Lecture 10 Introduction to Machine Learning and Linear Regression

Motivating Example: 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
```

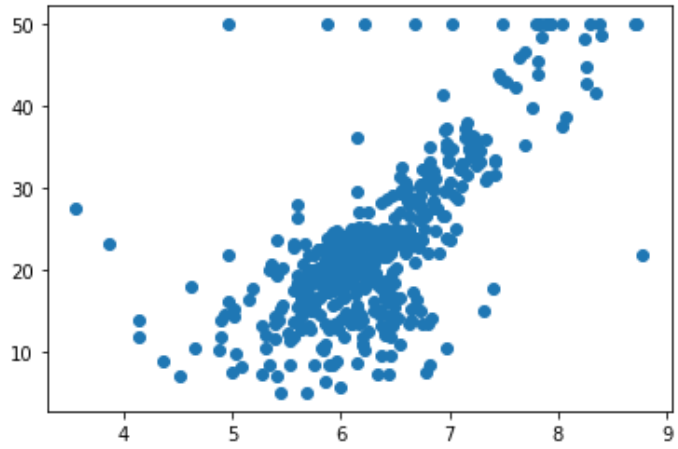
In [2]:
```
from sklearn import datasets
X, y = datasets.load_boston(return_X_y=True)
```

In [3]:
```
X.shape
```

Out[3]:
```
(506, 13)
```

In [4]:
```
import matplotlib.pyplot as plt
plt.scatter(X[:,5],y)
```

Out[4]:
```
<matplotlib.collections.PathCollection at 0x7fd096f55a90>
```



In [5]:
```
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)
```

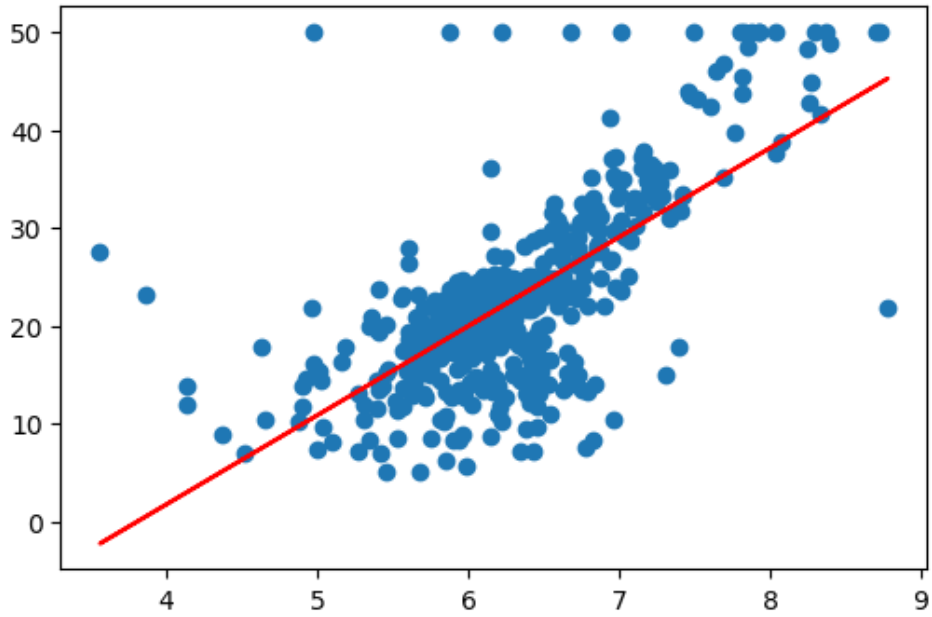
In [6]:
```
lreg.fit(X[:,5],y)
```

In [7]:
```
lreg.score(X[:,5],y)
```

Out[7]:
```
0.4835254559913341
```

In [8]:
```
fig = plt.figure(dpi = 100)
plt.scatter(X[:,5],y)
plt.plot(X[:,5],lreg.predict(X[:,5]),'r')
```

Out[8]:
```
[<matplotlib.lines.Line2D at 0x7fd09708dd50>]
```



In [9]:
```
from sklearn import linear_model # compare with the scikit learn package
lreg_sklearn = linear_model.LinearRegression()
lreg_sklearn.fit(X[:,5].reshape(-1,1),y) #only accept 2D-array as x
```

Out[9]:
```
LinearRegression()
```

In [10]:
```
print(lreg.w,lreg.b)
print(lreg_sklearn.coef_, lreg_sklearn.intercept_)
```

```
9.102108981180306 -34.67062077643854
[9.10210898] -34.67062077643857
```

In [11]:
```
lreg_sklearn.score(X[:,5].reshape(-1,1),y)
```

Out[11]:
```
0.48352545599133423
```

In [12]:
```
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.
```

(Materials in Midterm Exam end here)

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): 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): (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).

Linear Regression (Multivariate Case) - Ordinary Least Square (OLS) Approach

Recall the basic task of **supervised learning**: given the *training dataset* $(x^{(i)}, y^{(i)}), i = 1, 2, \ldots, N$ with $y^{(i)} \in \mathbb{R}^q$ (for simplicity, assume $q = 1$) denotes the *labels*, the supervised learning aims to find a mapping $y \approx \mathbf{f}(x) : \mathbb{R}^p \to \mathbb{R}$ that we can use it to make predictions on the test dataset.

Model Setup

Model assumption 1: Linear Mapping Assumption.

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

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

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

Using the whole training dataset, we can write as

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

where

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

is also called the augmented data matrix.

Model assumption 2: Gaussian Residual Assumption (L^2 loss assumption)

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

The residuals or errors $\epsilon^{(i)}$ are **assumed** 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 $y^j (i = 1, 2, \ldots, N)$ is

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

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

Equivalently, as the function of β, we can maximize

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

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)

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

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

We also have the prediction

$$\hat{y}^{(i)} = \tilde{x}^{(i)}\hat{\beta}.$$

Algorithm: Normal Equation

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. 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)

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 vertical to the plane spanned by $\tilde{X}_k$, which yields

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

Extensions: Regularization, Ridge Regression and LASSO

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

1) the basic concepts of Ridge regression and LASSO ;

2) where does the additional regularization terms come from ;

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

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

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

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

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

MAP (instead of MLE) Estimation in Bayesian Statistics

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

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

- Case 1: The prior distribution $\mathcal{P}(\beta_i = x) \propto \exp(-x^2), i \geq 1$ 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 $\mathcal{P}(\beta_i = x) \propto \exp(-|x|), i \geq 1$ 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 $\beta_i \to 0 (i \geq 1)$ 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.

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 [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
=42)
```

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

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

Ordinary Least Square (OLS) Linear Regression

In [4]:
```
from sklearn import linear_model
reg_ols = linear_model.LinearRegression()
reg_ols.fit(X_train,y_train) # train the parameters in training dataset
```

Out[4]:
```
LinearRegression()
```

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

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

In [7]:
```
y_pred_ols = reg_ols.predict(X_test) # generate predictions in test dataset
```

In [8]:
```
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)
```

```
2743.8800467688443 0.5514251914993505
```

In [9]:
```
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)
```

```
[  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 [10]:
```
reg_lasso = linear_model.Lasso(alpha=.05) # 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)
```

```
[   0.         -212.76030063  514.23777918  309.6748151  -131.90735899
   -0.         -215.96745627   34.17218616  479.55741824   61.49888891]
2650.840160539064 0.5666355317609786
```

In [11]:
```
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))
```

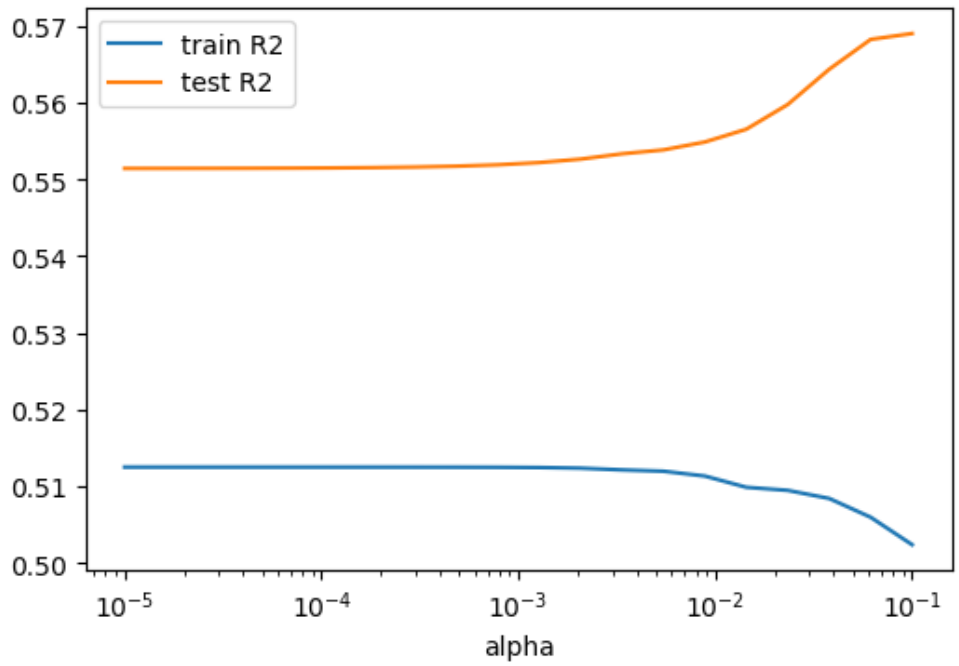
```
0.5125152248773208
0.5102072320833588
0.5072801848497961
```

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

In [12]:
```
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 [13]:
```
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[13]:
```
<matplotlib.legend.Legend at 0x7fd4b4170e90>
```



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.

In [53]:
```
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 [54]:
```
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 [46]:
```
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=
None, 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),             defa
ult=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 [55]:
```
import pandas as pd
scores_all = pd.DataFrame({"lasso": scores_lasso,"ols": scores_ols, "ridge":scores_ridge})
scores_all
```

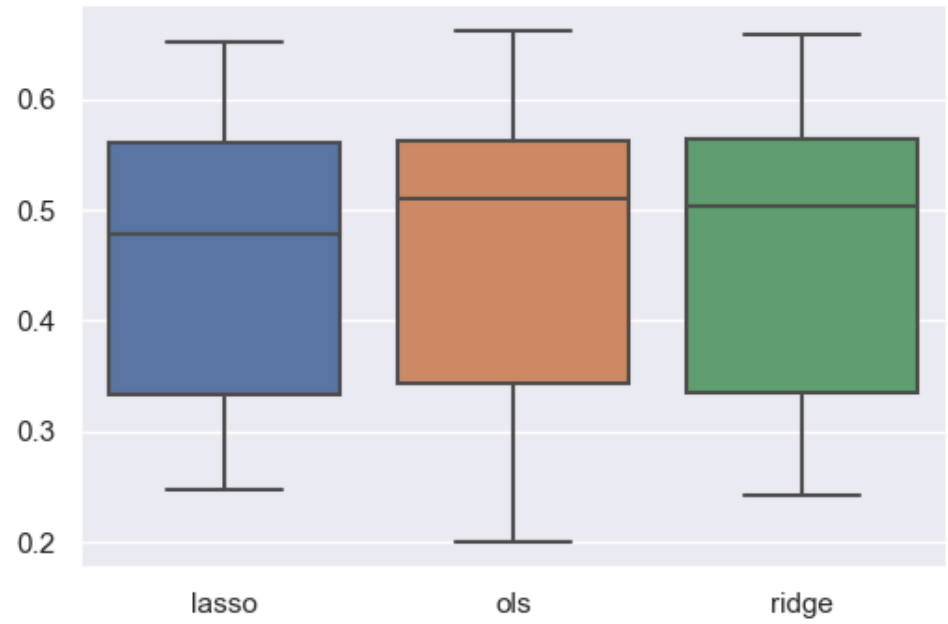
Out[55]:

	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

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

In [56]:
```
import seaborn as sns
sns.set_theme()
fig, ax = plt.subplots(dpi=100)
sns.boxplot(data = scores_all)
```

Out[56]: <AxesSubplot:>



In [57]:
```
scores_all.describe()
```

Out[57]:

	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 [26]:
```
reg_lasso.score(X_test,y_test)
```

Out[26]: 0.569007291247414

In [42]:
```
reg_ridge.score(X_test,y_test)
```

Out[42]: 0.5527661590071533

In [27]:
```
reg_ols.score(X_test,y_test)
```

Out[27]: 0.5514251914993505

Reference Reading Suggestions

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