified when using any of the DCA methods this value will be ignored and the energy cost of that setting will be used instead

`set_path`(*path*)
Sets the path to a folder from which the data is loaded and to which data is saved.

Parameters **`path`** – relative path to a folder

`set_sequence`(*sequence*)
Sets the sequence of contexts that will be used for testing the energy-efficiency of settings.

Parameters **`sequence`** – a sequence of contexts. Their ordering and proportions should be representative for the domain

`set_settings`(*setting_to_sequence*, *setting_to_energy=None*, *setting_fn_energy=None*)
Defines the possible settings and their performance/energy for the optimization.

Either `setting_to_energy` or `setting_fn_energy parameter` must be provided. Providing both or neither may result in unexpected behavior.

Parameters

- **`setting_to_sequence`** – a dictionary, where the keys are the possible settings and the values are lists of contexts. Each list represents the original sequence (see `set_sequence()`), classified using the corresponding setting
- **`setting_to_energy`** – a dictionary, where the keys are the possible settings and the values are the energy costs (per time unit) when using that setting
- **`setting_fn_energy`** – a function to be used if the `setting_to_energy` is None. This function should take a setting as the input and return energy costs (per time unit) when using that setting

2.2 Summarizing its properties

Use these functions for a quick look-up into created EnergyOptimizer object.

class `eecr.eeoptimizer.`**`EnergyOptimizer`**(*sequence=None*, *contexts=None*, *setting_to_energy=None*, *setting_to_sequence=None*, *quality_metric=None*, *sensors_off=None*, *sensors_on=None*, *path=None*)

`energy_quality`()
Returns a summary of all settings and their classification/energy performances.

Returns a dictionary, where the keys are the possible settings and values are (q,e) tuples, where the "q" represents the classification performance according to the specified quality metric and "e" represents the energy cost of that setting.

`quality`()
Returns a summary of all settings and their classification performances.

Returns a dictionary, where the keys are the possible settings and values the classification performances according the specified quality metric

`summary`()
Prints a summary (mathematical properties) of the dataset.

2.3 Searching for and evaluating configurations

The meat of the module. It is able to automatically find three types of energy-efficient solutions and predict their performance.

First type of solutions are the SCA configurations. Such a configuration is a list of settings (e.g. [s1, s2, s3]) where settings can be any hashable object. A configuration represents a system that works as follows: context c1 is detected, system switches to using setting s1, and keeps using it until a context change is detected. Then if the context c_i is detected the setting s_i is used (s_i is the i-th setting in the given configuration).

Second type of solutions are the DCA configurations. Such a configuration is a list of integers (e.g. [5,4,2]). A configuration represents a system that works as follows: context c1 is detected and the system stop working for (s_i)-1 time periods (s_i is the i-th entry in the configuration). Then it works for one (or more if so explicitly specified) time periods. The last classified context in this active period determines the length of the next sleeping period.

Third type of solutions are the SCA-DCA configurations. They essentially do both: switching both the setting in use and the length of the sleeping period. They are represented in the form (SCA_configuration, DCA_configuration).

Any such configuration can be evaluated using two different mechanism. One is the simulation, and the other is the mathematical prediction (based on the dataset properties). The latter approach is usually roughly 100 times faster and almost as accurate. In all cases the evaluations of configurations are returned as the list of tuples [(q1,e1), (q2,e2), . . .] where `q` represents the quality metrics of choice (e.g. accuracy) and `e` represents the energy cost.

All functions in this segment follow roughly the same pattern of naming, parameters and return types. Methods that evaluate configurations have the form similar to func:~*EnergyOptimizer.sca_model* - evaluates SCA configurations using the mathematical model. Methods that automatically find the configurations have the form similar to: func:~*EnergyOptimizer.find_sca_tradeoffs* - automatically finds SCA configurations.

class `eecr.eeoptimizer.`**`EnergyOptimizer`**(*sequence=None*, *contexts=None*, *setting_to_energy=None*, *setting_to_sequence=None*, *quality_metric=None*, *sensors_off=None*, *sensors_on=None*, *path=None*)

`dca_model`(*configurations*, *active=1*, *setting=None*, *cf=None*, *max_cycle=None*, *energy_costs=None*, *name=None*)

Tests the DCA configurations using the DCA mathematical model.

Parameters

- **`configurations`** – a list of DCA configurations. A configuration is a list of the same length as the number of contexts, each element being an integer >= 1. Optionally this parameter can be a single configuration instead of a list
- **`active`** – the length of the active period. For most purposes this length should be left as the default of 1
- **`setting`** – the setting that is in use while duty-cycling. If None is provided the function assumes that the classification accuracy is 100% and the energy cost is 1
- **`cf`** – uses the given confusion matrix (must be normalized, so each row sums to 1) instead of the one prescribed by the `setting` parameter
- **`max_cycle`** – the maximum duty-cycle length out of any found in configurations. If set the evaluation time will be slightly faster
- **`energy_costs`** – internal parameter (its value should not be changed)
- **`name`** – None or relative path to a file. In latter case the path will be used to save the generated configurations and trade-offs

Returns a list of trade-offs, in the form (quality, energy). i-th element represents the evaluation of the i-th configuration

dca_real(*configurations*, *setting=None*, *active=1*, *name=None*)

Tests the DCA configurations using a simulation.

This method reads from a sequence (either base one or one that corresponds to the given setting), skipping some in a way that simulates duty-cycling in a real-life environment. This is slower than using the mathematical model (`dca_model()`).

Parameters

- **configurations** – a list of DCA configurations. A configuration is a list of the same length as the number of contexts, each element being an integer >= 1. Optionally this parameter can be a single configuration instead of a list
- **name** – None or relative path to a file. In latter case the path will be used to save the generated configurations and trade-offs
- **setting** – the setting that is in use while duty-cycling. If None is provided the function assumes that the classification accuracy is 100% and the energy cost is 1
- **active** – the length of the active period. For most purposes this length should be left as the default of 1

Returns a list of trade-offs, in the form (quality, energy). i-th element represents the evaluation of the i-th configuration

find_dca_random(*n_samples=100*, *max_cycle=10*)

Returns `n_samples` random DCA configurations.

Parameters

- **n_samples** – number of configurations to return
- **max_cycle** – the maximum desired duty-cycle length for the configurations

Returns a list of configurations

find_dca_static(*max_cycle*, *name=None*, *setting=None*, *active=1*)

Returns all configurations where the same duty-cycle length is used for all contexts.

Parameters

- **max_cycle** – the maximum desired duty-cycle length for the configurations to be generated
- **name** – None or relative path to a file. In latter case the path will be used to save the generated configurations and trade-offs
- **setting** – the setting that is in use while duty-cycling. If None is provided the function assumes that the classification accuracy is 100% and the energy cost is 1
- **active** – the length of the active period. For most purposes this length should be left as the default of 1

Returns two lists, the first contains pareto-optimal configurations, the second their evaluations in the form (quality, energy).

find_dca_tradeoffs(*max_cycle=10*, *active=1*, *seeded=True*, *setting=None*, *cf=None*, *name=None*, *energy_costs=None*, *ngen=200*)

Attempts to find the best DCA trade-offs for the current dataset.

Parameters

- **max_cycle** – the maximum desired duty-cycle length for the configurations

- **active** – the length of the active period. For most purposes this length should be left as the default of 1
- **seeded** – if true, the search starts always start with two specific configuration in the starting population. One configuration uses the mininum and the other the maximum duty-cycle length
- **setting** – the setting that is in use while duty-cycling. If None is provided the function assumes that the classification accuracy is 100% and the energy cost is 1
- **cf** – uses the given confusion matrix (must be normalized, so each row sums to 1) instead of the one prescribed by the `setting` parameter
- **name** – None or relative path to a file. In latter case the path will be used to save the generated configurations and trade-offs
- **energy_costs** – internal parameter (its value should not be changed)
- **ngen** – the number of generations in the NSGA_II search

Returns two lists, the first contains pareto-optimal configurations, the second their evaluations in the form (quality, energy).

find_sca_dca_tradeoffs(*sca_configurations=None*, *sca_tradeoffs=None*, *dca_indices=None*, *n_points=5*, *binary_representation=False*, *name=None*, *max_cycle=10*, *active=1*, *verbose=False*, *cstree=False*)

Attempts to find the best SCA-DCA trade-offs for the current dataset.

Parameters

- **sca_tradeoffs** – if Pareto-optimal trade-ofs were already precalculated using *find_sca_tradeoffs()* they can be set using this parameter to avoid calculating them again
- **sca_configurations** – if `sca_tradeoffs` parameter is set, this parameter should list the configurations from which the `sca_tradeoffs` were generated
- **dca_indices** – index of the configurations in `sca_configurations` to be expanded using the DCA method. If not set, indices will be determined automatically by making them equidistant
- **n_points** – the number of sca configurations selected for expanding with the dca method
- **binary_representation** – a flag indicating that settings are represented by a binary list. In this case, a different mutation/crossover will be used for the NSGA-II algorithm
- **name** – None or relative path to a file. In latter case the path will be used to save the generated configurations and trade-offs
- **max_cycle** – the maximum desired duty-cycle length for the configurations
- **active** – the length of the active period. For most purposes this length should be left as the default of 1
- **verbose** – if true, the function prints out the current progress
- **cstree** – a flag indicating that cost-sensitive trees are used. Makes the evaluation more accurate if used

Returns two lists, the first contains pareto-optimal configurations, the second their evaluations in the form (quality, energy).

find_sca_random(*n_samples=100*)

Returns `n_samples` random SCA configurations.

Parameters `n_samples` – number of configurations to return

Returns a list of configurations

`find_sca_static`(*name=None*)

Returns all Pareto-optimal configurations where the same setting is used for all contexts.

Parameters `name` – None or relative path to a file. In latter case the path will be used to save the generated configurations and trade-offs

Returns two lists, the first contains pareto-optimal configurations, the second their evaluations in the form (quality, energy).

`find_sca_tradeoffs`(*binary_representation=False*, *name=None*, *cstree=False*)

Attempts to find best SCA trade-offs for the current dataset.

Uses the NSGA-II algorithm to search the space of different configurations and finds and returns the Pareto-optimal ones.

Parameters

- `binary_representation` – a flag indicating that settings are represented by a binary list. In this case, a different mutation/crossover will be used for the NSGA-II algorithm
- `name` – None or relative path to a file. In latter case the path will be used to save the generated configurations and trade-offs
- `cstree` – a flag indicating that cost-sensitive trees are used. Makes the evaluation more accurate if used

Returns two lists, the first contains pareto-optimal configurations, the second their evaluations in the form (quality, energy).

`sca_dca_model`(*configurations*, *active=1*, *cstree=False*, *name=None*)

Tests the SCA-DCA configurations using a simulation.

This combines the simulations from (see *sca_real()*) and (see *sca_real()*). It is slower than using the mathematical model (see *sca_dca_model()*).

Parameters

- `configurations` – a list of SCA-DCA configurations. A configuration can have the same syntax as the SCA configuration (see *sca_real()*) or it can be represented as a tuple, where the first element is a SCA configuration and the second element is a DCA configuration (see *sca_real()*)
- `active` – the length of the active period. For most purposes this length should be left as the default of 1
- `cstree` – a flag indicating that cost-sensitive trees are used. Makes the evaluation more accurate if used
- `name` – None or relative path to a file. In latter case the path will be used to save the generated configurations and trade-offs

Returns a list of trade-offs, in the form (quality, energy). i-th element represents the evaluation of the i-th configuration

`sca_dca_real`(*configurations*, *active=1*, *name=None*, *cstree_energy=False*)

Tests the SCA-DCA configurations using a simulation.

This combines the simulations from (see *sca_real()*) and (see *sca_real()*). It is slower than using the mathematical model (see *sca_dca_model()*).

Parameters

- **configurations** – a list of SCA-DCA configurations. A configuration can have the same syntax as the SCA configuration (see *sca_real()*) or it can be represented as a tuple, where the first element is a SCA configuration and the second element is a DCA configuration (see *sca_real()*)
- **active** – the length of the active period. For most purposes this length should be left as the default of 1
- **name** – None or relative path to a file. In latter case the path will be used to save the generated configurations and trade-offs
- **cstree_energy** – this flag slightly increases the accuracy of the simulation when using the cost-sensitive decision trees. In order to use it, the energy sequence must be precalculated, by using `cstree_energy` flag when generating the trees (e.g. in *add_csdt_weighted()* or *add_csdt_borders()*)

Returns a list of trade-offs, in the form (quality, energy). i-th element represents the evaluation of the i-th configuration

sca_model (*configurations*, *name=None*, *cstree=False*, *encrypted=False*)
Tests the SCA configurations using the SCA mathematical model.

Parameters

- **configurations** – a list of SCA configurations. A configuration is a list of the same length as the number of contexts, each element being a setting. Optionally this parameter can be a single configuration instead of a list
- **name** – None or relative path to a file. In latter case the path will be used to save the generated configurations and trade-offs
- **cstree** – a flag indicating that cost-sensitive trees are used. Makes the evaluation more accurate if used
- **encrypted** – a flag for internal use (do not change its value)

Returns a list of trade-offs, in the form (quality, energy). i-th element represents the evaluation of the i-th configuration

sca_real (*configurations*, *name=None*, *cstree_energy=False*)
Tests the SCA configurations using a simulation.

This method reads from different sequences (that correspond to different settings), switching between them as context changes. This simulates a real-life environment where contexts are classified one-by-one using different system settings. This is slower than using the mathematical model instead (*sca_model()*).

Parameters

- **configurations** – a list of SCA configurations. A configuration is a list of the same length as the number of contexts, each element being a setting. Optionally this parameter can be a single configuration instead of a list
- **name** – None or relative path to a file. In latter case the path will be used to save the generated configurations and trade-offs
- **cstree_energy** – this flag slightly increases the accuracy of the simulation when using the cost-sensitive decision trees. In order to use it, the energy sequence must be precalculated, by using `cstree_energy` flag when generating the trees (e.g. in *add_csdt_weighted()* or *add_csdt_borders()*)

Returns a list of trade-offs, in the form (quality, energy). i-th element represents the evaluation of the i-th configuration

sca_simple(*configurations*, *name=None*)

Tests the SCA configurations using a simple mathematical model.

The model used simply multiplies the performance of each setting with the expected proportion of time that setting is in use. E.g if configuration consists of two settings, one with accuracy 60% and other with 100%, and if both contexts appear equally often, then the expected accuracy is 80%.

Parameters

- **configurations** – a list of SCA configurations. A configuration is a list of the same length as the number of contexts, each element being a setting. Optionally this parameter can be a single configuration instead of a list
- **name** – None or relative path to a file. In latter case the path will be used to save the generated configurations and trade-offs

Returns a list of trade-offs, in the form (quality, energy). i-th element represents the evaluation of the i-th configuration

2.4 Saving and loading data

To avoid repeated processing of the data, many components can be quickly saved and loaded. Many functions have this functionality already built-in by using the `name` parameter. In addition, the following functions are specialized for the task. Make sure the path is correctly set before using them. Additionally note, that all saving and loading uses pickle in the background.

class eecr.eeoptimizer.**EnergyOptimizer**(*sequence=None*, *contexts=None*, *setting_to_energy=None*, *setting_to_sequence=None*, *quality_metric=None*, *sensors_off=None*, *sensors_on=None*, *path=None*)

load_config(*name*)

Loads the settings information from a file (instead of calling *set_settings()*)

Parameters **name** – relative path to a file

load_data(*name*)

Loads the base sequence from a file (instead of calling *set_sequence()*).

Parameters **name** – relative path to a file

load_data_config(*data_name=None*, *config_name=None*, *sample_dataset=None*)

Loads the sequence and settings information from files.

This is equivalent to sequentially calling the *load_data()* and *load_config()* functions.

Parameters

- **data_name** – relative path to a file with the sequence
- **config_name** – relative path to a file with the settings information
- **sample_dataset** – if specified, a sample dataset will be loaded (currently only "SHL" keyword works)

load_solution(*name*)

Loads a set of energy-efficient solutions from a file.

Parameters **name** – relative path to a file

Returns two lists representing different tradeoffs. First list contains the configurations, while the other their evaluations. Evaluations are represented by a tuple with the form (quality, energy)

`save_config`(*name*)
Save the settings information (set with `set_settings()`) to a file.

Parameters **`name`** – relative path to a file

`save_data`(*name*)
Saves the base sequence set with `set_settings()` to a file.

Parameters **`name`** – relative path to a file

`save_solution`(*configurations*, *values*, *name*)
Saves a set of energy-efficient solutions from a file.

Parameters

- **`configurations`** – a list of configurations
- **`values`** – a list of configuration evaluation in form of (quality, energy)
- **`name`** – relative path to a file

2.5 Automatic settings generators

Sometimes the generation of settings can be automated. Two of the methods listed (`add_csdt_weighted()`, `add_csdt_borders()`) automate only settings represented by the cost-sensitive decision trees. The last one (`add_subsets()`), automates the task of creating settings where each setting represents using a different attribute subset.

class `eecr.eeoptimizer.`**`EnergyOptimizer`**(*sequence=None*, *contexts=None*, *setting_to_energy=None*, *setting_to_sequence=None*, *quality_metric=None*, *sensors_off=None*, *sensors_on=None*, *path=None*)

`add_csdt_borders`(*cs_tree*, *x1*, *y1*, *x2*, *y2*, *x_p=None*, *y_p=None*, *test_fn=None*, *buffer_range=1*, *name=None*, *cstree_energy=False*, *verbose=False*, *weights_range=None*, *energy_range=None*, *n_tree=15*)
Generates different CS-DTs from data around different contexts.

When generating these cost-sensitive decision trees, only data from one contexts (and instances around it) are taken. This should create a tree that is good at recognizing that context and contexts to which it frequently transitions to. This method also tries different weights, same as the `add_csdt_weighted()` method. Generated CS-DT get added to the list of possible settings.

Parameters

- **`cs_tree`** – the base cost-sensitive tree object (`eecr.cstree.CostSensitiveTree`) to use for generation of its variants
- **`x1`** – a pandas dataframe of attributes to be used as the training set
- **`y1`** – a pandas series of labels for the instances in `x1`
- **`x2`** – a pandas dataframe of attributes to be used as the testing set
- **`y2`** – a pandas series of labels for the instances in `x2`
- **`x_p`** – (optional) a pandas dataframe of attributes to be used as the pruning set

- **y_p** – (optional) a pandas series of labels for the instances in `x_p`
- **buffer_range** – specifies how many instances to take after a contexts transitions
- **test_fn** – a function used for measuring energy while testing a CS-DT, if it need to be different than the one used for training
- **verbose** – if true, the function prints out the current progress
- **weights_range** – a tuple with two elements, representing the minimum and maximum weight to be used. Trees will use `n_tree` different weights equidistantly sampled between these two extreme values. Sensible weight range must be determined manually, but as a heuristic, it is usually inversely proportional to the energy costs. For example if energy costs are all >1 then weights should all be <1.
- **energy_range** – An alternative to the `weights_range` parameter, setting this to x is equivalent to setting `weights_range` to 1/n
- **n_tree** – number of different CS-DT to be generated
- **name** – None or relative path to a file. If latter, the generated settings will be saved
- **cstree_energy** – Generate accurate energy sequences that can be used to increase the accuracy of the *`sca_real()`*. It makes this function much slower

Returns the set of generated CS-DTs

add_csdt_weighted(*cs_tree, x1, y1, x2, y2, x_p=None, y_p=None, test_fn=None, verbose=True, weights_range=None, energy_range=None, n_tree=15, name=None, cstree_energy=False*)

Generates different CS-DTs with different ratios between energy and classification quality.

When generating cost-sensitive decision trees, at each node a decision is made on whether an attribute is worth using (based on its informativeness and cost). Different weights can be used to skew the decision in one way or another. Generated CS-DT get added to the list of possible settings.

Parameters

- **cs_tree** – the base cost-sensitive tree object (*`eecr.cstree.CostSensitiveTree`*) to use for generation of its variants
- **x1** – a pandas dataframe of attributes to be used as the training set
- **y1** – a pandas series of labels for the instances in `x1`
- **x2** – a pandas dataframe of attributes to be used as the testing set
- **y2** – a pandas series of labels for the instances in `x2`
- **x_p** – (optional) a pandas dataframe of attributes to be used as the pruning set
- **y_p** – (optional) a pandas series of labels for the instances in `x_p`
- **test_fn** – a function used for measuring energy while testing a CS-DT, if it need to be different than the one used for training
- **verbose** – if true, the function prints out the current progress
- **weights_range** – a tuple with two elements, representing the minimum and maximum weight to be used. Trees will use `n_tree` different weights equidistantly sampled between these two extreme values. Sensible weight range must be determined manually, but as a heuristic, it is usually inversely proportional to the energy costs. For example if energy costs are all >1 then weights should all be <1.
- **energy_range** – An alternative to the `weights_range` parameter, setting this to x is equivalent to setting `weights_range` to 1/n

- **`n_tree`** – number of different CS-DT to be generated
- **`name`** – None or relative path to a file. If latter, the generated settings will be saved
- **`cstree_energy`** – Generate accurate energy sequences that can be used to increase the accuracy of the *`sca_real()`*. It makes this function much slower

Returns the set of generated CS-DTs

`add_subsets`(*x1*, *y1*, *x2*, *y2*, *classifier*, *setting_to_energy=None*, *setting_fn_energy=None*, *subsettings=None*, *subsetting_to_features=None*, *n=0*, *feature_groups=None*, *setting_fn_features=None*, *name=None*, *y_p=None*, *x_p=None*, *csdt=False*, *csdt_fn_energy=None*, *cstree_energy=False*)

Automatically generates settings by taking different attribute subsets from a database

This method assumes you have a pandas dataframe where each column represent a different attribute. Each row represents an instance to be classify. However, each attribute has a cost and thus the goal is to classify the instances with as small subset as possible. The cost is often shared between the attributes: e.g. if two attributes are calculated from the GPS stream, then having one or both has the same cost (the cost of having the GPS open). It may also be the case that using some sensor increases or decreases the cost of another sensor as they share resources when used.

There are several ways of specifying which subsets to transform into settings. Use the one most convenient for the current domain. In all examples we will assume that data comes from different sensors (subsettings) and a setting is set of sensors that is in use. If the data (and costs) comes from different sources a similar logic applies.

1. Set `subsettings` parameter as a list of sensors and `subsetting_to_features` as a dictionary that maps each sensor to attribute list (attributes calculated from that sensor)
2. Set `feature_groups` as a list of lists. Each list represents features that should all be included or excluded in any attribute subset used (e.g. they come from the same sensor). Set `n` as the number of `feature_groups`.
3. Set `n` as the number of sensors, set `setting_fn_features` as a function that takes a binary string of length `n` and outputs a list of attributes. Character i in binary string represents whether i-th sensor is active.

This method is not a substitute for attribute selection.

Parameters

- **`x1`** – a pandas dataframe of attributes to be used as the training set
- **`y1`** – a pandas series of labels for the instances in `x1`
- **`x2`** – a pandas dataframe of attributes to be used as the testing set
- **`y2`** – a pandas series of labels for the instances in `x2`
- **`x_p`** – (optional) a pandas dataframe of attributes to be used as the pruning set
- **`y_p`** – (optional) a pandas series of labels for the instances in `x_p`
- **`classifier`** – any classifier (tested with sklearn and CS-DTs). Should provide functions “fit” and “predict”
- **`setting_to_energy`** – a dictionary, where the keys are the possible settings and the values are the energy costs (per time unit) when using that setting. Settings are binary string
- **`setting_fn_energy`** – a function to be used if the `setting_to_energy` is None. This function should take a setting as the input and return energy costs (per time unit) when using that setting. Settings are binary string

- **`subsettings`** – the list of subsettings
- **`subsetting_to_features`** – a dictionary that maps subsettings to attribute sets
- **`n`** – the number of subsettings
- **`feature_groups`** – a list of lists, elements of the sublists are always all included or excluded
- **`setting_fn_features`** – a function that maps settings to attribute sets
- **`name`** – None or relative path to a file. If latter, the generated settings will be saved
- **`x_p`** – (optional) a pandas dataframe of attributes to be used as the pruning set
- **`y_p`** – (optional) a pandas series of labels for the instances in `x_p`
- **`csdt`** – if the classifier is a CS-DT set this flag to true
- **`csdt_fn_energy`** – a function used for measuring energy while testing a CS-DT, if it need to be different than the one used for training
- **`cstree_energy`** – If using CS-DTs it generates accurate energy sequences that can be used to increase the accuracy of the *`sca_real()`*. It makes this function much slower

Returns a list of generated classifiers

2.6 Visualizing solutions

Most functions in the ee.eeutility module are meant for internal use. However, the `draw_tradeoffs()` provides an easy way to visualize the results achieved with other methods.

`eecr.eeutility.`**`draw_tradeoffs`**(*plots*, *labels*, *xlim=None*, *ylim=None*, *name=None*, *reverse=True*, *pareto=False*, *short=False*, *points=None*, *folder='artificial'*, *percentage=True*, *percentage_energy=False*, *scatter_indices=None*, *color_indices=None*, *text_factor=50*, *ylabel='Energy'*, *dotted_indices=None*, *thick_indices=None*, *xlabel='Classification error'*)

A tool for quick visualization of different trade-offs.

Essentially a wrapper around matplotlib.pyplot that makes drawing and comparing different sets of trade-offs easier. As an input it expects a list of trade-offs and can plot them in the way that is standard for Pareto fronts: step-wise with the quality axis reversed so that the ideal point lies in the lower-left corner. It streamlines some other aspects, for example labeling, saving and making sure only Pareto-optimal points are drawn.

While some utility parameters are presents for modifying the look of the graph, it is recommended to use matplotlib.pyplot library directly for any complex drawing task.

Parameters

- **`plots`** – a list of one or more trade-off sets. Each trade-off set is for example a result of a different energy-optimization function. They can be represented in two different ways. Either as list of tuples (each tuples representing quality, energy) or a tuple of two lists (first list containing quality, second energy). All outputs returned from energy-optimization functions already fit this criteria.
- **`labels`** – a list of labels in the same order as the trade-offs in the `plots` parameter
- **`xlim`** – a tuple with min and max for x-axis
- **`ylim`** – a tuple with min and max for y-axis

- **name** – None or name of a file where the figure is saved
- **reverse** – reverses the x-axis
- **pareto** – only Pareto-optimal points are drawn
- **short** – if true, the graph will be drawn as square instead of a rectangle
- **points** – a list of points of to be drawn [(x1,y1,s1), (x2,y2,s2)]. s1, s2, etc. are optional and provide a way to annotate points
- **folder** – None or name of a folder where the figure is saved
- **percentage** – multiplies the x-axis by 100 (in order to transform the accuracy of 0.45 into 45%)
- **percentage_energy** – multiplies the x-axis by 100
- **scatter_indices** – indices of trade-offs sets that should be drawn un-connected
- **color_indices** – if not None, each element of this list determines the index of a color (trade-offs with the same index will be drawn with the same color)
- **text_factor** – changing this parameter moves the label from/away the annotated `points`
- **ylabel** – label for the y-axis
- **dotted_indices** – indices of trade-offs sets that should be drawn with dots
- **thick_indices** – indices of trade-offs sets that should be drawn thicker than the rest
- **xlabel** – label for the x-axis

2.7 Alternative search methods

Methods below are based on the related work, but solve the same kind of problem as the methodology presented in this module. They are here included as possible alternatives and for easier comparison of the approaches.

class `eecr.eeoptimizer.`**EnergyOptimizer**(*sequence=None*, *contexts=None*, *setting_to_energy=None*, *setting_to_sequence=None*, *quality_metric=None*, *sensors_off=None*, *sensors_on=None*, *path=None*)

find_aimd_tradeoffs(*increase_range=None*, *decrease_range=None*, *name=None*, *active=1*)

Solves the duty-cycle assignment problem in a different ways than the DCA methods

Imagine a system with parameters `inc` and `dec` that works as follows: 1.) System duty-cycles with a cycle of length `len`. 2.) If the next detected context is the same as the previous one: `len = len + inc`. 3.) If the next detected context is different than the previous one: `len = len * dec`. This method finds good combinations of `inc` and `dec` from a specified range.

This method is based on paper: Au, Lawrence K., et al. "Episodic sampling: Towards energy-efficient patient monitoring with wearable sensors." 2009 Annual International Conference of the IEEE Engineering in Medicine and Biology Society. IEEE, 2009.

Parameters

- **increase_range** – a list of possible `inc` values
- **decrease_range** – a list of possible `dec` values

- **name** – None or relative path to a file. In latter case the path will be used to save the generated configurations and trade-offs
- **active** – the length of the active part of the duty-cycle

Returns two lists, the first contains pareto-optimal configurations in the form (`inc`, `dec`), the second their evaluations in the form (quality, energy).

find_coh_tradeoffs(*alphas=None*, *name=None*)
Finds the SCA configurations based on context transition probabilities.

While the method behind this is completely different than `find_sca_tradeoffs()`, it returns the same kind of output, including the same type of configurations.

This method is based on paper: Gordon, Dawud, Jürgen Czerny, and Michael Beigl. “Activity recognition for creatures of habit.” Personal and ubiquitous computing 18.1 (2014): 205-221.

Parameters

- **alphas** – a list of different values of parameter alpha to test (0 <= alpha <=1), see the paper for details
- **name** – None or relative path to a file. In latter case the path will be used to save the generated configurations and trade-offs

Returns two lists, the first contains pareto-optimal configurations, the second their evaluations in the form (quality, energy).

find_simple_tradeoffs(*name=None*)
Finds the SCA configurations based on simple mathematical model.

It works fast, but the results are often much worse than the alternatives. Finds either one or two trade-offs.

Parameters **name** – None or relative path to a file. In latter case the path will be used to save the generated configurations and trade-offs

Returns two lists, the first contains pareto-optimal configurations, the second their evaluations in the form (quality, energy).

2.8 Cost-sensitive decision trees

Cost-sensitive trees are a version of the decision tree classifier that is interested in both attribute informativeness and attribute costs when building the tree. To explain it briefly: at each tree node – instead of checking each attribute’s information gain or a similar metric – we compare the expected cost of misclassification if this node becomes a leaf, with the expected cost of the attribute and misclassification if the attribute is used to further divide the instances. The attribute that reduces the expected cost the most, if any, then expands this node.

This particular implementation of the cost-sensitive trees is optimized for the tasks in context-recognition. It can be used as a stand-alone method for reducing the energy consumption or it can be used in conjuction with other methods described in this module.

class eecr.cstree.**CostSensitiveTree**(*contexts*, *cost_function*, *feature_to_sensor=None*, *feature_groups=None*, *tree_type='default'*, *min_samples=1*, *extension=0*, *default=None*, *weight=1*)

__init__(*contexts*, *cost_function*, *feature_to_sensor=None*, *feature_groups=None*, *tree_type='default'*, *min_samples=1*, *extension=0*, *default=None*, *weight=1*)
Constructor for the cost-sensitive decision tree.

Either (but not both) `feature_to_sensor` or `feature_groups` must be set.

Parameters

- **`contexts`** – a list of contexts to be recognized
- **`cost_function`** – maps a set of sensors into the energy cost
- **`feature_to_sensor`** – a dictionary that maps each attribute to a sensor
- **`feature_groups`** – a dictionary that maps each sensor to a list of attributes
- **`tree_type`** – there are three types of trees that can be generated: "default", "pruned" and "batched". The pruned version prunes the tree using a dedicated pruning set after tree generation. "batched" version tries to place attributes that share energy costs as close together as possible. This results in a longer train time, but usually slightly better performance
- **`min_samples`** – the minimum number of samples in a non-leaf node
- **`extension`** – if this is set as >0 the tree building process will sometimes expand nodes even if energy-inefficient in anticipation that the decision will pay off later in the tree. It prunes these branches if the anticipation proves to be wrong. This results in a longer train time, but usually slightly better performance (in practice extension=1 proved best)
- **`default`** – context to be classified in case of an empty tree
- **`weight`** – weight of the energy cost when compared to the misclassification cost

`fit`(*x1*, *y1*, *x_p=None*, *y_p=None*)
Trains the tree classifier

Parameters

- **`x1`** – a pandas dataframe of attributes to be used as the training set
- **`y1`** – a pandas series of labels for the instances in `x1`
- **`x_p`** – (optional) a pandas dataframe of attributes to be used as the pruning set
- **`y_p`** – (optional) a pandas series of labels for the instances in `x_p`

`predict`(*x2*)
Tests the tree classifier

Parameters **`x2`** – a pandas dataframe of attributes to be used as the test set

Returns a list of predictions, where the i-th element is the prediction for i-th row in x2

`show`()
Prints the structure of the tree.

`show_sensors`()
Prints the structure of the tree, abstracted so only different sensors are shown.

CHAPTER

THREE

METHOD LIST

set_sequence	Sets the sequence of contexts that will be used for testing the energy-efficiency of settings.
set_path	Sets the path to a folder from which the data is loaded and to which data is saved.
set_settings	Defines the possible settings and their performance/energy for the optimization.
set_dca_costs	Sets the base energy costs for duty-cycling.
sca_real	Tests the SCA configurations using a simulation.
sca_simple	Tests the SCA configurations using a simple mathematical model.
sca_model	Tests the SCA configurations using the SCA mathematical model.
find_sca_tradeoffs	Attempts to find best SCA trade-offs for the current dataset.
find_sca_static	Returns all Pareto-optimal configurations where the same setting is used for all contexts.
find_sca_random	Returns `n_samples` random SCA configurations.
dca_real	Tests the DCA configurations using a simulation.
dca_model	Tests the DCA configurations using the DCA mathematical model.
find_dca_static	Returns all configurations where the same duty-cycle length is used for all contexts.
find_dca_tradeoffs	Attempts to find the best DCA trade-offs for the current dataset.
find_dca_random	Returns `n_samples` random DCA configurations.
sca_dca_real	Tests the SCA-DCA configurations using a simulation.
sca_dca_model	Tests the SCA-DCA configurations using a simulation.
find_sca_dca_tradeoffs	Attempts to find the best SCA-DCA trade-offs for the current dataset.
load_data_config	Loads the sequence and settings information from files.
load_data	Loads the base sequence from a file (instead of calling `set_sequence()`).
load_config	Loads the settings information from a file (instead of calling `set_settings()`)
load_solution	Loads a set of energy-efficient solutions from a file.
save_data	Saves the base sequence set with `set_settings()` to a file.
save_solution	Saves a set of energy-efficient solutions from a file.

Continued on next page

Table 1 – continued from previous page

save_config	Save the settings information (set with `set_settings()`) to a file.
add_subsets	Automatically generates settings by taking different attribute subsets from a database
add_csdt_weighted	Generates different CS-DTs with different ratios between energy and classification quality.
add_csdt_borders	Generates different CS-DTs from data around different contexts.

CHAPTER

FOUR

CODE EXAMPLES

4.1 Base case

4.1.1 Description

This module helps to create energy efficient solutions for a context recognition problem. It assumes that the problem can be solved using different system's settings (e.g. different frequencies, different feature sets, different sensor subsets etc.) Each of these settings has a different energy cost, but also different classification quality (e.g. accuracy). Settings can be switched in runtime (e.g. using low acceleromter frequency when user is resting, but high when they are running), and this module is tasked with finding good rules for switching between them. It does so in a domain independent way by analyzing the statistical properties of the dataset

First part of the required input is a sequence of contexts (contexts are the classes of the classification system: e.g walking, standing, running in an activity recognition system). This sequence should be representative in both context proportions and their ordering to the sequences expected in the target domain.

In this toy example the contexts are simply numbers 1-3 (they could be strings instead). They represent a domain when each context appears 4 times and then transitions to the one represented by the next number.

```
[3]: sequence = [0,0,0,0,  1,1,1,1,  2,2,2,2, 0]
```

In this hypothetical case, we have 6 different settings named "a"-"f". Their meaning is irrelevant to this module (perhaps they each represent a different sensor that could be used) and they can be presented by any hashable type. With each setting we classify the data from which the previous sequence was taken and store the results in a dictionary. We also supply the energy cost for using that setting for a one time unit.

```
[4]: classified = {}
     classified["a"] = [0,0,0,0,  1,1,1,1,  2,2,2,2, 0]
     classified["b"] = [1,1,1,1,  1,1,1,1,  1,1,1,1, 1]
     classified["c"] = [0,0,0,0,  2,1,2,1,  2,2,2,0, 0]
     classified["d"] = [0,0,0,0,  1,1,1,2,  0,1,2,2, 0]
     classified["e"] = [0,0,0,0,  0,0,0,0,  2,2,2,2, 0]
     classified["f"] = [1,1,1,1,  1,1,1,1,  2,2,2,2, 1]

     energy = {"a":3, "b":1, "c":2, "d":2, "e":1, "f":1}
```

A brief commentary on the classified sequences: Using setting "a" generated the same sequence to the original one, representing perfect classification. Its energy cost, however, its higher than all others. Using setting "b" every instance was classified as "1", but its energy cost was the low. Using setting "c" contexts "0" and "2" are accurately classified, while the context "1" is not. Using setting "d" contexts "0" and "1" are accurately classified, while the context "2" is not. Similarly for settings "e" and "f".

All three input paramters are entered into the EnergyOptimizer object.

```
[1]: from eecr import EnergyOptimizer
     from eecr import eeutility as util
```

```
[5]: optimizer = EnergyOptimizer(sequence=sequence,
                                 setting_to_sequence=classified,
                                 setting_to_energy=energy)
```

4.1.2 Data summary

The EnergyOptimizer assumes that the context sequence has the Markov property and in transforms the sequences into the mathematical description of corresponding Markov chains. Some basic properties can be shown.

```
[6]: optimizer.summary()
```

```
   Proportions  Average lengths
0     0.333333              4.0
1     0.333333              4.0
2     0.333333              4.0
```

For each setting, we can check its accuracy and energy consumption (they are shown in a tuple in this order). Accuracy can be substituted with any other quality metric (e.g. f-score) if needed.

```
[7]: optimizer.energy_quality()
```

```
[7]: {'a': (1.0, 3),
      'b': (0.3333333333333333, 1),
      'c': (0.75, 2),
      'd': (0.75, 2),
      'e': (0.6666666666666667, 1),
      'f': (0.6666666666666667, 1)}
```

If the desired context recognition system would always use one setting, only three out of six would come into consideration (printed below). Others are pareto dominated (they are worse in both accuracy and energy than one of the other solutions).

```
[8]: solutions, values = optimizer.find_sca_static()
     for (s,v) in zip(solutions, values):
         print (s, v)
```

```
a (1.0, 3)
c (0.7692307692307693, 2)
e (0.6923076923076923, 1)
```

4.1.3 SCA method

Suppose now, that we wish to change the setting based on the current context. E.g. if context 0 is detected, use setting “a”, but for contexts “1” and “2” use settings “d” and “e” respectively. The expected performance of such assignment can be mathematically modeled using the “sca_model” method.

```
[9]: optimizer.sca_model(["d","f","e"])
```

```
[9]: [(0.9535747446610956, 1.3514391829155061)]
```

Using this context switching scheme we could achieve 95% accuracy and on average consume 1.35 units of energy.

Note we assigned to use setting “d” when context 0 was detected. Since using this setting both “0” and “1” context are well classified (“1” always succeeds “0”), the assignment yield good results. If we used setting “d” that is good for recognizing contexts “0” and “2” instead, the results worsen.

```
[10]: optimizer.sca_model(["c","f","e"])
```

```
[10]: [(0.8952062430323299, 1.3801560758082498)]
```

The mathematical model (beside being much faster then a simulation on a large dataset) can capture some performance anomalies that would otherwise go undetected. For example, if setting “b” is assigned to context “1”, all classifications from this point on wozld be “1”. This is easy to miss when simulating on this short sequence (sca_real) but is detected by the mathematical model and assigned an appropriate low accuracy.

```
[11]: optimizer.sca_real(["a","b","a"])
```

```
[11]: [(0.6153846153846154, 1.6153846153846154)]
```

```
[12]: optimizer.sca_model(["a","b","a"])
```

```
[12]: [(0.3333333333333332, 1.0)]
```

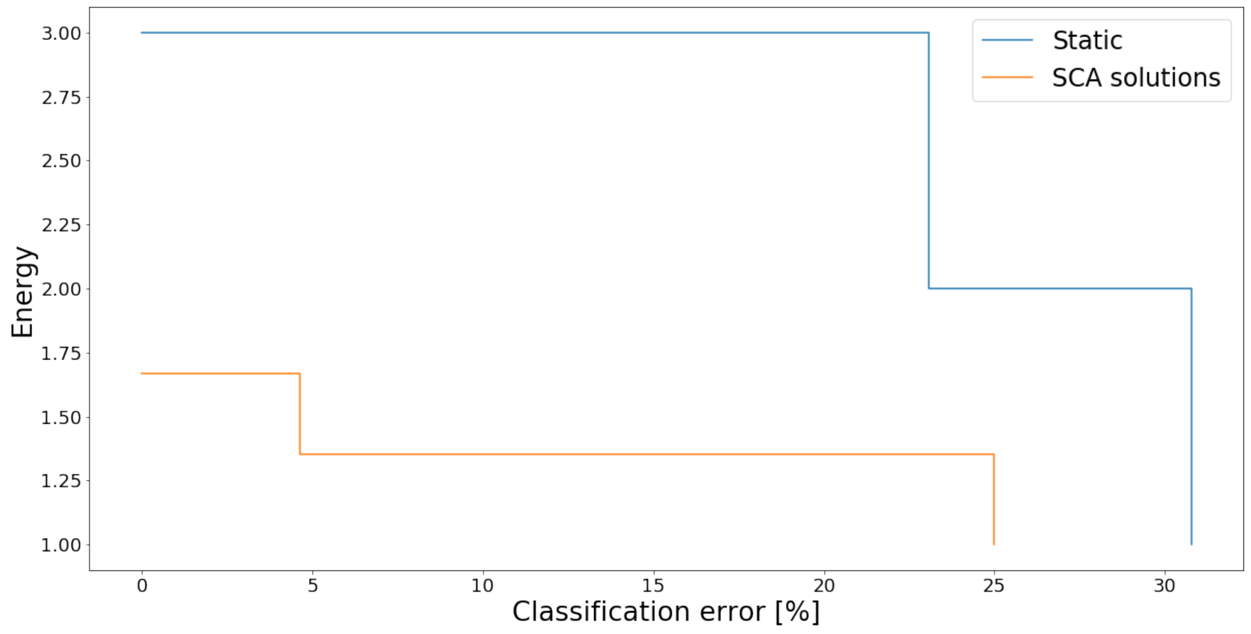
Good assignments can be found automatically using the “find_sca_tradeoffs” function.

```
[13]: solutions_sca, values_sca = optimizer.find_sca_tradeoffs()
      for (s,v) in zip(solutions_sca, values_sca):
          print (s, v)
```

```
['a', 'f', 'e'] (1.0, 1.6666666666666667)
['d', 'f', 'e'] (0.9535747446610956, 1.3514391829155061)
['f', 'f', 'e'] (0.7500000000000002, 1.0)
['b', 'f', 'e'] (0.7500000000000002, 1.0)
```

Performance can be easily visulized. Lower left corner presents the best solutions. It is immediately appearent that the solutions that use setting switching (SCA solutions) are better than those where the same one is always used (Static).

```
[14]: util.draw_tradeoffs([values, values_sca], ["Static", "SCA solutions"])
```



4.1.4 DCA method

Aside of using different settings, one can also use duty cycling. Duty-cycling means that sensors work for a given amount of time, than turn off for a given amount of time and repeat. Similar to the SCA setting switching case, the length of the sleeping period can be dynamically adjusted to the last classified context.

The functions that model the performance of the duty-cycling are similar, but use dca prefix instead of sca. The call below, for example, models how the system would beheave if duty-cycle length when activity "0" is detected would be 1 and so on. Note that duty-cycle length count period in both working and sleeping part of the cycle, thus the minimum for this parameter is 1 (1 period working, 0 periods sleeping).

```
[15]: optimizer.dca_model([1,2,3], setting = "a")
```

```
[15]: [(0.8682170542635659, 1.6976744186046506)]
```

Again, the solutions can be automatically found (only a subset is printed).

```
[16]: solutions_dca, values_dca = optimizer.find_dca_tradeoffs(setting = "d", max_cycle=5)
```

```
[17]: for (s,v) in list(zip(solutions_dca, values_dca))[0:8]:
          print (s, v)
```

```
[1, 1, 1] (0.7500000000000001, 2.0)
[1, 1, 2] (0.7336523125996811, 1.6299840510366825)
[1, 2, 1] (0.7214035087719299, 1.5564912280701755)
[1, 2, 2] (0.7209523809523811, 1.3447619047619046)
[1, 2, 3] (0.7018874994145473, 1.1944171233197507)
[1, 3, 2] (0.6903486141430177, 1.1702057286498644)
[1, 3, 3] (0.6790176340401228, 1.0633254801170842)
[2, 2, 2] (0.6770833333333334, 1.0)
```

4.1.5 SCA + DCA method combination

Both axis of optimizations can be joined togeteher, simultaneously switching between both the length of the duty-cycle and different system's settings.

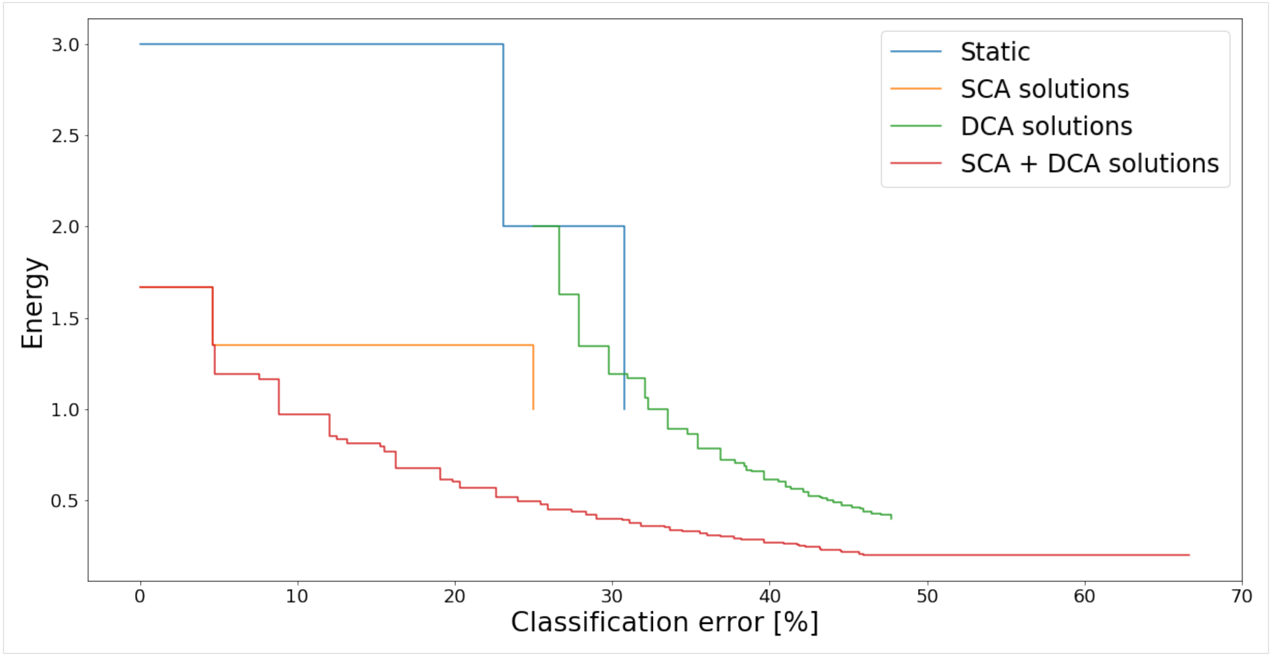
```
[18]: solutions_sca_dca, values_sca_dca = optimizer.find_sca_dca_tradeoffs(max_cycle=5)
```

```
[19]: for (s,v) in list(zip(solutions_sca_dca, values_sca_dca))[0:8]:
          print (s, v)
```

```
['a', 'f', 'e'] (1.0, 1.6666666666666667)
(['d', 'f', 'e'], [1, 1, 1]) (0.9535747446610957, 1.3514391829155064)
['d', 'f', 'e'] (0.9535747446610956, 1.3514391829155061)
(['a', 'f', 'e'], [2, 1, 1]) (0.9523809523809523, 1.1904761904761905)
(['d', 'f', 'e'], [1, 2, 1]) (0.9245284950744742, 1.1644404584582084)
(['a', 'f', 'e'], [2, 1, 2]) (0.9115646258503401, 0.9727891156462587)
(['d', 'f', 'e'], [2, 2, 1]) (0.8793818815385329, 0.8507989372176622)
(['a', 'f', 'e'], [2, 2, 2]) (0.875, 0.8333333333333333)
```

Solutions from all methods are visualized in the graph below. The method combination (as usual) produced the most energy-efficient solutions.

```
[20]: util.draw_tradeoffs([values, values_sca, values_dca, values_sca_dca],
                          ["Static", "SCA solutions", "DCA solutions", "SCA + DCA solutions"])
```



4.2 Test on real-life SHL dataset

Load the relevant parts of the dataset - the sequence of contexts, settings, their energy cost and misclassifications using each of the settings.

```
[21]: optimizer = EnergyOptimizer()
      optimizer.load_data_config(sample_dataset="SHL")
```

There are eight different contexts in this dataset.

```
[22]: optimizer.contexts
```

```
[22]: ['Bike', 'Bus', 'Car', 'Run', 'Still', 'Subway', 'Train', 'Walking']
```

```
[23]: optimizer.sequence
```

```
[23]: 0          Run
      1        Still
      2        Still
      3        Still
      4        Still
               ...
      4072      Bike
      4073      Bike
      4074      Bike
      4075      Bike
      4076     Train
      Name: Context, Length: 4077, dtype: object
```

Settings are sensor subset used. They are encoded in a binary tuple, where each element represents the existance of one sensors. Sensor order in tuple: accelerometer, magnetometer, barometer, pressure sensor, orientation sensor. The energy cost of a setting is the number of sensors * 10

[24]:
```
optimizer.setting_to_energy[(1,1,0,0,0)]
```

[24]:
```
20
```

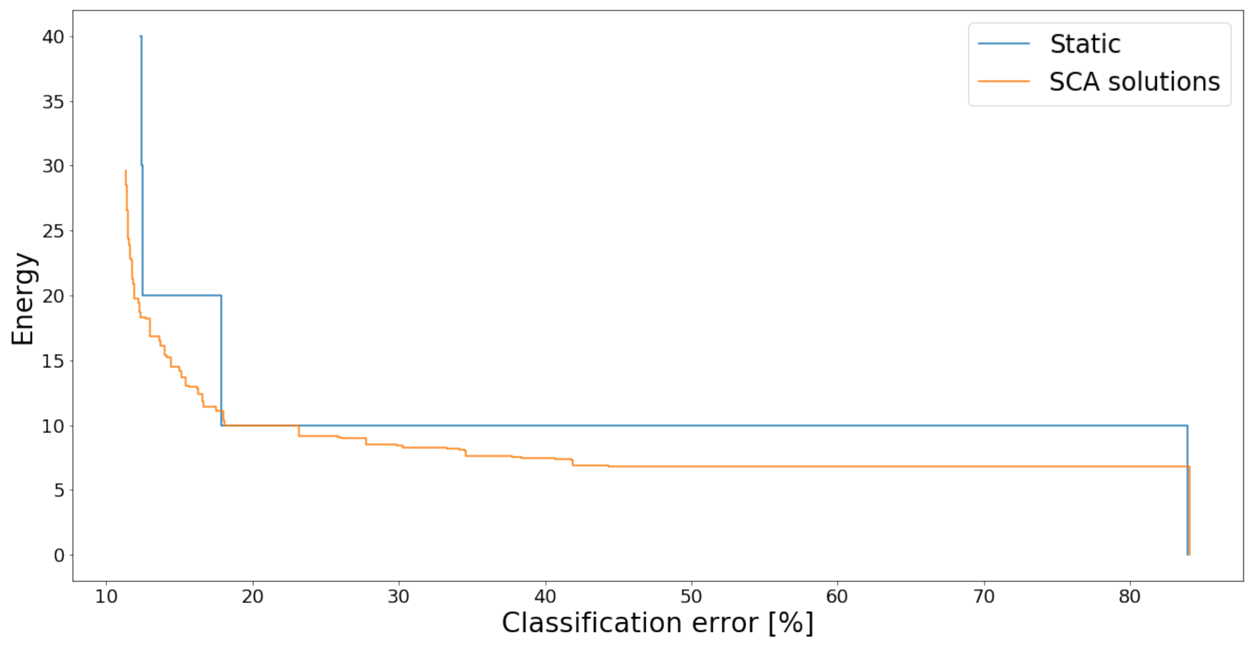
[25]:
```
#The context as predicted using only accelerometer and magnetometer
optimizer.setting_to_sequence[(1,1,0,0,0)]
```

[25]:
```
0         Run
1         Bus
2         Bus
3       Still
4       Still
         ...
4072     Bike
4073     Bike
4074     Bike
4075     Bike
4076    Train
Name: 11000, Length: 4077, dtype: object
```

Now we can run the exact same methods as in the artifical example case

[26]:
```
solutions, values = optimizer.find_sca_static()
solutions_sca, values_sca = optimizer.find_sca_tradeoffs()
```

[27]:
```
util.draw_tradeoffs([values, values_sca],
                    ["Static", "SCA solutions"])
```



Display a SCA assignment is a nice formatted way.

[47]:
```
sensor_list = ["acc","mag","pressure","gyr","ori"]
def display_assignment(index, letter=""):
    config,acc,energy = solutions_sca[index], values_sca[index][0], values_
↪sca[index][1]
    length = len(sensor_list)
```

(continues on next page)

(continued from previous page)

```
    print (" "*9+" ".join(sensor_list))
    for i,a in enumerate(optimizer.contexts):
        cfg = " ".join([("{0:>"+str(len(s))+"}").format(str(config[i][j])) for j,s
                        in enumerate(sensor_list)])
        print ("%7s: %s" % (a, cfg))
    print ("Accuracy: %1.3f     Energy: %1.1f"%(acc,energy))
    if len(letter)>0:
        return (acc,energy, letter)
```

[29]:
```
display_assignment(50)
```

```
         acc mag pressure gyr ori
   Bike:   1   0        0   0   0
    Bus:   1   1        0   0   0
    Car:   1   1        0   0   0
    Run:   1   0        0   0   0
  Still:   1   0        0   0   0
 Subway:   1   0        0   0   0
  Train:   1   1        0   0   0
Walking:   1   0        0   0   0
Accuracy: 0.856     Energy: 14.5
```

4.3 Cost-sensitive decision trees

The examples above showed how to change sensing settings based on the last classified context. An alternative is to change such settings based on the last feature data. This can be accomplished using the cost-sensitive decision trees adapted for the context recognition task. Such trees work by calculating only a subset of features in order to minimize the cost of attaining and calculating them. Based on the current branch used during the classification, different feature subset may be active for next instance.

[30]:
```
from eecr import CostSensitiveTree
import pandas as pd
```

We start by defining a toy dataset for activity recognition. Attribute values are inspired by real sensors, but are modified so most attributes can only distinguish between two different activities.

[31]:
```
activities = ["still", "walk", "walk","run","run","car","car","subway","subway","still
↪"]
gps_velocity = [1,0,1,0,1,66,68,66,68,0]
acc_motion = [1,4,5,4,5,0,1,0,1,0]
acc_period = [4,5,4,10,11,5,4,5,4,5]
mag_field = [49,48,49,48,49,48,49,102,203,48]
data = pd.DataFrame()
data["gps_velocity"] = gps_velocity
data["acc_motion"] = acc_motion
data["acc_period"] = acc_period
data["mag_field"] = mag_field
data["activity"] = activities
x = data.drop(["activity"], axis=1)
y = data["activity"]
contexts = y.unique()
```

[32]:
```
data
```

[32]:
```
   gps_velocity  acc_motion  acc_period  mag_field activity
0             1           1           4         49    still
1             0           4           5         48     walk
2             1           5           4         49     walk
3             0           4          10         48      run
4             1           5          11         49      run
5            66           0           5         48      car
6            68           1           4         49      car
7            66           0           5        102   subway
8            68           1           4        203   subway
9             0           0           5         48    still
```

Next we define which attributes belong to which sensors and their respective costs. Attributes that belong to the same sensor will have the same cost, regardless of how many of them are active.

[33]:
```
feature_groups = {"acc": ["acc_motion", "acc_period"], "gps": ["gps_velocity"], "mag":
↪["mag_field"]}

def sensor_costs(sensor_list):
    #print(sensor_list)
    cost = 20 #Base cost
    if "acc" in sensor_list:
        cost += 10
    if "gps" in sensor_list:
        cost+=30
    if "mag" in sensor_list:
        cost+=15
    #print (cost)
    return cost
```

A cost sensitive tree is created and fitted with the data.

[34]:
```
tree = CostSensitiveTree(contexts, sensor_costs, feature_groups=feature_groups,␣
↪weight = 0.01)
tree.fit(x,y)
```

We would expect that classifying the original data will yield the original label sequence and it does.

[35]:
```
tree.predict(x)
```

[35]:
```
['still',
 'walk',
 'walk',
 'run',
 'run',
 'car',
 'car',
 'subway',
 'subway',
 'still']
```

The resulting tree can be visualized in two different layers of abstraction.

[36]:
```
tree.show()
```

```
acc_motion == 1
  mag_field == 49
    gps_velocity == 1
```

(continues on next page)

(continued from previous page)

```
      --> still
    gps_velocity == 68
      --> car
  mag_field == 203
    --> subway
acc_motion == 4
  acc_period == 5
    --> walk
  acc_period == 10
    --> run
acc_motion == 5
  acc_period == 4
    --> walk
  acc_period == 11
    --> run
acc_motion == 0
  mag_field == 48
    gps_velocity == 66
      --> car
    gps_velocity == 0
      --> still
  mag_field == 102
    --> subway
```

[37]:
```
tree.show_sensors()
```

```
acc
  ---> ['walk', 'run']
  mag
    ---> ['subway']
    gps
      ---> ['still', 'car']
  mag
    ---> ['subway']
    gps
      ---> ['still', 'car']
```

Finally, we show that if we increase the weights, some sensors cease to be used. GPS is the most expensive and is the first one to go.

[38]:
```
tree = CostSensitiveTree(contexts, sensor_costs, feature_groups=feature_groups,␣
↪weight = 0.019)
tree.fit(x,y)
```

[39]:
```
tree.show_sensors()
```

```
acc
  ---> ['walk', 'run']
  mag
    ---> ['still', 'subway']
  mag
    ---> ['subway', 'car']
```

4.4 Automatic settings generation

[40]:
```
from sklearn.tree import DecisionTreeClassifier
```

[41]:
```
classifier = DecisionTreeClassifier()
```

[42]:
```
optimizer = EnergyOptimizer(y)
```

[43]:
```
list(feature_groups)
```

[43]:
```
['acc', 'gps', 'mag']
```

[44]:
```
feature_groups =[["acc_motion", "acc_period"], ["gps_velocity"], ["mag_field"]]
```

[45]:
```
def sensor_costs(setting):
    #print(setting)
    cost = 20
    if setting[0] == 1:
        cost += 10
    if setting[1] == 1:
        cost+=30
    if setting[2] == 1:
        cost+=15
    return cost
```

[46]:
```
classifiers = optimizer.add_subsets(x,y,x,y,classifier, feature_groups=feature_groups,
↪ n = 3,
                          setting_fn_energy = sensor_costs)
```

TO BE CONTINUED…

CHAPTER FIVE

INDICES AND TABLES

- genindex
- modindex
- search

Download `sample code` as Jupyter notebook.

For any questions or feedback: VitoJanko@ijs.si

PYTHON MODULE INDEX

e

eecr, 3
eecr.cstree, 16
eecr.eeoptimizer, 3
eecr.eeutility, 14

INDEX

__init__()eecr.cstree.CostSensitiveTree method, 16
__init__()eecr.eeoptimizer.EnergyOptimizer method, 3

add_csdt_borders()eecr.eeoptimizer.EnergyOptimizer method, 11
add_csdt_weighted()eecr.eeoptimizer.EnergyOptimizer method, 12
add_subsets()eecr.eeoptimizer.EnergyOptimizer method, 13

CostSensitiveTreeclass in eecr.cstree, 16

dca_model()eecr.eeoptimizer.EnergyOptimizer method, 5
dca_real()eecr.eeoptimizer.EnergyOptimizer method, 6
draw_tradeoffs()in module eecr.eeutility, 14

eecrmodule, 3
eecr.cstreemodule, 16
eecr.eeoptimizermodule, 3
eecr.eeutilitymodule, 14
energy_quality()eecr.eeoptimizer.EnergyOptimizer method, 4
EnergyOptimizerclass in eecr.eeoptimizer, 3–5, 10, 11, 15

find_aimd_tradeoffs()eecr.eeoptimizer.EnergyOptimizer method, 15
find_coh_tradeoffs()eecr.eeoptimizer.EnergyOptimizer method, 16
find_dca_random()eecr.eeoptimizer.EnergyOptimizer method, 6
find_dca_static()eecr.eeoptimizer.EnergyOptimizer method, 6
find_dca_tradeoffs()eecr.eeoptimizer.EnergyOptimizer method, 6
find_sca_dca_tradeoffs()eecr.eeoptimizer.EnergyOptimizer method, 7
find_sca_random()eecr.eeoptimizer.EnergyOptimizer method, 7
find_sca_static()eecr.eeoptimizer.EnergyOptimizer method, 8
find_sca_tradeoffs()eecr.eeoptimizer.EnergyOptimizer method, 8
find_simple_tradeoffs()eecr.eeoptimizer.EnergyOptimizer method, 16
fit()eecr.cstree.CostSensitiveTree method, 17

load_config()eecr.eeoptimizer.EnergyOptimizer method, 10
load_data()eecr.eeoptimizer.EnergyOptimizer method, 10
load_data_config()eecr.eeoptimizer.EnergyOptimizer method, 10
load_solution()eecr.eeoptimizer.EnergyOptimizer method, 10

predict()eecr.cstree.CostSensitiveTree method, 17

quality()eecr.eeoptimizer.EnergyOptimizer method, 4

save_config()eecr.eeoptimizer.EnergyOptimizer method, 11
save_data()eecr.eeoptimizer.EnergyOptimizer method, 11
save_solution()eecr.eeoptimizer.EnergyOptimizer method, 11
sca_dca_model()eecr.eeoptimizer.EnergyOptimizer method, 8
sca_dca_real()eecr.eeoptimizer.EnergyOptimizer method, 8
sca_model()eecr.eeoptimizer.EnergyOptimizer method, 9
sca_real()eecr.eeoptimizer.EnergyOptimizer method, 9
sca_simple()eecr.eeoptimizer.EnergyOptimizer method, 9
set_dca_costs()eecr.eeoptimizer.EnergyOptimizer method, 3
set_path()eecr.eeoptimizer.EnergyOptimizer method, 4
set_sequence()eecr.eeoptimizer.EnergyOptimizer method, 4
set_settings()eecr.eeoptimizer.EnergyOptimizer method, 4
show()eecr.cstree.CostSensitiveTree method, 17
show_sensors()eecr.cstree.CostSensitiveTree method, 17
summary()eecr.eeoptimizer.EnergyOptimizer method, 4