COSI101A Final Project
ASHRAE - Great Energy Predictor III

Ruiyang Hu, Hanyu Song, Yicheng Tao
Brandeis University

January 7, 2020

Abstract

To predict a building's raw energy consumption is important in that it could contrast and show whether a corporation's effort in energy reduction has been effective. This problem can be solved using supervised machine learning algorithms if provided with enough data. Our team focused on applying two widely-applied algorithms: Neural Network and LightGBM and we have obtained meaningful results. In this report, we will first walk you through the problem, then introduce our two distinctive approaches, and end it with our conclusion and credits. Throughout this report, you can see our changes in data manipulating methods and improvements in model tuning. We believe this project has granted us an invaluable experience.

Contents

Introduction

The purpose of our project is to solve a real-world problem that carries concrete economical and environmental impacts: to predict buildings' energy consumption. There are many factors that exhibit the importance of knowing the energy consumption of a building; One vital reason, which is also the reason that ASHARE company started this Kaggle project, is to use the actual energy a building consumes and contrast it against its consumption after the company spends millions on energy reduction, in order to test the efficiency of this expenditure.

Figure 1: AShare, the company that's sponsoring this Kaggle Project.

This project is very large in scale: the datasets contain tens of millions of rows of data, with each row consisting of dozens of building features possibly related to its energy consumption, such as weather, temperature, building's characters, and the kind of energy use we are predicting.Of course, there exists lost data, outliers and all kinds of trivial and non-trivial issues that may exist in a dataset.

In the next few sections, we are going to dissect this problem step by step. We will have a large proportion of this report focusing on two distinction approaches: Neural Network Algorithm and LightGBM (Gradient Boosting Machine) Decision Tree Algorithm. While they share similar data pre processing methods, the mechanisms behind these two methods are different in nature, so are the results obtained. Then, we will talk about our take-aways from this projects, including new skills learned, interesting methods found, and possible tips that may help our way out in our future attempt in Kaggle projects. Lastly, we will explicitly describe our contributions in each process.

Neural Network Approach

Neural Network Model Explained

Artificial Neural Network is a computing system vaguely inspired by biological neural networks that constitute animal brain. It consists of multiple neural layers where each layer contains a certain number of neurons, often associated with the number of features inputted into the algorithm. The gist of a neural network is that each neuron will carry a weight used to lever input features. When the input and output data are given, the network will gradually assign each neuron the optimal weight that can minimize the mean-squared error, with the output being the weighted sum of inputs:

$$y_i = \varphi_i(\Sigma_{j=1}^{n^i} w_j^i z_j^i + b^i) \tag{1}$$

where n_i is the total incoming connections, z_i is the input, w_i is the weight, b_i is the bias, and $\varphi_i(\cdot)$ is the activation function at the i-th node to limits the amplitude of the output the node into a certain range [3].

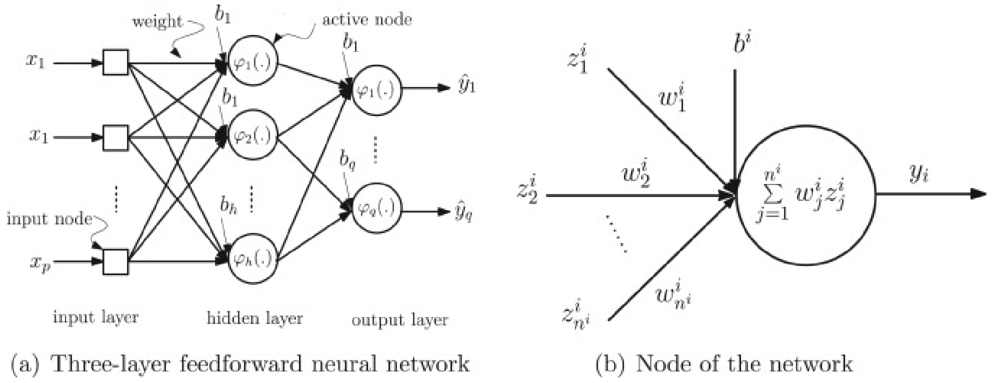


Figure 2: the neural networks have a specific structural architecture in which the nodes at a layer have forward connections from the nodes at its previous layer (Fig. 2(a)). A neuron is capable of processing information coming through the connection weights (Fig. 2(b)). [3]

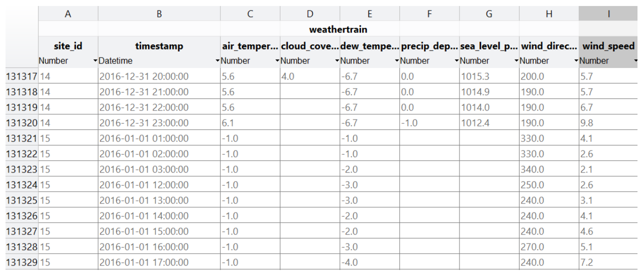
The neural network algorithm is widely applied. In this case, with the vast amount of data provided, we are using the supervised neural network, in which we use sets of paired inputs and desired outputs to train the model through the neural layers, then apply the trained algorithm onto the test file.

Data Pre-processing

We found it necessary to do some data preprocessing so that a "train.csv" file and a "test.csv" could be built from the smaller datasets given, in order to train and test our model. The first step was to merge "building_metadata.csv" "weather train.csv" and "train.csv" according to "side_id" "building_id" and "timestamp" so that we would have a training dataset that included all raw features we would use in our model later, since we plan to input all features into the neural network model at once and build neural layers based on that number of features. To facilitate the concatenating process, we used the "join" method in MATLAB.

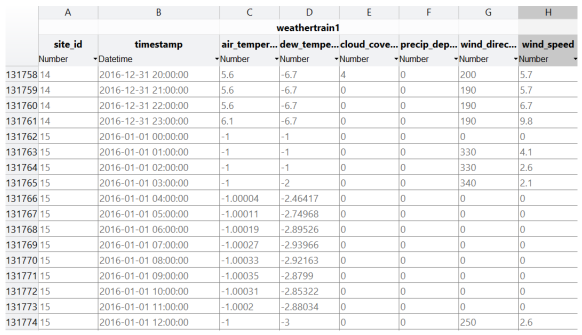
Yet, there are many typical issues in this dataset, which is totally understandable due to its size, such as missing data cells, which most frequently occurs in columns "cloud coverage", "floor count", "precip depth" and "sea level". There are also rows missing rows: in the below figure, from row 131323 to 131324, the timestamp jumped from 3 am to 12 pm, omitting 9 hours. Other than errors, there are also some challenges: there are categorical variables which are shown as Strings that can't be directly inputted into the neural network, such as the column 'primary use'.

We first tackled the missing value issues by updating our datasets into "weather_trian1.csv" and "weather_test1.csv" that filled those gaps using interpolation per site [1]. Then, we converted "timestamp" to integers in MATLAB and "primary_use" to one-hot codes, dropped the mis "floor_count" column, deleted rows that contains NaN, normalized continuous variables, and finally we got the $8088455 * 28$ train dataset "train1.csv" we would use. By doing the same procedures using "building_metadata.csv", "weather_test.csv", and "test.csv", we got the $41697600 * 28$ test dataset "test1.csv" that we would use.



	A	B	C	D	E	F	G	H	I
					weathertrain				
	site_id	timestamp	air_temper...	cloud_cove...	dew_tempe...	precip_dep...	sea_level_p...	wind_direc...	wind_speed
	Number	Datetime	Number	Number	Number	Number	Number	Number	Number
131317	14	2016-12-31 20:00:00	5.6	4.0	-6.7	0.0	1015.3	200.0	5.7
131318	14	2016-12-31 21:00:00	5.6		-6.7	0.0	1014.9	190.0	5.7
131319	14	2016-12-31 22:00:00	5.6		-6.7	0.0	1014.0	190.0	6.7
131320	14	2016-12-31 23:00:00	6.1		-6.7	-1.0	1012.4	190.0	9.8
131321	15	2016-01-01 01:00:00	-1.0		-1.0			330.0	4.1
131322	15	2016-01-01 02:00:00	-1.0		-1.0			330.0	2.6
131323	15	2016-01-01 03:00:00	-1.0		-2.0			340.0	2.1
131324	15	2016-01-01 12:00:00	-1.0		-3.0			250.0	2.6
131325	15	2016-01-01 13:00:00	-1.0		-3.0			240.0	3.1
131326	15	2016-01-01 14:00:00	-1.0		-2.0			240.0	4.1
131327	15	2016-01-01 15:00:00	-1.0		-2.0			240.0	4.6
131328	15	2016-01-01 16:00:00	-1.0		-3.0			270.0	5.1
131329	15	2016-01-01 17:00:00	-1.0		-4.0			240.0	7.2

Figure 3: This is the raw weather dataset where that has not been modified. It can be noticed that there are quite a few issues.

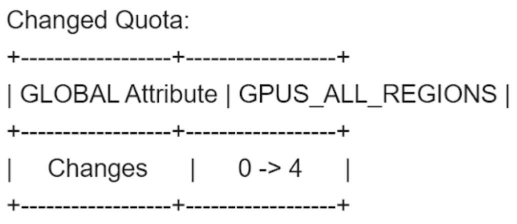


	A	B	C	D	E	F	G	H
				weathertrain1				
	site_id	timestamp	air_temper...	dew_tempe...	cloud_cove...	precip_dep...	wind_direc...	wind_speed
	Number	Datetime	Number	Number	Number	Number	Number	Number
131758	14	2016-12-31 20:00:00	5.6	-6.7	4	0	200	5.7
131759	14	2016-12-31 21:00:00	5.6	-6.7	0	0	190	5.7
131760	14	2016-12-31 22:00:00	5.6	-6.7	0	0	190	6.7
131761	14	2016-12-31 23:00:00	6.1	-6.7	0	0	190	9.8
131762	15	2016-01-01 00:00:00	-1	-1	0	0	0	0
131763	15	2016-01-01 01:00:00	-1	-1	0	0	330	4.1
131764	15	2016-01-01 02:00:00	-1	-1	0	0	330	2.6
131765	15	2016-01-01 03:00:00	-1	-2	0	0	340	2.1
131766	15	2016-01-01 04:00:00	-1.00004	-2.46417	0	0	0	0
131767	15	2016-01-01 05:00:00	-1.00011	-2.74968	0	0	0	0
131768	15	2016-01-01 06:00:00	-1.00019	-2.89526	0	0	0	0
131769	15	2016-01-01 07:00:00	-1.00027	-2.93966	0	0	0	0
131770	15	2016-01-01 08:00:00	-1.00033	-2.92163	0	0	0	0
131771	15	2016-01-01 09:00:00	-1.00035	-2.8799	0	0	0	0
131772	15	2016-01-01 10:00:00	-1.00031	-2.85322	0	0	0	0
131773	15	2016-01-01 11:00:00	-1.0002	-2.88034	0	0	0	0
131774	15	2016-01-01 12:00:00	-1	-3	0	0	250	2.6

Figure 4: This is the processed dataset, in which vacant rows and cells have been filled.

Virtual Machine Application

Due to the tremendous size of training and testing dataset, free computational resources provided by Google Colab seems to be not enough. We always ran out of memory when we were training a neural network model and were trying to use it to predict meter reading of test dataset, since maximum RAM provided by Colab is 25.5 GB and can not be upgraded, hence we decided to increase our computational resource by connecting Colab to Google Cloud Platform. First, we created a new project in GCP and named it 'cosi101final'. Then we submitted a request to increase our maximum usable computational resources and waited for Google to approve.



```
Changed Quota:
+------------------+------------------+
| GLOBAL Attribute | GPUS_ALL_REGIONS |
+------------------+------------------+
|     Changes      |      0 -> 4      |
+------------------+------------------+
```

We selected the Compute Engine product provided by Google and created a Virtual Machine instance, where its physical position was in US east, with multi-core CPU and one Nvidia Tesla GPU. More importantly, we then had a 64 GB RAM so we did not have to worry about memory space.

After the VM we requested was deployed, we now just had to connect Google Colab to the VM. We first installed Google Cloud SDK on our computers and use 'gcloud auth login' to login into our accounts. Our next step was to connect to the VM through our computers, which can be achieved by entering "gcloud beta compute –project "cosi101final" ssh –zone "us-east1-b" "instance-1" – -L 8080:localhost:8080" in Google Cloud SDK. If we entered "localhost:8080" in our browsers we could saw a Jupyter Lab was created and this meant that our computers were connected to the VM now. Finally, enter Google Colab and connect Colab to local runtime with backend port ID that equals to 8080. By entering and running the command line "!cat /proc/meminfo — grep Mem" and "!nvidia-smi" in Colab, we can check our computational resources and hence we would know whether we successfully connect Colab to the VM.

Another trick that we used was to use Kaggle API to connect Kaggle to Google Colab, to facilitate data import. We had to upload our own API token which could be downloaded in 'My Account' section in Kaggle to the hard drive of the VM, then use the below command line to connect:

```
!pip install -q kaggle
!mkdir -p ~/.kaggle
!cp kaggle.json ~/.kaggle/
!chmod 600 ~/.kaggle/kaggle.json
!kaggle competitions download -c ashrae-energy-prediction
```

Then we will have zipped raw data in our hard drive now. Unzip these files and we are done.

Model Fitting and Training

After the data has been pre-processed and Google Colab connected to Virtual Machine, the only thing left to be done was training. Due to one-hot encoding, we now had 28 features, so we would set input shape to 28. The Neural network model we used was fully connected and for activation functions we chose 'ReLu', which stands for Rectified Linear Unit. Number of neurons in each hidden layer are 14, 7 and 3. Reason why we need activation functions was to increase the scale of non-linearity of neural network. If there were no activation functions, the model would have been linear, which would have been equivalent to multiplication through matrices, which is no longer a neural network approach. There were multiple reasons why we used 'ReLu' here. Firstly, comparing to sigmoid, ReLu can be computed much faster. Secondly, 'ReLu' does not face potential errors caused by gradient vanishing. Finally, neural networks using 'ReLu' will be less likely to overfit. The following codes show our model construction. The first segment is where we create the neural layers and build up the model, while the second one is the part that takes the most time and memory space: fitting the model on our training data.

```
inputs = Input(shape=(28,))
encoded1 = Dense(14, activation='relu')(inputs)
encoded2 = Dense(7, activation='relu')(encoded1)
encoded3 = Dense(3, activation='relu')(encoded2)
outputs = Dense(1)(encoded3)
SSAE_model = Model(inputs, outputs)
```

```
SSAE_model.compile(optimizer='adam', loss='mse', metrics=['accuracy'])
SSAE_model.fit([train_X], [train_y], epochs=50, batch_size=100)
```

Evaluation of Results

Kaggle uses 'RMSLE' function to evaluate model performance. This function is close to 'RMSE' function except it takes the *log* of the actual and obtained value before computing their squared error, in which lower scores represent better models. The result for the neural network approach was not satisfying: our rmse was extremely high, which was more than ten thousand after it converged to the minimum. The Kaggle score, calculated by RMSLE, was 2.72, indicating predictions made by our model were way off from target values. Hence the first attempt was considered as totally unsuccessful.

We did save our model's weights, and they can be retrieved into another neural network of the same size using below code trunk:

```
modelweight = drive.CreateFile({'id':'16auI2HiH7vke9nLGcZkmVzY5Le2bPhFw'})
modelweight.GetContentFile('first_model.h5')
testmodel.compile(optimizer = 'adam', loss = 'mse', metrics = ['accuracy'])
testmodel.load_weights('first_model.h5')
```

We do have several reflections about this failure: for one, we brutally converted the timestamp into huge integers, which definitely lacks consideration. Although we attempted to normalize the data to decrease the huge discrepancies in scale between features, it didn't prevent the inaccuracy. Now that we think about it, the way we converted the timestamp was to convert everything into hours, where months' and years' weights completely dominated the integer's value while leaving days and hours literally useless. This taught us the importance to value every single information, instead of mixing some of them together, hoping to easily simplify the problem. After careful thought, we figured the correct thing to do is to create new columns named 'year', 'month', 'day' and 'hour', and plug the information containing in one timestamp into these four columns. Although it will create more features, it will significantly increase accuracy. The other fault that we made was that, during data pre-processing, we accidentally missed a column "sea level pressure", which turned out to be an influential feature to the target, which we will explain in our next approach.

Possible Improvements using Autoencoder

Our team attempted to boost our model using Stacked Autoencoder, which is a tool to pre-train neurons' weights before fitting the model with training data. We believe after fixing our previously stated problems and adding autoencoder, out neural network model will be much more effective.

Generally, an autoencoder, AE for short, consists of two subnetworks. The encoder network, which is formed by the input layer and the hidden layer, maps the original input to the hidden feature representations. The decoder network, which is formed by the hidden layer and the output layer, reconstructs the original input through the learned feature.

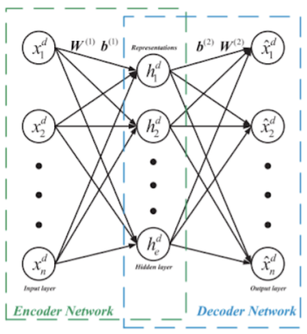
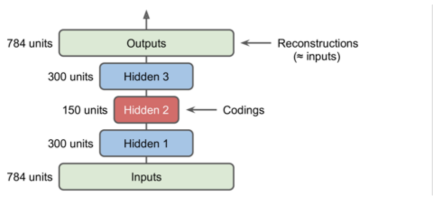


Figure 5: The architecture of a typical AE. n and e are dimensions of input respectively. [4]

When we consecutively use multiple AEs, we call them a stacked autoencoder (SAE), which is a technically just a neural network that contains a multi-layer AE where the output of encoding layer of each AE is used as the input to the next AE. Below Figure shows the whole architecture of SAE:



Stacked autoencoder is often used for pre-training of weights for deep neural networks (DNNs), such as our model. By doing this, DNN can converge quickly and have better classification ability. Below Figure shows the pre-training procedure. The weights obtained during the pre-training process will be used as the initial values of the weights of the DNN in the final global tuning phase. This layer-wise initialization approach has been shown to yield more meaningful local minima than random initialization, resulting in better generalization.

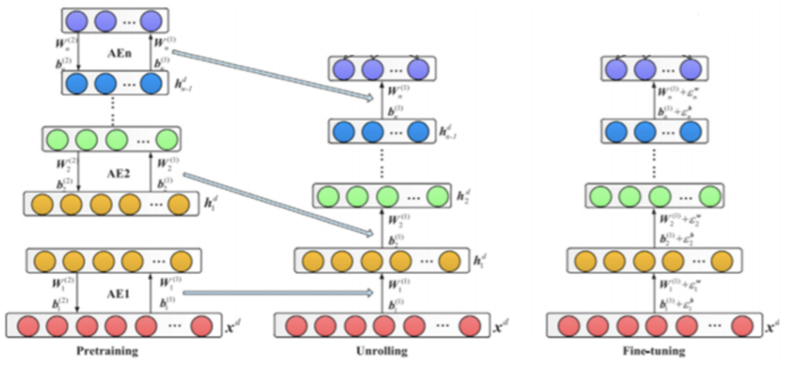


Figure 6: AE1 is first trained on the raw data to obtain weights (Wi) and bias (bi). After AE1 is trained, the AE1 encoding network is used to convert the raw data to the hidden representation vector hd 1 on which AE2 is trained to obtain weights (Wi) and bias (bi). And, similarly, hd 2 can be computed by AE2's encoding network, which is used as the input to train AE3. Repeat this process for every AE.

We have updated our code to incorporate the SAE technique:

```
# Pretrain AE1
ipt = Input(shape=(28,))
encoded = Dense(14, activation='relu')(ipt)
decoded = Dense(28)(encoded)
autoencoder = Model(ipt, decoded)
encoder = Model(ipt, encoded)
autoencoder.compile(optimizer='adam', loss='mse', metrics=['accuracy'])
autoencoder.fit([train_X], [train_X], epochs=50, batch_size=100)
code = encoder.predict([train_X])
SSAE_model.layers[1].set_weights(encoder.layers[-1].get_weights())

# Pretrain AE2
ipt = Input(shape=(14,))
encoded = Dense(7, activation='relu')(ipt)
```

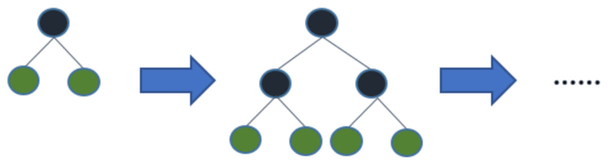
```
decoded = Dense(14)(encoded)
autoencoder = Model(ipt, decoded)
encoder = Model(ipt, encoded)
autoencoder.compile(optimizer='adam', loss='mse', metrics=['accuracy'])
autoencoder.fit([code], [code], epochs=50, batch_size=100)
code = encoder.predict([code])
SSAE_model.layers[2].set_weights(encoder.layers[-1].get_weights())

# Pretrain AE3
ipt = Input(shape=(7,))
encoded = Dense(3, activation='relu')(ipt)
decoded = Dense(7)(encoded)
autoencoder = Model(ipt, decoded)
encoder = Model(ipt, encoded)
autoencoder.compile(optimizer='adam', loss='mse', metrics=['accuracy'])
autoencoder.fit([code], [code], epochs=50, batch_size=100)
code = encoder.predict([code])
SSAE_model.layers[3].set_weights(encoder.layers[-1].get_weights())
```

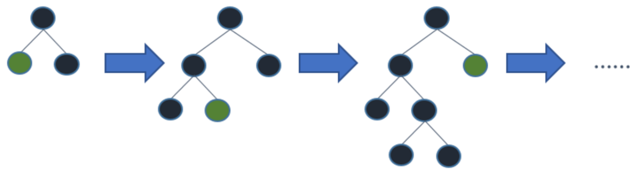
However, due to the previous unsatisfying result and more importantly, the pressing time, we updated the model but gave up a second running attempt, and shifted our focus to the second, and more effective approach: LightGBM.

LightGBM Approach

LightGBM Model Explained

Light GBM is a gradient boosting framework that uses tree based learning algorithm. It differs from other tree-based algorithm in that Light GBM grows tree vertically while other algorithm grows trees horizontally, meaning that Light GBM grows tree leaf-wise while other algorithm grows level-wise. It will choose the leaf with max delta loss to grow. When growing the same leaf, Leaf-wise algorithm can reduce more loss than a level-wise algorithm. Below are figures that show the distinctions between Light GBM's tree structure and normal tree structure.


Level-wise tree growth


Leaf-wise tree growth

The fact that the LGBM decision tree will not directly grow levels but grows on certain leaves shows its high speed, which is also the reason why it's been called "light". It uses the histogram-based algorithm to speed up the training process, reduce memory consumption and combines advanced network communication to optimize parallel learning, called parallel voting decision tree algorithm. It divides the training data into multiple machines, uses the local voting decision to select the top-k attributes and the global voting decision to receive the top2k attributes in each iteration are performed. It is suitable for large sets of data, and takes less memory space to run. However, it's been noted that LGBM, as well as some other boosting models, are susceptible to overfitting. For datasets that contains less than 10000 rows, LGBM will easily overfit - although powerful and fast, LGBM is not omnipotent.

Adjusting hyper-parameters is crucial for constructing any successful model, so is for Light GBM. There are multiple parameters in this algorithm that needs special attentions: 1. *Max_depth*: It describes the maximum depth of tree. This parameter is used to handle model overfitting. When the model is overfitted, we need to lower our max depth parameter. 2. *Num_leaves*: number of leaves in full tree. The optimal num of leaves differ from case to case, so we will need several rounds of training with all other parameters fixed and adjust the num_leaves to find the most optimal num_leaves. 3. *Learning_rate:* This determines the impact of each tree on the final outcome. GBM works by starting with an initial estimate which is updated using the output of each tree. The learning parameter controls the magnitude of this change in the estimates. Normally, higher learning_rate will converge faster while smaller learning_rate possesses higher accuracy. Thus, we used a 0.1 learning rate to do multiple trials to get the optimized num_leaves, then changed into smaller learning_rate to tune the model. 4. *Boosting*: this parameter defines the type of algorithm you want to run - gbdt: traditional Gradient Boosting Decision Tree, which is very generic and used prevalently. rf: random forest. dart: Dropouts meet Multiple Additive Regression Trees. goss: Gradient-based One-Side Sampling. We wanted to select 'dart' in this approach, yet dart doesn't allow early stop in iterations even though the rmse has converged, which basically means it takes forever to finish our algorithm. Therefore, we chose the default 'gbdt' function. The rest parameters, such as 'application' or 'metric' are relatively less important.

Data Pre-processing, Memory Reduction and Feature Engineering

As we have mentioned, there are some missing hours in the weather datasets. We fixed this problem in the following way: Firstly, we found missing hours by subtracting the list of timestamps for each site from the list of all timestamps in a year and then insert an empty placeholder row for each missing hour. This is down using the following code:

```
time_format = "%Y-%m-%d %H:%M:%S"
start_date =
    datetime.datetime.strptime(weather_df['timestamp'].min(),time_format)
end_date =
    datetime.datetime.strptime(weather_df['timestamp'].max(),time_format)
total_hours = int(((end_date - start_date).total_seconds() + 3600) / 3600)
hours_list = [(end_date -
    datetime.timedelta(hours=x)).strftime(time_format) for x in
    range(total_hours)]

missing_hours = []
for site_id in range(16):
    site_hours = np.array(weather_df[weather_df['site_id'] ==
        site_id]['timestamp'])
    new_rows =
        pd.DataFrame(np.setdiff1d(hours_list,site_hours),columns=['timestamp'])
    new_rows['site_id'] = site_id
    weather_df = pd.concat([weather_df,new_rows])

    weather_df = weather_df.reset_index(drop=True)
```

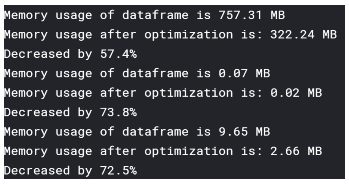
Next, we separated the "timestamp" column into four columns "datetime", "day", "week", "month", note that the sole purpose of creating these columns is to serve the next step, which is to calculate the averages. We will re-create columns in the feature engineering that will happen later. Thirdly, for each site in each day of each month, calculate the average of each weather feature value ("air_temperature", "cloud_coverage", etc...) and then fill it into the corresponding empty place. Below code gives an illustration of how we filled the empty spaces in "cloud_coverage" column:

```
# Step 1
cloud_coverage_filler =
    weather_df.groupby(['site_id','day','month'])['cloud_coverage'].mean()
# Step 2
cloud_coverage_filler =
    pd.DataFrame(cloud_coverage_filler.fillna(method='ffill'),columns=["cloud_coverage"])

weather_df.update(cloud_coverage_filler,overwrite=False)
```

Lastly, we wanted to extract more information with the inter-correlation between features. By using the Magnus approximation, we can calculate the relative humidity based on "dew_temperature" and "air_temperature" with the formula $RH = 100e^{(\frac{cb(TD-T)}{(c+T)(c+TD)})}$. By using "feels_like" and "Temp" function in the "meteocalc" package, we can calculate the "feels like" temperature based on "air temperature" (at), "relative_humidity" (rh), and "wind_speed" (ws) as feels like (Temp(at, unit = 'C'), rh, ws)). Below code trunk contains the function to create the "relative_humidity" and "feels_like" features:

```
def get_meteorological_features(data):
    def calculate_rh(df):
        df['relative_humidity'] = 100 * (np.exp((17.625 *
            df['dew_temperature']) / (243.04 + df['dew_temperature'])) /
            np.exp((17.625 * df['air_temperature'])/(243.04 +
            df['air_temperature'])))
    def calculate_fl(df):
        flike_final = []
        flike = []
        # calculate Feels Like temperature
        for i in range(len(df)):
            at = df['air_temperature'][i]
            rh = df['relative_humidity'][i]
            ws = df['wind_speed'][i]
            flike.append(feels_like(Temp(at, unit = 'C'), rh, ws))
        for i in range(len(flike)):
            flike_final.append(flike[i].f)
        df['feels_like'] = flike_final
        del flike_final, flike, at, rh, ws
    calculate_rh(data)
    calculate_fl(data)
    return data
    weather_df = get_meteorological_features(weather_df)
```

Before we began to merge datasets into one and start to perform feature engineering, we observed that existing two dataset already occupied huge amount of memory space. Although we always have the Virtual Machine backup plan. Yet, it would always be better to reduce some memory space to speed up the computations. To achieve this, we looked up and found some public resources that we could use [2]. The memory-reduction function iterates through all columns of the input data while checking the maximum and minimum values of each column, then update the datatype in that column with the most efficient datatype that is able to hold all the values. Using this method, we fit each column into the tightest boundary, number of bits occupied by each element in a column is less than before without changing contents. Below Figure shows the effect of this method:



```
Memory usage of dataframe is 757.31 MB
Memory usage after optimization is: 322.24 MB
Decreased by 57.4%
Memory usage of dataframe is 0.07 MB
Memory usage after optimization is: 0.02 MB
Decreased by 73.8%
Memory usage of dataframe is 9.65 MB
Memory usage after optimization is: 2.66 MB
Decreased by 72.5%
```

Figure 7: The effect of the memory-reduction function, where it decreased memory usages by over 70%.

Then we merged two datasets into one, where we will perform feature engineering. Compared to what we did in the neural network approach, it was much easier for Light GBM. Our focus in feature engineering lies on timestamp, since one of our biggest failures in the previous neural network approach was that we bluntly converted the timestamp column into integers. This time, we used 'datetime' packages and transformed timestamps into separated columns: year, month, day, hour, minute and second. Then we used the weekday (or dayofweek) function in pandas to encode days category, and created a new column to store current hour. Other than those two, we also add a new column that represents current quarter. By using the replace() function, we will substitute 12 months into 4 quarters of a year.

The rest part of feature engineering was trivial. In order to reduce the range of "square feet" and "meter reading", we use the log1p() function in numpy to rescale them. We also transformed "primary use" column from string to integer using label encoder function from sklearn. At this point, we are officially done with data preprocessing and feature engineering. It is time to train a LightGBM model with the data we have.

Model Fitting, Training and Validating

The process that we trained our model is the following: firstly, set 'building_id', 'site_id' and all other categorical features as 'categorical' so that the model would treat these data properly. Secondly, we ran the LightGBM model with 5-fold cross validation, where it splitted the whole train set into 5 folds and picked one as validation dataset while four left as train dataset each round. Furthermore, we calculated and printed the training and validation RMSE for each round. The variable 'predonTrain' was used to compare with actual meter reading in training dataset and score represented overall RMSE. Notice that a good overall RMSE did not necessarily represent a better model due to possible overfitting. Below is the code trunk where we fitted and trained our LGBM model:

```
for tr_idx, val_idx in tqdm(kf.split(train_df,train_df['month']), total =
    folds):
        tr_x, tr_y = train[features].iloc[tr_idx], train[target].iloc[tr_idx]
        vl_x, vl_y = train[features].iloc[val_idx],
            train[target].iloc[val_idx]
        tr_data = lgb.Dataset(tr_x, label = tr_y, categorical_feature =
            categorical)
        vl_data = lgb.Dataset(vl_x, label = vl_y, categorical_feature =
            categorical)
        clf = lgb.train(param, tr_data, num_rounds, valid_sets = [tr_data,
            vl_data], verbose_eval = 25,
                      early_stopping_rounds = 50)
```

Below is the line chart that shows the model performance when we change the number of leaves:

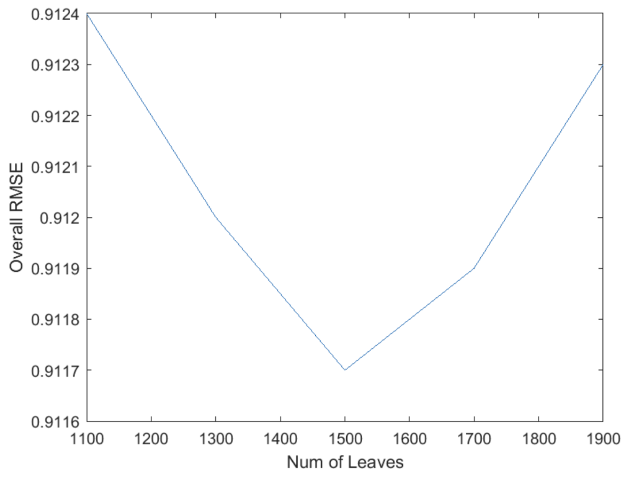


Figure 8: The change in model performance when given different number of leaves.

Therefore, it shows that 1500 is nearly the optimized number of leaves, and we will set the number of leaves to be around 1500 and re-train the model to obtain the best result.

Evaluation of Results

Our best Kaggle score is 1.089. Although its far from the best on the board, we are delighted that this is much better than that of our neural network model. As the previous section has already mentioned, we used a 5-fold validation. We focused on adjusting two hyper-parameters, which are learning rate and number of leaves. We made several submissions, and the best were set to 0.05 and 1580.

The reason why we couldn't obtain a better result could come from one or more of the following reasons: due to time limitation, we didn't try to adjust more hyper-parameter values. Also, it was common that most datasets have outliers in it, which will have a negative effect on model training. However, we did not do an outlier detection that will help to find abnormal data so that we could drop them. Next time, we will first visualize the dataset and run an outlier detection. Also, although we used a 5-fold method, we didn't exactly know what number of fold is the optimized number.

Conclusions and Reflections

Throughout this project, we have learned a great deal of knowledge and gained much invaluable experience which would be very helpful in our future practices in machine learning. We reflected upon ourselves and realized that we have improved not only in technical aspects, such as model selection, data manipulation, hyper-parameter improving, but also in non-technical aspects, like time management, teamwork and effective communications. However, we also realized that we made several mistakes and that our project could have been done better. In this section, we will summarize the stuff we have learned and places we could improve.

We have learned plenty of technical skills. What comes before the hardcore knowledge about machine learning is data manipulating skills. We learned data frame operations in python.pandas package, including referencing rows and columns, dropping columns, creating groups by columns and creating data frames. Sometimes we found MATLAB is more effective in table operations, where it treats everything as matrices so that it's really fast. Only with these knowledge of data processing, could we successfully perform feature engineering. We also gained a sense of how to wisely run machine learning on our laptops, such as running codes in Google Colab (with GPU!), using Kaggle API to foster data importing.

Equally important, we learned to properly utilize machine learning algorithms by carefully choosing hyper-parameters, such as in LightGBM where we did multiple pre-trains to obtain the optimized num_leaves parameter, and also lowered our learning rate in effort to obtain a more accurate result.

We are surprised, yet thrilled by the fact that the process of choosing a suitable machine learning algorithm is sometimes counterintuitive. As at the beginning of our practice, we immediately thought neural network as the best approach given the tremendous size of the datasets. Yet, we then realized the drawback of neural network - it takes forever to run and it's very sensitive about inputs and that the decision tree based, gradient boosting algorithm LightGBM is much more effective. This taught us to expand our thoughts and never jump to conclusions when choosing algorithms next time and this projects before the wrong model wastes us too much time.

As stated in previous sections, we have made some mistakes in this project that undermined our result. If we could have more time, we would implement our new data pre-processing and Autoencoder methods on neural network method; would try to adjust more parameters such as 'boosting' and 'metric' to further improve our LightGBM model; would also implement and run 'XGboost' and 'Catboost' models, then compare their results with that of LightGBM to discover their differences in efficiency.

Credits and Contributions

In this project, all of us have devoted a significant amount of time in learning, discussing and implementing. We constantly meet up and finished this project as a team. Below explicitly lists each of our contributions:

Ruiyang Hu:
Neural network approach – Data manipulation in Matlab (codes)
Neural network approach – VM instance construction, including connecting it to Colab and applying Kaggle API in Colab to facilitate importing datasets.
LightGBM approach: Data pre-processing (codes, writing)
LightGBM approach – Training and validating (codes, writing)

Hanyu Song:
Neural Network and LightGBM model explaining (writing)

Neural network approach – Data exploration and pre processing (coding, writing)
Neural network approach – Fitting and training the model (codes)
LightGBM approach – Date pre-processing (codes)
LightGBM approach – Training and validating (codes, writing)
Introduction and conclusion (writing)
LATEX creating and report drafting, organizing, and polishing.

Yicheng Tao:
Neural network approach – Data pre processing (coding, writing)
Neural network approach – Fitting and training the model (coding)
Neural network approach – Evaluation of results (coding, writing)
LightGBM approach – LightGBM Model and its Hyper-parameters (writing)
LightGBM approach – Data pre-processing and Feature engineering (codes, writing)
LightGBM approach – Results (writing)
Conclusion (writing)

References

[1] https://www.kaggle.com/hmendonca/clean-weather-data-eda.

[2] https://www.kaggle.com/gemartin/load-data-reduce-memory-usage, 2019.

[3] Varun Kumar Ojha, Ajith Abraham, and Václav Snášel. Metaheuristic design of feedforward neural networks: A review of two decades of research. *Engineering Applications of Artificial Intelligence*, 60:97–116, 2017.

[4] Zehan Zhang, Shuanghong Li, Yawen Xiao, and Yupu Yang. Intelligent simultaneous fault diagnosis for solid oxide fuel cell system based on deep learning. *Applied energy*, 233:930–942, 2019.