Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Institut für
Technische Informatik und
Kommunikationsnetze

Using Random Forest for Indoor Solar Energy Estimates on Embedded Systems

Semester Thesis

Jannik William
williamj@ethz.ch

Computer Engineering and Networks Laboratory
Department of Information Technology and Electrical Engineering
ETH Zürich

Supervisors:
Naomi Stricker
Stefan Draskovic
Prof. Dr. Lothar Thiele

July 6, 2021

Acknowledgements

I would like to thank my supervisors Naomi Stricker and Stefan Draskovic. The weekly meetings with them were always very interesting and productive and they supported me wherever they could.

I also want to thank Prof. Lothar Thiele and the Computer Engineerung Group for providing me with the opportunity to work on this interesting topic.

Abstract

As IoT systems have gained a lot of interest in the last years, powering these devices have become a major hurdle. Energy harvesting is a solution to this problem but comes with its own difficulties. As the harvested energy is very scarce and big energy storages are costly and have an environmental impact, the energy usage have to be scheduled carefully with power availability in mind. It was suggested, that random forest is suitable for this task.

In this thesis, I implement an energy prediction algorithm based on random forest on a microcontroller and evaluate its performance. Further a technique is implemented to improve models on the device itself.

Contents

Chapter 1

Introduction

Over the last years, there was an ever increasing number of IoT and CPS systems. This trend poses new challenges as scalability, maintenance and environmental impact become relevant topics. For these reasons, we do not want to power these devices with batteries, as these contain environmentally harmful materials and have to be replaced periodically. Energy harvesting in conjuction with a small rechargable energy storage, is a good alternative. This has become feasible through technological advances, mainly because both IoT devices and solar panels are getting more and more efficient.

When we have a small energy storage, energy consumption becomes much more sensitive to the short-term variability in harvested energy. In this case, we need to carefully plan energy usage to achieve good system performance. For this, one can exploit energy predictions, so that we can schedule energy intensive tasks when energy is abundant.

Indoor solar power forecasting is more difficult compared to outdoors. The available light is a combination of natural and artificial light. The artificial part of the light is subject to high volatility, as it usually depends on human presence, while the natural light is also unpredictable due to many obstacles in the environment. In addition to that, the available energy is about 0.1% compared to outdoors. Therefore, we need more involved algorithms for high-accuracy predictions. In a previous work [1], different prediction models have been assessed. It was concluded that random forest yields the best prediction accuracy and robustness, with the RMSE being around 10% lower than with a linear regression model.
However, these prediction models have only been tested on a high power computer. To demonstrate they are useful, it is necessary to actually implement and evaluate the models on a low power microcontroller.

In this thesis, I implement an indoor energy prediction algorithm based on random forest on a microcontroller, and evaluate the predictions with respects to accuracy and resource requirements. More specifically, the main contribution of this thesis is the implementation of a script, that allows me to convert a model, that is trained with Scikit-learn [2] to C code, which can be executed on a microcontroller.

In an attempt to improve the model during operation, a pruning technique is implemented, where low accuracy trees are deleted from the model.

I extensively evaluate the execution time, flash requirements and accuracy of the different implementation approaches and of models with various sizes. To ease this process, I automated the model training, translation, deployment and evaluation.

In Chapter 2, I talk about the theoretical aspect of my work. Chapter 3 gives an overview over the different steps beginning with training and model translation and ending with the deployment. A deep-dive into the model training is found in Chapter 4. In Chapter 5, I explain different approaches to implement random forest on a microcontroller, which are then evaluated in Chapter 6.

Chapter 2

Theory

In this chapter I give a theoretical overview of the system (Section 2.1), a discussion on how random forest works under the hood (Section 2.2) and an assessment of the deployability of random forest on a microcontroller (Section 2.3).

2.1 System Model

The system model (Figure 2.1) describes the model training and evaluation of the random forest. Furthermore, it describes the origin of the features and how they get selected. It comprises of the following parts:

- Input variables
- Logging stage
- Prediction model
- Model evaluation
- Output variable

On an IoT device, ambient sensors measuring temperature, humidity, pressure and light are easily integrable. Additionally, meteo data can sent from a central station to the devices. So in this thesis, the input variables of the model consists of data from multiple indoor measurement station and meteo data from MERRA-2 (described in Section 4.2). Namely, these are the current solar power, illuminance of the solar panels, ambient temperature, ambient air pressure, ambient relative humidity, outdoor temperature, outdoor pressure and solar irradiance.
In the logging stage, we get several values from each signal stream with different lags (1h, 2h, 3h, 4h, 24h and 1 week).

The feature selection is next, where only the most significant variables get selected and then fed into the prediction model. The importance of the features are already calculated in [1], so I use these results to select them.
As the model training is done offline, it is not part of the system model. It is however important for the evaluation, that the model can be either trained on data coming from the same node as it is later deployed and evaluated or trained with different nodes than it is later evaluated. This leads to significantly different model performance.
The output of the model is the estimated power. To evaluate the prediction accuracy, the true value and the predicted value are compared with different metrics.

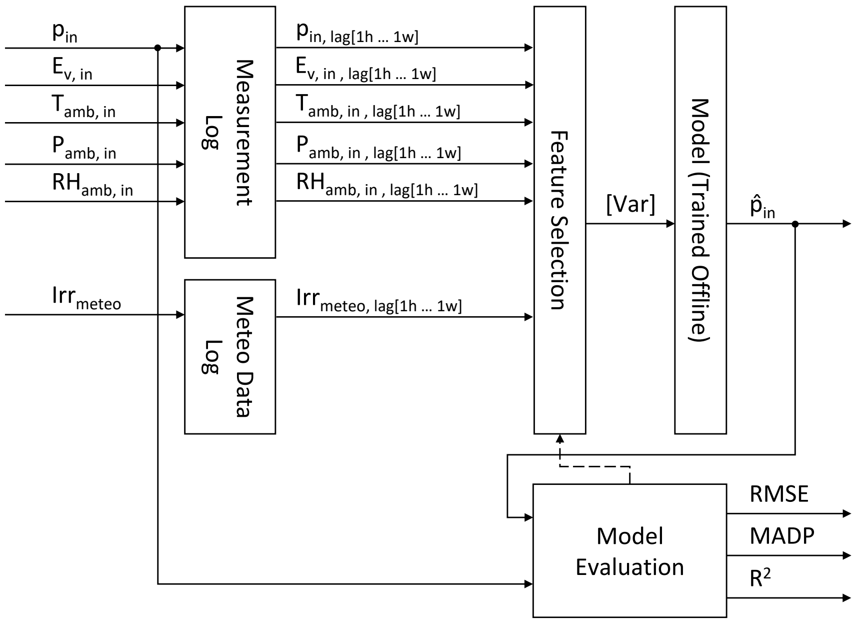

Figure 2.1: System Model: Prediction and Evaluation

2.2 Random Forest

Random forest [3] is an algorithm, that enables us to do regression. It uses desicion trees to make a prediction by averaging the output of each individual tree, resulting in a more robust and stable prediction than it would be with a single tree.

2.2.1 Decision Tree

A decision tree is a binary tree, which is traversed by a certain rule set. In Figure 2.2 you can see an example of such a tree. You start at the root node (with threshold c_1) and compare the feature (a value like temperature) with the threshold. Based on whether the feature is greater or smaller than the threshold, you take either the left child or the right one. This process is repeated, until you reach a leaf node. There, the output value (y_k) is held.
The depth of a tree d (2 in the example) is an important measurement of the complexity of the tree. In creating deeper trees, we are able to represent more complex real world behaviour in the model. The space required to store the tree is related to the depth by $\mathcal{O}(d^2)$.
To predict the solar power with multiple features, we have to expand the functionality of this simple decision tree by attaching a feature index to each threshold. This feature index maps each threshold to a feature allowing us to create trees using more than one feature.

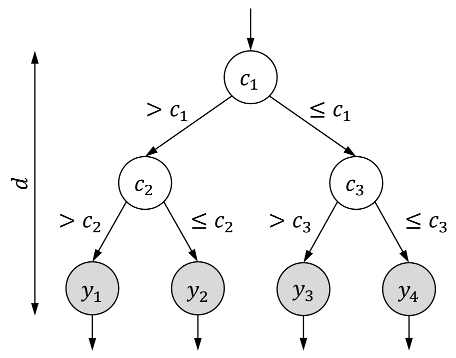

Figure 2.2: Simple Decision Tree with 1 Feature

2.2.2 Averaging Multiple Decision Trees

As we know, decision trees are prone to overfitting. By training multiple, independent trees and averaging their result, we are able to bypass this effect. In Figure 2.3, we can see a random forest with n trees. The input vector (features) $\vec{x}$ is fed into each tree, producing an output value $\tilde{y}_k$. The averaged value $\bar{\tilde{y}}$ is the predicted value of the random forest.

Independency is achieved by subsampling the dataset and randomly selecting a subset of features for each tree. For the sake of saving space, the number of trees and their depth is limited. To cut out unnecessary subtrees, the minimal number of samples per leaf is also specified.

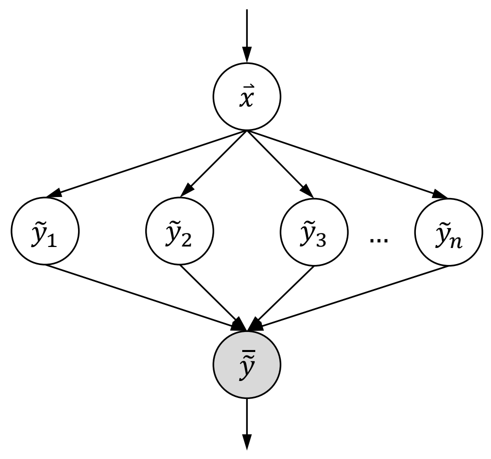

Figure 2.3: Random Forest Algorithm

2.2.3 Pruning

The idea behind pruning is that the model can be improved after training. This is done through cutting out under-performing trees from the forest.
As described in Section 2.1, the data for training the model can come from other nodes, than for evaluating it. This likely leads to trees, that perform significantly worse than others. Deleting them should lead to a better model overall.

2.3 Feasibility of Deployment

In [1] it was suggested that a forest size of 500, with a minimal number of samples per leaf of 5, and with no tuning for tree depth leads to optimal results, while improvements beyond this point are minimal. Without tuning the tree depth d however, it is impossible to predict the size of the model, as a larger dataset will result in a larger model. So we have to limit this parameter as well. In Chapter 6, I dive deeper into the selection of tree depth.

The required space for a random forest model on the microcontroller is dependent on the implementation (see Chapter 5). One can however calculate the minimal required storage, if one assumes, that each threshold and output value is stored as single precision float (4 bytes) and the feature index as short integer (2 bytes), that is stored for each node. For simplicity, we do not differentiate between normal nodes and leafs. To calculate the number of nodes, we have to know the tree depth d and the number of trees in the forest n. Assuming, that each tree is a full binary tree, we get following formula for the number of nodes per tree $n_{\text{nodes per tree}}$:

$$n_{\text{nodes per tree}} = 2^{d+1} - 1 \tag{2.1}$$

The number of total nodes $n_{\text{total nodes}}$ is calculated by the formula

$$n_{\text{total nodes}} = n \cdot n_{\text{nodes per tree}} \tag{2.2}$$

This leads us to a space requirement n_{bytes} in bytes

$$n_{\text{bytes}} = n_{\text{total nodes}}(s_{\text{th}} + s_{\text{idx}}) \tag{2.3}$$

whereas s_{th} is the size of the threshold/output value (4 bytes) and s_{idx} the size of the index value (2 bytes).

For example, we get 46'500 bytes for a random forest with 250 trees and a tree depth of 4. This however is the ideal case, as the size grows with additional required functions, debug information and struct padding.

Chapter 3

Framework

In this chapter I talk about the training, model translation and deployment process. The last section is about the communication protocol. Since this is is specifically implemented for the purpose of this work, the protocol is extensively described.

3.1 Model Training

The model training happens offline on a computer. For that, I use the machine learning library Scikit-learn [2] from Python. With the class `RandomForestRegressor`, one can implement a random forest regressor in a few lines of code. I parameterize the number of trained trees, the number of features used by each tree, the depth of the trees and the minimum number of samples required per tree. These parameters affect the accuracy and space requirements of the model.
The data (Section 4.2) used for training and testing the model is a time series of measurements. To preserve causality of the model, it is important that the training data is chronologically before the testing data.

3.2 Model Translation

The key contribution in this work is the model translation part. It converts the object oriented model from Scikit-learn and converts it to C code. Two different implementations exists. The code resides in `model_exporter.py`.
The main function is `export_to_file()`, which takes the trained `RandomForestRegressor` object, the implenentation and the output file as input. It saves the generted C code and its corresponding header files with the base algorithm and some helper functions, which are specific to the implementation at the specified location. How the different implementations work in detail is described in Section 5.

3.3 Deployment

The deployment process (compilation and flashing) is automated through Python. It uses the Arm Compiler with various optimization flags. This ensures, that minimal debug information is stored on the microcontroller. As a consequence, larger models can be stored on the internal flash.
Flashing is done with the tool Uniflash. It can be invoked through the command line with different options.

3.4 Microcontroller Communication

In order to test and evaluate the model, the microcontroller needs to communicate with the computer. That is done through a serial protocol. All parameters in the packets are LSB first.

3.4.1 Header

Each packet sent either from the computer or the microcontroller starts with a 3 byte header. This header contains a start byte (`0xA5`), a length byte indicating the length of the remaining packet (excluding the header) in bytes and a checksum byte (excluding the header). The checksum byte is calculated by simply adding up all the bytes in the packet and taking the least significant byte. An overview can be found in Table 3.1.

START	Start Byte	1 Byte
LEN	Length of Packet	1 Byte
CHKSM	Checksum of Packet	1 Byte

Table 3.1: Header

3.4.2 Request Packet

A request packet (Table 3.2), coming from the computer, starts with the header, followed by a command and subcommand byte. The command byte specifies, if predicting or pruning should be performed and the subcommand byte specifies the type.

The commands and subcommands with their corresponding functions are defined in Table 3.3.

HEAD	HEADER	3 BYTE
CMD	COMMAND	1 BYTE
SCMD	SUB COMMAND	1 BYTE
DATA	DATA	0 - 255 BYTES

Table 3.2: Request Packet

CMD	SCMD	Function
0x01	0x01	SIMPLE PREDICTION
	0x02	PREDICTION WORST TREE TO END
	0x03	PREDICTION MOVE N WORST TREE
0x02	0x01	SIMPLE PRUNING
	0x02	PRUNING N TREES

Table 3.3: Commands and Subcommands

Prediction

There are three different prediction subcommands. The ”Simple Prediction” just does the prediction with the provided features (Table 3.4). The features are provided as single precision float in the IEEE 754 format [4] with LSB first.

FEAT0	FEATURE 0	4 BYTES
FEAT1	FEATURE 1	4 BYTES
...	...	... ...
FEATN	FEATURE N	4 BYTES

Table 3.4: Simple Prediction Data

There are two options for sorting the trees based on accuracy while predicting. The first one (Table 3.5) only looks at the worst tree and moves it to the end of the tree array. The second one (Table 3.6) looks at multiple trees and moves them back based on two parameters. The rule for calculating the amount, that a tree is moved back in the array is $s = is_t + o$, while i is the index in the array with the worst trees (ordered by accuracy, higher accuracy first), s_t is the parameter ”Steps Per Tree” and o is the parameter ”Move Offset”.

TRUEP	TRUE PREDICTION	4 BYTES
FEAT0	FEATURE 0	4 BYTES
FEAT1	FEATURE 1	4 BYTES
...	...	... ...
FEATN	FEATURE N	4 BYTES

Table 3.5: Prediction and Move Worst Tree to the End of the Array Data

NTREES	Number of Trees to Move	4 Bytes
OFFS	Move Offset	4 Bytes
STEPS	Steps Per Tree	4 Bytes
TRUEP	True Prediction	4 Bytes
FEAT0	Feature 0	4 Bytes
FEAT1	Feature 1	4 Bytes
...	...	... ...
FEATN	Feature N	4 Bytes

Table 3.6: Prediction and Move Worst N Trees Data

Pruning

Two different pruning functions allows for easy pruning. ”Simple Pruning” (Table 3.7) prunes the worst tree and ”Pruning N Trees” (Table 3.8) prunes the worst N trees.

EMPTY	No Data	0 Bytes

Table 3.7: Simple Pruning Data

NTREES	Number of Trees to Prune	1 Byte

Table 3.8: Pruning N Trees Data

3.4.3 Answer Packet

A answer packet (Table 3.9), coming from the microcontroller, starts with the header, followed by a return and error code.

HEAD	Header	3 Bytes
RET	Return Code	1 Byte
ERR	Error Code	0 - 1 Bytes
DATA	Data	0 - 255 Bytes

Table 3.9: Answer Packet

The prediction commands defined in Section 3.4.2, always return a packet, defined in Table 3.10. The prediction is returned as single precision float in the IEEE 754 format and the computation time as 32 bit integer in microseconds.

PRED	Prediction	4 Bytes
CMPTIME	Computation Time	4 Bytes

Table 3.10: Prediction Answer Data

The prune commands defined in Section 3.4.2, always return a packet, defined in Table 3.11. The remaining trees are returned as 16 bit integer.

REMTREES	Remaining Trees	2 Bytes

Table 3.11: Pruning Answer Data

Chapter 4

Model Training

In this chapter I describe how the model training part is built. Especially, I talk about the data and how to preprocess it, as data preprocessing is one of the main aspects in any machine learning task.

4.1 System Overview

In Section 3.1, the tools that were used are already described. The model training part is essentially packed in one function called `create_model()`, residing in the source file `model_creation.py`. It takes the paths for data, model parameters (number of trees, tree depth, minimal samples per leaf, maximum features per tree, train test fraction, sampling settings) as arguments and outputs a trained `RandomForestRegressor` object with evaluation scores.
`create_model()` is dependent on `data_preprocessing.py`, where all the data preprocessing happens. The main functions there are `load_data()` to load and merge the different data sources (described in Section 4.2), `load_preprocessed_data()` to load and prepare already preprocessed data and `preprocess_data()` to preprocess the aggregated data from the `load_data()` function.

4.2 Data Basis

The main dataset [5] used in this work is provided by multiple measurement stations, that are located in the ETZ building at ETH Zurich. The measurement stations have a solar panel and ambient sensors to measure the generated solar power, illuminance, ambient pressure, ambient temperature and ambient relative humidity. The solar power is sampled at a rate of `10 Hz` and the ambient sensors at a rate of `1 Hz`. The available data ranges from April 2018 to October 2020.
A subsecond sampling interval is not reasonable for this task, because ambient

changes such as temperature are very slow, I resample the data to an interval of 1 hour with the mean of the interval. This will also be my main prediction time, as this is considered one of the more difficult intervals [1].
The outdoor measurements are taken from the MERRA-2 project [6] for the location of the ETZ building. They contain temperature, pressure and solar irradiance and are sampled with an interval of 10 minutes.

The preprocessing of the data consists of three steps. The first step is to join the different sources into one pandas [7] `DataFrame`. As a second step, I resample the data into the desired interval (1 hour), taking the mean over this interval. In the third step, I add computed and lagging columns to the `DataFrame`. The calculated columns are two boolean columns (working time, working day), which equals `1`, if the time is between 09:00 AM and 06:00 PM and between Mondays and Fridays and three time columns (hour, day of the week, day of the year). As already described in Section 2.1, for each signal we add several values with different lags (1h, 2h, 3h, 4h, 24h and 1 week). The signal value without lag is not considered in the model training, as this is not known

Chapter 5

Microcontroller Implementation

In this chapter, I implement and test two different approaches to represent a random forest on a microcontroller and make predictions and talk about their advantages and disadvantages.

5.1 System Overview

I use the Texas Instruments MSP432P401R microcontroller as my low-power embedded platform, for which I evaluate both implementation versions. It is based around an Arm Cortex-M4F processor with a 32 bit architecture, with 48 MHz clock speed, 256 KByte of flash memory and 64 KB of RAM.

The model to C-code conversion is defined in the `model_exporter.py` source file. The main function `export_to_file()` takes a trained `RandomForestRegressor` object, the implementation approach (described in the following sections), a file path for the generated code and some code settings as arguments. It then generates a `.c` file and a `.h` file with the trees and implementation specific helper functions.

The code for the microcontroller is organised in four different `.c` files with corresponding headers. In `main.c`, the system is setup and the interrupt handler for the serial communication is defined. It reads each byte of an individual packet and stores it in a buffer, until there is a whole packet. It then calls the communication handler, which resides in `communication.c`. This handler checks the validity of the packet and delegates it either to the prediction handler or the prune handler based on the CMD byte (see Table 3.4.2). The prediction handler calls the different predict functions in `tree.c`, that was previously generated by the model exporter script. Additional functions, that are necessary, are defined in `utility.c`.

5.2 If-Else Approach

The basic idea in this approach is, that you can represent a binary decision tree as a series of `if`/`else`-clauses. With that, a single tree is represented in a function, that takes the features as argument and returns the prediction. In Listing 1 you can see an example with a feature array of size 15 and a tree depth of 2. From the features, only feature 0, 3 and 6 are used.

The advantage of this approach is the easy conversion script. The storing method of Scikit-learn for trees allows us to traverse the tree recursively from the root in a depth first, preorder manner, as it stores the threshold, feature index and the references to its children. The right child is always taken, if the feature is lower than the threshold, otherwise the left child is taken. As we want to build the `if`-clause in the format `feature[i] <= threshold[k]`, we traverse the tree right handed. So as long as we go to a right child, we can add a new line with an `if`-statement. When we reach a leaf, we add a `return`-statement. If we approach a right child, we add an `else`-statement on a new line.

Listing 1 If-Else Decision Tree Prediction with Depth of 2 and 15 Features

```
float tree0(float features[15]) {
    if(feature[6] <= 1.30739273e+02) {
        if(feature[0] <= 1.65000000e+01) {
            return 2.65752328e-05;
        } else {
            return 1.21320995e-05;
        }
    } else {
        if(feature[3] <= 1.96884800e+02) {
            return 3.12212301e-05;
        } else {
            return 4.39161221e-05;
        }
    }
}
```

To now get to a full forest, we have n different `tree`-functions, which are all called during prediction. To do this efficiently, we store the pointers to the `tree`-functions in an array, so that we can loop through them.

As we want to prune trees, that are performing bad, we introduce a variable

called `prune_pointer`, which at the beginning equals n. If we prune now, we just decrement this variable. Now we just predict trees in the pointer array up to this index.
As this pruning approach requires the trees to be sorted by their accuracies, we need additional routines. A simple approach is to track the worst performing tree in a prediction with the true value and move it to the end of the function array. This is not a perfect sorting of the trees, but doing this, one can expect that the worst trees are in the lower half of the array.
A more sophisticated approach is, that we track multiple trees and move them back by a certain amount. This allows us to have more fine grained control on how much to alter the array.
There is however the disadvantage with this approach, that pruning can only save on execution time and not on saving space, because the individual functions can't be removed from memory.

5.3 Array Approach

A different approach is, that we store the trees in arrays of nodes, containing the threshold value and the feature index. Now we have a function (Listing 2), that takes such an array and features as input and returns the prediction. The array is built in a breadth first manner (lower depth nodes ave lower array indices). To get the child-node indices from a node with index k, you can use (5.1) and (5.2).

Listing 2 Array Based Decision Tree Prediction with 15 Features

```
float tree_prediction(float features[15], node* tree_array, size_t max_depth) {
    float prediction;
    size_t depth = 0;
    size_t index = 0;
    while(depth <= max_depth) {
        if(tree_array[index].feature_index == -1) {
            prediction = tree_array[index].value;
            break;
        }
        if(features[tree_array[index].feature_index] > tree_array[index].value) {{
            index = (2 * index) + 2;
        } else {
            index = (2 * index) + 1;
        }
        depth++;
    }
    return prediction;
}
```

$$i_{\text{left child}} = 2k + 1 \tag{5.1}$$

$$i_{\text{right child}} = 2k + 2 \tag{5.2}$$

The difficulty in the conversion is, that we have to convert the pointer based model to an breadth first array assuming a full binary tree and dealing with incomplete trees.
This can be solved however with a queue based algorithm (see Algorithm 1). The main idea is to pop the front of the queue and enqueueing the children. This converts the tree model to a breadth first array.

Algorithm 1 Tree to Array Conversion

Input: *root node*, *tree depth*
Output: *array*

```
 1: function treeToArray(tree)
 2:     initialize queue
 3:     initialize array
 4:     add root node to queue
 5:     for i in range $2^{tree\ depth+1} - 1$ do
 6:         node = pop element from queue
 7:         if node is undefined then
 8:             add empty element to array
 9:         else if node is leaf then
10:             add (node.value, leaf) to array
11:         else
12:             add (node.value, node.feature) to array
13:         end if
14:         add node.left and node.right to queue
15:     end for
16:     return array
17: end function
```

To now extend this to a forest, the function (Listing 2) is called for each tree array, averaging the output from each call.

Pruning is implemented the same way as described in Section 5.2, looping through the array with pointers to the tree array up to the index of `prune pointer`.

A main advantage of this approach is, that we do not have lots of conditionals in the compiled code, which saves space. Another advantage is, that now the

trees are no longer stored as functions but rather as variables. With dynamic memory allocation, we have now the opportunity to delete trees from the memory and allocate this space for different purposes.
The main disadvantage is, that a tree always occupies the same amount of space (given the same depth), as nonexistent subtrees are represented by empty elements in the array. If we have sparse trees, this implementation would not be space efficient. We assume however, that we nearly always have full trees, as we chose primarily low tree depths.

Chapter 6

Evaluation

In this chapter, I evaluate the different implementation approaches regarding compiled size (Section 6.1) and prediction runtime (Section 6.2). In Section 6.3, I evaluate the effectiveness of pruning.

6.1 Flash Size

As space is limited on the microcontroller, we want the model to be as small as possible to have enough space for the main functionality. There are two factors, influencing the flash size, that I look at. The first one is obviously the model size and the other one are the compiler settings. We first look at the compiler and use the best settings for all further measurements.

6.1.1 Compiler

The compiler (`ti-cgt-arm_20.2.5.LTS`) has two flags, that heavily influence the size of the binary. One is `--opt_for_speed` (speed vs size trade-offs) and the other one is `--opt_level` (optimization level). It should be obvious, what the best settings are, but I want to evaluate how big the impact is.

To evaluate the `--opt_for_speed` compiler flag, I turn off the `--opt_level` flag. After that, I set the best `--opt_for_speed` flag and vary the `--opt_level` flag. For this, I use the data from all measurement stations with a train - test split of 90% and 15 features. The model parameters are set to 250 trees and to a tree depth of 4.

In Figure 6.1 you can see the impact, that `--opt_for_speed` has on the compiled size. The difference is significant with a 10% size reduction.

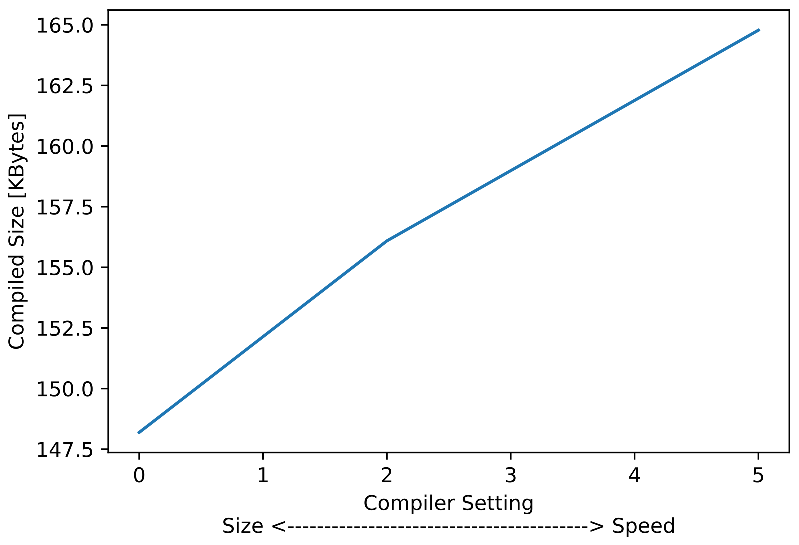

Figure 6.1: `--opt_for_speed` Impact on Compiled Size

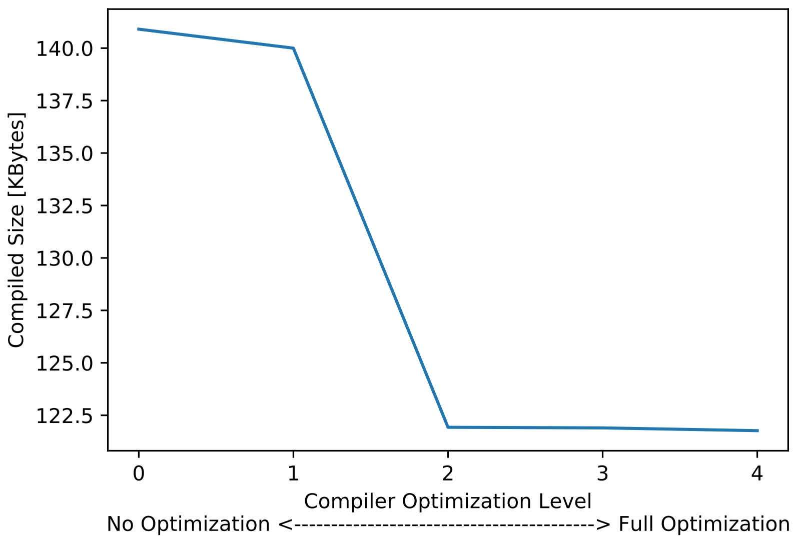

Figure 6.2: `--opt_level` Impact on Compiled Size

With the flag set to `0`, we can observe in Figure 6.2, that a major size reduction (around 13%) happens, when we change the `--opt_level` from `1` to `2`. The main reason is, that debug information is stripped out, making debugging harder. This leads to an overall size reduction of 26% comparing the optimal flags with no flags at all.

6.1.2 Model Size

The model size has a lower bound of space requirement calculated by (2.3). Analyzing this formula leads us to the assumption, that the space requirement has an exponential dependency on the tree depth d and a linear dependency on the number of trees n.
To prove this assumption, we train models on all stations (90% train - test split) with varying tree depths and number of trees and compile them to get the flash size. We do that for both the If Else Implementation and the Array Implementation.

As we can see in Figure 6.3, an increase in number of trees grows the size linearly, whereas there is an exponential growth on the tree depth axis. To calculate the size per node, we want to eliminate the space required by other functions. So we calculate the difference of flash size and of nodes with (2.2). This leads to a size per node of around `15.5 bytes`. That is over 2.5 times the ideal case.

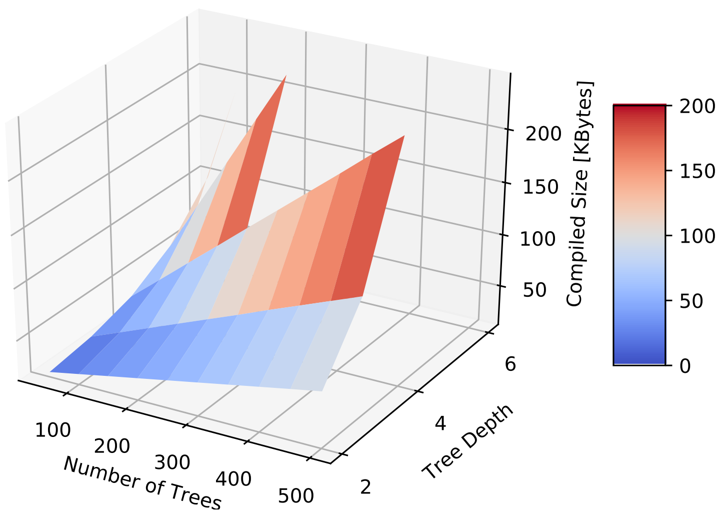

Figure 6.3: Model Parameter Impact on Compilation Size with the If Else Implementation

Looking at the Array Implementation in Figure 6.4, the situation looks different regarding the node size. Calculating it the same way leads us to a size a bit larger than `8 bytes`. This was to be expected, as we store each node as a struct containing a `4 bytes` floating point value and a `2 bytes` integer value. The remaining `2 bytes` are due to the struct padding, as the size of a struct has to be a multiple of `32 bits`.

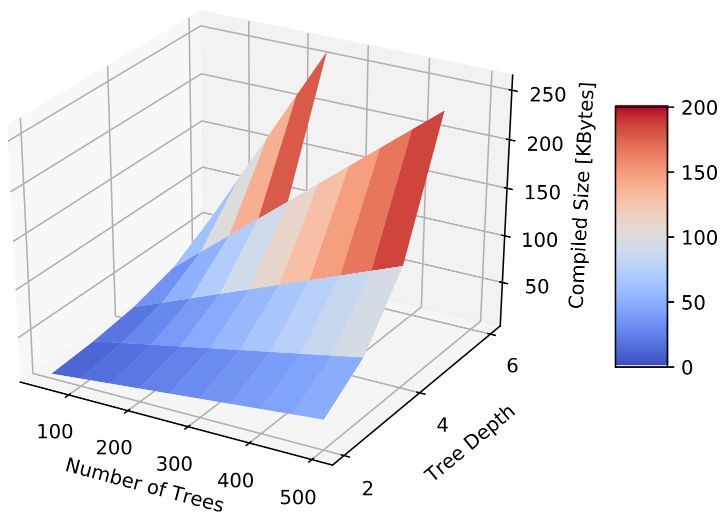

Figure 6.4: Model Parameter Impact on Compilation Size with the Array Implementation

To now evaluate, how the compiled size affects the model accuracy, so called global models are trained. Global models are models, that are trained on other measurement stations, than they are evaluated, In this evaluation, I trained on all stations but station 14 with a split at 30.08.2020 and evaluated with data from station 14 from this date onwards. The tree depth is varied between 2 and 6 and the number of trees between 50 and 500. The relative root mean squared error compared to the maximum power and the compiled size is then plotted for both implementation approaches.

In Figure 6.5 and 6.6, you can see, that we get the best accuracy for a certain amount of space with high tree depth. The number of trees however doesn't significantly improve the accuracy.

6.2 Runtime

Runtime is equally important as space efficiency, as this is an indirect measure on how much energy is needed to predict. This also depends on the compiler flags described in Section 6.1.1 and the model size.

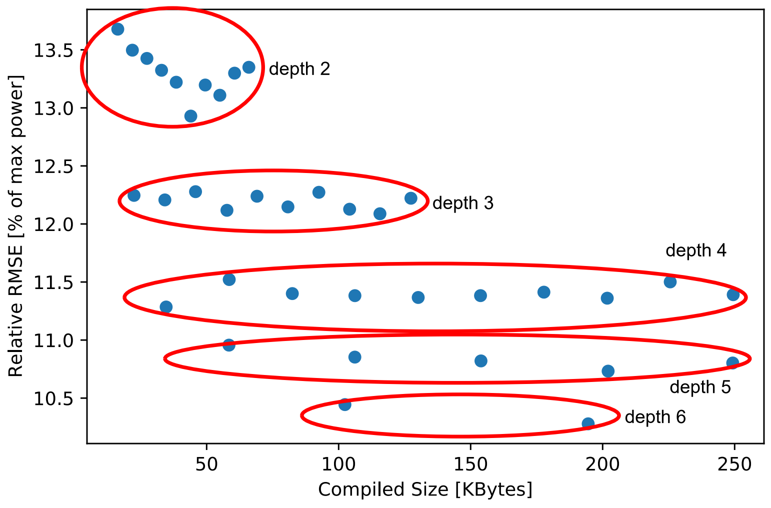

Figure 6.5: Relative RMSE Compared with Compiled Size If Else Approach

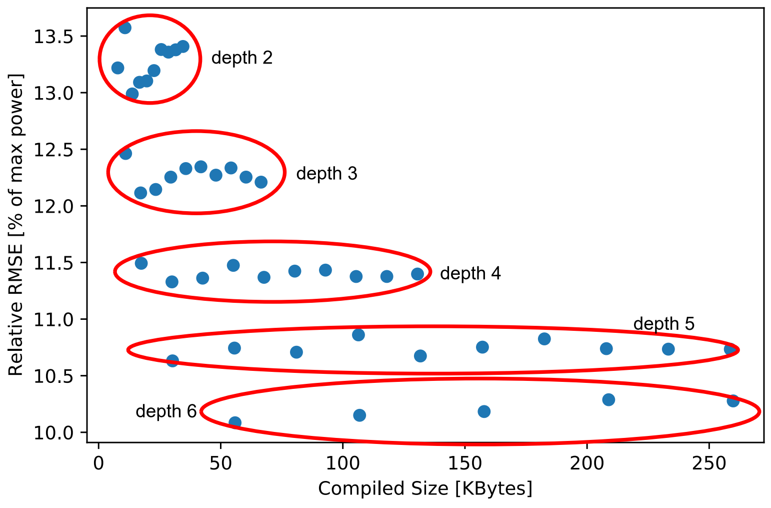

Figure 6.6: Relative RMSE Compared with Compiled Size Array Approach

6.2.1 Compiler

The setup looks the same way as in Section 6.1.1, but instead of the compiled size, I measure the runtime of a single prediction. I again measured the impact of the `--opt_for_speed` and the `--opt_level` compiler flag.

The impact of the `--opt_for_speed` flag behaves just the other way round as intended by the compiler. As you can see in Figure 6.7, if we optimize on speed, the execution time is actually a little bit higher (about 1%), than if we optimize on size.

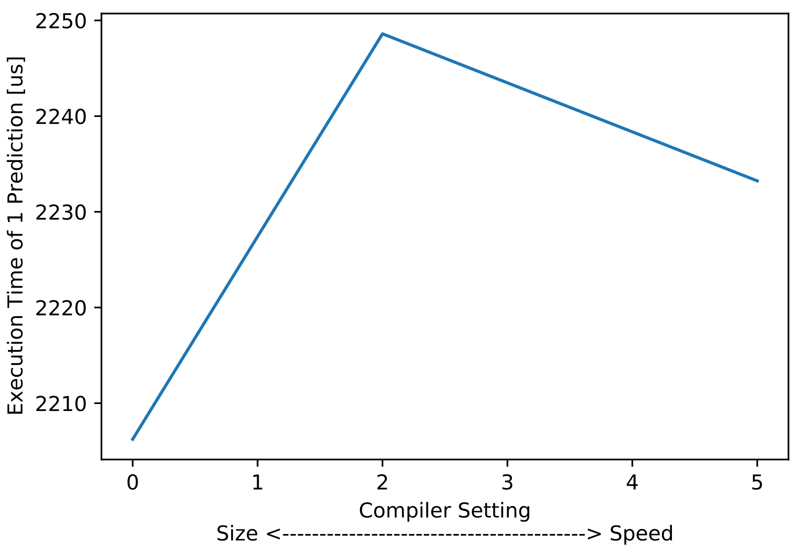

Figure 6.7: `--opt_for_speed` Impact on Runtime

However the `--opt_level` flag allows us to save about 4% of execution time compared to the worst setting, as you can see in Figure 6.8.

6.2.2 Model Size

In this section, I will compare different model sizes for each implementation approach and measure their single prediction runtime with the same setup as in Section 6.1.2. I expect the runtime to grow linearly both with increasing tree numbers and tree depth, because each tree adds the same amount of nodes to be evaluated and each level of depth adds a single node.

This may indeed be observed, if we look at Figure 6.9. This shows the execution time for the If Else Implementation Approach. To compare the different approaches, we calculate the time per node. To do that, we have to know, how many nodes are evaluated per prediction. As only one node is evaluated in each level, the number of nodes evaluated is $n(d+1)$. For this approach, it equals `1434 ns` per node.

Comparing the If Else Approach to the Array Approach (Figure 6.10, we can see quite a huge difference in the execution time. There, the time used per node

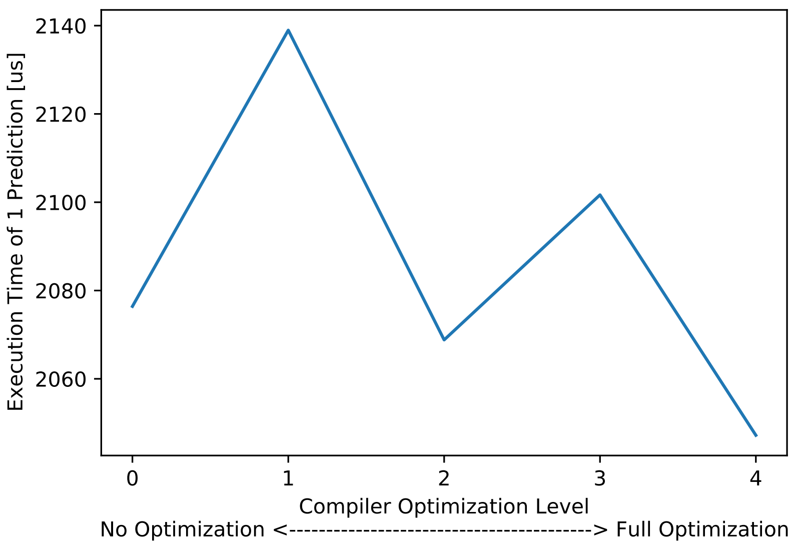

Figure 6.8: `--opt_level` Impact on Runtime

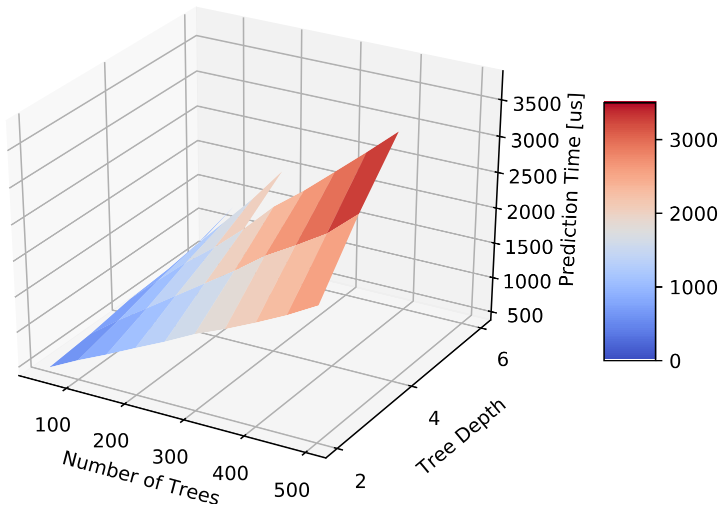

Figure 6.9: Execution Time per Prediction If Else Approach

is about `745 ns`, which is a time saving of 48%.

6.3 Pruning

To prove the effectiveness of pruning, I look at two different settings. One represents a local setting (use the same station for training and evaluating

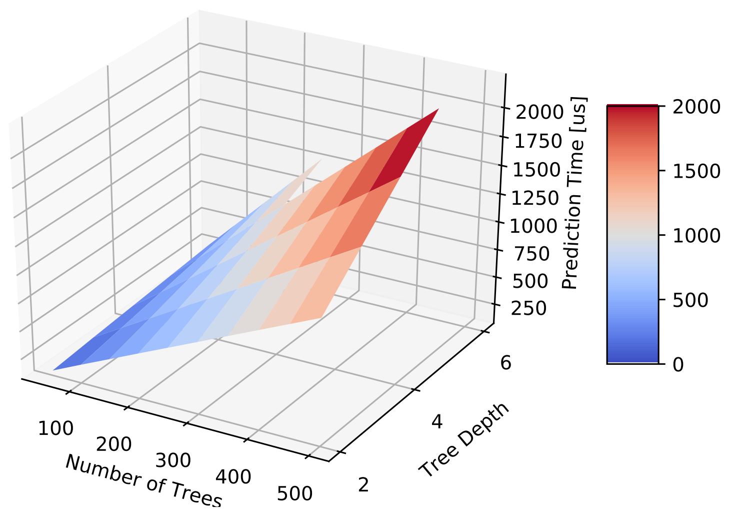

Figure 6.10: Execution Time per Prediction Array Approach

accuracy) and the other one represents a global setting (all but one station used for training, the remaining station is used for evaluating accuracy). I will look at a small number of trees (50) and a high tree depth (8).
The training now happens in two steps. The first step is the offline training of the model and the second step is prediction with pruning, where the trees get sorted by accuracy.
I train a new model for each pruning size I look at. This is to ensure, that I also see the variability in accuracy after the offline training.

In Figure 6.11, you can see, how the local setting perform. There is no initial improvement in accuracy than one might expect. It stays roughly the same up to about 80% pruned trees and after that it gets slightly worse. With only one tree left (technically no random forest anymore) the relative error increased about 2.5%.

A similar observation can be made with the global setting (Figure 6.12). If we smooth out the accuracy curve of the pruned model, we have a deterioration after pruning about 80% of the trees.

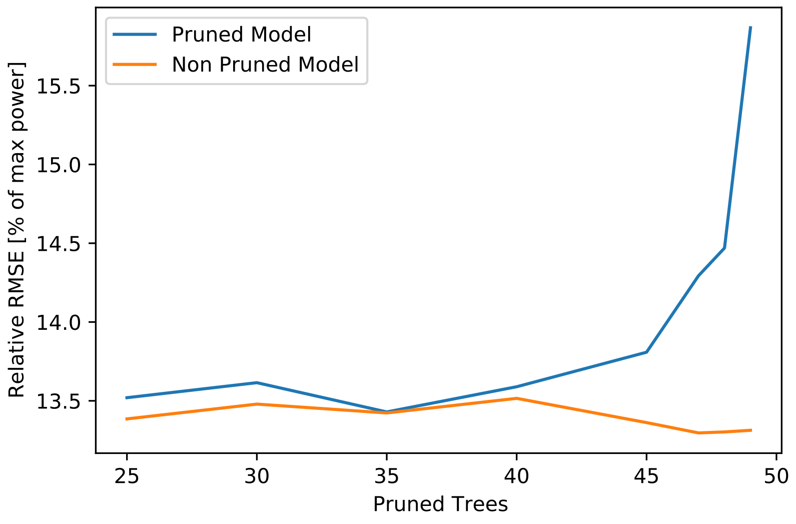

Figure 6.11: RMSE Pruned Models, Local Setting (Station 6)

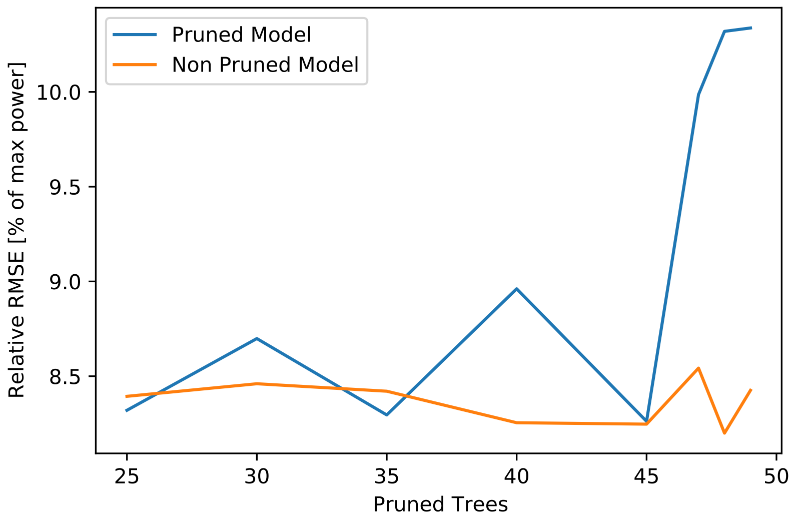

Figure 6.12: RMSE Pruned Models, Global Setting (Station 14)

Chapter 7
Conclusion

I implemented an indoor energy prediction algorithm based on a random forest in this thesis. Two implementation approaches with online improvement techniques were developed and evaluated.

The evaluation has shown, that with an array based implementation, you are able to store a model space efficient (25% more than the ideal case because of struct padding) on a microcontroller and have a fast prediction time. In comparison, the approach based on conditionals (if/else), uses almost double the amount of memory and double the amount of execution time for each prediction.

Trying to improve the model on-the-fly with pruning doesn't work that well, but because the model stays equally accurate with up to 80% of pruned trees, this could be used to free up space for the use of other purposes.

7.1 Further Work

As pruning doesn't seem to work as expected, one could consider looking into other techniques to improve the model with limited resources. For example, one could copy trees and add noise or maintain data from bad predictions and train new trees based on this data.

To prove the usefulness of this prediction algorithm, one should look into different scheduling algorithms and put them to the test with realistic workloads. That way, we would see, if the energy expenditure for predicting actually leads to a better system performance, than other energy harvesting scheduling algorithms.

Bibliography

[1] Colin Berner. Prediction Models for Indoor Solar Energy Harvesting. January 2019.

[2] F. Pedregosa, G. Varoquaux, A. Gramfort, V. Michel, B. Thirion, O. Grisel, M. Blondel, P. Prettenhofer, R. Weiss, V. Dubourg, J. Vanderplas, A. Passos, D. Cournapeau, M. Brucher, M. Perrot, and E. Duchesnay. Scikit-learn: Machine learning in Python. *Journal of Machine Learning Research*, 12: 2825–2830, 2011.

[3] Leo Breiman. Random forests. *Machine Learning*, 45(1):5–32, 2001. ISSN 1573-0565. doi: 10.1023/A:1010933404324. URL `https://doi.org/10.1023/A:1010933404324`.

[4] IEEE Standard for Floating-Point Arithmetic. *IEEE Std 754-2019 (Revision of IEEE 754-2008)*, pages 1–84, 2019. doi: 10.1109/IEEESTD.2019.8766229.

[5] Lukas Sigrist. Tracing indoor solar harvesting, November 2019. 2nd Workshop on Data: Acquisition to Analysis (SenSys/BuildSys 2019); Conference Location: New York City, NY, USA; Conference Date: November 10, 2019.

[6] Ronald Gelaro, Will McCarty, Max J. Suárez, Ricardo Todling, Andrea Molod, Lawrence Takacs, Cynthia A. Randles, Anton Darmenov, Michael G. Bosilovich, Rolf Reichle, Krzysztof Wargan, Lawrence Coy, Richard Cullather, Clara Draper, Santha Akella, Virginie Buchard, Austin Conaty, Arlindo M. da Silva, Wei Gu, Gi-Kong Kim, Randal Koster, Robert Lucchesi, Dagmar Merkova, Jon Eric Nielsen, Gary Partyka, Steven Pawson, William Putman, Michele Rienecker, Siegfried D. Schubert, Meta Sienkiewicz, and Bin Zhao. The Modern-Era Retrospective Analysis for Research and Applications, Version 2 (MERRA-2). *Journal of Climate*, 30(14):5419 – 5454, 2017. doi: 10.1175/JCLI-D-16-0758.1. URL `https://journals.ametsoc.org/view/journals/clim/30/14/jcli-d-16-0758.1.xml`.

[7] Wes McKinney. Data Structures for Statistical Computing in Python. In Stéfan van der Walt and Jarrod Millman, editors, *Proceedings of the 9th Python in Science Conference*, pages 56 – 61, 2010. doi: 10.25080/Majora-92bf1922-00a.