Assignment 2: House price predicting

N4A, 2018/10/24

1 Task Description

Use the following dataset to do house price predicting work.

1. https://www.kaggle.com/vikrishnan/boston-house-prices (https://www.kaggle.com/vikrishnan/boston-house-prices)
2. https://github.com/datasets/house-prices-uk (https://github.com/datasets/house-prices-uk)

Details: design a model to do house price predicting work. Linear Regression models including basic linear model based on polynomial, Ridge Regression, Lasso Regression and regression model based Decision Tree must be implemented. Regression models based on SVM and Deep Learning is optional.

2 Data Acquisition and Description

2.1 Boston house prices

Although we can download the data from https://www.kaggle.com/vikrishnan/boston-house-prices (https://www.kaggle.com/vikrishnan/boston-house-prices), here we use API $sklearn.datasets.load\_boston$ to simplify this process.

In [31]:

```
%matplotlib inline
import numpy as np
import matplotlib.pyplot as plt
from sklearn import datasets
```

In [32]:

```
boston = datasets.load_boston()
boston_X, boston_y = np.asarray(boston.data), np.asarray(boston.target)
boston_X = boston_X / np.max(boston_X, axis=0)
# show data shape
print boston_X.shape, boston_y.shape
```

```
(506, 13) (506,)
```

The dataset consists of 506 samples, the meaning of each dimension of feature x is summarized as bellow.

1. CRIM: per capita crime rate by town
2. ZN: proportion of residential land zoned for lots over 25,000 sq.ft.
3. INDUS: proportion of non-retail business acres per town
4. CHAS: Charles River dummy variable (= 1 if tract bounds river; 0 otherwise)
5. NOX: nitric oxides concentration (parts per 10 million)
   1https://archive.ics.uci.edu/ml/datasets/Housing (https://archive.ics.uci.edu/ml/datasets/Housing)
   123 20.2. Load the Dataset 124
6. RM: average number of rooms per dwelling
7. AGE: proportion of owner-occupied units built prior to 1940
8. DIS: weighted distances to five Boston employment centers
9. RAD: index of accessibility to radial highways
10. TAX: full-value property-tax rate per $10,000
11. PTRATIO: pupil-teacher ratio by town
12. B: 1000(Bk−0.63)2 where Bk is the proportion of blacks by town
13. LSTAT: % lower status of the population

y is Median value of owner-occupied homes in $1000s

In [33]:

```
# split train and test sets
from sklearn.model_selection import train_test_split

boston_x_train, boston_x_test, boston_y_train, boston_y_test = train_test_split(boston_X, boston
_y, test_size=0.1, shuffle=False)
```

2.2 UK house prices

UK house prices dataset can be aquired here(https://github.com/datasets/house-prices-uk (https://github.com/datasets/house-prices-uk)) and then we use pandas to load the csv data.

In [34]:

```
import pandas as pd
```

In [35]:

```
file_path = './uk_house_price.csv'
uk = pd.read_csv(file_path)
```

In [36]:

```
# check data size
print uk.shape
```

```
(261, 9)
```

In [37]:

```
# show 10 examples
uk[:10]
```

Out[37]:

	Date	Price (All)	Change (All)	Price (New)	Change (New)	Price (Modern)	Change (Modern)	Price (Older)	Change (Older)
0	1952-11-01	1891	0.0	2107	0.0	2020	0.0	1524	0.0
1	1953-02-01	1891	0.0	2107	0.0	2002	0.0	1542	0.0
2	1953-05-01	1891	0.0	2107	0.0	2002	0.0	1542	0.0
3	1953-08-01	1881	0.0	2117	0.0	2002	0.0	1524	0.0
4	1953-11-01	1872	-1.0	2117	0.5	1975	-2.2	1542	1.2
5	1954-02-01	1863	-1.5	2117	0.5	1957	-2.2	1524	-1.2
6	1954-05-01	1872	-1.0	2117	0.5	1984	-0.9	1515	-1.7
7	1954-08-01	1863	-1.0	2127	0.5	1948	-2.7	1524	0.0
8	1954-11-01	1853	-1.0	2127	0.5	1939	-1.8	1515	-1.7
9	1955-02-01	1900	2.0	2167	2.4	1984	1.4	1569	2.9

In [38]:

```
uk.dtypes
```

Out[38]:

```
Date                object
Price (All)          int64
Change (All)       float64
Price (New)          int64
Change (New)       float64
Price (Modern)       int64
Change (Modern)    float64
Price (Older)        int64
Change (Older)     float64
dtype: object
```

3 Algorithms introduction and implementation

3.1 Basic Linear Regression

The simplest linear model for regression is one that involves a linear combination of the input variables. Here, we donate input variables or features as $x = (x_1, \ldots, x_D)^T$ and then the target output is calculated as follows:

$$y(x, w) = w_0 + w_1 x_1 + \ldots + w_d x_D$$

Obviously, this imposes significant limitations on the model. For example, the model can not fit a ploynomial curve with input x being a scalar. We therefore extend the model by considering linear combinations of fixed nonlinear functions of the input variables, of the form

$$y(x, w) = w_0 + \sum_{j=1}^{M-1} w_j \phi_j(x)$$

where $\phi_j(x)$ are known as basic functions. In boston house price predicting problem, the input x is a 13 dimension vector. So we can simply define $\phi_j(x) = x_j$. On the other hand, in UK house price predicting problem, the input x is just a scalar. This is a polinomial curve fitting problem. So we can define $\phi_j(x) = x^j$ and then the curve could be well fitted by the polynomial function.

Besides, We can define $\phi_0(x) = 1$ so that

$$y(x, w) = \sum_{j=0}^{M-1} w_j \phi_j(x) = w^T \phi(x)$$

Then， Basic Linear Regression estimates the parameters by minimizing the residual sum of squares between the observed responses, donated as t, in the dataset and the responses($y(x, w)$) predicted by the linear approximation. Mathematically it solves a problem of the form:

$$\min_w ||y(x, w) - t||_2^2$$

Here, we use sklearn lib to implement the algrithm and the codes are as follows:

In [39]:

```
from sklearn import linear_model as lm


# define some basic functions
def rmse(preds, target):
    return np.sqrt(np.sum(np.square(preds - target))/ len(preds))

def plot_regression(preds, target, x_axis=None, name='regression'):
    plt.figure()
    if x_axis is None:
        plt.plot(preds, label='predict price')
        plt.plot(target, label='real price')
    else:
        plt.plot(x_axis, preds, label='predict price')
        plt.plot(x_axis, target, label='real price')
    plt.legend()
    plt.title(name)
    plt.show()

def model_train_test(model, x_train, y_train, x_test, y_test, plot_train=False, plot_x_axis=None
):
    model.fit(boston_x_train, boston_y_train)
    # test on boston data
    test_preds_price = model.predict(x_test)
    print('rmse on test set of boston data: {:.6f}'.format(rmse(test_preds_price, y_test)))
    # plot result
    plot_regression(test_preds_price, y_test, name='Regression result on test set')
    if plot_train:
        train_preds  = model.predict(x_train)
        plot_regression(train_preds, y_train,name='Regression result on train set')
```

In [40]:

```
# check basic linear regression model on boston set
print('check basic linear regression model on boston data set: ')
model_train_test(lm.LinearRegression(), boston_x_train, boston_y_train, boston_x_test, boston_y_
test)
```

```
check basic linear regression model on boston data set:
rmse on test set of boston data: 3.287687
```

3.2 Ridge Regression

Ridge regression addresses some of the problems of Ordinary Least Squares by imposing a penalty on the size of coefficients. The ridge coefficients minimize a penalized residual sum of squares,

$$\min_w ||y(x,w) - t||_2^2 + \alpha||w||_2^2$$

Here, $\alpha \geq 0$ is a complexity parameter that controls the amount of shrinkage: the larger the value of , the greater the amount of shrinkage and thus the coefficients become more robust to collinearity.

The sklearn codes are as follows:

In [41]:

```
print('check Ridge regression model on boston data set: ')
model_train_test(lm.Ridge(alpha=.5), boston_x_train, boston_y_train, boston_x_test, boston_y_test)
```

```
check Ridge regression model on boston data set:
rmse on test set of boston data: 3.190611
```

3.3 Lasso Regression

The Lasso is a linear model that estimates sparse coefficients. It is useful in some contexts due to its tendency to prefer solutions with fewer parameter values, effectively reducing the number of variables upon which the given solution is dependent. For this reason, the Lasso and its variants are fundamental to the field of compressed sensing. Under certain conditions, it can recover the exact set of non-zero weights (see Compressive sensing: tomography reconstruction with L1 prior (Lasso)).

Mathematically, it consists of a linear model trained with ℓ_1 prior as regularizer. The objective function to minimize is:

$$\min_w \frac{1}{2n_{samples}} ||y(x,w) - t||_2^2 + \alpha||w||_1$$

The lasso estimate thus solves the minimization of the least-squares penalty with $\alpha||w||_1$ added, where $||w||_1$ is a constant and is the ℓ_1-norm of the parameter vector.

The sklearn codes are as follows:

In [42]:

```
print('check Lasso regression model on boston data set: ')
model_train_test(lm.Lasso(alpha=.02), boston_x_train, boston_y_train, boston_x_test, boston_y_test)
```

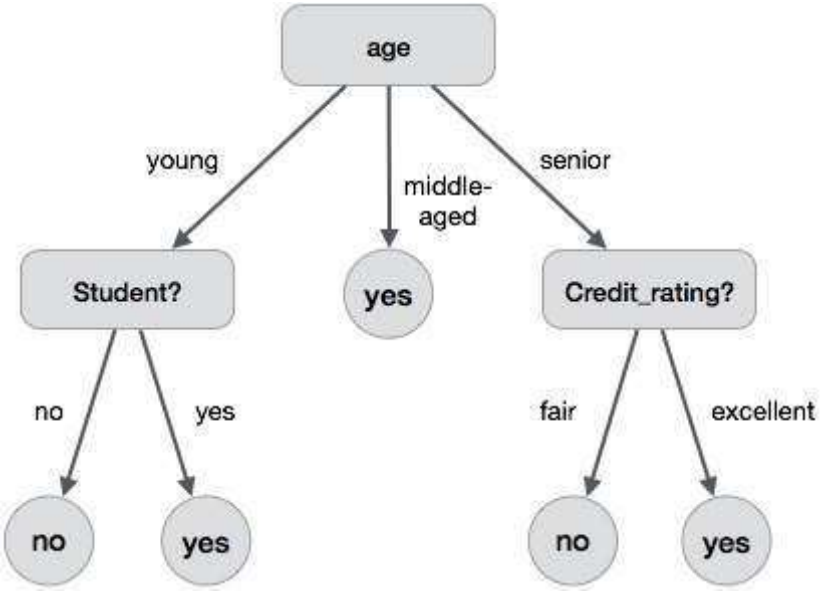
```
check Lasso regression model on boston data set:
rmse on test set of boston data: 3.108968
```

3.4 Decision Tree Regression

Decision Trees (DTs) are a non-parametric supervised learning method used for classification and regression. The goal is to create a model that predicts the value of a target variable by learning simple decision rules inferred from the data features.

A decision tree is a structure that includes a root node, branches, and leaf nodes. Each internal node denotes a test on an attribute, each branch denotes the outcome of a test, and each leaf node holds a class label. The topmost node in the tree is the root node.

The following decision tree is for the concept buy_computer that indicates whether a customer at a company is likely to buy a computer or not. Each internal node represents a test on an attribute. Each leaf node represents a class.



As for regression problem, in the example below, decision trees learn from data to approximate a sine curve with a set of if-then-else decision rules. The deeper the tree, the more complex the decision rules and the fitter the model.

Decision Tree Regression

The sklearn implementation codes of decision tree regression are as below:

In [43]:

```
from sklearn.tree import DecisionTreeRegressor
print('check Decision Tree Regression model with depth being 5 on boston data set: ')
model_train_test(DecisionTreeRegressor(max_depth=5), boston_x_train, boston_y_train, boston_x_test, boston_y_test)
```

```
check Decision Tree Regression model with depth being 5 on boston data set:
rmse on test set of boston data: 4.297973
```

In [44]:

```
print('check Decision Tree Regression model with depth being 2 on boston data set: ')
model_train_test(DecisionTreeRegressor(max_depth=2), boston_x_train, boston_y_train, boston_x_test, boston_y_test)
```

```
check Decision Tree Regression model with depth being 2 on boston data set:
rmse on test set of boston data: 4.147044
```

3.5 SVM Regression

SVM is also known as larger margin classifer which usually has a smaller generalization error. When a SVM classifier is trained, only the data points that are nearest to decision boundary are left for classify a new data point. These remaining data points called support vectors.

Of course, we can extend support vector machines to regression problems while at the same time preserving the property of sparseness. In ridge regression with the output being a scalar, the error function is

$$||y(x,w)-t||_2^2+\alpha||w||_2^2=\sum_{n=1}^{N}(y_n-t_n)^2+\alpha||w||_2^2$$

To obtain sparse solutions, the quadratic error function is replaced by an ϵ-insensitive error function that is given by

$$E_\epsilon(y(x)-t)=\begin{cases}0, & if\ |y(x)-t|<\epsilon\\ |y(x)-t|-\epsilon, & otherwise\end{cases}$$

We therefore minimize a regularized error function given by

$$C\sum_{n=1}^{N}E_\epsilon(y(x_n)-t_n)+||w||^2$$

By convention, the (inverse) regularization parameter, denoted by C, appears in front of the error term.

We can re-express the optimization problem by introducing slack varibles $\zeta_i,\zeta_i^*\geq 0$. Then the error optimization target can be written as

$$\begin{aligned}\min_{w,b,\zeta,\zeta^*}\ &\frac{1}{2}w^Tw+C\sum_{i=1}^{n}(\zeta_i+\zeta_i^*)\\ \text{subject to }&t_i-y(x_i)\leq\varepsilon+\zeta_i,\\ &y(x_i)-t_i\leq\varepsilon+\zeta_i^*,\\ &\zeta_i,\zeta_i^*\geq 0,i=1,\dots,n\end{aligned}$$

Its dual form is

$$\begin{aligned}\min_{\alpha,\alpha^*}\ &\frac{1}{2}(\alpha-\alpha^*)^TQ(\alpha-\alpha^*)+\varepsilon e^T(\alpha+\alpha^*)-y^T(\alpha-\alpha^*)\\ &\text{subject to }e^T(\alpha-\alpha^*)=0\\ &0\leq\alpha_i,\alpha_i^*\leq C,i=1,\dots,n\end{aligned}$$

where e is the vector of all ones, $C>0$ is the upper bound and the same as the C above, Q is an n by n positive semidefinite matrix, $Q_{ij}\equiv K(x_i,x_j)=\phi(x_i)^T\phi(x_j)$ is the kernel. Here training vectors are implicitly mapped into a higher (maybe infinite) dimensional space by the function .

sklearn codes are as follows:

In [45]:

```
from sklearn.svm import SVR

print('check SVM Regression model with linear kernel on boston data set: ')
model_train_test(SVR(kernel='linear'), boston_x_train, boston_y_train, boston_x_test, boston_y_test)
```

```
check SVM Regression model with linear kernel on boston data set:
rmse on test set of boston data: 3.726680
```

In [46]:

```
print('check SVM Regression model with sigmoid kernel on boston data set: ')
model_train_test(SVR(kernel='sigmoid'), boston_x_train, boston_y_train, boston_x_test, boston_y_test)
```

```
check SVM Regression model with sigmoid kernel on boston data set:
rmse on test set of boston data: 3.898799
```

In [47]:

```
print('check SVM Regression model with rbf kernel on boston data set: ')
model_train_test(SVR(kernel='rbf'), boston_x_train, boston_y_train, boston_x_test, boston_y_test
)
```

```
check SVM Regression model with rbf kernel on boston data set:
rmse on test set of boston data: 4.032612
```

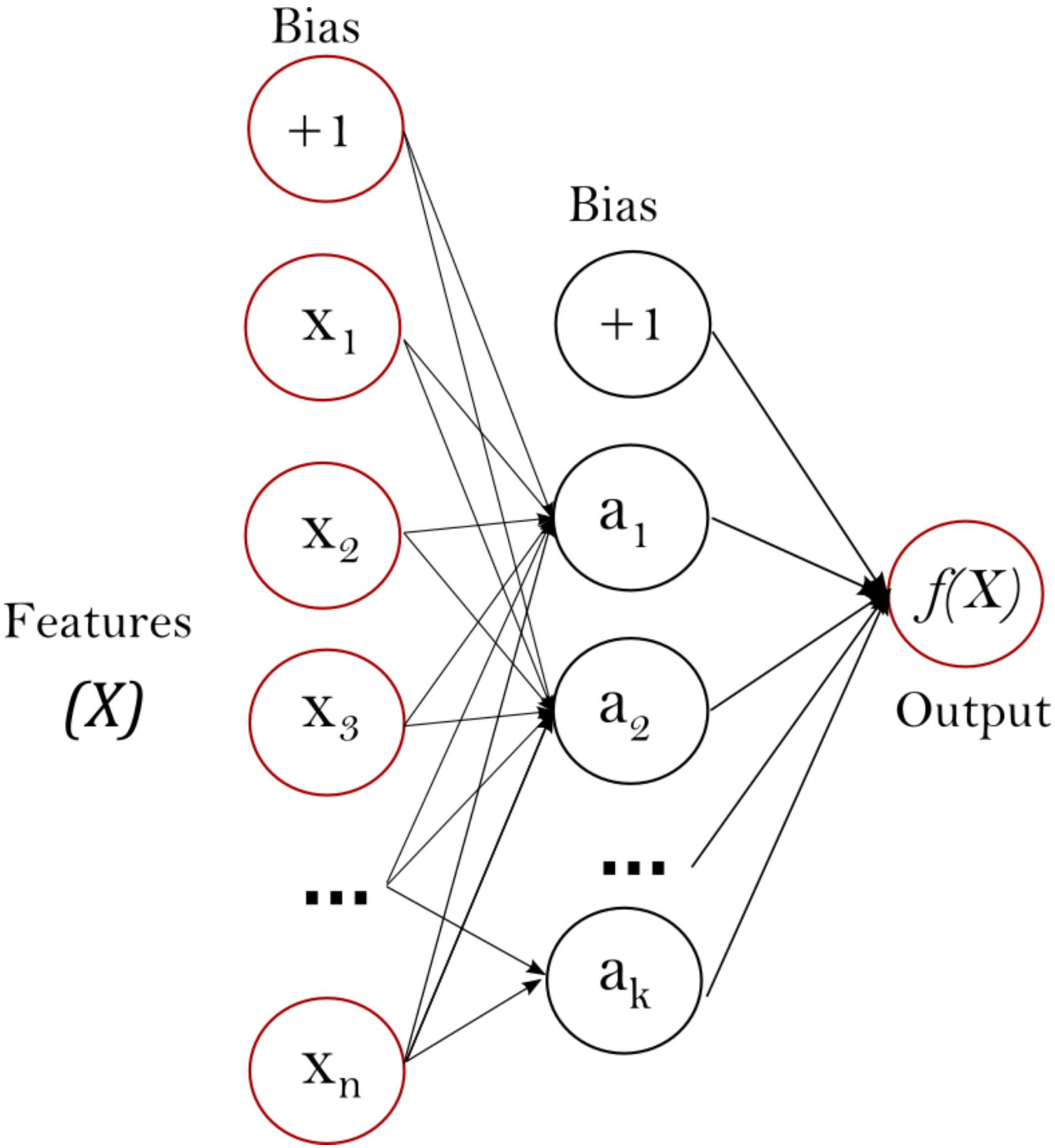
In [48]:

```
print('check SVM Regression model with poly kernel on boston data set: ')
model_train_test(SVR(kernel='poly'), boston_x_train, boston_y_train, boston_x_test, boston_y_tes
t)
```

```
check SVM Regression model with poly kernel on boston data set:
rmse on test set of boston data: 4.114710
```

3.6 Nerual network regression

Here we use the simple multi layer perceptron model to do this work. Multi-layer Perceptron (MLP) is a supervised learning algorithm that learns a function $f(\cdot) : R^m \rightarrow R^o$ by training on a dataset, where m is the number of dimensions for input and o is the number of dimensions for output while in this task o is 1. Given a set of features $X = x_1, x_2, \ldots, x_m$and a target , it can learn a non-linear function approximator for either classification or regression. It is different from logistic regression, in that between the input and the output layer, there can be one or more non-linear layers, called hidden layers. Figure 1 shows a one hidden layer MLP with scalar output.



The leftmost layer, known as the input layer, consists of a set of neurons $\{x_i|x_1, x_2, \ldots, x_m\}$ representing the input features. Each neuron in the hidden layer transforms the values from the previous layer with a weighted linear summation $w_1x_1 + w_2x_2 + \ldots + w_mx_m$, followed by a non-linear activation function $g(\cdot) : R \rightarrow R$ - like the hyperbolic tan function. The output layer receives the values from the last hidden layer and transforms them into output values.

We also use sklearn to implement it.

In [49]:

```
from sklearn.neural_network import MLPRegressor

print('check MLP Regression model with relu activation on boston data set: ')
model_train_test(MLPRegressor(activation='relu'), boston_x_train, boston_y_train, boston_x_test,
 boston_y_test)
```

```
check MLP Regression model with relu activation on boston data set:
rmse on test set of boston data: 3.977414
```

In [50]:

```
print('check MLP Regression model with no activation on boston data set: ')
model_train_test(MLPRegressor(activation='identity'), boston_x_train, boston_y_train, boston_x_test, boston_y_test)
```

```
check MLP Regression model with no activation on boston data set:
rmse on test set of boston data: 4.010192
```

In [51]:

```
print('check MLP Regression model with two hidden layers and relu activation on boston data se
t: ')
model_train_test(MLPRegressor(hidden_layer_sizes=(50, 25, ), activation='relu'), boston_x_train,
 boston_y_train, boston_x_test, boston_y_test)
```

```
check MLP Regression model with two hidden layers and relu activation on boston da
ta set:
rmse on test set of boston data: 3.634077
```

In [52]:

```
print('check MLP Regression model with tree hidden layers and relu activation on boston data se
t: ')
model_train_test(MLPRegressor(hidden_layer_sizes=(50, 30, 20,), activation='relu'),
                 boston_x_train, boston_y_train, boston_x_test, boston_y_test)
```

```
check MLP Regression model with tree hidden layers and relu activation on boston d
ata set:
rmse on test set of boston data: 4.017490
```

4 Experiments and Conclusion

4.1 Experiments on boston house prices data set

As described above, boston house price data has 506 samples. We split 10% of the data as a test set and the left exmaples as a train set. We use RMSE to evaluate the performance of each model. The best results of each model are showed in the table below. In this experiment, the Lasso Regression outperforms the others. See section 3 for details.

	LinearR	**RidgeR**	**LassoR**	**DTR**	**SVR**	**MLPR**
RMSE	3.1089	3.1906	3.1089	4.1470	3.7266	3.6340

We can see that SVR model and MLP model with default hyper-parameters are much less effective than we thought. So, here we tune some hyper-parameters for these two models

In [110]:

```
print('check SVM Regression model with linear kernel on boston data set: ')
# under this setting, the SVR model will be much better
model_train_test(SVR(C=100, kernel='poly'), boston_x_train, boston_y_train, boston_x_test, bosto
n_y_test)
```

```
check SVM Regression model with linear kernel on boston data set:
rmse on test set of boston data: 3.066366
```

In [147]:

```
print('check MLP Regression model with two hidden layers and relu activation on boston data se
t: ')
# not easy to tune the parameter. just a little better.
model_train_test(MLPRegressor(hidden_layer_sizes=(50, 25),alpha=0.01, max_iter=200, learning_rat
e_init=0.001, activation='relu'),
                 boston_x_train, boston_y_train, boston_x_test, boston_y_test)
```

```
check MLP Regression model with two hidden layers and relu activation on boston da
ta set:
rmse on test set of boston data: 3.519582
```

4.2 Experiments on uk house prices data set

This data set includes for types of prices that are chaning over time. The four types are All, New, Modern and Older. The prices was recorded every three months. So we can change the time with index as input features to predict four types prices. Here, we first preprocess the data set to fit our requirement.

In [53]:

```
uk.columns
```

Out[53]:

```
Index([u'Date', u'Price (All)', u'Change (All)', u'Price (New)',
       u'Change (New)', u'Price (Modern)', u'Change (Modern)',
       u'Price (Older)', u'Change (Older)'],
      dtype='object')
```

In [54]:

```
price_all = uk.iloc[:, 1].values.flatten()
price_new = uk.iloc[:, 3].values.flatten()
price_modern = uk.iloc[:, 5].values.flatten()
price_older = uk.iloc[:, 7].values.flatten()
time = np.arange(len(price_all))
```

In [71]:

```
# extend time to poly features
time_f = np.stack([time, time*time], axis=-1)
```

In [72]:

```
time_f[:4]
```

Out[72]:

```
array([[0, 0],
       [1, 1],
       [2, 4],
       [3, 9]])
```

In [73]:

```
def fit_data(model, y):
    model.fit(time_f,y)
    return model.predict(time_f)
```

In [74]:

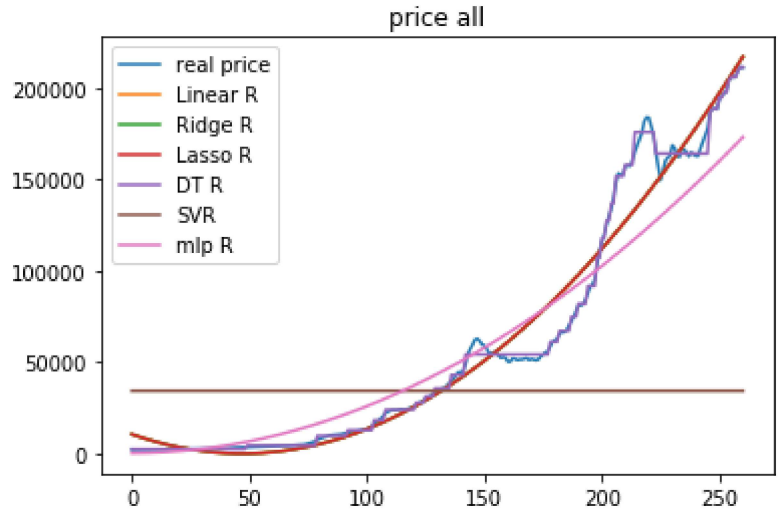
```
def fit_all_model(y, name='price'):
    lr_preds = fit_data(lm.LinearRegression(), y)
    rr_preds = fit_data(lm.Ridge(alpha=.5), y)
    lar_preds = fit_data(lm.Lasso(), y)
    dtr_preds = fit_data(DecisionTreeRegressor(max_depth=5), y)
    svr_preds = fit_data(SVR(), y)
    mlp_preds = fit_data(MLPRegressor(hidden_layer_sizes=(25, ), activation='relu'), y)

    # print rmse value
    print(' \tLieanrR\tRidgeR\tLassoR\tDTR\tSVR\tMLPR\n'
          'RMSE\t{:.1f}\t{:.1f}\t{:.1f}\t{:.1f}\t{:.1f}\t{:.1f}'.format(rmse(lr_preds, y), rmse(
rr_preds, y), rmse(lar_preds, y),
                                                                       rmse(dtr_preds, y),rmse(
svr_preds, y), rmse(mlp_preds, y)))
    # plot result
    plt.figure()
    plt.title(name)
    plt.plot(y, label='real price')
    plt.plot(lr_preds, label='Linear R')
    plt.plot(rr_preds, label='Ridge R')
    plt.plot(lar_preds, label='Lasso R')
    plt.plot(dtr_preds, label='DT R')
    plt.plot(svr_preds, label='SVR')
    plt.plot(mlp_preds, label='mlp R')
    plt.legend()
    plt.show()
```

In [75]:

```
# test all price
fit_all_model(price_all, name='price all')
```

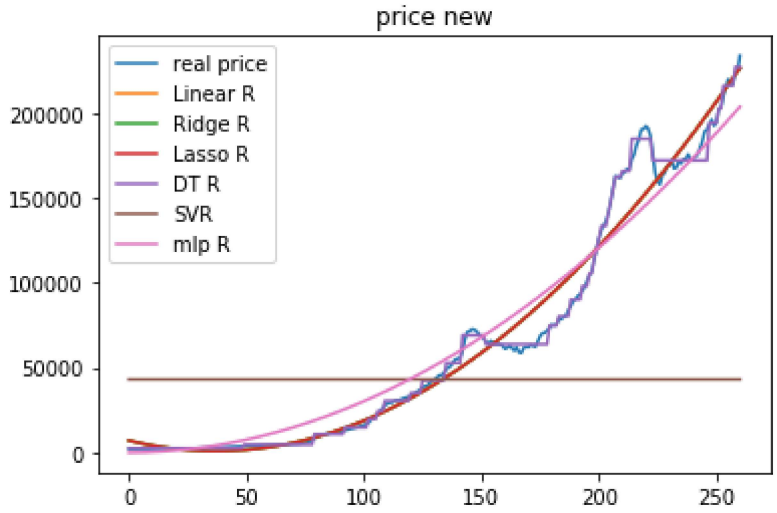
```
        LieanrR RidgeR  LassoR  DTR     SVR     MLPR
RMSE    12264.3 12264.3 12264.3 2708.1  70788.0 19498.8
```



In [76]:

```
# test new price
fit_all_model(price_new, name='price new')
```

```
        LieanrR RidgeR  LassoR  DTR     SVR     MLPR
RMSE    11604.8 11604.8 11604.8 3233.0  72281.7 14892.9
```



In [77]:

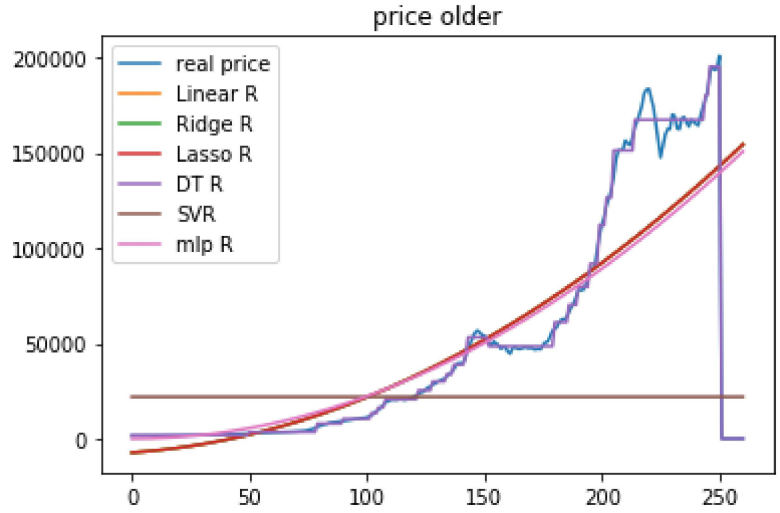
```
# test modern price
fit_all_model(price_modern, name='price modern')
```

```
        LieanrR RidgeR  LassoR  DTR     SVR     MLPR
RMSE    34627.0 34627.0 34627.0 3362.6  62233.6 34905.7
```

In [78]:

```
# test older price
fit_all_model(price_older, name='price older')
```

```
        LieanrR RidgeR  LassoR  DTR     SVR     MLPR
RMSE    37004.6 37004.6 37004.6 3314.2  66925.5 37149.6
```



All in all, Decision Tree Regression with max depth being 5 is overfitted. SVR with C(see 3.5) being default 1 is always underfitted. Linear Regression, Ridge Regression, Lasso Regression and MLP Regression are ok.

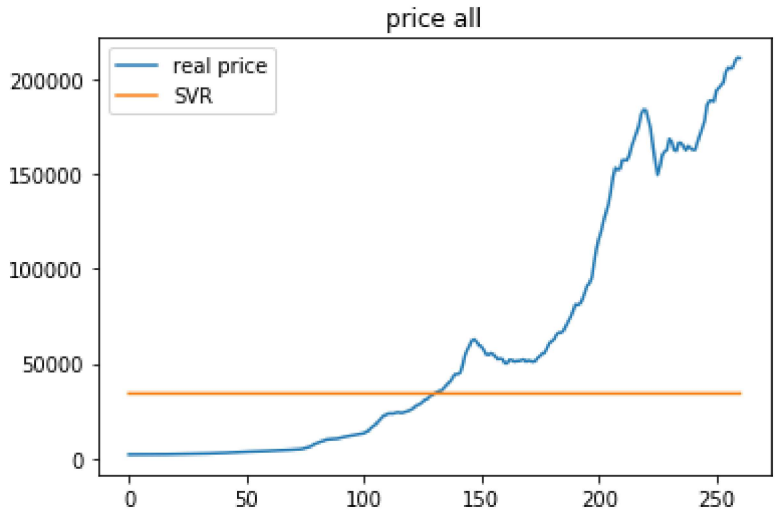
In [90]:

```
def fit_svr_model(y, name='price', C=1):
    svr_preds = fit_data(SVR(C=C), y)
    # print rmse value
    print(' \tSVR\n'
          'RMSE\t{:.1f}'.format(rmse(svr_preds, y)))
    # plot result
    plt.figure()
    plt.title(name)
    plt.plot(y, label='real price')
    plt.plot(svr_preds, label='SVR')
    plt.legend()
    plt.show()
```

In [91]:

```
# test all price
fit_svr_model(price_all, name='price all', C=1)
```

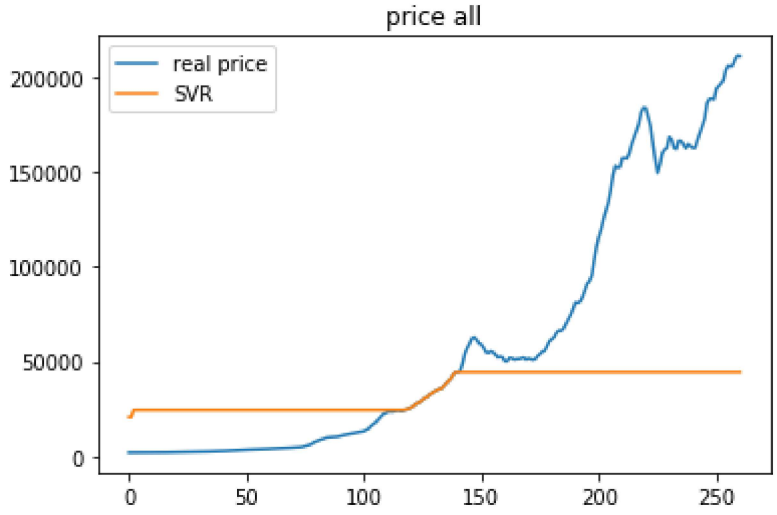
	SVR
RMSE	70788.0



In [93]:

```
# test all price
fit_svr_model(price_all, name='price all', C=1e4)
```

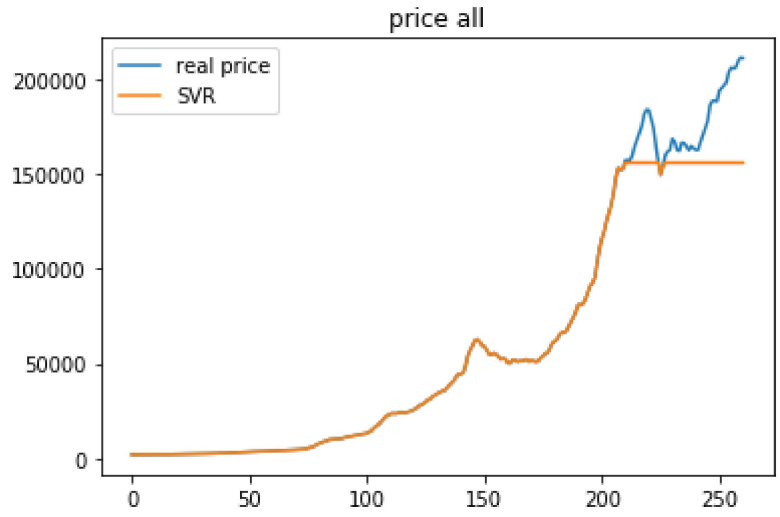
	SVR
RMSE	63856.1



In [94]:

```
# test all price
fit_svr_model(price_all, name='price all', C=1e5)
```

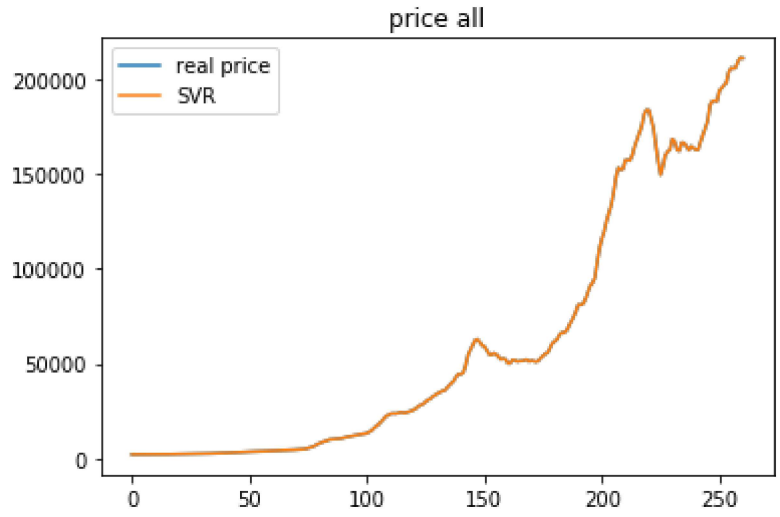
	SVR
RMSE	11699.8



In [95]:

```
# test all price
fit_svr_model(price_all, name='price all', C=1e6)
```

	SVR
RMSE	0.1



Note that C is inverse regularization parameter(see 3.5). C is penalty parameter of the error term. The larger is C, the more sensitive is the model to the error between predicted values and target values and the easier to be overffited. On the other hand, if C is too small, the model will be underfitted.