Assignment 2: Machine learning with Energy dataset

- *Exploratory Data Analysis:*
- First, we import dataset and get its overview like below:

```
In [4]: complete_df.describe()
```

Out[4]:

	Appliances	lights	T1	RH_1	T2	RH_2	T3	RH_3	T4	RH_4	...	
count	19735.000000	19735.000000	19735.000000	19735.000000	19735.000000	19735.000000	19735.000000	19735.000000	19735.000000	19735.000000	...	1973
mean	97.694958	3.801875	21.686571	40.259739	20.341219	40.420420	22.267611	39.242500	20.855335	39.026904	...	1
std	102.524891	7.935988	1.606066	3.979299	2.192974	4.069813	2.006111	3.254576	2.042884	4.341321	...	
min	10.000000	0.000000	16.790000	27.023333	16.100000	20.463333	17.200000	28.766667	15.100000	27.660000	...	1
25%	50.000000	0.000000	20.760000	37.333333	18.790000	37.900000	20.790000	36.900000	19.530000	35.530000	...	1
50%	60.000000	0.000000	21.600000	39.656667	20.000000	40.500000	22.100000	38.530000	20.666667	38.400000	...	1
75%	100.000000	0.000000	22.600000	43.066667	21.500000	43.260000	23.290000	41.760000	22.100000	42.156667	...	2
max	1080.000000	70.000000	26.260000	63.360000	29.856667	56.026667	29.236000	50.163333	26.200000	51.090000	...	2

8 rows × 28 columns

then we have a look on every feature's data type:

```
In [5]: complete_df.dtypes
```

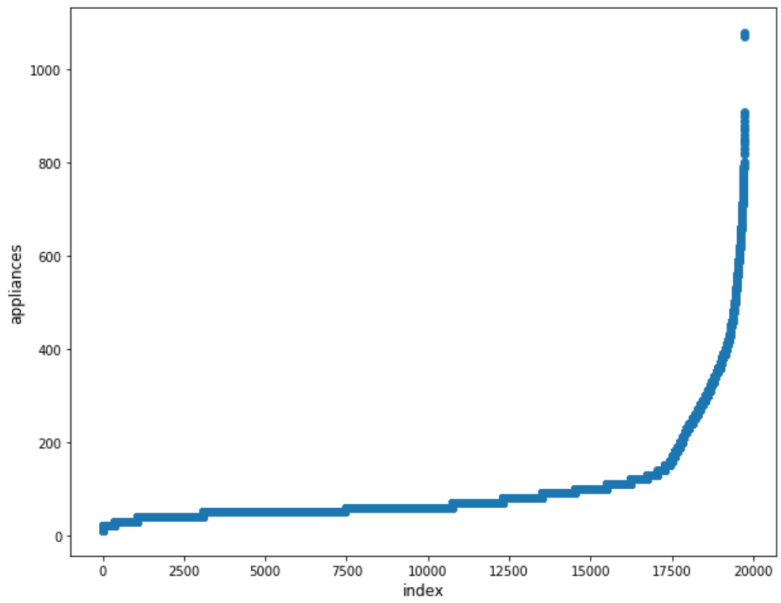
Out[5]:

```
date            object
Appliances       int64
lights           int64
T1             float64
RH_1           float64
T2             float64
RH_2           float64
T3             float64
RH_3           float64
T4             float64
RH_4           float64
T5             float64
RH_5           float64
T6             float64
RH_6           float64
T7             float64
RH_7           float64
T8             float64
RH_8           float64
T9             float64
RH_9           float64
T_out          float64
Press_mm_hg    float64
RH_out         float64
Windspeed      float64
Visibility     float64
Tdewpoint      float64
rv1            float64
rv2            float64
dtype: object
```

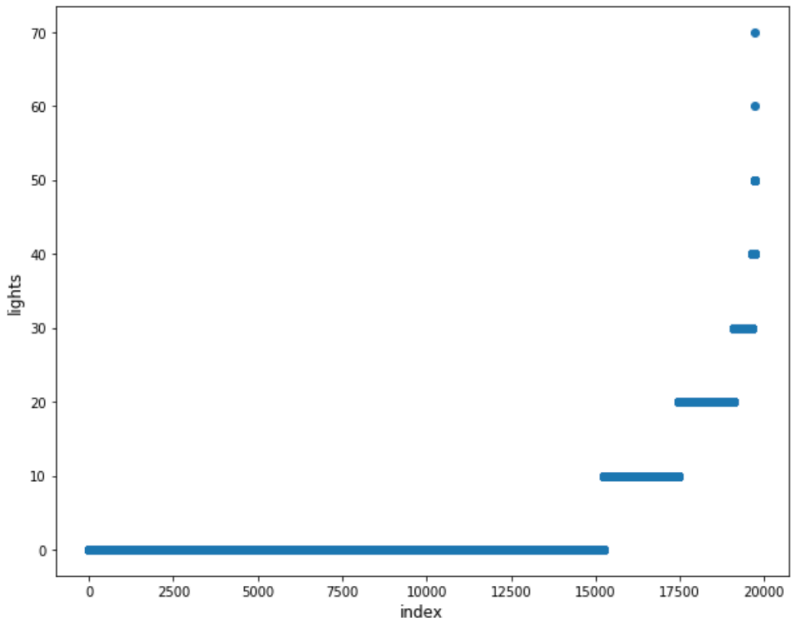
We can get except the column of date is object, all other data is numeric.

- Then we draw distribution graphs for every feature to see if there has some outliers and its overall trend. I list 2 of them below:

In [7]:
```
plt.figure(figsize = (10,8))
plt.scatter(range(complete_df.shape[0]),np.sort(complete_df.Appliances.values))
plt.xlabel('index',fontsize=12)
plt.ylabel('appliances',fontsize=12)
plt.show()
```



In [9]:
```
plt.figure(figsize = (10,8))
plt.scatter(range(complete_df.shape[0]),np.sort(complete_df.lights.values))
plt.xlabel('index',fontsize=12)
plt.ylabel('lights',fontsize=12)
plt.show()
```



We can see there are some outliers and some features' data is low variance.

- After that, we calculate the correlation between all features and get a table below:

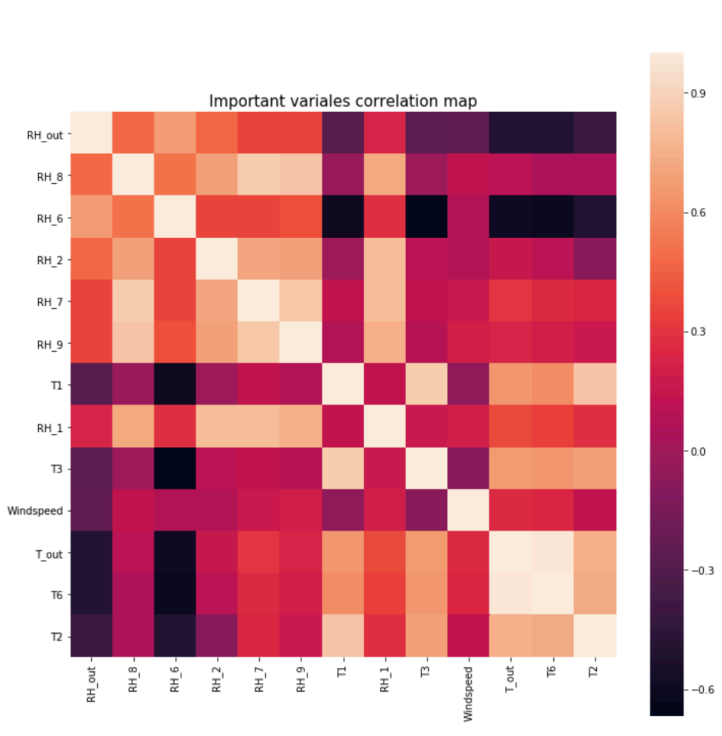
```
In [6]: complete_df.corr()
```

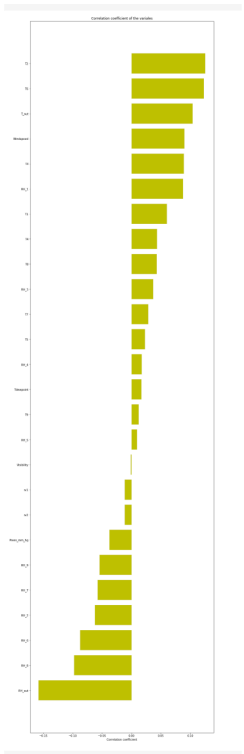
Out[6]:

	Appliances	lights	T1	RH_1	T2	RH_2	T3	RH_3	T4	RH_4	...	T9	RH_9	T_ou
Appliances	1.000000	0.197278	0.055447	0.086031	0.120073	-0.060465	0.085060	0.036292	0.040281	0.016965	...	0.010010	-0.051462	0.09915
lights	0.197278	1.000000	-0.023528	0.106968	-0.005622	0.050985	-0.097393	0.131161	-0.008859	0.114936	...	-0.157592	-0.008766	-0.07442
T1	0.055447	-0.023528	1.000000	0.164006	0.836834	-0.002509	0.892402	-0.028550	0.877001	0.097861	...	0.844777	0.071756	0.68284
RH_1	0.086031	0.106968	0.164006	1.000000	0.269839	0.797535	0.253230	0.844677	0.106180	0.880359	...	0.115263	0.764001	0.34076
T2	0.120073	-0.005622	0.836834	0.269839	1.000000	-0.165610	0.735245	0.121497	0.762066	0.231563	...	0.675535	0.157346	0.79225
RH_2	-0.060465	0.050985	-0.002509	0.797535	-0.165610	1.000000	0.137319	0.678326	-0.047304	0.721435	...	0.054544	0.676467	0.03367
T3	0.085060	-0.097393	0.892402	0.253230	0.735245	0.137319	1.000000	-0.011234	0.852778	0.122737	...	0.901324	0.134602	0.69941
RH_3	0.036292	0.131161	-0.028550	0.844677	0.121497	0.678326	-0.011234	1.000000	-0.140457	0.898978	...	-0.195270	0.833538	0.11820
T4	0.040281	-0.008859	0.877001	0.106180	0.762066	-0.047304	0.852778	-0.140457	1.000000	-0.048650	...	0.889439	-0.025549	0.66347
RH_4	0.016965	0.114936	0.097861	0.880359	0.231563	0.721435	0.122737	0.898978	-0.048650	1.000000	...	-0.044518	0.856591	0.29328
T5	0.019760	-0.078745	0.885247	0.205797	0.720550	0.110409	0.888169	-0.050062	0.871813	0.091812	...	0.911055	0.072308	0.65132
RH_5	0.006955	0.141233	-0.014782	0.303258	0.029595	0.250271	-0.066355	0.375422	-0.076489	0.352591	...	-0.138509	0.272197	-0.05312
T6	0.117638	-0.079029	0.654769	0.316141	0.801186	-0.009670	0.686882	0.076833	0.652350	0.259047	...	0.667177	0.184424	0.97478
RH_6	-0.083178	0.153756	-0.615045	0.245126	-0.580372	0.389933	-0.647672	0.514912	-0.703149	0.392178	...	-0.738940	0.391943	-0.64157
T7	0.025801	-0.135347	0.838705	0.021397	0.663660	-0.051422	0.847374	-0.250090	0.877763	-0.131204	...	0.944776	-0.077690	0.63129
RH_7	-0.055642	0.035069	0.135182	0.801122	0.229212	0.690584	0.172624	0.832685	0.043527	0.894301	...	0.028055	0.858686	0.29419
T8	0.039572	-0.071458	0.825413	-0.030053	0.578191	-0.041023	0.795283	-0.283228	0.796256	-0.167066	...	0.869338	-0.156820	0.50284
RH_8	-0.094039	0.012915	-0.006441	0.736196	0.068534	0.679777	0.044427	0.828822	-0.095192	0.847259	...	-0.113014	0.855812	0.11714
T9	0.010010	-0.157592	0.844777	0.115263	0.675535	0.054544	0.901324	-0.195270	0.889439	-0.044518	...	1.000000	-0.008683	0.66822
RH_9	-0.051462	-0.008766	0.071756	0.764001	0.157346	0.676467	0.134602	0.833538	-0.025549	0.856591	...	-0.008683	1.000000	0.22327
T_out	0.099155	-0.074424	0.682846	0.340767	0.792255	0.033674	0.699417	0.118207	0.663478	0.293289	...	0.668220	0.223270	1.00000
Press_mm_hg	-0.034885	-0.010576	-0.150574	-0.293957	-0.133028	-0.255646	-0.189974	-0.233274	-0.075292	-0.250748	...	-0.156828	-0.183739	-0.14324
RH_out	-0.152282	0.068543	-0.345481	0.274126	-0.505291	0.584911	-0.281718	0.356192	-0.388602	0.336813	...	-0.318848	0.359377	-0.57419
Windspeed	0.087122	0.060281	-0.087654	0.204932	0.052495	0.069190	-0.100776	0.263188	-0.185747	0.300192	...	-0.177756	0.238655	0.19293
Visibility	0.000230	0.020038	-0.076210	-0.021057	-0.069721	-0.005368	-0.102310	0.017041	-0.104768	0.002636	...	-0.103915	0.008667	-0.07736
Tdewpoint	0.015353	-0.036322	0.571309	0.639106	0.582602	0.499152	0.645886	0.414387	0.519471	0.616509	...	0.581483	0.540328	0.79066
rv1	-0.011145	0.000521	-0.006203	-0.000699	-0.011087	0.006275	-0.005194	-0.000477	-0.001815	-0.001787	...	-0.001227	-0.002955	-0.01521
rv2	-0.011145	0.000521	-0.006203	-0.000699	-0.011087	0.006275	-0.005194	-0.000477	-0.001815	-0.001787	...	-0.001227	-0.002955	-0.01521

28 rows × 28 columns

For seeing correlation intuitively, we draw a correlation map and a histogram of correlation coefficient.



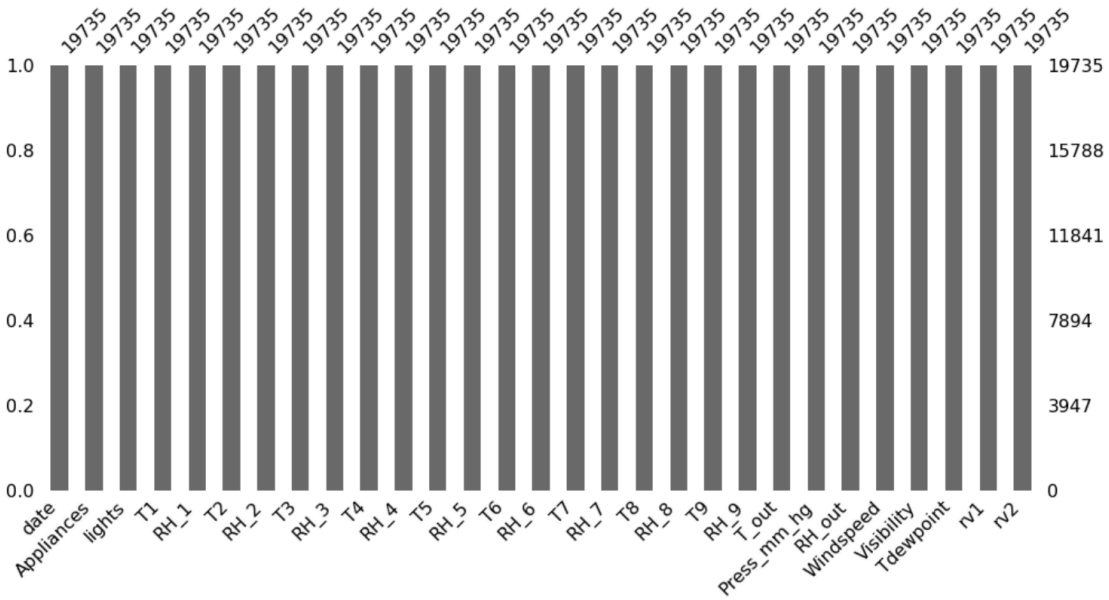


We can tell there are some features have too strong relevance, we can use one feature to stand for all these relevant features.

- In order to know whether there are some NaN values, we draw a graph like below:

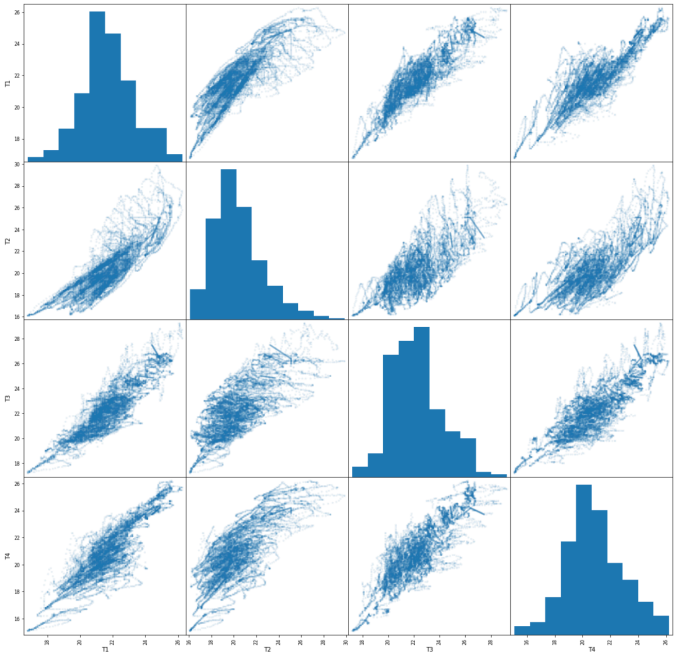
```
In [16]: msno.bar(df_copy,
                  figsize=(16,7))
```

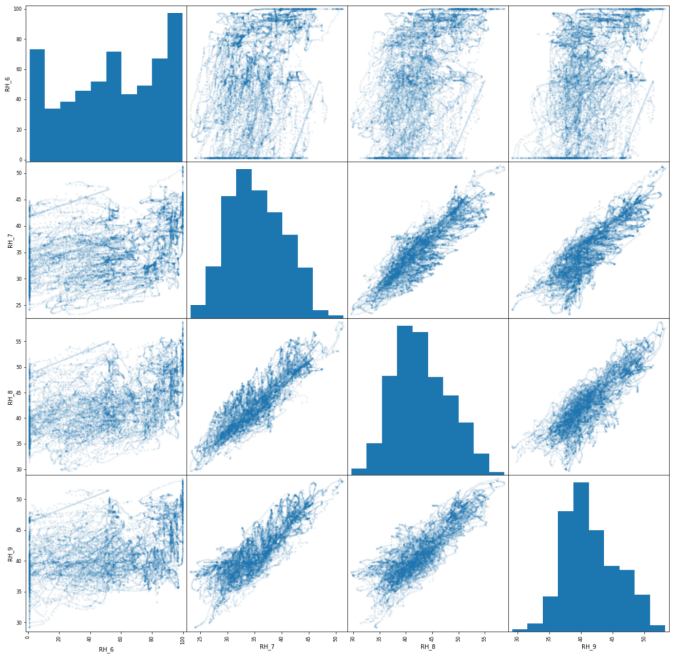
```
Out[16]: <matplotlib.axes._subplots.AxesSubplot at 0x15da8378438>
```



We can see there is no NaN value in all these features so we have no need to do work of removing missing value.

- In order to see some specific features correlation, we draw graphs like below:





We can see T1, T2, T3, T4 have a positive correlation from the left graph and R7, R8,R9 have a positive correlation but R6 has no correlation with them from right graph.

- At last, we use pandas profiling to get a comprehensive summarize.

Pandas profiling

```
In [24]: pandas_profiling.ProfileReport(df_copy)
```

Out[24]:

Overview

Dataset info

Number of variables	29
Number of observations	19735
Total Missing (%)	0.0%
Total size in memory	4.4 MiB
Average record size in memory	232.0 B

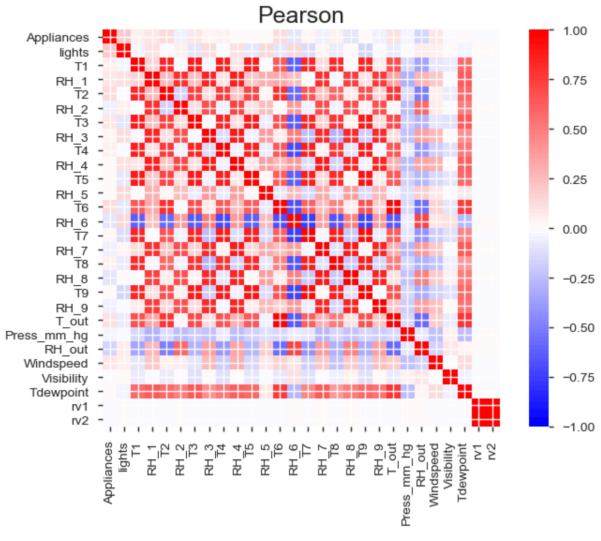
Variables types

Numeric	25
Categorical	0
Boolean	0
Date	0
Text (Unique)	1
Rejected	3
Unsupported	0

Warnings

- `T9` is highly correlated with `T7` (ρ = 0.94478) Rejected
- `T_out` is highly correlated with `T6` (ρ = 0.97479) Rejected
- `lights` has 15252 / 77.3% zeros Zeros
- `rv2` is highly correlated with `rv1` (ρ = 1) Rejected

Correlations



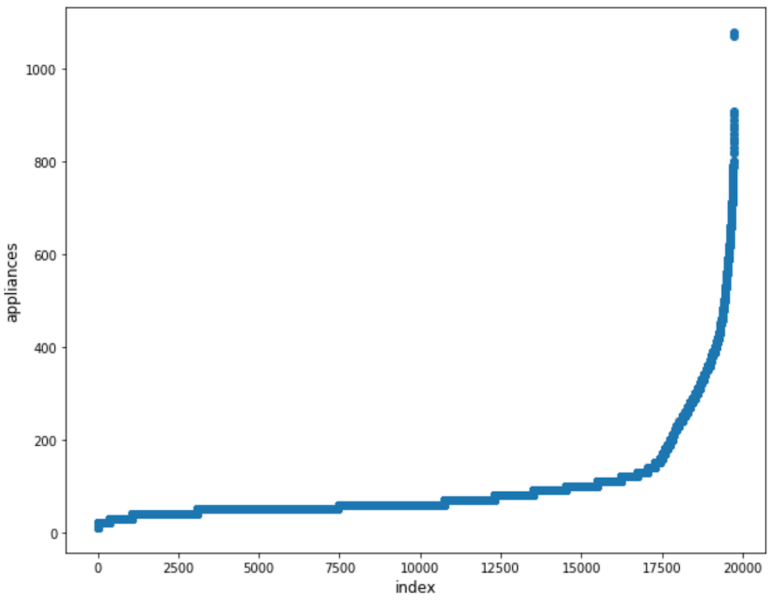
We can get some same conclusions as we already know above, so let's jump into feature engineering part to do more impressing work.
All these graphs are in this PPT:
https://docs.google.com/presentation/d/15BCYZdk0XMXENPqZk5FjavM_JoTbkMX2Uy4CFFxzgso/edit?usp=sharing

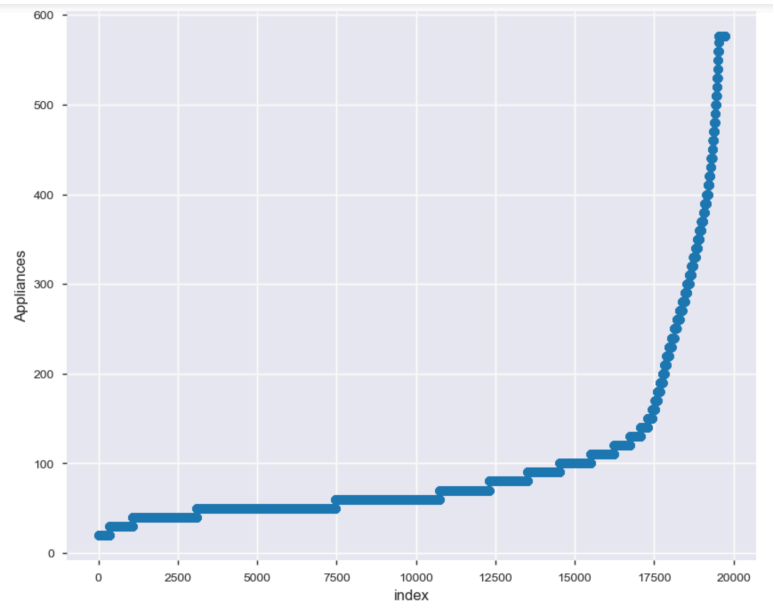
• *Feature Engineering:*

According to our exploratory data analysis, we need do some work on our dataset to make training data more accurate, then to select most important feature to train our model.

- First, we already find there are some outliers in our data, so let's remove this outliers. We choose to remove least 1% and most 1% data from all features.

```
In [8]: ulimit = np.percentile(complete_df.Appliances.values,99)
        llimit = np.percentile(complete_df.Appliances.values,1)
        complete_df['Appliances'].ix[complete_df['Appliances'] > ulimit] = ulimit
        complete_df['Appliances'].ix[complete_df['Appliances'] < llimit] = llimit

        plt.figure(figsize=(12,8))
        sns.distplot(complete_df.Appliances.values, bins=50, kde=False)
        plt.xlabel('Appliances', fontsize=12)
        plt.show()
```





You can see the outliers have been removed.

Then, we remove features with low variance. It removes all features whose variance doesn't meet some threshold.

```
def fea_sel_variancethreshold(x_pre):
    sel = VarianceThreshold(threshold=(.8*(1-.8)))
    x_post = sel.fit_transform(x_pre)
    return x_post
```

- Secondly, we use pre-processing in Sklearn package to make dataset more suitable for our model training. For this part, we need choose all numeric data to proceed data standardization, normalization, scaling and generating polynomial features.

Standardization of datasets is a common requirement for many machine learning estimators implemented in scikit-learn; they might behave badly if the individual features do not more or less look like standard normally distributed data: Gaussian with zero mean and unit variance.

Standardization

```
#Method_1
#
#numeric_data_complete
complete = np.array(numeric_data_complete)
# calculate mean
complete_mean = complete.mean(axis=0)
# calculate variance
complete_std = complete.std(axis=0)
# standardize
complete1 = (complete-complete_mean)/complete_std
# use function preprocessing.scale to standardize
complete_scale = preprocessing.scale(complete)
```

```
complete_scale
```

```
array([[-0.36767572,  3.30126384, -1.11864475, ...,  0.3669753 ,
        -0.80797358, -0.80797358],
       [-0.36767572,  3.30126384, -1.11864475, ...,  0.34313479,
        -0.44024015, -0.44024015],
       [-0.46521548,  3.30126384, -1.11864475, ...,  0.31929428,
         0.25210868,  0.25210868],
       ...,
       [ 1.68065927,  0.78103476,  2.37445166, ...,  2.26626907,
         0.29049435,  0.29049435],
       [ 3.14375569,  0.78103476,  2.37445166, ...,  2.25832223,
        -1.28759013, -1.28759013],
       [ 3.24129545,  0.78103476,  2.37445166, ...,  2.2503754 ,
         0.6298737 ,  0.6298737 ]])
```

An alternative standardization is scaling features to lie between a given minimum and maximum value, often between zero and one, or so that the maximum absolute value of each feature is scaled to unit size. This can be achieved using MinMaxScaler or MaxAbsScaler, respectively, We use MinMaxScaler here to make sure all these data is between -1 to 1.

Scaling features to a range

```
min_max_scaler = preprocessing.MinMaxScaler(feature_range=( -1, 1))
complete_minmax = min_max_scaler.fit_transform(complete)
```

```
complete_minmax
```

```
array([[-0.90654206, -0.14285714, -0.34530095, ...,  0.07692308,
        -0.46910219, -0.46910219],
       [-0.90654206, -0.14285714, -0.34530095, ...,  0.0678733 ,
        -0.25583421, -0.25583421],
       [-0.92523364, -0.14285714, -0.34530095, ...,  0.05882353,
         0.14569532,  0.14569532],
       ...,
       [-0.51401869, -0.71428571,  0.83949314, ...,  0.79788839,
         0.16795719,  0.16795719],
       [-0.23364486, -0.71428571,  0.83949314, ...,  0.79487179,
        -0.74725708, -0.74725708],
       [-0.21495327, -0.71428571,  0.83949314, ...,  0.7918552 ,
         0.36478114,  0.36478114]])
```

Normalization is the process of scaling individual samples to have unit norm. This process can be useful if you plan to use a quadratic form such as the dot-product or any other kernel to quantify the similarity of any pair of samples.

Normalization

```
#Method_1
complete_normalized = preprocessing.normalize(complete, norm='l2')
```

```
complete_normalized
```

```
array([[0.07858774, 0.03929387, 0.02605184, ..., 0.00694192, 0.01738811,
        0.01738811],
       [0.07859288, 0.03929644, 0.02605354, ..., 0.00681138, 0.02437191,
        0.02437191],
       [0.06553308, 0.03931985, 0.02606906, ..., 0.00668437, 0.03754084,
        0.03754084],
       ...,
       [0.32931037, 0.01219668, 0.03110154, ..., 0.01618093, 0.03561323,
        0.03561323],
       [0.47736441, 0.01136582, 0.02898284, ..., 0.01504077, 0.00718636,
        0.00718636],
       [0.48538035, 0.01128792, 0.02878418, ..., 0.01490005, 0.03851307,
        0.03851307]])
```

It’s useful to add complexity to the model by considering nonlinear features of the input data. A simple and common method to use is polynomial features,

which can get features' high-order and interaction terms. It is implemented in PolynomialFeatures.

```
degree = np.arange(0, 21)
train_score, val_score = validation_curve(gen_rf_pipeline(0.8,5,25,), x_train, y_train,
                                          'polynomialfeatures__degree', scoring='precision' degree, cv=5)

plt.plot(degree, np.median(train_score, 1), color='blue', label='training score')
plt.plot(degree, np.median(val_score, 1), color='red', label='validation score')
plt.legend(loc='best')
plt.ylim(0, 1)
plt.xlabel('degree')
plt.ylabel('score');
```

And we will do the feature transformation according to the feature importances found.

- Thirdly, we select univariate feature. Univariate feature selection works by selecting the best features based on univariate statistical tests. It can be seen as a preprocessing step to an estimator. Sklearn exposes feature selection routines as objects that implement the transform method and code is like below:

```
def Kbest_by_f_regressor(num,x_pre,y):
    x_post = SelectKBest(f_regression, k=num).fit_transform(x_pre, y)
    return x_post

def Kbest_by_mutual_info(num,x_pre,y):
    x_post = SelectKBest(mutual_info_regression, k=num).fit_transform(x_pre, y)
    return x_post

def perc_by_f_regressor(num,x_pre,y):
    if(num > 100 or num < 0):
        print('wrong percentage: {}%'.format(num))
        return
    x_post = SelectPercentile(f_regression, percentile=num).fit_transform(x_pre, y)
    return x_post

def perc_by_mutual_info(num,x_pre,y):
    if(num > 100 or num < 0):
        print('wrong percentage: {}%'.format(num))
        return
    x_post = SelectPercentile(mutual_info_regression, percentile=num).fit_transform(x_pre, y)
    return x_post
```

- Fourthly, we can select best features by an external estimator. Given an external estimator that assigns weights to features (e.g., the coefficients of a linear model), recursive feature elimination (RFE) is to select features by recursively considering smaller and smaller sets of features. First, the estimator is trained on the initial set of features and the importance of each feature is obtained either through a coef_attribute or through a

feature_importance_attribute. Then, the least important features are pruned from current set of features.That procedure is recursively repeated on the pruned set until the desired number of features to select is eventually reached. Code is below.

```
def recursively_sel(regressor,num, x_pre,y):
    rfe = RFE(estimator=regressor, n_features_to_select=num, step = 1)
    rfe.fit(x_pre,y)
    x_post = rfe.transform(x_pre)
    return x_post
```

```
x_transformed = fea_sel_variancethreshold(x_train)
print(x_transformed)
```

- *Model Training:*

As requested, we use Linear regression, Random forest, Neural networks as our training model.

- First, we build these 3 models.
- Second, we compute RMS, MAPE, R2 and MAE for every models.
- According to these values given above, Random forest is the best model for this case for now.

- *Model Validation and Selection:*

- Conclusion: we implemented validation_curve in a loop to see the scores of the model with different degrees. Finally, we found that the model got the highest score when degree equals 1. In other words, the model is in best performance when degree is 1.
- Why we use these model validation technology, such as cross validation techniques, bias-variance tradeoff.

Cross validation techniques: one disadvantage of using a holdout set for model validation is that we have lost a portion of our data to the model training. In the above case, half the dataset does not contribute to the training of the model! This is not optimal, and can cause problems – especially if the initial set of training data is small.

One way to address this is to use cross-validation; that is, to do a sequence of fits where each subset of the data is used both as a training set and as a validation set.

Cross validation

```
1 score_result = cross_val_score(nn, x_test, y_test, cv = 10)
```

```
1 score_result
```

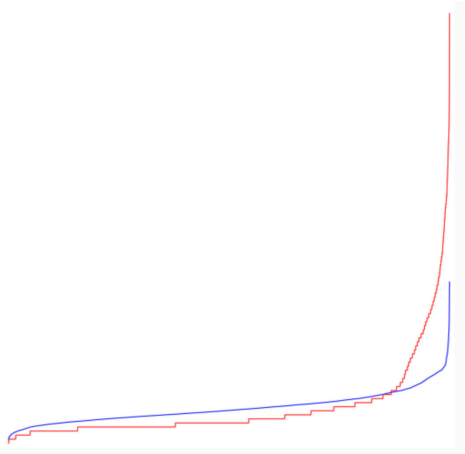
Bias-variance tradeoff: fundamentally, the question of "the best model" is about finding a sweet spot in the tradeoff between bias and variance. This technology help us choose appropriate model complexity.

Validation_curve for different pipelines(to find the best degree)

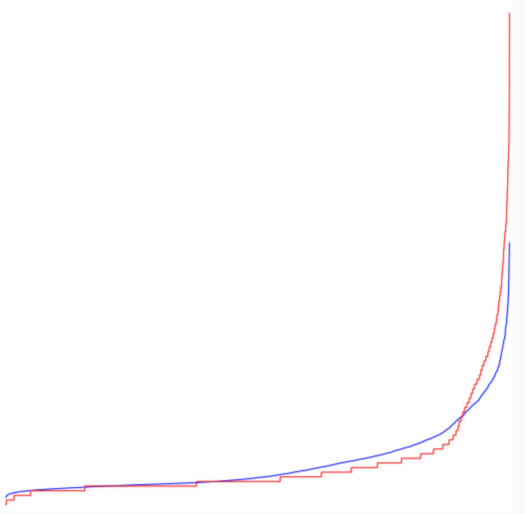
```
degree = np.arange(0, 21)
train_score, val_score = validation_curve(gen_rf_pipeline(0.8,5,25,), x_train, y_train,
                                          'polynomialfeatures__degree', scoring='precision' degree, cv=5)

plt.plot(degree, np.median(train_score, 1), color='blue', label='training score')
plt.plot(degree, np.median(val_score, 1), color='red', label='validation score')
plt.legend(loc='best')
plt.ylim(0, 1)
plt.xlabel('degree')
plt.ylabel('score');
```

After all these work done, we drew several plots to visualize the performance of three different models, the result is as follows:

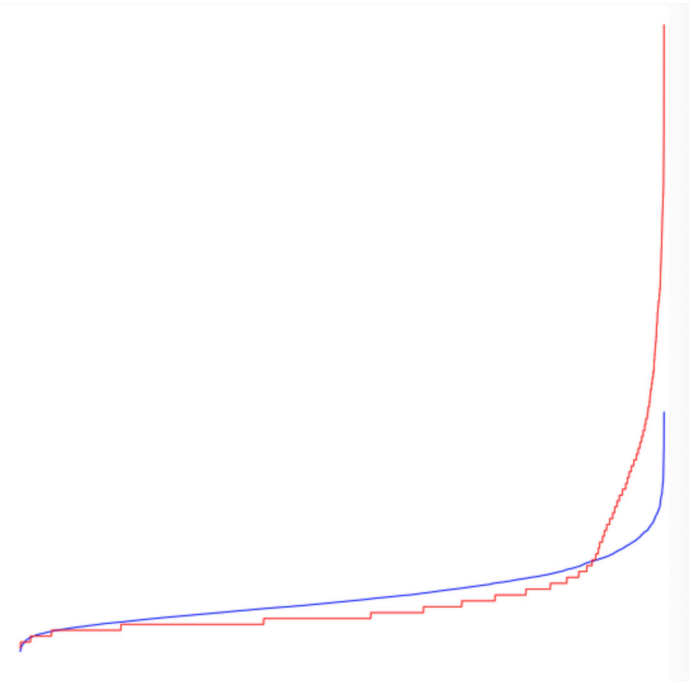
1. Linear Regression Model:



2. Random Forest Model:



3. Neural Network Model:



- *Final Pipeline:*

Pipelines

```
def gen_linear_pipeline(vp, up, num, degree=2):
    return make_pipeline(low_variance_selector(vp),
                         univariant_selector(up),
                         RFE_selector(num),
                         pp.PolynomialFeatures(degree),
                         LinearRegression()
                        )

def gen_rf_pipeline(vp, up, num, md, ne, degree=2):
    return make_pipeline(low_variance_selector(vp),
                         univariant_selector(up),
                         RFE_selector(num),
                         pp.PolynomialFeatures(degree),
                         RandomForestRegressor(max_depth=md, n_estimators=ne)
                        )

def gen_nn_pipeline(vp, up, num, hls, al=0.0001, degree=2):
    return make_pipeline(low_variance_selector(vp),
                         univariant_selector(up),
                         RFE_selector(num),
                         min_max_scaler(),
                         pp.PolynomialFeatures(degree),
                         MLPRegressor(hidden_layer_sizes=hls, alpha=al)
                        )
```