Towards reproducible state-of-the-art energy disaggregation

Nipun Batra
nipun.batra@iitgn.ac.in
IIT Gandhinagar, India

Rithwik Kukunuri
kukunuri.sai@iitgn.ac.in
IIT Gandhinagar, India

Ayush Pandey
ayush20pandey@gmail.com
NIT Agartala, India

Raktim Malakar
raktimmalakar2015@gmail.com
IIEST Shibpur, India

Rajat Kumar
201811024@daiict.ac.in
DAIICT, India

Odysseas Krystalakos
odysseaskrst@gmail.com
Aristotle University of Thessaloniki, Greece

Mingjun Zhong
MZhong@lincoln.ac.uk
University of Lincoln, UK

Paulo Meira
pmeira@ieee.org
University of Campinas, Brazil

Oliver Parson
oliver.parson@hivehome.com
Centrica Hive Limited, UK

ABSTRACT

Non-intrusive load monitoring (NILM) or energy disaggregation is the task of separating the household energy measured at the aggregate level into constituent appliances. In 2014, the NILM toolkit (NILMTK) was introduced in an effort towards making NILM research reproducible. Despite serving as the reference library for data set parsers and reference benchmark algorithm implementations, few publications presenting algorithmic contributions within the field went on to contribute implementations back to the toolkit. This paper describes two significant contributions to the NILM community in an effort towards reproducible state-of-the-art research: i) a rewrite of the disaggregation API and a new experiment API which lower the barrier to entry for algorithm developers and simplify the definition of algorithm comparison experiments, and ii) the release of NILMTK-contrib; a new repository containing NILMTK-compatible implementations of 3 benchmarks and 9 recent disaggregation algorithms. We have performed an extensive empirical evaluation using a number of publicly available data sets across three important experiment scenarios to showcase the ease of performing reproducible research in NILMTK.

CCS CONCEPTS

• **Computing methodologies → Machine learning algorithms**.

KEYWORDS

energy disaggregation; non-intrusive load monitoring; smart meters

ACM Reference Format:
Nipun Batra, Rithwik Kukunuri, Ayush Pandey, Raktim Malakar, Rajat Kumar, Odysseas Krystalakos, Mingjun Zhong, Paulo Meira, and Oliver Parson. 2019. Towards reproducible state-of-the-art energy disaggregation. In *The 6th ACM International Conference on Systems for Energy-Efficient Buildings, Cities, and Transportation (BuildSys '19), November 13–14, 2019, New York, NY, USA.* ACM, New York, NY, USA, 10 pages. https://doi.org/10.1145/3360322.3360844

1 INTRODUCTION

Non-intrusive load monitoring (NILM) or energy disaggregation is the task of separating a building's energy measured at the aggregate level into constituent appliances. The problem was originally studied by Hart in the early 1980s [9] and has seen a renewed interest in recent years owing to the availability of larger data sets, smart meter rollouts, and amidst climate change concerns.

Despite more than three decades of research in the field, three factors primarily affected reproducibility, and therefore empirical comparison of NILM algorithms: i) it was hard to assess generality of NILM approaches as most works were evaluated on a single data set, ii) there was lack of comparison using the same benchmarks due to the lack of availability of open-source benchmark implementations, and iii) different metrics were used based on the use case under consideration. The open source non-intrusive load monitoring toolkit (NILMTK) [3] was released in early 2014 against this background to enable easy comparison of NILM algorithms in a reproducible fashion. The main contributions of the toolkit were: i) NILMTK-DF (data format): the standard energy disaggregation data structure used by NILMTK; ii) parsers for six existing data sets; iii) implementations of two benchmark NILM algorithms; iv) statistical and diagnostic functions for understanding data sets; v) a suite of accuracy metrics across a range of use cases. Later in 2014, NILMTK v0.2 was released [12] which added support for out-of-core computation, motivated by release of very large data sets such as Dataport data set [18].

Since these two releases, NILMTK has become the energy disaggregation field's reference library for data set parsers and reference benchmark algorithm implementations. However, few publications presenting algorithmic contributions within the field went on to contribute implementations back to the toolkit. As a result, new publications generally compare a novel algorithm with a baseline benchmark algorithm instead of the state-of-the-art within the field. Consequently, it is still not possible to compare the performance of state-of-the-art algorithms side-by-side, therefore limiting progress within the field.

Against this background, this paper describes two significant contributions to the NILM research community. First, we have rewritten the disaggregation API and implemented a new experiment API, which respectively lowers the barrier to entry for algorithm developers and simplify the definition of algorithm comparison experiments. We have also made a number of practical improvements to the toolkit's installation process, data set parsers and documentation. Second, we have released NILMTK-contrib[1]; a new repository containing NILMTK-compatible implementations of three benchmarks and nine recently published disaggregation algorithms [13, 14, 16, 21, 22]. For the first time, algorithm developers will be able to compare the performance of a new approach with state-of-the-art algorithms in a range of different settings.

To demonstrate the potential of these releases, we have performed an extensive empirical evaluation using a number of publicly available data sets. We demonstrate the versatility of the new experiment API by conducting experiments across the following three train/test scenarios: i) train and test on a different building from the same data set, ii) train on multiple buildings from different data sets and iii) train and test on artificially generated aggregate data (by summing appliance usage rather than measuring the building aggregate). Furthermore, we demonstrate the potential of NILMTK-contrib by comparing the performance of the three benchmarks and nine disaggregation algorithms included in NILMTK-contrib in each of these settings. This evaluation is the most extensive empirical comparison of energy disaggregation algorithms to date. However, we have only evaluated algorithm performance using a single accuracy metric, and although many metrics are available within NILMTK, a comparison across multiple metrics is beyond the scope of this paper.

This represents a significant release of NILMTK and addresses a key issue faced by the community, as evidenced by GitHub issue queue. With this release, we believe that we have lowered the barrier to entry for the NILM community to embrace reproducible research. In addition, we believe that this release will also pave the way for future NILM competitions.

The remainder of this paper is structured as follows. First, we formally introduce the energy disaggregation problem and discuss the issues with existing toolkit. We then describe changes to the core toolkit, including updates to the experiment API and the new model interface. Next, we introduce the NILMTK-contrib repository and provide an overview for the supported algorithms. We then demonstrate the value of such releases through an empirical comparison, before summarising our contributions.

2 BACKGROUND

In this section, we introduce the mathematical definition of energy disaggregation and provide an overview of the NILM research field, before summarising NILMTK and discussing its current limitations.

2.1 The NILM Model

Suppose we have observed a time series of aggregate measurements $Y = (Y_1, Y_2, \cdots, Y_T)$, where $Y_t \in R_+$ represents the energy or power measured in Watt-hours or Watts by an electricity meter at time t. This signal is assumed to be the aggregation of energy consumed by the component appliances in a building. In the following, we assume there are I appliances, and for each appliance the energy signal is represented as $X_i = (x_{i1}, x_{i2}, \cdots, x_{iT})$ where $x_{it} \in R_+$. The mains readings can then be represented as the summation of the appliance signals with the following form: $Y_t = \sum_{i=1}^{I} x_{it} + \epsilon_t$ where ϵ_t is an error term. The aim of the energy disaggregation problem, i.e., NILM, is to recover the unknown signals X_i given only the observed aggregate measurements Y.

[1]https://github.com/nilmtk/nilmtk-contrib

2.2 Overview of NILM Research

The field of NILM was introduced by George Hart in the early 1980s [9]. In the past decade, research in the field has accelerated due to smart meter rollouts across different countries and the efforts towards emission reduction. The availability of public data sets (more than 10 across different geographies) in recent years has also generated significant research activity.

Various algorithms have been proposed for solving NILM since the inception of the field. These mainly include signal processing methods that use explicit features of appliances for the purpose of disaggregation [10, 11], and machine learning approaches including, unsupervised and supervised learning. Many methods [15, 19, 22, 23] used probabilistic approaches to explicitly model appliances' energy consumption via a hidden Markov model (HMM). Another category of methods [2, 5, 14] leverage the low-rank structure of energy consumption to perform energy disaggregation using factorisation techniques. Furthermore, the success of neural networks in other domains has spurred innovation in NILM as evidenced by recent deep learning methods [13, 16, 21]. Most of the methods described above work well for sampling rates of 1Hz to 1/600 Hz (a reading every 10 minutes). However, there is also a significant amount of work [6, 7, 17] in the NILM literature leveraging power data collected at much higher frequencies.

Despite this recent interest in NILM, it remained virtually impossible to empirically compare NILM research and *truly* understand the state of the art. Three factors primarily affected reproducibility and prevented empirical comparison of NILM algorithms. First, it was hard to assess the generality of NILM approaches as most work was evaluated on a single data set, and furthermore, researchers would often sub-sample data sets to select specific households, appliances and time periods, making experimental results more difficult to reproduce. Second, there was a lack of comparison using the same benchmarks, which was primarily due to the lack of availability of open source benchmark implementations. Thus, most research papers compared their work against variants of their main algorithms or trivial baseline algorithms. Third, a large number of metrics were used based on the use case under consideration, making it practically impossible to determine the state-of-the-art.

2.3 NILMTK

The open source non-intrusive load monitoring toolkit (NILMTK) [3] was released in early 2014 to enable easy comparison of NILM algorithms in a reproducible fashion. The main contributions of the toolkit were: i) NILMTK-DF (data format): the standard energy disaggregation data structure used by NILMTK; ii) parsers for six existing data sets (REDD, SMART*, AMPds, UK-DALE, iAWE and

PecanStreet data set); iii) implementations of two benchmark NILM algorithms (combinatorial optimization and exact factorial hidden Markov model); iv) statistical and diagnostic functions for understanding data sets; v) a suite of accuracy metrics across a range of use cases. Later in 2014, motivated by the release of large data sets such as Dataport data set [18], NILMTK v0.2 was released [12], which was a major rewrite of the toolkit specifically designed to support out-of-core computation.

Since its release, NILMTK has become the energy disaggregation field's reference library for data set parsers and reference benchmark algorithm implementations. The GitHub repository[2] has been starred by 385 people, forked 243 times, is cloned roughly 20 times a day and has an average of 600 page views every day from roughly 100 unique users. The toolkit was also awarded the best demonstration award at ACM BuildSys 2014.

However, few publications presenting algorithmic contributions within the field of energy disaggregation subsequently contributed implementations back to NILMTK. This is likely due to the requirement for algorithm authors to understand internal concepts of the toolkit, such as the `ElecMeter` and `MeterGroup` objects and their associated methods. Furthermore, the disaggregator interface itself has become overly complex with the support for out-of-core disaggregation. As a result of the lack of availability of such algorithm implementations, it still remains a major challenge to reproduce the results of recent contributions in the field and determine which algorithms represent the state of the art in different scenarios.

More recently, Kelly et al. conducted a survey[3] in an effort to design a competition for energy disaggregation algorithms. The insights gathered were useful not only in designing such a contest, but also analysing the current needs of the community. The survey responses were submitted by a diverse range of NILM enthusiasts. While the community has shown a lot of interest in a NILM competition, the learning curve was too high to integrate new algorithms into NILMTK, further supporting the above assertions. Clearly, significant improvements to the toolkit are required to support and measure progress within the NILM research field.

3 IMPROVEMENTS TO NILMTK

In this section we first describe two core changes made to NILMTK; namely a new experiment interface and a rewrite of the model interface. We then go on to summarise the user-raised issues that have been addressed since the release of NILMTK v0.2, and also describe a number of practical improvements to the toolkit's installation process, data set parsers and documentation.

3.1 Experiment Interface

We have introduced `ExperimentAPI`; a new NILMTK interface which reduces the barrier-to-entry for specifying experiments for NILM research. This allows NILMTK users to focus on which experiments to run rather than on the code required to run such experiments. Our new interface is inspired by declarative visualisation library `Vega-Lite`.[4] Declarative syntax decouples *what we want to do* from *how we do it* (the latter being imperative style).

[2]https://github.com/nilmtk/nilmtk/

[3]http://jack-kelly.com/a_competition_for_energy_disaggregation_algorithms

[4]https://vega.github.io/vega-lite/

```
1  experiment = {
2    'power': {'mains': ['active'], 'appliance': ['active']},
3    'sample_rate': 60, 'artificial_aggregate':False,
4    'appliances': ['fridge', 'washing machine'],
5    'methods': {'CO': {},
6      'FHMM_EXACT': {'num_of_states': 2},
7      'Seq2Point': {
8        'n_epochs': 1,
9        'pre-processing': {
10         'appliance_params': {
11           'washing machine': {
12             'mean': 400,'std': 700},
13           'fridge': {
14             'mean': 200,'std': 400 },
15         }
16       },
17     }
18     },
19   'train': {
20     'datasets': {
21       'REDD': {
22         'path': '/data/REDD/redd.h5',
23         'buildings': {
24           1: {'start_time': '2011-04-01','end_time': '2011-04-30'},
25           2: {'start_time': '2011-04-01','end_time': '2011-04-30'}
26         }
27       }
28     }
29   },
30   'test': {
31     'datasets': {
32       'IAWE': {
33         'path': '/data/IAWE/iawe.h5',
34         'buildings': {
35           1: {'start_time': '2015-08-05','end_time': '2015-08-10'}
36         }
37       }
38     },
39     'metrics': ['mae', 'rmse']
40   }
41 }
```

Listing 1: `ExperimentAPI`: Simplifying the definition of algorithm comparison experiments

Listing 1 shows the new experiment interface implemented in NILMTK. This interface aims to encapsulate the parameters required for training and testing over data sets using NILMTK. Previously, these parameters were spread across multiple NILMTK modules and toolkit users had to be considerate about providing the right training and testing parameters during the respective function calls. The listing describes an experiment where:

- mains and appliances use active power (L2).
- with a sampling rate of 60 seconds and not using artificial aggregate, i.e. using true aggregate reading (L3).
- for appliances: fridge and washing machine (L4).
- three algorithms are used for disaggregation (CO, FHMM and Seq2Point – L5, 6, 7 respectively) and their corresponding parameters are specified (L6 for FHMM and L8-14 for Seq2Point).
- training parameters are specified on L19-29, where the different training data sets are specified: REDD data set specified from L21-26, where the path for the data set is specified in L22 and the start and end time for building number 1 and 2 are specified in L24 and 25.
- the test parameters are added in a similar format to the training parameters from L31-38.
- the set of evaluation metrics are on L39.

The new API drastically reduces the workload for the toolkit user. As an example, for the experiment described in Listing 1, the previous version of NILMTK (which was imperative in design) required the users to iterate over chunks of data across different buildings and across different datasets, then combine the predictions across these individual chunks and pass them through the interface for metrics. Not only was the previous interface more time consuming; but, also more error prone. Besides, the end-user had to gain sufficient familiarity with NILMTK specific data structures

```
1  class Disaggregator(object):
2      def train(self, metergroup):
3          """Parameters
4          ----------
5          metergroup : a nilmtk.MeterGroup object
6          """
7          raise NotImplementedError()
8
9      def train_on_chunk(self, chunk, meter):
10         """Parameters
11         ----------
12         chunk : pd.DataFrame where each column represents a
13             disaggregated appliance
14         meter : ElecMeter for this chunk
15         """
16         raise NotImplementedError()
17
18     def disaggregate(self, mains, output_datastore):
19         """Parameters
20         ----------
21         mains : nilmtk.ElecMeter (single-phase) or
22             nilmtk.MeterGroup (multi-phase)
23         output_datastore : instance of nilmtk.DataStore or str of
24             datastore location
25         """
26         raise NotImplementedError()
27
28     def disaggregate_chunk(self, mains):
29         """Parameters
30         ----------
31         mains : pd.DataFrame
32
33         Returns
34         -------
35         appliances : pd.DataFrame where each column represents a
36             disaggregated appliance
37         """
38         raise NotImplementedError()
39
40
41     def _save_metadata_for_disaggregation(self, output_datastore,
42                                           sample_period, measurement,
43                                           timeframes, building,
44                                           meters=None, num_meters=None,
45                                           supervised=True):
46         """Parameters
47         ----------
48         output_datastore : nilmtk.DataStore subclass object
49             The datastore to write metadata into.
50         sample_period : int
51             The sample period, in seconds, used for both the
52             mains and the disaggregated appliance estimates.
53         measurement : 2-tuple of strings
54             In the form (<physical_quantity>, <type>) e.g.
55             ("power", "active")
56         timeframes : list of nilmtk.TimeFrames or nilmtk.TimeFrameGroup
57             The TimeFrames over which this data is valid for.
58         building : int
59             The building instance number (starting from 1)
60         supervised : bool, defaults to True
61             Is this a supervised NILM algorithm?
62         meters : list of nilmtk.ElecMeters, optional
63             Required if `supervised=True`
64         num_meters : int
65             Required if `supervised=False`
66         """
```

Listing 2: The old `Model Interface` in NILMTK required interfacing with NILMTK intricacies

It is important to note that handing over so much flexibility to the user does require the user to be somewhat familiar with the data set, but this part of the process is supported by NILMTK as data exploration is simple and well documented. We make extensive use of the `ExperimentAPI` for our empirical evaluations in Section 5.

3.2 Model Interface

Previously, the disaggregator class (Listing 2) required intricate knowledge of NILMTK objects such as `ElecMeter` and `MeterGroup` as used in functions at L2, L9, L18, L28, L41. This dependence on knowledge of NILMTK's internal implementation proved to be a barrier for community algorithm authors.

The new `Model Interface` (Listing 3) eliminates the dependence on NILMTK objects. The class definition has been simplified in terms of input and output formats and is consistent throughout the new API, but also all of the new functions in L2, L12 and L20

```
1  class Disaggregator(object):
2    def partial_fit(self, train_mains,
3        train_appliances, **load_kwargs):
4      """
5      Parameters
6      ----------
7      train_main: list of pd.DataFrames with pd.DatetimeIndex as index and 1 or
         ↪ more power columns
8      train_appliances: list of (appliance_name, list of pd.DataFrames) with the
         ↪ same pd.DatetimeIndex as index as train_main and the same 1 or more
         ↪ power columns as train_main
9      """
10     raise NotImplementedError()
11
12   def disaggregate_chunk(self, test_mains):
13     """
14     Parameters
15     ----------
16     test_mains : list of pd.DataFrames
17     """
18     raise NotImplementedError()
19
20   def call_preprocessing(self, train_mains,
21       train_appliances):
22     """
23     Parameters
24     ----------
25     train_main: list of pd.DataFrames with pd.DatetimeIndex as index and 1 or
         ↪ more power columns
26     train_appliances: list of (appliance_name, list of pd.DataFrames) with the
         ↪ same pd.DatetimeIndex as index as train_main and the same 1 or more
         ↪ power columns as train_mains
27     """
28     return train_mains, train_appliances
```

Listing 3: The new `Model Interface` in NILMTK requires only the Python data science ecosystem

are independent of NILMTK objects. The algorithm developer only needs to know the PyData stack (`pandas, numpy, scikit-learn`) to be able to write new algorithms for NILMTK. The new interface also decouples the training and loading of data, and the user only has to handle a chunk of data loaded by the API according to the parameters specified by the user. Only two new functions are mandatory for implementation:

(1) `partial_fit`: The training method that is called repeatedly over new chunks of data and keeps on fitting the new data and improving the model.
(2) `disaggregate_chunk`: The disaggregation method that receives aggregate test data which it disaggregates into predicted constituent appliances and returns the predictions.

The `call_preprocessing` method is optional and is specific to algorithms requiring preprocessing. It was added to address the time consuming preprocessing required by neural network models. It allows users to store their preprocessed data in an HDF5-formatted file. Users do not need to repeatedly preprocess the same data, since the data can be loaded from the saved file.

We now explain the design of the input data structures in the newly introduced `Model Interface`. These were carefully designed after extensive discussions with community members to meet the requirements of benchmark and modern NILM algorithms. We define the following two terms to explain the data structure:
Chunk: A contiguous portion of time-series indexed power data that fits in memory. Users can specify the chunk size depending on their system memory.
Window: A portion of a chunk on which the model trains in one iteration. The concept of a window is akin to a sequence in which various features are extracted in traditional machine learning algorithms (such as mean, median). The windows could be rolling (with or without overlap) or be completely dis-contiguous. Each window of data is a DataFrame (like the chunk it is derived from) which contains a timeseries index and various power features as columns.

```
 1 class Mean(Disaggregator):
 2     def __init__(self, model_parameters):
 3         self.model = {}
 4         self.MODEL_NAME = 'Mean'
 5         self.save_model_path = model_parameters.get('save-model-path',None)
 6         self.load_model_path = model_parameters.get('pretrained-model-path',None
       ↪ )
 7         self.chunk_wise_training = model_parameters.get('chunk_wise_training',
       ↪ True)
 8         if self.load_model_path:
 9             self.load_model(self.load_model_path)
10
11     def partial_fit(self, train_main, train_appliances, **load_kwargs):
12         train_main = pd.concat(train_main,axis=0)
13         for appliance_name, power in train_appliances:
14             power_ = pd.concat(power,axis=0)
15             appliance_dict = self.model.get(appliance_name,{'sum':0,'n_elem':0})
16             appliance_dict['sum']+=int(np.nansum(power_.values))
17             appliance_dict['n_elem']+=len(power_[~np.isnan(power_)])
18             self.model[appliance_name] = appliance_dict
19         if self.save_model_path:
20             self.save_model(self.save_model_path)
21
22     def disaggregate_chunk(self, test_mains):
23         test_predictions_list = []
24         for test_df in test_mains:
25             appliance_powers = pd.DataFrame()
26             for i, appliance_name in enumerate(self.model):
27                 model = self.model[appliance_name]
28                 predicted_power = [model['sum']/model['n_elem'] for j in range
       ↪ (0, test_df.shape[0])]
29                 appliance_powers[appliance_name] = pd.Series(predicted_power,
       ↪ index=test_df.index, name=i)
30             test_predictions_list.append(appliance_powers)
31         return test_predictions_list
32
33     def save_model(self,folder_name):
34         string_to_save = json.dumps(self.model)
35         os.makedirs(folder_name, exist_ok=True)
36         with open(os.path.join(folder_name,"model.txt","w")) as f:
37             f.write(string_to_save)
38
39     def load_model(self,folder_name):
40         with open(os.path.join(folder_name,"model.txt","r")) as f:
41             model_string = f.read().strip()
42             self.model = json.loads(model_string)
```

Listing 4: `Model Interface` definition for the *Mean* baseline in NILMTK

The input to the `Model Interface` is a list of mains `windows` and a list of appliance `windows`. Traditional algorithms like FHMM would have a single window (equal to the chunk length), whereas newer neural algorithms like Seq2Point would consider overlapping windows (with a time difference of 1 sample) of much smaller length than the chunk size.

We now illustrate the ease of writing a new algorithm using the new `Model Interface` in Listing 4. We discuss what we refer to as the *mean algorithm* (refer to Section 4.1 for more details about this algorithm). The main idea of the algorithm is that for each appliance we estimate its mean usage from the training set and predict the same mean usage for the test set, irrespective of the observed aggregate. We now discuss the listing:

- L5-L7: show the parameters that can be specified as input to the algorithm: where to save or load the model from.
- L13-L18: update the appliance-wise sum and total length observed in an online fashion.
- L26-30: take a mains dataframe as input and outputs an appliance-wise usage dataframe based on stored mean.
- L33-37 (optional): saves the model after the training process.
- L39-42 (optional): load a pre-trained model.

As it can be seen from the listing, writing a new algorithm in NILMTK now does not require knowledge of NILMTK internals.

3.2.1 Preprocessing. This release supports data preprocessing by providing 3 important features in the experiment API and the model interface: (a) allowing users to specify the preprocessing parameters of the algorithm (Listing 1, L9-L16), (b) providing a common data structure to handle changes made to the dataframe shapes during preprocessing, which are consistent throughout the methods in Listing 3, and (c) allowing the user to store and load the preprocessed data for individual methods (Listing 1, L5). All of these additions to the interface collectively allow users to include preprocessing methods such as mean centering, standardisation, min-max scaling, etc. The `call_preprocessing` method (Listing 3, L20) acts as an interface for handling the preprocessed dataframes and function calls to the individual preprocessing methods inside each algorithm.

3.3 Practical Improvements to NILMTK

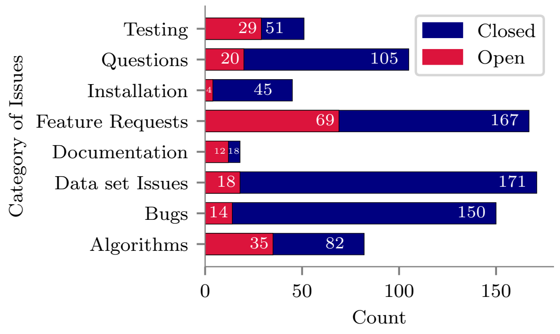

Figure 1: Types of issues on NILMTK's GitHub repository

The NILMTK GitHub issue queue totals at 727 issues, whose distribution is shown in Figure 1. The issues have been categorised as shown with some issues appearing in multiple categories. Most of the issues have been fixed and the documentation has improved significantly. Addressing these issues and improving NILMTK involved work from 21 contributors. 1700+ commits and 72 pull requests later, 617 issues have been closed as of now. We now discuss our efforts towards some of the issue categories.

3.3.1 Easier installation: Installation issues have been popping up at regular intervals ever since NILMTK gained widespread usage in the community. A total of 49 issues (Figure 1) were directly related to installation. The installation procedure required the user and developers to clone the `nilmtk` and `nilm_metadata` repositories, setup the environment using the `YAML` file provided and then run the `setup.py` files manually, in a specific sequence, for the process to complete. The process varied across platforms, and the installation on Windows was particularly problematic due to the dependencies such as `psycopg` (support for `PostgreSQL`) being available only in source code format at the time, requiring other tools such as `Microsoft Build Tools` (C/C++ compiler tools). This was coupled with dependencies on specific versions of `pandas` and `scikit-learn`, and newer versions of such libraries were often incompatible, leading to a growing number of installation conflicts as the dependencies evolved. This was a convoluted process, and understandably caused problems for the average new user.

Such issues motivated a permanent solution to the NILMTK installation process. With the advent of the Anaconda Python distribution, especially the `conda` package manager and `conda-forge` community repository of packages, NILMTK's dependencies are now available in an accessible manner. This enables the entire toolkit to be built into a package that is hosted on Anaconda cloud, that reduces the complex installation process to a single command:

```
conda install -c nilmtk -c conda-forge nilmtk
```

3.3.2 New data converters and improvements: This category includes datastore, format conversion and pre-processing issues. Most NILMTK code revolves around processing HDF5 files as all data sets are first converted into a consistent HDF5 file format.
1. Data set converters: Major issues regarding existing data sets have been addressed and resolved. We have also added converters for the Dutch residential energy data set (DRED) [20] and the Smart* data set [1]. These converters currently allow NILMTK to support 12 different data sets.
2. Datastore issues: A host of issues were related errors in format conversion, data set loaders, metadata representation and HDF datastores. Most of the bugs have been fixed.

3.3.3 Documentation Changes: The NILMTK documentation has gone through the following incremental changes and major overhauls motivated by the community issues:
1. Installation Documentation: The new installation documentation now guides the user on how to create, activate and deactivate conda environments and documents the single line commands to add conda-forge and install NILMTK.
2. Meter Selection and Basic Statistics: We have added a host of data exploration plots such as Sankey plot, autocorrelation plot, etc. We have also written documentation for various methods of `MeterGroup` and `ElecMeter` object selection, etc.
3. Algorithm Updates and Testing: The disaggregation algorithms CO, FHMM and Edge detection have been refined to handle most errors and use cases. Documentation has been added in the repository's manual to reflect these changes.

4 NILMTK-CONTRIB

This section describes the algorithms contained within the NILMTK-contrib repository. We have chosen to house bleeding-edge algorithms in a separate repository to the core toolkit to encourage algorithm publishers to own the implementation of their algorithm. We expect algorithms to eventually move into the main NILMTK repository as they gain maturity. Each of the algorithms described in this section have been implemented in accordance with the new disaggregator class described earlier. We have included simpler benchmarks in the NILMTK-contrib repository (and described them here) during the transition to the new Model Interface.

4.1 Mean

The Mean algorithm is a simple benchmark designed to provide a well-understood baseline against which more complex algorithms can be compared. Furthermore, the Mean algorithm can be used in the documentation to provide a sample algorithm implementation which algorithm contributors can use as a guide. The trained mean model calculates and stores only the mean power state for each appliance. The mean for each appliance is dynamically updated each time the same appliance is encountered (in each chunk). Prediction is equally simple - for each value of aggregate reading, the mean model predicts all appliances to be ON and returns the mean power value for all appliances. As such, it is similar to the commonly used "always on" benchmark algorithm. Despite its simple nature, the Mean algorithm is a solid baseline and performs comparably to complex disaggregation algorithms on a number of metrics, and has been used as a baseline in prior work [14].

4.2 Edge Detection

Proposed by George Hart in 1985 [8], this algorithm is often used as a baseline model for the NILM problem. The technique is based on edge detection within the power signal, which divides the time series into steady and transient time periods. An edge is defined as the magnitude difference between two steady states and typically corresponds to an appliance changing state (e.g. turning on or off). Although this algorithm is theoretically unsupervised as it does not require appliance-level data for training, this means that the algorithm output (e.g. appliance 1, 2 etc.) need to be mapped to appliance categories (e.g. refrigerator, washing machine etc.). In our implementation, we use the best case mapping, in which algorithm outputs are assigned to the appliance categories which, maximise the algorithm's accuracy.

4.3 Combinatorial Optimisation (CO)

The combinatorial optimisation (CO) algorithm [9] has served as a baseline algorithm in the NILM literature [4, 20]. The CO algorithm is similar to the well-studied knapsack and subset sum problem. The main assumption in CO is that each appliance can be in a given state (1 of K where K is a small number), where each state has an associated power consumption. The goal of the algorithm is to assign states to appliances in a way that the difference between the household aggregate reading and the sum of power usage of the different appliances is minimised. CO's time complexity is exponential in the number of appliances and thus does not scale well.

4.4 Discriminative Sparse Coding (DSC)

Sparse coding approximates the original energy matrix by representing it as a product of over-complete bases and their activations [14]. For each appliance i, we approximate the data matrix as $X_i \approx B_i A_i^*$, where X_i corresponds to usage, B_i corresponds to bases and A_i^* corresponds to activations for the i^{th} appliance. The activations A_i^* are the activations that minimise the reconstruction error for the i^{th} appliance. Hence, they are the optimal activations, that we wish to obtain when we disaggregate on the mains reading.

Disaggregation is performed by following the sparse coding approach on the observed mains reading by concatenating the bases of all appliances to obtain the activations ($\hat{A}$). The appliance-wise activations are extracted and are multiplied with the reconstruction bases to obtain the disaggregated usage. Discriminative Sparse Coding is an approach which modifies the sparse coding bases to produce activations that are closer to the optimal solution. The extracted activations are then multiplied with reconstruction bases to produce the disaggregated usage.

4.5 Exact Factorial Hidden Markov Model (ExactFHMM)

The factorial hidden Markov model (FHMM) is a natural model to represent the NILM problem [15, 19, 22]. In an FHMM, each appliance is represented by a hidden Markov model with K_i states so that the component signal X_i has a finite set of states $\mu_i = (\mu_{i1}, \cdots, \mu_{iK_i})$. Thus, $x_{it} \approx \widetilde{\mu}_{it}$ where $\widetilde{\mu}_{it} \in \mu_i$ for the i^{th} appliance at time t. We use a binary vector $S_{it} = (S_{it1}, S_{it2}, \cdots, S_{itK_i})^T$ to represent the state of the i^{th} appliance at time t such that $S_{itk} = 1$ when the appliance is at state k and for all $j \neq k$, $S_{itj} = 0$. The parameters of FHMM are learned using the training data. These parameters are μ_{ik} representing the state mean values for appliance i at state k; the initial probabilities $\pi_i = (\pi_{i1}, \cdots, \pi_{iK_i})^T$ where $\pi_{ik} = P(S_{i1k} = 1)$; and the transition probabilities $p^{(i)}_{jk} = P(S_{itj} = 1 | S_{i,t-1,k} = 1)$. The posterior distribution of the state variables is: $P(S|Y) \propto \prod_{i=1}^{I} P(S_{i1}) \prod_{t=2}^{T} \prod_{i=1}^{I} P(S_{it}|S_{i,t-1}) \prod_{t=1}^{T} p(Y_t|S_t)$. Given the model parameters denoted by θ, our aim is to infer the sequence over time of hidden states S_i for each appliance. In the *exact* formulation, we create a *super* HMM combining the individual HMMs in a Kronecker product fashion. For instance, if we had two appliances with OFF and ON states, our new *super* HMM would have four states (appliance 1 OFF, appliance 2 OFF; appliance 1 OFF, appliance 2 ON; appliance 1 ON, appliance 2 OFF, and appliance 1 ON, appliance 2 ON). As with CO, the *exact* model scales poorly as it is exponential in the number of appliances.

4.6 Approximate Factorial Hidden Markov Model (ApproxFHMM)

Inference of exact solutions in an FHMM is expensive and often becomes stuck in a local optimum. The approximate FHMM aims to alleviate these issues by relaxing the state values into $[0, 1]$ and transforming the FHMM inference problem to a convex program [22]. To achieve this purpose, we define a $K_i \times K_i$ variable matrix $H^{it} = (h^{it}_{jk})$ such that $h^{it}_{jk} = 1$ when $S_{i,t-1,k} = 1$ and $S_{itj} = 1$, and otherwise $h^{it}_{jk} = 0$. Therefore $S_{itk} \in [0, 1]$ and $h^{it}_{jk} \in [0, 1]$. The objective function becomes $\mathcal{L}(S, H, \sigma^{-2}) = -\sum_{i,t,k,j} h^{it}_{jk} \log p^{(i)}_{jk} - \sum_{i=1}^{I} S^T_{i1} \log \pi_i + \frac{1}{2} \sum_{t=1}^{T} \log \sigma^2_t + \frac{1}{2} \sum_{t=1}^{T} \frac{1}{\sigma^2_t} \left(Y_t - \sum_{i=1}^{I} S^T_{it} \mu_i \right)^2$. Denote the constraints $Q_S = \{\sum_{k=1}^{K_i} S_{itk} = 1, 0 \leq S_{itk} \leq 1, \forall i, t\}$, $Q_H = \{\sum_{l=1}^{K_i} H^{it}_{l.} = S^T_{i,t-1}, \sum_{l=1}^{K_i} H^{it}_{.l} = S^T_{i,t}, 0 \leq h^{it}_{jk} \leq 1, \forall i, t\}$. We solve the following convex quadratic program (CQP):

$$\min_{S,H,\sigma^2} \mathcal{L}(S, H, \sigma^2), \text{ st } Q_S \cup Q_H$$

4.7 FHMM with Signal Aggregate Constraints (FHMM+SAC)

The FHMM with signal aggregate constraints is an extension to the baseline FHMM, where the aggregate value of each appliance i over a time period is expected to be a certain value μ_{i0}. Under this assumption, the SAC expects $\sum_{t=1}^{T} x_{it} = \mu_{i0}$. Combining FHMM with SAC results in the following optimisation problem:

$$\min_S - \log P(S|Y), \qquad \text{subject to} \qquad \left(\sum_{t=1}^{T} \mu_i^T S_{it} - \mu_{i0} \right)^2 \leq \epsilon_i, \forall i,$$

which can be transformed to a relaxed convex program [22].

4.8 Denoising Autoencoder (DAE)

The denoising autoencoder is a specific deep neural network architecture designed to extract a particular component from noisy input. Well-known applications of a DAE include removing grain from images and reverb from speech signals. In a similar way, Kelly et al. proposed using the DAE for NILM by considering the mains signal to be a noisy representation of the appliance power signal [13]. As such, the mains reading Yt is assumed to be the sum of the power consumption of the target appliance x_{jt} and noise η_t.

Multiple trained models are required in order to disaggregate a group of appliances using a DAE since it denoises on a per-appliance basis. Moreover, the DAE receives a window of the mains readings of fixed length and outputs the inferred appliance consumption for the same time window. The length of the input vector can be tuned for each appliance to maximise performance. The architecture proposed by Kelly et al. is the following:

(1) Input with length optimised to the appliance
(2) 1D Convolution: {filters: 8, kernel size: 4, activation: linear}
(3) Fully connected: {size:input length × 8, activation: ReLU}
(4) Fully connected: {size:128, activation: ReLU}
(5) Fully connected: {size:input length × 8, activation: ReLU}
(6) 1D Convolution: {filters: 1, kernel size: 4, activation: linear}

4.9 Recurrent Neural Network (RNN)

Recurrent Neural Networks are a specific type of neural network that allows for connections between neurons of the same layer. This makes RNNs well suited for sequential data, much like the readings of power consumption in NILM. Motivated by this, Kelly et al. proposed an RNN that receives a sequence of mains readings and outputs a single value of power consumption of the target appliance [13]. To overcome the vanishing gradient problem, the network utilises long short-term memory (LSTM) units which are special neurons designed to store values in their built-in memory cell. The proposed architecture in detail is the following:

(1) Input with length optimised to the appliance
(2) 1D Convolution: {filters: 16, kernel size: 4, activation: linear}
(3) Bidirectional LSTM: {number of units: 128, activation: tanh}
(4) Bidirectional LSTM: {number of units: 256, activation: tanh}
(5) Fully connected: {size:128, activation: tanh}
(6) Fully connected: {size:1, activation: linear}

4.10 Sequence-to-Sequence (Seq2seq)

The sequence to sequence learning model [21] learns a regression map from the mains sequence to the corresponding target appliance sequence. We denote the mains and the target appliance sequences as $Y_{t:t+W-1}$ and $X_{t:t+W-1}$ respectively. The seq2seq model is then defined by the regression $x_{t:t+W-1} = f(Y_{t:t+W-1}, \theta) + \epsilon_t$ where the W-dimensional Gaussian noise variable $\epsilon_t \sim \mathcal{N}(0, \sigma^2_t I)$. Again f is a neural network. For learning f, we adopt the architecture proposed in [21] using stride 1 convolutions and ReLU activations for all layers except the final one. The other hyperparameters are:

(1) Input sequence with length W: $Y_{t:t+W-1}$
(2) 1D Convolution: {# filters: 30; filter size: 10}
(3) 1D Convolution: {# filters: 30; filter size: 8}
(4) 1D Convolution: {# filters: 40; filter size: 6}
(5) 1D Convolution: {# filters: 50; filter size: 5}

(6) 1D Convolution: {# filters: 50; filter size: 5}
(7) Fully connected: {# units: 1024}
(8) Output: {Number of units:W}

4.11 Sequence-to-Point (Seq2point)

Following the work in [21], sequence to point learning (seq2point) models the input of the network as a mains window $Y_{t:t+W-1}$, and the output as the midpoint element $x_{\tau(t,W)}$ of the corresponding window of the target appliance, where $\tau(t, W) = t + \lfloor W/2 \rfloor$. The intuition behind this method is that the midpoint of the target appliance should have a strong correlation with the mains information before and after that time point. Seq2point learning could be viewed as a non-linear regression $x_{\tau(t,W)} = f(Y_{t:t+W-1}, \theta) + \epsilon_t$, where θ are the parameters, for any input sequence $Y_{t:t+W-1}$ and output point $x_{\tau(t,W)}$. The function f is represented by a neural network. For learning f, we adopt the architecture proposed in [21] using stride 1 convolutions and ReLU activations for all layers except the final one. The netork is as follows:
(1) Input sequence with length W: $Y_{t:t+W-1}$
(2) 1D Convolution: {# filters: 30; filter size: 10}
(3) 1D Convolution: {# filters: 30; filter size: 8}
(4) 1D Convolution: {# filters: 40; filter size: 6}
(5) 1D Convolution: {# filters: 50; filter size: 5}
(6) 1D Convolution: {# filters: 50; filter size: 5}
(7) Fully connected: {# units: 1024}
(8) Output: {Number of units:1, activation: linear}

4.12 Online GRU

Based on the RNN of section 4.9, Krystalakos et al. proposed a similar architecture that attempts to reduce the computational demand while maintaining the same performance [16]. This version has replaced the LSTM units with light-weight Gated Recurrent Units (GRU) and optimised the recurrent layer sizes to reduce redundancy. As a result, this architecture manages to decrease the number of trainable parameters by 60%, relative to the original RNN model.

When deployed. the Online GRU model receives the last W available mains readings $Y_{t:t+W-1}$ as input and uses them to calculate the power consumption $x_{j(t+W-1)}$ of a single appliance j, for the last time point. The window size W can be optimised for each appliance individually. The architecture in detail is the following:
(1) Input with length optimised to the appliance
(2) 1D Convolution: {filters: 16, kernel size: 4, activation: linear}
(3) Bidirectional GRU: {# units: 64, activation: tanh, dropout: 0.5}
(4) Bidirectional GRU: {# units: 128, activation: tanh, dropout: 0.5}
(5) Fully connected: {size:128, activation: tanh, , dropout: 0.5}
(6) Fully connected: {size:1, activation: linear}

5 EXPERIMENTAL RESULTS

We now describe the settings used for our experiments before demonstrating usage of NILMTK across a range of application scenarios. The focus of this evaluation is to demonstrate the flexibility of NILMTK as a tool for algorithmic comparison across different scenarios. As such, an exhaustive discussion of each algorithm's performance is beyond the scope of this work. All our experiments are completely reproducible and have been pushed to Github[5]. In the interest of space, we will discuss the listing only for Section 5.3.

5.1 Settings

The tests were run on virtual machines with 2x8 GB vRAM Nvidia Tesla M60 GPU's with Intel(R) Xeon CPU (12 Cores @2.6 GHz) and 128 GB RAM. The sample period was 60 seconds. All neural algorithms were trained for 50 epochs with a batch-size of 1024, except for OnlineGRU which was trained for 30 epochs. All neural networks were also optimised by fine-tuning the algorithm parameters such as *sequence-length* and the appliance-wise parameters such as the *max-value*, etc. The optimal values for the parameters were provided by the algorithm authors.

5.2 Train and test across buildings from the same data set

In this experiment, we train and test across multiple buildings from the Dataport data set. We trained the models on 10 buildings and then tested them on 5 unseen buildings. The training duration was 14 days and the testing duration was 7 days, with the models training and disaggregating 4 appliances for each building. The main results for this experiment can be found in Table 1. The neural network models perform comparably, with Seq2Point and Seq2Seq achieving the best performance. Interestingly, the edge detection algorithm achieves good performance for fridges. This can be explained by the fact that for simple appliances with a single ON-OFF component (in this case, a compressor-controlled duty cycle), simple edge detection is likely to work well. This is an important finding, in that it appears that only more complex appliances motivate the use of complex disaggregation algorithms. The mean algorithm performs reasonably well for washing machine. This finding suggests that accurately disaggregating sparsely used appliances is still non-trivial for modern algorithms. However, a different metric that is better suited to handle class imbalance would reveal the inaccuracy of the Mean model.

Algorithms	Fridge	Air Conditioner	Electric Furnace	Washing Machine
Mean	63.3±07.7	224.8±16.4	81.5±01.6	5.07±00.8
Edge detection	41.1±18.1	86.8±30.5	30.2±11.2	4.8±01.3
CO	65.7±42.3	98.5±85.7	56.9±55.4	105±19.0
DSC	78.4±56.5	71.5±36.0	39.1±17.9	6.5±05.7
ExactFHMM	66.7±23.5	45.5±44.6	95.3±110.5	59.9±17.5
ApproxFHMM	63.8±08.0	139.9±130.2	**26.5±12.0**	30.7±21.3
FHMM+SAC	59.2 ±05.7	97.0±40.3	35.1±19.0	3.8±00.7
DAE	32.2±11.8	39.3±27.9	29.4±15.3	3.1±01.6
RNN	38.4±07.9	46.6±30.6	33.9±20.6	3.5±01.2
Seq2Seq	28.1±09.5	32.3±25.2	27.9±15.3	**2.3±01.2**
Seq2Point	**23.5±12.1**	**24.8±20.9**	27.5±15.0	2.4±00.9
OnlineGRU	28.8±11.4	25.3±17.1	34.5±15.0	3.0±01.4

Table 1: MAE Mean ± Std. Error: train/test on different set of buildings, same data set

[5]https://github.com/nilmtk/buildsys2019-paper-notebooks

```
1  d = {
2    'power': {
3      'mains': ['apparent', 'active'], 'appliance': ['apparent', 'active']
4    },
5    'sample_rate': 60,'artificial_aggregate':False,
6    'appliances': ['washing machine', 'fridge'],
7    'methods': {
8      'Mean': {}, 'CO': {},
9      'FHMM_EXACT': {
10       'num_of_states': 2
11     },
12     'AFHMM': {}, 'AFHMM_SAC': {}, 'Seq2Point': {}, 'Seq2Seq': {},'RNN': {},
13     'WindowGRU': {}, 'DAE': {}, 'DSC': {},
14   },
15
16   'train': {
17     'datasets': {
18       'UKDALE': {
19         'path': '/data/UKDALE/ukdale.h5',
20         'buildings': {1: {'start_time': '2017-01-05', 'end_time': '2017-03-05'},
21                 }
22       },
23     }
24   },
25   'test': {
26     'datasets': {
27       'DRED': {
28         'path': '/data/DRED/DRED.h5',
29         'buildings': {1: {'start_time': '2015-09-21', 'end_time': '2015-10-01'}
30         }
31       },
32       'REDD': {
33         'path': '/data/REDD/redd.h5',
34         'buildings': {1: {'start_time': '2011-04-17','end_time': '2011-04-27'}
35         }
36       }
37     },
38     'metrics': ['mae', 'rmse']
39   }
40 }
```

Listing 5: `Experiment Interface` for section 5.3

5.3 Train and test across multiple buildings from multiple data sets across data sets

In this experimental setup, we trained models on 2 appliances across a building from UK-DALE and then tested them on 1 building from DRED and 1 building from REDD. The total training duration was 2 months and the testing duration was 10 days for each building. Listing 5 corresponds to the declarative definition of this experiment, where the line numbers are referenced as follows:

- L5: sampling frequency is 60 seconds and disable artificial aggregate mode.
- L6: appliances are washing machine and fridge.
- L8-L14: specify the methods to be used for disaggregation.
- L16-L24: train on 2 months of data from building 1 from UKDALE.
- L25 - L37: use ten days of data from building 1 of DRED (L32-40) and building 1 of REDD for testing (L42-48).
- L38: specify the metrics to be reported.

It should be noted that an empty dictionary passed to an algorithm means that the algorithm is using default parameters for training. In previous versions of NILMTK, no documentation for executing such experiments existed owing to the non-trivial code required. In fact, in an attempt to create such functionality in previous version of NILMTK, the core contributors had to write a different function which would train across different buildings.

As can be seen from the result in Table 2, all the algorithms had noticeably poor performance on the test home for the REDD data set as compared to the DRED data set. This is likely due to the fact that the model was trained on a European Data set (UK), and thus the metrics were reasonable for the data set based in Europe, i.e. DRED (Netherlands) whereas the models perform worse for the REDD data set (USA), owing differences in appliances and usage behaviour between Europe and North America. The moderate performance of modern neural network based algorithms can be potentially attributed to their overfitting to the appliance patterns of the training geography. Further, the mean baseline performs very close to the best algorithm for washing machine usage for DRED, but does much worse for REDD. This can likely be explained by the difference in baseline appliance energy usages across Europe and USA. Experiments such as this open up the interesting possibilities of transfer learning in the energy disaggregation domain [2].

Algorithms	**REDD - Home 1**		**DRED - Home 1**	
	Fridge	Washing Machine	Fridge	Washing Machine
Mean	62.3	47.2	43.4	25.6
Edge detection	37.0	57.1	21.8	40.7
CO	99.3	171.1	45.9	47.2
DSC	61.5	48.9	34.3	12.1
ExactFHMM	95.9	179.6	32.3	19.1
ApproxFHMM	67.0	227.9	34.6	94.5
FHMM+SAC	48.1	30.4	31.1	19.0
DAE	41.7	50.1	**16.9**	3.8
RNN	50.9	**19.2**	27.8	7.3
Seq2Seq	40.9	23.3	18.5	**2.9**
Seq2Point	44.1	25.5	17.1	3.1
OnlineGRU	**36.4**	29.5	24.3	6.9

Table 2: MAE: train/test across multiple buildings and data sets

5.4 Train and test on artificial aggregate

In the this experiment, we train and test on 2 buildings from the Dataport data set, using true (obtained from smart meter) and artificial aggregate (calculated by summing the power readings of the appliances to be disaggregated). The artificial aggregate does not contain either structured noise from appliances which were not sub-metered or unstructured noise contributed by the mains sensor hardware. This scenario is often used for algorithm comparisons despite its lack of realism. This is due to the requirement for training data to be available for all appliances present in the aggregate signal, which is a common issue in data sets due to the practical difficultly in sub-metering all appliances within a building. The training was done on the first 20 days and the testing was done on the next 7 days for each building. The 2 most commonly used appliances were chosen for disaggregation.

Table 3 shows the main result where we can notice the superior disaggregation performance on the artificial aggregate for all algorithms. The only exception being the mean algorithm, that is independent of the aggregate data. The performance of neural networks on the air conditioner has improved significantly with artificial aggregate. However, the most significant improvement comes from the FHMM variants. This can be explained by the fact that FHMM variants have a state space that is exponential in the number of appliances and FHMM can explain the noise in true aggregate via a wrong appliance state space combination. In the absence of noise, the probability of estimating the correct state space combination is much more likely. This experiment represents the ideal scenario for energy disaggregation and might be prevalent

Algorithms	True Aggregate		Artificial Aggregate	
	Fridge	AC	Fridge	AC
Mean	43±04	176±40	43±04	176±40
Edge Detection	43±04	156±31	15±06	89±65
CO	104±08	90±34	18±03	15±07
DSC	67±05	130±29	53±14	55±20
ExactFHMM	56±10	83±16	14±01	24±03
ApproxFHMM	46±06	210±78	41±07	77±24
FHMM+SAC	32±04	132±47	36±07	125±28
DAE	22±05	34±08	14±01	8±02
RNN	31±01	78±24	11±01	10±03
Seq2Point	**14±02**	**20±06**	**5±01**	**4 ±01**
Seq2Seq	16±02	22±05	9±01	9±02
WindowGRU	20±04	23±09	8±02	7±03

Table 3: MAE: train/test across same buildings with true and artificial aggregate

in geographies where majority of the energy consumption can be attributed to a small set of appliances.

6 CONCLUSION

In this paper, we have have described two key improvements to NILMTK; a rewritten model interface to simplify authoring of new disaggregation algorithms, and a new experiment API through which algorithmic comparisons can be specified with relatively little model knowledge. In addition, we have introduced NILMTK-contrib, a new repository containing 3 benchmarks and 9 modern disaggregation algorithms. Furthermore, we have demonstrated these contributions through the most comprehensive algorithmic comparison to date. Taken together, these toolkit contributions enable empirical evaluations to be easily reproduced, therefore increasing the rate of progress within the field.

In the short-term, future work will focus on an exhaustive empirical evaluation of the algorithms presented in NILMTK-contrib across all publicly available data sets and a range of accuracy metrics. Longer-term future work will include collaboration with the community to ensure new algorithmic advances are incorporated within the NILMTK-contrib repository. In addition, such algorithms will be continuously evaluated in a range of pre-defined scenarios to produce an ongoing NILM competition.

7 ACKNOWLEDGEMENTS

We would like to sincerely thank Mario Berges for his detailed inputs during the shepherding phase. We would also like to thank the anonymous reviewers. Nipun Batra would like to thank NVidia for their GPU award.

REFERENCES

[1] Sean Barker, Aditya Mishra, David Irwin, Emmanuel Cecchet, Prashant Shenoy, and Jeannie Albrecht. [n. d.]. Smart*: An Open Data Set and Tools for Enabling Research in Sustainable Homes.

[2] Nipun Batra, Yiling Jia, Hongning Wang, and Kamin Whitehouse. 2018. Transferring decomposed tensors for scalable energy breakdown across regions. In *Thirty-Second AAAI Conference on Artificial Intelligence.*

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

[4] Nipun Batra, Amarjeet Singh, and Kamin Whitehouse. 2015. If you measure it, can you improve it? exploring the value of energy disaggregation. In *Proceedings of the 2nd ACM International Conference on Embedded Systems for Energy-Efficient Built Environments.* ACM, 191–200.

[5] Nipun Batra, Hongning Wang, Amarjeet Singh, and Kamin Whitehouse. 2017. Matrix factorisation for scalable energy breakdown. In *Thirty-First AAAI Conference on Artificial Intelligence.*

[6] Manoj Gulati, Shobha Sundar Ram, and Amarjeet Singh. 2014. An in depth study into using EMI signatures for appliance identification. In *Proceedings of the 1st ACM Conference on Embedded Systems for Energy-efficient Buildings.* ACM, 70–79.

[7] Sidhant Gupta, Matthew S Reynolds, and Shwetak N Patel. 2010. ElectriSense: single-point sensing using EMI for electrical event detection and classification in the home. In *Proceedings of the 12th ACM international conference on Ubiquitous computing.* ACM, 139–148.

[8] George W Hart. 1985. Prototype nonintrusive appliance load monitor. In *MIT Energy Laboratory Technical Report, and Electric Power Research Institute Technical Report.*

[9] G. W. Hart. 1992. Nonintrusive appliance load monitoring. *Proc. IEEE* 80, 12 (Dec. 1992), 1870–1891. https://doi.org/10.1109/5.192069

[10] K. He, L. Stankovic, J. Liao, and V. Stankovic. 2018. Non-Intrusive Load Disaggregation Using Graph Signal Processing. *IEEE Transactions on Smart Grid* 9, 3 (May 2018), 1739–1747. https://doi.org/10.1109/TSG.2016.2598872

[11] Yuanwei Jin, Eniye Tebekaemi, Mario Berges, and Lucio Soibelman. 2011. A time-frequency approach for event detection in non-intrusive load monitoring. In *Signal Processing, Sensor Fusion, and Target Recognition XX*, Vol. 8050. International Society for Optics and Photonics, 80501U.

[12] Jack Kelly, Nipun Batra, Oliver Parson, Haimonti Dutta, William Knottenbelt, Alex Rogers, Amarjeet Singh, and Mani Srivastava. 2014. Nilmtk v0. 2: a non-intrusive load monitoring toolkit for large scale data sets: demo abstract. In *Proceedings of the 1st ACM Conference on Embedded Systems for Energy-efficient Buildings.* ACM, 182–183.

[13] Jack Kelly and William Knottenbelt. 2015. Neural NILM: Deep Neural Networks Applied to Energy Disaggregation. In *Proceedings of the 2Nd ACM International Conference on Embedded Systems for Energy-Efficient Built Environments (BuildSys '15).* ACM, New York, NY, USA, 55–64. https://doi.org/10.1145/2821650.2821672 event-place: Seoul, South Korea.

[14] J. Z. Kolter, Siddharth Batra, and Andrew Y. Ng. 2010. Energy Disaggregation via Discriminative Sparse Coding. In *Advances in Neural Information Processing Systems 23*, J. D. Lafferty, C. K. I. Williams, J. Shawe-Taylor, R. S. Zemel, and A. Culotta (Eds.). Curran Associates, Inc., 1153–1161. http://papers.nips.cc/paper/4054-energy-disaggregation-via-discriminative-sparse-coding.pdf

[15] J Zico Kolter and Matthew J Johnson. 2011. REDD: A public data set for energy disaggregation research.

[16] Odysseas Krystalakos, Christoforos Nalmpantis, and Dimitris Vrakas. 2018. Sliding Window Approach for Online Energy Disaggregation Using Artificial Neural Networks. In *Proceedings of the 10th Hellenic Conference on Artificial Intelligence.*

[17] Henning Lange and Mario Bergés. 2016. Bolt: Energy disaggregation by online binary matrix factorization of current waveforms. In *Proceedings of the 3rd ACM International Conference on Systems for Energy-Efficient Built Environments.* ACM.

[18] Oliver Parson, Grant Fisher, April Hersey, Nipun Batra, Jack Kelly, Amarjeet Singh, William Knottenbelt, and Alex Rogers. 2015. Dataport and nilmtk: A building data set designed for non-intrusive load monitoring. In *2015 IEEE Global Conference on Signal and Information Processing (GlobalSIP).* IEEE, 210–214.

[19] Oliver Parson, Siddhartha Ghosh, Mark Weal, and Alex Rogers. 2012. Non-intrusive load monitoring using prior models of general appliance types. In *Twenty-Sixth AAAI Conference on Artificial Intelligence.*

[20] Akshay S.N. Uttama Nambi, Antonio Reyes Lua, and Venkatesha R. Prasad. 2015. LocED: Location-aware Energy Disaggregation Framework. In *Proceedings of the 2Nd ACM International Conference on Embedded Systems for Energy-Efficient Built Environments (BuildSys '15).* ACM, New York, NY, USA, 45–54. https://doi.org/10.1145/2821650.2821659

[21] Chaoyun Zhang, Mingjun Zhong, Zongzuo Wang, Nigel Goddard, and Charles Sutton. 2018. Sequence-to-Point Learning With Neural Networks for Non-Intrusive Load Monitoring. In *Thirty-Second AAAI Conference on Artificial Intelligence.* https://www.aaai.org/ocs/index.php/AAAI/AAAI18/paper/view/16623

[22] Mingjun Zhong, Nigel Goddard, and Charles Sutton. 2014. Signal Aggregate Constraints in Additive Factorial HMMs, with Application to Energy Disaggregation. In *Advances in Neural Information Processing Systems 27*, Z. Ghahramani, M. Welling, C. Cortes, N. D. Lawrence, and K. Q. Weinberger (Eds.). Curran Associates, Inc., 3590–3598.

[23] Mingjun Zhong, Nigel Goddard, and Charles Sutton. 2015. Latent Bayesian melding for integrating individual and population models. In *Advances in Neural Information Processing Systems 28*, C. Cortes, N. D. Lawrence, D. D. Lee, M. Sugiyama, and R. Garnett (Eds.). Curran Associates, Inc., 3618–3626.