6.5 Confirm gradient and Hessian calculations

Confirm that the gradient and Hessian of the Cross Entropy cost are as shown in Section 6.2.7.

Sol: $\nabla \sigma(x) = \left(\frac{1}{1+e^{-x}}\right)' = \frac{e^{-x}}{(1-e^{-x})^2} = \frac{1}{1+e^{-x}} \times \left(1 - \frac{1}{1+e^{-x}}\right)$

$$g_p(w) = \begin{cases} -\log(\sigma(x_p^T w)) & \text{if } y_p = 1 \\ -\log(1-\sigma(x_p^T w)) & \text{if } y_p = 0 \end{cases}$$

$$g(w) = -\frac{1}{P}\sum_{p=1}^{P} y_p \log(\sigma(\mathring{x}_p^T w)) - (1-y_p)\log(1-\sigma(\mathring{x}_p^T w))$$

$$\Rightarrow \quad \nabla g_p(w) = -y_p \frac{\sigma(\mathring{x}_p^T w)(1-\sigma(\mathring{x}_p^T w))}{\sigma(\mathring{x}_p^T w)} \mathring{x}_p^T - (1-y_p) \frac{\sigma(\mathring{x}_p^T w)(1-\sigma(\mathring{x}_p^T w))}{1-\sigma(\mathring{x}_p^T w)} \cdot \mathring{x}_p^T$$

$$= -y_p \mathring{x}_p (1-\sigma(\mathring{x}_p^T w)) - (1-y_p)\mathring{x}_p^T \sigma(\mathring{x}_p^T w)$$

$$= -(y_p - \sigma(\mathring{x}_p^T w))\mathring{x}_p$$

$$\therefore \nabla g(w) = -\frac{1}{P}\sum_{p=1}^{P} (y_p - \sigma(\mathring{x}_p^T w))\mathring{x}_p$$

For $\nabla g_p(w) = (\sigma(\mathring{x}_p^T w))' \mathring{x}_p$

$$\frac{\partial}{\partial w_i} \cdot \frac{\partial}{\partial w_j} = \sigma(\mathring{x}_p^T w)(1-\sigma(\mathring{x}_p^T w))\, \mathring{x}_{pi} \cdot \mathring{x}_{pj}$$

$$\therefore \nabla^2 g(w) = \frac{1}{P}\sum_{p=1}^{P} \sigma(\mathring{x}_p^T w)(1-\sigma(\mathring{x}_p^T w))\, \mathring{x}_p \cdot \mathring{x}_p^T$$

6.10 **The perceptron cost is convex**
Show that the Perceptron cost given in Equation (6.33) is convex using the zero-order definition of convexity described in Exercise 5.8.

$$g(\mathbf{w}) = \frac{1}{P}\sum_{p=1}^{P} \max\left(0, -y_p \mathring{\mathbf{x}}_p^T \mathbf{w}\right). \tag{6.33}$$

Sol:

We need to prove that $\lambda g(w_1) + (1-\lambda) g(w_2) \geq g(\lambda w_1 + (1-\lambda) w_2)$ ⇒ According to the zero-order definition of convexity

$\max(0, -y_p \mathring{x}_p^T(\lambda w_1 + (1-\lambda) w_2)) \leq \lambda \max(0, -y_p \mathring{x}_p^T w_1) + (1-\lambda)\max(0, -y_p \mathring{x}_p^T w_2)$

$\max(0, -(y_p \mathring{x}_p^T w_1 \lambda + y_p \mathring{x}_p^T w_2 (1-\lambda))) \leq \max(0, -y_p \mathring{x}_p^T w_1 \lambda) + \max(0, -y_p \mathring{x}_p^T w_2 (1-\lambda))$

① if $-y_p \mathring{x}_p^T w_1 \lambda < 0$ & $-y_p \mathring{x}_p^T w_2 (1-\lambda) < 0$

right and left side will be both 0.

② if $-y_p \mathring{x}_p^T w_1 \lambda < 0$ & $-y_p \mathring{x}_p^T w_2 (1-\lambda) > 0$

$\max(0, -(y_p \mathring{x}_p^T w_1 \lambda + y_p \mathring{x}_p^T w_2 (1-\lambda))) \leq -y_p \mathring{x}_p^T w_2 (1-\lambda)$

③ if $-y_p \mathring{x}_p^T w_1 \lambda > 0$ & $-y_p \mathring{x}_p^T w_2 (1-\lambda) < 0$

$\max(0, -(y_p \mathring{x}_p^T w_1 \lambda + y_p \mathring{x}_p^T w_2 (1-\lambda))) \leq -y_p \mathring{x}_p^T w_1 \lambda$

④ if $-y_p \mathring{x}_p^T w_1 \lambda > 0$ & $-y_p \mathring{x}_p^T w_2 (1-\lambda) > 0$

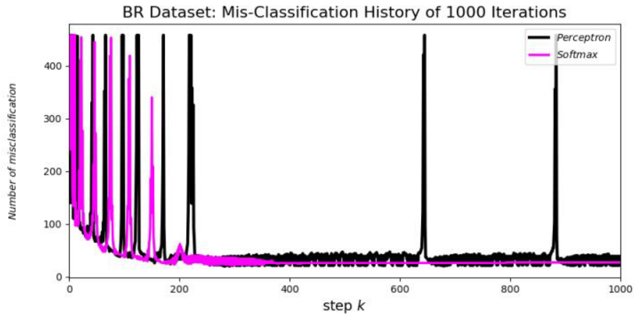
$-(y_p \mathring{x}_p^T w_1 \lambda + y_p \mathring{x}_p^T w_2 (1-\lambda)) \leq -y_p \mathring{x}_p^T w_1 \lambda - y_p \mathring{x}_p^T w_2 (1-\lambda)$

This equation is equal.

∴ Therefore, The perceptron cost is always convex

6.13 **Compare the efficacy of two-class cost functions I**

Compare the efficacy of the Softmax and the Perceptron cost functions in terms of the minimal number of misclassifications each can achieve by proper minimization via gradient descent on a breast cancer dataset. This dataset consists of $P = 699$ data points, each point consisting of $N = 9$ input of attributes of a single individual and output label indicating whether or not the individual does or does not have breast cancer. You should be able to achieve around 20 misclassifications with each method.



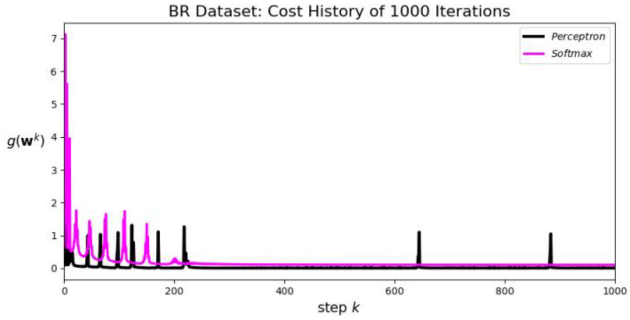
Cost function:

Softmax: $g(w) = \sum_{p=1}^{P} \log(1 + e^{-y_p \mathring{x}_p^T w})$

Here, study rate: 0.5 Iteration: 1000

Perceptron: $g(w) = \sum_{p=1}^{P} \max(0, 1 - y_p \mathring{x}^T w)$

Here, study rate: 0.1 Iteration: 1000



The minimum misclassification num is

$$\begin{cases} 20 & \text{For softmax} \\ 22 & \text{For perceptron} \end{cases}$$

both around 20

- **6_13 Compare the efficacy of two-class cost functions I**

```
import sys
import autograd.numpy as np
import autograd
from mlrefined_libraries.math_optimization_library import static_plotter

plotter = static_plotter.Visualizer()
sys.path.append('../')


class basic_ml_function(object):
    def __init__(self, data_set, stdlize=True):
        self.x = data_set[:-1, :]
        self.y = data_set[-1:, :]
        self.x_std = self.x.std(axis=1)[:, np.newaxis]
        self.x_mean = self.x.mean(axis=1)[:, np.newaxis]
        self.w0 = self.decent_initializer()
        if stdlize:
            self.data_initialization(data_set)

    def data_initialization(self):
        # The whole data processing piplne
        self.x_mean = np.nanmean(self.x, axis=1)
        x = self.data_recovery(self.x, self.x_mean)
        self.deviation_regulartor()
        self.x_mean = x.mean(axis=1)[:, np.newaxis]
        self.data_normalization(x)

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

    def deviation_regulartor(self):
        regulator = np.zeros(self.x_std.shape)
        for i in range(len(self.x_std)):
            if self.x_std[i] <= 0.1:
                regulator[i] = 1.0
                self.x_std += regulator
            else:
                pass
```

```python
def decent_initializer(self):
    w = 0.1 * np.random.randn(self.x.shape[0] + 1, 1)
    return w

@staticmethod
def data_recovery(x, mean):
    for i in np.argwhere(np.isnan(x) == True):
        x[i[0], i[1]] = mean[i[0]]
    return x

@staticmethod
def sigmoid(t):
    return 1 / (1 + np.exp(-t))

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

# cost function
def softmax(self, w):
    cost = np.sum(np.log(1 + np.exp(-self.y * self.linear_model(w))))
    return cost / float(np.size(self.y))

def perceptron(self, w):
    cost = np.sum(np.maximum(0, -self.y * self.linear_model(w)))
    return cost / float(np.size(self.y))

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

def least_absolute_deviations(self, w):
    cost = np.sum(np.abs(self.linear_model(w) - self.y))
    return cost / float(np.size(self.y))

def cross_entropy(self, w):
    a = self.sigmoid(self.linear_model(w))
    ind = np.argwhere(self.y == 0)[:, 1]
    cost = -np.sum(np.log(1 - a[:, ind]))
    ind = np.argwhere(self.y == 1)[:, 1]
    cost -= np.sum(np.log(a[:, ind]))
    return cost / self.y.size

# Optimization function
```

```python
    def gradient_decent(self, Loss_function, study_rate, iteration):
        """
        Gradient decent to minimize the cost function
        """
        if Loss_function == 'LSM':
            Loss_fun = self.least_squares_mean
        elif Loss_function == 'LAD':
            Loss_fun = self.least_absolute_deviations
        elif Loss_function == 'Softmax':
            Loss_fun = self.softmax
        elif Loss_function == 'Perceptron':
            Loss_fun = self.perceptron
        elif Loss_function == 'CrossEntropy':
            Loss_fun = self.perceptron
        else:
            raise Exception("Error Function Name")
        w = self.w0
        Gradient = autograd.grad(Loss_fun)
        weight_history = [w]
        cost_history = [Loss_fun(w)]
        for k in range(1, iteration + 1):
            grad_decent = Gradient(w)
            w = w - study_rate * grad_decent
            weight_history.append(w)
            cost_history.append(Loss_fun(w))
        if Loss_function == "LSM":
            cost_history = [cost ** 0.5 for cost in cost_history]
        return weight_history, cost_history

    def predict(self, w_trained):
        predicted_y = np.sign(self.linear_model(w_trained))
        return predicted_y

    def counting_mis_classification(self, weight_history):
        mismatching_his = []
        for w_p in weight_history:
            index = np.argwhere(self.y != self.predict(w_trained=w_p))
            mismatching_his.append(index.shape[0])
        return mismatching_his


if __name__ == "__main__":
    data_bcd = np.loadtxt('../mlrefined_datasets/superlearn_datasets/breast_cancer_data.csv',
delimiter=',')
```

```
BCD = basic_ml_function(data_bcd, stdlize=False)
weight_history_BCD_Per, cost_history_BCD_Per = BCD.gradient_decent('Perceptron', study_rate=0.1,
iteration=1000)
weight_history_BCD_Sof, cost_history_BCD_Sof = BCD.gradient_decent('Softmax', study_rate=0.5,
iteration=1000)
mismatch_his_Per = BCD.counting_mis_classification(weight_history_BCD_Per)
mismatch_his_Sof = BCD.counting_mis_classification(weight_history_BCD_Sof)
plotter.plot_mismatching_histories(histories=[mismatch_his_Per, mismatch_his_Sof], start=0,
                                   labels=['$ Perceptron $', '$ Softmax $'],
                                   title="BR Dataset: Mis-Classification History of 1000 Iterations")
plotter.plot_cost_histories(histories=[cost_history_BCD_Per, cost_history_BCD_Sof], start=0,
                            labels=['$ Perceptron $', '$ Softmax $'],
                            title="BR Dataset: Cost History of 1000 Iterations")
mini_percept = np.min(mismatch_his_Per)
mini_soft = np.min(mismatch_his_Sof)
print('mini_per: ' + str(mini_percept) + "mini_soft:" + str(mini_soft))
```

6.15 Credit check

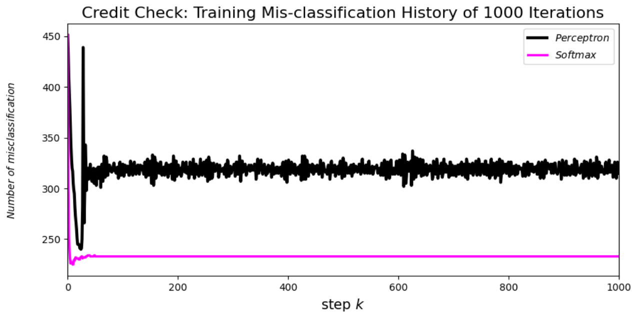
Repeat the experiment described in Example 6.11. Using an optimizer of your choice, try to achieve something close to the results reported. Make sure you *standard normalize* the input features of the dataset – as detailed in Section 9.3 – prior to optimization.

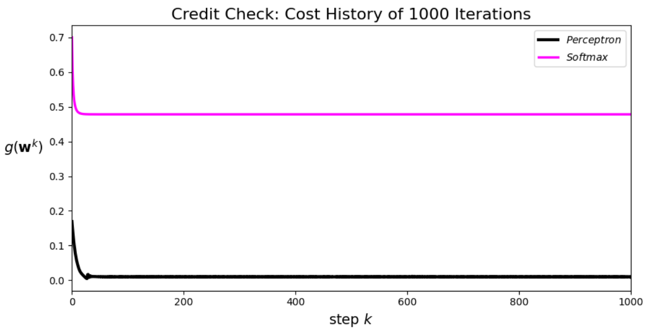
Example 6.11 Credit check

In this example we examine a two-class classification dataset consisting of $P = 1000$ samples, each a set of statistics extracted from loan application to a German bank (taken from [24]). Each input has an associated label: either a *good* (700 examples) or *bad* (300 examples) credit risk as determined by financial professionals. In learning a classifier for this dataset we create an automatic credit risk assessment tool that can help decide whether or not future applicants are good candidates for loans.

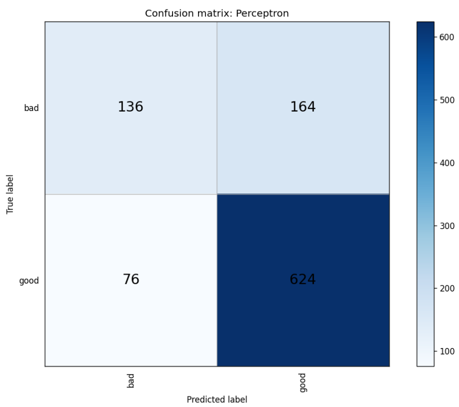
The $N = 20$ dimensional input features in this dataset include: the individual's current account balance with the bank (feature 1), the duration (in months) of previous credit with the bank (feature 2), the payment status of any prior credit taken out with the bank (feature 3), and the current value of their savings/stocks (feature 6). Properly minimizing the Perceptron cost we can achieve a 75 percent accuracy over the entire dataset, along with the following confusion matrix.

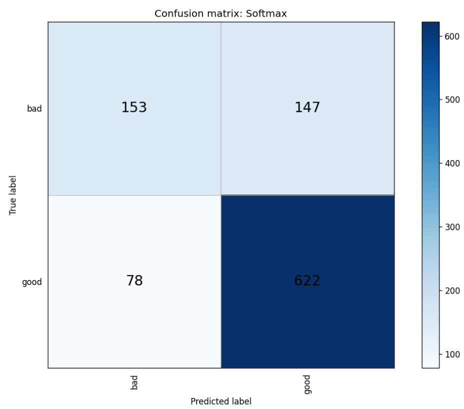
		Predicted	
		bad	good
Actual	bad	285	15
	good	234	466





below are the confusion matrixs of : Perceptron & Softmax





Accuracy (Perceptron): $\frac{136 + 624}{164 + 76 + 136 + 624} = 72\%$

Accuracy (Softmax): $\frac{153 + 622}{1000} = 77.5\%$

Both overall accuracy is close to 75%

- ## 6_15 Credit Check: With both Perceptron and SoftMax

```python
import sys
import autograd.numpy as np
import autograd
from mlrefined_libraries.math_optimization_library import static_plotter
from sklearn.metrics import confusion_matrix
import matplotlib.pyplot as plt

plotter = static_plotter.Visualizer()
sys.path.append('../')


class basic_ml_function(object):
    def __init__(self, data_set, stdlize=True):
        self.mismatching_his = None
        self.x = data_set[:-1, :]
        self.y = data_set[-1:, :]
        self.w0 = self.decent_initializer()
        if stdlize:
            self.data_initialization()

    def data_initialization(self):
        # The whole data processing piplne
        self.deviation_regulartor(self.x)
        x = self.data_recovery(self.x)
        self.x_mean = x.mean(axis=1)[:, np.newaxis]
        self.data_normalization(x)

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

    def deviation_regulartor(self, x):
        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] <= 0.1:
                regulator[i] = 1.0
                self.x_std += regulator
            else:
                pass
```

```python
def decent_initializer(self):
    w = 0.1 * np.random.randn(self.x.shape[0] + 1, 1)
    return w

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

@staticmethod
def sigmoid(t):
    return 1 / (1 + np.exp(-t))

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

# cost function
def softmax(self, w):
    cost = np.sum(np.log(1 + np.exp(-self.y * self.linear_model(w))))
    return cost / float(np.size(self.y))

def perceptron(self, w):
    cost = np.sum(np.maximum(0, -self.y * self.linear_model(w)))
    return cost / float(np.size(self.y))

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

def least_absolute_deviations(self, w):
    cost = np.sum(np.abs(self.linear_model(w) - self.y))
    return cost / float(np.size(self.y))

def cross_entropy(self, w):
    a = self.sigmoid(self.linear_model(w))
    ind = np.argwhere(self.y == 0)[:, 1]
    cost = -np.sum(np.log(1 - a[:, ind]))
    ind = np.argwhere(self.y == 1)[:, 1]
    cost -= np.sum(np.log(a[:, ind]))
    return cost / self.y.size

# Optimization function
```

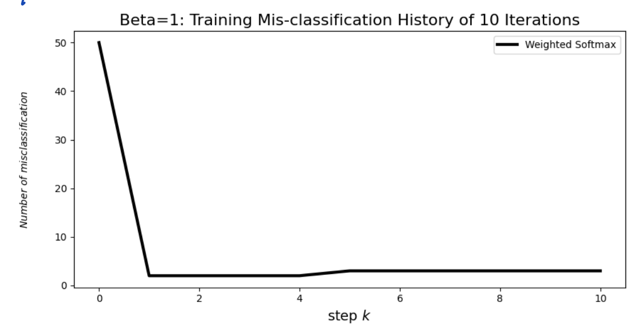
```python
    def gradient_decent(self, Loss_function, study_rate, iteration):
        """
        Gradient decent to minimize the cost function
        """
        if Loss_function == 'LSM':
            Loss_fun = self.least_squares_mean
        elif Loss_function == 'LAD':
            Loss_fun = self.least_absolute_deviations
        elif Loss_function == 'Softmax':
            Loss_fun = self.softmax
        elif Loss_function == 'Perceptron':
            Loss_fun = self.perceptron
        elif Loss_function == 'CrossEntropy':
            Loss_fun = self.cross_entropy
        else:
            raise Exception("Error Function Name")
        w = self.w0
        Gradient = autograd.grad(Loss_fun)
        weight_history = [w]
        cost_history = [Loss_fun(w)]
        for k in range(1, iteration + 1):
            grad_decent = Gradient(w)
            w = w - study_rate * grad_decent
            weight_history.append(w)
            cost_history.append(Loss_fun(w))
        if Loss_function == "LSM":
            cost_history = [cost ** 0.5 for cost in cost_history]
        return weight_history, cost_history

    def predict(self, w_trained):
        y_pred = np.sign(self.linear_model(w_trained))
        return y_pred

    def counting_mis_classification(self, weight_history):
        mismatching_his = []
        for w_p in weight_history:
            index = np.argwhere(self.y != self.predict(w_trained=w_p))
            mismatching_his.append(index.shape[0])
        self.mismatching_his = mismatching_his
        return mismatching_his

    # Plotting
    def confusion_matrix(self, weight_his, labels, normalize=False, title='Confusion Matrix',
precision="%0.f"):
```

```
        self.counting_mis_classification(weight_his)
        ind = np.argmin(self.mismatching_his)
        w_p = weight_his[ind]
        tick_marks = np.array(range(len(labels))) + 0.5
        cm = confusion_matrix(self.y[0], self.predict(w_p)[0])
        if normalize:
            cm = cm.astype('float') / cm.sum(axis=1)[:, np.newaxis]
            title = "Normalized " + title
            precision = "%0.2f"
        plt.figure(figsize=(12, 8), dpi=120)
        ind_array = np.arange(len(labels))
        x, y = np.meshgrid(ind_array, ind_array)
        for x_val, y_val in zip(x.flatten(), y.flatten()):
            c = cm[y_val][x_val]
            if c > 0.0:
                plt.text(x_val, y_val, precision % (c,), color='k', fontsize=17, va='center', ha='center')
        plt.gca().set_xticks(tick_marks, minor=True)
        plt.gca().set_yticks(tick_marks, minor=True)
        plt.gca().xaxis.set_ticks_position('none')
        plt.gca().yaxis.set_ticks_position('none')
        plt.grid(True, which='minor', linestyle='-')
        plt.gcf().subplots_adjust(bottom=0.15)
        plt.imshow(cm, interpolation='nearest', cmap='Blues')
        plt.title(title)
        plt.colorbar()
        xlocations = np.array(range(len(labels)))
        plt.xticks(xlocations, labels, rotation=90)
        plt.yticks(xlocations, labels)
        plt.ylabel('True label')
        plt.xlabel('Predicted label')
        plt.show()


if __name__ == "__main__":
    data_CD = np.loadtxt('../mlrefined_datasets/superlearn_datasets/credit_dataset.csv', delimiter=',')
    CD = basic_ml_function(data_CD, )

    weight_history_BCD_Per, cost_history_BCD_Per = CD.gradient_decent('Perceptron', study_rate=0.1,
iteration=1000)
    CD.confusion_matrix(weight_history_BCD_Per, labels=["bad", "good"], normalize=False,
                        title="Confusion matrix: Perceptron")
    mismatch_his_Per = CD.counting_mis_classification(weight_history_BCD_Per)

    weight_history_BCD_Sof, cost_history_BCD_Sof = CD.gradient_decent('Softmax', study_rate=1,
```

```
iteration=1000)
    CD.confusion_matrix(weight_history_BCD_Sof, labels=["bad", "good"], normalize=False,
                     title="Confusion matrix: Softmax")
    mismatch_his_Sof = CD.counting_mis_classification(weight_history_BCD_Sof)

    plotter.plot_mismatching_histories(histories=[mismatch_his_Per, mismatch_his_Sof], start=0,
                                     labels=['$ Perceptron $', '$ Softmax $'],
                                     title="Credit Check: Training Mis-classification History of 1000
Iterations")
    plotter.plot_cost_histories(histories=[cost_history_BCD_Per, cost_history_BCD_Sof], start=0,
                              labels=['$ Perceptron $', '$ Softmax $'],
                              title="Credit Check: Cost History of 1000 Iterations")
```

6.16 **Weighted classification and balanced accuracy**
Repeat the experiment described in Example 6.12 and shown in Figure 6.25. You need not reproduce the plots shown in the figure to confirm your implementation works properly, but should be able to achieve similar results to those reported there.

- $\beta = 1$



Balanced Acc = 80%

Overall Acc = 96.4%

Example 6.12 **Class imbalance and weighted classification**

In the left panel of Figure 6.25 we show a toy dataset with severe class imbalance. Here we also show the linear decision boundary resulting from minimizing the Softmax cost over this dataset using five steps of Newton's method, and color each region of the space based on how this trained classifier labels points. There are only three (of a total of 55) points in total misclassified here (one blue and two red – giving an accuracy close to 95 percent); however, those that are misclassified constitute almost half of the minority (red) class. While this is not reflected in a gross misclassification or accuracy metric, it is reflected in a balanced accuracy (see Section 6.8.4) which is significantly lower, at around 79 percent.

In the middle and right panels we show the result of increasing the weights of each member of the minority class from $\beta = 1$ to $\beta = 5$ (middle panel) and $\beta = 10$ (right panel). These weights are denoted visually in the figure by increasing the radius of the points in proportion to the value of β used (thus their radius increases from left to right). Also shown in the middle and right panels is the result of properly minimizing the weighted Softmax cost in Equation (6.83) using the same optimization procedure (i.e., five steps of Newton's method). As the value of β is increased on the minority class, we encourage fewer misclassifications of its members (at the expense here of additional misclassifications of the majority class). In the right panel of the figure – where $\beta = 10$ – we have one more misclassification than in the original run with an accuracy of 93 percent. However, with the assumption that misclassifying minority class members is far more perilous than misclassifying members of the majority class, here the trade-off is well worth it as no members of the minority class are misclassified. Moreover, we achieve a significantly improved *balanced accuracy score* of 96 percent over the 79 percent achieved with the original (unweighted) run.

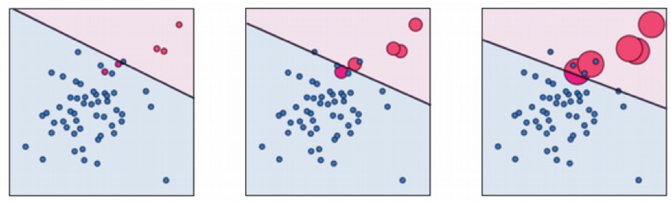


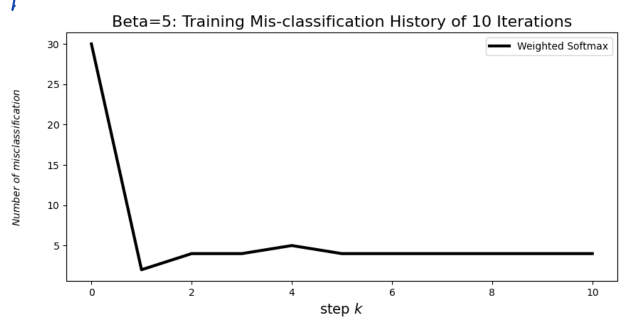
Figure 6.25 Figure associated with Example 6.12. See text for details.

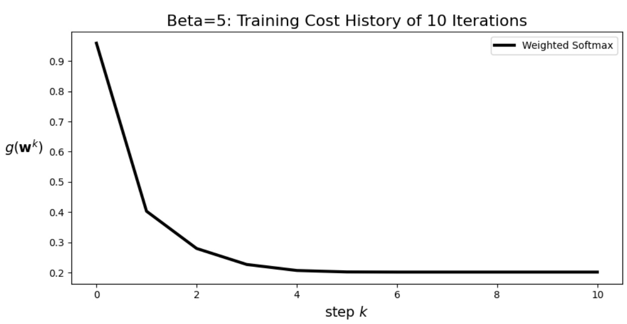
$$A_{+1} = 1 - \frac{1}{|\Omega_{+1}|}\sum_{p\in\Omega_{+1}} I(\hat{y}_p, y_p)$$

$$A_{-1} = 1 - \frac{1}{|\Omega_{-1}|}\sum_{p\in\Omega_{-1}} I(\hat{y}_p, y_p)$$

$$A_{balanced} = \frac{A_{+1} + A_{-1}}{2}$$

- $\beta = 5$

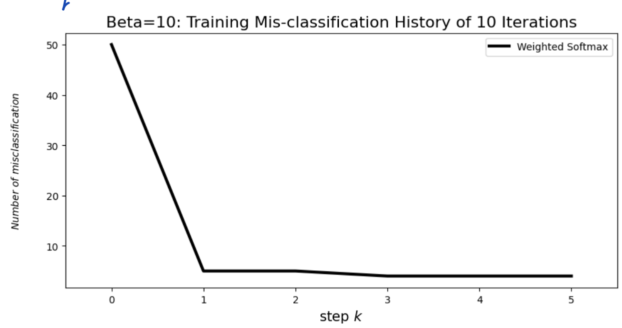


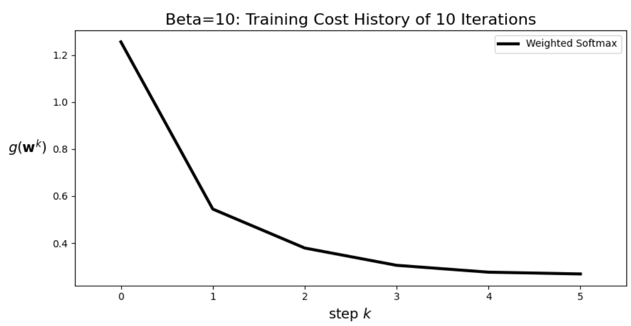


Balanced Acc = 89%

Overall Acc = 96.4%

- $\beta = 10$





Balanced Acc : 96%

Overall Acc : 92.7%

With the increase of β, balanced accuracy will increase significantly while overall accuracy will have a slight drop.

- ## 5_9 Weight classification and balanced accuracy

```python
import sys
import autograd.numpy as np
import autograd
from mlrefined_libraries.math_optimization_library import static_plotter
from sklearn.metrics import confusion_matrix
import matplotlib.pyplot as plt

plotter = static_plotter.Visualizer()
sys.path.append('../')


class basic_ml_function(object):
    def __init__(self, data_set, stdlize=True, beta=1):
        self.x = data_set[:-1, :]
        self.y = data_set[-1:, :]
        self.w0 = self.decent_initializer()
        if stdlize:
            self.data_initialization()
        self.mismatching_his = None
        self.balanced_acc = None
        self.overall_acc = None
        self.beta = np.array([1.0, 1.0 * beta])

    def data_initialization(self):
        # The whole data processing pipeline
        self.deviation_regulartor(self.x)
        x = self.data_recovery(self.x)
        self.x_mean = x.mean(axis=1)[:, np.newaxis]
        self.data_normalization(x)

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

    def deviation_regulartor(self, x):
        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
```

```python
        else:
            pass

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

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

    @staticmethod
    def sigmoid(t):
        return 1 / (1 + np.exp(-t))

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

    # cost function
    def softmax(self, w):
        cost = np.sum(np.log(1 + np.exp(-self.y * self.linear_model(w))))
        return cost / float(np.size(self.y))

    def perceptron(self, w):
        cost = np.sum(np.maximum(0, -self.y * self.linear_model(w)))
        return cost / float(np.size(self.y))

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

    def least_absolute_deviations(self, w):
        cost = np.sum(np.abs(self.linear_model(w) - self.y))
        return cost / float(np.size(self.y))

    def weighted_softmax(self, w):
        a = self.sigmoid(self.linear_model(w))
        ind = np.argwhere(self.y == -1)[:, 1]
        cost = -self.beta[0] * np.sum(np.log(1 - a[:, ind]))
        ind = np.argwhere(self.y == 1)[:, 1]
        cost -= self.beta[1] * np.sum(np.log(a[:, ind]))
```

```python
        return cost / self.y.size

    # Optimization function
    def gradient_decent(self, Loss_function, study_rate, iteration):
        """
        Gradient decent to minimize the cost function
        """
        if Loss_function == 'LSM':
            Loss_fun = self.least_squares_mean
        elif Loss_function == 'LAD':
            Loss_fun = self.least_absolute_deviations
        elif Loss_function == 'Softmax':
            Loss_fun = self.softmax
        elif Loss_function == 'Perceptron':
            Loss_fun = self.perceptron
        elif Loss_function == 'WS':
            Loss_fun = self.weighted_softmax
        else:
            raise Exception("Error Function Name")
        w = self.w0
        Gradient = autograd.grad(Loss_fun)
        weight_history = [w]
        cost_history = [Loss_fun(w)]
        for k in range(1, iteration + 1):
            grad_decent = Gradient(w)
            w = w - study_rate * grad_decent
            weight_history.append(w)
            cost_history.append(Loss_fun(w))
        if Loss_function == "LSM":
            cost_history = [cost ** 0.5 for cost in cost_history]
        return weight_history, cost_history

    def newtons_method(self, Loss_function, iteration, **kwargs):
        w = self.w0
        if Loss_function == 'LSM':
            Loss_fun = self.least_squares_mean
        elif Loss_function == 'LAD':
            Loss_fun = self.least_absolute_deviations
        elif Loss_function == 'Softmax':
            Loss_fun = self.softmax
        elif Loss_function == 'Perceptron':
            Loss_fun = self.perceptron
        elif Loss_function == 'WS':
            Loss_fun = self.weighted_softmax
```

```
        else:
            raise Exception("Error Function Name")
        gradient = autograd.grad(Loss_fun)
        hess = autograd.hessian(Loss_fun)
        epsilon = 10 ** (-10)
        if 'epsilon' in kwargs:
            epsilon = kwargs['epsilon']
        weight_history = [np.array(w)]
        cost_history = [np.array(Loss_fun(w))]
        for k in range(iteration):
            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(Loss_fun(w)))
        return weight_history, cost_history

    def predict(self, w_trained):
        y_pred = np.sign(self.linear_model(w_trained))
        return y_pred

    def balanced_accuracy(self, weight_history):
        miss_1 = []
        miss_2 = []
        ind = np.argmin(self.mismatching_his)
        w_p = weight_history[ind]
        index_class_1 = np.argwhere(self.y == -1)
        for v in index_class_1:
            miss_1.append(v[1])
        true_sample1 = np.argwhere(self.y[:, miss_1] == self.predict(w_p)[:, miss_1])
        acc1 = len(true_sample1) / len(miss_1)

        index_class_2 = np.argwhere(self.y == 1)
        for v in index_class_2:
            miss_2.append(v[1])
        true_sample2 = np.argwhere(self.y[:, miss_2] == self.predict(w_p)[:, miss_2])
        acc2 = len(true_sample2) / len(miss_2)
        self.balanced_acc = (acc1 + acc2) / 2
        self.overall_acc = (len(true_sample1) + len(true_sample2)) / (len(miss_1) + len(miss_2))

    def counting_mis_classification(self, weight_history):
```

```
        mismatching_his = []
        for w_p in weight_history:
            index = np.argwhere(self.y != self.predict(w_trained=w_p))
            mismatching_his.append(index.shape[0])
        self.mismatching_his = mismatching_his
        return mismatching_his

if __name__ == "__main__":
    data_3D =
np.loadtxt('../mlrefined_datasets/superlearn_datasets/3d_classification_data_v2_mbalanced.csv',
                    delimiter=',')

    """
    Beta = 1
    """
    JQ1 = basic_ml_function(data_3D, stdlize=False, beta=1)
    weight_history_JQ2_Per, cost_history_JQ2_Per = JQ1.newtons_method('WS', study_rate=0.1,
iteration=10)
    mismatch_his_Sof = JQ1.counting_mis_classification(weight_history_JQ2_Per)
    JQ1.balanced_accuracy(weight_history_JQ2_Per)
    plotter.plot_mismatching_histories(histories=[mismatch_his_Sof], start=0, labels=['Weighted
Softmax'],
                                       title="Beta=1: Training Mis-classification History of 10 Iterations")
    plotter.plot_cost_histories(histories=[cost_history_JQ2_Per], start=0, labels=['Weighted Softmax'],
                                title="Beta=1: Training Cost History of 10 Iterations")
    print("the balanced acc is:" + str(JQ1.balanced_acc))
    print("the overall acc is:" + str(JQ1.overall_acc))
    """
    Beta = 5
    """
    JQ2 = basic_ml_function(data_3D, stdlize=False, beta=5)
    weight_history_JQ2_Per, cost_history_JQ2_Per = JQ2.newtons_method('WS', study_rate=0.1,
iteration=10)
    mismatch_his_Sof = JQ2.counting_mis_classification(weight_history_JQ2_Per)
    JQ2.balanced_accuracy(weight_history_JQ2_Per)
    plotter.plot_mismatching_histories(histories=[mismatch_his_Sof], start=0, labels=['Weighted
Softmax'],
                                       title="Beta=5: Training Mis-classification History of 10 Iterations")
    plotter.plot_cost_histories(histories=[cost_history_JQ2_Per], start=0, labels=['Weighted Softmax'],
                                title="Beta=5: Training Cost History of 10 Iterations")
    print("the balanced acc is:" + str(JQ2.balanced_acc))
    print("the overall acc is:" + str(JQ2.overall_acc))
    """
    Beta = 10
```

```
    """
    JQ3 = basic_ml_function(data_3D, stdlize=False, beta=10)
    weight_history_JQ2_Per, cost_history_JQ2_Per = JQ3.newtons_method('WS', study_rate=0.1, iteration=5)
    mismatch_his_Sof = JQ3.counting_mis_classification(weight_history_JQ2_Per)
    # JQ3.balanced_accuracy(weight_history_JQ2_Per)
    JQ3.balanced_accuracy(weight_history_JQ2_Per)
    plotter.plot_mismatching_histories(histories=[mismatch_his_Sof], start=0, labels=['Weighted 
Softmax'],
                                    title="Beta=10: Training Mis-classification History of 10 Iterations")
    plotter.plot_cost_histories(histories=[cost_history_JQ2_Per], start=0, labels=['Weighted Softmax'],
                             title="Beta=10: Training Cost History of 10 Iterations")
    print("the balanced acc is:" + str(JQ3.balanced_acc))
    print("the overall acc is:" + str(JQ3.overall_acc))
```