**Data Mining and Analysis Project**

**on**

# Temperature Prediction

**A Project Report submitted in partial fulfilment of the requirements for the award of**

## Bachelor of Engineering

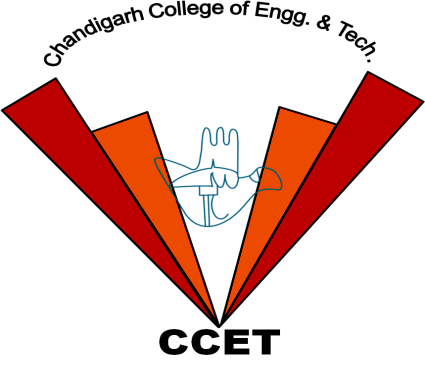
**IN COMPUTER SCIENCE AND ENGINEERING**

**Submitted by :**

**CO18345(Saurabh Gudwal)**

**CO18351(Tamanna)**

### Under the supervision of Dr.Varun Gupta



### CHANDIGARH COLLEGE OF ENGINEERING AND TECHNOLOGY

**(DEGREE WING)**

Government Institute under Chandigarh (UT) Administration, Affiliated to Panjab University , Chandigarh Sector-26, Chandigarh. PIN-160019

**INTRODUCTION:**

**Temperature Prediction using Multiple Linear Regression**

Machine learning is an application of artificial intelligence (AI) that provides systems the ability to automatically learn and improve from experience without being explicitly programmed.

Machine learning focuses on the development of computer programs that can access data and use it learn for themselves.

We have used regression to predict the temperature of a room.

In statistics, linear regression is a linear approach to modeling the relationship between a scalar response (or dependent variable) and one or more explanatory variables (or independent variables). The case of one explanatory variable is called simple linear regression. For more than one explanatory variable, the process is called multiple linear regression.

So, linear regression is a mechanism in which we build a linear model which uses one or more than one input parameters(x1, x2, x3..) and uses them to determine an output(y). In mathematical terms, y is a linear combination of the input parameters x1, x2, x3 ..The form of the model mathematically is:

y = w0 +w1.x1 + w2.x2 + w3.x3 + ….

So, its essentially taking into consideration the effect of each parameter on the output and the effect of each parameter is determined by a coefficient/weight.

Physicists define climate as a “complex system”. While there are a lot of interpretations about it, in this specific case we can consider “complex” to be “unsolvable in analytical ways”.

This may seems discouraging, but it actually paves the way to a wide range of numerical algorithms that aim to solve the climate challenges. With the computational developments of the last years, Machine Learning algorithms are certainly part of them.

**METHODOLOGY:**

* We are using regression to predict temperature of the area.
* The geographically weighed regression (GWR) method is used to realize the simulation of surface temperature based on the current date. The deep learning prediction network based on convolution and long short‐term memory (LSTM) networks was constructed to predict the spatial distribution of surface temperature on the next observation date.

**Temperature Prediction in Python**

Forecasting Temperature is the use of science and technology to predict the atmospheric conditions of a given space and time. Humans have been trying to predict the weather informally for thousands of years since the 19th century. Weather forecasts are made by collecting details of the current state of the atmosphere, land, and sea and using weather patterns to predict how the atmosphere will change in a particular area.

When calculated manually based on changes in barometric pressure, current weather, and weather or cloud cover, weather forecasting is now based on computer-based models that take into account many celestial objects. Individual input is still required to select the best weather model on which to predict, which includes pattern recognition skills, telephone communication, model performance information, and model bias information. Accuracy of forecasting is due to the hot air condition, the great calculation power required to solve calculations that describe the atmosphere, land and sea, the error involved in estimating the initial conditions, and incomplete understanding of wind and related processes. Therefore, the predictions are less accurate as the difference between the current time and the time the forecast is made (the range of the forecast) increases. The use of ensembles and harmonization model helps reduce error and provides a level of confidence in the forecast.

There are many different ways to use the end of the weather. Weather warnings are important predictions because they are used to protect health and property. Forecasts based on temperature and rainfall are important for agriculture, so for traders in the middle of the commodity markets. Temperature forecasts are used by state-owned companies to measure demand in the coming days. Every day, many use the weather forecast to determine what to wear on a particular day. Since outdoor activities are severely limited by heavy rain, snow, and cool air, forecasts can be used to plan activities around these events, and to plan ahead and survive.

Climate forecasting is part of the economy, for example, in 2009, the US spent about $ 5.1 billion on weather forecasting, generating six times as much profit.

**Libraries**

The libraries used are the most popular in data analysis, architecture and mathematical operations (pandas, matplotlib, numpy). There are some of them to visualize advanced data.

from \_\_\_\_ import partition

import csv, sys, re, time, stats

from input dasetsets within sklearn, linear\_model, pre-processing, neural\_network

from sklearn.utils to import column\_or\_1d

from data import time, timedelta, date

from dateutil.relativedelta import relativedelta

import os

import errno

import matplotlib.pyplot as plt

import numpy as np

import pandas as pd

from sklearn.svm to import SVR

from matplotlib import dates as mPlotDATEs

**Editing:**

umathop

**Algorithm:**

Multi-Line Repetition

**Source Code**

from \_\_future\_\_ import division

import csv, sys, re, timeit, math

from sklearn import datasets, linear\_model, preprocessing, neural\_network

from sklearn.utils import column\_or\_1d

from datetime import datetime, timedelta, date

from dateutil.relativedelta import relativedelta

import os

import errno

import matplotlib.pyplot as plt

import numpy as np

import pandas as pd

from sklearn.svm import SVR

from matplotlib import dates as mPlotDATEs

try:

int(s)

return True

except ValueError:

return False

def RepresentsFloat(s):

try:

float(s)

return True

except ValueError:

return False

elif (option == 'hour'):

time\_diff = time\_obj - data time (time\_obj.year, time\_obj.month, time\_obj.day, time\_obj.hour, 0)

other:

expand SystemError ("option is not a valid thread!")

return int (time\_diff.total\_seconds () / 60)

# do the translation

def interpolate\_df (df, features):

df\_re = df

print ("len (df.index) = {}". format (len (df.index)))

# check all data float data and convert the data type to float64

for col in features:

# df [col] = df [col] .astype (float)

temp = df [df [col] .isnull ()]

#print (test title)

Print ("===")

# print (test title (n = 1))

print ("{} type {}". format (col, df [col] .dtype))

print ("{} type contains {} np.NaN". format (col, len (temp.index)))

Print ("===")

print ("len (df.index) = {}". format (len (df.index)))

# can be time to use as a reference and set method = 'time'

# df.to\_csv ("df\_before\_interpolate.csv")

# df [features] = df [features] .interpolate (method = 'time')

# df.loc [:, features] = df [features] .interpolate (method = 'time')

# somehow, df (input) will be updated or used inplace = False

df\_re.loc [:, features] = df [features] .interpolate (method = 'time', inplace = False)

# df.to\_csv ("df\_after\_interpolate.csv")

#print ("df =")

#print (df)

# capture real nan values

df\_nan\_interpolate = df.loc [df\_nan.index.values]

print ("len (df\_nan\_interpolate.index) = {}". format (len (df\_nan\_interpolate.index)))

df\_nan\_interpolate.to\_csv ("df\_nan\_interpolate.csv")

if (df\_re.notnull (). all (axis = 1) .all (axis = 0)):

print ("CHECK: No value in df\_re.")

replace df\_re

df\_train = df.loc [(df.index> data\_start) & (df.index <= data\_end) ,:]]

# do interpolate on training set only

df\_train = interpolate\_df (df\_train, features)

df\_train.to\_csv ('df\_train\_clean.csv')

X\_train = df\_train [features]

y\_train = df\_train [target]

# configure test data

data\_start = time (data\_test\_yr\_start, 1, 1, 0, 0, 0)

data\_end = time (data\_test\_yr\_end, 12, 31, 23, 59, 59)

df\_test = df.loc [(df.index> data\_start) & (df.index <= data\_end) ,:]]

# drops the number lines of the NaN test set

(irowu\_old, col\_old) = df\_test.shape

print ("Before you draw the NaN number of the test set, df\_test.shape = {}". format (df\_test.shape))

df\_test = df\_test [df\_test.notnull (). all (axis = 1)]

(row, column) = df\_test.shape

print ("After dragging the NaN number of the test set, df\_test.shape = {}". format (df\_test.shape))

print ("Drop Level = {0: .2f}". format (float (1 - (line / line\_knee))))

df\_test.to\_csv ('df\_test\_clean.csv')

X\_test = df\_test [features]

y\_test = df\_test [stones]

# familiarity and training scale / test set

# use robust\_scaler to protect misleading merchants

# scaler = processing StandardScaler ()

# use robust\_scaler to protect misleading merchants

scale = advancement.RebustScaler ()

X\_train = scaler.fit\_transform (X\_train)

X\_test = scaler.transform (X\_test)

return (X\_train, y\_train, X\_test, y\_test)

def normalization (df\_train, df\_test, targets, features):

# use robust\_scaler to protect misleading merchants

scale = advancement.RebustScaler ()

X\_train = scaler.fit\_transform (X\_train)

X\_test = scaler.transform (X\_test)

return (X\_train, y\_train, X\_test, y\_test)

# structure y\_test

def plot\_y\_test (regr, X\_test, y\_test, ask\_user):

(r\_test, c\_test) = X\_test.shape

# for i in range (c\_test):

# plt. scatter (X\_test [:, i], y\_test)

# plt.plot (X\_test [:, i], regr.predict (X\_test), color = 'blue', linewidth = 3)

y\_predict = regr.predict (X\_test)

#print ("==> y\_test type = {}". format (type (y\_test)))

#print ("y\_test.index = {}". format (y\_test.index))

#print ("y\_test = {}". format (y\_test))

#print ("y\_predict = {}". format (y\_predict))

df\_plot = y\_test

#print (df\_plot)

#print ("DATE")

#print ("# # # # # # # # # # # # # # # # # # #

df\_plot = df\_plot.reset\_index (level = ['DATE'])

df\_plot.loc [:, 'predict\_temp\_C'] = y\_cognize

# go back green DATE time 1day\_later

df\_plot.loc [:, "raw\_DATE"] = df\_plot ['DATE']. apply (lambda time\_obj: time\_obj + relativedelta (days = 1))

df\_plot.rename (columns = {'1days\_later\_temp\_C': 'raw\_temp\_C', 'DATE': 'label\_DATE'}, inplace = True)

df\_plot = df\_plot.set\_index ("green\_DATE")

#print (df\_plot)

#print ("# # # # # # # # # # # # # # # # # #

# the time range for the default layout

icebo\_yr = 2016

structure\_month = 10

icebo\_day = 5

duration = 10

df\_test.to\_csv('df\_test\_clean.csv')

X\_test = df\_test[features]

y\_test = df\_test[targets]

# normalization and scale for training/test set

# use robust\_scaler to avoid misleading outliers

# scaler = preprocessing.StandardScaler()

# use robust\_scaler to avoid misleading outliers

scaler = preprocessing.RobustScaler()

X\_train = scaler.fit\_transform(X\_train)

X\_test = scaler.transform(X\_test)

return (X\_train, y\_train, X\_test, y\_test)

def normalization(df\_train, df\_test, targets, features):

# use robust\_scaler to avoid misleading outliers

scaler = preprocessing.RobustScaler()

X\_train = scaler.fit\_transform(X\_train)

X\_test = scaler.transform(X\_test)

return (X\_train, y\_train, X\_test, y\_test)

# plot y\_test

def plot\_y\_test(regr, X\_test, y\_test, ask\_user):

(r\_test, c\_test) = X\_test.shape

# for i in range(c\_test):

# plt.scatter(X\_test[:, i], y\_test)

# plt.plot(X\_test[:, i], regr.predict(X\_test), color='blue', linewidth=3)

y\_predict = regr.predict(X\_test)

# print("==> y\_test type = {}".format(type(y\_test)) )

# print("y\_test.index = {}".format(y\_test.index))

# print("y\_test = {}".format(y\_test) )

# print("y\_predict = {}".format(y\_predict) )

df\_plot = y\_test

# print(df\_plot)

# print("DATE")

# print("#########################")

df\_plot = df\_plot.reset\_index(level=['DATE'])

df\_plot.loc[:,'predict\_temp\_C'] = y\_predict

# shift back to raw DATE time 1day\_later

df\_plot.loc[:,"raw\_DATE"] = df\_plot['DATE'].apply(lambda time\_obj: time\_obj + relativedelta(days=1))

df\_plot.rename(columns={'1days\_later\_temp\_C': 'raw\_temp\_C', 'DATE':'label\_DATE'}, inplace=True)

df\_plot = df\_plot.set\_index("raw\_DATE")

# print(df\_plot)

# print("#########################")

# default plot time range

plot\_yr = 2016

plot\_month = 10

plot\_day = 5

duration = 10

range\_start = datetime(plot\_yr, plot\_month, plot\_day, 0, 0, 0)

range\_end = datetime(plot\_yr, plot\_month, plot\_day, 0, 0, 0) + relativedelta(days=duration)

if (range\_start < datetime(2016,1,2,0,0,0) or range\_end > datetime(2017,1,1,0,0,0) ):

raise SystemExit("Input date is out of range! Please try again!")

else:

print("Correct format and time range!")

if (ask\_user == True):

)

print("years, month, day, ploting duration(days) \n")

print("For example, enter: {}, {}, {}, {}".format(plot\_yr, plot\_month, plot\_day, duration) )

input\_format\_ok = False

while(input\_format\_ok == False):

user\_input = input()

print("Your input is {}".format(user\_input) )

try:

plot\_yr = int(user\_input[0])

plot\_month = int(user\_input[1])

plot\_day = int(user\_input[2])

duration = int(user\_input[3])

range\_start = datetime(plot\_yr, plot\_month, plot\_day, 0, 0, 0)

range\_end = datetime(plot\_yr, plot\_month, plot\_day, 0, 0, 0) + relativedelta(days=duration)

if (range\_start < datetime(2016,1,2,0,0,0) or range\_end > datetime(2017,1,1,0,0,0) ):

print("Input date is out of range! Please try again!")

else:

print("Correct format and time range!")

input\_format\_ok = True

except:

print("Incorrect format, please try again!")

df\_plot = df\_plot[range\_start.strftime('%Y-%m-%d %H:%M:%S') : range\_end.strftime('%Y-%m-%d %H:%M:%S')]

DateFormatter('%Y-%m-%d %H:%M:%S')

ax.xaxis.set\_major\_formatter(xfmt)

ax.xaxis\_date()

# plt.scatter(y\_test.index, y\_test)

# plt.plot(y\_test.index, y\_predict, color='blue', linewidth=3)

# plt.scatter(y\_test.index[0:25], y\_test[0:25])

# plt.plot(y\_test.index[0:25], y\_test[0:25], color='red', linewidth=3)

# plt.plot(y\_test.index[0:25], y\_predict[0:25], color='blue', linewidth=3)

# plt.subplot(121)

plt.xlabel("time range")

plt.ylabel("degree C")

plt.title("raw data (red) v.s. predict data (blue)")

plt.grid()

plt.plot(datenums, value\_raw, linestyle='-', marker='o', markersize=5, color='r', linewidth=2, label="raw temp C")

plt.plot(datenums, value\_predict, linestyle='-', marker='o', markersize=5, color='b', linewidth=2, label="predict temp C")

plt.legend(loc="best")

plt.show()

plt.figure()

# plt.subplot(122)

plt.xlabel("raw data (degree C)")

plt.ylabel("predict data (degree C)")

plt.title("perfect match (red) v.s. model (blue)")

plt.grid()

plt.plot(value\_raw, value\_raw, linestyle='--', marker='o', markersize=5, color='r', linewidth=1, label="perfect match line")

plt.scatter(value\_raw, value\_predict, marker='o', s=10, color='b', label="predict temp C")

# plt.plot(value\_predict, marker='o', markersize=3, color='b', label="predict temp C")

plt.legend(loc="best")

plt.show()

# poly\_degree = int, interaction\_only = True

def linear\_regr(X\_train, y\_train, X\_test, y\_test, poly\_degree, interaction\_only, print\_coef, plot, ask\_user, model\_result):

# create more features

poly = preprocessing.PolynomialFeatures(poly\_degree, interaction\_only=interaction\_only)

X\_train = poly.fit\_transform(X\_train)

X\_test = poly.fit\_transform(X\_test)

(s\_n, f\_n) = X\_train.shape

# l\_n = int(math.ceil(1.5\*f\_n))

l\_n = int(math.ceil(1.2\*f\_n))

## model\_name = "SGDRegressor"

## model\_rt\_start = timeit.default\_timer()

## regr = linear\_model.SGDRegressor(penalty='elasticnet', alpha=0.01, l1\_ratio=0.25, fit\_intercept=True)

## model\_rt\_stop = timeit.default\_timer()

## model\_runtime = model\_rt\_stop - model\_rt\_start

## # test score: 0.83

## model\_name = "ElasticNet"

## model\_rt\_start = timeit.default\_timer()

## regr = linear\_model.ElasticNet(alpha = 0.01)

## model\_rt\_stop = timeit.default\_timer()

## model\_runtime = model\_rt\_stop - model\_rt\_start

if (model == 0):

# test score: 0.84

alpha = 0

model\_name = "linear\_model.LinearRegression"

regr = linear\_model.LinearRegression(copy\_X=True, fit\_intercept=True, n\_jobs=1, normalize=False)

model\_rt\_start = timeit.default\_timer()

regr.fit(X\_train, column\_or\_1d(y\_train) )

model\_rt\_stop = timeit.default\_timer()

model\_runtime = model\_rt\_stop - model\_rt\_start

model\_result = evaluation(X\_train, y\_train, X\_test, y\_test, poly\_degree, interaction\_only, print\_coef, plot, ask\_user,

model\_result, model\_name, model\_runtime, regr, alpha)

elif (model == 1):

for alpha in [0.0001, 0.001, 0.01, 0.1, 1, 3, 10]:

# test score: 0.83

model\_name = "linear\_model.Lasso"

regr\_lasso = linear\_model.Lasso(alpha = alpha)

model\_rt\_start = timeit.default\_timer()

regr\_lasso.fit(X\_train, column\_or\_1d(y\_train) )

model\_rt\_stop = timeit.default\_timer()

model\_runtime = model\_rt\_stop - model\_rt\_start

model\_result = evaluation(X\_train, y\_train, X\_test, y\_test, poly\_degree, interaction\_only, print\_coef, plot, ask\_user,

model\_result, model\_name, model\_runtime, regr\_lasso, alpha)

elif (model == 2):

for alpha in [0.0001, 0.001, 0.01, 0.1, 1, 3, 10]:

# for alpha in [0.0000001, 0.00001, 0.001, 0.01, 0.1, 1, 3, 10, 30, 100, 300, 10\*\*3, 10\*\*4, 10\*\*5]:

# test score: 0.84

model\_name = "linear\_model.Ridge"

regr\_ridge = linear\_model.Ridge(alpha = alpha)

model\_rt\_start = timeit.default\_timer()

regr\_ridge.fit(X\_train, column\_or\_1d(y\_train) )

model\_rt\_stop = timeit.default\_timer()

model\_runtime = model\_rt\_stop - model\_rt\_start

model\_result = evaluation(X\_train, y\_train, X\_test, y\_test, poly\_degree, interaction\_only, print\_coef, plot, ask\_user,

model\_result, model\_name, model\_runtime, regr\_ridge, alpha)

\_rt\_stop - model\_rt\_start

model\_result = evaluation(X\_train, y\_train, X\_test, y\_test, poly\_degree, interaction\_only, print\_coef, plot, ask\_user,

model\_result, model\_name, model\_runtime, regr, alpha)

elif (model == 4):

if (poly\_degree <= 3):

for alpha in [1, 10, 1000]:

# for alpha in [0.00001]:

# for layer\_n in [3, 7, 11]:

for layer\_n in [7, 11]:

# for layer\_n in [3]:

# test score: 0.83, runtime longer

model\_name = "neural\_network.MLPRegressor, layer = " + str(layer\_n)

if(layer\_n == 3):

regr = neural\_network.MLPRegressor(random\_state=True,hidden\_layer\_sizes=(l\_n,l\_n,l\_n),alpha=alpha)

if(layer\_n == 7):

# regr = neural\_network.MLPRegressor(random\_state=True,hidden\_layer\_sizes=(l\_n,l\_n,l\_n,l\_n,l\_n,l\_n,l\_n),alpha=alpha)

regr = neural\_network.MLPRegressor(random\_state=True,hidden\_layer\_sizes=(l\_n,l\_n,l\_n,l\_n,l\_n,l\_n,l\_n),alpha=alpha, learning\_rate='invscaling')

if(layer\_n == 11):

regr = neural\_network.MLPRegressor(random\_state=True,hidden\_layer\_sizes=(l\_n,l\_n,l\_n,l\_n,l\_n,l\_n,l\_n,l\_n,l\_n,l\_n,l\_n),alpha=alpha)

model\_rt\_start = timeit.default\_timer()

regr.fit(X\_train, column\_or\_1d(y\_train) )

model\_rt\_stop = timeit.default\_timer()

model\_runtime = model\_rt\_stop - model\_rt\_start

model\_result = evaluation(X\_train, y\_train, X\_test, y\_test, poly\_degree, interaction\_only, print\_coef, plot, ask\_user,

model\_result, model\_name, model\_runtime, regr, alpha)

else:

raise SystemExit("Model selection out of range!!!")

return model\_result

print("Coefficients: {}\n", regr.coefs\_)

with open("logs/log\_" + log\_timestr +".txt", "a") as logfile:

logfile.write("Coefficients: {}\n".format(regr.coefs\_) )

lt\_timer()

predict\_test = regr.predict(X\_test)

model\_rt\_predict\_test\_stop = timeit.default\_timer()

model\_runtime\_predict\_test = model\_rt\_predict\_test\_stop - model\_rt\_predict\_test\_start

mse\_test = float(np.mean( (predict\_test - column\_or\_1d(y\_test) ) \*\* 2) )

score\_test = regr.score(X\_test, y\_test)

# The mean squared error

print("Mean squared error (test): {0:.3f} \n".format( mse\_test ) )

# Explained variance score: 1 is perfect prediction

print("Variance score (test): {0:.3f} \n".format( score\_test ) )

print("model\_runtime (predict test set) = {0:.3f} (seconds) \n".format(model\_runtime\_predict\_test))

with open("logs/log\_" + log\_timestr +".txt", "a") as logfile:

logfile.write("====================\n")

logfile.write("Features polynomial degree: {} \n".format( poly\_degree ) )

logfile.write("Model: {} \n".format( model\_name ) )

logfile.write("Alpha (Regularization strength): {} \n".format( alpha ) )

logfile.write("X\_train.shape = {} \n".format(X\_train.shape) )

logfile.write("y\_train.shape = {} \n".format(y\_train.shape) )

logfile.write("X\_test.shape = {} \n".format(X\_test.shape) )

logfile.write("y\_test.shape = {} \n".format(y\_test.shape) )

logfile.write("For training set: \n")

logfile.write("Mean squared error (train): {0:.3f} \n".format( mse\_train ) )

logfile.write("Variance score (train): {0:.3f} \n".format( score\_train ) )

logfile.write("For test set: \n")

logfile.write("Mean squared error (test): {0:.3f} \n".format( mse\_test ) )

logfile.write("Variance score (test): {0:.3f} \n".format( score\_test ) )

logfile.write("model\_runtime (training) = {0:.3f} (seconds) \n".format(model\_runtime))

logfile.write("model\_runtime (predict train set) = {0:.3f} (seconds) \n".format(model\_runtime\_predict\_train))

logfile.write("model\_runtime (predict test set) = {0:.3f} (seconds) \n".

# print shape

if (plot == True):

plot\_y\_test(regr, X\_test, y\_test, ask\_user)

return model\_result

# def run\_fit(postfix, df\_run, targets\_run, features\_run, poly\_d\_max, inter\_only, print\_coef, plot):

def run\_fit(postfix, df\_run\_train, df\_run\_test, targets\_run, features\_run, poly\_d\_max, inter\_only, print\_coef, plot, ask\_user):

text = "RUNNING... df" + postfix

print("{0:{fill}{align}16}".format(text, fill='=', align='^'))

(X\_train, y\_train, X\_test, y\_test) = (0, 0, 0, 0)

# (X\_train, y\_train, X\_test, y\_test) = data\_gen(df\_run, targets\_run, features\_run, 2006, 2015, 2016, 2016)

(X\_train, y\_train, X\_test, y\_test) = normalization(df\_run\_train, df\_run\_test, targets\_run, features\_run)

# data = []

# (data[0], data[1], data[2], data[3]) = data\_gen(df\_run, features\_run)

print("df{} X\_train.shape = {}".format(postfix, X\_train.shape))

print("df{} y\_train.shape = {}".format(postfix, y\_train.shape))

print("df{} X\_test.shape = {}".format(postfix, X\_test.shape))

print("df{} y\_test.shape = {}".format(postfix, y\_test.shape))

print("df\_run\_train target + features = {}".format(df\_run\_train.columns.values))

print("=====")

elif (option == 'hour'):

time\_diff = time\_obj - data time (time\_obj.year, time\_obj.month, time\_obj.day, time\_obj.hour, 0)

other:

expand SystemError ("option is not a valid thread!")

return int (time\_diff.total\_seconds () / 60)

# do the translation

def interpolate\_df (df, features):

df\_re = df

print ("len (df.index) = {}". format (len (df.index)))

# check all data float data and convert the data type to float64

for col in features:

# df [col] = df [col] .astype (float)

temp = df [df [col] .isnull ()]

#print (test title)

Print ("===")

# print (test title (n = 1))

print ("{} type {}". format (col, df [col] .dtype))

print ("{} type contains {} np.NaN". format (col, len (temp.index)))

Print ("===")

print ("len (df.index) = {}". format (len (df.index)))

# can be time to use as a reference and set method = 'time'

# df.to\_csv ("df\_before\_interpolate.csv")

# df [features] = df [features] .interpolate (method = 'time')

# df.loc [:, features] = df [features] .interpolate (method = 'time')

# somehow, df (input) will be updated or used inplace = False

df\_re.loc [:, features] = df [features] .interpolate (method = 'time', inplace = False)

# df.to\_csv ("df\_after\_interpolate.csv")

#print ("df =")

#print (df)

# capture real nan values

df\_nan\_interpolate = df.loc [df\_nan.index.values]

print ("len (df\_nan\_interpolate.index) = {}". format (len (df\_nan\_interpolate.index)))

df\_nan\_interpolate.to\_csv ("df\_nan\_interpolate.csv")

if (df\_re.notnull (). all (axis = 1) .all (axis = 0)):

print ("CHECK: No value in df\_re.")

replace df\_re

df\_train = df.loc [(df.index> data\_start) & (df.index <= data\_end) ,:]]

# do interpolate on training set only

df\_train = interpolate\_df (df\_train, features)

df\_train.to\_csv ('df\_train\_clean.csv')

X\_train = df\_train [features]

y\_train = df\_train [target]

# configure test data

data\_start = time (data\_test\_yr\_start, 1, 1, 0, 0, 0)

data\_end = time (data\_test\_yr\_end, 12, 31, 23, 59, 59)

df\_test = df.loc [(df.index> data\_start) & (df.index <= data\_end) ,:]]

# drops the number lines of the NaN test set

(irowu\_old, col\_old) = df\_test.shape

print ("Before you draw the NaN number of the test set, df\_test.shape = {}". format (df\_test.shape))

df\_test = df\_test [df\_test.notnull (). all (axis = 1)]

(row, column) = df\_test.shape

print ("After dragging the NaN number of the test set, df\_test.shape = {}". format (df\_test.shape))

print ("Drop Level = {0: .2f}". format (float (1 - (line / line\_knee))))

df\_test.to\_csv ('df\_test\_clean.csv')

X\_test = df\_test [features]

y\_test = df\_test [stones]

# familiarity and training scale / test set

# use robust\_scaler to protect misleading merchants

# scaler = processing StandardScaler ()

# use robust\_scaler to protect misleading merchants

scale = advancement.RebustScaler ()

X\_train = scaler.fit\_transform (X\_train)

X\_test = scaler.transform (X\_test)

return (X\_train, y\_train, X\_test, y\_test)

def normalization (df\_train, df\_test, targets, features):

# use robust\_scaler to protect misleading merchants

scale = advancement.RebustScaler ()

X\_train = scaler.fit\_transform (X\_train)

X\_test = scaler.transform (X\_test)

return (X\_train, y\_train, X\_test, y\_test)

# structure y\_test

def plot\_y\_test (regr, X\_test, y\_test, ask\_user):

(r\_test, c\_test) = X\_test.shape

# for i in range (c\_test):

# plt. scatter (X\_test [:, i], y\_test)

# plt.plot (X\_test [:, i], regr.predict (X\_test), color = 'blue', linewidth = 3)

y\_predict = regr.predict (X\_test)

#print ("==> y\_test type = {}". format (type (y\_test)))

#print ("y\_test.index = {}". format (y\_test.index))

#print ("y\_test = {}". format (y\_test))

#print ("y\_predict = {}". format (y\_predict))

df\_plot = y\_test

#print (df\_plot)

#print ("DATE")

#print ("# # # # # # # # # # # # # # # # # # #

df\_plot = df\_plot.reset\_index (level = ['DATE'])

df\_plot.loc [:, 'predict\_temp\_C'] = y\_cognize

# go back green DATE time 1day\_later

df\_plot.loc [:, "raw\_DATE"] = df\_plot ['DATE']. apply (lambda time\_obj: time\_obj + relativedelta (days = 1))

df\_plot.rename (columns = {'1days\_later\_temp\_C': 'raw\_temp\_C', 'DATE': 'label\_DATE'}, inplace = True)

df\_plot = df\_plot.set\_index ("green\_DATE")

#print (df\_plot)

#print ("# # # # # # # # # # # # # # # # # #

# the time range for the default layout

icebo\_yr = 2016

structure\_month = 10

icebo\_day = 5

duration = 10

model\_re = {}

# for poly\_d in range(1, poly\_d\_max+1):

for poly\_d in range(1, poly\_d\_max+1):

model\_re = linear\_regr(X\_train, y\_train, X\_test, y\_test, poly\_degree = poly\_d,

interaction\_only = inter\_only, print\_coef = print\_coef, plot = plot, ask\_user = ask\_user, model\_result = model\_re)

return model\_re

elif (option == 'hour'):

time\_diff = time\_obj - data time (time\_obj.year, time\_obj.month, time\_obj.day, time\_obj.hour, 0)

other:

expand SystemError ("option is not a valid thread!")

return int (time\_diff.total\_seconds () / 60)

# do the translation

def interpolate\_df (df, features):

df\_re = df

print ("len (df.index) = {}". format (len (df.index)))

# check all data float data and convert the data type to float64

for col in features:

# df [col] = df [col] .astype (float)

temp = df [df [col] .isnull ()]

#print (test title)

Print ("===")

# print (test title (n = 1))

print ("{} type {}". format (col, df [col] .dtype))

print ("{} type contains {} np.NaN". format (col, len (temp.index)))

Print ("===")

print ("len (df.index) = {}". format (len (df.index)))

# can be time to use as a reference and set method = 'time'

# df.to\_csv ("df\_before\_interpolate.csv")

# df [features] = df [features] .interpolate (method = 'time')

# df.loc [:, features] = df [features] .interpolate (method = 'time')

# somehow, df (input) will be updated or used inplace = False

df\_re.loc [:, features] = df [features] .interpolate (method = 'time', inplace = False)

# df.to\_csv ("df\_after\_interpolate.csv")

#print ("df =")

#print (df)

# capture real nan values

df\_nan\_interpolate = df.loc [df\_nan.index.values]

print ("len (df\_nan\_interpolate.index) = {}". format (len (df\_nan\_interpolate.index)))

df\_nan\_interpolate.to\_csv ("df\_nan\_interpolate.csv")

if (df\_re.notnull (). all (axis = 1) .all (axis = 0)):

print ("CHECK: No value in df\_re.")

replace df\_re

df\_train = df.loc [(df.index> data\_start) & (df.index <= data\_end) ,:]]

# do interpolate on training set only

df\_train = interpolate\_df (df\_train, features)

df\_train.to\_csv ('df\_train\_clean.csv')

X\_train = df\_train [features]

y\_train = df\_train [target]

# configure test data

data\_start = time (data\_test\_yr\_start, 1, 1, 0, 0, 0)

data\_end = time (data\_test\_yr\_end, 12, 31, 23, 59, 59)

df\_test = df.loc [(df.index> data\_start) & (df.index <= data\_end) ,:]]

# drops the number lines of the NaN test set

(irowu\_old, col\_old) = df\_test.shape

print ("Before you draw the NaN number of the test set, df\_test.shape = {}". format (df\_test.shape))

df\_test = df\_test [df\_test.notnull (). all (axis = 1)]

(row, column) = df\_test.shape

print ("After dragging the NaN number of the test set, df\_test.shape = {}". format (df\_test.shape))

print ("Drop Level = {0: .2f}". format (float (1 - (line / line\_knee))))

df\_test.to\_csv ('df\_test\_clean.csv')

X\_test = df\_test [features]

y\_test = df\_test [stones]

# familiarity and training scale / test set

# use robust\_scaler to protect misleading merchants

# scaler = processing StandardScaler ()

# use robust\_scaler to protect misleading merchants

scale = advancement.RebustScaler ()

X\_train = scaler.fit\_transform (X\_train)

X\_test = scaler.transform (X\_test)

return (X\_train, y\_train, X\_test, y\_test)

def normalization (df\_train, df\_test, targets, features):

# use robust\_scaler to protect misleading merchants

scale = advancement.RebustScaler ()

X\_train = scaler.fit\_transform (X\_train)

X\_test = scaler.transform (X\_test)

return (X\_train, y\_train, X\_test, y\_test)

# structure y\_test

def plot\_y\_test (regr, X\_test, y\_test, ask\_user):

(r\_test, c\_test) = X\_test.shape

# for i in range (c\_test):

# plt. scatter (X\_test [:, i], y\_test)

# plt.plot (X\_test [:, i], regr.predict (X\_test), color = 'blue', linewidth = 3)

y\_predict = regr.predict (X\_test)

#print ("==> y\_test type = {}". format (type (y\_test)))

#print ("y\_test.index = {}". format (y\_test.index))

#print ("y\_test = {}". format (y\_test))

#print ("y\_predict = {}". format (y\_predict))

df\_plot = y\_test

#print (df\_plot)

#print ("DATE")

#print ("# # # # # # # # # # # # # # # # # # #

df\_plot = df\_plot.reset\_index (level = ['DATE'])

df\_plot.loc [:, 'predict\_temp\_C'] = y\_cognize

# go back green DATE time 1day\_later

df\_plot.loc [:, "raw\_DATE"] = df\_plot ['DATE']. apply (lambda time\_obj: time\_obj + relativedelta (days = 1))

df\_plot.rename (columns = {'1days\_later\_temp\_C': 'raw\_temp\_C', 'DATE': 'label\_DATE'}, inplace = True)

df\_plot = df\_plot.set\_index ("green\_DATE")

#print (df\_plot)

#print ("# # # # # # # # # # # # # # # # # #

# the time range for the default layout

icebo\_yr = 2016

structure\_month = 10

icebo\_day = 5

duration = 10

t3 = t1

t3.loc[:, new\_target] = t2[new\_target]

# df\_time\_train = t3['2006':'2014']

range\_start = datetime(train\_yr\_start, 1, 1, 0, 0, 0)

range\_end = datetime(train\_yr\_start, 1, 1, 0, 0, 0) + relativedelta(years=train\_years)

df\_time\_train = t3[range\_start.strftime('%Y-%m-%d %H:%M:%S') : range\_end.strftime('%Y-%m-%d %H:%M:%S')]

print("df\_time\_train.shape = {}".format(df\_time\_train.shape))

df\_time\_train.loc[:, new\_target] = df\_time\_train[new\_target].interpolate(method='time')

# df\_time\_test = t3['2015']

range\_start = datetime(test\_yr\_start, s\_month, s\_day, 0, 0, 0)

range\_end = datetime(test\_yr\_start, s\_month, s\_day, 0, 0, 0) + relativedelta(days=365)

# range\_end = datetime(2017, 3, 05, 0, 0, 0)

# range\_end = datetime(2017, 3, 05, 0, 0, 0)

df\_time\_test = t3[range\_start.strftime('%Y-%m-%d %H:%M:%S') : range\_end.strftime('%Y-%m-%d %H:%M:%S')]

print("df\_time\_test.shape = {}".format(df\_time\_train.shape))

# drop NaN number rows of test set

(row\_old, col\_old) = df\_time\_test.shape

print("Before drop NaN number of test set, df\_time\_test.shape = {}".format(df\_time\_test.shape))

df\_time\_test = df\_time\_test[ df\_time\_test.notnull().all(axis=1) ]

(row, col) = df\_time\_test.shape

print("After drop NaN number of test set, df\_time\_test.shape = {}".format(df\_time\_test.shape))

print("Drop rate = {0:.2f} ".format(float(1 - (row/row\_old)) ) )

print("===================== \n")

print("### Experiment = {} \n".format( experiment ) )

print("new\_target = {} \n".format( new\_target ) )

print("new\_features = {} \n".format( new\_features ) )

with open("logs/log\_" + log\_timestr +".txt", "a") as logfile:

logfile.write("===================== \n")

logfile.write("### Experiment = {} \n".format( experiment ) )

logfile.write("new\_target = {} \n".format( new\_target ) )

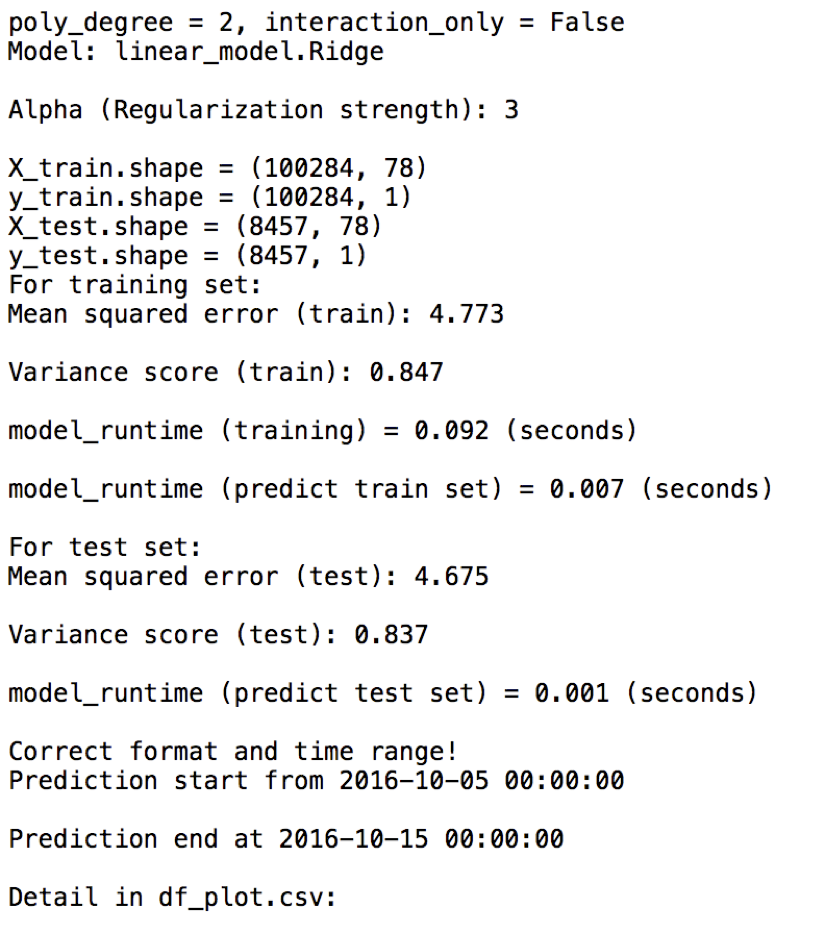
logfile.write("new\_features = {} \n".format( new\_features ) )

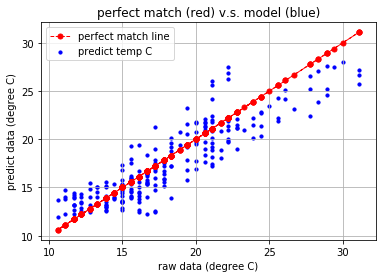
False, ask\_user=False)

print("### Experiment = {} \n".format( experiment ) )

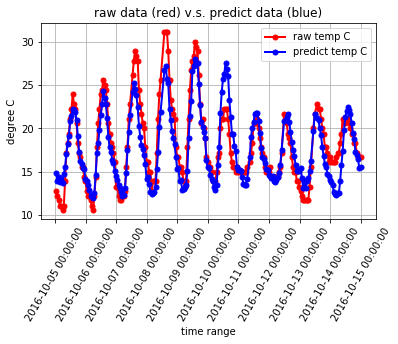
# print("model\_re = {}".format(model\_re))

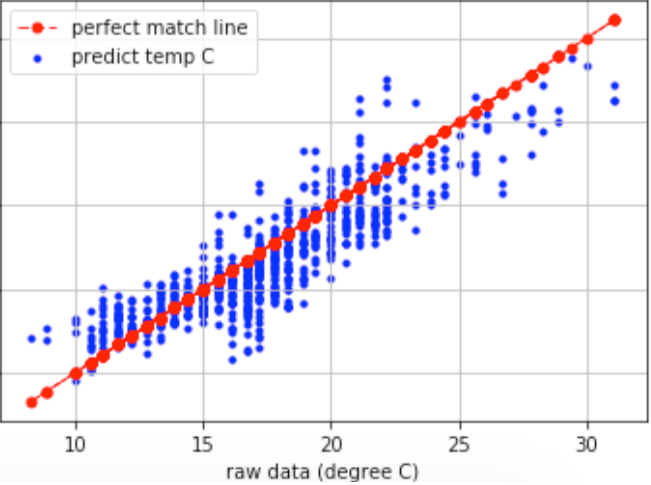
**Output:**





Old Data:





**Conclusions**

These methods are extremely easy to adopt as they don’t require any specific computational power.

Nonetheless, predictions perfectly fit in the error range designed by the dataset itself. It is important to consider that we only have examined monthly average values while it may be interesting to consider daily values too and have daily predictions.

**Refrences:**

<https://en.wikipedia.org/wiki/Machine_learning>

<https://www.hindawi.com/journals/scn/2018/1635081/>