Q-4 Regression

- Akshay Bankar (2019201011)

Household power consumption data - Time series forcasting :

```
A time series is a sequence of observations taken sequentially in time.
```

Time series adds an explicit order dependence between observations: a time dimension. This additional dimension is both a constraint and a structure that provides a source of additional information.

The household power consumption data is a multivariate series comprised of seven variables (besides the date and time); they are:

```
global_active_power: The total active power consumed by the household
(kilowatts).
global_reactive_power: The total reactive power consumed by the househo
ld (kilowatts).
voltage: Average voltage (volts).
global_intensity: Average current intensity (amps).
sub_metering_1: Active energy for kitchen (watt-hours of active energ
y).
sub_metering_2: Active energy for laundry (watt-hours of active energ
y).
sub_metering_3: Active energy for climate control systems (watt-hours o
f active energy).
```

- In the given problem we are asked to perform regression over the dataset of global active power values.
- We are supposed to take the active power values in the past one hour and predict the next active power value

Hence it becomes a Univariate time series problem where dataset comprises of a single series of observations with a temporal ordering

In [0]:
```
import numpy as np
import pandas as pd
```

Data preparation

> Using read_csv() function of pandas to load the data and combine the first two columns into a single date-time column that can be used as an index.

In [10]:
```
dataframe = pd.read_csv('/content/drive/My Drive/household_power_consumptic
```

```
/usr/local/lib/python3.6/dist-packages/IPython/core/interactiveshell.py:2718:
DtypeWarning: Columns (2,3,4,5,6,7) have mixed types.Specify dtype option on i
mport or set low_memory=False.
  interactivity=interactivity, compiler=compiler, result=result)
```

In [4]:
```
dataframe.head()
```

Out[4]:

datetime	Global_active_power	Global_reactive_power	Voltage	Global_intensity	Sub_metering_1	Sub_me
2006-12-16 17:24:00	4.216	0.418	234.840	18.400	0.000	
2006-12-16 17:25:00	5.360	0.436	233.630	23.000	0.000	
2006-12-16 17:26:00	5.374	0.498	233.290	23.000	0.000	
2006-12-16 17:27:00	5.388	0.502	233.740	23.000	0.000	
2006-12-16 17:28:00	3.666	0.528	235.680	15.800	0.000	

Fill missing values : mark all missing values indicated with a '?' character with a NaN value.

Using **forward filling** (Walk-Forward), we fill the missing values with the previous days' values as this is logical for a time series data that the pattern of values will be very close to values with previous timestamps.

Walk-Forward : the actual data for that hour is made available to the model so that it can be used as the basis for making a prediction on the subsequent hour.

In [0]:
```
def fill_missing(values):
    one_day = 60 * 24
    for row in range(values.shape[0]):
        for col in range(values.shape[1]):
            if np.isnan(values[row, col]):
                values[row, col] = values[row - one_day, col]

# mark all missing values
dataframe.replace('?', np.nan , inplace=True)
# make dataset numeric
dataframe = dataframe.astype('float32')
# fill missing
fill_missing(dataframe.values)
```

Extract the "Global_active_power" column from the data so that the dataset is a **Univariate** now.

In [12]:
```
df = dataframe['Global_active_power']
df.head()
```

Out[12]:
```
datetime
2006-12-16 17:24:00    4.216
2006-12-16 17:25:00    5.360
2006-12-16 17:26:00    5.374
2006-12-16 17:27:00    5.388
2006-12-16 17:28:00    3.666
Name: Global_active_power, dtype: float32
```

Create data samples :

Divide the sequence into multiple input/output patterns called samples, where 60 observations corresponding to an hour are used as input and one time step is used as output for the one-step prediction that is being learned.

In [13]:
```
from sklearn.model_selection import train_test_split
def split_sequence(sequence, n_steps):
      X, y = list(), list()
      for i in range(len(sequence)):
        end_ix = i + n_steps
        if end_ix > len(sequence)-1:
            break
        seq_x, seq_y = sequence[i:end_ix], sequence[end_ix]
        X.append(seq_x)
        y.append(seq_y)
      return np.array(X), np.array(y)

steps = 60
X, y = split_sequence(df.to_numpy(), n_steps=steps)
print("Number of input-output samples :",np.shape(X))
X_train, X_test, y_train, y_test = train_test_split(X, y, train_size=0.8, r
```

```
Number of input-output samples : (2075199, 60)
```

Regression using MLP

Import keras libraries to be used to build MLP model.

In [0]:
```
from tensorflow.keras.models import Model
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Dense
from tensorflow.keras.backend import variable
```

Define model with one, two and three hidden layer with RELU activation function and with loss function as mean-square-error.

In [0]:
```
##### Model with one hidden layer #####
model_1layer = Sequential()
model_1layer.add(Dense(100, activation='relu', input_dim=steps))
model_1layer.add(Dense(1))
model_1layer.compile(optimizer='adam', loss='mse')

##### Model with two hidden layers #####
model_2layer = Sequential()
model_2layer.add(Dense(100, activation='relu', input_dim=steps))
model_2layer.add(Dense(100, activation='relu', input_dim=100))
model_2layer.add(Dense(1))
model_2layer.compile(optimizer='adam', loss='mse')

##### Model with three hidden layer #####
model_3layer = Sequential()
model_3layer.add(Dense(100, activation='relu', input_dim=steps))
model_3layer.add(Dense(100, activation='relu', input_dim=100))
model_3layer.add(Dense(100, activation='relu', input_dim=100))
model_3layer.add(Dense(1))
model_3layer.compile(optimizer='adam', loss='mse')
```

Fit the above three models defined with batch_sizev= 64 and 10 epochs

In [24]:
```
model_1layer.fit(X_train, y_train, epochs=5, batch_size = 64, verbose=0)
model_2layer.fit(X_train, y_train, epochs=5, batch_size = 64, verbose=0)
model_3layer.fit(X_train, y_train, epochs=5, batch_size = 64, verbose=0)
```

Out[24]:
```
<tensorflow.python.keras.callbacks.History at 0x7f8890036940>
```

Predict on test data

In [0]:
```
y_pred_1layer = model_1layer.predict(X_test, verbose=0)
y_pred_2layer = model_2layer.predict(X_test, verbose=0)
y_pred_3layer = model_3layer.predict(X_test, verbose=0)
```

Calculate MSE and R2-score for the three models.

Observation : The model with two hidden layers performs well.

In [26]:
```
from sklearn.metrics import mean_squared_error, r2_score

y_test = y_test.reshape(y_test.shape[0],1)

print("Mean squared error with one layer: %.2f" % mean_squared_error(y_test
print('Variance score with one layer: %.2f' % r2_score(y_test, y_pred_1laye

print("Mean squared error with two layers: %.2f" % mean_squared_error(y_tes
print('Variance score with two layers: %.2f' % r2_score(y_test, y_pred_2lay

print("Mean squared error with three layers: %.2f" % mean_squared_error(y_t
print('Variance score with three layers: %.2f' % r2_score(y_test, y_pred_3l
```

```
Mean squared error with one layer: 0.07
Variance score with one layer: 0.94
Mean squared error with two layers: 0.07
Variance score with two layers: 0.94
Mean squared error with three layers: 0.07
Variance score with three layers: 0.94
```

Evaluation with different activation functions :

In [0]:
```
steps = 60
##### Model with sigmoid activation function #####
model_2layer_sig = Sequential()
model_2layer_sig.add(Dense(100, activation='sigmoid', input_dim=steps))
model_2layer_sig.add(Dense(100, activation='sigmoid', input_dim=100))
model_2layer_sig.add(Dense(1))
model_2layer_sig.compile(optimizer='adam', loss='mse')

##### Model with tanh activation function #####
model_2layer_tanh = Sequential()
model_2layer_tanh.add(Dense(100, activation='tanh', input_dim=steps))
model_2layer_tanh.add(Dense(100, activation='tanh', input_dim=100))
model_2layer_tanh.add(Dense(1))
model_2layer_tanh.compile(optimizer='adam', loss='mse')

##### Model with linear activation function #####
model_2layer_lin = Sequential()
model_2layer_lin.add(Dense(100, activation='linear', input_dim=steps))
model_2layer_lin.add(Dense(100, activation='linear', input_dim=100))
model_2layer_lin.add(Dense(1))
model_2layer_lin.compile(optimizer='adam', loss='mse')
```

In [18]:
```
model_2layer_sig.fit(X_train, y_train, epochs=5, batch_size = 64, verbose=0
model_2layer_tanh.fit(X_train, y_train, epochs=5, batch_size = 64, verbose=
model_2layer_lin.fit(X_train, y_train, epochs=5, batch_size = 64, verbose=0
```

Out[18]:
```
<tensorflow.python.keras.callbacks.History at 0x7f8878233860>
```

In [19]:
```
y_pred_2layer_sig = model_2layer_sig.predict(X_test, verbose=0)
y_pred_2layer_tanh = model_2layer_tanh.predict(X_test, verbose=0)
y_pred_2layer_lin = model_2layer_lin.predict(X_test, verbose=0)

from sklearn.metrics import mean_squared_error, r2_score

y_test = y_test.reshape(y_test.shape[0],1)

print("Mean squared error with sigmoid activation: %.2f" % mean_squared_err
print('Variance score with sigmoid activation: %.2f' % r2_score(y_test, y_p

print("Mean squared error with tanh activation: %.2f" % mean_squared_error(
print('Variance score with tanh activation: %.2f' % r2_score(y_test, y_pred

print("Mean squared error with linear activation: %.2f" % mean_squared_erro
print('Variance score with linear activation: %.2f' % r2_score(y_test, y_pr
```

```
Mean squared error with sigmoid activation: 0.07
Variance score with sigmoid activation: 0.94
Mean squared error with tanh activation: 0.07
Variance score with tanh activation: 0.94
Mean squared error with linear activation: 0.07
Variance score with linear activation: 0.94
```

Taking observation window of more than an hour

> Observation window of two hours

In [21]:
```
##### Observation window of two hours #####
steps = 120
X, y = split_sequence(df.to_numpy(), n_steps=steps)
print("Number of input-output samples :",np.shape(X))
X_train, X_test, y_train, y_test = train_test_split(X, y, train_size=0.8, r

##### Model with two hidden layers #####
model_2layer = Sequential()
model_2layer.add(Dense(100, activation='relu', input_dim=steps))
model_2layer.add(Dense(100, activation='relu', input_dim=100))
model_2layer.add(Dense(1))
model_2layer.compile(optimizer='adam', loss='mse')
model_2layer.fit(X_train, y_train, epochs=5, batch_size = 64, verbose=0)

y_pred_2layer = model_2layer.predict(X_test, verbose=0)

print("Mean squared error with two-hour window: %.2f" % mean_squared_error(
print('Variance score with two-hour window: %.2f' % r2_score(y_test, y_pred
```

```
Number of input-output samples : (2075139, 120)
Mean squared error with two-hour window: 0.07
Variance score with two-hour window: 0.94
```

Observation window of three hours

In [22]:
```
##### Observation window of three hours #####
steps = 180
X, y = split_sequence(df.to_numpy(), n_steps=steps)
print("Number of input-output samples :",np.shape(X))
X_train, X_test, y_train, y_test = train_test_split(X, y, train_size=0.8, r

##### Model with two hidden layers #####
model_2layer = Sequential()
model_2layer.add(Dense(100, activation='relu', input_dim=steps))
model_2layer.add(Dense(100, activation='relu', input_dim=100))
model_2layer.add(Dense(1))
model_2layer.compile(optimizer='adam', loss='mse')
model_2layer.fit(X_train, y_train, epochs=5, batch_size = 64, verbose=0)

y_pred_2layer = model_2layer.predict(X_test, verbose=0)

print("Mean squared error with three-hour window: %.2f" % mean_squared_erro
print('Variance score with three-hour window: %.2f' % r2_score(y_test, y_pr
```

```
Number of input-output samples : (2075079, 180)
Mean squared error with three-hour window: 0.07
Variance score with three-hour window: 0.94
```

Using Linear regression

Define the Linear regression class (using the class definition from previous assignment)

In [0]:

```
class LinearRegression:

    def fit(self, X, y, lr = 0.001, iters=1000, verbose=True, batch_size=1)
        X = self.add_bias(X)
        self.weights = np.zeros(len(X[0]))
        for i in range(iters):
            idx = np.random.choice(len(X), batch_size)
            X_batch, y_batch =  X[idx], y[idx]
            self.weights -= lr * self.get_gradient(X_batch, y_batch)
            if i % 1000 == 0 and verbose:
                print('Iterations: %d - Error : %.4f' %(i, self.get_loss(X,

    def predict(self, X):
        return self.predict_(self.add_bias(X))

    def get_loss(self, X, y):
        return np.mean((y - self.predict_(X)) ** 2)

    def predict_(self, X):
        return np.dot(X,self.weights)

    def add_bias(self,X):
        return np.insert(X, 0, np.ones(len(X)), axis=1)

    def get_gradient(self, X, y):
        return -1.0 * np.dot(y - self.predict_(X), X) / len(X)

    def evaluate(self, X, y):
        return self.get_loss(self.add_bias(X), y)
```

Fit the train data using linear regression

In [31]:

```
lin_reg = LinearRegression()
lin_reg.fit(X_train, y_train, iters = 11000)
```

```
Iterations: 0 - Error : 2.2557
Iterations: 1000 - Error : 0.1640
Iterations: 2000 - Error : 0.1170
Iterations: 3000 - Error : 0.1924
Iterations: 4000 - Error : 0.0932
Iterations: 5000 - Error : 0.0953
Iterations: 6000 - Error : 0.1168
Iterations: 7000 - Error : 0.2455
Iterations: 8000 - Error : 0.0850
Iterations: 9000 - Error : 0.0799
Iterations: 10000 - Error : 0.0863
```

Calculate the MSE and R2-score for the linear regression model.

Observation : The model seems to converge with 10K- 11K iteratons.

In [32]:

```
from sklearn.metrics import mean_squared_error, r2_score
import matplotlib.pyplot as plt

y_pred = lin_reg.predict(X_test)
print("Mean squared error: %.2f"
      % mean_squared_error(y_test, y_pred))
# Explained variance score: 1 is perfect prediction
print('Variance score: %.2f' % r2_score(y_test, y_pred))
```

```
Mean squared error: 0.11
Variance score: 0.90
```