Module 2 Class Exercises

Class Exercises here are from related sections of the textbook

In [20]:
```
pip install dmba
```

```
Requirement already satisfied: dmba in c:\users\akhil\anaconda3\lib\site-pac
kages (0.2.4)
Requirement already satisfied: graphviz in c:\users\akhil\anaconda3\lib\site
-packages (from dmba) (0.20.1)
Requirement already satisfied: matplotlib in c:\users\akhil\anaconda3\lib\si
te-packages (from dmba) (3.7.2)
Requirement already satisfied: numpy in c:\users\akhil\anaconda3\lib\site-pa
ckages (from dmba) (1.24.3)
Requirement already satisfied: pandas in c:\users\akhil\anaconda3\lib\site-p
ackages (from dmba) (2.0.3)
Requirement already satisfied: scikit-learn in c:\users\akhil\anaconda3\lib
\site-packages (from dmba) (1.3.0)
Requirement already satisfied: scipy in c:\users\akhil\anaconda3\lib\site-pa
ckages (from dmba) (1.11.1)
Requirement already satisfied: contourpy>=1.0.1 in c:\users\akhil\anaconda3
\lib\site-packages (from matplotlib->dmba) (1.0.5)
Requirement already satisfied: cycler>=0.10 in c:\users\akhil\anaconda3\lib
\site-packages (from matplotlib->dmba) (0.11.0)
Requirement already satisfied: fonttools>=4.22.0 in c:\users\akhil\anaconda3
\lib\site-packages (from matplotlib->dmba) (4.25.0)
Requirement already satisfied: kiwisolver>=1.0.1 in c:\users\akhil\anaconda3
\lib\site-packages (from matplotlib->dmba) (1.4.4)
Requirement already satisfied: packaging>=20.0 in c:\users\akhil\anaconda3\l
ib\site-packages (from matplotlib->dmba) (23.1)
Requirement already satisfied: pillow>=6.2.0 in c:\users\akhil\anaconda3\lib
\site-packages (from matplotlib->dmba) (9.4.0)
Requirement already satisfied: pyparsing<3.1,>=2.3.1 in c:\users\akhil\anaco
nda3\lib\site-packages (from matplotlib->dmba) (3.0.9)
Requirement already satisfied: python-dateutil>=2.7 in c:\users\akhil\anacon
da3\lib\site-packages (from matplotlib->dmba) (2.8.2)
Requirement already satisfied: pytz>=2020.1 in c:\users\akhil\anaconda3\lib
\site-packages (from pandas->dmba) (2023.3.post1)
Requirement already satisfied: tzdata>=2022.1 in c:\users\akhil\anaconda3\li
b\site-packages (from pandas->dmba) (2023.3)
Requirement already satisfied: joblib>=1.1.1 in c:\users\akhil\anaconda3\lib
\site-packages (from scikit-learn->dmba) (1.2.0)
Requirement already satisfied: threadpoolctl>=2.0.0 in c:\users\akhil\anacon
da3\lib\site-packages (from scikit-learn->dmba) (2.2.0)
Requirement already satisfied: six>=1.5 in c:\users\akhil\anaconda3\lib\site
-packages (from python-dateutil>=2.7->matplotlib->dmba) (1.16.0)
Note: you may need to restart the kernel to use updated packages.
```

In [21]:
```
%matplotlib inline
from pathlib import Path
import numpy as np
import pandas as pd
from sklearn.model_selection import train_test_split
from sklearn.metrics import r2_score
from sklearn.linear_model import LinearRegression
import dmba

import matplotlib.pylab as plt
```

Table 2.3

Load the West Roxbury data set

In [22]:
```
housing_df = dmba.load_data('WestRoxbury.csv')
```

Determine the shape of the data frame. It has 5802 rows and 14 columns

In [23]:
```
housing_df.shape
```

Out[23]:
```
(5802, 14)
```

Show the top rows of the dataframe

In [24]:
```
housing_df.head()
```

Out[24]:

	TOTAL VALUE	TAX	LOT SQFT	YR BUILT	GROSS AREA	LIVING AREA	FLOORS	ROOMS	BEDROOMS	FULL BATH	HALF BATH
0	344.2	4330	9965	1880	2436	1352	2.0	6	3	1	1
1	412.6	5190	6590	1945	3108	1976	2.0	10	4	2	1
2	330.1	4152	7500	1890	2294	1371	2.0	8	4	1	1
3	498.6	6272	13773	1957	5032	2608	1.0	9	5	1	1
4	331.5	4170	5000	1910	2370	1438	2.0	7	3	2	0

Cleanup

Preprocessing and cleaning up data is an important aspect of data analysis.

Show the column names.

In [25]:
```
housing_df.columns
```

Out[25]:
```
Index(['TOTAL VALUE ', 'TAX', 'LOT SQFT ', 'YR BUILT', 'GROSS AREA ',
       'LIVING AREA', 'FLOORS ', 'ROOMS', 'BEDROOMS ', 'FULL BATH',
       'HALF BATH', 'KITCHEN', 'FIREPLACE', 'REMODEL'],
      dtype='object')
```

Note that some column titles end with spaces and some consist of two space separated words. For further analysis it's more convenient to have column names which are single words.

In the rename command you can specify individual columns by name and provide their new name using a dictionary. Note that we use the `inplace` argument here. This means that the data frame is modified directly. By default, the modification is done on a copy and the copy returned by the method.

In [26]:
```
housing_df = housing_df.rename(columns={'TOTAL VALUE ': 'TOTAL_VALUE'})
housing_df.columns
```

Out[26]:
```
Index(['TOTAL_VALUE', 'TAX', 'LOT SQFT ', 'YR BUILT', 'GROSS AREA ',
       'LIVING AREA', 'FLOORS ', 'ROOMS', 'BEDROOMS ', 'FULL BATH',
       'HALF BATH', 'KITCHEN', 'FIREPLACE', 'REMODEL'],
      dtype='object')
```

We therefore strip trailing spaces and replace the remaining spaces with an underscore _. Instead of using the `rename` method, we create a modified copy of `columns` and assign to the `columns` field of the dataframe.

In [27]:
```
housing_df.columns = [s.strip().replace(' ', '_') for s in housing_df.columns]
housing_df.columns
```

Out[27]:
```
Index(['TOTAL_VALUE', 'TAX', 'LOT_SQFT', 'YR_BUILT', 'GROSS_AREA',
       'LIVING_AREA', 'FLOORS', 'ROOMS', 'BEDROOMS', 'FULL_BATH', 'HALF_BAT
H',
       'KITCHEN', 'FIREPLACE', 'REMODEL'],
      dtype='object')
```

Accessing subsets of the data

Pandas uses two methods to access rows in a data frame; `loc` and `iloc`. The `loc` method is more general and allows accessing rows using labels. The `iloc` method on the other hand only allows using integer numbers. To specify a range of rows use the slice notation, e.g. `0:9`.

To show the first four rows of the data frame, you can use the following commands.

In [28]:
```
housing_df.loc[0:3]  # for loc, the second index in the slice is inclusive
```

Out[28]:

	TOTAL_VALUE	TAX	LOT_SQFT	YR_BUILT	GROSS_AREA	LIVING_AREA	FLOORS	ROOMS
0	344.2	4330	9965	1880	2436	1352	2.0	6
1	412.6	5190	6590	1945	3108	1976	2.0	10
2	330.1	4152	7500	1890	2294	1371	2.0	8
3	498.6	6272	13773	1957	5032	2608	1.0	9

In [29]:
```
housing_df.iloc[0:4]  # for loc, the second index in the slice is exclusive
```

Out[29]:

	TOTAL_VALUE	TAX	LOT_SQFT	YR_BUILT	GROSS_AREA	LIVING_AREA	FLOORS	ROOMS
0	344.2	4330	9965	1880	2436	1352	2.0	6
1	412.6	5190	6590	1945	3108	1976	2.0	10
2	330.1	4152	7500	1890	2294	1371	2.0	8
3	498.6	6272	13773	1957	5032	2608	1.0	9

Note the difference in the two methods with respect to the slice notation! For consistency with how slices are defined in Python, we will use the `iloc` method mostly from here on.

Next, show the first ten rows of the first column

In [30]:
```
housing_df['TOTAL_VALUE'].iloc[0:10]
housing_df.iloc[0:10]['TOTAL_VALUE']  # the order is not important
housing_df.iloc[0:10].TOTAL_VALUE
```

Out[30]:
```
0    344.2
1    412.6
2    330.1
3    498.6
4    331.5
5    337.4
6    359.4
7    320.4
8    333.5
9    409.4
Name: TOTAL_VALUE, dtype: float64
```

Show the fifth row of the first 10 columns. The `iloc` methods allows specifying the rows and columns within one set of brackets. `dataframe.iloc[rows, columns]`

In [31]:
```
housing_df.iloc[4][0:10]
housing_df.iloc[4, 0:10]  # this is equivalent
```

Out[31]:
```
TOTAL_VALUE    331.5
TAX             4170
LOT_SQFT        5000
YR_BUILT        1910
GROSS_AREA      2370
LIVING_AREA     1438
FLOORS           2.0
ROOMS              7
BEDROOMS           3
FULL_BATH          2
Name: 4, dtype: object
```

If you prefer to preserve the data frame format, use a slice for the rows as well.

In [32]:
```
housing_df.iloc[4:5, 0:10]
```

Out[32]:

	TOTAL_VALUE	TAX	LOT_SQFT	YR_BUILT	GROSS_AREA	LIVING_AREA	FLOORS	ROOMS
4	331.5	4170	5000	1910	2370	1438	2.0	7

Use the `pd.concat` method if you want to combine non-consecutive columns into a new data frame. The `axis` argument specifies the dimension along which the concatenation happens, 0=rows, 1=columns.

In [33]:
```
pd.concat([housing_df.iloc[4:6,0:2], housing_df.iloc[4:6,4:6]], axis=1)
```

Out[33]:

	TOTAL_VALUE	TAX	GROSS_AREA	LIVING_AREA
4	331.5	4170	2370	1438
5	337.4	4244	2124	1060

To specify a full column, use the `:` on its own.

```
housing_df.iloc[:,0:1]
```

A often more practical way is to use the column name as follows

In [34]:
```
housing_df['TOTAL_VALUE']
```

Out[34]:
```
0       344.2
1       412.6
2       330.1
3       498.6
4       331.5
        ...
5797    404.8
5798    407.9
5799    406.5
5800    308.7
5801    447.6
Name: TOTAL_VALUE, Length: 5802, dtype: float64
```

We can subset the column using a slice

In [35]:
```
housing_df['TOTAL_VALUE'][0:10]
```

Out[35]:
```
0    344.2
1    412.6
2    330.1
3    498.6
4    331.5
5    337.4
6    359.4
7    320.4
8    333.5
9    409.4
Name: TOTAL_VALUE, dtype: float64
```

Pandas provides a number of ways to access statistics of the columns.

In [36]:
```
print('Number of rows ', len(housing_df['TOTAL_VALUE']))
print('Mean of TOTAL_VALUE ', housing_df['TOTAL_VALUE'].mean())
```

```
Number of rows  5802
Mean of TOTAL_VALUE  392.6857149258877
```

A data frame also has the method `describe` that prints a number of common statistics

In [37]:
```
housing_df['TOTAL_VALUE'].describe()
```

Out[37]:
```
count    5802.000000
mean      392.685715
std        99.177414
min       105.000000
25%       325.125000
50%       375.900000
75%       438.775000
max      1217.800000
Name: TOTAL_VALUE, dtype: float64
```

In [38]:
```
housing_df.describe()
```

Out[38]:

	TOTAL_VALUE	TAX	LOT_SQFT	YR_BUILT	GROSS_AREA	LIVING_AREA
count	5802.000000	5802.000000	5802.000000	5802.000000	5802.000000	5802.000000
mean	392.685715	4939.485867	6278.083764	1936.744916	2924.842123	1657.065322
std	99.177414	1247.649118	2669.707974	35.989910	883.984726	540.456726
min	105.000000	1320.000000	997.000000	0.000000	821.000000	504.000000
25%	325.125000	4089.500000	4772.000000	1920.000000	2347.000000	1308.000000
50%	375.900000	4728.000000	5683.000000	1935.000000	2700.000000	1548.500000
75%	438.775000	5519.500000	7022.250000	1955.000000	3239.000000	1873.750000
max	1217.800000	15319.000000	46411.000000	2011.000000	8154.000000	5289.000000

Table 2.4

Use the `sample` method to retrieve a random sample of observations. Here we sample 5 observations without replacement.

In [39]:
```
housing_df.sample(5)
```

Out[39]:

	TOTAL_VALUE	TAX	LOT_SQFT	YR_BUILT	GROSS_AREA	LIVING_AREA	FLOORS	ROO
3030	396.6	4989	5500	1920	3551	1918	2.0	
5144	347.2	4367	5000	1920	2437	1476	2.0	
3508	715.0	8994	12189	1950	4234	2388	2.0	
3437	507.9	6389	6996	1885	3492	2192	2.0	
3465	424.9	5345	5000	1956	2800	1876	2.0	

The sample method allows to specify weights for the individual rows. We use this here to oversample houses with over 10 rooms.

In [40]:
```
weights = [0.9 if rooms > 10 else 0.01 for rooms in housing_df.ROOMS]
housing_df.sample(5, weights=weights)
```

Out[40]:

	TOTAL_VALUE	TAX	LOT_SQFT	YR_BUILT	GROSS_AREA	LIVING_AREA	FLOORS	ROO
1607	621.6	7819	8106	2001	2946	2788	2.0	
3585	590.2	7424	8562	1904	5005	3085	2.0	
5218	687.9	8653	9364	2004	4228	3022	2.0	
5716	376.2	4732	6110	1940	2810	1768	2.0	
2873	645.7	8122	10569	1901	4760	2933	2.0	

Table 2.5

In [41]:
```
housing_df.columns
```

Out[41]:
```
Index(['TOTAL_VALUE', 'TAX', 'LOT_SQFT', 'YR_BUILT', 'GROSS_AREA',
       'LIVING_AREA', 'FLOORS', 'ROOMS', 'BEDROOMS', 'FULL_BATH', 'HALF_BAT
H',
       'KITCHEN', 'FIREPLACE', 'REMODEL'],
      dtype='object')
```

The REMODEL column is a factor, so we need to change it's type.

In [42]:
```
print(housing_df.REMODEL.dtype)
housing_df.REMODEL = housing_df.REMODEL.astype('category')
print(housing_df.REMODEL.cat.categories)  # It can take one of three levels
print(housing_df.REMODEL.dtype)  # Type is now 'category'
```

```
object
Index(['Old', 'Recent'], dtype='object')
category
```

Other columns also have types.

In [43]:
```
print(housing_df.BEDROOMS.dtype)  # BEDROOMS is an integer variable
print(housing_df.TOTAL_VALUE.dtype)  # Total_Value is a numeric variable
```

```
int64
float64
```

It's also possible to the all columns data types

In [44]:
```
housing_df.dtypes
```

Out[44]:
```
TOTAL_VALUE     float64
TAX               int64
LOT_SQFT          int64
YR_BUILT          int64
GROSS_AREA        int64
LIVING_AREA       int64
FLOORS          float64
ROOMS             int64
BEDROOMS          int64
FULL_BATH         int64
HALF_BATH         int64
KITCHEN           int64
FIREPLACE         int64
REMODEL        category
dtype: object
```

Table 2.6

Pandas provides a method to convert factors into dummy variables. In older versions of pandas, the missing values were treated as a separate category. In the book code, we therefore removed the first dummy variable. With newer versions of pandas, we can now call `get_dummies` with the default value of `drop_first=False`.

In [45]:
```
# the missing values will create a third category
# use the arguments drop_first and dummy_na to control the outcome
housing_df = pd.get_dummies(housing_df, prefix_sep='_', dtype=int)
housing_df.columns
```

Out[45]:
```
Index(['TOTAL_VALUE', 'TAX', 'LOT_SQFT', 'YR_BUILT', 'GROSS_AREA',
       'LIVING_AREA', 'FLOORS', 'ROOMS', 'BEDROOMS', 'FULL_BATH', 'HALF_BAT
H',
       'KITCHEN', 'FIREPLACE', 'REMODEL_Old', 'REMODEL_Recent'],
      dtype='object')
```

In [47]:
```
print(housing_df.loc[:, 'REMODEL_Old':'REMODEL_Recent'].head(5))
```

```
   REMODEL_Old  REMODEL_Recent
0            0               0
1            0               1
2            0               0
3            0               0
4            0               0
```

Table 2.7

To illustrate missing data procedures, we first convert a few entries for bedrooms to NA's. Then we impute these missing values using the median of the remaining values.

In [48]:
```
print('Number of rows with valid BEDROOMS values before: ',
      housing_df['BEDROOMS'].count())
missingRows = housing_df.sample(10).index
housing_df.loc[missingRows, 'BEDROOMS'] = np.nan
print('Number of rows with valid BEDROOMS values after setting to NAN: ',
      housing_df['BEDROOMS'].count())
housing_df['BEDROOMS'].count()
```

```
Number of rows with valid BEDROOMS values before:  5802
Number of rows with valid BEDROOMS values after setting to NAN:  5792
```

Out[48]:
```
5792
```

In [49]:
```
# remove rows with missing values
reduced_df = housing_df.dropna()
print('Number of rows after removing rows with missing values: ', len(reduced_
```

```
Number of rows after removing rows with missing values:  5792
```

Replace the missing values using the median of the remaining values.

By default, the `median` method of a pandas dataframe ignores NA values. This is in contrast to R where this must be specified explicitly.

In [50]:
```
medianBedrooms = housing_df['BEDROOMS'].median()
housing_df.BEDROOMS = housing_df.BEDROOMS.fillna(value=medianBedrooms)
print('Number of rows with valid BEDROOMS values after filling NA values: ',
      housing_df['BEDROOMS'].count())
```

```
Number of rows with valid BEDROOMS values after filling NA values:  5802
```

Table - scaling data

In [52]:
```
from sklearn.preprocessing import MinMaxScaler, StandardScaler
df = housing_df.copy()

# Normalizing a data frame

# pandas:
norm_df = (housing_df - housing_df.mean()) / housing_df.std()

# scikit-learn:
scaler = StandardScaler()
norm_df = pd.DataFrame(scaler.fit_transform(housing_df),
                       index=housing_df.index, columns=housing_df.columns)
# the result of the transformation is a numpy array, we convert it into a data

# Rescaling a data frame
# pandas:
rescaled_df = (housing_df - housing_df.min()) / (housing_df.max() - housing_df

# scikit-learn:
scaler = MinMaxScaler()
rescaled_df = pd.DataFrame(scaler.fit_transform(housing_df),
                       index=housing_df.index, columns=housing_df.columns)
```

The standardization of the dataset may give a `DataConversionWarning` . This informs you that the integer columns in the dataframe are automatically converted to real numbers (`float64`). This is expected and you can therefore ignore this warning. If you want to suppress the warning, you can explicitly convert the integer columns to real numbers

```
# Option 1: Identify all integer columns, remove personal loan,
# and change their type
intColumns = [c for c in housing_df.columns if housing_df[c].dtype ==
'int']
housing_df[intColumns] = housing_df[intColumns].astype('float64')
```

Alternatively, you can suppress the warning as follows:

```
# Option 2: use the warnings package to suppress the display of the w
arning
import warnings
with warnings.catch_warnings():
    warnings.simplefilter('ignore')
    norm_df = pd.DataFrame(scaler.fit_transform(housing_df),
                           index=housing_df.index, columns=housing_df.col
umns)
```

Table 2.9

Split the dataset into training (60%) and validation (40%) sets. Randomly sample 60% of the dataset into a new data frame `trainData` . The remaining 40% serve as validation.

In [53]:
```
# random_state is set to a defined value to get the same partitions when re-ru
trainData= housing_df.sample(frac=0.6, random_state=1)
# assign rows that are not already in the training set, into validation
validData = housing_df.drop(trainData.index)

print('Training   : ', trainData.shape)
print('Validation : ', validData.shape)
print()

# alternative way using scikit-learn
trainData, validData = train_test_split(housing_df, test_size=0.40, random_sta
print('Training   : ', trainData.shape)
print('Validation : ', validData.shape)
```

```
Training   :  (3481, 15)
Validation :  (2321, 15)

Training   :  (3481, 15)
Validation :  (2321, 15)
```

Partition the dataset into training (50%), validation (30%), and test sets (20%).

In [54]:
```
# randomly sample 50% of the row IDs for training
trainData = housing_df.sample(frac=0.5, random_state=1)
# sample 30% of the row IDs into the validation set, drawing only from records
# not already in the training set; 60% of 50% is 30%
validData = housing_df.drop(trainData.index).sample(frac=0.6, random_state=1)
# the remaining 20% rows serve as test
testData = housing_df.drop(trainData.index).drop(validData.index)

print('Training   : ', trainData.shape)
print('Validation : ', validData.shape)
print('Test       : ', testData.shape)
print()

# alternative way using scikit-learn
trainData, temp = train_test_split(housing_df, test_size=0.5, random_state=1)
validData, testData = train_test_split(temp, test_size=0.4, random_state=1)
print('Training   : ', trainData.shape)
print('Validation : ', validData.shape)
print('Test       : ', testData.shape)
```

```
Training   :  (2901, 15)
Validation :  (1741, 15)
Test       :  (1160, 15)

Training   :  (2901, 15)
Validation :  (1740, 15)
Test       :  (1161, 15)
```

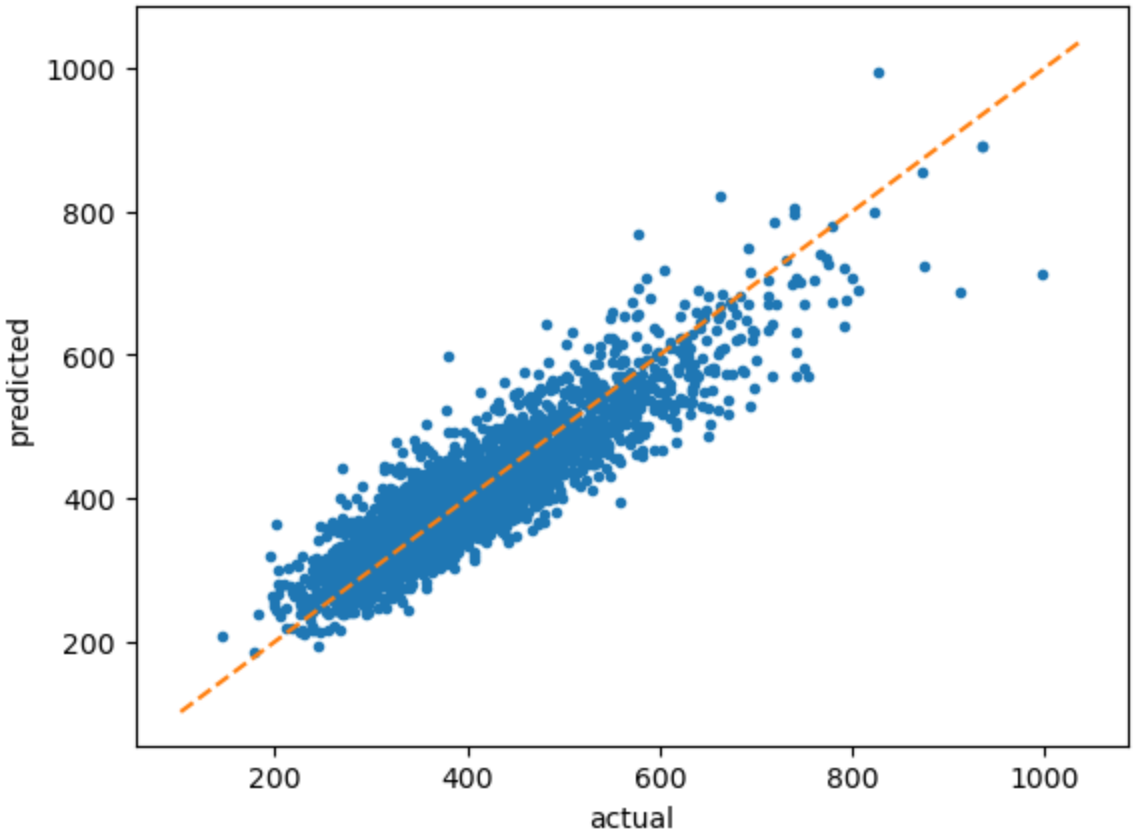
Table 2.11

The statsmodels package allows to define linear regression models using a formula definition similar to R. In contrast to R, all variables need to be specified explicitly. We construct a formula excluding the dependent variable and the `TAX` column

In [55]:
```
# Data loading and preprocessing
housing_df = dmba.load_data('WestRoxbury.csv')
housing_df.columns = [s.strip().replace(' ', '_') for s in housing_df.columns]
housing_df = pd.get_dummies(housing_df, prefix_sep='_', drop_first=False, dtyp

excludeColumns = ('TOTAL_VALUE', 'TAX')
predictors = [s for s in housing_df.columns if s not in excludeColumns]
outcome = 'TOTAL_VALUE'

# partition data
X = housing_df[predictors]
y = housing_df[outcome]
train_X, valid_X, train_y, valid_y = train_test_split(X, y, test_size=0.4, ran

model = LinearRegression()
model.fit(train_X, train_y)

train_pred = model.predict(train_X)
train_results = pd.DataFrame({
    'TOTAL_VALUE': train_y,
    'predicted': train_pred,
    'residual': train_y - train_pred
})
print(train_results.head())
```

```
      TOTAL_VALUE   predicted   residual
2024        392.0  387.726258   4.273742
5140        476.3  430.785540  45.514460
5259        367.4  384.042952 -16.642952
421         350.3  369.005551 -18.705551
1401        348.1  314.725722  33.374278
```

In [56]:
```
plt.plot(train_results.TOTAL_VALUE, train_results.predicted, '.')
plt.xlabel('actual')  # set x-axis label
plt.ylabel('predicted')  # set y-axis label
axes = plt.gca()
plt.plot(axes.get_xlim(), axes.get_xlim(), '--')
plt.show()
```



Predict the validation data

In [57]:
```
valid_pred = model.predict(valid_X)
valid_results = pd.DataFrame({
    'TOTAL_VALUE': valid_y,
    'predicted': valid_pred,
    'residual': valid_y - valid_pred
})
print(valid_results.head())
```

```
      TOTAL_VALUE   predicted   residual
1822        462.0  406.946377  55.053623
1998        370.4  362.888928   7.511072
5126        407.4  390.287208  17.112792
808         316.1  382.470203 -66.370203
4034        393.2  434.334998 -41.134998
```

In [58]:
```
plt.plot(valid_results.TOTAL_VALUE, valid_results.predicted, '.')
plt.xlabel('actual')  # set x-axis label
plt.ylabel('predicted')  # set y-axis label
axes = plt.gca()
plt.plot(axes.get_xlim(), axes.get_xlim(), '--')
plt.show()
```

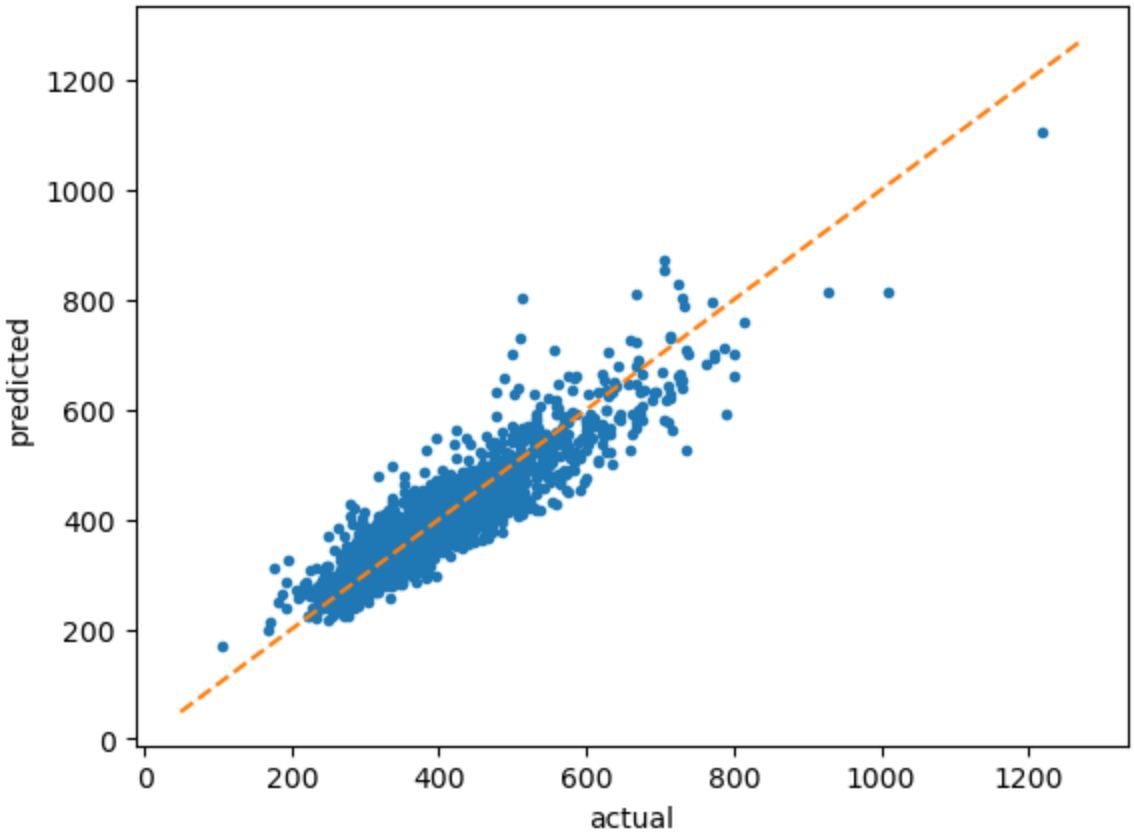


Table 2.13

We can use the metrics that scikit-learn provides.

In [59]:
```
print('Training set r2: ', r2_score(train_results.TOTAL_VALUE, train_results.p
print('Validation set r2: ', r2_score(valid_results.TOTAL_VALUE, valid_results
```

```
Training set r2:  0.8097361461091853
Validation set r2:  0.8171327286147877
```

In [60]:
```
# import the utility function regressionSummary
from dmba import regressionSummary

# training set
regressionSummary(train_results.TOTAL_VALUE, train_results.predicted)

# validation set
regressionSummary(valid_results.TOTAL_VALUE, valid_results.predicted)
```

```
Regression statistics

                    Mean Error (ME) : -0.0000
         Root Mean Squared Error (RMSE) : 43.0306
              Mean Absolute Error (MAE) : 32.6042
            Mean Percentage Error (MPE) : -1.1116
Mean Absolute Percentage Error (MAPE) : 8.4886

Regression statistics

                    Mean Error (ME) : -0.1463
         Root Mean Squared Error (RMSE) : 42.7292
              Mean Absolute Error (MAE) : 31.9663
            Mean Percentage Error (MPE) : -1.0884
Mean Absolute Percentage Error (MAPE) : 8.3283
```

Table 2.14

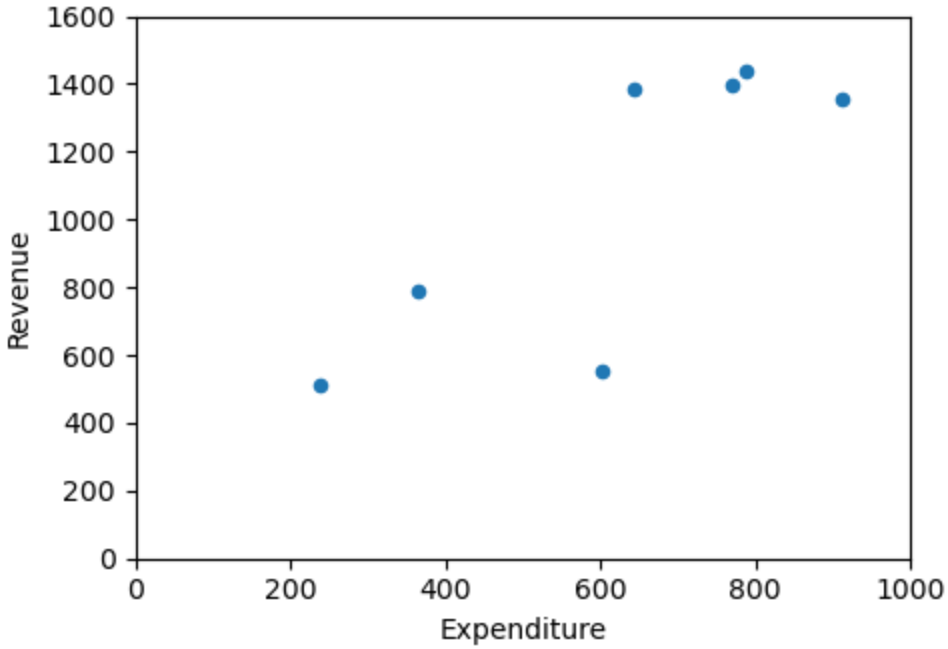
Figure 2.2 and 2.3

In [61]:
```
df = pd.DataFrame({'Expenditure': [239, 364,602, 644, 770, 789, 911],
                   'Revenue': [514, 789, 550, 1386, 1394, 1440, 1354]})
df
```

Out[61]:

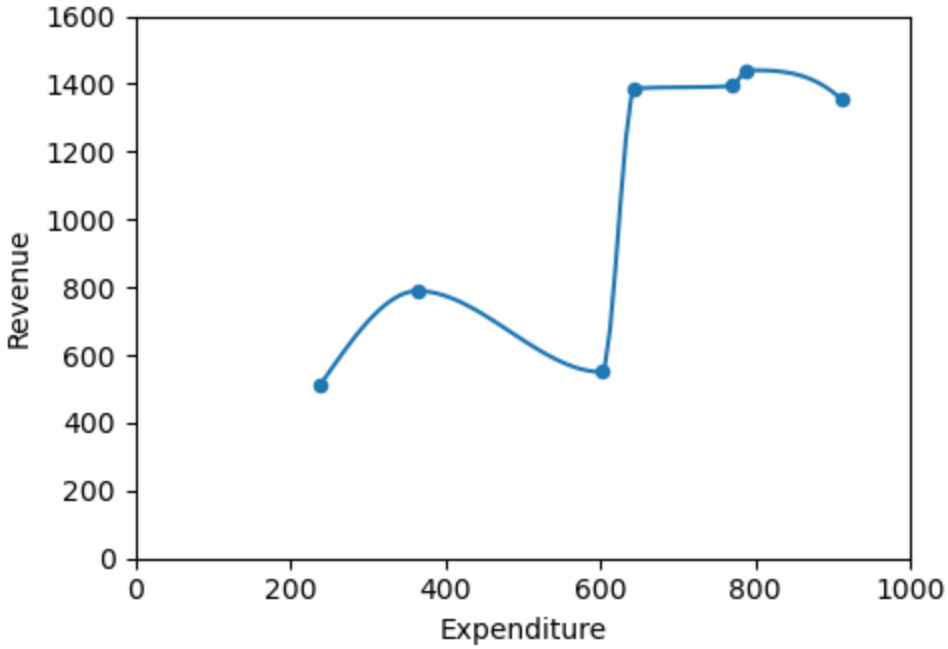
	Expenditure	Revenue
0	239	514
1	364	789
2	602	550
3	644	1386
4	770	1394
5	789	1440
6	911	1354

In [62]:
```
df.plot.scatter(x='Expenditure', y='Revenue', xlim=(0, 1000), ylim=(0, 1600),
plt.tight_layout()  # Increase the separation between the plots
plt.show()
```



In [67]:
```
x = list(df.Expenditure)
y = list(df.Revenue)

from scipy import interpolate
f = interpolate.PchipInterpolator(x, y)

x_new = np.linspace(x[0], x[-1], 100)
y_new = [f(xi) for xi in x_new]
```

In [68]:
```
df.plot.scatter(x='Expenditure', y='Revenue', xlim=(0, 1000), ylim=(0, 1600),
plt.plot(x_new, y_new)
plt.tight_layout()  # Increase the separation between the plots
plt.show()
```



In [ ]:
```

```