I. <u>Abstract:</u>

ASHRAR- Great Energy Predictor mainly deals with finding accurate models of metered buildings energy. There are various areas provided such as chilled water, electric, hot water and steam water. The data is derived from more than 1000 buildings within a three-year timeframe. The major motive behind this project is to predict continues values of energy consumption hence a Regression Problem where we develop a model custom tailored for different use case scenarios such the consumers could benefit from reduced costs compared to the conventional retro fitted model. Since conventional models do not scale well since a specific meter type doesn't work with different building type.

II. <u>Approach:</u>

Upon analyzing the current work done on Kaggle we found out that for datasets which are very large (typically multiple gigabytes of data having 10,000+ rows of data elements) Light Gradient Boosting Model (LGBM) scales really well giving us a Root Mean squared error (RMSE) of less than 0.90. We further have different approaches to modifying our baseline model built using LGBM. These approaches include splitting the data into 2 equal halves such that training metric- RMSE is improved further. Another way of improving the model is by utilizing k-Fold splits on our dataset where we split the data into 3 or 4 parts and train our entire model on top of it. **However, since LGBM is outside the scope of this class, we implemented a LSTM Neural Network along with a traditional Fully Connected Neural Network and performed several optimization techniques essentially hyperparameter tuning to have our best performing network.** It is important to note here that we already know that LSTM will perform far worse on our existing dataset compared to LGBM. However, since we are concerned with building a Neural Network and not utilizing any decision trees algorithm hence, we focus on making our LSTM based network as good as it could be by utilizing all possible parameter tweaks. Also, for our fully connected neural network we carry out certain parameter tweaks to have optimum results.

These tweaks involve changing the Batch Size, Activation Functions at the hidden and output layer, Learning rate , Number of Epoch's, Number of Neurons, Loss Function , Optimizer type, Number of Layers in the network, splitting the data into different parts for train and validation, adding early stopping ,adding dropout etc.
Thus, in this project we have 5 models which can be described below:

Model 1: Light Gradient Boosting Baseline Model (LGBM)giving us a training RMSE of less than 0.761 while the validation RMSE is close to 0.764. These are the best model values after tweaking a lot of things. Since this is not a neural network we move on to other alternatives for the scope of this project. It must however be remembered that decision trees give great results especially when we are dealing with large amounts of data
Model 3 (Neural Network Model): This is a LSTM network along with several optimizations that we performed as discussed later. The training RMSE with approach with default network settings turned out to be around 2.0 while the validation RMSE is around 1.7
Model 4 (Neural Network Model): This is a traditional MLP network along with several optimizations that we performed as discussed later. In this approach we use Keras Embeddings layer API to build our Neural Network. The training RMSE with this approach turned out to be 1.1 while the validation RMSE is about 1.02.
This is actually our best performing model among Neural Network Considerations.

III. Methodology:

1) **Import all necessary libraries**

- For our implementation we utilize TensorFlow & Keras. Apart from TF & Keras we also make use of NumPy, Seaborn, Matplotlib & Pandas for data visualization & data pre-processing

2) **Import the dataset**

- We obtain the data from Kaggle's competition page. The dataset essentially consists of 5 .csv files with several characteristics along with 1 submission.csv file to be submitted on Kaggle.

3) **Explore & Visualize the data to get an intuition of various attributes.**

- To get a proper intuition of our data we apply several built-in methods from Keras and TensorFlow to get the dimensions and label size for both the training as well as test dataset.
  Data visualization in this project is extremely critical since it lets us depict a clear picture of how meter energy usage under different use case scenarios such as based on industry type, a particular day of the week, peak time etc. Once we have a proper understanding of how the data is distributed, we can do pre-processing upon it to remove unwanted outliers such that the memory usage footprint is reduced, and we have an optimized model.

4) **Data Preprocessing**
   This step involves handling of missing values of several entities such as building and weather data. Since the weather information and building

information is located in different files, we also need to combine them into a single train data file i.e. data merging
We also need to create new feature data; for example, create new features which are cyclic based on timestamp data

5) **Model 1 Light Gradient Boosting Model**

- Simulations for various changed parameters such as number of Epoch's, Activation Functions, Learning rate, Optimizers, Batch size, Loss functions etc. Since this model (not a neural network) is not of our primary concern, I have included only the most optimized values in the code. i.e. the on the current values my model performs best

6) **Model 2 LSTM Neural Network:**

- Building a Recurrent Neural Network for train and prediction along with several parameter tweaks

7) **Model 3 MLP Neural Network:**

- Building a fully connected neural network with several parameter tweaks.

8) **Comparison of performance metrics of different networks**

- Results for model loss, accuracy for training, test and validation set after the models are subjected to various design tweaks (hyperparameter tuning)

IV. <u>**Dataset:**</u>

The 5 main files that are utilized in this project along with a submission.csv file described as follows:

<u>**Train.csv:**</u>

- building_id - Foreign key for the building metadata.
- meter - The meter id code. Read as {0: electricity, 1: chilledwater, 2: steam, 3: hotwater}. Not every building has all meter types.
- timestamp - When the measurement was taken
- meter_reading - The target variable. Energy consumption in kWh (or equivalent). Note that this is real data with measurement error, which we expect will impose a baseline level of modeling error.

Shape of this file -> 1048576 x 4

<u>**Building_meta.csv:**</u>

- site_id - Foreign key for the weather files.
- building_id - Foreign key for training.csv

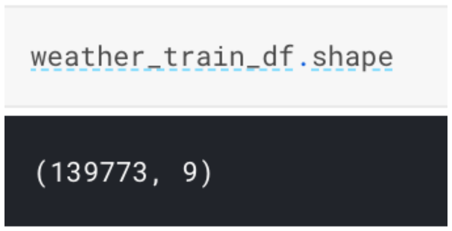
- primary_use - Indicator of the primary category of activities for the building based on EnergyStar property type definitions
- square_feet - Gross floor area of the building
- year_built - Year building was opened
- floor_count - Number of floors of the building.

Shape-> 1450 x 6

<u>Weather train/test.csv:</u>

- Weather data from a meteorological station as close as possible to the site.
- air temperature - Degrees Celsius
- cloud_coverage - Portion of the sky covered in clouds
- dew_temperature - Degrees Celsius
- precip_depth_1_hr - Millimeters
- sea_level_pressure - Millibar/hectopascals
- wind_direction - Compass direction (0-360)
- wind_speed - Meters per second

The weather train csv is of the following shape->**139773 x 9**



```
weather_train_df.shape
```

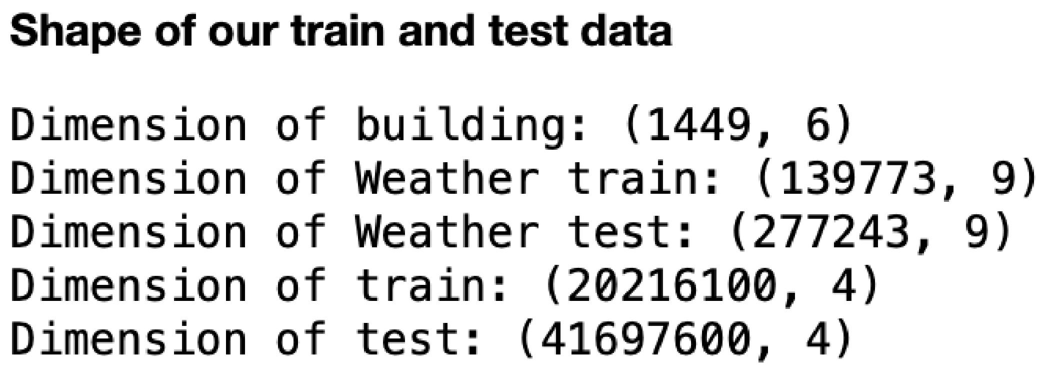
```
(139773, 9)
```

The shape of weather test file is -> 277244 x 9

<u>The weather and building data are merged together so as to get all the necessary features in 1 single file.</u>

<u>Test.csv:</u> The submission files utilize row numbers for ID codes in order to save space.   This file has no feature data, only aims to get the prediction output values in correct order

Shape->1048576 x 4

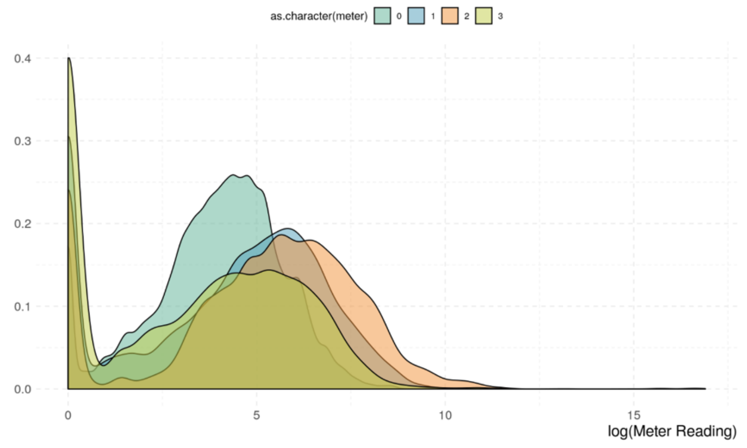

Shape of our train and test data

```
Dimension of building: (1449, 6)
Dimension of Weather train: (139773, 9)
Dimension of Weather test: (277243, 9)
Dimension of train: (20216100, 4)
Dimension of test: (41697600, 4)
```

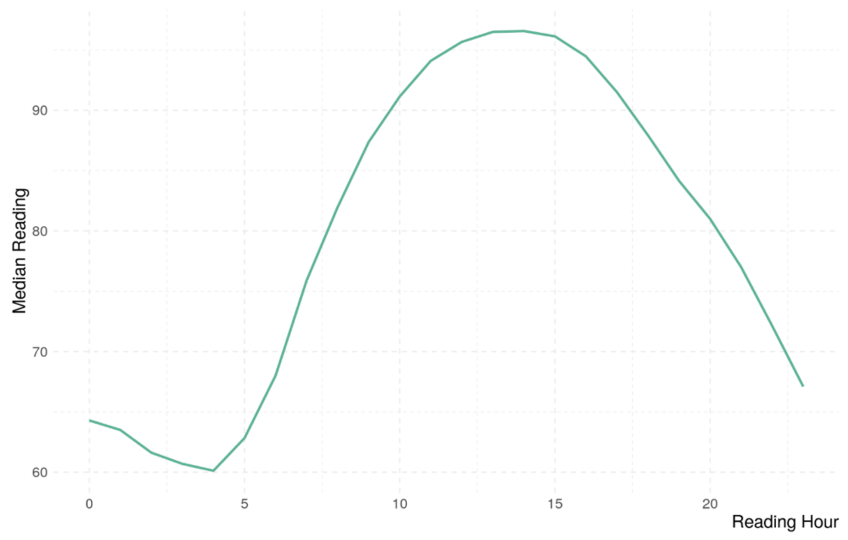
V. Data Visualization:

- There are 4 different meter types which are given as follows:
  0: electricity
  1: chilled water
  2: steam
  3: hot water

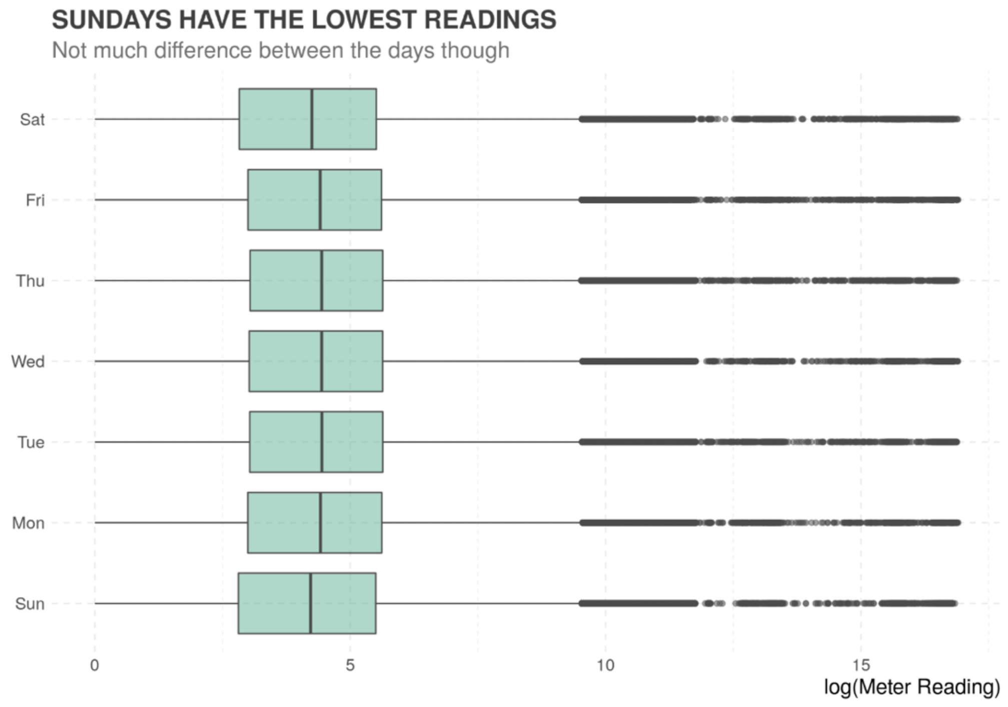
- The steam meter type is the least efficient while the electricity is the most efficient



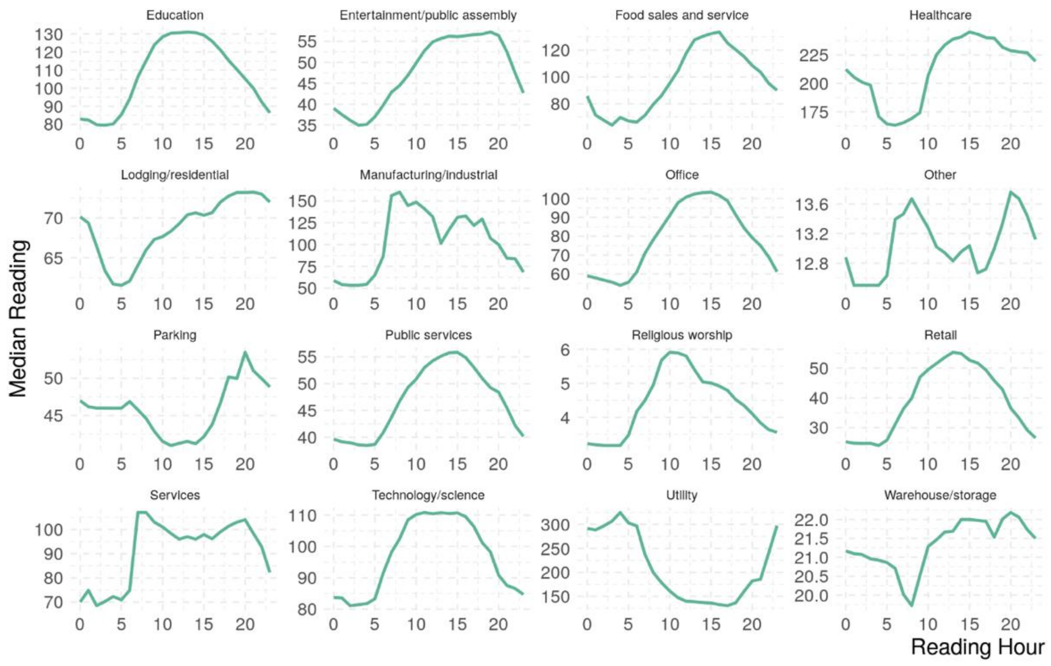
- The readings have a peak during the middle of the day. This is evident from the graph below:



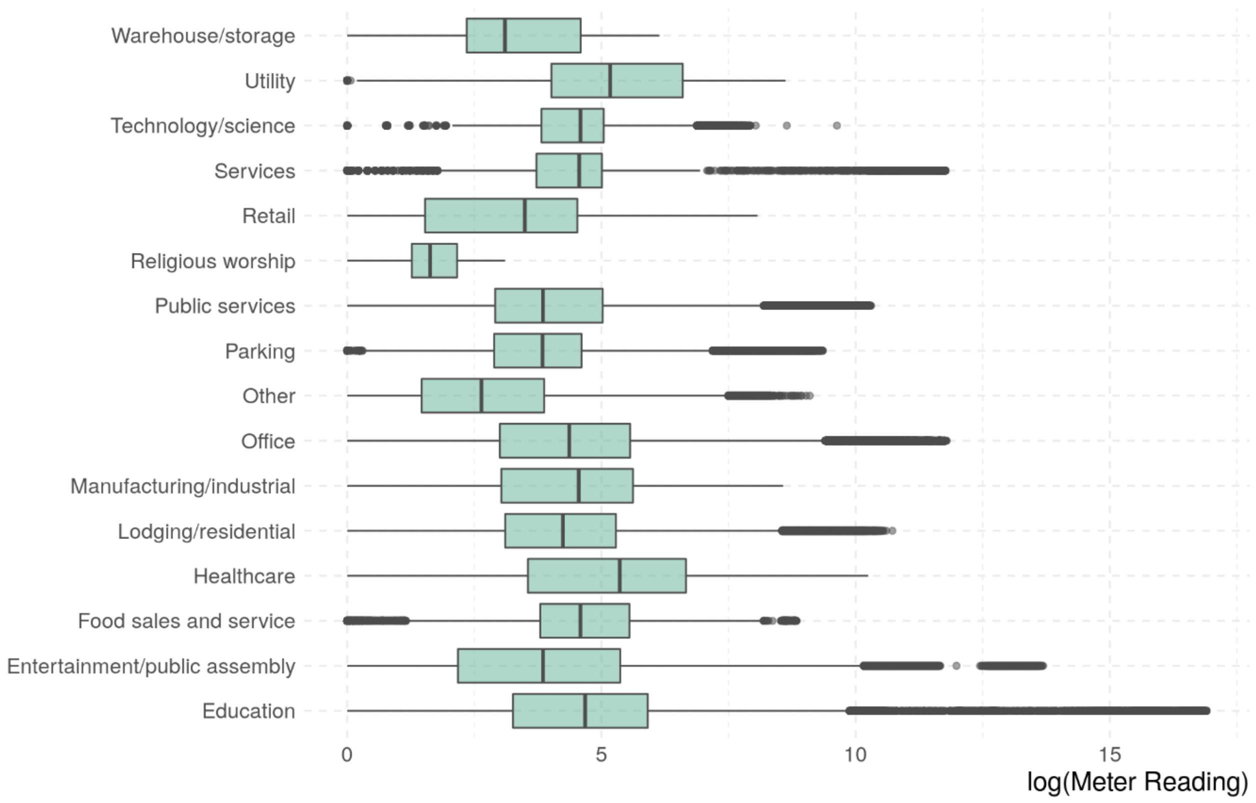
- Sundays have the lowest readings; although there isn't much difference between the days



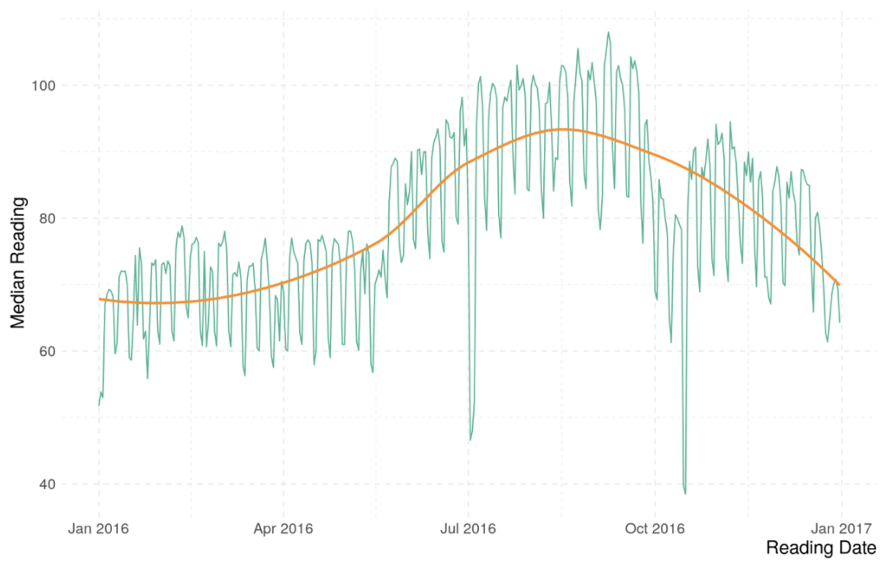
- The readings based on building type can be shown as below:



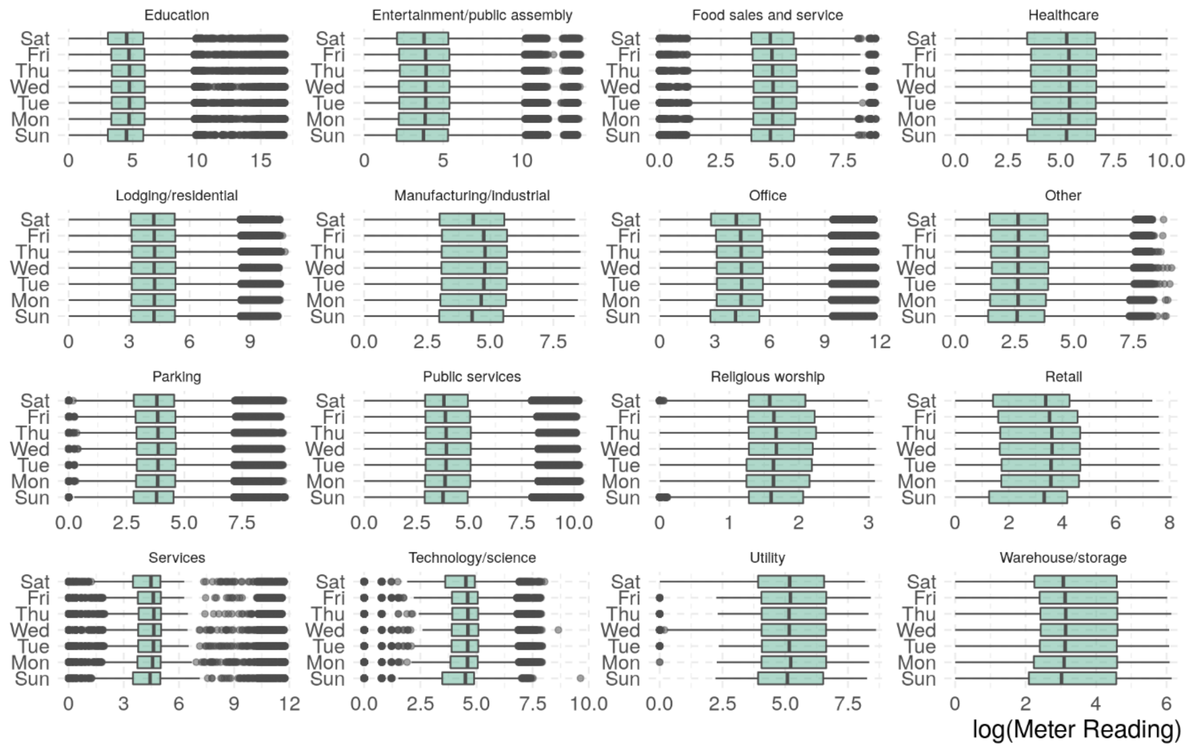
- It is also observed that the utilities and healthcare have the highest readings while religious worship places have the lowest.



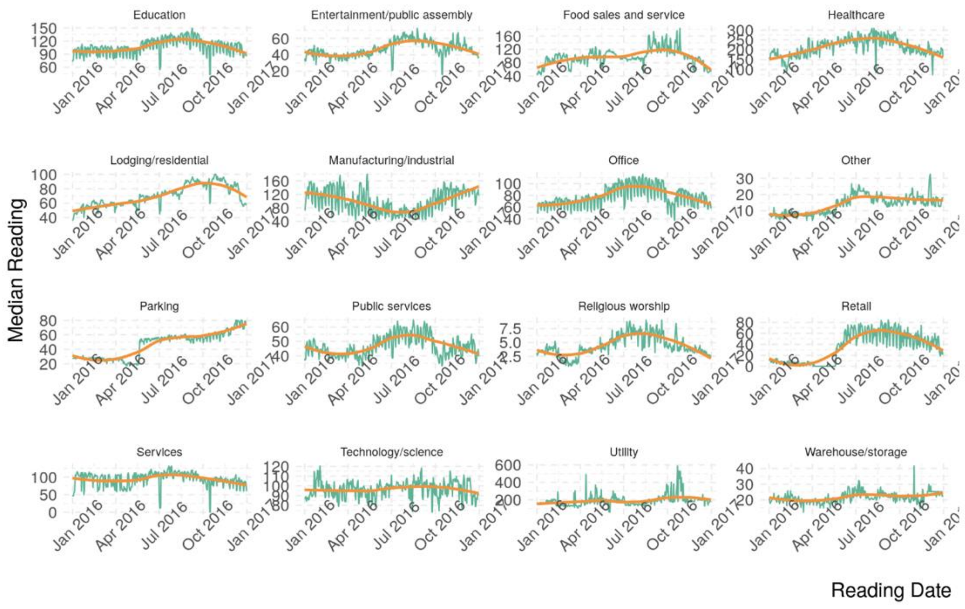
- Another important observation is that the readings have peaked from May to October



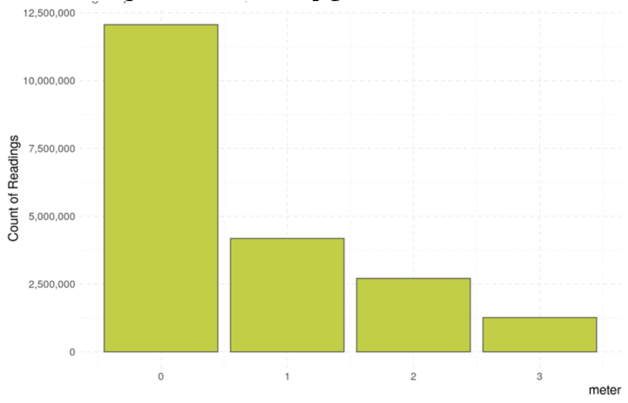
- Religious Worship places have higher readings on weekends as can be observed below.



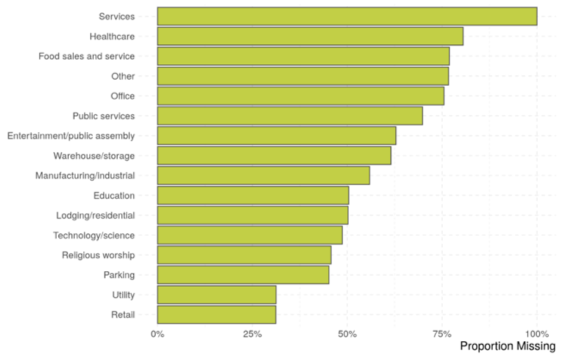
- Some business types remained fairly stable over the period while manufacturing has a lot of activity especially from Jan 2016-2017 period



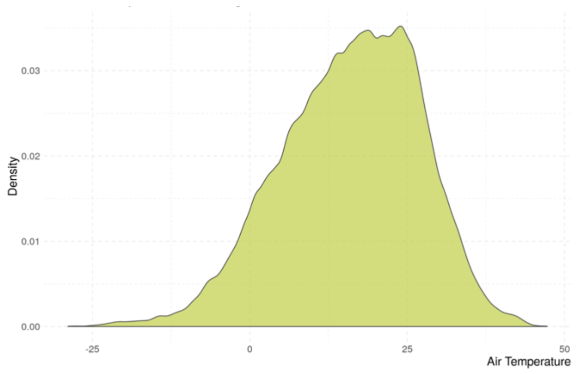
- Electricity is the most frequent meter type measured



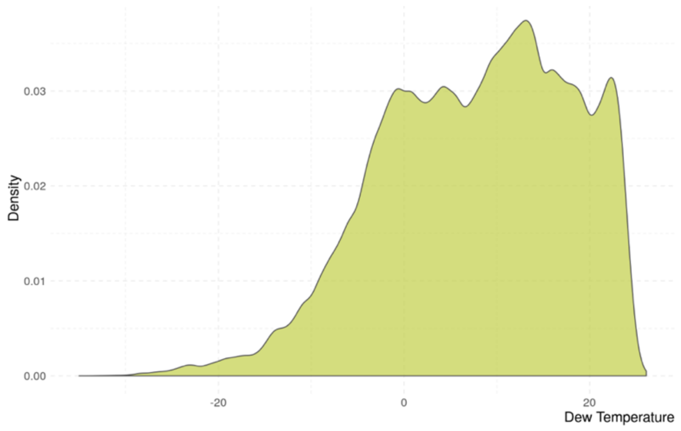
- One thing worth mentioning is that 11-16 building types have over half missing year-built records. This is evident from the graph given below:



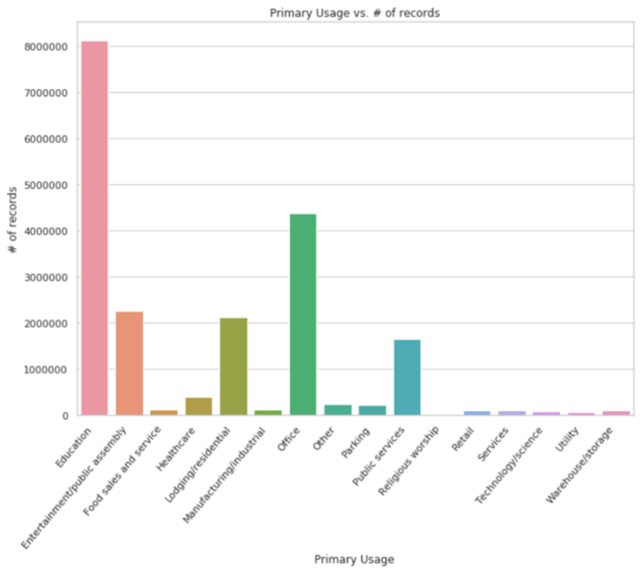
- Most temperature range obtained is between 13-25 degrees.



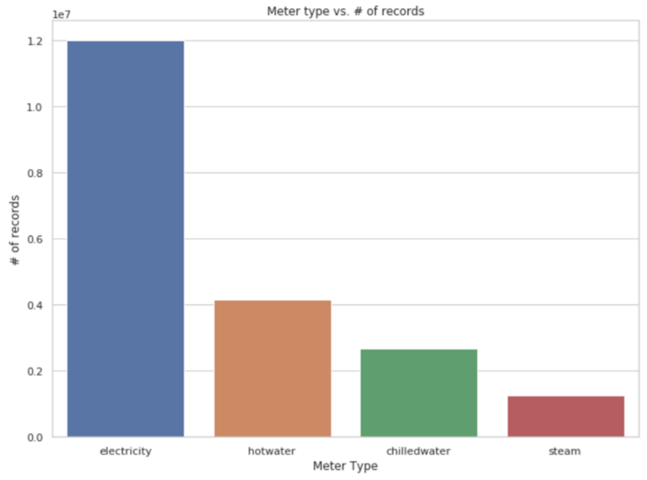
- There are 100,140 missing dew temperature records in our combined training set.



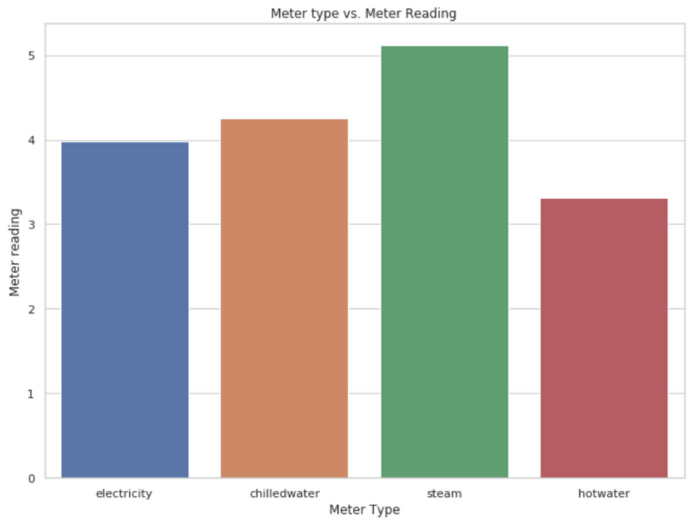
- Distribution of primary usage of buildings can be given as:



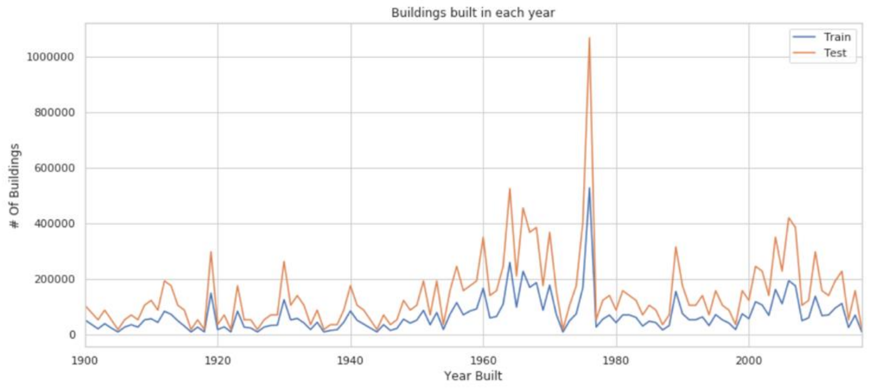
- The distribution of meter types is given as follows:



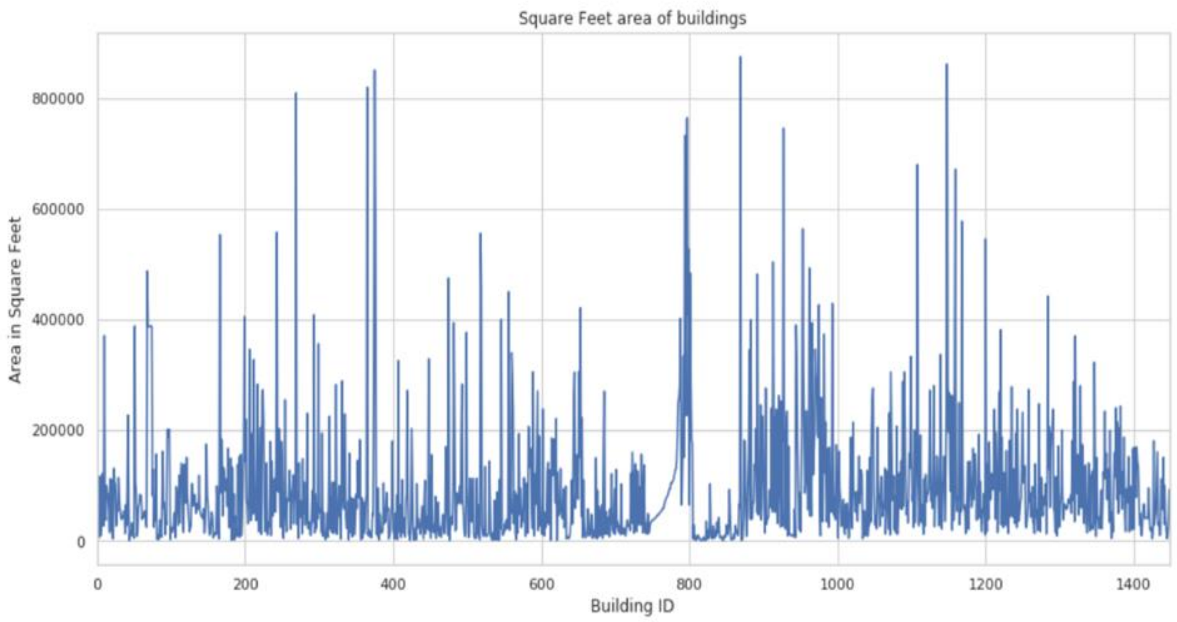
- The distribution of meter readings for each meter type is illustrated in the bar graph below:



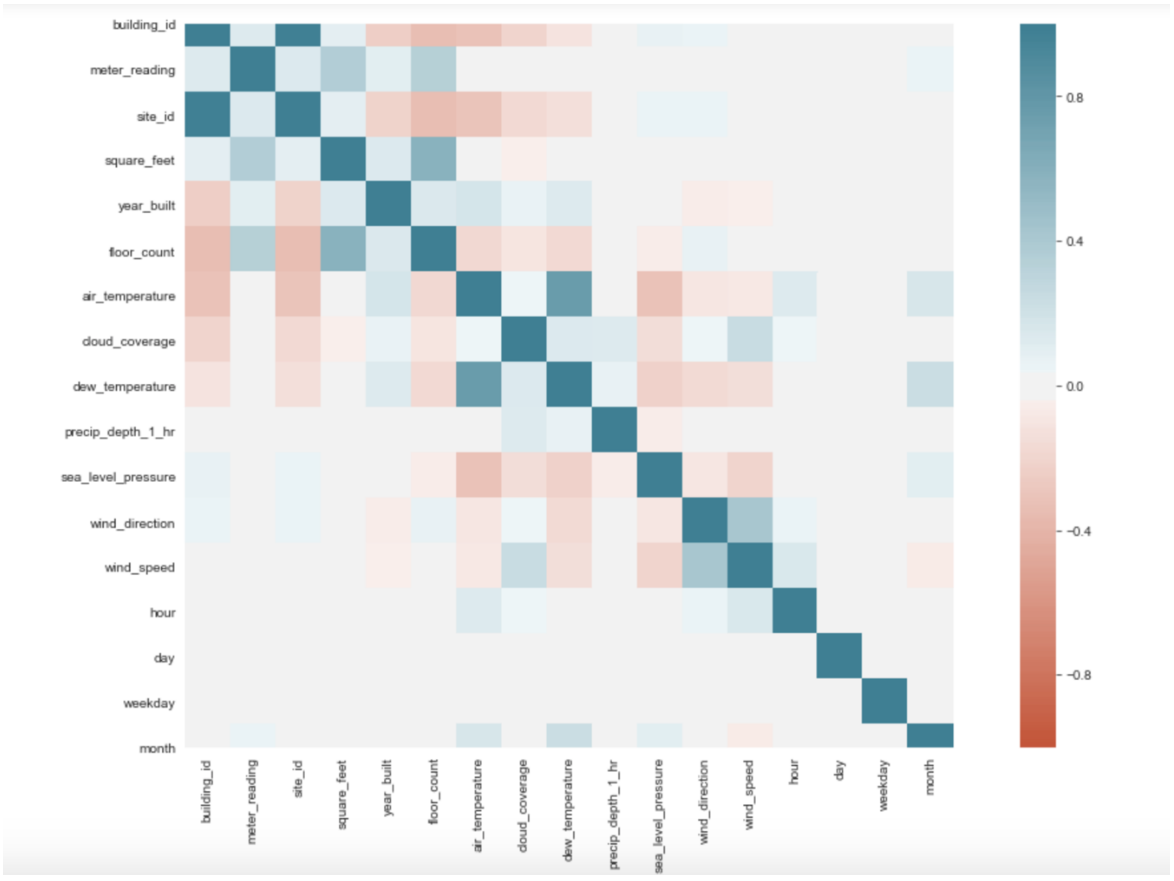
- Distribution of buildings for both training and test datasets is given as follows:



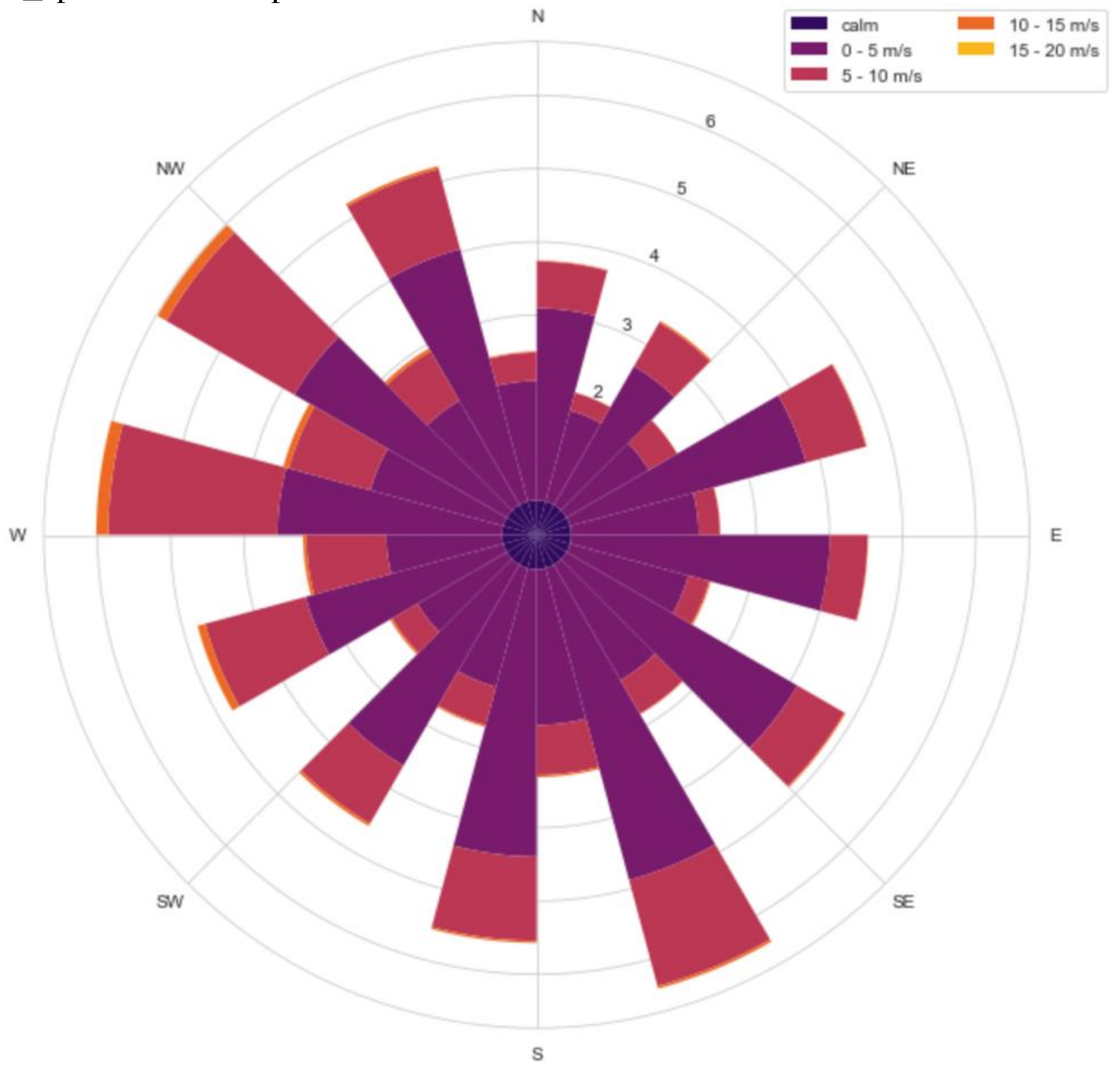
- Distribution of square feet area of buildings is given below:



- Correlation between meter readings and numeric value



- In heatmap plot wind direction and wind speed are highly correlated. So, it is good to plot together.
- wind_direction - Compass direction (0-360)
- wind_speed - Meters per second

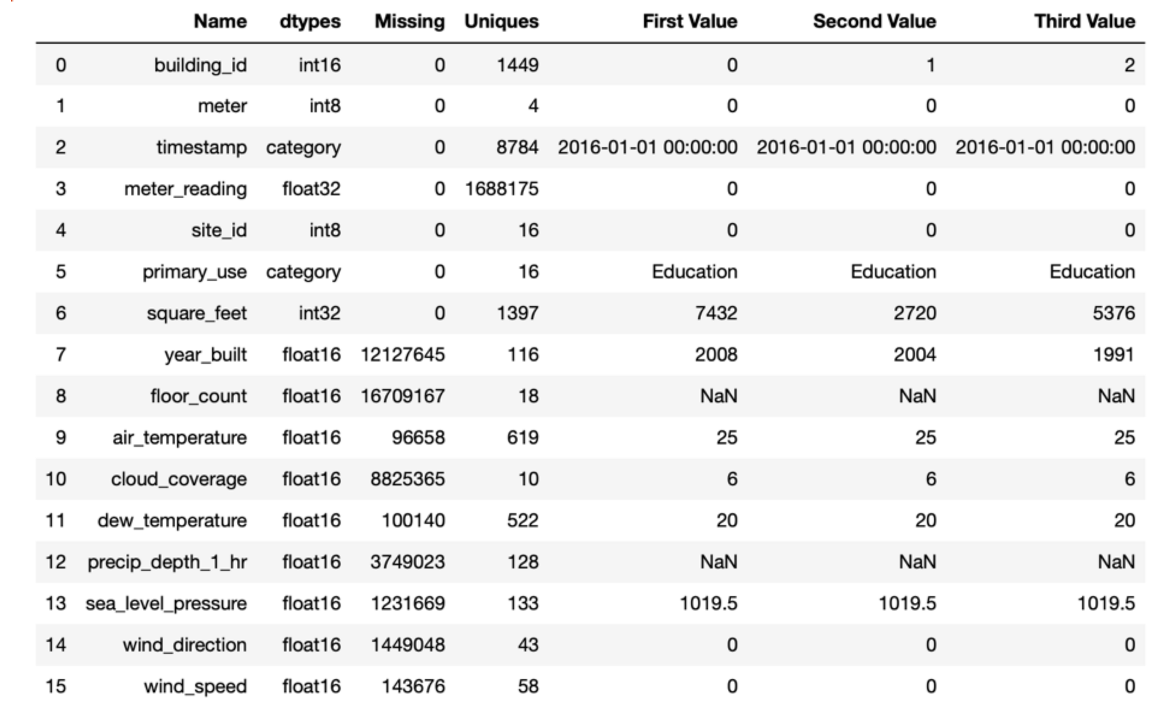


VI. Data Pre-processing:

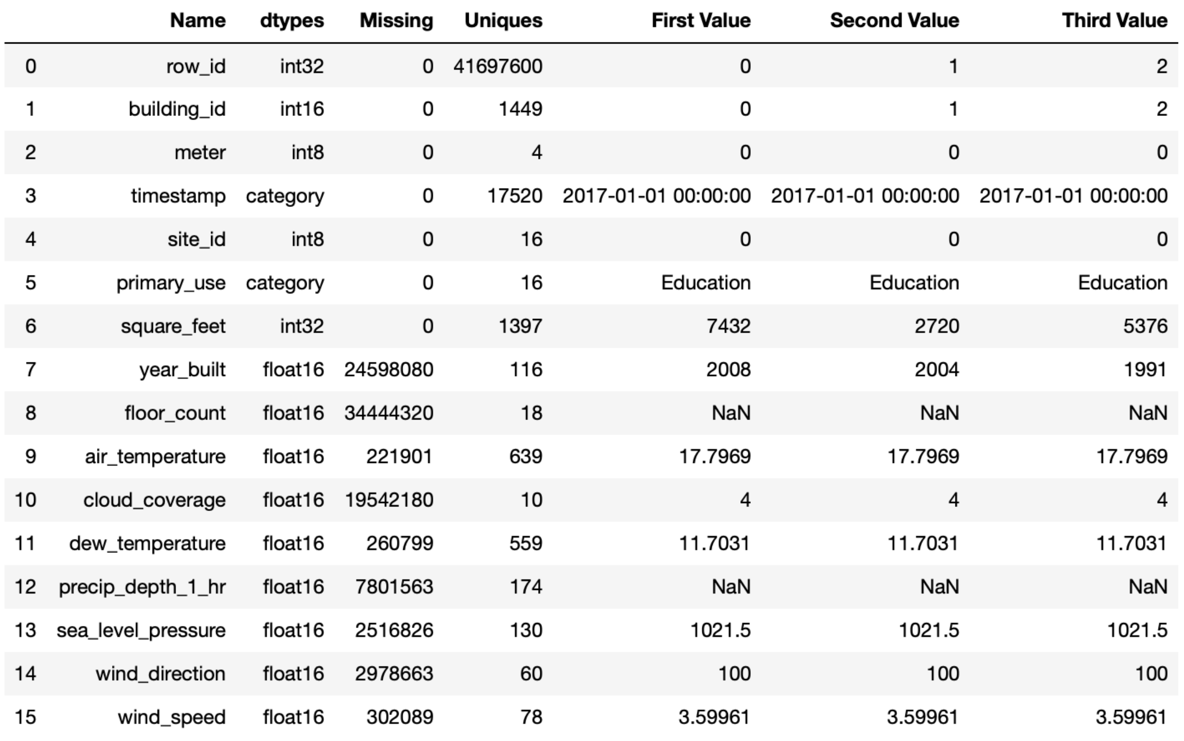
1) Since the dataset is extremely huge data preprocessing is critical in this project. Many outliers have to be processed carefully since we would want to reduce the redundant values and optimize the memory footprint for our model.

- **Delete Unnecessary data frames thereby improving the model performance and making it more robust**

Variable Description of train Data:



	Name	dtypes	Missing	Uniques	First Value	Second Value	Third Value
0	building_id	int16	0	1449	0	1	2
1	meter	int8	0	4	0	0	0
2	timestamp	category	0	8784	2016-01-01 00:00:00	2016-01-01 00:00:00	2016-01-01 00:00:00
3	meter_reading	float32	0	1688175	0	0	0
4	site_id	int8	0	16	0	0	0
5	primary_use	category	0	16	Education	Education	Education
6	square_feet	int32	0	1397	7432	2720	5376
7	year_built	float16	12127645	116	2008	2004	1991
8	floor_count	float16	16709167	18	NaN	NaN	NaN
9	air_temperature	float16	96658	619	25	25	25
10	cloud_coverage	float16	8825365	10	6	6	6
11	dew_temperature	float16	100140	522	20	20	20
12	precip_depth_1_hr	float16	3749023	128	NaN	NaN	NaN
13	sea_level_pressure	float16	1231669	133	1019.5	1019.5	1019.5
14	wind_direction	float16	1449048	43	0	0	0
15	wind_speed	float16	143676	58	0	0	0

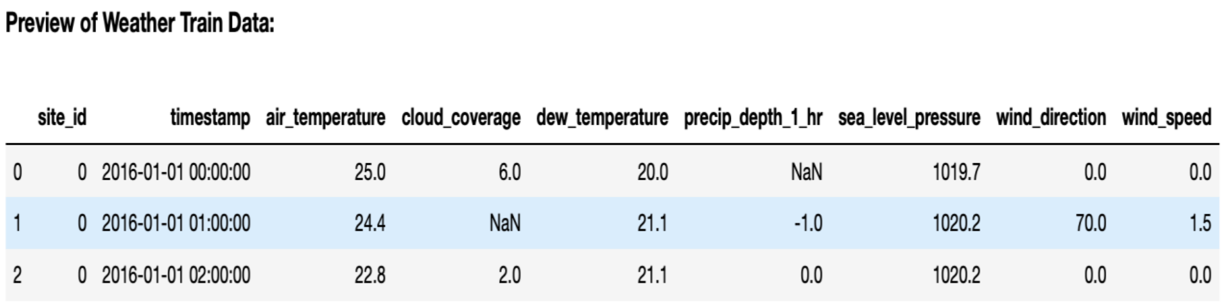


	Name	dtypes	Missing	Uniques	First Value	Second Value	Third Value
0	row_id	int32	0	41697600	0	1	2
1	building_id	int16	0	1449	0	1	2
2	meter	int8	0	4	0	0	0
3	timestamp	category	0	17520	2017-01-01 00:00:00	2017-01-01 00:00:00	2017-01-01 00:00:00
4	site_id	int8	0	16	0	0	0
5	primary_use	category	0	16	Education	Education	Education
6	square_feet	int32	0	1397	7432	2720	5376
7	year_built	float16	24598080	116	2008	2004	1991
8	floor_count	float16	34444320	18	NaN	NaN	NaN
9	air_temperature	float16	221901	639	17.7969	17.7969	17.7969
10	cloud_coverage	float16	19542180	10	4	4	4
11	dew_temperature	float16	260799	559	11.7031	11.7031	11.7031
12	precip_depth_1_hr	float16	7801563	174	NaN	NaN	NaN
13	sea_level_pressure	float16	2516826	130	1021.5	1021.5	1021.5
14	wind_direction	float16	2978663	60	100	100	100
15	wind_speed	float16	302089	78	3.59961	3.59961	3.59961

2) As evident from the above 2 graphs both the training as well as test data files have a lot of missing values in them. We need to address this situation by taking care of unnecessary redundant data as well Nans

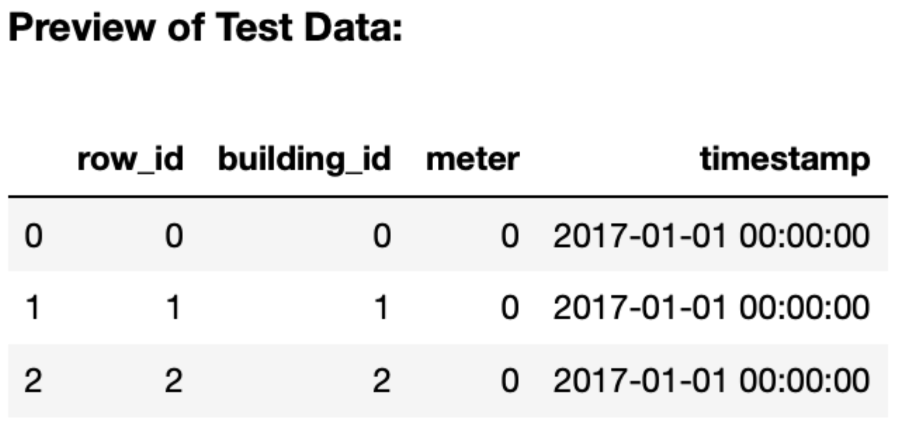
- **<u>Aligning Timestamps:</u>**

  i) The timestamp information in the train files must be aligned with respect to a same time such that it is better for our model to estimate the energy usages.



Preview of Weather Train Data:

	site_id	timestamp	air_temperature	cloud_coverage	dew_temperature	precip_depth_1_hr	sea_level_pressure	wind_direction	wind_speed
0	0	2016-01-01 00:00:00	25.0	6.0	20.0	NaN	1019.7	0.0	0.0
1	0	2016-01-01 01:00:00	24.4	NaN	21.1	-1.0	1020.2	70.0	1.5
2	0	2016-01-01 02:00:00	22.8	2.0	21.1	0.0	1020.2	0.0	0.0

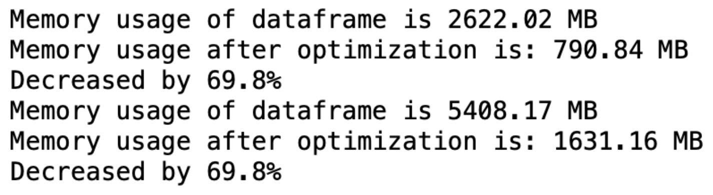


Preview of Test Data:

	row_id	building_id	meter	timestamp
0	0	0	0	2017-01-01 00:00:00
1	1	1	0	2017-01-01 00:00:00
2	2	2	0	2017-01-01 00:00:00

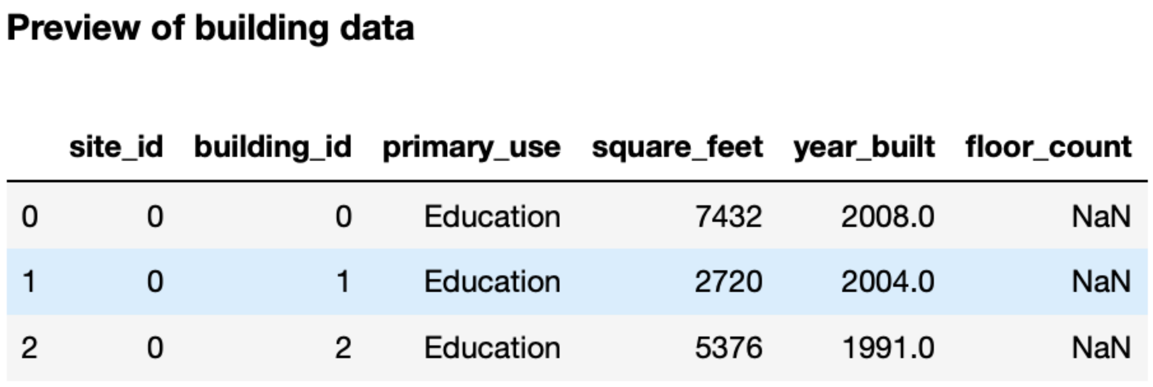
- **<u>Reduce Memory Usages:</u>**

  We also need to significantly reduce the memory footprint since this is especially a large dataset and hence, we need a roust performing model.



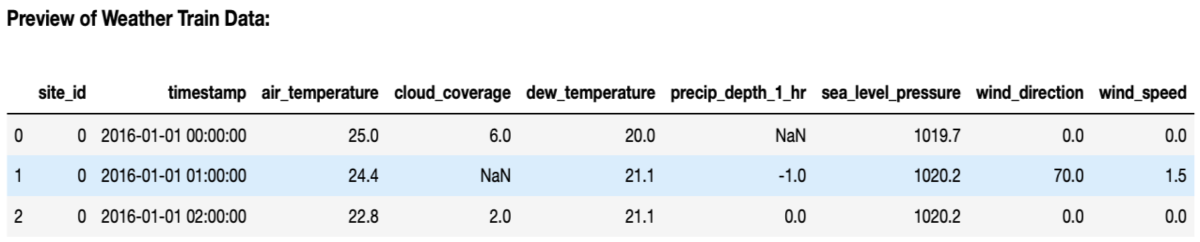
```
Memory usage of dataframe is 2622.02 MB
Memory usage after optimization is: 790.84 MB
Decreased by 69.8%
Memory usage of dataframe is 5408.17 MB
Memory usage after optimization is: 1631.16 MB
Decreased by 69.8%
```

- **<u>Fill Nans:</u>**



Preview of building data

	site_id	building_id	primary_use	square_feet	year_built	floor_count
0	0	0	Education	7432	2008.0	NaN
1	0	1	Education	2720	2004.0	NaN
2	0	2	Education	5376	1991.0	NaN

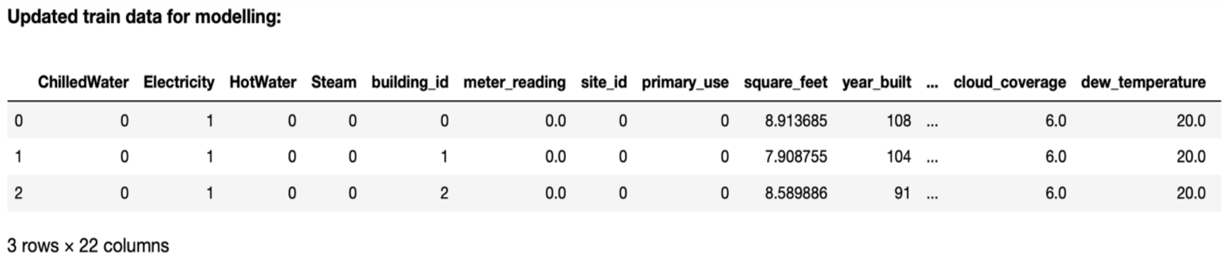


Preview of Weather Train Data:

	site_id	timestamp	air_temperature	cloud_coverage	dew_temperature	precip_depth_1_hr	sea_level_pressure	wind_direction	wind_speed
0	0	2016-01-01 00:00:00	25.0	6.0	20.0	NaN	1019.7	0.0	0.0
1	0	2016-01-01 01:00:00	24.4	NaN	21.1	-1.0	1020.2	70.0	1.5
2	0	2016-01-01 02:00:00	22.8	2.0	21.1	0.0	1020.2	0.0	0.0

i) The NaNs in weather data are filled by interpolation. There are also some redundant entities that must be removed

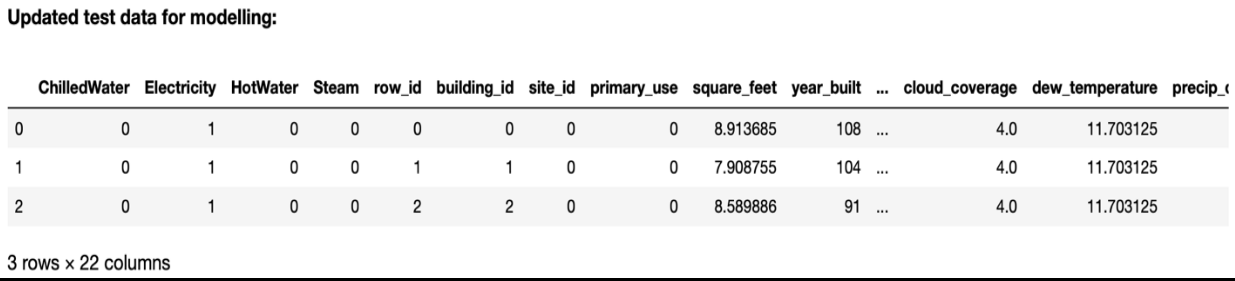
ii) The updated train and test data file along with their attributes are given below:



Updated train data for modelling:

	ChilledWater	Electricity	HotWater	Steam	building_id	meter_reading	site_id	primary_use	square_feet	year_built	...	cloud_coverage	dew_temperature
0	0	1	0	0	0	0.0	0	0	8.913685	108	...	6.0	20.0
1	0	1	0	0	1	0.0	0	0	7.908755	104	...	6.0	20.0
2	0	1	0	0	2	0.0	0	0	8.589886	91	...	6.0	20.0

3 rows × 22 columns

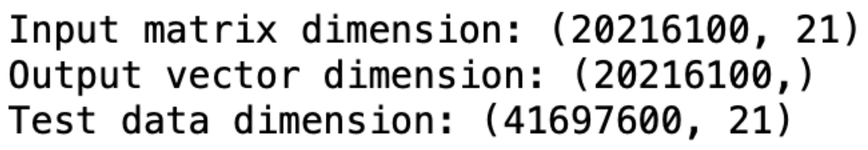


Updated test data for modelling:

	ChilledWater	Electricity	HotWater	Steam	row_id	building_id	site_id	primary_use	square_feet	year_built	...	cloud_coverage	dew_temperature	precip_
0	0	1	0	0	0	0	0	0	8.913685	108	...	4.0	11.703125	
1	0	1	0	0	1	1	0	0	7.908755	104	...	4.0	11.703125	
2	0	1	0	0	2	2	0	0	8.589886	91	...	4.0	11.703125	

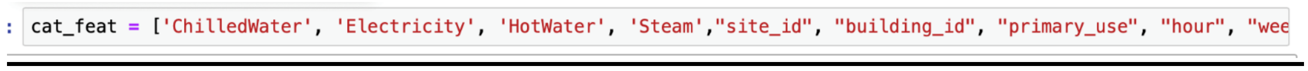
3 rows × 22 columns

iii) The data dimensions for modelling are given as follows:



```
Input matrix dimension: (20216100, 21)
Output vector dimension: (20216100,)
Test data dimension: (41697600, 21)
```

Also, the categorical features are:

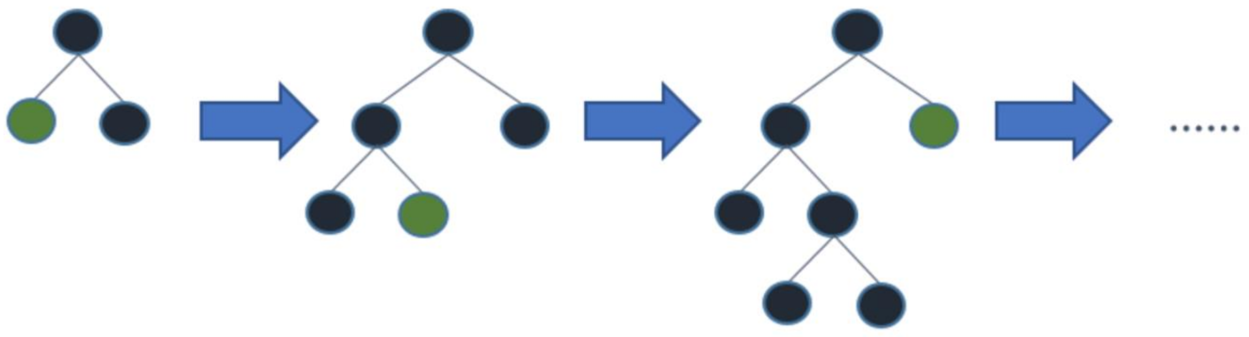


```
: cat_feat = ['ChilledWater', 'Electricity', 'HotWater', 'Steam',"site_id", "building_id", "primary_use", "hour", "wee
```

It is important to note that the categorical variables are obtained after encoding them and extracting the nominal variables for one hot encoding of test and train data

VII. <u>Light Gradient Boosting (LGB)</u>

- Light Gradient Boosting is a Machine Learning framework which basically uses tree-based learning for gradient boosting. It is different from other decision trees since the data scales leaf-wise (vertically) instead of level wise growth. The major factor here that benefits LGBM is that it chooses the leaf with max delta loss to grow. Thus, using LGB can help us reduce loss to a greater extent compared to any level wise algorithm.



Leaf-wise tree growth

- Another important reason that LGB is being utilized in many applications is the ability to use high end graphic cards to enable GPU based learning. Thus, it can greatly enhance performance especially such that we can use some of the powerful cards from Nvidia or AMD.
- Moreover, LGBM provides more than 100 parameters to be tweaked via its Python API such that many optimization techniques can be implemented as per the application. LGBM is ideally suited for very large datasets typically for data with 10,000+ rows of elements.

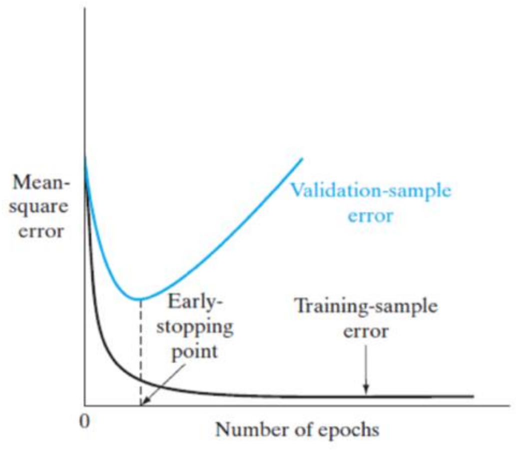
VIII. Key Considerations while developing any neural network:

- **Cross validation**

Cross validation technique is necessary to avoid the overfitting to the data while training the model. It assures the good generalization and improve performance of our model.

- **Early Stopping**

When we start training the model both training, and the test errors start decreasing up to a certain point. If we continue to train our model it will start over fitting to the training data, so the training error continues to decrease but the generalization error starts increasing due to overfitting. So, to prevent this we should stop training the model after this point called the early stopping point. This can be illustrated from the figure below:



- **Confusion Matrix**

Confusion Matrix gives us insight not only into the misclassifications being made by a classifier but also the types of errors that are being made. In this project, confusion matrix is observed for various MLP configurations used to observe the errors made. In this project the shape of confusion is 10 x 10 since the total number of classes are ten.

Thus, A confusion matrix consists of summary of probable predictions. Thus, confusion matrix is mainly used for error analysis to find False positives and True Positives in our classification problem. Correct classification data is always available at the diagonal while non diagonal elements show misclassified data i.e. one class represented falsely into some other classification label. We use it for error analysis. We get correctly classification data at the diagonal.

- **One hot Encoding**

One hot encoding can be described as a methodology in which categorical variables are transformed into a form that can be fed to Machine Learning tasks to do a better job in prediction. If the output of our model is going to show the probabilities of where an image should be categorized as a prediction, each element represents the predicting probability of each classes.

Model 1 Light Gradient Boosting Model

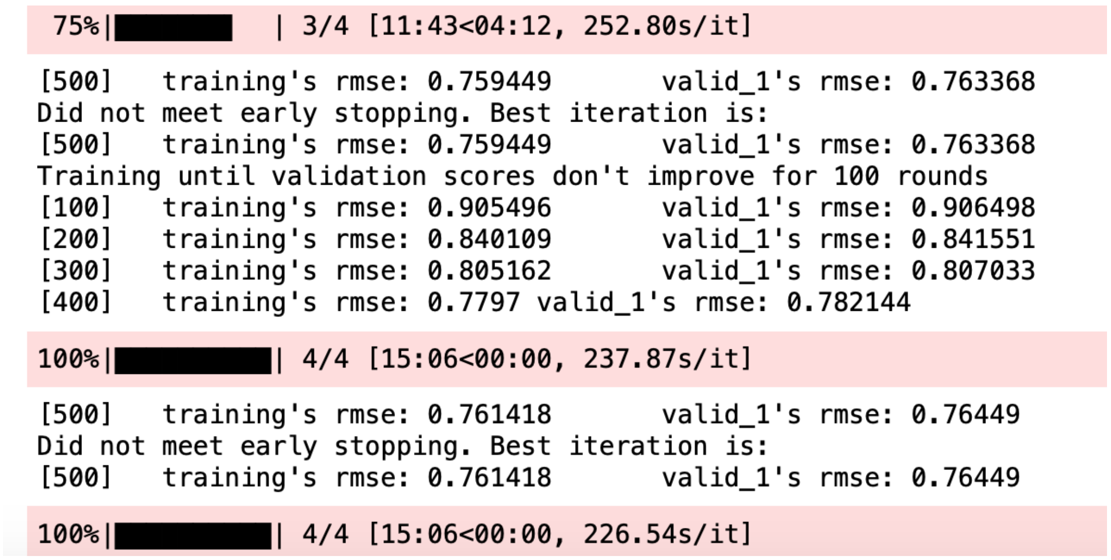
Default Network Architecture Settings:

1. Boosting Type- Gradient Boost
2. Objective- Regression
3. Learning rate – 0.3
4. Number of folds for train/rest split – 4
5. Subsample – 0.25
6. Seed=55
7. Early Stopping rounds = 100
8. Metric: RMSE
9. Num_leaves -20
10. Lambda L1 =1
11. Lambda L2 =1

Training RMSE	Validation RMSE
0.761	0.766

- The difference between the training accuracy and test accuracy can be called as overfitting. It happens when a ML model behaves worse on unseen data, i.e. the one it did not encounter in the training phase

  The results for training vs validation RMSE with early stopping utilized to avoid overfitting



```
 75%|███████▌  | 3/4 [11:43<04:12, 252.80s/it]

[500]   training's rmse: 0.759449       valid_1's rmse: 0.763368
Did not meet early stopping. Best iteration is:
[500]   training's rmse: 0.759449       valid_1's rmse: 0.763368
Training until validation scores don't improve for 100 rounds
[100]   training's rmse: 0.905496       valid_1's rmse: 0.906498
[200]   training's rmse: 0.840109       valid_1's rmse: 0.841551
[300]   training's rmse: 0.805162       valid_1's rmse: 0.807033
[400]   training's rmse: 0.7797 valid_1's rmse: 0.782144

100%|██████████| 4/4 [15:06<00:00, 237.87s/it]

[500]   training's rmse: 0.761418       valid_1's rmse: 0.76449
Did not meet early stopping. Best iteration is:
[500]   training's rmse: 0.761418       valid_1's rmse: 0.76449

100%|██████████| 4/4 [15:06<00:00, 226.54s/it]
```

<u>**Early Stopping:**</u>

We use early stopping rounds of 100 so that we can avoid overfitting condition while training with an iterative scheme such as the gradient descent

Early Stopping proves to be a vital regularization technique such that the model can better fit the data as it moves along each iteration

Values tried: 10 & 100

<u>**Boosting type: Gradient Boosting**</u>

Gradient Boosting helps the growth of data where there is mac loss involved. As a result, the model produces prediction techniques using decision trees mechanism. ‘gdb’ allows optimization of loss function

Other boosting types such as goss and dart were also tweaked but gdb appears to be a better general solution.

Learning Rate:

Learning rate is a hyper-parameter that controls the adjustment of weights of the network in context to loss gradient. If the value is lower, we travel slower along the downward slope. If learning rate is too high, gradient descent can overshoot the minimum. It may fail to converge, or even diverge. The learning rate has a huge impact on how quickly our model can converge to the respective local minima.
Values tried: 0.1, 0.3

Number of folds for train test split

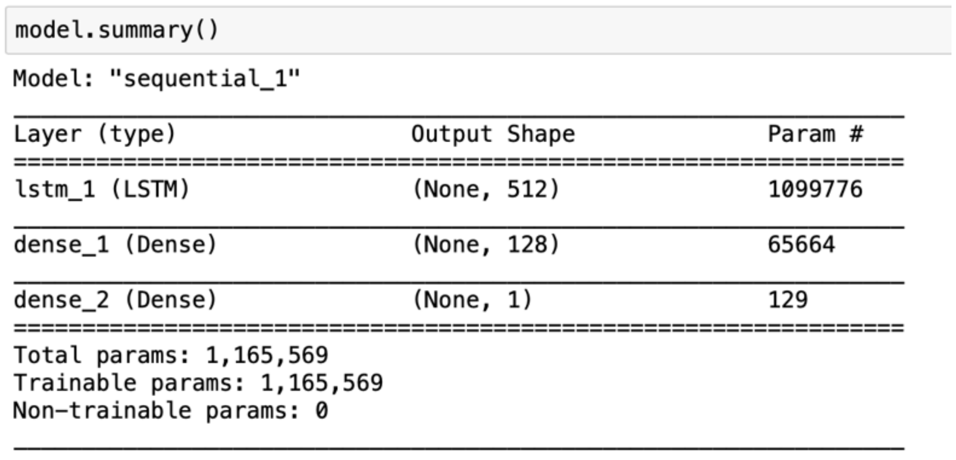
This involves splitting the data into parts and essentially training our model on those parts. Values tried: 2 and 4 folds

Model 2: LSTM (Neural Network)

Default Network Architecture Settings:

1. Number of Epochs=1
2. Train & validation split: Training size = 60%
3. Batch Size -1024
4. Number of Dense Layers = 1
5. Number of Neurons in the Input layer: 8
6. Activation Function in the Input layer: Relu
7. Activation Function in the Hidden layer: -----
8. Activation Function in the Output layer: Relu
9. Loss Function- MSE
10. Optimizer: Adam

Thus, many variations were utilized in the model such as changes in- number of epochs, activation functions, training set size, type of optimizers, loss type etc. were varied so as to come up with a good baseline model.



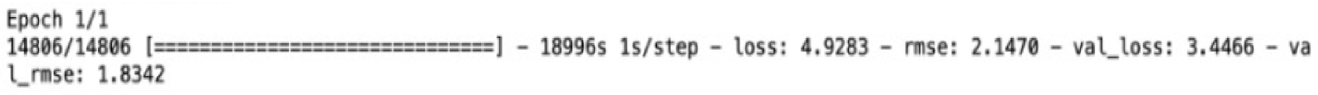
```
model.summary()

Model: "sequential_1"
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
lstm_1 (LSTM)                (None, 512)               1099776
_________________________________________________________________
dense_1 (Dense)              (None, 128)               65664
_________________________________________________________________
dense_2 (Dense)              (None, 1)                 129
=================================================================
Total params: 1,165,569
Trainable params: 1,165,569
Non-trainable params: 0
_________________________________________________________________
```

Following are the results:

Training RMSE: 2.14, Loss – 4.92

Validation RMSE: 1.83, Loss – 3.46



```
Epoch 1/1
14806/14806 [==============================] - 18996s 1s/step - loss: 4.9283 - rmse: 2.1470 - val_loss: 3.4466 - va
l_rmse: 1.8342
```

The training RMSE turned out to be 2.14 and can be improved further with more epochs and hyper parameter tuning as discussed in later part.

Following are the observations made by changing different parameters:

<u>**Changing the Number of Epoch's:**</u>

When the entire dataset is passed during both forward as well as backward during Back propagation it is termed as 1 complete Epoch. Thus, it is a metric using which the number of times all of the training vectors are used once to update the weights. The numbers of epochs are related to how diverse the data is. one epoch is not enough, and hence we need to pass the dataset over multiple times to the same neural network so as to get a better accuracy.
As the number of epochs increases, a greater number of times the weights are changed in the neural network and the curve goes from underfitting to optimal to overfitting curve.
Final model has number of epoch's set to 2. This is only to reduce the training time. However, with epochs set to around 25, an overall greater metric can be easily achieved.

Varying the Batch Size:

Batch size directly dictates the accuracy of the rough calculation of the error gradient. This is done during the training phase of neural network. Minibatch gradient descent, Stochastic & Batch, and are the 3 main variations of this learning algorithm.
When batch size= 1024, i.e.1024 samples from the training dataset will be used to find out the error gradient. This is done before the model weights are transformed. 1 training epoch means that the learning algorithm has made one pass through the training dataset.

Values Utilized: 64, 512 and 1024

Varying the Activation functions:

Activation function is an important parameter since it decides, whether a neuron should be fired or not This is done by computing weighted addition and summing bias to it. The prime motive of the activation function is to introduce non- linearity into the output of any neural network architecture.

The main reason behind using different activation functions is to analyze what is its effect on prediction and performance metrics. In this project three activation functions have been explored: sigmoid, hyperbolic tangent and ReLU. Each function has different characteristics which make them interesting to explore. Sigmoid function has a well-defined non-zero derivative everywhere, allowing gradient descent to make some progress at every step. Tanh function varies smoothly between [-1, 1] whereas ReLU remains dormant till [0, 0] and then varies linearly.

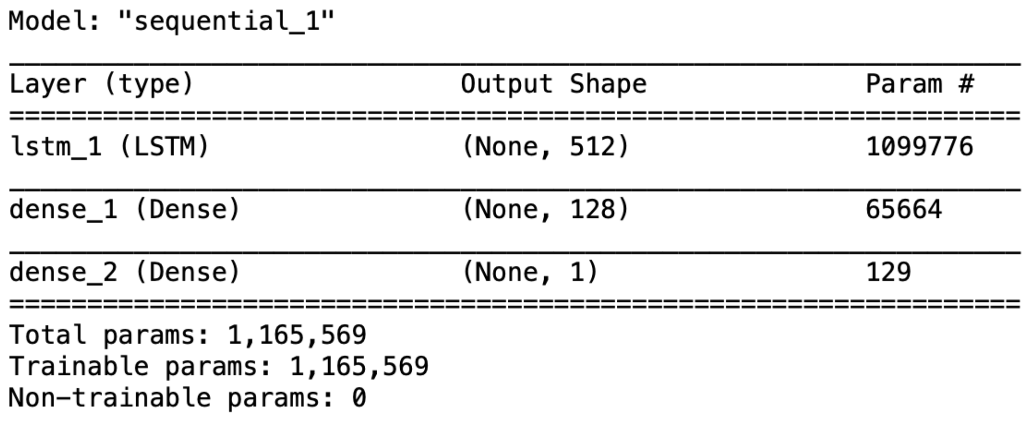
Advantages of using ReLUs- reduced probability of vanishing gradient.
This is due to the fact that the gradient has a constant value in contrast, the gradient of sigmoid becomes increasingly small as the absolute value of x increases. The constant gradient of ReLUs results in faster learning.
We use Relu in all iterations with other parameters changed

Varying the Number of Neurons:

A large number of neurons in the hidden layer can drastically increase the time to train the network. More importantly the amount of training time can inflate to the point where it is not possible to properly train the neural network.

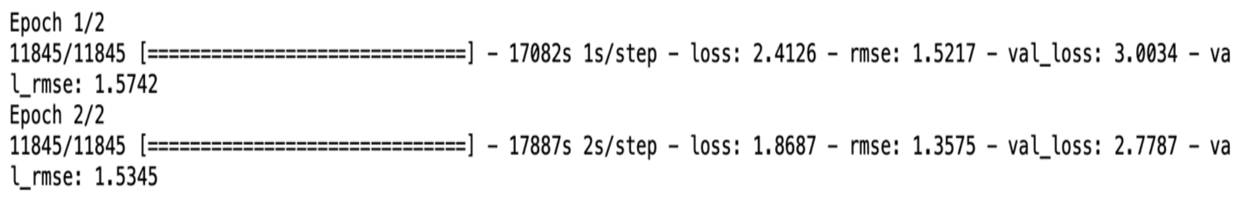
Underfitting occurs when there are too few neurons in the hidden layers to adequately detect the signals in a complicated data set.
Using too many neurons in the hidden layers can result in several problems. Thus, it can be said that Overfitting occurs when the neural network has so much information processing capacity that the limited amount of information contained in the training set is not enough to train all of the neurons in the hidden layers.

selection of number of neurons for any particular architecture will be better if we use trial and error out method. This is due to the fact that if we equip trial and error method with 256 neurons, we can obtain the desired highest accuracy.
The RMSE metrics after changing the number of neurons to 512 in the input layer along with adding a hidden layer with 128 neurons is given as follows:



```
Model: "sequential_1"
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
lstm_1 (LSTM)                (None, 512)               1099776
_________________________________________________________________
dense_1 (Dense)              (None, 128)               65664
_________________________________________________________________
dense_2 (Dense)              (None, 1)                 129
=================================================================
Total params: 1,165,569
Trainable params: 1,165,569
Non-trainable params: 0
```

<u>**RMSE Results**</u>



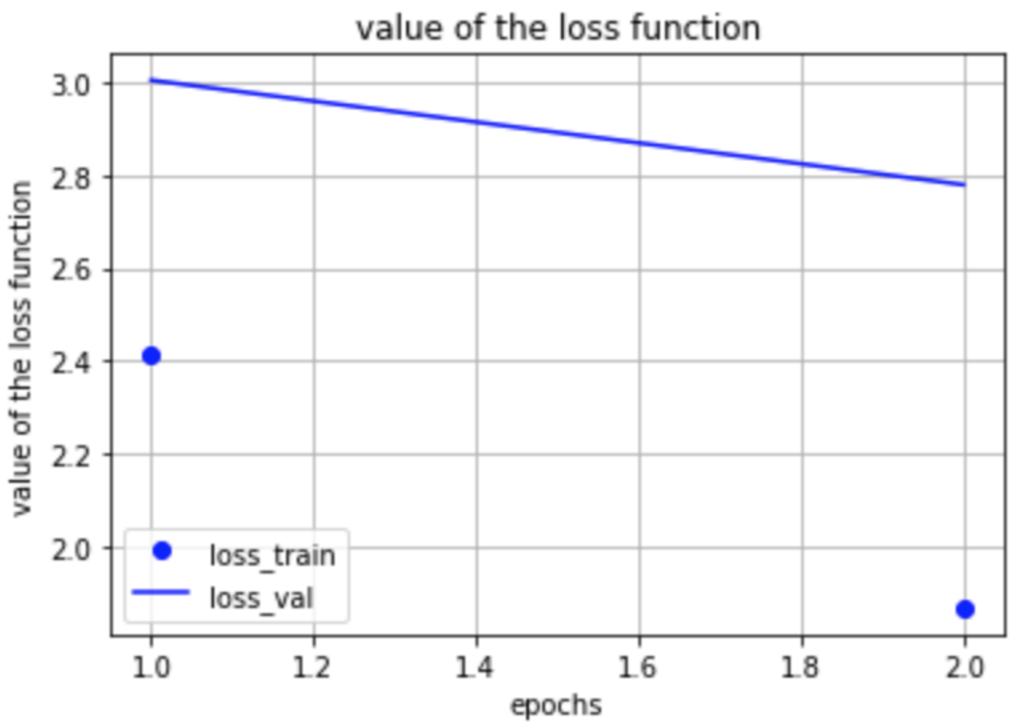
```
Epoch 1/2
11845/11845 [==============================] - 17082s 1s/step - loss: 2.4126 - rmse: 1.5217 - val_loss: 3.0034 - va
l_rmse: 1.5742
Epoch 2/2
11845/11845 [==============================] - 17887s 2s/step - loss: 1.8687 - rmse: 1.3575 - val_loss: 2.7787 - va
l_rmse: 1.5345
```

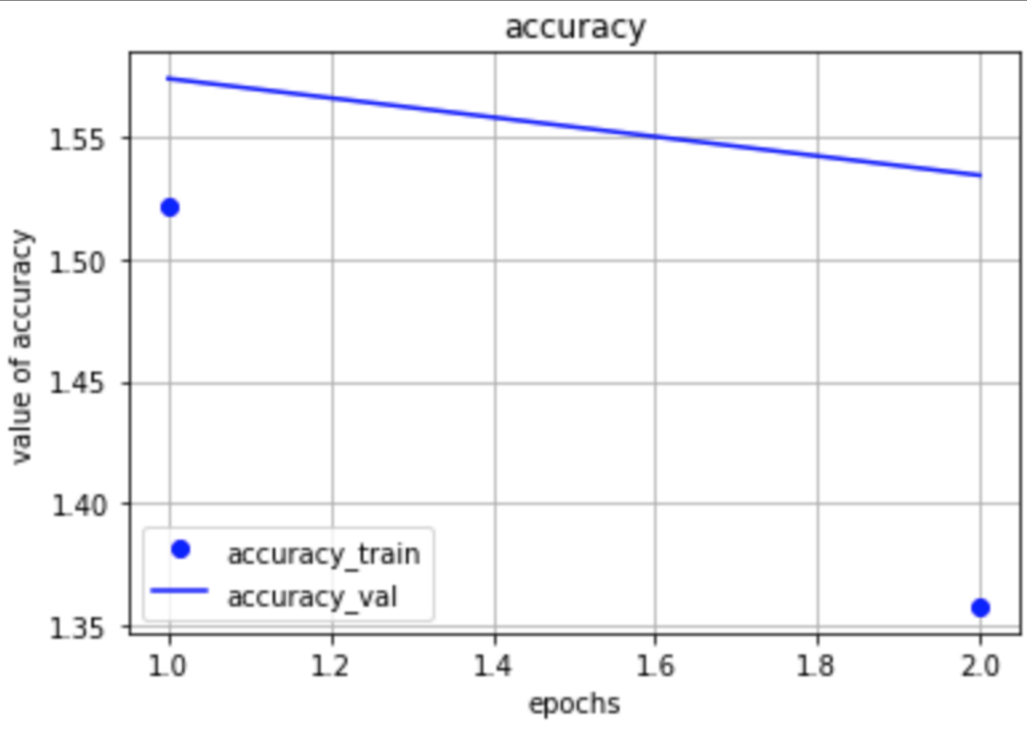
Training RMSE: 1.35; Loss- 1.86

Validation RMSE: 1.53; Loss- 2.77

The model loss does decrease when the number of neurons in the hidden layers increases. However, there is a limit to this, and hence too many neurons must be avoided to make sure that the test accuracy is sufficiently high.

Note: The graph below doesn't have a history(curve) for loss train values since it is being trained for very few epochs. If we keep relatively high epochs, we will get a curve with history over certain number of iterations





<u>Varying the Learning rate:</u>

Learning rate is a hyper-parameter that controls the adjustment of weights of the network in context to loss gradient. If the value is lower, we travel slower along the downward slope. If learning rate is too high, gradient descent can overshoot the minimum. It may fail to converge, or even diverge. The learning rate has a huge impact on how quickly our model can converge to the respective local minima.

On the other hand, if the learning rate that we are utilizing for our neural architecture is too small then this can further contribute to the process being stuck. However, a larger learning rate always ensures that the resulting model can converge quickly. A learning rate that is too small can lead to the process being stuck while on the other hand a learning rate that is too big can cause the model to converge too quickly.
The challenge of training neural networks involves carefully selecting the learning rate.

<u>Varying the training set size.</u>

When we change the training set size it has a significant performance effect on our system. In practice more training samples always lead to a better accuracy for both train as well as test cases. But this isn’t guaranteed every single time.
In this implementation we have divided the data such that 60% belongs to training while the rest for validation

<u>Varying the type of Optimizer:</u>

Theoretically Adam should be better than a lot of other optimizers that are being compared above since it computes individual learning rates for different parameters. But the performance does depend a lot on other parameters too. Also, batch size affects Adam optimizer. Common batch sizes 16, 32, and 64 can be used. Results show that there is a sweet spot for batch size, where a model performs best.

<u>Effect of loss functions:</u>

MSE: It is a type of regression loss in which the sum of distances between target and predicted values is squared off.

Reason LSTM performs worse:

RNNs make classifications based heavily on context. An RNN will build up an internal state that is an abstract representation of what it has seen so far and use this to make a prediction. The hidden state is usually initialized to all 0s, and it has to build up to something meaningful, so the predictions towards the beginning of a sequence are much less accurate.

RNNs make classifications based heavily on context. An RNN will build up an internal state that is an abstract representation of what it has seen so far and use this to make a prediction. The hidden state is usually initialized to all 0s, and it has to build up to something meaningful, so the predictions towards the beginning of a sequence are much less accurate.

Model 3: Fully Connected (FC) Neural network

Default Network Architecture Settings:

1. Number of Epochs=2
2. Number of dense layers =4
3. Dropout after each layer: Yes (0.1)
4. Early Stopping- Yes
5. Learning rate: 0.001
6. Loss- mse_loss
7. Number of folds for train test split: 2
8. Batch Size - 64
9. Optimizer: Adam
10. Activation Function in the Input layer: Relu
11. Activation Function in the Output layer: Relu

We use Keras Embeddings Layer in this Neural Network.

The embedding layer is basically a matrix which can be considered a transformation from discrete and sparse 1-hot-vector into a continuous and dense latent space. To save the computation, we don't actually do the matrix multiplication, as it is redundant in the case of 1-hot-vectors.
So, if we have a vocabulary size of 5000, as input dimension - and we want to find a 256-dimension output representation of it - we will have a (5000,256) shape matrix, which should multiply the 1-hot vector representation to get the latent vector.

Thus, many variations were utilized in the model such as changes in- number of epochs, activation functions, training set size, type of optimizers, loss type etc. were varied so as to come up with a good baseline model.

Following are the results for the baseline model:

Training RMSE- 1.251, Validation RMSE: 1.10
Training Loss- 1.61, Validation Loss: 1.46

Varying the Learning rate:

Learning rate is a hyper-parameter that controls the adjustment of weights of the network in context to loss gradient. If the value is lower, we travel slower along the downward slope. If learning rate is too high, gradient descent can overshoot the minimum. It may fail to converge, or even diverge. The learning rate has a huge impact on how quickly our model can converge to the respective local minima.
Learning rate in one of the most crucial hyperparameter that needs to be tweaked. This is because it directly corresponds to how quick and efficiently the corresponding model can adapt to our classification job. If we equip smaller learning rate, we might need a greater number of training Epoch's. This is since we any type of changes are made the weights get updated and as a result if our learning rate is large enough, we can move across it in fast succession thereby utilizing fewer number of epoch's during the training phase. On the other hand, if the learning rate that we are utilizing for our neural architecture is too small then this can further contribute to the process being stuck.
However, a larger learning rate always ensures that the resulting model can converge quickly. A learning rate that is too small can lead to the process being stuck while on the other hand a learning rate that is too big can cause the model to converge too quickly. The challenge of training neural networks involves carefully selecting the learning rate. Higher learning rate yields a better result, although marginally.
We change the learning rate from 0.01 to 0.1 and observe the results. At this stage both the learning rate and # of epochs are changed, and the results are shown below

Varying the Number of Epoch's

Epoch can be defined as a measure of the number of times the training vectors are utilized in order to update the corresponding weights. The numbers of epochs represent how different the data is. In our implementation following results are obtained: When the number of Epoch's are increased the accuracy increases by a lot especially where the training accuracy benefits are evident.

Thus, there is a direct correlation between the accuracy and loss when we tweak number of epochs. As the number of epochs increases, a greater number of times the weight is changed in the neural network and the curve goes from underfitting to optimal to overfitting curve. The numbers of epochs are related to how diverse your data is. When the entire dataset is passed during both forward as well as backward during Back propagation it is termed as 1 complete Epoch. Thus, it is a metric using which the number of times all of the training vectors are used once to update the weights. The numbers of epochs are related to how diverse the data is. one epoch is not enough, and hence we need to pass the dataset over multiple times to the same neural network so as to get a better accuracy.

The following are the results with all previous configuration settings and epochs set to 1. Lowering # of epochs results in a **performance degradation i.e. Higher RMSE (about 2x degradation when we reduce the epoch from 2 to 1)**

Model	RMSE (training)	Loss(training)	RMSE Valid	Loss valid
1 epoch	2.09	4.41	2.04	4.38
2 epochs	1.21	1.61	1.1	1.4

Varying the training set size.

When we change the training set size it has a significant performance effect on our system. In practice more training samples always lead to a better accuracy for both train as well as test cases. But this isn't guaranteed every single time.

Thus, it is evident that since we have a higher training data in case where we split the data in training and validation into equal half. The model also is lower in case of more training data. This is not always the case as lots of training data may lead to the model performing better on the train set but not necessarily on the test. Thus, it can be concluded that we validation set reduces i.e. more training set hence it likely that we will get a better accuracy both in terms of test as well as train.

In this implementation we separate the train and set split by using 4 or 2 folds and then training our entire data over it

Changing activation functions in the output layer:

The main task of the activation function is to put non- linearity relation into the output of a neuron. It decides where a neuron should be activated or not. Activation function is an important parameter since it decides, whether a neuron should be fired or not This is done by computing weighted addition and summing bias to it. The prime motive of the activation function is to introduce non- linearity into the output of any neural network architecture.

The main reason behind using different activation functions is to analyze what is its effect on prediction and performance metrics. sigmoid, hyperbolic tangent Softmax and ReLU are some of the choices available. Each function has different characteristics which make them interesting to explore. Sigmoid function has a well-defined non-zero derivative everywhere, allowing gradient descent to make some progress at every step. Tanh function varies smoothly between [-1, 1] whereas ReLU remains dormant till [0, 0] and then varies linearly.

RELU: Mainly used in hidden layers of Neural network. RELU is less computationally expensive than tanh and sigmoid since it focusses on simpler mathematical operations. Only a few neurons are activated making the network sparse making it efficient and easy for computation.
RELU has an advantage that it can learn much quicker compared to other activation functions. it is a better idea to use SoftMax at the output layer such that we get classification as a probabilistic distribution because we are concerned with multi classification

Sigmoid: The range of Sigmoid function is between [0,1] and the result can be calculated to be 1 if value > 0.5 and 0 if value< 0.5. Sigmoid happens to be a better choice especially in cases where binary classification is concerned

Tanh: Helpful in hidden layers of a neural network as its values lies in [-1 to 1] and the average/mean for hidden layer is almost 0 or very close to it, as a result it helps in positioning the data at the center by bringing mean close to 0. As a result, the learning for the next layer is much easier.

Advantages of using ReLUs- reduced probability of vanishing gradient.
This is due to the fact that the gradient has a constant value in contrast, the gradient of sigmoid becomes increasingly small as the absolute value of x increases. The constant gradient of ReLUs results in faster learning.

Advantages of using Softmax: At the end of a network, we can either use nothing (logits) and get a multi parameter regression or a sigmoid and get a number between 0 and 1 for each output, this is useful when we have several possible and independent answers or use SoftMax where all the outputs sum to 1, this useful when we want one best answer, we get a probability distribution.
In one to one comparison if we are well aware of our function, Softmax seems to be a better choice although a lot of other parameters also contribute to a great extent.

Additionally, Relu/SoftMax is in general a better choice in the hidden layer since it is much faster computationally as an activation function. RELU/Softmax learns much faster than sigmoid and Tanh function. Hence, we observe that SoftMax got the highest accuracy

In our approach we tried using sigmoid but then switched back to Relu throughout as it's an ideal choice in our case.

Effect of Batch Normalization:

Batch Normalization is another method to regularize a neural network. On top of a regularizing effect, batch normalization also gives your network a resistance to vanishing gradient during training. This can decrease training time and result in better performance. Batch normalization is used along with dropout as a regularization technique. It is already implemented in the above implementations.

We experimented with removing the Batch Normalization

Changing loss functions:

Neural networks require a loss function to calculate the model error as the training takes place with the help of an optimization process

The loss function has an important task in that it must filter all parts of the model down into a single number in such a way that betterments in that number are a sign of a better performing model

We use 'mse' throughout

Varying the Batch Size:

Batch size is one of the most important hyperparameters to tune in neural network systems. Experts often want to use a larger batch size to train their model as it allows computational fast processing mechanisms using the parallelism of GPUs. Although, it is also true that too big of a batch size will contribute to poor generalization.

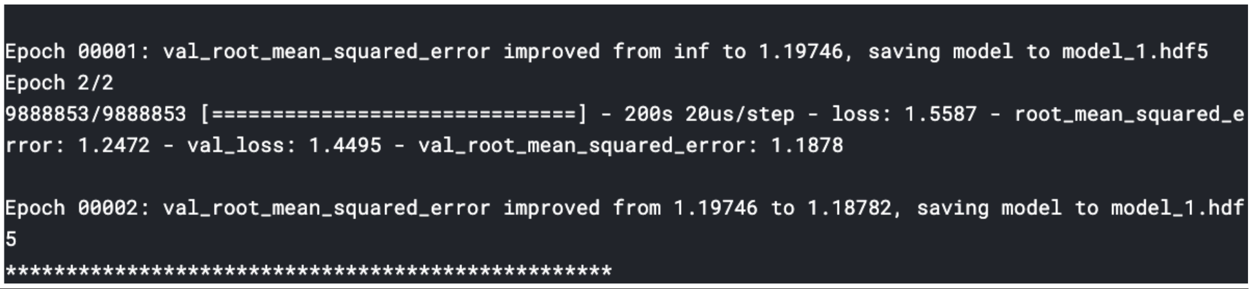
Batch size directly dictates the accuracy of the rough calculation of the error gradient. This is done during the training phase of neural network. Minibatch gradient descent, Stochastic & Batch, and are the 3 main variations of this learning algorithm.

When batch size= 64, 64 samples from the training dataset will be used to find out the error gradient. This is done before the model weights are transformed. 1 training epoch means that the learning algorithm has made one pass through the training dataset.
Following results are obtained when we change Batch size in various configurations:
The results are as follows:

Batch Size	1024	64
Epoch	1	1
RMSE (train)	1.24	2.09
Loss(train)	1.5	4.41

Some important things to consider:

- lower asymptotic test accuracy can be obtained with higher batch sizes
- lost test accuracy can be recovered from a larger batch size by increasing the learning rate
- If the number of sample's are similar, then larger batch sizes make larger gradient steps than the corresponding smaller batch sizes.



```
Epoch 00001: val_root_mean_squared_error improved from inf to 1.19746, saving model to model_1.hdf5
Epoch 2/2
9888853/9888853 [==============================] - 200s 20us/step - loss: 1.5587 - root_mean_squared_e
rror: 1.2472 - val_loss: 1.4495 - val_root_mean_squared_error: 1.1878

Epoch 00002: val_root_mean_squared_error improved from 1.19746 to 1.18782, saving model to model_1.hdf
5
**************************************************
```

Varying Neurons in the dense layer:

One of the most critical design parameters is deciding on the # of neurons in a hidden layer architecture of a typical neural network. These layers are termed hidden since we never have access to the weights present inside them, i.e. they represent an intermediate layer between the input layer and output layer. A large number of neurons in the hidden layer can drastically increase the time to train the network. More importantly the amount of training time can inflate to the point where it is not possible to properly train the neural network.

Underfitting occurs when there are too few neurons in the hidden layers to adequately detect the signals in a complicated data set.

Using too many neurons in the hidden layers can result in several problems. Thus, it can be said that Overfitting occurs when the neural network has so much information processing capacity that the limited amount of information contained in the training set is not enough to train all of the neurons in the hidden layers. Selection of number of neurons for any particular architecture will be better if we use trial and error out method. This is due to the fact that if we equip trial and error method with more neurons, we can obtain the desired highest accuracy. Also, the loss lowers down by a little, although other parameters need to be tweaked to observe significant performance metric.

Apart from the default config of 64-32-32-16 we also tried 256-128-128-128-64. This is included in the result above

Conclusion:
Thus, various performance enhancements were observed after tweaking many parameters. Below represents our best Model Architecture Settings:

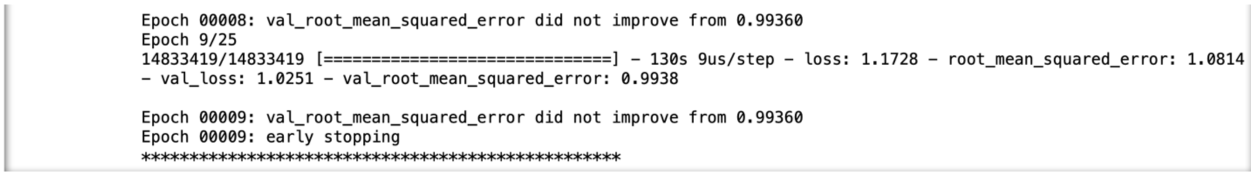
Default Network Architecture Settings:

1. Number of Epochs=**25 (ideally should be >50; but will consume training time)**
2. Number of dense layers =**4**
3. Dropout after each layer: Yes (**0.1**)
4. Learning rate: **0.001**
5. Loss- mse_loss
6. Number of folds for train test split: **4**
7. Batch Size – **1024 (if using 1024 need to tweak learning rate too)**
8. Optimizer: **Adam**

9. Early Stopping- Yes
10. Activation Function in the Input layer: **Relu**
11. Activation Function in the Output layer: **Relu**

Train RMSE=1.08; Validation RMSE =0.993

Train Loss=1.17; Validation Loss=1.02



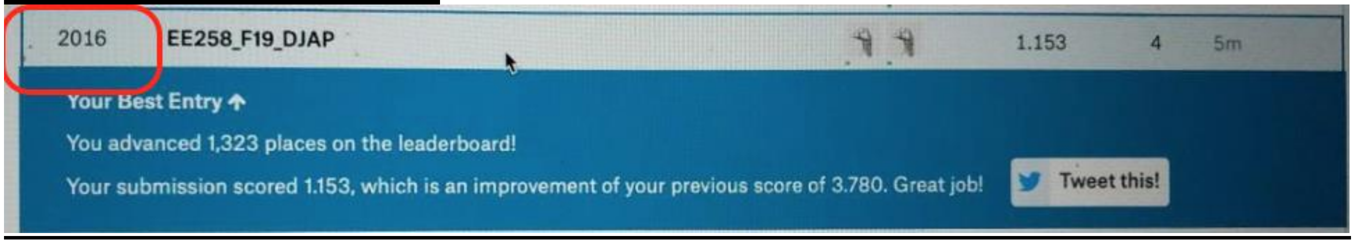
```
Epoch 00008: val_root_mean_squared_error did not improve from 0.99360
Epoch 9/25
14833419/14833419 [==============================] - 130s 9us/step - loss: 1.1728 - root_mean_squared_error: 1.0814
- val_loss: 1.0251 - val_root_mean_squared_error: 0.9938

Epoch 00009: val_root_mean_squared_error did not improve from 0.99360
Epoch 00009: early stopping
**************************************************
```

Observations:

- We observed that number of neurons play a huge role, but it is also an important design criterion that too many neurons wouldn't contribute to an optimum design mechanism.
- To prevent overfitting dropout must be implemented. This can be done by adding dropout as a parameter
- Batch size plays a critical role in the training accurate as it affects the learning rate of the model. Hence in this we prefer a batch size of 1024
- Validation accuracy gets a huge degradation after making changes to batch size. In theory however with large number of epoch's and dropout along with batch normalization should yield best performing model which is being calculated at epochs=25 along with other design tweaks.

Kaggle Rank: 2016



2016 EE258_F19_DJAP 1.153 4 5m

Your Best Entry

You advanced 1,323 places on the leaderboard!

Your submission scored 1.153, which is an improvement of your previous score of 3.780. Great job!

Issues/ Things that I wanted to try:

- Play with some more parameters and multiple values. Although I tweaked around 15 values, I could only compare 2 values against each other for example in terms of epochs I did 2 and 12. Better things can be tried out such as cross comparisons between 5 types of optimizers batch size, learning rate etc. Same goes for other hyper parameters.
- # of epoch's plays a critical role but due to time constraints and Kaggle kernel automatically shutting off randomly I couldn't use their cloud-based GPU's and hence running everything locally took a whole lot time. Thus, I had to change multiple parameters at once instance since running for each with epoch>10 would be time consuming.