



**SOLAR ENERGY
TECHNOLOGIES OFFICE**
U.S. Department Of Energy



Distribution System Model Calibration Algorithm Documentation

I. Table of Contents

I.	Table of Contents	1
I.	Introduction.....	2
II.	Phase Identification.....	2
1.	Spectral Clustering Ensemble Method	2
2.	Sensor-based Method	6
III.	Meter to Transformer Pairing	11
1.	Overview.....	11
2.	Output Description	12
3.	Function Descriptions.....	12
IV.	Sample Data.....	15
1.	Overview.....	15
2.	Injected Data Issues.....	15
3.	Included Data Files	16
V.	Code Notes	18
VI.	Related Publications from the Project.....	18
VII.	Acknowledgments	19
VIII.	References	19

I. Introduction

The code in this library was developed by Sandia National Laboratories under funding provided by the U.S. Department of Energy Solar Energy Technologies Office as part of the project titled “Physics-Based Data-Driven grid Modelling to Accelerate Accurate PV Integration” Agreement Number 34226. More information about the project and resulting publications can be found at <https://www.researchgate.net/project/Intelligent-Model-Fidelity-IMoFi>.



Three distribution system model calibration algorithms are included in this release. There are two algorithms for performing phase identification: one based on an ensemble spectral clustering approach and one based on leveraging additional sensors placed on the medium voltage. Start with the `CA_Ensemble_SampleScripts.py` file and the `SensorMethod_SampleScript.py` file, respectively. The third algorithm identifies the connection between service transformers and low-voltage customers (meter to transformer pairing algorithm). Start with the `MeterToTransPairingScripts.py` file.

There is also a sample dataset included to facilitate the use of the code. The dataset will load automatically when using one of the sample scripts provided. For more details, please see Section IV.

This code and data are released without any guarantee of robustness under conditions different from the ones tested during development.

Questions or inquiries can be directed to Logan Blakely (Sandia National Laboratories) at lblakel@sandia.gov.

II. Phase Identification

1. Spectral Clustering Ensemble Method

The code for the spectral clustering ensemble method is broken into three files: `CA_Ensemble_SampleScripts.py`, `CA_Ensemble_Funcs.py`, and `PhaseIdent_Utills.py`. Start with the `CA_Ensemble_SampleScripts.py` file; it will automatically load the required sample data and run the spectral clustering ensemble algorithm. `CA_Ensemble_Funcs.py` contains the primary functions that make up the method, and `PhaseIdent_Utills.py` contains helper functions.

For more details on this method, please see [1].

a. Overview

This algorithm performs phase identification of customers by using their voltage timeseries measurements to cluster similarly related customers, without any other data requirements of topology information, customer power measurements, or measurements from the substation or transformers. It takes as input the voltage timeseries from advanced metering infrastructure (AMI) meters which have been converted into per-unit representation and transformed into a change in voltage timeseries by taking the difference in adjacent measurements. The algorithm loops through the available data in interval sizes specified by the `windowSize` parameter. A `windowSize` of 384 may be a good place to start; values between 96 and 384 will likely work fine. Larger window sizes tend to work better, but there is a tradeoff depending on the amount of missing data in the dataset. At each iteration, customers with missing data during that window are excluded, and the remaining customers are clustered using the `sklearn` implementation of spectral clustering. The clustering results from each window are used to build a co-association matrix where the entries for customers that were clustered

together are incremented. After all windows have been processed, the co-association matrix is normalized by dividing each entry by the number of windows where that pair of customers were both included in the window (i.e. neither customer was excluded due to missing data). The normalized co-association matrix is then used as a pre-computed affinity matrix to the spectral clustering algorithm for the final clustering determination. The number of final clusters can be set based on the feeder topology. 4 or 7 final clusters might be a good place to start, however this parameter may change from feeder to feeder. For example, if the feeder has voltage regulation devices a larger number of clusters may be required. The resulting clusters will be groupings by phase, but the final mapping from the cluster to a particular phase is left to a subsequent step. If the original utility labels are available and you believe they are at least 50% accurate, then the final mapping can be done using a majority vote with the original labels. This is what is done in the sample script provided.

b. Output Description

The two primary outputs from the spectral clustering ensemble phase identification method are `finalClusterLabels` from the `CAEnsemble` function and `predictedPhases` from the `CalcPredictedPhaseNoLabels` function. The `finalClusterLabels` field contains the results from the final cluster assignment; these cluster labels correspond to phase groupings but are not mapped to a particular phase. If the original phase labels are available, and you believe they are at least 50% accurate, then they can be used to do the mapping between the clusters in `finalClusterLabels` and particular phases. This is the `predictedPhases` field where each customer has been assigned a predicted phase based on the final cluster and a majority vote using the original utility labels.

```
Spectral Clustering Ensemble Phase Identification Results
There are 31 customers with different phase labels compared to the original phase labeling.

The accuracy of the predicted phase is 100.0% after comparing to the ground truth phase labels
There are 0 incorrectly predicted customers
There are 0 customers not predicted due to missing data

In [45]:
```

Figure 1 - Output screenshot for the spectral clustering ensemble phase identification algorithm using the sample data

c. Function Descriptions

This section contains function descriptions for each function in the `CA_EnsembleFuncs.py` file.

i. *CAEnsemble*

This function implements the ensemble of Spectral Clustering for the task of phase identification task. The ensemble size is determined by the number of sliding windows available given the `windowSize` parameter. In each window, the cluster labels are returned by the spectral clustering algorithm and that clustering is then used to update a co-association matrix based on pairwise paired/unpaired information in the cluster labels. That weight matrix is then used for a final clustering into the final clusters which represent phase groupings. The original utility phase labels are not used in this function. The mapping of the final clusters to particular phases is left to a subsequent step. For more details, please see this paper: L. Blakely and M. J. Reno, "Phase Identification Using Co-Association Matrix Ensemble Clustering," IET Smart Grid, no. Machine Learning Special Issue, Jun. 2020.

Parameters

voltage: ndarray of float (measurements, customers) -
voltage timeseries for each customer. The timeseries

should be pre-processed into per-unit, difference (delta) representation. This pre-processing is an essential step.

kVector: ndarray of int - a vector of the possible values of k for the windows

kFinal: int - Number of clusters for the final clustering

custID: list of str - list of customer ids

windowSize: int - The size (in number of measurements) of the sliding window

lowWindowsThresh: int - the minimum number of windows before printing a warning that some customers had few windows due to missing data. The default value is set to 4 if this parameter is not specified.

printLowWinWarningFlag: boolean - allows suppression of the printout if customer has only a few windows in the ensemble. The default value is True. If a customer is only present in a small number of windows the co-association matrix will not be built adequately for that customer (although it will not affect other customers). Thus, results for customers with few windows should be considered low confidence predictions and likely discarded

Returns

finalClusterLabels: ndarray of int (1, customers)
array of the final cluster labels representing the phases, but they will not match in number to the actual phases. Determining which cluster number goes with which real phase is left for a future step. This parameter is one that depends on the topology of the feeder. For more discussion see the paper by B.D. Pena listed above. Starting values to try for this parameter might be 4 or 7, topologies with voltage regulators in the feeder may require a larger number of final clusters.

noVotesIndex: list of int - list of customer indices that did not receive any votes (i.e. were removed from all windows). This occurs due to missing data for a customer. If all windows for that customer contain missing data, then that customer will be eliminated from the analysis.

noVotesIDs: list of str - list of customer ids that did not receive any votes (i.e. were removed from all windows due to missing data)

clusteredIDs: list of str (customers) - list of customers IDs that were clustered during the ensemble. The length of clusteredIDs plus the length of noVotesIDs should equal the total number of customers

custWindowCounts: ndarray of int (customers) - the count, for each customer, of the number of windows that were included in the analysis, i.e. the number of windows that

were not excluded due to missing data. This count is significantly affected by the value chosen for the `windowSize` parameter. Customers with a low number of windows in the ensemble should be considered low confidence in the final prediction as they will not populate the co-association matrix properly.

ii. *SPClustering*

This function takes a window of timeseries data for the total number of customers and the number of desired clusters and performs the spectral clustering algorithm on that data, returning the cluster labels for each customer. This is the internal spectral clustering function which is called for each window (and each value in `kVector`). These results are used to build the co-association matrix. The kernel function has been hardcoded here to be the Radial Basis Function ('rbf') based on the results of this research.

Parameters

`features`: ndarray of float (customers, measurements) - a 'window' of time series measurements where customers with missing data are removed. Any NaN values in this matrix will cause the Spectral Clustering function to fail.
`k`: int - Number of clusters

Returns

`clusterLabels`: list of int - The resulting cluster label of each customer (1-k)

iii. *SPClustering_Precomp*

This function takes a precomputed affinity matrix, in the form of a co-association matrix generated by `CAEnsemble` and will use that to construct the final clusters representing the three phases.

Parameters

`aggWM`: ndarray of float, shape (customers, customers) affinity matrix of paired/unpaired weights aggregated over all available windows.
`kFinal`: int - the number of final clusters. This parameter should be set based on the feeder topology. Setting this parameter to 4 or 7 is a good place to start. If the feeder in question has voltage regulating devices a larger number of final clusters may be required.

Returns

`clusterLabels`: list of int - The resulting cluster label of each customer (1-k)

2. Sensor-based Method

The code for the sensor-based phase identification method is broken into three files:

SensorMethod_Funcs.py, SensorMethod_SampleScript.py, and PhaseIdent_Utils.py. Sensor_Method_Funcs.py contains the primary functions for the sensor-based method. PhaseIdent_Utils.py contains helper functions. SensorMethod_SampleScript.py is the place to start; the sample data will load automatically and run the algorithm.

For more details on this method, please see [2].

a. Overview

The sensor-based phase identification method takes voltage timeseries from AMI meters and sensors located on the medium-voltage distribution system (this research used IntelliRupters®). Based on the known phases of the medium-voltage sensors located around the feeder, the phase of each AMI meter is determined based on the correlations to the phase voltage measurements from the other sensors. The voltage timeseries should be pre-processed into a per-unit representation and then converted to a change in voltage timeseries by taking the difference of adjacent measurements. A window ensemble approach is employed where intervals of data, specified by the windowSize parameter, are taken independently. Correlation coefficients are calculated between each customer and each sensor data stream (3 data streams per sensor, one for each phase). Any customers with missing data during a window are excluded from consideration during that window. The other parameter to set is the CC Separation Filter threshold (CCSepThresh); this parameter filters the correlation coefficients produced by individual windows using the Correlation Coefficient Separation Score. Using a window size of 96 and a CC Separation filter value of 0.06 might be a reasonable place to start. Ranges of 96-384 for the window size and 0.02 – 0.06 for the CC Separation filter appear to be reasonable choices for the data we've tested. Once all available data has been used, the mean of the correlation coefficients is taken. The highest correlated sensors with each customer then vote on the predicted phase for each customer. The number of votes should be determined by the number of available sensors on the feeder and other considerations such as the number of voltage regulation devices in the system. Our work uses 5 votes. Finally, confidence metrics are calculated for each customer to give an indication of confidence in the phase prediction.

b. Output Description

The primary output of the sensor-based phase identification method is the predictedPhaseLabels field. This will contain the phase labels for each customer (excluding those omitted due to missing data) which were assigned by the sensor-based method. Figure 2 shows the output produced by running the SensorMethod_SampleScript.py using the included sample data. The predictions are compared both to the original utility labeling and the ground-truth phase labeling.

```
Sensor-based Phase Identification Results

Results compared to the original phase labels:
There were 31 customers whose predicted phase labels are different from the original phase labels
After filtering using confidence scores, there are 31 customers with different phase labels

Results compared to the ground truth phase labels:
There are 0 customers with incorrect phase labels
The accuracy of the predicted labels compared to the ground truth is 100.0%

In [32]:
```

Figure 2 - Output screenshot for the sensor-based phase identification method using the sample data

c. Function Descriptions

i. *AssignPhasesUsingSensors*

This function takes customer voltage timeseries and voltage timeseries from other sensors and assigns a phase label to the customer based on votes from the highest correlated sensors. For more details, see the paper listed above.

Parameters

voltageCust: ndarray of float (measurements, customers)
AMI voltage timeseries for each customer. Each column corresponds to a single customers timeseries. This data should be in per-unit, difference (delta) representation. That preprocessing is a critical step.

voltageSens: ndarray of float (measurements, sensors*phases*datastreams)
voltage timeseries for the sensor datastream. Each column corresponds to a sensor datastream timeseries. This data should be in per-unit, difference (delta) representation. The measurements dimension should match in length and measurement interval with voltageCust

custIDInput: list of str - the list of customer IDs. The length and indexing should match with the customer dimension (axis 1) of voltageCust

sensIDInput: list of str - the list of sensor IDs. Each sensor will likely have measurements for all three phases, so this list will have repeating groups in that case. The length and indexing should match axis 1 of voltageSens

phaseLabelsSens: ndarray of int (1, sensors*3) - the phase labels for each sensor. This assumes that each sensor will have measurements for all three phases.

windowSize: int - the number of samples to use in each window

numVotes: int - the number of sensors to use in making the phase prediction for each customer. The default for this parameter is 5. Make sure this number is less than or equal to the total number of sensors available.

dropLowCCSepFlag: boolean - If true this flag calls the function to drop CC values where the CC Separation is below the ccSepThresh value. The default value is False, mean all CC values are considered.

ccSepThresh: float - if dropLowCCSepFlag is set to True, then this parameter must also be passed. CC values in each window, where the CC Separation is below this value are dropped.

minWindowThreshold: int - the minimum number of windows that must be available for each customer. If a customer has fewer windows then they are omitted from the analysis.

Returns

newPhaseLabels: ndarray of int (1, customers) - the

predicted phase labels for each customer
 ccMatrix: ndarray of float (customers, customers) - the final, median, correlation coefficients for all customers over all windows
 custIDFound: list of str - the customer IDs for customers which were predicted, i.e. not eliminated due to missing data. This list will match the dimensions of all other outputs of this function
 noVotesIndex: list of int - the indices of customers who were removed from all windows
 noVotesIDs: list of str - the customer IDs of customers who were removed from all windows
 omittedCust: dict - two keys minWindows and missDataOrFiltered
 minWindows is a list of customer IDs which were omitted from the analysis because they did not meet the minimum number of windows, and missDataOrFiltered is a list of customer IDs which were omitted due, either to missing data or being filtered due to the CC Separation filter
 sensVotesConfScore: list of float - each entry is the percentage of sensors which agree on the phase prediction for that customer
 winVotesConfScore: list of float - each entry is the percentage of windows which have the same phase vote for each customer. This is calculated by considering the phase of the sensor datastream with the highest CC in each window as a vote
 ccSeparation: list of float - each entry is the difference between the highest CC and the next highest CC, considering the mean CC across windows.
 confScoreCombined: list of float - the combination of the sensVotes and the ccSeparation. Obtained by multiplying those scores together
 custWindowCounts: ndarray of int (customers) - the count, for each customer, of the number of windows that were included in the analysis, i.e. the number of windows that were not excluded due to missing data. This count is significantly affected by the value chosen for the windowSize parameter

ii. *CCSensVoting*

This function takes a row of correlation coefficients for one customer and uses the highest correlated datastreams to vote on the predicted phase based on this row of CC. If all the votes come from different sensors (as they should) then the function takes the mode of votes. If a sensor has repeated votes, this indicates more than one phase of the sensor is highly correlated with this customer. This sometimes occurs with sensors near the substation but is undesirable. In this case, those sensors are eliminated from voting, and only the remaining sensors, in the numVotes highest correlated sensors, are used for the prediction.

Parameters

ccRow: ndarray of float (numSensors) - the correlation coefficients between one customer and the set of sensors

numVotes: int - the number of votes to use in the prediction
phaseLabelsSens: ndarray of int (1, numSensors) - the phase labels for each sensor. The dimensions should match the length of ccRow
sensID: list of str - the list of sensors IDs. The length of this should match the dimensions of ccRow and phaseLabelsSens

Returns

phasePrediction: int - the phase predictions for this customer
votes: list of int - the votes that led to this prediction
voteIndices: list of int - the indices of the sensors used in the phase prediction
voteIDs: list of str - the sensor IDs contributing to the votes

iii. *CalcConfidenceScores4Sensors*

This function calculates 4 per-customer confidence scores for the Sensor/IntelliRupter phase identification method. 1. sensVotes is the percentage of sensors that agree on the predicted phase for the customers. Only the sensors included in the votes are considered. 2. winVotes is the percentage of windows that agree on the predicted phase. 3. ccSeparation is the difference between the CC on the predicted phase versus the next highest CC. 4. CombConfScore is the winVotes and the sensVotes multiplied together. Note that these scores measure the consistency of the method across windows and do not necessarily imply accuracy. Note that if any customers were not predicted due to missing data (entries in noVotesIDs) all of the confidence metrics will be recorded as 0 for those customers. If sensPhasesInput only has three entries, the function will omit the sensor agreement and combined score metrics, assuming that the substation was used instead of the sensors.

Parameters

ccMatrixAllWindows: ndarray of float (num customers, number of sensors, number of windows) - the correlation coefficient values between a particular customer and a set of sensors. The indexing of axis 0 must match the indexing for custIDList. The indexing of axis 1 must match the indexing for sensIDInput
meanCCMatrix: ndarray of float (num customers, number of sensors) - The mean correlation coefficients over all windows. If any customers were eliminated due to missing data, then the whole column will be NaN
custIDList: list of str - the list of customer IDs. This list must match the indexing for ccMatrixInput axis 0
sensIDInput: list of str - the list of sensors IDs. This list must match the indexing for ccMatrixInput axis 1
sensPhasesInput: ndarray of int (1, sensor datastreams) - the phase labels for each sensor datastream
noVotesIDs: list of str - the list of customers which were excluded from the phase prediction due to missing data
numVotes: int - the number of sensor votes included in the phase prediction
allWindowVotes: ndarray of int (customers, windows) - the phase vote for each customer in each window. Windows where the customer was

omitted contain -999

allSensVotes: list of ndarrays of int - the sensor votes for each customer

The length of the main list is the number of customers. Each list item is the sensor votes from the highest correlated sensors, up to numVotes sensors. The length of these arrays may vary if some sensors were excluded from voting.

analysis

newPhasLabels: ndarray of int (1, customers) - the predicted phase labels for each customer

Returns

sensVotesConfScore: list of float - each entry is the percentage of sensors which agree on the phase prediction for that customer. This may be 0 if sensPhasesInput had 3 entries, i.e. in the substation case

winVotesConfScore: list of float - each entry is the percentage of windows which have the same phase vote for each customer. This is calculated by considering the phase of the sensor datastream with the highest CC in each window as a vote

ccSeparation: list of float - each entry is the difference between the highest CC and the next highest CC, considering the mean CC across windows.

confScoreCombined: list of float - the combination of the sensVotes and the ccSeparation. Obtained by multiplying those scores together. This may be 0 if sensPhasesInput had 3 entries, i.e. in the substation case.

numWindows: list of int - the number of windows included in each customers window voting score

iv. *AssignPhasesUsingSubstation*

Note this function is not included in the sample script. It is simply included as a comparison algorithm if desired.

This function takes customer voltage timeseries and voltage timeseries from the substation and assigns a phase label to the customer based on the highest correlation with one of the substation phases. This is a comparison method to the sensor-based method

Parameters

voltageCust: ndarray of float (measurements, customers)
full-length voltage timeseries for each customer. This should be in per-unit and delta voltage form

voltageSub: ndarray of float (measurements, phases)
full-length voltage profiles for each substation phase. This should be in per-unit and in delta voltage form.

custIDInput: list of str - the list of customer IDs

subIDInput: list of str - the list of substation IDs. This

will likely be the substation name repeated three times.
This is necessary for the confidence score function
phaseLabelsSub: ndarray of int (1, num phases) - the phase labels for each of the substation datastreams
windowSize: int - the number of samples to use in each window
Returns

ccMatrixSub: ndarray of float (customers, numPhases) - the final, median, correlation coefficients for all customers over all windows
custIDUsed: list of str - the list of customer IDs which were predicted. If no customers were lost due to missing data this will match custIDInput.
noVotesIndex: list of int - the indices of customers who were removed from all windows
noVotesIDs: list of str - the customer IDs of customers who were removed from all windows
predictedPhases: ndarray of int (1, customers) - the predicted phase labels based on correlation with the substation
winVotesConfScore: list of float - each entry is the percentage of windows which have the same phase vote for each customer. This is calculated by considering the phase of the sensor datastream with the highest CC in each window as a vote
ccSeparation: list of float - each entry is the difference between the highest CC and the next highest CC, considering the mean CC across windows.
""""

III. Meter to Transformer Pairing

The code that implements the meter to transformer pairing task is broken into three files: M2TFuncs.py, M2Utils.py, and MeterToTransPairingScripts.py. M2TFuncs.py implements the primary functions for the methodology. M2Utils.py implements helper functions. MeterToTransPairingScripts.py is the place to start; this file will load the sample data and run the algorithm.

For more details on this method, please see [3].

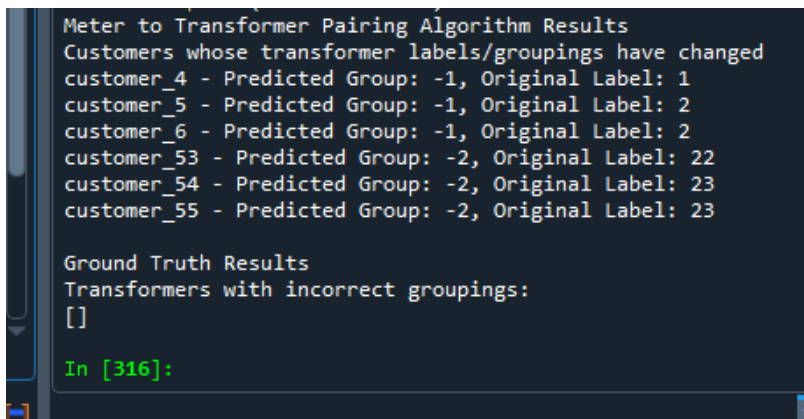
1. Overview

This method uses correlation coefficients analysis of the customer voltages to flag transformer groupings which likely contain errors, and then a linear regression methodology using voltage, real power, and reactive power data is used to correctly group the customers by service transformer. The input to the meter to transformer pairing algorithm is voltage magnitude, real power, and reactive power timeseries AMI data. The algorithm is divided into two stages. In the first stage, pairwise correlation coefficients are calculated between all pairs of customers. Then, the original transformer labels are used to inspect the correlation coefficients for each transformer grouping. If any of the pairwise correlation coefficients are below a specified threshold then the transformer is flagged for inspection in the second stage of the algorithm. Stage 1 is done using a ranked approach with multiple thresholds such that stage 1 results in a ranked list of flagged transformers, so that the

earlier in the list the transformer appears, the worse the correlation coefficients were in that transformer grouping. In stage 2, a pairwise linear regression is done between all customers. This produces a mean-squared-error (MSE) value that functions as a type of goodness-of-fit metric for the regression, a resistance coefficient, and a reactance coefficient. The resistance and reactance values function as a type of distance matrix between customers. The MSE values and are used as a filter for the reactance matrix, where pairs with high MSE are discarded. The algorithm currently sets the MSE threshold by finding the minimum MSE value and adding a small amount to that value. This parameter could also be set manually if desired. The resulting, filtered, reactance matrix is used to assign new transformer groups to the flagged transformers/customers from Stage 1. Customer pairs serviced by the same transformer will have reactance 'distances' lower than the reactance due to the influence of two transformers.

2. Output Description

The primary output of the meter to transformer pairing algorithm is the predictedTransLabels field returned by the CorrectFlaggedTransErrors function. This field contains a list of transformer labels, unchanged labels remaining the same, and new transformer groupings labeled with negative integers. There is not a straightforward way to map the new groupings (designated with negative integers) to physical transformers; that is left as a subsequent task. The sample data produces output as shown in Figure 3. The two customers whose labels were changed (customer_4 and customer_53) are in new transformer groups with the other customers serviced by their respective transformers. You can see this by comparing the results in predictedTransLabels to the ground-truth transformer labels in transLabelsTrue variable. The 'Transformers with incorrect groupings' list is empty because the algorithm was 100% successful in finding the correct transformer groupings. Any transformers whose groupings are incorrect when compared to the ground-truth labels would be listed here.



```
Meter to Transformer Pairing Algorithm Results
Customers whose transformer labels/groupings have changed
customer_4 - Predicted Group: -1, Original Label: 1
customer_5 - Predicted Group: -1, Original Label: 2
customer_6 - Predicted Group: -1, Original Label: 2
customer_53 - Predicted Group: -2, Original Label: 22
customer_54 - Predicted Group: -2, Original Label: 23
customer_55 - Predicted Group: -2, Original Label: 23

Ground Truth Results
Transformers with incorrect groupings:
[]

In [316]:
```

Figure 3 - Output screenshot for the meter to transformer pairing algorithm

3. Function Descriptions

a. RankFlaggingBySweepingThreshold

This function takes a vector of possible threshold values (probably correlation coefficients) and creates a ranked list of flagged transformers based on which had relatively lower cc values when they were flagged.

Parameters

transLabelsInput: ndarray of float (1, customers) - the transformer

label for each customer
notMemberThresholdVector: list of float - the list of possible thresholds below which customers are considered not on the same transformer
ccMatrix: ndarray of float (customers, customers) - the array of pairwise correlation coefficients.

Returns

allflaggedTrans: list of list of int - each entry in the list contains the list of flagged transformers for that threshold value. This allows identification of the threshold value responsible for flagging each transformer
allNumFlagged: list of int - the number of flagged transformers for each threshold value. The length of this list will match the length of notMemberThresholdVector
rankedFlaggedTrans: list of int - a ranked list containing all flagged transformers over all threshold values. Values at the beginning of the list had relatively lower cc threshold values when they were flagged.
rankedTransThresholds: list of float - a list of thresholds that correspond in index to rankedFlaggedTrans denoting which CC threshold flagged that transformer

b. CCTransErrIdent

This function uses the correlation coefficients to detect errors in transformer groupings. This is simply used to flag customers as potential errors.

Parameters

transLabelsInput: ndarray of float (1, customers) - the transformer label for each customer
notMemberThreshold: float - the threshold below which customers are considered not on the same transformer
ccMatrix: ndarray of float (customers, customers) - the array of pairwise correlation coefficients.

Returns

flaggedCust: list of int - the list of indices of the flagged customers

c. AdjustDistFromThreshold

Uses the specified threshold of the comparison matrix to replace values in a second matrix. Values less than 0 in the matrix2Adjust are replaced. For example, the way this is usually used is if the comparison matrix is the MSE matrix and the threshold is 0.2, then for any cell in the MSE matrix with a higher MSE than 0.2, the corresponding cell in the second matrix, for example, the reactance distance matrix, is set to the replacement

value which is usually a very high value. Effectively this function filters a matrix, like the reactance distance matrix, by high values in the MSE matrix. Values less than 0 in the reactance distance matrix are known to be bad values.

Parameters

compMatrix: ndarray of float (1, customers) - the matrix containing the values used as a threshold to remove values from matrix2Adjust. This matrix must be a 'distance'-type matrix. Often this is the pairwise MSE matrix containing the mean-squared error results from a pairwise regression formulation

matrix2Adjust: ndarray of float (1, customers) - the second matrix that will have values replaced.

threshold: float - value used to flag values in the comparison matrix above the threshold for which the corresponding cells in matrix2Adjust will be replaced by a given replacement value.

replacementValue: float - value used to replace flagged cells in matrix2Adjust
In the meter to transformer pairing task this is often set to the max value in the matrix2Adjust field, so that the value is discarded, but remains within the range of the variable

Returns

adjustedMatrix: ndarray of float (1, customers) - the final matrix adjusted by replacing the cells in matrix2Adjust corresponding with the thresholded cells from compMatrix with a given replacement value.

d. [CorrectFlaggedTransErrors](#)

This function takes a list of flagged transformers and produces a list of predicted customer labeling errors and a prediction for the correct labeling

Parameters

flaggedTrans: list of flagged transformers

transLabelsInput: ndarray of float (1, customers) - the transformer label for each customer

custIDInput: list of str - the customer ID's

ccMatrix: ndarray of float (customers, customers) - the matrix of pairwise correlation coefficients

notMemberThreshold: float - the threshold for flagging customers as not being connected to the same transformer as another customer

mseMatrix: ndarray of float (customers, customers) - the mse values from a pairwise linear regression

xDistAdjusted: ndarray of float (customers, customers) - the reactance distance, adjusted by the r-squared values

reactanceThreshold: float - the threshold for considering a customer to be the only customer on a transformer. The default value of 0.046 which was determined as an average value for reactance across two transformers.

Returns

predictedTransLabels: ndarray of int (1, customers) - the predicted transformer labels for each customer. 'New' labels are given by negative labels. Those should reflect correct transformer groupings but not the original labels themselves.

allChangedIndices: list of lists of int - each entry in the list is the list of indices which changed under each successive flagged transformer

allChangedOrgTrans: list of lists of int - each entry in the list is the list of the original transformer label for customers which changed their label

allChangedpredTrans: list of lists of int - each entry in the list is the list of the predicted transformer label for each customer which changed their label. 'New' transformer labels will be negative. Thus, the customer groupings should match physical transformers but the utility label is to be determined.

IV. Sample Data

1. Overview

The sample data included in this release consists of timeseries data for 200 single-phase customers, 10 sensors, and the substation at 15-minute intervals with a total of 11,520 measurement points, or approximately four months. The customer advanced metering infrastructure (AMI) data, as well as the sensor and substation, consist of voltage magnitude, real power, and reactive power timeseries. The voltage magnitude data is in Volts and the real power and reactive power are in Watts and Var, respectively. There are also phase labels, transformer labels, and ID's for each AMI and sensor. The data is included as .npy files which is the numpy data format. More information on the file format can be found here, <https://numpy.org/devdocs/reference/generated/numpy.lib.format.html>

The data was synthetically generated using real load profiles from customers in the southern U.S. which was then placed on EPRI Ckt. 5. A quasi-static time series (QSTS) simulation was run using Open DSS was to produce voltage magnitudes and reactive power profiles for the AMI meters, sensors, and substation.

2. Injected Data Issues

The sample data includes several data issues that have been injected into the original data files. These are measurement noise, missing data, incorrect phase labels, and incorrect transformer labels.

a. Measurement Noise

For the voltage data, normally distributed noise was injected with a specified standard deviation. For the AMI data, the standard deviation was set to 0.07% of the mean (240V in this case). This value was determined

using a meter testing report provided by one of our utility partners. 0.07% was the demonstrated standard deviation of noise in the 0.5 meter class they were testing. Using 0.07% standard deviation also ensures that 99% of the noise is within 0.2% which is the maximum deviation allowed by the 0.2 meter class. For the sensors, a standard deviation of 0.04% was used. This value was determined via analysis of the sensor data provided by our utility partners. For the real and reactive power, small amounts of uniformly distributed noise were added to the measurements, up to a maximum of +/- 100W and 100VAR respectively.

b. Missing Data

Approximately 0.2% missing data was injected into the voltage, real power, and reactive power AMI timeseries. No missing data was injected into the sensor data. We assumed that all measurements would be missing simultaneously from the AMI meter, thus the missing data is in the same locations between all three data streams on a given meter. A minimum missing interval (1 datapoint) and a maximum missing interval (48 datapoints) was specified, and missing data was injected into each customer independently.

c. Incorrect Phase Labels

Two phase label files are included for the AMI data, one contains the ground-truth phase labels and the other contains a set of phase labels where 18 customers have been given an incorrect phase label. The set of phase labels with errors included is intended to represent a set of labels for the utility model that contains some unknown amount of errors. The phase labels for the sensor and substation data streams are the ground-truth phase labels; no error injection was done to those labels.

d. Incorrect Transformer Labels

There are two transformer label files included for the AMI data, one contains the ground-truth transformer labels and the other contains a set of phase labels where two selected customers have been given an incorrect transformer label. These are as follows: customer_3 transformer was changed from 1 to 2 and customer_53 transformer was changed from 23 to 22. Thus, there are two customers with incorrect transformer labels, but there are 4 transformers whose customer groups are now incorrect. Transformers 1, 2, 23, and 22 have incorrect groups. Transformers 1 and 23 are missing one customer each and Transformers 2 and 22 have one additional customer each.

3. Included Data Files

CustomerIDs_AMI.npy

custIDInput: list of str - This is a list of customer IDs as strings. The length of this list should match the customer dimension of voltageInput

PhaseLabels_Sensor.npy

sensPhases: ndarray of int (1, sensor datastreams) - the phase labels of the sensors in integer form. 1 - Phase A, 2 - Phase B, 3 - Phase C. The number of sensor datastreams should match the dimensions and indexing of axis 1 in voltageInputSens

PhaseLabelsTrue_AMI.npy

phaseLabelsTrue: ndarray of int (1, customers) - This contains the ground-truth phase labels for each customer, if available. Note that, in practice this may not be available, but for testing purposes this is provided along with functions to evaluate the phase identification accuracy against the ground-truth labels.

PhaseLabelsErrors_AMI.npy

phaseLabelsErrors: ndarray of int (1, customers) - This contains the phase labels for each customer in integer form (i.e. 1 - Phase A, 2 - Phase B, 3 - Phase C). Any integer notation may be used for this field; it is only used to assign the final phase predictions. In practice, this field could even be omitted, the phase identification results from CAEnsemble will still be grouped by phase, and the assignment of final phase labels could be left for a post-processing step by the utility. The dimensions of this matrix should be (1, customers). These are assumed to be the original, utility labels, which may contain some number of errors. The sample data included with these scripts has ~9% of phase labels injected with errors. This can be seen by comparing this field with the entries in phaseLabelsTrue which contains the ground-truth phase labels

ReactivePowerData_AMI.npy

qDataInput: ndarray of float (measurements, customers) - the reactive power measurements for each customer in VAR

RealPowerData_AMI.npy

pDataInput: ndarray of float (measurements, customers) - the real power measurements for each customer in Watts

SensorsIDs.npy

sensIDs: list of str - the IDs for the sensors in string form. The length of this list should match the length and indexing of the sensor datastreams dimension of voltageInputSens and sensPhases. (The IDs will be repeated for the different phase datastreams of the same sensor)

TransformerLabelsErrors_AMI.npy

transLabelsErrors: ndarray of int (1, customers) - the transformer labels for each customer which may contain errors. In the sample data, customer_4 transformer was changed from 1 to 2 and customer_53 transformer was changed from 23 to 22

TransformerLabelsTrue_AMI.npy

transLabelsTrue: ndarray of int (1, customers) - the transformer labels for each customer as integers. This is the ground truth transformer labels

VoltageData_AMI.npy

voltageInput: ndarray of float - This matrix contains the voltage measurements (in volts) for all customers under consideration. The matrix should be in the form (measurements, customers) where each column represents

one customer AMI timeseries. It is recommended that the timeseries interval be at least 15-minute sampling, although the algorithm will still function using 30-minute or 60-minute interval sampling

VoltageData_Sensors.npy

voltageInputSens: ndarray of float (measurements, sensor datastreams) – the sensor voltage timeseries. Each column should be a sensor datastream with measurements in volts. The length of the measurement's dimension should match in length and timestamp with the voltageInputCust field. Each data stream will correspond to one measurement field and phase for a particular sensor. Currently, our work is using the average voltage field from the sensors. For example, if there are 10 sensors in the system, and the average voltage field is used, each sensor will have measurements from each of the three phases A, B, C. Thus the length of the sensor datastreams dimension will be 30. Our work utilized IntelliRupters which take measurements on either side of the device, thus it might be expedient to downselect to one of the two datastreams, as the two datastreams will either be repeated (closed devices) or measuring two different sections of the grid (open devices).

V. Code Notes

The included code was developed in Python 3.7.3. Standard libraries are noted below with the version used in the development of this code.

- numpy (1.20.2)
- pathlib (2.3.5)
- scikit-learn (0.24.2)
- matplotlib (3.3.4)
- seaborn (0.11.1)
- scipy (1.6.2)
- pickle
- datetime
- warnings
- copy
- sys

VI. Related Publications from the Project

This section contains references for other publications related to this work and produced during the course of this project.

For a discussion of parameter tuning for the spectral clustering ensemble phase identification method, please see [4].

For a direct comparison of phase identification methods, including between voltage-based methods and power-based methods, please see [5].

For an analysis of common types of errors found in distribution systems, please see [6].

For discussion on AMI data quality issues and requirements, please see [7], [8].

For work on estimating distribution system parameters such as wire type and length, please see [9].

For work on locating and estimating parameters for behind-the-meter PV installations, please see [10]–[12].

VII. Acknowledgments

Additional support for this work was provided by Matthew J. Reno and Bethany Pena at Sandia National Laboratories. Thank you to them for their contributions.

This material is based upon work supported by the U.S. Department of Energy’s Office of Energy Efficiency and Renewable Energy (EERE) under Solar Energy Technologies Office (SETO) Agreement Number 34226. Sandia National Laboratories is a multimission laboratory managed and operated by National Technology and Engineering Solutions of Sandia, LLC., a wholly owned subsidiary of Honeywell International, Inc., for the U.S. Department of Energy’s National Nuclear Security Administration under contract DE-NA0003525. This paper describes objective technical results and analysis. Any subjective views or opinions that might be expressed in the paper do not necessarily represent the views of the U.S. Department of Energy or the United States Government. SAND2021-13869 O

VIII. References

- [1] L. Blakely and M. J. Reno, “Phase Identification Using Co-Association Matrix Ensemble Clustering,” *IET Smart Grid*, no. Machine Learning Special Issue, Jun. 2020.
- [2] L. Blakely, M. J. Reno, B. Jones, and A. Furlani Bastos, “Leveraging Additional Sensors for Phase Identification in Systems with Voltage Regulators,” presented at the Power and Energy Conference at Illinois (PECI), Apr. 2021.
- [3] L. Blakely and M. J. Reno, “Identification and Correction of Errors in Pairing AMI Meters and Transformers,” presented at the Power and Energy Conference at Illinois (PECI), Apr. 2021.
- [4] B. D. Pena, L. Blakely, and M. J. Reno, “Parameter Tuning Analysis for Phase Identification Algorithms in Distribution System Model Calibration,” presented at the KPEC, Apr. 2021.
- [5] F. Therrien, L. Blakely, and M. J. Reno, “Assessment of Measurement-Based Phase Identification Methods,” *IEEE Open Access Journal of Power and Energy*, publication pending 2021.
- [6] Blakely, M. J. Reno, and J. Peppanen, “Identifying Common Errors in Distribution System Models,” presented at the Photovoltaic Specialists Conference (PVSC), Chicago, IL, USA, Jun. 2019.
- [7] L. Blakely, M. J. Reno, and K. Ashok, “AMI Data Quality And Collection Method Consideration for Improving the Accuracy of Distribution System Models,” presented at the IEEE Photovoltaic Specialists Conference (PVSC), Chicago, IL, USA, 2019.
- [8] K. Ashok, M. J. Reno, D. Divan, and L. Blakely, “Systematic Study of Data Requirements and AMI Capabilities for Smart Meter Connectivity Analytics,” presented at the IEEE Smart Energy Grid Engineering (SEGE), Oshawa, Ontario, Canada, 2019.
- [9] M. Lave, M. J. Reno, R. J. Broderick, and J. Peppanen, “Full-Scale Demonstration of Distribution System Parameter Estimation to Improve Low-Voltage Circuit Models,” presented at the IEEE Photovoltaic Specialists Conference (PVSC), Washington, DC, USA, 2017.

- [10] K. Mason, M. J. Reno, L. Blakely, S. Vejdani, and S. Grijalva, "A Deep Neural Network Approach for Behind-the-Meter Residential PV Size, Tilt, and Azimuth Estimation," *Solar Energy*, vol. 196, pp. 260–269, Jan. 2020.
- [11] S. Grijalva, A. U. Khan, J. S. Mbeleg, C. Gomez-Peces, M. J. Reno, and L. Blakely, "Estimation of PV Location in Distribution Systems Based on Voltage Sensitivities," presented at the NAPS, Mar. 2021.
- [12] X. Zhang and S. Grijalva, "A Data-Driven Approach for Detection and Estimation of Residential PV Installations," *IEEE Transactions on Smart Grid*, vol. 7, no. 5, pp. 2477–2485, Sep. 2016, doi: 10.1109/TSG.2016.2555906.