climate-change-hackathon-1

April 3, 2024

0.1 Libraries

[1]:
```
#Import Libraries

import pandas as pd
import numpy as np

#Data Visualization
import seaborn as sns
import matplotlib.pyplot as plt
%matplotlib inline
plt.style.use('seaborn-dark')

#DateTime
import datetime as dt

#Models
from sklearn.linear_model import LinearRegression
from lightgbm import LGBMRegressor

#Sklearn
from sklearn.metrics import mean_squared_error
from sklearn.model_selection import KFold, cross_val_score, StratifiedKFold,␣
↪train_test_split
from sklearn.preprocessing import StandardScaler

#Time to run Program
import time
```

```
<IPython.core.display.HTML object>
```

0.2 Load Data

[2]:
```
def load_data():
    '''
    Function to Load the Train, Test and Submission Data

    returns: train, test, submission dataframes
```

```
    '''

    train = pd.read_csv('../input/d2c-climate-change-hackathon/new_train.csv')
    test = pd.read_csv('../input/d2c-climate-change-hackathon/new_test.csv')
    submission = pd.read_csv('../input/d2c-climate-change-hackathon/
 ↪sample_submission.csv')

    return train, test, submission
```

[3]:
```
#Declare Traget and Feature
TARGET = 'temp'
feature = ['date']
```

[4]:
```
train, test, submission = load_data()
```

[5]:
```
train.head()
```

[5]:

	date	temp
0	01-01-1980	4.16
1	02-01-1980	4.06
2	03-01-1980	7.12
3	04-01-1980	9.23
4	05-01-1980	3.20

[6]:
```
test.head()
```

[6]:

	date
0	01-01-2011
1	02-01-2011
2	03-01-2011
3	04-01-2011
4	05-01-2011

[7]:
```
submission.head()
```

[7]:

	prediction
0	5.57
1	5.57
2	5.57
3	5.57
4	5.57

0.3 Functions

[8]:
```python
#RMSE
def rmse():
  y_pred = train.iloc[10000:11322, 2]
  y = train.iloc[10000:11322, 0]
  metric = np.sqrt(mean_squared_error(y, y_pred))
  print(f"RMSE of Data is: {metric}")

#Hackathon Metric
def predict(model, model_features):
  pred_train = model.predict(X_train[model_features])
  pred_val = model.predict(X_val[model_features])

  print(f"Train RMSE = {np.sqrt(mean_squared_error(y_train, pred_train))}")
  print(f"Test RMSE = {np.sqrt(mean_squared_error(y_val, pred_val))}")

def run_gradient_boosting(clf, fit_params, train, test, features):
  N_SPLITS = 5
  oofs = np.zeros(len(train))
  preds = np.zeros((len(test)))

  target = train[TARGET]

  folds = StratifiedKFold(n_splits = N_SPLITS)
  stratified_target = pd.qcut(train[TARGET], 10, labels = False,␣
↪duplicates='drop')

  feature_importances = pd.DataFrame()

  for fold_, (trn_idx, val_idx) in enumerate(folds.split(train,␣
↪stratified_target)):
    print(f'\n------------- Fold {fold_ + 1} -------------')

    ### Training Set
    X_trn, y_trn = train[features].iloc[trn_idx], target.iloc[trn_idx]

    ### Validation Set
    X_val, y_val = train[features].iloc[val_idx], target.iloc[val_idx]

    ### Test Set
    X_test = test[features]

    scaler = StandardScaler()
    _ = scaler.fit(X_trn)

    X_trn = scaler.transform(X_trn)
```

```
    X_val = scaler.transform(X_val)
    X_test = scaler.transform(X_test)

    _ = clf.fit(X_trn, y_trn, eval_set = [(X_val, y_val)], **fit_params)

    fold_importance = pd.DataFrame({'fold': fold_ + 1, 'feature': features, 'importance': clf.feature_importances_})
    feature_importances = pd.concat([feature_importances, fold_importance], axis=0)

    ### Instead of directly predicting the classes we will obtain the probability of positive class.
    preds_val = clf.predict(X_val)
    preds_test = clf.predict(X_test)

    fold_score = metric(y_val, preds_val)
    print(f'\nRMSE score for validation set is {fold_score}')

    oofs[val_idx] = preds_val
    preds += preds_test / N_SPLITS


  oofs_score = metric(target, oofs)
  print(f'\n\nRMSE for oofs is {oofs_score}')

  feature_importances = feature_importances.reset_index(drop = True)
  fi = feature_importances.groupby('feature')['importance'].mean().sort_values(ascending = False)[:20][::-1]
  fi.plot(kind = 'barh', figsize=(12, 6))

  return oofs, preds, fi

def metric(y_true, y_pred):
  return np.sqrt(mean_squared_error(y_true, y_pred))

def download_preds(preds_test, file_name = 'hacklive_sub.csv'):

  ## 1. Setting the target column with our obtained predictions
  submission['prediction'] = preds_test

  ## 2. Saving our predictions to a csv file

  submission.to_csv(file_name, index = False)

  ## 3. Downloading and submitting the csv file
  from google.colab import files
  files.download(file_name)
```

```python
#Download Submission File
def download(model, model_features, file_name = 'prophet.csv'):

  pred_test = model.predict(model_features)

  #Setting the target column with our obtained predictions
  submission['prediction'] = pred_test

  #Saving our predictions to a csv file
  submission.to_csv(file_name, index = False)

  #Downloadingthe csv file
  files.download(file_name)

def join_df(train, test):

  df = pd.concat([train, test], axis=0).reset_index(drop = True)
  features = [c for c in df.columns if c not in [feature, TARGET]]
  df[TARGET] = df[TARGET].apply(lambda x: np.log1p(x))

  return df, features

def split_df_and_get_features(df, train_nrows):

  train, test = df[:train_nrows].reset_index(drop = True), df[train_nrows:].
↪reset_index(drop = True)
  features = [c for c in train.columns if c not in [feature, TARGET]]

  return train, test, features
```

0.4 EDA and Data Preprocessing

[9]:
```python
#Combine Train and Test Dataframe
df, features = join_df(train, test)
```

[10]:
```python
df.head()
```

[10]:

	date	temp
0	01-01-1980	1.640937
1	02-01-1980	1.621366
2	03-01-1980	2.094330
3	04-01-1980	2.325325
4	05-01-1980	1.435085

0.4.1 Data Details

[11]:
```
print(f"train.shape: {train.shape}")
print(f"test.shape: {test.shape}")
```

```
train.shape: (11323, 2)
test.shape: (3561, 1)
```

[12]:
```
train.describe()
```

[12]:

	temp
count	11323.000000
mean	15.573259
std	7.877191
min	-5.110000
25%	8.390000
50%	15.990000
75%	22.055000
max	32.390000

[13]:
```
#Check Datatypes
train.dtypes
```

[13]:
```
date     object
temp    float64
dtype: object
```

Datatype of `date` is incorrect. It should be datetime. We will correct it in a later stage.

0.4.2 Null Values

[14]:
```
print(f"Train Null Value Count: {train.isnull().sum()}")
print(f"Test Null Value Count: {test.isnull().sum()}")
```

```
Train Null Value Count: date    0
temp    0
dtype: int64
Test Null Value Count: date    0
dtype: int64
```

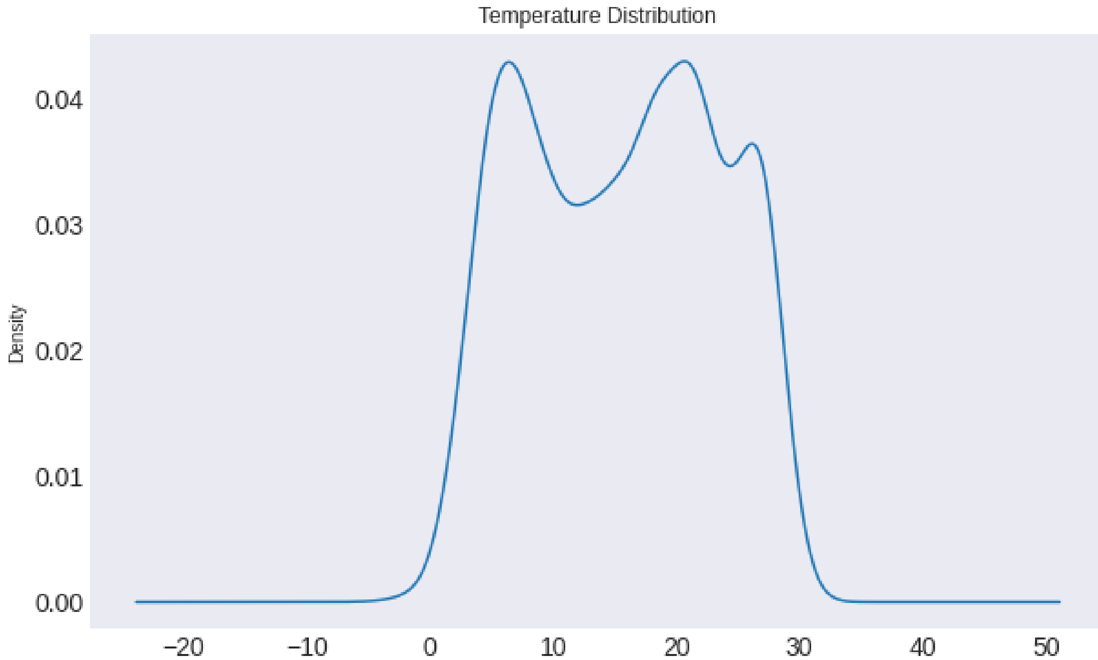
Thus, Training Data does not have any null values.

0.4.3 Target Distribution

Let us check the distribution of the TARGET i.e., Temperature.

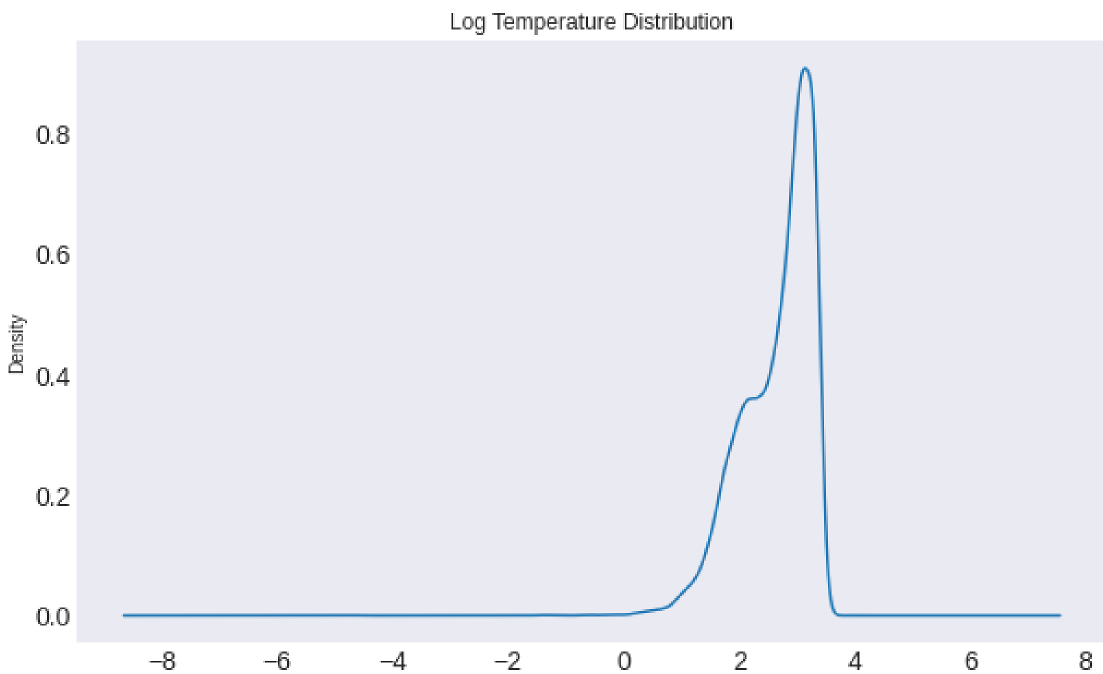
[15]:
```
#Temperature Distribution
train[TARGET].plot(kind = 'density', title = 'Temperature Distribution',␣
 ↪fontsize=14, figsize=(10, 6))
```

[15]: <AxesSubplot:title={'center':'Temperature Distribution'}, ylabel='Density'>



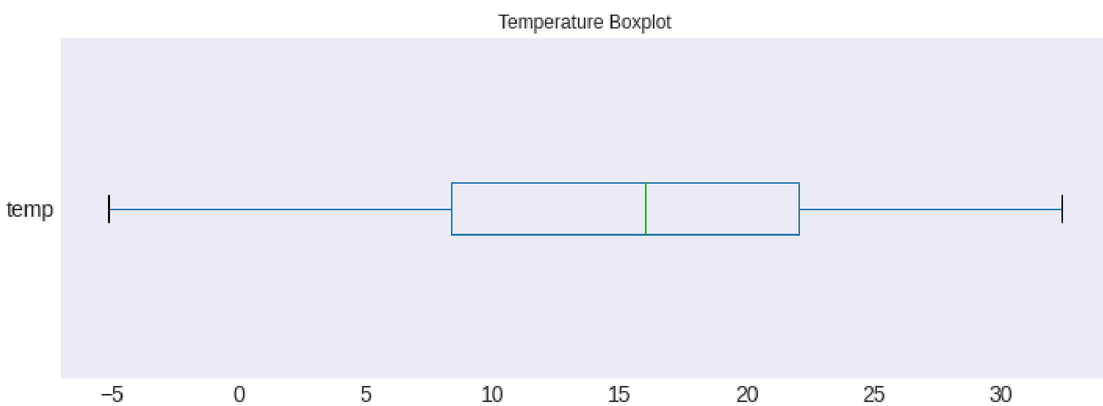
[16]:
```
#Log Temperature Distribution
_ = pd.Series(np.log1p(train[TARGET])).plot(kind = 'density', title = 'Log␣
↪Temperature Distribution', fontsize=14, figsize=(10, 6))
```

```
/opt/conda/lib/python3.7/site-packages/pandas/core/arraylike.py:274:
RuntimeWarning: invalid value encountered in log1p
  result = getattr(ufunc, method)(*inputs, **kwargs)
```



[17]:
```
#Temperature Boxplot
train[TARGET].plot(kind = 'box', vert=False, figsize=(12, 4), title =␣
↪'Temperature Boxplot', fontsize=14)
```

[17]: <AxesSubplot:title={'center':'Temperature Boxplot'}>

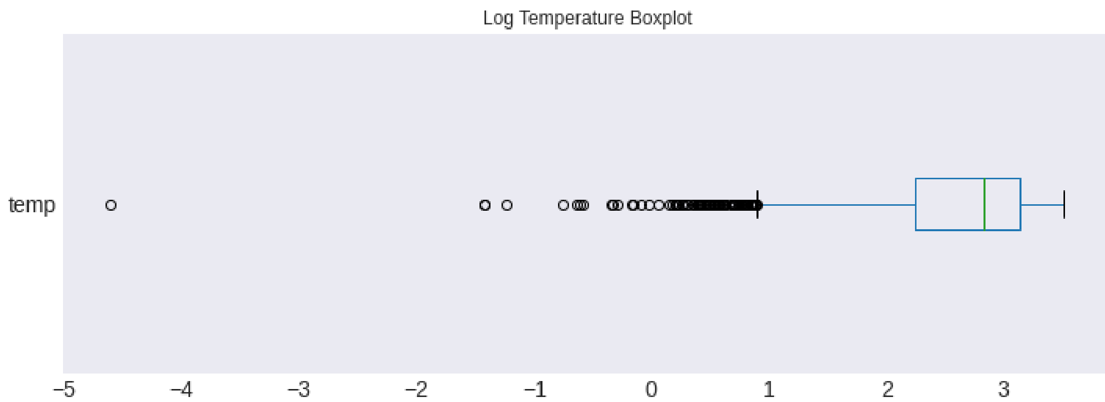


[18]:
```
#Log Temperature BoxPlot
pd.Series(np.log1p(train[TARGET])).plot(kind = 'box', vert=False, figsize=(12,␣
↪4), title = 'Log Temperature Boxplot', fontsize=14)
```

/opt/conda/lib/python3.7/site-packages/pandas/core/arraylike.py:274:

```
RuntimeWarning: invalid value encountered in log1p
  result = getattr(ufunc, method)(*inputs, **kwargs)
```

[18]:
```
<AxesSubplot:title={'center':'Log Temperature Boxplot'}>
```



0.5 Date Feature

Now we shall create some features using the `date` column.

[19]:
```
#Convert `date` column datatype to `datetime`
df['date'] = pd.to_datetime(df['date'])

df.dtypes
```

[19]:
```
date    datetime64[ns]
temp           float64
dtype: object
```

[20]:
```
print(f"Train Null Value Count: {train.isnull().sum()}")
print(f"Test Null Value Count: {test.isnull().sum()}")
```

```
Train Null Value Count: date    0
temp    0
dtype: int64
Test Null Value Count: date    0
dtype: int64
```

[21]:
```
#Make basic datetime features
# df['day_of_week'] = df['date'].dt.dayofweek
df['year'] = df['date'].dt.year
df['month'] = df['date'].dt.month
df['week'] = df['date'].dt.isocalendar().week
```

```
#Get Train and Test sets from df
train, test, features = split_df_and_get_features(df, train.shape[0])

#Define the features
features = [c for c in df.columns if c not in [feature, TARGET]]
features = features[1:]
features
```

[21]:
```
['year', 'month', 'week']
```

There are many more functions in `datetime` library which you can try out for yourself. Check the documentation for more such functions.

[22]:
```
df.head()
```

[22]:
```
         date      temp  year  month  week
0  1980-01-01  1.640937  1980      1     1
1  1980-02-01  1.621366  1980      2     5
2  1980-03-01  2.094330  1980      3     9
3  1980-04-01  2.325325  1980      4    14
4  1980-05-01  1.435085  1980      5    18
```

[23]:
```
train.fillna(np.mean(train['temp']), inplace=True)
```

0.6 Model

[24]:
```
#Declare Features and Target from Training Dataset
X = train[features]
y = train[TARGET]

#Split Training and Validation Datasets
X_train, X_val, y_train, y_val = train_test_split(X, y, test_size=0.3, random_state = 42)
```

[25]:
```
X.shape, y.shape
```

[25]:
```
((11323, 3), (11323,))
```

We begin with a simple Linear Regression baseline model and then move ahead with more complex algorithms.

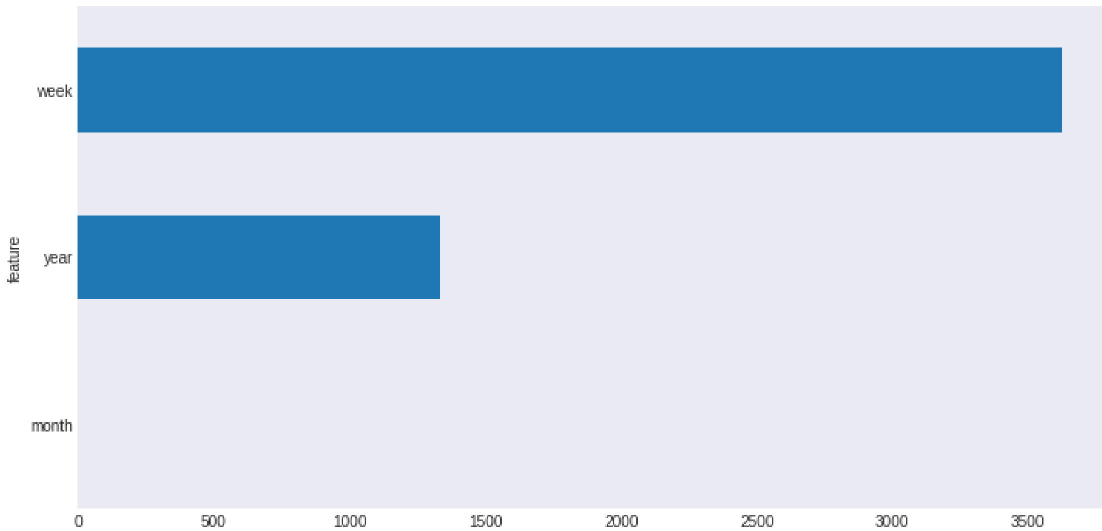
[26]:
```
#Linear Regression
model = LinearRegression()

model.fit(X_train[features], y_train)

predict(model, features)
```

```
Train RMSE = 0.5928992504796073
Test RMSE = 0.6142380849281177
```

[27]:
```python
model = LGBMRegressor(n_estimators = 5000,
                      learning_rate = 0.01,
                      colsample_bytree = 0.76,
                      metric = 'None',
                      )
fit_params = {'verbose': 300, 'early_stopping_rounds': 200, 'eval_metric':␣
↪'rmse'}

lgb_oofs, lgb_preds, fi = run_gradient_boosting(clf = model, fit_params =␣
↪fit_params, train = train, test = test, features = features)
```

```
------------- Fold 1 -------------
Training until validation scores don't improve for 200 rounds
Early stopping, best iteration is:
[46]    valid_0's rmse: 0.516931

RMSE score for validation set is 0.5169311565544552

------------- Fold 2 -------------
Training until validation scores don't improve for 200 rounds
[300]   valid_0's rmse: 0.549336
Early stopping, best iteration is:
[154]   valid_0's rmse: 0.505732

RMSE score for validation set is 0.5057316383185315

------------- Fold 3 -------------
Training until validation scores don't improve for 200 rounds
[300]   valid_0's rmse: 0.478351
Early stopping, best iteration is:
[126]   valid_0's rmse: 0.458806

RMSE score for validation set is 0.45880612894406014

------------- Fold 4 -------------
Training until validation scores don't improve for 200 rounds
[300]   valid_0's rmse: 0.459342
Early stopping, best iteration is:
[262]   valid_0's rmse: 0.458651

RMSE score for validation set is 0.45865072074994945

------------- Fold 5 -------------
Training until validation scores don't improve for 200 rounds
```

```
[300]	valid_0's rmse: 0.429767
Early stopping, best iteration is:
[238]	valid_0's rmse: 0.42859

RMSE score for validation set is 0.4285897355787363


RMSE for oofs is 0.47488107436049726
```



0.6.1 Preprocess Data

[28]:
```python
#Load Data
train, test, submission = load_data()
```

[29]:
```python
#Convert `date` column to datetime
train.date = pd.to_datetime(train.date)
```

[ ]:
```python
#Set `date` as index
train.set_index('date', inplace = True)
```

[ ]:
```python
train.head()
```

[30]:
```python
train.describe()
```

[30]:

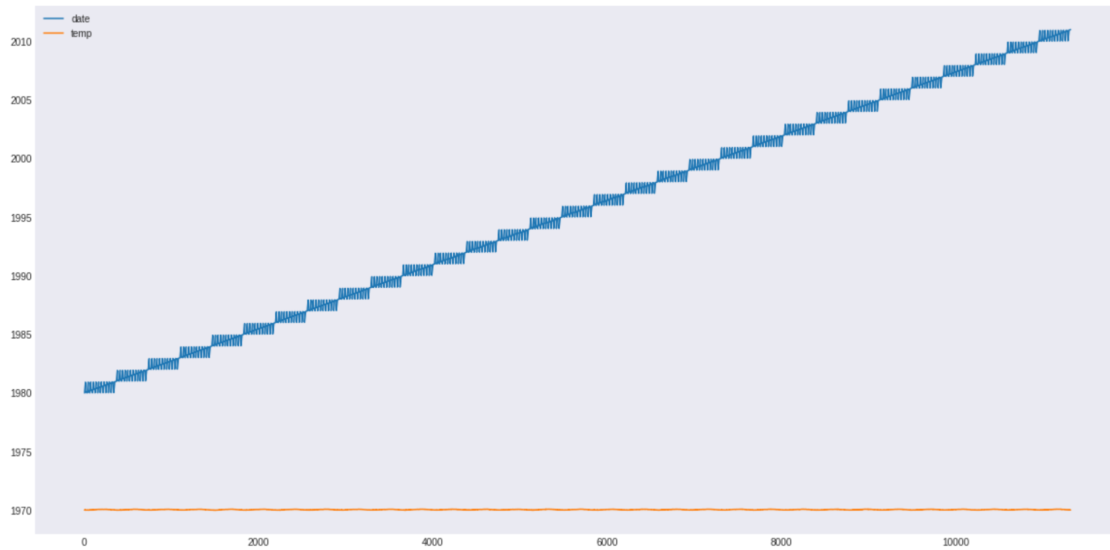
	temp
count	11323.000000
mean	15.573259
std	7.877191
min	-5.110000

```
25%         8.390000
50%        15.990000
75%        22.055000
max        32.390000
```

0.6.2 Visualize Data

[31]:
```
train.plot(figsize = (20, 10))
```

[31]: <AxesSubplot:>



From the plot we observe that the data is seasonal.

0.6.3 Make Data Stationary

To check if the data is stationary, we perform the `adfuller` test.

[32]:
```
#Import adfuller test
from statsmodels.tsa.stattools import adfuller
```

[33]:
```
# test_result = adfuller(train.temp)
```

[34]:
```
#H0: It is not stationary
#H1: It is stationary

def adfuller_test(temp):
    result=adfuller(temp)
    labels = ['ADF Test Statistic','p-value','#Lags Used','Number of␣
 ↪Observations Used']
```

```
    for value,label in zip(result,labels):
        print(label+' : '+str(value) )
    if result[1] <= 0.05:
        print("strong evidence against the null hypothesis(Ho), reject the null␣
↪hypothesis. Data has no unit root and is stationary")
    else:
        print("weak evidence against null hypothesis, time series has a unit␣
↪root, indicating it is non-stationary ")
```

[35]:
```
adfuller_test(train.temp)
```

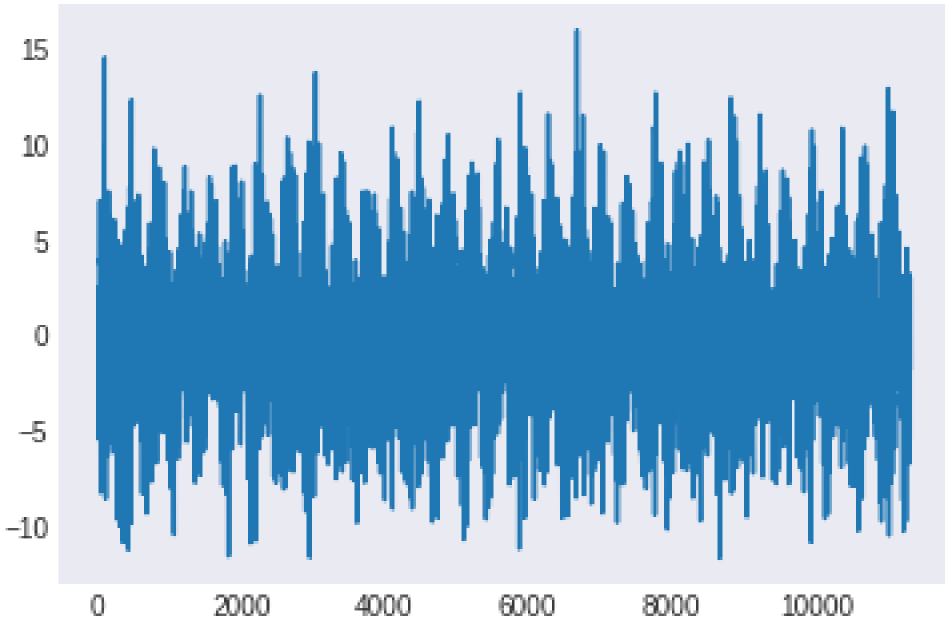
```
ADF Test Statistic : -10.24507580260355
p-value : 4.6405019261563845e-18
#Lags Used : 40
Number of Observations Used : 11282
strong evidence against the null hypothesis(Ho), reject the null hypothesis.
Data has no unit root and is stationary
```

Since data is already stationary, we do not need to perform differencing and can set `d=0` directly.

[36]:
```
train['Seasonal First Difference']=train['temp']-train['temp'].shift(12)␣
↪#Because 1 year has 12 months

## Again test dickey fuller test
adfuller_test(train['Seasonal First Difference'].dropna())

train['Seasonal First Difference'].plot()
```

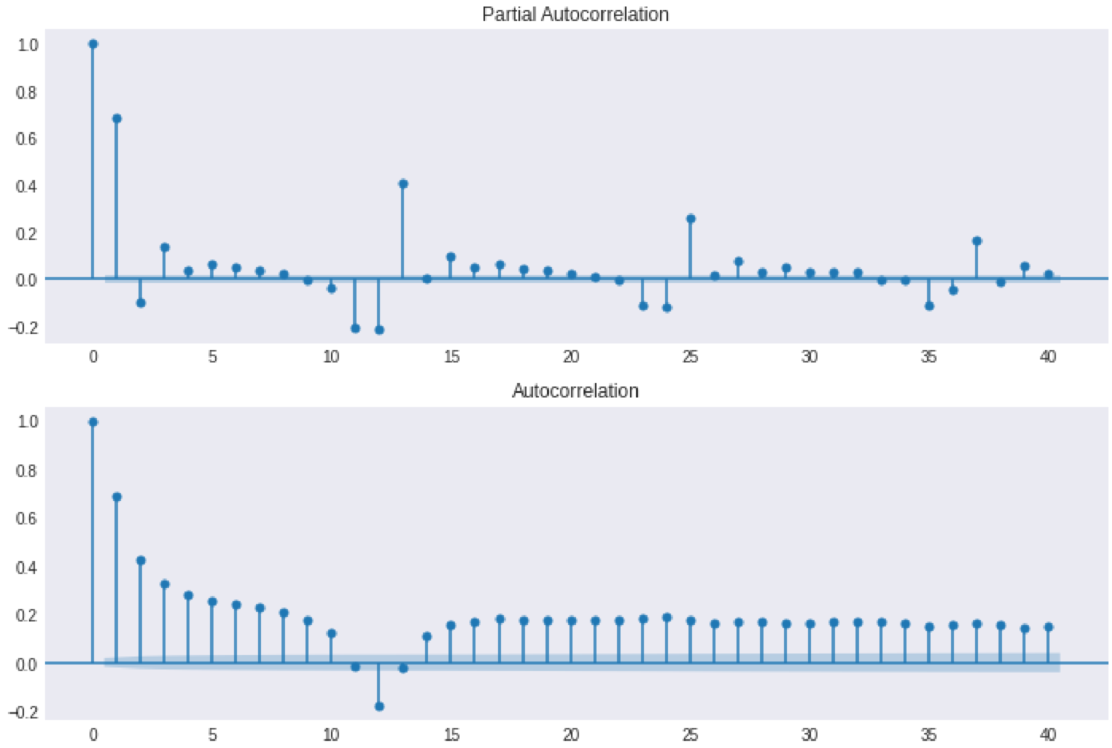
```
ADF Test Statistic : -8.151279157068146
p-value : 9.681017925787857e-13
#Lags Used : 40
Number of Observations Used : 11270
strong evidence against the null hypothesis(Ho), reject the null hypothesis.
Data has no unit root and is stationary
```

[36]:
```
<AxesSubplot:>
```



[38]:
```
from statsmodels.graphics.tsaplots import plot_pacf ,plot_acf
```

[39]:
```
fig = plt.figure(figsize = (12, 8))
ax1 = fig.add_subplot(211)
fig = plot_pacf(train['Seasonal First Difference'].iloc[13:],lags=40,ax=ax1)
ax2 = fig.add_subplot(212)
fig = plot_acf(train['Seasonal First Difference'].iloc[13:],lags=40,ax=ax2)
```

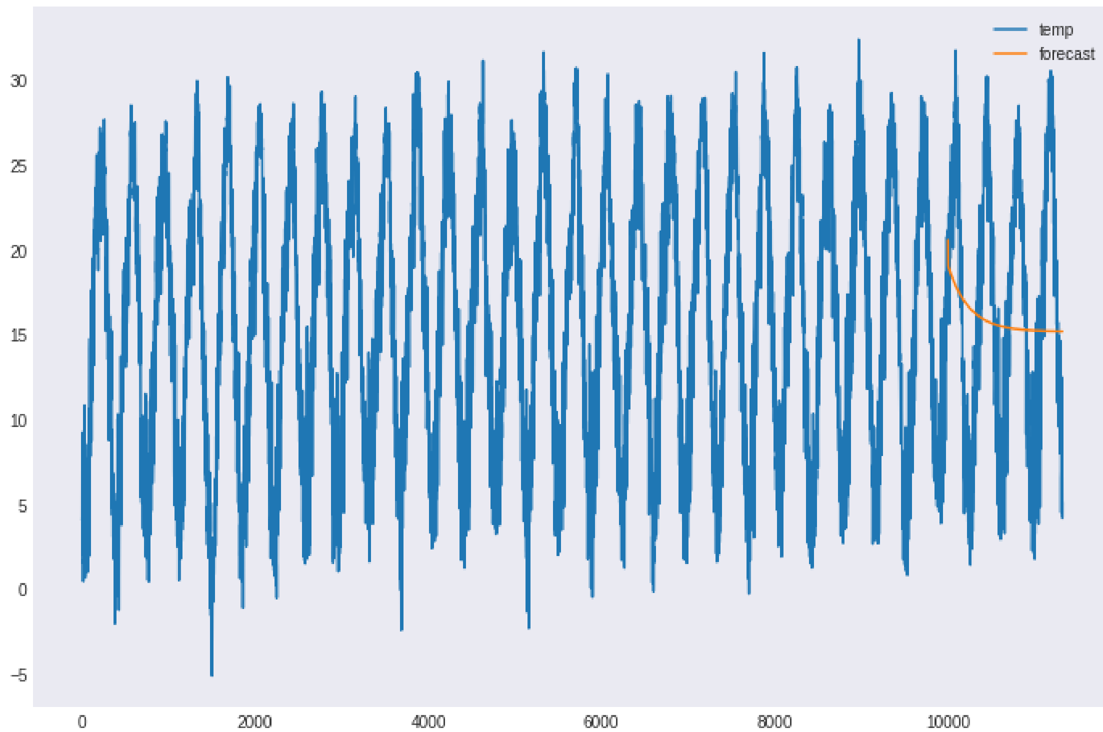


[40]:
```python
from statsmodels.tsa.arima_model import ARIMA

model=ARIMA(train['temp'],order=(2,0,2))
model_fit=model.fit()

model_fit.summary()

train['forecast']=model_fit.predict(start=10000,end=11321,dynamic=True)
train[['temp','forecast']].plot(figsize=(12,8))
```

```
/opt/conda/lib/python3.7/site-packages/statsmodels/tsa/arima_model.py:472:
FutureWarning:
statsmodels.tsa.arima_model.ARMA and statsmodels.tsa.arima_model.ARIMA have
been deprecated in favor of statsmodels.tsa.arima.model.ARIMA (note the .
between arima and model) and
statsmodels.tsa.SARIMAX. These will be removed after the 0.12 release.

statsmodels.tsa.arima.model.ARIMA makes use of the statespace framework and
is both well tested and maintained.

To silence this warning and continue using ARMA and ARIMA until they are
removed, use:
```

```
import warnings
warnings.filterwarnings('ignore', 'statsmodels.tsa.arima_model.ARMA',
                        FutureWarning)
warnings.filterwarnings('ignore', 'statsmodels.tsa.arima_model.ARIMA',
                        FutureWarning)

  warnings.warn(ARIMA_DEPRECATION_WARN, FutureWarning)
```

[40]: <AxesSubplot:>



The forecast is poor because data is seasonal, so we need to use `SARIMAX`