# Data Science and Global Warming: simple machine learning techniques applied

Yassine BENAZZOU, Charles DOGNIN, Jean-Michel ROUFOSSE

December 23, 2016

**Abstract**

We use machine learning techniques to study the evolution of climate change in the world since 1800. The dataset comes from the Kaggle website[1], displaying the evolution since 1750 in degree Celsius of the Land Average Temperature, the Land Average Temperature Uncertainty (the 95% confidence interval around the mean), the Land Max Temperature, the Land Max Temperature Uncertainty, the Land Min Temperature,the Land Min Temperature Uncertainty, the Land And Ocean Average Temperature, and finally the Land And Ocean Average Temperature Uncertainty. The data gathered by Kaggle come from a far-reaching study, The Berkeley Earth Surface Temperature Study which "combines 1.6 billion temperature reports from 16 pre-existing archives"[2]. The second set of data comes from the the US Government's Earth System Research Laboratory, Global Monitoring Division and register the level of CO2 globally on average since 1958.

[1]https://www.kaggle.com/berkeleyearth/climate-change-earth-surface-temperature-
[2]http://berkeleyearth.org/data/

# Contents

**1 Introduction: Why climate change? 3**

**2 Data description and preparation 3**

**3 Exploratory Data Analysis 5**

**4 Predictive Modelling 10**

4.1 Linear Regression . . . 10
4.2 Polynomial Regression . . . 13
4.3 Correlation Matrix . . . 16
4.4 Random Forest . . . 17
4.5 XGBoost . . . 18
4.6 K-Nearest Neighbors . . . 19
4.7 Minimization of the MED . . . 20
4.7.1 Manual Minimization . . . 20
4.8 Impact of a new variable on our models . . . 24
4.8.1 Random Forest . . . 25
4.8.2 XGBoost . . . 27

**5 Country Clustering 30**

**6 Zoom in: Paris Temperature Peak? 35**

**7 General Conclusion 39**

## 1 Introduction: Why climate change?

The emergence of the Big Data era is an opportunity for scientists to test their theories and to refine their models: *currently, about 25 laboratories across the world support almost 50 climate models, forming the basis of climate projections (predictions) assessed by the Intergovernmental Panel on Climate Change (IPCC), which was established by the United Nations in 1988 and received the 2007 Nobel Peace Prize (shared with former Vice President Al Gore).*[3].But we wanted to see if we could at our level and using simple machine learning models assess climate change. We naturally use simpler machine learning models than those used by climate experts. The Big Data is also an opportunity for the public to realise how serious environmental challenges are. More data indeed imply more graphs, plots that can easily be understood by people without scientific knowledge. In this project we thus also stress the variety and interactivity of the plotting, majorly using a very innovative package called bokeh[3]. We show in the report almost all the graphs we did but in order to fully take advantage of their interactivity, we invite the reader to use our notebook. The next step was to find a significant database. We found one on the website Kaggle. Their are five files in the dataset: the monthly global temperatures since 1750 globally (1), by country (2), by state (3), by major city (4) or by city. In this project, we use the the files (1), (2) and (4). In the last part, we add one variable, the level of CO2 pollution and evaluate its impact on our models.

## 2 Data description and preparation

We began by importing the global temperatures. We import as well the standard data analysis packages Numpy and Pandas, some plotting packages Maplotlib, Seaborn and Bokeh. Finally, the machine learning packages that we use in our models from scikit-learn and XGBoost.

Listing 1: Imports

```
# Imports

# pandas
import pandas as pd
from pandas import Series,DataFrame

# numpy, matplotlib, seaborn, bokeh
import numpy as np
import matplotlib.pyplot as plt
import seaborn as sns
import bokeh as bk
sns.set_style('whitegrid')
%matplotlib inline

# machine learning sklearn
from sklearn import linear_model
from sklearn import metrics
from sklearn.svm import SVC, LinearSVC
from sklearn.ensemble import RandomForestClassifier
```

[3]Notebook precision: the reader has to run the concerned cells to see the bokeh graphs appear

```python
from sklearn.ensemble import RandomForestRegressor
from sklearn.neighbors import KNeighborsClassifier
from sklearn.naive_bayes import GaussianNB

#machine learning xgboost
import xgboost
```

Then we visualise and clean our data. We want to keep only two variables: the Land Average Temperature and the Uncertainty at 95% regarding the temperature. The first problem we faced was the number of NaN values on the dataset. Back in the 18th century, it was probably much more difficult to evaluate the global Land Average Temperature. After this cleaning, using the function *dropna()* we realise that the data set becomes really significant from 1800 onwards. It is consistent with the fact that climate experts generally use temperature data that start in the 19th century[3].

Listing 2: Visualisation, Cleaning

```python

# get global temp csv file as a DataFrame
global_temps = pd.read_csv("GlobalTemperatures.csv", sep=',')

# preview the data
global_temps.head()

#Delete all columns except dates, Land Average Temperatures,
#Land Average Temperature Uncertainty
global_temps=global_temps.drop(global_temps.columns[3:], axis=1)

global_temps.head()

#Delete all empty lines
global_temps=global_temps.dropna()
```

We understand that the data will not be easily displayed if we keep the monthly temperatures. We resize the data yearly and build a simpler dataframe.

Listing 3: Resizing

```python
#Resizing the datasets, we only keep the year averages

date=global_temps['dt'].apply(lambda x: x[:4])
years = np.unique(date)
mean_temps_global=[]
mean_temps_uncertainty=[]

for y in years:

    mean_temps_global.append(global_temps[y==date]
    ['LandAverageTemperature'].mean())
    mean_temps_uncertainty.append(global_temps[y==date]
    ['LandAverageTemperatureUncertainty'].mean())

#Defining a function that classifies the average temperatures

def heat(x):

    if type(x) is str:
        return x
    elif x<=7:
        return 3
    elif x>7 and x<=8:
        return 2
    else:
        return 1

#Building of a new, simpler dataframe

years=pd.to_numeric(years)

d1={'Dates': pd.Series(years),'Temperatures':pd.Series(mean_temps_global),
'Uncertainty':pd.Series(mean_temps_uncertainty)}

df_global_mean=pd.DataFrame(d1)

#Adding the heat function

e=df_global_mean.applymap(heat)

d2={'Dates': pd.Series(years),'Temperatures':pd.Series(mean_temps_global),
'Uncertainty':pd.Series(mean_temps_uncertainty),
    'Heat':e['Temperatures']}

df_global_mean=pd.DataFrame(d2)
```

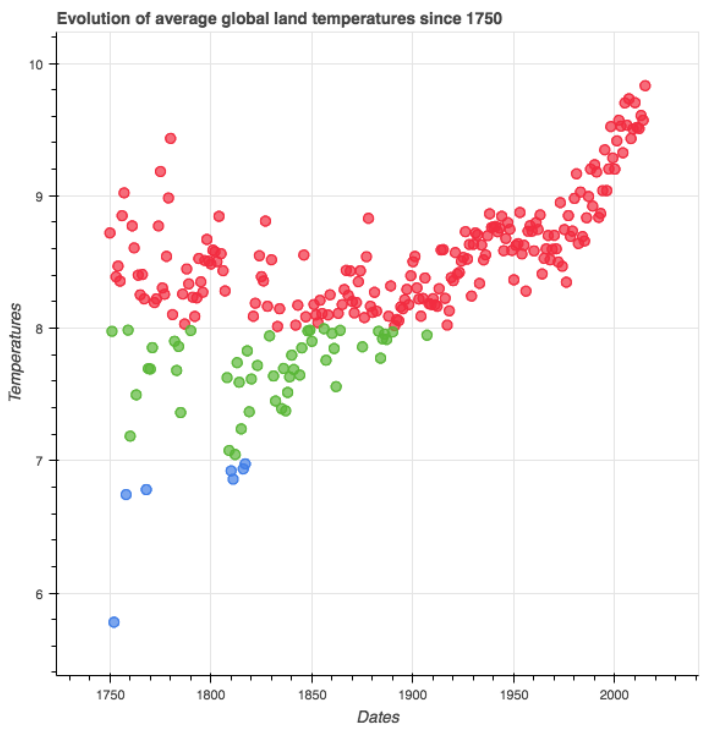
## 3 Exploratory Data Analysis

We plot the first graph using bokeh:

Listing 4: Evolution of Average Global Temperatures since 1750

```python
#Global Average Temperatures plot using Bokeh

from bokeh.io import output_notebook
from bokeh.charts import Scatter, show
from bokeh.plotting import figure
output_notebook()
p = Scatter(df_global_mean, x='Dates', y='Temperatures',
color='Heat', legend=None, background_fill_color="white",
title="Evolution of average global land temperatures since 1750")
show(p)
```

Figure 1: Evolution of average global land temperatures since 1750



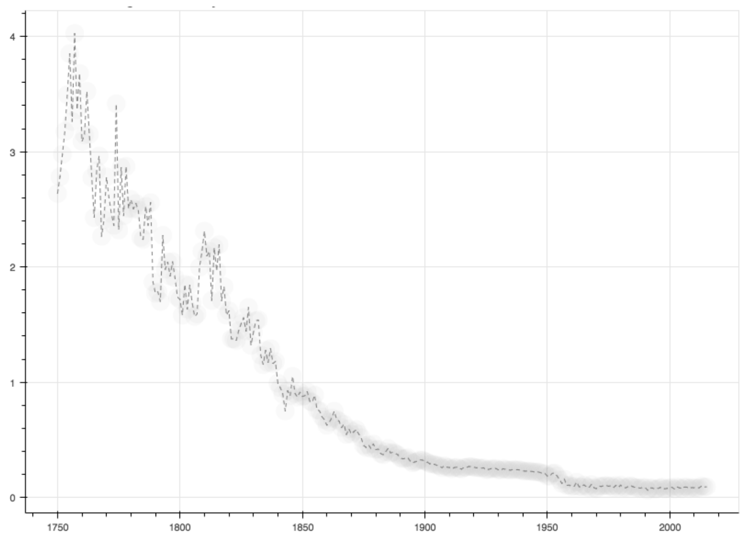
We see a clear path towards increasing temperatures. From 1900 onwards, the average global temperatures variance is decreasing dramatically and from that date, the temperatures are increasing at an ever faster pace. Now we take a look at the uncertainty (in celcius degree) regarding the accuracy of the measured temperatures.

Listing 5: Evolution of the Uncertainty regarding the Average Global Temperatures since 1750

```python
#Now, we plot the 95% Uncertainty regarding the temperature
    evolution

from bokeh.plotting import figure, show
from bokeh.models import ColumnDataSource, Circle, HoverTool,
    CustomJS
from bokeh.io import output_notebook

output_notebook()

(x, y) = (df_global_mean['Dates'], df_global_mean['Uncertainty'
    ])

# Basic plot setup
p = figure(width=800, height=600, toolbar_location=None,
title='A clear decreasing of uncertainty')

p.line(x, y, line_dash="4 4", line_width=1, color='gray')

```

```
# Add a circle, that is visible only when selected
source = ColumnDataSource({'x': x, 'y': y})
invisible_circle = Circle(x='x', y='y', fill_color='gray',
fill_alpha=0.05, line_color=None, size=20)
visible_circle = Circle(x='x', y='y', fill_color='firebrick',
fill_alpha=0.5,
line_color=None, size=20)
cr = p.add_glyph(source, invisible_circle,
selection_glyph=visible_circle, nonselection_glyph=
    invisible_circle)

# Add a hover tool, that selects the circle
code = "source.set('selected', cb_data['index']);"
callback = CustomJS(args={'source': source}, code=code)
p.add_tools(HoverTool(tooltips=None, callback=callback,
renderers=[cr], mode='hline'))

show(p)
```

Figure 2: Evolution of uncertainty regarding the average global land temperatures since 1750



The uncertainty is clearly decreasing as well and is near 0 for the past few years. Global warming is a reality, not a fiction. To be able to fully appreciate the interactivity of the graph, we encourage the reader to take a look at the notebook.

After doing the same manipulation as those done for the average, we explore the evolution of the extremes and we plot them.

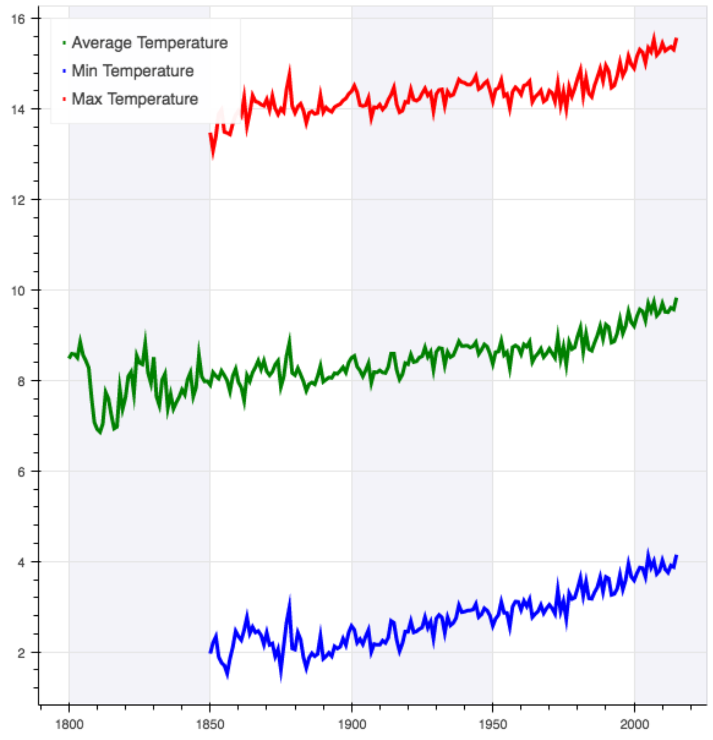
Listing 6: Evolution of the Min and Max Temperatures

```
#Loading the Min and Max data

global_temps2 = pd.read_csv("GlobalTemperatures.csv", sep=',')

# Starting in 1800 when data is more significant (less NaN)
```

```python

list_of_years_1800=years[50:]

# Compute Average, Max, Min Temperature by Year since 1800

date=global_temps2['dt'].apply(lambda x: x[:4])
list_of_years_1800=np.unique(date)
list_of_years_1800=list_of_years_1800[50:]

mean_temps_1800=[]
min_temps_1800=[]
max_temps_1800=[]

for y in list_of_years_1800:

    mean_temps_1800.append(global_temps2[y==date]
    ['LandAverageTemperature'].mean())
    max_temps_1800.append(global_temps2[y==date]
    ['LandMaxTemperature'].mean())
    min_temps_1800.append(global_temps2[y==date]
    ['LandMinTemperature'].mean())

#Building of a new, simpler dataframe

list_of_years_1800=pd.to_numeric(list_of_years_1800)

d1={'Average Temperature':pd.Series(mean_temps_1800),
'Dates': pd.Series(list_of_years_1800),
'Min Temperature':pd.Series(min_temps_1800),
'Max Temperature':pd.Series(max_temps_1800)}

global_temps_1800=pd.DataFrame(d1)

global_temps_1800=global_temps_1800[['Dates','Average
    Temperature',
'Min Temperature','Max Temperature']]

from bokeh.plotting import figure, show
from bokeh.models import ColumnDataSource, Circle, HoverTool,
    CustomJS
from bokeh.io import output_notebook

output_notebook()

x = global_temps_1800['Dates']
y = global_temps_1800['Average Temperature']
z = global_temps_1800['Min Temperature']
t = global_temps_1800['Max Temperature']

p = figure(plot_width=600, plot_height=600)


p.line(x, y, legend="Average Temperature", line_color="green",
    line_width=3)

p.line(x,z, legend="Min Temperature", line_color="blue",
    line_width=3)

p.line(x, t, legend="Max Temperature", line_color="red",
    line_width=3)

# change just some things about the x-grid
```

```
p.xgrid.band_fill_alpha = 0.05
p.xgrid.band_fill_color = "navy"

p.legend.location = "top_left"
p.legend.glyph_width = 2
p.legend.label_width=2

show(p)
```

Figure 3: Comparison Extreme and Average Temperatures since 1800



We notice the lack of data regarding the extremes between 1800 and 1850. The upward trend is confirmed.

The result of our first Exploratory Data Analysis helps us understand the Average Temperature trend globally over the past few centuries. We then train and test several machine learning models on the dataset.

## 4 Predictive Modelling

After having explored the data visualy. We try several machine learning models. To evaluate their accuracy, we define a metric that measures the fit of the model to the data. We call it Mean Euclidean Distance (MED), it is defined as:

**Definition 1.**

$$\mathbb{E}[||\beta - \hat{\beta}||]$$

*With $\beta$ being the true value of the temperature, $\hat{\beta}$ the predicted value and $||.||$ being the Euclidean Norm.*

We use this metric for almost every machine learning model we devised. The only exceptions being the Linear and Polynomial Regression for which we use more classical metrics like the mean squared error or the explained variance score. Furthermore, the visual aspect of regression is sufficient to see if it fits or not the data.

### 4.1 Linear Regression

We use first the simpler continuous machine learning model: the linear regression. The output variable is the temperature, the input variables are the dates. We evaluate the relationships between time and temperature. First, we train the model on the whole dataset. To test the accuracy of the model, we use graphs and classical regression metrics.

Listing 7: Linear Regression

```python
#Regression on all the data (without prediction) => See if a
    regression can be a good estimation

import matplotlib.pyplot as plt
import numpy as np
from sklearn import linear_model


#Define the input and output
temp=global_temps_1800['Average Temperature']
time=global_temps_1800['Dates']


# The data sets
dates_X = time.iloc[0:,]

# The targets
temperature_y = temp.iloc[0:,]

# Create linear regression object
reg = linear_model.LinearRegression()

# Train the model using the data sets
reg.fit(dates_X.to_frame(),temperature_y.to_frame())

# The estimation
output_reg=reg.predict(dates_X.to_frame())
output_simple=np.copy(output_reg)
output=pd.DataFrame(index=pd.DataFrame(output_simple).index.
    values)
```

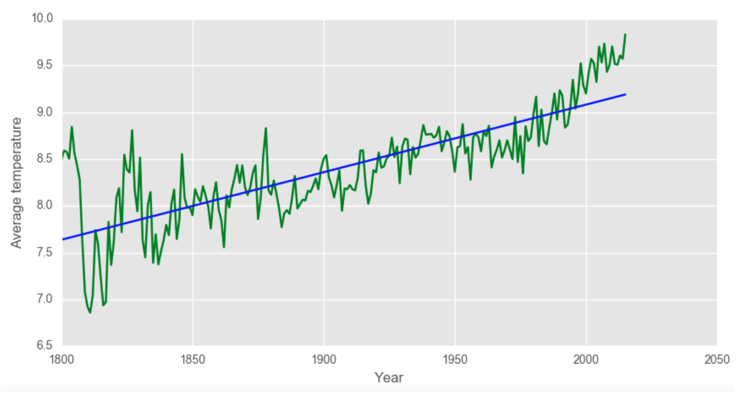
```
# The coefficients
print('Coefficients: \n', reg.coef_)
# The intercept
print('Intercept: \n', reg.intercept_)


# The mean squared error
print("Mean squared error: %.2f" % np.mean((reg.predict(dates_X.
    to_frame()) - temperature_y.to_frame()) ** 2))

# Explained variance score: 1 is perfect prediction
print('Variance score: %.2f' % reg.score(dates_X.to_frame(),
    temperature_y.to_frame()))
```

We obtain the following results:

```
Coefficients:[ 0.00721859]
Intercept:[-5.35578585]
Mean squared error: 0.12
Variance score: 0.64
```

And the following graph:

Figure 4: Linear Regression 1



As shown by the graphe, the regression seems to give accurate estimates of our real temperatures. Then we try to use it as a predictive model, separating the train and test databases. The training database spans from 1800 to 1949 and the test database from 1950 to 2015.

Listing 8: Linear Regression

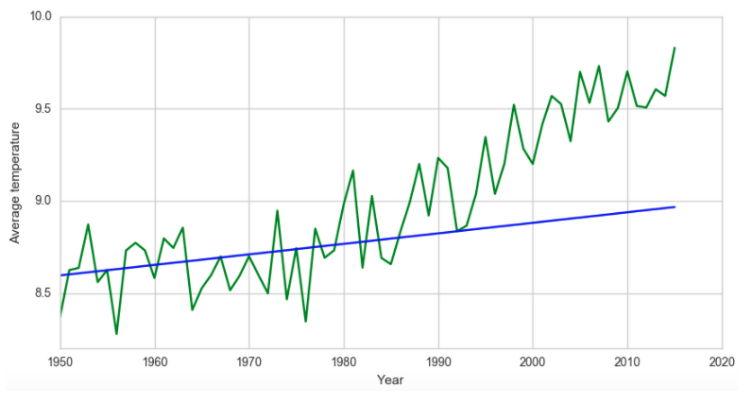
```
import matplotlib.pyplot as plt
import numpy as np
from sklearn import linear_model

#Define the input and output
temp=global_temps_1800['Average Temperature']
time=global_temps_1800['Dates']

# Split the data into training/testing sets
dates_X_train = time.iloc[0:150,]
dates_X_test = time.iloc[150:,]
```

```

# Split the targets into training/testing sets
temperature_y_train = temp.iloc[0:150,]
temperature_y_test = temp.iloc[150:,]

# Create linear regression object
reg = linear_model.LinearRegression()

# Train the model using the training sets
reg.fit(dates_X_train.to_frame(),temperature_y_train.to_frame())

# The coefficients
print('Coefficients: \n', reg.coef_)
# The intercept
print('Intercept: \n', reg.intercept_)

# The prediction
output_simple=reg.predict(dates_X_test.to_frame())

# The mean squared error
print("Mean squared error: %.2f"
      % np.mean((reg.predict(dates_X_test.to_frame())
      - temperature_y_test.to_frame()) ** 2))

# Explained variance score: 1 is perfect prediction
print('Variance score: %.2f' % reg.score(dates_X_test.to_frame()
    ,
 temperature_y_test.to_frame()))

# Linear regression curve on the predicted values

fig, axes = plt.subplots(figsize=(12,10))
axes.set_ylabel('Average temperature')
axes.set_xlabel('Year')
plt.plot(dates_X_test, temperature_y_test, label='Annual Mean
    temperature',
color='g')
plt.plot(dates_X_test,output_simple,label='Forecast Temperature
    by Regression',
color='b')
```

The metrics of the output are:

```
Coefficients:[ 0.00570766]
Intercept:[-2.53305848]
Mean squared error: 0.14
Variance score: 0.16
```

Figure 5: Linear Regression 2



We deduce from the metrics and the graph that the linear regression is not relevant when it comes to testing. Whereas the parameters of the model were accurately fitting the evolution of the temperatures when they took their information from the whole dataset, they are not relevant anymore when they are built from a part of the dataset. Therefore, the linear regression is not an efficient predictive model.

## 4.2 Polynomial Regression

"*Historically, the standard way to extend linear regression to settings in which the relationship between the predictors and the response is nonlinear has been to replace the standard linear model*"[4]. It seems to be the case for time and temperature. We thus try to implement a Polynomial Regression with $\deg(\mathbb{P}) \leq 3$. We realise the same experience as regarding the linear regression. First, we train the model on the whole dataset. To evaluate the fit, we use once again a graph and traditional regression metrics.

Listing 9: Polynomial Regression 2

```
#Polynomial Regression t^2 + t^3

#Regression on all the data (without prediction) => See if a
    regression can be a good estimation

#Define the input and output
time=pd.DataFrame()
time["Dates"]=global_temps_1800['Dates']
time["Dates^2"]=global_temps_1800['Dates']*global_temps_1800['
    Dates']
time["Dates^3"]=global_temps_1800['Dates']*global_temps_1800['
    Dates']*global_temps_1800['Dates']
temp=global_temps_1800['Average Temperature']


# The data sets
dates_X = time.iloc[0:,0]

```

[4] [2]

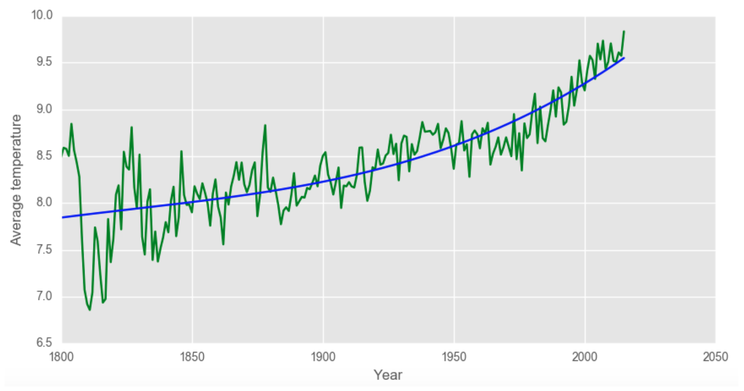
```
# The targets
temperature_y = temp.iloc[0:,]

# Create linear regression object
reg = linear_model.LinearRegression()

# Train the model using the training sets
reg.fit(time.iloc[0:,],temperature_y.to_frame())

# The prediction
output_reg=reg.predict(time.iloc[0:,])
output_multiple=np.copy(output_reg)
output=pd.DataFrame(index=pd.DataFrame(output_simple).index.
    values)
# The coefficients
print('Coefficients: \n', reg.coef_)
# The intercept
print('Intercept: \n', reg.intercept_)

# The mean squared error
print("Mean squared error: %.2f" % np.mean((output_reg -
    temperature_y.to_frame()) ** 2))

# Explained variance score: 1 is perfect prediction
print('Variance score: %.2f' % reg.score(time.iloc[0:,],
    temperature_y.to_frame()))

# Euclidian Distance
output['Predicted Temperature']=pd.DataFrame(output_multiple)
output["Average Temperature"]=temp
```

The results are the following ones:

```
Coefficients:[1.54583760e+00 -8.43972200e-04 1.53912499e-07]
Intercept:[-937.81103859]
Mean squared error: 0.10
Variance score: 0.69
```

And the graph:

Figure 6: Linear Regression 2



The model seems to fit the data even better than the first linear regression. We study then as before the predictive power of it. Once again, the train database spans from 1800 to 1949 and the test database from 1950 to 2015.

Listing 10: Polynomial Regression 2

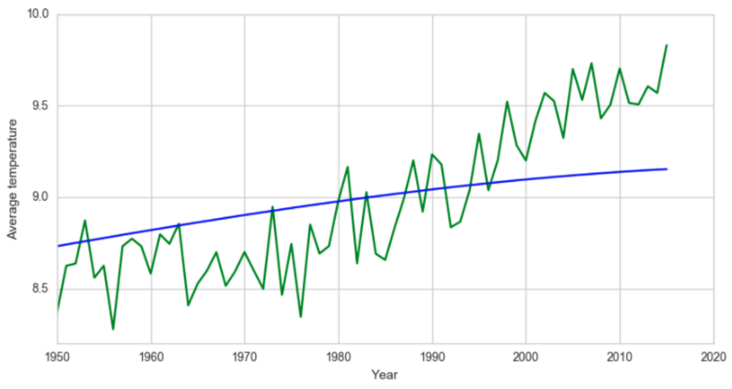
```python
#Polynomial Regression t^2 + t^3

#Define the input and output
temp=global_temps_1800['Average Temperature']
time=pd.DataFrame()
time["Dates"]=global_temps_1800['Dates']
time["Dates^2"]=global_temps_1800['Dates']*global_temps_1800['
    Dates']
time["Dates^3"]=global_temps_1800['Dates']
*global_temps_1800['Dates']*global_temps_1800['Dates']

# Split the data into training/testing sets
dates_X_train = time.iloc[0:150,0]
dates_X_test = time.iloc[150:,0]

# Split the targets into training/testing sets
temperature_y_train = temp.iloc[0:150,]
temperature_y_test = temp.iloc[150:,]

# Create linear regression object
reg = linear_model.LinearRegression()

# Train the model using the training sets
reg.fit(time.iloc[0:150,],temperature_y_train.to_frame())

# The coefficients
print('Coefficients: \n', reg.coef_)
# The intercept
print('Intercept: \n', reg.intercept_)

# The prediction
output_multiple=reg.predict(time.iloc[150:,])

# The mean squared error
print("Mean squared error: %.2f"
      % np.mean((reg.predict(time.iloc[150:,])
      - temperature_y_test.to_frame()) ** 2))

# Explained variance score: 1 is perfect prediction
print('Variance score: %.2f' % reg.score(time.iloc[150:,]
, temperature_y_test.to_frame()))

# Multiple Linear Regression: t,t^2, t^3:

fig, axes = plt.subplots(figsize=(12,10))
axes.set_ylabel('Average temperature')
axes.set_xlabel('Year')
# Plot
plt.plot(dates_X_test, temperature_y_test,
label='Annual Mean temperature', color='g')
plt.plot(dates_X_test,output_multiple,label='Forecast Reg',color
    ='b')
```

The results are:

Coefficients:[-3.37849536e+00 1.75645409e-03 -3.03542634e-07]
Intercept:[2168.61282322]

Mean squared error: 0.10
Variance score: 0.42

Figure 7: Polynomial Regression



The variance score is significantly higher than the 2nd linear regression's one. Still, we cannot consider it as a relevant predictive model for our data. We turn to the correlation matrix to see if we can learn more about our variables.

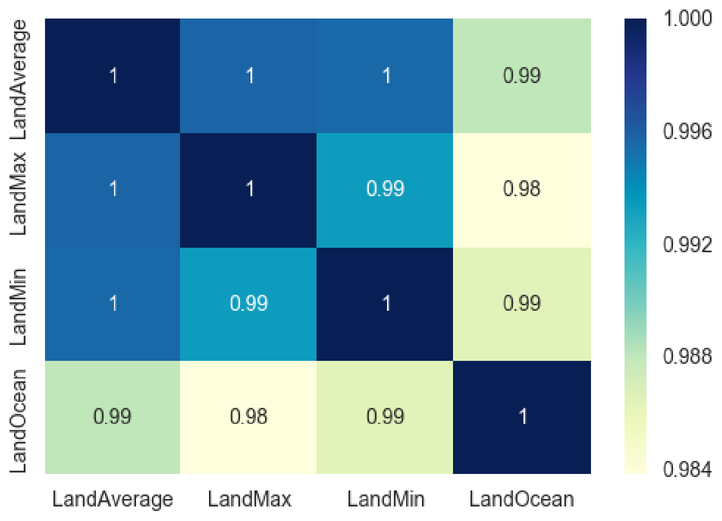
## 4.3 Correlation Matrix

We calculate the correlations between some of our variables. We only look at all the temperatures variables, we disregard here the uncertainty.

Listing 11: Correlation Matrix

```python
#Correlation Matrix for the following features :
#LandAverageTemperature, LandMaxTemperature,
#LandMinTemperature,LandAndOceanAverageTemperature

global_temperatures=pd.read_csv("GlobalTemperatures.csv", sep=',
    ')
global_temperatures['dt'] = pd.to_datetime(global_temperatures['
    dt'])
global_temperatures=global_temperatures.drop(
['LandAverageTemperatureUncertainty',"
    LandMaxTemperatureUncertainty","
    LandMinTemperatureUncertainty","
    LandAndOceanAverageTemperatureUncertainty"],axis=1)
global_temperatures=global_temperatures.dropna()
corr=global_temperatures.corr()

# Heatmap for the previous correlation matrix
sns.heatmap(corr,xticklabels=['LandAverage','LandMax','LandMin',
    'LandOcean'], yticklabels=['LandAverage','LandMax',
'LandMin','LandOcean'], annot=True,cmap="YlGnBu")
```

Figure 8: Correlation Matrix



The variables are very linked. Only the variable *LandandOceanTemperature* is slightly less correlated with than the others between them. We turn to another predictive model, the Random Forest Regressor.

## 4.4 Random Forest

We use the Random Forest Regressor proposed by the scikit-learn package. We tested different values regarding the number of trees and we realised that $n = 10$ was the most efficient. We decide to choose a larger training database than for the previous regressions. We regress the average temperatures on the landandocean, min and max temperatures, one month before. Otherwise, the output (average) would be based on inputs (min, max, landandoceanaverage) of the same year, which would biaised the result. The train database spans from 1800 to 1980 and the test database from 1981 onwards. It allows for a finer tuning of our model. To test the fit of the model, we use the metric defined above, what we named the Mean Euclidean Distance (MED).

Listing 12: First Random Forest

```

# Train
train_df_copy=pd.DataFrame.copy(global_temperatures[
    global_temperatures['dt'].
map(lambda x: x.year)<1980])
train_df=pd.DataFrame.copy(train_df_copy)
train_df.drop(train_df.index[:1], inplace=True)
train_df_copy.drop(train_df.index[-1],inplace=True)

# To predict each average land temperature we use the
#previous years' temperatures (ocean, min, max)
for j in (train_df.index):
    for column in train_df.columns[1:-1]:
        train_df.loc[j,column]=train_df_copy.loc[j-1,column]
```

```python
forest = RandomForestRegressor(n_estimators=10)
train_df=train_df.drop(['dt'],axis=1)
forest = forest.fit(train_df.drop(["LandAverageTemperature"],
    axis=1),
train_df["LandAverageTemperature"])

# Test
test_df_copy=pd.DataFrame.copy(global_temperatures[
    global_temperatures['dt'].
map(lambda x: x.year)>=1980])
test_df=pd.DataFrame.copy(test_df_copy)
test_df.drop(test_df.index[:1], inplace=True)
test_df_copy.drop(test_df.index[-1], inplace=True)

# To predict each average land temperature we use the
#previous year's temperatures (landandocean, min , max)
for j in (test_df.index):
    for column in test_df.columns[1:-1]:
        test_df.loc[j,column]=test_df_copy.loc[j-1,column]
test_df=test_df.drop(['dt'],axis=1)
output = forest.predict(test_df.drop(["LandAverageTemperature"],
    axis=1))

#Adding the predicted temperature to our output DF
test_df['Predicted_temp']=output

# Distance computing
MED=abs(test_df['LandAverageTemperature']-test_df['
    Predicted_temp']).mean()
```

We found a MED of 0.21886310904872408. We store this value and will compare it to the values obtained with other models.

## 4.5 XGBoost

After having used the Random Forest Regressor we use the eXtreme Gradient Boosting Regressor, XGBoostRegressor. The training and test sets, the output and input variables are the same as before. We decide to test the model with the default parameters and to analyse the outcome.

Listing 13: First XGBoost

```python

# Packages
import xgboost
from xgboost import XGBRegressor

# Train
train_df_copy_2=pd.DataFrame.copy(global_temperatures[
    global_temperatures['dt'].
map(lambda x: x.year)<1980])
train_df_2=pd.DataFrame.copy(train_df_copy_2)
train_df_2.drop(train_df_2.index[:1], inplace=True)
train_df_copy_2.drop(train_df_2.index[-1], inplace=True)

# To predict each average land temperature we use the
#previous year's temperatures (landandocean, min , max)
for j in (train_df_2.index):
    for column in train_df_2.columns[1:-1]:
        train_df_2.loc[j,column]=train_df_copy_2.loc[j-1,column]
```

```
forest2 = XGBRegressor()
train_df_2=train_df_2.drop(['dt'],axis=1)
forest2 = forest2.fit(train_df_2.drop(["LandAverageTemperature"
    ],
axis=1),train_df_2["LandAverageTemperature"])

# Test
test_df_copy_2=pd.DataFrame.copy(global_temperatures[
    global_temperatures['dt'].
map(lambda x: x.year)>=1980])
test_df_2=pd.DataFrame.copy(test_df_copy_2)
test_df_2.drop(test_df_2.index[:1], inplace=True)
test_df_copy_2.drop(test_df_2.index[-1], inplace=True)

# To predict each average land temperature we use the previous
#year's temperatures (landandocean, min , max)
for j in (test_df_2.index):
    for column in test_df_2.columns[1:-1]:
        test_df_2.loc[j,column]=test_df_copy_2.loc[j-1,column]
test_df_2=test_df_2.drop(['dt'],axis=1)
output = forest2.predict(test_df_2.drop(["LandAverageTemperature
    "],axis=1))

# Adding the predicted temperature to our output DF
test_df_2['Predicted_temp']=output

# Distance computing
MED=abs(test_df_2['LandAverageTemperature']-test_df_2['
    Predicted_temp']).mean()
```

We obtain a MED of 0.20515603448730613. The model is thus more accurate than the Random Forest Regressor.

## 4.6 K-Nearest Neighbors

We turn to the K-Nearest Neighbors machine learning method. After having tested a large amount of different values, we find that we minimize the MED with $k = 2$. The training and test sets, the output and input variables are the same as before.

Listing 14: K-Nearest Neighbors

```

## Packages
from sklearn.neighbors import KNeighborsRegressor

# Train
neigh = KNeighborsRegressor(n_neighbors=2)
train_df_copy_3=pd.DataFrame.copy(global_temperatures[
    global_temperatures['dt'].map(lambda x: x.year)<1980])
train_df_3=pd.DataFrame.copy(train_df_copy_3)
train_df_3.drop(train_df_3.index[:1], inplace=True)
train_df_copy_3.drop(train_df_3.index[-1], inplace=True)

# To predict each average land temperature we use the
#previous year's temperatures (landandocean, min , max)
for j in (train_df_3.index):
    for column in train_df_3.columns[1:-1]:
        train_df_3.loc[j,column]=train_df_copy_3.loc[j-1,column]
train_df_3=train_df_3.drop(['dt'],axis=1)
```

```
neighbors=neigh.fit(train_df_3.drop(["LandAverageTemperature"],
    axis=1),
train_df_3["LandAverageTemperature"])

# Test
test_df_copy_3=pd.DataFrame.copy(global_temperatures[
    global_temperatures['dt'].
map(lambda x: x.year)>=1980])
test_df_3=pd.DataFrame.copy(test_df_copy_3)
test_df_3.drop(test_df_3.index[:1], inplace=True)
test_df_copy_3.drop(test_df_3.index[-1], inplace=True)

# To predict each average land temperature we use the
#previous year's temperatures (landandocean, min , max)
for j in (test_df_3.index):
    for column in test_df_3.columns[1:-1]:
        test_df_3.loc[j,column]=test_df_copy_3.loc[j-1,column]
test_df_3=test_df_3.drop(['dt'],axis=1)
neigh_output = neighbors.predict(test_df_3.drop(["
    LandAverageTemperature"],axis=1))

# Adding the predicted temperature to our output DF
test_df_3['Predicted_temp']=neigh_output

MED=abs(test_df_3['LandAverageTemperature']-test_df_3['
    Predicted_temp']).mean()
```

We obtain a MED of 0.23760904872389807, which is higher than both Random Forest and XGBoost. We now want to see if we can find hyperparameters that would give us a lower MED for our most efficient model, the XGB Regressor. We use two hyperparameter optimization techniques: the grid-search technique, with the function *GridSearchCV* proposed by scikit-learn and a manual minimization.

## 4.7 Minimization of the MED

### 4.7.1 Manual Minimization

We first attempt to find the hyperparameters of the XGBoost that minimize the MED manually. We decide to restrain the possible values of the hyperparameters for the following reasons: if max_depth is too large, the model tends to overfit, regarding the learning_rate we tested several parameters and decided to choose a specific efficient range. As for the n_estimators, we choose not to go beyond 389 because once again, we tested different values.

Listing 15: Manual Minimization

```


#Definition of the function we want to minimize

def MED_XGB(n,maxi,learn,
        X_train, y_train, X_test,y_test,
        MED_limit):

    '''
    n          : n_estimators    -> Number of boosted trees to fit
    maxi       : max_depth       -> Maximum tree depth for base
        learners
    learn     : learning_rate -> Boosting learning rate
    ...
    MED_limit : -> threshold for minimum MED required
    '''

    xgb_model = xgboost.XGBRegressor(n_estimators = n ,
        learning_rate = learn, max_depth = maxi )
    model_fit = xgb_model.fit(X_train , y_train )
    y_predict = model_fit.predict(X_test)
    MED=abs(y_test-y_predict).mean()

    if MED<MED_limit:
        return MED , n, learn, maxi
    else:
        return 0

#### Data Sets ####

#Train
train_df_copy=pd.DataFrame.copy(global_temperatures[
    global_temperatures['dt'].map(lambda x: x.year)<1980])
train_df=pd.DataFrame.copy(train_df_copy)
train_df.drop(train_df.index[:1], inplace=True)
train_df_copy.drop(train_df.index[-1], inplace=True)
for j in (train_df.index):
    for column in train_df.columns[1:-1]:
        train_df.loc[j,column]=train_df_copy.loc[j-1,column]

train_df=train_df.drop(['dt'],axis=1)

#Test
test_df_copy=pd.DataFrame.copy(global_temperatures[
    global_temperatures['dt'].map(lambda x: x.year)>=1980])
test_df=pd.DataFrame.copy(test_df_copy)
test_df.drop(test_df.index[:1], inplace=True)
test_df_copy.drop(test_df.index[-1], inplace=True)
for j in (test_df.index):
    for column in test_df.columns[1:-1]:
        test_df.loc[j,column]=test_df_copy.loc[j-1,column]

test_df=test_df.drop(['dt'],axis=1)


X_train = train_df.drop("LandAverageTemperature", axis = 1)
y_train = train_df['LandAverageTemperature']

X_test = test_df.drop("LandAverageTemperature", axis = 1)
y_test = test_df['LandAverageTemperature']

#### Hyperparameters ####

max_depth = [2+i for i in range(5)]
learning_rate = [0.01*(i+1) for i in range(5)]
n_estimators = [370+i for i in range(20)]
MED_limit =  0.21
l=[]

#### Very long loop depending on parameters ####

for n in n_estimators:
    for learn in learning_rate:
        for maxi in max_depth:
            a = MED_XGB(n,maxi,learn,X_train,y_train,X_test,
                y_test,MED_limit)
            if a != 0:
                l.append(a)
                l.sort()

l[:5] # 5 Best MED Results
```

We obtain the following results:

[(0.2024272751996246, 385, 0.04, 2),
(0.2024369978097088, 386, 0.04, 2),
(0.20244117096460612, 387, 0.04, 2),
(0.20250207667782244, 384, 0.04, 2),
(0.20251076595633877, 383, 0.04, 2)]

Those results show us that there are hyperparameters that allow us to beat our previos XGBoost result. We want to compare this result with a Grid Search method.

Listing 16: Minimization using Grid Search

```

#### GridSearch (Take some time to compute) ####

from xgboost import plot_importance,to_graphviz, plot_tree
from sklearn.grid_search import GridSearchCV

xgb_model = xgboost.XGBRegressor()

clf = GridSearchCV(xgb_model,
                   {'max_depth': max_depth,
                    'n_estimators': n_estimators,
                   'learning_rate': learning_rate}, verbose=1)

clf.fit(X_train,y_train)
clf.best_score_, clf.best_params_
```

We obtain the following results:

(0.9921251007122502, 'learning_rate': 0.02, 'max_depth': 2, 'n_estimators': 381)

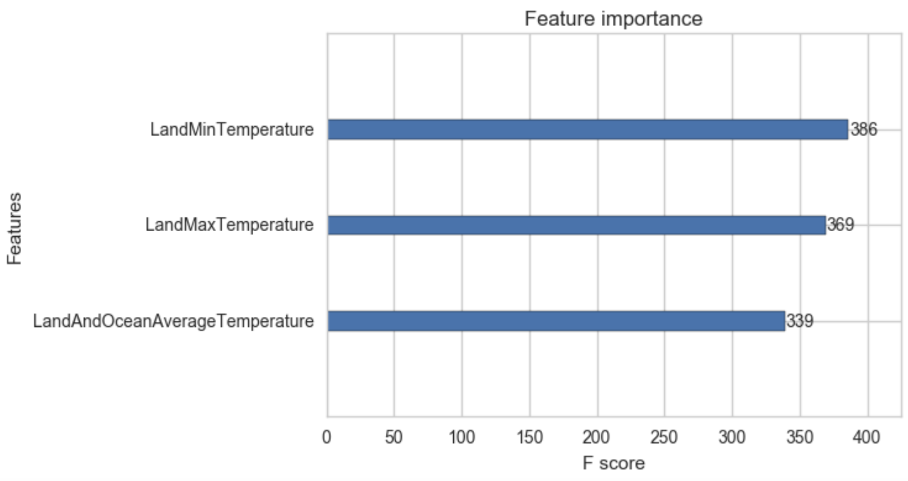
The first number being a score, the others, the hyperparameters that minimize the MED. We observe that these hyperparamaters give us an output MED of 0.21173377635362803 which is higher than the XGBoost taking the default hyperparameters and also higher than the MED we obtain with a manual minimization. We thus decide to take the first result of our manual optimization. Taking these hyperparameters to run the XGBoost, we evaluate the importance of the different features in the XGBoost Regressor prediction.

Listing 17: Model Features

```

#Using the results of our minimization
maxi = 2
```

```
learn = 0.04
n = 385

xgb_model = xgboost.XGBRegressor(max_depth = maxi,
                                 learning_rate = learn,
                                 n_estimators = n)

model_fit = xgb_model.fit(X_train,y_train)

#Ranking the features'importance
plot_importance(model_fit)
```

Figure 9: Features Importance

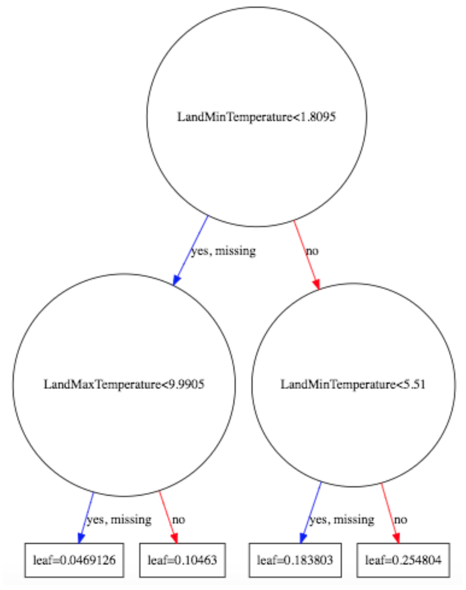


We obtain that the most discriminant feature is the LandMinTemperature, closely followed by the LandMaxTemperature. We then plot the most efficient tree of the XGBoost model.

Listing 18: Decision Tree

```
#Plotting an exemple of node and leafs
to_graphviz(model_fit)
```

Figure 10: Decision Tree



Naturally, we observe that the most discriminatory feature LandMinTemperature appears at the top of the tree.

We seem to have found hyperparameters that allow us to minimize significantly our MED, as well as identified the most discriminatory features of the model. Now, we evaluate the impact of the introduction of a new variable, the CO2 level into our analysis.

## 4.8 Impact of a new variable on our models

We decide to add a CO2 variable to our dataset. The CO2 data[5] spans from 1958 to 2016. We drop the 2016 year because our initial database stops in 2015. We also decide to assess the impact on our two most efficient models to date: Random Forest Regressor and XGBoost Regressor. Here we use directly the optimization techniques seen before, in order to find the most efficient hyperparameters. Before, we import the new data and merge them with the initial dataset.

Listing 19: Import CO2 variable

```
# Adding the CO2 level as a feature

CO2=pd.read_csv("co2-mm-mlo.csv",sep=",")

# We drop the year 2016 since we don't have the temperatures of
    2016
CO2 = CO2.drop(CO2.index[-10:])
CO2["Date"]=CO2["Date"]+"-01"
CO2["Date"]=pd.to_datetime(CO2["Date"])
CO2=CO2.drop(["Decimal Date","Interpolated","Trend","Number of
    Days"],axis=1)
```

[5]http://data.okfn.org/data/core/

```

# We merge the temperatures data with the CO2 data and we keep
#only the months where we have the average CO2 levels
temperatures=pd.merge(left=global_temperatures,right=CO2,
    left_on='dt', right_on='Date', how='outer')
temp=pd.DataFrame.copy(temperatures[temperatures['dt'].map(
    lambda x: x.year)<1980])
temp=temp.dropna()
```

We can now use our predictive models on the new dataset. This time, we decide to directly use the hyperparameters optimization techniques. The set on which we are minimizing was chosen after having done several trials.

### 4.8.1 Random Forest

Listing 20: Random Forest Optimization

```
#Sets

# Train
train_df_copy=pd.DataFrame.copy(temp)
train_df=pd.DataFrame.copy(train_df_copy)
train_df.drop(train_df.index[:1], inplace=True)
train_df_copy.drop(train_df.index[-1], inplace=True)

X_train = train_df.drop(["LandAverageTemperature","dt","Date"],
    axis = 1)
y_train = train_df['LandAverageTemperature']

# Test
test_df_copy=pd.DataFrame.copy(temperatures[temperatures['dt'].
    map(lambda x: x.year)>=1980])
test_df=pd.DataFrame.copy(test_df_copy)
test_df.drop(test_df.index[:1], inplace=True)
test_df_copy.drop(test_df.index[-1], inplace=True)

X_test = test_df.drop(["LandAverageTemperature","dt","Date"],
    axis = 1 )
y_test = test_df['LandAverageTemperature']

#Definition of the Minimization Function

def MED_RF(n,maxi,
       X_train, y_train, X_test,y_test,
       MED_limit):

#Note that there is no learning_rate for the random forest
    regressor

    '''
    n          : n_estimators  -> Number of boosted trees to fit
    maxi       : max_depth     -> Maximum tree depth for base
        learners
    ...
    MED_limit : -> threshold for minimum MED required
    '''

    forest = RandomForestRegressor(n_estimators=n,max_depth =
        maxi)
    model_fit = forest.fit(X_train , y_train )
    y_predict = model_fit.predict(X_test)
    MED=abs(y_test-y_predict).mean()

    if MED<MED_limit:
        return MED, n, maxi
    else:
        return 0

#Hyperparameters

max_depth = [8+i for i in range(7)]
n_estimators = [i+1 for i in range(100)]
MED_limit = 0.2
l=[]

#### Minimization (Very long loop depending on parameters) ####

for n in n_estimators:
    for maxi in max_depth:
        a = MED_RF(n,maxi,X_train,y_train,X_test,y_test,
            MED_limit)
        if a != 0:
            l.append(a)
            l.sort()

l[:5] # 5 Best MED Results

# GridSearch (very long)

clf = GridSearchCV(forest,
                   {'max_depth': max_depth,
                    'n_estimators': n_estimators}, verbose=1)

clf.fit(X_train,y_train)
clf.best_score_, clf.best_params_
```

We obtain the following output for the manual minimization:

```
[(0.12805185110461015, 23, 13),
 (0.13384931231904434, 36, 9),
 (0.1344451730051446, 92, 13),
 (0.1360461221964427, 25, 10),
 (0.1361788863109051, 20, 13)]
```

And for the GridSearch we obtain:

```
(0.9994277108488541, 'max_depth': 14, 'n_estimators': 54)
```

Note that we are minimizing only two hyperparameters. Contrary to the XG-Boost, Random Forest does not have a learning_rate hyperparameter. We test the parameters obtained with the Grid Search.

Listing 21: Random Forest-Grid Search Testing

```
#Random Forest (GridSearch MED Test)

# To predict each average land temperature we use the previous
    year's temperatures (landandocean, min , max) and the CO2
    level of the current year
```

```python
for j in (train_df.index):
    for column in train_df.columns[1:-2]:
        train_df.loc[j,column]=train_df_copy.loc[j-1,column]
forest = RandomForestRegressor(max_depth=14,n_estimators=54)
train_df=train_df.drop(['dt',"Date"],axis=1)
forest = forest.fit(train_df.drop(["LandAverageTemperature"],
    axis=1),train_df["LandAverageTemperature"])

for j in (test_df.index):
    for column in test_df.columns[1:-2]:
        test_df.loc[j,column]=test_df_copy.loc[j-1,column]
test_df=test_df.drop(['dt',"Date"],axis=1)
output = forest.predict(test_df.drop(["LandAverageTemperature"],
    axis=1))

# Distance computing
test_df['Predicted_temp']=output
MED=abs(test_df['LandAverageTemperature']-test_df['
    Predicted_temp']).mean()
MED
```

The result is: 0.15592940620434984. The GridSearch is again less efficient than the manual minimization, whose top result was 0.12805185110461015. Without the CO2 variable, the best Random Forest MED was 0.21886310904872408. The introduction of the CO2 variable decreases substantially our MED, which means that the model, here the Random Forest Regressor, better fits the new, richer data set. We turn to the XGB Regressor.

### 4.8.2 XGBoost

Listing 22: XGBoost Optimization

```python
#Sets

# Train
train_df_copy_2=pd.DataFrame.copy(temperatures[temperatures['dt'
    ].map(lambda x: x.year)<1980])
train_df_2=pd.DataFrame.copy(train_df_copy_2)
train_df_2.drop(train_df_2.index[:1], inplace=True)
train_df_copy_2.drop(train_df_2.index[-1], inplace=True)

# Test
test_df_copy_2=pd.DataFrame.copy(temperatures[temperatures['dt'
    ].map(lambda x: x.year)>=1980])
test_df_2=pd.DataFrame.copy(test_df_copy_2)
test_df_2.drop(test_df_2.index[:1], inplace=True)
test_df_copy_2.drop(test_df_2.index[-1], inplace=True)

#Hyperparameters
max_depth = [2+i for i in range(4)]
learning_rate = [0.01+0.01*i for i in range(5)]
n_estimators = [450+i*10 for i in range(10)]
MED_limit = 2
l=[]

#### Minimization (Very long loop depending on parameters) ####

for n in n_estimators:
    for learn in learning_rate:
        for maxi in max_depth:
            a = MED_XGB(n,maxi,learn,X_train,y_train,X_test,y_test,MED_limit)
            if a != 0:
                l.append(a)
                l.sort()

l[:5] # 5 Best MED Results

# GridSearch (very long)

clf = GridSearchCV(forest2,
                   {'max_depth': max_depth,
                    'n_estimators': n_estimators,
                   'learning_rate': learning_rate}, verbose=1)

clf.fit(X_train,y_train)
clf.best_score_, clf.best_params_

#XGBoost (GridSearch MED Test)

# To predict each average land temperature we use the previous year's temperatures (landandocean, min, max) and the CO2 level of the current year

for j in (train_df_2.index):
    for column in train_df_2.columns[1:-2]:
        train_df_2.loc[j,column]=train_df_copy_2.loc[j-1,column]
forest2 = XGBRegressor(learning_rate=0.01, max_depth=4, n_estimators=450)
forest2 = forest2.fit(train_df_2.drop(["LandAverageTemperature","dt","Date"],axis=1),train_df_2["LandAverageTemperature"])


for j in (test_df_2.index):
    for column in test_df_2.columns[1:-2]:
        test_df_2.loc[j,column]=test_df_copy_2.loc[j-1,column]
output = forest2.predict(test_df_2.drop(["LandAverageTemperature","dt","Date"],axis=1))

# Distance computing
test_df_2['Predicted_temp']=output
MED=abs(test_df_2['LandAverageTemperature']-test_df_2['Predicted_temp']).mean()
MED
```

The results are:

```
[(0.2016687891311823, 460, 0.04, 2),
(0.20183603119020402, 470, 0.04, 2),
(0.20186279160130058, 450, 0.04, 2),
(0.20253749801830465, 480, 0.04, 2),
(0.20266475269025555, 490, 0.04, 2)]
```

The results or GridSearch Optimization are:

```
(0.9227900204652463, 'learning_rate': 0.01, 'max_depth': 4, 'n_estimators':
450)
```

And the MED XGBoost result:

$$0.3102404183345716$$

We observe that the XGBoost MED results, 0.2016687891311823 at best are sensibly lower than the Random Forest's ones, 0.12805185110461015 at best. Remember that we find above that without the new variable, the XGB Regressor is more efficient than the Random Forest. We explain this new counter-intuitive result by two facts. First, the training database is far smaller than before, because the new variable does not have values before 1958. Second, our computing power is limited and we believe that if we had been able to minimize our function on a bigger set, we would have found more promising results for the XGBRegressor. We conclude that the introduction of the new variable has a significant impact on the MED regarding the Random Forest Regressor. As for the XGB Regressor, the impact on the MED is not significant (0.2016687891311823 here against 0.2024272751996246 without the new variable). We deduce that The Random Forest Regressor better fits our new dataset.

## 5 Country Clustering

In this section, we study the existing similarities in terms of climate path between countries throughout the world. We clean and prepare the dataframe to adapt it to the clustering.

Listing 23: Cleaning and Preparing the Dataframe

```python
country_temps = pd.read_csv("GlobalLandTemperaturesByCountry.csv", sep=',')

#Drop Uncertainty for the clustering

country_temps = country_temps.drop("AverageTemperatureUncertainty", axis = 1)

# Delete all empty lines, change time, and put time as Index

country_temps['dt'] = pd.to_datetime(country_temps['dt'])

country_temps.tail()

#Setting the Countries as Index, time as Columns and Average Temperature as the main Data
pivot = country_temps.pivot("Country","dt","AverageTemperature")

pivot = pivot.drop(pivot.columns[[i for i in range(1274)]+[3238]], axis = 1)

"""
    Data before 1850 is deleted because

    *30%  of Database is NaN if dt starts at 1773
    *18%  of Database is NaN if dt starts at 1800
    *5.2% of Database is NaN if dt starts at 1850

    Numbers obtained doing :
        -> np.count_nonzero(~np.isnan(pivot))

    Plus, the last month (9/2013) doesn't have any data

"""

pivot.head()

#Getting rid of all countries with no data (Only one country eliminated)

pivot = pivot.dropna(how='all')

#We decide to replace all the missing values by the median temperature of the concerned country. The mean would have been too influenced by extreme values.

from sklearn.preprocessing import Imputer

imp = Imputer(strategy="median", axis = 1)
# Beware: here axis = 1 means rows -> http://scikit-learn.org/stable/modules/generated/sklearn.preprocessing.Imputer.html


```

```python
#Imputer changes type of DataFrame.

trans_pivot = imp.fit_transform(pivot)
nonan_pivot = pd.DataFrame(trans_pivot)

nonan_pivot.columns = pivot.columns
nonan_pivot.index = pivot.index

nonan_pivot.tail()

print(np.count_nonzero(~np.isnan(nonan_pivot)))

#No missing data, and let's now call the DataFrame : clusterdf
clusterdf = nonan_pivot
```

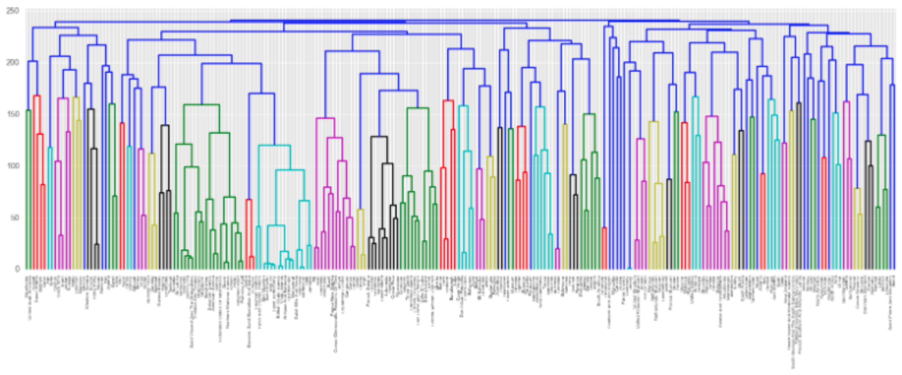
We define our Clustering function, using a scikit-learn package and plot it.

Listing 24: Clustering

```python
def Clustering(df,x,y,orientation,size):

    """

    df          : DataFrame
    x,y         : plt dimensions
    orientation : top, bottom, right or left
    size        : leaf_font_size

    """
    from sklearn.cluster import AgglomerativeClustering
    from scipy.cluster.hierarchy import dendrogram
    import matplotlib.pyplot as plt
    plt.style.use('ggplot')

    ward = AgglomerativeClustering(linkage='ward',
        compute_full_tree=True).fit(df)
    dendro = [ ]
    for a,b in ward.children_:
        dendro.append([a,b,float(len(dendro)+1),len(dendro)+1])

    fig = plt.figure( figsize=(x,y) )
    ax = fig.add_subplot(1,1,1)
    r = dendrogram(
        dendro,
        color_threshold=None,
        labels=df.index,
        distance_sort = True,
        show_leaf_counts=False,
        leaf_font_size = size,
        ax=ax,
        orientation = str(orientation)
    )

def Clustering_truncated(df,x,y,orientation,p,size):

    """

    df          : DataFrame
    x,y         : plt dimensions
    orientation : top, bottom, right or left
    p           : Only the last p merged clusters are shown
    size        : leaf_font_size
    """
    from sklearn.cluster import AgglomerativeClustering
    from scipy.cluster.hierarchy import dendrogram
    import matplotlib.pyplot as plt
    plt.style.use('ggplot')

    ward = AgglomerativeClustering(linkage='ward',
        compute_full_tree=True).fit(df)
    dendro = [ ]
    for a,b in ward.children_:
        dendro.append([a,b,float(len(dendro)+1),len(dendro)+1])

    fig = plt.figure( figsize=(x,y) )
    ax = fig.add_subplot(1,1,1)
    r = dendrogram(
        dendro,
        color_threshold=None,
        labels=df.index,
        show_leaf_counts=False,
        truncate_mode = 'lastp',  # show only the last p merged
            clusters
        p = p,  # show only the last p merged clusters
        leaf_font_size = size,
        show_contracted = True,  # to get a distribution
            impression in truncated branches
        ax=ax,
        orientation = str(orientation)

    )
    return r["ivl"]

#We can now plot the clustering
Clustering(clusterdf,20,6,"top",6)
```

We obtain the following graph:

Figure 11: First Clustering



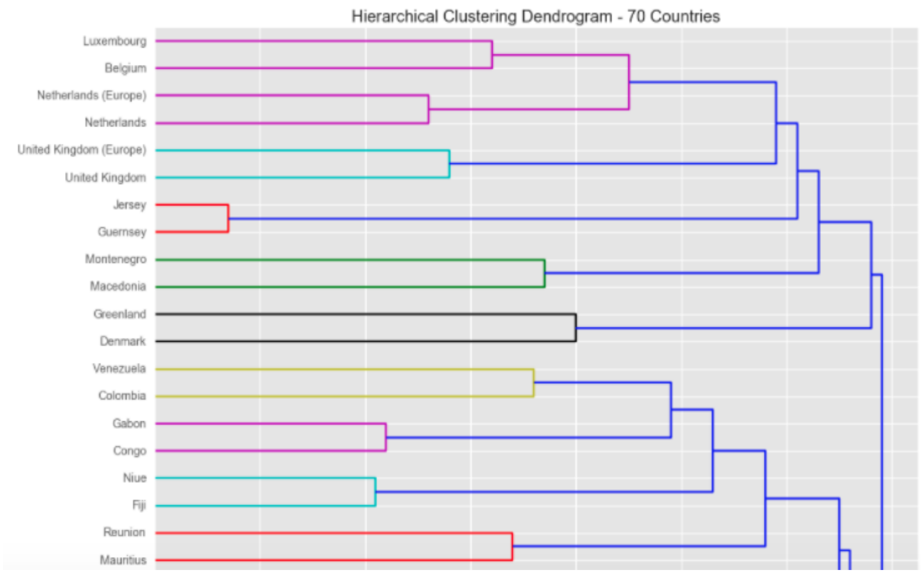
We enlarge the graph and put it vertically. Given the important size of the graph, we decide not to display it here. The reader can see it in the notebook. As shown by the previous graph, many countries do not share any similarities with others. The similarity is measured by a certain distance. The more important the distance, the less relevant the merging of the countries into clusters is. We thus create a second clustering which gathers only the countries that closely share climate similarities.

Listing 25: 2nd Clustering

```

#We remove from the cluster the most interesting countries in
    terms of shared climate similarities

Cluster_70 = Clustering_truncated(clusterdf,12,30,"right",200,8)
plt.title('Hierarchical Clustering Dendrogram (Truncated)')
plt.xlabel('Distance')
plt.ylabel('Countries')

#We define a function that only keeps the interesting #countries

def delete_symbol(l,sym):
    for i in range(l.count(sym)):
        l.remove(sym)
    return l, len(l)

cntry_to_del = delete_symbol(Cluster_70,'')
cntry_to_del[1]

#We deleted 172 countries
clusterdf_new = clusterdf.drop(cntry_to_del[0], axis = 0)

#We can now plot the new dendrogram
Clustering(clusterdf_new,12,30,"right",10)
plt.title('Hierarchical Clustering Dendrogram - 70 Countries')
plt.xlabel('Distance')
plt.ylabel('Countries')
```

We display the top part of the graph.

Figure 12: Second Clustering



We cannot see it here but the distance axis that spanned from 0 to 250 in the previous graphs stops here at 70. We thus have only the country, whose temperature trends are very similar.

## 6 Zoom in: Paris Temperature Peak?

We use the file GlobalLandTemperaturesByMajorCity.csv and focus on Paris. Once again, we decide to keep only the years from 1800 onwards given the numerous NaN we have before. We design an interactive heating-map using the bokeh package, displaying the temperature in Paris, every month since 1800, classified by colors. The reader can hover over the map on the notebook.

Listing 26: Zoom in, Paris

```

#Import of Paris'data

import pandas as pd
df=pd.read_csv('GlobalLandTemperaturesByMajorCity.csv')
df_Paris=df[df.City == 'Paris']
df_Paris=df_Paris.dropna()
df_Paris=df_Paris.drop(df_Paris.columns[2:], axis=1)

#Starts at 1800

df_Paris=df_Paris.iloc[602:]

#Building of a new Dataframe adapted to the map

date=df_Paris['dt'].apply(lambda x: x[:4])
years = np.unique(date)
mean_temps_Paris=[]

for y in years:

    mean_temps_Paris.append(df_Paris[y==date]['
        AverageTemperature'].mean())

jan=[]
for i in range(0,2562,12):
        jan.append(df_Paris.iloc[i]['AverageTemperature'])
feb=[]
for i in range(1,2562,12):
        feb.append(df_Paris.iloc[i]['AverageTemperature'])
mar=[]
for i in range(2,2562,12):
        mar.append(df_Paris.iloc[i]['AverageTemperature'])
apr=[]
for i in range(3,2562,12):
        apr.append(df_Paris.iloc[i]['AverageTemperature'])
may=[]
for i in range(4,2562,12):
        may.append(df_Paris.iloc[i]['AverageTemperature'])
jun=[]
for i in range(5,2562,12):
        jun.append(df_Paris.iloc[i]['AverageTemperature'])
jul=[]
for i in range(6,2562,12):
        jul.append(df_Paris.iloc[i]['AverageTemperature'])
aug=[]
for i in range(7,2562,12):
        aug.append(df_Paris.iloc[i]['AverageTemperature'])
sep=[]
for i in range(8,2562,12):
        sep.append(df_Paris.iloc[i]['AverageTemperature'])
octb=[]
for i in range(9,2562,12):
        octb.append(df_Paris.iloc[i]['AverageTemperature'])
nov=[]
for i in range(10,2562,12):
        nov.append(df_Paris.iloc[i]['AverageTemperature'])
dec=[]
for i in range(11,2562,12):
        dec.append(df_Paris.iloc[i]['AverageTemperature'])

#New Dataframe

df_Paris=pd.DataFrame({'Annual':pd.Series(mean_temps_Paris),
'Year':years,'Jan':pd.Series(jan),
                       'Feb':pd.Series(feb),'Mar':pd.Series(mar),
                       'Apr':pd.Series(apr),'May':pd.Series(may),
                      'Jun':pd.Series(jun),'Jul':pd.Series(jul),
                      'Aug':pd.Series(aug),'Sep':pd.Series(sep),
                      'Oct':pd.Series(octb),
                      'Nov':pd.Series(nov),'Dec':pd.Series(dec)})

df_Paris=df_Paris[['Year','Jan','Feb','Mar','Apr','May','Jun','Jul','Aug',
'Sep','Oct','Nov','Dec','Annual']]
df_Paris=df_Paris.dropna()

#Interactive map with monthly average temperatures in Paris since 1800

from collections import OrderedDict
import pandas as pd
import numpy as np
from bokeh.io import output_notebook, show
from bokeh.plotting import (ColumnDataSource, figure)
from bokeh.models import HoverTool

output_notebook()

# Read in the data with pandas. Convert the year column to string
df_Paris['Year'] = [str(x) for x in df_Paris['Year']]
years = list(df_Paris['Year'])
months = ["Jan","Feb","Mar","Apr","May","Jun","Jul","Aug","Sep","Oct",
"Nov","Dec"]
df_Paris = df_Paris.set_index('Year')

# this is the colormap from the original plot

colors = ["#75968f", "#75968f", "#a5bab7", "#c9d9d3", "#e2e2e2", "#dfccce",
          "#ddb7b1", "#cc7878", "#933b41", "#550b1d","#75968f","#75968f"]

# Set up the data for plotting. We will need to have values for every
# pair of year/month names. Map the Temperature to a color.

```

```python
def mapping_to_colors(x):
    if x<=0:
        return 1
    elif x>=0 and x<4:
        return 2
    elif x>=4 and x<8:
        return 3
    elif x>=8 and x<12:
        return 4
    elif x>=12 and x<14:
        return 5
    elif x>=14 and x<19:
        return 6
    elif x>=19 and x<22:
        return 7
    else:
        return 8

month = []
year = []
color = []
Temperature = []
for y in years:
    for m in months:
        month.append(m)
        year.append(y)
        Monthly_Temperature = df_Paris[m][y]
        Temperature.append(Monthly_Temperature)
        color.append(colors[mapping_to_colors(
            Monthly_Temperature)])

source = ColumnDataSource(
    data=dict(month=month, year=year, color=color, Temperature=
        Temperature)
)

TOOLS = "resize,hover,save,pan,box_zoom,wheel_zoom"

#Display parameters
p = figure(title="Paris Temperature (1800 - 2012)",
    x_range=years, y_range=list(reversed(months)),
    x_axis_location="above", plot_width=3000, plot_height=500,
    toolbar_location="left", tools=TOOLS)

p.rect("year", "month", 1, 1, source=source,
    color="color", line_color=None)

p.grid.grid_line_color = None
p.axis.axis_line_color = None
p.axis.major_tick_line_color = None
p.axis.major_label_text_font_size = "5pt"
p.axis.major_label_standoff = 0
p.xaxis.major_label_orientation = np.pi/3

#Hovers parameters
hover = p.select(dict(type=HoverTool))
hover.tooltips = OrderedDict([
    ('date', '@month @year'),
    ('Temperature', '@Temperature'),])

show(p)
```

Figure 13: Paris Heating Map Beginning

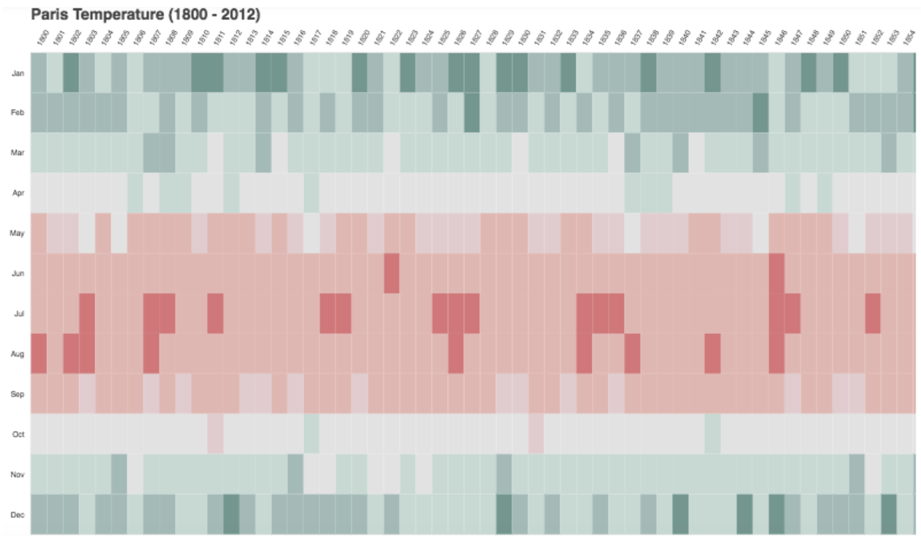
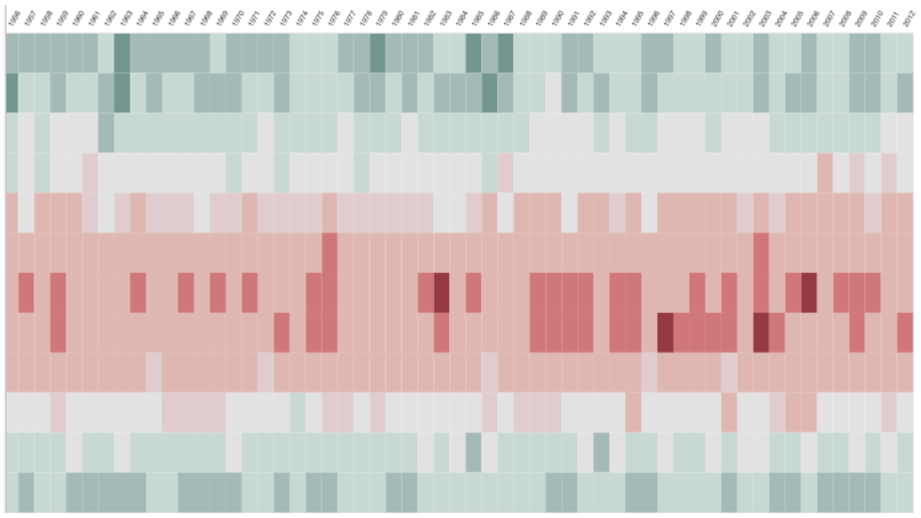


Figure 14: Paris Heating Map Last seventy years



We clearly see that the temperatures are rising in the French capital, with a significant acceleration starting in the 1980's. The center of the map, corresponding to the summer registers more and more extreme values. The borders of the map correponding to the winter, spring and autumn register less and less extremely cold values and becomes more homogeneous. This zoom on Paris echoes the recent pollution peak the city experienced. It is now clear that Paris' temperatures are growing and pollution peaks can only worsen the phenomenon.

## 7 General Conclusion

Following our data exploratory section, we tried out several machine learning models and tested their accuracy. Then we clustered the countries by temperature and finally we plotted an interactive heatmap of Paris. The efficient models are according to the MED, the Random Forest Regressor and the XGBoost Regressor regarding the initial dataset. When adding a new variable, we observe that the Random Forest Regressor produces better results. We regret the very slow process of minimization of the MED, whether it is through the GridSearch or via manual optimization. A more powerful computing power would have certainly helped us obtain finer results. There are no reason why the model would have a limited lifespan, except if the global warming becomes boiling and make human disappear. And we find no reason why the model would lose efficiency with time. Besides, our models can perfectly be trained with higher datasets.

In order to spare costs, we could only take the yearly temperature data. Our models could be used and executed by artificially intelligent robots if we automatise the data scraping and the structuration of the dataframes. It is also possible to detect the failing data updates with a simple findnan function scanning through the new data.

Regarding the business opportunity surrounding our models. We think an application would not spark interest because of the numerous weather related app already existing. We could nonetheless sell this model to industries that do not trust the already existing weather-predicting models and want to have another opinion. Agriculture that heavily relies on the climate could take advantage of our predictive models. Hedge Funds or Banks that trade on complex weather-derivatives or simply on commodities-related derivatives could also take advantage of these predictive models. An Excel extension could be proposed.

With the CO2 variable included, our models become more accurate. The MED is around 0.1 which is very small. In order to further reduce it, we have to add more variables that have relationships with the evolution of the temperatures. The level of human related pollutions generally, the deforestation, the ocean levels could enrich our models and reduce the errors. Nowadays, climate scientists actually use those kind of models, including many more variables.[3]

## List of Figures

1 Evolution of average global land temperatures since 1750 . . . . 6
2 Evolution of uncertainty regarding the average global land temperatures since 1750 . . . . . . . . . . . . . . . . . . . . . . . . . 7
3 Comparison Extreme and Average Temperatures since 1800 . . . 9
4 Linear Regression 1 . . . . . . . . . . . . . . . . . . . . . . . . . . 11
5 Linear Regression 2 . . . . . . . . . . . . . . . . . . . . . . . . . . 13
6 Linear Regression 2 . . . . . . . . . . . . . . . . . . . . . . . . . . 14
7 Polynomial Regression . . . . . . . . . . . . . . . . . . . . . . . . 16
8 Correlation Matrix . . . . . . . . . . . . . . . . . . . . . . . . . . 17
9 Features Importance . . . . . . . . . . . . . . . . . . . . . . . . . 23
10 Decision Tree . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 24
11 First Clustering . . . . . . . . . . . . . . . . . . . . . . . . . . . 32
12 Second Clustering . . . . . . . . . . . . . . . . . . . . . . . . . . 34
13 Paris Heating Map Beginning . . . . . . . . . . . . . . . . . . . . 38
14 Paris Heating Map Last seventy years . . . . . . . . . . . . . . . . 38

## References

[1] Jerome Friedman, Trevor Hastie, and Robert Tibshirani. *The elements of statistical learning*, volume 1. Springer series in statistics Springer, Berlin, 2001.

[2] Gareth James, Daniela Witten, Trevor Hastie, and Robert Tibshirani. *An introduction to statistical learning*, volume 6. Springer, 2013.

[3] Claire Monteleoni, Gavin A Schmidt, and Scott McQuade. Climate informatics: accelerating discovering in climate science with machine learning. *Computing in Science & Engineering*, 15(5):32–40, 2013.

[4] Sebastian Raschka. *Python Machine Learning*. Packt Publishing Ltd, 2015.

[5] Willi Richert. *Building machine learning systems with python*. Packt Publishing Ltd, 2013.