HW1_submission

January 27, 2025

Cleaned Automobile Dataset Documentation Dataset Overview: The cleaned automobile dataset is derived from the UCI Machine Learning Repository's "Automobile Data Set." This dataset contains various continuous features related to automobile specifications and a target variable (price) representing the price of the car.

File Details: * File Name: cleaned_automobile_data.pt * File Format: PyTorch .pt file (serialized dictionary) * Size: Varies based on the number of valid observations. * Contents of the File: * The .pt file contains the following keys:

- features: PyTorch tensor of shape (n_samples, 13)
- Description: Continuous numerical features related to automobile characteristics.
- target: PyTorch tensor of shape (n_samples, 1)
- Description: Price of the automobile (target variable).
- feature_names: List of strings (length 13)
- Description: Names of the feature columns:['wheel-base', 'length', 'width', 'height', 'curb-weight', 'engine-size', 'bore', 'stroke', 'compression-ratio', 'horsepower', 'peak-rpm', 'city-mpg', 'highway-mpg']
- target_name: String
- Description: Name of the target variable (price).

Data Cleaning Details: * Missing Values Handling: Rows with missing values for any of the 13 continuous features or price were removed. * Continuous Features: Only continuous features relevant to car specifications are retained. * Target Variable: The price column was cleaned and reshaped for use in regression tasks. * Dataset Shape: After cleaning, the dataset contains approximately 195 samples

[2]:
```
import torch

# === Step 2: Loading the Dataset === #
def load_dataset(file_path):
    """
    Loads the saved PyTorch .pt dataset from the specified file path.

    This function reads a dataset saved in PyTorch format and extracts its␣
↪features, target values,
    and relevant metadata. It prints the shapes of the feature matrix and␣
↪target vector,
    as well as the feature and target names.
```

```
    Parameters:
    -----------
    file_path : str
        The file path to the saved .pt dataset.

    Returns:
    --------
    tuple
        A tuple containing:
        - features (torch.Tensor): A tensor containing the feature values.
        - target (torch.Tensor): A tensor containing the target values.
        - feature_names (list of str): A list of names corresponding to the␣
↪feature columns.
        - target_name (str): The name of the target variable.

    Example:
    --------
    >>> features, target, feature_names, target_name =␣
↪load_dataset("cleaned_automobile_train_dataset.pt")
    Loaded dataset successfully!
    Features shape: torch.Size([100, 13]), Target shape: torch.Size([100])
    Feature Names: ['wheel-base', 'length', 'width', ...], Target Name: 'price'
    """
    dataset = torch.load(file_path)
    print("Loaded dataset successfully!")
    print(f"Features shape: {dataset['features'].shape}, Target shape:␣
↪{dataset['target'].shape}")
    print(f"Feature Names: {dataset['feature_names']}, Target Name:␣
↪{dataset['target_name']}")
    return dataset["features"], dataset["target"], dataset["feature_names"],␣
↪dataset["target_name"]

if __name__ == "__main__":
    #Load the dataset and display information
    print('loading train_dataset from cleaned_automobile_train_dataset.pt')
    X_train, Y_train, feature_names, target_name =␣
↪load_dataset("cleaned_automobile_train_dataset.pt")
    print('number of training samples={}'.format(X_train.shape[0]))
    print('dimension of features={}'.format(X_train.shape[1]))
    print('X_train shape={}'.format(list(X_train.shape)))
    print('Y_train shape={}'.format(list(Y_train.shape)))

    print('loading test_dataset from cleaned_automobile_test_dataset.pt')
    X_test, Y_test, feature_names, target_name =␣
↪load_dataset("cleaned_automobile_test_dataset.pt")
    print('number of test samples={}'.format(X_test.shape[0]))
```

```
    print('dimension of features={}'.format(X_test.shape[1]))
    print('X_test shape={}'.format(list(X_test.shape)))
    print('Y_test shape={}'.format(list(Y_test.shape)))
```

```
loading train_dataset from cleaned_automobile_train_dataset.pt
Loaded dataset successfully!
Features shape: torch.Size([156, 13]), Target shape: torch.Size([156, 1])
Feature Names: ['wheel-base', 'length', 'width', 'height', 'curb-weight',
'engine-size', 'bore', 'stroke', 'compression-ratio', 'horsepower', 'peak-rpm',
'city-mpg', 'highway-mpg'], Target Name: price
number of training samples=156
dimension of features=13
X_train shape=[156, 13]
Y_train shape=[156, 1]
loading test_dataset from cleaned_automobile_test_dataset.pt
Loaded dataset successfully!
Features shape: torch.Size([39, 13]), Target shape: torch.Size([39, 1])
Feature Names: ['wheel-base', 'length', 'width', 'height', 'curb-weight',
'engine-size', 'bore', 'stroke', 'compression-ratio', 'horsepower', 'peak-rpm',
'city-mpg', 'highway-mpg'], Target Name: price
number of test samples=39
dimension of features=13
X_test shape=[39, 13]
Y_test shape=[39, 1]
```

```
/tmp/ipykernel_4630/1056467191.py:33: FutureWarning: You are using `torch.load`
with `weights_only=False` (the current default value), which uses the default
pickle module implicitly. It is possible to construct malicious pickle data
which will execute arbitrary code during unpickling (See
https://github.com/pytorch/pytorch/blob/main/SECURITY.md#untrusted-models for
more details). In a future release, the default value for `weights_only` will be
flipped to `True`. This limits the functions that could be executed during
unpickling. Arbitrary objects will no longer be allowed to be loaded via this
mode unless they are explicitly allowlisted by the user via
`torch.serialization.add_safe_globals`. We recommend you start setting
`weights_only=True` for any use case where you don't have full control of the
loaded file. Please open an issue on GitHub for any issues related to this
experimental feature.
  dataset = torch.load(file_path)
```

[3]:
```
X_test, Y_test, feature_names, target_name =␣
 ↪load_dataset("cleaned_automobile_test_dataset.pt")
```

```
Loaded dataset successfully!
Features shape: torch.Size([39, 13]), Target shape: torch.Size([39, 1])
Feature Names: ['wheel-base', 'length', 'width', 'height', 'curb-weight',
'engine-size', 'bore', 'stroke', 'compression-ratio', 'horsepower', 'peak-rpm',
'city-mpg', 'highway-mpg'], Target Name: price
```

```
/tmp/ipykernel_4630/1056467191.py:33: FutureWarning: You are using `torch.load`
with `weights_only=False` (the current default value), which uses the default
pickle module implicitly. It is possible to construct malicious pickle data
which will execute arbitrary code during unpickling (See
https://github.com/pytorch/pytorch/blob/main/SECURITY.md#untrusted-models for
more details). In a future release, the default value for `weights_only` will be
flipped to `True`. This limits the functions that could be executed during
unpickling. Arbitrary objects will no longer be allowed to be loaded via this
mode unless they are explicitly allowlisted by the user via
`torch.serialization.add_safe_globals`. We recommend you start setting
`weights_only=True` for any use case where you don't have full control of the
loaded file. Please open an issue on GitHub for any issues related to this
experimental feature.
  dataset = torch.load(file_path)
```

[4]:
```
X_test.shape, Y_test.shape
```

[4]:
```
(torch.Size([39, 13]), torch.Size([39, 1]))
```

0.1 1.a)

[5]:
```
X_train, Y_train, feature_names, target_name =␣
↪load_dataset("cleaned_automobile_train_dataset.pt")
```

```
Loaded dataset successfully!
Features shape: torch.Size([156, 13]), Target shape: torch.Size([156, 1])
Feature Names: ['wheel-base', 'length', 'width', 'height', 'curb-weight',
'engine-size', 'bore', 'stroke', 'compression-ratio', 'horsepower', 'peak-rpm',
'city-mpg', 'highway-mpg'], Target Name: price
```

```
/tmp/ipykernel_4630/1056467191.py:33: FutureWarning: You are using `torch.load`
with `weights_only=False` (the current default value), which uses the default
pickle module implicitly. It is possible to construct malicious pickle data
which will execute arbitrary code during unpickling (See
https://github.com/pytorch/pytorch/blob/main/SECURITY.md#untrusted-models for
more details). In a future release, the default value for `weights_only` will be
flipped to `True`. This limits the functions that could be executed during
unpickling. Arbitrary objects will no longer be allowed to be loaded via this
mode unless they are explicitly allowlisted by the user via
`torch.serialization.add_safe_globals`. We recommend you start setting
`weights_only=True` for any use case where you don't have full control of the
loaded file. Please open an issue on GitHub for any issues related to this
experimental feature.
  dataset = torch.load(file_path)
```

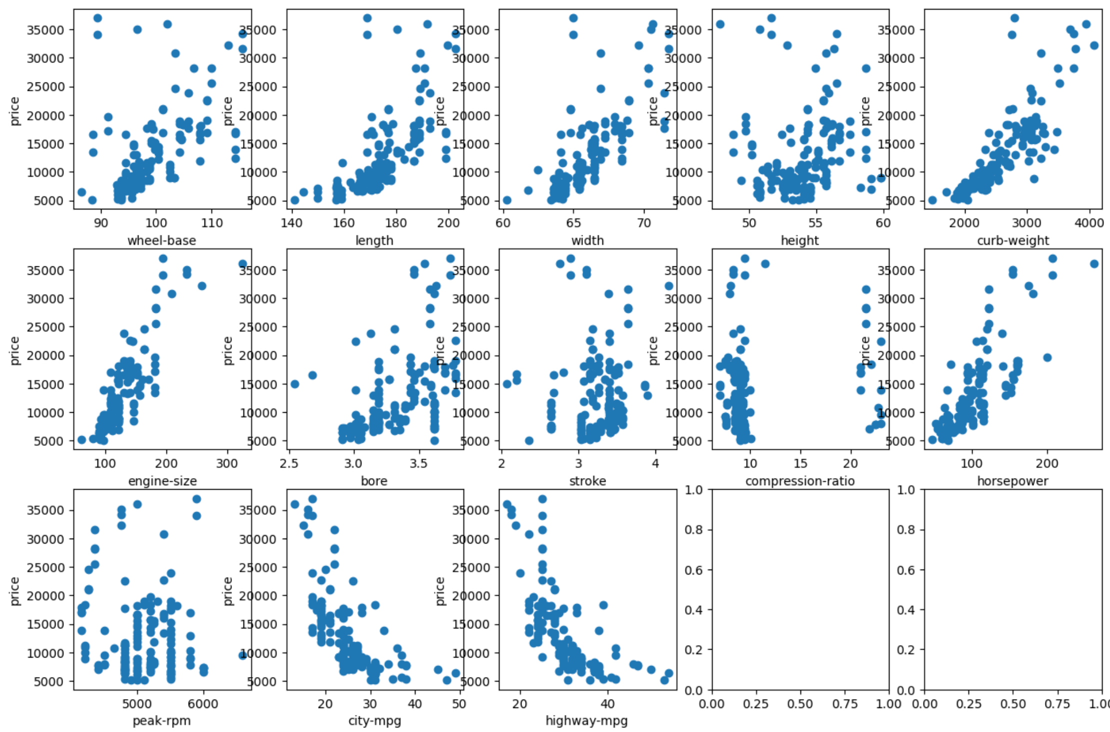
[6]:
```
X_train.shape, Y_train.shape, feature_names, target_name
```

[6]:
```
(torch.Size([156, 13]),
 torch.Size([156, 1]),
 ['wheel-base',
  'length',
  'width',
  'height',
  'curb-weight',
  'engine-size',
  'bore',
  'stroke',
  'compression-ratio',
  'horsepower',
  'peak-rpm',
  'city-mpg',
  'highway-mpg'],
 'price')
```

0.2 1.b)

[7]:
```python
from matplotlib import pyplot as plt

# 13 subplots
# make it 3 * 5
fig, ax = plt.subplots(3, 5, figsize=(15, 10))
for i in range(13):
    ax[i//5, i%5].scatter(X_train[:, i], Y_train)
    ax[i//5, i%5].set_xlabel(feature_names[i])
    ax[i//5, i%5].set_ylabel(target_name)
plt.show()
```



0.3 1.c)

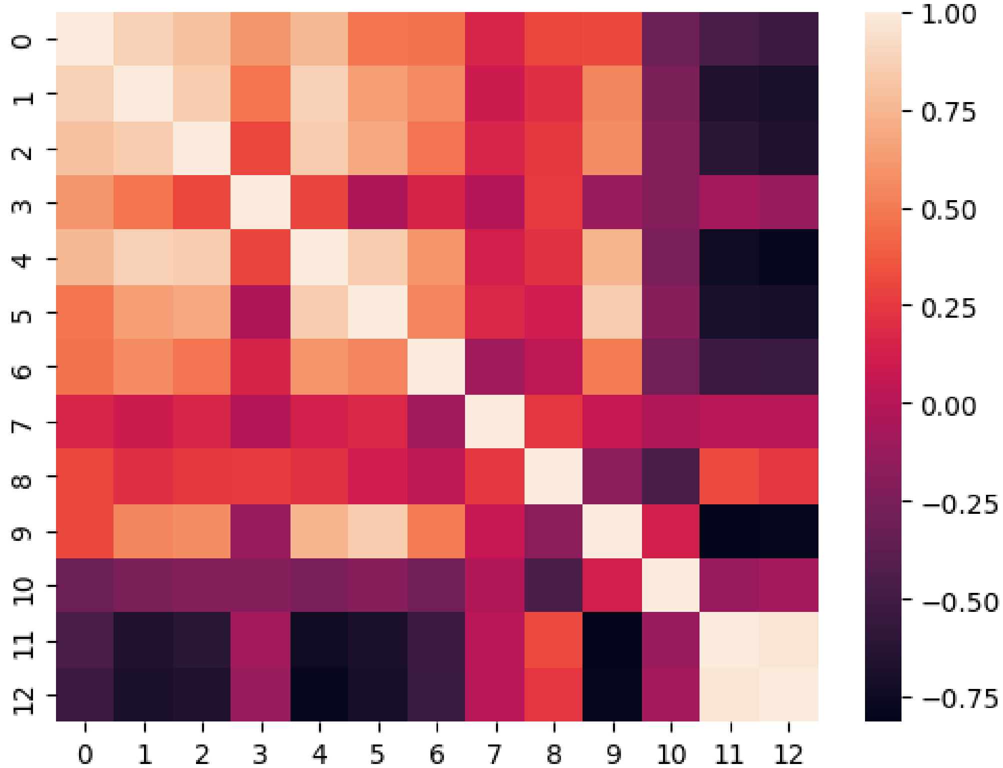
In general, some features show stronger positive correlation with the target, those includes `wheel-base`, `width`, etc, and some features show negative correlation with the target, including `city-mpg`, `highway-mpg`. This makes sense because we expect a higher price from a bigger car and a car with lower mileage.

0.4 1.d)

[8]:
```
# pair-wise correlation

import numpy as np
import seaborn as sns
corr_mat = X_train.T.corrcoef()
sns.heatmap(corr_mat)
```

[8]: <Axes: >

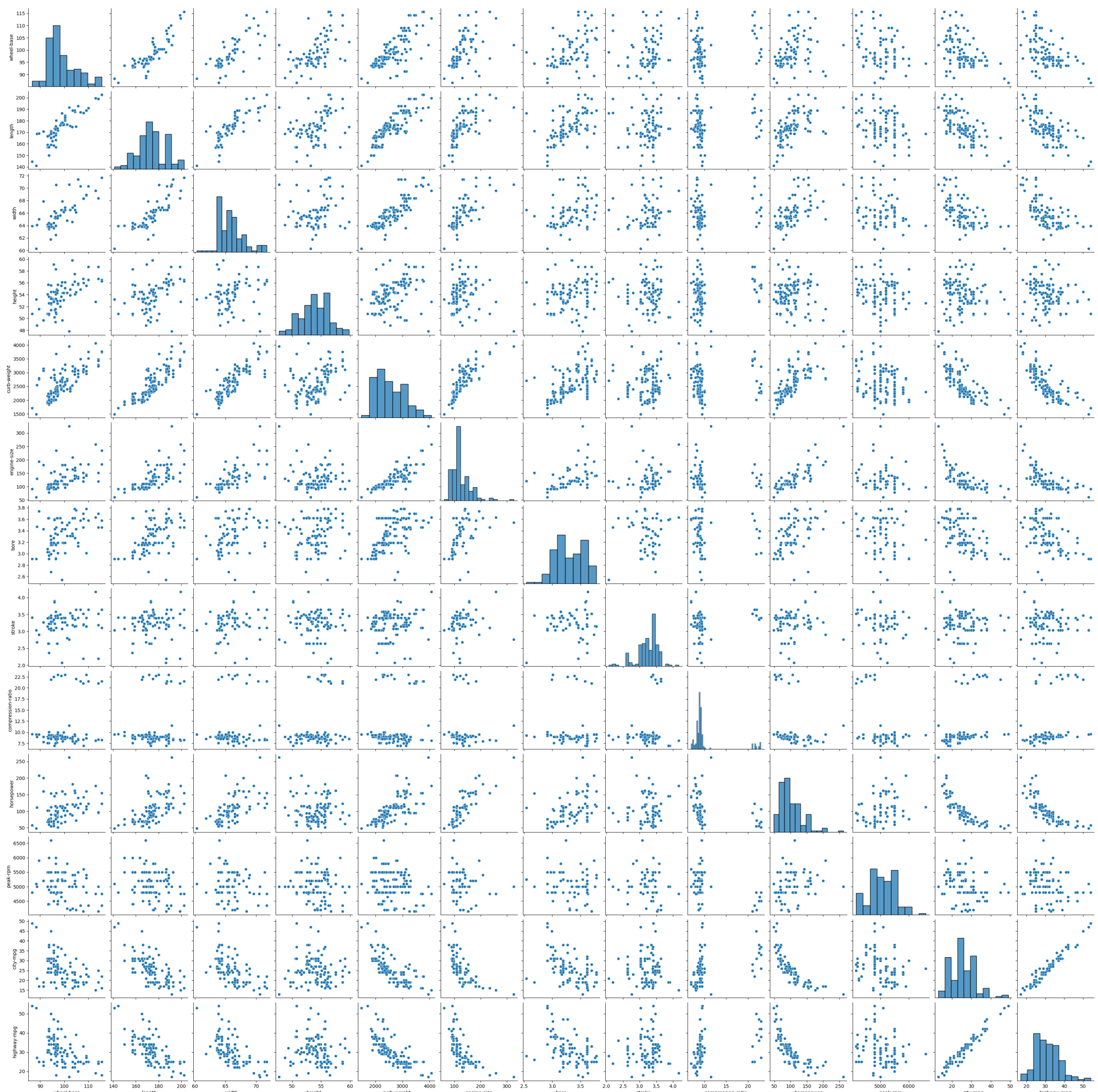


[9]:
```
# pair-wise scattle plot

import pandas as pd

sns.pairplot(pd.DataFrame(X_train, columns=feature_names))
```

[9]:
```
<seaborn.axisgrid.PairGrid at 0x7f211f502660>
```



0.5 1.e)

[10]:
```
feature_names
```

[10]:
```
['wheel-base',
 'length',
 'width',
 'height',
 'curb-weight',
 'engine-size',
 'bore',
 'stroke',
```

```
'compression-ratio',
'horsepower',
'peak-rpm',
'city-mpg',
'highway-mpg']
```

From both plots above, we can see that first three feature, and the last two features, have really strong, positive correlations. This makes sense because one should expect `city-mpg` to be correlated with `highway-mpg` and the `length`, `width`, `height` to follow some ratio.

0.6 1.f)

[ ]:
```

```

0.7 2.a)

[58]:
```
def transform(X):
    X_transformed = torch.ones((X.shape[0], X.shape[1] + 1))

    n = X.shape[1]

    # add x, 1/x , log x, x^2, x^0.5 x^3 features
    X_transformed[:, 0:n] = X[:]
    # X_transformed[:, n:2*n] = 1 / X[:]
    # X_transformed[:, 2*n:3*n] = np.log(X[:])
    # X_transformed[:, 3*n:4*n] = X[:] ** 2
    # X_transformed[:, 4*n:5*n] = X[:] ** 0.5
    # X_transformed[:, 5*n:6*n] = X[:] ** 3
    # bias
    return X_transformed
```

[73]:
```
X_train_transformed = transform(X_train)
X_train_transformed.shape, X_train.shape
```

[73]:
```
(torch.Size([156, 14]), torch.Size([156, 13]))
```

[74]:
```
X_train_transformed
```

[74]:
```
tensor([[ 99.8000, 177.3000,  66.3000,  …,  19.0000,  25.0000,   1.0000],
        [ 97.0000, 172.0000,  65.4000,  …,  24.0000,  29.0000,   1.0000],
        [ 99.1000, 186.6000,  66.5000,  …,  21.0000,  28.0000,   1.0000],
        …,
        [ 97.2000, 173.4000,  65.2000,  …,  27.0000,  34.0000,   1.0000],
        [ 94.5000, 159.3000,  64.2000,  …,  24.0000,  29.0000,   1.0000],
        [114.2000, 198.9000,  68.4000,  …,  19.0000,  24.0000,   1.0000]])
```

0.8 2.b)

[80]:
```
# fit a ordinal least squares model using X and Y
X = X_train_transformed
Y = Y_train

beta_opt = torch.linalg.solve(X.T @ X, X.T @ Y)
beta_opt.shape, beta_opt
```

```
/tmp/ipykernel_4630/346199305.py:5: DeprecationWarning: __array_wrap__ must
accept context and return_scalar arguments (positionally) in the future.
(Deprecated NumPy 2.0)
  beta_opt = torch.linalg.solve(X.T @ X, X.T @ Y)
```

[80]:
```
(torch.Size([14, 1]),
 tensor([[ 6.2408e+01],
         [-1.0142e+02],
         [ 6.6769e+02],
         [ 2.6359e+02],
         [ 2.1389e+00],
         [ 9.4635e+01],
         [-1.0336e+03],
         [-2.6506e+03],
         [ 3.1078e+02],
         [ 4.9542e+01],
         [ 1.3534e+00],
         [-3.2724e+02],
         [ 2.5385e+02],
         [-5.3711e+04]]))
```

[81]:
```
from sklearn.metrics import r2_score, mean_squared_error as mse
import numpy as np

# compute R^2 for X_test and Y_test
X_test_transformed = transform(X_test)
Y_test_pred = X_test_transformed @ beta_opt

if not isinstance(Y_test, np.ndarray):
    Y_test = Y_test.numpy()

if not isinstance(Y_test_pred, np.ndarray):
    Y_test_pred = Y_test_pred.numpy()

R2 = r2_score(Y_test, Y_test_pred)
R2
```

[81]:
```
0.8499779105186462
```

0.9 2.c)

[82]:
```python
# train set mse

X_train_transformed = transform(X_train)
X_test_transformed = transform(X_test)
Y_train_pred = X_train_transformed @ beta_opt
Y_test_pred = X_test_transformed @ beta_opt

# convert all to numpy
Y_train_pred = Y_train_pred.numpy()
Y_test_pred = Y_test_pred.numpy()

train_mse = mse(Y_train, Y_train_pred)
print('train mse', train_mse)

train_r2 = r2_score(Y_train, Y_train_pred)

print('train r2', train_r2)

# test set mse

test_mse = mse(Y_test, Y_test_pred)
print('test mse', test_mse)


# test set r2
test_r2 = r2_score(Y_test, Y_test_pred)
print('test r2', test_r2)
```

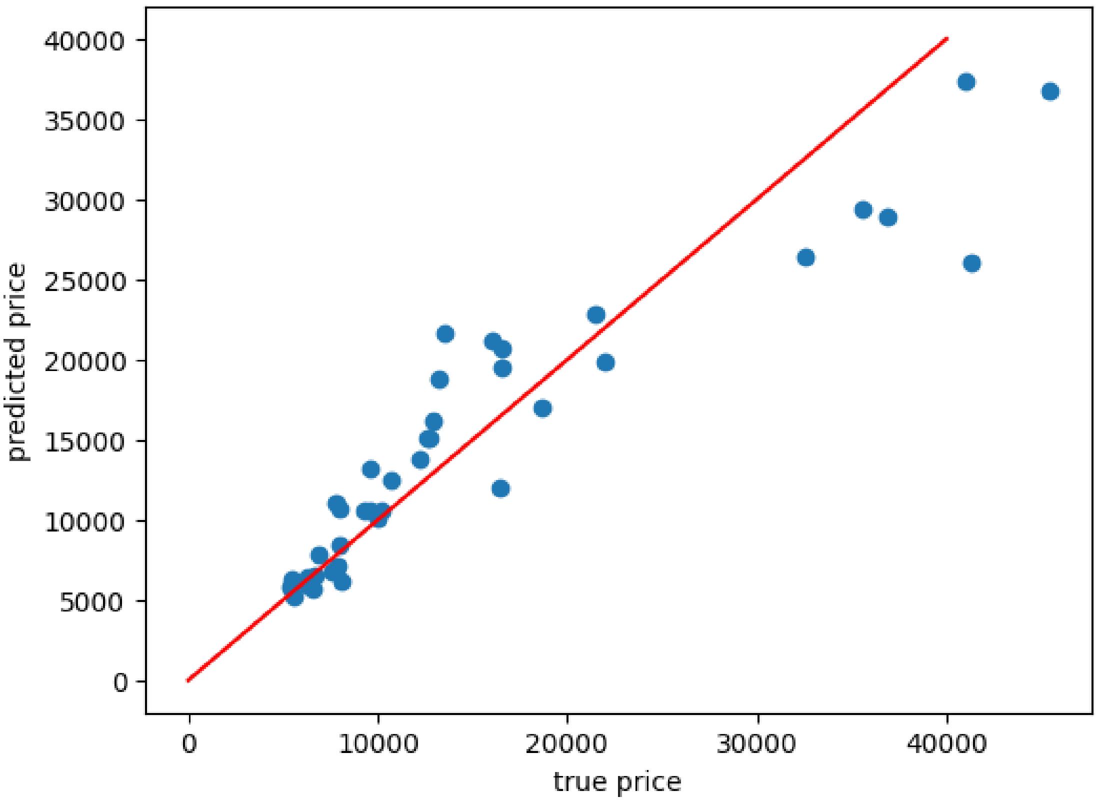
```
train mse 7592032.0
train r2 0.8456655740737915
test mse 18193230.0
test r2 0.8499779105186462
```

[84]:
```python
# put above result into dataframe
result_df = pd.DataFrame({
    'mse': [train_mse, test_mse],
    'r2': [train_r2, test_r2]
}, index=['train', 'test'], dtype='float64')
print(result_df)
print(f'generalization gap: {test_mse - train_mse}')
```

```
              mse        r2
train   7592032.0  0.845666
test   18193230.0  0.849978
generalization gap: 10601198.0
```

0.10 2.d)

[85]:
```
# plot true price vs predicted price
Y_test_pred = X_test_transformed @ beta_opt
plt.scatter(Y_test, Y_test_pred)
plt.xlabel('true price')
plt.ylabel('predicted price')
# plot the line y = x
plt.plot([0, 40000], [0, 40000], color='red')
plt.show()
```



0.11 3.a)

[86]:
```
def p3_transform(X):
    # simply copy X
    return X
```

[90]:
```
# OLS with p3_transform

X_train_transformed = p3_transform(X_train)
X_test_transformed = p3_transform(X_test)
```

```
# find the beta_opt
beta_opt = torch.linalg.solve(X_train_transformed.T @ X_train_transformed,␣
↪X_train_transformed.T @ Y_train)

print('beta_opt', beta_opt.squeeze(-1))


y_pred = X_test_transformed @ beta_opt
y_true = Y_test

# make sure everything is numpy
if not isinstance(y_pred, np.ndarray):
    y_pred = y_pred.numpy()

if not isinstance(y_true, np.ndarray):
    y_true = y_true.numpy()

R2 = r2_score(y_true, y_pred)

print('R2:', round(R2, 5))
```

```
beta_opt tensor([ 1.2890e+02, -8.5229e+01,  2.7172e+01,  5.1442e+01,
3.2452e+00,
         8.9445e+01, -2.4821e+03, -2.9824e+03,  3.6812e+02,  4.7040e+01,
         7.7293e-01, -3.4168e+02,  1.6393e+02])
R2: 0.83259
```

```
/tmp/ipykernel_4630/1495427163.py:7: DeprecationWarning: __array_wrap__ must
accept context and return_scalar arguments (positionally) in the future.
(Deprecated NumPy 2.0)
  beta_opt = torch.linalg.solve(X_train_transformed.T @ X_train_transformed,
X_train_transformed.T @ Y_train)
```

0.12 3.b)

[94]:
```
# MSE and R2 for trainset

X_train_transformed = p3_transform(X_train)
X_test_transformed = p3_transform(X_test)

Y_train_pred = X_train_transformed @ beta_opt
Y_test_pred = X_test_transformed @ beta_opt

# convert all to numpy
if not isinstance(Y_train_pred, np.ndarray):
    Y_train_pred = Y_train_pred.numpy()
```

```
if not isinstance(Y_test_pred, np.ndarray):
    Y_test_pred = Y_test_pred.numpy()

train_R2 = r2_score(Y_train, Y_train_pred)
test_R2 = r2_score(Y_test, Y_test_pred)

train_mse = mse(Y_train, Y_train_pred)
test_mse = mse(Y_test, Y_test_pred)

# to dataframe
result_df = pd.DataFrame({
    'mse': [train_mse, test_mse],
    'r2': [train_R2, test_R2]
}, index=['train', 'test'], dtype='float64')
print(result_df)
print(f'generalization gap: {test_mse - train_mse}')
```

```
              mse        r2
train   8168751.5  0.833942
test   20301772.0  0.832591
generalization gap: 12133020.5
```

0.13 3.c)

Comparing these results, I believe that the model devised in part 2 (with transformation) gives a better R^2 result (0.8500 vs 0.8325) and smaller generalization gap (10601198.0 vs 12133020.5). I would choose model in part 2 for both purposes.

```
[ ]:
```