

Contents

[**Introduction**](about:blank)

[**Data Extraction**](about:blank)

**Data Processing**

[**Target Variables**](about:blank)

[**Predictive Variables**](about:blank)

[**Pre-modeling**](about:blank)

[**Modeling**](about:blank)

**Conclusions and Future Work**

Introduction

In this project, we aimed to build a forecasting model for individual user electricity consumption. We used data from the UCI Individual Household Electric Power Consumption dataset, which contains the electricity load of 370 users over a period of 4 years. Our ultimate goal is to forecast each user's electricity load for one year into the future, with a granularity of one day.

As detailed in this document, we explored different model architectures and feature engineering techniques in order to achieve the best possible predictive power. Through careful iteration and experimentation, we created an accurate and efficient final model. Furthermore, the model can be scaled to include new users and new data.

The purpose of this document is to provide a detailed technical overview of:

1. Design specifications and parameters
2. Data inclusions and exclusions
3. Predictive variable creation process
4. Target variable definition
5. Iterative model building process
6. Metric evaluation process

Diagram

Description automatically generated

A brief visualization of the modeling process

Data Extraction

**Workflow**

The dataset we used for our forecasting problem is [Electricity Load Diagrams 20112014 Data Set](https://archive.ics.uci.edu/ml/datasets/ElectricityLoadDiagrams20112014) from UCI Machine Learning Repository. The data is in one CSV file from the zipped folder. With the Python Pandas package, we are able to read in the CSV file as Pandas dataframe, and convert to a Numpy array later if necessary.

Our team works locally and collaboratively (on Google Colab). The original data is stored in both local machines (permanently) and in Colab cloud (temporary, download needed every time you start the virtual machine). Due to limited RAM on Colab, we would work locally with sophisticated modeling and upload changes to Colab.

Data Processing

**Data Overview**

The dataset is in one CSV file, it contains the electricity load of 370 anonymous users recorded in kW for each 15-minute interval from 2011 to 2014.

Below are plots of electricity load from 3 random selected users.

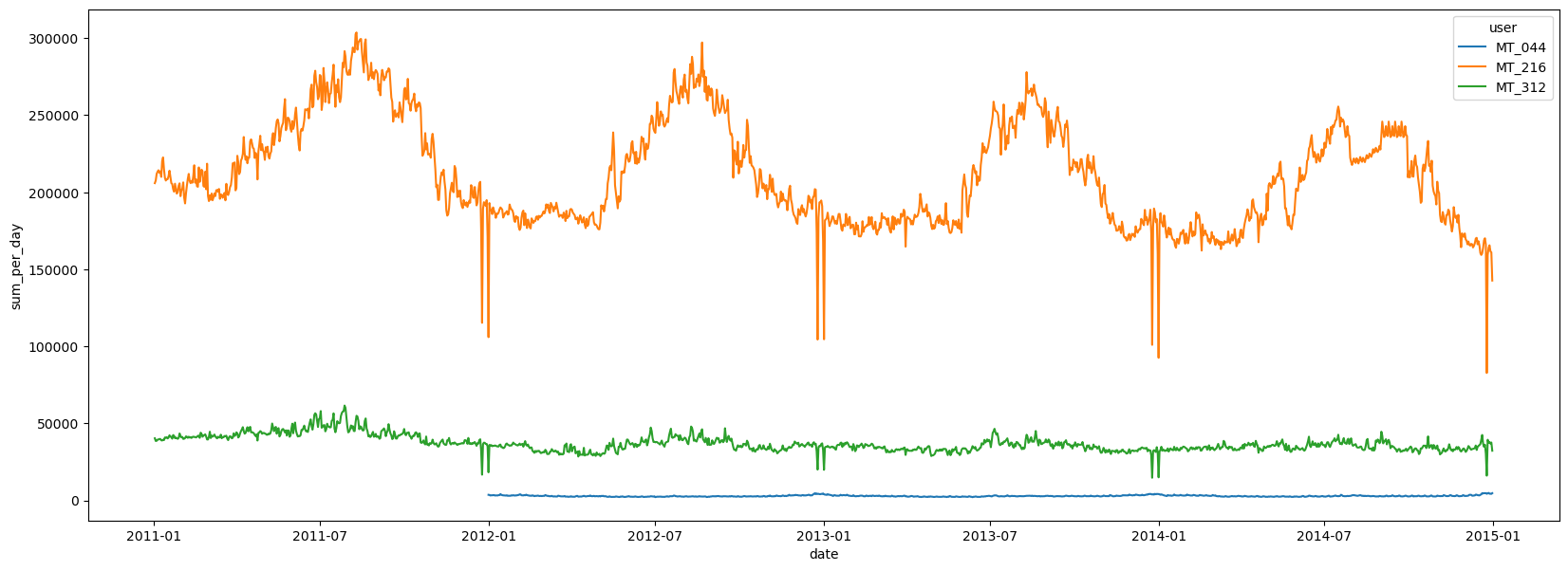


Figure 1: Random selected users MT\_044, MT\_216, MT\_311 in same scale

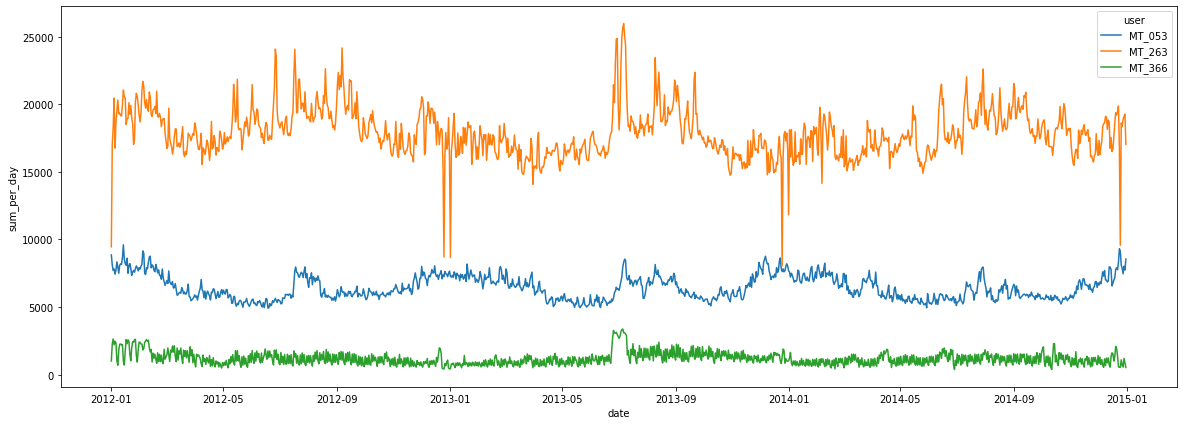


Figure 2: Random selected users MT\_053, MT\_263, MT\_366 in same scale

**Data Diagnostics**

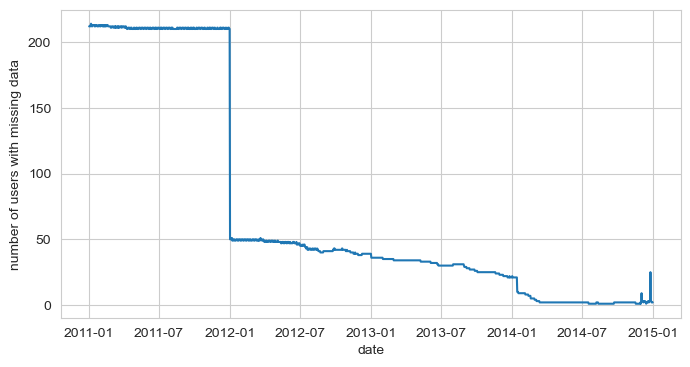
In the given CSV file, the rows span every 15 minutes from the start of 2011 to the end of 2014, and the columns span the electricity load for 370 users among 4 years. The size of data is (140256, 370). There are no missing values in this dataset, but some clients were created after 2011 and the unrecorded data are padded with zero. Upon further inspection, it is highly possible that missing data is recorded as 0 within the data – not much context is given regarding the dataset. According to daylight saving, March time change days have only 23 hours and the values between 1:00 am and 2:00 am are zeros; October/November time change days have 25 hours and the values between 1:00 am and 2:00 am aggregate the consumption of two hours.

**Modeling Data Creation**

The first step is to convert the input data to datetime timestamps (time) and numerics (electricity load). Then it is very important to melt the wide data matrix to a long data matrix so that our input data is one entry instead of one data point. Considering that this dataset is very large, we will merge the data so that the input data is aggregated from load per 15 minutes to daily load. For every user, we calculate the total electricity consumption and the standard deviation in the day. Then, we have the data in 4 years, 1461 days for all 370 users. The aggregated dataset is of shape (540570, 4), where the four columns are user, date, total consumption in the day and the standard deviation in the day respectively.

**Missing Data Analysis**

According to the description of the dataset, some users lack the data in 2011 (or even further). Therefore, we define the data as missing if the total electricity consumption in a day is 0. By counting the number of users that lack data in every day, we have the following figure

Figure 3: The number of users that lack data by date

From the figure, we can see that more than half of the users lack data in 2011 and at least 50 users also have missing data after 2012. Thus, we decided to limit our analysis to data in 2012-2014.

In addition, since some users still lack data even after 2012, we check the number of days that lack data (total usage = 0 for the whole day) for every user. Considering some accidental failures, we list the users that lack data from more than 2 days. From figure 4, we can see that 56 users lack data for more than 2 days since 2012. While users with more than 50 days of missing data are rightfully excluded from the study, we will manually check the pattern of users that have 3-50 days of missing data. We found that there are 3 of them: MT\_003, MT\_146 and MT\_288.

From figure 5, we can see that MT\_003 and MT\_288 have abnormal patterns that should be excluded from the study while MT\_146 only has a few days in the beginning of 2012. Therefore, 55 users are excluded from the study and the cleaned dataset contains 315 users. We also generated a cleaning method that dropped the leading entries for users that have consecutive zeros greater than some threshold. Those who do not have leading zeros will retain data from 2011.

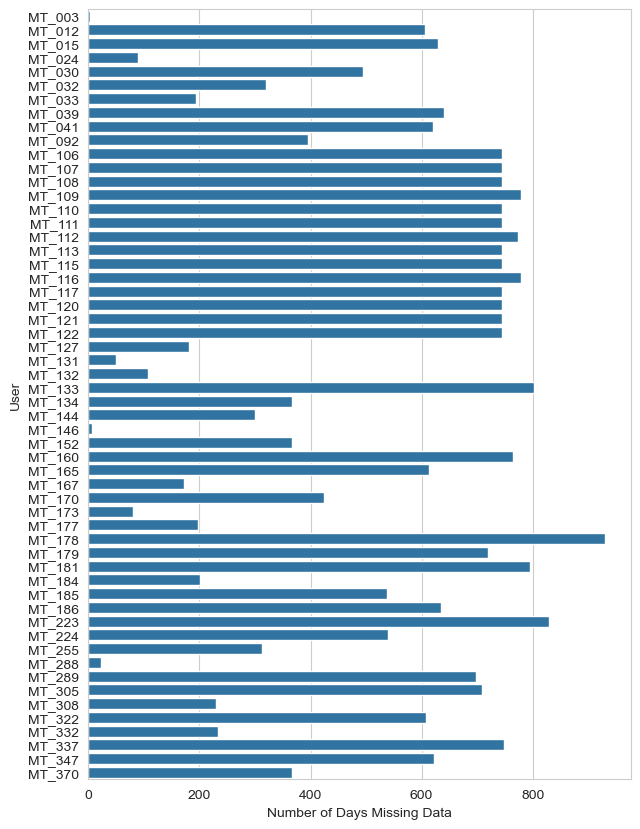


Figure 4: The number of days missing data since 2012 by user

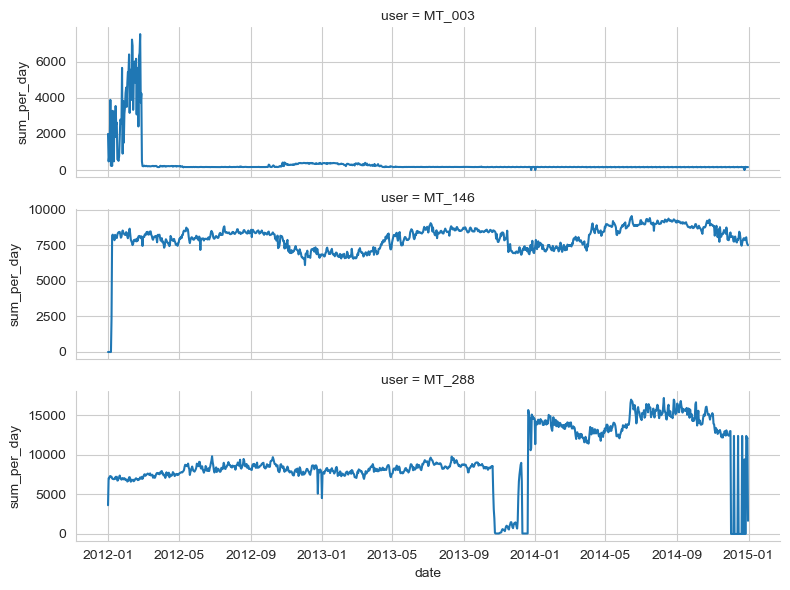


Figure 5: Electricity Usage of Suspicious User for Missing Data

The dataset given is in one, so we split the training set and testing set by roughly 75:25 with time in order (not shuffled). The test cutoff is at 2014-04-01 00:00:00 so it is the last 9 months, all entries after the cutoff are considered as the testing set.

Target Variables

There is a single variable to predict: the total electricity consumption in kW on a particular day. This is an aggregated value from the raw data, which is reported use every 15 minutes. Furthermore, to evaluate the robustness of models, we predicted the daily consumption at different time periods into the future. We made predictions at the following intervals:

* 1 day into the future (first day of the test set)
* 30 days into the future (30 days into the test set)
* 1 year into the future (last day of the test set)

While evaluating error for 30 days and 1 year, these predictions incorporate predictions into the *input* of the model.

Multi-step Predictions (Many-to-Many)

For LSTM or general RNN architecture, the model target would be a sequence of length n, with n being the number of output time steps. For example, if we are making many-to-many predictions for future 5 days starting from the past 30 days (i.e. input sequence length = 30, output sequence length = 5), then the model target would be a 5 by 1 vector denoting future 5 days’ loads. For batched training, the dimension of the target tensor would be (batch\_size, output\_seq\_length, 1).

As we applied sliding windows of stride 1 one the training data, the total amount of training data/targets would be

Custom Torch Dataset and DataLoader

To train many-to-many models, we implemented a custom dataset class. The dataset class defines the interface that PyTorch's dataloader class will use to load data. Each item in the dataset is a tuple of (sequential\_featrue\_tensor, user\_feature\_tensor, target\_tensor). The sequential feature tensor has the shape of (input\_seq\_len, feature\_dim), and target tensor has the shape of (output\_seq\_len, 1). User feature tensor (input\_seq\_len, 1) contains the user id repeated for input\_seq\_len times.

The custom torch dataset makes sure that the data is correctly formatted so that each training sample only contains one user. This is important for RNN models, which assume that the input is a sequence.

Predictive Variables

During our analysis, we categorized predictive variables into two categories:

1. Direct variables – These variables were directly from the dataset with no change.
2. Derived variables – These variables were created by manipulating the direct variables.

The majority of our features were derived or engineered from the original data, which was relatively straightforward. The only information included was user, datetime, and electric load at that time. Our derived variables aim to capture variance contained in lagged data values as well as any seasonality related to time.

**Variable List**

**Direct:**

1. user – some particular customer associated with the energy use.
2. load\_per\_15min – the amount of electricity used in a 15 minute interval. Note – this variable is not included in any input as it is aggregated in load\_per\_day.

**Derived:**

1. load\_per\_day – the amount of electricity used in a 1-day/24-hour interval, which is the sum of load\_per\_15min for each day.
2. feat\_one\_day\_lag – the amount of electricity one day before load\_per\_day.
3. feat\_one\_week\_lag – the amount of electricity one week before load\_per\_day.
4. feat\_one\_month\_lag – the amount of electricity one month (30 days) before load\_per\_day.
5. feat\_one\_year\_lag – the amount of electricity one year (365 days) before load\_per\_day.
6. feat\_one\_week\_mean – the average load in the window of the past week before the predicted day.
7. feat\_one\_month\_mean – the average load in the window of the past month (30 days) before the predicted day.
8. feat\_one\_week\_std – the standard deviation in the window of the past week.
9. feat\_one\_month\_std – the average load in the window of the past month (30 days).
10. feat\_year – the year of the timestamp associated with load\_per\_day.
11. feat\_month – the month of the timestamp associated with load\_per\_day.
12. feat\_day – the day of the timestamp associated with load\_per\_day.
13. feat\_day\_of\_week – the day of the timestamp associated with load\_per\_day, ex. Mon -> 0, Sun -> 6.

Pre-modeling

**Training-Testing Split**

With the data in a format that can be used for modeling, we then split it into training and testing sets (as well as validation for tuning). This is important to avoid overfitting, or building a model that perfectly predicts the training data but does not perform well on new, unseen data. We split the data along years, with the years 2011-2013 from the data going into the training set and 2014 going into the testing set. No splitting was done for users, meaning that all users' data were used in both the training and test sets.

As previously mentioned, we utilized a validation subset. We broke the training (or development) set into smaller pieces iteratively as training and validation sets, to conduct temporal cross-validation for model selection.

**Model Clustering**

With a diverse dataset where every user may have different trends, we clustered all users that had similar trends and fit multiple models for each cluster. The clustering method we used is Dynamic Time Warping distance instead of regular Euclidean distance with K-Means.

Since the dataset is huge, we used PCA to create a smaller dataset for finding the best number of clusters from the range [2,6]. We were able to reduce from 731 columns to 196 and capture 99% of variance.

By looking at the Figure 6, we can see that k=3 is the best cluster number for clustering as all clusters passed the average silhouette score; however, the clusters are not balanced (approximate ratio 1:2:4). *Maybe the next group can try to improve this part.* Then we can fit the original dataset (731 columns) to DTW K-Means clustering method for labeling users.



Figure 6: Silhouette Analysis plot on n\_clusters = 3

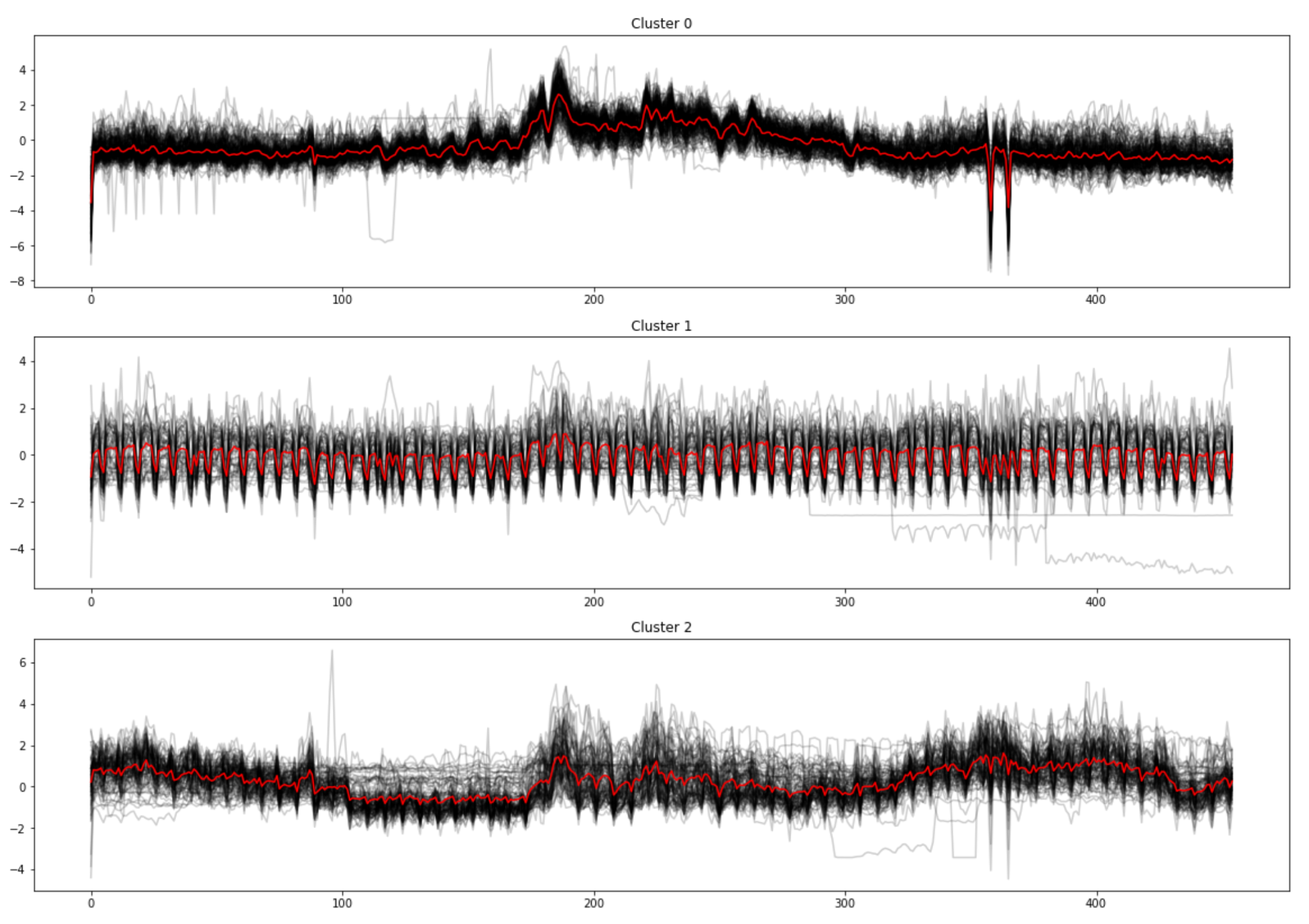


Figure 7: red – cluster center, black – user data in the same cluster

**Feature Scaling**

Standardization was used on both the covariate space and target variable for models that this is relevant (all but tree-based methods). This was done for two primary reasons: first, to account for differences in the scale of different features, which could lead to some features having disproportionately large impact on the model simply due to their scale; and second, to speed up training by putting all features on the same scale. To standardize, we fit the StandardScaler from sklearn.preprocessing on the training data only, and then used those fitted parameters to standardize both the training and testing data.

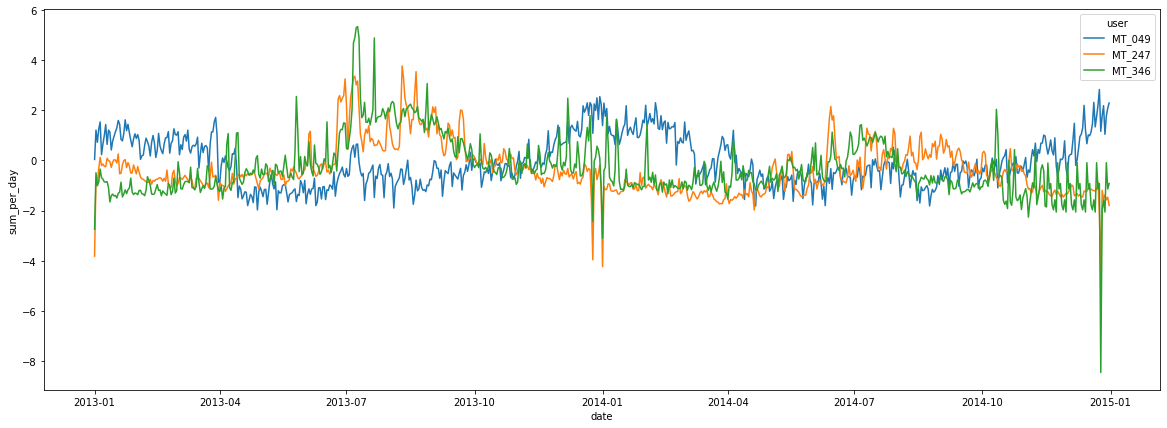


Figure 8: Random selected users MT\_049, MT\_247, MT\_346 after normalization

**Exploring Models**

**Future Training/Testing Pipeline**

We will first conduct a clustering that groups similar users together based on features derived from both temporal and spectral space. After that, we will train models for each cluster and further conduct cross-validation for model selection. To evaluate the model on the test set, we first find the cluster that the user belongs to, and then use the pre-trained model within that cluster to make a prediction.

**Metric**

The metric we used for performance evaluation was Mean Absolute Percentage Error (MAPE), but also considered other options such as Mean Squared Error (MSE) and Root Mean Squared Logarithmic Error (RMSLE). These can be good at understanding the overall performance for the time period, while MAPE is better at understanding the day to day performance.

Modeling

**Introduction**

Our team explored a variety of models. For our prediction problems, we considered using OLS, Gradient Boosting (XGBoost), and also a toy PyTorch model. Due to the size of our data (50M rows), we conducted cross validation or hyperparameter tuning only for our chosen final model. The 1-day into the future prediction problem was the simplest among the three, so we did most of our initial modeling experiments along this timeframe. Explicitly this means that inputs in the test set were original and true values. We used the accuracy of this task to benchmark for the more difficult prediction tasks, 30-days ahead and 1-year ahead.

**Algorithmic Solution Design**

| **Target** | **Category** | **Algorithms/Approach** |
| --- | --- | --- |
| 1-day electricity load ahead | 1-step prediction | OLS  Prophet  Random Forest  Gradient Boosting  Neural Net |
| 30-day forward prediction | 1-step prediction  N-steps prediction/forecasting | Prophet  Gradient Boosting (rolling)  Neural Net |
| 9-month/1-year forward prediction | N-steps prediction/forecasting | Prophet  Gradient Boosting (rolling)  Neural Net |

**Model Results**

1. OLS

Our baseline model is a standard OLS regression, regressing the next-hour electricity load on all derived features. This model is the simplest and fastest to train, and can be used as a baseline to compare the performance of other models.

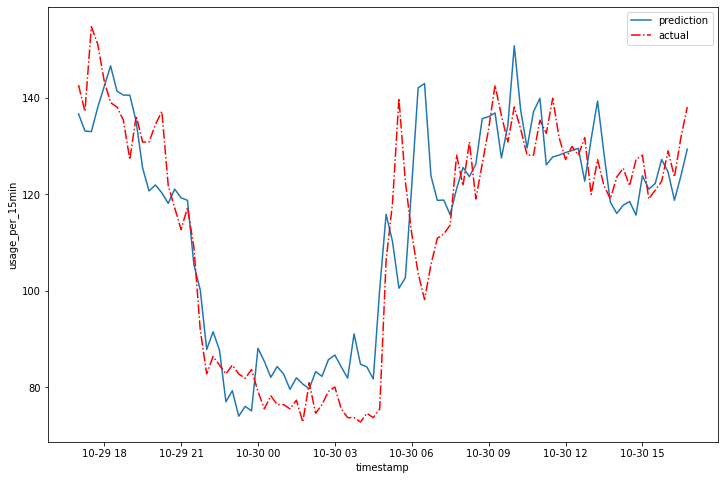


Figure 9: 1-hour prediction for 24 hrs on user MT\_300 with vanilla OLS regression model

The above figure demonstrates the performance of our linear regression model on the test set (user: MT\_300, 1-hour prediction for 24hrs). We can see that the model is able to capture the general trend of the load, but struggles with predicting spikes. Given that the model is only required to predict the next-hour load, it is not too surprising that the simple linear model is able to predict the general trend in data. The absolute residuals of training & testing sets are plotted below:

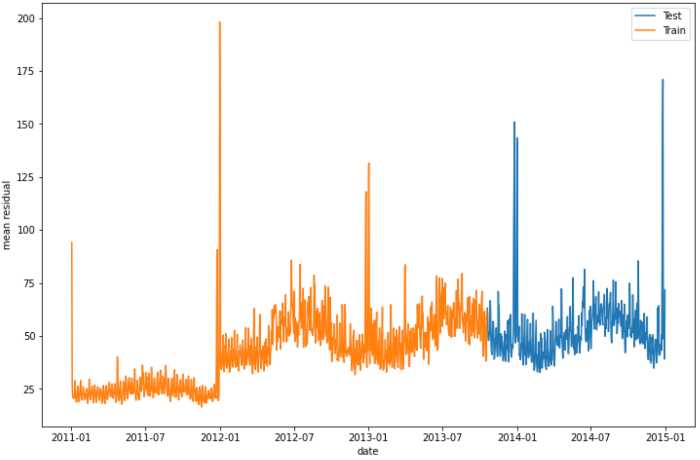


Figure 10: Absolute residuals of training & testing with OLS on hourly prediction

For OLS we also fitted on daily data with multiple models by clusters, the performance is below. There is an overfitting sign in Cluster 0 as the Testing MAPE is much higher than Training MAPE, but not in Cluster 1 and Cluster 2.

| Methods: OLS | **Training Set Performance** | **Testing Set Performance** |
| --- | --- | --- |
| Multi Model Cluster 0, MAPE | 1.6378 | 2.7999 |
| Multi Model Cluster 1, MAPE | 1.5365 | 1.2923 |
| Multi Model Cluster 2, MAPE | 2.2906 | 2.2560 |

Let’s look at one user to see the improvement with clustering. From Figures below, we can see that with clustering, there is less noise when predicting trends. Looking at other users in different clusters, we can see that clustering has positive effect on performance.

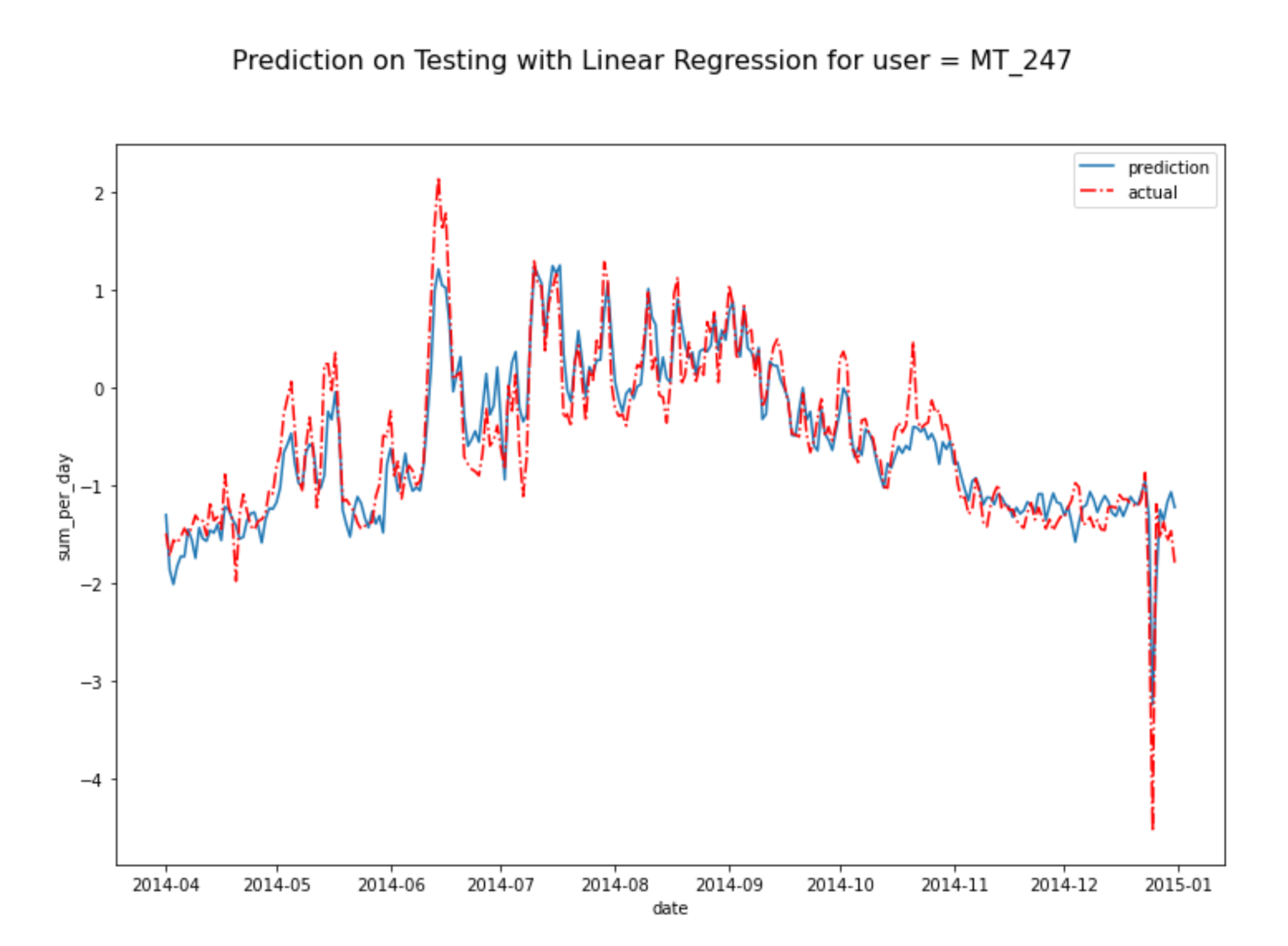


Figure 11: Prediction on user MT\_247 for single model OLS

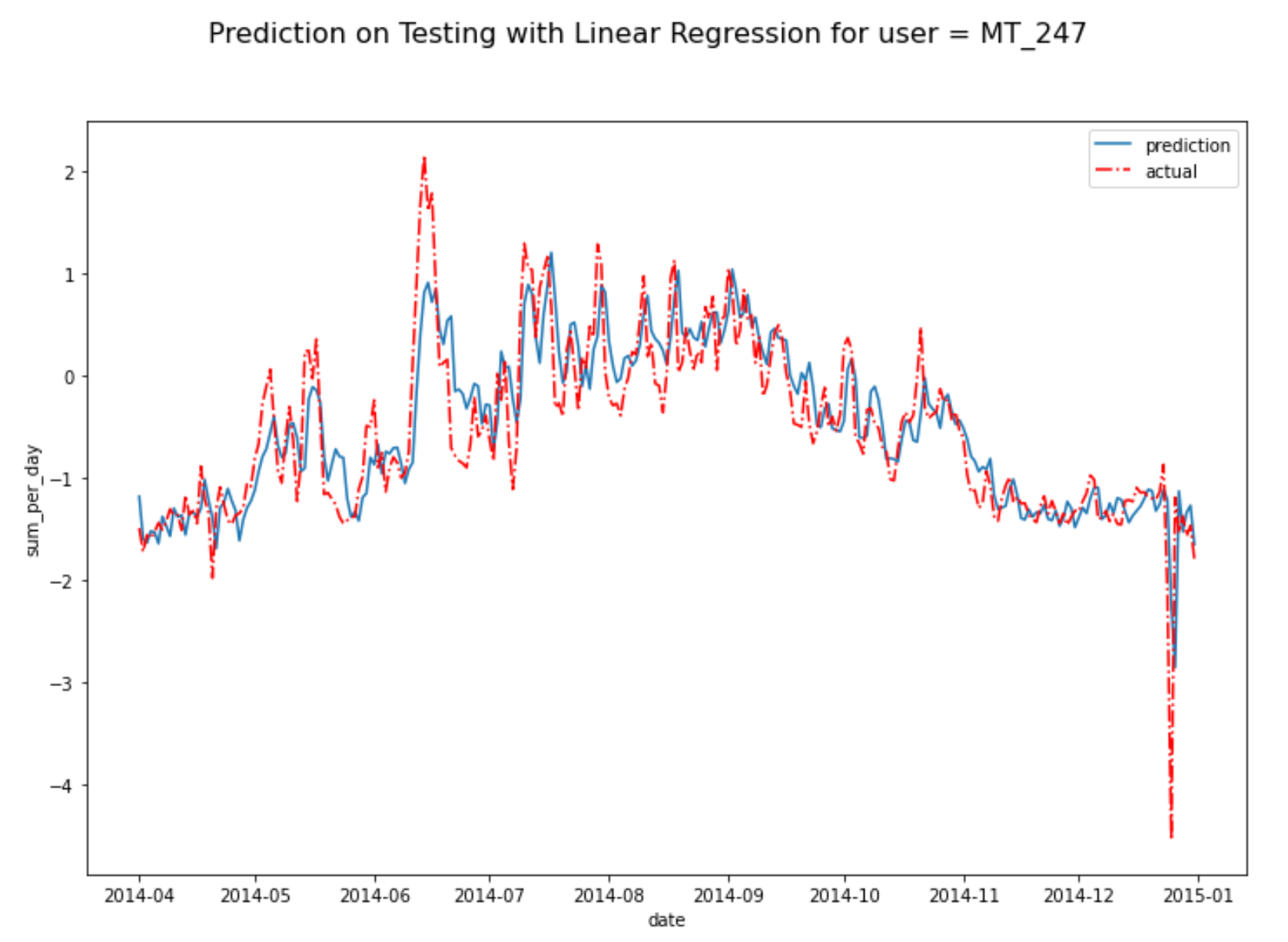


Figure 12: Prediction on user MT\_247 for multi model OLS by cluster

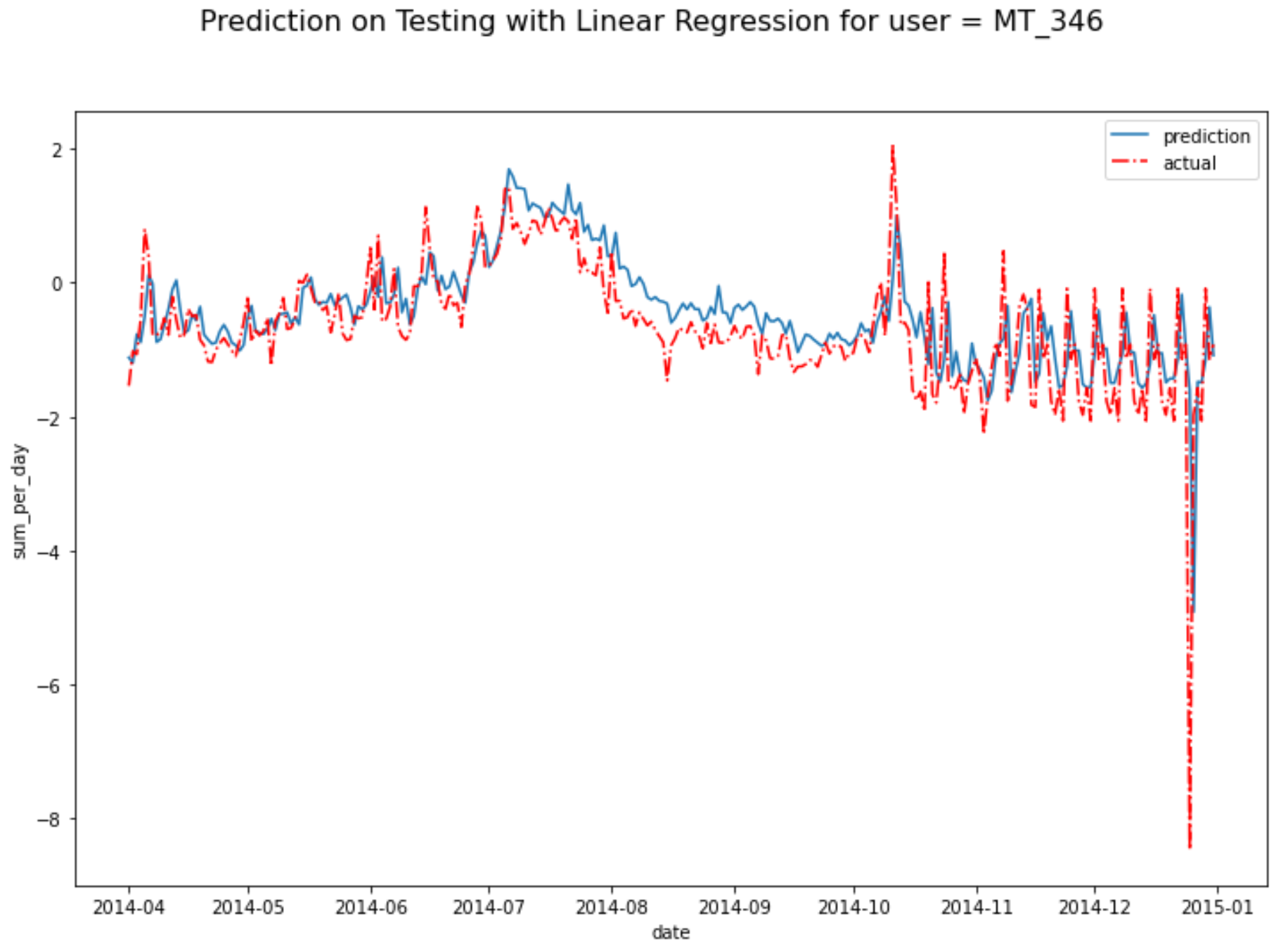


Figure 13: Prediction on user MT\_346 for multi model OLS by cluster

2. Prophet

In addition to OLS, we used the Facebook Prophet package to create a baseline for prediction. It automatically fits and uses time-series models to time-series data, making it extremely simple to use. There is no need to engineer any features or tweak hyperparameters, it simply just works. The performance was overall similar to OLS, performing better in some clusters and worse in others. Looking at the fit of some individual users, some were good and some were poor, indicating that the model was somewhat inconsistent.

| Methods: Prophet | **Training Set Performance** | **Testing Set Performance** |
| --- | --- | --- |
| Multi Model Cluster 0, MAPE | 1.0962 | 1.1300 |
| Multi Model Cluster 1, MAPE | 3.0234 | 3.1984 |
| Multi Model Cluster 2, MAPE | 3.4737 | 2.2673 |
| All data, MAPE | 1.2676 | 1.0979 |

Because Prophet creates a time-series model, it does not require inputs beyond a date. Thus, predictions further from the start of the test set period can be understood as rolling predictions. For example, a prediction 90 days from April 1, 2014 does not see any true data beyond what it was trained on. This differs from our Machine Learning models as those use lagged values in the inputs and must be set up differently to calculate anything beyond a single day into the future.

| Methods: Prophet | **1st 3-months** | **2nd 3-months** | **3rd 3-months** |
| --- | --- | --- | --- |
| All data, MAPE | 2.0450 | 2.4343 | 2.3261 |

As one may expect, the predictions get worse further into the future. It performed slightly better in the third period than second, but only slightly so.

3. Random Forests

Random Forest is the first model we have tried that is capable of capturing non-linear relationships in data. Compared to OLS, Random Forest overfit in all clusters. Overwhelmingly the performance of the model was better than OLS or Prophet, but took comparatively a lot longer to train.

| Methods: Random Forest | **Training Set Performance** | **Testing Set Performance** |
| --- | --- | --- |
| Multi Model Cluster 0, MAPE | 0.0149 | 0.0404 |
| Multi Model Cluster 1, MAPE | 0.0503 | 0.1324 |
| Multi Model Cluster 2, MAPE | 0.0220 | 0.1162 |

We can look at the prediction plot of the same three users as above. The visualization makes it obvious that the fit is much more suitable.

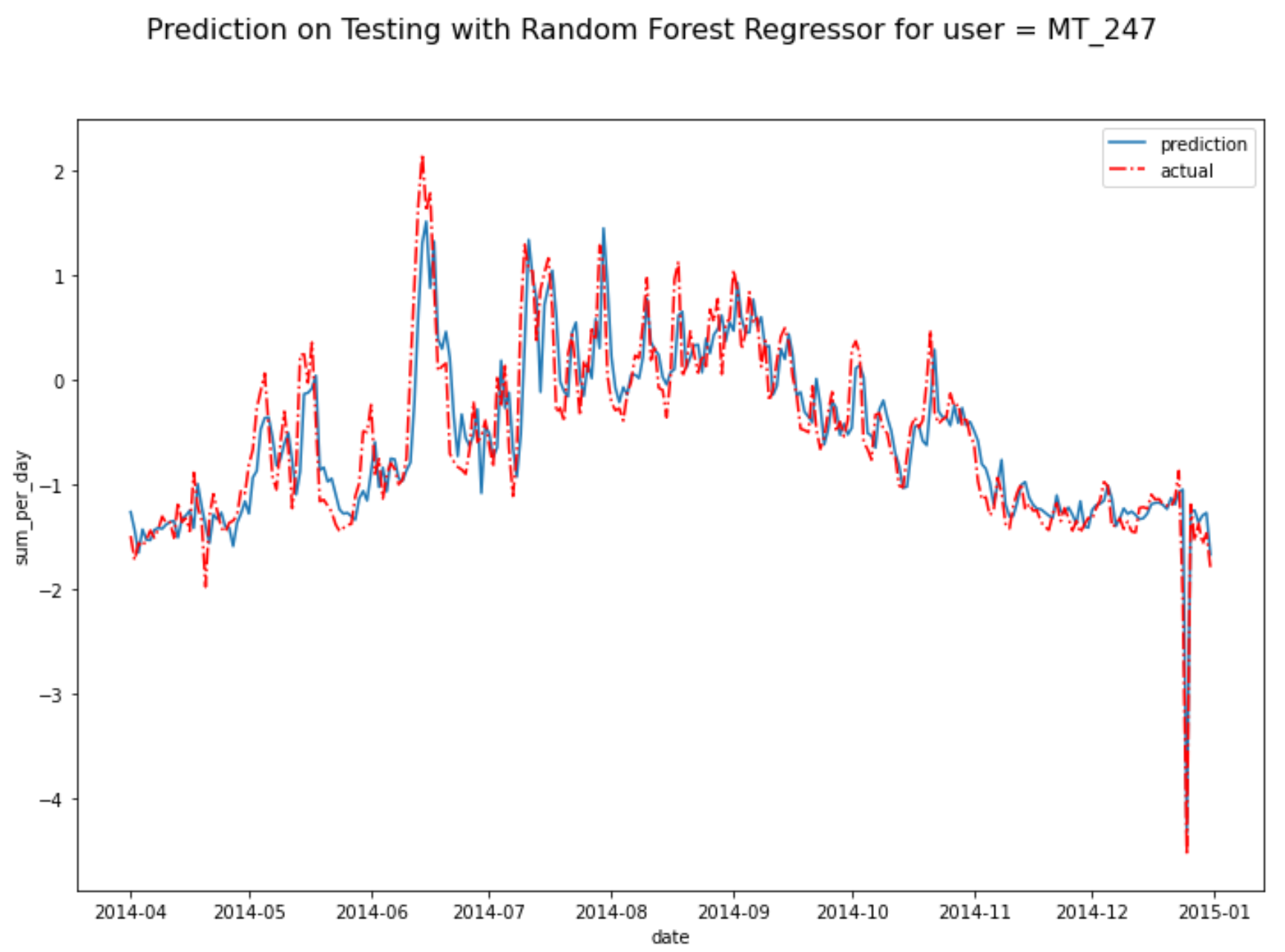


Figure 14: Prediction on user MT\_247 for multi model Random Forest by cluster

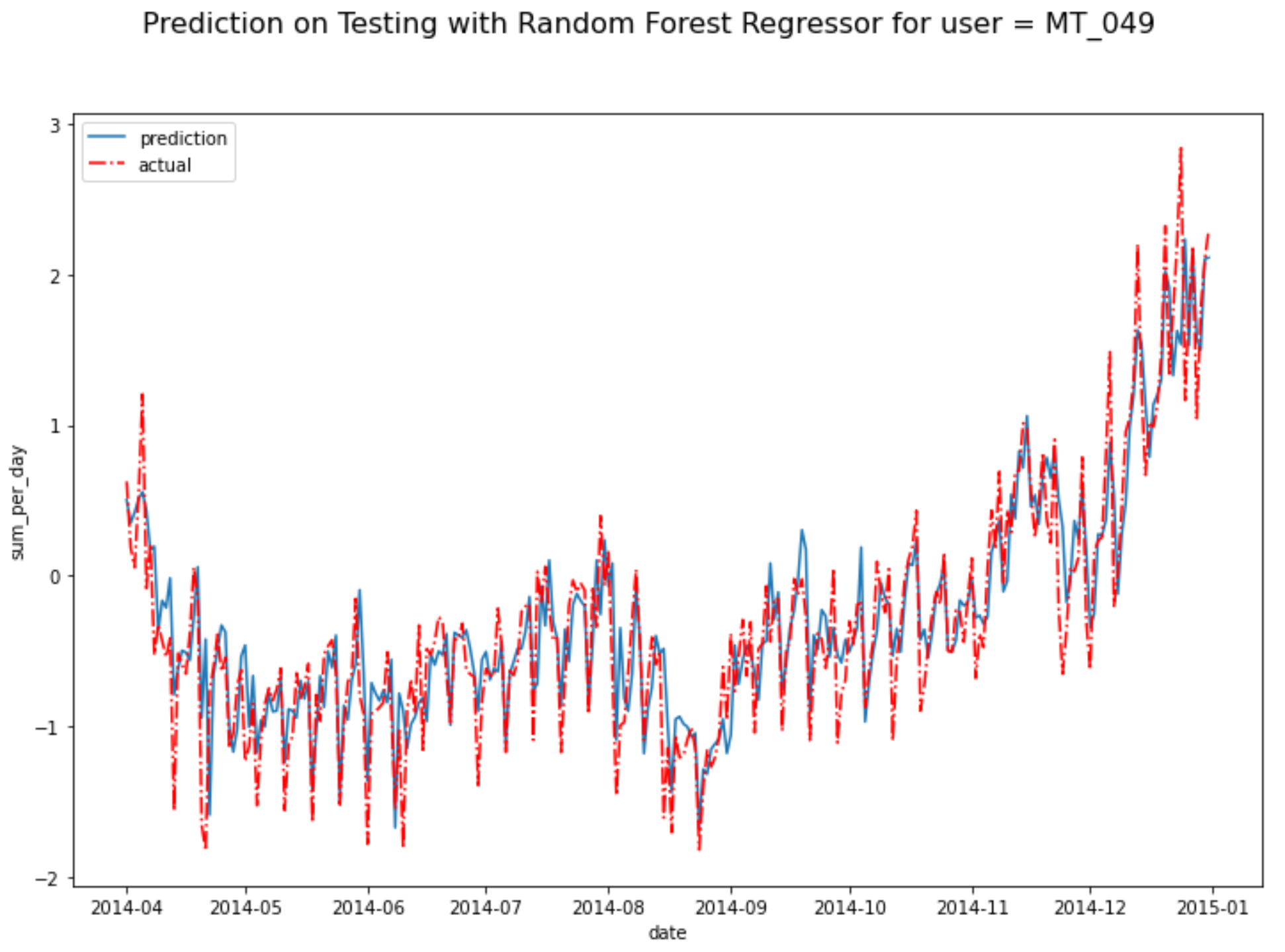


Figure 15: Prediction on user MT\_049 for multi model Random Forest by cluster

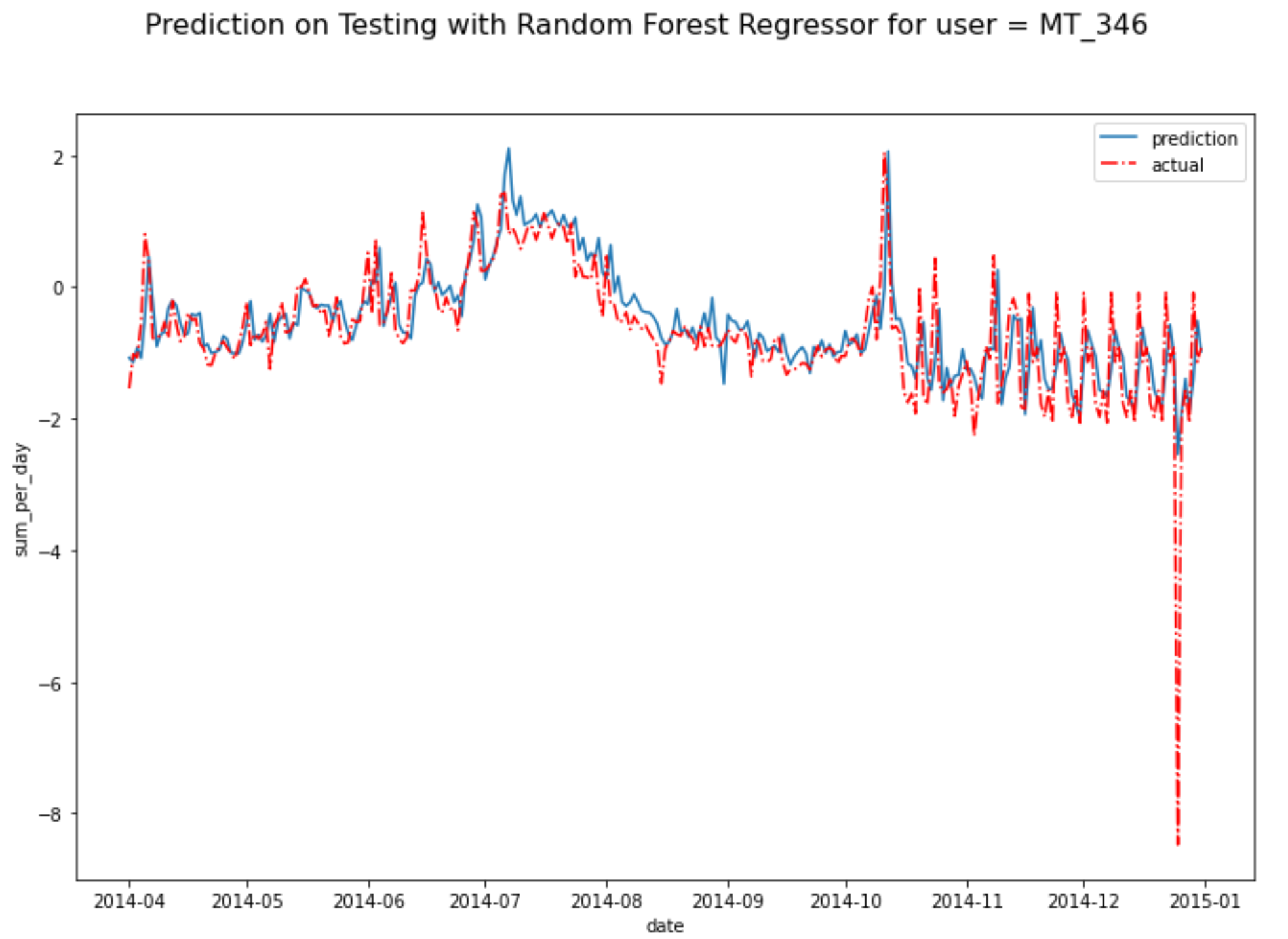


Figure 16: Prediction on user MT\_346 for multi model Random Forest by cluster

4. Gradient Boosting (XGBoost)

*1-day future prediction:*

Similar to Random Forest, we tried Gradient Boosting (specifically XGBoost) as a more efficient and effective tree-based model to capture non-linear relationships. In our benchmarking, the XGBoost model with default hyperparameters performed similarly to Random Forest, but much more efficiently. The Random Forest model took several minutes to train while XGBoost took only a few seconds. The efficiency was even *faster* than the simpler Prophet and OLS models. This was extremely important to note moving forward as we consider the tradeoffs between accuracy and efficiency. The flexibility of the model also allowed us to easily train and test on all of the data, without clustering it previously.

| Methods: XGBoost (Default) | **Training Set Performance** | **Testing Set Performance** |
| --- | --- | --- |
| Multi Model Cluster 0, MAPE | 0.0485 | 0.0613 |
| Multi Model Cluster 1, MAPE | 0.1501 | 0.1902 |
| Multi Model Cluster 2, MAPE | 0.1390 | 0.0888 |
| All data, MAPE | 0.0975 | 0.0847 |

As shown above, the performance is similar to the Random Forest model. In some model experimentation, we tested its performance *with* user information which actually gave slightly worse performance with off the shelf hyperparameters. This is useful to know, however, in case one would like to predict the future electricity use of a certain user.

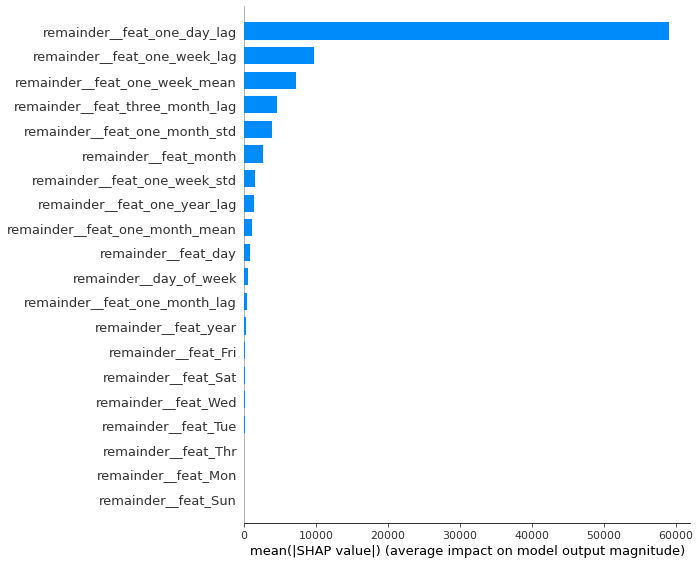


Figure 17: SHAP Plot of Feature Importances in All Data XGBoost Model

Above is a SHAP plot of feature importances, which can roughly interpret which features are predictive of the model. The energy use from one day before was significantly more important than any other feature which aligns with one’s intuition, that day to day electricity use does not change drastically. This is followed by energy use from a week before as the next most important, which could indicate a small amount of weekly seasonality. This visualization could be important moving forward if one were to iterate on the feature engineering, because some of the features may not be worth having at all. Further experimentation with adding or removing features could be worthwhile when determining data cost to performance tradeoffs.

In some experimentation, we tried a Bayesian Search with 5-fold cross validation to tune the hyperparameters. We could only achieve a trivially different MAPE on the test set. However, in other iterations of our data set, the search improved performance a small amount. The full process took a few hours each time, so we were reluctant to keep trying it. XGBoost is sensitive to changes, and any tweak to the dataset or search space could change the outcome of this. With all that being said, it could offer a slight performance boost without any significant changes to the data.

*Rolling prediction evaluation (30-day, 1-year):*

Satisfied with the off-the-shelf performance of the XGBoost model, we evaluated its performance with creating rolling predictions. Each prediction after the last was based on the previous prediction. So, one can intuitively assume that the model should perform worse the further out the prediction is.

| Methods: XGBoost Rolling | **1st 3-months** | **2nd 3-months** | **3rd 3-months** |
| --- | --- | --- | --- |
| All data, MAPE | 0.1695 | 0.2694 | 0.2087 |

Strangely, the model performed slightly better in the last period than the middle. This could possibly be explained by anomalies in the data that were not accounted for, or users coming online in the middle third. Because the test set is 9 months, we did not evaluate its performance in predicting a year into the future. This would require adjustment of the train/test split, and is something that could be investigated in future work.

5. TensorFlow Simple NN

We will be using a sliding window method for NN where the dataset is sliced to sequence subsets. The sequence retains the time-series order, and we shuffle the subsets after slicing.

With sliding window methods, the input will be a sequence of length 30, spanning from t=0 to t=29, the output will be a single prediction 1 day after t=30.

| Methods: Simple NN | **Training Set Performance** | **Validation Set Performance** | **Testing Set Performance** |
| --- | --- | --- | --- |
| Simple NN, MSE | 0.1530 | 0.1569 | 0.2323 |
| Simple NN, MAE | 0.2592 | 0.2586 | 0.2753 |

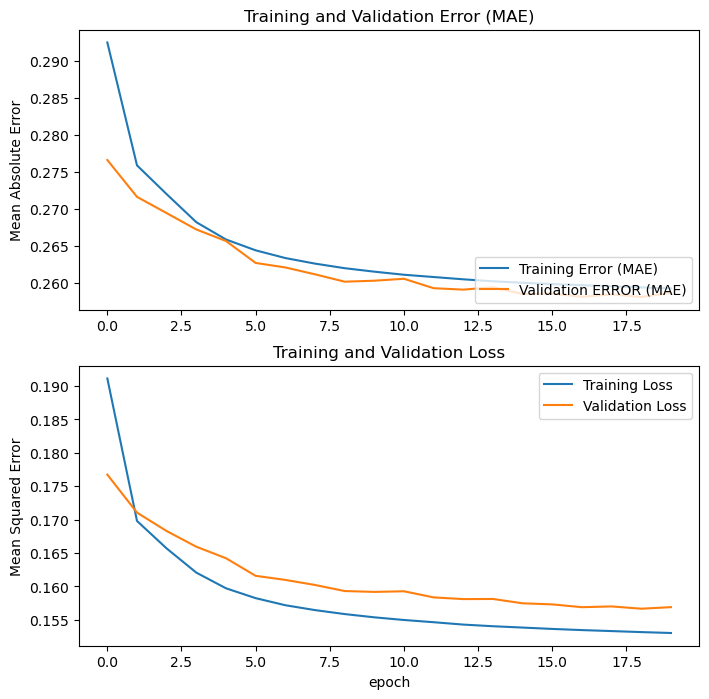


Figure 18: Training and Validation Error/Loss with Simple NN Model

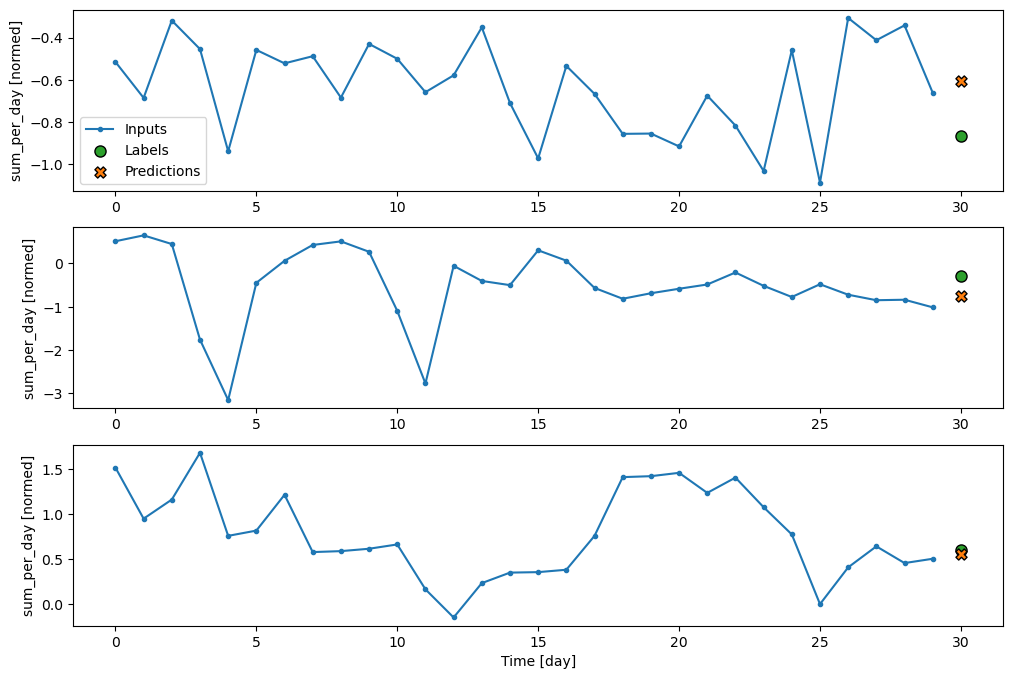


Figure 19: Example prediction with Simple NN Model

6. TensorFlow Single-step LSTM

With sliding window methods, the input will be a sequence of length 30, spanning from t=0 to t=29, the output will be a sequence of same length prediction 1 day after, spanning from t=1 to t=31.

| Methods: Single LSTM | **Training Set Performance** | **Validation Set Performance** | **Testing Set Performance** |
| --- | --- | --- | --- |
| Single LSTM, MSE | 0.1655 | 0.1885 | 0.3155 |
| Single LSTM, MAE | 0.2688 | 0.2792 | 0.2843 |

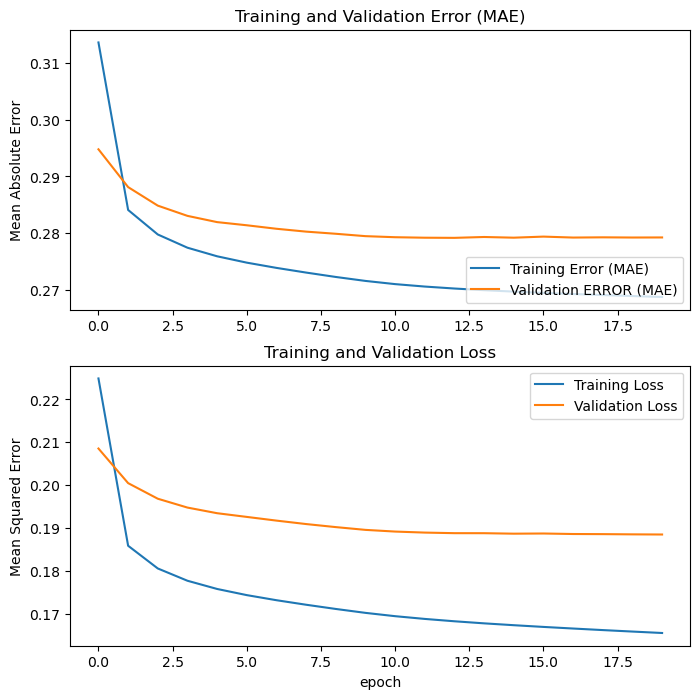


Figure 20: Training and Validation Error/Loss with Single-step LSTM Model

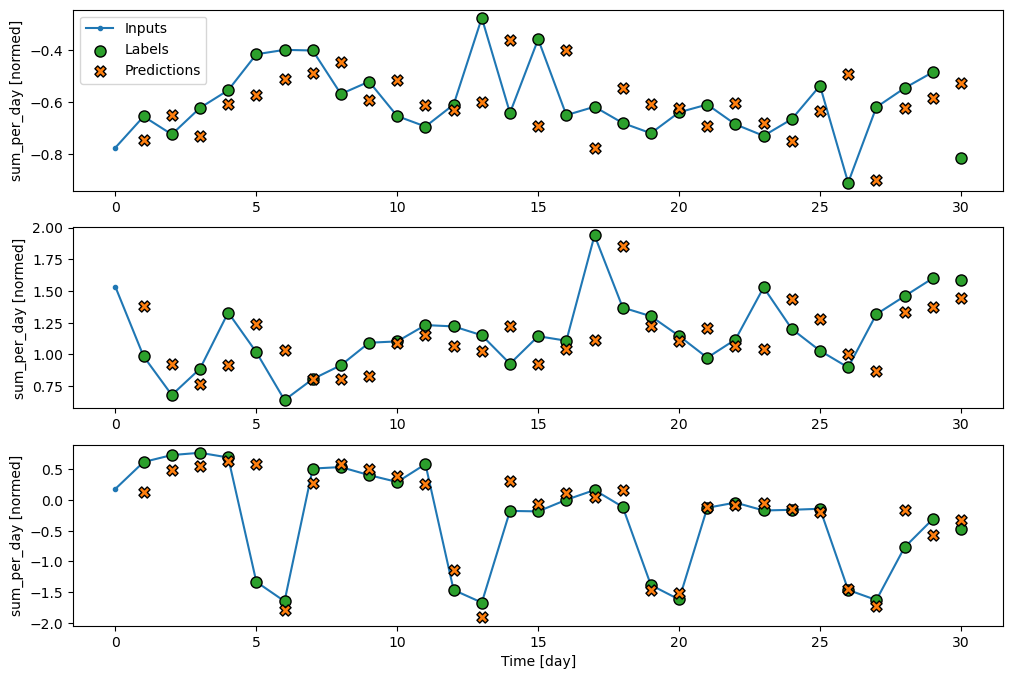


Figure 21: Example prediction with Single-step LSTM Model

7. TensorFlow Multi-step LSTM

With sliding window methods, the input will be a sequence of length 30, spanning from t=0 to t=29, the output will be a sequence of same length prediction 30 days after, spanning from t=30 to t=59.

This will be the final LSTM model for testing set evaluation, in which the input has only one set (30 days before the start of the testing set), the output of the current window will be the input of the next window. However, there are some problems (we haven’t figured out) for further predictions, in which the final MAPE after 9-month is significantly huge compared to the prior MAPE. By looking at Figure 24, there is a big jump after ~240 days. Maybe you could make it better.

| Methods: Multi LSTM | **Training Set Performance** | **Validation Set Performance** | **Testing Set Performance** |
| --- | --- | --- | --- |
| Multi LSTM, MSE | 0.3786 | 0.4448 | 0.7140 |
| Multi LSTM, MAE | 0.4320 | 0.4703 | 0.4999 |

| Methods: LSTM | **1st 3-months** | **2nd 3-months** | **3rd 3-months** |
| --- | --- | --- | --- |
| All data, MAPE | 0.4164 | 0.4443 | 8.347e16 |

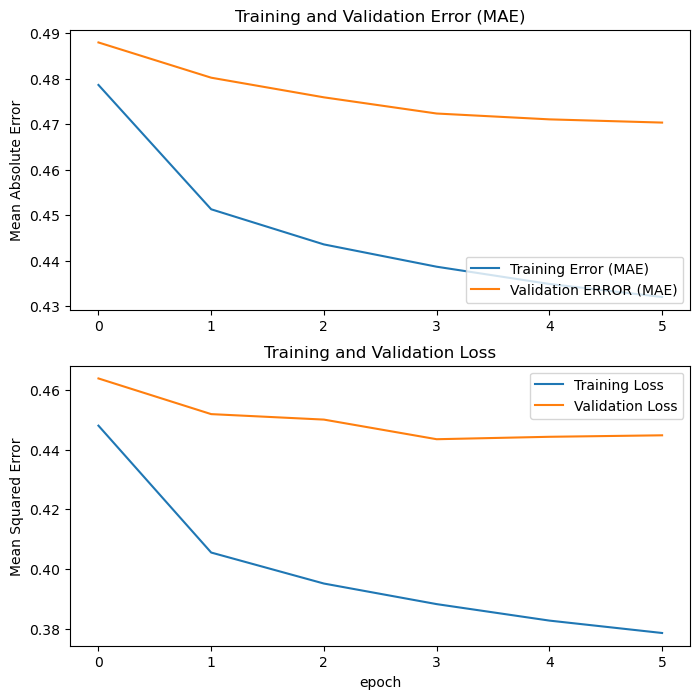


Figure 22: Training and Validation Error/Loss with Multi-step LSTM Model

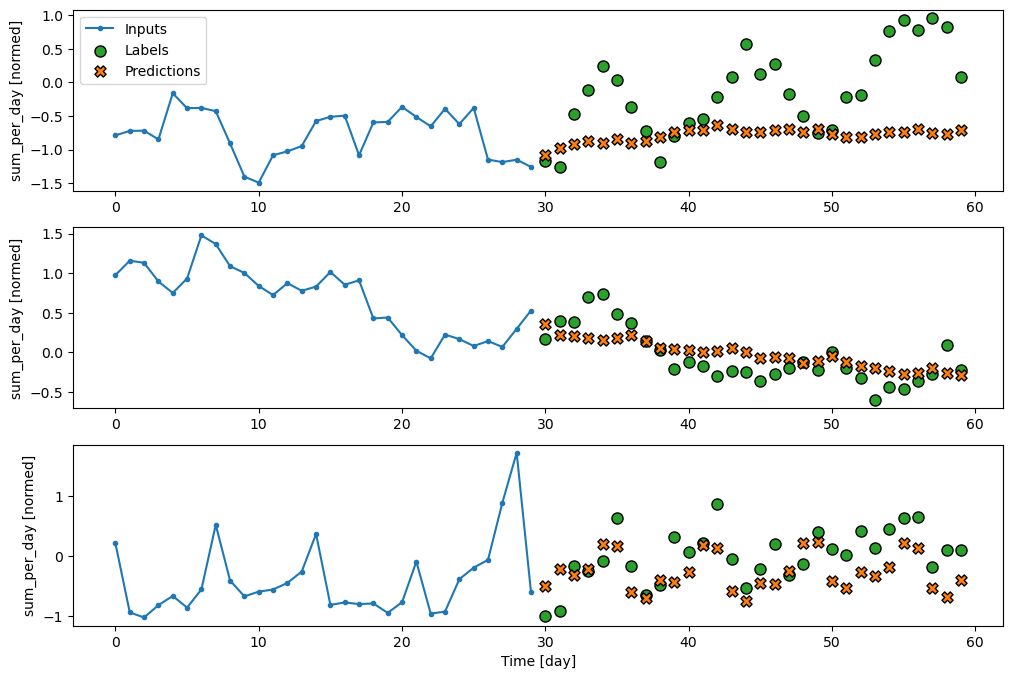


Figure 23: Example prediction with Multi-step LSTM Model

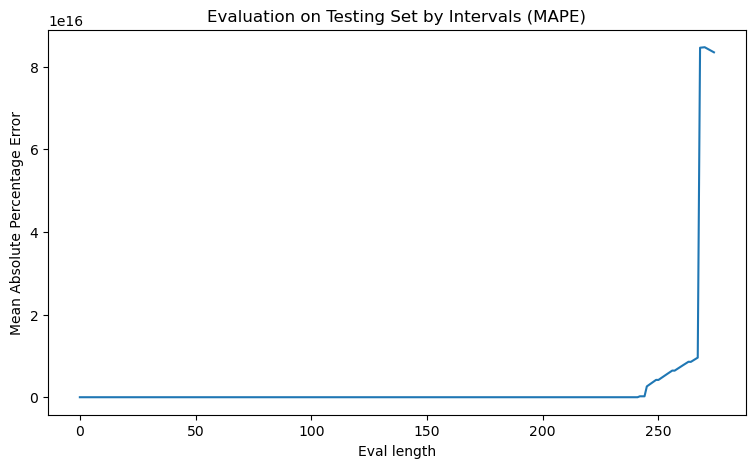


Figure 24: Final MAPE with Multi-step LSTM Model on Testing set by evaluate sequence length

8. Pytorch Seq2seq with Bahdanau attention

**Model Overview**

Because we intend to conduct many-to-many predictions (i.e. predicting multiple future time steps given multiple past time steps), we believe that an encoder-decoder architecture with attention would be well suited to this task. The encoder-decoder architecture allows the model to have different input and output sequence lengths. For example, we can have an input sequence length of 180 (i.e. past 180 days), and output sequence length of 60 (i.e. future 60 days). The attention mechanism was first proposed by Bahdanau et al., and the idea behind attention is to allow the decoder to focus on different parts of encoders’ output when decoding (generating outputs). Therefore, the attention mechanism allows the model to focus on different parts of the input sequence while making predictions for each output time step.



Figure 25: Seq2seq with Bahdanau Attention Architecture (source: [FLOYDHUB](https://blog.floydhub.com/attention-mechanism/))

The intuition of this architecture is that the model can use past information (hidden states from Encoder cells) to predict future time steps (decoder). In our model, encoder and decoder are both GRU (Gated Recurrent Unit) cells. We choose to use GRU over LSTM cells because there are less tensor operations required to compute each hidden state. A single LSTM cell has two gates (input and forget gate), while a single GRU cell only has one reset gate. This means that a GRU cell has fewer parameters to train, and therefore can potentially converge faster during training.

**Model Parameters**

| batch\_size | The number of samples in a mini-batch that train through the network in parallel. We recommend it to be a power of 2. | default: 512 |
| --- | --- | --- |
| input\_seq\_len | Length of the input sequence (the number of days in the past that we want to use as input) | default: 180 |
| output\_seq\_len | Length of the output sequence (the number of days in the future that we want to predict) | default: 60 |
| encoder\_hidden\_dim: | Dimension of the hidden state in the Encoder GRU cells | default: 128 |
| decoder\_hidden\_dim: | Dimension of the hidden state in the Decoder GRU cells | default: 128 |
| encoder\_num\_layers | Number of stacked GRU cells in the Encoder | default: 6 |
| decoder\_num\_layers | Number of stacked GRU cells in the Decoder | default: 3 |
| encoder\_ bidirectional | Boolean, indicating whether to use a bidirectional Encoder. A bidirectional Encoder reads the input sequence forwards and backwards, which can potentially help the model to learn long-term dependencies. | default: True |
| embed\_size | embedding size on user. Embedding layer functions as a lookup table that maps from discrete objects (such as user ids) to vectors of continuous values. These vectors are then passed into the GRU cells as initial hidden states to help condition on input/output sequence. The embedding weights are learnt along with model training. | default: 20 |
| dropout | dropout rate applied to both the Encoder and Decoder GRU cells. Dropout is a regularization technique that randomly drops out (sets to 0) some of the connections between nodes during training, in order to prevent overfitting. | default: 0.3 |
| teacher\_ force\_ratio | The probability of using the ground truth output sequence as the target input sequence to train the model, instead of using the predicted output as the target. This is known as “teacher forcing”, and can potentially help the model to learn faster. Teacher forcing is automatically disabled during inference. | default: 0.5 |

**Forward Pass Explained**

The forward pass of our model can be divided into two parts: encoding and decoding.

*Encoding*

The inputs from data loaders are three separate tensors: sequential features (batch\_size, input\_seq\_len, feature\_dim), user features (batch\_size, input\_seq\_len, 1), and target sequence (batch\_size, output\_seq\_len, 1). The sequential features are the time series data that we want to use as input, which include the electricity load for the input timestamps and their corresponding positional encoded seasonality information. For user feature tensor, each row [i, :, :] contains the user\_id that has been repeated for input\_seq\_len times. There are at most (batch\_size) unique users in each mini-batch. Therefore, we flatten the user tensor & extract the unique users [:, 0, :], and feed the new tensor of dimension (batch\_size, ) into the embedding layer to obtain user embeddings of dimension (batch\_size, embed\_size).

Now we have the user embeddings, there are several ways to feed this user information into encoders:

1. Concatenate the user embeddings with the sequential features and feed this into the encoder.
2. Use the user embeddings to initialize hidden state of the encoder GRU cells.

In this project, we choose the second method. We input the embedding tensor of (batch\_size, embed\_size) into a linear layer and obtain a new tensor with the same dimension as the hidden states of encoder GRU cells. This new tensor has the shape of (num\_encoder\_layer \* num\_directions, batch\_size, encoder\_hidden\_size) and is then used to initialize the hidden states of all encoder GRU cells. This makes sure that the user information is encoded into the model from the very beginning, which can potentially help to improve model performance.

GRU Layer (stacked GRU cells) takes 2 inputs:

1. sequential feature tensor (batch\_size, input\_seq\_len, feature\_dim), and
2. initial hidden state tensor (num\_layers \* num\_directions, batch\_size, hidden\_size).

The GRU layer outputs a tuple of (outputs, hidden state). *'outputs'* tensor has the shape of (batch\_size, input\_seq\_len, hidden\_size \* num\_directions), which contains the hidden state information of every timestamp in the last GRU cell in the stack. 'hidden state' tensor has the shape of (num\_layers \* num\_directions, batch\_size, hidden\_size), which contains the hidden state information of the last timestamp in the input sequence for every GRU cell in the stack. The `hidden state` tensor is also called 'context vector' and is used to help decoders to decode the target sequence.

*Decoding*

With 'outputs' and 'hidden state' tensors from the encoding stage, we are now able to start decoding the target sequence. Decoder module is very similar to the encoder module, except that we are applying the attention mechanism at every timestep during decoding. If our output sequence length is 60, the decoder module will run for 60 timesteps, and at each timestep, we will:

1. Form decoder GRU input tensor
2. retrieve decoder GRU hidden state from previous decoder
3. make prediction for the current timestep
4. pass the hidden state to the next decoder

Form decoder GRU input tensor

For the decoder GRU input tensor, we first apply attention on the `outputs` tensor from the encoding stage to obtain a new context vector. The new context vector will be a weighted sum of the `outputs` tensor. We then concatenate the new context vector with the user embedding tensor and the last predicted value from the previous decoder. This new tensor will be the input for the decoder at the current timestep.

retrieve decoder GRU hidden state from previous decoder

The hidden state is retrieved from the previous decoder by simply taking the hidden state output from the previous decoder. This hidden state is then fed into the current decoder as its initial hidden state.

make prediction for the current timestep

After getting the decoder's GRU outputs, we apply a linear layer on the concatenated GRU outputs and context vector to obtain the prediction for the current timestep. The prediction is a real value and is used to form the target sequence.

pass the hidden state to the next decoder

After making predictions, we pass the hidden state of current decoder to next decoder as its initial hidden state.

The process repeats until we reach the end of the output sequence.

**Training**

We trained the model using Adam optimizer with a learning rate of 0.001 for 20 epochs on a subset of the training data to test the feasibility of the model. The training and validation loss are plotted in the figure below.

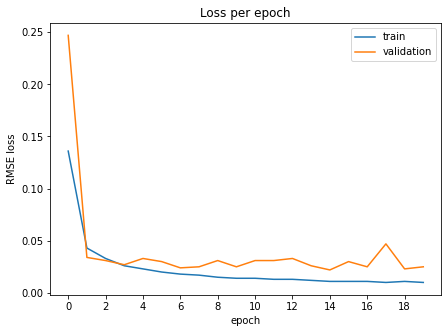


Figure 26: Seq2seq Training/Validation Loss

9. Pytorch MA+FC

Realizing that the data have a strong seasonality, we first use the moving average to separate the series into trend part (the moving average) and the seasonality part (the residue apart from moving average). In addition, we get temporal information from the timestamp of input series. Trigonometric transformation is applied on cyclical features such as Month\_in\_Year, Day\_in\_Month, Day\_in\_Week.

After that, we simply apply fully connected linear layers on the trend part, seasonality part and temporal part respectively and add them up to approach the output series.

Due to the limited length of the dataset, we only train the model with 180-day input series and 30-day output series. To predict the time series in one year, we apply the model repeatedly. The predicted output is fed as the input of the model with window slide to later time.

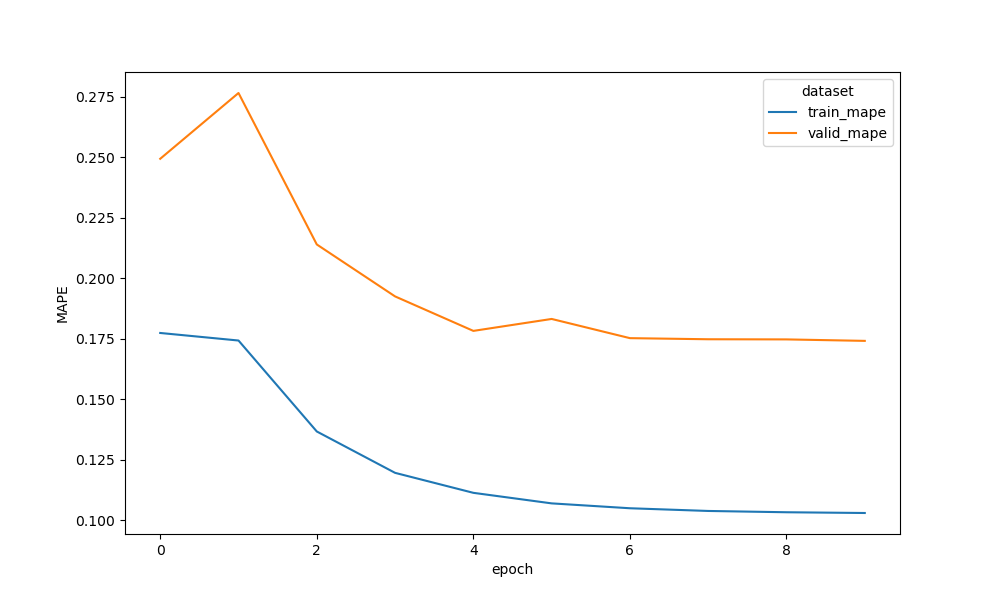


Figure 27: MAPE during training in training set and validation set

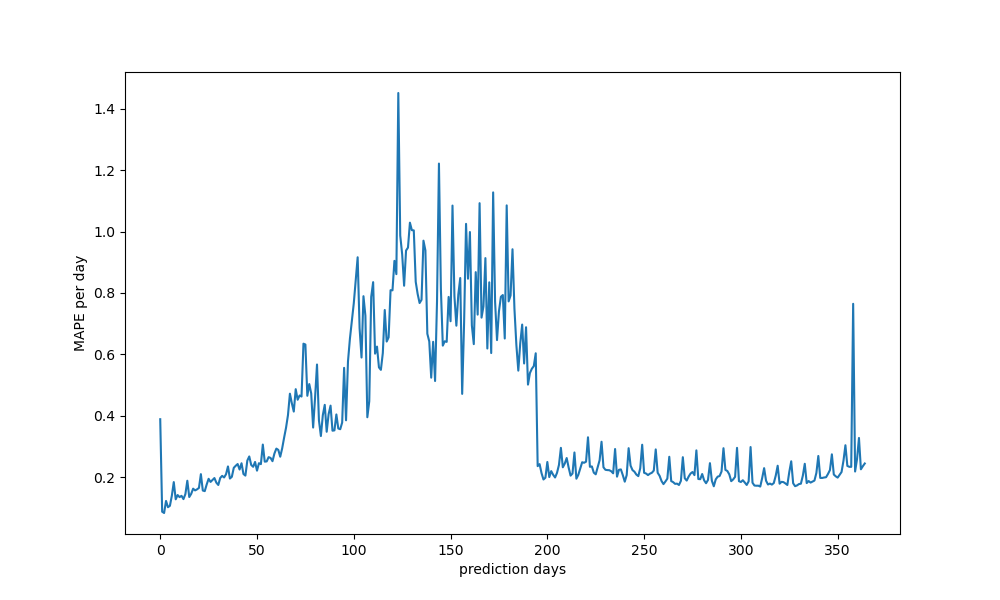


Figure 28: MAPE trend in one year of prediction

Conclusions and Future Work

**Model selection**

From the results, it is easy to see that the more sophisticated models performed better than the simpler OLS/Prophet models. **Gradient Boosting** seemed to offer the best performance to cost ratio, as it was extremely accurate even without hyperparameter tuning. Random Forest offered similar and better performance off-the-shelf, but took much longer to train and run. Also, although the neural network models potentially *could* be more accurate with more tweaking and experimentation, it is unclear if this is worth the tradeoff in cost. At several points during our project, Python crashed, ran out of memory, or ate up cloud credits when working with the neural networks. After all, it is a very large dataset to work with after our transformations to get it into a workable form. Furthermore, Gradient Boosting retains some explainability (like in the SHAP plot) that allows us to derive insight regarding what is important for prediction. Neural networks famously tend to be black boxes and are difficult to explain what goes into prediction.

| Rolling MAPE | **1st 3-months** | **2nd 3-months** | **3rd 3-months** |
| --- | --- | --- | --- |
| Prophet | 2.0450 | 2.4343 | 2.3261 |
| XGBoost | 0.1695 | 0.2694 | 0.2087 |
| LSTM | 0.4164 | 0.4443 | 8.347e16 |
| MA+FC | 0.2712 | 0.7510 | 0.2958 |

**Future Work**

As we documented above, we can further explore the clustering model for all users to have different forecasting models in each group. Thinking about the forecasting problem in real-life situations, if our models are conducted by an electricity company that wants to forecast usage for a given region and allocate enough resources, there will be much more users than we have right now (370 users in this dataset). In which case the user trends can be more diverse and may cluster in more groups. It is hard for a one-to-all model that captures all trends by every user, if we have more time (or more diverse dataset) clustering methods can improve overall performance.

Another approach to clustering is gather more user information like location, users in Southern Hemisphere may have opposite trends versus users in Northern Hemisphere, users live near equator may use more electricity in summer (cooling) than users live near polar regions, vise versa that users live near polar regions may use more electricity in winter (heat).

Furthermore, there are several aspects of our current work that could be improved. For example, our data cleaning function handles parts of the data that make modeling tricky, but there could be another way that retains more data for training. Or, it could be that some user just behaves strangely such that the electricity load increases significantly after one day (may include a new family member). Beyond things like this, there could also be more features to engineer from the existing data that we missed that could provide a signal boost.

We didn’t have time/resources to finalize some sophisticated models (ex. Pytorch Seq2Seq, MA+FC) nor to train models with 15-min interval. For example, with a 15-min interval, we can capture the trends with day and night – some users use more electricity at night as they work in different locations during the day, or during another night many users use a lot of electricity as there is a world-wise event happening (Olympic sports). This can be expanded to code holiday/event features (ex. Christmas, Thanksgiving) where people will group with their family for a big dinner. Overall, bringing in additional data to find signal would be helpful in improving performance.

Finally, there are many improvements on the current stage, you could explore more based on what we had right now.