Boston house price prediction

刘天润 SC24219058

Table of contents

Import libraries

In [1]:
```
# Import libraries for data manipulation
import pandas as pd
import numpy as np

# Import libraries for data visualization
import matplotlib.pyplot as plt
import seaborn as sns

# Import libraries for building linear regression model
from statsmodels.formula.api import ols
import statsmodels.api as sm

# Import library for preparing data
from sklearn.model_selection import train_test_split

import warnings
warnings.filterwarnings("ignore")
```

Data analysis

load data

In [2]:
```
df = pd.read_excel("BostonHousingData.xlsx")
df.info()
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 506 entries, 0 to 505
Data columns (total 14 columns):
 #   Column   Non-Null Count  Dtype
---  ------   --------------  -----
 0   CRIM     506 non-null    float64
 1   ZN       506 non-null    float64
 2   INDUS    506 non-null    float64
 3   CHAS     506 non-null    int64
 4   NOX      506 non-null    float64
 5   RM       506 non-null    float64
 6   AGE      506 non-null    float64
 7   DIS      506 non-null    float64
 8   RAD      506 non-null    int64
 9   TAX      506 non-null    int64
 10  PTRATIO  506 non-null    float64
 11  B        506 non-null    float64
 12  LSTAT    506 non-null    float64
 13  MEDV     506 non-null    float64
dtypes: float64(11), int64(3)
memory usage: 55.5 KB
```

There are a total of 506 non-null observations in each of the columns. This indicates that there are no missing values in the data. There are 13 columns in the dataset and every column is of numeric data type.

Exploratory Data Analysis

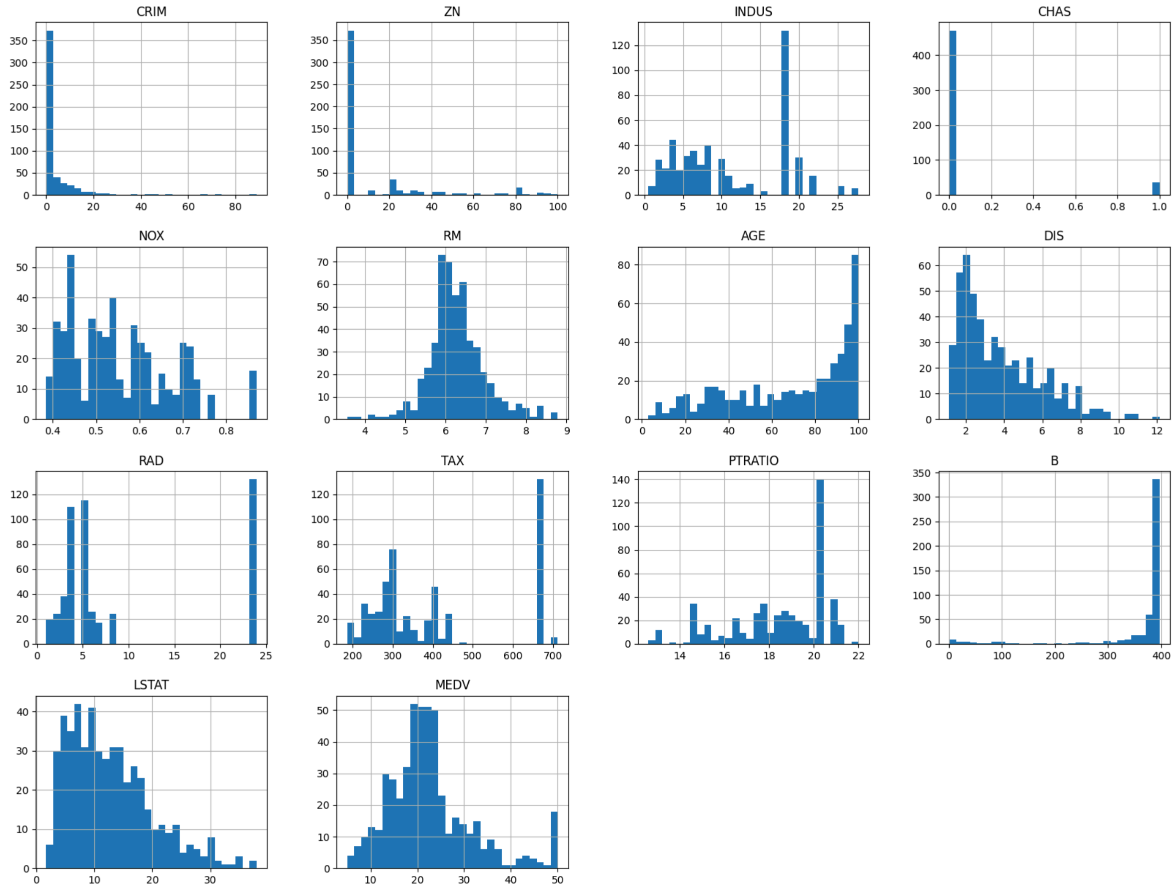
In [3]:
```python
df.describe()
```

Out[3]:

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

Univariate analysis

In [4]:
```python
df.hist(bins=30, figsize=(20,15))
plt.show()
```



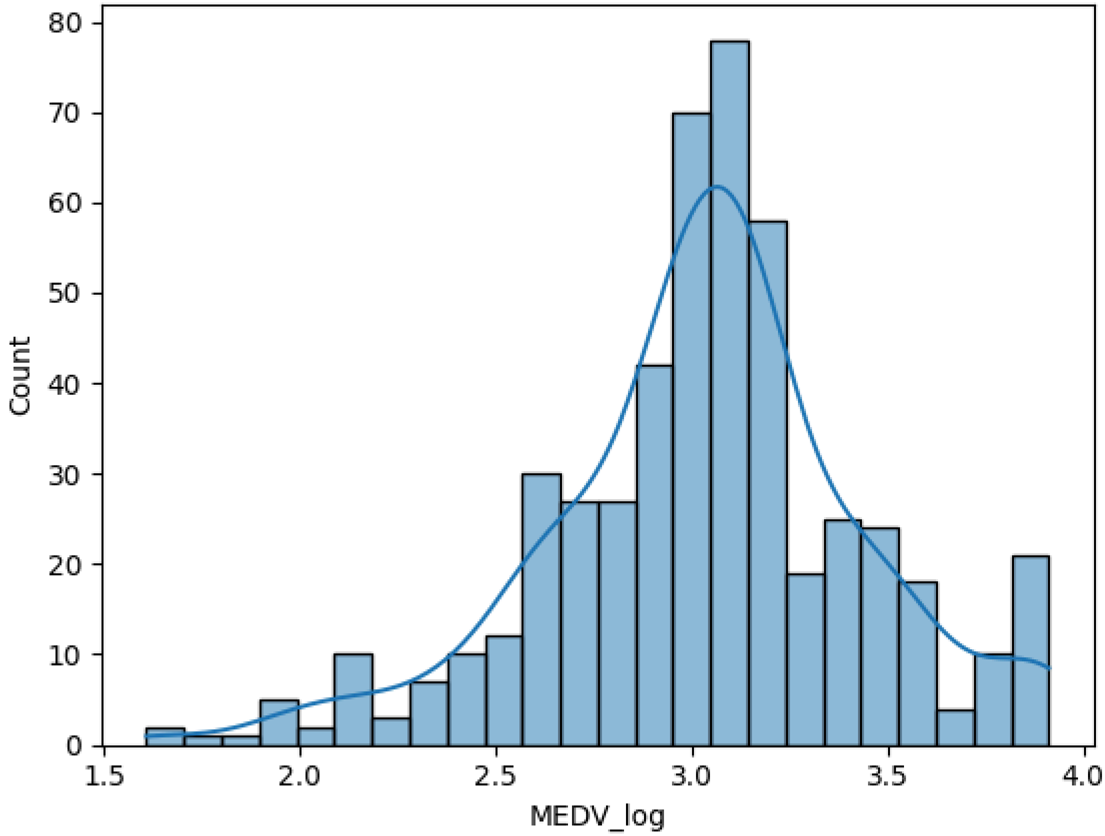
MEDV: Median value of owner-occupied homes in 1000 dollars Slightly skewed. As this is our dependent variable will need to take action to **normalize** it.

Least squares regression models assume the residuals are normal, and a non-normal dependent variable will produce non-normal residual errors. Therefore, as the dependent variable is sightly skewed, we need to apply a log transformation on the 'MEDV' column and check the distribution of the transformed column.

Note: Using methods like quantile regression and robust regression can use non-normal dependent variables.

In [5]:
```
df['MEDV_log'] = np.log(df['MEDV'])
sns.histplot(data = df, x = 'MEDV_log', kde = True)
```

Out[5]: <Axes: xlabel='MEDV_log', ylabel='Count'>

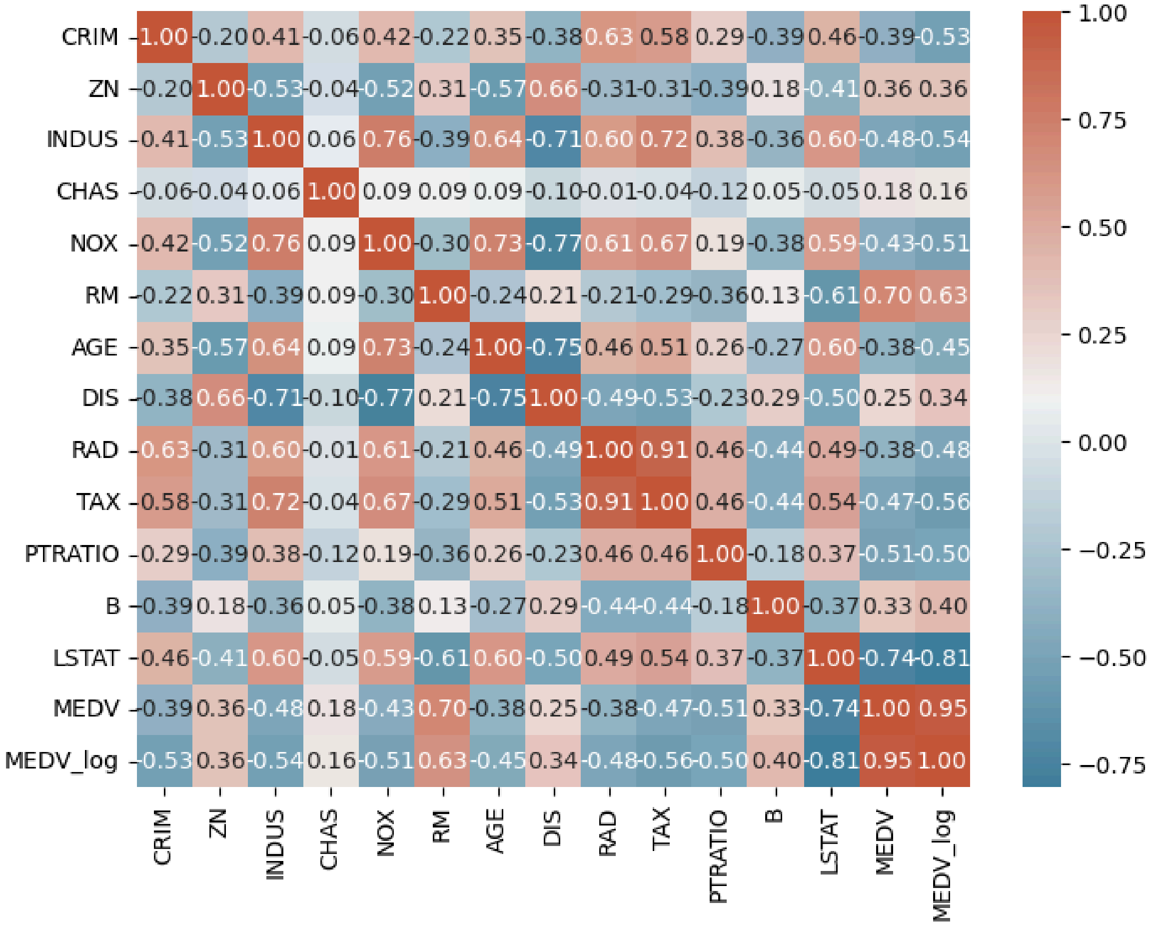


The log-transformation (MEDV_log) appears to have a nearly normal distribution without skew, therefore we can proceed.

Bivariate Analysis

Check the correlation using heatmap

In [6]:
```
plt.figure(figsize = (8, 6))

cmap = sns.diverging_palette(230, 20, as_cmap = True)

sns.heatmap(df.corr(), annot = True, fmt = '.2f', cmap = cmap)

plt.show()
```



Strong correlations (>= 0.7 or <= -0.7) between MEDV_log and:

LSTAT: Negative Correlation

Strong correlations (>= 0.7 or <= -0.7) not involving our dependent variable:

Positive Correlation between NOX and INDUS. Positive Correlation between NOX and AGE.

Negative Correlation between DIS and INDUS, DIS and NOX, DIS and AGE.

Positive Correlation between TAX and INDUS. Very high Positive Correlation between TAX and RAD.

We can reduce the complexity of the model and improve its performance by dropping some highly related features.

Split dataset

Split the data into the dependent and independent variables and further split it into train and test set in a ratio of 9:1 for train and test sets.

In [7]:
```
Y = df['MEDV_log']

X = df.drop(columns = {'MEDV', 'MEDV_log'})
```

```
# Add the intercept term
X = sm.add_constant(X)
```

Intercept Term: allows the regression line to be shifted up or down on the y-axis to better fit the data. The value of the intercept term can be interpreted as the expected value of the dependent variable when all independent variables are set to zero.

In [8]:
```
X_train, X_test, y_train, y_test = train_test_split(X, Y, test_size = 0.10, rand

# drop highly related features
X_train.drop(columns={'TAX','NOX','DIS'})
```

Out[8]:

	const	CRIM	ZN	INDUS	CHAS	RM	AGE	RAD	PTRATIO	B	LST
242	1.0	0.10290	30.0	4.93	0	6.358	52.9	6	16.6	372.75	11.
5	1.0	0.02985	0.0	2.18	0	6.430	58.7	3	18.7	394.12	5.
168	1.0	2.30040	0.0	19.58	0	6.319	96.1	5	14.7	297.09	11.
490	1.0	0.20746	0.0	27.74	0	5.093	98.0	4	20.1	318.43	29.
62	1.0	0.11027	25.0	5.13	0	6.456	67.8	8	19.7	396.90	6.
...	...	...	...	...	...	...	...	...	...	...	
255	1.0	0.03548	80.0	3.64	0	5.876	19.1	1	16.4	395.18	9.
72	1.0	0.09164	0.0	10.81	0	6.065	7.8	4	19.2	390.91	5.
396	1.0	5.87205	0.0	18.10	0	6.405	96.0	24	20.2	396.90	19.
235	1.0	0.33045	0.0	6.20	0	6.086	61.5	8	17.4	376.75	10.
37	1.0	0.08014	0.0	5.96	0	5.850	41.5	5	19.2	396.90	8.

455 rows × 11 columns

Model Building

Linear Regression Model

In [9]:
```
# Create the model using ordinary least squared
model1 = sm.OLS(y_train,X_train).fit()

# Get the model summary
model1.summary()
```

Out[9]:

OLS Regression Results

Dep. Variable:	MEDV_log	**R-squared:**	0.787
Model:	OLS	**Adj. R-squared:**	0.781
Method:	Least Squares	**F-statistic:**	125.4
Date:	Sun, 06 Apr 2025	**Prob (F-statistic):**	6.20e-139
Time:	15:28:27	**Log-Likelihood:**	112.88
No. Observations:	455	**AIC:**	-197.8
Df Residuals:	441	**BIC:**	-140.1
Df Model:	13		
Covariance Type:	nonrobust		

	coef	std err	t	P>|t|	[0.025	0.975]
const	4.1763	0.216	19.344	0.000	3.752	4.601
CRIM	-0.0105	0.001	-7.819	0.000	-0.013	-0.008
ZN	0.0015	0.001	2.409	0.016	0.000	0.003
INDUS	0.0018	0.003	0.697	0.486	-0.003	0.007
CHAS	0.1008	0.036	2.811	0.005	0.030	0.171
NOX	-0.8260	0.163	-5.068	0.000	-1.146	-0.506
RM	0.0849	0.018	4.718	0.000	0.050	0.120
AGE	0.0002	0.001	0.378	0.706	-0.001	0.001
DIS	-0.0525	0.009	-6.084	0.000	-0.070	-0.036
RAD	0.0147	0.003	5.189	0.000	0.009	0.020
TAX	-0.0006	0.000	-3.948	0.000	-0.001	-0.000
PTRATIO	-0.0371	0.006	-6.723	0.000	-0.048	-0.026
B	0.0004	0.000	3.125	0.002	0.000	0.001
LSTAT	-0.0286	0.002	-13.352	0.000	-0.033	-0.024

Omnibus:	52.314	**Durbin-Watson:**	1.998
Prob(Omnibus):	0.000	**Jarque-Bera (JB):**	205.743
Skew:	0.423	**Prob(JB):**	2.11e-45
Kurtosis:	6.184	**Cond. No.**	1.50e+04

Notes:
[1] Standard Errors assume that the covariance matrix of the errors is correctly specified.

[2] The condition number is large, 1.5e+04. This might indicate that there are strong multicollinearity or other numerical problems.

R-squared is at 78.7%, not bad but can be improved.

From the above it may be noted that the regression coefficients corresponding to ZN, AGE, and INDUS are not statistically significant at level α = 0.05. In other words, the regression coefficients corresponding to these three are not significantly different from 0 in the population.

Check performance

1. Check for mean of residuals

In [10]:
```
residuals = model1.resid

np.mean(residuals)
```

Out[10]:
```
2.4063779052424823e-15
```

The mean of residuals approach 0 hence satisfied the assumption

2. Check for homoscedasticity

In [11]:
```
from statsmodels.stats.diagnostic import het_white

from statsmodels.compat import lzip

import statsmodels.stats.api as sms
name = ["F statistic", "p-value"]

test = sms.het_goldfeldquandt(y_train, X_train)

lzip(name, test)
```

Out[11]:
```
[('F statistic', 1.1609356413798142), ('p-value', 0.13816348189672661)]
```

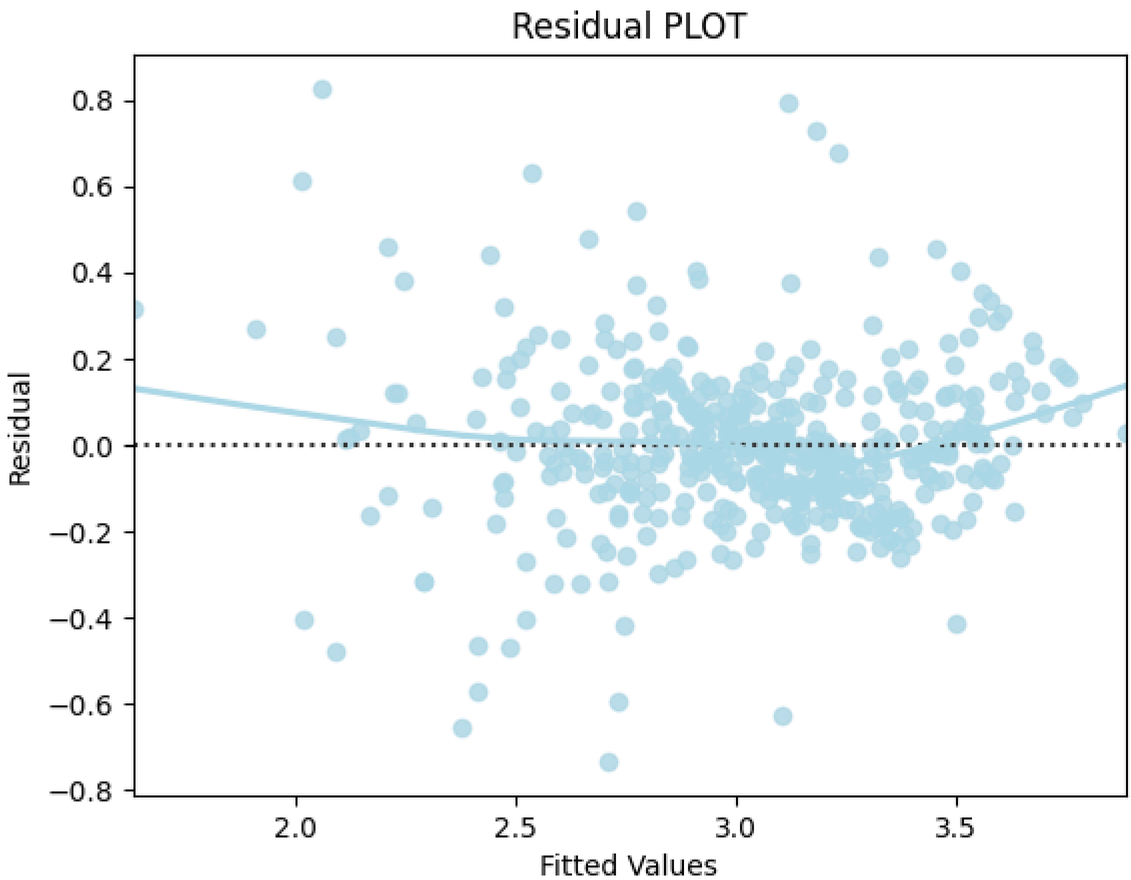
Since the p-value > 0.05, we cant reject the Null-Hypothesis.

3. Linearity of variables

Goldfeld-Quandt Test: This test involves dividing the data into two subgroups based on a selected variable and then comparing the variances of the residuals in these subgroups. In Python, you can use het_goldfeldquandt from the statsmodels.stats.diagnostic module. If the variances in the two subgroups are significantly different, it suggests heteroscedasticity.

In [12]:
```
# Predicted values
fitted = model1.fittedvalues

# sns.set_style("whitegrid")
```

```
sns.residplot(x = fitted, y = residuals, color = "lightblue", lowess = True)

plt.xlabel("Fitted Values")

plt.ylabel("Residual")

plt.title("Residual PLOT")

plt.show()
```

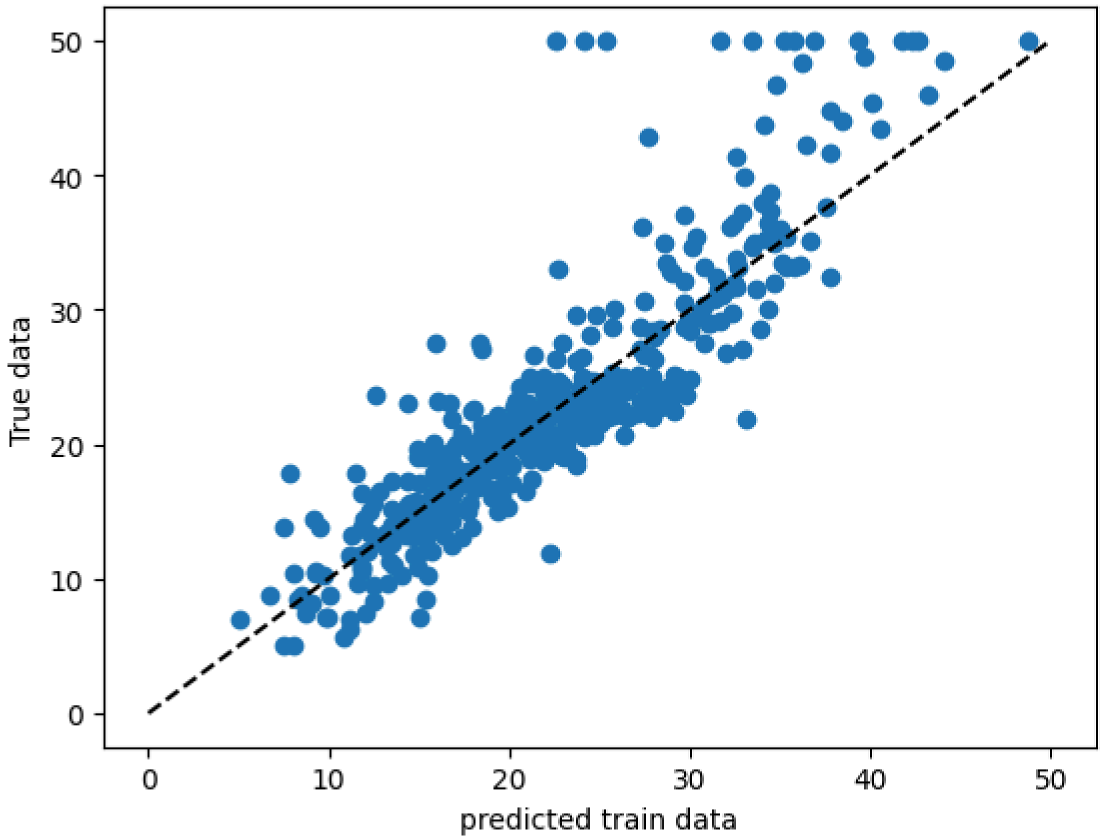


There is no pattern in the residual vs fitted values, therefore the assumption is satesfied.

In [13]:
```
# RMSE
def mse(predictions, targets):
    return ((targets - predictions) ** 2).mean()

# Model Performance on test and train data
def model_perf(olsmodel, x_train, x_test):

    # In-sample Prediction
    y_pred_train = olsmodel.predict(x_train)
    y_true_train = y_train

    # Prediction on test data
    y_pred_test = olsmodel.predict(x_test)
    y_true_test = y_test

    plt.scatter(np.e**y_pred_train, np.e**y_true_train)
    plt.plot([0, 50], [0, 50], '--k')
    plt.xlabel('predicted train data')
    plt.ylabel('True data')
```

```
    # plt.scatter(y_pred_train,y_true_train)

    print(
        pd.DataFrame(
            {
                "Data": ["Train", "Test"],
                "MSE": [
                    mse(np.exp(y_pred_train), np.exp(y_true_train)),
                    mse(np.exp(y_pred_test), np.exp(y_true_test)),
                    # mse(y_pred_train,y_true_train)
                ],

            }
        )
    )


# Checking model performance

model_perf(model1, X_train, X_test)
```

```
    Data        MSE
0  Train  19.084023
1   Test  16.959608
```



Conclusion:

The train and test RMSE are very close, therefore our model is **not overfitted and generalizes well**.

Neural Network

import libraries

In [14]:
```
import torch
from torch import nn
from torch.nn import functional as F
import torch.utils.data as data
from torch import optim
from sklearn import datasets
from sklearn.preprocessing import StandardScaler
from sklearn.preprocessing import MinMaxScaler
from sklearn.metrics import mean_absolute_error
from sklearn.model_selection import train_test_split
```

Make train and test dataset

In [15]:
```
y = df['MEDV']
X = df.drop(columns = {'MEDV', 'MEDV_log'})

X_train, X_test, y_train, y_test = train_test_split(X, y, random_state=1)
n_samples, n_features = X.shape
```

Feature Scaling

In [16]:
```
sc = StandardScaler()
X_train = sc.fit_transform(X_train)
X_test = sc.transform(X_test)
```

In [17]:
```
# dataset = pd.read_excel("BostonHousingData")
device = torch.device("cuda:0" if torch.cuda.is_available() else "cpu")
np.random.seed(21)
torch.manual_seed(21)
```

Out[17]:
```
<torch._C.Generator at 0x2e3ff997b50>
```

Load data in batch size

In [18]:
```
def load_array(data_arrays, batch_size, is_train=True):
    # create PyTorch data iterator
    dataset = data.TensorDataset(*data_arrays)
    return data.DataLoader(dataset, batch_size, shuffle=is_train)

batch_size = 25
# transform data type
X_train_tensor = torch.tensor(X_train, dtype=torch.float32)
y_train_tensor = torch.tensor(y_train, dtype=torch.float32).unsqueeze(1)
X_test_tensor = torch.tensor(X_test, dtype=torch.float32)
y_test_tensor = torch.tensor(y_test.values, dtype=torch.float32)

print(X_train_tensor.size())
data_iter = load_array((X_train_tensor, y_train_tensor), batch_size)
test_loader=load_array((X_test_tensor,y_test_tensor),batch_size)
```

```
torch.Size([379, 13])
```

Define the NN model

In [19]:
```
# network model
class Model(nn.Module):
    def __init__(self, n_features, hiddenA, hiddenB):
        super(Model, self).__init__()
        self.linearA = nn.Linear(n_features, hiddenA)
        self.linearB = nn.Linear(hiddenA, hiddenB)
        self.linearC = nn.Linear(hiddenB, 1)

    def forward(self, x):
        yA = F.relu(self.linearA(x))
        yB = F.relu(self.linearB(yA))
        return self.linearC(yB)
```

In [20]:
```
#define NN input num、hidden layer A、B number
net = Model(n_features, 50,20)

# loss function
loss = nn.MSELoss()

#define optimization
trainer = torch.optim.Adam(net.parameters(), lr=0.01)
```

Training

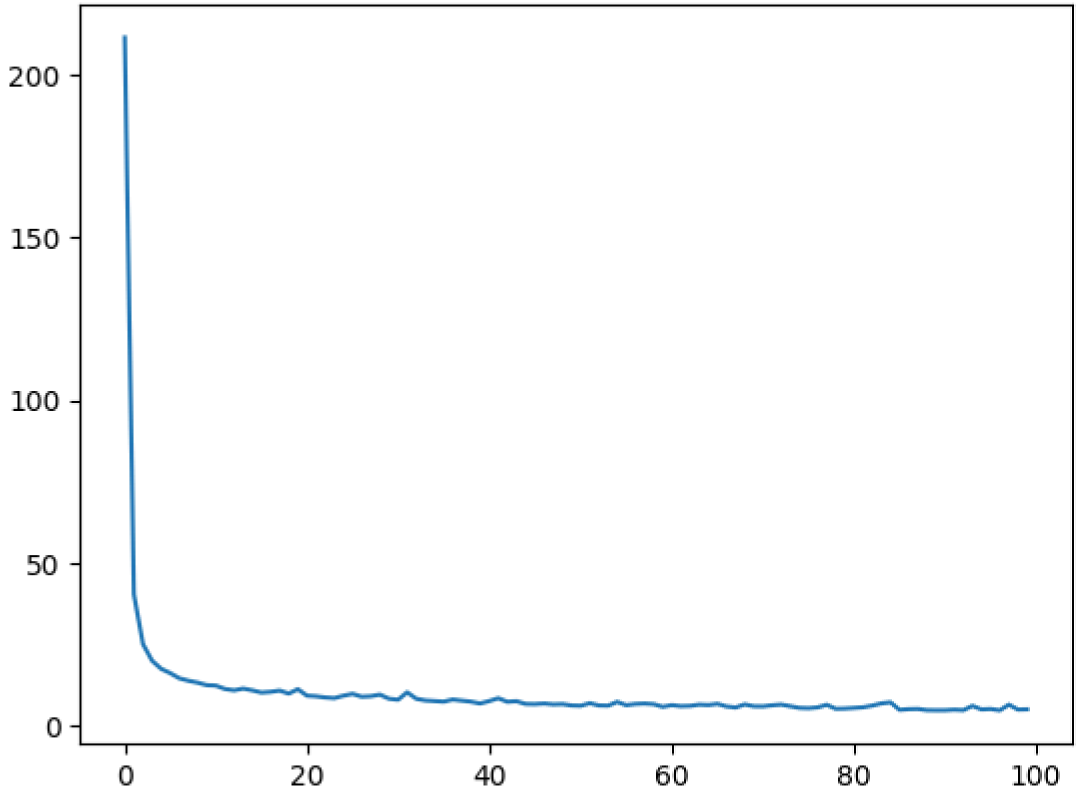
In [21]:
```
num_epochs = 100
all_losses=[]
for epoch in range(num_epochs):
    for X, y in data_iter:
        l = loss(net(X) ,y)
        trainer.zero_grad()
        l.backward()
        trainer.step()
    l = loss(net(X_train_tensor), y_train_tensor)
    print(f'epoch {epoch + 1}, loss {l:f}')
    all_losses.append(l)

all_losses_np = torch.tensor(all_losses).detach().numpy()
plt.plot(all_losses_np)
```

```
epoch 1, loss 211.178375
epoch 2, loss 40.431667
epoch 3, loss 25.148335
epoch 4, loss 20.025293
epoch 5, loss 17.553223
epoch 6, loss 16.230713
epoch 7, loss 14.640662
epoch 8, loss 13.898292
epoch 9, loss 13.323333
epoch 10, loss 12.551755
epoch 11, loss 12.411380
epoch 12, loss 11.375528
epoch 13, loss 10.966515
epoch 14, loss 11.510386
epoch 15, loss 10.964094
epoch 16, loss 10.283645
epoch 17, loss 10.493580
epoch 18, loss 10.890614
epoch 19, loss 9.970551
epoch 20, loss 11.315365
epoch 21, loss 9.327151
epoch 22, loss 9.141327
epoch 23, loss 8.770189
epoch 24, loss 8.569515
epoch 25, loss 9.307537
epoch 26, loss 9.893481
epoch 27, loss 9.018070
epoch 28, loss 9.157809
epoch 29, loss 9.658569
epoch 30, loss 8.404288
epoch 31, loss 8.071660
epoch 32, loss 10.364773
epoch 33, loss 8.381832
epoch 34, loss 7.867637
epoch 35, loss 7.682922
epoch 36, loss 7.487062
epoch 37, loss 8.150660
epoch 38, loss 7.814933
epoch 39, loss 7.494398
epoch 40, loss 6.925875
epoch 41, loss 7.667480
epoch 42, loss 8.558464
epoch 43, loss 7.447286
epoch 44, loss 7.632472
epoch 45, loss 6.789209
epoch 46, loss 6.733048
epoch 47, loss 6.920307
epoch 48, loss 6.648250
epoch 49, loss 6.753411
epoch 50, loss 6.315160
epoch 51, loss 6.235201
epoch 52, loss 7.023687
epoch 53, loss 6.325701
epoch 54, loss 6.265414
epoch 55, loss 7.409880
epoch 56, loss 6.408016
epoch 57, loss 6.782548
epoch 58, loss 6.916611
epoch 59, loss 6.726186
epoch 60, loss 5.926729
```

```
epoch 61, loss 6.422053
epoch 62, loss 6.108149
epoch 63, loss 6.192948
epoch 64, loss 6.584384
epoch 65, loss 6.455864
epoch 66, loss 6.816649
epoch 67, loss 6.032435
epoch 68, loss 5.704159
epoch 69, loss 6.644650
epoch 70, loss 6.079269
epoch 71, loss 6.042371
epoch 72, loss 6.374020
epoch 73, loss 6.603501
epoch 74, loss 6.093718
epoch 75, loss 5.575790
epoch 76, loss 5.483064
epoch 77, loss 5.735765
epoch 78, loss 6.526803
epoch 79, loss 5.311162
epoch 80, loss 5.378872
epoch 81, loss 5.564423
epoch 82, loss 5.766329
epoch 83, loss 6.280193
epoch 84, loss 6.955698
epoch 85, loss 7.293020
epoch 86, loss 4.991150
epoch 87, loss 5.191741
epoch 88, loss 5.260571
epoch 89, loss 4.891850
epoch 90, loss 4.853191
epoch 91, loss 4.860579
epoch 92, loss 5.032619
epoch 93, loss 4.890618
epoch 94, loss 6.195663
epoch 95, loss 5.114611
epoch 96, loss 5.245399
epoch 97, loss 4.839276
epoch 98, loss 6.559350
epoch 99, loss 5.067227
epoch 100, loss 5.138278
```

Out[21]:

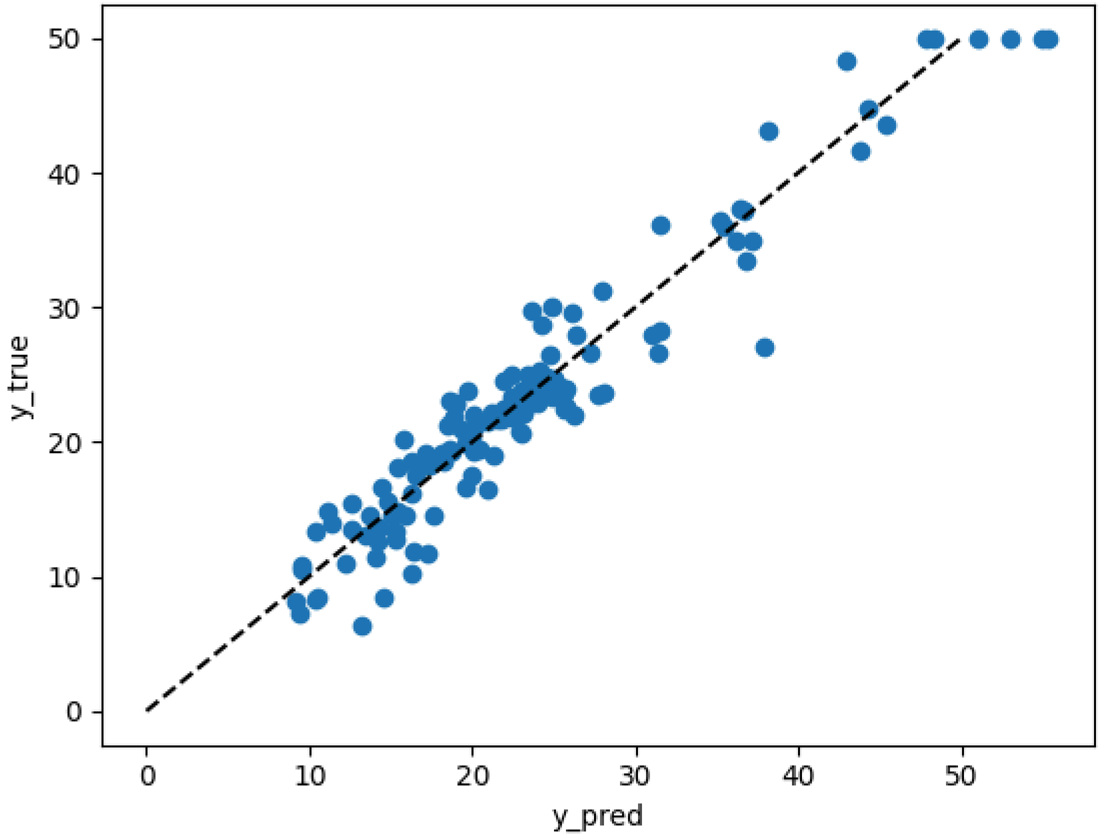
```
[<matplotlib.lines.Line2D at 0x2e396013750>]
```



Model evaluation

In [22]:
```
y_pred = []
y_true = []
net.train(False)
for inputs, targets in test_loader:
    y_pred.extend(net(inputs).data.numpy())
    y_true.extend(targets.numpy())
plt.scatter(y_pred, y_true)
plt.xlabel('y_pred')
plt.ylabel('y_true')
plt.plot([0, 50], [0, 50], '--k')
print("MAE:", mean_absolute_error(y_true, y_pred))
```

```
MAE: 2.2000093
```



Neural network with 2 hidden layers's MAE on testing data is only 2.2, which is much beter than 17 that linear regression model's result. So **NN has the better performance** than simple linear regression.