**9.2    MNIST classification: pixels versus edge-based features**

Repeat the experiment outlined in Example 9.3 and create a pair of cost function/misclassification history plots like the ones shown in Figure 9.11. Your results may vary slightly from those reported in the example depending on the details of your implementation.

**Example 9.3    Handwritten digit recognition**

In this example we look at the problem of handwritten digit recognition (introduced in Example 1.10), and compare the training effectiveness of mini-batch gradient descent (20 steps/epochs with a learning rate of $\alpha = 10^{-2}$ and batch size of 200 applied to a multi-class Softmax cost) using $P = 50,000$ raw (pixel-based) data points from the MNIST handwritten digit recognition dataset (introduced in Example 7.10), to the effectiveness of precisely the same setup applied to edge histogram based features extracted from these same data points.
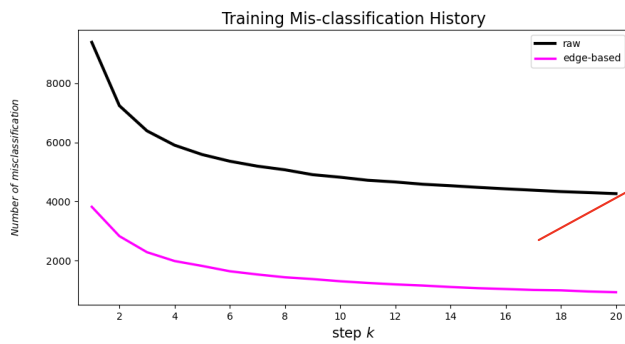
cost function:   multiclass_softmax

study rate: 0·01      train for 20 iterations

batch size: 200      P = 50000

Training Cost History
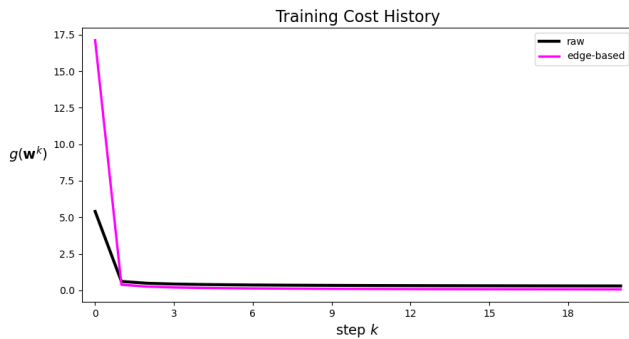
both kinds of feature have similar converge speed

The edge-based feature achieves better result in classification (About 4000 less misclassifications)

Training Mis-classification History

← x axis limit ⇒ iteration 1 to 20

## ● 9-2 MNIST classification

```python
import sys
from mlrefined_libraries.math_optimization_library import static_plotter
import autograd.numpy as np
from autograd.misc.flatten import flatten_func
from autograd import grad as gradient
import edge_extract
from sklearn.datasets import fetch_openml


sys.path.append('..')


plotter = static_plotter.Visualizer()


def linear_model(x, w):
    a = w[0] + np.dot(x.T, w[1:])
    return a.T


def multiclass_softmax(w, x, y, iter):
    x_p = x[:, iter]
    y_p = y[:, iter]
    all_evals = linear_model(x_p, w)
    a = np.log(np.sum(np.exp(all_evals), axis=0))
    b = all_evals[y_p.astype(int).flatten(), np.arange(np.size(y_p))]
    cost = np.sum(a - b)
    return cost / float(np.size(y_p))


class MNIST_Classification(object):
    def __init__(self, n_sample):
        x, y = fetch_openml('mnist_784', version=1, return_X_y=True)
        self.x = x.T
        self.y = np.array([int(v) for v in y])[np.newaxis, :]
        self.shuffle_data(n_sample, self.x, self.y)
        self.standard_normalizer(self.x_rand.T)
        self.x_rand = self.normalizer(self.x_rand.T).T
        self.x_edge = self.hog_extractor(self.x_rand)
        self.cost_function = multiclass_softmax
        self.mismatching_his = None


    def shuffle_data(self, n_sample, x, y):
        inds = np.random.permutation(y.shape[1])[:n_sample]
        self.x_rand = np.array(x)[:, inds]
        self.y_rand = y[:, inds]
```

```python
    def standard_normalizer(self, x):
        x_ave = np.nanmean(x, axis=1)[:, np.newaxis]
        x_std = np.nanstd(x, axis=1)[:, np.newaxis]
        self.normalizer = lambda data: (data - x_ave) / x_std


    @staticmethod
    def hog_extractor(x):
        kernels = np.array([[[-1, -1, -1], [0, 0, 0], [1, 1, 1]], [[-1, -1, 0],[-1, 0, 1], [0, 1, 1]],
        [[-1, 0, 1], [-1, 0, 1], [-1, 0, 1]], [[0, 1, 1],[-1, 0, 1], [-1, -1, 0]],
        [[1, 0, -1], [1, 0, -1], [1, 0, -1]], [[0, -1, -1],[1, 0, -1], [1, 1, 0]],
        [[1, 1, 1], [0, 0, 0], [-1, -1, -1]], [[1, 1, 0],[1, 0, -1], [0, -1, -1]]])

        extractor = edge_extract.tensor_conv_layer()
        x_transformed = extractor.conv_layer(x.T, kernels).T
        return x_transformed


    def gradient_descent(self, loss_fun, w, x_train, y_train, alpha, max_its, batch_size):
        g_flat, unflatten, w = flatten_func(loss_fun, w)
        grad = gradient(g_flat)
        num_train = y_train.size
        w_hist = [unflatten(w)]
        train_hist = [g_flat(w, x_train, y_train, np.arange(num_train))]
        num_batches = int(np.ceil(np.divide(num_train, batch_size)))
        for k in range(max_its):
            for b in range(num_batches):
                batch_inds = np.arange(b * batch_size, min((b + 1) * batch_size, num_train))
                grad_eval = grad(w, x_train, y_train, batch_inds)
                grad_eval.shape = np.shape(w)
                w = w - alpha * grad_eval
            train_cost = g_flat(w, x_train, y_train, np.arange(num_train))
            w_hist.append(unflatten(w))
            train_hist.append(train_cost)
        return w_hist, train_hist


    def predict(self, x, w_trained):
        class_y = np.argmax(linear_model(x, w_trained), axis=0)
        return class_y


    @staticmethod
    def misclass_cost(w, x, y):
        all_evals = linear_model(x, w)
        y_predict = (np.argmax(all_evals, axis=0))[np.newaxis, :]
        count = np.sum(np.argwhere(y != y_predict))
        return count
```

```python
if __name__ == "__main__":
    mnist = MNIST_Classification(n_sample=50000)
    N = mnist.x_rand.shape[0]
    C = len(np.unique(mnist.y_rand))
    w = 0.1 * np.random.randn(N + 1, C)
    weight_his, cost_his = mnist.gradient_descent(mnist.cost_function, w, mnist.x_rand, mnist.y_rand,
alpha=0.01,
                                                  max_its=20, batch_size=200)
    N = mnist.x_edge.shape[0]
    w = 0.1 * np.random.randn(N + 1, C)
    weight_edge_his, cost_edge_his = mnist.gradient_descent(mnist.cost_function, w, mnist.x_edge,
mnist.y_rand,
                                                            alpha=0.01,
                                                            max_its=20, batch_size=200)
    mis1 = [a / 25000 for a in [mnist.misclass_cost(v, mnist.x_rand, mnist.y_rand) for v in weight_his]]
    mis2 = [a / 25000 for a in
            [mnist.misclass_cost(v, mnist.x_edge, mnist.y_rand) for v in weight_edge_his]]


    plotter.plot_mismatching_histories(histories=[mis1, mis2], start=1,
                                       labels=['raw', 'edge-based'],
                                       title="Training Mis-classification History")
    plotter.plot_cost_histories(histories=[cost_his, cost_edge_his], start=0,
                                labels=['raw', 'edge-based'],
                                title="Training Cost History")
```

## 10.4 Moore's law

Gordon Moore, co-founder of Intel corporation, predicted in a 1965 paper [47] that the number of transistors on an integrated circuit would double approximately every two years. This conjecture, referred to nowadays as Moore's law, has proven to be sufficiently accurate over the past five decades. Since the processing power of computers is directly related to the number of transistors in their CPUs, Moore's law provides a trend model to predict the computing power of future microprocessors. Figure 10.18 plots the transistor counts of several microprocessors versus the year they were released, starting from Intel 4004 in 1971 with only 2300 transistors, to Intel's Xeon E7 introduced in 2014 with more than 4.3 billion transistors.

**Figure 10.18** Figure associated with Exercise 10.4. See text for details.
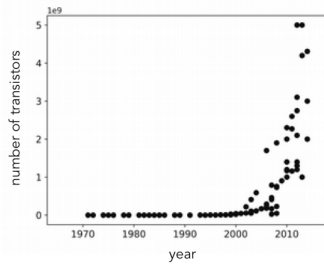


(a) Propose a single feature transformation for the Moore's law dataset shown in Figure 10.18 so that the transformed input/output data is related linearly. *Hint: to produce a linear relationship you will end up having to transform the output, not the input.*
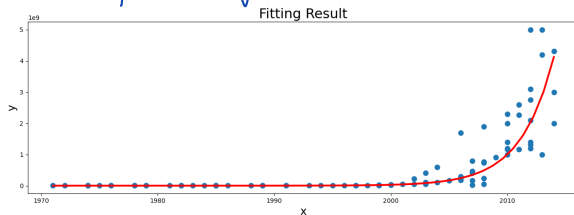
Sol: We can apply a log on y, then, log(y) and x will be linearly related.
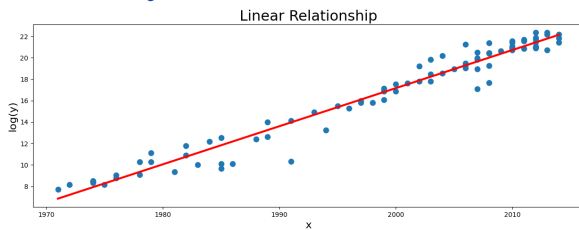


(b) Formulate and minimize a Least Squares cost function for appropriate weights, and fit your model to the data in the original data space as shown in Figure 10.18.

Sol: Below is the fitting result :

$\begin{cases} \text{Cost function: Least square} \\ \\ \text{Optimization function: Newtons method.} \end{cases}$

# ● 10-4 Moore's Law

```python
import numpy as np
import pandas as pd
import autograd.numpy as np
from autograd import hessian, grad
import matplotlib.pyplot as plt


class Moores_law(object):
    def __init__(self, filename):
        data = np.asarray(pd.read_csv(filename, header=None))
        self.x = data[:, 0]
        self.x.shape = (1, len(self.x))
        self.y = data[:, 1]
        self.y.shape = (1, len(self.y))
        self.y_log = np.log(self.y)


    def data_plotting_linear_transform(self, w):
        fig = plt.figure(figsize=(16, 5))
        ax1 = fig.add_subplot(1, 1, 1)  # panel for original space
        ax1.scatter(self.x, self.y_log, linewidth=3)
        s = np.linspace(np.min(self.x), np.max(self.x))
        t = w[0] + w[1] * s
        ax1.plot(s, t, linewidth=3, color='r')
        ax1.set_xlabel('x', fontsize=16)
        ax1.set_ylabel('log(y)', rotation=90, fontsize=16)
        ax1.set_title('Linear Relationship', fontsize=22)
        plt.show()

    @staticmethod
    def model(x, w):
        a = w[0] + np.dot(x.T, w[1:])
        return a.T


    def least_squares_mean(self, w):
        cost = np.sum((self.model(self.x, w) - self.y_log) ** 2)
        return cost / float(np.size(self.y_log))


    def newtons_method(self, g, max_its, w, **kwargs):
        gradient = grad(g)
        hess = hessian(g)
```

```python
        epsilon = 10 ** (-10)
        if 'epsilon' in kwargs:
            epsilon = kwargs['epsilon']
        weight_history = [np.array(w)]
        cost_history = [np.array(g(w))]
        for k in range(max_its):
            grad_eval = gradient(w)
            hess_eval = hess(w)
            hess_eval.shape = (int((np.size(hess_eval)) ** (0.5)), int((np.size(hess_eval)) ** (0.5)))
            A = hess_eval + epsilon * np.eye(w.size)
            b = grad_eval
            w = np.linalg.solve(A, np.dot(A, w) - b)
            weight_history.append(np.array(w))
            cost_history.append(np.array(g(w)))
        self.w = w
        return weight_history, cost_history


    def data_plotting(self, w):
        fig = plt.figure(figsize=(16, 5))
        ax1 = fig.add_subplot(1, 1, 1)  # panel for original space
        ax1.scatter(self.x, self.y, linewidth=3)
        s = np.linspace(np.min(self.x), np.max(self.x))
        t = np.exp(w[0] + w[1] * s)
        ax1.plot(s, t, linewidth=3, color='r')
        ax1.set_xlabel('x', fontsize=18)
        ax1.set_ylabel('y', rotation=90, fontsize=18)
        ax1.set_title('Fitting Result', fontsize=22)
        plt.show()



if __name__ == "__main__":
    filename = '../mlrefined_datasets/nonlinear_superlearn_datasets/transistor_counts.csv'
    w0 = np.random.randn(2, 1)
    Moores = Moores_law(filename)
    Moores.newtons_method(Moores.least_squares_mean, max_its=1000, w=w0)
    Moores.data_plotting_linear_transform(Moores.w)
    Moores.data_plotting(Moores.w)
```
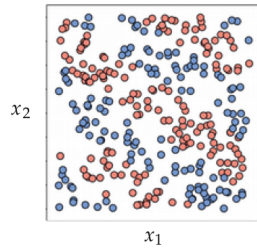
**10.8** **Engineering features for a two-class classification dataset**

Propose a nonlinear model for the dataset shown in Figure 10.20 and perform nonlinear two-class classification. Your model should be able to achieve perfect classification on this dataset.
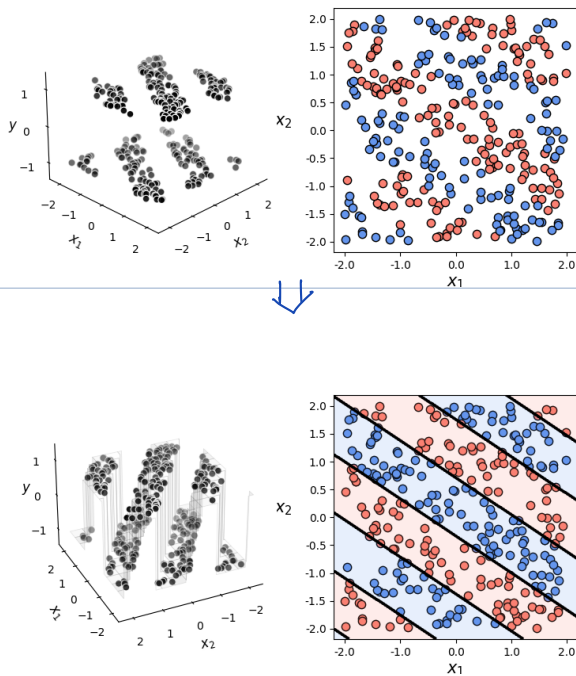
$x_2$

$x_1$

We can use "sin" to transform input data

The model is : $f(x,w) = \sin(b + \sum_{p=1}^{P} \mathring{x}_p^T w_p)$  here, we only have $f(x,y) = \sin(b + x_1 w_1 + x_2 w_2)$

trained with "softmax" study_rate = 0.1 for 2000 iteration





As the figure illustrates, our model it can achieve perfect classification

over this dataset.

# ● 10-8 Engineering features transformation

```python
from mlrefined_libraries.nonlinear_superlearn_library import nonlinear_classification_visualizer as ncv
from mlrefined_libraries.nonlinear_superlearn_library import basic_runner
import autograd.numpy as np


Visualizer = ncv.Visualizer



class Engineering_Feature_Transformation(Visualizer):
    def __init__(self, file_path):
        super().__init__(file_path)
        self.decent_initializer(2)


    def decent_initializer(self, scale):
        self.w0 = [scale * np.random.randn(3, 1), scale * np.random.randn(2, 1)]


    @staticmethod
    def feature_transforms(x, w):
        f = np.sin(w[0] + np.dot(x.T, w[1:])).T
        return f


    def train(self, loss_fun, study_rate, iters, normalize):
        model = basic_runner.Setup(self.x.T, self.y, self.feature_transforms, loss_fun,
normalize=normalize)
        model.fit(w=self.w0, alpha_choice=study_rate, max_its=iters)
        ind = np.argmin(model.cost_history)
        self.w_best = model.weight_history[ind]
        self.model = model


    def visulizer(self):
        self.static_N2_simple(self.w_best, self.model, view=[30, 160])



if __name__ == "__main__":
    file_path = '../mlrefined_datasets/nonlinear_superlearn_datasets/diagonal_stripes.csv'
    Eng_trans = Engineering_Feature_Transformation(file_path)
    Eng_trans.plot_data()
    Eng_trans.train(loss_fun='softmax', study_rate=0.1, iters=2000)
    Eng_trans.visulizer()
```

Repeat the first experiment outlined in Example 11.15, producing five naively cross-validated polynomial models to fit different training–validation splits of the two-class classification dataset shown in Figure 11.46. Compare the efficacy – in terms of number of misclassifications over the entire dataset – of each individual model and the final bagged model.

train-set portion: $\frac{2}{3}$    validation portion: $\frac{1}{3}$

5 bags  , degree of polynomial: 1-8

model trained for 50 iterations for each degree

Optimized with Newtons method.
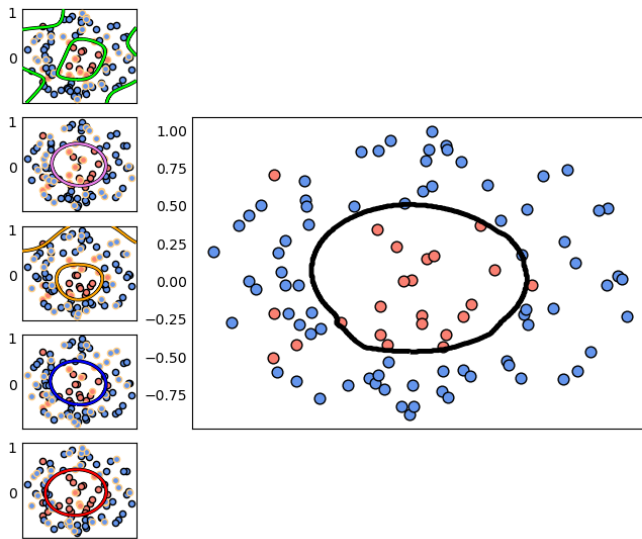


each one of the subplot corresponds to a individual model, and most of them are already overfitting.

By contrast, the final model (mode of 5 individual models) does not appear obvious overfitting

**Example 11.15    Bagging cross-validated two-class classification models**

In the set of small panels in the left column of Figure 11.46 we show five different training–validation splits of the prototypical two-class classification dataset first described in Example 11.7, where $\frac{2}{3}$ of the data in each instance is used for training and $\frac{1}{3}$ is used for validation (the boundaries of these points are colored yellow). Plotted with each split of the original data is the nonlinear decision boundary corresponding to each cross-validated model found via naive cross-validation of the full range of polynomial models of degree 1 to 8. Many of these cross-validated models perform quite well, but some of them (due to the particular training–validation split on which they are based) severely *overfit* the original dataset. By bagging these models using the most popular prediction to assign labels (i.e., the *mode* of these cross-validated model predictions) we produce an appropriate decision boundary for the data shown in the right panel of the figure.
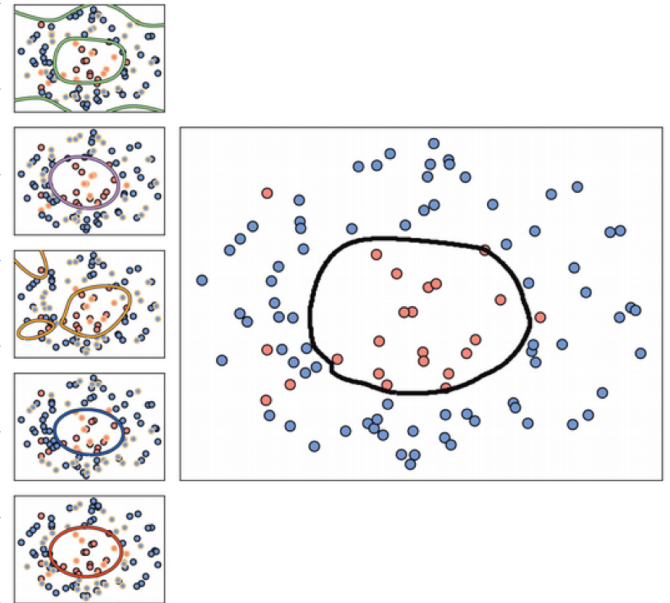


**Figure 11.46**  Figure associated with Example 11.15. (left column) Five models cross-validated on random training–validation splits of the data, with the validation data in each instance highlighted with a yellow outline. The corresponding nonlinear decision boundary provided by each model is shown in each panel. Some models, due to the split of the data on which they were built, severely overfit. (right column) The original dataset with the decision boundary provided by the bag (i.e., mode) of the five cross-validated models. See text for further details.

# ● 11-7 Bagging two-class classification models

```python
import copy
import autograd.numpy as np
from mlrefined_libraries.nonlinear_superlearn_library.classification_bagging_visualizers_v2 import
Visualizer
from mlrefined_libraries.nonlinear_superlearn_library.reg_lib.super_setup import Setup


class Bagging_Two_Class_Classification(Visualizer):
    def __init__(self, file_path):
        data = np.loadtxt(file_path, delimiter=',')
        Visualizer.__init__(self, file_path)
        self.x = data[:-1, :]
        self.y = data[-1:, :]
        self.visulizer = Visualizer(file_path)

    def split_dataset(self, train_portion):
        self.train_portion = train_portion
        r = np.random.permutation(self.x.shape[1])
        train_num = int(np.round(train_portion * len(r)))
        self.train_inds = r[:train_num]
        self.valid_inds = r[train_num:]

        self.x_train = self.x[:, self.train_inds]
        self.x_valid = self.x[:, self.valid_inds]

        self.y_train = self.y[:, self.train_inds]
        self.y_valid = self.y[:, self.valid_inds]

    def train(self, num_bag, train_portion, degree):
        model = []
        for j in range(num_bag):
            model_11_7 = Setup(self.x, self.y)
            model_11_7.preprocessing_steps(name="standard")
            model_11_7.split_dataset(train_portion)
            for d in range(1, degree + 1):
                model_11_7.choose_cost(name='softmax')
                model_11_7.choose_features(name='polys', degree=d)
                model_11_7.fit(algo='newtons_method', max_its=50, verbose=False, lam=10 ** (-8))
            val_costs = [np.min(model_11_7.valid_count_histories[i]) for i in range(degree)]
            min_ind = np.argmin(val_costs)
            min_val = val_costs[min_ind]
```

```python
            # get minor of minor
            smallest_ind = np.argmin(model_11_7.valid_count_histories[min_ind])
            model_11_7.train_cost_histories = model_11_7.train_cost_histories[min_ind][smallest_ind]
            model_11_7.valid_cost_histories = model_11_7.valid_cost_histories[min_ind][smallest_ind]
            model_11_7.train_count_histories = model_11_7.train_count_histories[min_ind][smallest_ind]
            model_11_7.valid_count_histories = model_11_7.valid_count_histories[min_ind][smallest_ind]
            model_11_7.weight_histories = model_11_7.weight_histories[min_ind][smallest_ind]
            model_11_7.choose_features(name='polys', degree=min_ind + 1)

            # store
            model.append(copy.deepcopy(model_11_7))
        return model


    def visulization(self, model):
        self.visulizer.show_runs(model)



if __name__ == "__main__":
    file_path = '../mlrefined_datasets/nonlinear_superlearn_datasets/new_circle_data.csv'
    bagging8 = Bagging_Two_Class_Classification(file_path=file_path)
    trained_model = bagging8.train(num_bag=5, train_portion=0.67, degree=8)
    bagging8.show_runs(trained_model)
```

**11.10    Classification of diabetes**

Perform K-fold cross-validation using a linear model and the $\ell_1$ regularizer over a popular two-class classification genomics dataset consisting of $P = 72$ data-points, each of which has input dimension $N = 7128$. This will tend to produce a sparse predictive linear model – as detailed in Example 11.18 – which is helpful

in determining a small number of genes that correlate with the output of this two-class classification dataset (which is whether each individual represented in the dataset has diabetes or not).

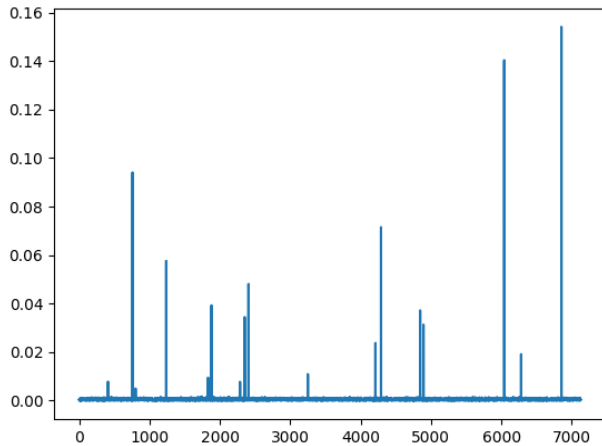**Example 11.18    Genome-wide association studies**

Genome-wide association studies (GWAS) aim at understanding the connections between tens of thousands of genetic markers (input features), taken from across the human genome of several subjects, with medical conditions such as high blood pressure, high cholesterol, heart disease, diabetes, various forms of cancer, and many others (see Figure 11.52). These studies typically involve a relatively small number of patients with a given affliction (as compared to the very large dimension of the input). As a result, regularization based cross-validation is a useful tool for learning meaningful (linear) models for such data. Moreover, using a (sparsity-inducing) regularizer like the $\ell_1$ norm can help researchers identify the handful of genes critical to the affliction under study, which can both improve our understanding of it and perhaps provoke development of gene-targeted therapies. See Exercise 11.10 for further details.

$L_1:$ regularization

$$C = C_0 + \frac{\lambda}{n} \sum_\omega |\omega|$$

① chose best $\lambda$ for the regularization

10-folds validation     study rate: $\frac{1}{k}$ (K ⇒ iteration)

Best $\lambda$ for $L_1$ regularization $\approx 12.245$

best lambda is:12.244897959183675

# ● 11-10 Classification of diabetes

```python
import copy
from autograd import value_and_grad
from autograd.misc.flatten import flatten_func
import autograd.numpy as np
import matplotlib.pyplot as plt
from IPython.display import clear_output
from mlrefined_libraries.nonlinear_superlearn_library.kfolds_reg_lib.superlearn_setup import Setup


class Diabetes_Classification(Setup):
    def __init__(self, file_path, K_fold):
        data = np.loadtxt(file_path, delimiter=',')
        self.x = data[:-1, :]
        self.y = data[-1:, :]
        super().__init__(self.x, self.y)
        self.fold_num = self.fold_assigning(self.y.size, K_fold)
        self.x_mean = np.nanmean(self.x, axis=1)[:, np.newaxis]
        self.x_std = np.nanstd(self.x, axis=1)[:, np.newaxis]
        self.K_fold = K_fold
        self.lam = None


    def decent_ini(self):
        self.w_0 = 0.1 * np.random.randn(self.x.shape[0] + 1, 1)


    @staticmethod
    def fold_assigning(L, K):
        order = np.random.permutation(L)
        c = np.ones((L, 1))
        L = int(np.round((1 / K) * L))
        for s in np.arange(0, K - 2):
            c[order[s * L:(s + 1) * L]] = s + 2
        c[order[(K - 1) * L:]] = K
        return c


    def data_initialization(self):
        self.deviation_regulartor(self.x)
        x = self.data_recovery(self.x)
        self.x_mean = np.nanmean(x, axis=1)[:, np.newaxis]
        self.data_normalization(x)


    def deviation_regulartor(self, x):
```

```python
        self.x_std = np.nanstd(x, axis=1)[:, np.newaxis]
        regulator = np.zeros(self.x_std.shape)
        for i in range(len(self.x_std)):
            if self.x_std[i] <= 0.01:
                regulator[i] = 1.0
                self.x_std += regulator
            else:
                pass


    def data_normalization(self, x):
        # Generate the normalization function
        normalize = lambda x: (x - self.x_mean) / self.x_std
        self.x = normalize(x)


    def data_recovery(self, x):
        mean = np.nanmean(self.x, axis=1)
        for i in np.argwhere(np.isnan(x)):
            x[i[0], i[1]] = mean[i[0]]
        return x


    @staticmethod
    def linear_model(x, w):
        a = w[0] + np.dot(x.T, w[1:])
        return a.T


    def make_train_test_split(self, k):
        train_ind = [v[0] for v in np.argwhere(self.fold_num != k)]
        valid_ind = [v[0] for v in np.argwhere(self.fold_num == k)]
        self.train_x = self.x[:, train_ind]
        self.train_y = self.y[:, train_ind]
        self.valid_x = self.x[:, valid_ind]
        self.valid_y = self.y[:, valid_ind]


    def gradient_descent(self, g, x, y, alpha_choice, max_its, batch_size):
        w = self.w_0
        # flatten the input function, create gradient based on flat function
        g_flat, unflatten, w = flatten_func(g, w)
        grad = value_and_grad(g_flat)
        num_train = y.size
        w_hist = [unflatten(w)]
        train_hist = [g_flat(w, x, y, np.arange(num_train))]
        num_batches = int(np.ceil(np.divide(num_train, batch_size)))
        alpha = 0
```

```python
        for k in range(max_its):
            if alpha_choice == 'diminishing':
                alpha = 1 / float(k + 1)
            else:
                alpha = alpha_choice

            for b in range(num_batches):
                batch_inds = np.arange(b * batch_size, min((b + 1) * batch_size, num_train))
                cost_eval, grad_eval = grad(w, x, y, batch_inds)
                grad_eval.shape = np.shape(w)
                w = w - alpha * grad_eval

            train_cost = g_flat(w, x, y, np.arange(num_train))
            w_hist.append(unflatten(w))
            train_hist.append(train_cost)
        return w_hist, train_hist


    def softmax(self, w, x, y, iter):
        x_p = x[:, iter]
        y_p = y[:, iter]
        cost = np.sum(np.log(1 + np.exp(-y_p * (self.linear_model(x_p, w)))))
        cost += (self.lam * np.sum(np.abs(w[1:])))
        return cost / float(np.size(y_p))


    def counting_mis_classification(self, w, x, y):
        y_predict = np.sign(self.linear_model(x, w))
        num_misclass = len(np.argwhere(y != y_predict))
        return num_misclass


    def train(self, lams, max_its, study_rate):
        all_train_counts = []
        all_valid_counts = []
        for k in range(self.K_fold):
            print("---------fold" + str(k + 1) + "------")
            print("---------****---------")
            # self.data_normalization(self.x)
            self.choose_normalizer(name='standard')
            self.make_train_test_split(k)
            self.find_best_reg(lams, max_its, study_rate)
            all_train_counts.append(copy.deepcopy(self.train_count_misclass))
            all_valid_counts.append(copy.deepcopy(self.valid_count_misclass))
        print("finish")
        best_lam = self.count_mis_class_total(all_train_counts, all_valid_counts)
        return best_lam
```

```python
    def single_trail(self, lams, max_its, study_rate):
        self.data_normalization(self.x)
        train_inds = np.argwhere(self.fold_num != -1)
        train_inds = [v[0] for v in train_inds]
        valid_inds = np.argwhere(self.fold_num == -1)
        valid_inds = [v[0] for v in valid_inds]
        self.valid_x = self.x[:, valid_inds]
        self.valid_y = self.y[:, valid_inds]
        self.train_x = self.x[:, train_inds]
        self.train_y = self.y[:, train_inds]
        self.find_best_reg(lams, max_its, study_rate)


    def find_best_reg(self, lams, max_its, study_rate):
        self.train_count_misclass = []
        self.valid_count_misclass = []
        self.weights = []
        batch_size = np.size(self.train_y)
        self.decent_ini()
        for i in range(len(lams)):
            print('running ' + str(i + 1) + ' of ' + str(len(lams)) + ' rounds')
            self.lam = lams[i]
            weight_history, cost_history = self.gradient_descent(self.softmax, self.train_x,
                                                self.train_y, alpha_choice=study_rate,
                                                max_its=max_its, batch_size=batch_size)
            w_p = weight_history[np.argmin(cost_history)]
            self.weights.append(w_p)
            self.train_count_misclass.append(self.counting_mis_classification(w_p, self.train_x,
self.train_y))
            self.valid_count_misclass.append(self.counting_mis_classification(w_p, self.valid_x,
self.valid_y))
        bset_ind = np.argmin(self.valid_count_misclass)
        self.best_lam = lams[bset_ind]
        self.best_weights = self.weights[bset_ind]


    @staticmethod
    def count_mis_class_total(all_train_counts, all_valid_counts):
        all_train_counts = np.array(all_train_counts)
        train_totals = np.sum(all_train_counts, 0)
        all_valid_counts = np.array(all_valid_counts)
        valid_totals = np.sum(all_valid_counts, 0)
        best_valid_ind = np.where(valid_totals == valid_totals.min())[0][-1]
        best_lam = lamda[best_valid_ind]
        return best_lam
```

```python
if __name__ == "__main__":
    k_fold = 10
    lamda = np.linspace(0, 20, 50)
    file_path = '../mlrefined_datasets/nonlinear_superlearn_datasets/new_gene_data.csv'
    diabetes = Diabetes_Classification(file_path=file_path, K_fold=k_fold)
    lamda_chose = diabetes.train(lamda, max_its=100, study_rate='diminishing')
    print("best lambda is:" + str(lamda_chose))
    clear_output()
    diabetes2 = Diabetes_Classification(file_path=file_path, K_fold=k_fold)
    diabetes2.single_trail(lams=np.array([lamda_chose]), max_its=200, study_rate='diminishing')
    plt.plot(np.abs(diabetes2.best_weights[1:]))
    plt.show()
```

```python
if __name__ == "__main__":
    k_fold = 10
    lamda = np.linspace(0, 20, 50)
    file_path = '../mlrefined_datasets/nonlinear_superlearn_datasets/new_gene_data.csv'
    diabetes = Diabetes_Classification(file_path=file_path, K_fold=k_fold)
```