# assignment6_mlr

April 18, 2021

## 1 LINEAR REGRESSION

Objective: To gets hands on experience on multiple linear regression. * Cost Function for regression * Gradient Descent, Learning rates * Feature Normalization * Iterative method vs Direct Method for regression * Using scikit-learn library for regression * How to work with categorical variables / one hot encoding

Problem: We will solve two set of problems 1. Predicting house price 2. Predicting weather station maintenence request counts

## 2 INTRODUCTION

Linear Regression is a supervised machine learning algorithm where the predicted output is continuous and has a constant slope. Is used to predict values within a continuous range. (e.g. sales, price)

Simple Regression: Simple linear regression uses traditional slope-intercept form, where $m$ and $b$ are the variables our algorithm will try to "learn" to produce the most accurate predictions. $x$ represents our input data and $y$ represents our prediction.

$$\hat{y} = \theta_1 x + \theta_0$$

In order to compute the values of $m$ and $b$ we need to minimize a cost function. In the case of Linear Regression it is given by

$$J(\theta) = \frac{1}{2m} \sum_{i=i}^{m} (\hat{y}_i - y_i)^2$$

In multivariate regression we seek a set of parameters

$$\Theta = [\theta_0 \quad \theta_1 \quad \cdots \quad \theta_i \quad \cdots \quad \theta_m]^T$$

which minimizes the the cost function: Hypothesis $$\mathbf{H}=\mathbf{X}\ \Theta$$

Cost function is

$$\mathbf{J(\Theta)} = \frac{1}{2m} (\mathbf{X\Theta} - \mathbf{Y})^T (\mathbf{X\Theta} - \mathbf{Y})$$

The advantages of Linear Regression are that they computationally efficient, simple and easy to interpret. However, the algorithm fails to capture non-linear behavior.

[1]:
```
import warnings
warnings.filterwarnings('ignore')
import numpy as np
import matplotlib.pyplot as plt
#from sklearn import datasets,linear_model
```

## 3 Loading the data from a csv file

The columns represent the surface area, number of rooms and the price of the apartment.

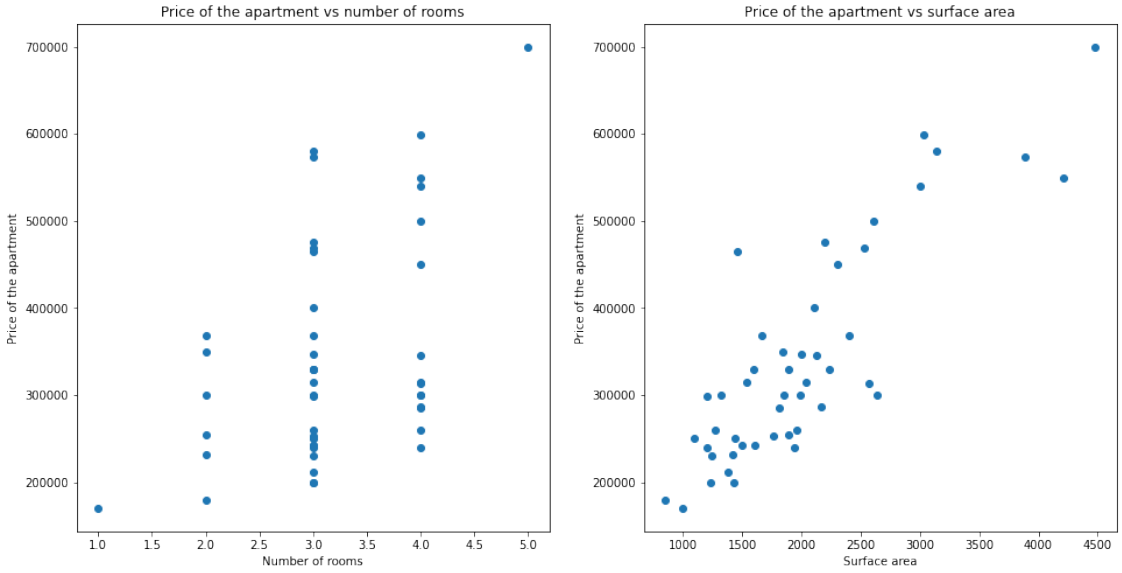
[2]:
```
data = np.loadtxt('../data/Housing/housing_data.csv', delimiter=",")
X = data[:,:2]
y = data[:,2]
m = len(y)
```

## 4 Data Exploration

Plot the data and have a look Note the large difference in the scale of "number of room variables" and "surface area". * "Number of room" has a range [1.0,5.0] * "Surface area" has a range [500,4500]

This causes problems during the cost minimization step (gradient descent).

[3]:
```
plt.figure(figsize=(16,8))
plt.subplot(121)
plt.scatter(X[:,1],y)
plt.xlabel("Number of rooms")
plt.ylabel("Price of the apartment")
plt.title("Price of the apartment vs number of rooms")
plt.subplot(122)
plt.scatter(X[:,0],y)
plt.xlabel("Surface area")
plt.ylabel("Price of the apartment")
plt.title("Price of the apartment vs surface area")
plt.show()
```




## 5 Feature normalization.

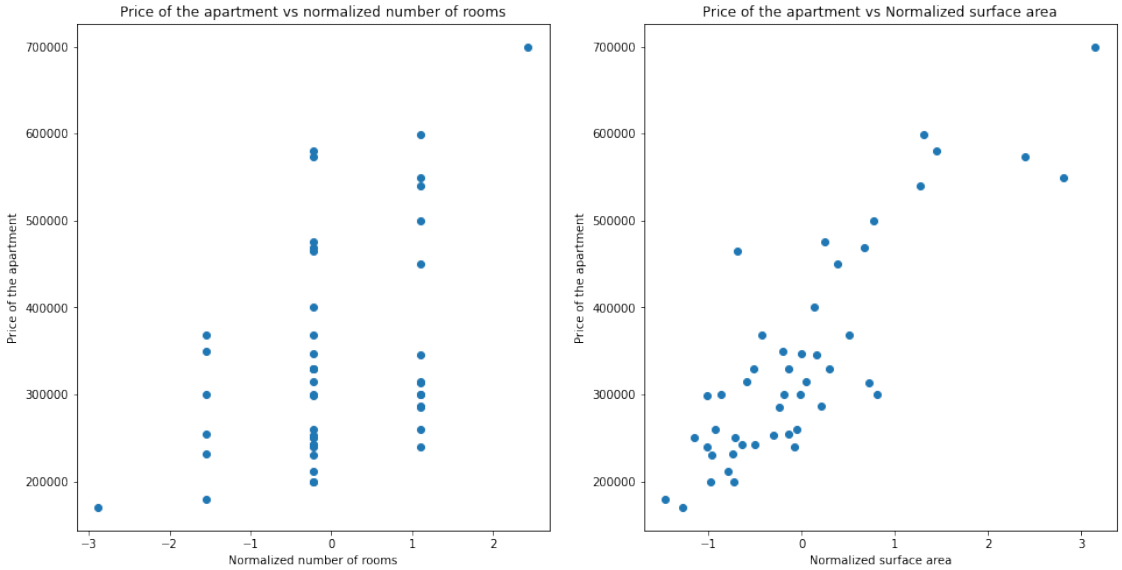
The input feature vectors are normalized so that they have comparable scales.Here we do so by subtracting the mean and dividing by the standard deviation.

[4]:
```
# Normalize features
def featureNormalize(X):
    X_norm = X.copy()
    mu    = np.zeros((1, X.shape[1]))
    sigma = np.zeros((1, X.shape[1]))
    for i in range(X.shape[1]):
        mu[:,i] = np.mean(X[:,i])
        sigma[:,i] = np.std(X[:,i])
        X_norm[:,i] = (X[:,i] - float(mu[:,i]))/float(sigma[:,i])
    return X_norm, mu, sigma
```

[5]:
```
X_norm, mu, sigma = featureNormalize(X)
```

[6]:
```
plt.figure(figsize=(16,8))
plt.subplot(121)
plt.scatter(X_norm[:,1],y)
plt.xlabel("Normalized number of rooms")
plt.ylabel("Price of the apartment")
plt.title("Price of the apartment vs normalized number of rooms")
plt.subplot(122)
plt.scatter(X_norm[:,0],y)
plt.xlabel("Normalized surface area")
```

```
plt.ylabel("Price of the apartment")
plt.title("Price of the apartment vs Normalized surface area")
plt.show()
```




## 6 Bias Term

We simply add a column vector of 1.

[7]:
```
X_padded = np.column_stack((np.ones((m,1)), X_norm))
```

## 7 Cost Function involving multiple variables

$$\mathbf{J}(\Theta) = \frac{1}{2m}(\mathbf{X}\Theta - \mathbf{Y})^T(\mathbf{X}\Theta - \mathbf{Y})$$

[8]:
```
def computeCost(X,y,theta):
    m=len(y)
    Cost=0.0;
    y=y.reshape(-1,1)
    Cost=1.0/2.0/m*(np.dot((X.dot(theta)-y).T,(X.dot(theta)-y)))
    return Cost;
```

[9]:
```
def gradientDescent(X, y, alpha, num_iters):
    m,n = X.shape
    theta = np.zeros((X.shape[1],1))
    Cost_history = np.zeros((num_iters, 1))
    for i in range(num_iters):
```

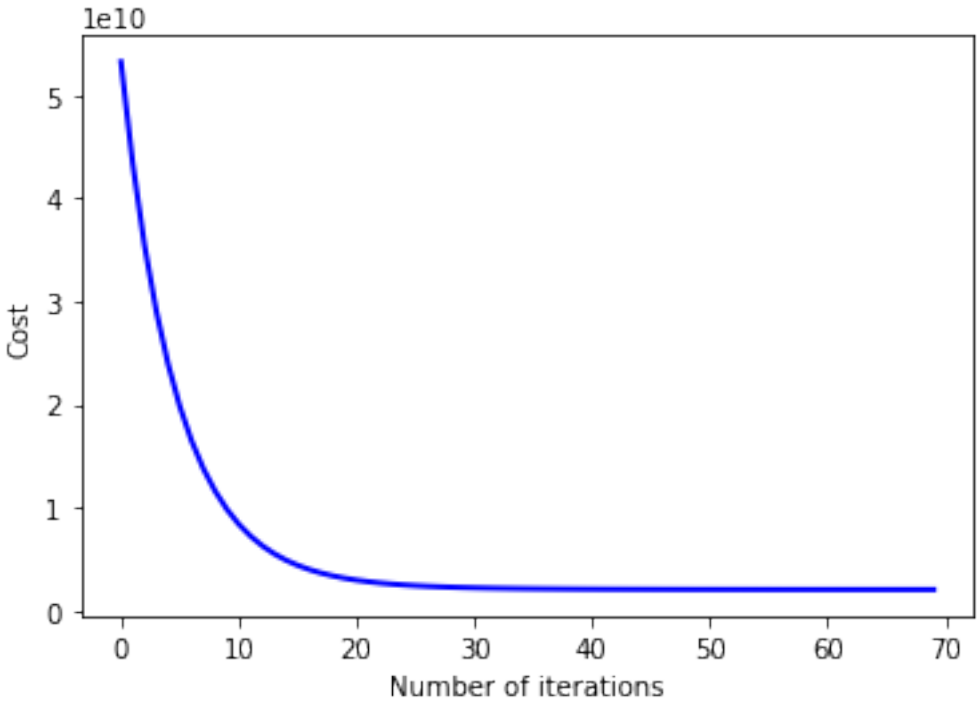
```
        theta = theta - alpha*(1.0/m) * np.transpose(X).dot(X.dot(theta) - np.
↪transpose([y]))
        Cost_history[i] = computeCost(X, y, theta)
    return theta, Cost_history
```

[10]:
```
def SolveGradientDescent(X,y,alpha, num_iters):
    m,n=X.shape
    theta, Loss_history = gradientDescent(X, y, alpha, num_iters)
    plt.plot(range(Loss_history.size), Loss_history, "-b", linewidth=2 )
    plt.xlabel('Number of iterations')
    plt.ylabel('Cost')
    plt.show(block=False)
    theta.shape
    return theta
```

## 8 Evolution of the cost function

Try solving the same problem but without feature normalization and convince yourself the importance of the step.

[11]:
```
theta_GD=SolveGradientDescent(X=X_padded,y=y,alpha=0.1,num_iters=70)
print('Theta computed from gradient descent: ',theta_GD.T)
```



```
Theta computed from gradient descent:  [[340199.36423635 106961.59217624
-4092.86897639]]
```

## 9 Make predictions using the trained model

Remember that the input features were normalized so we need to do the same with the new input features on which we want to make make predictions.

[12]:
```
house_norm_padded = np.array([1, (1650-mu[0,0])/sigma[0,0], (3-mu[0,1])/
 ↪sigma[0,1]])
price_GD = np.array(house_norm_padded).dot(theta_GD)
print("Predicted price of a 1650 sq-ft, 3 br house (using gradient descent):",␣
 ↪price_GD)
```

```
Predicted price of a 1650 sq-ft, 3 br house (using gradient descent):
[293415.1732852]
```

## 10 Solution using Normal Equation

Generally for very large problems due to memory requirements gradient descent algorithm is used for minimization but since here we are working with relatively small dataset with smaller number of feature vectors $X$, we have used direct method for finiding the minima. This involves invoking the closed-form solution to linear regression. Convince yourself with a little calculation that:

$$= (\mathbf{X^T X})^{-1} \mathbf{X^T Y}$$

[13]:
```
# Solve using direct method
def normalEqn(X, y):
    theta = np.zeros((X.shape[1], 1))
    theta = np.linalg.pinv(np.transpose(X).dot(X)).dot(np.transpose(X).dot(y))
    return theta
```

[14]:
```
theta_Normal = normalEqn(X_padded, y)
print("Theta calculated by Normal Equation ",theta_Normal)
price_Normal = np.array(house_norm_padded).dot(theta_Normal)
print("Predicted price of a 1650 sq-ft, 3 br house (using normal equation):",␣
 ↪price_Normal)
```

```
Theta calculated by Normal Equation  [340412.65957447 109447.79646964
-6578.35485416]
Predicted price of a 1650 sq-ft, 3 br house (using normal equation):
293081.4643348961
```

## 11 Regression using scikit-learn library in two lines

[15]:
```
from sklearn import linear_model
lr = linear_model.LinearRegression()
lr.fit(X_padded,y)
print("Theta calculated by SK-learn regression ",lr.coef_)
print("Predicted price ",lr.predict(house_norm_padded.reshape(1,-1)))
```

```
Theta calculated by SK-learn regression  [     0.         109447.79646964
-6578.35485416]
Predicted price  [293081.4643349]
```

## 12 Example 2

Data Description

- date : yyyy-mm-dd format
- calendar_code : 0 or 1 (a code describing certain calendar events)
- request_count : an integer (the number of support requests received on that date)
- site_count : an integer (the number of sites operating on that date)
- max_temp : a float (max temperature for that day in degrees Celsius)
- min_temp : a float (min temperature for that day in degrees Celsius)
- precipitation : a float (millimeters of precipitation on that date)
- events : a string (description of weather events on that date)

Our aim is to predict the request_count when the other paeameters are given.

[16]:
```
import pandas as pd
import numpy as np
from sklearn.preprocessing import LabelEncoder, OneHotEncoder
from sklearn.model_selection import train_test_split
from sklearn import linear_model
import matplotlib.pyplot as plt
import seaborn as sns
```

[17]:
```
#ALso remember to parse the date column. This will be helpful in the next step
training_data=pd.read_csv('../data/Met/Met_train.
 ↪csv',sep=',',parse_dates=['date'])
training_data.head()
```

[17]:

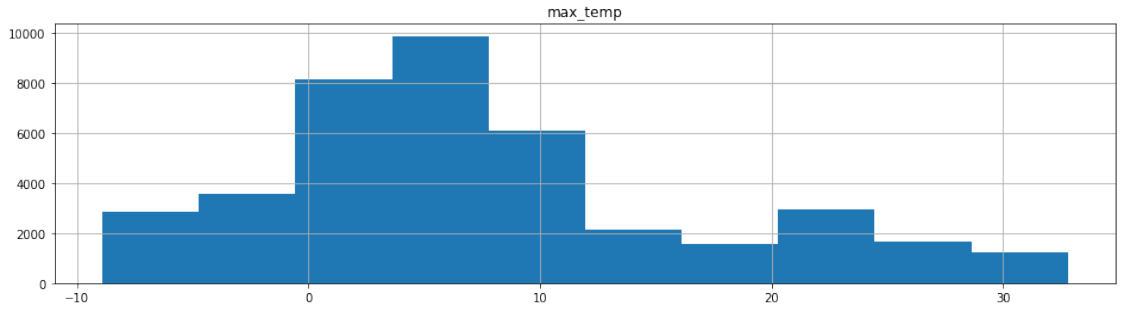
| | date | calendar_code | request_count | site_count | max_temp | min_temp | \ |
|---|---|---|---|---|---|---|---|
| 0 | 2014-09-01 | 0.0 | 165 | 6 | 30.6 | 22.8 | |
| 1 | 2014-09-02 | 1.0 | 138 | 7 | 32.8 | 22.8 | |
| 2 | 2014-09-03 | 1.0 | 127 | 7 | 29.4 | 18.3 | |
| 3 | 2014-09-04 | 1.0 | 174 | 7 | 29.4 | 17.2 | |
| 4 | 2014-09-05 | 1.0 | 196 | 7 | 30.6 | 21.7 | |

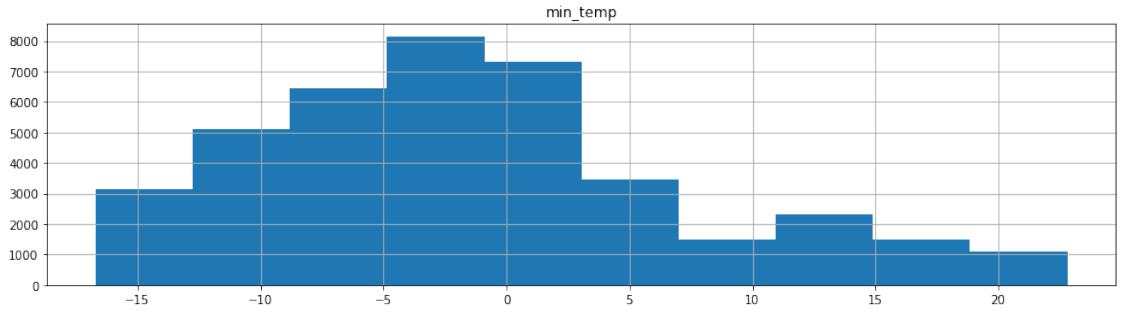
| | precipitation | events |
|---|---|---|

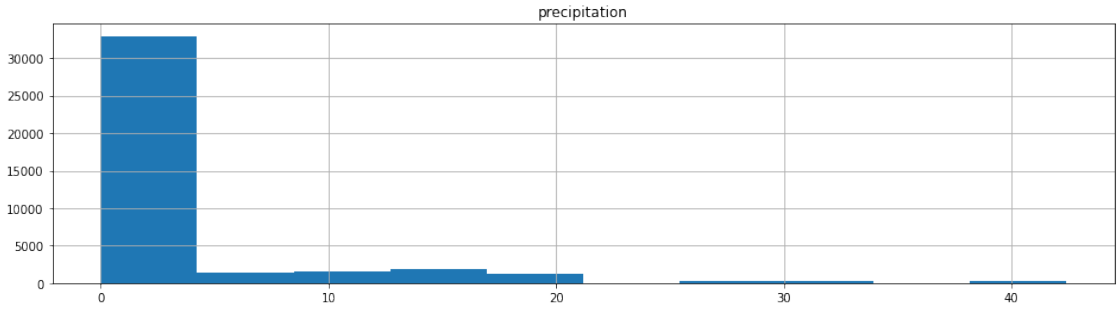
```
0            0.0                Rain
1           15.5  Rain-Thunderstorm
2            0.0                None
3            0.0                None
4            0.0                 Fog
```

## 13 Data Exploration

[18]:
```
training_data.
 ↪hist('max_temp',weights=training_data['request_count'],figsize=(16,4))
training_data.
 ↪hist('min_temp',weights=training_data['request_count'],figsize=(16,4))
training_data.
 ↪hist('precipitation',weights=training_data['request_count'],figsize=(16,4))
plt.show()
```



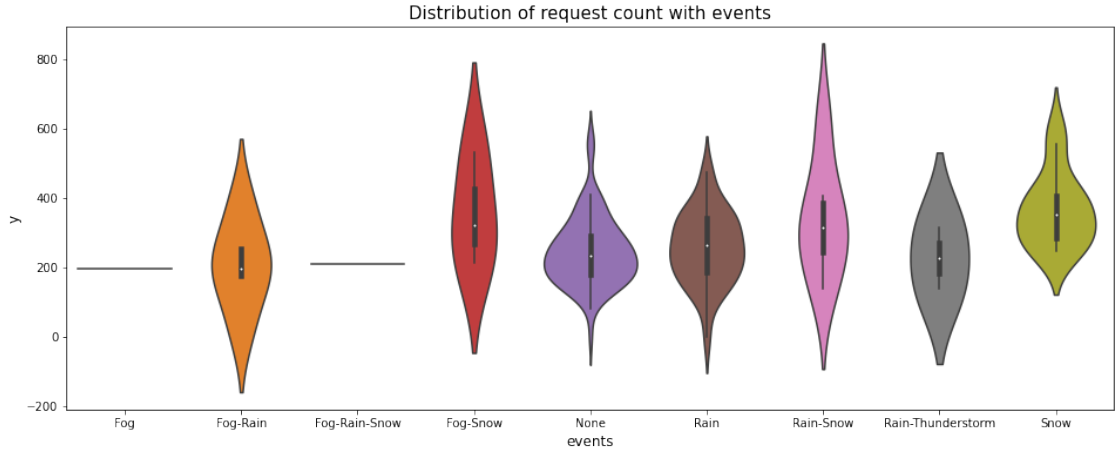






From the above histograms we see that most of the request comes when 1. maximum temperature is below 10C 2. min temperature is below 2C 3. When there is zero precipitation

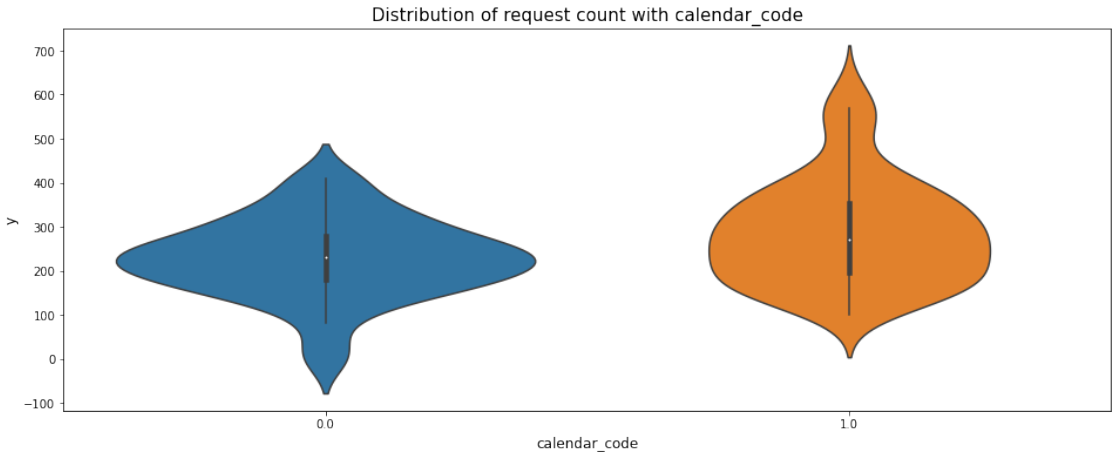
We use violin plot for dependence on the categorical variables (https://blog.modeanalytics.com/violin-plot-examples/)

[19]:
```
var_name = "events"
col_order = np.sort(training_data[var_name].unique()).tolist()
plt.figure(figsize=(16,6))
sns.violinplot(x=var_name, y='request_count', data=training_data,␣
 ↪order=col_order)
plt.xlabel(var_name, fontsize=12)
plt.ylabel('y', fontsize=12)
plt.title("Distribution of request count with "+var_name, fontsize=15)
plt.show()
```




[20]:
```
var_name = "calendar_code"
col_order = np.sort(training_data[var_name].unique()).tolist()
plt.figure(figsize=(16,6))
```

```
sns.violinplot(x=var_name, y='request_count', data=training_data,␣
 ↪order=col_order)
plt.xlabel(var_name, fontsize=12)
plt.ylabel('y', fontsize=12)
plt.title("Distribution of request count with "+var_name, fontsize=15)
plt.show()
```




[21]:
```
training_data['day_of_week'] = training_data['date'].dt.dayofweek
training_data['week_day'] = training_data['date'].dt.day_name()
training_data.head()
```

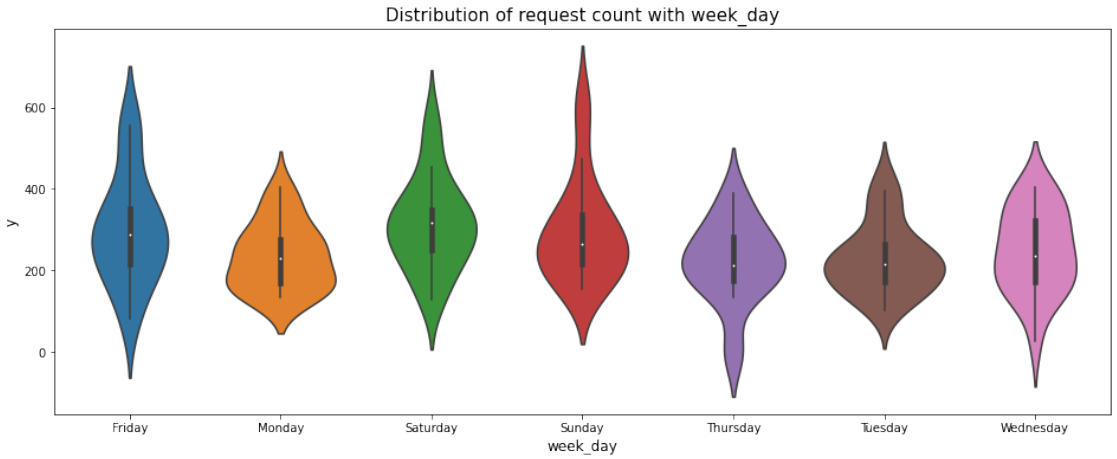
[21]:

| | date | calendar_code | request_count | site_count | max_temp | min_temp | \ |
|---|---|---|---|---|---|---|---|
| 0 | 2014-09-01 | 0.0 | 165 | 6 | 30.6 | 22.8 | |
| 1 | 2014-09-02 | 1.0 | 138 | 7 | 32.8 | 22.8 | |
| 2 | 2014-09-03 | 1.0 | 127 | 7 | 29.4 | 18.3 | |
| 3 | 2014-09-04 | 1.0 | 174 | 7 | 29.4 | 17.2 | |
| 4 | 2014-09-05 | 1.0 | 196 | 7 | 30.6 | 21.7 | |

| | precipitation | events | day_of_week | week_day |
|---|---|---|---|---|
| 0 | 0.0 | Rain | 0 | Monday |
| 1 | 15.5 | Rain-Thunderstorm | 1 | Tuesday |
| 2 | 0.0 | None | 2 | Wednesday |
| 3 | 0.0 | None | 3 | Thursday |
| 4 | 0.0 | Fog | 4 | Friday |

[22]:
```
var_name = "week_day"
col_order = np.sort(training_data[var_name].unique()).tolist()
plt.figure(figsize=(16,6))
sns.violinplot(x=var_name, y='request_count', data=training_data,␣
 ↪order=col_order)
plt.xlabel(var_name, fontsize=12)
```

```
plt.ylabel('y', fontsize=12)
plt.title("Distribution of request count with "+var_name, fontsize=15)
plt.show()
```




## 14 Dealing with categorical variables

In most of the machine learning algorithms barringa a few (ex. Decision Tree) we need numerical values for the input features. In the current data events are categorical variable which should be converted into a unique numerical identifiers. This will result in an additional column "events_code"

[23]:
```
training_data['events_code'] = pd.Categorical(training_data["events"]).codes
training_data.head()
```

[23]:

| | date | calendar_code | request_count | site_count | max_temp | min_temp | \ |
|---|---|---|---|---|---|---|---|
| 0 | 2014-09-01 | 0.0 | 165 | 6 | 30.6 | 22.8 | |
| 1 | 2014-09-02 | 1.0 | 138 | 7 | 32.8 | 22.8 | |
| 2 | 2014-09-03 | 1.0 | 127 | 7 | 29.4 | 18.3 | |
| 3 | 2014-09-04 | 1.0 | 174 | 7 | 29.4 | 17.2 | |
| 4 | 2014-09-05 | 1.0 | 196 | 7 | 30.6 | 21.7 | |

| | precipitation | events | day_of_week | week_day | events_code |
|---|---|---|---|---|---|
| 0 | 0.0 | Rain | 0 | Monday | 5 |
| 1 | 15.5 | Rain-Thunderstorm | 1 | Tuesday | 7 |
| 2 | 0.0 | None | 2 | Wednesday | 4 |
| 3 | 0.0 | None | 3 | Thursday | 4 |
| 4 | 0.0 | Fog | 4 | Friday | 0 |

Since request count is the target variable, we store it separately as "y"

[24]:
```
y=training_data["request_count"]
```

Drop the redundant columns now “date”,“events”,“request_count”

[25]:
```
training_data = training_data.
 ↪drop(["date","events","request_count","week_day"],axis=1)
training_data.head()
```

[25]:
```
   calendar_code  site_count  max_temp  min_temp  precipitation  day_of_week  \
0            0.0           6      30.6      22.8            0.0            0
1            1.0           7      32.8      22.8           15.5            1
2            1.0           7      29.4      18.3            0.0            2
3            1.0           7      29.4      17.2            0.0            3
4            1.0           7      30.6      21.7            0.0            4

   events_code
0            5
1            7
2            4
3            4
4            0
```

## 15 One-Hot-Encoding

The numerical values of day_of_week, events_code and calender code do not signify anything so they need to be one-hot-encoded to be used as a feature input vector.

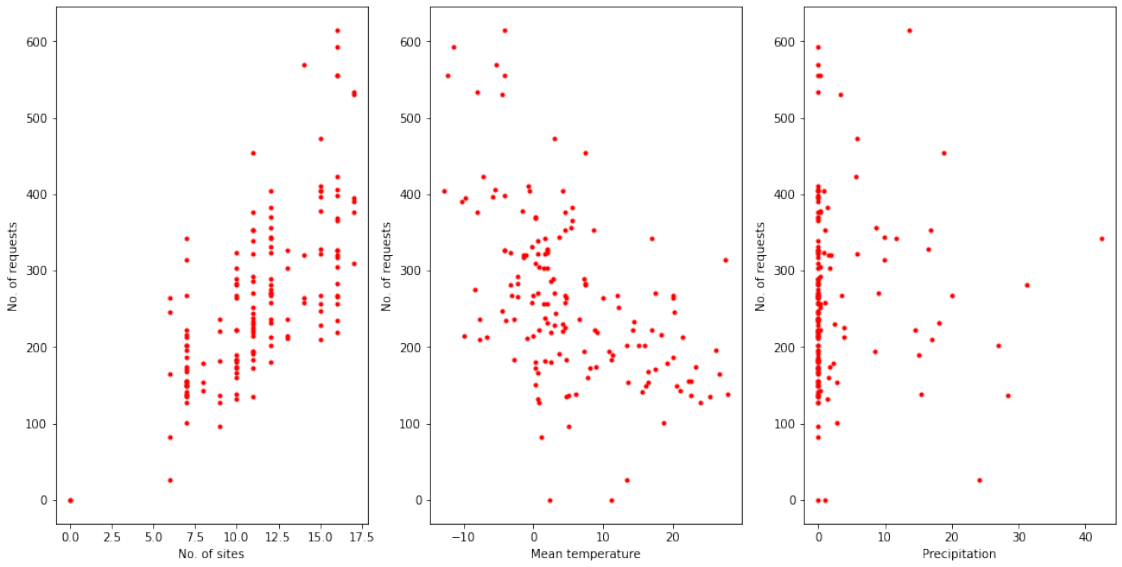
[26]:
```
training_data= pd.
 ↪get_dummies(training_data,columns=["calendar_code","events_code","day_of_week"],prefix=["ca
training_data.head()
```

[26]:
```
   site_count  max_temp  min_temp  precipitation  calendar_0.0  calendar_1.0  \
0           6      30.6      22.8            0.0             1             0
1           7      32.8      22.8           15.5             0             1
2           7      29.4      18.3            0.0             0             1
3           7      29.4      17.2            0.0             0             1
4           7      30.6      21.7            0.0             0             1

   event_0  event_1  event_2  event_3  …  event_6  event_7  event_8  week_0  \
0        0        0        0        0  …        0        0        0       1
1        0        0        0        0  …        0        1        0       0
2        0        0        0        0  …        0        0        0       0
3        0        0        0        0  …        0        0        0       0
4        1        0        0        0  …        0        0        0       0

   week_1  week_2  week_3  week_4  week_5  week_6
0       0       0       0       0       0       0
1       1       0       0       0       0       0
2       0       1       0       0       0       0
```

```
3       0       0       1       0       0       0
4       0       0       0       1       0       0

[5 rows x 22 columns]
```

Call the feature vectors X

[27]:
```
X=training_data.values
plt.figure(figsize=(16,8))
plt.subplot(131)
plt.plot(X[:,0],y[:],'r.')
plt.xlabel("No. of sites")
plt.ylabel("No. of requests")
plt.subplot(132)
plt.plot((X[:,1]+X[:,2])/2.0,y[:],'r.')
plt.xlabel("Mean temperature")
plt.ylabel("No. of requests")
plt.subplot(133)
plt.plot(X[:,3],y[:],'r.')
plt.xlabel("Precipitation")
plt.ylabel("No. of requests")
plt.show()
```




## 16 Task 1: Feature Engineering

It appeats that the no of requests has some kind of a quadratic dependence on the mean temperature so in addition to max and min temperature we should construct a new feature $((minx+maxx)/2)^2$. Feature engineering is one way of brining in domain knowledge or experience into the analysis. Add

a new feature vector to the X matrix

[28]:
```
training_data["feature_eng"] = ((training_data["min_temp"] +␣
 ↪training_data["max_temp"])/2)**2
X=training_data.values
print(X.shape)
training_data.head()
```

```
(152, 23)
```

[28]:

| | site_count | max_temp | min_temp | precipitation | calendar_0.0 | calendar_1.0 | \ |
|---|---|---|---|---|---|---|---|
| 0 | 6 | 30.6 | 22.8 | 0.0 | 1 | 0 | |
| 1 | 7 | 32.8 | 22.8 | 15.5 | 0 | 1 | |
| 2 | 7 | 29.4 | 18.3 | 0.0 | 0 | 1 | |
| 3 | 7 | 29.4 | 17.2 | 0.0 | 0 | 1 | |
| 4 | 7 | 30.6 | 21.7 | 0.0 | 0 | 1 | |

| | event_0 | event_1 | event_2 | event_3 | … | event_7 | event_8 | week_0 | week_1 | \ |
|---|---|---|---|---|---|---|---|---|---|---|
| 0 | 0 | 0 | 0 | 0 | … | 0 | 0 | 1 | 0 | |
| 1 | 0 | 0 | 0 | 0 | … | 1 | 0 | 0 | 1 | |
| 2 | 0 | 0 | 0 | 0 | … | 0 | 0 | 0 | 0 | |
| 3 | 0 | 0 | 0 | 0 | … | 0 | 0 | 0 | 0 | |
| 4 | 1 | 0 | 0 | 0 | … | 0 | 0 | 0 | 0 | |

| | week_2 | week_3 | week_4 | week_5 | week_6 | feature_eng |
|---|---|---|---|---|---|---|
| 0 | 0 | 0 | 0 | 0 | 0 | 712.8900 |
| 1 | 0 | 0 | 0 | 0 | 0 | 772.8400 |
| 2 | 1 | 0 | 0 | 0 | 0 | 568.8225 |
| 3 | 0 | 1 | 0 | 0 | 0 | 542.8900 |
| 4 | 0 | 0 | 1 | 0 | 0 | 683.8225 |

```
[5 rows x 23 columns]
```

## 17 Splitting the data into training and test set

Generally we can always tune the hyperparameters so much that the model performs well on the data but it fails to generalize on the unseen data. To avoid this we split the available data into traing and test sets. We want somewhat similar performance on both the sets.

## 18 Task 2: Split the data into training and test and call them

X_train, X_test, y_train, y_test

[29]:
```
from sklearn.model_selection import train_test_split

X_train, X_test, Y_train, Y_test = train_test_split(training_data, y,␣
 ↪shuffle=True, test_size=0.3, random_state=1)
```

```
print(X_train.shape, X_test.shape)
```

```
(106, 23) (46, 23)
```

# 19 Task 3: Conduct a MLR on the dataset and report MSE both on the training and the test data

[30]:
```
from sklearn import linear_model
lr = linear_model.LinearRegression()

X_train_bias = np.column_stack((np.ones((X_train.shape[0],1)), X_train))
X_test_bias  = np.column_stack((np.ones((X_test.shape[0],1)), X_test))
print(X_train_bias.shape, X_test_bias.shape)

lr.fit(X_train_bias, Y_train)
theta = lr.coef_

print(theta.shape)
print(theta)
```

```
(106, 24) (46, 24)
(24,)
[ 0.00000000e+00  1.93210377e+01 -3.93484417e+00 -1.58011628e+00
  6.52488772e-01 -4.59374179e+01  4.59374179e+01 -1.99816034e+01
  6.15410476e+01 -2.66367402e+02  3.74551487e+01  1.97645085e+01
  5.92448075e+01  6.04256171e+01 -1.20312199e+01  5.99490955e+01
 -4.20398588e+01 -4.19322338e+01 -3.43314882e+01 -3.51184551e+01
  4.01085984e+01  7.17789153e+01  4.15345222e+01  1.83915048e-01]
```

[31]:
```
Y_train_pred = X_train_bias @ theta
#print(Y_train_pred)
#print(Y_train.to_numpy())

error_train = Y_train_pred - Y_train.to_numpy()
#print(error_train)
mse_train = (np.mean(error_train ** 2))
print("Train:", mse_train)
```

```
Train: 2624.2671285609345
```

[32]:
```
Y_test_pred = X_test_bias @ theta
#print(Y_test_pred)
#print(Y_test.to_numpy())

error_test = Y_test_pred - Y_test.to_numpy()
#print(error_test)
```

```
mse_test = (np.mean(error_test ** 2))
print("Test:", mse_test)
```

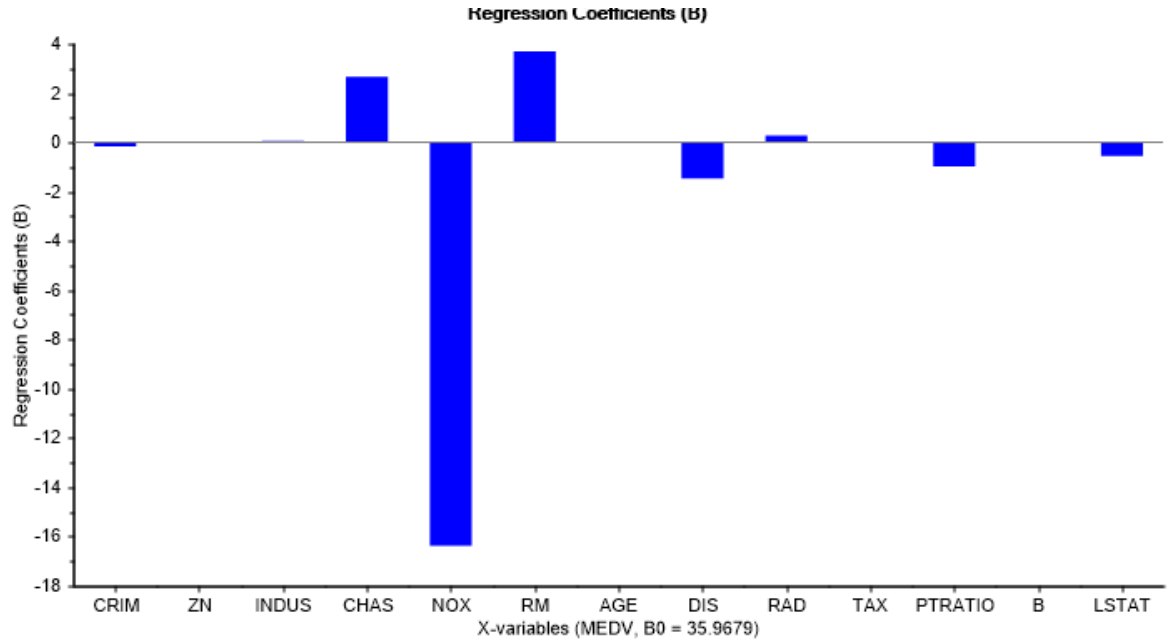
```
Test: 3050.6314507217157
```

**Assignment 6 – Unscrambler part**

We see INDUS and NOX are highly positively correlated which is expected. More industry can mean larger emission of NOX gasses. Somewhat surprisingly NOX and AGE are highly positively correlated.

The predicted vs reference plot shows that many of the predicted data points are close to the reference value, meaning that the model works well. We also see the residuals are around zero, with some outliers, and most are within -10 to 10 range. Considering the different scale on the variables this could be acceptable. In the ANOVA table we have Rsquared of 0.74. This is decent, although ideally we would like to be closer to 1.

The p-values for INDUS and AGE are a lot larger than the others, and are way above normal limits like 0.05 or 0.1. This means that the effect of these variables are not certain.




The regression coefficients plot tells us that the most important variables for house value is the number of rooms RM and the CHAS variable (I don't understand what this is supposed to represent), and if the area is highly polluted by NOX gasses the price drops significantly. DIS pulls down some which can make some sense because houses close to employment centers are probably not inhabited by the richest people. RAD pulls up a little bit which also makes some sense since access to to highways makes commutes easier and is a bonus for most people.

Recalculating with only CHAS, NOX and RM gave a much smaller R square of 0.52 meaning that only these variables are not enough to have a good prediction. The residuals are a little more spread out than before.

The RMSE of the training set was 4.72 and for the testing set it was 4.88. They are similar which indicates that the models predicts the values well.