Time series analysis to predict variability of office space heating costs at UBS in London - Karl Merisalu

Background: There are many factors when it comes to optimising company costs. At UBS, one of areas that needs to be planned for and optimised is real estate operating costs. More specifically, heating and cooling costs of the office space.

Problem situation: When it comes to strategic planning of costs it is usually done on an annual basis, often with little detail into monthly/weekly variations.

Making cost projections with precision on a monthly/weekly basis would present an opportunity to: 1) better manage my company's cash liquidity position and 2) being able to allocate money better to areas which need it the most at given times.

However, it can be difficult to predict when exactly more needs to be spent on heating/cooling and when less. In London, the building needs to be cooled in the summer and heated in the winter, and perhaps the system needs to be switched off in the night time, but it is difficult to tell when exactly are likely peak heating and cooling times in the year and how much the temperature needs to be adjusted then. Or for example, how much the building temperature needs to be adjusted every day/week throughout the whole year?

Task: In order to shed more light on heating and cooling patterns of UBS London office throughout the year, I will conduct a time series analysis, based on daily UK weather temperature data of 38 years.

In my role as a Business Manager in UBS EMEA COO's Office I work closely with the Real Estate management team. I plan desk allocations in the building and escalate building related issues, including the temperature.

I also work with Finance teams, who are in charge of revenue and cost predictions.

As such, this analysis is important for me an my firm, because it will help the firm plan better for expected temperature fluctuations and also make our heating cost projections more accurate.

As a Business Manager, I act as a bridge between many functions and this analysis will be a great starting point for me to start discussions on abovementioned topics.

This project will focus on analysing historical weather temperature data with SARIMAX (Seasonal Autoregressive Integrated Moving Average) model.

I will be using Python ecosystem for this task, because it is good in dealing with large quantities of data and given good results, the model would be easy to replicate for our offices in other locations.

I will train the model based on historical data and then use it to make predictions for future weather temperatures as an example.

Limitations: There are many factors that affect temperature in office rooms and each office is different from it's systems and design perspective. The analysis seeks to provide an overall guidance for planning ahead.

Given the simplicity of the model and recent irregular weather patterns I don't expect the model to be highly accurate. However, at the same time, weather patterns tend to be similar over years (i.e. winter is colder than summer) and we're looking for a rough guidance for Real Estate and Finance team, so our analysis may be sufficient for this purpose.

Summary of actions: To summarize, this is a time series project, where I'm going to predict weather temperature in London based on past weather temperature data. This project will be useful for my firm and myself because it will hopefully assist in planning office heating costs more accurately, and for me to start a conversation on this topic with relevant teams. Based on model predictions, I will make suggestions to optimise our heating costs at UBS as a conclusion

Following is a step by step guide on how I'm going to accomplish it:
1) Importing libraries *(enables me to use pre-developed complex functions on the data)*
2) Loading data *(38 years of daily weather data)*
3) Preparing and cleaning data *(identify and fix shortcomings/flaws, data into series format)*
4) Seasonal decomposition of time series data *(into: trend, seasonality, noise components)*
5) Model and transform *(perform statistical checks to ensure data is stationary/suitable to be represented with the time series model I plan to fit and check modelling assumptions)*
6) Building the model
7) Plotting the model against original data *(to verify how well the model is able to reproduce data on which it was fitted)*
8) Forecasting *(what's the temperature going to be like?)*
9) Evaluation and conclusion *(including suggestions for further research)*
APPENDIX

1) Importing libraries

In [0]:

```
import pandas as pd
import numpy as np
import matplotlib.pylab as plt
import matplotlib.pyplot
%matplotlib inline
```

2) Loading data

Dataset summary:

The UK daily temperature data contain maximum and minimum temperatures (air, grass and concrete slab) measured over a period of up to 24 hours. The measurements were recorded by observation stations operated by the Met Office across the UK. The available data span from 1853 to 2018. For data download and further details please visit: https://catalogue.ceda.ac.uk/uuid/b37382e8c1e74b849831a5fa13afdcae (https://catalogue.ceda.ac.uk/uuid/b37382e8c1e74b849831a5fa13afdcae)

This data is free to download and for this analysis I will be using only part of the available data: from 1980 to end of 2017, which makes it 38 years of daily temperatures. This should be more than enough for this purpose.

UBS has 4 data confidentiality classes:
- Strictly confidential – not to be further distributed
- Confidential – only for targeted audience
- Internal – can be shared with colleagues internally
- Public – can be shared internally and externally

This weather dataset qualifies as public data, so there is no need to hide/transform any of the data.

As for location I'm going to use data from Heathrow weather station beause:
1) It is one of the longest serving station,
2) Has least missing data compared to other London weather stations
3) It is relatively close to our office in the City of London
4) The data is being measured twice a day in 12 hour intervals: from 09:00-21:00 and from 21:00-09:00

The data comes in annual batches, so first I need to load all (38) of these files and combine (concatenate) them into 1 set.

In [2]:

```
df80 = pd.read_csv('midas-open_uk-daily-temperature-obs_dv-201901_greater-london_00708_
heathrow_qcv-1_1980.csv', skiprows=90) # skipping 90 rows which includes unnecessary da
ta.
df81 = pd.read_csv('midas-open_uk-daily-temperature-obs_dv-201901_greater-london_00708_
heathrow_qcv-1_1981.csv', skiprows=90)
df82 = pd.read_csv('midas-open_uk-daily-temperature-obs_dv-201901_greater-london_00708_
heathrow_qcv-1_1982.csv', skiprows=90)
df83 = pd.read_csv('midas-open_uk-daily-temperature-obs_dv-201901_greater-london_00708_
heathrow_qcv-1_1983.csv', skiprows=90)
df84 = pd.read_csv('midas-open_uk-daily-temperature-obs_dv-201901_greater-london_00708_
heathrow_qcv-1_1984.csv', skiprows=90)
df85 = pd.read_csv('midas-open_uk-daily-temperature-obs_dv-201901_greater-london_00708_
heathrow_qcv-1_1985.csv', skiprows=90)
df86 = pd.read_csv('midas-open_uk-daily-temperature-obs_dv-201901_greater-london_00708_
heathrow_qcv-1_1986.csv', skiprows=90)
df87 = pd.read_csv('midas-open_uk-daily-temperature-obs_dv-201901_greater-london_00708_
heathrow_qcv-1_1987.csv', skiprows=90)
df88 = pd.read_csv('midas-open_uk-daily-temperature-obs_dv-201901_greater-london_00708_
heathrow_qcv-1_1988.csv', skiprows=90)
df89 = pd.read_csv('midas-open_uk-daily-temperature-obs_dv-201901_greater-london_00708_
heathrow_qcv-1_1989.csv', skiprows=90)
df90 = pd.read_csv('midas-open_uk-daily-temperature-obs_dv-201901_greater-london_00708_
heathrow_qcv-1_1990.csv', skiprows=90)
df91 = pd.read_csv('midas-open_uk-daily-temperature-obs_dv-201901_greater-london_00708_
heathrow_qcv-1_1991.csv', skiprows=90)
df92 = pd.read_csv('midas-open_uk-daily-temperature-obs_dv-201901_greater-london_00708_
heathrow_qcv-1_1992.csv', skiprows=90)
df93 = pd.read_csv('midas-open_uk-daily-temperature-obs_dv-201901_greater-london_00708_
heathrow_qcv-1_1993.csv', skiprows=90)
df94 = pd.read_csv('midas-open_uk-daily-temperature-obs_dv-201901_greater-london_00708_
heathrow_qcv-1_1994.csv', skiprows=90)
df95 = pd.read_csv('midas-open_uk-daily-temperature-obs_dv-201901_greater-london_00708_
heathrow_qcv-1_1995.csv', skiprows=90)
df96 = pd.read_csv('midas-open_uk-daily-temperature-obs_dv-201901_greater-london_00708_
heathrow_qcv-1_1996.csv', skiprows=90)
df97 = pd.read_csv('midas-open_uk-daily-temperature-obs_dv-201901_greater-london_00708_
heathrow_qcv-1_1997.csv', skiprows=90)
df98 = pd.read_csv('midas-open_uk-daily-temperature-obs_dv-201901_greater-london_00708_
heathrow_qcv-1_1998.csv', skiprows=90)
df99 = pd.read_csv('midas-open_uk-daily-temperature-obs_dv-201901_greater-london_00708_
heathrow_qcv-1_1999.csv', skiprows=90)
df00 = pd.read_csv('midas-open_uk-daily-temperature-obs_dv-201901_greater-london_00708_
heathrow_qcv-1_2000.csv', skiprows=90)
df01 = pd.read_csv('midas-open_uk-daily-temperature-obs_dv-201901_greater-london_00708_
heathrow_qcv-1_2001.csv', skiprows=90)
df02 = pd.read_csv('midas-open_uk-daily-temperature-obs_dv-201901_greater-london_00708_
heathrow_qcv-1_2002.csv', skiprows=90)
df03 = pd.read_csv('midas-open_uk-daily-temperature-obs_dv-201901_greater-london_00708_
heathrow_qcv-1_2003.csv', skiprows=90)
df04 = pd.read_csv('midas-open_uk-daily-temperature-obs_dv-201901_greater-london_00708_
heathrow_qcv-1_2004.csv', skiprows=90)
df05 = pd.read_csv('midas-open_uk-daily-temperature-obs_dv-201901_greater-london_00708_
heathrow_qcv-1_2005.csv', skiprows=90)
df06 = pd.read_csv('midas-open_uk-daily-temperature-obs_dv-201901_greater-london_00708_
heathrow_qcv-1_2006.csv', skiprows=90)
df07 = pd.read_csv('midas-open_uk-daily-temperature-obs_dv-201901_greater-london_00708_
heathrow_qcv-1_2007.csv', skiprows=90)
df08 = pd.read_csv('midas-open_uk-daily-temperature-obs_dv-201901_greater-london_00708_
heathrow_qcv-1_2008.csv', skiprows=90)
```

```
df09 = pd.read_csv('midas-open_uk-daily-temperature-obs_dv-201901_greater-london_00708_
heathrow_qcv-1_2009.csv', skiprows=90)
df10 = pd.read_csv('midas-open_uk-daily-temperature-obs_dv-201901_greater-london_00708_
heathrow_qcv-1_2010.csv', skiprows=90)
df11 = pd.read_csv('midas-open_uk-daily-temperature-obs_dv-201901_greater-london_00708_
heathrow_qcv-1_2011.csv', skiprows=90)
df12 = pd.read_csv('midas-open_uk-daily-temperature-obs_dv-201901_greater-london_00708_
heathrow_qcv-1_2012.csv', skiprows=90)
df13 = pd.read_csv('midas-open_uk-daily-temperature-obs_dv-201901_greater-london_00708_
heathrow_qcv-1_2013.csv', skiprows=90)
df14 = pd.read_csv('midas-open_uk-daily-temperature-obs_dv-201901_greater-london_00708_
heathrow_qcv-1_2014.csv', skiprows=90)
df15 = pd.read_csv('midas-open_uk-daily-temperature-obs_dv-201901_greater-london_00708_
heathrow_qcv-1_2015.csv', skiprows=90)
df16 = pd.read_csv('midas-open_uk-daily-temperature-obs_dv-201901_greater-london_00708_
heathrow_qcv-1_2016.csv', skiprows=90)
df17 = pd.read_csv('midas-open_uk-daily-temperature-obs_dv-201901_greater-london_00708_
heathrow_qcv-1_2017.csv', skiprows=90)

frames = [df80, df81, df82, df83, df84,df85, df86, df87, df88,
          df89, df90, df91, df92, df93, df94, df95, df96, df97,
          df98, df99, df00, df01, df02, df03, df04, df05, df06,
          df07, df08, df09, df10, df11, df12, df13, df14, df15,
          df16, df17]

df = pd.concat(frames)
df = df.reset_index(drop=True)
df.head(3)
```

Out[2]:

	ob_end_time	id_type	id	ob_hour_count	version_num	met_domain_name	src_id	re
0	1980-01-01 09:00:00	DCNN	5113.0	12.0	1.0	NCM	708.0	
1	1980-01-01 21:00:00	DCNN	5113.0	12.0	1.0	NCM	708.0	
2	1980-01-02 09:00:00	DCNN	5113.0	12.0	1.0	NCM	708.0	

In [3]:

```
df.shape
```

Out[3]:

```
(27798, 22)
```

Comment: Bases on above it looks like the concatenation has worked well as we have roughly the right number of entires for 38 years and equivalent number of days.

As a side note, above is a neat example how to consolidate whatever repetitive tasks when one is given data in small chunks. For convenience, I'm going to save the combined file in a new csv file:

In [0]:

```
df.to_csv('HeathrowWeatherData1980-2017.csv')
```

In [0]:

```
df = pd.read_csv('HeathrowWeatherData1980-2017.csv')
```

3) Preparing and cleaning data

I will be using only time/date (ob_end_time), max and min observation temperatures (max_air_temp and min_air_temp) data. So before I do anything else, I will get rid of the data I don't need:

In [6]:

```
df_temp = df[['ob_end_time','max_air_temp', 'min_air_temp']]
df_temp.tail(3)
```

Out[6]:

	ob_end_time	max_air_temp	min_air_temp
27795	31/12/2017 09:00	13.0	9.0
27796	31/12/2017 21:00	11.6	8.1
27797	end data	NaN	NaN

Let's find out next if I need to format data and check if I have any missing values in the dataset.

In [7]:

```
df_temp.dtypes # checking data types
```

Out[7]:

```
ob_end_time      object
max_air_temp    float64
min_air_temp    float64
dtype: object
```

In [8]:

```
df_temp.isnull().values.any() # checking if there are any missing values
```

Out[8]:

```
True
```

Comment: Temperature formats are what I need, but I need to change time to datetime format. The dataset also seems to have missing values, so I need to investigate closer

In [9]:

```
df_temp.info()
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 27798 entries, 0 to 27797
Data columns (total 3 columns):
 #   Column        Non-Null Count  Dtype  
---  ------        --------------  -----  
 0   ob_end_time   27798 non-null  object 
 1   max_air_temp  27760 non-null  float64
 2   min_air_temp  27760 non-null  float64
dtypes: float64(2), object(1)
memory usage: 651.6+ KB
```

Comment: I seem to have 38 missing datapoints for max and min air temperature. Initially I had 38 smaller datasets that I combined into 1. This is probably related.

In [10]:

```
missing_values = df_temp[df_temp.isnull().any(axis=1)]
missing_values.head(3)
```

Out[10]:

	ob_end_time	max_air_temp	min_air_temp
732	end data	NaN	NaN
1463	end data	NaN	NaN
2194	end data	NaN	NaN

Great, I have uncovered missing values above, which are likely the ending values of individual smaller datasets. It's safe to get rid of them, it won't affect my analysis, so I'll drop them using the function *pd.DataFrame.dropna()*

In [11]:

```
df_temp_clean = df_temp.dropna()
df_temp_clean.info()
```

```
<class 'pandas.core.frame.DataFrame'>
Int64Index: 27760 entries, 0 to 27796
Data columns (total 3 columns):
 #   Column        Non-Null Count  Dtype  
---  ------        --------------  -----  
 0   ob_end_time   27760 non-null  object 
 1   max_air_temp  27760 non-null  float64
 2   min_air_temp  27760 non-null  float64
dtypes: float64(2), object(1)
memory usage: 867.5+ KB
```

Comment: the dataset now seems clean as there are no missing values. What I need to do next is to keep only date values where the temperature is measured between 09:00 and 21:00. This is because I'm interested in temperatures in the daytime only rather that at nighttime (heating/cooling may be switched to minimum at night if noone is at office).

In [12]:

```
indexNames = df_temp_clean[df_temp_clean['ob_end_time'].str.endswith('09:00:00')].index
# Selecting indexes where measurement taken at 09:00
df_temp_clean_day2 = df_temp_clean.drop(indexNames) # Dropping 09:00:00 measurements
df_temp_clean_day2 = df_temp_clean_day2.reset_index(drop=True) # Resetting the index
df_temp_clean_day2.head(3)
```

Out[12]:

	ob_end_time	max_air_temp	min_air_temp
0	1980-01-01 21:00:00	1.0	-2.3
1	1980-01-02 21:00:00	2.3	-3.9
2	1980-01-03 21:00:00	7.7	0.8

In [13]:

```
df_temp_clean_day2.tail(3)
```

Out[13]:

	ob_end_time	max_air_temp	min_air_temp
14242	30/12/2017 21:00	14.4	10.8
14243	31/12/2017 09:00	13.0	9.0
14244	31/12/2017 21:00	11.6	8.1

NOTE: I discovered that date format is inconsistent throughout the dataset. I'll fix it next by also dropping 09:00 measurements from year 2017. I will then format the outlier dates (2017) into standard datetime format in Python.

In [14]:

```
indexNames = df_temp_clean_day2[df_temp_clean_day2['ob_end_time'].str.endswith('09:00'
)].index # Selecting indexes where measurement taken at 09:00
df_temp_clean_day3 = df_temp_clean_day2.drop(indexNames) # Dropping 09:00 measurements
df_temp_clean_day3 = df_temp_clean_day3.reset_index(drop=True) # Resetting the index
df_temp_clean_day3.tail(3)
```

Out[14]:

	ob_end_time	max_air_temp	min_air_temp
13877	29/12/2017 21:00	9.3	5.0
13878	30/12/2017 21:00	14.4	10.8
13879	31/12/2017 21:00	11.6	8.1

In [15]:

```
df_temp_clean_day3.head()
```

Out[15]:

	ob_end_time	max_air_temp	min_air_temp
0	1980-01-01 21:00:00	1.0	-2.3
1	1980-01-02 21:00:00	2.3	-3.9
2	1980-01-03 21:00:00	7.7	0.8
3	1980-01-04 21:00:00	9.7	5.1
4	1980-01-05 21:00:00	8.0	4.1

In [16]:

```
# Importing required libraries
from datetime import datetime

# First getting the slice of dataset where year is 2017
df_temp_clean_day3_2017 = df_temp_clean_day3[df_temp_clean_day3['ob_end_time'].str.ends
with('2017 21:00')]

# Formatting 2017 data into datetime format
df_temp_clean_day3_2017['ob_end_time'] = pd.to_datetime(df_temp_clean_day3_2017['ob_end
_time'], format ='%d/%m/%Y %H:%M')
```

```
/usr/local/lib/python3.6/dist-packages/ipykernel_launcher.py:7: SettingWit
hCopyWarning:
A value is trying to be set on a copy of a slice from a DataFrame.
Try using .loc[row_indexer,col_indexer] = value instead

See the caveats in the documentation: https://pandas.pydata.org/pandas-doc
s/stable/user_guide/indexing.html#returning-a-view-versus-a-copy
  import sys
```

Next I combine again formatted 2017 data slice and the rest, so that formatted numbers will prevail over "old" ones. I also make sure that the "old" dataset gets proper datetime formatting.

In [17]:

```
df_temp_clean_day4 = df_temp_clean_day3_2017.combine_first(df_temp_clean_day3)
df_temp_clean_day4['ob_end_time'] = pd.to_datetime(df_temp_clean_day4['ob_end_time'])
df_temp_clean_day4.head()
```

Out[17]:

	ob_end_time	max_air_temp	min_air_temp
0	1980-01-01 21:00:00	1.0	-2.3
1	1980-01-02 21:00:00	2.3	-3.9
2	1980-01-03 21:00:00	7.7	0.8
3	1980-01-04 21:00:00	9.7	5.1
4	1980-01-05 21:00:00	8.0	4.1

In [18]:

```
df_temp_clean_day4.tail()
```

Out[18]:

	ob_end_time	max_air_temp	min_air_temp
13875	2017-12-27 21:00:00	3.5	1.4
13876	2017-12-28 21:00:00	5.3	0.1
13877	2017-12-29 21:00:00	9.3	5.0
13878	2017-12-30 21:00:00	14.4	10.8
13879	2017-12-31 21:00:00	11.6	8.1

In [19]:

```
df_temp_clean_day4.shape
```

Out[19]:

```
(13880, 3)
```

Comment: I now have datetime as I need, but I need to make a decision on what kind of temperature I'll be using. Ideally, I would use max air temp when it's warm and min air temp when it's cold outside to get an idea how much work our heating/cooling systems need to do (and perhaps define a neutral band when no heating is necessary at all), but as a reasonable simplification for this analysis I will use the average of max and min temperatures.

In [20]:

```
df_temp_clean_day4['AvgDayTemp'] = (df_temp_clean_day4['max_air_temp'] + df_temp_clean_
day4['min_air_temp']) / 2
df_temp_clean_day5 = df_temp_clean_day4[['ob_end_time','AvgDayTemp']] # Keeping only da
te and average temperature columns
df_temp_clean_day5 = df_temp_clean_day5.rename(columns={"ob_end_time": "Date"}) # Renam
ing date column to Date
df_temp_clean_day5.head()
```

Out[20]:

	Date	AvgDayTemp
0	1980-01-01 21:00:00	-0.65
1	1980-01-02 21:00:00	-0.80
2	1980-01-03 21:00:00	4.25
3	1980-01-04 21:00:00	7.40
4	1980-01-05 21:00:00	6.05

Next, I convert my daily data to weekly data as another simplification and also to greatly reduce computing times later on in the project.

In [21]:

```
df_temp_clean_day5["weekday"] = df_temp_clean_day5["Date"].dt.weekday
df_temp_clean_day5.head()
```

Out[21]:

	Date	AvgDayTemp	weekday
0	1980-01-01 21:00:00	-0.65	1
1	1980-01-02 21:00:00	-0.80	2
2	1980-01-03 21:00:00	4.25	3
3	1980-01-04 21:00:00	7.40	4
4	1980-01-05 21:00:00	6.05	5

In [22]:

```
df_temp_clean_day5["AvgWeekTemp"] = df_temp_clean_day5["AvgDayTemp"].rolling(7, center=
True).mean()

df_temp_clean_week = df_temp_clean_day5[df_temp_clean_day5['weekday'] == 3]
df_temp_clean_week2 = df_temp_clean_week.dropna()
df_temp_clean_week3 = df_temp_clean_week2[['Date','AvgWeekTemp']]
df_temp_clean_week3.head(15)
```

Out[22]:

	Date	AvgWeekTemp
9	1980-01-10 21:00:00	2.728571
16	1980-01-17 21:00:00	1.807143
23	1980-01-24 21:00:00	4.242857
30	1980-01-31 21:00:00	6.400000
37	1980-02-07 21:00:00	8.178571
44	1980-02-14 21:00:00	8.421429
51	1980-02-21 21:00:00	7.392857
58	1980-02-28 21:00:00	6.657143
65	1980-03-06 21:00:00	6.728571
72	1980-03-13 21:00:00	6.178571
79	1980-03-20 21:00:00	4.042857
86	1980-03-27 21:00:00	9.392857
93	1980-04-03 21:00:00	9.750000
100	1980-04-10 21:00:00	11.471429
107	1980-04-17 21:00:00	14.071429

Comment: I'm almost done with initial data prep! I now need to change Date column to datetime format for Python to understand it is dealing with time and to be able to perform further time series analysis.

Next, I import datetime library to start converting data from dataframe to a series (I want "date" column to be the "index" essentially)

In [23]:

```
# setting date column to datetime value and setting it as index value
df_temp_clean_week3.set_index('Date', inplace = True)
df_temp_clean_week3.head()
```

Out[23]:

Date	AvgWeekTemp
1980-01-10 21:00:00	2.728571
1980-01-17 21:00:00	1.807143
1980-01-24 21:00:00	4.242857
1980-01-31 21:00:00	6.400000
1980-02-07 21:00:00	8.178571

In [24]:

```
# assigning AvgDayTemp column to a new time series variable
ts_temp = df_temp_clean_week3['AvgWeekTemp']
type(ts_temp)
```

Out[24]:

```
pandas.core.series.Series
```

Comment: Note, that the type of the dataset is now series

In [25]:

```
ts_temp.head()
```

Out[25]:

```
Date
1980-01-10 21:00:00    2.728571
1980-01-17 21:00:00    1.807143
1980-01-24 21:00:00    4.242857
1980-01-31 21:00:00    6.400000
1980-02-07 21:00:00    8.178571
Name: AvgWeekTemp, dtype: float64
```

Next, taking a quick look at the distribution of temperatures

In [26]:

```
ts_temp.describe()
```

Out[26]:

```
count    1982.000000
mean       12.635145
std         5.792467
min        -3.007143
25%         8.042857
50%        12.417857
75%        17.467857
max        28.671429
Name: AvgWeekTemp, dtype: float64
```

Interestingly the max temperature is way above the 75% distribution. This suggests that we have some short term spikes in the dataset, which I can confirm with plotting the data.

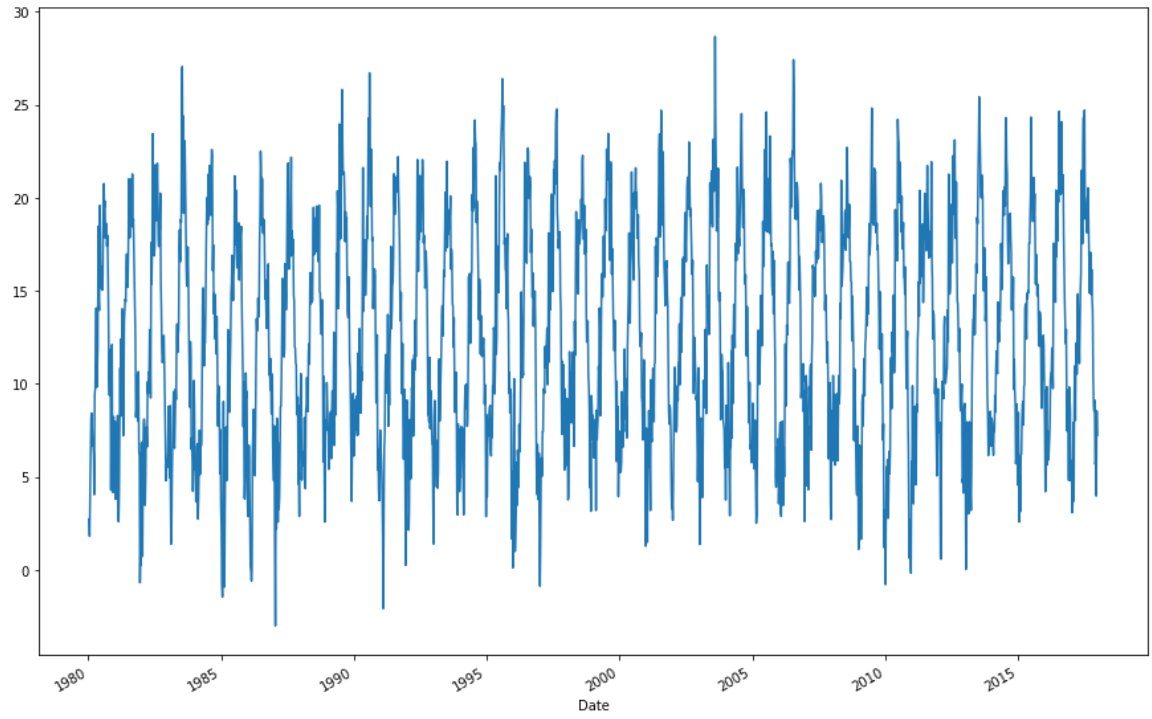
Ideally, I would want the data to be following similar patterns over time and avoid one-off extreme datapoints.

In [27]:

```
plt.figure(figsize=(15,10)) # Defining size of the plot
ts_temp.plot()
```

Out[27]:

```
<matplotlib.axes._subplots.AxesSubplot at 0x7f23fc863588>
```



Comment: extreme outliers confirmed. However, top and low end temperatures seems to be happening regularly (not frequently), which is good for our analysis.

Next, I will work on decomposing data to identify trend, seasonality and residual information.

I will first give it a try with the full dataset, however, based on the above time series chart, I would expect we need slice the data to analyse a more similar (stationary) period.

4) Decomposition of data

I will be using seasonal_decompose library to decompose data into 3:
1) Trend (The general direction data mean values are travelling (like upwards or downwards)
2) Seasonality (Cycles that can be identified repeating in the data)
3) Noise (Random variation that occurs naturally in the data)

In [28]:

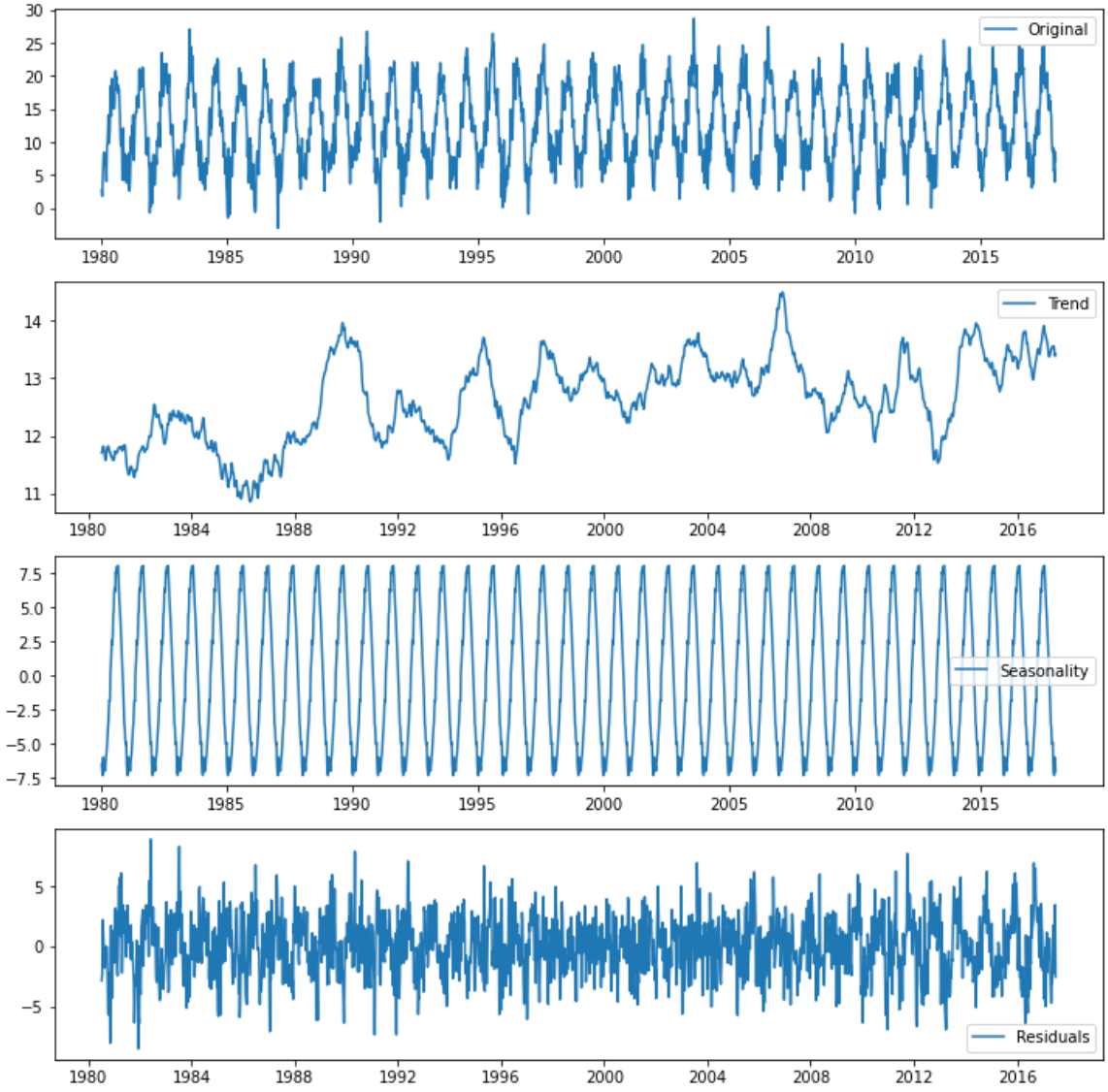
```
from statsmodels.tsa.seasonal import seasonal_decompose
decomposition = seasonal_decompose(ts_temp, freq=52)

# creating 3 new variables of decomposed parts
trend = decomposition.trend
seasonal = decomposition.seasonal
residual = decomposition.resid

plt.figure(figsize=(10,10))

plt.subplot(411)
plt.plot(ts_temp, label = 'Original')
plt.legend(loc = 'best')
plt.subplot(412)
plt.plot(trend, label = 'Trend')
plt.legend(loc = 'best')
plt.subplot(413)
plt.plot(seasonal, label = 'Seasonality')
plt.legend(loc = 'best')
plt.subplot(414)
plt.plot(residual, label = 'Residuals')
plt.legend(loc = 'best')
plt.tight_layout()
```

```
/usr/local/lib/python3.6/dist-packages/statsmodels/tools/_testing.py:19: F
utureWarning: pandas.util.testing is deprecated. Use the functions in the
public API at pandas.testing instead.
  import pandas.util.testing as tm
```



Note on the above: The data seems to have seasonality element and interestingly, starting from around 2013 mark, there has seems to be a trend towards higher average weekly temperatures. Based on the chart II could perhaps say that the past 3-4 years have been slightly warmer on average.

Next: Selecting the length of the dataset, which will be used to build the forecasting model. In order to have a decent size dataset I should have at least few years of data. As a general knowedge, it is also understood that Earth's climate can change over time, so perhaps the weather temperature back in the 1980's doesn't have that much in common with today's weather as for example last year's weather, this is why I will decide to focus this analysis on the last 8 years of available data. Next, I will slice the dataset accordingly.

In [29]:

```
ts_temp_recent = ts_temp[ts_temp.index >= '2010-01-01 21:00:00']
ts_temp_recent.head()
```

Out[29]:

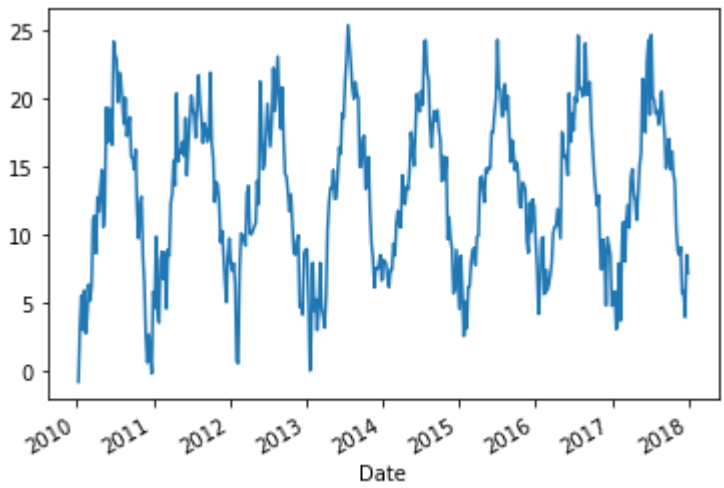
```
Date
2010-01-07 21:00:00   -0.792857
2010-01-14 21:00:00    3.207143
2010-01-21 21:00:00    5.542857
2010-01-28 21:00:00    3.000000
2010-02-04 21:00:00    5.921429
Name: AvgWeekTemp, dtype: float64
```

In [30]:

```
ts_temp_recent.plot()
```

Out[30]:

```
<matplotlib.axes._subplots.AxesSubplot at 0x7f23fc30c470>
```



In [31]:

```
ts_temp_recent.describe()
```

Out[31]:

```
count    417.000000
mean      13.011648
std        5.750166
min       -0.792857
25%        8.542857
50%       13.128571
75%       17.771429
max       25.421429
Name: AvgWeekTemp, dtype: float64
```

Comment: weather temperatures from 2010-2017 compared to the full dataset, have slightly lower high end temperatures and higher low end temperatures.

Next, I run the same decomposing algorithms on 2010-2017 data

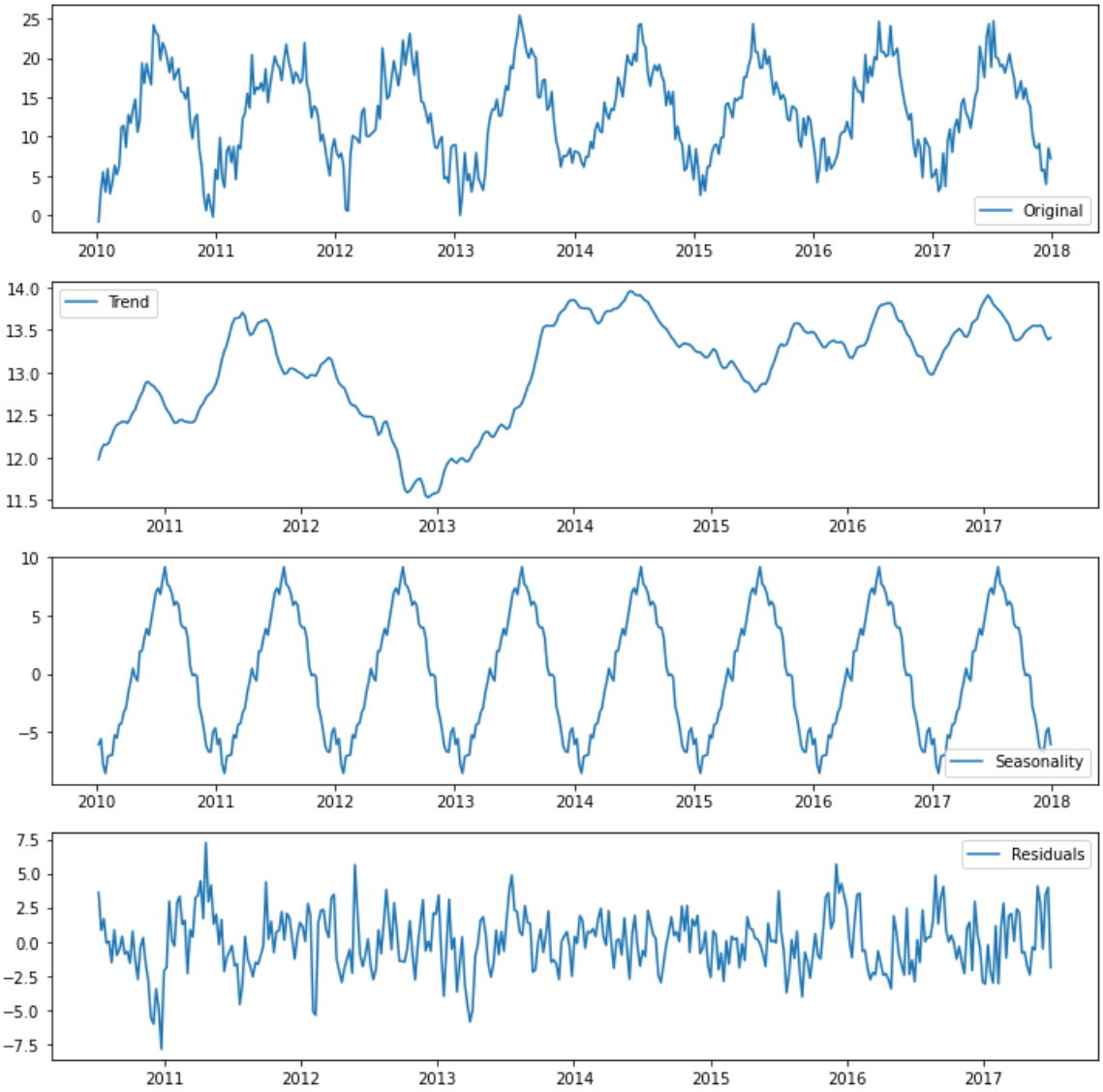
In [32]:

```
# running the same decomposing method for year 2015-2017 data
decomposition = seasonal_decompose(ts_temp_recent, freq=52)

trend = decomposition.trend
seasonal = decomposition.seasonal
residual = decomposition.resid

plt.figure(figsize=(10,10))

plt.subplot(411)
plt.plot(ts_temp_recent, label = 'Original')
plt.legend(loc = 'best')
plt.subplot(412)
plt.plot(trend, label = 'Trend')
plt.legend(loc = 'best')
plt.subplot(413)
plt.plot(seasonal, label = 'Seasonality')
plt.legend(loc = 'best')
plt.subplot(414)
plt.plot(residual, label = 'Residuals')
plt.legend(loc = 'best')
plt.tight_layout()
```


Original
Trend
Seasonality
Residuals

5) Model and transform

In order to build a valid model, I need to make sure that data is stationary - that it's mean, variance and covariance are constant over time. Essentially, this means that the dataset wouldn't change over time. It is needed to make our assumptions work for the model that we're going to use. There are some models that are able to automatically deal with non-stationary data, but this is outside of the scope of this course.

The 0-hypothesis is the following: Daily weather temperature change is not significantly different from 0. According to this, if I determine that daily weather temperature change is not significantly different from 0, the data is stationary. Otherwise, the temperature may have a trend or changing variance, for which I would need to differentiate the dataset and hypothesis test again to make it work fo the model.

1) If the temperature change is significantly different from 0, we reject 0-hypothesis (and data is not stationary)

2) If the temperature change is not significantly different from 0, we accept 0-hypothesis (and data is stationary - this is what I want to achieve)

To assess this I check the **p-value**. Higher than 0.05 p-value means that chances of weather temperature change being different from 0 are **insignificant** (this is what I want!)

Lower than 0.05 p-value means that chances of temperature change being different from 0 are significant.

In [33]:

```
from statsmodels.tsa.stattools import kpss
kpss(ts_temp_recent)
```

```
/usr/local/lib/python3.6/dist-packages/statsmodels/tsa/stattools.py:1685:
FutureWarning: The behavior of using lags=None will change in the next rel
ease. Currently lags=None is the same as lags='legacy', and so a sample-si
ze lag length is used. After the next release, the default will change to
be the same as lags='auto' which uses an automatic lag length selection me
thod. To silence this warning, either use 'auto' or 'legacy'
  warn(msg, FutureWarning)
/usr/local/lib/python3.6/dist-packages/statsmodels/tsa/stattools.py:1711:
InterpolationWarning: p-value is greater than the indicated p-value
  warn("p-value is greater than the indicated p-value", InterpolationWarni
ng)
```

Out[33]:

```
(0.0438307007569328,
 0.1,
 18,
 {'1%': 0.739, '10%': 0.347, '2.5%': 0.574, '5%': 0.463})
```

In this case, the p-value is greater than 0.05 (it is 0.1), meaning I can **accept the null hypothesis** that the data is stationary (constant variance and constant mean).

Next, I will continue with time series analysis by using the transformed dataset. I can proceed without having to transform the data to make it stationary.

6) Building the model

ARIMA Model (p, d, q)

Autoregressive Integrated Moving Average model with exogenous variables. More info on the model is available here: https://www.machinelearningplus.com/time-series/arima-model-time-series-forecasting-python/ (https://www.machinelearningplus.com/time-series/arima-model-time-series-forecasting-python/)

First, I import relevant libraries for ARIMA model

In [0]:

```
# importing relevant libraries for ARIMA model
from sklearn.metrics import mean_squared_error
from statsmodels.tsa.arima_model import ARIMA
import statsmodels.api as sm
```

Next, I define hyperparameters for the model - p, d and q. There are several ways to do it, but let's move on by walking through defining them one by one.

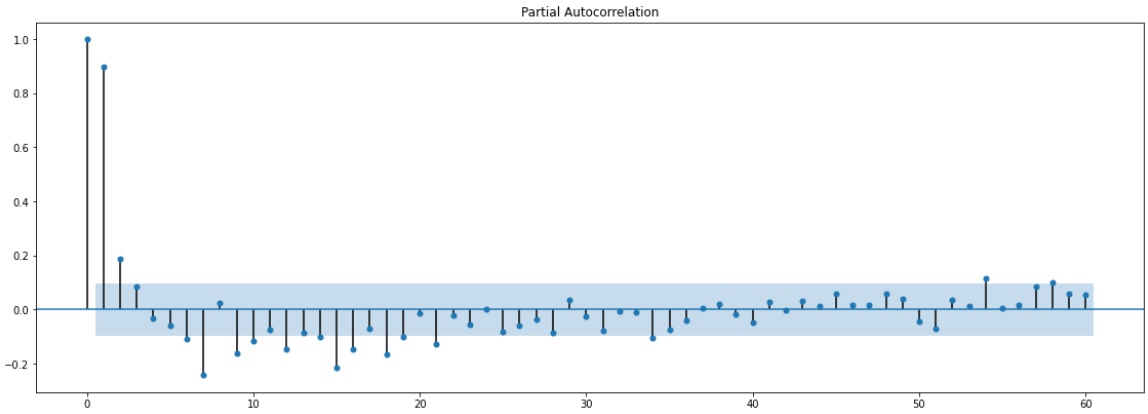
I start with 'd' hyperparameter, which shows how many times I need to differentiate the data in order to have a stationary dataset. I conducted this exercise above, where I concluded that the dataset is already stationary. **This means d = 0**

Now I define p (Autoregressive - AR(p) model), which shows with how many (time)lags is the dataset correlated. I can find it out by plotting the Partial Autocorrelation Function (PACF) as below (PACF as opposed to ACF, also eliminates the correlation between lags themselves).

In [35]:

```
def set_size(w,h, ax=None):
    """ w, h: width, height in inches """
    if not ax: ax=plt.gca()
    l = ax.figure.subplotpars.left
    r = ax.figure.subplotpars.right
    t = ax.figure.subplotpars.top
    b = ax.figure.subplotpars.bottom
    figw = float(w)/(r-l)
    figh = float(h)/(t-b)
    ax.figure.set_size_inches(figw, figh)

sm.graphics.tsa.plot_pacf(ts_temp_recent, lags=60)
set_size(15,5);
```



Comment: the above chart shows that that today's temperature is heavily correlated with today -1 week average temperatures. I can also identify significant correlation with lag 2, lag 6, lag 7 and onwards. Interestingly there is a significant correlation also with lag 54 - this is probably due to seasonality effects because I'm using weekly data where 52 weeks makes a year.

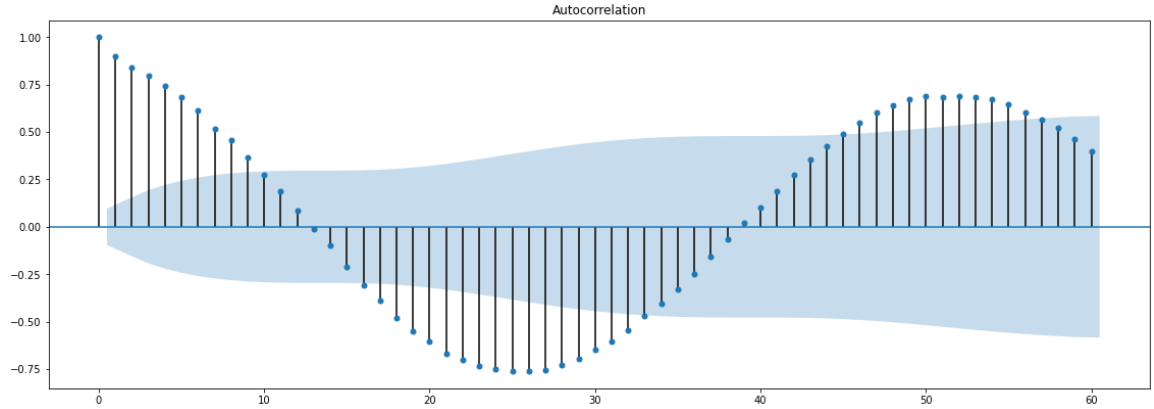
Based on this I define p = 1 (at the very least)

Finally, I need to define q (Moving Average - MA(q) model), which shows how the data is correlated with its own past resduals/errors. For this I use Autocorrelation Function (ACF). In this case correlation between lags is not eliminated (as was with PACF). I see so many significant correlations popping up below because each lag is likely correlated to the their following lag.

In [36]:

```
def set_size(w,h, ax=None):
    """ w, h: width, height in inches """
    if not ax: ax=plt.gca()
    l = ax.figure.subplotpars.left
    r = ax.figure.subplotpars.right
    t = ax.figure.subplotpars.top
    b = ax.figure.subplotpars.bottom
    figw = float(w)/(r-l)
    figh = float(h)/(t-b)
    ax.figure.set_size_inches(figw, figh)

sm.graphics.tsa.plot_acf(ts_temp_recent, lags=60)
set_size(15,5);
```



Comment: the above chart shows that that today's temperature errors are heavily correlated with past weekly errors.

There are also significant negative correlations with around -25 week (half a year) lags. It makes sense that for example summer and winter temperatures are negatively correlated.

Finally, we can see positive correlations with current week's temperature at -52 week timelag (1 year ago), which also is expected.

From that I set **q = 1** for as starting point..

Based on the analysis above I have a starting point of building **ARIMA (p=1, d=0, q=1 model)**. However, I'm taking a step forward to look into Seasonal ARIMA model, because I know that the data is seasonal.

SARIMAX (p, d, q) x (P, D, Q, m) model

Building on the starting blocks above, SARIMAX model introduces the seasonal element to ARIMA, where P, D, Q stand for seasonal hyperparameters and m the number of periods in a season, which is 52 (weeks) for us.

Instead of going ahead and using a p=1, d=0, q=1 model as defined above, I will now introduce **grid search**, which is essentially a very powerful trial and error algorithm, to find the best fitting hyperparameters (in the defined range) for modelling the dataset. To shed more light on this, I first create a grid with n x m dimensions. Then I define the range of our hyper parameters (for example 0 < p < 5) and finally, I populate the grid with all possible combinations of hyperparameters to be tried out. The best model fit will be selected and used in the next steps.

In order to optimise computing time, I will already eliminate non-valid combinations of hyperparameters from the grid. As defined above, these will be parameters which don't satisfy p >= 1, d >= 0, q >= 1 conditions.

I start with importing required libraries for SARIMAX

In [0]:

```
from statsmodels.tsa.statespace.sarimax import SARIMAX
import itertools
import warnings
```

Next, I define a range of parameters which I want the algorithm to try out and create list of all possible combinations of these parameters. I fix the seasonal element to 52 periods.

In [0]:

```
p = d = q = range(0, 4)

# Generate all different combinations of p, q and q triplets
pdq = list(itertools.product(p, d, q))

# Generate all different combinations of seasonal p, q and q triplets
seasonal_pdq = [(x[0], x[1], x[2], 52) for x in list(itertools.product(p, d, q))]
```

Out of all possible p, d, q combinations I only keep/overwrite with the ones satisfying the validity condition:

In [0]:

```
pdq = [(1, 0, 1),
 (1, 0, 2),
 (1, 0, 3),
 (1, 1, 1),
 (1, 1, 2),
 (1, 1, 3),
 (1, 2, 1),
 (1, 2, 2),
 (1, 2, 3),
 (1, 3, 1),
 (1, 3, 2),
 (1, 3, 3),
 (2, 0, 1),
 (2, 0, 2),
 (2, 0, 3),
 (2, 1, 1),
 (2, 1, 2),
 (2, 1, 3),
 (2, 2, 1),
 (2, 2, 2),
 (2, 2, 3),
 (2, 3, 1),
 (2, 3, 2),
 (2, 3, 3),
 (3, 0, 1),
 (3, 0, 2),
 (3, 0, 3),
 (3, 1, 1),
 (3, 1, 2),
 (3, 1, 3),
 (3, 2, 1),
 (3, 2, 2),
 (3, 2, 3),
 (3, 3, 1),
 (3, 3, 2),
 (3, 3, 3)]
```

Finally, I use the below function to try out all hyperparameter combinations (in defined range) to identify the best fit. In order to determine best fit, I use AIC measure (https://en.wikipedia.org/wiki/Akaike_information_criterion (https://en.wikipedia.org/wiki/Akaike_information_criterion)). The lower the result the better.

Below is a code snippet that will be run in **Google COLAB environment** (free online cloud solution for jupyter notebooks) separately, because I don't have enough memory on my local device to perform this task. Using Google COLAB is a very cost effective way to do this analysis, as it radically reduces hardware and time costs, which were a big barrier to entry to such data analysis for many until recently. Performing this task will, however, still take along time, so it will be convenient to leave it running for few hours/days on the background.

In addition to using COLAB, I got in touch with a colleague from the IT department, who recommended me to introduce multi/parallel processing, which makes it possible to try out different hyperparameters at the same time. So if I have more than 1 processing core in my computer hardware, I can split the calculations between them and get to the end result quicker. In the case of Google COLAB, there are 2 processing "cores". Below, I will also try to benefit from it, which should speed up this analysis roughly by 2 times.

In [0]:

```
# import multiprocessing
# from multiprocessing import Process

# warnings.filterwarnings("ignore") # specify to ignore warning messages

# def grid_search():

#   best_result = [0, 0, 10000000]
#   for param in pdq:
#       for param_seasonal in seasonal_pdq:
#           try:
#               mod = sm.tsa.statespace.SARIMAX(ts_temp_recent,
#                                               order=param,
#                                               seasonal_order=param_seasonal,
#                                               enforce_stationarity=False,
#                                               enforce_invertibility=False)

#               results = mod.fit()

#               print('ARIMA{} x {} - AIC: {}'.format(param, param_seasonal, results.ai
c))

#               if results.aic < best_result[2]:
#                   best_result = [param, param_seasonal, results.aic]
#           except:
#               continue

#   print('\nBest Result:', best_result)

# if __name__ == '__main__':
#     p = Process(target=grid_search)
#     p.start()
#     p.join()
```

...After **5 days** of processing on the background in Google COLAB, I have identified that the best result (according to the lowerst AIC measure) is:

ARIMA(3, 1, 1) x (3, 3, 1, 52) - AIC: 569.313219423569

See APPENDIX section at the end of notebook for more details on cell output.

However, even if the the above result is valid by parameters, most of the variables are not significant and when plotting the model overfits and produces unrealistic results when forecasting out of sample. So by filtering the results according to AIC (done in MS Excel for simplicity), fitting various models, to make sure they're not overfitting and that variables are significant, I've come to the conclusion that the best valid model for me for my purpose is:

ARIMA(1, 0, 1) x (3, 1, 0, 52) - AIC: 1698.070

I'll now create this valid best model and see how well this model performs:

In [41]:

```
model = SARIMAX(ts_temp_recent, order= (1,0,1), seasonal_order=(3,1,0,52))
```

```
/usr/local/lib/python3.6/dist-packages/statsmodels/tsa/base/tsa_model.py:1
65: ValueWarning: No frequency information was provided, so inferred frequ
ency W-THU will be used.
  % freq, ValueWarning)
```

In [0]:

```
results = model.fit()
```

In [43]:

```
# summary of the model
results.summary()
```

Out[43]:

Statespace Model Results

Dep. Variable:	AvgWeekTemp	**No. Observations:**	417
Model:	SARIMAX(1, 0, 1)x(3, 1, 0, 52)	**Log Likelihood**	-843.035
Date:	Tue, 28 Apr 2020	**AIC**	1698.070
Time:	10:37:12	**BIC**	1721.470
Sample:	01-07-2010	**HQIC**	1707.369
	- 12-28-2017		
Covariance Type:	opg		

	coef	std err	z	P>|z|	[0.025	0.975]
ar.L1	0.6493	0.084	7.701	0.000	0.484	0.815
ma.L1	-0.2489	0.109	-2.280	0.023	-0.463	-0.035
ar.S.L52	-0.7846	0.061	-12.877	0.000	-0.904	-0.665
ar.S.L104	-0.4715	0.077	-6.111	0.000	-0.623	-0.320
ar.S.L156	-0.1759	0.072	-2.453	0.014	-0.316	-0.035
sigma2	5.3795	0.427	12.594	0.000	4.542	6.217

Ljung-Box (Q):	55.00	**Jarque-Bera (JB):**	0.37
Prob(Q):	0.06	**Prob(JB):**	0.83
Heteroskedasticity (H):	0.72	**Skew:**	-0.07
Prob(H) (two-sided):	0.07	**Kurtosis:**	2.92

Warnings:
[1] Covariance matrix calculated using the outer product of gradients (complex-step).

Comment: The model seems to be fitting the data quite well. According to p-values, **all variables** are significant, which is a great result (this doesn't happen very often).

There is still a chance that the model is overfitting the data, meaning, it would perform well on existing data, but not so well outside of our sample. I will test for this next.

7) Plotting the model against original data

1-step ahead prediction

Plotting the model on real data next to see how well our model performs predicting 1-step (1 week) ahead:

In [0]:

```
pred = results.get_prediction(start=364, dynamic=False)
pred_ci = pred.conf_int()
```

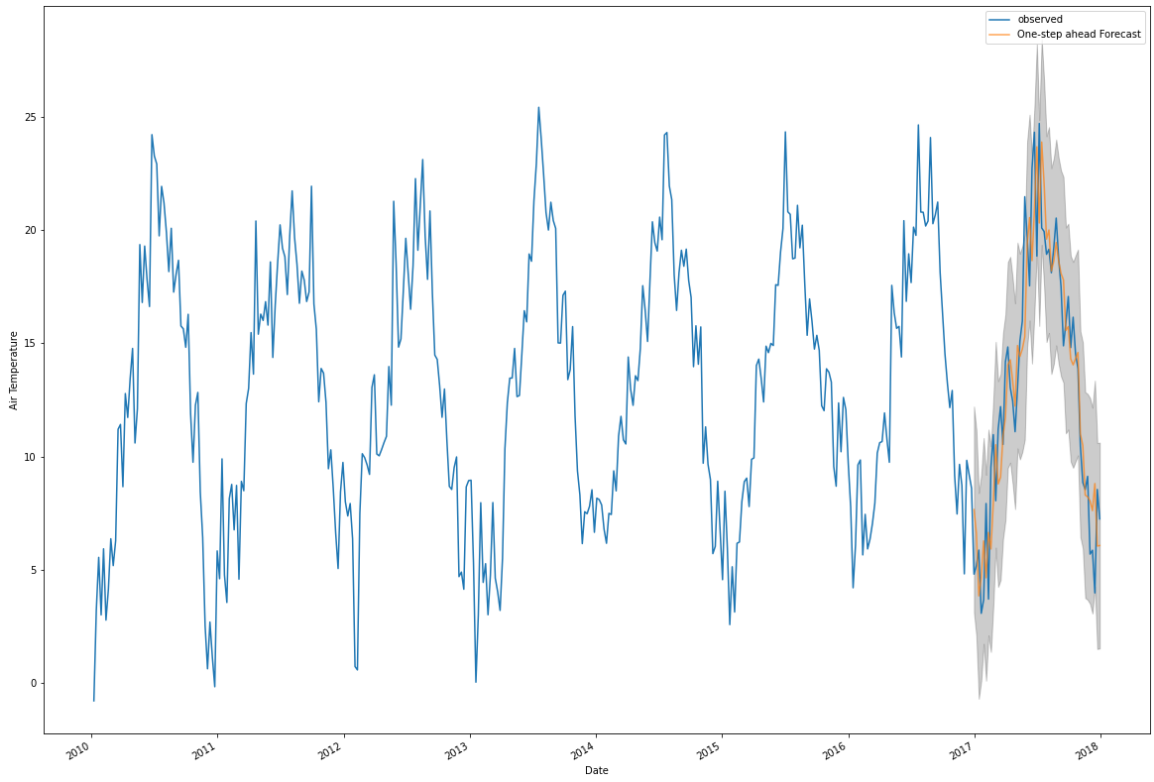
In [45]:

```
ax = ts_temp_recent.plot(label='observed', figsize=(20, 15))
pred.predicted_mean.plot(ax=ax, label='One-step ahead Forecast', alpha=.7)

ax.fill_between(pred_ci.index,
                pred_ci.iloc[:, 0],
                pred_ci.iloc[:, 1], color='k', alpha=.2)

ax.set_xlabel('Date')
ax.set_ylabel('Air Temperature')
plt.legend()

plt.show()
```



Comment: The model seems to be fitting actual data quite well according to the chart. Next, I assess the model's accuracy by calculating mean squared error (MSE)

MSE of the 1-step prediction model:

In [46]:

```
# Extracting the predicted and true values of our time series
ts_temp_recent_forecasted = pred.predicted_mean
ts_temp_recent_truth = ts_temp_recent[364:]

# Computing the mean square error
mse = ((ts_temp_recent_forecasted - ts_temp_recent_truth) ** 2).mean()
print('The Mean Squared Error of our forecasts is {}'.format(round(mse, 2)))
```

```
The Mean Squared Error of our forecasts is 5.7
```

Dynamic prediction

Next, plotting the model on real data to see how well the model performs dynamically. That is, predicting temperatures for year 2017 based on previous 2 years (2015 an 2016) data: **1-year ahead dynamic predition**

In [0]:

```
pred_dynamic = results.get_prediction(start=364, dynamic=True, full_results=True)
pred_dynamic_ci = pred_dynamic.conf_int()
```

In [48]:

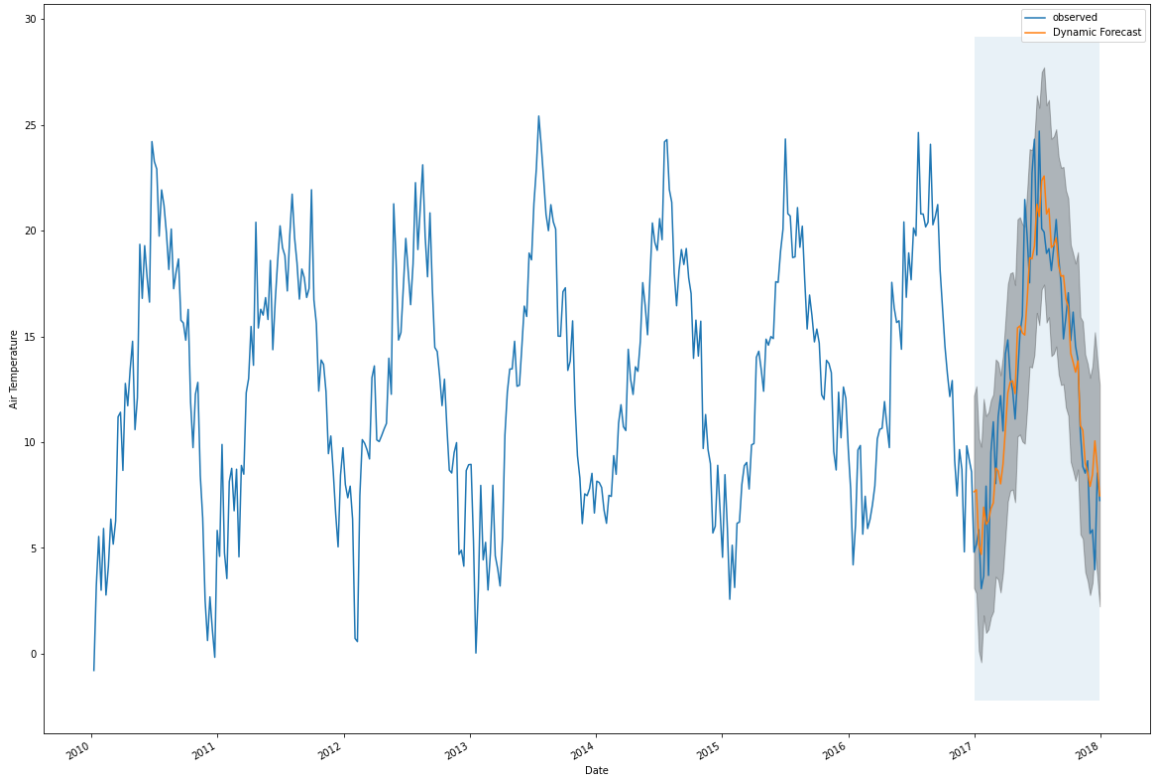
```
ax = ts_temp_recent.plot(label='observed', figsize=(20, 15))
pred_dynamic.predicted_mean.plot(label='Dynamic Forecast', ax=ax)

ax.fill_between(pred_dynamic_ci.index,
                pred_dynamic_ci.iloc[:, 0],
                pred_dynamic_ci.iloc[:, 1], color='k', alpha=.25)

ax.fill_betweenx(ax.get_ylim(), pd.to_datetime('2017-01-01'), ts_temp_recent.index[-1],
                 alpha=.1, zorder=-1)

ax.set_xlabel('Date')
ax.set_ylabel('Air Temperature')

plt.legend()
plt.show()
```



Comment: Again, the model seems to be fitting actual data quite well according to the chart, but when looking closely the confidence interval zones have expanded and the prediction is missing out on more of the spikes in temperature change compared to 1-step ahead prediction. This is expected, because previously I was predicting 1 step ahead, whereas, now up to 52 steps ahead. Next, I again assess the model's accuracy by calculating mean squared error (MSE), which I also expect to be higher than for 1-step ahead prediction.

MSE of the 1-year dynamic prediction model:

In [49]:

```
# Extract the predicted and true values of our time series
ts_temp_recent_forecasted = pred_dynamic.predicted_mean
ts_temp_recent_truth = ts_temp_recent[364:]

# Compute the mean square error
mse = ((ts_temp_recent_forecasted - ts_temp_recent_truth) ** 2).mean()
print('The Mean Squared Error of our forecasts is {}'.format(round(mse, 2)))
```

```
The Mean Squared Error of our forecasts is 6.06
```

8) Forecasting

Out of sample prediction

In this section I will forecast weather temperatures in London Heathrow in the future for 1 year ahead. I plot the prediction on the same chart with training data. Essentially, this is a dynamic prediction, but starting from the end of the dataset.

In [0]:

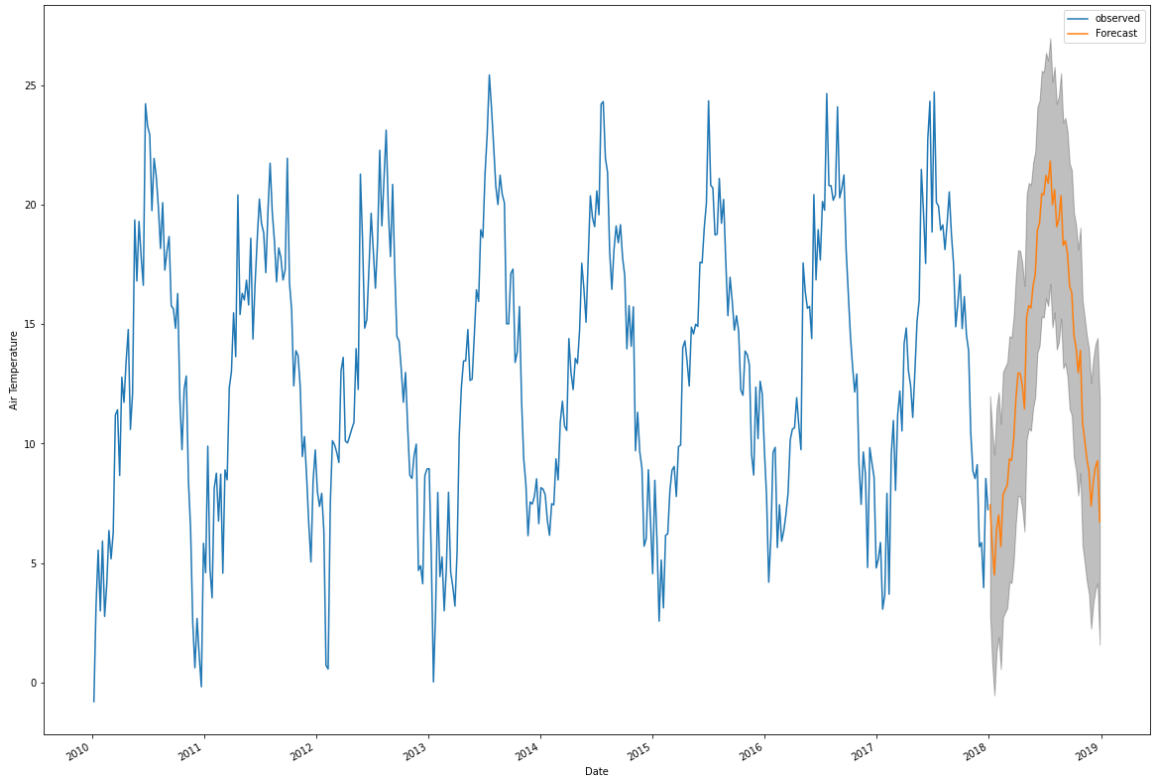
```
# Get forecast 52 steps ahead in future
pred_uc = results.get_forecast(steps=52)

# Get confidence intervals of forecasts
pred_ci = pred_uc.conf_int()
```

In [51]:

```
ax = ts_temp_recent.plot(label='observed', figsize=(20, 15))
pred_uc.predicted_mean.plot(ax=ax, label='Forecast')
ax.fill_between(pred_ci.index,
                pred_ci.iloc[:, 0],
                pred_ci.iloc[:, 1], color='k', alpha=.25)
ax.set_xlabel('Date')
ax.set_ylabel('Air Temperature')

plt.legend()
plt.show()
```



Comment: 1-year ahead forecast looks reasonable by the looks of it compared to past temperature fluctuations.

9) Evaluation and conclusion

Summary findings:
Air temperature change at Heathrow has very strong seasonal effects, as expected, but the temperature has also been relatively stable over the years for a given time in a year. Based on the forecast, air temperature is also expected to be relatively stable in the future (in 2018).

For example, based on the forecast chart, I can say with 95% confidence that in the peak summer next year (2018), the temperature will be between 26C and 15C between 9am and 9pm.

In peak winter time in 2018, the air temperature will be between 15C and 2C with 95% certainty.

Real life implications:
These findings are important, because it makes a case for dynamic budgeting for heating and cooling within my company. Having confidence intervals for temperature fluctuations is helpful, because it will enable the firm to plan cooling/heating costs for these periods with adequate buffers.

For example in winter of 2018 UBS could budget its heating costs for the bulding based on 0C* minimum temperature and we could do this for each season and even each month.

One could argue that these numbers are based on past data and as such may not be very accurate, but given the consistency of temperatures across years, at the very least, I will be able to start a conversation on this topic and recommend our Finance and Real Estate management teams to introduce precise monthly cost planning initiatives for office heating/cooling purposes.

Dynamic cost planning is important, because precise month by month planning will enable the firm to plan better its liquidity position by optimising cash buffers and to allocate money to areas within the bank, where it is needed most at each point in time.

The beauty of this analysis in Python is that it could be replicated very quickly for various locations. For example, for our offices in India, Singapore and Hong Kong, where cooling costs in Summer make up a more significant part of overall costs.

Additionally, this model could also be expanded to other departments in our company, which could find it useful to analyse financials of companies where heating costs play a major role like greenhouse farms or server warehouses.

Suggestions for further analysis:
This analysis presents a step-by-step guide of the process of time series analysis. It is relatively high level, limited by available data and could be definitely improved in accuracy. As for further improvements I would recommend the following:

1) Conduct the same analysis with up to 2019 data
2) Define optimal temperature in required building and translate outside air temperature fluctuations into approximate heating/cooling costs to make the analysis more tangible
3) Secondly, expand on SARIMAX model to potentially improve model accuracy by using other models like Artificial Neural Networks (ANN) or Prophet model developed by Facebook's data team.

APPENDIX

Grid Search input and output results that were run separately:

In [0]:

```
# **In:**

# import multiprocessing
# from multiprocessing import Process

# warnings.filterwarnings("ignore") # specify to ignore warning messages

# def grid_search():

#   best_result = [0, 0, 10000000]
#   for param in pdq:
#       for param_seasonal in seasonal_pdq:
#           try:
#               mod = sm.tsa.statespace.SARIMAX(ts_temp_recent,
#                                               order=param,
#                                               seasonal_order=param_seasonal,
#                                               enforce_stationarity=False,
#                                               enforce_invertibility=False)

#               results = mod.fit()

#               print('ARIMA{} x {} - AIC: {}'.format(param, param_seasonal, results.ai
c))

#               if results.aic < best_result[2]:
#                   best_result = [param, param_seasonal, results.aic]
#           except:
#               continue

#   print('\nBest Result:', best_result)

# if __name__ == '__main__':
#     p = Process(target=grid_search)
#     p.start()
#     p.join()
```

In [0]:

```
# **Out:**

# ARIMA(1, 0, 1) x (0, 0, 0, 52) - AIC  1919.206716
# ARIMA(1, 0, 1) x (0, 0, 1, 52) - AIC  1665.411595
# ARIMA(1, 0, 1) x (0, 0, 2, 52) - AIC  1429.298031
# ARIMA(1, 0, 1) x (0, 1, 0, 52) - AIC  1817.559819
# ARIMA(1, 0, 1) x (0, 1, 1, 52) - AIC  1406.055719
# ARIMA(1, 0, 1) x (0, 1, 2, 52) - AIC  9834.958631
# ARIMA(1, 0, 1) x (0, 2, 0, 52) - AIC  1904.496894
# ...
# ARIMA(3, 1, 1) x (3, 2, 0, 52) - AIC  755.1597026
# ARIMA(3, 1, 1) x (3, 2, 1, 52) - AIC  752.2618436
# ARIMA(3, 1, 1) x (3, 3, 1, 52) - AIC  569.3132194
# ARIMA(3, 1, 2) x (3, 2, 0, 52) - AIC  757.131687
# ARIMA(3, 1, 2) x (3, 2, 1, 52) - AIC  754.2377963
# ARIMA(3, 1, 2) x (3, 3, 1, 52) - AIC  570.4193534

# Best Result: [(3, 1, 1), (3, 3, 1, 52), 569.313219423569]
```