Standardization + Cross-Validation

- Avoid leaking information between training and validation/testing data

```
chemistry = pd.read_csv('chemistry_samples.csv')
X = chemistry.loc[:, chemistry.columns != 'lc50']
X = standardise(X)
y = chemistry.loc[:, 'lc50']
# Later ...
folds = cross_val_split(X.shape[0], num_folds)
for i in range(len(folds)):
    X_train, y_train = # Some code
    X_val, y_val = # Some code

    beta = train(X_train, y_train)
    y_hat = predict(beta, X_val)
    score = evaluate(y_hat, y_val)
```

Ridge Regression: Augmentation + Regularization

- Augmentation means that β vector includes the intercept β_0.
- Regularization penalty should be independent on the intercept.

```
def ridge_estimate(X_aug, y, lambd):
    _, p = X_aug.shape
    I = np.identity(p)
    beta_ridge = np.linalg.solve(X_aug.T @ X_aug + lambd * I, X_aug.T @ y)
    return beta_ridge
```

Ridge Regression: Augmentation + Regularization

- Augmentation means that β vector includes the intercept β_0.
- Regularization penalty should be independent on the intercept.

```python
def ridge_estimate(X_aug, y, lambd):
    _, p = X_aug.shape
    I = np.identity(p)
    I[0, 0] = 0 # If augmented from the left side
    beta_ridge = np.linalg.solve(X_aug.T @ X_aug + lambd * I, X_aug.T @ y)
    return beta_ridge
```

- If you standardize your data (to zero-mean and unit-variance), you don't need to augment and fit for β_0
- But if you standardize and still augment?!

Lasso Regression: Augmentation + Regularization

```python
def minimize_ls_huber(X, y, lambd, n_iters, step_size):
    n, p = X.shape
    XX = X.T @ X / n
    Xy = X.T @ y / n

    # initialise betas
    beta = np.zeros(shape=(p, 1))

    # gradient descent
    for i in range(n_iters):
        grad_huber_beta = grad_huber(beta)
        grad = XX @ beta - Xy + lambd * grad_huber_beta
        # gradient descent update
        beta -= step_size * grad

    return beta
```

Lasso Regression: Gradient Descent

```python
def minimize_ls_huber(X, y, lambd, n_iters, step_size):
    n, p = X.shape
    XX = X.T @ X / n
    Xy = X.T @ y / n

    # initialise betas
    beta = np.zeros(shape=(p, 1))

    # gradient descent
    for i in range(n_iters):
        grad_huber_beta = grad_huber(beta)
        grad_huber_beta[0] = 0
        grad = XX @ beta - Xy + lambd * grad_huber_beta
        # gradient descent update
        beta -= step_size * grad

    return beta
```

Lasso Regression: Gradient Descent

```
def minimize_ls_huber(X, y, lambd, n_iters, step_size):
    n, p = X.shape
    XX = X.T @ X / n
    Xy = X.T @ y / n

    # initialise betas
    beta = np.zeros(shape=(p, 1))

    # gradient descent
    for i in range(n_iters):
        grad_huber_beta = grad_huber(beta)
        grad_huber_beta[0] = 0
        grad = XX @ beta - Xy + lambd * np.sum(grad_huber_beta)
        # gradient descent update
        beta -= step_size * grad

    return beta
```

Lasso Regression: Gradient Descent

```
def minimize_ls_huber(X, y, lambd, n_iters, step_size):
    n, p = X.shape
    XX = X.T @ X / n
    Xy = X.T @ y / n

    # initialise betas
    beta = np.zeros(shape=(p, 1))

    # gradient descent
    for i in range(n_iters):
        grad_huber_beta = grad_huber(beta)
        grad_huber_beta[0] = 0
        grad = XX @ beta - Xy + lambd * grad_huber_beta
        # gradient descent update
        beta -= step_size * grad

    return beta
```