Lecture 10 Introduction to Machine Learning and Linear Regression

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

Problem

Given the *training dataset* $(x^{(i)} \in \mathbb{R}, y^{(i)} \in \mathbb{R}), i = 1, 2, \ldots, N$, 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

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

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)

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

  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 loss function -- note that this is the quadratic function of w and b, and we can analytically solve $\partial_w L = \partial_b L = 0$, 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 learned 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 for 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.

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

In [1]:
```
import numpy as np

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

    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 i
ntercept b (float)
         '''

        cov_mat = np.cov(x,y,bias=True) # covariance matrix, bias = True makes the fa
ctor is 1/N -- but it doesn't matter actually, since the factor will be cancelled
        self.w = cov_mat[0,1] / cov_mat[0,0] # the (0,1) element is COV(X,Y) and (0,
0) element is Var(X). (1,1) is Var(Y)
        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 corre
sponding x
        '''

        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 [2]:
```python
import pandas as pd
house = pd.read_csv('kc_house_data.csv')
house.sample(5)
```

Out[2]:

	id	date	price	bedrooms	bathrooms	sqft_living	sqft_lot	floors	waterfront	view
16792	6388930420	20140805T000000	582000.0	3	2.50	2380	19860	2.0	0	(
7648	7625703945	20140701T000000	345000.0	2	1.00	1080	7775	1.0	0	(
9653	1402600110	20150226T000000	392000.0	4	2.25	2360	7733	2.0	0	(
21563	9406530090	20141020T000000	337000.0	4	2.50	2470	5100	2.0	0	(
19437	5379803386	20140801T000000	289950.0	4	1.75	1500	8400	1.0	0	(

5 rows × 21 columns

In [3]:
```python
house.drop(['id','date','zipcode','lat','long','yr_built','yr_renovated'],axis = 1, i
nplace = True)
house
```

Out[3]:

	price	bedrooms	bathrooms	sqft_living	sqft_lot	floors	waterfront	view	condition	grade	sqft_above
0	221900.0	3	1.00	1180	5650	1.0	0	0	3	7	1180
1	538000.0	3	2.25	2570	7242	2.0	0	0	3	7	2170
2	180000.0	2	1.00	770	10000	1.0	0	0	3	6	770
3	604000.0	4	3.00	1960	5000	1.0	0	0	5	7	1050
4	510000.0	3	2.00	1680	8080	1.0	0	0	3	8	1680
...	...	...	...	...	...	...	...	...	...	...	...
21608	360000.0	3	2.50	1530	1131	3.0	0	0	3	8	1530
21609	400000.0	4	2.50	2310	5813	2.0	0	0	3	8	2310
21610	402101.0	2	0.75	1020	1350	2.0	0	0	3	7	1020
21611	400000.0	3	2.50	1600	2388	2.0	0	0	3	8	1600
21612	325000.0	2	0.75	1020	1076	2.0	0	0	3	7	1020

21613 rows × 14 columns

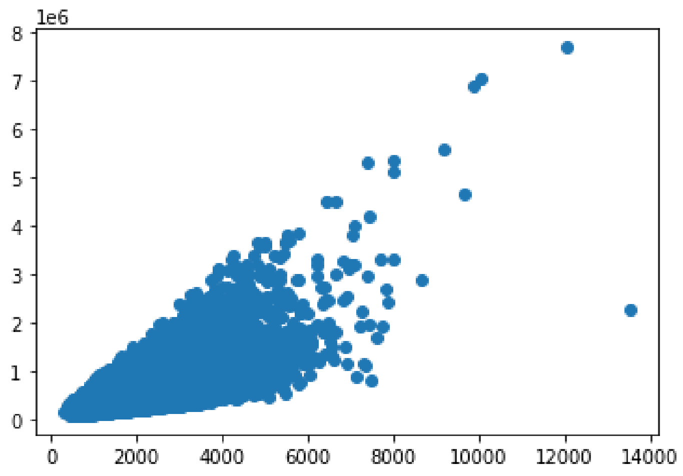
In [4]:
```python
X = house.iloc[:,1:].to_numpy()
y = house['price'].to_numpy()
```

In [5]:
```python
X.shape
```

Out[5]:
```
(21613, 13)
```

In [6]:
```
import matplotlib.pyplot as plt
x = X[:,2]
plt.scatter(x,y)
```

Out[6]:
```
<matplotlib.collections.PathCollection at 0x7fd05deda3d0>
```



In [7]:
```python
lreg = MyLinearRegression1D() # initialize the instance of one estimator
help(lreg)
```

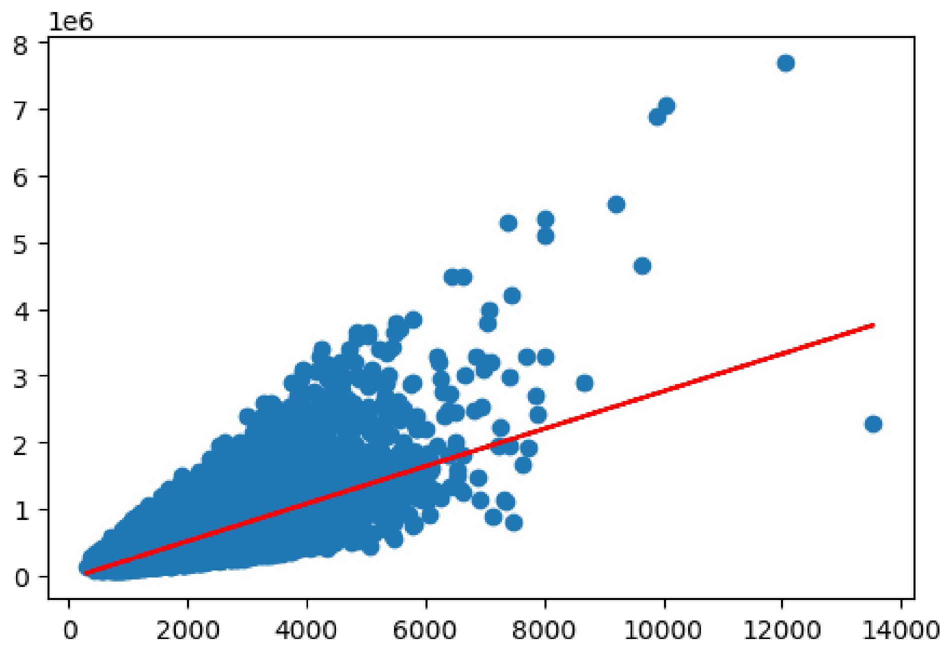
```
Help on MyLinearRegression1D in module __main__ object:

class MyLinearRegression1D(builtins.object)
 |  The single-variable linear regression estimator -- writing in the style of sklea
rn package
 |  
 |  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
intercept 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 corr
esponding 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 [8]:
```python
lreg.fit(x,y)
```

In [9]:
```python
lreg.score(x,y)
```

Out[9]:
```
0.49286538652201417
```

In [10]:
```
fig = plt.figure(dpi = 100)
plt.scatter(x,y)
plt.plot(x,lreg.predict(x),'r')
```
Out[10]:
```
[<matplotlib.lines.Line2D at 0x7fd05e253a90>]
```


In [11]:
```
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[11]:
```
LinearRegression()
```
In [12]:
```
print(lreg.w,lreg.b)
print(lreg_sklearn.coef_, lreg_sklearn.intercept_)
```
```
280.8066899295006 -43867.60153385543
[280.80668993] -43867.601533855544
```
In [13]:
```
lreg_sklearn.score(x.reshape(-1,1),y)
```
Out[13]:
```
0.49286538652201417
```

In [14]:
```
help(lreg_sklearn)
```

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

class LinearRegression(sklearn.base.MultiOutputMixin, sklearn.base.RegressorMixin, L
inearModel)
 |  LinearRegression(*, fit_intercept=True, normalize=False, copy_X=True, n_jobs=Non
e)
 |  
 |  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* $(x^{(i)}, y^{(i)}), i = 1, 2, \dots, N$, this time with $y^{(i)} \in \mathbb{R}$ and $x^{(i)} \in \mathbb{R}^p$, we fit the multi-variable linear function

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

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

Here β is called regression coefficients, and β_0 specially referred to intercept.

Using the whole training dataset, we can write as

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

where

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

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 time 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 find β such that $\tilde{X}\beta$ is as close as Y.

Training the model

- With the training dataset, define the loss function $L(\beta)$ of parameters β, which is also called **mean squared error** (MSE)
  $$L(\beta) = \frac{1}{N}\sum_{i=1}^{N}\left(\hat{y}^{(i)} - y^{(i)}\right)^2 = \frac{1}{N}\sum_{i=1}^{N}(y^{(i)} - \tilde{x}^{(i)}\beta)^2,$$
  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 $\nabla L(\beta) = 0$.

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

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

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 (https://en.wikipedia.org/wiki/Ordinary_least_squares)](https://en.wikipedia.org/wiki/Ordinary_least_squares)

Denote $\tilde{X} = (\tilde{X}_0, \tilde{X}_1, \ldots, \tilde{X}_p)$, then $\tilde{X}\beta = \sum_{k=0}^{p} \beta_k \tilde{X}_k$. We require that the residual $Y - \tilde{X}\beta$ is 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.

```
In [ ]: 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)
```

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

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

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 $(x, x^2, \ldots, x^p)$ 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 discribe 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 (https://towardsdatascience.com/understanding-the-bias-variance-tradeoff-165e6942b229)](https://towardsdatascience.com/understanding-the-bias-variance-tradeoff-165e6942b229): Simple models -- large bias, low variance. Complex models -- low bias, large variance

[No Free Lunch Theorem (https://analyticsindiamag.com/what-are-the-no-free-lunch-theorems-in-data-science/#:~:text=Once%20Upon%20A%20Time,that%20they%20brought%20a%20drink)](https://analyticsindiamag.com/what-are-the-no-free-lunch-theorems-in-data-science/#:~:text=Once%20Upon%20A%20Time,that%20they%20brought%20a%20drink): (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 (https://en.wikipedia.org/wiki/No_free_lunch_in_search_and_optimization)](https://en.wikipedia.org/wiki/No_free_lunch_in_search_and_optimization).

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

Note: The detailed mathematical derivations below are optional material. You only need to know (for quiz/exam):

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 prabability to observe $\epsilon^{(i)}$ within the small interval $[z, z + \Delta z]$ is roughly

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

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

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

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

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

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

Equivalently, 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)

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

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

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

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 maximaze the posterior distribution. It is formally termed as **Maximum A-Posteriori Estimation (MAP)**. Note that

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

- 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 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** (https://en.wikipedia.org/wiki/Lasso_(statistics)).

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 $\beta_0 = \bar{y}$.
- If $\lambda \to 0$, it will yield the same results with OLS.

Why control the complexity? Recall the bias-variance tradeoff -- sometimes reduce the complexity of model **might** help to improve performace in 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 (https://www.cs.ubc.ca/~schmidtm/Documents/2005_Notes_Lasso.pdf) 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): $1 - \frac{\sum_{i=1}^N (y^{(i)} - \hat{y}^{(i)})^2}{\sum_{i=1}^N (y^{(i)} - \bar{y})^2}$

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 (https://scikit-learn.org/stable/index.html) to load the data and run regression. More tutorials about linear models can be found here (https://scikit-learn.org/stable/modules/linear_model.html).

Data from this paper (https://web.stanford.edu/~hastie/Papers/LARS/LeastAngle_2002.pdf) by Professor Robert Tibshirani et al (https://statweb.stanford.edu/~tibs/index.html).

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

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

Generate the training and test dataset by random splitting

```
In [ ]: 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
        =42)
```

```
In [ ]: print(X_train.shape)
        print(y_test.shape)
```

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

Ordinary Least Square (OLS) Linear Regression

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

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

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

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

```
In [ ]: 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 fitti
        ng in test dataset?
        print(mse_ols,R2_ols)
```

```
In [ ]: reg_ridge = linear_model.Ridge(alpha=.02) # alpha is proportional to the lambda above
        -- only up to the constant
        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)
```

In [ ]:
```python
reg_lasso = linear_model.Lasso(alpha=0.1) # alpha is proportional to the lambda above
-- only up to the constant
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)
```

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

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

In [ ]:
```python
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 [ ]:
```python
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()
```

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 (https://scikit-learn.org/stable/modules/cross_validation.html)

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 (https://en.wikipedia.org/wiki/Training,_validation,_and_test_sets#Test_dataset).

In [ ]:
```python
from sklearn.model_selection import cross_val_score
scores_lasso = cross_val_score(reg_lasso, X_train, y_train, cv=10) # cross-validation
function in sklearn
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 [ ]:
```python
print(scores_lasso)
print(scores_ridge)
print(scores_ols)
```

In [ ]:
```python
help(cross_val_score)
```

```
In [ ]: import pandas as pd
        scores_all = pd.DataFrame({"lasso": scores_lasso,"ols": scores_ols, "ridge":scores_ri
        dge})
        scores_all
```

Besides mean and standard deviation, we can also use the boxplot (https://towardsdatascience.com/understanding-boxplots-5e2df7bcbd51) to visualize the results.

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

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

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

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

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

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

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

```
In [ ]: # your code here
```

Reference Reading Suggestions

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