TU Delft

Machine Learning for Electrical Engineering
EE4C12

Fault Detection for Residential PV Systems

23rd October 2022

Gabriele Corbo 5854083
G.Corbo@student.tudelft.nl
Giacomo Mussita 5852862
G.Mussita@student.tudelft.nl

Summary

In this report is described the framework we followed to develop an ML model that uses Photovoltaic system power output to detect malfunctions. Using binary classification, we have separated faulty system conditions from healthy system conditions. With multiclass classification, it is possible to determine the type of fault that is causing a system to malfunction. Additionally, the different meteorological conditions like incident solar irradiance, sun position and ambient temperature can be utilized to learn complex relations between these features. Another aspect which plays a role is system age, as different fault types are more likely to occur at specific moments of the system's lifetime. We approached the tasks by implementing many models of ascending complexity in order to find the most accurate one and then tested it to determine its goodness.

1 ML pipeline

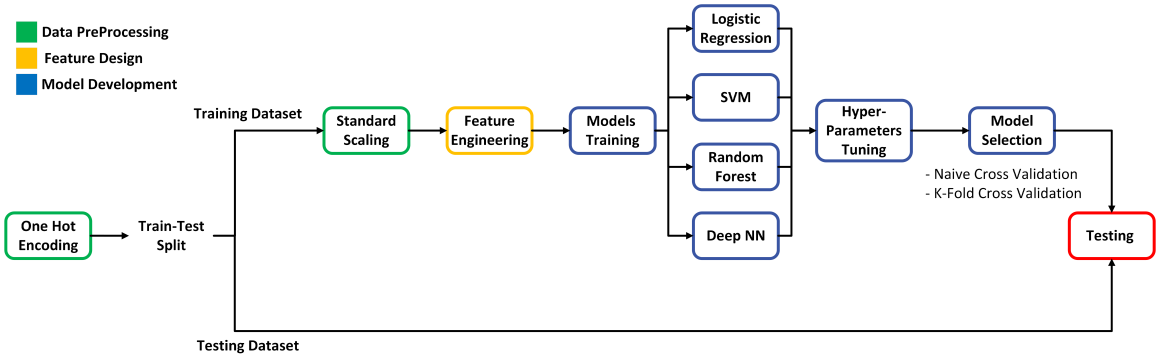

Figure 1: Pipeline used to develop machine learning models for fault detection of residential PV systems. The pipeline includes the fundamental steps needed for binary and multiclass classification (only the models applied are different). The steps of the pipeline are described in details in the sections of this report.

2 Data Pre-Processing

After having imported the `Project_Data_EE4C12_SET_PV.csv` dataset, we started by converting the column `Fault_Type` into a one-hot encoding of the fault type classes. By doing so, we removed any potential undesired ordinal relationship between labels. Additionally, we looked through the dataset for any missing values, but none were found.

Then, using the `train_test_split()` [1] function, we divided the dataset into two sets: one set contained the data for the training operations, while the other one was used for the testing of the final model. The latter set must not take part in any of the following pipeline steps, as it would bias the results, so it was put aside. Instead, the training set includes the data we used for both the model parameter optimization and the cross-validation of the models. Moreover, we were able to separate the feature matrix from the output class vectors thanks to the split. Later, we proceed with a further split of the training data into a set used for model fitting and one for validation.

Afterwards, we computed the correlation between the six features present in the training set. Figure 2 shows an heat-map of the matrix: it is possible to see how the `Irradiance` is highly correlated with the `System_Power` (reasonable for a photovoltaic system), while the `Ambient_Temperature` is rather correlated with the `Irradiance` and the `Sun_Elevation`, as we would expect from these physical quantities. However, except for the diagonal that contains the variance of each feature, most of the entries of the matrix are far below 0.8. For this reason we decided not to apply any feature reduction technique, like the principal component analysis. In this way we were able to preserve the special contribution that each individual feature could provide.

The `StandardScaler()` [1] function was then used to perform the normalization and standardization of the training set. In this way, all the features have zero mean

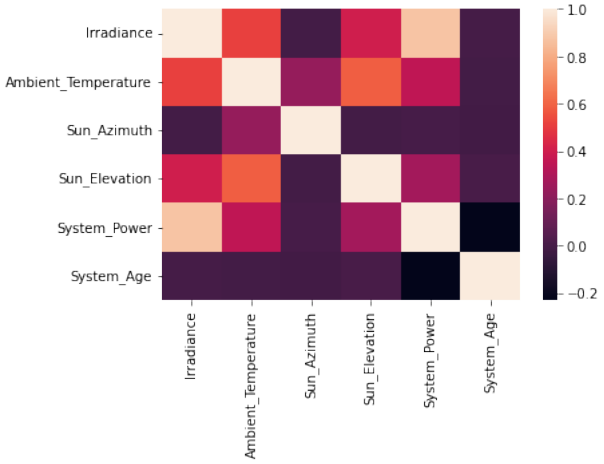

Figure 2: Correlation matrix

and unitary variance. Since all the quantities fall within the same range of values and the loss function has a more circular shape (because of the standardization), gradient descent can converge more quickly, making model learning simpler.
Ultimately, we thought about increasing the number of features, as six seemed not enough for some models, and we were curious how the performances would have changed with more complex features. Therefore, we computed a polynomial transformation of order 2 of the feature space. To convert the dataset, we used the `PolynomialFeatures()` [1] function. We trained most of the models on both sets (original and polynomial).

3 Binary Classification

3.1 Performance Metrics

In order to develop a binary classification model to predict damaged systems, we built a variety of classifiers and compared their performances on the validation set to choose the best. Our search was primarily driven by accuracy:

$$Accuracy = \frac{True\,Healthy + True\,Faulty}{All}$$

Yet, we also took recall and precision into account:

$$Precision = \frac{True\,Faulty}{True\,Faulty + False\_Faulty}$$

$$Recall = \frac{True\,Faulty}{True\,Faulty + False\,Healthy}$$

We made the decision to take precision into consideration since, if this value is too low, the model would tends to classify as faulty also system that are not. This may be a concern since we would perform maintenance even when it is not necessary. At the same time, recall must be high, otherwise, we risk ignoring numerous faults,

which is detrimental for safety. Moreover, for each model we also compute the following metrics:

$$Overfit = \frac{Training\,Accuracy - Validation\,Accuracy}{Validation\,Accuracy}$$

We called this $Overfit$ because it gives an idea of whether the model is overfitting or not. If its value is high, it indicates that the model's performance on unseen data is not as good as the performance on training data.

3.2 Models Training and Hyper-Parameters Optimization

The collection of models that we trained is reported in Table 3. Since we were facing a binary classification problem, a logistic regression trained on the set of six original features was the first model we decided to explore. By reducing the cross entropy over training samples, logistic regression learns how to predict which class a datapoint belongs to. To guarantee the convergence of gradient descent with the desired tolerance (the default tolerance is 0.0001), we set the `max_iter` [1] parameter to 1000. The results we obtained were far from satisfactory, and when we looked at the accuracy value, we realized the model was underfitting the data. Therefore, we opt to use polynomial features. As a result, performances improved, since the model became more complex due to the transformed features set.
However, we were not happy with the accuracy thus far, since it was below 80%, so we tried training various support vector classifiers (SVC). We trained SVCs on the two different feature sets with two different kernels: linear and gaussian. After fitting, we noticed that all the models presented a significant unbalance between the recall and the precision. An example is shown in Table 1. Thus, we decided to weight more the faulty datapoints in the cost function of the SVC (using `class_weight` [1] parameter). The accuracy did not improve, but we obtained a better trade-off between recall and precision (see Table 2).

		Predicted	
		Healthy	Faulty
Actual	Healthy	682	86
	Faulty	290	472

Table 1: confusion matrix of SVC with Gaussian kernel, trained on six features set. The classifier is not able to detect many faulty systems.

		Predicted	
		Healthy	Faulty
Actual	Healthy	552	216
	Faulty	168	594

Table 2: confusion matrix of SVC with Gaussian kernel, trained on six features set, with weighted classes (0.9 for healthy, 1.1 for faulty)

Afterward, we proceeded with the tuning of the regularization parameter `C` of the `SVC` class of `sklearn` [1] for each support vector machines. This parameter represents how much misclassified points are penalised within the loss function:

$$g(b, \boldsymbol{w}, \xi) = \frac{1}{2}||\boldsymbol{w}||_2^2 + C\sum_{i=1}^{P}\xi_i$$

ξ_i are slack variables that represent the distance of a misclassified point from the correct boundary decision, based on the following constraints:

$$y_i(\boldsymbol{w}^T \boldsymbol{x}_i + b) \geq 1 - \xi_i$$

$$\xi_i \geq 0 \quad i = 1, \ldots, P$$

In order to tune `C`, we trained the model with several values of the hyperparameter, and then we tested each model's accuracy using the validation dataset. After first increasing the hyperparameter by an order of magnitude each time, we concentrated on a range of values where the model performed well. We did not search for the value that corresponded to the maximum, since doing so would have resulted in an overfitting of the validation data. However, we still observe from Table 3 that the SVC models overfit the validation set: indeed the $Overfit$ metrics are negative for three out of four models, meaning that the models perform better on the validation set than on the training set.
Eventually, we decided to train also a `RandomForestClassifier` [1]. This model is a bagged ensemble of several decision trees, so the model leverages the potential of every single tree. Also the main hyperparameter of this model, the number of trees, was tuned through recurring evaluations of the accuracy on the validation set. Following that, we decided to tune also the maximum depth of each tree. If we left the default value, the tree would be enlarged until all of the training datapoints are accurately categorised. Clearly, this causes the training data to be overfitted. Therefore, we performed cross validation to find a reasonable maximum depth that guaranteed a trade-off between accuracy and overfitting (Figure 3).

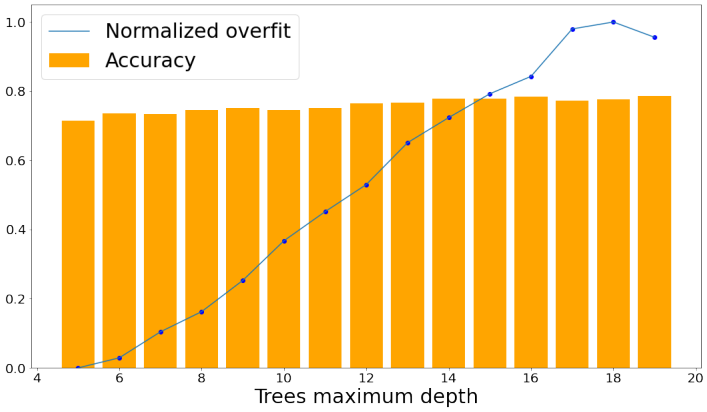

Figure 3: accuracy and overfit of the random forest classifiers with respect to the maximum depth of the tree. The overfit score is normalised between 0 and 1 to ease the visualization. As it is possible to see, the accuracy does not change much as we increase the depth of the trees, but the models tend to overfit more the training data.

After training all these models, we were not satisfied with the results yet, especially because we thought the accuracy was too low. In order to find a model able to capture the complexity of the phenomena and increase accuracy, we chose to investigate neural networks. We build several nets with different shapes using `Keras` [2]. We

investigate networks having two, three, and four levels, with nine to four nodes in each layer. A layer can extract more features the more nodes it has, while as we increase the network's depth (number of layers), the complexity of the models and of the features rises. In this sense, the neural network serves as a tool for feature learning, because cross-validation nets are taken into account in each layer. Since we struggled to identify a suitable collection of features for the task, neural networks seemed to be the best option.
As for the hyperparameters of the networks, we chose to perform 5000 epochs with a batch size (number of samples per mini-batch gradient update) of 32 (default value). Because we used early stopping with a `Patience` [2] parameter of 50, the actual number of training iterations was always far lower than 5000. This means that the model stops learning if the validation accuracy does not improve after 50 iterations. In this way, we improved the generalization capability of the model and we prevented overfitting of the training data. This choice appeared to be rather effective, because it provides for a good level of complexity without relying too heavily on the training data.
To select the best architecture, we performed k-fold cross-validation (Section 3.3) over different networks. It turns out that the best performance is obtained for the network in Figure 4.

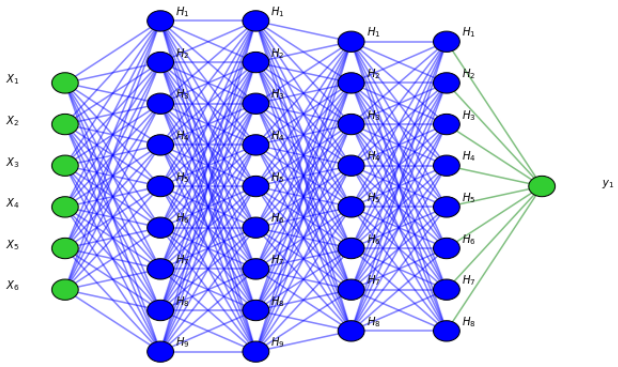

Figure 4: architecture of the best neural network. This networks presents 9 nodes for the first two layers, and 8 nodes for the last two layers. Apparently, this structure is able to describe with sufficient complexity the PV system.

3.3 K-Fold Cross Validation

We utilized k-fold cross-validation with five splits of the data into training and validation sets to compare the models and choose the one that performed the best. We calculated the average validation accuracy over the various splits for each model. Since the number of splits reduces the variance of the accuracy on the validation set, it is significantly more realistic than the accuracy obtained with naive cross-validation. The results are reported in Table 3.

Models comparison					
Model	**Accuracy**	**Recall**	**Precision**	**F1-score**	**Overfit**
Logistic regression	67.95%	68.24%	68.51%	68.38%	-1.13%
Logistic regression with polynomial features	77.12%	71.13%	80.65%	75.59%	-2.58%
Linear SVC	67.27%	67.59%	68.58%	68.08%	9.61%
Linear SVC with polynomial features	76.73%	74.41%	77.88%	76.11%	-2.43%
Gaussian SVC	81.72%	80.97%	83.95%	82.43%	-9.59%
Gaussian SVC with polynomial features	82.03%	80.97%	82.60%	81.78%	-8.73%
Random Forest	77.18%	76.51%	78.57%	77.53%	20.29%
Deep Neural Network	84.15%	78.35%	89.10%	83.38%	17.77%

Table 3: scores of the chosen models. Accuracy is computed using k-fold cross-validation (except for models trained on polynomial features), while other metrics are evaluated on the validation set. For Deep Neural Networks metrics refers to the best performing architecture. The SVC scores are based on the weighted models.

As can be seen, the accuracy gets better as we make the model more sophisticated. With the exception of the SVC with gaussian kernel, we can state that using polynomial features instead of the original ones results in higher performances. It is crucial to note that the models trained using polynomial features have not undergone k-fold cross validation. As a result, the corresponding accuracy might not be entirely trustworthy.
Therefore, we can conclude that the best model is the Deep Neural Network, since it has the best accuracy and the best F1 score, which combine both the recall and the precision.

3.4 Testing

Finally, we evaluated the best model on the testing set, to see how the model performs on unseen data. We trained the neural network on the training set first, and then we took the model at the epochs where the maximum accuracy was achieved.
Then, we predict the labels of the test set samples. Keep in mind that before using the test set, it must first be scaled using the `StandardScaler` [1] created during the pre-processing (Section 2). The results are reported below:

$$Accuracy = 82.89\% \qquad F1\_score = 81.73\%$$

$$Recall = 77.24\% \qquad Precision = 86.78\%$$

We can be satisfied with our Deep Neural Network model, since the performance on unseen data are comparable to the one obtained during training. The model generalizes rather well.

4 Multiclass classification

4.1 Models Training and Selection

In order to develop a multiclass classification model that allows the detection of faulty systems, we tried multiple options. We basically repeated all the steps done for the binary classification, so we first implemented a Multiclass Logistic Regression using the six features resulting in an unsatisfactory result in terms of accuracy. We implemented the same model utilising the previously discussed Polynomial features, which led to a consistent improvement in the goodness of fit. Then we experimented with the implementation of a Random forest classifier and a Support vector machine model with a gaussian kernel using both linear and polynomial combinations of features, performing also a hyperparameters optimization. Furthermore, a K-fold cross-validation was performed for most of the models, but without producing any significant results; hence, it was decided not to employ it. The Gaussian SVM proved to be the most accurate model among these, therefore we also attempted implementing a Bagging Classifier of such a model. Finally, we decided to build a Deep neural network with 3 hidden layers. Figure 5 shows the dimensions of the implemented network. The implementation was carried out using the functions of the library `keras`[2] and we trained it with a batch size of 10 and a large number of Epochs.

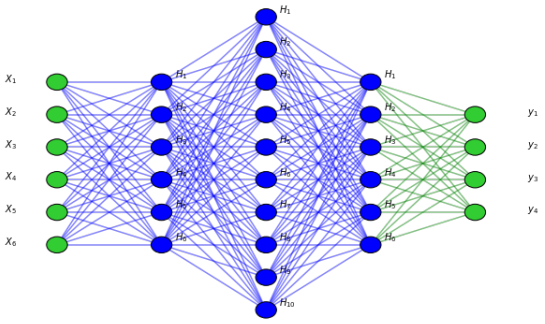

Figure 5: Deep Neural Network

The obtained results are:

Models comparison	
Model	Accuracy
Logistic Regression	63.20%
Polynomial Logistic Regression	71.96%
Gaussian SVM with $C = 800$	78.30%
Gaussian SVM with Polynomial features	76.40%
Bagging Gaussian SVM	77.05%
Random Forest Classifier	71.11%
Deep Neural Network	80.13%

Table 4: scores for various models

We decided to select the Deep Neural Network model because have the best performances in terms of Accuracy score and confusion matrix:

	0	**1**	**2**	**3**
0	**688**	60	18	2
1	71	**174**	4	1
2	94	27	**119**	14
3	1	8	4	**245**

Table 5: Confusion matrix on the **Validation** set, where on the column are represented the Predicted Labels and on the rows the True ones, as in the above confusion matrices.

4.2 Shapley values of features

In conclusion, we computed the Shapley value of each feature in order to understand their contribution to the prediction made by the neural network. The Shapley value is the average expected marginal contribution of one feature after all possible combinations have been considered. This value helps to determine a payoff for all of the features when each player might have contributed more or less than the others and is calculated by computing a weighted average payoff gain that feature i provides when included in all coalitions that exclude i:

$$\Phi_i(v) = \sum_{S \subseteq N \setminus \{i\}} \frac{|S|!(N - |S| - 1)!}{N!} (v(S \cup \{i\}) - v(S))$$

where N is the maximum coalition with all the features, $S \subseteq N \setminus \{i\}$ every possible coalition and $v(S)$ its value. In our study case, the value of a partition $v(S)$ is a $1x4$ array corresponding to the output of the Deep neural network using the features in the partition S. We used the `shap.DeepExplainer`[3] to compute the Shapley values of our features in the Deep neural network:

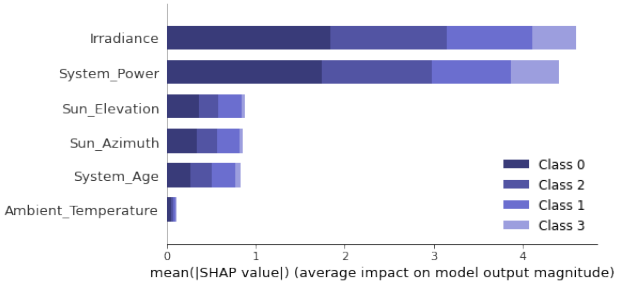

Figure 6: Shapley values of the six features

We can notice how all the features have a positive contribution, with the `Irradiance` that seems to be the most important one with a significant Shapley value for all the four possible response classes.

4.3 Testing

Finally, we evaluated the selected model on the Testing set, to see how the model performs on unseen data. We trained the neural network on the complete dataset first, and then we took the model at the epochs where the maximum accuracy was achieved. The obtained results are:

$$Accuracy = 79.78\%$$

	0	**1**	**2**	**3**
0	**797**	69	42	0
1	90	**205**	11	5
2	99	23	**151**	15
3	0	6	4	**283**

Table 6: Confusion matrix on the **Testing** set, where on the column are represented the Predicted Labels and on the rows the True ones, as in the above confusion matrices.

We can be satisfied with our Deep Neural Network model, since the performance on unseen data are comparable to the one obtained during training. The model generalizes rather well.

Conclusions

The obtained results are satisfactory and enable us to classify PV failures with good performance. We will employ a 4-layers Deep Neural Network as a binary classifier to find broken systems for new unseen operational scenarios. While using a 3-layers Deep Neural Network as a multiclass classifier to categorise the operational status of a system for new, unknown operational circumstances. Indeed, the neural networks provide a solution to the problem of identifying a meaningful set of transformed features.
To conclude, the models may be improved by gathering more data of different kinds, which would necessitate the use of more sensors, for example inner temperature sensors or infrared grid sensors (pointing towards the cells).

References

[1] F. Pedregosa et al. 'Scikit-learn: Machine Learning in Python'. In: *Journal of Machine Learning Research* 12 (2011), pp. 2825–2830.

[2] Francois Chollet et al. *Keras*. 2015. URL: `https://github.com/fchollet/keras`.

[3] Scott M. Lundberg et al. 'From local explanations to global understanding with explainable AI for trees'. In: *Nature Machine Intelligence* 2.1 (2020), pp. 2522–5839.