Linear Regression

CSE474/574 Introduction to Machine Learning

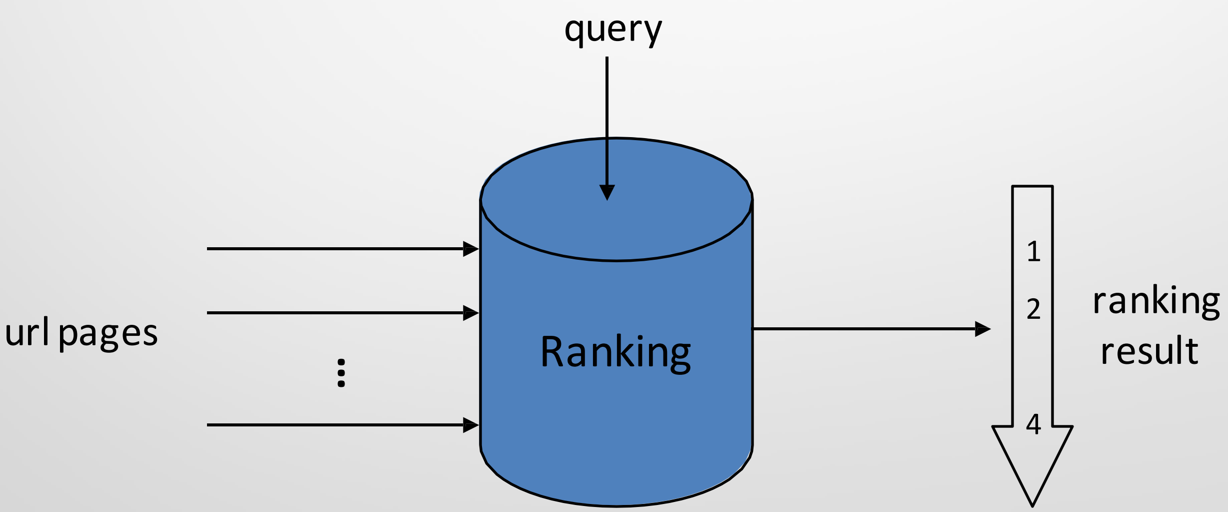
Jun Chu

jchu6@buffalo.edu

Letor4.0 Dataset

Goal: given queries and a documents/urls, estimate the Web search results (relevance) of the pages to the queries.

Ranking the pages via a relevance function.



Letor4.0 Dataset

- LETOR is a package of benchmark data sets for research on Learning Rank released by Microsoft Research Asia.
- The latest version, 4.0, can be found at http://research.microsoft.com/en-us/um/beijing/projects/letor/letor4dataset.aspx (It contains 8 datasets for four ranking settings derived from the two query sets and the Gov2 web page collection.)
- For this project, one dataset of MQ2007 is used (supervised ranking):
    - Querylevelnorm_t.csv
    - Querylevelnorm_X.csv

Synthetic Dataset

- **Procedurally generated:** generate synthesized data using some sort of mathematical formula
  - input.csv
  - output.csv

$$y = f(\mathbf{x}) + \varepsilon$$

f: a deterministic function

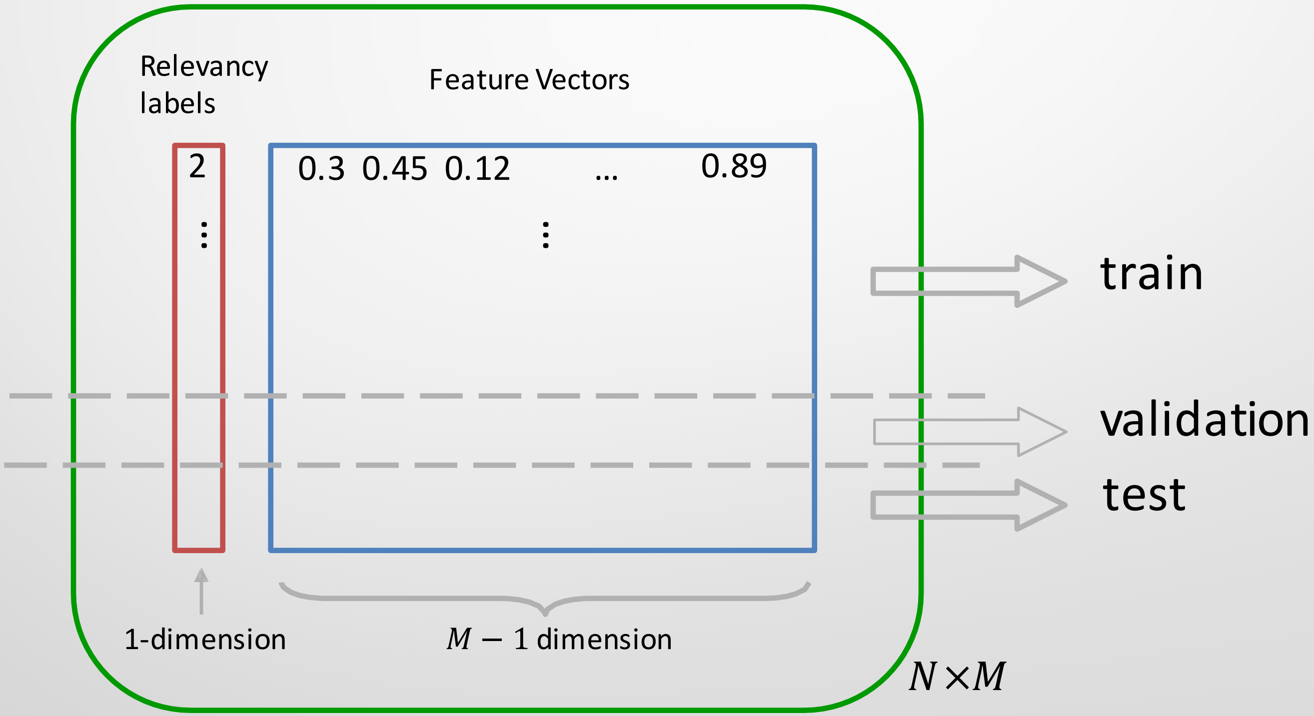
Ɛ: random noise

Import Data Set

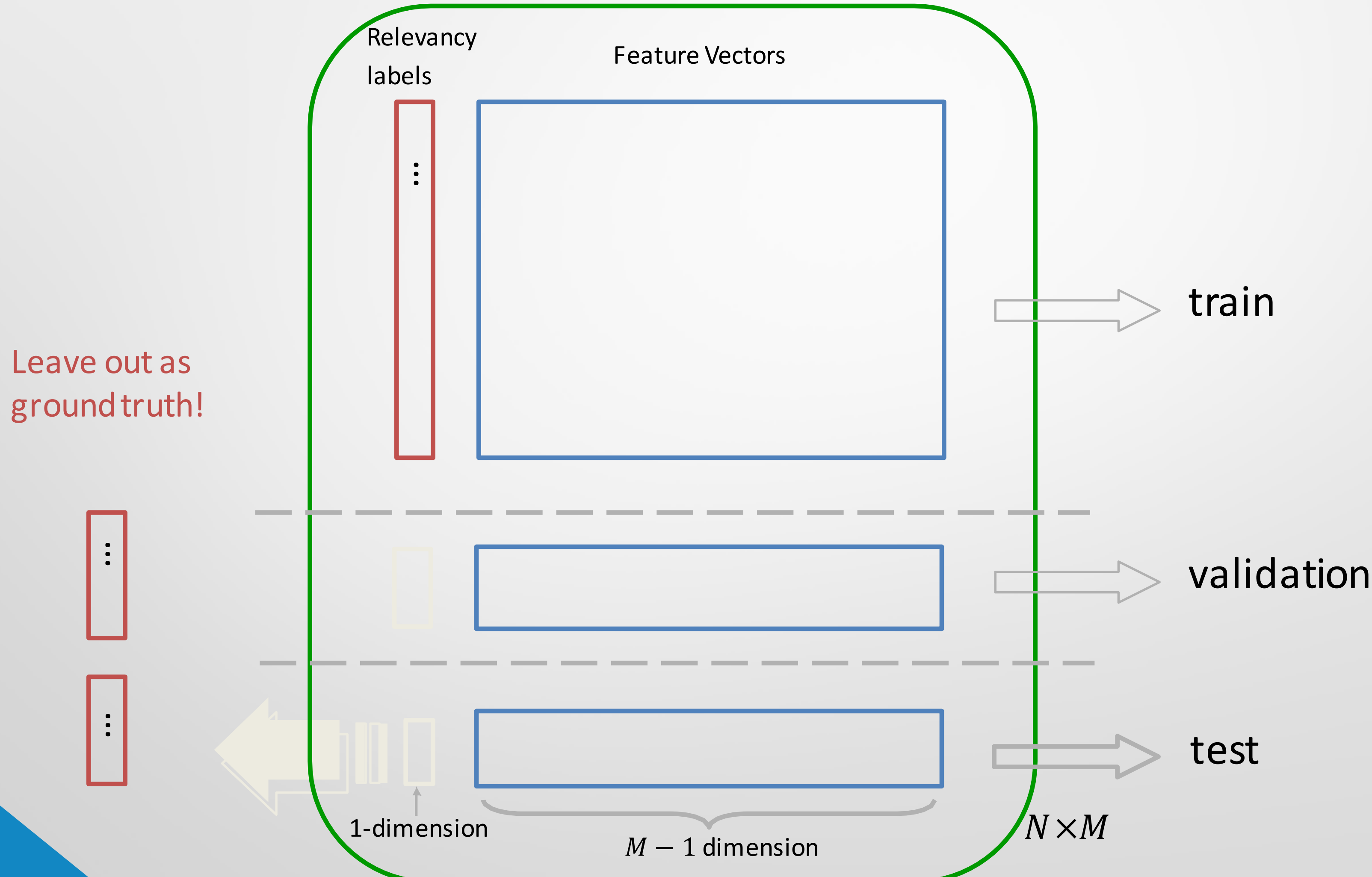
Use numpy function: genfromtxt

```
import numpy as np
syn_input_data = np.genfromtxt('datafiles/input.csv', delimiter=',')
syn_output_data = np.genfromtxt(
    'datafiles/output.csv', delimiter=',').reshape([-1, 1])
letor_input_data = np.genfromtxt(
    'datafiles/Querylevelnorm_X.csv', delimiter=',')
letor_output_data = np.genfromtxt(
    'datafiles/Querylevelnorm_t.csv', delimiter=',').reshape([-1, 1])
```

Partition of the datasets



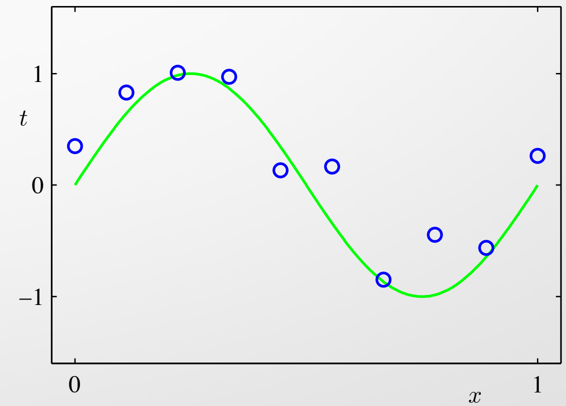
Train/Validation/Test Sets



Linear Regression

Problem: We want a general way of obtaining a linear model (model is linear in the parameters) that fits to observed data.

General set up:
Given a set of training examples $(\boldsymbol{x}_n, t_n), n = 1, \dots N$
Goal: learn a function $y(x)$ to minimize some
loss function (error function): $E(y, t)$



Linear Basis function Model:

$$y(\mathbf{x}, \mathbf{w}) = \mathbf{w}^\top \boldsymbol{\phi}(\mathbf{x}) \qquad \boldsymbol{\phi}(\mathbf{x}) = [\phi_0(\mathbf{x}), \phi_1(\mathbf{x}), .., \phi_{M-1}(\mathbf{x})]^\top, \quad \phi_0(\mathbf{x}) \equiv 1$$

Different Gaussian parameter settings

w: M dimension weight vector

Linear Regression for Project

Project Goal: To predict the value of one or more continuous target variables t given the value of a D-dimensional vector x of input variables.

Gaussian Basis Function $$\phi_j(\mathbf{x}) = \exp\left(-\frac{1}{2}(\mathbf{x}-\boldsymbol{\mu}_j)^\top \Sigma_j^{-1}(\mathbf{x}-\boldsymbol{\mu}_j)\right)$$

Each basis function takes in one data points (D elements) and gives out one scalar

μ_j: centers
Σ_j: spreads

Compute the design matrix

$$\mathbf{\Phi} = \begin{bmatrix} \phi_0(\mathbf{x}_1) & \phi_1(\mathbf{x}_1) & \phi_2(\mathbf{x}_1) & \cdots & \phi_{M-1}(\mathbf{x}_1) \\ \phi_0(\mathbf{x}_2) & \phi_1(\mathbf{x}_2) & \phi_2(\mathbf{x}_2) & \cdots & \phi_{M-1}(\mathbf{x}_2) \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ \phi_0(\mathbf{x}_N) & \phi_1(\mathbf{x}_N) & \phi_2(\mathbf{x}_N) & \cdots & \phi_{M-1}(\mathbf{x}_N) \end{bmatrix}$$

a single data

N x M design matrix

a basis function

Compute the design matrix

Computing one entry is straight forward.

$$\phi_j(\mathbf{x}) = \exp\left(-\frac{1}{2}(\mathbf{x} - \boldsymbol{\mu}_j)^\top \Sigma_j^{-1} (\mathbf{x} - \boldsymbol{\mu}_j)\right)$$

How to vectorize the process to avoid looping and make computation more efficient?

Start from computing one col of the design matrix at a time

Compute the design matrix

Assume we stack all data into a 2-D matrix, each row corresponding to 1 data point:

$$X = \begin{bmatrix} \mathbf{x}_1^\top \\ \mathbf{x}_2^\top \\ \vdots \\ \mathbf{x}_N^\top \end{bmatrix}_{N\times D} \qquad Y_j = \begin{bmatrix} (\mathbf{x}_1 - \boldsymbol{\mu}_j)^\top \\ (\mathbf{x}_2 - \boldsymbol{\mu}_j)^\top \\ \vdots \\ (\mathbf{x}_N - \boldsymbol{\mu}_j)^\top \end{bmatrix}_{N\times D}$$

Use matrix multiplication:

$$Y_j \Sigma_j^{-1} = \begin{bmatrix} (\mathbf{x}_1 - \boldsymbol{\mu}_j)^\top \Sigma_j^{-1} \\ (\mathbf{x}_2 - \boldsymbol{\mu}_j)^\top \Sigma_j^{-1} \\ \vdots \\ (\mathbf{x}_N - \boldsymbol{\mu}_j)^\top \Sigma_j^{-1} \end{bmatrix}_{N\times D}$$

Perform row-wise dot product. This could be done using element-wise product and adding along the row dim:

$$Y_j \Sigma_j^{-1} \circledast Y_j = \begin{bmatrix} (\mathbf{x}_1 - \boldsymbol{\mu}_j)^\top \Sigma_j^{-1} (\mathbf{x}_1 - \boldsymbol{\mu}_j) \\ (\mathbf{x}_2 - \boldsymbol{\mu}_j)^\top \Sigma_j^{-1} (\mathbf{x}_2 - \boldsymbol{\mu}_j) \\ \vdots \\ (\mathbf{x}_N - \boldsymbol{\mu}_j)^\top \Sigma_j^{-1} (\mathbf{x}_N - \boldsymbol{\mu}_j) \end{bmatrix}_{N\times 1} = \begin{bmatrix} \phi_j(\mathbf{x}_1) \\ \phi_j(\mathbf{x}_2) \\ \vdots \\ \phi_j(\mathbf{x}_N) \end{bmatrix}_{N\times 1}$$

Compute the design matrix

Stack all centers and spreads together and use broadcast to compute the design matrix in one statement:

```
def compute_design_matrix(X, centers, spreads):
    # use broadcast
    basis_func_outputs = np.exp(
        np.sum(
            np.matmul(X - centers, spreads) * (X - centers),
            axis=2
        ) / (-2)
    ).T
    # insert ones to the 1st col
    return np.insert(basis_func_outputs, 0, 1, axis=1)
```

Compute closed-form solution

Least squares solution:

$$\mathbf{w}_{\mathrm{ML}} = (\mathbf{\Phi}^\top \mathbf{\Phi})^{-1} \mathbf{\Phi}^\top \mathbf{t}$$

Least squares solution with weight decay:

$$\mathbf{w}_{\mathrm{ML}} = (\lambda \mathbf{I} + \mathbf{\Phi}^\top \mathbf{\Phi})^{-1} \mathbf{\Phi}^\top \mathbf{t}$$

Compute closed-form solution

```
def closed_form_sol(L2_lambda, design_matrix, output_data):
    return np.linalg.solve(
        L2_lambda * np.identity(design_matrix.shape[1]) +
            np.matmul(design_matrix.T, design_matrix),
        np.matmul(design_matrix.T, output_data)
    ).flatten()
```

Compute gradient descent solution

It is just another way of computing **w**, anything else remains the same.

General Form:

$$\mathbf{w}^{(\tau+1)} = \mathbf{w}^{(\tau)} + \Delta\mathbf{w}^{(\tau)}$$

$$\Delta\mathbf{w}^{(\tau)} = -\eta^{(\tau)}\nabla E$$

$$\nabla E = \nabla E_D + \lambda\nabla E_W$$

$$\nabla E_D = -(t_n - \mathbf{w}^{(\tau)\top}\boldsymbol{\phi}(\mathbf{x}_n))\boldsymbol{\phi}(\mathbf{x}_n), \quad \nabla E_W = \mathbf{w}^{(\tau)}$$

$\phi(\mathbf{x}_n)$ is one row of the design matrix

Φ_n : 1 x M vector

Learning Rate η:

Start with $\eta = 1$, check if it converges.

Compute gradient descent solution

Stopping Criteria:

The error decreasing is very small between iterations.

Initialization:

Since the optimization is convex, we can start from anywhere

Mini-batch SGD:

In reality, we accumulate a bunch of ∇Es before updating **w**

Compute gradient descent solution

```
def SGD_sol(learning_rate,
            minibatch_size,
            num_epochs,
            L2_lambda,
            design_matrix,
            output_data):
    N, _ = design_matrix.shape
    # You can try different mini-batch size size
    # Using minibatch_size = N is equivalent to standard gradient descent
    # Using minibatch_size = 1 is equivalent to stochastic gradient descent
    # In this case, minibatch_size = N is better
    weights = np.zeros([1, 4])
    # The more epochs the higher training accuracy. When set to 1000000,
    # weights will be very close to closed_form_weights. But this is unnecessary
```

Compute gradient descent solution

```
for epoch in range(num_epochs):
    for i in range(N / minibatch_size):
        lower_bound = i * minibatch_size
        upper_bound = min((i+1)*minibatch_size, N)
        Phi = design_matrix[lower_bound : upper_bound, :]
        t = output_data[lower_bound : upper_bound, :]
        E_D = np.matmul(
            (np.matmul(Phi, weights.T)-t).T,
            Phi
        )
        E = (E_D + L2_lambda * weights) / minibatch_size
        weights = weights - learning_rate * E
    print np.linalg.norm(E)
return weights.flatten()
```

Randomly pick up 3 basis functions

```
N, D = input_data.shape
# Assume we use 3 Gaussian basis functions M = 3
# shape = [M, 1, D]
centers = np.array([np.ones((D))*1, np.ones((D))*0.5, np.ones((D))*1.5])
centers = centers[:, np.newaxis, :]
# shape = [M, D, D]
spreads = np.array([np.identity(D), np.identity(D), np.identity(D)]) * 0.5
# shape = [1, N, D]
X = input_data[np.newaxis, :, :]
design_matrix = compute_design_matrix(X, centers, spreads)
```

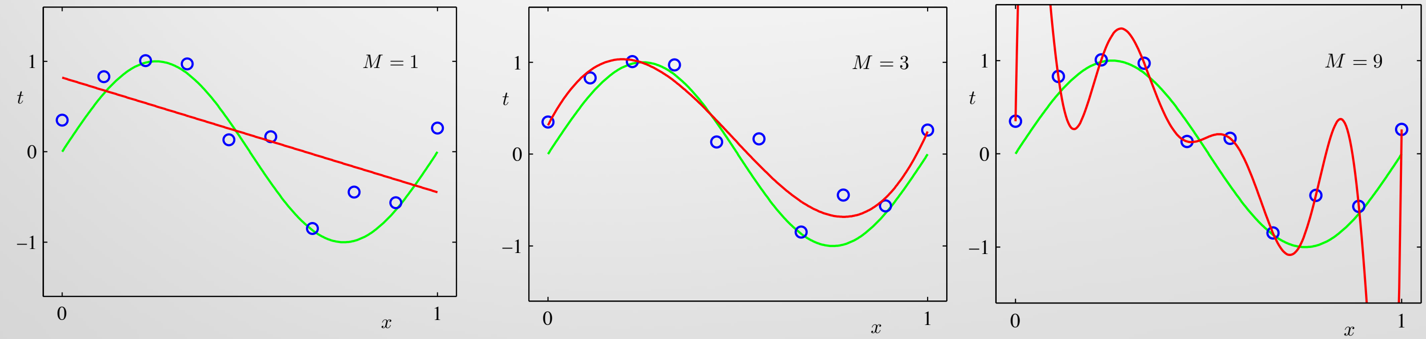
Print out the solutions

```
# Closed-form solution
print closed_form_sol(L2_lambda=0.1,
                      design_matrix=design_matrix,
                      output_data=output_data)
# Gradient descent solution
print SGD_sol(learning_rate=1,
              minibatch_size=N,
              num_epochs=10000,
              L2_lambda=0.1,
              design_matrix=design_matrix,
              output_data=output_data)
```

Overfitting Issue

What can we do to curb overfitting?

- Use less complex model
- Use more training examples
- Regularization



Experimental Phases

Training

Determine format of your model

Train the model you have selected

learn weights **w**

Model

Validation

Adjusting following:

# of basis func. **M**
Regularization **λ**
Hyperparameter **μ,s**

etc.

Model with tuned parameters

Test

Evaluating the final model

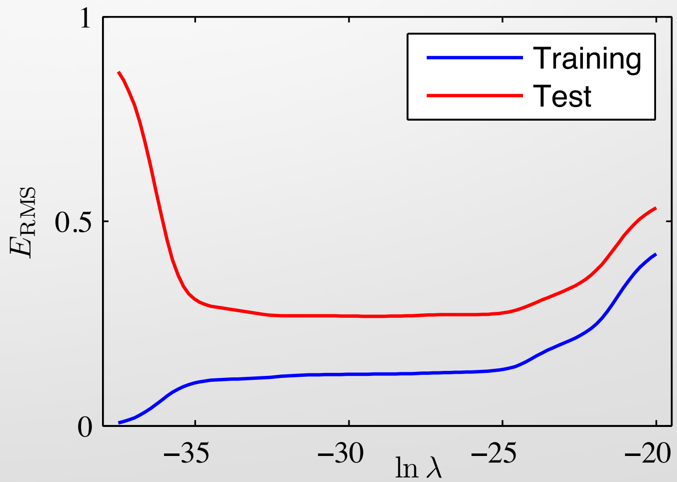
Report test error

Unacceptable **validation error**

Evaluation Metrics

Express results as Root Mean Square Error: ***E_{RMS}***

$$E_{RMS}(\mathbf{w}) = \sqrt{\frac{2E_D(\mathbf{w})}{N}}$$



N: number of data in data set

E_D(w): sum of square error function
(data-dependent error)

Project Report

- Explain the problem and how you choose your model.

- Elaborate your validating process.

  - The intuitive choice of parameters)

    There are no limitation on setting parameters and there could be infinity choices. You can define some range or choose some specific values.

  - Description of how you went about avoiding overfitting.

- Generate graphs showing how error changes with the adjusting of parameters.

- Report final result and evaluating model performance.