5.

least square regression: $E_{LS}(w) = \frac{1}{2}\|\tilde{\Phi}w - \hat{y}\|^2$

insert $\tilde{\Phi}, \hat{y}$: $\tilde{\Phi}w - \hat{y} = \begin{pmatrix}\Phi w \\ \sqrt{\lambda} w\end{pmatrix} - \begin{pmatrix} y \\ 0_m\end{pmatrix} = \begin{pmatrix}\Phi w - y \\ \sqrt{\lambda} w\end{pmatrix}$

$\Rightarrow E_{LS}(w) = \frac{1}{2}\left\|\begin{pmatrix}\Phi w - y \\ \sqrt{\lambda} w\end{pmatrix}\right\|^2 = \frac{1}{2}(\|\Phi w - y\|^2 + \|\sqrt{\lambda} w\|^2)$

$= \frac{1}{2}\|\Phi w - y\|^2 + \frac{\lambda}{2}\|w\|^2$ ← which is ridge regression

6. a)

best_error is initialized to -1, which is always less than any legitimate MSE. This causes the condition (error <= best_error) or (best_error == -1) to always evaluate as true during the first iteration, making the comparison ineffective and always overwriting best_error and best_degree.

b) initialize best_error to a very large value, such as float('inf'). This ensures the first calculated error will be smaller and will update best_error and best_degree appropriately.

7.

Bayes Rule: $p(w, \beta | D) \propto p(y | \Phi, w, \beta)\, p(w, \beta)$

Taking logarithm: $\log p(w, \beta | D) = \log p(\Phi, w, \beta) + \log p(w, \beta) + \text{const.}$

Likelihood term: $\log p(y | \Phi, w, \beta) = \log\left(\prod_{i=1}^{N} \mathcal{N}(y_i | w^T \phi(x_i), \beta^{-1})\right) =$

$$-\frac{N}{2}\log(2\pi) + \frac{N}{2}\log\beta - \frac{\beta}{2}\sum_{i=1}^{N}(y_i - w^T\phi(x_i))^2$$

Prior for w: $\log p(w | \beta) = -\frac{M}{2}\log(2\pi) + \frac{1}{2}\log|S_0| + \frac{M}{2}\log\beta -$

$$\frac{\beta}{2}(w - m_0)^T S_0^{-1}(w - m_0)$$

Gamma prior for β: $\log p(\beta) = (a_0 - 1)\log\beta - b_0\beta + \text{cost.}$

$S_N = (S_0^{-1} + \beta\Phi^T\Phi)^{-1}$, $m_N = S_N(S_0^{-1}m_0 + \Phi^T y)$

$a_N = a_0 + \frac{N}{2}$, $b_N = b_0 + \frac{1}{2}(y^T y + m_0^T S_0^{-1} m_0 - m_N^T S_N^{-1} m_N)$

8.

$\nabla_w E_{ridge}(w) = \Phi^T(\Phi w - y) + \lambda w = 0 \Rightarrow w = (\Phi^T\Phi + \lambda I)^{-1}\Phi^T y$

The matrix $\Phi^T\Phi$ becomes singular/non-invertible

Regularization adds term λI, which ensures $\Phi^T\Phi + \lambda I$ is always invertible

9.

a) $w_{new} = w^*/a$

b) $\lambda_{new} = \lambda/a^2$

a) given: $X_{new} = aX$, $a>0$, model prediction: $y = Xw^*$

For X_{new}: $y_{new} = X_{new} w_{new} = (aX) w_{new}$

To have same prediction: $y_{new} = y \Rightarrow aX w_{new} = Xw^* \Rightarrow w_{new} = \frac{w^*}{a}$

b) insert $w_{new} = w^* a \Rightarrow \frac{\lambda_{new}}{2} w_{new}^T w_{new} = \frac{\lambda_{new}}{2} a^2 w^{*T} w^*$

in order to have same prediction: $\frac{\lambda_{new}}{2} a^2 w^{*T} w^* = \frac{\lambda}{2} w^{*T} w^*$

$\Rightarrow \lambda_{new} = \lambda/a^2$

Programming Task: Linear Regression

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

from sklearn.datasets import fetch_california_housing
from sklearn.model_selection import train_test_split
```

Your task

This notebook provides a code skeleton for performing linear regression. Your task is to complete the functions where required. You are only allowed to use built-in Python functions, as well as any `numpy` functions. No other libraries / imports are allowed.

In the beginning of every function there is docstring which specifies the input and and expected output. Write your code in a way that adheres to it. You may only use plain python and anything that we imported for you above such as numpy functions (i.e. no scikit-learn classifiers).

Load and preprocess the data

In this assignment we will work with the Boston Housing Dataset. The data consists of 506 samples. Each sample represents a district in the city of Boston and has 13 features, such as crime rate or taxation level. The regression target is the median house price in the given district (in $1000's).

More details can be found here: http://lib.stat.cmu.edu/datasets/boston (http://lib.stat.cmu.edu/datasets/boston)

In [2]:
```
X , y = fetch_california_housing(return_X_y=True)

# Add a vector of ones to the data matrix to absorb the bias term
# (Recall slide #7 from the lecture)
X = np.hstack([np.ones([X.shape[0], 1]), X])
# From now on, D refers to the number of features in the AUGMENTED dataset (i.e.

# Split into train and test
test_size = 0.9
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=test_size)
```

Task 1: Fit standard linear regression

In [3]:
```
def fit_least_squares(X, y):
    """Fit ordinary least squares model to the data.

    Parameters
    ----------
    X : array, shape [N, D]
        (Augmented) feature matrix.
    y : array, shape [N]
        Regression targets.

    Returns
    -------
    w : array, shape [D]
        Optimal regression coefficients (w[0] is the bias term).

    """
    ### YOUR CODE HERE ###
    w = np.linalg.inv(X.T @ X) @ X.T @ y

    return w
```

Task 2: Fit ridge regression

In [4]:
```
def fit_ridge(X, y, reg_strength):
    """Fit ridge regression model to the data.

    Parameters
    ----------
    X : array, shape [N, D]
        (Augmented) feature matrix.
    y : array, shape [N]
        Regression targets.
    reg_strength : float
        L2 regularization strength (denoted by lambda in the lecture)

    Returns
    -------
    w : array, shape [D]
        Optimal regression coefficients (w[0] is the bias term).

    """
    ### YOUR CODE HERE ###
    D = X.shape[1]

    I = np.eye(D)

    I[0, 0] = 0

    w = np.linalg.inv(X.T @ X + reg_strength * I) @ X.T @ y

    return w
```

Task 3: Generate predictions for new data

In [5]:
```
def predict_linear_model(X, w):
    """Generate predictions for the given samples.

    Parameters
    ----------
    X : array, shape [N, D]
        (Augmented) feature matrix.
    w : array, shape [D]
        Regression coefficients.

    Returns
    -------
    y_pred : array, shape [N]
        Predicted regression targets for the input data.

    """
    ### YOUR CODE HERE ###
    y_pred = X @ w

    return y_pred
```

Task 4: Mean squared error

In [6]:
```
def mean_squared_error(y_true, y_pred):
    """Compute mean squared error between true and predicted regression targets.

    Reference: `https://en.wikipedia.org/wiki/Mean_squared_error`

    Parameters
    ----------
    y_true : array
        True regression targets.
    y_pred : array
        Predicted regression targets.

    Returns
    -------
    mse : float
        Mean squared error.

    """
    ### YOUR CODE HERE ###
    mse = np.mean((y_true - y_pred) ** 2)

    return mse
```

Compare the two models

The reference implementation produces

- MSE for Least squares ≈ **0.5347**
- MSE for Ridge regression ≈ **0.5331**

You results might be slightly (i.e. $\pm 1\%$) different from the reference soultion due to numerical reasons.

In [7]:
```
# Load the data
np.random.seed(1234)
X , y = fetch_california_housing(return_X_y=True)
X = np.hstack([np.ones([X.shape[0], 1]), X])
test_size = 0.9
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=test_size)

# Ordinary least squares regression
w_ls = fit_least_squares(X_train, y_train)
y_pred_ls = predict_linear_model(X_test, w_ls)
mse_ls = mean_squared_error(y_test, y_pred_ls)
print('MSE for Least squares = {0}'.format(mse_ls))

# Ridge regression
reg_strength = 1
w_ridge = fit_ridge(X_train, y_train, reg_strength)
y_pred_ridge = predict_linear_model(X_test, w_ridge)
mse_ridge = mean_squared_error(y_test, y_pred_ridge)
print('MSE for Ridge regression = {0}'.format(mse_ridge))
```

```
MSE for Least squares = 0.5347102426024742
MSE for Ridge regression = 0.5341685503387622
```

In [ ]:
```

```