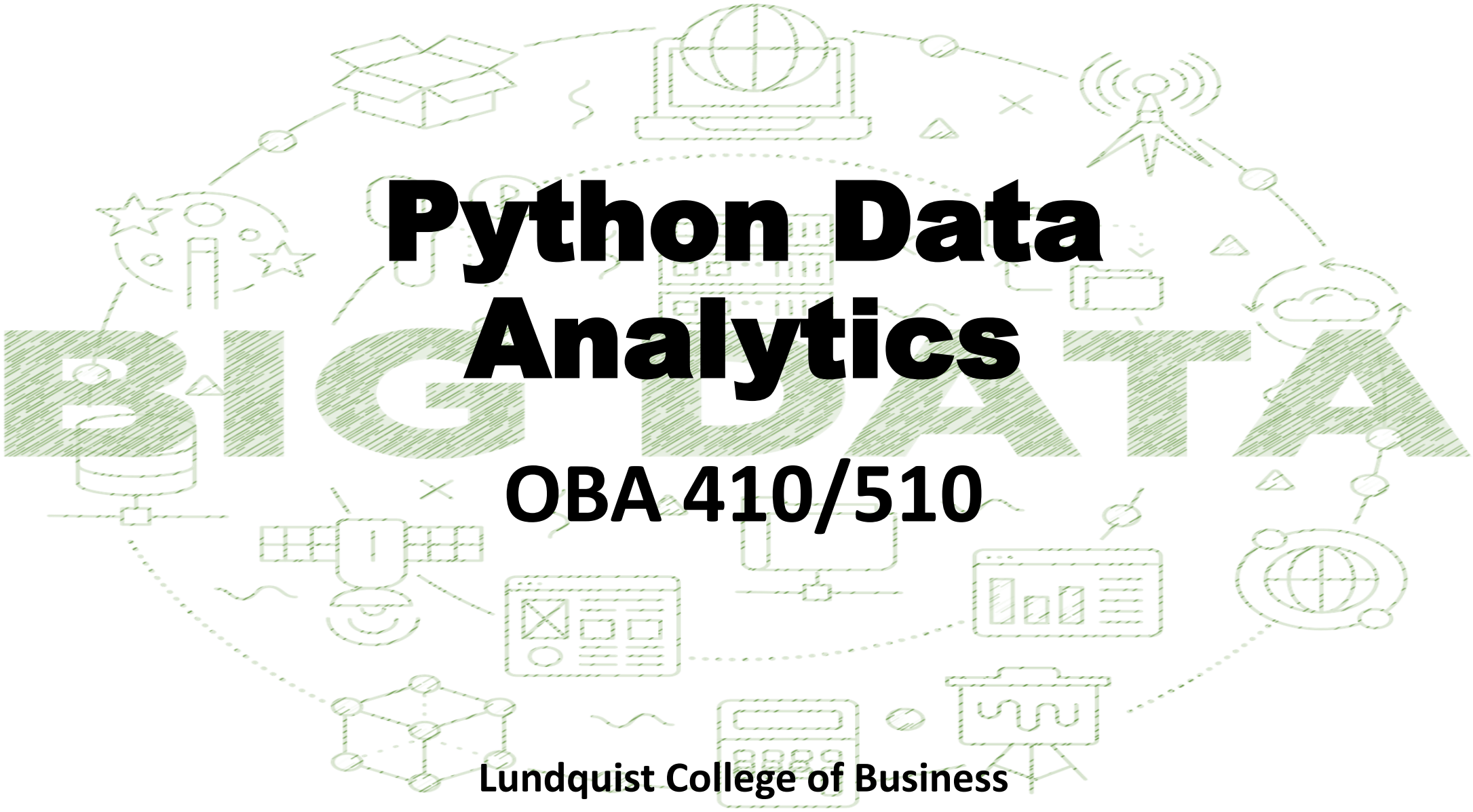


Python Data Analytics

OBA 410/510

Lundquist College of Business

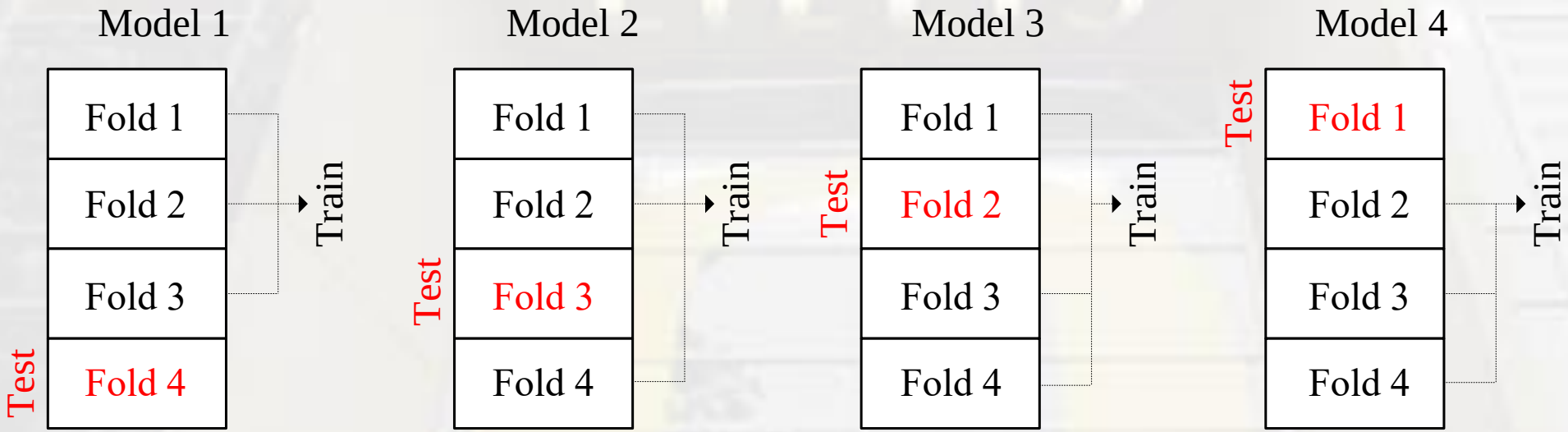
Background image from www. givweb.com

Some of the slides and contents are adopted from Statistical Learning course by Dr. Trevor Hastie

Cross-Validation

- While the method of splitting data into training and test is workable and commonly used, it is sensitive to how exactly the data is split
- ***Cross-validation*** is a more stable method for evaluating the generalizability of models
- The most commonly used version of cross-validation is ***k-fold cross-validation***
- ***k*** is user specified and usually is 5 or 10

Cross-Validation



- 4-fold cross-validation
- Data is partitioned into 4 parts of equal size, called ***folds***
- 4 models are trained
- First model is trained using the first 3 folds, and uses fold 4 as test to be evaluated
- This process will be repeated 4 times and each time we have an accuracy value

Cross-Validation in scikit-learn

- Cross-validation is implemented in scikit-learn using ***cross_val_score*** function
- Parameters are the model we want to evaluate, features of our data, and the target labels
- Default is 3-fold, we can change it using ***cv*** parameter
- We usually consider the mean of the models' accuracy on different folds

Cross-Validation in scikit-learn

```
cancer=pd.read_csv('breast_cancer_data.csv',index_col=0)
X, y=cancer.iloc[:,1:], cancer['diagnosis']
```

```
]: from sklearn.model_selection import cross_val_score
   knn=KNeighborsClassifier(n_neighbors=7)
```

```
]: scores=cross_val_score(knn,X,y,cv=5)
   scores
```

```
]: array([0.87826087, 0.93913043, 0.94690265, 0.94690265, 0.92035398])
```

```
]: print('Average cross-validation accuracy is {:.3f}'.format(scores.mean()))
```

```
Average cross-validation accuracy is 0.926
```

Cross-Validation in scikit-learn

- Practice cross validation on Boston housing data using k-NN regressor (Use **6 folds**)

- Note: For regression problems it is always better to shuffle the data before using cross-validation
- The reason is that data might be sorted and cross validation just divides it to k partition in order, therefore the partitioning might not be random

```
In [41]:
1 boston=pd.read_csv('boston_housing_data.csv',
2 boston.head()
```

Out[41]:

ID	crim	zn	indus	chas	nox	rm	age	dis	ra
1	0.00632	18.0	2.31	0	0.538	6.575	65.2	4.0900	
2	0.02731	0.0	7.07	0	0.469	6.421	78.9	4.9671	
4	0.03237	0.0	2.18	0	0.458	6.998	45.8	6.0622	
5	0.06905	0.0	2.18	0	0.458	7.147	54.2	6.0622	
7	0.08829	12.5	7.87	0	0.524	6.012	66.6	5.5605	

```
In [42]:
1 boston=boston.sample(frac=1,random_state=0)
```

Cross-Validation in scikit-learn

UNIVERSITY OF OREGON
Lundquist College of Business

In [43]:
```
1 X_boston, y_boston=boston.iloc[:,:-1],boston['target_medv']
```

In [49]:
```
1 X_boston.head(n=1)
```

Out[49]:

ID	crim	zn	indus	chas	nox	rm	age	dis	rad	tax	ptratio	lstat
84	0.03551	25.0	4.86	0	0.426	6.167	46.7	5.4007	4	281	19.0	7.51

In [45]:
```
1 y_boston.head(n=2)
```

Out[45]:
```
ID
84    22.9
81    28.0
Name: target_medv, dtype: float64
```

In [46]:
```
1 kNN_cls=KNeighborsRegressor(n_neighbors=5)
```

In [47]:
```
1 scores1=cross_val_score(kNN_cls,X_boston,y_boston,cv=6)
2 scores1
```

Out[47]:
```
array([0.60460082, 0.50661903, 0.56194114, 0.38164372, 0.73683233,
       0.6239456 ])
```

In [48]:
```
1 scores1.mean()
```

Out[48]:
```
0.5692637726200642
```

Cross-Validation benefits

- train_test_split performs a random split, so,
  - If we are lucky, all easy-to-classify data points might be in the test set, and our test score will be unrealistically high
  - If we are unlucky, all hard-to-classify data points might be in the test set, and our test score will be unrealistically low
- However, in cross-validation, every data point will be in a test set exactly once, therefore, model needs to generalize well for all data points
- Having multiple splits of data provides some information about the sensitivity of our model to the selection of training dataset
- In cross-validation we use our data more effectively, for instance, in 10-fold cross validation, we use 90% data to train the model every time. More data usually results in more accurate models.

Generalization, Overfitting, and Underfitting

- In supervised learning, we want to build a model on training data, and then be able to make accurate predictions on new/unseen data
- If a model can make accurate predictions on unseen data, we say it is able to ***generalize*** from the training dataset to test dataset.
- If we develop very complex models, we can be very accurate on training data, but we might not get accurate predictions on test data
- Therefore, we want models that work good on unseen datasets

Generalization, Overfitting, and Underfitting

- Building models that are too complex for the amount of information we have is called ***overfitting***.
- Overfitting occurs when you fit a model too closely to the particularities of the training set
- Overfitted models work really well on training data, but they cannot generalize to test/new data

Overfitting

- "if a customer is older than 50, and has less than 3 children or is not divorced, then they buy a boat"

Table 2-1. Example data about customers

Age	Number of cars owned	Owns house	Number of children	Marital status	Owns a dog	Bought a boat
66	1	yes	2	widowed	no	yes
52	2	yes	3	married	no	yes
22	0	no	0	married	yes	no
25	1	no	1	single	no	no
44	0	no	2	divorced	yes	no
39	1	yes	2	married	yes	no
26	1	no	2	single	no	no
40	3	yes	1	married	yes	no
53	2	yes	2	divorced	no	yes
64	2	yes	3	divorced	no	no
58	2	yes	2	married	yes	yes
33	1	no	1	single	no	no

Generalization, Overfitting, and Underfitting

- If your model is too simple, then you might not be able to capture all aspects and variability in the data
- In this case, your model will do badly even on the training set
- Choosing too simple a model is called ***underfitting***
- There is sweet spot in between that will yield the best generalization performance

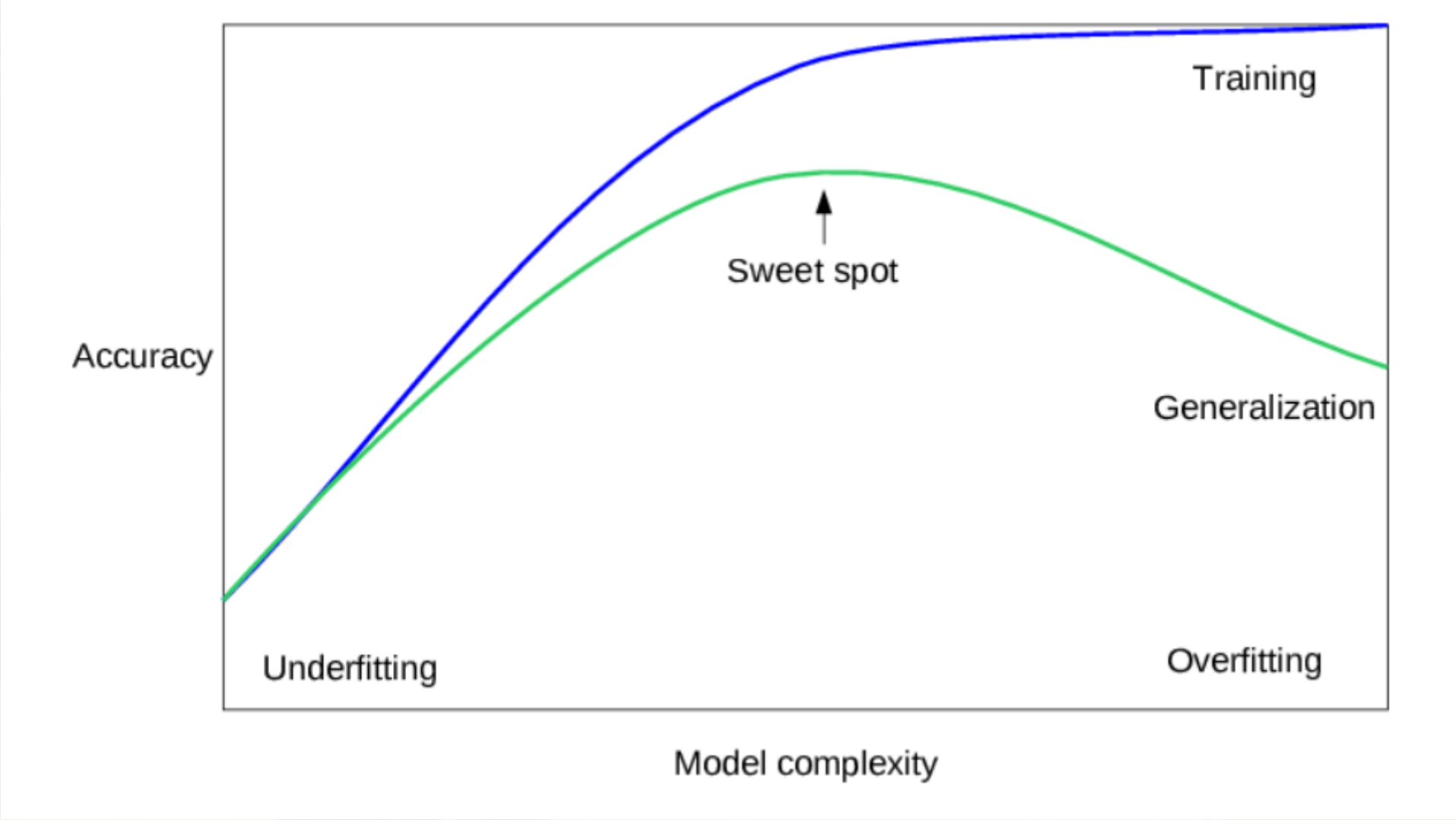
Underfitting

- "Everybody who owns a house will buy a boat"

Table 2-1. Example data about customers

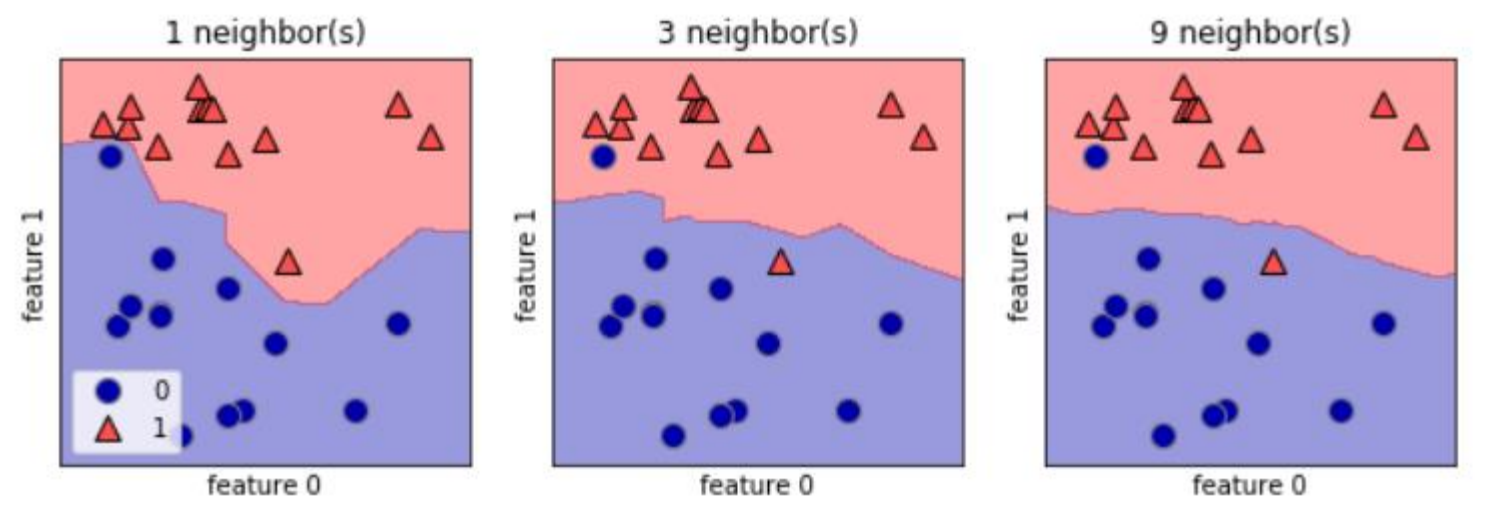
Age	Number of cars owned	Owns house	Number of children	Marital status	Owns a dog	Bought a boat
66	1	yes	2	widowed	no	yes
52	2	yes	3	married	no	yes
22	0	no	0	married	yes	no
25	1	no	1	single	no	no
44	0	no	2	divorced	yes	no
39	1	yes	2	married	yes	no
26	1	no	2	single	no	no
40	3	yes	1	married	yes	no
53	2	yes	2	divorced	no	yes
64	2	yes	3	divorced	no	no
58	2	yes	2	married	yes	yes
33	1	no	1	single	no	no

Generalization, Overfitting, and Underfitting



Analyzing KNeighborsClassifier

- For 2-dimensional datasets, we can illustrate the prediction for all possible test points in the xy-plane
- Decision boundary, divides between where algorithm assigns class 0 versus class 1



Analyzing KNeighborsClassifier

- Using a single neighbor results in a decision boundary that follows the training data closely
- Considering more and more neighbors leads to a smoother decision boundary
- A smoother decision boundary corresponds to simpler model (using few neighbors: complex models, using many neighbors: simpler model)
- The extreme case: number of neighbors is the number of all data points in the training set
  - all predictions will be the same: the class that is more frequent in the training set

Analyzing KNeighborsClassifier

```
import pandas as pd
import numpy as np
from sklearn.model_selection import train_test_split
```

```
cancer=pd.read_csv('breast_cancer_data.csv',index_col=0)
cancer.head()
```

id	diagnosis	radius_mean	texture_mean
842302	M	17.99	10.38

```
X, y=cancer.iloc[:,1:], cancer['diagnosis']
X_train, X_test, y_train, y_test=train_test_split(X, y, random_state=0)
```

```
from sklearn.neighbors import KNeighborsClassifier
knn1=KNeighborsClassifier(n_neighbors=1)
knn5=KNeighborsClassifier(n_neighbors=5)
knn9=KNeighborsClassifier(n_neighbors=9)
knn29=KNeighborsClassifier(n_neighbors=29)
```

```
knn1.fit(X_train,y_train)
knn5.fit(X_train,y_train)
knn9.fit(X_train,y_train)
knn29.fit(X_train,y_train)
```

Analyzing KNeighborsClassifier

```
print('knn1 acc on train:', knn1.score(X_train,y_train), 'knn1 acc on test:', knn1.score(X_test,y_test))
print('knn5 acc on train:', knn5.score(X_train,y_train), 'knn5 acc on test:', knn5.score(X_test,y_test))
print('knn9 acc on train:', knn9.score(X_train,y_train), 'knn9 acc on test:', knn9.score(X_test,y_test))
print('knn29 acc on train:', knn29.score(X_train,y_train), 'knn29 acc on test:', knn29.score(X_test,y_test))
```

```
knn1 acc on train: 1.0 knn1 acc on test: 0.916083916083916
knn5 acc on train: 0.9413145539906104 knn5 acc on test: 0.9370629370629371
knn9 acc on train: 0.9366197183098591 knn9 acc on test: 0.958041958041958
knn29 acc on train: 0.9154929577464789 knn29 acc on test: 0.9370629370629371
```

Analyzing KNeighborsRegressor

- Boston housing data

```
# reading the data
boston=pd.read_csv('boston_housing_data.csv',index_col=0)
boston.head(3)
```

ID	crim	zn	indus	chas	nox	rm	age	dis	rad	tax	ptratio	lstat	target_medv
1	0.00632	18.0	2.31	0	0.538	6.575	65.2	4.0900	1	296	15.3	4.98	24.0
2	0.02731	0.0	7.07	0	0.469	6.421	78.9	4.9671	2	242	17.8	9.14	21.6
4	0.03237	0.0	2.18	0	0.458	6.998	45.8	6.0622	3	222	18.7	2.94	33.4

```
# creating features and target sets
X_boston, y_boston=boston.iloc[:,:-1], boston.iloc[:,-1]
X_boston.head(3)
```

ID	crim	zn	indus	chas	nox	rm	age	dis	rad	tax	ptratio	lstat
1	0.00632	18.0	2.31	0	0.538	6.575	65.2	4.0900	1	296	15.3	4.98
2	0.02731	0.0	7.07	0	0.469	6.421	78.9	4.9671	2	242	17.8	9.14
4	0.03237	0.0	2.18	0	0.458	6.998	45.8	6.0622	3	222	18.7	2.94

Analyzing KNeighborsRegressor

```
# splitting to train and test
X_train1, X_test1, y_train1, y_test1=train_test_split(X_boston, y_boston, random_state=0)
```

```
X_boston.shape
```

```
(333, 12)
```

```
X_train1.shape
```

```
(249, 12)
```

```
# defining the models
kNN_reg2=KNeighborsRegressor(n_neighbors=2)
kNN_reg5=KNeighborsRegressor(n_neighbors=6)
kNN_reg25=KNeighborsRegressor(n_neighbors=25)
```

```
# fitting the models
kNN_reg2.fit(X_train1,y_train1)
kNN_reg5.fit(X_train1,y_train1)
kNN_reg25.fit(X_train1,y_train1)
```

```
KNeighborsRegressor(algorithm='auto', leaf_size=30, metric='minkowski',
          metric_params=None, n_jobs=None, n_neighbors=25, p=2,
          weights='uniform')
```

Analyzing KNeighborsRegressor

```
]: # evaluating the models
print('knn1 acc on train: {:.2%}'.format(kNN_reg2.score(X_train1,y_train1)))
print('knn1 acc on test: {:.2%}'.format(kNN_reg2.score(X_test1,y_test1)),'\n')
print('knn5 acc on train: {:.2%}'.format(kNN_reg5.score(X_train1,y_train1)))
print('knn5 acc on test: {:.2%}'.format(kNN_reg5.score(X_test1,y_test1)),'\n')
print('knn9 acc on train: {:.2%}'.format(kNN_reg25.score(X_train1,y_train1)))
print('knn9 acc on test: {:.2%}'.format(kNN_reg25.score(X_test1,y_test1)))
```

```
knn1 acc on train: 88.21%
knn1 acc on test: 64.22%

knn5 acc on train: 68.80%
knn5 acc on test: 64.36%

knn9 acc on train: 34.73%
knn9 acc on test: 33.39%
```

Linear Regression

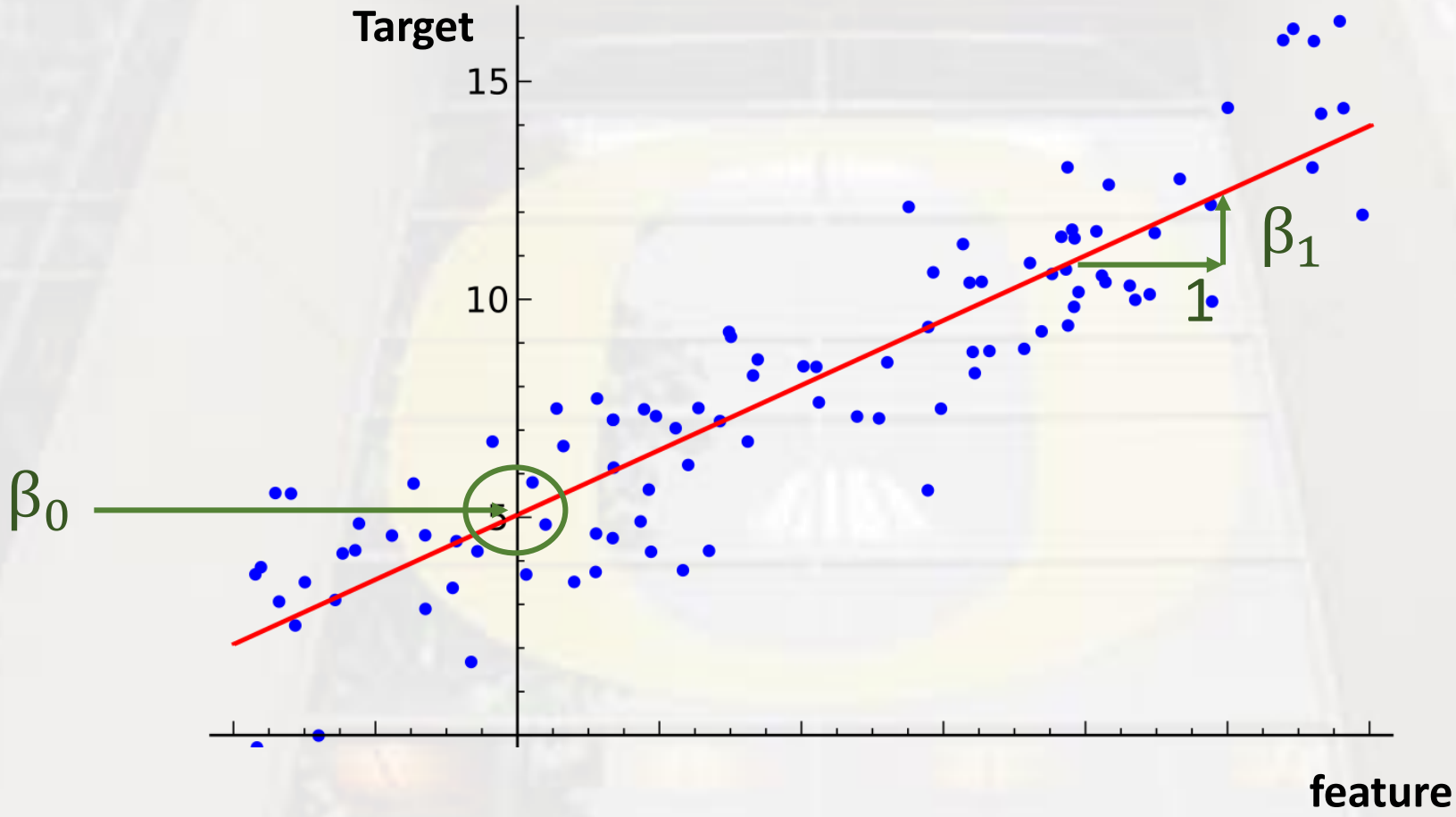
Linear Regression

- Basic model in Supervised Learning
- Useful tool for predicting a quantitative target
- Been around for a long time
- Many advanced models are extensions of linear regression

Linear Regression

- Simple linear regression
  - Predict a target variable “y” using one feature (predictor variable) “X”
  - For instance, predicting the price of a house using its size
  - $y \approx \beta_0 + \beta_1 x_1$
  - Our goal is to estimate β_0 and β_1

Linear Regression



Linear Regression

- Multiple linear regression
  - Predict a target variable “y” using a linear combination of one or more features (predictor variables) “X”
  - $y \approx \beta_0 + \beta_1 x_1 + \beta_2 x_2 + \cdots + \beta_p x_p$
  - Our goal is to estimate $\beta_0, \beta_1, \beta_2 \cdots \beta_p$
  - Intercept & slope coefficients : parameters of the fitted model
    - Intercept: β_0
    - Slope coefficients: $\beta_1, \beta_2 \cdots \beta_p$

Example : Real estate market

- Target: House List Price (y)
- Features (X)
  - Square Foot
  - Year Built
  - Beds, Bath
  - Lot Size
  - Parking Spots
  - Garage (0/1)
  - Zip,
  - Public School Rating
- $\text{List Price} \approx \beta_0 + \beta_1\ \text{SQFT} + \beta_2\ \text{Age} + \cdots + \beta_p\ \text{School Rating}$

Example : Used Car Sales

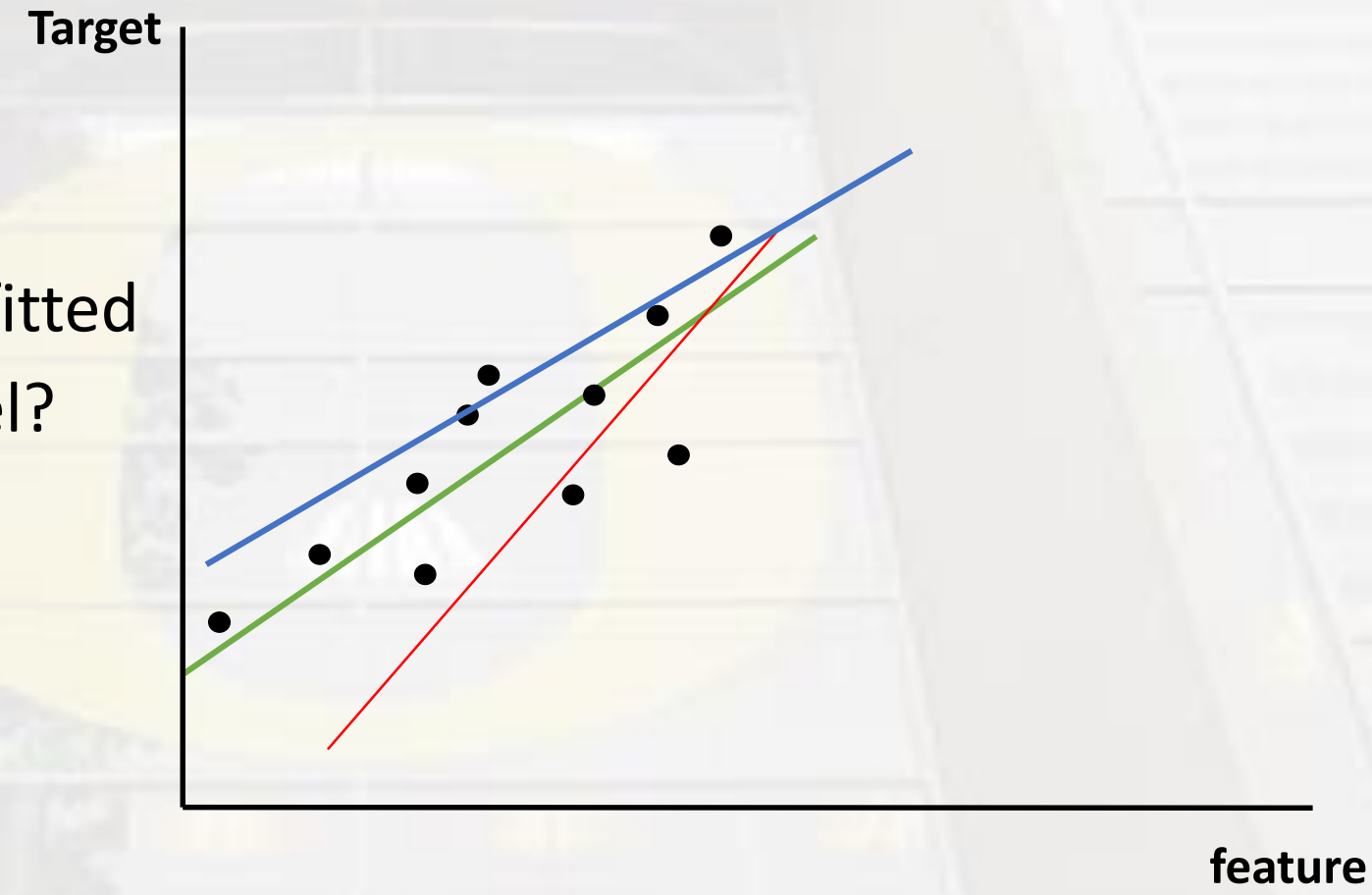
- Target: Sale Price (y)
- Features (X)
  - Age in months
  - Mileage
  - Fuel type (Petrol, Diesel, CNG)
  - Horsepower
  - Metallic color? (Yes = 1, No = 0)
  - Automatic (Yes = 1, No = 0)
  - Number of doors
- $\text{Sale Price} \approx \beta_0 + \beta_1 \text{Age} + \beta_2 \text{Miles} + \cdots + \beta_p \text{Doors}$

Example : New Route Air fare

- Target: Fare (y)
- Features (X)
  - Start and End City
  - New Air carriers entering the route
  - Market concentration
  - Start and End City Average Income
  - Start and End City Average Population
  - Distance
- $\text{Fare} \approx \beta_0 + \beta_1 \text{ Start City} + \beta_2 \text{ End City} + \cdots + \beta_p \text{ Distance}$

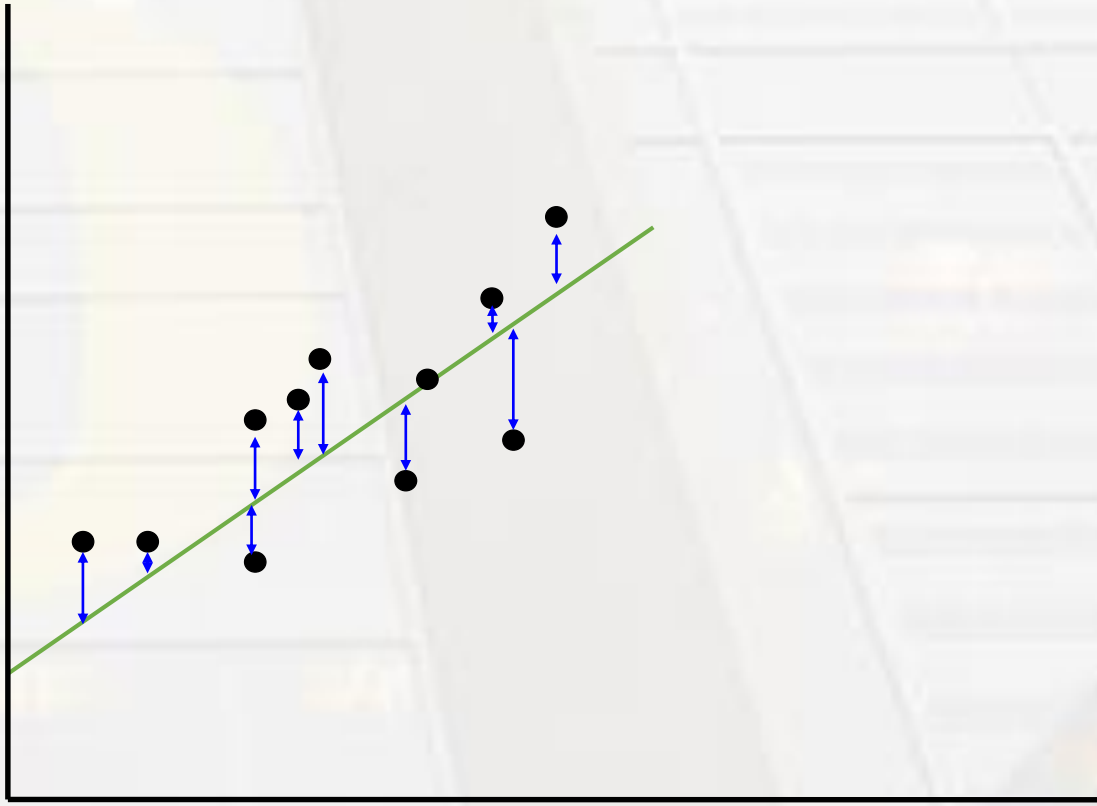
Linear Regression

- Which line is the best fitted linear regression model?



Linear Regression

- How we estimate the parameters (intercept and coefficients)?
- Linear regression finds the parameters that minimizes the mean squared errors between the predictions and the true target values on the training data
- $e_i = y_i - \hat{y}_i$
- MSE= $\frac{1}{n}\sum_{i=1}^{n} e_i^2$



Linear Regression

- Linear regression in Pyhton
  - Import the algorithm
  - Train (or fit) the algorithm using training data by applying *fit* method
    - Model parameters (intercept and coefficients) are estimated after model is trained
    - Intercept is stored in the *intercept_* attribute
    - Slope coefficients (also called weights) are stored in the *coef_* attribute
  - Evaluate the performance of the algorithm using *score* method
  - Predict new test datasets using the *predict* method

Linear Regression

```
3]:  1 house1=pd.read_csv('House_1feature.csv')
     2 house1.head()
```

3]:

	Square_feet	Price
0	1349	147703
1	897	178239
2	660	160805
3	854	146418
4	916	168025

```
4]:  1 X_house, y_house=house1[['Square_feet']], house1['Price']
     2 X_house.head()
```

4]:

	Square_feet
0	1349

Linear Regression

```
1  from sklearn.model_selection import train_test_split
2  X_train, X_test, y_train, y_test=train_test_split(X_house,y_house,random_state=0,)
```

```
1  print('Train features shape:',X_train.shape)
2  print('Test features shape:',X_test.shape)
3  print('Train target shape:',y_train.shape)
4  print('Test target shape:',y_test.shape)
```

```
Train features shape: (27, 1)
Test features shape: (9, 1)
Train target shape: (27,)
Test target shape: (9,)
```

Linear Regression

- Underscore at the end of coef_ and intercept_ attributes: Scikit-learn always stores anything that is derived from the training data in attributes that end with a trailing underscore. That is to separate them from parameters that are set by user.

```
: 1 #importing the algorithm
  2 from sklearn.linear_model import LinearRegression
```

```
: 1 lr=LinearRegression()
  2 lr.fit(X_train,y_train)
```

```
: LinearRegression(copy_X=True, fit_intercept=True, n_jobs=None,
         normalize=False)
```

```
: 1 lr.intercept_
```

```
: 107080.61733508084
```

```
: 1 lr.coef_
```

```
: array([70.8783014])
```

Linear Regression

```
]:
1 # Evaluating the performance
2 print('R-square on train data is:',lr.score(X_train,y_train))
3 print('R-square on test data is:',lr.score(X_test,y_test))
```

```
R-square on train data is: 0.7220076883072996
R-square on test data is: 0.819617028100509
```

```
]:
1 lr.predict([[5000],[6230],[9000]])
```

```
]: array([461472.12432632, 548652.43504617, 744985.32991932])
```

```
:
1 # Evaluating the performance
2 print('R-square on train data is: {:.1%}'.format(lr.score(X_train,y_train)))
3 print('R-square on test data is: {:.2f}'.format(lr.score(X_test,y_test)))
```

```
R-square on train data is: 72.2%
R-square on test data is: 0.82
```

Fixing the format of the output

Linear Regression

- Try it with real dataset, Boston Housing

```
In [124]: house_data=pd.read_csv('boston_housing_data.csv', index_col=0)

In [132]: house_data.head(n=2)
```

Out[132]:

ID	crim	zn	indus	chas	nox	rm	age	dis	rad	tax	ptratio	lstat	target_medv
1	0.00632	18.0	2.31	0	0.538	6.575	65.2	4.0900	1	296	15.3	4.98	24.0
2	0.02731	0.0	7.07	0	0.469	6.421	78.9	4.9671	2	242	17.8	9.14	21.6

```
In [129]: X_house, y_house=house_data.iloc[:,:-1],house_data['target_medv']
```

Linear Regression

```
]: 1 X_house, y_house=boston.iloc[:,:-1],boston['target_medv']
   2 X_house.head(n=2)
```

]:

ID	crim	zn	indus	chas	nox	rm	age	dis	rad	tax	ptratio	lstat
1	0.00632	18.0	2.31	0	0.538	6.575	65.2	4.0900	1	296	15.3	4.98
2	0.02731	0.0	7.07	0	0.469	6.421	78.9	4.9671	2	242	17.8	9.14

```
]: 1 X_train, X_test, y_train, y_test=train_test_split(X_house,y_house,random_state=0,)
```

```
]: 1 print('Train features shape:',X_train.shape)
   2 print('Test features shape:',X_test.shape)
   3 print('Train target shape:',y_train.shape)
   4 print('Test target shape:',y_test.shape)
```

```
Train features shape: (249, 12)
Test features shape: (84, 12)
Train target shape: (249,)
Test target shape: (84,)
```

Linear Regression

```
In [ ]:
1 lr2=LinearRegression()
2 lr2.fit(X_train,y_train)
```

```
Out[ ]:
LinearRegression(copy_X=True, fit_intercept=True, n_jobs=None,
         normalize=False)
```

```
In [ ]:
1 lr2.intercept_
```

```
Out[ ]:
41.72463933066167
```

```
In [ ]:
1 lr2.coef_
```

```
Out[ ]:
array([-1.83853404e-01,  3.40723088e-02,  9.16637921e-02,  4.36098629e+00,
       -2.06471577e+01,  3.36616016e+00,  7.04582792e-03, -1.42586713e+00,
        3.41928111e-01, -1.23677528e-02, -7.83100065e-01, -6.93085300e-01])
```

```
In [ ]:
1 X_house.columns
```

```
Out[ ]:
Index(['crim', 'zn', 'indus', 'chas', 'nox', 'rm', 'age', 'dis', 'rad', 'tax',
       'ptratio', 'lstat'],
      dtype='object')
```

```
In [ ]:
1 pd.DataFrame(data=lr2.coef_,
2              index=X_house.columns,
3              columns=['coefficients'])
```

	coefficients
crim	-0.183853
zn	0.034072
indus	0.091664
chas	4.360986
nox	-20.647158
rm	3.366160

Linear Regression

```
4]:  1  # Evaluating the performance
     2  print('R-square on train data is: {:.1%}'.format(lr2.score(X_train,y_train)))
     3  print('R-square on test data is: {:.2f}'.format(lr2.score(X_test,y_test)))
```

```
R-square on train data is: 73.0%
R-square on test data is: 0.67
```

Overfitting in Linear Regression

- In datasets with many features (many predictors), there is a high chance for linear regression to overfit
- A linear regression model with too many features is a complex model
- A relatively significant difference between our model's performance on training and test sets is a sign of overfitting
- Upload another version of Boston housing data that has more features (variables): *boston_housing_data_v2.csv*

Overfitting in Linear Regression

```
13]: boston2=pd.read_csv('boston_housing_data_v2.csv')
     print('boston 2 shape is:',boston2.shape)
     X=boston2.iloc[:,1:]
     y=boston2['target_medv']
     X_train, X_test, y_train, y_test=train_test_split(X,y,random_state=0)
```

```
boston 2 shape is: (333, 62)
```

```
11]: lr.fit(X_train,y_train)
     print('Training set score: {:.3f}'.format(lr.score(X_train,y_train)))
     print('Test set score: {:.3f}'.format(lr.score(X_test,y_test)))
```

```
Training set score: 0.789
Test set score: 0.607
```

- We should try to find a model that allows us to control complexity

More on Regression

Ridge Regression

- $e_i = y_i - \hat{y}_i$
- Regular regression (ordinary least square) minimizes MSE to estimate the coefficients
  - $MSE = \frac{1}{n}\sum_{i=1}^{n} e_i^2$
- Ridge regression minimizes the following to estimate the coefficients
  - $MSE + \alpha \sum_{j=0}^{p} \beta_j^2$
- Adding this additional constraint is an example of *regularization*
- The regularization used by ridge regression is known as L2 regularization

Ridge Regression in scikit-learn

```
|: from sklearn.linear_model import Ridge
   r_reg=Ridge()
   r_reg.fit(X_train,y_train)
   print('Training set score: {:.3f}'.format(r_reg.score(X_train,y_train)))
   print('Test set score: {:.3f}'.format(r_reg.score(X_test,y_test)))
```

```
Training set score: 0.752
Test set score: 0.685
```

- Training score is lower and test score is higher, which is consistent with our expectation
- With linear regression we were over-fitting our data
- Ridge is a more restricted model, so we are less likely to overfit

Ridge Regression in scikit-learn

- The ridge model makes a trade-off between the simplicity of the model (near-zero coefficients) and its performance on the training set
- This can be controlled by user, using the ***alpha*** parameter
- By default alpha=1
- Increasing the alpha forces coefficients to move more toward zero, which decrease the training performance and might help generalization (after some point it might even decrease the test score)
- The best value for alpha depends on data we have

Ridge Regression in scikit-learn

```
]: # smaller alpha, more complex model, higher chance of over-fitting
   ridge1=Ridge(alpha=.01)

   # something in between
   ridge2=Ridge(alpha=1)

   # greater alpha, less complex model (maybe too simple),
   # lower chance of over-fitting (might do under-fitting)
   ridge3=Ridge(alpha=10)
```

```
]: ridge1.fit(X_train, y_train)
   ridge2.fit(X_train, y_train)
   ridge3.fit(X_train, y_train)
```

```
]: Ridge(alpha=10, copy_X=True, fit_intercept=True, max_iter=None, normalize=False,
         random_state=None, solver='auto', tol=0.001)
```

Ridge Regression in scikit-learn

```
]: # evaluate
print('ridge1 on train: {:.2%}'.format(ridge1.score(X_train, y_train)))
print('ridge1 on test: {:.2%}'.format(ridge1.score(X_test, y_test)),'\n')

print('ridge2 on train: {:.2%}'.format(ridge2.score(X_train, y_train)))
print('ridge2 on test: {:.2%}'.format(ridge2.score(X_test, y_test)),'\n')

print('ridge3 on train: {:.2%}'.format(ridge3.score(X_train, y_train)))
print('ridge3 on test: {:.2%}'.format(ridge3.score(X_test, y_test)))
```

```
ridge1 on train: 78.44%
ridge1 on test: 66.67%

ridge2 on train: 75.19%
ridge2 on test: 68.49%

ridge3 on train: 73.62%
ridge3 on test: 68.13%
```

Lasso

- Lasso minimizes the following to estimate the coefficients
  - $MSE + \alpha \sum_{j=0}^{p} |\beta_j|$
- This means some features will be entirely ignored by the model
- This can be seen as a form of automatic feature selection
- It can reveal the most important features (predictors) for the model
- Lasso's models are simpler and easier to interpret
- Lasso's regularization is called L1 regularization

Lasso in scikit-learn

```
from sklearn.linear_model import Lasso
l_reg=Lasso()
l_reg.fit(X_train,y_train)
print('Training set score: {:.3f}'.format(l_reg.score(X_train,y_train)))
print('Test set score: {:.3f}'.format(l_reg.score(X_test,y_test)))
```

```
Training set score: 0.661
Test set score: 0.604
```

- As we can see, lasso did quite badly, both on train and the test
- This indicate that we are underfitting
- Let’s check out how many features we have used in this model

Lasso in scikit-learn

- We might also need to increase the **max iterations**

```
# smaller alpha -> more complex model (less simple) ->
# higher chance of over-fitting(lower chance of under-fitting)
lasso1=Lasso(alpha=.0001, max_iter=100000)
lasso2=Lasso(alpha=.001, max_iter=100000)
lasso3=Lasso(alpha=.1, max_iter=100000)
# greater alpha -> less complex (maybe too simple) ->
# lower chance of over-fitting (HIGHER chance of under-fitting)
lasso4=Lasso(alpha=1,max_iter=100000)
```

```
lasso1.fit(X_train, y_train)
lasso2.fit(X_train, y_train)
lasso3.fit(X_train, y_train)
lasso4.fit(X_train, y_train)
```

```
Lasso(alpha=1, copy_X=True, fit_intercept=True, max_iter=100000,
      normalize=False, positive=False, precompute=False, random_state=None,
      selection='cyclic', tol=0.0001, warm_start=False)
```

Lasso in scikit-learn

UNIVERSITY OF OREGON
Lundquist College of Business

```
]: print('lasso 1 on train: {:.2%}'.format(lasso1.score(X_train, y_train)))
   print('lasso 1 on test: {:.2%}'.format(lasso1.score(X_test, y_test)))
   print('non-zeero coef for lasso1: ', sum(lasso1.coef_!=0),'\n')

   print('lasso 2 on train: {:.2%}'.format(lasso2.score(X_train, y_train)))
   print('lasso 2 on test: {:.2%}'.format(lasso2.score(X_test, y_test)))
   print('non-zeero coef for lasso2: ', sum(lasso2.coef_!=0),'\n')

   print('lasso 3 on train: {:.2%}'.format(lasso3.score(X_train, y_train)))
   print('lasso 3 on test: {:.2%}'.format(lasso3.score(X_test, y_test)))
   print('non-zeero coef for lasso3: ', sum(lasso3.coef_!=0),'\n')

   print('lasso 4 on train: {:.2%}'.format(lasso4.score(X_train, y_train)))
   print('lasso 4 on test: {:.2%}'.format(lasso4.score(X_test, y_test)))
   print('non-zeero coef for lasso4: ', sum(lasso4.coef_!=0))
```

```
lasso 1 on train: 78.93%
lasso 1 on test: 62.31%
non-zeero coef for lasso1:  57

lasso 2 on train: 78.27%
lasso 2 on test: 68.07%
non-zeero coef for lasso2:  49

lasso 3 on train: 72.48%
lasso 3 on test: 66.94%
non-zeero coef for lasso3:  14

lasso 4 on train: 66.06%
lasso 4 on test: 60.40%
non-zeero coef for lasso4:  9
```