Renewable Energy Analysis Lab
Undergraduate Research

Wind Power Scenario Generation with Machine Learning Algorithms

Daehyun Kim

Electrical Engineering
University of Washington

Supervised by:
PhD candidate Yishen Wang

Faculty Advisor:
Professor Daniel Kirschen

April 4, 2017

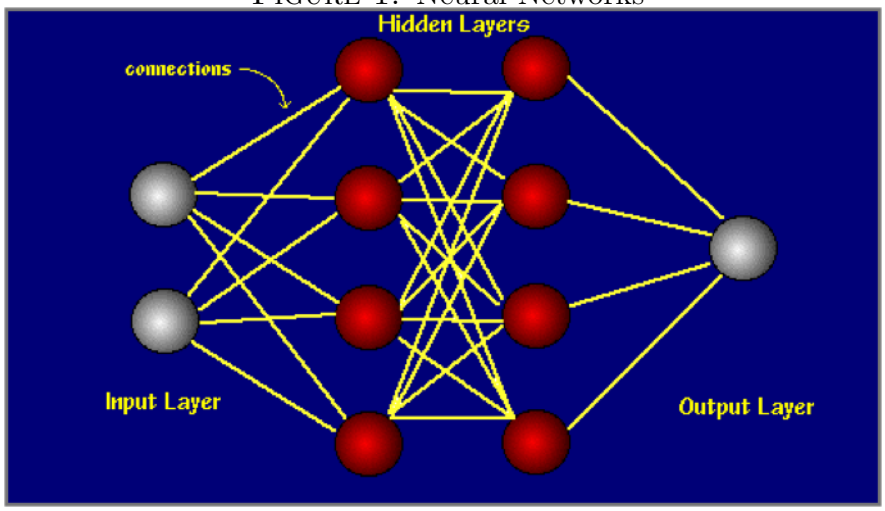
Introduction

Though the fossil fuel has accounted for the greatest portion of energy sources since it started being used in 1800s, concerns about its limited amount and the destruction of the environment caused by it have led to growing demands for alternative energy sources. Out of many developing alternative energy sources, wind energy is one of the key sustainable energy sources due to multiple advantages it has. Wind energy does not cause air pollution as does the fossil fuel, and according to the Wind Vision report, wind energy has the potential to reduce overall greenhouse gas emissions by 14%, saving $400 billion in avoided damage by 2050 [4]. Wind energy is also a cost-effective, inexhaustible energy source from the prolific wind supply in the U.S. [4]. Despite these plus points of the wind energy, its nature of uncertainty and variability present some challenges. Uncertainty basically refers to unpredictability in that it is impossible to have a perfect forecast of wind power. Variability, on the contrary, lies in the oscillating nature of wind power. In order to manage the wind power generating system in the most stable and effective manner given the challenges, it is necessary to develop and test advanced wind modeling and forecasting algorithms. In this paper, multiple machine learning algorithms that are widely utilized in the field will be evaluated through the comparison between the predicted and real values. The algorithms will be tested with the Western Wind Integration Data Set in 2006 from the National Renewable Energy Laboratory, NREL, which was designed for wind integration studies in the United States. The input data will be pre-processed in Python scripts to generate training and testing set, and these feature sets will be implemented in the open-source machine learning package called Scikit-Learn to produce wind power modeling. Then, the resulting prediction will be evaluated based on the errors from the real data set. Finally, other variations as well as further studies will be discussed.

Background

There exist many prediction algorithms that are developed by researchers to provide advanced data modeling. This paper will explore a portion of the widely-used regression algorithms, which include Neural Networks, Random Forest, Gradient Boosting, Nearest Neighbor, Support Vector Machines, Kernel Ridge Regression, and the most basic Linear Regression. This section will briefly discuss each algorithm's structure and function. Neural Networks are modeled after "the neuronal structure of the mamalian cerebral cortex" [8]. Neural Networks are composed of multi-perceptron layers. Among these layers are the input layer through which patterns are presented to, the hidden layer where the actual processing is done by weighted connections, and the outer layer which outputs the result as in Figure 1 below. Neural Networks are good with noisy data and when the function mapping between input and output data is unknown. It, however, is difficult to comprehend the resulting weights and may also take longer than other methods.

FIGURE 1. Neural Networks



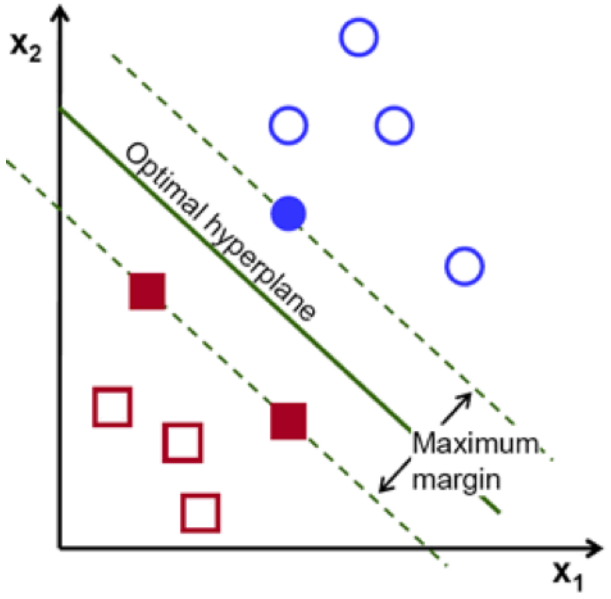
Random Forest is one of the ensemble methods. Ensemble methods combine individual weak approximations to form a strong approximation to the data. Random Forest is a strong approximation made up of many weak decision trees where a decision tree is a tree in which an input is entered at the top and the data gets divided into smaller sets down the tree [2]. In a Random Forest, a new input is run down all of the trees and the average of the terminal nodes is generally the final output. This algorithm is relatively fast and good with unbalanced data, whereas there is a chance of overfitting noisy data.

Another popular ensembling method is Gradient Boosting. It also combines weak learners into a strong learner, but in a sequential manner which is called boosting compared to bootstrapping of Random Forest. Here, the loss function, $y = ax + b + e$ where e is the error, is gradually minimized with the Gradient Descent algorithm [5].

k Nearest Neighbors in regression is a simple algorithm which outputs the average of the values of k nearest neighbors around the known object value. It can assign weight to the neighbors so that the nearer neighbors contribute more to the output than the farther ones. The lazy algorithm is effective with large, noisy data set, but there is the need to tune parameter k [7].

Support Vector Machines, usually called SVM, utilize a separating hyperplane that maximizes the margin of the training data as in Figure 2. In other words, they find a hyperplane that best divides the training set into two classes. SVM are good with smaller, clean datasets and more space-efficient from using a subset of training data. They are, however, not as good with larger, noisy datasets [6].

FIGURE 2. Support Vector Machine



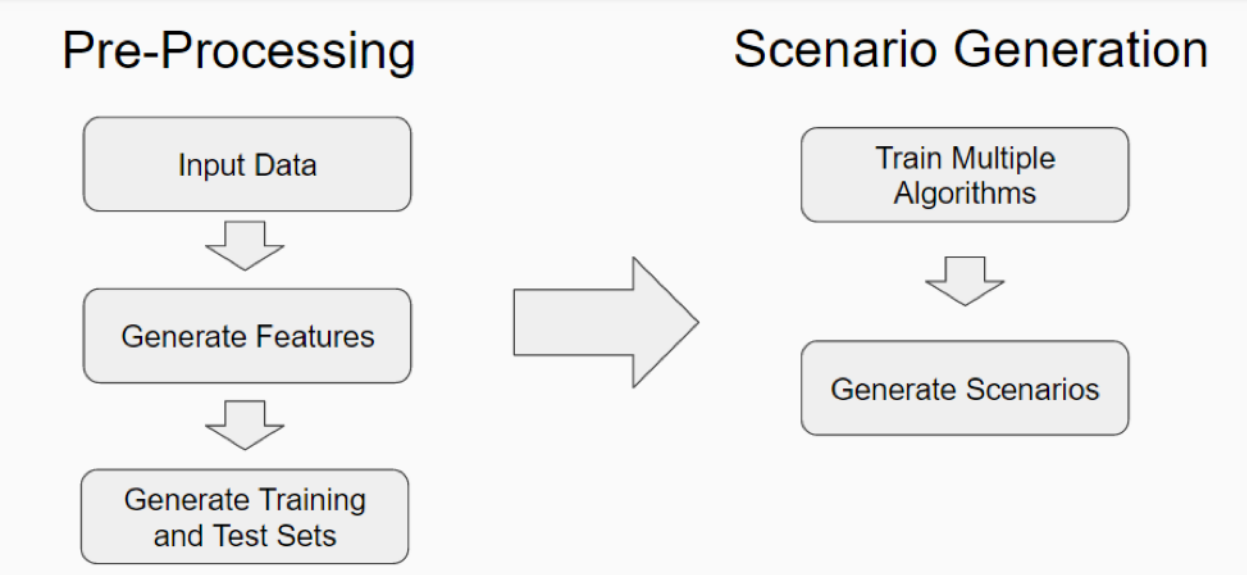
Kernel Ridge Regression combines Ridge Regression with the kernel trick. Ridge regression adds a regularization penalty to the cost, and Kernel Ridge Regression performs this in feature space by mapping data to higher dimensional space, which is called the kernel method [1].

Linear regression, the most basic form of regression methods, was implemented as well. Linear regression models the relationship between variables by fitting a linear equation to observed data [3]. Most commonly used technique for fitting a line is the least-squares approximation. This method minimizes the sum of the squares of the deviations from each data to the line.

Implementation

The wind data was collected from the Western Wind Integration Data Set in 2006 from the NREL. The entire implementation process consists of two stages: pre-processing and scenario generation. Figure 3 demonstrates a flowchart of the entire process.

FIGURE 3. Implementation Flowchart



Pre-Processing

The pre-processing phase collects wind speed and power data from the NREL West in 2006. The data is in the form of csv files containing wind power and wind speed in 10-min resolution for 230 wind sites. These wind sites are then grouped into 19 wind farms corresponding to their locations and the wind speed and power values are averaged for each wind farm in an hourly resolution. These values are normalized and divided by 30MW, which is the capacity of the wind farm. This process is done in the Python script called "read" using NumPy package. Another script called "parameter_gen" is a simple program that loads multiple parameters given the input in this case the wind power. Some of the parameters include "time lead"-the number of hours ahead of the prediction-, local historical hours, and resolution. The time lead has a big influence on the accuracy of modeling, and for this prediction, it is set to be 1 day. The local historical hours indicate the effect of recent information, and it is set to be 7 days as a number that is not too big nor too small. The last sub-script is called "feature_build" and this script builds the input features and output target from known time series utilizing the results from "read" and "parameter_gen". The input features include the most recent power and near future prediction, and the output target refers to the actual power output. All three scripts are called in the main script as

functions. The resulting input features and output target are then used for building training and testing sets. Day 124, day 221, and day 306 were randomly chosen out of the year for testing and prediction hours are up to 7 days, the number of local historical hours, after those days. For generating the prediction curves, only three farms-farm 1, farm 4, and farm 7- were tested, though these example testing days and farms can be manipulated in the script. The length of the training sets were initialized at 2160 hours which equals to 90 days. There are four sets to be generated: Feature_Train, Target_Train, Feature_Test, and Target_Test (These are coded as xTr, yTr, xTe, and yTe respectively). Feature_Train and Target_Train contain feature and target data for the 90-day period before the testing day. These two sets will be used to fit the model for each algorithm. Feature_Test and Target_Test contain feature and target data for a week after the testing day. These two sets are used to evaluate the performance of the trained model for each algorithm. All of the pre-processing scripts can be accessed in the Appendix section.

Scenario Generation

FIGURE 4. Scikit-Learn

After generating training and testing sets, the open-source package called Scikit-Learn (in Figure 4) is implemented in the main script to fit the model. This process is exercised in a simple manner given the built-in functions in the package. For example, a function called "fit" will fit the model using the training sets, and another function called "predict" will predict the target power using the Feature_Test set, which would be later compared to the real Target_Test set as a measure of each algorithms performance. After this prediction step, the prediction result is filtered to be in between 0 and 1 because wind power cannot be

less than 0 or more than 1 after normalized. In order to quantify the performance of each machine learning algorithm, root mean squared error and mean absolute error are calculated using the Target_Test set and the prediction result. The package also has a built-in measure of coefficients of determination, also known as "R sqaured," which stands for the proportion of the variance in the dependent variable that is predictable from the independent variable. The additional mean of evaluating algorithms includes the computing time which measures the time it takes for each algorithm to fit the model and predict. After the process with the package, MatplotLib in Python is utilized to create graphical results for each testing day and testing farm demonstrating how each algorithm compares to the real target data. Moreover, the calculated errors, coefficients of determination, and computing time are visualized for all 19 farms, which provide the quantitative measures of the performance of each algorithm. Lastly, the same visual representations will be proceeded with varied time leads to understand what impact time lead has on the performance of each algorithm. The three different main scripts that contain the scenario generation phase are attached in the Appendix section.

Results

The following three figures are the results of the prediction process. The first figure represents the predicted power generated for 7 days from the requested testing day for the Farm 102, the first farm. The other two figures are the same visuals but for Farm 117 and Farm 120, the fourth farm and the seventh farm respectively. These images merely provide the general sense of which algorithms prediction diverges away from the real data, but it is hard to observe the images and have a quantitative understanding of each algorithms performance. Some consistent pattern we can deduce from these three graphs is that Nearest Neighbor represented by a pink curve constantly deviates from the read data represented by a dotted line. Also another interesting aspect is the result for the seventh farm, Farm 120, indicates that Random Forest and Gradient Boosting, represented by blue and sky blue curves respectively, have a poor prediction unlike for the other two results.

FIGURE 5. Prediction for Farm102

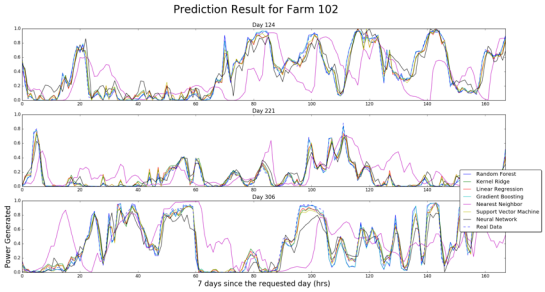


FIGURE 6. Prediction for Farm117

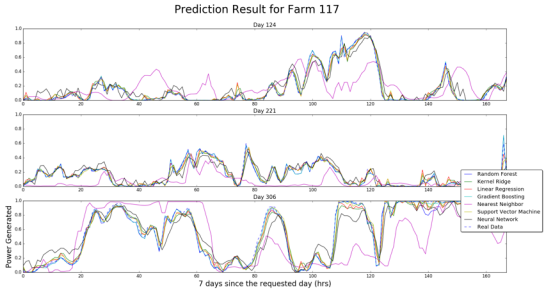
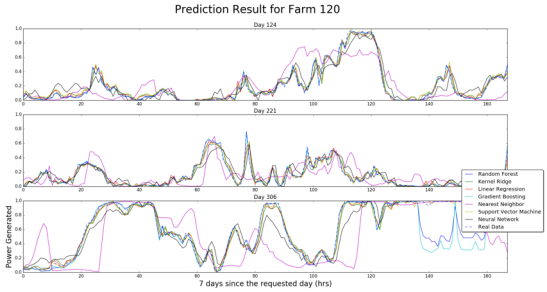


FIGURE 7. Prediction for Farm120



In order to evaluate the performance of each algorithm in a quantitative manner, a simple command in the package that calculates the coefficient of determination, the proportion of a dependent variable that can be predicted from an independent variable, was implemented (1-the highest possible and 0-the lowest). Other measures that were implemented in the code include root mean squared error (RMSE), mean absolute error (MAE), and computing time, which are self-explanatory. These performance measurements were calculated not only based on the three example farms, but on the entire 19 farms.

FIGURE 8. RMSE

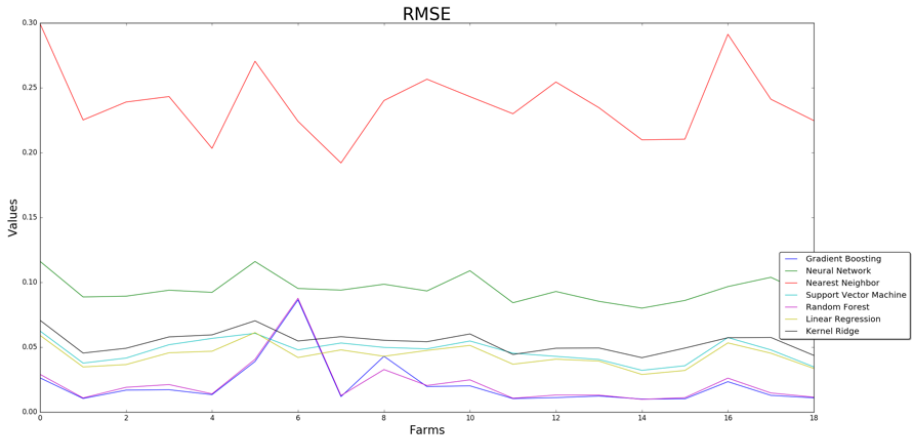
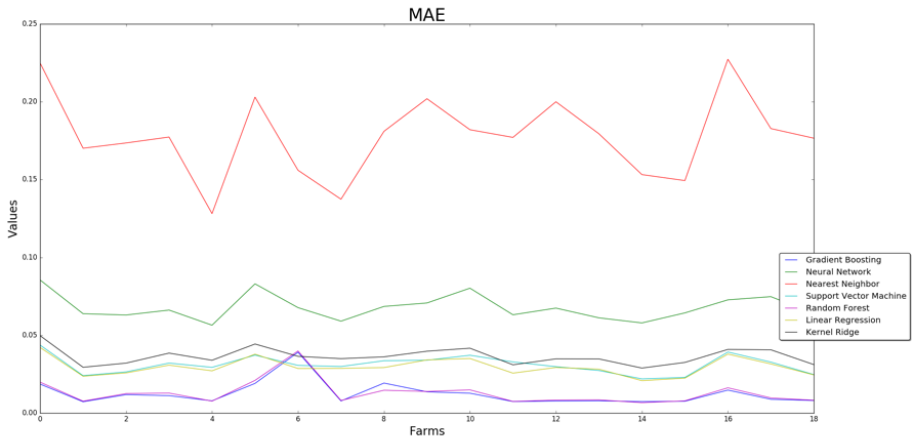
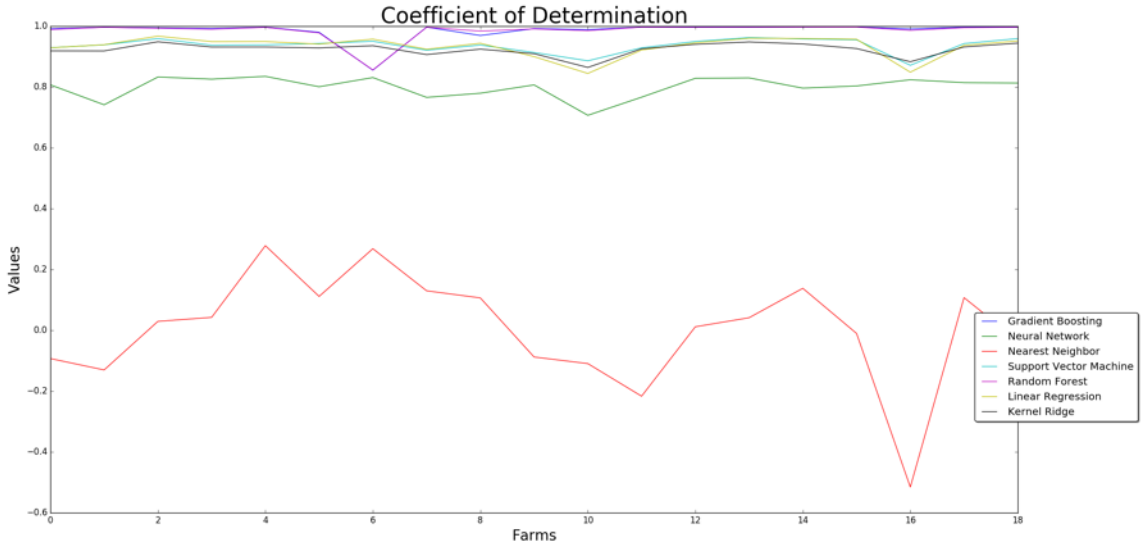


FIGURE 9. MAE



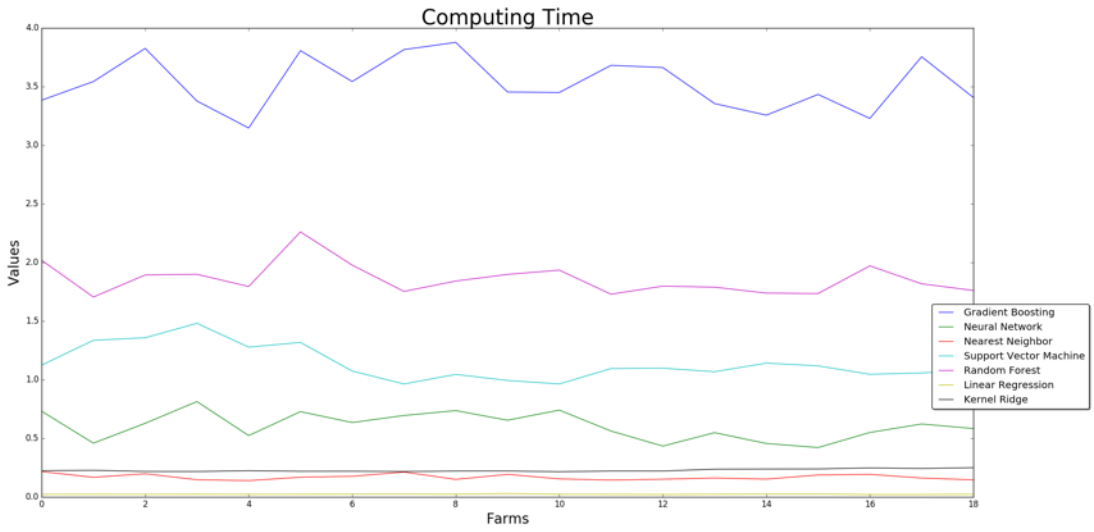
From Figure 8 and Figure 9, both RMSE and MAE graphs were consistent in giving Nearest Neighbor, the red line, as the most poorly predicted algorithm. The second worst algorithm was the Nearest Neighbor, the green line. The best two algorithms for the prediction were Random Forest and Gradient Boosting, purple and blue respectively. As mentioned before, these two best algorithms had some issues on the seventh farm so they have a sudden spike in error at that point. Kernel Ridge Regression had the biggest errors out of the three middle ground algorithms. The other two of the three, Support Vector Machine and unexpectedly Linear Regression had slightly better errors than Kernel Ridge, but it can be concluded that these three had similar errors.

FIGURE 10. Coefficient of Determination



Coefficient of Determination graph demonstrates a similar pattern as the error graphs. The top two, Gradient Boosting and Random Forest mostly have the coefficient of determination very close to the maximum 1.0, with the same spike on the seventh farm. Other four algorithms except Nearest Neighbor exhibits fairly high coefficients of determination higher than 0.8. The Nearest Neighbor is far off from the rest with coefficient of determination around 0. These error and R squared graphs have indicated that some of these algorithms are extremely incompetent in predicting the output correctly compared to the other algorithms. This is not completely true, because there barely was parameter tuning stage if at all, due to lack of time. For example, the two poorly done algorithmsNearest Neighbor and Neural Networksare known to be greatly influenced by their parameters such as k, number of nearest neighbors and number of neurons in the hidden layer respectively. This parameter tuning would have enhanced the performance of the algorithms to a large degree and it could be verified in the further studies.

FIGURE 11. Computing Time



The computing time graph says something different from the other measures. The best two algorithms in terms of the prediction result, Gradient Boosting and Random Forest were the worst two here with the highest computing time. Linear Regression expectedly was the best algorithm in terms of computing time given its simple nature. Except Linear Regression, the general pattern here was the algorithms with good prediction results took longer computing time and those with bad results did not have as much computing time.

Variations

For the main prediction process, time lead, number of hours ahead of the prediction, was pre-defined as 1 day. The time lead has a significant impact on the prediction performance because a smaller time lead results in a better modeling, while a bigger time lead causes more errors while simulating a realistic forecast. With forecast data given, the time lead can be manipulated to varied numbers as large as 24 days before running the prediction process. As mentioned above, making the time lead that large would indicate that the prediction is truly exercising the short-term forecast.

FIGURE 12. Farm102_1 day

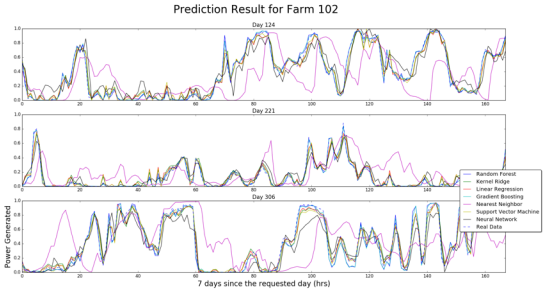
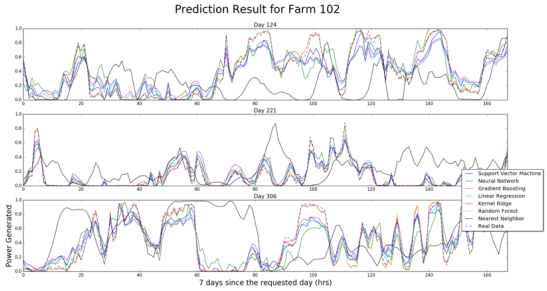


FIGURE 13. Farm102_24 days



The result of increasing time lead from 1 day to 24 days for the first farm, Farm 102, is displayed on the above two graphs. It is evident that the increased time lead has enlarged the deviation errors to a great degree. Not only the previously poorly done Nearest Neighbor but also are the rest of the algorithms far off the real data. Only Gradient Boosting shown by the orange line fairly overlaps the real data, the dotted line. The similar results for the other two example farms, 117 and 120, are included in the Appendix. This correlation is even more noticeable in the charts that compute the errors and coefficient of determination for the varied time leads. The following charts averaged the errors and coefficient of determination among the first 10 farms-all 19 farms consume too much time- for multiple time leads: 1, 2, 4, 6, 12, and 24 days.

FIGURE 14. RMSE_Variation

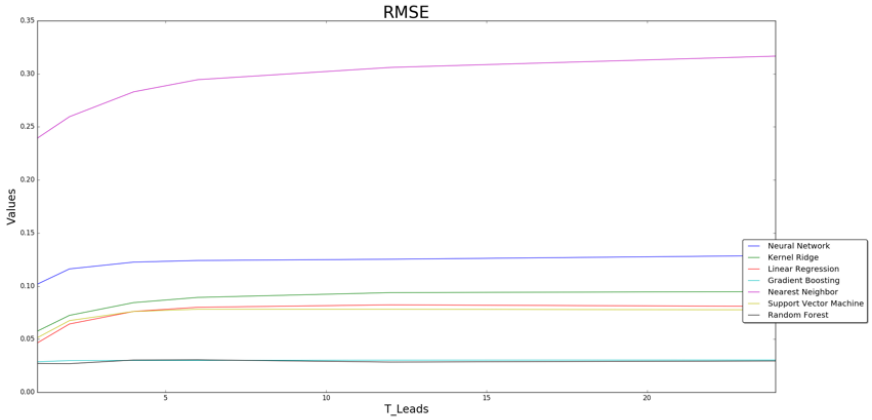
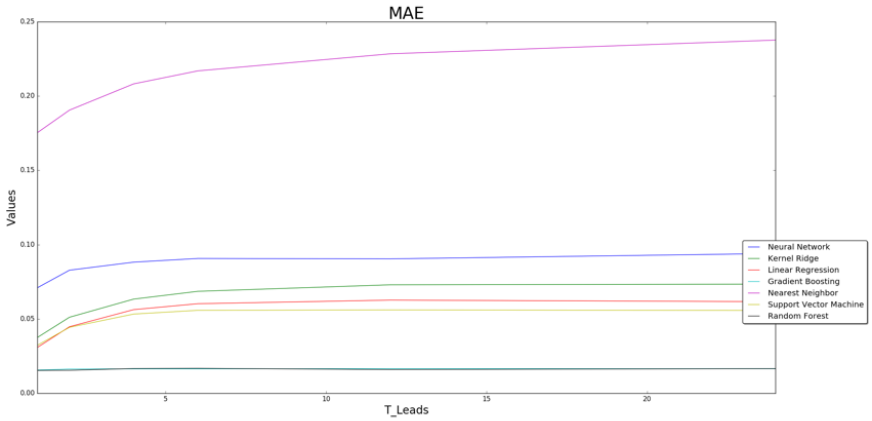
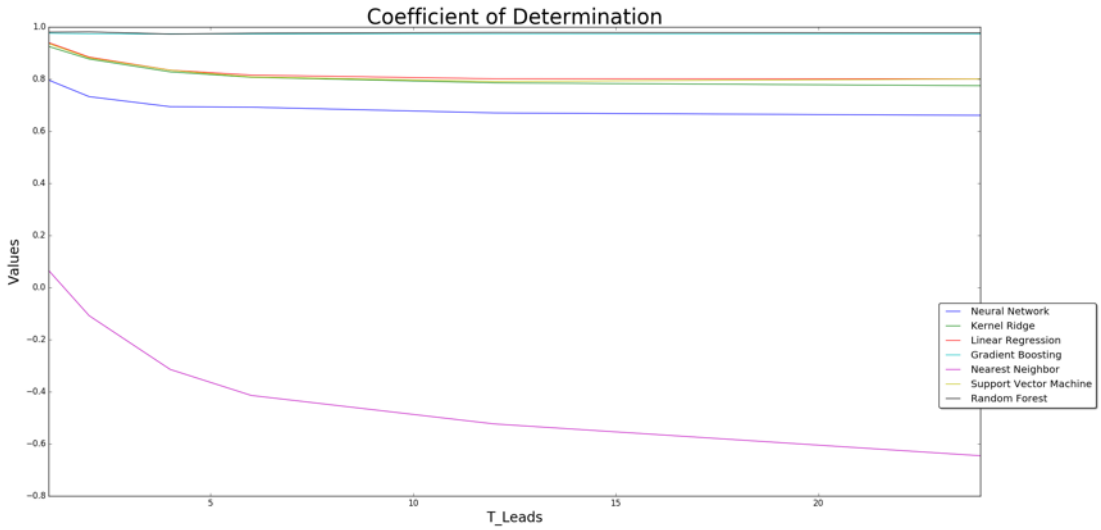


FIGURE 15. MAE_Variation



Both RMSE and MAE charts for varied time leads indicate a similar pattern that the longer the time lead, the larger the deviation errors. Even though Gradient Boosting and Random Forest algorithms show consistently low errors, the other 5 algorithms all had their root mean squared and mean absolute errors rise noticeably. The increase in the errors is steeper towards the fourth time lead value, 6 days, and then flattens out after.

FIGURE 16. CoD_Variation



The same trend occurs in the coefficient of determination graph, but in the opposite direction since larger errors correspond to smaller coefficients of determination. Likewise, the coefficients of determination for the two best algorithms stay constant right below 1.0. For the other 5 algorithms, coefficients of determination decrease rapidly up to time lead at 6 days, and relatively slows down. Overall, these three charts lead to the conclusion that an increased time lead causes a less accurate modeling and a bigger deviation error.

Conclusion

Through examining results of machine learning implementation on wind power series the report has developed an original insight into the the performance of several machine learning algorithms from an open-source package. The ultimate goal in this project was to test out the modeling done by the open-source machine learning algorithms with the real wind power data and visually represent the results on multiple farms with and without variations in time lead. Open-source machine learning packages like Scikit-Learn provide a convenient method of evaluating data modeling, but it has its drawbacks in that it may lead to implementing the algorithms without full understanding their principles. From the overall analysis it was concluded the best algorithms that closely predicted the wind power output were Random Forest and Gradient Boosting despite their large computing time, while the worst were Nearest Neighbor and Neural Networks. As discussed previously, these results were expected since Nearest Neighbor and Neural Networks are more responsive to parameter tuning which was not exercised in the current project than are other algorithms. Additional analysis on variation of time lead has led to the verification of the hypothesis that the increase in time lead causes more deviation errors for all algorithms. There still are many other areas to work on in the further studies. As mentioned above, there is an opportunity to boost the performance of some algorithms by tuning their parameters. Also, cross validation stage and spatial features can be added to the process which would further reduce the deviation errors for all algorithms. On the other hand, variations on differentiating the testing process include testing with different open-source packages like TensorFlow and testing with different wind data such as those from the East coast. Furthermore, compared to sampling only one scenario from each algorithm in the current project, sampling multiple scenarios at once will give more interesting outlooks. This project along with its potentially improved parts in the further studies will offer the essential asset to the growing development of predicting wind power with machine learning.

References

[1] Kristin Bennet. *Kernel Ridge Regression*. Ed. by Rensselaer Polytechnic Institute. Accessed March 3, 2017. URL: http://homepages.rpi.edu/~bennek/class/mds/lecture/lecture6-06.pdf.

[2] Dan Benyamin. *A Gentle Introduction to Random Forests, Ensembles, and Performance Metrics in a Commercial System*. Ed. by CitizenNet. Accessed March 3, 2017. URL: http://blog.citizennet.com/blog/2012/11/10/random-forests-ensembles-and-performance-metrics.

[3] Yale University Statistics Department, ed. *Linear Regression*. Accessed March 3, 2017. URL: http://www.stat.yale.edu/Courses/1997-98/101/linreg.htm.

[4] Energy.gov. Office of Energy Efficiency Renewable Energy., ed. *Advantages and Challenges of Wind Energy*. Accessed March 3 , 2017. URL: https://energy.gov/eere/wind/advantages-and-challenges-wind-energy.

[5] Sunil Ray. *Quick Introduction to Boosting Algorithms in Machine Learning*. Ed. by Analytics Vidhya. Accessed March 3, 2017. URL: https://www.analyticsvidhya.com/blog/2015/11/quick-introduction-boosting-algorithms-machine-learning/.

[6] Sunil Ray. *Understanding Support Vector Machine algorithm from examples (along with code)*. Ed. by Analytics Vidhya. Accessed March 3, 2017. URL: https://www.analyticsvidhya.com/blog/2015/10/understaing-support-vector-machine-example-code/.

[7] Statistica, ed. *k-Nearest Neighbors*. Accessed March 3, 2017. URL: http://www.statsoft.com/textbook/k-nearest-neighbors.

[8] University of Wisconsin Computer Science, ed. *A Basic Introduction to Neural Networks*. Accessed March 3 , 2017. URL: http://pages.cs.wisc.edu/~bolo/shipyard/neural/local.html.

Appendix

The following figures are side-by-side graphs of predictions along with their variation graphs with time lead at 24 days.

FIGURE 17. Farm117_1 day

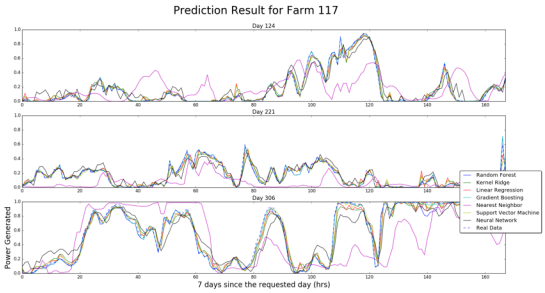


FIGURE 18. Farm117_24 days

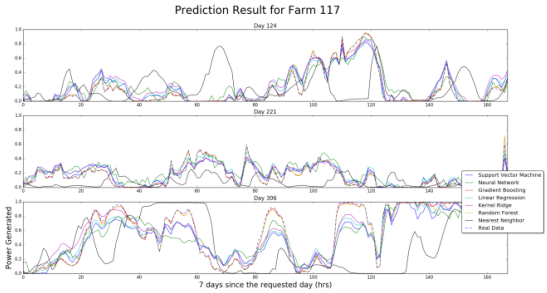


FIGURE 19. Farm120_1 day

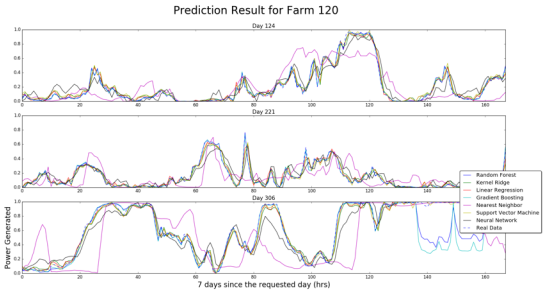
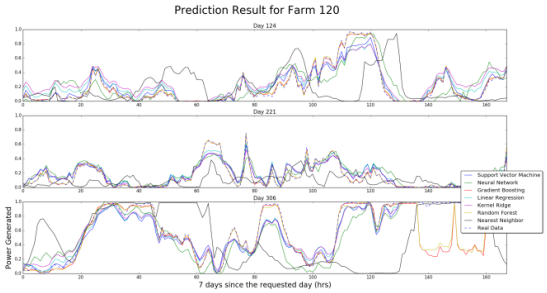


FIGURE 20. Farm120_24 days



Here are some information about the machine, and software versions.

Machine Information:

```
Computer           : Asus Q550L
Operating system   : Windows 10 (C) 2016 Microsoft Corporation
System type        : 64-bit Operating System, x64-based processor
Processor          : Intel(R) Core(TM) i7-4712HQ CPU @2.3GHz
Memory             : 16.0GB
```

Software Information:

```
Eclipse
Python 3.5.1
```

The following is the Code "read.py" for data pre-processing. It obtains the wind turbine parameters from NREL west wind dataset.

```
import csv

def readData(wf_name, wf_idx, resolution, year):
#     get the wind turbine parameters from NREL west wind dataset
#     each csv file contains a 10-min wind output in a year with 30MW
#     input:
#         wf_name: the cell matrix for wind farm info, nLocation*1
#                  matrix, each cell contains the wf name list in csv
#         wf_idx: the matrix store the numbered folder contans csv
#         year: if year is 2006 or 2005, have 52560 measurements
#         resolution: the resolution within one hour
#     output:
#         speed: averaged wind speed among local area, nLocation*1 cell in
#                desired resolution
#         gen: total wind power generation among local area, nLocation*1 cell
#              in desired resolution
#         wind_param: the output wind csv data file in the cell format, each
#                     cell is one Location, contains n turbine
#         capacity: the capacity of each wind farm, nLocation*1 matrix
#  data:
#       load data under the local folder '2006/'
#  ========================================================================

    nLocation = len(wf_name) # number of sites

    if year % 4 == 0:        # number of measurements
        nRow = 6*24*366      # lunar year
    else:
        nRow = 6*24*365

    # initialization
    wind_param = []
    speed_temp = []
    speed = []
    gen = []
    gen_temp = []
    capacity = []

    for iLocation in range(nLocation):
```

```
        wf_id = wf_idx[iLocation] # the name (number) of iLocation in RTS
        farm_idx = wf_name[iLocation] # pick wind sites in ith Location
        nSite = len(farm_idx) # number of sites in iLocation
        turbine_param = np.zeros((nSite, nRow, 4)) # parameters in each farm

        # copy from csv files into the matrices
        for iSite in range(nSite):
            with open('./2006/2006/' + str(wf_id) + '/' + str(farm_idx[iSite]) + '.csv') as f:
                reader = csv.reader(f)
                next(reader)
                count = 0
                for row in reader:
                    turbine_param[iSite, count, :] = row[1:]
                    count += 1
        # capacity
        loc_capacity = 30*nSite
        capacity.append(loc_capacity)
        wind_param.append(turbine_param)
        speed_temp.append(np.mean(turbine_param[:,:,0],axis=0))
        gen_temp.append(np.sum(turbine_param[:,:,3], axis=0)/(loc_capacity))

        # 1-hr resolution
        if resolution == 1:
            speed_per_hour = np.reshape(speed_temp[iLocation], (nRow//6, 6))
            gen_per_hour = np.reshape(gen_temp[iLocation], (nRow//6, 6))
            speed.append(np.mean(speed_per_hour, axis=1)/30)
            gen.append(np.mean(gen_per_hour, axis=1))
        # 10-min resolution
        elif resolution == 6:
            speed.append(speed_temp[iLocation]/30)
            gen.append(gen_temp[iLocation])
        else:
            print ("desired resolution is not valid.")

    speed = np.array(speed)
    gen = np.array(gen)
    wind_param = np.array(wind_param)
    capacity = np.array(capacity)

    return speed, gen, wind_param, capacity
```

The following is the Code "parameter_gen.py" for data pre-processing. It obtains program parameters based on given input.

```
from collections import namedtuple
import numpy as np

def parameter_gen(x, t, t_scale, t_lead, space_bool):

#  obtain program parameters based on given input
#  ==============================================
#  input:
#     x: a cell array, each cell is a data series
#     t: number of days considering for the feature
#     t_scale: number of points in one hour
#     t_lead:  leading time for prediction = t_horizon
#     space_bool: 1 if space considered, 0 otherwise
#  output:
#     para: a structure indicating many parameters

# ========= the system parameters ==================
    # initialize a structure of parameters called para
    para = namedtuple("para", "nFarm nSeries horizon resolution fea_hist fea_pred fea_type spa
    [nFarm, nSeries] = np.shape(x) # number of wind farms and overall datapoints
    horizon = t_lead               # forecast horizon, lead time
    resolution = t_scale           # hourly data

# ========== feature building ====================
    fea_hist = 24*t                # input feature length for history hours before prediction
    fea_pred = 24*t//2     # input feature length for day-ahead predictions
    fea_type = 1                   # number of features type include power and speed

    if (space_bool == 0):
        spa_hist = 0                   # input feature length for nearby farm history days
        spa_pred = 0                   # input feature length for nearby farm day-ahead predict
        spa_nloc = 0                   # number of extra locations builds
    elif (space_bool == 1):
        spa_hist = 24*t
        spa_pred = 24*t//2
        spa_nloc = 3

    drop_length = resolution*fea_hist + horizon
    # dropped data length
```

```
    nSample = nSeries-drop_length  # total sample size
    nFeature = ((fea_hist+fea_pred)*fea_type+(spa_hist+spa_pred)*spa_nloc)*resolution
    # total length for each input vector

# =========== evaluation criteria ==================
    evaluation = 'RMSE'            # evaluation criteria: MAE or RMSE

    p = para(nFarm, nSeries, horizon, resolution, fea_hist, fea_pred, fea_type, spa_hist, spa_
    return p
```

The following is the Code "feature_build.py" for data pre-processing. It builds the input vector and output from known time series without spatial features.

```
import numpy as np

def feature_build(power, speed, para):
#  build the input vector and output from known time series without spatial
#  ============================================================
#  skip the first part that don't have the data
#  input features include the most recent power and near future prediction
#  output target is the the actual power output
#  features are in the recent order, the prediction, the more recent,
#  and nearby sites the index is smaller
#  input:
#     power: nLoc*1 cell, each with wind generation, also as input feature
#     speed: nLoc*1 cell, each with wind speed, as input feature
#     para:  parameters to decide the whole model information
#  output:
#     feature:  nLoc*1 cell, each contains m_sample*nFeatures
#     target:   nLoc*1 cell, each contains m_sample*2,col1_true,col2_pred

    nFarm = para.nFarm

    nDrop = para.drop_length           #  length of dropped data = fea_hist + horizon
    nSample = para.nSample             #  number of whole sample excluding dropped data

    nFeaTotal = para.nFeature          #  total feature length = fea_hist+fea_pred if no space

    nFeaHist = para.fea_hist*para.resolution
    #  feature length for power series (fea_hist)

    nFeaSpeed = nFeaHist//2             #  feature length for speed series (fea_pred)

    feature = []
    target = []

    # building features
    for iFarm in range(nFarm):
        fea_temp = np.empty((nSample, nFeaTotal))
        # set up input feature
        for iFea1 in range(nFeaHist):
            # add history as input feature
```

```
            fea_temp[:,iFea1] = power[iFarm][nDrop-para.horizon-iFea1 : para.nSeries-para.horiz

        for iFea2 in range(nFeaSpeed):
            fea_temp[:,nFeaHist+iFea2] = speed[iFarm][nDrop-iFea2 : para.nSeries-iFea2]

        # set up target output, throw away the drop_length data
        temp = [power[iFarm][nDrop:para.nSeries]]
        target.append(np.transpose(temp))
        feature.append(fea_temp)

    feature = np.array(feature)
    target= np.array(target)

    return feature, target
```

The following is the Code "main1.py". It performs data processing and scenario generation and ends with graphical prediction results for three testing days and three testing farms.

```
import os
import numpy as np
import math
import time
import matplotlib.pyplot as plt
from Input.read import readData
from Input.parameter_gen import parameter_gen
from Input.feature_build import feature_build

from sklearn import linear_model
from sklearn import svm
from sklearn.ensemble import RandomForestRegressor
from sklearn.ensemble import GradientBoostingRegressor
from sklearn import neural_network
from sklearn import kernel_ridge
from sklearn import neighbors

# WIND FORECAST CORRECTIVE SCENARIOS for 19 wind farms
# This script ends with graphical prediction results for
# three testing days and three testing farms.
#   ================================================================
#   generate wind scenarios based on historic data
#   considering spatial and temporal correlation
#   provide scenarios with better forecast
#   provide scenarios with uncertainty quantification
#   provide reasonable boundary with scenarios
#   combining multiple data mining techniques
#   including Random Forest, SVM, Linear Regression, KNN, NN
#   the data is based on the NREL Western Wind Dataset

# Load Data
start_time = time.time()
year = 2006
resolution = 1  # 1 hr resolution
speed = []
gen = []
# directory where the data is stored
dataDir = os.listdir('./' + str(year) + '/' + str(year))
# number of wind farms in the directory
```

```
nLocation = len(dataDir)

# list storing names of wind farms
wf_idx = []
for dirname in dataDir:
    wf_idx.append(dirname)

nSites = 0
# list storing names of wind sites in each wind farm
wf_name = []
for dirname in dataDir:
    temp = []
    for filename in os.listdir('./' + str(year) + '/' + str(year) + '/' + dirname):
        temp.append(os.path.splitext(filename)[0])
        nSites = nSites+1
    wf_name.append(temp)

wf_idx = np.array(wf_idx)
wf_name = np.array(wf_name)
print(nSites) # 230 in total

# Output cleaned wind speed and power based on the given data
speed, gen, wind_param, capacity = readData(wf_name, wf_idx, resolution, year)
# Load Parameters
para = parameter_gen(gen, 5, resolution, 1, 0)

# Build Feature and Target
feature, target = feature_build(gen, speed, para)
print(np.shape(feature[0]))
print(np.shape(target[0]))

# Build Training and Test sets
days = [124, 221, 306] # testing days: can be manipulated
farms = [0, 3, 6] # testing farms: can be manipulated
farm_axis = np.arange(nLocation)

for f in range(len(farms)):
    fig = plt.figure()
    for i in range(len(days)):
        # prediction hours: 7 days
        test_hour = np.arange((days[i]-1) * 24, (days[i]+6) * 24) - para.drop_length
        test_time = np.transpose(test_hour)
```

```
train_length = 2160 # length of training sets

nFarm = nLocation
xTr = []
yTr = []
xTe = []
yTe = []

# build training and testing sets here
for iFarm in range(nFarm):
    xTr1 = feature[iFarm][test_time[0]-train_length : test_time[0]]
    yTr1 = target[iFarm][test_time[0]-train_length : test_time[0]]
    xTe1 = feature[iFarm][test_time[0]:test_time[len(test_time)-1]+1]
    yTe1 = target[iFarm][test_time[0]:test_time[len(test_time)-1]+1]

    xTr.append(xTr1)
    yTr.append(yTr1)
    xTe.append(xTe1)
    yTe.append(yTe1)

xTr = np.array(xTr)
yTr = np.array(yTr)
xTe = np.array(xTe)
yTe = np.array(yTe)

print(np.shape(xTr[0]))
print(np.shape(yTr[0]))
print(np.shape(xTe[0]))
print(np.shape(yTe[0]))

# Scikit-Learn commands for multiple algorithms
Estimators = {
            "Linear Regression": linear_model.LinearRegression(),
            "Support Vector Machine": svm.LinearSVR(),
            "Kernel Ridge": kernel_ridge.KernelRidge(),
            "Random Forest": RandomForestRegressor(),
            "Gradient Boosting": GradientBoostingRegressor(),
            "Neural Network": neural_network.MLPRegressor(),
            "Nearest Neighbor": neighbors.KNeighborsRegressor()
}

# dictionary form to store prediction results
```

```
    y_test_predict = dict()

    for name, estimator in Estimators.items():
        t1 = time.time() # for computing time
        print (name, "------")
        # fit the training sets
        estimator.fit(xTr[farms[f]], yTr[farms[f]].reshape(len(yTr[farms[f]]),))
        # predict using each algorithm
        y_test_predict[name] = estimator.predict(xTe[farms[f]])

        # the wind power should be in the range of 0 to 1, so outliers should be taken care
        for h in range(len(y_test_predict[name])):
            if (y_test_predict[name][h] < 0):
                y_test_predict[name][h] = 0
            elif (y_test_predict[name][h] > 1):
                y_test_predict[name][h] = 1

        # root mean squared error
        rmse = math.sqrt(np.mean((y_test_predict[name] - yTe[farms[f]].reshape(len(yTe[farm
        # mean absolute error
        mae = np.mean(abs(y_test_predict[name] - yTe[farms[f]].reshape(len(yTe[farms[f]]),)
        t2 = time.time()
        # Print the results of the performance of each algorithm
        print ("Coefficient of Determination:", estimator.score(xTe[farms[f]], yTe[farms[f]
        print ("Root-Mean-Squared Error:", rmse)
        print ("Mean Absolute Error:", mae)
        print ("Time for each algorithm:", t2-t1)
        print()

    # Visualize the prediction results using MatplotLib
    ax = plt.subplot('%d%d%d' %(len(days),1,i+1))
    for name, estimator in Estimators.items():
        ax.plot(y_test_predict[name], label=name)

    ax.plot(yTe[farms[f]], label="Real Data", linestyle='--')
    ax.set_title('Day %d' % days[i], fontsize=15)
    ax.set_xlim(0,167)
    ax.set_ylim(0,1)
fig.suptitle('Prediction Result for Farm %s' % wf_idx[farms[f]], fontsize=30)
plt.xlabel('7 days since the requested day (hrs)', fontsize=20)
plt.ylabel('Power Generated', fontsize=20)
plt.legend(loc='center left', bbox_to_anchor=(0.9, 1),
```

```
                fancybox=True, shadow=True)
plt.show()
end_time = time.time()
print("Entire Program time: ", end_time - start_time)
```

The following is the Code "main2.py". It performs data processing and scenario generation and ends with graphical results for errors, coefficients of determination, and computing time for all 19 farms.

```
import os
import numpy as np
import math
import time
import matplotlib.pyplot as plt
from Input.read import readData
from Input.parameter_gen import parameter_gen
from Input.feature_build import feature_build

from sklearn import linear_model
from sklearn import svm
from sklearn.ensemble import RandomForestRegressor
from sklearn.ensemble import GradientBoostingRegressor
from sklearn import neural_network
from sklearn import kernel_ridge
from sklearn import neighbors

# WIND FORECAST CORRECTIVE SCENARIOS for 19 wind farms and ERRORS
# This script ends with graphical results for errors,
# coefficients of determination, and computing time for all 19 farms
#   ================================================================
#   generate wind scenarios based on historic data
#   considering spatial and temporal correlation
#   provide scenarios with better forecast
#   provide scenarios with uncertainty quantification
#   provide reasonable boundary with scenarios
#   combining multiple data mining techniques
#   including Random Forest, SVM, Linear Regression, KNN, NN
#   the data is based on the NREL Western Wind Dataset

# Load Data
start_time = time.time()
year = 2006
resolution = 1  # 1 hr resolution
speed = []
gen = []
# directory where the data is stored
dataDir = os.listdir('./' + str(year) + '/' + str(year))
```

```
# number of wind farms in the directory
nLocation = len(dataDir)

# list storing names of wind farms
wf_idx = []
for dirname in dataDir:
    wf_idx.append(dirname)

# list storing names of wind sites in each wind farm
wf_name = []
for dirname in dataDir:
    temp = []
    for filename in os.listdir('./' + str(year) + '/' + str(year) + '/' + dirname):
        temp.append(os.path.splitext(filename)[0])
    wf_name.append(temp)

wf_idx = np.array(wf_idx)
wf_name = np.array(wf_name)

# Output cleaned wind speed and power based on the given data
speed, gen, wind_param, capacity = readData(wf_name, wf_idx, resolution, year)
# Load Parameters
para = parameter_gen(gen, 5, resolution, 1, 0)

# Build Feature and Target
feature, target = feature_build(gen, speed, para)
print(np.shape(feature[0]))
print(np.shape(target[0]))

# Build Training and Test sets
days = [124, 221, 306] # testing days: can be manipulated
farms = [0, 3, 6] # testing farms: can be manipulated
# x-axis with all 19 farms
farm_axis = np.arange(nLocation)

# Scikit-Learn commands for multiple algorithms
Estimators = {
                      "Linear Regression": linear_model.LinearRegression(),
                      "Support Vector Machine": svm.LinearSVR(),
                      "Kernel Ridge": kernel_ridge.KernelRidge(),
                      "Random Forest": RandomForestRegressor(),
                      "Gradient Boosting": GradientBoostingRegressor(),
```

```
                    "Neural Network": neural_network.MLPRegressor(),
                    "Nearest Neighbor": neighbors.KNeighborsRegressor(),
}
rmse_avg = dict()
mae_avg = dict()
CoDet_avg = dict()
time_avg = dict()
for name, estimator in Estimators.items():
    rmse_avg[name] = np.empty((len(days),len(farm_axis)))
    mae_avg[name] = np.empty((len(days),len(farm_axis)))
    CoDet_avg[name] = np.empty((len(days),len(farm_axis)))
    time_avg[name] = np.empty((len(days),len(farm_axis)))


for i in range(len(days)):
    # prediction hours: 7 days
    test_hour = np.arange((days[i]-1) * 24, (days[i]+6) * 24) - para.drop_length
    test_time = np.transpose(test_hour)
    train_length = 2160  # length of training sets

    nFarm = nLocation
    xTr = []
    yTr = []
    xTe = []
    yTe = []

    # build training and testing sets here
    for iFarm in range(nFarm):
        xTr1 = feature[iFarm][test_time[0]-train_length : test_time[0]]
        yTr1 = target[iFarm][test_time[0]-train_length : test_time[0]]
        xTe1 = feature[iFarm][test_time[0]:test_time[len(test_time)-1]+1]
        yTe1 = target[iFarm][test_time[0]:test_time[len(test_time)-1]+1]

        xTr.append(xTr1)
        yTr.append(yTr1)
        xTe.append(xTe1)
        yTe.append(yTe1)

    xTr = np.array(xTr)
    yTr = np.array(yTr)
    xTe = np.array(xTe)
    yTe = np.array(yTe)
```

```
    # dictionary form to store prediction results
    y_test_predict = dict()
    for name, estimator in Estimators.items():
        temprmse = []
        tempmae = []
        tempcoef = []
        temptime = []
        for f in range(len(farm_axis)):
            t1 = time.time()  # for computing time
            # fit the training sets
            estimator.fit(xTr[f], yTr[f].reshape(len(yTr[f]),))
            # predict using each algorithm
            y_test_predict[name] = estimator.predict(xTe[f])

            # the wind power should be in the range of 0 to 1, so outliers should be taken care
            for h in range(len(y_test_predict[name])):
                if (y_test_predict[name][h] < 0):
                    y_test_predict[name][h] = 0
                elif (y_test_predict[name][h] > 1):
                    y_test_predict[name][h] = 1

            # root mean squared error
            rmse = math.sqrt(np.mean((y_test_predict[name] - yTe[f].reshape(len(yTe[f]),))**2))
            # mean absolute error
            mae = np.mean(abs(y_test_predict[name] - yTe[f].reshape(len(yTe[f]),)))
            t2 = time.time()
            temprmse.append(rmse)
            tempmae.append(mae)
            tempcoef.append(estimator.score(xTe[f], yTe[f].reshape(len(yTe[f]),)))
            temptime.append(t2-t1)
        rmse_avg[name][i] = np.array(temprmse)
        mae_avg[name][i] = np.array(tempmae)
        CoDet_avg[name][i] = np.array(tempcoef)
        time_avg[name][i] = np.array(temptime)

# Visualize the error results for 19 farms using MatplotLib
# Root Mean Squared Error
fig1 = plt.figure()
plt.title("RMSE", fontsize=30)
plt.xlabel("Farms", fontsize=20)
plt.ylabel("Values", fontsize=20)
```

```
for name, estimator in Estimators.items():
    plt.plot(farm_axis, np.mean(rmse_avg[name], axis=0), label=name)
plt.legend(loc='center left', bbox_to_anchor=(0.95, 0.3),
           fancybox=True, shadow=True)

# Mean Absolute Error
fig2 = plt.figure()
plt.title("MAE", fontsize=30)
plt.xlabel("Farms", fontsize=20)
plt.ylabel("Values", fontsize=20)
for name, estimator in Estimators.items():
    plt.plot(farm_axis, np.mean(mae_avg[name], axis=0), label=name)
plt.legend(loc='center left', bbox_to_anchor=(0.95, 0.3),
           fancybox=True, shadow=True)

# Coefficient of Determination
fig3 = plt.figure()
plt.title("Coefficient of Determination", fontsize=30)
plt.xlabel("Farms", fontsize=20)
plt.ylabel("Values", fontsize=20)
for name, estimator in Estimators.items():
    plt.plot(farm_axis, np.mean(CoDet_avg[name], axis=0), label=name)
plt.legend(loc='center left', bbox_to_anchor=(0.95, 0.3),
           fancybox=True, shadow=True)

# Computing Time
fig4 = plt.figure()
plt.title("Computing Time", fontsize=30)
plt.xlabel("Farms", fontsize=20)
plt.ylabel("Values", fontsize=20)
for name, estimator in Estimators.items():
    plt.plot(farm_axis, np.mean(time_avg[name], axis=0), label=name)
plt.legend(loc='center left', bbox_to_anchor=(0.95, 0.3),
           fancybox=True, shadow=True)
plt.show()
end_time = time.time()
print("Entire Program time: ", end_time - start_time)
```

The following is the Code "main3.py". It performs data processing and scenario generation and ends with graphical prediction results for three testing days and three testing farms but with different time leads. It also provides errors and coefficients of determination for different time leads.

```
import os
import numpy as np
import math
import time
import matplotlib.pyplot as plt
from Input.read import readData
from Input.parameter_gen import parameter_gen
from Input.feature_build import feature_build

from sklearn import linear_model
from sklearn import svm
from sklearn.ensemble import RandomForestRegressor
from sklearn.ensemble import GradientBoostingRegressor
from sklearn import neural_network
from sklearn import kernel_ridge
from sklearn import neighbors


# WIND FORECAST CORRECTIVE SCENARIOS for 19 wind farms with VARIATION in time_lead
# This script ends with graphical prediction results for three testing days and
# three testing farms but with different time_leads. It also provides
# errors and coefficients of determination for different time_leads.
#   ================================================================
#   generate wind scenarios based on historic data
#   considering spatial and temporal correlation
#   provide scenarios with better forecast
#   provide scenarios with uncertainty quantification
#   provide reasonable boundary with scenarios
#   combining multiple data mining techniques
#   including Random Forest, SVM, Linear Regression, KNN, NN
#   the data is based on the NREL Western Wind Dataset

# Load Data
start_time = time.time()
year = 2006
resolution = 1  # 1 hr resolution
speed = []
gen = []
```

```
# directory where the data is stored
dataDir = os.listdir('./' + str(year) + '/' + str(year))
# number of wind farms in the directory
nLocation = len(dataDir)

# list storing names of wind farms
wf_idx = []
for dirname in dataDir:
    wf_idx.append(dirname)

# list storing names of wind sites in each wind farm
wf_name = []
for dirname in dataDir:
    temp = []
    for filename in os.listdir('./' + str(year) + '/' + str(year) + '/' + dirname):
        temp.append(os.path.splitext(filename)[0])
    wf_name.append(temp)

wf_idx = np.array(wf_idx)
wf_name = np.array(wf_name)

# Output cleaned wind speed and power based on the given data
speed, gen, wind_param, capacity = readData(wf_name, wf_idx, resolution, year)

# varied Time Leads
t_lead = [1, 2, 4, 6, 12, 24]
rmse_avg = dict()
mae_avg = dict()
CoDet_avg = dict()
days = [124, 221, 306] # testing days: can be manipulated
farms = [0, 3, 6] # testing farms: can be manipulated
# x-axis with 10 farms
farm_axis = np.arange(10)

# Scikit-Learn commands for multiple algorithms
Estimators = {
                          "Linear Regression": linear_model.LinearRegression(),
                          "Support Vector Machine": svm.LinearSVR(),
                          "Kernel Ridge": kernel_ridge.KernelRidge(),
                          "Random Forest": RandomForestRegressor(),
                          "Gradient Boosting": GradientBoostingRegressor(),
                          "Neural Network": neural_network.MLPRegressor(),
```

```
                        "Nearest Neighbor": neighbors.KNeighborsRegressor(),
}
for name, estimator in Estimators.items():
    rmse_avg[name] = np.empty((len(t_lead), len(days)))
    mae_avg[name] = np.empty((len(t_lead), len(days)))
    CoDet_avg[name] = np.empty((len(t_lead), len(days)))
for t in range(len(t_lead)):
# Load Parameters
    para = parameter_gen(gen, 5, resolution, t_lead[t], 0)

    # Build Feature and Target
    feature, target = feature_build(gen, speed, para)

    # Build Training, Validation, and Test sets
    for i in range(len(days)):
        # prediction hours: 7 days
        test_hour = np.arange((days[i]-1) * 24, (days[i]+6) * 24) - para.drop_length
        test_time = np.transpose(test_hour)
        train_length = 2160 # length of training sets

        nFarm = nLocation
        xTr = []
        yTr = []
        xTe = []
        yTe = []

        # build training and testing sets here
        for iFarm in range(nFarm):
            xTr1 = feature[iFarm][test_time[0]-train_length : test_time[0]]
            yTr1 = target[iFarm][test_time[0]-train_length : test_time[0]]
            xTe1 = feature[iFarm][test_time[0]:test_time[len(test_time)-1]+1]
            yTe1 = target[iFarm][test_time[0]:test_time[len(test_time)-1]+1]

            xTr.append(xTr1)
            yTr.append(yTr1)
            xTe.append(xTe1)
            yTe.append(yTe1)

        xTr = np.array(xTr)
        yTr = np.array(yTr)
        xTe = np.array(xTe)
        yTe = np.array(yTe)
```

```
        # dictionary form to store prediction results
        y_test_predict = dict()
        for name, estimator in Estimators.items():
            temprmse = []
            tempmae = []
            tempcoef = []
            temptime = []
            for f in range(len(farm_axis)):
                # fit the training sets
                estimator.fit(xTr[f], yTr[f].reshape(len(yTr[f]),))
                # predict using each algorithm
                y_test_predict[name] = estimator.predict(xTe[f])

                # the wind power should be in the range of 0 to 1, so outliers should be taken
                for h in range(len(y_test_predict[name])):
                    if (y_test_predict[name][h] < 0):
                        y_test_predict[name][h] = 0
                    elif (y_test_predict[name][h] > 1):
                        y_test_predict[name][h] = 1

                # root mean squared error
                rmse = math.sqrt(np.mean((y_test_predict[name] - yTe[f].reshape(len(yTe[f]),))
                # mean absolute error
                mae = np.mean(abs(y_test_predict[name] - yTe[f].reshape(len(yTe[f]),)))
                temprmse.append(rmse)
                tempmae.append(mae)
                tempcoef.append(estimator.score(xTe[f], yTe[f].reshape(len(yTe[f]),)))
            rmse_avg[name][t][i] = np.mean(temprmse)
            mae_avg[name][t][i] = np.mean(tempmae)
            CoDet_avg[name][t][i] = np.mean(tempcoef)

# Visualize the error results for different time leads using MatplotLib
fig1 = plt.figure()
plt.title("RMSE", fontsize=30)
plt.xlabel("T_Leads", fontsize=20)
plt.ylabel("Values", fontsize=20)
for name, estimator in Estimators.items():
    plt.plot(t_lead, np.mean(rmse_avg[name], axis=1), label=name)
plt.legend(loc='center left', bbox_to_anchor=(0.95, 0.3),
                fancybox=True, shadow=True)
plt.xlim(1,24)
```

```
fig2 = plt.figure()
plt.title("MAE", fontsize=30)
plt.xlabel("T_Leads", fontsize=20)
plt.ylabel("Values", fontsize=20)
for name, estimator in Estimators.items():
    plt.plot(t_lead, np.mean(mae_avg[name], axis=1), label=name)
plt.legend(loc='center left', bbox_to_anchor=(0.95, 0.3),
                 fancybox=True, shadow=True)
plt.xlim(1,24)
fig3 = plt.figure()
plt.title("Coefficient of Determination", fontsize=30)
plt.xlabel("T_Leads", fontsize=20)
plt.ylabel("Values", fontsize=20)
for name, estimator in Estimators.items():
    plt.plot(t_lead, np.mean(CoDet_avg[name], axis=1), label=name)
plt.legend(loc='center left', bbox_to_anchor=(0.95, 0.3),
                 fancybox=True, shadow=True)
plt.xlim(1,24)

plt.show()
end_time = time.time()
print("Entire Program time: ", end_time - start_time)
```