Algorithms for Energy Disaggregation

Albert Fiol

Master in Innovation and Research in Informatics
Director: Josep Carmona
Co-director: Jorge Castro

Barcelona, 2016

Facultat d'Informàtica de Barcelona
Barcelona School of Informatics

Contents

1 Abstract

Energy disaggregation is the problem of separating an aggregate energy signal into the consumption of individual appliances in a household. This is useful because having a breakdown of the consumption of all the devices encourages users to consume less energy and gives them indications on how to do so. We focus on the problem of non-intrusive load monitoring, which attempts to perform energy disaggregation without using individual meters for every device.

In this project we compare three different solutions to the energy disaggregation problem: one based on sparse coding, another based on particle filters, and another based on quadratic programming. We test all three algorithms on a reference dataset, and finally we present Endivia, a program designed to easily visualize the disaggregate information.

We hope that this project will help us understand better the problem of energy disaggregation, giving us insights on the hardware and software requirements for potential commercial solutions.

2 Introduction

2.1 Energy disaggregation

Energy saving has been a concern for years and is now more important than ever. It is highly desirable for reasons such as saving money or being environmentally friendly, but no matter the reason, it has a big impact on our lives, from a personal to a national level.

While energy saving concerns us more and more as time goes by, we have reached a point in which energy consumption data is widely available, and machine learning has advanced up to a point in which we can build robust models of energy consumption.

Between these two ideas lays energy disaggregation. Energy disaggregation is, in essence, a signal processing and machine learning problem. It consists in taking the "aggregate" energy consumption signal (the power consumed by all the devices in a household) and extracting finer-grained readings, ideally corresponding to individual appliances in a single home. Figure 1 shows how different devices may contribute to an aggregate power signal.

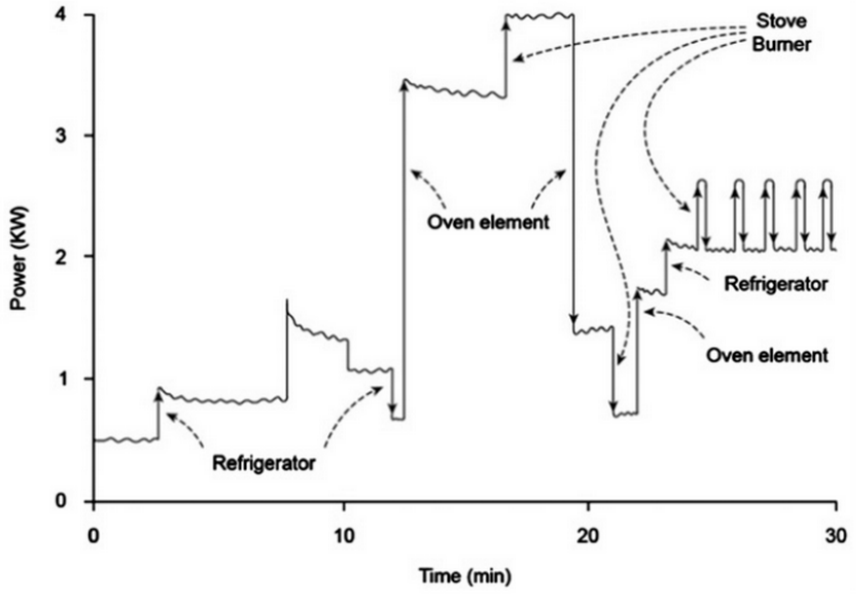


Figure 1: *Disaggregation attempts to identify the individual contribution of appliances in the overall energy signal. Original figure from Hart, 1992.*

This process is also known as Non-intrusive Load Monitoring (NILM) or Non-intrusive Appliance Load Monitoring (NIALM, also NALM). It was first proposed by George W. Hart, Ed Kern and Fred Schweppe of MIT in the 80s and further developed by Hart in the 90s [1]. The idea behind NILM is to measure only the voltage and current that go inside a household, and to use only that data to deduce the individual states of the appliances. In our case we use the *power consumption* of the household, which is commonly used to measure device usage and it is measured in watts (W). However, disaggregation can be done using various other features like the voltage waveform and its harmonics.

2.1.1 A concrete example

Let us go through a small disaggregation example. Training data comes to us as a list of power readings for every device in a household. In a hypothetical house with four devices, this would look as follows:

	Oven	TV	Dishwasher	Laptop
t_0	0	120	300	20
t_1	0	120	300	60
t_2	0	120	300	50
t_3	0	80	300	50
t_4	450	80	300	20
t_5	450	120	300	20
...	...	...	...	...

Table 1: *Example of training data for an energy disaggregation problem. For each device, its power consumption in watts (W) is given at every time sample.*

For the sake of the example, timesteps in the table are around 10 minute intervals. Now, once the system is trained, a disaggregation problem is presented as the following:

$$P = (440, 440, 400, 450, 750, 750, 750, 830, 850, 750, 570, 570, 570, 590)$$

The vector P represents a sequence of power readings (in W) which have to be disaggregated. A *possible* solution is the following:

	Total consumption	Oven	TV	Dishwasher	Laptop	Other
t_0	440	0	120	300	20	0
t_1	440	0	120	300	20	0
t_2	400	0	0	300	50	50
t_3	450	80	120	300	50	0
t_4	750	450	0	300	0	0
t_5	750	450	0	300	0	0
t_6	750	450	0	300	0	0
t_7	830	450	80	300	0	0
t_8	850	450	80	300	20	0
t_9	750	450	0	300	0	0
...	...	...	...	...	...	...

Table 2: *A possible solution for the disaggregation example problem.*

There are a few interesting things to note here. First, the fact that solutions are not unique: one can have two different solutions that minimize the disaggregation error: to discriminate between the solutions, domain knowledge or other methods must be used to select the best one. Another interesting problem is the occasional "lost power" which is not explained by the system, corresponding to either devices working in unexpected ways, or to unseen devices in the household.

2.2 Benefits of appliance monitoring

There have been several studies on the behaviour of consumers once they are presented with a detailed view of their energy consumption habits [2] [3]. Overall, the conclusion is that having feedback affects the users, and it can lead to energy saving behavior. In fact, according to these reports, detailed energy consumption information in households is almost invisible to the end user: energy companies tend to show the user a basic overview of the power consumed during a certain period of time.

The main idea behind appliance monitoring is that as users gain information on how their habits affect the energy consumption from any form of feedback, they have a "starting point" from which they can change their behaviour. This change is measured and provides the users with more knowledge on how the energy system works and how they can affect it. After a period of learning and adapting, users finally adopt new patterns of use resulting in a net saving of energy.

There are several types of feedback to consider, but the most important ones are *direct* and *indirect* feedback. Direct feedback consists in having direct displays, or reading the energy meters themselves. It is basically energy feedback on demand. Contrary to direct feedback, indirect feedback consists on data that is being processed by the energy company and later sent to the users. Indirect feedback can come in form of more frequent bills, or bills with disaggregated values.

While the effectiveness of feedback is known, recent studies have focused on what is the best kind of feedback and what is the best way of putting it in a user-friendly context. Direct feedback is instantaneous, but might be hard to understand for some people. On the other hand, indirect feedback may be presented in a more user-friendly way, but since it is less frequent it makes positive changes harder to notice and act upon.

A synthesis of common themes in past researches is provided in [2]; according to it, feedback is more useful if it is provided frequently, over an extended period of time, and presented in a simple, meaningful way (among other requirements). These are all achievable using energy disaggregation algorithms, even if they do not reach perfect accuracy. According to [3], direct feedback results in savings in the 5%-15% region, and similarly, indirect feedback results in savings in the 0%-10% region.

In a time where saving energy and money is one of the most important aspects in our daily life, achieving savings of only 5% in every household at a low price can have a huge impact at a global scale.

2.3 Goals of this thesis

The goal of this project is to overview recent advances in energy disaggregation, and to provide a starting point from which further research can be done. We will first survey the literature for candidate algorithms that can perform non-intrusive load monitoring, possibly in different sampling environments. We will implement them, trying to verify the original results.

More importantly, we want to compare all the algorithms on the same data, experimenting with them to figure out which algorithms perform better in which situations: we want to extract practical knowledge about their requirements, accuracy, or scalability. In the future, this practical knowledge can translate to a commercial application.

3 State of the art

As we have seen, energy disaggregation was first introduced by Hart in 1992 [1]. His proposal was to look at the power changes and cluster together the ones that appeared to be caused by the same device (or set of devices).

His work, and following disaggregation efforts, all worked on a set of assumptions that are common across literature. Before reviewing current work on this field we will first list these assumptions.

One of these assumptions is the *one-at-a-time assumption*, which assumes that appliances switch on and off one at a time. In other words, there are no events in which two appliances change state at the same time. It is easy to see that the strength of the assumption is related to the sampling rate: data sampled at, for example, 100 Hz has very low probability of having multiple state changes at the same time. However, data sampled at 1 per minute may have two devices turning on at the same data point. In our case, this assumption is needed for AFAMAP.

The second assumption is the *steady state load* assumption. It states that most appliances consume a constant amount of power while they stay in the same state. In practice the amount of power is not exactly constant, but with denoising techniques we can approximate the signal to a piecewise constant one.

The combination of these two assumptions is present in the first algorithms for energy disaggregation: changes in power are caused by some device changing state; the same change of state of the same device will always produce a similar change in the power demand, and this can be identified and used to disaggregate the whole signal.

In 1998, Cole and Albicki proposed their solution that took into account power spikes [4], which typically occur when a device is switched on or off. These spikes and slopes happen at the instant a device is turned on due to reactions in the electrical grid, an can be used as a "signature" to identify devices. These events are also known as *transients*.

In the early 2000s, algorithms went beyond just consumed power and added other features. As the technology behind the meters and recorders improved, new features became available such as the signal waveform and its harmonics. Signal processing solutions, such as Fourier's transform, were used to further detect "power signatures" of devices based on the transients and harmonics of the signal. [5] [6].

In the recent years we have seen a boom in machine learning approaches due to advances in both algorithms and networking, and also due to the sheer volume of data these algorithms can work with. Because of this, the last generation of energy disaggregation algorithms take advantage of big training sets and high performance computing.

One of these algorithms was introduced by Kolter, Batra and Ng in 2010 [7]. Their algorithm, called DDSC, will be tested in this project along two other algorithms. DDSC poses the disaggregation problem as a machine learning sparse coding problem. It does not take into account power spikes or transients, but it assumes that the learned model will capture these behaviours well. Another very recent paper treating energy disaggregation as a purely machine learning problem is [8], which tries to disaggregate a signal using neural networks.

Finally, some recent solutions move away from pure machine learning tasks and try to apply more complex inference techniques on the data, without necessarily taking transients or power spikes into account. Among these algorithms we can find particle filtering solutions such as PALDi [9], or optimization approaches such as AFAMAP [10].

Energy disaggregation has become a hot topic in the last years. Initiatives such as the NILM workshop[1] and a focus from big companies such as IBM[2] show a growing field with potential effects on our daily lives and wallets. One interesting initiative is

[1] http://nilmworkshop.org/

[2] http://researcher.watson.ibm.com/researcher/view_group_pubs.php?grp=4989

the NILM Toolkit (NILMTK) [11], which is an attempt to standardize the various NILM algorithms and datasets. NILMTK is interesting to us because it is open for extension: it allows us the integration of dataset converters, or disaggregation algorithms. In the experiments section we will adapt our algorithms to work with the NILMTK data format, in order to be able to compare them against other existing algorithms.

3.1 Hardware requirements for energy disaggregation

In this work we have mostly focused on the algorithmic side of energy disaggregation. However, for these solutions to be implemented we need some basic hardware infrastructure to provide us with data on which to train and run our algorithms.

The easiest way to obtain appliance-specific data for training is to use special plugs to which the appliances connect [12]. These plugs can be inconvenient to operate, and they present an additional cost for the consumers. Even more so, if consumers buy a special plug to read energy data from a device, they can obtain that data without the need for a disaggregation algorithm.

Some whole-house monitoring solutions already perform energy disaggregation for some devices, but the installation can cost up to hundreds of dollars, and once installed, the performance may not be as good as desired [12].

Finally, smart meters may present us with the solution to the data acquisition problem. These smart meters are currently replacing the old, analog power meters which we used to measure the power drawn by a house. Smart meters seem to be the first step towards a useful disaggregation infrastructure, as they are installed by the electrical companies. However, they currently send readings at very low sampling rates (between 15 minutes and 1 hour), which seems incompatible with most disaggregation algorithms.

As smart meters increase in complexity, we will start to see off-the-shelf solutions based on higher-frequency data, without the need to buy special plugs to gather training data for the algorithm. Until then, we can use algorithms designed for low sampling frequencies.

4 Definitions

In this section we will lay out the terms used in the following sections.

4.1 Markov chains

Markov chains are *stochastic, memory-less processes* that have a finite number of states. A process is a system whose state is represented by a set of variables. These variables are indexed, usually by time, and together form a series that represents the overall evolution of the process. A stochastic process is a process in which the state changes are random. The memory-less property (known as *Markov property*) states that the state of a system at any time t is only dependant on the state of the system at the previous timestep $t-1$. From now on we will assume that the Markov chains work at discrete timesteps.

Markov chains can be represented as a directed graph, in which each node corresponds to one of the n possible states, which form a *state space* that we call $\mathcal{S} = \{1, 2, \ldots, n\}$. At each timestep, the model has one state in $\mathcal{S}$, and then it changes through what we call a *transition*. As we mentioned earlier, Markov chains are stochastic processes, and thus, given the state of the model, the possible transitions follow a certain probability distribution. Each transition P_{ij} corresponds to one of the edges in the graph and it is defined as the probability of having a transition from state i to state j given that the model is found at state i. Note that transitions to the same state (self-loops in the directed graph) are possible. Since transitions are probabilities, it must hold true that, for all i:

$$\sum_j P_{ij} = 1$$

Transition probabilities are usually grouped in a *transition matrix* P, which is an $n \times n$ matrix that holds each transition probability P_{ij}. Finally, the Markov chain definition also includes an *initial state distribution*, which is the prior belief over the state of the system at initial time $t = 1$, and we will denote $p(X_1)$. This is usually represented as a vector of probabilities. With this, we can formally represent a Markov chain as the tuple $\langle \mathcal{S}, P, p(X_1) \rangle$.

Given the initial state x_1 and P, we can compute a distribution of states at time $t = 2$ by just multiplying $x_1 P$, and in fact we can compute the distribution of states at time $t = k$ by multiplying $x_1 P^k$. With this, we can now formalize the memory-less property of the Markov chain at any time t:

$$P(X_{t+1} = x \mid X_t = x_t) = P(X_{t+1} = x \mid X_0 = x_0, X_1 = x_1, \ \ldots \ , X_t = x_t)$$

A graphical representation of a Markov chain as a directed graph with $\mathcal{S} = \{1, 2, 3\}$ and transitions represented as edge labels can be seen in Figure 2.

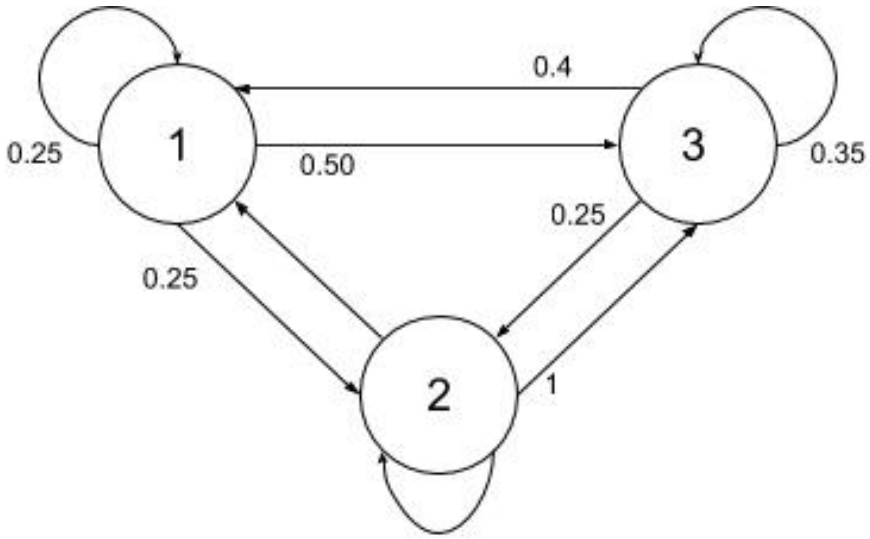


Figure 2: *Basic Markov chain represented in graph form.*

In this case, the transition matrix is the following:

$$P = \begin{pmatrix} 0.25 & 0.25 & 0.5 \\ 0 & 0 & 1 \\ 0.4 & 0.25 & 0.35 \end{pmatrix}$$

Now, if we imagine that the initial state is $x_1 = (0.5, 0.5, 0)$, we can see that the distribution over the states at time $t = 1$ is $x_1 P = (0.125, 0.125, 0.75)$. We can also compute the probability distribution over the states of the system at any time t, as we said before.

Markov chains are a specific case of *Markov processes*, which are defined as any process that satisfies the memory-less property. Markov chains satisfy it, but they also have additional constraints like a finite state space and a discrete timestep. Markov chains are very useful to model time series, and they have been used for many different applications such as ranking websites in a popular search engine [13], defining the entropy of a language via modelling it as a Markov process [14], or generating sequences of numbers that follow a desired distribution [15]. However, in many situations, the true state of the model cannot be measured, and Markov chains fall short - in those cases we need a more powerful model which is the hidden Markov model.

4.2 Hidden Markov models

A hidden Markov model (HMM) is a specific case of a Markov process, in which the system that follows the Markov property is hidden from the observer. Instead, we can only measure output values whose probability distribution depends on the state the hidden Markov process is in.

Formally, we represent the hidden Markov process as we did with the Markov chain: they can be represented as a directed graph, where each node corresponds to one state from $\mathcal{S} = \{1, 2, \ldots, n\}$, and where each edge corresponds to one transition. The HMM also has a random variable X that takes values from $\mathcal{S}$, using X_t to denote the value of X at time t. However, remember that this variable is hidden; what we, as observers, can measure, is a random variable Y, the *output* or *emission* of the HMM, with Y_t representing the value of Y at time t. In the general case, we say that $Y_t = f(X_t)$; f can be any function: a simple probability distribution, a constant value, a more complex function... A graphical example of the evolution of a hidden Markov model can be found in Figure 3.

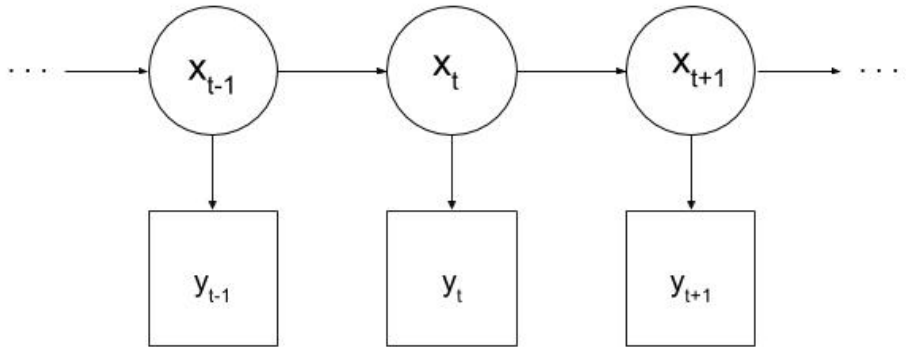


Figure 3: *Evolution of a hidden Markov model. In the top row, the states of the system at every time. In the bottom row, the emissions of each one of these states.*

In the picture, the states $x_{t-1}, x_t, x_{t+1}, \ldots$ are generated from a Markov chain, but observers only have access to the values $y_{t-1}, y_t, y_{t+1}, \ldots$. It is worth to note that the outputs also follow the Markov property with respect to the states in the hidden process: if we know $X_1, X_2, \ldots, X_t$, then it is true that:

$$P(Y_t = y_t \mid X_t = x_t) = P(Y_t = y_t \mid (X_1 = x_1, X_2 = x_2, \ldots, X_t = x_t)$$

Hidden Markov models are also a generalization of Markov chains: a HMM with identity as the emission function is essentially a Markov chain. However, hidden Markov models are much more powerful, and in fact they involve many inference problems. For example, computing the most probable sequence of states given the sequence of emissions, learning the parameters (transition probabilities and emission functions) of a HMM given a sequence of observations, or computing the most probable sequence of emissions given a HMM.

Because of the high expressive power of hidden Markov models, many problems can be modelled as HMM inference problems. For example, speech recognition [16], part-of-speech tagging [17], or even predicting protein topology [18]. Even so, there are still some problems that cannot be easily modelled as an inference problem on a single HMM. For those kinds of problems we need to combine multiple HMMs in what is known as a factorial hidden Markov model.

4.3 Factorial hidden Markov models

Even though HMMs are very useful for solving a whole family of inference problems, they lack in terms of representational capacity. For example, if we want to model a time series with 20 bits of information (as we would do to model the evolution of a household composed by 20 on/of devices), a HMM can only represent that state space with 2^{20} states, which is an scaling that makes many instances of problems unfeasible to be solved with HMM.

Factorial hidden Markov models (FHMMs) are a general type of HMM that has more than one variable to encode the state. This means, that while a HMM before had only X, a FHMM has k different state variables $X^{(0)}, X^{(1)}, \dots X^{(k)}$. Now, the same time series with 20 bits of information can be modelled with a FHMM that has only 20 binary variables. In fact, a FHMM with k models with n states each is equivalent to a HMM with n^k states. However, this higher expressive power comes with an increased computing cost, especially for inference algorithms.

Formally, we can represent a FHMM as a set of k HMMs that evolve simultaneously and independently over time, where we use $X^{(i)}$ as the variable that represents the state of the ith HMM. Then, at each timestep, there is a sequence of variables for each HMM in the FHMM: $X_0^{(i)}, X_1^{(i)}, \dots, X_t^{(i)}$ for any time t. Each HMM in the model has its own number of states, and we let n_i be the number of states of the ith HMM. Each HMM has its own emission functions, as well, which we will denote as $f_j^{(i)}$ to refer to the emission function of HMM i for state j.

Each HMM in the factorial model has its own output, but this output is not visible to the observer. Now, the visible output at time t, Y_t, is a combination of the outputs of each one of the HMMs in the model, $X_t^{(0)}, \dots, X_t^{(k)}$. For example, a FHMM equivalent of Figure 3 (a representation of the evolution of a HMM) is given in Figure 4.

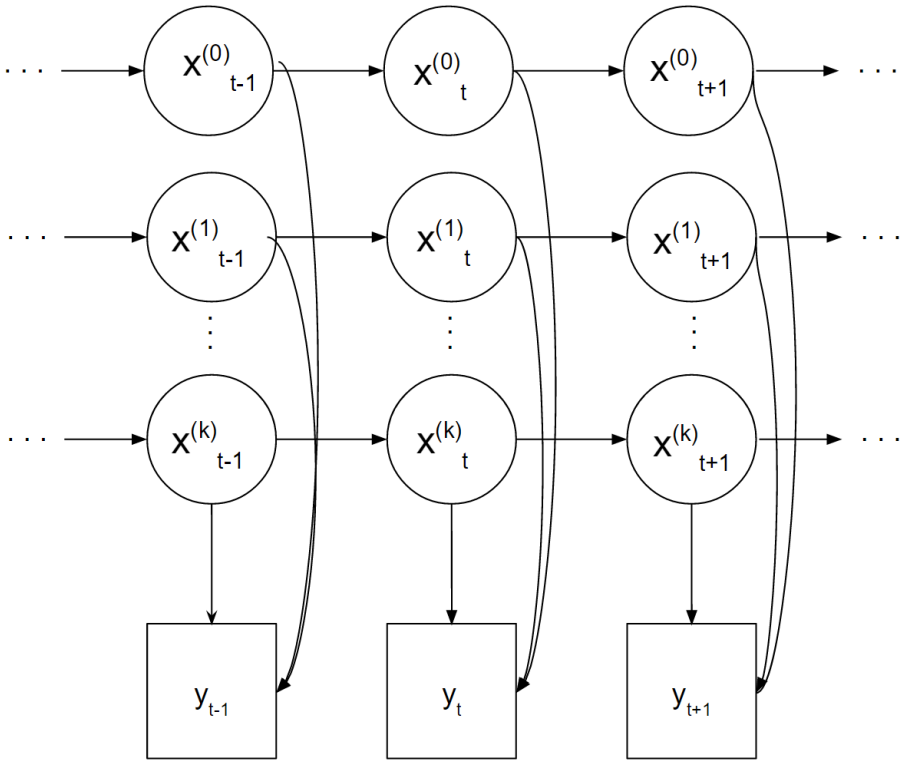


Figure 4: *A representation of a FHMM. The top three rows correspond to three independent Markov chains, whose states condition the output at every point in time.*

In this case, we have k HMMs that change state independently. Each one of them satisfies the Markov property. Their outputs at every time t $Y_t^{(i)}$ are combined to form the output of the FHMM, which is found in the bottom row as Y_t.

FHMMs were first introduced in the 90s, with Ghahramani and Jordan proposing a tractable algorithm to learn the parameters of a FHMM [19]. Since then, FHMMs have been used to solve various problems which were not easily tractable with HMMs, including again more robust speech recognition [20], bioinformatics [21], or source separation of audio signals [22], which is closely related to energy disaggregation.

4.3.1 Additive factorial hidden Markov models

In the following sections we will refer to a special type of FHMM which we will call *additive factorial hidden Markov model*, following Kolter and Jaakola's terminology [10]. These FHMMs output a value that is the sum of all the values outputted by the individual HMMs, hence the name *additive*. In other words, the output of an additive FHMM at time t is

$$Y_t = \sum_{i=0}^{k} Y_t^{(i)}$$

where $Y_t^{(i)}$ is the output of the ith HMM.

Additive FHMMs are of special interest to us, because they offer a very intuitive way of modelling a household. As we will see later, we can model each device as a single HMM in which the different states correspond to the different states of the device (which might be just 'on' and 'off', but also maybe a low-power state, or an 'on' with several power levels). Each state has an emission corresponding to a power consumption, and the transitions depend on the device usage pattern.

If we are modelling a household, we can group several devices, each modelled with a HMM, into a single FHMM that models the entire house. This has to be an additive FHMM, since the "aggregate" signal that we see is just the sum of all the power consumptions of all the devices in the house. For this reason, in the rest of this document, we will refer to additive FHMMs just as FHMMs, making a clear distinction only if necessary.

5 Disaggregation algorithms

In this section we will explain the three disaggregation algorithms, mentioning any necessary implementation details.

5.1 Technology overview

All three algorithms have been implemented mainly in Python[3] using the necessary libraries for linear algebra and plotting (mainly NumPy[4] [23] and SciPy[5] [24]). In the case of AFAMAP, the quadratic program is solved using Gurobi[6]. The first two algorithms, which only require Python (and NumPy and SciPy) have been adapted to work under NILMTK[7].

One of the most important differences between the algorithms is that DDSC and EDPF are both *supervised* learning algorithms, while AFAMAP is *unsupervised*. This is a key difference, and in the realm of practical energy disaggregation it is very important because supervised learning requires training data, and to obtain power consumption data for training we need special meters attached to every device that we want monitored in a household. Because of this, a good disaggregation algorithm may fail to reach out to consumers if they find the data acquisition step to be tedious or expensive. In contrast, AFAMAP works as a "plug and play" algorithm, which does not require extra hardware or cost.

5.2 DDSC: Discriminative Disaggregation Sparse Coding

The first algorithm we will consider is called DDSC. It was introduced by Kolter, Batra, and Ng in 2010 [7]. It is based on *sparse coding*, which is a class of unsupervised learning methods that try to learn succinct representations of data [25]. It is an algorithm that is designed to work with data measured at a low sample rate (around one lecture per hour).

The low-sampling rate feature of DDSC makes it hard to compare to the other algorithms meaningfully, but we will compare them anyway to see if DDSC is also good in environments with a high sampling rate. If it is not, we can now that we have an alternative to the rest of algorithms when data comes in hourly measurements.

Sparse coding (also known as *sparse dictionary learning*) consists in given a vector x, trying to learn a set of k over-complete bases θ such that:

$$x = \sum_{i=1}^{k} a_i \theta_i$$

where a_i is the ith coefficient for the base θ_i. This is similar to Principal Component Analysis (PCA) [26], which tries to learn a mutually linearly independent, complete set of bases that represent the data. Essentially, it transforms the data to a new system of coordinates, so that the bases of this system (called *principal components*) are aligned with the directions in which the dataset projects more variability.

In other words, the first principal component "'explains" the highest percentage of the data, the second one explains the second highest, etc. If the first few principal components explain a high enough percentage of the data, the rest of them can be dropped to have a compact representation of the input.

Unlike PCA, which learns a complete set of bases, sparse coding tries to learn an *over-complete* set of bases. If PCA can represent an n-dimensional vector with k

[3] http://www.python.org/
[4] http://www.numpy.org/
[5] http://www.scipy.org/
[6] http://www.gurobi.com/
[7] http://github.com/nilmtk/nilmtk

bases, with $n > k$, sparse coding tries to represent it with $k > n$. The reason for this is that an over-complete set of bases allows it to better capture patterns that are found in the input data (in the case of energy disaggregation, these patterns are the device use patterns).

However, learning an over-complete set of bases leads to the problem that the set of coefficients $a_1, a_2, \ldots, a_k$ is no longer uniquely determined by the input and the bases. In order to solve this, sparse coding enforces a criterion of *sparsity* over the coefficients, penalizing those that are far from zero. The problem of sparse coding a set of m vectors, then, is fully defined as:

$$minimize_{a_i^{(j)},\theta_i} \sum_{j=1}^{m} \left\| x^{(j)} - \sum_{i=1}^{k} a_i^{(j)} \theta_i \right\| + \lambda \sum_{i=1}^{k} S(a_i^{(j)})$$

where $x^{(j)}$ is the j-th vector in the set, $a^{(j)}$ is the vector of coefficients that represent the j-th vector in the set, θ_i is the ith basis, and $a_i^{(j)}$ is the ith coefficient that is used to represent the jth vector. $S(a)$ is a function that returns a cost based on the sparsity of a, and λ is a parameter used to vary the degree in which the sparsity criterion is enforced. This is an optimization problem which needs to optimize in two dimensions at the same time. One of them is finding the appropriate coefficients for every vector of the input set, and the second one is finding the appropriate set of basis, common for all the vectors in the input set.

5.2.1 Basic algorithm

DDSC is a supervised learning method. Let m be the number of houses (1 in our tests, since we test each house separately), and let T be the number of data points we have for each house, which should be sampled with a low sampling rate, according to the source paper. Finally, let k be the number of different classes of devices we want to disaggregate.

The input to DDSC is a $T \times m$ matrix of power readings, where each column contains the power readings for each house. The input also contains $X_1, X_2, \ldots, X_k$, which are also $T \times m$ matrices representing the separate signals for each class of device. Now, DDSC applies sparse coding on all the training matrices X_i to be able to approximate $X_i = B_i A_i$, where B_i are the bases and A_i are the coefficients (also called activations).

Once the base B_i and the activations A_i have been learned for each class, disaggregating an input signal $\bar{X}$ becomes a matter of solving the optimization problem

$$\hat{A}_{1..k} = \underset{A_{1..k} \geq 0}{\arg\min} \left\| \bar{X} - B_{1..k} A_{1..k} \right\|^2 + \lambda \sum_{i,p,q} (A_i)_{pq} \equiv F(\bar{X}, B_{1..k}, A_{1..k})$$

where $A_{1..k}$ is used to denote $A_1, A_2, \ldots, A_k$, and A_{pq} is the element in the pth row and qth column of A. Note how we abbreviate this optimization function in the right side, since we will use this notation later. This is again an optimization problem that assumes that the bases B_i have learned to capture device use patterns. These bases are now used to find suitable activations that are sparse, enforced with the second term in the formula with the λ parameter. These activations $\hat{A}_1, \hat{A}_2, \ldots, \hat{A}_k$ are then used to disaggregate the signal $\bar{X}$ the following way:

$$\hat{X}_i = B_i \hat{A}_i,$$

where $\hat{X}_i$ is the signal corresponding to the ith class of devices.

The intuition behind this method is that the bases B_i are trained to learn the use patterns of every device, so they are better suited to represent the ith portion of the signal than any other basis. This is the desired behaviour, but unfortunately in practice

this first part does not produce good results, because the bases are trained to learn the device use patterns with the hope that they will produce a small disaggregation error, but they are not built explicitly to minimize this error. To fix this, DDSC introduces a method to improve the bases, which is based in structured prediction [27].

The second step is based in the intuition that the best optimal activations of $\hat{A}$, called $A*$, are the ones obtained from the sparse coding step. The idea, then, is to optimize the bases $B_{1..k}$ so that when performing the optimization over $\hat{A}_{1..k}$, the value of $\hat{A}$ is as close as $A*$ as possible. However, changing the bases $B_{1..k}$ also affects the optimal $A*$, so the bases that are being optimized, now called $\tilde{B}$ (referred to as the *disaggregation bases*) need to be different to the ones obtained in the sparse coding step (referred to as the *reconstruction bases*). The problem is formulated as finding bases $\tilde{B}_{1..k}$ such that:

$$A*_{1..k} = \underset{A_{1..k} \geq 0}{\arg\min} \left\| \bar{X} - \tilde{B}_{1..k} A_{1..k} \right\|^2 + \lambda \sum_{i,p,q} (A_i)_{pq}$$

To find this optimal bases, the authors of DDSC use a method based on the structured perceptron [28]. The process is done iteratively, first computing the value of $\hat{A}_{1..k}$, and then doing an update on the bases $\tilde{B}_{1..k}$, repeating this until convergence.

The training phase of the DDSC algorithm ends here. Now, with the reconstruction bases $B_{1..k}$, given an aggregated test example $\bar{X}'$, we can compute $\hat{A}'_{1..k}$, and predict $\bar{X}'_i = B_i \hat{A}'_i$. The complete DDSC method is shown in Figure 5.

Algorithm 1 Discriminative disaggregation sparse coding

Input: data points for each individual source $\mathbf{X}_i \in \mathbb{R}^{T \times m}$, $i = 1, \ldots, k$, regularization parameter $\lambda \in \mathbb{R}_+$, gradient step size $\alpha \in \mathbb{R}_+$.

Sparse coding pre-training:

1. Initialize $\mathbf{B}_i$ and $\mathbf{A}_i$ with positive values and scale columns of $\mathbf{B}_i$ such that $\|\mathbf{b}_i^{(j)}\|_2 = 1$.
2. For each $i = 1, \ldots, k$, iterate until convergence:
   (a) $\mathbf{A}_i \leftarrow \arg\min_{\mathbf{A} \geq 0} \|\mathbf{X}_i - \mathbf{B}_i \mathbf{A}\|_F^2 + \lambda \sum_{p,q} \mathbf{A}_{pq}$
   (b) $\mathbf{B}_i \leftarrow \arg\min_{\mathbf{B} \geq 0, \|\mathbf{b}^{(j)}\|_2 \leq 1} \|\mathbf{X}_i - \mathbf{B}\mathbf{A}_i\|_F^2$

Discriminative disaggregation training:

3. Set $\mathbf{A}^\star_{1:k} \leftarrow \mathbf{A}_{1:k}$, $\tilde{\mathbf{B}}_{1:k} \leftarrow \mathbf{B}_{1:k}$.
4. Iterate until convergence:
   (a) $\hat{\mathbf{A}}_{1:k} \leftarrow \arg\min_{\mathbf{A}_{1:k} \geq 0} F(\bar{\mathbf{X}}, \tilde{\mathbf{B}}_{1:k}, \mathbf{A}_{1:k})$
   (b) $\tilde{\mathbf{B}} \leftarrow \left[\tilde{\mathbf{B}} - \alpha \left((\bar{\mathbf{X}} - \tilde{\mathbf{B}}\hat{\mathbf{A}})\hat{\mathbf{A}}^T - (\bar{\mathbf{X}} - \tilde{\mathbf{B}}\mathbf{A}^\star)(\mathbf{A}^\star)^T \right) \right]_+$
   (c) For all i, j, $\mathbf{b}_i^{(j)} \leftarrow \mathbf{b}_i^{(j)} / \|\mathbf{b}_i^{(j)}\|_2$.

Given aggregated test examples $\bar{\mathbf{X}}'$:

5. $\hat{\mathbf{A}}'_{1:k} \leftarrow \arg\min_{\mathbf{A}_{1:k} \geq 0} F(\bar{\mathbf{X}}', \tilde{\mathbf{B}}_{1:k}, \mathbf{A}_{1:k})$
6. Predict $\hat{\mathbf{X}}'_i = \mathbf{B}_i \hat{\mathbf{A}}'_i$.

Figure 5: *Detailed DDSC algorithm. Figure taken from [7].*

The algorithm has been implemented in Python following the original paper's methods. However, we have not succeeded in implementing the extensions that are presented in the original paper (total energy priors, group lasso, etc.).

5.3 EDPF: Energy Disaggregation via Particle Filtering

EDPF is a supervised algorithm that is based on particle filters, which are a common name for Sequential Monte Carlo methods[29]. The name comes from the fact that they are used to solve the *filtering problem*.

Contrary to DDSC, EDPF needs data that is sampled at a high rate (around 1 Hz). The reason for this, as we will see later, is that EDPF needs lots of data in order to build a model that represents each device accurately. With a granularity of hours it is not possible to know certainly how often does a device turn on and off.

The filtering problem is, in essence, a problem of inference on hidden Markov models: it consists in finding the current state of a hidden Markov model (or the whole series of states since time $t = 0$) based on only the observations of the model up to the present[30]. Several algorithms try to solve the filtering problem, given the nature of the model. The most basic on is the Bayesian filter[31].

Formally, we have a Markov process with states $\mathcal{S} = \{1, 2, \ldots, n\}$, with a random variable $X_t \in \mathcal{S}$ that is the state of the process at time t. The process follows the Markov property, and we denote the output at time t as Y_t. The filtering problem consists in estimating $X_1, X_2, \ldots, X_{t-1}$ given $Y_1, Y_2, \ldots, Y_t$. In order to do this, we need to know the posterior distribution

$$p(X_t|Y_1 = y_1, Y_2 = y_2, \ldots, Y_t = y_t)$$

to know which state is the most likely at time t.

From now on we will use $X_{1:t}$ as shorthand for $X_1, X_2, \ldots, X_t$. If we apply Bayes' rule, we have:

$$p(X_t|Y_{1:t}) = \frac{p(Y_{1:t}|X_t)p(X_t)}{p(Y_{1:t})}.$$

If we know the parameters of the model (transition probabilities, emission functions, initial state distribution), then we can compute this distribution recursively in two steps. First, the *prediction step*:

$$p(X_t|Y_{1:t-1}) = \sum p(X_t|X_{t-1})p(X_{t-1}|Y_{1:t-1})$$

This corresponds to calculating the prior probability of the next state, having seen all the observations until time t. After this comes the *update step*, in which after we make a new measurement, we refine our beliefs by computing the estimate a posteriori using Bayes' rule:

$$p(X_t|Y_{1:t}) = p(X_t|Y_{1:t-1}, Y_t) = \frac{p(Y_t|X_t, Y_{1:t-1})p(X_t|Y_{1:t-1})}{p(Y_t|Y_{1:t-1})}$$

In the fraction, the first factor is a parameter of the model (probability of an observation given a state), the second factor is the prior, and the denominator acts as a normalization constant. For this reason, we can omit the denominator and normalize the probabilities at a later step.

This is the general layout of a recursive Bayesian filter. At each step, it makes a new prediction a priori, and then updates its beliefs after seeing the evidence. The advantage of this approach is that it can be done online, without need to store all the learned distributions at every timestep.

There are other methods to solve the filtering problem, and other kinds of filters such as the Kalman filter, that can analytically compute the distributions if the shape of the posterior is Gaussian[32] [33]. However, these methods are based on assumptions over the shape of the distribution, or the linearity of the emission functions.

In our case, energy disaggregation is not necessarily gaussian nor linear, given the power signature of some devices - these are not on/off devices with very clear power consumptions, but they may be devices such as a drill, that have a continuous range of power activity. However, as the model increases in complexity, inference becomes

computationally expensive. For these cases we use approximate methods to solve the filtering problem, and particle filters are one of them.

5.3.1 Particle filters

Particle filters try to implement a recursive Bayesian filter by successive Monte Carlo simulations [34]. They use a number of *particles* to represent the posterior distribution of the state given the observations. These particles have weights which intuitively correspond to the likelihood of the state they represent in the target distribution. As the number of particles increases, we can better approximate the posterior distribution [35].

The particle filter works as follows: suppose we have a probability distribution π that we want to approximate. The Sequential Monte Carlo method approach consists in sampling this distribution independently N times, giving us N independent variables (*particles*) X^i for $i \in [1, 2, \ldots N]$. Then, we can approximate the original distribution by doing:

$$\hat{\pi}(x) = \frac{1}{N} \sum_{i=1}^{N} \delta(X_i, x)$$

where $\delta(x, y)$ is a function that is defined as 1 if $x = y$, and 0 otherwise. In literature this is done using the *dirac delta function* [30]. In the end, the empirical approximation of the density of x is obtained by sampling the original distribution N times, and counting how many times we sampled x.

This can also be done if the distribution is over a state space with t multiple values; in other words, the distribution is over states of the form $x_{1:t} = (x_1, x_2, \ldots, x_t)$. The previous formula can be applied:

$$\hat{\pi}(x_{1:t}) = \frac{1}{N} \sum_{i=1}^{N} \delta(X^i_{1:t}, x_{1:t})$$

This is usually a problem when the original distribution cannot be sampled from because it is too complex, or it is unknown. In these cases *importance sampling* is commonly used [36], which consists in sampling from a separate distribution $q(x)$ which shares some properties with the target distribution, and samples are weighted according to the ratio between the two distributions:

$$\hat{\pi}(x_{1:t}) = \sum_{i=1}^{N} w_i \delta(X^i_{1:t}, x_{1:t})$$

where $X_{1:t} \sim q(x)$ and w_i is calculated from the ratio $\frac{\pi(x)}{q(x)}$.

5.3.2 Basic algorithm

EDPF works by modelling every device as a HMM, fitting each HMM to the training data for that device, and then using this information to run a particle filter on the aggregate signal. We can treat the whole household as a FHMM with k discrete state variables (one for every device), or as a single HMM with n^k states (assuming each device has n states). In order to learn these HMMs, EDPF needs disaggregated training data corresponding to the power values of every device during a period of time. EDPF uses this data to fit HMMs to every device, and in the disaggregation phase it uses these HMMs to sample states and weights for a particle filter.

In any case, if we have the parameters of each HMM, we can compute the initial state distribution $p(X_1)$, the transition probabilities $p(X_t|X_{t-1})$, and the emission probabilities $p(Y_t|X_t)$ of the whole system. For this last term, we assign probability very close to 1 to a power value Y_t that is close to the sum of the individual power consumptions of every device, and 0 otherwise. Now, we have a good approximation

of the distribution $p(X_t|Y_{1:t})$ (using Bayes' rule), and since it follows the Markov property, we can easily sample values from it.

Specifically, EDPF runs a bootstrap filter [37]. Particles are represented as a tuple $\{x_{1:t}, w_t\}$, where $x_{1:t}$ is the sequence of states from the beginning of the algorithm, and w_t is the weight of each particle at time t. The algorithm is the following:

Algorithm 1 Bootstrap filter

$\mathcal{P} = \{x_1^{(i)} \sim p(X_1), w_1^{(i)} = \frac{1}{N}\}, i \in [1..N]$
for every $t > 1$ **do**
    for every particle p_i **do**
        Sample next state: $x_t^{(i)} \sim p(X_t|X_{t-1})$
        Compute particle weight: $w_t^{(i)} = p(X_t = x_t^{(i)}|Y_t)$
    Normalize all weights: $w_t^{(i)} = w_t^{(i)} / \sum w_t^{(j)}$.
    for every particle p_i **do**
        $p_i = resample(\mathcal{P})$
return $predict(\mathcal{P})$

The algorithm works intuitively by having many simulations of the model, each one corresponding to one particle, with the weights corresponding to the likelihood of this simulation. At every step in time, we generate our prior belief over the state of the system, and weight that prediction based on the probability of it happening once we have made the observation. After having done this for all particles, we do a *resampling* step.

Resampling is done to discard very unlikely hypotheses and replace them for more likely ones. The function $resample(\mathcal{P})$ selects one particle $p_i \in \mathcal{P}$ with probability w_i, since those are normalized weights, and they define a probability distribution. This results in more likely particles being resampled more times, increasing the chances of approximating the desired distribution of the model.

At the end, we select a winner explanation which corresponds to the most likely sequence of states given the observations on the model. The function $predict(\mathcal{P})$ selects, in the case of EDPF, the particle with the highest weight, but this prediction can be made in many different forms: we could combine several of the particles with the highest weights into a single explanation, or even further process the winner particle given some other information.

The whole bootstrap filtering process is shown in Figure 6.

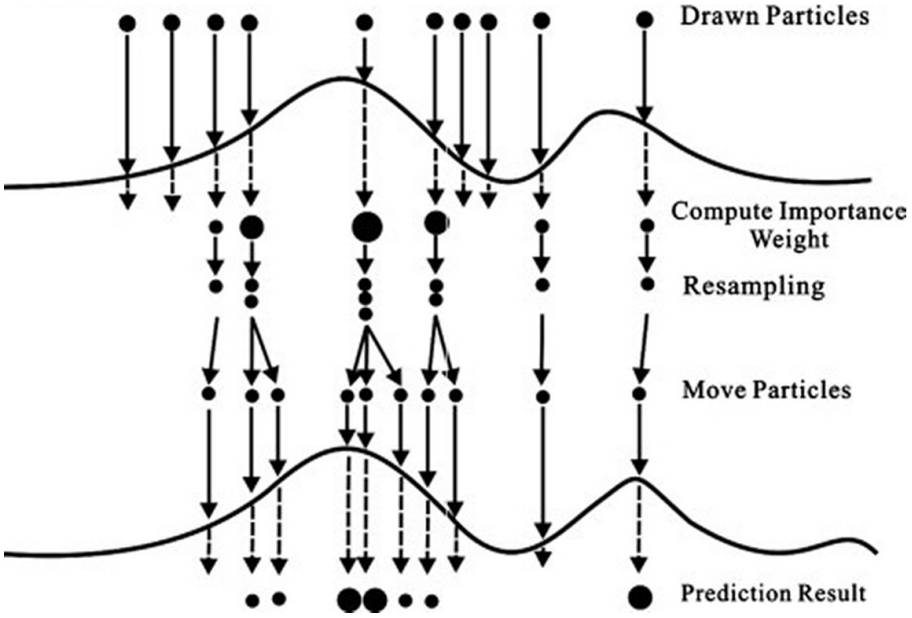


Figure 6: *Visualization of the bootstrap algorithm.*

5.3.3 Learning device HMMs from data

In order for EDPF to work, we need to know each device's HMM parameters to compute the joint probability distributions. As we said before, we need training data for every device, which corresponds to sequences of values $Y_1^{(k)}, Y_2^{(k)}, \ldots, Y_t^{(k)}$ for every device k. EDPF greatly benefits of high sampling rates, as HMMs are fitted by counting methods, and we need transitions to happen in a very fine way.

Once we have data for every device, we are faced with the problem of fitting a HMM to that data. There are several ways of doing so, for example using the expectation-maximization algorithm (EM) [38] or the Baum-Welch algorithm [39] [40]; in our case, we will take a simpler approach. First, we preprocess the signal and approximate it to a piecewise constant signal, and then we build a HMM with as many states as different power levels we have observed, with transition probabilities computed by counting how many timesteps the device was found in each state, and emissions of each state being that same power value. We have tried two ways of approximating the signal to a piecewise constant signal: the *clustering approach*, and the *signal denoising approach*.

The clustering approach uses k-means [41] to identify k important "power levels" for the device. The most basic clustering is a one-dimensional clustering and corresponds to an on/off device, but often devices have several power levels (for example, TVs have a stand-by mode, or microwave ovens work at different powers). In this case, it is required to fine-tune the k parameter.

A possible option is to include some heuristics during this process to make the clustering automatic: if with a certain k there are clusters that have very few observations (for example, less than 2%), then the whole process is repeated with a smaller k until groups are representative enough. Once all observations are clustered, each point in the signal is replaced by the representative of its group (computed as the mean or median).

5.3.4 Signal denoising approach

The signal denoising approach uses total variation denoising (TV-denoising) [42] to achieve a piecewise constant signal. This method tries to minimize the total variation of a signal y, which is defined as:

$$TV(y) = \sum_{n>0} |y_n - y_{n-1}|$$

Informally, it corresponds to the sum of the changes in the signal's magnitude. This function is interesting because signals with higher noise tend to have higher total variation, so if we reduce the total variation while still being close to the original signal, we will have removed some undesired noise. This idea is very useful for image processing, and in fact TV-denoising can be used to enhance and remove noise in images [43].

Total variation denoising is basically an optimization problem. The TV-denoised signal x for a given signal y is defined as:

$$x = argmin_x \left(\sum_n (y_n - x_n)^2 + \lambda \, TV(x) \right)$$

The λ parameter controls the degree of smoothing. The bigger λ is, the less fluctuation will be in the resulting signal. It is easy to see that for every signal there exists a value λ_{max} in which performing TV-denoising with $\lambda \geq \lambda_{max}$ results in the trivial solution $x = mean(y)$ (a solution with no fluctuation that minimizes the square error).

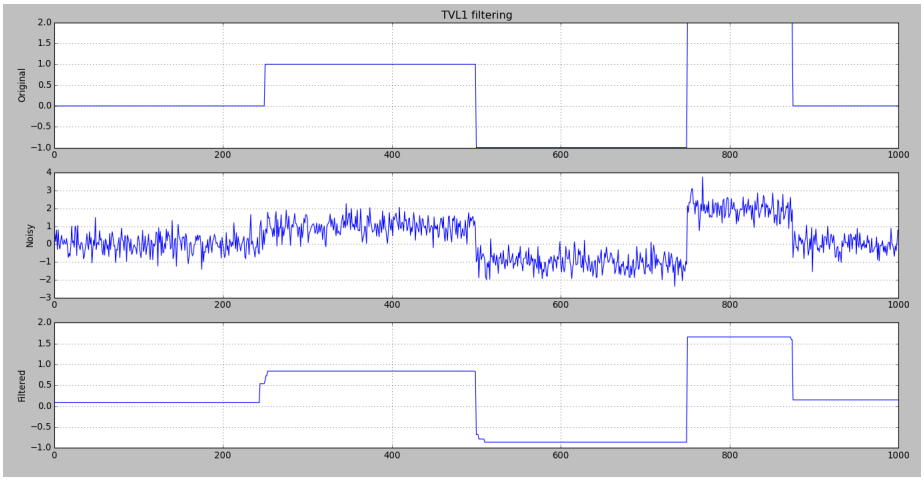


Figure 7: *Total variation denoising allows us to reduce the total variation of a signal while keeping close to the original signal. Top: original signal. Middle: noisy signal. Bottom: denoised signal using TV denoising.*

Once the signal has been denoised, a second pass over the signal transforms it to a true piecewise constant signal by greedily assigning the non-constant values to the closest constant power level.

5.4 AFAMAP: Additive Factorial Approximate Maximum A Posteriori

AFAMAP is an algorithm introduced by Kolter and Jaakkola that performs Maximum A Posteriori inference in additive FHMMs [10]. A key advantage of AFAMAP over the rest of algorithms in this report is that AFAMAP is an unsupervised algorithm: it works over the aggregated signal without the need of a priori knowledge of the devices of the house. However, it also benefits from having a detailed model of the house appliances. In this project we will work with a priori information on the house appliances.

For this reason (and the one-at-a-time assumption, which states that at every data point, at most one device is changing state), AFAMAP needs a high sampling rate (of around 1 Hz) to ensure optimal results.

AFAMAP works with two simultaneous models: the *additive model* and the *difference model*. The additive model corresponds to the individual contribution of each device to the aggregate signal. On the other hand, the difference model corresponds to changes in the overall power consumption at each timestep: under some assumptions, changes in the aggregate signal value can be explained by a single device turning on/off.

5.4.1 Basic algorithm

In this section we will formalize both the additive and the inference factorial models which are the base of AFAMAP. Then we will develop the MAP inference problem as a quadratic programming problem.

The *additive factorial model* is modelled as follows:

$$
\begin{aligned}
x_1^{(i)} &\sim Mult(\phi^{(i)}) \\
x_t^{(i)} | x_{t-1}^{(i)} &\sim Mult(P^{(i)}_{x_{t-1}^{(i)}}) \\
\bar{y}_t | x_t^{(1:N)} &\sim \mathcal{N}(\sum_{i=1}^{N} \mu^{(i)}_{x_t^{(i)}}, \sigma),
\end{aligned}
$$

where N is the number of devices (and thus, the number of HMMs in the model), and m_i is the number of states in the ith HMM, which corresponds to the different number of states a device can be in. Also, T is the number of timesteps; $x_t^{(i)}$ is the state of the ith HMM at time t; $\bar{y}_t$ is the observed output at time t, and $\mu_j^{(i)}$ is the mean of the ith HMM for state j. Finally, $\phi^{(i)}$ is the initial state distribution for the ith HMM, σ is the observation variance, and $P^{(i)}$ is the transition matrix for the ith HMM.

As one can see, this model follows the Markov property: the first formula represents the state of the model at the first timestep. Then, the state at successive timesteps only depends on the previous one, and the probabilities are derived from the transition matrix. Finally, the output at any time t only depends on the state of the system at that time, and it corresponds to the sum of individual outputs.

The *difference factorial model* is modelled similarly. It is still a FHMM, and thus the HMMs inside it follow the Markov property, and we use the same parameters to describe it. However, the output at time t, which is denoted $\Delta\bar{y}_t$ is defined as:

$$
\Delta\bar{y}_t | x_t^{(1:N)} \sim \mathcal{N}(\sum_{i=1}^{N} (\mu^{(i)}_{x_t^{(i)}} - \mu^{(i)}_{x_{t-1}^{(i)}}), \sigma)
$$

This means that the output of the difference factorial model at every time t is the total increase in power with respect to the previous timestep. A comparison between the two models is shown in Figure 8.

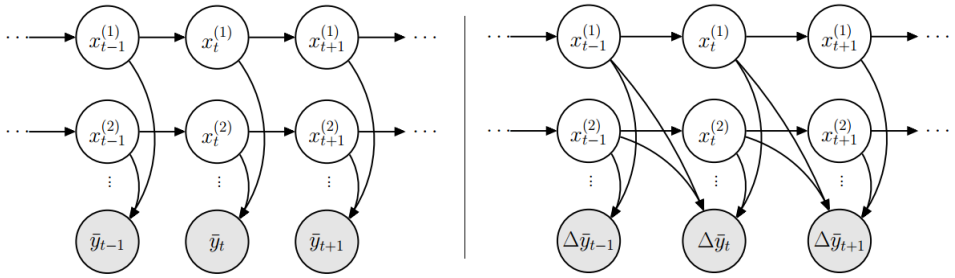


Figure 8: *On the left, the additive factorial model. On the right, the difference factorial model. Original figures from Kolter and Jaakkola's paper[10].*

Now, both systems are used together, because the additive model captures the total aggregate output very well, and at the same time the difference model encodes the changes in the power signal at every timestep.

Apart from this, a separate third component is added to both additive and difference models to make the model more robust against outliers. It is a component than can take any value but that is regularized via total variation. In the additive model, it is denoted by z_t, while in the difference model it is denoted by Δz_t. This extra signal will be used to represent the "unexplained" power which is not assigned to any device, and the total variation regularizer encourages it to take piecewise constant values, which will represent the unexplained power coming from other devices.

Now, let us model the disaggregation problem as a MAP inference problem. We will first model the exact MAP inference problem, but afterwards we will relax it using some key observations in order to develop the final AFAMAP algorithm. The exact MAP inference is an optimization problem over the variables

$$\mathcal{Q} = \{Q(x_t^{(i)}) \in \mathbb{R}^{m_i}, Q(x_{t-1}^{(i)}, x_t^{(i)}) \in \mathbb{R}^{m_i \times m_i}\}$$

which are indicator variables that satisfy that $Q(x_t^{(i)})_j = 1 \iff x_t^{(i)} = j$, and $Q(x_{t-1}^{(i)}, x_t^{(i)})_{j,k} = 1 \iff x_{t-1}^{(i)} = j, x_t^{(i)} = k$. These variables are subject to the following set of constraints $\mathcal{L}$:

$$\mathcal{L} = \begin{array}{rcl} \sum_{j=1}^{m_i} Q(x_t^{(i)})_j & = & 1 \\ \sum_{k=1}^{m_i} Q(x_{t-1}^{(i)}, x_t^{(i)})_{j,k} & = & Q(x_{t-1}^{(i)})_j \\ \sum_{k=1}^{m_i} Q(x_{t-1}^{(i)}, x_t^{(i)})_{k,j} & = & Q(x_t^{(i)})_j \\ Q(x_t^{(i)})_j, Q(x_{t-1}^{(i)}, x_t^{(i)})_{k,j} & \geq & 0 \end{array}$$

The first constraint imposes that every device must be in exactly one state at all times. The second and third constraints enforce the relationship between the two types of indicator variables: if $Q(x_{t-1}^{(i)}, x_t^{(i)})_{j,k}$ is true, then both $Q(x_{t-1}^{(i)})_j$ and $Q(x_t^{(i)})_k$ must also be true.

Now, exact MAP inference can be expressed as optimizing the likelihood of the models subject to the constraints in $\mathcal{L}$, using binary variables (enforcing $\mathcal{Q} \in \{0, 1\}$).

This results in a mixed-integer quadratic program, which is hard to solve[44], so the authors of AFAMAP have relaxed it using some key observations.

The first observation is the *one-at-a-time condition*, which is reasonable to accept under high sampling rates (e.g. 1 Hz). The one-at-a-time condition is used to further limit the set of distributions that could explain the observation. Formally, this can be represented with this additional constraint:

$$\mathcal{O} = \sum_{i,j,k \neq j} Q(x_{t-1}^{(i)}, x_t^{(i)})_{j,k} \leq 1$$

This constraint simplifies the math, allowing the quadratic component in the problem to be replaced by a linear component.

The second observation is that we can drop the integer constraint on the variables, since the nature of the problem encourages integer solutions anyway. Dropping this constraint turns the problem into a convex linear program which can be solved much more easily.

The final AFAMAP algorithm is shown in Figure 9:

Input: $\bar{y}_{1:T} \in \mathbb{R}^n$, aggregate output signal; $\mu_{1:m_i}^{(1:N)} \in \mathbb{R}^n$, state means for N HMMs; $\Sigma_1, \Sigma_2 \in \mathbb{R}^{n \times n}$, $\lambda_1, \lambda_2 \in \mathbb{R}_+$, covariance and regularization parameters

minimize over $\{\mathcal{Q} \in \mathcal{L} \cap \mathcal{O}, z_{1:T}\}$

$$\frac{1}{2} \sum_t \left\| \bar{y}_t - \Sigma_1^{1/2} z_t - \sum_{i,j} \mu_j^{(i)} Q(x_t^{(i)})_j \right\|_{\Sigma_1^{-1}}^2$$

$$+ \frac{1}{2} \sum_{t,i,j,k \neq j} \left\| \Delta \bar{y}_t - \Delta \mu_{k,j}^{(i)} \right\|_{\Sigma_2^{-1}}^2 Q(x_{t-1}^{(i)}, x_t^{(i)})_{j,k}$$

$$+ \frac{1}{2} \sum_t D(\Sigma_2^{-1/2} \Delta \bar{y}_t, \lambda_2) \left(1 - \sum_{i,j} Q(x_{t-1}^{(i)}, x_t^{(i)})_{j,j} \right)$$

$$+ \sum_{t,i,j,k} Q(x_{t-1}^{(i)}, x_t^{(i)})_{j,k} (-\log P_{k,j}^{(i)})$$

$$+ \lambda_1 \sum_t \|z_t - z_{t-1}\|_1$$

Output: $\hat{y}_{1:T}^{(1:N)}$, predicted individual HMM output

$$\hat{y}_t^{(i)} = \sum_j \mu_j^{(i)} Q(x_t^{(i)})_j$$

Figure 9: *The AFAMAP algorithm. Original figure from Kolter and Jaakkola[10].*

The first term is optimizing the likelihood of the additive factorial model; the second term corresponds to the difference model. The third component includes the

Huber loss function [45], which corresponds to the error the the signal Δz in the difference FHMM model, and intuitively is the error of the signal Δz if there has been no change in the devices, and 0 otherwise. The Huber loss function is the result of analytically optimising the error of Δz using the one-at-a-time assumption. The fourth component is penalizing unlikely transitions, and finally the fifth component is the regularization term on the "unexplained" signal z_t, which encourages it to take on piece-wise constant values.

The problem can be solved with commercial solvers (such as Gurobi), but in practice, the total variation term (the fifth term in the AFAMAP problem) is not handled very well by the software. For this reason, AFAMAP first fixes z, solves the remaining problem with a commercial solver, and then fixes $\mathcal{Q}$ and solves the remaining problem, which is a total variation regularization problem that can be solved efficiently [46]. This process is repeated until convergence.

5.4.2 AFAMAP as an unsupervised energy disaggregation algorithm

We mentioned that AFAMAP is an unsupervised algorithm, which is a clear advantage compared to the other algorithms presented. However, it still needs information about the devices in a house: specifically, the state means for the devices (i.e. the mean power consumed in each state) and also the transition matrix for every device (i.e. the usage pattern of each device).

Kolter and Jaakola [10] propose to use total variation regularization to approximate the signal as a piecewise constant series, which makes the data easier to work with. The next step consists on finding all the cases in which a device was turned on (making the power consumption go up by a certain amount) and eventually went back to the original value. If the sampling is fast enough some individual device power consumptions can be captured this way, and these are modelled as HMMs. The means of the states correspond to the observed power change, and the transition probabilities can be obtained by counting how much time each device spends in each state.

After this, AFAMAP runs spectral clustering to group some of the similar HMMs together, and the result is fed to the quadratic program that does the MAP inference. This results in a very powerful algorithm that does not need any training data.

This results in an algorithm that can run in an unsupervised fashion, while at the same time greatly benefiting of accurate training data and powerful modelling algorithms.

5.5 NILMTK algorithms: COOP and FHMM

We also consider some of the algorithms in NILMTK in our tests, to have a better insight on how our implementations are working, and how do they compare to others. Specifically, we will run tests with two of the algorithms in NILMTK.

The first one (abbreviated as *CO* in the original paper, and *COOP* here) is based on combinatorial optimisation [11]. At training time, it clusters each device's power consumption values in order to have a set of power states. For example, if the power consumption of the kth device is $(0, 0, 0, 0, 30, 35, 33, 35, 28, 35, 0, 0)$, COOP clusters the values and finds two power states: $(0, 30)$.

It then builds a 2d matrix in which each column represents a device, and each row represents every possible combination of the power values of the devices (in other words, the Cartesian product of all the device power values). For example, the next matrix corresponds to a set of 3 on/off devices, with power consumptions 20W, 100W, and 50W respectively:

$$\begin{bmatrix} 0 & 0 & 0 \\ 0 & 0 & 50 \\ 0 & 100 & 0 \\ 0 & 100 & 50 \\ 20 & 0 & 0 \\ 20 & 0 & 50 \\ 20 & 100 & 0 \\ 20 & 100 & 50 \end{bmatrix} \Rightarrow \begin{bmatrix} 0 \\ 50 \\ 100 \\ 150 \\ 20 \\ 70 \\ 120 \\ 170 \end{bmatrix}$$

At disaggregation time, COOP just finds the row whose sum is nearest to the aggregated signal. For example, if the signal to be disaggregated is $y = (55, 115, 120)$, then we can now that at time $t = 1$ we had only the 3rd device on; at time $t = 2$ and $t = 3$, we had the first one and the second, etc.

The second algorithm, abbreviated as *FHMM* [11], is very similar to EDPF. In the training phase, it fits a HMM to every device given the disaggregated training data; then combines all the HMMs into a single one. At runtime, it runs the Viterbi algorithm [47] to find the most likely sequence of states given the observations. FHMM works with the *hmmlearn* Python library[8], which includes many methods to build and perform inference on HMMs.

[8] https://github.com/hmmlearn/hmmlearn

5.6 Implementation

As we said in the beginning of this section, we have implemented all three algorithms DDSC, EDPF and AFAMAP in Python, using libraries such as NumPy and SciPy. However, our implementation differs from the source material in a few ways.

In the case of DDSC, we have implemented the basic algorithm using the same methods shown in the original paper [7]. However, I was not able to successfully implement and test the proposed extensions. One of particular interest is the Group Lasso [48] extension, which tries to group the activations of a device through the set of houses, but unfortunately this extension had to be dropped from the final algorithm due to it performing poorly, and also because of the testing environment using only one house.

In the case of EDPF, the implementation of the bootstrap filter is straightforward. To compute the updated weights given an observation, I assumed that every state of the HMM outputs a constant value (the steady state load assumption), but for practical purposes I modelled them as a normal distribution $\mathcal{N} \sim (\mu, \sigma^2)$ with μ equal to the state's power value, and σ set to a small constant. Then, the weight can be computed as the probability of the observation being sampled from a normal distribution $\mathcal{N}_t$ which is a sum of normal distributions:

$$\mathcal{N}_t \sim (\sum_{i=1}^{k} \mu_i, \sum_{i=1}^{k} \sigma_i^2)$$

In the case of AFAMAP, I implemented the quadratic program as seen in the original paper [10]. However, I used [49] to solve the TV regularization problem to compute the new z for a fixed $\mathcal{Q}$.

Also, I departed from the proposed approach by running a supervised version of AFAMAP, applying the same method we used in EDPF to build the HMMs. Once we have the HMMs for the devices, we pass them to the MAP inference quadratic program. The main reasons behind this choice are simplicity and ease of comparison with the other algorithms, as now they are all trained and tested using the same data.

6 Datasets

The initial scope for this project considered working on different datasets, but due to time constraints I have reduced the testing to only one of them, REDD. This is so because it is the most commonly used dataset for energy disaggregation: both the AFAMAP authors and the NILMTK developers use it for benchmarks. However, for reference, I will also include in this section GREEND, which is another potential dataset for testing our algorithms.

6.1 REDD

REDD (Reference Energy Disaggregation Dataset) is a dataset built for the purpose of energy disaggregation, using readings from 6 different houses [50]. The frequency of the readings is around 1 Hz, but in some cases the readings are done every 3 seconds.

The dataset includes power readings as well as high frequency voltage data that can be also used for disaggregation. However, since our algorithms only deal with power consumption, we will ignore the high frequency data.

REDD is the dataset on which tests are run on the original AFAMAP paper. However, as we mentioned earlier, they do not use the training data; they only take the aggregate signal and try to disaggregate from there. In our case we will use the original disaggregate data to build the device HMMs and from there we will run AFAMAP.

We will access the dataset through NILMTK, as it provides a converter to its file format. In the experiments, we will run the disaggregation with $k = 5$ devices, which are selected as the top 5 consuming devices in each household.

6.2 GREEND

GREEND (GREEND Electrical ENergy Dataset) is a dataset built with energy readings from houses in Italy and Austria [51]. It features readings from 8 different houses. Readings are done with a frequency of 1 per second.

To gather the data, a small computer and storage device was introduced in each house, along with a series of sensing outlets that would read the appliance energy consumption at fixed intervals. For more information, the introductory paper to the dataset contains more details on the setup and case studies. [51]

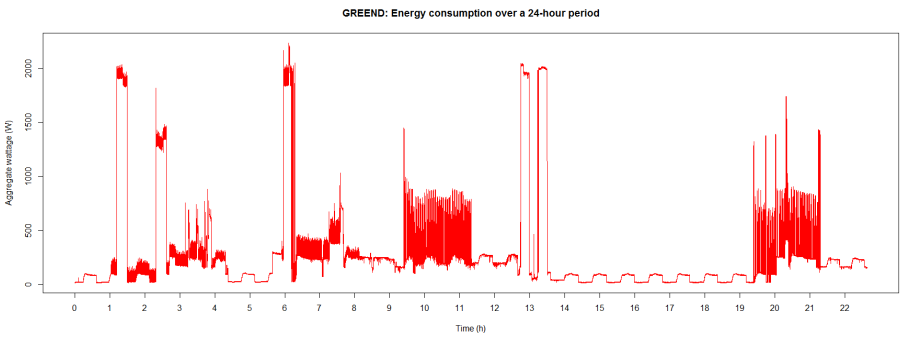


Figure 10: *Aggregate energy consumption of GREEND's building*0 *over the course of a 24-hour period. Different chunks with very different behaviours can be identified looking at the aggregate signal.*

Apart from energy disaggregation (which is tested by the authors with a particle-filter-based disaggregator), the GREEND paper introduces more problems such as occupancy detection or appliance usage modelling. The first problem, although interesting, is of less use to us as the second one. Appliance usage modelling consists on mining the patterns in which devices are used by residents, which can be later used as extra information for the disaggregation process.

Since GREEND contains fairly consistent 1 Hz readings, there is little preprocessing concerning this issue. However, there are some instances in which the devices failed to report any power consumption, producing lots of NA values. NILMTK allows to easily parse the GREEND dataset and read the data while applying a desired function on the NA values (for example, drop them, but also copy the last non-NA value seen, for example).

7 Methodology

In this section we outline the main experimentation procedure. We have designed these experiments with the following goals in mind:

- Find out if disaggregation is possible, and if it can be done producing useful results. We also want the experiments to give us insights on the requirements of energy disaggregation algorithms and their feasibility.
- Compare the algorithms with respect to three criteria: accuracy, running time, and scalability.

To do this, we will train and test the algorithms with different power series from REDD. First, we will train and test the algorithms with the same data. We will compare runtime and accuracy. This serves as a baseline, to see how good the algorithms are at just running on learned data.

Second, we will train the algorithms on a dataset representing a household's consumption over a day or a week, and we will test the models the same house, but during a different period of time.

Third, we will run different tests on very short amounts of time in order to test AFAMAP against other algorithms.

The specific metrics we will use to compare the algorithms are:

- **Run time**. Measured in seconds to train, and seconds to disaggregate.
- **F-score (F_1)**. Introduced by NILMTK, it is defined as the harmonic mean of precision and recall of properly classifying devices as on/off (higher F1-scores are better):

$$F_1 = 2\,\frac{P \cdot R}{P + R}$$

- **Average RMSE (per device)**. The average of the mean square errors between the prediction and the real value of each device. This is a continuous value, and is defined as follows, where $\hat{y}$ denotes the prediction and y denotes the ground truth value (lower errors are better):

$$RMSE = \sqrt{\frac{1}{T}\sum_{t}^{T}(y_t - \hat{y}_t)^2}$$

The majority of the experiments have run on a machine with 16 GB of memory and a i7-6700k CPU running at 4.00 GHz. The AFAMAP experiments have been run on a machine with 8 GB of memory and a i5-3230M CPU running at 2.60 GHz, due to AFAMAP requiring 3rd party software that I was unable to get working on the first machine.

8 Results and discussion

8.1 Exploratory tests: REDD

We will run an exploratory test on the first building of REDD in order to have an idea of how the disaggregators perform. To do this, we will first train and test the algorithms on a day worth of data of the first building of REDD, sampling at 1 Hz. Then we will redo the test with 8 days worth of data, sampling once every 30 seconds, to see how they scale. The disaggregation here attempts to predict the power consumption of the top 5 devices in the building.

As we mentioned in section 5, these algorithms are designed to work in environments with different sampling rates: DDSC works with hourly readings, while the rest of the algorithms work with a sampling frequency of 1 Hz. For this reason, we expect DDSC to perform poorly in the following experiments, as they are ran on datasets with high sampling rate.

The data looks like this:

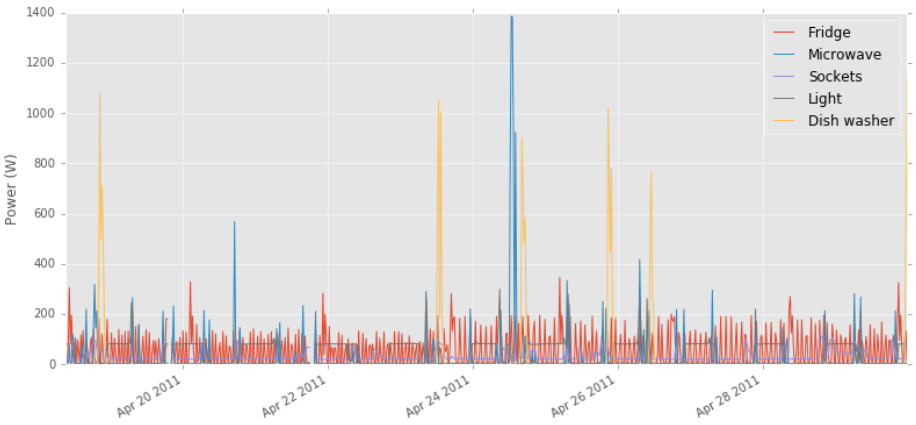


Figure 11: *Data for REDD's building 1. Data for the top 5 appliances is shown between 19th and 30th of April, 2011.*

	Training time	Prediction time	Average F1-score	Average RMSE
FHMM	43.2	9.53	0.84	66.79
COOP	1.45	0.811	0.37	72.34
EDPF	47.96	121.95	0.67	102.48
DDSC	62.25	2.3	0.54	78.78

Table 3: *Results on training and testing on one same day (sampling at 1 s).*

	Training time	Prediction time	Average F1-score	Average RMSE
FHMM	16.06	5.21	0.58	136.40
COOP	1.46	2.56	0.45	121.00
EDPF	15.95	33.73	0.45	176.50
DDSC	13.37	1.44	0.33	134.76

Table 4: *Results on training and testing on 8 days (sampling at 30 s)*

We can get an idea of how the disaggregators behave from this short summary. First, we can see that both FHMM and EDPF take roughly the same amount of time to train, because they both fit HMMs to the data. Combinatorial optimization seems to train pretty quickly, and this is so because it just does some clustering on the input chunks. DDSC is the algorithm that takes most time to train, probably due to the size of the matrices it deals with.

In case of disaggregating, all the algorithms run fairly quickly except EDPF, which for every data point needs to iterate over all the particles. It takes roughly 10x the time of FHMM, since it is using 10 particles. On the results, we can see how FHMM performs best, with the least error and the highest average F1-score. EDPF and DDSC do okay in F1-score, but EDPF gets the highest error. In following experiments it may be useful to reduce the sample period for EDPF and increase the number of particles, to be able to explore more state space.

8.2 Testing all disaggregators

In this section we will expand the testing to other buildings in REDD. The training has been done on 60% of the data, and disaggregators will be tested on the remaining 40%. Here are the results for each house:

		Training time	Prediction time	Average F1-score	Average RMSE
House 1	**FHMM**	167.54	5.69	0.45	176.66
	COOP	2.82	1.48	0.36	197.81
	EDPF	182.31	61.14	0.5	240.39
	DDSC	297.62	10.05	0.2	147.57
House 3	**FHMM**	86.4	13.87	0.59	124.89
	COOP	2.62	3.16	0.45	132.72
	EDPF	93.49	158.57	0.6	266.44
	DDSC	144.0	5.4	0.2	198.98
House 4	**FHMM**	111.06	15.38	0.43	72.37
	COOP	2.1	3.71	0.31	71.49
	EDPF	117.72	169.65	0.38	91.53
	DDSC	173.53	6.14	0.2	48.86
House 5	**FHMM**	20.51	2.59	0.5	91.15
	COOP	1.82	1.42	0.26	114.32
	EDPF	22.72	16.53	0.42	291.46
	DDSC	24.46	1.52	0.2	83.61
House 6	**FHMM**	77.55	13.95	0.44	100.68
	COOP	2.51	3.83	0.44	104.00
	EDPF	83.95	149.13	0.46	211.94
	DDSC	120.94	5.10	0.33	158.98

Table 5: *Results on all the REDD houses with sampling time 5 seconds.*

	Average F1-score	Average RMSE
FHMM	0.48	113.15
COOP	0.36	124.07
EDPF	0.47	220.352
DDSC	0.23	127.6

Table 6: *Performance summary of the four algorithms: values are computed by averaging each house's results from Table 5.*

In these results we can see that even though results tend to be quite different from house to house, FHMM consistently reports better scores than the other algorithms in terms of F1-score and RMSE. The performance of the whole set of algorithms is quite variable across houses, probably due to the density of the data we have for each one.

An interesting result we can notice here is that EDPF usually achieves an average F1-score which is higher than the others, but on the other hand its average RMSE error is the worst of the whole set. In practice, this means that EDPF identifies better than the others when is a device on, but it fails to assign it a proper power value. This is probably due to how EDPF models the power states of each device: a normal distribution centered at all the potential power state centers, with a constant standard deviation.

Conversely, DDSC reports consistently bad F1 scores, but surprisingly, the average RMSE error is better than other algorithms. This means that DDSC has many false positives/negatives (probably due to the fact that it does not know about each device's power), but DDSC on average catches the power value of the devices more effectively. F1 scores of DDSC could be improved by providing it with more information about the devices, or simply by cutting off low power values returned by the algorithm. It may be that there may be many instances in which devices are assigned a very low value (maybe just a few watts) due to the nature of the matrix computations, so in this next experiment we will re-run the same tests, but now only with DDSC with a slight modification so it cuts off power values lesser than 30W.

	Average F1-score	Average RMSE
House 1	0.2	147.72
House 3	0.2	199.16
House 4	0.2	48.78
House 5	0.2	83.26
House 6	0.33	160.98

Table 7: *Results for DDSC when we cut off the appliance power whenever it's less than 30W.*

Results for the 30W cutoff can be found in Table 7. Unfortunately, the very-low power values are not the cause of the low F1 scores. After looking into the results more in detail, these F1 scores correspond to results in which one or two devices are not disaggregated properly, while the rest are. These 1 or 2 failing devices have a very low F1-score that drives the average result down.

8.3 Testing EDPF with more particles

Now, let's see how EDPF scales with the number of particles. The hypotheses is that runtime will scale linearly with the number of particles, but the results may not even improve: we need to have enough particles to be "lucky" and explore the right part of

the solution space. We will re-run EDPF with the same data as before, now with 20, 30, 50, and 100 particles. Here are the results:

	Training time	Prediction time	Average F1-score	Average RMSE
House 1	182.63	112.08	0.49	242.56
House 3	96.50	291.44	0.63	265.11
House 4	116.92	313.41	0.39	85.66
House 5	21.11	29.11	0.42	291.31
House 6	82.48	276.93	0.47	214.72

Table 8: *Results of running EDPF with 20 particles.*

	Training time	Prediction time	Average F1-score	Average RMSE
House 1	180.14	166.80	0.49	231.84
House 3	93.74	443.60	0.63	276.78
House 4	117.47	472.79	0.40	79.11
House 5	22.93	44.39	0.42	286.05
House 6	86.65	427.35	0.47	208.24

Table 9: *Results of running EDPF with 30 particles.*

	Training time	Prediction time	Average F1-score	Average RMSE
House 1	186.85	282.15	0.50	230.38
House 3	102.65	743.15	0.60	265.50
House 4	125.51	786.23	0.40	79.10
House 5	20.98	72.39	0.44	285.81
House 6	83.04	694.43	0.49	205.68

Table 10: *Results of running EDPF with 50 particles.*

	Training time	Prediction time	Average F1-score	Average RMSE
House 1	189.63	545.35	0.51	227.95
House 3	98.31	1437.31	0.61	247.1
House 4	121.56	1540.51	0.40	75.95
House 5	21.79	142.05	0.39	291.70
House 6	88.19	1375.17	0.50	205.33

Table 11: *Results of running EDPF with 100 particles.*

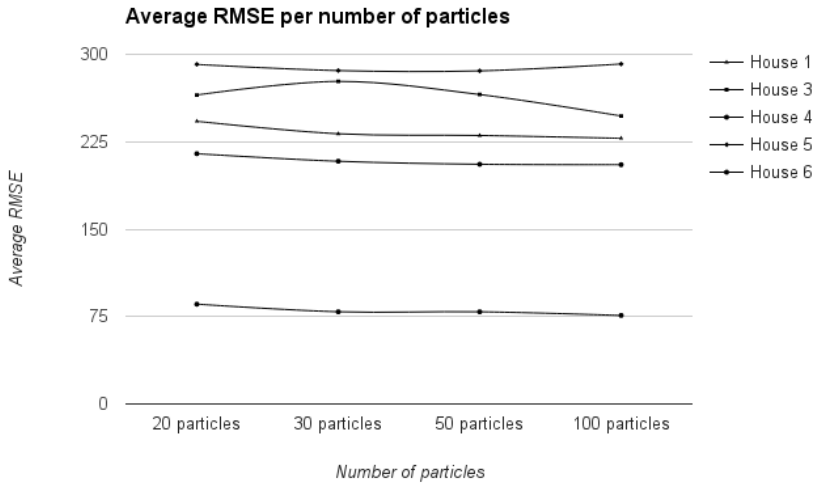


Figure 12: *Scaling of EDPF's average RMSE for different number of particles.*

We can see that increasing the number of particles offers marginal improvements in both the F1-score and RMSE, and that in some times we get worse results (due to the random nature of the algorithm). This means that the particles are not the cause for the low scores, and that the cause is probably the training phase: EDPF needs to fit better the devices at training time, and no matter how many particles we use, we cannot explore the true state space if the models we have are wrong. The positive point about EDPF is that it is very parallelizable, since the scaling in time is linear with respect to the number of particles, and the sequential Monte Carlo method can run many of them in parallel.

8.4 Testing DDSC with lower sampling rates

We mentioned that DDSC is designed to work in environments with low sampling rate, but all the previous tests were performed with sampling rates of less than a minute. We will now compare how DDSC works with different sampling periods. Due to missing data, we have only been able to get results from houses 1 and 6 with hourly sampling rates during a week. This is the performance of DDSC with different sampling rates:

	10 minutes		30 minutes		60 minutes	
	RMSE	F1	RMSE	F1	RMSE	F1
House 1	124.84	0.28	78.11	0.20	57.95	0.20
House 6	164.45	0.32	167.11	0.31	147.62	0.29

Table 12: *DDSC results with lower sampling rates in houses 1 and 6 from REDD dataset.*

As expected, DDSC tends to perform better the lower the sampling rate is; with the same data, lower sampling rates give lower RMSE, but also lower F1 score. This means that while DDSC gets worse at classifying what devices are on/off, it gets better at assigning powers to each one of them. This may be so due to the amount of bases

that were used (5 in this test), which may be too few to disaggregate data at a higher sampling rate, but they work just fine with 1-hour readings.

8.5 Testing AFAMAP: reduced datasets

Unfortunately, managing the amounts of data that the other disaggregators were using proved to be too much for my implementation of AFAMAP, making Gurobi run out of memory pretty early in the execution. For this reason, I have reduced the REDD dataset to multiple shorter-length periods - which AFAMAP can properly handle. Also, since AFAMAP was implemented under Windows, and the NILMTK installation failed in that OS, the only algorithms being used in the comparison will be EDPF and DDSC; the metrics are the same as computed by NILMTK, but manually implemented using the scikit-learn Python library. Here are the results for House 1 with a sampling rate of 60 seconds:

	AFAMAP		DDSC		EDPF	
	RMSE	F1	RMSE	F1	RMSE	F1
House 1	51687.50	0.85	177.17	0.72	58393.8	0.75

Table 13: *AFAMAP, DDSC and EDPF on 1 day worth of data, with a sample period of 60 seconds.*

This quick test gives us a bit of insight on how AFAMAP works. The run time of AFAMAP is not included in this table, as experiments were done in a different machine but it is very large compared to the other algorithms - AFAMAP takes around 5 minutes to disaggregate one day worth of data.

Second, it is interesting to see such high RMSE reported by AFAMAP (and also by EDPF). Upon further inspection, this corresponds to some periods of failing to disaggregate high-profile devices that consume hundreds of watts, which produces such a very high number. EDPF suffers from this as well, and the reason is the low sampling rate. This fits exactly with the assumption explained in the AFAMAP section earlier: it works based on the "one-at-a-time" assumption, which says that at every data point, only one device at most should be switched on or off. In order to test AFAMAP better, we need to increase the sampling rate without running out of memory, which means that we need to take shorter periods of time:

		AFAMAP		DDSC		EDPF	
	Hour	RMSE	F1	RMSE	F1	RMSE	F1
House 1	1	3.00	1.0	14.04	1.0	10.08	0.99
	2	10.65	1.0	18.06	1.0	7.29	0.99
	3	268.43	1.0	6.1	1.0	347.96	1.0
	4	10.58	1.0	15.20	1.0	94.16	0.99
House 3	1	4.94	0.96	9.18	0.94	7.55	0.95
	2	7.40	0.78	2.19	0.91	66.18	0.92
	3	1546.78	0.93	73.78	0.99	3954.64	0.99
	4	373.78	0.86	15.02	0.78	416.63	0.89
House 4	1	4.94	0.96	45.34	0.95	619.80	0.94
	2	2.75	1.0	9.41	1.0	11.71	1.0
	3	14.56	0.93	8.72	0.92	51.36	0.99
	4	323.27	1.0	13.97	1.0	207.41	0.99

Table 14: *AFAMAP, DDSC and EDPF on 1 hour intervals with a sample period of 10 seconds.*

AFAMAP results are much more sensible now that the sampling rate is increased. Unfortunately, AFAMAP takes much longer to run (up to half an hour for some instances), and usually runs out of memory in the machine I used for testing it. For this reason, we are not able to provide results with bigger samples or offer meaningful run time comparisons.

It is interesting to see that AFAMAP and EDPF perform poorly on mostly the same instances. This is probably due to, again, the way we are fitting the HMMs to the real data. Since AFAMAP and EDPF use the same models, they struggle when the fitting is not good enough. However, AFAMAP always works better than EDPF, which provides us with an upper-bound on how good EDPF could be.

When comparing across each house and all these test instances, we can see that AFAMAP performs as well as the others, and in some cases outperforms them. However, it has a few disadvantages that make it less preferable than, for example, DDSC, in this set of experiments. The disadvantages here are the combination of the need for a high sampling rate and the computational requirements, making AFAMAP hard to test with in this machine. In a real-life scenario, however, AFAMAP can be deployed as an unsupervised algorithm, and as we said before, this alone can make all the difference for consumers.

8.6 Conclusions

Thanks to these experiments we have gained some insight into the workings of each one of the algorithms. It is clear that they have all strengths and weaknesses, and in some cases it may be preferable to use one before the others.

One of the key factors to decide on a suitable algorithm is the existence of training data. In a real life application, it may be too hard to acquire training data for most of the devices in a house. This can be solved by only monitoring a subset of the devices (e.g. the top 5 appliances in a house), but in the end it is an extra cost that we need to take into account, and if we have access to this kind of data then we do not need a disaggregation algorithm. The solution to this problem is to train algorithms across multiple houses and use the results to test on new houses, but this requires further research.

Another key factor is the sampling rate: it is more expensive to monitor the power signal of a household at high sampling rates, and maybe it is just more practical to have a low sampling rate, lower accuracy algorithm.

For practical purposes, DDSC seems to be the algorithm that could be deployed right now, as current smart meters make energy readings at hourly ratings [12]. However, the algorithm needs training data on the home appliances, and for this reason, DDSC should be trained across multiple houses, offline, in order to learn common usage patterns for devices. This remains as future work, as we have not explored this possibility in this project.

The algorithms that require a high sampling rate first need infrastructure to measure power signal at higher frequencies. They would need high-end smart meters, or more specific solutions as discussed in [12]. If we ignore this issue, the most attractive algorithm is AFAMAP, because it does not require training data. However, it should be tuned to automatically label the learned device sets with easy to understand names, maybe with a database of common device behaviours - otherwise AFAMAP will provide non-actionable feedback, which is not very useful.

We could also apply the same strategy to any other algorithm, and have all algorithms go through a phase of offline training. If this is the case, and we assume that we have good enough training data, it seems that FHMM would be the best choice; it is slower than the others, but we have plenty of time to run it, as feedback does not need to be real-time (it can come with a few hours of delay, or even it can be presented as a daily report).

In conclusion, we have the algorithms to disaggregate our energy signals, but what we lack right now is infrastructure to deploy them. Unless we are willing to pay for custom meters and plugs, our best disaggregator candidate right now is DDSC - always assuming it has access to the smart meter readings.

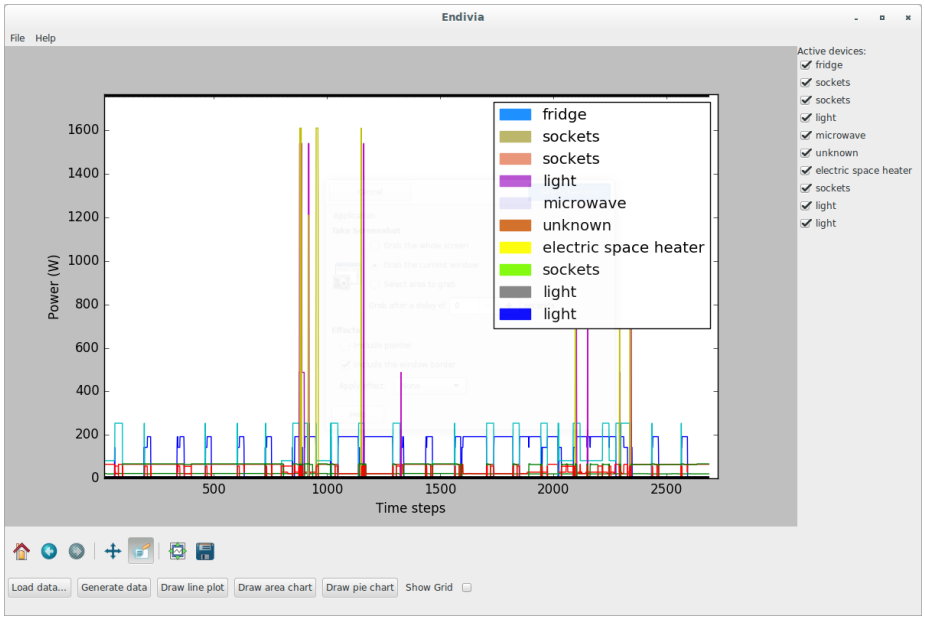
9 Visualizing results

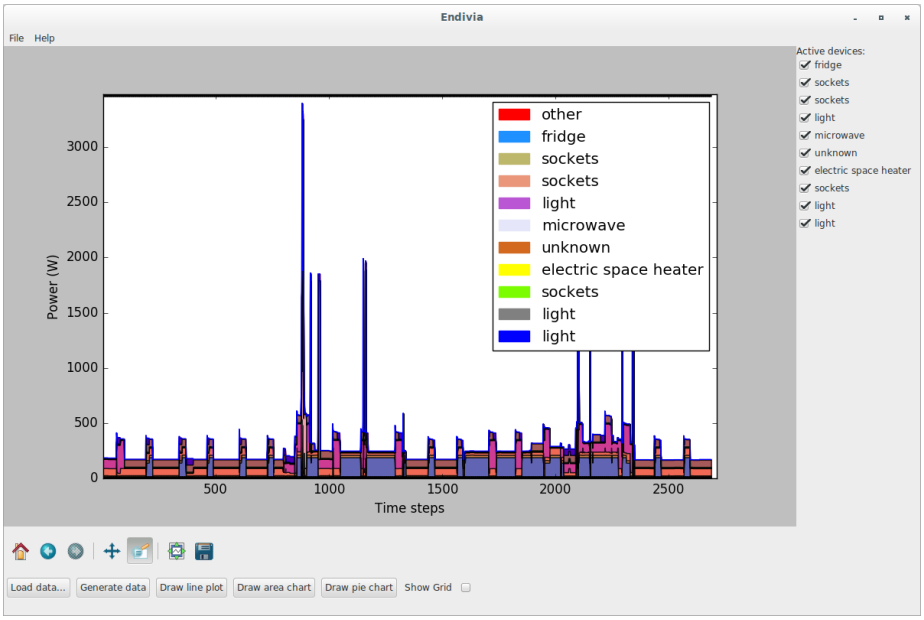
For desaggregation to be useful for a regular user, results have to be presented meaningfully and visually. To achieve this, we have created Endivia (ENergy DIsaggregation VIsualizing Application) to give users an easy-to-understand graph with a summary of the disaggregation info.

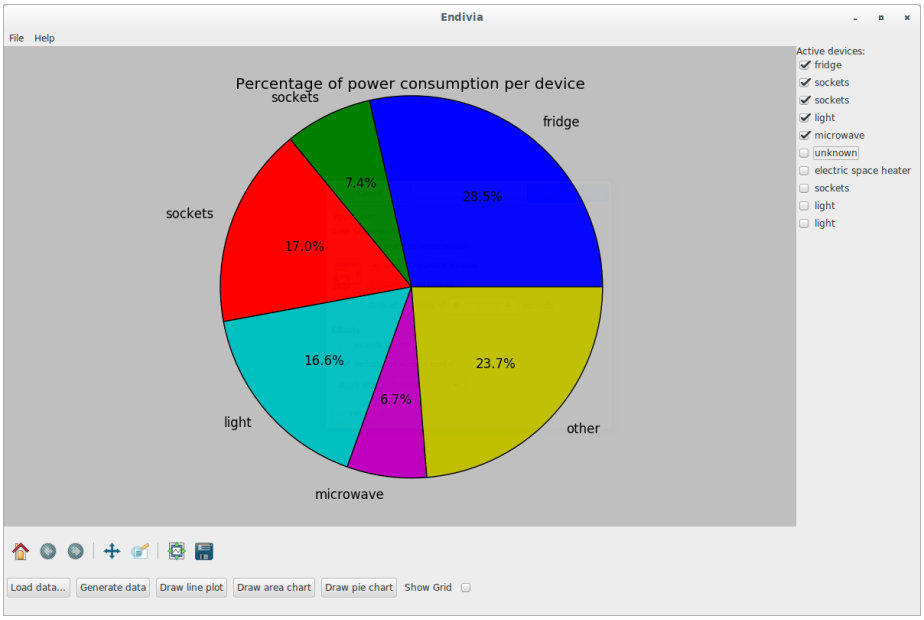
Endivia is a simple Python script that plots the disaggregated signal in a meaningful way. It works on HDFS files, which are produced by NILMTK and represent the disaggregated signals of every device, as well as more information. With this data Endivia allows the user to understand their energy consumption profile at a glance.

Endivia plots the disaggregated data as a line plot, an area chart, and a pie chart. The line plot is very useful to see the devices usage patterns and their relative consumptions to the others along the time. However, it is hard to combine multiple devices in this plot, so the area chart stacks them to show which are the most consuming at every point in time. Finally, the pie chart represents the percentage of power consumed of each device, so it is very easy to see what devices should the consumers prioritize on if they want to reduce their power consumption.

Here are some screenshots of Endivia's plots, with the data of house 1 in the REDD dataset, disaggregating the top 10 appliances and doing it at a sample rate of 30 seconds:







In a real-life application, Endivia could be deployed on a smartphone or tablet, and it would show the disaggregated signal in real time. For now, the initial version of Endivia only works on batch mode, but it could be easily deployed as a real-time application. As we have seen in the different studies, a real-time feedback would be very beneficial to encourage energy saving habits.

10 Future work

Even though the presented algorithms manage to disaggregate a signal well, they can be improved a long way to get even better results. This line of research could be continued in several ways:

- **Explore other solutions.** As we have seen in section 3, energy disaggregation can be approached from many different angles. One interesting approach is the use of deep learning, as this has already been explored for source separation [52], but it can be worth to explore energy disaggregation as a problem for LSTM (long short-term memory) neural networks [53]. We also have not used the high-frequency voltage data, which can be added into the models for more powerful disaggregation.
- **Training on multiple houses.** We may be able to disaggregate the power signal of a house with training data of that house, but if we want to make a viable disaggregation solution, we will need to train on a test of houses that is different from the consumer houses.
- **Algorithm extensions.** The algorithms presented here can all be extended. In particular, the original DDSC paper presents three separate extensions that affect the results in different ways; particle filters can be extended with plenty of heuristics, and the original AFAMAP paper presents a way of learning device HMMs from the aggregate signal - it would also benefit from being run in a more powerful machine, or also a cluster. EDPF can also benefit from using existing libraries to fit HMMs such as *hmmlearn*.
- **Further testing and tweaking.** Before deploying these algorithms in a real-life scenario, we need to further test as many cases as we want and tweak the algorithms so they can automatically adjust to each problem's particular signature.
- **Endivia development and testing.** In the future the visualizer could be deployed on smartphones and tablets to provide users with a quick overview of their power consumption. As conveying information is hard, there needs to be some user experience research in order to design a usable visualizer.
- **Focus on unsupervised methods.** The methods we have presented all need training data to work. However, DDSC can be used to learn appropriate bases from a training set and then use those to disaggregate the signal of a different set of houses; also, EDPF and AFAMAP can be run in an unsupervised way. This would prove to be quite useful to the end user, avoiding the need for installing smart meters in their homes. Energy companies could use the solution to provide the users with real-time or almost real-time feedback.
- **Commercialization.** One possible outcome of this research is the creation and sale of a new product. To do this, a market study needs to be carried out and then a strategy must be designed around the study. Given the state of the energy industry and the introduction of smart meters, this could potentially be a commercially successful product.

11 Conclusion

In this project we have implemented and compared multiple algorithms for energy disaggregation. This process, also known as non-intrusive load monitoring, promises to be a great tool to estimate our power consumption and grow healthy habits, as multiple studies have shown. On the technical side, NILM can be performed in many different ways: from basic signal processing techniques, to more sophisticated probabilistic analysis, machine learning or optimization problems. On the consumer side, NILM is a tool that has shown to have a positive impact on users' habits, and can make consumers save money and protect the environment.

Specifically, we have explored three algorithms: DDSC, based on sparse coding; EDPF, based on particle filters; and AFAMAP, based on integer programming. We have integrated DDSC and EDPF in NILMTK, which is an open-source toolkit that aims to unify development and testing of NILM algorithms. We have tested the different algorithms on the REDD dataset, and the algorithms have shown acceptable results, sometimes outperforming NILMTK's algorithms. Maybe more important than this, we have gained some insight into the disaggregation process and the algorithms' behaviours and requirements.

Energy disaggregation remains an open problem, but many research groups are tackling the problem from different fronts, and our interest in this area grows every year. As more powerful techniques (for example, deep learning) start being a part of many commercial products, and as more data is available, we will start to see commercial disaggregators for the public to use. Only at that point we will be able to see whether energy disaggregation is a viable solution for us to acquire better habits.

References

[1] George W Hart. Nonintrusive appliance load monitoring. *Proceedings of the IEEE*, 80(12):1870–1891, 1992.

[2] B Neenan, J Robinson, and RN Boisvert. Residential electricity use feedback: A research synthesis and economic framework. *Electric Power Research Institute*, 2009.

[3] Sarah Darby et al. The effectiveness of feedback on energy consumption. *A Review for DEFRA of the Literature on Metering, Billing and direct Displays*, 486:2006, 2006.

[4] Agnim Cole, Alexander Albicki, et al. Algorithm for nonintrusive identification of residential appliances. In *Circuits and Systems, 1998. ISCAS'98. Proceedings of the 1998 IEEE International Symposium on*, volume 3, pages 338–341. IEEE, 1998.

[5] Christopher Laughman, Kwangduk Lee, Robert Cox, Steven Shaw, Steven Leeb, Les Norford, and Peter Armstrong. Power signature analysis. *Power and Energy Magazine, IEEE*, 1(2):56–63, 2003.

[6] Shwetak N Patel, Thomas Robertson, Julie A Kientz, Matthew S Reynolds, and Gregory D Abowd. At the flick of a switch: Detecting and classifying unique electrical events on the residential power line. *Lecture Notes in Computer Science*, 4717:271–288, 2007.

[7] J Zico Kolter, Siddharth Batra, and Andrew Y Ng. Energy disaggregation via discriminative sparse coding. In *Advances in Neural Information Processing Systems*, pages 1153–1161, 2010.

[8] Jack Kelly and William Knottenbelt. Neural nilm: Deep neural networks applied to energy disaggregation. *arXiv preprint arXiv:1507.06594*, 2015.

[9] Dominik Egarter, Venkata Pathuri Bhuvana, and Wilfried Elmenreich. Paldi: Online load disaggregation via particle filtering. *Instrumentation and Measurement, IEEE Transactions on*, 64(2):467–477, 2015.

[10] J Zico Kolter and Tommi Jaakkola. Approximate inference in additive factorial hmms with application to energy disaggregation. In *International conference on artificial intelligence and statistics*, pages 1472–1482, 2012.

[11] Nipun Batra, Jack Kelly, Oliver Parson, Haimonti Dutta, William Knottenbelt, Alex Rogers, Amarjeet Singh, and Mani Srivastava. Nilmtk: an open source toolkit for non-intrusive load monitoring. In *Proceedings of the 5th international conference on Future energy systems*, pages 265–276. ACM, 2014.

[12] K Carrie Armel, Abhay Gupta, Gireesh Shrimali, and Adrian Albert. Is disaggregation the holy grail of energy efficiency? the case of electricity. *Energy Policy*, 52:213–234, 2013.

[13] Lawrence Page, Sergey Brin, Rajeev Motwani, and Terry Winograd. The pagerank citation ranking: bringing order to the web. 1999.

[14] Claude Elwood Shannon. A mathematical theory of communication. *ACM SIGMOBILE Mobile Computing and Communications Review*, 5(1):3–55, 2001.

[15] Walter R Gilks. *Markov chain monte carlo*. Wiley Online Library, 2005.

[16] Lawrence R Rabiner. A tutorial on hidden markov models and selected applications in speech recognition. *Proceedings of the IEEE*, 77(2):257–286, 1989.

[17] Julian Kupiec. Robust part-of-speech tagging using a hidden markov model. *Computer Speech & Language*, 6(3):225–242, 1992.

[18] Anders Krogh, BjoÈrn Larsson, Gunnar Von Heijne, and Erik LL Sonnhammer. Predicting transmembrane protein topology with a hidden markov model: application to complete genomes. *Journal of molecular biology*, 305(3):567–580, 2001.

[19] Zoubin Ghahramani and Michael I Jordan. Factorial hidden markov models. *Machine learning*, 29(2-3):245–273, 1997.

[20] Tuomas Virtanen. Speech recognition using factorial hidden markov models for separation in the feature space. In *INTERSPEECH*. Citeseer, 2006.

[21] Bernd Fischer, Volker Roth, Franz Roos, Jonas Grossmann, Sacha Baginsky, Peter Widmayer, Wilhelm Gruissem, and Joachim M Buhmann. Novohmm: a hidden markov model for de novo peptide sequencing. *Analytical chemistry*, 77(22):7265–7273, 2005.

[22] Gautham J Mysore, Paris Smaragdis, and Bhiksha Raj. Non-negative hidden markov modeling of audio with application to source separation. In *Latent variable analysis and signal separation*, pages 140–148. Springer, 2010.

[23] Travis E Oliphant. *A guide to NumPy*, volume 1. Trelgol Publishing USA, 2006.

[24] Eric Jones, Travis Oliphant, and Pearu Peterson. {SciPy}: open source scientific tools for {Python}. 2014.

[25] Honglak Lee, Alexis Battle, Rajat Raina, and Andrew Y Ng. Efficient sparse coding algorithms. In *Advances in neural information processing systems*, pages 801–808, 2006.

[26] Ian Jolliffe. *Principal component analysis*. Wiley Online Library, 2002.

[27] Ben Taskar, Vassil Chatalbashev, Daphne Koller, and Carlos Guestrin. Learning structured prediction models: A large margin approach. In *Proceedings of the 22nd international conference on Machine learning*, pages 896–903. ACM, 2005.

[28] Michael Collins. Discriminative training methods for hidden markov models: Theory and experiments with perceptron algorithms. In *Proceedings of the ACL-02 conference on Empirical methods in natural language processing-Volume 10*, pages 1–8. Association for Computational Linguistics, 2002.

[29] Adrian Smith, Arnaud Doucet, Nando de Freitas, and Neil Gordon. *Sequential Monte Carlo methods in practice*. Springer Science & Business Media, 2013.

[30] Arnaud Doucet and Adam M Johansen. A tutorial on particle filtering and smoothing: Fifteen years later. *Handbook of Nonlinear Filtering*, 12(656-704):3, 2009.

[31] Dieter Fox, Jeffrey Hightower, Lin Liao, Dirk Schulz, and Gaetano Borriello. Bayesian filtering for location estimation. *IEEE pervasive computing*, (3):24–33, 2003.

[32] Richard J Meinhold and Nozer D Singpurwalla. Understanding the kalman filter. *The American Statistician*, 37(2):123–127, 1983.

[33] PJ Hargrave. A tutorial introduction to kalman filtering. In *Kalman Filters: Introduction, Applications and Future Developments, IEE Colloquium on*, pages 1–1. IET, 1989.

[34] Christopher Z Mooney. *Monte carlo simulation*, volume 116. Sage Publications, 1997.

[35] M Sanjeev Arulampalam, Simon Maskell, Neil Gordon, and Tim Clapp. A tutorial on particle filters for online nonlinear/non-gaussian bayesian tracking. *Signal Processing, IEEE Transactions on*, 50(2):174–188, 2002.

[36] John Geweke. Bayesian inference in econometric models using monte carlo integration. *Econometrica: Journal of the Econometric Society*, pages 1317–1339, 1989.

[37] Neil J Gordon, David J Salmond, and Adrian FM Smith. Novel approach to nonlinear/non-gaussian bayesian state estimation. In *Radar and Signal Processing, IEE Proceedings F*, volume 140, pages 107–113. IET, 1993.

[38] Tood K Moon. The expectation-maximization algorithm. *Signal processing magazine, IEEE*, 13(6):47–60, 1996.

[39] Leonard E Baum, Ted Petrie, George Soules, and Norman Weiss. A maximization technique occurring in the statistical analysis of probabilistic functions of markov chains. *The annals of mathematical statistics*, 41(1):164–171, 1970.

[40] Lloyd R Welch. Hidden markov models and the baum-welch algorithm. *IEEE Information Theory Society Newsletter*, 53(4):10–13, 2003.

[41] James MacQueen et al. Some methods for classification and analysis of multivariate observations. In *Proceedings of the fifth Berkeley symposium on mathematical statistics and probability*, volume 1, pages 281–297. Oakland, CA, USA., 1967.

[42] Leonid I Rudin, Stanley Osher, and Emad Fatemi. Nonlinear total variation based noise removal algorithms. *Physica D: Nonlinear Phenomena*, 60(1):259–268, 1992.

[43] Antonin Chambolle. An algorithm for total variation minimization and applications. *Journal of Mathematical imaging and vision*, 20(1-2):89–97, 2004.

[44] Daniel Bienstock. Computational study of a family of mixed-integer quadratic programming problems. *Mathematical programming*, 74(2):121–140, 1996.

[45] Peter J Huber et al. Robust estimation of a location parameter. *The Annals of Mathematical Statistics*, 35(1):73–101, 1964.

[46] Alvaro Barbero and Suvrit Sra. Fast newton-type methods for total variation regularization. In *Proceedings of the 28th International Conference on Machine Learning (ICML-11)*, pages 313–320, 2011.

[47] G David Forney Jr. The viterbi algorithm. *Proceedings of the IEEE*, 61(3):268–278, 1973.

[48] Ming Yuan and Yi Lin. Model selection and estimation in regression with grouped variables. *Journal of the Royal Statistical Society: Series B (Statistical Methodology)*, 68(1):49–67, 2006.

[49] Laurent Condat. A direct algorithm for 1d total variation denoising. *IEEE Signal Processing Letters*, 20(11):1054–1057, 2013.

[50] J. Zico Kolter and J. Johnson. REDD: A public data set for energy disaggregation research. *Proceedings of the SustKDD workshop on Data Mining Applications in Sustainability*, 2011.

[51] Andrea Monacchi, Dominik Egarter, Wilfried Elmenreich, Salvatore D'Alessandro, and Andrea M Tonello. Greend: An energy consumption dataset of households in italy and austria. In *Smart Grid Communications (SmartGridComm), 2014 IEEE International Conference on*, pages 511–516. IEEE, 2014.

[52] Po-Sen Huang, Minje Kim, Mark Hasegawa-Johnson, and Paris Smaragdis. Deep learning for monaural speech separation. In *Acoustics, Speech and Signal Processing (ICASSP), 2014 IEEE International Conference on*, pages 1562–1566. IEEE, 2014.

[53] Felix A Gers, Jürgen Schmidhuber, and Fred Cummins. Learning to forget: Continual prediction with lstm. *Neural computation*, 12(10):2451–2471, 2000.