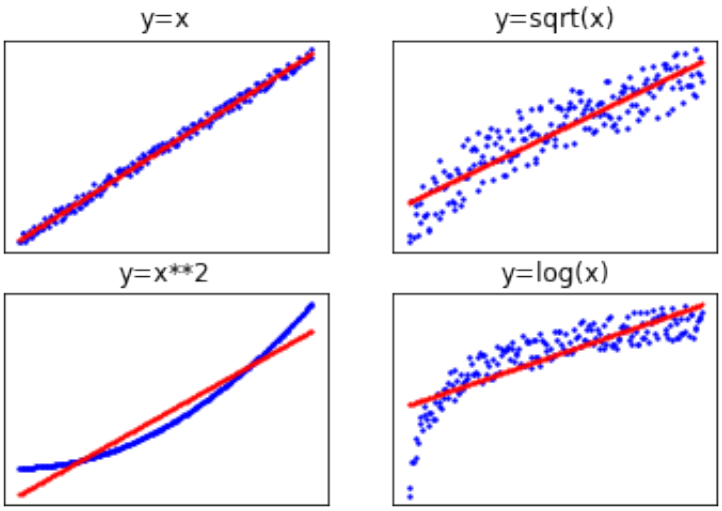
Linear Regression

As in the previous notebook, first we need to input all the packages we need.

In [1]:
```
%matplotlib inline
import matplotlib.pyplot as plt
import seaborn as sns
import pandas as pd
import numpy as np
from sklearn import datasets,linear_model
```

In [2]:
```
def original(x):
    return x

def square(x):
    return x*x

def change_func(N,func,noise_factor,title,i):
    data_x = np.arange(1,N).reshape(-1,1) # Generate an array with
    rdm = (np.random.rand(N-1)-0.5).reshape(-1,1) # Random noise ge
    data_y1 = (func(data_x) + rdm*noise_factor)  # different functi
    # Create and train linear regression model
    regr = linear_model.LinearRegression()
    regr.fit(data_x,data_y1)
    predicted_y = regr.predict(data_x)
    plt.subplot(2,2,i)
    plt.title(title)
    plt.scatter(data_x,data_y1,color="b",s=2)
    plt.scatter(data_x,predicted_y,color="r",s=1)
    plt.xticks([]), plt.yticks([])

N=200
plt.figure(1)
change_func(N,original,15,"y=x",1)
change_func(N,np.sqrt,5,"y=sqrt(x)",2)
change_func(N,square,20,"y=x**2",3)
change_func(N,np.log,1,"y=log(x)",4)
plt.show()
```



Reading Datasets

In order to perform machine learning, we typically need a significant amount of data. By understanding the data, analysing patterns and training our algorithms, we can achieve meaningful results. Scikit-learn makes it easy for us to access some pre-defined 'toy' datasets to practice our understanding.

In this example, we'll use the "diabetes" dataset, which contains records for 442 diabetes patients. The 10 features in the dataset represent each patient's age, sex, body mass index, average blood pressure, and six blood serum measurements. The response of interest is a quantitative measure of disease progression one year after baseline. We'll use this to find a regression to predict a patient's disease progression based on any of their features.

Read through the code below to understand how this particular data is structured. Later, we'll apply a linear regression to the data, but some features will be better suited to this than others. Try to find which feature has the most linear relationship to the target.

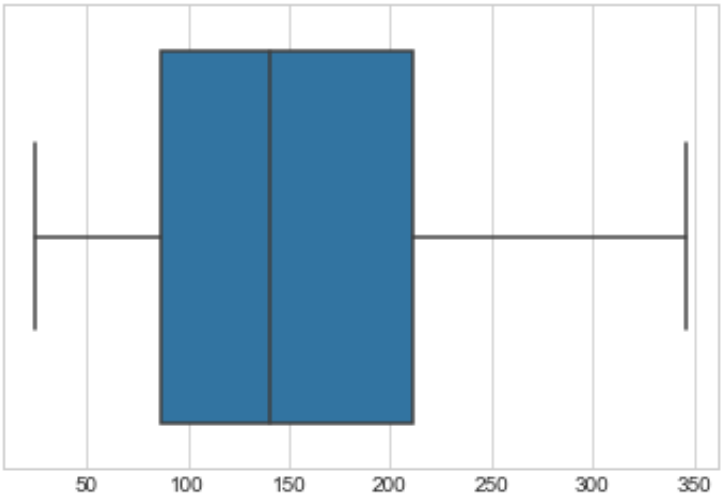
Data Exploration

First, we input the datset of diabetes

In [3]:
```
# load the diabetes dataset
diabetes = datasets.load_diabetes()
data=pd.DataFrame(diabetes.data,columns=diabetes.feature_names)
target=pd.DataFrame(diabetes.target,columns=["target"])
df=pd.concat([data,target],axis=1)
```

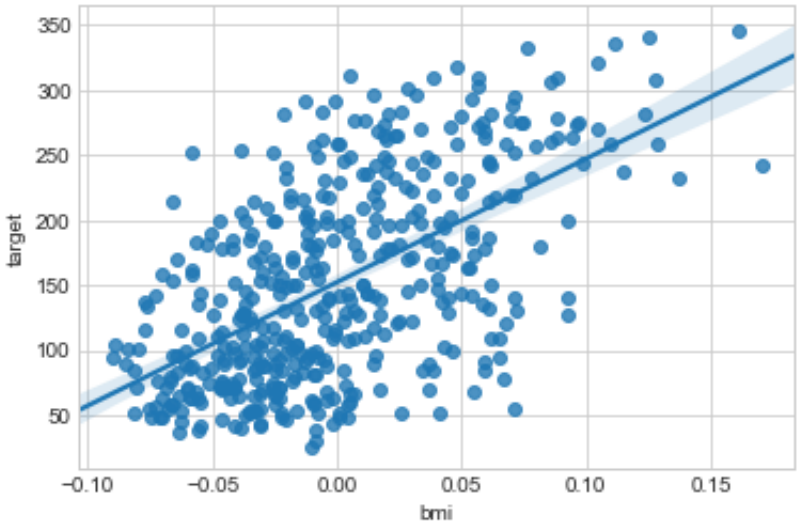
Sometimes you need to see the data plotted out to understand more. Seaborn is a library which is a wrapper over Matplotlib (the standard Python library for data plotting) and is extremely convenient to use. For example, the below plot shows a box plot of target values of all records.

In [4]:
```
sns.set_style("whitegrid")
sns.boxplot(x=target) #Box plot
plt.show()
```



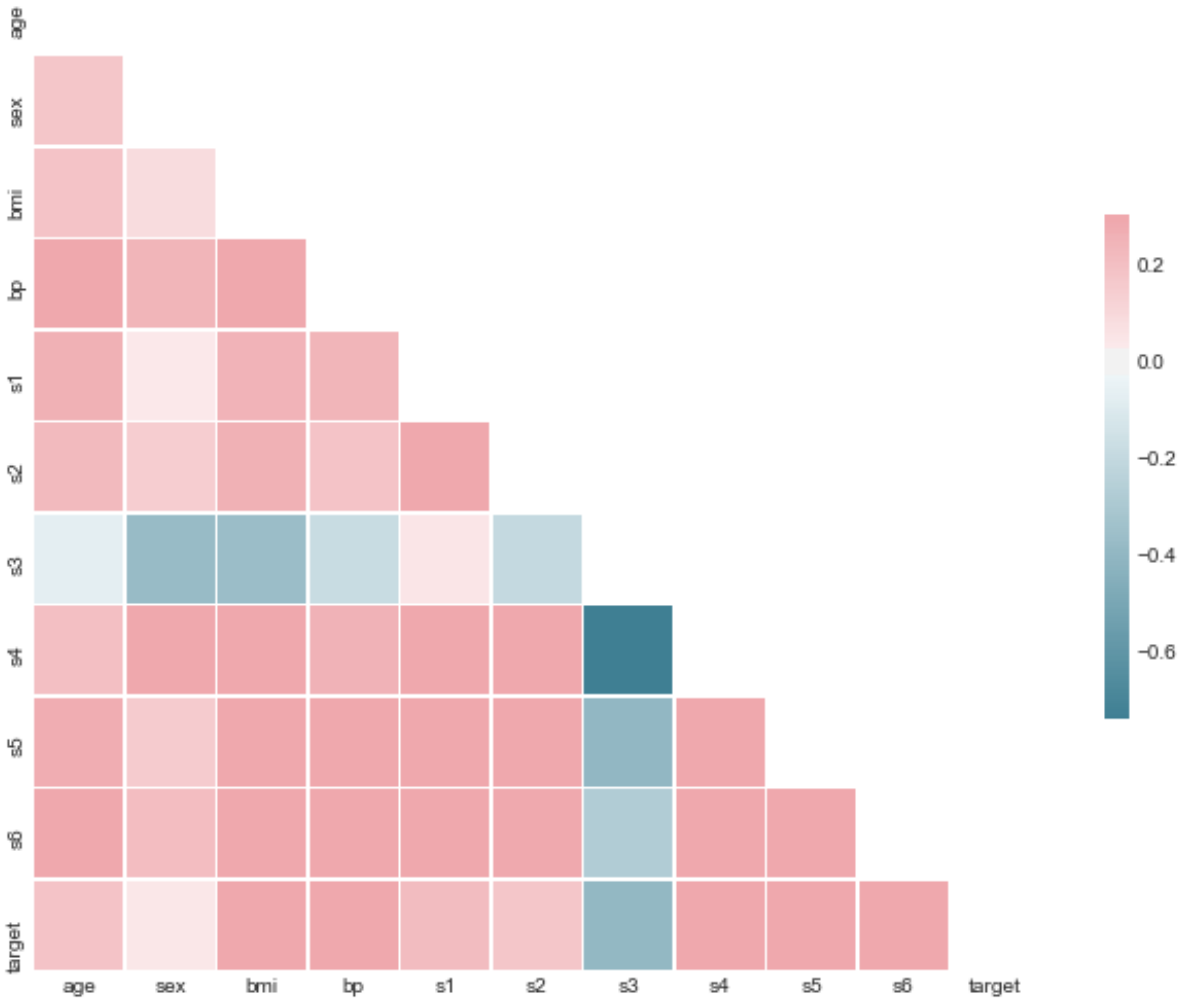
Using Seaborn, plot a scatter plot between bmi and target value.

In [5]:
```
#TODO
sns.regplot(x='bmi',y='target',data=df)
plt.show()
```

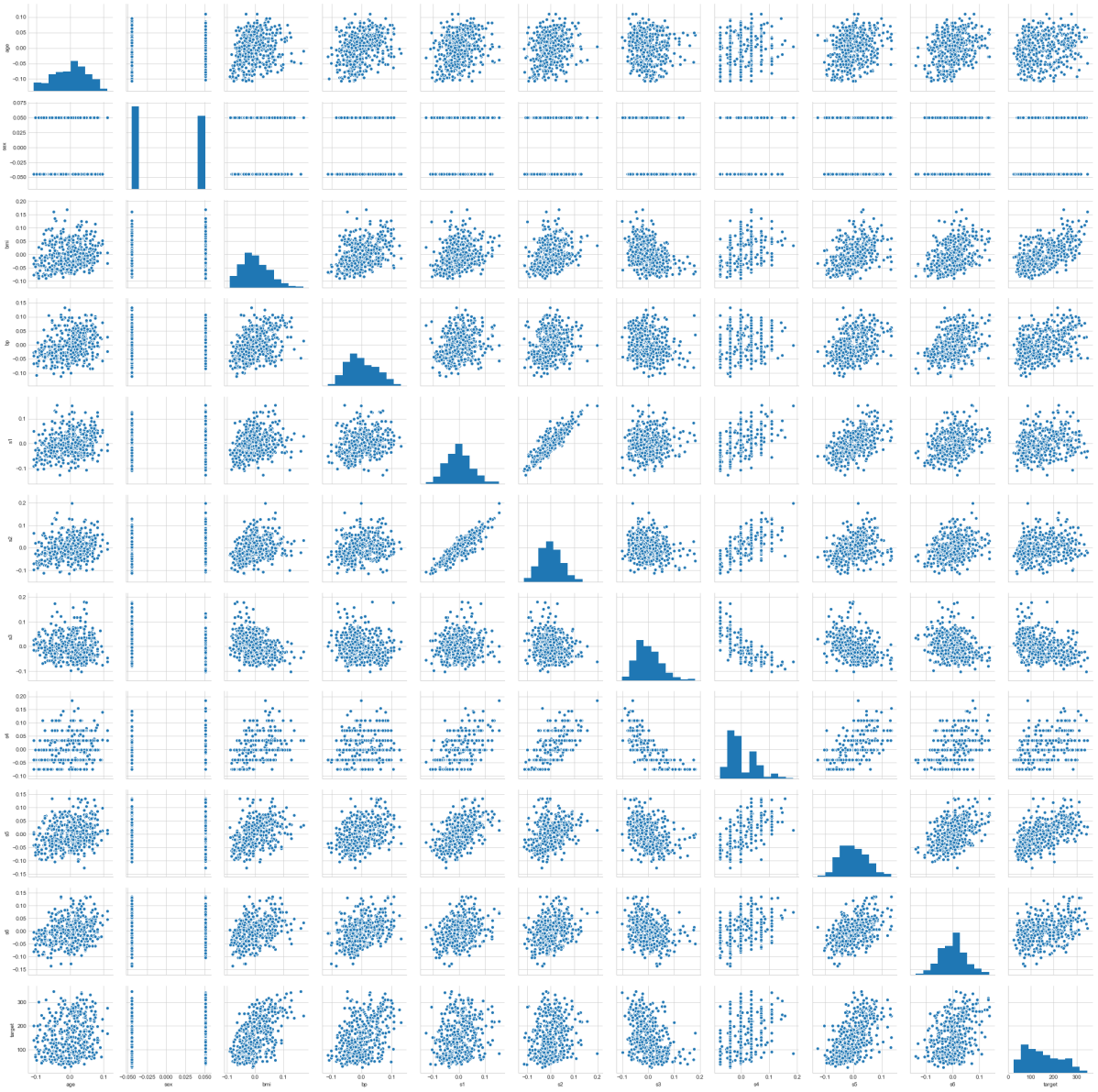


Let's view the correlation heatmap for the different features for some more inspiration.

In [6]:
```
# Compute the correlation matrix
corr = df.corr()

# Generate a mask for the upper triangle
mask = np.zeros_like(corr, dtype=np.bool)
mask[np.triu_indices_from(mask)] = True

# Set up the matplotlib figure
f, ax = plt.subplots(figsize=(11, 9))

# Generate a custom diverging colormap
cmap = sns.diverging_palette(220, 10, as_cmap=True)

# Draw the heatmap with the mask and correct aspect ratio
sns.heatmap(corr, mask=mask, cmap=cmap, vmax=.3, center=0,
            square=True, linewidths=.5, cbar_kws={"shrink": .5})

plt.show()
```



In [7]:
```
sns.pairplot(df)#
plt.show()
```



Now that we have a dataset loaded, and a feature of interest selected, we can try to fit a linear regression. Scikit-learn has methods for accomplishing this very simply. There are two steps involved:

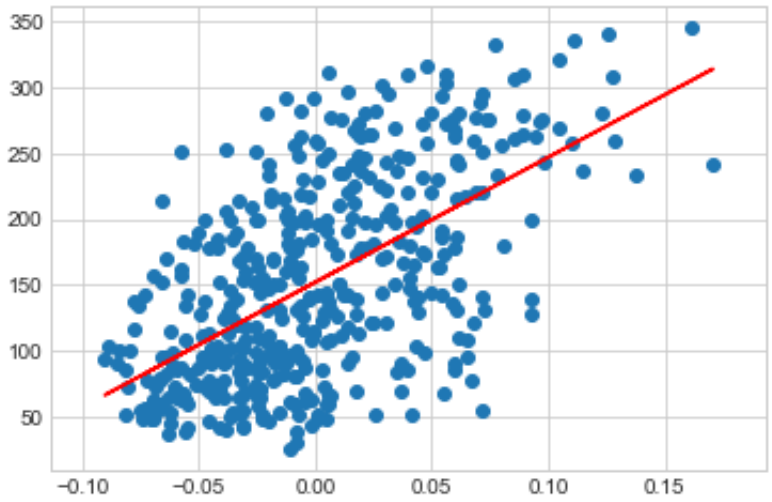
1. Training the model: the linear regression model must be trained by supplying it with a sample feature set, and the corresponding targets.
2. Prediction: once the model is trained, you can provide it with a number of sample points and it will return its predicted targets.

After training the model, we can see the coefficients of the model; i.e., find the *regression coefficient* a and the *intercept* b in the (univariate) linear regression formula $y = ax + b$. We can also calculate the mean squared error of the model points to give us something to compare models and feature choice.

Fill in the code necessary to load the dataset, train and visualise the linear regression.

In [8]:
```
#load the diabetes dataset
diabetes = datasets.load_diabetes()
#Extract a single feature
fnum = 2 ## <-- Whichever feature you chose
data_x = diabetes.data[:,np.newaxis,fnum]
data_y = diabetes.target

#Create linear regression model
regr = linear_model.LinearRegression()
#Train the model
regr.fit(data_x,data_y)
#Predict the targets
predicted_y = regr.predict(data_x)

#Output the coefficient
print('Coefficients: ', regr.coef_[0], regr.intercept_)
#Calculate the Mean Squared Error
mse = np.mean((predicted_y - data_y) ** 2)
print ("Mean squared error: %.2f" % mse)

plt.scatter(data_x,data_y)
plt.plot(data_x,predicted_y,color="red")
plt.show()
```

```
Coefficients:  949.4352603839491 152.1334841628967
Mean squared error: 3890.46
```



If done correctly, you should end up with a linear regression (i.e. straight line) which approximately follows the shape of the scattered data, with an MSE under 4000. You can try using other features in the dataset to see how the linearity of the data affects the MSE.

Test vs. Training set

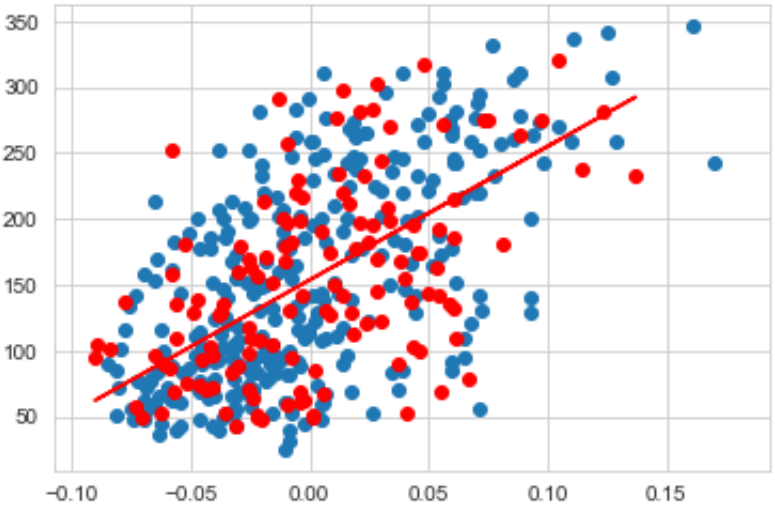
Typically, we do not want to test our model on the data with which we trained it, as this will not provide an accurate assessment of the model. However, we only have a limited amount of data to work with in this case. One way around this issue is to split the data into a 'training set' and a much smaller 'test set'. In this example, we'll take 20 data points out of the dataset to use as our test set, and leave the rest for training.

Change the code below to use the test and training set rather than the dataset as a whole. Note, the array indexing we use to extract the test set is provided by the Numpy module.

How is the mean squared error (MSE) affected when we check with the test set? Why? Try adjusting N and see how the model accuracy is affected.

In [9]:
```
from sklearn.model_selection import train_test_split
#load the data
diabetes = datasets.load_diabetes()
fnum = 2 # selected by prior knowledge or the maximun correlations.
data_x = diabetes.data[:,np.newaxis,fnum]
data_y = diabetes.target

# split data
train_x, test_x, train_y, test_y = train_test_split(
    data_x, data_y, test_size=0.30, random_state=0)
#Create linear regression model
regr = linear_model.LinearRegression()
#Train the model
regr.fit(train_x,train_y)
#Predict the targets
predicted_y =regr.predict(test_x)

#Output the coefficient
print('Coefficients: ', regr.coef_[0], regr.intercept_)
#Calculate the Mean Squared Error
mse = np.mean((predicted_y - test_y) ** 2)
print ("Mean squared error: %.2f" % mse)

plt.scatter(train_x,train_y)
plt.scatter(test_x,test_y, color="red")
plt.plot(test_x,predicted_y,color="red")
plt.show()
```

```
Coefficients:  1013.17358256885 153.4350903922729
Mean squared error: 3921.37
```



Other kinds of measure for linear regression are used:

In [10]:
```
from sklearn.metrics import explained_variance_score,mean_squared_e
def performance_metrics(y_true,y_pred):
    rmse = mean_squared_error(y_true,y_pred)
    r2 = r2_score(y_true,y_pred)
    explained_var_score = explained_variance_score(y_true,y_pred)
    return rmse,r2,explained_var_score

rmsa,r2,explained_var_score=performance_metrics(test_y,predicted_y)
print ("Root of mean squared error: %.2f" % rmsa)
print ("R2-score: %.2f" % r2)
print ("Explained variance score: %.2f" % explained_var_score)
```

```
Root of mean squared error: 3921.37
R2-score: 0.23
Explained variance score: 0.23
```