Section 10 Introduction to Machine Learning and Linear Regression

Motivating Example I: Single-variable (1D) Linear Regression

Problem

Given the *training dataset* $\left(x^{(i)} \in \mathbb{R}, y^{(i)} \in \mathbb{R}\right)$, $i = 1, 2, \ldots, N$ (in other words the x-values and y-values are real numbers), we want to find the linear function

$$y \approx f(x) = wx + b$$

that fits the relations between $x^{(i)}$ and $y^{(i)}$. So that given any new x^{test} in the **test** dataset, we can make the prediction

$$y^{pred} = wx^{test} + b$$

To compare to section 9's notation:

$$\theta = (w, b)$$

$$y \approx f(x; \theta) = f(x; w, b) = wx + b$$

In the above line, the semicolon separates variables into two categories: data variable x to the left, and function parameters θ (or w, b) on the right. Our model depends on the parameters, but they won't change after we find the line of best fit.

For the rest of section 10, we will be dropping the mention of θ and it's associated variables in our notation for f.

Training the model

- With the training dataset, define the loss function $L(w, b)$ of parameter w and b, which is also called **mean squared error** (MSE)

  $$\text{MSE} = L(w, b) = \frac{1}{N} \sum_{i=1}^{N} \left(\hat{y}^{(i)} - y^{(i)}\right)^2 = \frac{1}{N} \sum_{i=1}^{N} \left((wx^{(i)} + b) - y^{(i)}\right)^2,$$

  where $\hat{y}^{(i)}$ denotes the predicted value of y at $x^{(i)}$, i.e. $\hat{y}^{(i)} = wx^{(i)} + b$.

- Then find the minimum of the loss function. Note $L(w, b)$ is a quadratic function of w and b, and we can analytically solve $\partial_w L = \partial_b L = 0$ (exactly the same as finding the minimum in multivariable calculus -- Math 2D), and yields

$$w^* = \frac{\sum_{i=1}^N (x^{(i)} - \bar{x})(y^{(i)} - \bar{y})}{\sum_{i=1}^N (x^{(i)} - \bar{x})^2} = \frac{\frac{1}{N}\sum_{i=1}^N (x^{(i)} - \bar{x})(y^{(i)} - \bar{y})}{\frac{1}{N}\sum_{i=1}^N (x^{(i)} - \bar{x})^2} = \frac{\mathrm{Cov}(X,Y)}{\mathrm{Var}(X)},$$

$$b^* = \bar{y} - w^* \bar{x},$$

where $\bar{x}$ and $\bar{y}$ are the mean of x and of y, and $\mathrm{Cov}(X,Y)$ denotes the estimated covariance (or called sample covariance) between X and Y (a little difference with what you learn in statistics is that we have the normalization factor $1/N$ instead of $1/(N-1)$ here), and $\mathrm{Var}(Y)$ denotes the sample variance of Y (the normalization factor is still $1/N$). This is just about convention -- in statistics, they pursue the unbiased estimator.)

Evaluating the model

- MSE: The smaller MSE indicates better performance
- R-Squared: The larger R^2 (closer to 1) indicates better performance. Compared with MSE, R-squared is **dimensionless**, not dependent on the units of variable.

$$R^2 = \frac{\mathrm{Var}(Y) - \mathrm{MSE}}{\mathrm{Var}(Y)}$$

The above calculation is a theoretical explanation for R^2. The equation below is for simpler calculation.

$$R^2 = 1 - \frac{\sum_{i=1}^N (y^{(i)} - \hat{y}^{(i)})^2}{\sum_{i=1}^N (y^{(i)} - \bar{y})^2} = 1 - \frac{\frac{1}{N}\sum_{i=1}^N (y^{(i)} - \hat{y}^{(i)})^2}{\frac{1}{N}\sum_{i=1}^N (y^{(i)} - \bar{y})^2} = 1 - \frac{\mathrm{MSE}}{\mathrm{Var}(Y)}$$

In [2]:

```
import numpy as np

class MyLinearRegression1D:
    '''
    The single-variable linear regression estimator -- writing in the style of sklearn
    '''

    def fit(self, x, y):
        '''
        Determine the optimal parameters w, b for the input data x and y

        Parameters
        ----------
           x : 1D numpy array with shape (n_samples,) from training data
           y : 1D numpy array with shape (n_samples,) from training data

        Returns
        -------
        self : returns an instance of self, with new attributes slope w (float) and int
         '''

        cov_mat = np.cov(x,y,bias=True) # covariance matrix, bias = True makes the fact
        #array of form [[Var(x), Cov(x,y)],
        #               [Cov(x,y), Var(y)]]
        self.w = cov_mat[0,1] / cov_mat[0,0] # the (0,1) element is COV(X,Y) and (0,0)
        self.b =  np.mean(y)-self.w * np.mean(x)
```

```
    def predict(self,x):
        '''
        Predict the output values for the input value x, based on trained parameters

        Parameters
        ----------
           x : 1D numpy array from training or test data

        Returns
        -------
        returns 1D numpy array of same shape as input, the predicted y value of corresp
        '''

        return self.w*x+self.b

    def score(self, x, y):
        '''
        Calculate the R-squared on the dataset with input x and y

        Parameters
        ----------
           x : 1D numpy array with shape (n_samples,) from training or test data
           y : 1D numpy array with shape (n_samples,) from training or test data

        Returns
        -------
        returns float, the R^2 value
        '''

        y_hat = self.predict (x) # predicted y
        mse = np.mean((y-y_hat)**2) # mean squared error
        return 1- mse / np.var(y) # return R-squared
```

In [3]:

```
import pandas as pd
house = pd.read_csv('kc_house_data.csv')
house.sample(5)
```

Out[3]:

	id	date	price	bedrooms	bathrooms	sqft_living	sqft_lot	floors	wat
9246	8121101380	20140813T000000	475000.0	3	1.00	1380	4635	1.0	
12707	8019200845	20150218T000000	245000.0	2	1.00	1020	15000	1.5	
6331	5416500660	20150430T000000	426500.0	4	2.50	2960	4640	2.0	
27	3303700376	20141201T000000	667000.0	3	1.00	1400	1581	1.5	
11347	4315700505	20150210T000000	535000.0	4	1.75	1570	3250	1.5	

5 rows × 21 columns

In [4]:

```
house.drop(['id','date','zipcode','lat','long','yr_built','yr_renovated'],axis = 1, inp
house
```

Out[4]:

	price	bedrooms	bathrooms	sqft_living	sqft_lot	floors	waterfront	view	condition	grade
0	221900.0	3	1.00	1180	5650	1.0	0	0	3	7
1	538000.0	3	2.25	2570	7242	2.0	0	0	3	7
2	180000.0	2	1.00	770	10000	1.0	0	0	3	6
3	604000.0	4	3.00	1960	5000	1.0	0	0	5	7
4	510000.0	3	2.00	1680	8080	1.0	0	0	3	8
...	...	...	...	...	...	...	...	...	...	...
21608	360000.0	3	2.50	1530	1131	3.0	0	0	3	8
21609	400000.0	4	2.50	2310	5813	2.0	0	0	3	8
21610	402101.0	2	0.75	1020	1350	2.0	0	0	3	7
21611	400000.0	3	2.50	1600	2388	2.0	0	0	3	8
21612	325000.0	2	0.75	1020	1076	2.0	0	0	3	7

21613 rows × 14 columns

(This cell is a bookmark to the assignment of X below)

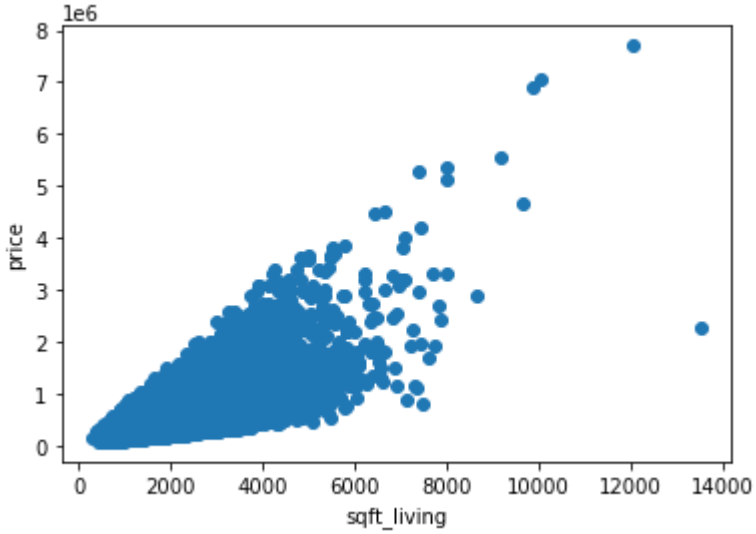
In [5]:
```
X = house.iloc[:,1:].to_numpy() #every column except price
y = house['price'].to_numpy()
```

In [6]:
```
X.shape
```

Out[6]:
```
(21613, 13)
```

In [7]:
```
import matplotlib.pyplot as plt
x = X[:,2]
plt.scatter(x,y)
plt.xlabel('sqft_living')
plt.ylabel('price')
```

Out[7]:
```
Text(0, 0.5, 'price')
```



In [9]:

```
lreg = MyLinearRegression1D() # initialize the instance of one estimator
help(lreg)
```

```
Help on MyLinearRegression1D in module __main__ object:

class MyLinearRegression1D(builtins.object)
 |  The single-variable linear regression estimator -- writing in the style of sklearn p
ackage
 |
 |  Methods defined here:
 |
 |  fit(self, x, y)
 |      Determine the optimal parameters w, b for the input data x and y
 |
 |      Parameters
 |      ----------
 |         x : 1D numpy array with shape (n_samples,) from training data
 |         y : 1D numpy array with shape (n_samples,) from training data
 |
 |      Returns
 |      -------
 |      self : returns an instance of self, with new attributes slope w (float) and inte
rcept b (float)
 |
 |  predict(self, x)
 |      Predict the output values for the input value x, based on trained parameters
 |
 |      Parameters
 |      ----------
 |         x : 1D numpy array from training or test data
 |
 |      Returns
 |      -------
 |      returns 1D numpy array of same shape as input, the predicted y value of correspo
nding x
 |
 |  score(self, x, y)
 |      Calculate the R-squared on the dataset with input x and y
 |
 |      Parameters
 |      ----------
 |         x : 1D numpy array with shape (n_samples,) from training or test data
 |         y : 1D numpy array with shape (n_samples,) from training or test data
 |
```

```
|      Returns
|      -------
|      returns float, the R^2 value
|
|  ----------------------------------------------------------------------
|  Data descriptors defined here:
|
|  __dict__
|      dictionary for instance variables (if defined)
|
|  __weakref__
|      list of weak references to the object (if defined)
```

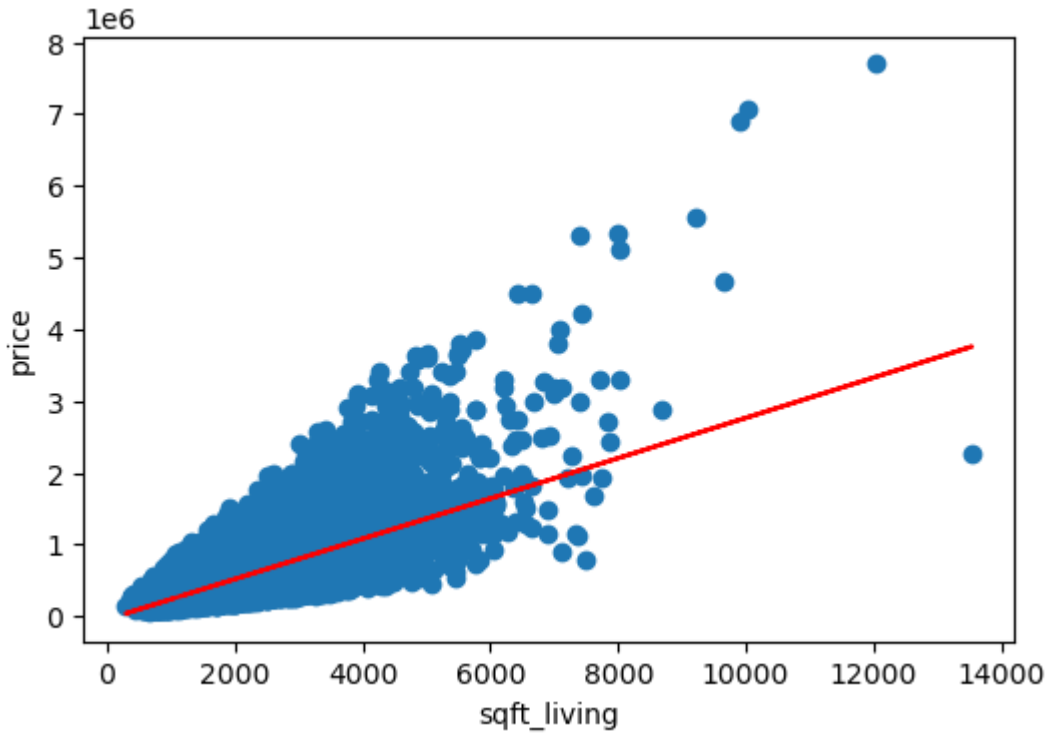
In [10]:
```
lreg.fit(x,y)
```

In [11]:
```
lreg.score(x,y)
```

Out[11]:
```
0.49286538652201417
```

In [12]:
```
fig = plt.figure(dpi = 100)
plt.scatter(x,y)
plt.xlabel('sqft_living')
plt.ylabel('price')
plt.plot(x,lreg.predict(x),'r')
```

Out[12]:
```
[<matplotlib.lines.Line2D at 0x2bd1132e880>]
```



In [13]:
```
from sklearn import linear_model # compare with the scikit learn package
lreg_sklearn = linear_model.LinearRegression()
lreg_sklearn.fit(x.reshape(-1,1),y) #only accept 2D-array as x
```

Out[13]:
```
LinearRegression()
```

In [14]:
```
print(lreg.w,lreg.b) #coeffs from our method
print(lreg_sklearn.coef_, lreg_sklearn.intercept_) #coeffs from scikit learn
```

```
280.8066899295006 -43867.60153385543
[280.80668993] -43867.601533854846
```

In [15]:
```
lreg_sklearn.score(x.reshape(-1,1),y) #same R^2 regression score up to 15 significant d
```

Out[15]:
```
0.4928653865220143
```

In [20]:
```
help(lreg_sklearn)
```

```
Help on LinearRegression in module sklearn.linear_model._base object:

class LinearRegression(sklearn.base.MultiOutputMixin, sklearn.base.RegressorMixin, Linea
rModel)
 |  LinearRegression(*, fit_intercept=True, normalize=False, copy_X=True, n_jobs=None)
 |
 |  Ordinary least squares Linear Regression.
 |
 |  LinearRegression fits a linear model with coefficients w = (w1, ..., wp)
 |  to minimize the residual sum of squares between the observed targets in
 |  the dataset, and the targets predicted by the linear approximation.
 |
 |  Parameters
 |  ----------
 |  fit_intercept : bool, default=True
 |      Whether to calculate the intercept for this model. If set
 |      to False, no intercept will be used in calculations
 |      (i.e. data is expected to be centered).
 |
 |  normalize : bool, default=False
 |      This parameter is ignored when ``fit_intercept`` is set to False.
 |      If True, the regressors X will be normalized before regression by
 |      subtracting the mean and dividing by the l2-norm.
 |      If you wish to standardize, please use
 |      :class:`sklearn.preprocessing.StandardScaler` before calling ``fit`` on
 |      an estimator with ``normalize=False``.
 |
 |  copy_X : bool, default=True
 |      If True, X will be copied; else, it may be overwritten.
 |
 |  n_jobs : int, default=None
 |      The number of jobs to use for the computation. This will only provide
 |      speedup for n_targets > 1 and sufficient large problems.
 |      ``None`` means 1 unless in a :obj:`joblib.parallel_backend` context.
 |      ``-1`` means using all processors. See :term:`Glossary <n_jobs>`
 |      for more details.
 |
 |  Attributes
 |  ----------
 |  coef_ : array of shape (n_features, ) or (n_targets, n_features)
 |      Estimated coefficients for the linear regression problem.
 |      If multiple targets are passed during the fit (y 2D), this
 |      is a 2D array of shape (n_targets, n_features), while if only
 |      one target is passed, this is a 1D array of length n_features.
 |
 |  rank_ : int
 |      Rank of matrix `X`. Only available when `X` is dense.
```

```
|
|  singular_ : array of shape (min(X, y),)
|      Singular values of `X`. Only available when `X` is dense.
|
|  intercept_ : float or array of shape (n_targets,)
|      Independent term in the linear model. Set to 0.0 if
|      `fit_intercept = False`.
|
|  See Also
|  --------
|  sklearn.linear_model.Ridge : Ridge regression addresses some of the
|      problems of Ordinary Least Squares by imposing a penalty on the
|      size of the coefficients with l2 regularization.
|  sklearn.linear_model.Lasso : The Lasso is a linear model that estimates
|      sparse coefficients with l1 regularization.
|  sklearn.linear_model.ElasticNet : Elastic-Net is a linear regression
|      model trained with both l1 and l2 -norm regularization of the
|      coefficients.
|
|  Notes
|  -----
|  From the implementation point of view, this is just plain Ordinary
|  Least Squares (scipy.linalg.lstsq) wrapped as a predictor object.
|
|  Examples
|  --------
|  >>> import numpy as np
|  >>> from sklearn.linear_model import LinearRegression
|  >>> X = np.array([[1, 1], [1, 2], [2, 2], [2, 3]])
|  >>> # y = 1 * x_0 + 2 * x_1 + 3
|  >>> y = np.dot(X, np.array([1, 2])) + 3
|  >>> reg = LinearRegression().fit(X, y)
|  >>> reg.score(X, y)
|  1.0
|  >>> reg.coef_
|  array([1., 2.])
|  >>> reg.intercept_
|  3.0000...
|  >>> reg.predict(np.array([[3, 5]]))
|  array([16.])
|
|  Method resolution order:
|      LinearRegression
|      sklearn.base.MultiOutputMixin
|      sklearn.base.RegressorMixin
|      LinearModel
|      sklearn.base.BaseEstimator
|      builtins.object
|
|  Methods defined here:
|
|  __init__(self, *, fit_intercept=True, normalize=False, copy_X=True, n_jobs=None)
|      Initialize self.  See help(type(self)) for accurate signature.
|
|  fit(self, X, y, sample_weight=None)
|      Fit linear model.
|
|      Parameters
|      ----------
|      X : {array-like, sparse matrix} of shape (n_samples, n_features)
|          Training data
|
|      y : array-like of shape (n_samples,) or (n_samples, n_targets)
|          Target values. Will be cast to X's dtype if necessary
|
```

```
|      sample_weight : array-like of shape (n_samples,), default=None
|          Individual weights for each sample
|
|          .. versionadded:: 0.17
|             parameter *sample_weight* support to LinearRegression.
|
|      Returns
|      -------
|      self : returns an instance of self.
|
|  ----------------------------------------------------------------------
|  Data and other attributes defined here:
|
|  __abstractmethods__ = frozenset()
|
|  ----------------------------------------------------------------------
|  Data descriptors inherited from sklearn.base.MultiOutputMixin:
|
|  __dict__
|      dictionary for instance variables (if defined)
|
|  __weakref__
|      list of weak references to the object (if defined)
|
|  ----------------------------------------------------------------------
|  Methods inherited from sklearn.base.RegressorMixin:
|
|  score(self, X, y, sample_weight=None)
|      Return the coefficient of determination R^2 of the prediction.
|
|      The coefficient R^2 is defined as (1 - u/v), where u is the residual
|      sum of squares ((y_true - y_pred) ** 2).sum() and v is the total
|      sum of squares ((y_true - y_true.mean()) ** 2).sum().
|      The best possible score is 1.0 and it can be negative (because the
|      model can be arbitrarily worse). A constant model that always
|      predicts the expected value of y, disregarding the input features,
|      would get a R^2 score of 0.0.
|
|      Parameters
|      ----------
|      X : array-like of shape (n_samples, n_features)
|          Test samples. For some estimators this may be a
|          precomputed kernel matrix or a list of generic objects instead,
|          shape = (n_samples, n_samples_fitted),
|          where n_samples_fitted is the number of
|          samples used in the fitting for the estimator.
|
|      y : array-like of shape (n_samples,) or (n_samples, n_outputs)
|          True values for X.
|
|      sample_weight : array-like of shape (n_samples,), default=None
|          Sample weights.
|
|      Returns
|      -------
|      score : float
|          R^2 of self.predict(X) wrt. y.
|
|      Notes
|      -----
|      The R2 score used when calling ``score`` on a regressor uses
|      ``multioutput='uniform_average'`` from version 0.23 to keep consistent
|      with default value of :func:`~sklearn.metrics.r2_score`.
|      This influences the ``score`` method of all the multioutput
|      regressors (except for
```

```
|      :class:`~sklearn.multioutput.MultiOutputRegressor`).
|
|  ----------------------------------------------------------------------
|  Methods inherited from LinearModel:
|
|  predict(self, X)
|      Predict using the linear model.
|
|      Parameters
|      ----------
|      X : array_like or sparse matrix, shape (n_samples, n_features)
|          Samples.
|
|      Returns
|      -------
|      C : array, shape (n_samples,)
|          Returns predicted values.
|
|  ----------------------------------------------------------------------
|  Methods inherited from sklearn.base.BaseEstimator:
|
|  __getstate__(self)
|
|  __repr__(self, N_CHAR_MAX=700)
|      Return repr(self).
|
|  __setstate__(self, state)
|
|  get_params(self, deep=True)
|      Get parameters for this estimator.
|
|      Parameters
|      ----------
|      deep : bool, default=True
|          If True, will return the parameters for this estimator and
|          contained subobjects that are estimators.
|
|      Returns
|      -------
|      params : mapping of string to any
|          Parameter names mapped to their values.
|
|  set_params(self, **params)
|      Set the parameters of this estimator.
|
|      The method works on simple estimators as well as on nested objects
|      (such as pipelines). The latter have parameters of the form
|      ``<component>__<parameter>`` so that it's possible to update each
|      component of a nested object.
|
|      Parameters
|      ----------
|      **params : dict
|          Estimator parameters.
|
|      Returns
|      -------
|      self : object
|          Estimator instance.
```

Motivating Example II: Multi-variable Linear Regression (OLS -- Ordinary Least Square)

Problem

Given the *training dataset* $\left(x^{(i)}, y^{(i)}\right), i = 1, 2, \ldots, N$, this time with $y^{(i)} \in \mathbb{R}$ and $x^{(i)} \in \mathbb{R}^p$, we fit the multi-variable linear function

$$y \approx \mathbf{f}(x) = \beta_0 + \beta_1 x_1 + .. + \beta_p x_p = \tilde{x}\beta,$$

$$\tilde{x} = (1, x_1, .., x_p) \in \mathbb{R}^{1\times(p+1)}, \beta = (\beta_0, \beta_1, .., \beta_p)^T \in \mathbb{R}^{(p+1)\times 1}.$$

We call β the regression coefficients, and β_0 specially refers to the intercept.

When $p = 2$, we are finding a plane of best fit. When $p > 2$, we are finding a hyperplane of best fit.

Note that the dot product of $\tilde{x}\beta = f(x)$ (try this out).

Using the whole training dataset, we can write as

$$Y = \begin{pmatrix} y^{(1)} \\ y^{(2)} \\ \cdots \\ y^{(N)} \end{pmatrix} \approx \begin{pmatrix} \mathbf{f}(x^{(1)}) \\ \mathbf{f}(x^{(2)}) \\ \cdots \\ \mathbf{f}(x^{(N)}) \end{pmatrix} = \begin{pmatrix} \tilde{x}^{(1)}\beta \\ \tilde{x}^{(2)}\beta \\ \cdots \\ \tilde{x}^{(N)}\beta \end{pmatrix} = \begin{pmatrix} \tilde{x}^{(1)} \\ \tilde{x}^{(2)} \\ \cdots \\ \tilde{x}^{(N)} \end{pmatrix} \beta = \tilde{X}\beta,$$

where

$$\tilde{X} = \begin{pmatrix} \tilde{x}^{(1)} \\ \tilde{x}^{(2)} \\ \cdots \\ \tilde{x}^{(N)} \end{pmatrix} = \begin{pmatrix} 1 & x_1^{(1)} & \cdots & x_p^{(1)} \\ 1 & x_1^{(2)} & \cdots & x_p^{(2)} \\ \cdots & & & \\ 1 & x_1^{(N)} & \cdots & x_p^{(N)} \end{pmatrix}$$

is also called the augmented data matrix.

- **Question**: To get unknown β, can we directly solve the linear equation $\tilde{X}\beta = Y$?
- **Answer**: Most times no, because:
    1. typically there are more equations than variables ($N >> (p+1)$)
    2. the linear model is merely the approximation to the real mapping
    3. there are noises in the data points -- it's highly possible that there is NO solution at all!
- **Strategy**: Instead of solving $\tilde{X}\beta = Y$ exactly, we want β such that $\tilde{X}\beta$ is as close as Y as possible.

Training the model

- With the training dataset, define the loss function $L(\beta)$ of parameters β, which is also called **mean squared error** (MSE)

$$\text{MSE} = L(\beta) = \frac{1}{N}\sum_{i=1}^{N}\left(\hat{y}^{(i)} - y^{(i)}\right)^2 = \frac{1}{N}\sum_{i=1}^{N}\left(\tilde{x}^{(i)}\beta - y^{(i)}\right)^2,$$

where $\hat{y}^{(i)}$ denotes the predicted value of y at $x^{(i)}$, i.e.

$$\hat{y}^{(i)} = \beta_0 + \beta_1 x_1^{(i)} + .. + \beta_p x_p^{(i)} = \tilde{x}^{(i)}\beta.$$

Now the problem becomes

$$\min_{\beta} L(\beta),$$

i.e. find the minimizer of a multi-variable (p+1 dimensions) function.

- Then find the minimum of loss function -- There are two ways, either by numerical optimization (will be introduced in discussion) or by solving linear systems (introduced below), which is also called the **normal equation** approach.

To solve the critical points, we have $\nabla L(\beta) = 0$ (same as in multivariable calculus).

$$\frac{\partial L}{\partial \beta_0} = 2\sum_{i=1}^{N}(\tilde{x}^{(i)}\beta - y^{(i)}) = 0,$$
$$\frac{\partial L}{\partial \beta_k} = 2\sum_{i=1}^{N} x_k^{(i)}(\tilde{x}^{(i)}\beta - y^{(i)}) = 0, \quad k = 1, 2, .., p.$$

In Matrix form, it can be expressed as (left as exercise)

$$\tilde{X}^T\tilde{X}\beta = \tilde{X}^T Y,$$

also called the **normal equation** of linear regression. The optimal parameter $\hat{\beta} = \text{argmin} L(\beta)$ is also called the ordinary least square (**OLS**) estimator in statistics community.

Then the OLS estimator can be solved as

$$\hat{\beta} = (\tilde{X}^T\tilde{X})^{-1}\tilde{X}^T Y.$$

Geometrical Interpretation

Denote $\tilde{X} = (\tilde{X}_0, \tilde{X}_1, .., \tilde{X}_p)$, then $\tilde{X}\beta = \sum_{k=0}^{p}\beta_k\tilde{X}_k$. We require that the residual $Y - \tilde{X}\beta$ is orthogonal (i.e. normal, or vertical) to the plane spanned by $\tilde{X}_k$, which yields

$$\tilde{X}_k^T(Y - \tilde{X}\beta) = 0, \quad k = 0, 1, \ldots, p$$

Exercise: Check that when $p = 1$, the solution is equivalent to the single-variable regression.

Prediction in Test Data

Given the new observation called $x^{(test)}$, we have the prediction as

$$\hat{y}^{(test)} = \hat{\beta}_0 + \hat{\beta}_1 x_1^{(test)} + .. + \hat{\beta}_p x_p^{(test)} = \tilde{x}^{(test)}\hat{\beta}.$$

Evaluating the model

- MSE: The smaller MSE indicates better performance
- R-Squared: The larger R^2 (closer to 1) indicates better performance. Compared with MSE, R-squared is **dimensionless**, not dependent on the units of variable.

$$R^2 = 1 - \frac{\sum_{i=1}^{N}(y^{(i)} - \hat{y}^{(i)})^2}{\sum_{i=1}^{N}(y^{(i)} - \bar{y})^2} = 1 - \frac{\frac{1}{N}\sum_{i=1}^{N}(y^{(i)} - \hat{y}^{(i)})^2}{\frac{1}{N}\sum_{i=1}^{N}(y^{(i)} - \bar{y})^2} = 1 - \frac{\text{MSE}}{\text{Var}(Y)}$$

The coding of multi-variable linear regression left as the homework this week. Below we will call the function in sklearn directly.

Link to where X and y are defined in our notebook

In [16]:
```
from sklearn import linear_model # compare with the scikit learn package
lreg_sklearn = linear_model.LinearRegression()
lreg_sklearn.fit(X,y)
lreg_sklearn.score(X,y)
```

Out[16]:
```
0.6070919341230853
```

In [17]:
```
lreg_sklearn.coef_
```

Out[17]:
```
array([-3.46669591e+04, -1.48698203e+04,  1.34078969e+02,  2.67695148e-02,
       -3.00709892e+03,  5.85782331e+05,  5.98910372e+04,  5.38601479e+04,
        1.00892579e+05,  5.22508347e+01,  8.18281340e+01,  1.10527941e+01,
       -7.49796930e-01])
```

In [18]:
```
lreg_sklearn.intercept_
```

Out[18]:
```
-690582.9505202449
```

Motivating Example III: Single-variable Polynomial Regression (Special Case of Multivariable Linear Regression)

Problem

Given the *training dataset* $(x^{(i)}, y^{(i)}), i = 1, 2, \ldots, N$, this time with $y^{(i)} \in \mathbb{R}$ and $x^{(i)} \in \mathbb{R}$, we fit the single-variable polynomial function of p-th order

$$y \approx f(x) = w_0 + w_1 x + w_2 x^2 + \ldots + w_p x^p$$

Remark: A basic conclusion in numerical analysis is that with N points, we can have a polynomial of order (N-1) that fits every point perfectly.

Strategy

Single-variable **polynomial regression** is a special case of multi-variable **linear** regression, because we can construct a dataset of p variables by defining each row as $(1, x, x^2, \ldots, x^p)$ for each observation at x.

Machine Learning: Overview of the whole picture

Possible hierarchies of machine learning concepts:

- **Problems**: Supervised Learning (Regression,Classification), Unsupervised Learning (Dimension Reduction, Clustering), Reinforcement Learning (Not covered in this course)
- **Models**:
  - (Supervised) Linear Regression, Logistic Regression, K-Nearest Neighbor (kNN) Classification/Regression, Decision Tree, Random Forest, Support Vector Machine, Ensemble Method, Neural Network...
  - (Unsupervised) K-means, Hierachical Clustering, Principle Component Analysis, Manifold Learning (MDS, IsoMap, Diffusion Map, tSNE), Auto Encoder...
- **Algorithms**: Gradient Descent, Stochastic Gradient Descent (SGD), Back Propagation (BP),Expectation–Maximization (EM)...

For the same **problem**, there may exist multiple **models** to describe it. Given the specific **model**, there might be many different **algorithms** to solve it.

Why there is so much diversity? The following two fundamental principles of machine learning may provide theoretical insights.

Bias-Variance Trade-off: Simple models -- large bias, low variance. Complex models -- low bias, large variance. It's totally possible for a model with a worse fit on training data has a better fit on test data.

No Free Lunch Theorem: (in plain language) There is no one model that works best for every problem. (more quantitatively) Any two models are equivalent when their performance averaged across all possible problems. --Even true for optimization algorithms.

Extensions of OLS: MLE, Regularization, Ridge Regression and LASSO

Note: The detailed mathematical derivations below are optional material. What you need to know for quizzes/exams is:

1) what is the relation between MLE (most likelihood estimation) and the loss function in OLS regression (ordinary least-square) ;

2) the basic concepts of Ridge regression and LASSO ;

3) where does the additional regularization terms in the loss function of Ridge and LASSO come from ;

4) which model has the best performance on training/test dataset? (or is there any theoretical guarantee?)

Most Likelihood Estimation (MLE) and loss function in OLS

We already known what the loss function looks like in OLS. Here we first provide a mathematical explanation of this loss function from the perspective of Most Likelihood Estimation (MLE).

Recall that in linear regression, our **model assumption** is

$$y^{(i)} = \tilde{x}^{(i)}\beta + \epsilon^{(i)}, i = 1, 2, .., N$$

Now we further **assume** that residuals or errors $\epsilon^{(i)}$ are as independent Gaussian random variables with identical distribution $\mathcal{N}(0, \sigma^2)$ which has mean 0 and standard deviation σ.

From the density function of Gaussian distribution, the probability to observe $\epsilon^{(i)}$ within the small interval $[z, z + \Delta z]$ is roughly

$$\mathbb{P}(z < \epsilon^{(i)} < z + \Delta z) = \frac{1}{\sqrt{2\pi}\sigma}\exp\left(-\frac{z^2}{2\sigma^2}\right)\Delta z.$$

From the data, we know indeed $z = y^{(i)} - \tilde{x}^{(i)}\beta$. Therefore, given $x^{(i)}$ as fixed, the conditional probability density (likelihood) to observe $y^{(i)}$ is roughly

$$l(y^{(i)}|\beta; x^{(i)}) = \frac{1}{\sqrt{2\pi}\sigma}\exp\left(-\frac{(y^{(i)} - \tilde{x}^{(i)}\beta)^2}{2\sigma^2}\right).$$

Using the *independence* assumption, the overall likelihood to observe the response data $y^{(i)}$ $(i = 1, 2, \ldots, N)$ is

$$\mathcal{P}(y^{(i)}, 1 \leq i \leq N|\beta; x^{(i)}) = \prod_{i=1}^{N} l(y^{(i)}|\beta; x^{(i)})$$

The famous **Maximum Likelihood Estimation (MLE)** theory in statistics **assumes** that we aim to find the unknown parameter β that maximizes the conditional distribution $\mathcal{P}(\beta \,|\, x^{(i)}, y^{(i)}, 1 \leq i \leq N)$ by treating $x^{(i)}$ and $y^{(i)}$ as fixed numbers.

Equivalently (using the properties of logarithms), as the function of β, we can maximize

$$\ln \mathcal{P} = \sum_{i=1}^{N} \ln l(y^{(i)}|\beta; x^{(i)}).$$

By removing the constants, we finally arrives at the **minimization** problem of L^2 loss function (whose difference with **MSE -- mean squared error** is only up to the factor 1/N)

$$N \cdot \text{MSE} = N \cdot L(\beta) = \sum_{i=1}^{N} (y^{(i)} - \tilde{x}^{(i)}\beta)^2 = ||Y - \tilde{X}\beta||_2^2.$$

MAP (instead of MLE) Estimation in Bayesian Statistics

Recall the likelihood function -- we interpret it as the probability of observing the response data, given the parameter β as fixed, i.e. conditional probability

$$\mathcal{P}(y^{(i)}, 1 \le i \le N | x^{(i)}, \beta) = \prod_{i=1}^{N} l(y^{(i)} | x^{(i)}, \beta)$$

Now we take a Bayesian approach -- assume β is the random variable with **prior distirbution** $\mathcal{P}(\beta)$. Then the **posterior distribution** of β given the data is

$$\mathcal{P}(\beta | x^{(i)}, y^{(i)}, 1 \le i \le N) \propto \mathcal{P}(\beta)\mathcal{P}(y^{(i)}, 1 \le i \le N | \beta, x^{(i)}).$$

The **Bayesian** estimation aims to maximize the posterior distribution. It is formally termed as **Maximum A-Posteriori Estimation (MAP)**. Note that

$$\text{argmax}_\beta \mathcal{P}(\beta | x^{(i)}, y^{(i)}, 1 \le i \le N) = \text{argmax}_\beta \ln \mathcal{P}(\beta | x^{(i)}, y^{(i)}, 1 \le i \le N)$$

In other words, β maximizes one function if and only if it maximizes the other, and vice versa.

- Case 1: The prior distribution $\mathcal{P}(\beta_i = x) \propto \exp(-x^2), i \ge 1$ is Gaussian-like, and different β_i are independent. Now the minimization problem becomes

  $$\min_\beta ||Y - \tilde{X}\beta||_2^2 + \lambda ||\beta||_2^2.$$

  here $||\beta||_2^2 = \sum_{i=1}^{p} \beta_i^2$. This is called **Ridge Regression**.

- Case 2: The prior distribution $\mathcal{P}(\beta_i = x) \propto \exp(-|x|), i \ge 1$ is double-exponential (Laplace) like, and different β_i are independent. Now the minimization problem becomes

  $$\min_\beta ||Y - \tilde{X}\beta||_2^2 + \lambda \sum_{i=1}^{p} |\beta_i|$$

  This is called **LASSO Regression**).

In general, these additional terms are called the **regularization terms**. In statistics, regularization is equivalent to Bayesian prior. Here λ is the adjustable parameter in algorithm -- its choice is empirical while sometimes very important for model performance (where the word "alchemy" arises in machine learning) Roughly it controls the **complexity** of the model:

- If $\lambda \to \infty$, we have $\beta_i \to 0 (i \ge 1)$ and $\beta_0 = \bar{y}$. (no complexity in the model)
- If $\lambda \to 0$, it will yield the same results with OLS. (Same complexity in model as OLS)

Why control the complexity? Recall the bias-variance tradeoff -- sometimes reducing the complexity of a model **might** help to improve performance in the test dataset.

Algorithm consideration

The optimization for ridge regression is similar to OLS -- try to derive the analytical solution your self. The optimization for LASSO is non-trival and is the important topic in convex optimization.

Prediction in Test Data

Now from the same training dataset, we have three β estimated from three different models, namely $\hat{\beta}^{OLS}, \hat{\beta}^{Ridge}, \hat{\beta}^{Lasso}$ because they are the minimizers of three different loss functions.

Given the new observation called $x^{(test)}$, the formal expression of predictions from different methods are the same

$$\hat{y}^{(test)} = \hat{\beta}_0 + \hat{\beta}_1 x_1^{(test)} + .. + \hat{\beta}_p x_p^{(test)} = \tilde{x}^{(test)} \hat{\beta}.$$

The **only** difference is what $\hat{\beta}$ we use. Of course, the corresponded prediction values are also different.

Model Performance Evaluation

- Mean Square Error (MSE) -- the lower, the better (in test data): $\frac{1}{N} \sum_{i=1}^{N} (y^{(i)} - \hat{y}^{(i)})^2$
- R-squared (coefficient of determination, R^2) -- the larger, the better (in test data): $1 - \frac{\sum_{i=1}^{N} (y^{(i)} - \hat{y}^{(i)})^2}{\sum_{i=1}^{N} (y^{(i)} - \bar{y})^2} = 1 - \frac{MSE}{Var(Y)}$

Question: What about on the training dataset?

Conclusion: **By definition**, compared with Ridge or LASSO regression, OLS **will be sure** to have the smallest MSE (hence largest R^2) on **training dataset**. Think why!

Example: Diabetes Dataset

We use the scikit-learn package to load the data and run regression. More tutorials about linear models can be found here.

Data from this paper by Professor Robert Tibshirani et al.

In [1]:
```
from sklearn import datasets
X,y= datasets.load_diabetes(return_X_y = True)
```

In [2]:
```
help(datasets.load_diabetes)
```

```
Help on function load_diabetes in module sklearn.datasets._base:

load_diabetes(*, return_X_y=False, as_frame=False)
    Load and return the diabetes dataset (regression).

    ==============   ==================
    Samples total    442
```

```
Dimensionality   10
Features         real, -.2 < x < .2
Targets          integer 25 - 346
==============   ==================

Read more in the :ref:`User Guide <diabetes_dataset>`.

Parameters
----------
return_X_y : bool, default=False.
    If True, returns ``(data, target)`` instead of a Bunch object.
    See below for more information about the `data` and `target` object.

    .. versionadded:: 0.18

as_frame : bool, default=False
    If True, the data is a pandas DataFrame including columns with
    appropriate dtypes (numeric). The target is
    a pandas DataFrame or Series depending on the number of target columns.
    If `return_X_y` is True, then (`data`, `target`) will be pandas
    DataFrames or Series as described below.

    .. versionadded:: 0.23

Returns
-------
data : :class:`~sklearn.utils.Bunch`
    Dictionary-like object, with the following attributes.

    data : {ndarray, dataframe} of shape (442, 10)
        The data matrix. If `as_frame=True`, `data` will be a pandas
        DataFrame.
    target: {ndarray, Series} of shape (442,)
        The regression target. If `as_frame=True`, `target` will be
        a pandas Series.
    feature_names: list
        The names of the dataset columns.
    frame: DataFrame of shape (442, 11)
        Only present when `as_frame=True`. DataFrame with `data` and
        `target`.

        .. versionadded:: 0.23
    DESCR: str
        The full description of the dataset.
    data_filename: str
        The path to the location of the data.
    target_filename: str
        The path to the location of the target.

(data, target) : tuple if ``return_X_y`` is True

    .. versionadded:: 0.18
```

Generate the training and test dataset by random splitting

In [3]:

```
from sklearn.model_selection import train_test_split
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.1, random_state=4
```

In [4]:

```
print(X_train.shape)
print(y_test.shape)
```

```
(397, 10)
(45,)
```

In [5]:
```
help(train_test_split)
```

```
Help on function train_test_split in module sklearn.model_selection._split:

train_test_split(*arrays, **options)
    Split arrays or matrices into random train and test subsets

    Quick utility that wraps input validation and
    ``next(ShuffleSplit().split(X, y))`` and application to input data
    into a single call for splitting (and optionally subsampling) data in a
    oneliner.

    Read more in the :ref:`User Guide <cross_validation>`.

    Parameters
    ----------
    *arrays : sequence of indexables with same length / shape[0]
        Allowed inputs are lists, numpy arrays, scipy-sparse
        matrices or pandas dataframes.

    test_size : float or int, default=None
        If float, should be between 0.0 and 1.0 and represent the proportion
        of the dataset to include in the test split. If int, represents the
        absolute number of test samples. If None, the value is set to the
        complement of the train size. If ``train_size`` is also None, it will
        be set to 0.25.

    train_size : float or int, default=None
        If float, should be between 0.0 and 1.0 and represent the
        proportion of the dataset to include in the train split. If
        int, represents the absolute number of train samples. If None,
        the value is automatically set to the complement of the test size.

    random_state : int or RandomState instance, default=None
        Controls the shuffling applied to the data before applying the split.
        Pass an int for reproducible output across multiple function calls.
        See :term:`Glossary <random_state>`.


    shuffle : bool, default=True
        Whether or not to shuffle the data before splitting. If shuffle=False
        then stratify must be None.

    stratify : array-like, default=None
        If not None, data is split in a stratified fashion, using this as
        the class labels.

    Returns
    -------
    splitting : list, length=2 * len(arrays)
        List containing train-test split of inputs.

        .. versionadded:: 0.16
            If the input is sparse, the output will be a
            ``scipy.sparse.csr_matrix``. Else, output type is the same as the
            input type.

    Examples
    --------
    >>> import numpy as np
    >>> from sklearn.model_selection import train_test_split
```

```
>>> X, y = np.arange(10).reshape((5, 2)), range(5)
>>> X
array([[0, 1],
       [2, 3],
       [4, 5],
       [6, 7],
       [8, 9]])
>>> list(y)
[0, 1, 2, 3, 4]

>>> X_train, X_test, y_train, y_test = train_test_split(
...     X, y, test_size=0.33, random_state=42)
...
>>> X_train
array([[4, 5],
       [0, 1],
       [6, 7]])
>>> y_train
[2, 0, 3]
>>> X_test
array([[2, 3],
       [8, 9]])
>>> y_test
[1, 4]

>>> train_test_split(y, shuffle=False)
[[0, 1, 2], [3, 4]]
```

Ordinary Least Square (OLS) Linear Regression

In [6]:
```
from sklearn import linear_model
reg_ols = linear_model.LinearRegression()
reg_ols.fit(X_train,y_train) # train the parameters in training dataset
```

Out[6]:
```
LinearRegression()
```

In [25]:
```
dir(reg_ols)
```

Out[25]:
```
['__abstractmethods__',
 '__class__',
 '__delattr__',
 '__dict__',
 '__dir__',
 '__doc__',
 '__eq__',
 '__format__',
 '__ge__',
 '__getattribute__',
 '__getstate__',
 '__gt__',
 '__hash__',
 '__init__',
 '__init_subclass__',
 '__le__',
 '__lt__',
 '__module__',
 '__ne__',
 '__new__',
 '__reduce__',
 '__reduce_ex__',
 '__repr__',
```

```
'__setattr__',
'__setstate__',
'__sizeof__',
'__str__',
'__subclasshook__',
'__weakref__',
'_abc_impl',
'_check_n_features',
'_decision_function',
'_estimator_type',
'_get_param_names',
'_get_tags',
'_more_tags',
'_preprocess_data',
'_repr_html_',
'_repr_html_inner',
'_repr_mimebundle_',
'_residues',
'_set_intercept',
'_validate_data',
'coef_',
'copy_X',
'fit',
'fit_intercept',
'get_params',
'intercept_',
'n_features_in_',
'n_jobs',
'normalize',
'predict',
'rank_',
'score',
'set_params',
'singular_']
```

In [7]:
```
reg_ols.coef_
```

Out[7]:
```
array([  19.92576904, -262.55453086,  509.19112446,  336.09693678,
       -849.29530342,  480.22076125,  120.68418641,  236.71853501,
        716.61035542,   70.41045019])
```

In [8]:
```
y_pred_ols = reg_ols.predict(X_test) # generate predictions in test dataset
y_pred_ols
```

Out[8]:
```
array([143.06621271, 177.70923973, 134.80159283, 288.66523611,
       123.58429291,  96.64399491, 252.70865552, 183.51563317,
        93.96508916, 109.83316004,  98.04648824, 168.61502622,
        58.09759262, 206.5896178 , 102.4078438 , 130.25693511,
       218.0570909 , 245.9207401 , 193.24351477, 214.36945188,
       208.82778064,  90.55665059,  74.15304744, 187.1216387 ,
       156.36442036, 157.46376883, 184.17736744, 177.18027887,
        52.24263585, 110.66673778, 174.05918425,  90.89850309,
       133.07968763, 183.22988596, 173.93725211, 189.85248233,
       125.86458581, 121.53390004, 148.94895292,  60.82842472,
        76.36312191, 106.40220555, 162.20473499, 153.15077269,
       174.23003255])
```

In [9]:
```
from sklearn.metrics import mean_squared_error
mse_ols = mean_squared_error(y_test, y_pred_ols)
R2_ols =  reg_ols.score(X_test,y_test) # the R-squared value -- how good is the fitting
print(mse_ols,R2_ols)
```

```
2743.8800467688457 0.5514251914993504
```

In [10]:
```
reg_ridge = linear_model.Ridge(alpha=.02) # alpha is proportional to the lambda above -
reg_ridge.fit(X_train,y_train)
print(reg_ridge.coef_)

y_pred_ridge = reg_ridge.predict(X_test)
mse_ridge = mean_squared_error(y_test, y_pred_ridge)
R2_ridge =  reg_ridge.score(X_test,y_test)
print(mse_ridge,R2_ridge)
```

```
[  21.70557246 -252.8105591   507.97196544  328.21420703 -280.47609687
   37.89517179 -127.46013757  163.28415598  497.87046059   77.00701528]
2735.677504142067 0.5527661590071533
```

In [11]:
```
reg_lasso = linear_model.Lasso(alpha=0.1) # alpha is proportional to the lambda above -
reg_lasso.fit(X_train,y_train)
print(reg_lasso.coef_)

y_pred_lasso = reg_lasso.predict(X_test)
mse_lasso = mean_squared_error(y_test, y_pred_lasso)
R2_lasso =  reg_lasso.score(X_test,y_test)
print(mse_lasso,R2_lasso)
```

```
[   0.         -173.66792464  510.09537263  286.77901824  -64.43166585
   -0.         -226.79324775    0.          453.3557073    44.84749075]
2636.332383001763 0.569007291247414
```

In [12]:
```
print(reg_ols.score(X_train,y_train)) # note that we calculate score on TRAINING datase
print(reg_ridge.score(X_train,y_train))
print(reg_lasso.score(X_train,y_train))
```

```
0.5125152248773208
0.5102072320833588
0.5024076500883519
```

By definition, OLS has the smallest MSE (largest R-squared) on **training dataset**. What about on the test dataset?

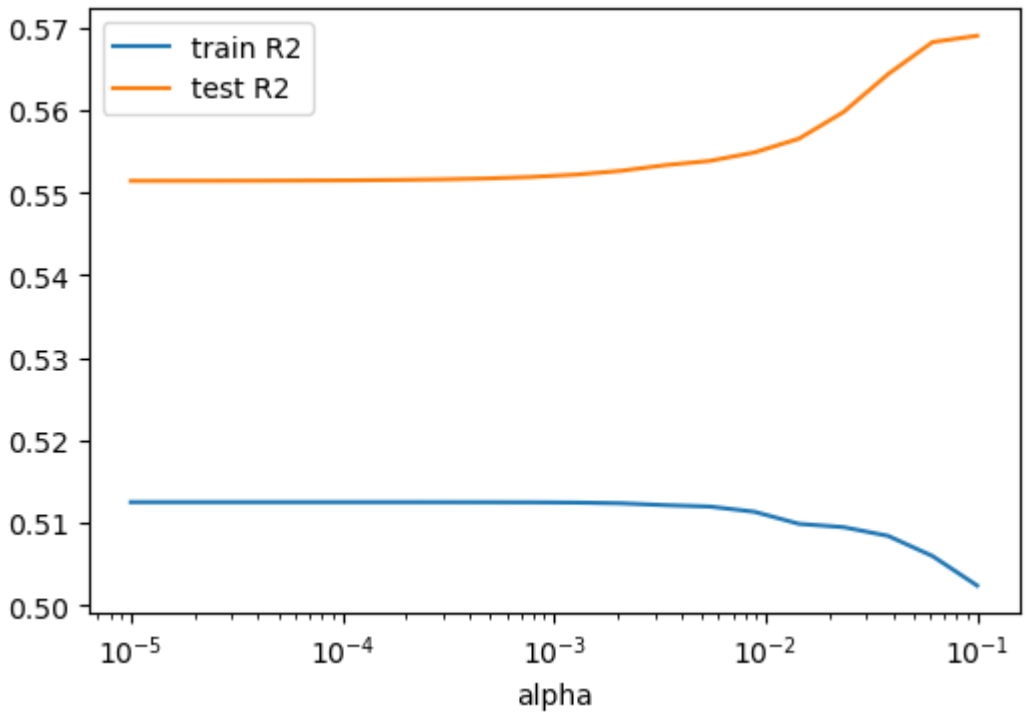
In [13]:
```
import numpy as np
train_errors = list()
test_errors = list()
alphas = np.logspace(-5, -1, 20)
for alpha in alphas:
    reg_lasso.set_params(alpha=alpha) # change the parameter of reg_lasso
    reg_lasso.fit(X_train, y_train)
    train_errors.append(reg_lasso.score(X_train, y_train))
    test_errors.append(reg_lasso.score(X_test, y_test))
```

In [14]:
```
print(alphas)
```

```
[1.00000000e-05 1.62377674e-05 2.63665090e-05 4.28133240e-05
 6.95192796e-05 1.12883789e-04 1.83298071e-04 2.97635144e-04
 4.83293024e-04 7.84759970e-04 1.27427499e-03 2.06913808e-03
 3.35981829e-03 5.45559478e-03 8.85866790e-03 1.43844989e-02
 2.33572147e-02 3.79269019e-02 6.15848211e-02 1.00000000e-01]
```

In [15]:
```
import matplotlib.pyplot as plt
fig = plt.figure(dpi=100)
plt.semilogx(alphas,train_errors,label = 'train R2')
plt.semilogx(alphas,test_errors,label = 'test R2')
plt.xlabel('alpha')
plt.legend()
```

Out[15]: <matplotlib.legend.Legend at 0x1423aeb5f40>



Therefore, a good model in training dataset does not mean it's a good model in the final test dataset. Then how can we use the best model (i.e. model with best regularization parameters)?

Cross Validation

What if we don't know the true labels in test, but the performance in test is so important to us so that we really want to select a model with greater confidence with traning dataset?

As discussed previously, we can use traning dataset to make 10 "quizzes" (each "quiz" is called a validation dataset), and let the three models to compete based on the 10 "competitions". This is called 10-fold cross-validation.

For the more detailed discussion and distinguishment between training, validation and test datasets, you can refer to this wikipedia link.

In [16]:
```
from sklearn.model_selection import cross_val_score
scores_lasso = cross_val_score(reg_lasso, X_train, y_train, cv=10) # cross-validation f
scores_ridge = cross_val_score(reg_ridge, X_train, y_train, cv=10)
scores_ols = cross_val_score(reg_ols, X_train, y_train, cv=10)
```

In [17]:
```
print(scores_lasso)
print(scores_ridge)
print(scores_ols)
```

```
[0.24777555 0.59326777 0.47897959 0.5352791  0.32317178 0.47569164
 0.6518041  0.56942576 0.25184587 0.36446431]
[0.24342237 0.57522902 0.52325584 0.53031117 0.34021405 0.48194162
 0.6585968  0.57423334 0.24263773 0.33362724]
[0.23604669 0.57037558 0.53700808 0.52611281 0.34264557 0.49282279
 0.66256801 0.57878559 0.19975324 0.34375095]
```

In [36]:

```
help(cross_val_score)
```

```
Help on function cross_val_score in module sklearn.model_selection._validation:

cross_val_score(estimator, X, y=None, *, groups=None, scoring=None, cv=None, n_jobs=Non
e, verbose=0, fit_params=None, pre_dispatch='2*n_jobs', error_score=nan)
    Evaluate a score by cross-validation

    Read more in the :ref:`User Guide <cross_validation>`.

    Parameters
    ----------
    estimator : estimator object implementing 'fit'
        The object to use to fit the data.

    X : array-like of shape (n_samples, n_features)
        The data to fit. Can be for example a list, or an array.

    y : array-like of shape (n_samples,) or (n_samples, n_outputs),             default=
None
        The target variable to try to predict in the case of
        supervised learning.

    groups : array-like of shape (n_samples,), default=None
        Group labels for the samples used while splitting the dataset into
        train/test set. Only used in conjunction with a "Group" :term:`cv`
        instance (e.g., :class:`GroupKFold`).

    scoring : str or callable, default=None
        A str (see model evaluation documentation) or
        a scorer callable object / function with signature
        ``scorer(estimator, X, y)`` which should return only
        a single value.

        Similar to :func:`cross_validate`
        but only a single metric is permitted.

        If None, the estimator's default scorer (if available) is used.

    cv : int, cross-validation generator or an iterable, default=None
        Determines the cross-validation splitting strategy.
        Possible inputs for cv are:

        - None, to use the default 5-fold cross validation,
        - int, to specify the number of folds in a `(Stratified)KFold`,
        - :term:`CV splitter`,
        - An iterable yielding (train, test) splits as arrays of indices.

        For int/None inputs, if the estimator is a classifier and ``y`` is
        either binary or multiclass, :class:`StratifiedKFold` is used. In all
        other cases, :class:`KFold` is used.

        Refer :ref:`User Guide <cross_validation>` for the various
        cross-validation strategies that can be used here.

        .. versionchanged:: 0.22
```

```
        ``cv`` default value if None changed from 3-fold to 5-fold.

n_jobs : int, default=None
    The number of CPUs to use to do the computation.
    ``None`` means 1 unless in a :obj:`joblib.parallel_backend` context.
    ``-1`` means using all processors. See :term:`Glossary <n_jobs>`
    for more details.

verbose : int, default=0
    The verbosity level.

fit_params : dict, default=None
    Parameters to pass to the fit method of the estimator.

pre_dispatch : int or str, default='2*n_jobs'
    Controls the number of jobs that get dispatched during parallel
    execution. Reducing this number can be useful to avoid an
    explosion of memory consumption when more jobs get dispatched
    than CPUs can process. This parameter can be:

        - None, in which case all the jobs are immediately
          created and spawned. Use this for lightweight and
          fast-running jobs, to avoid delays due to on-demand
          spawning of the jobs

        - An int, giving the exact number of total jobs that are
          spawned

        - A str, giving an expression as a function of n_jobs,
          as in '2*n_jobs'

error_score : 'raise' or numeric, default=np.nan
    Value to assign to the score if an error occurs in estimator fitting.
    If set to 'raise', the error is raised.
    If a numeric value is given, FitFailedWarning is raised. This parameter
    does not affect the refit step, which will always raise the error.

    .. versionadded:: 0.20

Returns
-------
scores : array of float, shape=(len(list(cv)),)
    Array of scores of the estimator for each run of the cross validation.

Examples
--------
>>> from sklearn import datasets, linear_model
>>> from sklearn.model_selection import cross_val_score
>>> diabetes = datasets.load_diabetes()
>>> X = diabetes.data[:150]
>>> y = diabetes.target[:150]
>>> lasso = linear_model.Lasso()
>>> print(cross_val_score(lasso, X, y, cv=3))
[0.33150734 0.08022311 0.03531764]

See Also
---------
:func:`sklearn.model_selection.cross_validate`:
    To run cross-validation on multiple metrics and also to return
    train scores, fit times and score times.

:func:`sklearn.model_selection.cross_val_predict`:
    Get predictions from each split of cross-validation for diagnostic
    purposes.
```

```
    :func:`sklearn.metrics.make_scorer`:
        Make a scorer from a performance metric or loss function.
```

In [18]:
```
import pandas as pd
scores_all = pd.DataFrame({"lasso": scores_lasso,"ols": scores_ols, "ridge":scores_ridg
scores_all
```

Out[18]:

	lasso	ols	ridge
0	0.247776	0.236047	0.243422
1	0.593268	0.570376	0.575229
2	0.478980	0.537008	0.523256
3	0.535279	0.526113	0.530311
4	0.323172	0.342646	0.340214
5	0.475692	0.492823	0.481942
6	0.651804	0.662568	0.658597
7	0.569426	0.578786	0.574233
8	0.251846	0.199753	0.242638
9	0.364464	0.343751	0.333627

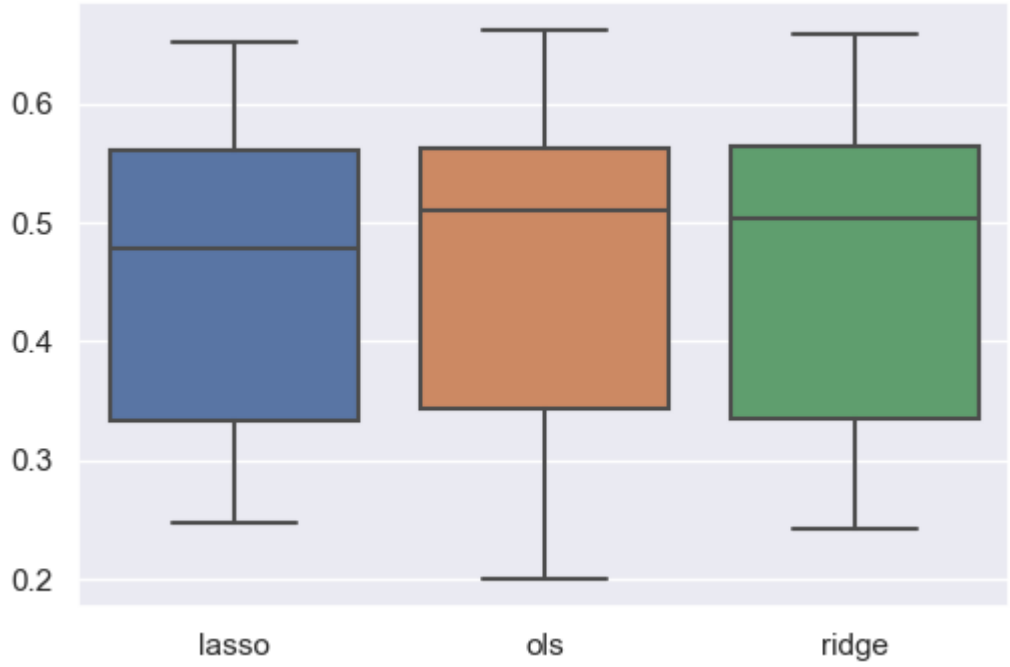
In [25]:
```
type(scores_lasso)
```

Out[25]:
```
numpy.ndarray
```

Besides mean and standard deviation, we can also use the boxplot to visualize the results.

In [19]:
```
import seaborn as sns
sns.set_theme()
fig, ax = plt.subplots(dpi=100)
sns.boxplot(data = scores_all)
```

Out[19]:
```
<AxesSubplot:>
```



In [20]:
```
scores_all.describe()
```

Out[20]:

	lasso	ols	ridge
count	10.000000	10.000000	10.000000
mean	0.449171	0.448987	0.450347
std	0.144468	0.157290	0.148598
min	0.247776	0.199753	0.242638
25%	0.333495	0.342922	0.335274
50%	0.477336	0.509468	0.502599
75%	0.560889	0.562034	0.563253
max	0.651804	0.662568	0.658597

Of course, the final judgement is still in the test dataset.

In [21]:
```
reg_lasso.score(X_test,y_test)
```

Out[21]: 0.569007291247414

In [22]:
```
reg_ridge.score(X_test,y_test)
```

Out[22]: 0.5527661590071533

In [23]:
```
reg_ols.score(X_test,y_test)
```

Out[23]: 0.5514251914993504

Exercise: Use cross-validation to select the `alpha` parameter in LASSO

In [42]:
```
# your code here
#test_errors = list()
alphas = np.logspace(-5, -1, 20)
train_errors = np.zeros_like(alphas)
for ind, alpha in enumerate(alphas):
    reg_lasso.set_params(alpha=alpha) # change the parameter of reg_lasso
    #reg_lasso.fit(X_train, y_train)
    scores_lasso = cross_val_score(reg_lasso, X_train, y_train, cv=10)
    #train_errors.append(reg_lasso.score(X_train, y_train))
    #score_mean = scores_lasso.mean()
    train_errors[ind]= scores_lasso.mean()
    #test_errors.append(reg_lasso.score(X_test, y_test))
```

In [43]:
```
print(train_errors)
print(train_errors.max())
print(train_errors.argmax())
print(alphas[train_errors.argmax()])
```

```
[0.44898873 0.44898985 0.44899167 0.44899459 0.4489993  0.44900684
 0.44901877 0.44903735 0.44906542 0.44910546 0.44919526 0.44935973
 0.44972176 0.44992911 0.44901409 0.44909977 0.45044024 0.45092872
 0.45140977 0.44917055]
0.4514097740599895
18
0.06158482110660261
```

In [44]:
```
reg_lasso_new = linear_model.Lasso(alpha=alphas[train_errors.argmax()]) # alpha is prop
reg_lasso_new.fit(X_train,y_train)
print(reg_lasso_new.coef_)

y_pred_lasso_new = reg_lasso_new.predict(X_test)
mse_lasso_new = mean_squared_error(y_test, y_pred_lasso_new)
R2_lasso_new =  reg_lasso_new.score(X_test,y_test)
print(mse_lasso_new,R2_lasso_new)
```

```
[   0.         -202.49710284  513.13597299  303.12781379 -101.29843297
   -0.         -237.82654836    0.          474.55221916   58.20204323]
2641.0204812969946 0.5682408718853533
```

Reference Reading Suggestions

- ISLR: Chapter 2,3,6
- ESL: Chapter 1,2,3
- PML: Chapter 1,2,3,4,7,11
- DL: Chapter 5