Boston house price prediction

The problem that we are going to solve here is that given a set of features that describe a house in Boston, our machine learning model must predict the house price. To train our machine learning model with boston housing data, we will be using scikit-learn's boston dataset.

In this dataset, each row describes a boston town or suburb. There are 506 rows and 13 attributes (features) with a target column (price). [https://archive.ics.uci.edu/ml/machine-learning-databases/housing/housing.names (https://archive.ics.uci.edu/ml/machine-learning-databases/housing/housing.names)](https://archive.ics.uci.edu/ml/machine-learning-databases/housing/housing.names)

In [1]:
```
# Importing the Libraries
import pandas as pd
import numpy as np
from sklearn import metrics
import matplotlib.pyplot as plt
import seaborn as sns
%matplotlib inline
```

In [2]:
```
# Importing the Boston Housing dataset
from sklearn.datasets import load_boston
boston = load_boston()
```

In [3]:
```
# Initializing the dataframe
data = pd.DataFrame(boston.data)
```

In [4]:
```
# See head of the dataset
data.head()
```

Out[4]:

	0	1	2	3	4	5	6	7	8	9	10	11	12
0	0.00632	18.0	2.31	0.0	0.538	6.575	65.2	4.0900	1.0	296.0	15.3	396.90	4.98
1	0.02731	0.0	7.07	0.0	0.469	6.421	78.9	4.9671	2.0	242.0	17.8	396.90	9.14
2	0.02729	0.0	7.07	0.0	0.469	7.185	61.1	4.9671	2.0	242.0	17.8	392.83	4.03
3	0.03237	0.0	2.18	0.0	0.458	6.998	45.8	6.0622	3.0	222.0	18.7	394.63	2.94
4	0.06905	0.0	2.18	0.0	0.458	7.147	54.2	6.0622	3.0	222.0	18.7	396.90	5.33

In [5]:
```
#Adding the feature names to the dataframe
data.columns = boston.feature_names
data.head()
```

Out[5]:

	CRIM	ZN	INDUS	CHAS	NOX	RM	AGE	DIS	RAD	TAX	PTRATIO	B	LST
0	0.00632	18.0	2.31	0.0	0.538	6.575	65.2	4.0900	1.0	296.0	15.3	396.90	4.
1	0.02731	0.0	7.07	0.0	0.469	6.421	78.9	4.9671	2.0	242.0	17.8	396.90	9.
2	0.02729	0.0	7.07	0.0	0.469	7.185	61.1	4.9671	2.0	242.0	17.8	392.83	4.
3	0.03237	0.0	2.18	0.0	0.458	6.998	45.8	6.0622	3.0	222.0	18.7	394.63	2.
4	0.06905	0.0	2.18	0.0	0.458	7.147	54.2	6.0622	3.0	222.0	18.7	396.90	5.

CRIM per capita crime rate by town
ZN proportion of residential land zoned for lots over 25,000 sq.ft.
INDUS proportion of non-retail business acres per town
CHAS Charles River dummy variable (= 1 if tract bounds river; 0 otherwise)
NOX nitric oxides concentration (parts per 10 million)
RM average number of rooms per dwelling
AGE proportion of owner-occupied units built prior to 1940
DIS weighted distances to five Boston employment centres
RAD index of accessibility to radial highways
TAX full-value property-tax rate per 10,000usd
PTRATIO pupil-teacher ratio by town
B 1000(Bk - 0.63)^2 where Bk is the proportion of blacks by town
LSTAT % lower status of the population

Each record in the database describes a Boston suburb or town.

In [6]:
```
#Adding target variable to dataframe
data['PRICE'] = boston.target
# Median value of owner-occupied homes in $1000s
```

In [7]:
```
#Check the shape of dataframe
data.shape
```

Out[7]:
```
(506, 14)
```

In [8]:
```
data.columns
```

Out[8]:
```
Index(['CRIM', 'ZN', 'INDUS', 'CHAS', 'NOX', 'RM', 'AGE', 'DIS', 'RAD', 'TA
X',
       'PTRATIO', 'B', 'LSTAT', 'PRICE'],
      dtype='object')
```

In [9]:
```
data.dtypes
```

Out[9]:
```
CRIM       float64
ZN         float64
INDUS      float64
CHAS       float64
NOX        float64
RM         float64
AGE        float64
DIS        float64
RAD        float64
TAX        float64
PTRATIO    float64
B          float64
LSTAT      float64
PRICE      float64
dtype: object
```

In [10]:
```
# Identifying the unique number of values in the dataset
data.nunique()
```

Out[10]:
```
CRIM       504
ZN          26
INDUS       76
CHAS         2
NOX         81
RM         446
AGE        356
DIS        412
RAD          9
TAX         66
PTRATIO     46
B          357
LSTAT      455
PRICE      229
dtype: int64
```

In [11]:
```
# Check for missing values
data.isnull().sum()
```

Out[11]:
```
CRIM       0
ZN         0
INDUS      0
CHAS       0
NOX        0
RM         0
AGE        0
DIS        0
RAD        0
TAX        0
PTRATIO    0
B          0
LSTAT      0
PRICE      0
dtype: int64
```

In [13]:
```
# See rows with missing values
data[data.isnull().any(axis=1)]
```

Out[13]:

CRIM	ZN	INDUS	CHAS	NOX	RM	AGE	DIS	RAD	TAX	PTRATIO	B	LSTAT	PRICE

In [14]:
```
# Viewing the data statistics
data.describe()
```

Out[14]:

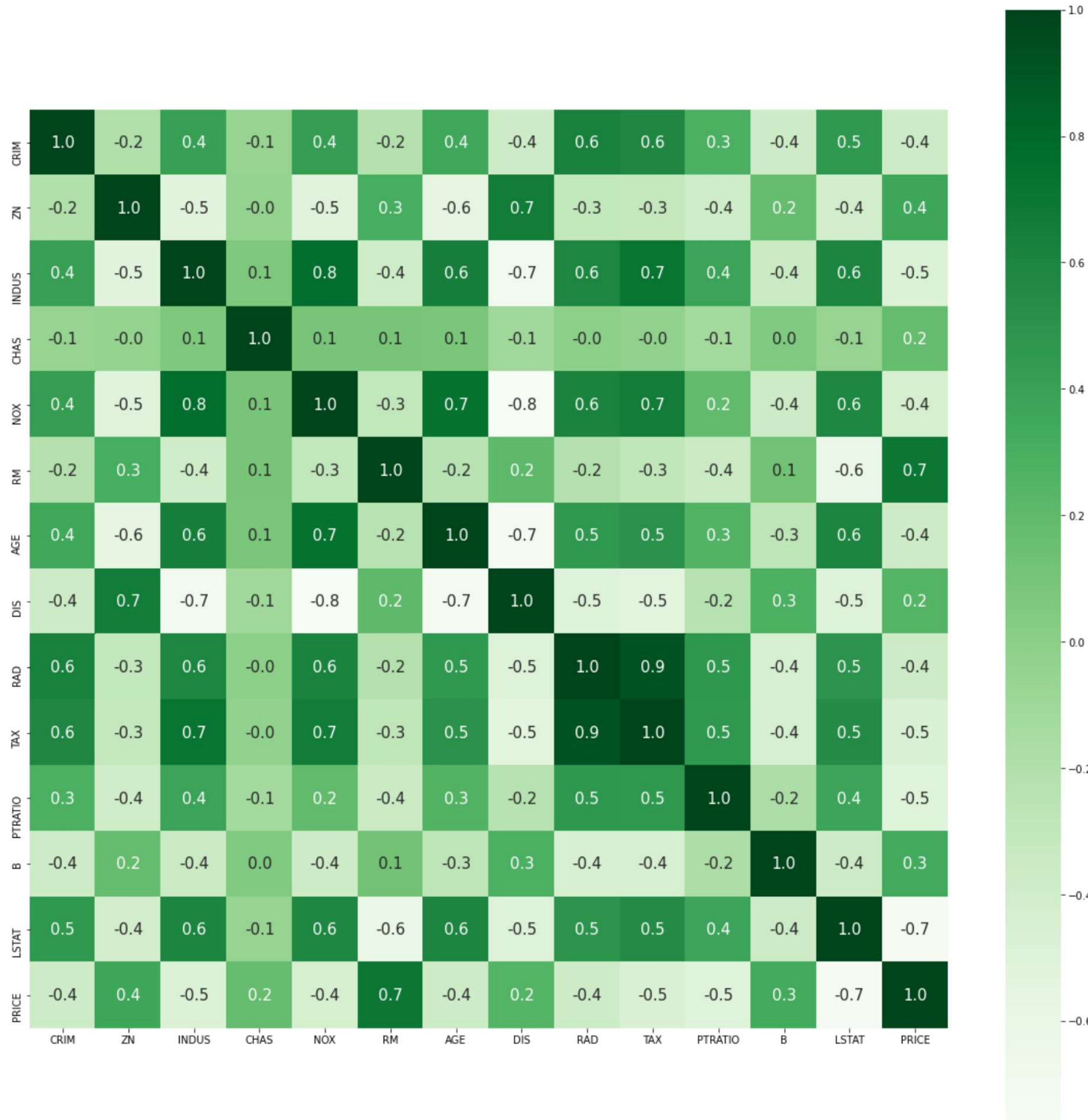
	CRIM	ZN	INDUS	CHAS	NOX	RM	AGE	
count	506.000000	506.000000	506.000000	506.000000	506.000000	506.000000	506.000000	506
mean	3.613524	11.363636	11.136779	0.069170	0.554695	6.284634	68.574901	3
std	8.601545	23.322453	6.860353	0.253994	0.115878	0.702617	28.148861	2
min	0.006320	0.000000	0.460000	0.000000	0.385000	3.561000	2.900000	1
25%	0.082045	0.000000	5.190000	0.000000	0.449000	5.885500	45.025000	2
50%	0.256510	0.000000	9.690000	0.000000	0.538000	6.208500	77.500000	3
75%	3.677083	12.500000	18.100000	0.000000	0.624000	6.623500	94.075000	5
max	88.976200	100.000000	27.740000	1.000000	0.871000	8.780000	100.000000	12

In [15]:
```
# Finding out the correlation between the features
corr = data.corr()
corr.shape
```

Out[15]: (14, 14)

In [16]:
```python
# Plotting the heatmap of correlation between features
plt.figure(figsize=(20,20))
sns.heatmap(corr, cbar=True, square= True, fmt='.1f', annot=True, annot_kw
```

Out[16]: <AxesSubplot:>



In [17]:
```python
# Spliting target variable and independent variables
X = data.drop(['PRICE'], axis = 1)
y = data['PRICE']
```

In [18]:
```python
# Splitting to training and testing data

from sklearn.model_selection import train_test_split
X_train, X_test, y_train, y_test = train_test_split(X,y, test_size = 0.3,
```

Linear regression

Training the model

In [19]:
```
# Import library for Linear Regression
from sklearn.linear_model import LinearRegression

# Create a Linear regressor
lm = LinearRegression()

# Train the model using the training sets
lm.fit(X_train, y_train)
```

Out[19]: LinearRegression()

In [20]:
```
# Value of y intercept
lm.intercept_
```

Out[20]: 36.357041376595205

In [21]:
```
#Converting the coefficient values to a dataframe
coeffcients = pd.DataFrame([X_train.columns,lm.coef_]).T
coeffcients = coeffcients.rename(columns={0: 'Attribute', 1: 'Coefficient
coeffcients
```

Out[21]:

	Attribute	Coefficients
0	CRIM	-0.12257
1	ZN	0.055678
2	INDUS	-0.008834
3	CHAS	4.693448
4	NOX	-14.435783
5	RM	3.28008
6	AGE	-0.003448
7	DIS	-1.552144
8	RAD	0.32625
9	TAX	-0.014067
10	PTRATIO	-0.803275
11	B	0.009354
12	LSTAT	-0.523478

Model Evaluation

In [22]:
```
# Model prediction on train data
y_pred = lm.predict(X_train)
```

In [23]:
```
# Model Evaluation
print('R^2:',metrics.r2_score(y_train, y_pred))
print('Adjusted R^2:',1 - (1-metrics.r2_score(y_train, y_pred))*(len(y_tra
print('MAE:',metrics.mean_absolute_error(y_train, y_pred))
print('MSE:',metrics.mean_squared_error(y_train, y_pred))
print('RMSE:',np.sqrt(metrics.mean_squared_error(y_train, y_pred)))
```

```
R^2: 0.7465991966746854
Adjusted R^2: 0.736910342429894
MAE: 3.08986109497113
MSE: 19.07368870346903
RMSE: 4.367343437774162
```

R^2 : It is a measure of the linear relationship between X and Y. It is interpreted as the proportion of the variance in the dependent variable that is predictable from the independent variable.

Adjusted R^2 :The adjusted R-squared compares the explanatory power of regression models that contain different numbers of predictors.

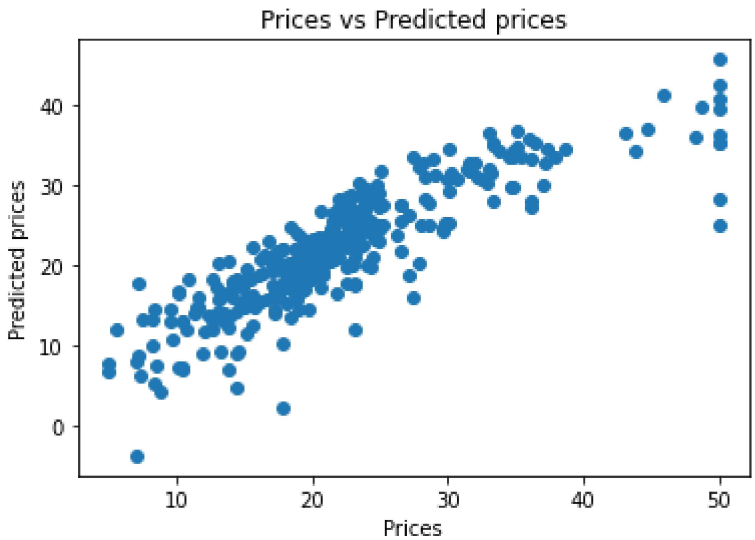
MAE : It is the mean of the absolute value of the errors. It measures the difference between two continuous variables, here actual and predicted values of y.

MSE: The mean square error (MSE) is just like the MAE, but squares the difference before summing them all instead of using the absolute value.

RMSE: The mean square error (MSE) is just like the MAE, but squares the difference before summing them all instead of using the absolute value.

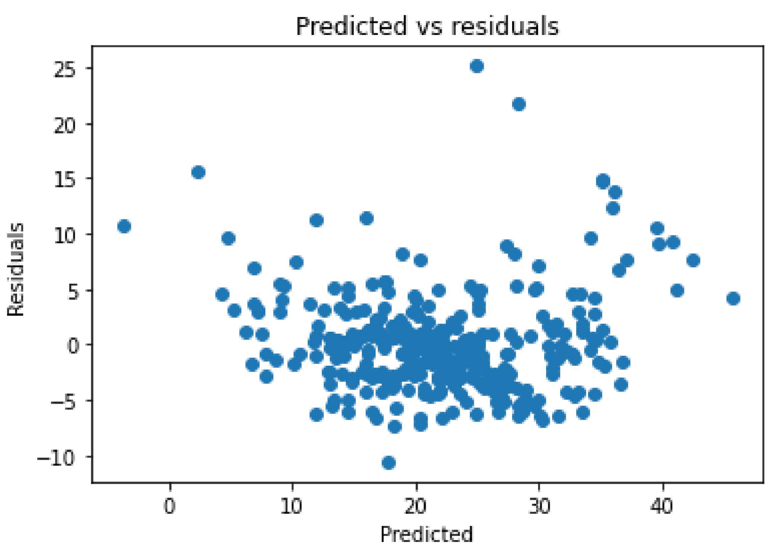
In [24]:

```
# Visualizing the differences between actual prices and predicted values
plt.scatter(y_train, y_pred)
plt.xlabel("Prices")
plt.ylabel("Predicted prices")
plt.title("Prices vs Predicted prices")
plt.show()
```



In [25]:

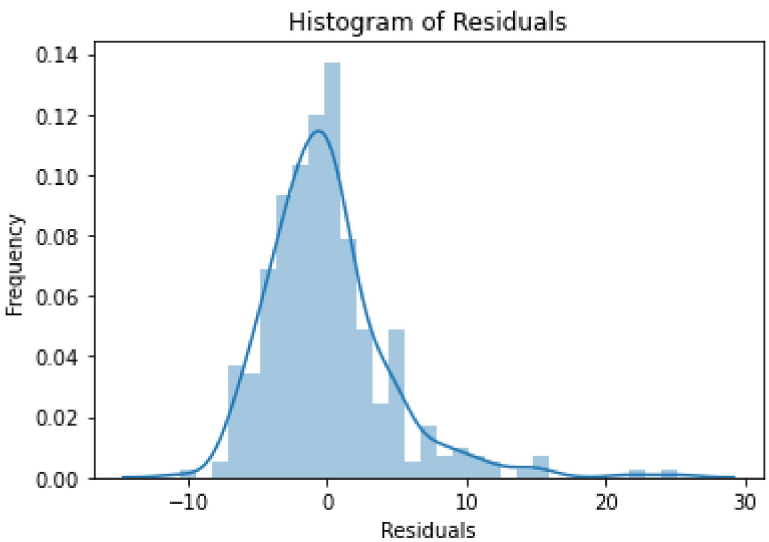
```
# Checking residuals
plt.scatter(y_pred,y_train-y_pred)
plt.title("Predicted vs residuals")
plt.xlabel("Predicted")
plt.ylabel("Residuals")
plt.show()
```



There is no pattern visible in this plot and values are distributed equally around zero. So Linearity assumption is satisfied

In [26]:
```
# Checking Normality of errors
sns.distplot(y_train-y_pred)
plt.title("Histogram of Residuals")
plt.xlabel("Residuals")
plt.ylabel("Frequency")
plt.show()
```

```
C:\Users\admin\anaconda3\lib\site-packages\seaborn\distributions.py:2557: FutureWarning: `distplot` is a deprecated function and will be removed in a future version. Please adapt your code to use either `displot` (a figure-level function with similar flexibility) or `histplot` (an axes-level function for histograms).
  warnings.warn(msg, FutureWarning)
```



Here the residuals are normally distributed. So normality assumption is satisfied

For test data

In [27]:
```
# Predicting Test data with the model
y_test_pred = lm.predict(X_test)
```

In [28]:
```
# Model Evaluation
acc_linreg = metrics.r2_score(y_test, y_test_pred)
print('R^2:', acc_linreg)
print('Adjusted R^2:',1 - (1-metrics.r2_score(y_test, y_test_pred))*(len(y
print('MAE:',metrics.mean_absolute_error(y_test, y_test_pred))
print('MSE:',metrics.mean_squared_error(y_test, y_test_pred))
print('RMSE:',np.sqrt(metrics.mean_squared_error(y_test, y_test_pred)))
```

```
R^2: 0.7121818377409195
Adjusted R^2: 0.6850685326005713
MAE: 3.8590055923707407
MSE: 30.053993307124127
RMSE: 5.482152251362974
```

Here the model evaluations scores are almost matching with that of train data. So the model is not overfitting.

Random Forest Regressor

Train the model

In [29]:
```
# Import Random Forest Regressor
from sklearn.ensemble import RandomForestRegressor

# Create a Random Forest Regressor
reg = RandomForestRegressor()

# Train the model using the training sets
reg.fit(X_train, y_train)
```

Out[29]:
```
RandomForestRegressor()
```

Model Evaluation

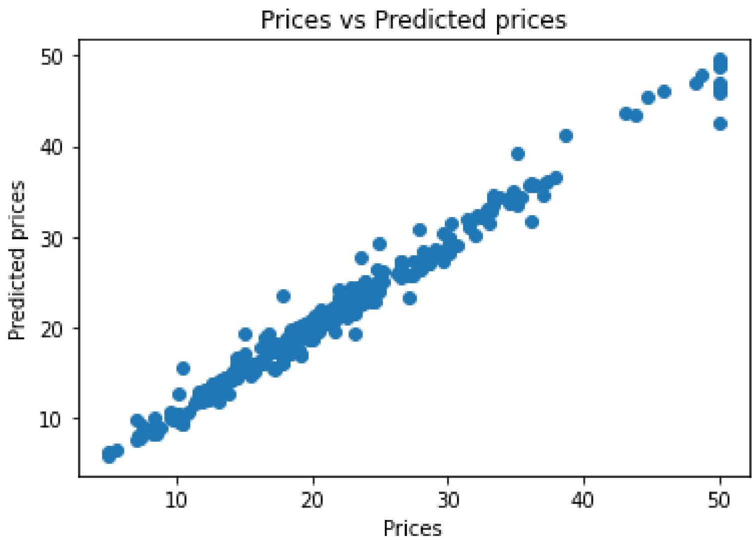
In [30]:
```
# Model prediction on train data
y_pred = reg.predict(X_train)
```

In [31]:
```
# Model Evaluation
print('R^2:',metrics.r2_score(y_train, y_pred))
print('Adjusted R^2:',1 - (1-metrics.r2_score(y_train, y_pred))*(len(y_tra
print('MAE:',metrics.mean_absolute_error(y_train, y_pred))
print('MSE:',metrics.mean_squared_error(y_train, y_pred))
print('RMSE:',np.sqrt(metrics.mean_squared_error(y_train, y_pred)))
```

```
R^2: 0.9785878017207631
Adjusted R^2: 0.9777691000218511
MAE: 0.8499576271186446
MSE: 1.611713929378534
RMSE: 1.2695329571848595
```

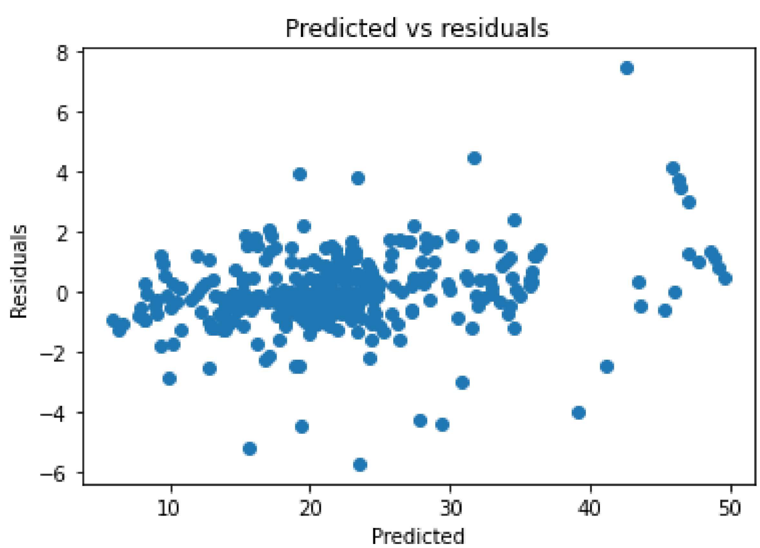
In [32]:

```
# Visualizing the differences between actual prices and predicted values
plt.scatter(y_train, y_pred)
plt.xlabel("Prices")
plt.ylabel("Predicted prices")
plt.title("Prices vs Predicted prices")
plt.show()
```



In [33]:

```
# Checking residuals
plt.scatter(y_pred,y_train-y_pred)
plt.title("Predicted vs residuals")
plt.xlabel("Predicted")
plt.ylabel("Residuals")
plt.show()
```



For test data

In [34]:
```
# Predicting Test data with the model
y_test_pred = reg.predict(X_test)
```

In [35]:
```
# Model Evaluation
acc_rf = metrics.r2_score(y_test, y_test_pred)
print('R^2:', acc_rf)
print('Adjusted R^2:',1 - (1-metrics.r2_score(y_test, y_test_pred))*(len(y
print('MAE:',metrics.mean_absolute_error(y_test, y_test_pred))
print('MSE:',metrics.mean_squared_error(y_test, y_test_pred))
print('RMSE:',np.sqrt(metrics.mean_squared_error(y_test, y_test_pred)))
```

```
R^2: 0.8302539268272174
Adjusted R^2: 0.8142633547167379
MAE: 2.5325328947368426
MSE: 17.724897230263167
RMSE: 4.210094681864431
```

XGBoost Regressor

Training the model

In [36]:
```
# Import XGBoost Regressor
from xgboost import XGBRegressor

#Create a XGBoost Regressor
reg = XGBRegressor()

# Train the model using the training sets
reg.fit(X_train, y_train)
```

```
---------------------------------------------------------------------------
ModuleNotFoundError                       Traceback (most recent call last)
<ipython-input-36-c5d0a9b9906d> in <module>
      1 # Import XGBoost Regressor
----> 2 from xgboost import XGBRegressor
      3 
      4 #Create a XGBoost Regressor
      5 reg = XGBRegressor()

ModuleNotFoundError: No module named 'xgboost'
```

max_depth (int) – Maximum tree depth for base learners.

learning_rate (float) – Boosting learning rate (xgb's "eta")

n_estimators (int) – Number of boosted trees to fit.

gamma (float) – Minimum loss reduction required to make a further partition on a leaf node of the tree.

min_child_weight (int) – Minimum sum of instance weight(hessian) needed in a child.

subsample (float) – Subsample ratio of the training instance.

colsample_bytree (float) – Subsample ratio of columns when constructing each tree.

objective (string or callable) – Specify the learning task and the corresponding learning objective or a custom objective function to be used (see note below).

nthread (int) – Number of parallel threads used to run xgboost. (Deprecated, please use n_jobs)

scale_pos_weight (float) – Balancing of positive and negative weights

Model Evaluation

In [37]:

```
# Model prediction on train data
y_pred = reg.predict(X_train)
```

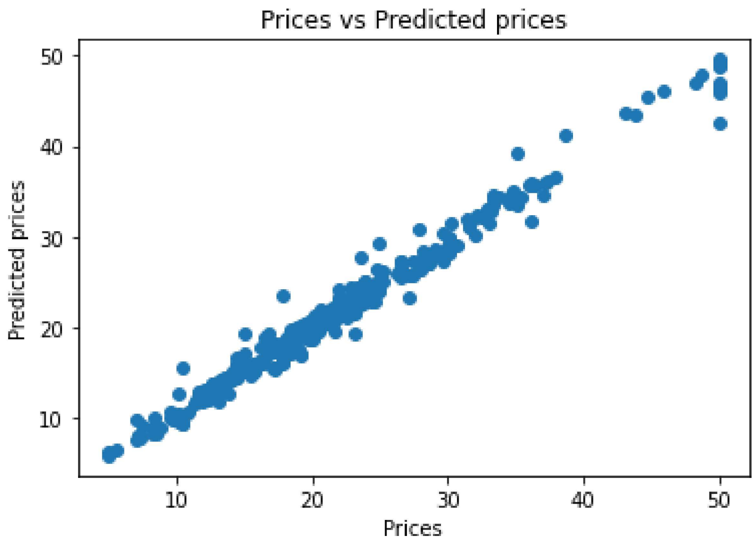
In [38]:

```
# Model Evaluation
print('R^2:',metrics.r2_score(y_train, y_pred))
print('Adjusted R^2:',1 - (1-metrics.r2_score(y_train, y_pred))*(len(y_tr
print('MAE:',metrics.mean_absolute_error(y_train, y_pred))
print('MSE:',metrics.mean_squared_error(y_train, y_pred))
print('RMSE:',np.sqrt(metrics.mean_squared_error(y_train, y_pred)))
```

```
R^2: 0.9785878017207631
Adjusted R^2: 0.9777691000218511
MAE: 0.8499576271186446
MSE: 1.611713929378534
RMSE: 1.2695329571848595
```

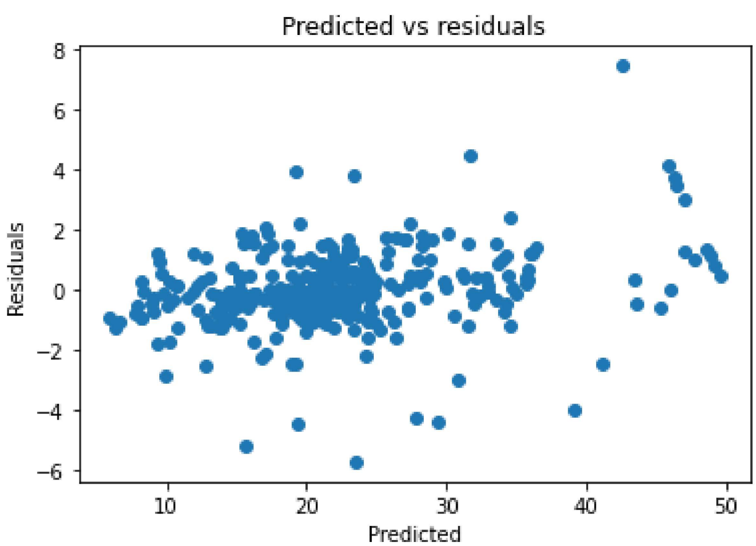
In [39]:

```
# Visualizing the differences between actual prices and predicted values
plt.scatter(y_train, y_pred)
plt.xlabel("Prices")
plt.ylabel("Predicted prices")
plt.title("Prices vs Predicted prices")
plt.show()
```



In [40]:

```
# Checking residuals
plt.scatter(y_pred,y_train-y_pred)
plt.title("Predicted vs residuals")
plt.xlabel("Predicted")
plt.ylabel("Residuals")
plt.show()
```



For test data

In [41]:
```
#Predicting Test data with the model
y_test_pred = reg.predict(X_test)
```

In [42]:
```
# Model Evaluation
acc_xgb = metrics.r2_score(y_test, y_test_pred)
print('R^2:', acc_xgb)
print('Adjusted R^2:',1 - (1-metrics.r2_score(y_test, y_test_pred))*(len(
print('MAE:',metrics.mean_absolute_error(y_test, y_test_pred))
print('MSE:',metrics.mean_squared_error(y_test, y_test_pred))
print('RMSE:',np.sqrt(metrics.mean_squared_error(y_test, y_test_pred)))
```

```
R^2: 0.8302539268272174
Adjusted R^2: 0.8142633547167379
MAE: 2.5325328947368426
MSE: 17.724897230263167
RMSE: 4.210094681864431
```

SVM Regressor

In [43]:
```
# Creating scaled set to be used in model to improve our results
from sklearn.preprocessing import StandardScaler
sc = StandardScaler()
X_train = sc.fit_transform(X_train)
X_test = sc.transform(X_test)
```

Train the model

In [44]:
```
# Import SVM Regressor
from sklearn import svm

# Create a SVM Regressor
reg = svm.SVR()
```

In [45]:
```
# Train the model using the training sets
reg.fit(X_train, y_train)
```

Out[45]:
```
SVR()
```

C : float, optional (default=1.0): The penalty parameter of the error term. It controls the trade off between smooth decision boundary and classifying the training points correctly.

kernel : string, optional (default='rbf'): kernel parameters selects the type of hyperplane used to separate the data. It must be one of 'linear', 'poly', 'rbf', 'sigmoid', 'precomputed' or a callable.

degree : int, optional (default=3): Degree of the polynomial kernel function ('poly'). Ignored by all other kernels.

gamma : float, optional (default='auto'): It is for non linear hyperplanes. The higher the gamma value it tries to exactly fit the training data set. Current default is 'auto' which uses 1 / n_features.

coef0 : float, optional (default=0.0): Independent term in kernel function. It is only significant in

Model Evaluation

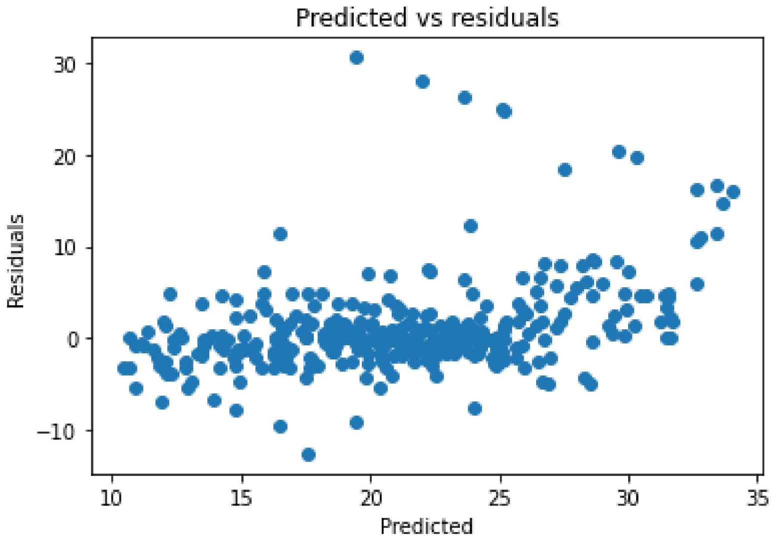
In [46]:
```
# Model prediction on train data
y_pred = reg.predict(X_train)
```

In [47]:
```
# Model Evaluation
print('R^2:',metrics.r2_score(y_train, y_pred))
print('Adjusted R^2:',1 - (1-metrics.r2_score(y_train, y_pred))*(len(y_tr
print('MAE:',metrics.mean_absolute_error(y_train, y_pred))
print('MSE:',metrics.mean_squared_error(y_train, y_pred))
print('RMSE:',np.sqrt(metrics.mean_squared_error(y_train, y_pred)))
```

```
R^2: 0.6419097248941195
Adjusted R^2: 0.628218037904777
MAE: 2.9361501059460284
MSE: 26.953752101332935
RMSE: 5.191700309275655
```

In [48]:
```
# Visualizing the differences between actual prices and predicted values
plt.scatter(y_train, y_pred)
plt.xlabel("Prices")
plt.ylabel("Predicted prices")
plt.title("Prices vs Predicted prices")
plt.show()
```

In [49]:
```
# Checking residuals
plt.scatter(y_pred,y_train-y_pred)
plt.title("Predicted vs residuals")
plt.xlabel("Predicted")
plt.ylabel("Residuals")
plt.show()
```



For test data

In [50]:
```
# Predicting Test data with the model
y_test_pred = reg.predict(X_test)
```

In [51]:
```
# Model Evaluation
acc_svm = metrics.r2_score(y_test, y_test_pred)
print('R^2:', acc_svm)
print('Adjusted R^2:',1 - (1-metrics.r2_score(y_test, y_test_pred))*(len(
print('MAE:',metrics.mean_absolute_error(y_test, y_test_pred))
print('MSE:',metrics.mean_squared_error(y_test, y_test_pred))
print('RMSE:',np.sqrt(metrics.mean_squared_error(y_test, y_test_pred)))
```

```
R^2: 0.5900158460478174
Adjusted R^2: 0.5513941503856553
MAE: 3.7561453553021673
MSE: 42.81057499010247
RMSE: 6.542979060802691
```

Evaluation and comparision of all the models

In [52]:

```
models = pd.DataFrame({
    'Model': ['Linear Regression', 'Random Forest', 'XGBoost', 'Support Ve
    'R-squared Score': [acc_linreg*100, acc_rf*100, acc_xgb*100, acc_svm*1
models.sort_values(by='R-squared Score', ascending=False)
```

Out[52]:

	Model	R-squared Score
1	Random Forest	83.025393
2	XGBoost	83.025393
0	Linear Regression	71.218184
3	Support Vector Machines	59.001585