深度学习中的正则化

朱明超
Email: deityrayleigh@gmail.com
Github: github.com/MingchaoZhu/DeepLearning

正则化的目标是减少模型泛化误差，为此提出了各种方法。本篇提出的正则化方法主要是考虑**当训练误差较小，但泛化误差较大**的情况下。

1 参数范数惩罚

许多正则化方法 (如神经网络、线性回归、逻辑回归) 通过对目标函数 J 添加一个参数范数惩罚 $\Omega(\theta)$，限制模型的学习能力。将正则化后的目标函数记为 $\tilde{J}$：

$$\tilde{J}(\theta;\boldsymbol{X},\boldsymbol{y}) = J(\theta;\boldsymbol{X},\boldsymbol{y}) + \alpha\Omega(\theta) \tag{1}$$

其中 $\alpha \in [0,+\infty)$ 是衡量参数范数惩罚程度的超参数。$\alpha = 0$ 表示没有正则化，α 越大对应正则化惩罚越大。

在神经网络中，参数包括每层线性变换的权重和偏置，我们**通常只对权重做惩罚而不对偏置做正则惩罚**；使用向量 $\boldsymbol{w}$ 表示应受惩罚影响的权重，用向量 θ 表示所有参数。

1.1 L^2 正则化

L^2 参数正则化 (也称为岭回归、Tikhonov 正则) 通常被称为权重衰减 (`weight decay`)，是通过向目标函数添加一个正则项 $\Omega(\theta) = \frac{1}{2}\|\boldsymbol{w}\|_2^2$ ，使权重更加接近原点。

目标函数：

$$\tilde{J}(\boldsymbol{w};\boldsymbol{X},\boldsymbol{y}) = J(\boldsymbol{w};\boldsymbol{X},y) + \frac{\alpha}{2}\boldsymbol{w}^\top\boldsymbol{w} \tag{2}$$

计算梯度：

$$\nabla_{\boldsymbol{w}}\tilde{J}(\boldsymbol{w};\boldsymbol{X},\boldsymbol{y}) = \nabla_{\boldsymbol{w}}J(\boldsymbol{w};\boldsymbol{X},\boldsymbol{y}) + \alpha\boldsymbol{w} \tag{3}$$

更新权重：

$$\boldsymbol{w} \leftarrow \boldsymbol{w} - \epsilon(\alpha\boldsymbol{w} + \nabla_{\boldsymbol{w}}J(\boldsymbol{w};\boldsymbol{X},\boldsymbol{y})) = (1-\epsilon\alpha)\boldsymbol{w} - \epsilon\nabla_{\boldsymbol{w}}J(\boldsymbol{w};\boldsymbol{X},\boldsymbol{y}) \tag{4}$$

从上式可以看出，加入权重衰减后会导致学习规则的修改，即在每步执行梯度更新前先收缩权重 (乘以 $(1-\epsilon\alpha)$)。

以第六章介绍的代价函数 $J(\theta) = -\frac{1}{m}\sum_{i=1}^{m}\left(\boldsymbol{y}^{(i)}\log\hat{\boldsymbol{y}}^{(i)} + (1-\boldsymbol{y}^{(i)})\log(1-\hat{\boldsymbol{y}}^{(i)})\right)$ 为例，在增加 L^2 正则化后，代价函数变为：

$$J_{regularized} = \underbrace{-\frac{1}{m}\sum_{i=1}^{m}\left(\boldsymbol{y}^{(i)}\log\left(\hat{\boldsymbol{y}}^{(i)} + (1-\boldsymbol{y}^{(i)})\log\left(1-\hat{\boldsymbol{y}}^{(i)}\right)\right)\right.}_{\text{cross-entropy cost}} + \underbrace{\frac{1}{m}\frac{\lambda}{2}\sum_{l}\sum_{k}\sum_{j}W_{k,j}^{[l]2}}_{\text{L2 regularization cost}} \tag{5}$$

而在反向传播的时候，须加上正则化项的梯度：

$$\frac{d}{d\boldsymbol{W}}(\frac{1}{2}\frac{\lambda}{m}\boldsymbol{W}^2) = \frac{\lambda}{m}\boldsymbol{W} \tag{6}$$

1.2 L^1 正则化

将参数惩罚项 $\Omega(\theta)$ 由权重衰减项修改为各个参数的绝对值之和，可以得到 L^1 正则化：

$$\Omega(\theta) = \|\boldsymbol{w}\|_1 = \sum_i |w_i| \tag{7}$$

目标函数：

$$\tilde{J}(\boldsymbol{w};\boldsymbol{X},\boldsymbol{y}) = J(\boldsymbol{w};\boldsymbol{X},\boldsymbol{y}) + \alpha\|\boldsymbol{w}\|_1 \tag{8}$$

计算梯度：

$$\nabla_{\boldsymbol{w}}\tilde{J}(\boldsymbol{w};\boldsymbol{X},\boldsymbol{y}) = \nabla_{\boldsymbol{w}}J(\boldsymbol{w};\boldsymbol{X},\boldsymbol{y}) + \alpha sgn(\boldsymbol{w}) \tag{9}$$

其中 $sgn(x)$ 为符号函数，取各个元素的正负号。

更新权重：

$$\boldsymbol{w} \leftarrow \boldsymbol{w} - \epsilon(\alpha sgn(\boldsymbol{w}) + \nabla_{\boldsymbol{w}}J(\boldsymbol{w};\boldsymbol{X},\boldsymbol{y})) \tag{10}$$

同样，在增加 L^1 正则化后，代价函数变为：

$$J_{regularized} = \underbrace{-\frac{1}{m}\sum_{i=1}^{m}\left(\boldsymbol{y}^{(i)}\log\hat{\boldsymbol{y}}^{(i)} + (1-\boldsymbol{y}^{(i)})\log(1-\hat{\boldsymbol{y}}^{(i)})\right)}_{\text{cross-entropy cost}} + \underbrace{\frac{1}{m}\frac{\lambda}{2}\sum_{l}\sum_{k}\sum_{j}|W_{k,j}^{[l]}|}_{\text{L1 regularization cost}} \tag{11}$$

而在反向传播的时候，须加上正则化项的梯度：

$$\frac{d}{d\boldsymbol{W}}(\frac{\lambda}{m}\|\boldsymbol{W}\|) = \frac{\lambda}{m}sgn(\boldsymbol{W}) \tag{12}$$

[1]:
```
from abc import ABC, abstractmethod
import numpy as np
from PIL import Image
%matplotlib inline
import matplotlib.pyplot as plt
import math
import re
import time
import progressbar
from sklearn.model_selection import train_test_split
from sklearn.preprocessing import StandardScaler
```

[2]:
```
class RegularizerBase(ABC):

    def __init__(self, **kwargs):
        super().__init__()

    @abstractmethod
    def loss(self, **kwargs):
        raise NotImplementedError

    @abstractmethod
    def grad(self, **kwargs):
        raise NotImplementedError

class L1Regularizer(RegularizerBase):

    def __init__(self, lambd=0.001):
        super().__init__()
        self.lambd = lambd

    def loss(self, params):
        loss = 0
        pattern = re.compile(r'^W\d+')
        for key, val in params.items():
            if pattern.match(key):
                loss +=  0.5 * np.sum(np.abs(val)) * self.lambd
        return loss

    def grad(self, params):
        for key, val in params.items():
            grad = self.lambd * np.sign(val)
        return grad

class L2Regularizer(RegularizerBase):

    def __init__(self, lambd=0.001):
        super().__init__()
        self.lambd = lambd

    def loss(self, params):
        loss = 0
```

```
        for key, val in params.items():
            loss +=  0.5 * np.sum(np.square(val)) * self.lambd
        return loss

    def grad(self, params):
        for key, val in params.items():
            grad = self.lambd * val
        return grad

class RegularizerInitializer(object):

    def __init__(self, regular_name="l2"):
        self.regular_name = regular_name

    def __call__(self):
        r = r"([a-zA-Z]*)=([^,)]*)"
        regular_str = self.regular_name.lower()
        kwargs = dict([(i, eval(j)) for (i, j) in re.findall(r, regular_str)])
        if  "l1" in regular_str.lower():
            regular = L1Regularizer(**kwargs)
        elif "l2" in regular_str.lower():
            regular = L2Regularizer(**kwargs)
        else:
            raise ValueError("Unrecognized regular: {}".format(regular_str))
        return regular
```

我们对第六章介绍的 DFN 引入正则项，我们将第六章中介绍的函数存储在 chapter6.py 中。

[3]:
```
from chapter6 import WeightInitializer, ActivationInitializer, LayerBase, CrossEntropy, OrderedDict, softmax
```

[4]:
```
class FullyConnected(LayerBase):
    """
    定义全连接层，实现 a=g(x*W+b)，前向传播输入 x，返回 a；反向传播输入
    """
    def __init__(self, n_out, acti_fn, init_w, optimizer=None):
        """
        参数说明：
        acti_fn：激活函数， str 型
        init_w：权重初始化方法， str 型
        n_out：隐藏层输出维数
        optimizer：优化方法
        """
        super().__init__(optimizer)
        self.n_in = None  # 隐藏层输入维数， int 型
        self.n_out = n_out  # 隐藏层输出维数， int 型
        self.acti_fn = ActivationInitializer(acti_fn)()
        self.init_w = init_w
        self.init_weights = WeightInitializer(mode=init_w)
        self.is_initialized = False  # 是否初始化， bool 型变量

    def _init_params(self):
        b = np.zeros((1, self.n_out))
        W = self.init_weights((self.n_in, self.n_out))
        self.params = {"W": W, "b": b}
        self.gradients = {"W": np.zeros_like(W), "b": np.zeros_like(b)}
        self.is_initialized = True

    def forward(self, X, retain_derived=True):
        """
        全连接网络的前向传播，原理见上文 反向传播算法 部分。
```

```
        参数说明：
        X：输入数组，为（n_samples, n_in），float 型
        retain_derived：是否保留中间变量，以便反向传播时再次使用，bool 型
        """
        if not self.is_initialized:  # 如果参数未初始化，先初始化参数
            self.n_in = X.shape[1]
            self._init_params()
        W = self.params["W"]
        b = self.params["b"]
        z = X @ W + b
        a = self.acti_fn.forward(z)
        if retain_derived:
            self.X.append(X)
        return a

    def backward(self, dLda, retain_grads=True, regular=None):
        """
        全连接网络的反向传播，原理见上文 反向传播算法 部分。

        参数说明：
        dLda：关于损失的梯度，为（n_samples, n_out），float 型
        retain_grads：是否计算中间变量的参数梯度，bool 型
        regular：正则化项
        """
        if not isinstance(dLda, list):
            dLda = [dLda]
        dX = []
        X = self.X
        for da, x in zip(dLda, X):
            dx, dw, db = self._bwd(da, x, regular)
            dX.append(dx)
            if retain_grads:
                self.gradients["W"] += dw
                self.gradients["b"] += db
        return dX[0] if len(X) == 1 else dX

    def _bwd(self, dLda, X, regular):
        W = self.params["W"]
        b = self.params["b"]
        Z = X @ W + b
        dZ = dLda * self.acti_fn.grad(Z)
        dX = dZ @ W.T
        dW = X.T @ dZ
        db = dZ.sum(axis=0, keepdims=True)
        if regular is not None:
            n = X.shape[0]
            dW_norm = regular.grad(self.params) / n
            dW += dW_norm
        return dX, dW, db

    @property
    def hyperparams(self):
        return {
            "layer": "FullyConnected",
            "init_w": self.init_w,
            "n_in": self.n_in,
            "n_out": self.n_out,
            "acti_fn": str(self.acti_fn),
```

```
            "optimizer": {
                "hyperparams": self.optimizer.hyperparams,
            },
            "components": {
                k: v for k, v in self.params.items()
            }
        }
```

[5]:
```
def minibatch(X, batchsize=256, shuffle=True):
    """
    函数作用：将数据集分割成 batch， 基于 mini batch 训练，具体可见第 8 章。
    """
    N = X.shape[0]
    idx = np.arange(N)
    n_batches = int(np.ceil(N / batchsize))
    if shuffle:
        np.random.shuffle(idx)
    def mb_generator():
        for i in range(n_batches):
            yield idx[i * batchsize : (i + 1) * batchsize]

    return mb_generator(), n_batches

class DFN(object):

    def __init__(
        self,
        hidden_dims_1=None,
        hidden_dims_2=None,
        optimizer="sgd(lr=0.01)",
        init_w="std_normal",
        regular_act=None,
        loss=CrossEntropy()
    ):
        self.optimizer = optimizer
        self.init_w = init_w
        self.loss = loss
        self.regular_act = regular_act
        self.regular = None
        self.hidden_dims_1 = hidden_dims_1
        self.hidden_dims_2 = hidden_dims_2
        self.is_initialized = False

    def _set_params(self):
        """
        函数作用：模型初始化
        FC1 -> Sigmoid -> FC2 -> Softmax
        """
        self.layers = OrderedDict()
        self.layers["FC1"] = FullyConnected(
            n_out=self.hidden_dims_1,
            acti_fn="sigmoid",
            init_w=self.init_w,
            optimizer=self.optimizer
        )
        self.layers["FC2"] = FullyConnected(
            n_out=self.hidden_dims_2,
            acti_fn="affine(slope=1, intercept=0)",
            init_w=self.init_w,
```

```
            optimizer=self.optimizer
        )
        if self.regular_act is not None:
            self.regular = RegularizerInitializer(self.regular_act)()
        self.is_initialized = True

    def forward(self, X_train):
        Xs = {}
        out = X_train
        for k, v in self.layers.items():
            Xs[k] = out
            out = v.forward(out)
        return out, Xs

    def backward(self, grad):
        dXs = {}
        out = grad
        for k, v in reversed(list(self.layers.items())):
            dXs[k] = out
            out = v.backward(out, regular=self.regular)
        return out, dXs

    def update(self):
        """
        函数作用：梯度更新
        """
        for k, v in reversed(list(self.layers.items())):
            v.update()
        self.flush_gradients()

    def flush_gradients(self, curr_loss=None):
        """
        函数作用：更新后重置梯度
        """
        for k, v in self.layers.items():
            v.flush_gradients()

    def fit(self, X_train, y_train, n_epochs=20, batch_size=64, verbose=False):
        """
        参数说明：
        X_train：训练数据
        y_train：训练数据标签
        n_epochs：epoch 次数
        batch_size：每次 epoch 的 batch size
        verbose：是否每个 batch 输出损失
        """
        self.verbose = verbose
        self.n_epochs = n_epochs
        self.batch_size = batch_size
        if not self.is_initialized:
            self.n_features = X_train.shape[1]
            self._set_params()

        prev_loss = np.inf
        for i in range(n_epochs):
            loss, epoch_start = 0.0, time.time()
            batch_generator, n_batch = minibatch(X_train, self.batch_size, shuffle=True)
            for j, batch_idx in enumerate(batch_generator):
                batch_len, batch_start = len(batch_idx), time.time()
```

```
            X_batch, y_batch = X_train[batch_idx], y_train[batch_idx]
            out, _ = self.forward(X_batch)
            y_pred_batch = softmax(out)
            batch_loss = self.loss(y_batch, y_pred_batch)
            # 正则化损失
            if self.regular is not None:
                for _, layerparams in self.hyperparams['components'].items():
                    assert type(layerparams) is dict
                    batch_loss += self.regular.loss(layerparams)
            grad = self.loss.grad(y_batch, y_pred_batch)
            _, _ = self.backward(grad)
            self.update()
            loss += batch_loss
            if self.verbose:
                fstr = "\t[Batch {}/{}] Train loss: {:.3f} ({:.1f}s/batch)"
                print(fstr.format(j + 1, n_batch, batch_loss, time.time() - batch_start))

        loss /= n_batch
        fstr = "[Epoch {}] Avg. loss: {:.3f}  Delta: {:.3f} ({:.2f}m/epoch)"
        print(fstr.format(i + 1, loss, prev_loss - loss, (time.time() - epoch_start) / 60.0))
        prev_loss = loss

    def evaluate(self, X_test, y_test, batch_size=128):
        acc = 0.0
        batch_generator, n_batch = minibatch(X_test, batch_size, shuffle=True)
        for j, batch_idx in enumerate(batch_generator):
            batch_len, batch_start = len(batch_idx), time.time()
            X_batch, y_batch = X_test[batch_idx], y_test[batch_idx]
            y_pred_batch, _ = self.forward(X_batch)
            y_pred_batch = np.argmax(y_pred_batch, axis=1)
            y_batch = np.argmax(y_batch, axis=1)
            acc += np.sum(y_pred_batch == y_batch)
        return acc / X_test.shape[0]

    @property
    def hyperparams(self):
        return {
            "init_w": self.init_w,
            "loss": str(self.loss),
            "optimizer": self.optimizer,
            "regular": str(self.regular_act),
            "hidden_dims_1": self.hidden_dims_1,
            "hidden_dims_2": self.hidden_dims_2,
            "components": {k: v.params for k, v in self.layers.items()}
        }
```

[6]:
```
def load_data(path="../data/mnist/mnist.npz"):
    f = np.load(path)
    X_train, y_train = f['x_train'], f['y_train']
    X_test, y_test = f['x_test'], f['y_test']
    f.close()
    return (X_train, y_train), (X_test, y_test)

(X_train, y_train), (X_test, y_test) = load_data()
y_train = np.eye(10)[y_train.astype(int)]
y_test = np.eye(10)[y_test.astype(int)]
X_train = X_train.reshape(-1, X_train.shape[1]*X_train.shape[2]).astype('float32')
X_test = X_test.reshape(-1, X_test.shape[1]*X_test.shape[2]).astype('float32')
print(X_train.shape, y_train.shape)
```

```
N = 20000   # 取 20000 条数据用以训练
indices = np.random.permutation(range(X_train.shape[0]))[:N]
X_train, y_train = X_train[indices], y_train[indices]
print(X_train.shape, y_train.shape)
X_train /= 255
X_train = (X_train - 0.5) * 2
X_test /= 255
X_test = (X_test - 0.5) * 2
```

```
(60000, 784) (60000, 10)
(20000, 784) (20000, 10)
```

[7]:
```
"""
不引入正则化
"""
model = DFN(hidden_dims_1=200, hidden_dims_2=10)
model.fit(X_train, y_train, n_epochs=20, batch_size=64)
```

```
[Epoch 1] Avg. loss: 2.286  Delta: inf (0.01m/epoch)
[Epoch 2] Avg. loss: 2.209  Delta: 0.078 (0.01m/epoch)
[Epoch 3] Avg. loss: 1.993  Delta: 0.215 (0.01m/epoch)
[Epoch 4] Avg. loss: 1.640  Delta: 0.353 (0.01m/epoch)
[Epoch 5] Avg. loss: 1.305  Delta: 0.335 (0.01m/epoch)
[Epoch 6] Avg. loss: 1.063  Delta: 0.242 (0.01m/epoch)
[Epoch 7] Avg. loss: 0.898  Delta: 0.166 (0.01m/epoch)
[Epoch 8] Avg. loss: 0.781  Delta: 0.117 (0.01m/epoch)
[Epoch 9] Avg. loss: 0.696  Delta: 0.085 (0.01m/epoch)
[Epoch 10] Avg. loss: 0.634  Delta: 0.062 (0.01m/epoch)
[Epoch 11] Avg. loss: 0.586  Delta: 0.048 (0.01m/epoch)
[Epoch 12] Avg. loss: 0.549  Delta: 0.037 (0.01m/epoch)
[Epoch 13] Avg. loss: 0.518  Delta: 0.031 (0.02m/epoch)
[Epoch 14] Avg. loss: 0.493  Delta: 0.025 (0.02m/epoch)
[Epoch 15] Avg. loss: 0.473  Delta: 0.021 (0.01m/epoch)
[Epoch 16] Avg. loss: 0.454  Delta: 0.018 (0.01m/epoch)
[Epoch 17] Avg. loss: 0.439  Delta: 0.015 (0.01m/epoch)
[Epoch 18] Avg. loss: 0.425  Delta: 0.014 (0.01m/epoch)
[Epoch 19] Avg. loss: 0.414  Delta: 0.012 (0.01m/epoch)
[Epoch 20] Avg. loss: 0.404  Delta: 0.010 (0.01m/epoch)
```

[8]:
```
print("without regularization -- accuracy:{}".format(model.evaluate(X_test, y_test)))


###### if show params ######
# print("regular", model.hyperparams["regular"], "\nparams:", model.hyperparams["components"])
```

```
without regularization -- accuracy:0.8961
```

[9]:
```
"""
引入 l2 正则化
"""
model_re = DFN(hidden_dims_1=200, hidden_dims_2=10, regular_act="l2(lambd=0.01)")
model_re.fit(X_train, y_train, n_epochs=20)
```

```
[Epoch 1] Avg. loss: 2.363  Delta: inf (0.02m/epoch)
[Epoch 2] Avg. loss: 2.284  Delta: 0.079 (0.02m/epoch)
[Epoch 3] Avg. loss: 2.068  Delta: 0.216 (0.02m/epoch)
[Epoch 4] Avg. loss: 1.729  Delta: 0.339 (0.02m/epoch)
[Epoch 5] Avg. loss: 1.428  Delta: 0.301 (0.02m/epoch)
[Epoch 6] Avg. loss: 1.226  Delta: 0.202 (0.02m/epoch)
[Epoch 7] Avg. loss: 1.096  Delta: 0.130 (0.02m/epoch)
[Epoch 8] Avg. loss: 1.013  Delta: 0.083 (0.02m/epoch)
[Epoch 9] Avg. loss: 0.958  Delta: 0.055 (0.02m/epoch)
```

```
[Epoch 10] Avg. loss: 0.923  Delta: 0.035 (0.01m/epoch)
[Epoch 11] Avg. loss: 0.899  Delta: 0.024 (0.01m/epoch)
[Epoch 12] Avg. loss: 0.883  Delta: 0.016 (0.01m/epoch)
[Epoch 13] Avg. loss: 0.872  Delta: 0.011 (0.01m/epoch)
[Epoch 14] Avg. loss: 0.865  Delta: 0.007 (0.01m/epoch)
[Epoch 15] Avg. loss: 0.860  Delta: 0.004 (0.01m/epoch)
[Epoch 16] Avg. loss: 0.858  Delta: 0.002 (0.01m/epoch)
[Epoch 17] Avg. loss: 0.858  Delta: 0.001 (0.01m/epoch)
[Epoch 18] Avg. loss: 0.858  Delta: -0.000 (0.02m/epoch)
[Epoch 19] Avg. loss: 0.859  Delta: -0.001 (0.01m/epoch)
[Epoch 20] Avg. loss: 0.860  Delta: -0.001 (0.01m/epoch)
```

[10]:
```
print("with L2 regularization -- accuracy:{}".format(model_re.evaluate(X_test, y_test)))


###### if show params ######
# print("regular", model_re.hyperparams["regular"], "\nparams:", model_re.hyperparams["components"])
```

```
with L2 regularization -- accuracy:0.8958
```

1.3 总结

相比 L^2 正则化，L^1 正则化会产生更稀疏的解。

假设 $\boldsymbol{w}^*$ 为未正则化的目标函数取得最优时的权重向量，并假设原目标函数有二阶导，将 $J(\boldsymbol{w})$ 在 $\boldsymbol{w}^*$ 处二阶泰勒展开 (最优值点一阶导数为 0)：

$$J(\boldsymbol{w}) \approx J(\boldsymbol{w}^*) + \frac{1}{2}(\boldsymbol{w} - \boldsymbol{w}^*)^\top \mathrm{H}(\boldsymbol{w} - \boldsymbol{w}^*) \tag{13}$$

其中 H 是 $J(\boldsymbol{w})$ 在 $\boldsymbol{w}^*$ 处的海森矩阵。$J(\boldsymbol{w})$ 最小时满足上式导数为 0，于是：

$$\nabla J(\boldsymbol{w}) = \mathrm{H}(\boldsymbol{w} - \boldsymbol{w}^*) = 0 \tag{14}$$

我们再考虑 L^2 正则化条件下，$\Omega(\theta) = \alpha\frac{1}{2}\|\boldsymbol{w}\|_2^2$，则可以得到：

$$\nabla J(\boldsymbol{w}) = \mathrm{H}(\boldsymbol{w} - \boldsymbol{w}^*) + \alpha\boldsymbol{w} = 0 \tag{15}$$

于是，我们可以得到新的最优解 $\tilde{\boldsymbol{w}}$ 满足：

$$\tilde{\boldsymbol{w}} = (\mathrm{H} + \alpha\boldsymbol{I})^{-1}\mathrm{H}\boldsymbol{w}^* \tag{16}$$

如果考 Hessian 矩阵是对角正定矩阵，我们得到 L^2 正则化的最优解是 $\tilde{w}_i = \frac{\mathrm{H}_{i,i}}{\mathrm{H}_{i,i}+\alpha}w_i^*$。如果 $w_i^* \neq 0$，则 $\tilde{w}_i \neq 0$，这说明 L^2 正则化不会使参数变得稀疏。

我们再看 L^1 正则化的最优解，同样，我们得到考虑 L^1 正则化条件下的最优解，此时需要满足：

$$\nabla J(\tilde{\boldsymbol{w}}) = \mathrm{H}(\tilde{\boldsymbol{w}} - \boldsymbol{w}^*) + \alpha sgn(\tilde{\boldsymbol{w}}) = 0 \tag{17}$$

为了简化讨论，我们假设 H 为对角阵，即 $\mathrm{H} = diag[\mathrm{H}_{1,1}, \mathrm{H}_{2,2}, ...\mathrm{H}_{n,n}], \mathrm{H}_{i,i} > 0$ (可以用 PCA 预处理输入特征得到)，此时

$$\tilde{w}_i = w_i^* - \frac{\alpha}{\mathrm{H}_{i,i}}sgn(\tilde{w}_i) \tag{18}$$

从这个式子也可以明显看出 $\tilde{\boldsymbol{w}}$ 和 $\boldsymbol{w}^*$ 是同号的。所以有：

$$\tilde{w}_i = w_i^* - \frac{\alpha}{\mathrm{H}_{i,i}}sgn(w_i^*) = sgn(w_i^*)\left(|w_i^*| - \frac{\alpha}{\mathrm{H}_{i,i}}\right) \tag{19}$$

同样，既然 $\tilde{\boldsymbol{w}}$ 和 $\boldsymbol{w}^*$ 是同号的，两边同乘 $sgn(\tilde{\boldsymbol{w}})$，得到：

$$|w_i^*| - \frac{\alpha}{\mathrm{H}_{i,i}} = |\tilde{w}_i| \geq 0 \tag{20}$$

于是刚才的式子可以进一步写为：

$$\tilde{w}_i = sgn(w_i^*)\max\left\{|w_i^*| - \frac{\alpha}{\mathrm{H}_{i,i}}, 0\right\} \tag{21}$$

可以看出，L^1 正则化有可能通过足够大的 α 实现稀疏。

- 正则化策略可以被解释为最大后验 (MAP) 贝叶斯推断。(详细内容见第五章)
  - L^2 正则化相当于权重是高斯先验的 MAP 贝叶斯推断；
  - L^1 正则化相当于权重是 `Laplace` 先验的 MAP 贝叶斯推断。

1.4 作为约束的范数惩罚

考虑参数范数正则化的代价函数：

$$\tilde{J}(\theta; \boldsymbol{X}, \boldsymbol{y}) = J(\theta; \boldsymbol{X}, \boldsymbol{y}) + \alpha\Omega(\theta) \tag{22}$$

如果想约束 $\Omega(\theta) < k$，k 是某个常数，可以构造广义 Lagrange 函数：

$$\mathcal{L}(\theta, \alpha; \boldsymbol{X}, \boldsymbol{y}) = J(\theta; \boldsymbol{X}, \boldsymbol{y}) + \alpha(\Omega(\theta) - k) \tag{23}$$

该约束问题的解是：

$$\theta^* = \arg\min_{\theta} \max_{\alpha, \alpha \geq 0} \mathcal{L}(\theta, \alpha) \tag{24}$$

对于该问题，可以通过调节 α 与 k 的值来扩大或缩小权重的约束区域。较大的 α 将得到一个较小的约束区域；而较小的 α 将得到一个较大的约束区域。

另一方面，重新考虑 `1.1` 和 `1.2` 中的正则化，正则化式等价于带约束的目标函数中的约束项。例如以平方损失函数和 L^2 正则化为例，优化模型如下:

$$\begin{aligned} J(\boldsymbol{\theta}; \boldsymbol{X}, \boldsymbol{y}) &= \sum_{i=1}^{n} (y_i - \theta^\top \mathrm{x_i})^2 \\ s.t. \|\boldsymbol{\theta}\|_2^2 &<= \mathrm{C} \end{aligned} \tag{25}$$

采用拉格朗日乘积算子法可以转化为无约束优化问题，即：

$$J(\boldsymbol{\theta}; \boldsymbol{X}, \boldsymbol{y}) = \sum_{i=1}^{n} (y_i - \boldsymbol{\theta}^\top \boldsymbol{x}_i)^2 + \lambda(\|\boldsymbol{\theta}\|_2^2 - \mathrm{C}) \tag{26}$$

1.5 欠约束问题

机器学习中许多线性模型，如线性回归和 PCA，都依赖于矩阵 $\boldsymbol{X}^\top\boldsymbol{X}$ 求逆。如果 $\boldsymbol{X}^\top\boldsymbol{X}$ 不可逆，这些方法就会失效。这种情况下，**正则化的许多形式对应求逆 $\boldsymbol{X}^\top\boldsymbol{X} + \alpha\boldsymbol{I}$，且这个正则化矩阵是可逆的**。大多数正则化方法能够保证应用于欠定问题的迭代方法收敛。

例如线性回归的损失函数是平方误差之和：$(\boldsymbol{X}\boldsymbol{w} - \boldsymbol{y})^\top(\boldsymbol{X}\boldsymbol{w} - \boldsymbol{y})$。

我们添加 L^2 正则项后，目标函数变为 $(\boldsymbol{X}\boldsymbol{w} - \boldsymbol{y})^\top(\boldsymbol{X}\boldsymbol{w} - \boldsymbol{y}) + \frac{1}{2}\alpha\boldsymbol{w}^\top\boldsymbol{w}$。

这将普通方程的解从 $\boldsymbol{w} = (\boldsymbol{X}^\top\boldsymbol{X})^{-1}\boldsymbol{X}^\top\boldsymbol{y}$ 变为 $\boldsymbol{w} = (\boldsymbol{X}^\top\boldsymbol{X} + \alpha\boldsymbol{I})^{-1}\boldsymbol{X}^\top\boldsymbol{y}$。

2 数据增强

2.1 数据集增强

数据集增强是解决数据量有限的问题，**让机器学习模型泛化得更好的最好办法是使用更多的数据进行训练** (这在对象识别 (`object detection`) 问题很有效)。

方法:

- 类别不改变，平移不变性。比如对图像的平移，旋转，缩放。
- 注入噪声，可以使模型对噪声更健壮 (如去噪自编码器)。

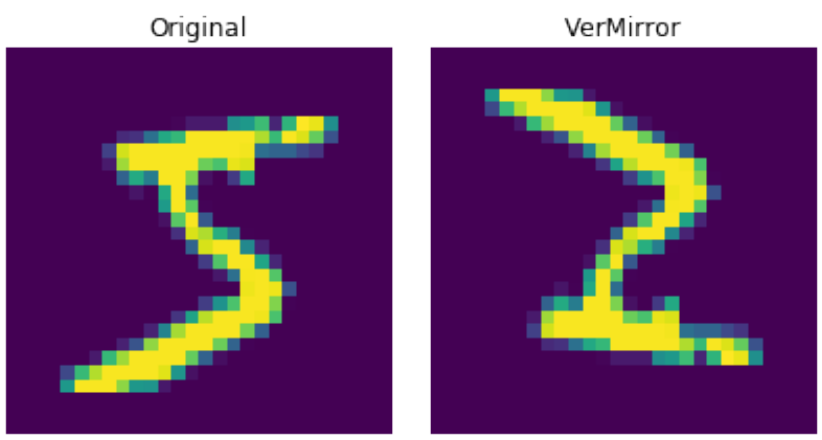
[11]:
```python
class Image(object):

    def __init__(self, image):
        self._set_params(image)

    def _set_params(self, image):
        self.img = image
        self.row = image.shape[0] # 图像高度
        self.col = image.shape[1] # 图像宽度
        self.transform = None

    def Translation(self, delta_x, delta_y):
        """
        平移。

        参数说明:
        delta_x: 控制左右平移，若大于 0 左移，小于 0 右移
```

```
        delta_y: 控制上下平移，若大于 0 上移，小于 0 下移
        """
        self.transform = np.array([[1, 0, delta_x],
                                   [0, 1, delta_y],
                                   [0,  0,  1]])

    def Resize(self, alpha):
        """
        缩放。

        参数说明:
        alpha: 缩放因子，不进行缩放设置为 1
        """
        self.transform = np.array([[alpha, 0, 0],
                                   [0, alpha, 0],
                                   [0,  0,  1]])

    def HorMirror(self):
        """
        水平镜像。
        """
        self.transform = np.array([[1,  0,  0],
                                   [0, -1, self.col-1],
                                   [0,  0,  1]])

    def VerMirror(self):
        """
        垂直镜像。
        """
        self.transform = np.array([[-1, 0, self.row-1],
                                   [0,  1,  0],
                                   [0,  0,  1]])

    def Rotate(self, angle):
        """
        旋转。

        参数说明:
        angle: 旋转角度
        """
        self.transform = np.array([[math.cos(angle),-math.sin(angle),0],
                                   [math.sin(angle), math.cos(angle),0],
                                   [    0,              0,         1]])

    def operate(self):
        temp = np.zeros(self.img.shape, dtype=self.img.dtype)
        for i in range(self.row):
            for j in range(self.col):
                temp_pos = np.array([i, j, 1])
                [x,y,z] = np.dot(self.transform, temp_pos)
                x = int(x)
                y = int(y)
                if x>=self.row or y>=self.col or x<0 or y<0:
                    temp[i,j,:] = 0
                else:
                    temp[i,j,:] = self.img[x,y]
        return temp

    def __call__(self, act):
```

```
        r = r"([a-zA-Z]*)=([^,)]*)"
        act_str = act.lower()
        kwargs = dict([(i, eval(j)) for (i, j) in re.findall(r, act_str)])
        if "translation" in act_str:
            self.Translation(**kwargs)
        elif "resize" in act_str:
            self.Resize(**kwargs)
        elif "hormirror" in act_str:
            self.HorMirror(**kwargs)
        elif "vermirror" in act_str:
            self.VerMirror(**kwargs)
        elif "rotate" in act_str:
            self.Rotate(**kwargs)
        return self.operate()
```

[12]:
```
"""
导入数据， 手写体数字
"""
def load_data(path="../data/mnist/mnist.npz"):
    f = np.load(path)
    X_train, y_train = f['x_train'], f['y_train']
    X_test, y_test = f['x_test'], f['y_test']
    f.close()
    return (X_train, y_train), (X_test, y_test)

(X_train, y_train), (X_test, y_test) = load_data()
```

[13]:
```
img = X_train[0].reshape(28, 28, 1)
Img = Image(img)
ax = plt.subplot(121)
plt.tight_layout()
plt.imshow(img.reshape(28,28))
plt.title('Original')
plt.axis('off')
ax = plt.subplot(122)
plt.imshow(Img('vermirror').reshape(28,28))
plt.title('VerMirror')
plt.axis('off')
plt.show()
```



2.2 噪声鲁棒性

- 将噪声**加入到输入**，等同于**数据集增强**。
- 将噪声**加入到权重**，能够**表现权重的不确定性**，这项技术主要用于循环神经网络。这可以被解释为关于权重的贝叶斯推断的随机实现，贝叶斯学习过程将权重视为不确定的，并且可以通过概率分布表示这种不确定性，向权重添加噪声是反映这种不确定性的一种实用的随机方法。
- 将噪声**加入到输出**，对噪声建模 (滤波)，**标签平滑**。由于大多数数据集的 $\boldsymbol{y}$ 标签都有一定错误，错误的 $\boldsymbol{y}$ 不利于最大化 $\log p(\boldsymbol{y} \mid \boldsymbol{x})$，避免这种情况的一种方法是显式地对标签上的噪声进行建模。

3 训练方案

3.1 半监督学习

监督学习指训练样本都是带标记的。然而在现实中，获取数据是容易的，但是收集到带标记的数据却是非常昂贵的。**半监督学习指的是既包含部分带标记的样本也有不带标记的样本**，通过这些数据来进行学习。在半监督学习的框架下，$P(\boldsymbol{x})$ 产生的未标记样本和 $P(\boldsymbol{x}, \boldsymbol{y})$ 中的标记样本都用于估计 $P(\boldsymbol{y} \mid \boldsymbol{x})$。在深度学习的背景下，半监督学习通常指的是学习一个表示 $h = f(\boldsymbol{x})$，**学习表示的目的是使相同类中的样本有类似的表示**。

我们可以构建这样一个模型，其中生成模型 $P(\boldsymbol{x})$ 或 $P(\boldsymbol{x}, \boldsymbol{y})$ 与判别模型 $P(\boldsymbol{y} \mid \boldsymbol{x})$ 共享参数，而不用分离无监督和监督部分。例如，我们可以这么做深度学习下的半监督学习，在损失函数中同时考虑两部分损失，一部分是有监督损失，另一部分是无监督损失 (无监督标签为 Pseudo Label，即直接取网络对无标签数据的预测的最大值为标签)。如果我们还鼓励网络学习数据内在的不变性，则可以构造无监督代价是对同一个输入在不同的正则和数据增强条件下的一致性。即要求在不同的条件下，模型的估计要一致。

3.2 多任务学习

多任务学习是基于共享表示，**把多个相关的任务放在一起学习**的一种机器学习方法。当模型的一部分被多个额外的任务共享时，这部分将被约束为良好的值，通常会带来**更好的泛化能力**。

从深度学习的观点看，**底层的先验知识**为：**能解释数据变化的因素中，某些因素是跨多个任务共享的**。可以考虑对复杂的问题，分解为简单且相互独立的子问题来单独解决。做单任务学习时，各个任务之间的模型空间是相互独立的，但这忽略了问题之间所富含的丰富的关联信息；而多任务学习便把多个相关的任务放在一起学习，学习过程中通过一个在浅层的共享表示来互相分享、互相补充学习到的领域相关的信息，互相促进学习，提升泛化的效果。

多任务学习是正则化的一种方法，对于**与主任务相关的任务**，可以看作是添加额外信息，**数据增强**；与**主任务不相关的任务**，可以看作是**引入噪音**，从而提高泛化。

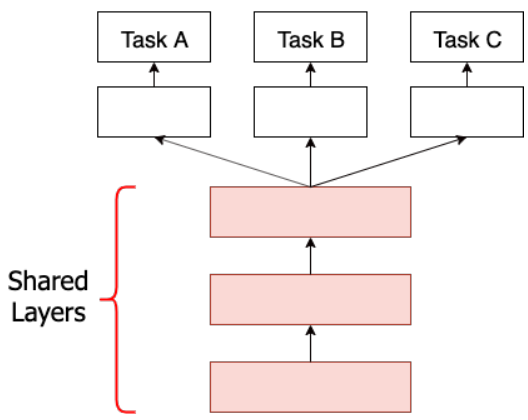


图 1. 多任务学习框架示意图，A, B, C 三个任务共享底层。

3.3 提前终止

当训练次数过多时会经常遭遇过拟合，此时训练误差会随时间推移减少，而验证集误差会再次上升。提前终止 (`Early Stopping`) 是一种交叉验证策略，我们将一部分训练集作为验证集 (`Validation Set`)。当我们看到验证集的性能越来越差时，我们立即停止对该模型的训练。

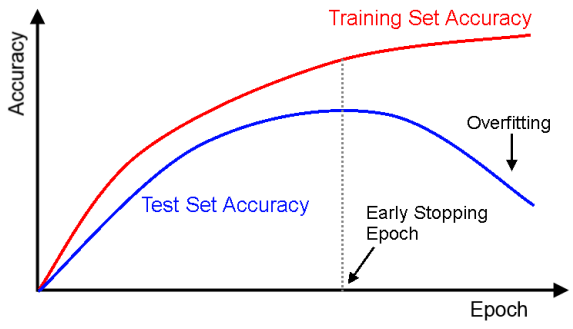


图 2. 学习曲线与提前终止。当测试集准确率下降时，可以提前终止。

提前终止的步骤如下：

- 将训练数据划分为训练集和测试集。
- 在训练集上训练，并在一段时间间隔内在测试集上预测。
- 当验证集上的误差高于上次时立即停止训练。
- 使用上一时刻中所得的权重来作为最终权重。

具体终止实现的策略可以多种：

- 策略一: 当泛化损失大于某个阈值时立即停止。

- 策略二: 假定过拟合仅仅发生在训练误差变化缓慢时。
- 策略三: 当 s 个连续时刻的泛化误差增大时停止。

一般而言，除非网络性能的小改进比训练时间更重要，否则选择第一个策略。

将测试集重新融入训练集 为了更好的利用所有数据，我们需要在完成提前终止的首次训练之后进行第二轮的训练，在第二轮中，所有的数据都被包括在内。对此我们有两个基本策略：

- 再次初始化模型，然后使用所有数据再次训练，在第二轮训练中，我们采用第一轮提前终止训练确定的最佳步数。
- 保持从第一轮训练获得的参数，然后使用验证集的数据继续训练，直到验证集的平均损失函数低于提前终止过程终止时的目标值。

提前终止具有正则化效果，其真正机制可理解**为将优化过程的参数空间限制在初始参数值 θ_0 的小邻域内**。考虑平方误差的简单线性模型，采用梯度下降法，可以证明假如学习率为 ϵ ，进行 τ 次训练迭代，则 $\frac{1}{\epsilon\tau}$ 等价于权重衰减系数 α 。

[14]:
```
"""
策略: 连续 4 个时刻验证集正确率没有增加
"""
def early_stopping(valid):
    """
    参数说明:
    valid: 验证集正确率列表
    """
    if len(valid) > 5:
        if valid[-1] < valid[-5] and valid[-2] < valid[-5] and valid[-3] < valid[-5] and valid[-4] < valid[-5]:
            return True
    return False
```

4 模型表示

4.1 参数绑定与共享

参数范数惩罚或约束是相对于固定区域或点，如 L^2 正则化是对参数偏离 0 进行惩罚。有时我们需要对模型参数之间的相关性进行惩罚，使模型参数尽量接近或者相等：

参数共享：强迫模型某些参数相等：

- 主要应用：卷积神经网络 (CNN)
- 优点：显著降低了 CNN 模型的参数数量 (CNN 模型参数数量经常是千万量级以上)，减少模型所占用的内存，并且显著提高了网络大小而不需要相应的增加训练数据。

4.2 稀疏表示

稀疏表示也是卷积神经网络经常用到的正则化方法。L^1 正则化会诱导稀疏的参数，使得许多参数为 0；而稀疏表示是惩罚神经网络的激活单元，稀疏化激活单元。换言之，稀疏表示是使得每个神经元的输入单元变得稀疏，很多输入是 0。如下图所示，相比于全连接层，隐藏层的 h 接受到稀疏的输入 x。

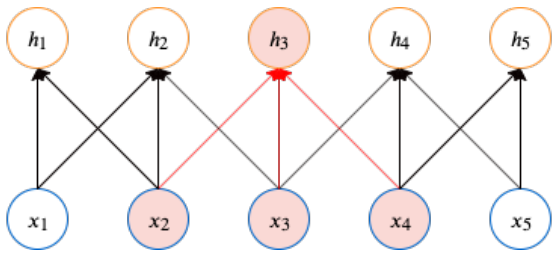


图 3. 稀疏表示示意图，每个隐藏层神经元最多连接三个输入单元。

4.3 Bagging 及其他集成方法

整合多个弱分类器，成为一个强大的分类器，这是集合学习的思想。集成学习主要有 Boosting 和 Bagging 两种思路。

4.3.1 Bagging 方法

在深度学习中，可以考虑用 Bagging 的思路来正则化。Bagging (`Bootstrap Aggregating`) 是通过重复采样生成新数据集 (Bootstrap)，再在新数据集上分别训练弱分类器，将多个弱分类器汇总成强分类器 (Aggregating)，从而可以降低泛化误差的技术，具体来说 Bagging 步骤：

- **构造 k 个不同的数据集**，每个数据集是从原始数据集中**重复采样**构成，和原始数据集具有相同数量的样本；
- 分别用这 k 个数据集去训练网络，**得到 k 个网络模型**；
- **最终输出的结果**可以用对 k 个网络模型的输出用**加权平均法**或者**投票法**来决定。

其中，我们通常选取的基础弱分类器是 CART 分类器 (见第五章)。而每个数据集 D_i 的构造：假设一共 m 个样本，则在单个数据集中 (需重复采样样本 m 次)，样本不会被采样到的概率约为 $(1-\frac{1}{m})^m \approx \frac{1}{e} = 36.8\%$。

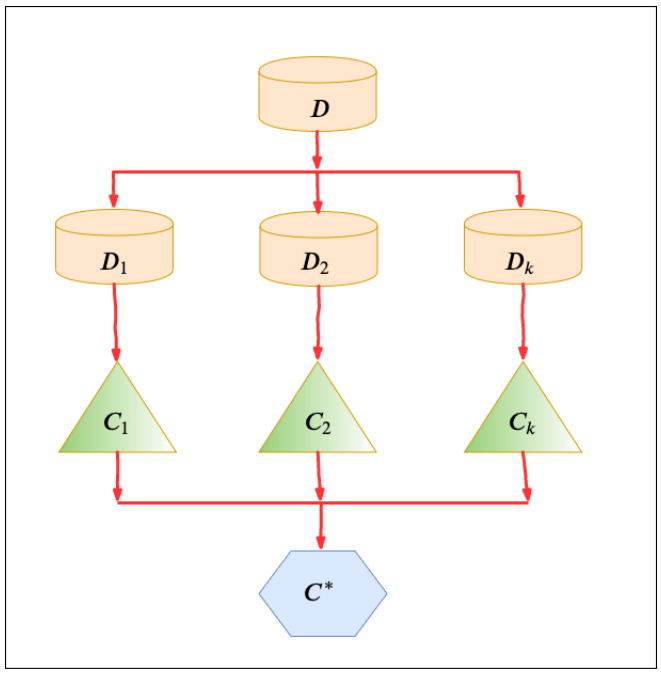


图 4. Bagging 方法示意图。

这种策略在机器学习中被称为模型平均 (`Model Averaging`)。模型平均是一个减少泛化误差的非常强大可靠的方法，例如我们假设有 k 个回归模型, 每个模型误差是 ϵ_i，误差服从零均值、方差为 v 、协方差为 c 的多维正态分布，则模型平均预测的误差为 $\frac{1}{k}\sum_i \epsilon_i$，均方误差的期望为

$$\mathbb{E}\left[(\frac{1}{k}\sum_i \epsilon_i)^2\right] = \frac{1}{k^2}\mathbb{E}\left[\sum_i(\epsilon_i^2 + \sum_{i\neq j}\epsilon_i\epsilon_j)\right] = \frac{1}{k}v + \frac{k-1}{k}c \tag{27}$$

可见，在误差完全相关即 $c = v$ 的情况下，均方误差为 v，模型平均没有帮助。在误差完全不相关即 $c = 0$ 时，模型平均的均方误差的期望仅为 $\frac{1}{k}v$ ，这说明集成平方误差的期望随集成规模的增大而线性减少。

Bagging 方法的优缺点：

- 优点：Bagging 利用集成学习的优势，其中多个弱学习器的表现优于单个强学习器。它有助于减少方差，从而帮助我们避免过拟合。
- 缺点：该模型缺乏可解释性。如果建模不当，则可能存在高偏差的问题 (弱分类器很差，集成后预测性能仍然很差)。另一个重要的缺点是，尽管 Bagging 可以提高准确性，但计算昂贵。尤其在网络模型中使用，我们的网络模型本来就比较复杂，参数很多，现在参数又增加了 k 倍，从而导致训练这样的网络要花更加多的时间和空间。因此一般 k 的个数不能太多，比如 5 - 10 个就可以了。

4.3.2 随机森林

随机森林是对 Bagging 决策树的改进。像 CART 这样的决策树的问题在于它们的贪婪性，他们使用最小化误差的贪婪算法选择的要分割的特征变量。这样，即使使用 Bagging，各个决策树也可以具有很多结构相似性，进而它们的预测具有高度相关性。

如果来自子模型的预测不相关或充其量是弱相关的，则将多个模型中的预测组合在一起会更好。这是随机森林改动的动机，它更改了学习子模型的方式。这是一个简单的调整。在 CART 中，选择分割点时，允许学习算法浏览所有特征变量和所有变量值，以选择最佳的分割点。随机森林算法更改了此过程，学习算法仅限于要搜索特征的随机样本。

每个子模型可以搜索的特征的数目 p 是模型的超参数，可以用交叉验证优化，默认参数为：

- 分类问题：$p = \sqrt{n}$
- 回归问题：$p = n/3$

其中 p 是可以在分割点搜索的随机选择特征的数目，n 是输入特征变量的数目。

接下来，描述一下随机森林的训练过程：

1. 从训练数据中创建数据子集，从全部 n 个特征中随机选择 p 个特征，其中 $p \ll n$。
2. 在 p 个特征中，执行 CART 的步骤，生成一棵决策树。
3. 通过重复步骤 1 至 2 执行 k 次来构建森林，以创建数量为 k 的树。

随机森林的预测过程：

1. 采取测试特征并使用每个随机创建的决策树的规则来预测结果并存储预测结果 (目标)。
2. 计算每个预测目标的票数。
3. 将最高得票的预测目标视为随机森林算法的最终预测。

与其他分类技术相比，随机森林的优点：

- 对于分类问题中的应用，随机森林将避免过拟合问题。
- 对于分类和回归任务，可以使用相同的随机森林算法。
- 随机森林可用于从训练数据集中识别最重要的特征，换句话说，就是特征工程。

4.3.3 方法解决过拟合

在“偏差和方差”中我们介绍到 (第五章)，一个分类器的总期望误差是由偏差和方差这两部分之和构成的。Bagging 方法能够通过减少方差分量来降低期望误差值，包含的分类器越多，方差减少量就越大。

从泛化稳定性的角度来看，当学习方法不稳定时，即输入数据的微小变化能导致生成差别相当大的分类器。正则化技术是通过调整输入权重分配和校准输出的思路来缓解该问题的。而集成学习提供了另一种解决问题的思路，实际上，尽可能地使学习方法不稳定，增加集成分类器中的多样性，有助于提高分类器性能。当对决策树使用 Bagging 技术时，决策树已经是不稳定的，如果不对树进行剪枝，经常可以获得更好的性能，而这会使决策树变得不稳定。

更多关于集成方法及其衍生会在本章最后补充介绍（ Boosting，GBDT，XGBoost ）。

[15]:
```
from chapter5 import ClassificationTree
```

[16]:
```
# 进度条
bar_widgets = [
    'Training: ', progressbar.Percentage(), ' ', progressbar.Bar(marker="-", left="[", right="]"),
    ' ', progressbar.ETA()
]

def get_random_subsets(X, y, n_subsets, replacements=True):
    """ 从训练数据中抽取数据子集 (默认可重复抽样)"""
    n_samples = np.shape(X)[0]
    # 将 X 和 y 拼接，并将元素随机排序
    Xy = np.concatenate((X, y.reshape((1, len(y))).T), axis=1)
    np.random.shuffle(Xy)
    subsets = []
    # 如果抽样时不重复抽样，可以只使用 50% 的训练数据；如果抽样时可重复抽样，使用全部的训练数据，默认可重复抽样
    subsample_size = int(n_samples // 2)
    if replacements:
        subsample_size = n_samples
    for _ in range(n_subsets):
        idx = np.random.choice(
            range(n_samples),
            size=np.shape(range(subsample_size)),
            replace=replacements)
        X = Xy[idx][:, :-1]
        y = Xy[idx][:, -1]
        subsets.append([X, y])
    return subsets


class Bagging():
    """
    Bagging 分类器。使用一组分类树，这些分类树使用特征训练数据的随机子集。
    """
    def __init__(self, n_estimators=100, max_features=None, min_samples_split=2,
                 min_gain=0, max_depth=float("inf")):
        self.n_estimators = n_estimators    # 树的数目
        self.min_samples_split = min_samples_split   # 分割所需的最小样本数
        self.min_gain = min_gain            # 分割所需的最小纯度 (最小信息增益)
        self.max_depth = max_depth          # 树的最大深度
        self.progressbar = progressbar.ProgressBar(widgets=bar_widgets)
        # 初始化决策树
        self.trees = []
        for _ in range(n_estimators):
```

```
            self.trees.append(
                ClassificationTree(
                    min_samples_split=self.min_samples_split,
                    min_impurity=min_gain,
                    max_depth=self.max_depth))

    def fit(self, X, y):
        # 对每棵树选择数据集的随机子集
        subsets = get_random_subsets(X, y, self.n_estimators)
        for i in self.progressbar(range(self.n_estimators)):
            X_subset, y_subset = subsets[i]
            # 用特征子集和真实值训练一棵子模型 (这里的数据也是训练数据集的随机子集)
            self.trees[i].fit(X_subset, y_subset)

    def predict(self, X):
        y_preds = np.empty((X.shape[0], len(self.trees)))
        # 每棵决策树都在数据上预测
        for i, tree in enumerate(self.trees):
            # 基于特征做出预测
            prediction = tree.predict(X)
            y_preds[:, i] = prediction
        y_pred = []
        # 对每个样本，选择最常见的类别作为预测
        for sample_predictions in y_preds:
            y_pred.append(np.bincount(sample_predictions.astype('int')).argmax())
        return y_pred

    def score(self, X, y):
        y_pred = self.predict(X)
        accuracy = np.sum(y == y_pred, axis=0) / len(y)
        return accuracy
```

[17]:

```
class RandomForest():
    """
    随机森林分类器。使用一组分类树，这些分类树使用特征的随机子集训练数据的随机子集。
    """
    def __init__(self, n_estimators=100, max_features=None, min_samples_split=2,
                 min_gain=0, max_depth=float("inf")):
        self.n_estimators = n_estimators    # 树的数目
        self.max_features = max_features    # 每棵树的最大使用特征数
        self.min_samples_split = min_samples_split    # 分割所需的最小样本数
        self.min_gain = min_gain            # 分割所需的最小纯度 (最小信息增益)
        self.max_depth = max_depth          # 树的最大深度
        self.progressbar = progressbar.ProgressBar(widgets=bar_widgets)
        # 初始化决策树
        self.trees = []
        for _ in range(n_estimators):
            self.trees.append(
                ClassificationTree(
                    min_samples_split=self.min_samples_split,
                    min_impurity=min_gain,
                    max_depth=self.max_depth))

    def fit(self, X, y):
        n_features = np.shape(X)[1]
        # 如果 max_features 没有定义，取默认值 sqrt(n_features)
        if not self.max_features:
            self.max_features = int(math.sqrt(n_features))
        # 对每棵树选择数据集的随机子集
```

```
        subsets = get_random_subsets(X, y, self.n_estimators)
        for i in self.progressbar(range(self.n_estimators)):
            X_subset, y_subset = subsets[i]
            # 选择特征的随机子集
            idx = np.random.choice(range(n_features), size=self.max_features, replace=True)
            # 保存特征的索引用于预测
            self.trees[i].feature_indices = idx
            # 选择索引对应的特征
            X_subset = X_subset[:, idx]
            # 用特征子集和真实值训练一棵子模型 (这里的数据也是训练数据集的随机子集)
            self.trees[i].fit(X_subset, y_subset)

    def predict(self, X):
        y_preds = np.empty((X.shape[0], len(self.trees)))
        # 每棵决策树都在数据上预测
        for i, tree in enumerate(self.trees):
            # 使用该决策树训练使用的特征
            idx = tree.feature_indices
            # 基于特征做出预测
            prediction = tree.predict(X[:, idx])
            y_preds[:, i] = prediction
        y_pred = []
        # 对每个样本，选择最常见的类别作为预测
        for sample_predictions in y_preds:
            y_pred.append(np.bincount(sample_predictions.astype('int')).argmax())
        return y_pred

    def score(self, X, y):
        y_pred = self.predict(X)
        accuracy = np.sum(y == y_pred, axis=0) / len(y)
        return accuracy
```

用自定义的 Bagging，乳腺癌数据集测试

[18]:
```
column_names = ['Sample code number','Clump Thickness',
                'Uniformity of Cell Size','Uniformity of Cell Shape',
                'Marginal Adhesion','Single Epithelial Cell Size',
                'Bare Nuclei','Bland Chromatin','Normal Nucleoli','Mitoses','Class']
data = pd.read_csv('../data/cancer/breast-cancer-wisconsin.data',names=column_names)
data = data.replace(to_replace='?',value=np.nan)  # 非法字符替代
data = data.dropna(how='any')  # 去掉空值，any；出现空值行则删除
print(data.shape)
# 随机采样 25% 的数据用于测试，剩下 75% 用于构建训练集
X_train,X_test,y_train,y_test = train_test_split(data[column_names[1:10]], data[column_names[10]],
                                                 test_size=0.25, random_state=1111)
# 再查看训练样本的数量和类别分布
print(y_train.value_counts())
# 修改标签为 0 和 1
print(y_train.shape)
y_train[y_train==2] = 0
y_train[y_train==4] = 1
y_test[y_test==2] = 0
y_test[y_test==4] = 1
# 再查看训练样本的数量和类别分布
print(y_train.value_counts())
# 数据标准化预处理，保证每个维度特征均值为 0，方差为 1
ss = StandardScaler()
X_train = ss.fit_transform(X_train)
X_test = ss.transform(X_test)
```

```
y_train = y_train.as_matrix()
y_test = y_test.as_matrix()
```

```
(683, 11)
2    328
4    184
Name: Class, dtype: int64
(512,)
0    328
1    184
Name: Class, dtype: int64
```

[19]:
```
model = Bagging(n_estimators=20)
model.fit(X_train, y_train)
print(model.score(X_test, y_test))
```

```
Training: 100% [-----------------------------------------------] Time: 0:00:04

0.9473684210526315
```

用 sklearn 的 Bagging，乳腺癌数据集测试

[20]:
```
from sklearn.ensemble import BaggingClassifier
from sklearn import tree
model = BaggingClassifier(tree.DecisionTreeClassifier(random_state=1))
model.fit(X_train, y_train)
model.score(X_test,y_test)
```

[20]: 0.9473684210526315

用自定义的随机森林，乳腺癌数据集测试

[21]:
```
model = RandomForest(n_estimators=20)
model.fit(X_train, y_train)
print(model.score(X_test, y_test))
```

```
Training: 100% [-----------------------------------------------] Time: 0:00:01

0.9649122807017544
```

用 sklearn 的随机森林，乳腺癌数据集测试

[22]:
```
from sklearn.ensemble import RandomForestClassifier
model= RandomForestClassifier(random_state=1)
model.fit(X_train, y_train)
model.score(X_test,y_test)
```

[22]: 0.9415204678362573

4.4 Dropout

Dropout 的原理为：在每个迭代过程中，**随机选择某些神经元**，并且删除它们在网络中的前向和后向连接，相当于是**“去掉”这些神经元**。如图 5 所示，在每批样本训练时，将原始网络中部分隐藏层单元 “去掉”。当然，Dropout 并不意味着这些神经元永远的消失了，在下一批数据迭代前，我们会把网络恢复成最初的全连接网络，然后再用随机的方法去掉部分隐藏层的神经元，接着去迭代更新 $\boldsymbol{W}$, $\boldsymbol{b}$。Dropout 思想可以理解为每次训练时放弃部分神经元对剩下的神经元加重训练，使剩下的神经元具有更强的能力。

每个迭代过程都会有不同的神经元节点的组合，从而导致不同的输出。这可以看成机器学习中的集成方法 (`Ensemble Technique`)。集成模型一般优于单一模型，因为它们可以捕获更多的随机性；相似地，Dropout 使得神经网络模型优于正常的模型。

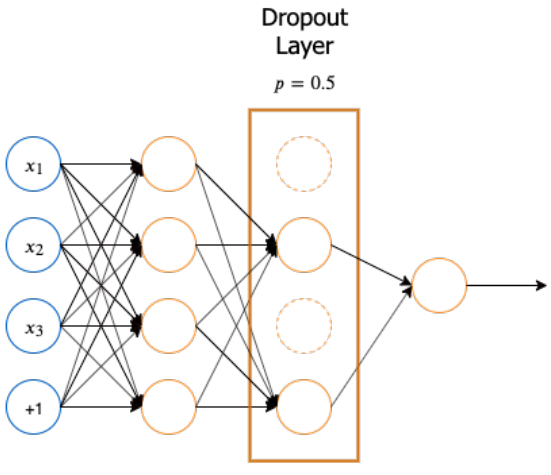


图 5. Dropout 示意图，此处 p 为 0.5，训练阶段随机“去掉”一半神经元。

Dropout 的实现步骤：

- 每次加载小批量样本，然后随机采样二值掩码，对于每个单元，掩码是独立采样的。**(将一些单元的输出乘零就能有效的删除一个单元)**；
- 传统的 Dropout，在训练阶段，用了 Dropout 的层，每个神经元以 p 的概率保留（或以 $1-p$ 的概率关闭），然后在测试阶段，不执行 Dropout，也就是所有神经元都不关闭，但是对训练阶段应用了 Dropout 的层上的神经元，为保证强度一致其输出激活值要乘以 p。**原理：为保证训练阶段与测试阶段强度一致，预测时对于每个隐层的输出，都乘以概率** p。可以从数学期望的角度去理解，我们考虑一个神经元的输出为 x (没有 Dropout 的情况下)，则它输出的数学期望为 $px+(1-p)0$，于是在测试阶段，我们直接把每个输出 x 都变换为 px，是可以保持一样的数学期望。而现在我们在训练阶段应用 Dropout 时并没有让神经元 a 的输出激活值除以 p，因此其期望值为 pa，在测试阶段如果不用 Dropout，所有神经元都保留，则输出期望值为 x，为了让测试阶段神经元的输出期望值和训练阶段保持一致 (这样才能正确评估训练出的模型)，就要给测试阶段的输出激活值乘上 p，使其输出期望值保持为 px；
- 现在主流的方法是 Inverted Dropout，和传统的 Dropout 方法有两点不同：在训练阶段，对执行了 Dropout 操作的层，其输出激活值要除以 p；测试阶段则不执行任何操作，既不执行 Dropout，也不用对神经元的输出乘 p。
- 其余部分，与之前一样，运行前向传播、反向传播和学习更新。

Dropout 不仅可以应用在隐含层，也可以应用在输入层。选择保留多少单元的概率值 p 是一个超参数。通常输入单元被保留的概率为 0.8，隐藏单元被保留的概率为 0.5。

Dropout 优点：

- 计算方便，训练过程中使用 Dropout 产生 n 个 (神经单元数目) 随机二进制数与状态相乘即可。
- 适用广 (几乎在所有使用分布式表示且可以用随机梯度下降训练的模型上都表现很好，如前馈神经网络、概率模型、受限波尔兹曼机、循环神经网络等)。
- 相比其他正则化方法 (如权重衰减、过滤器约束和稀疏激活) 更有效，也可与其他形式的正则化合并，得到进一步提升。

Dropout 缺点：

- 不适合宽度太窄的网络，否则大部分网络没有输入到输出的路径。
- 不适合训练数据太小（如小于 5000）的网络，训练数据太小时，Dropout 没有其他方法表现好。
- 不适合非常大的数据集，数据集大的时候正则化效果有限 (大数据集本身的泛化误差就很小)，使用 Dropout 的代价可能超过正则化的好处。

Dropout 与 Bagging 的比较

Dropout 模型中的参数 $\boldsymbol{W}$，$\boldsymbol{b}$ 是共享的，Dropout 下网络迭代时，更新的是同一组 $\boldsymbol{W}$，$\boldsymbol{b}$；而 Bagging 正则化中每个模型有自己的一套参数，相互之间是独立的。当然两种策略都是每次使用基于原始数据集得到的分批的数据集来训练模型。

[23]:
```python
class Dropout(ABC):

    def __init__(self, wrapped_layer, p):
        """
        参数说明:
        wrapped_layer: 被 dropout 的层
        p: 神经元保留率
        """
        super().__init__()
        self._base_layer = wrapped_layer
        self.p = p
        self._init_wrapper_params()

    def _init_wrapper_params(self):
        self._wrapper_derived_variables = {"dropout_mask": None}
        self._wrapper_hyperparams = {"wrapper": "Dropout", "p": self.p}
```

```
    def flush_gradients(self):
        """
        函数作用：调用 base layer 重置更新参数列表
        """
        self._base_layer.flush_gradients()

    def update(self):
        """
        函数作用：调用 base layer 更新参数
        """
        self._base_layer.update()

    def forward(self, X, is_train=True):
        """
        参数说明：
        X：输入数组；
        is_train：是否为训练阶段，bool 型；
        """
        mask = np.ones(X.shape).astype(bool)
        if is_train:
            mask = (np.random.rand(*X.shape) < self.p) / self.p
            X = mask * X
        self._wrapper_derived_variables["dropout_mask"] = mask
        return self._base_layer.forward(X)

    def backward(self, dLda):
        return self._base_layer.backward(dLda)

    @property
    def hyperparams(self):
        hp = self._base_layer.hyperparams
        hpw = self._wrapper_hyperparams
        if "wrappers" in hp:
            hp["wrappers"].append(hpw)
        else:
            hp["wrappers"] = [hpw]
        return hp
```

[24]:
```
def minibatch(X, batchsize=256, shuffle=True):
    """
    函数作用：将数据集分割成 batch， 基于 mini batch 训练，具体可见第 8 章。
    """
    N = X.shape[0]
    idx = np.arange(N)
    n_batches = int(np.ceil(N / batchsize))
    if shuffle:
        np.random.shuffle(idx)
    def mb_generator():
        for i in range(n_batches):
            yield idx[i * batchsize : (i + 1) * batchsize]
    return mb_generator(), n_batches

class DFN(object):

    def __init__(
        self,
        hidden_dims_1=None,
        hidden_dims_2=None,
        optimizer="sgd(lr=0.01)",
```

```
        init_w="std_normal",
        p=1.0,
        loss=CrossEntropy()
    ):
        self.optimizer = optimizer
        self.init_w = init_w
        self.loss = loss
        self.p = p
        self.hidden_dims_1 = hidden_dims_1
        self.hidden_dims_2 = hidden_dims_2
        self.is_initialized = False

    def _set_params(self):
        """
        函数作用：模型初始化
        FC1 -> Sigmoid -> FC2 -> Softmax
        """
        self.layers = OrderedDict()
        self.layers["FC1"] = Dropout(  # 这里引入 dropout
            FullyConnected(
                n_out=self.hidden_dims_1,
                acti_fn="sigmoid",
                init_w=self.init_w,
                optimizer=self.optimizer
            ), self.p
        )
        self.layers["FC2"] = FullyConnected(
                n_out=self.hidden_dims_2,
                acti_fn="affine(slope=1, intercept=0)",
                init_w=self.init_w,
                optimizer=self.optimizer
        )
        self.is_initialized = True

    def forward(self, X_train, is_train=True):
        Xs = {}
        out = X_train
        for k, v in self.layers.items():
            Xs[k] = out
            try:  # 考虑 dropout
                out = v.forward(out, is_train=is_train)
            except:
                out = v.forward(out)
        return out, Xs

    def backward(self, grad):
        dXs = {}
        out = grad
        for k, v in reversed(list(self.layers.items())):
            dXs[k] = out
            out = v.backward(out)
        return out, dXs

    def update(self):
        """
        函数作用：梯度更新
        """
        for k, v in reversed(list(self.layers.items())):
            v.update()
```

```
        self.flush_gradients()

    def flush_gradients(self, curr_loss=None):
        """
        函数作用：更新后重置梯度
        """
        for k, v in self.layers.items():
            v.flush_gradients()

    def fit(self, X_train, y_train, n_epochs=20, batch_size=64, verbose=False):
        """
        参数说明：
        X_train：训练数据
        y_train：训练数据标签
        n_epochs：epoch 次数
        batch_size：每次 epoch 的 batch size
        verbose：是否每个 batch 输出损失
        """
        self.verbose = verbose
        self.n_epochs = n_epochs
        self.batch_size = batch_size
        if not self.is_initialized:
            self.n_features = X_train.shape[1]
            self._set_params()
        prev_loss = np.inf
        for i in range(n_epochs):
            loss, epoch_start = 0.0, time.time()
            batch_generator, n_batch = minibatch(X_train, self.batch_size, shuffle=True)
            for j, batch_idx in enumerate(batch_generator):
                batch_len, batch_start = len(batch_idx), time.time()
                X_batch, y_batch = X_train[batch_idx], y_train[batch_idx]
                out, _ = self.forward(X_batch, is_train=True)
                y_pred_batch = softmax(out)
                batch_loss = self.loss(y_batch, y_pred_batch)
                grad = self.loss.grad(y_batch, y_pred_batch)
                _, _ = self.backward(grad)
                self.update()
                loss += batch_loss
                if self.verbose:
                    fstr = "\t[Batch {}/{}] Train loss: {:.3f} ({:.1f}s/batch)"
                    print(fstr.format(j + 1, n_batch, batch_loss, time.time() - batch_start))
            loss /= n_batch
            fstr = "[Epoch {}] Avg. loss: {:.3f}  Delta: {:.3f} ({:.2f}m/epoch)"
            print(fstr.format(i + 1, loss, prev_loss - loss, (time.time() - epoch_start) / 60.0))
            prev_loss = loss

    def evaluate(self, X_test, y_test, batch_size=128):
        acc = 0.0
        batch_generator, n_batch = minibatch(X_test, batch_size, shuffle=True)
        for j, batch_idx in enumerate(batch_generator):
            batch_len, batch_start = len(batch_idx), time.time()
            X_batch, y_batch = X_test[batch_idx], y_test[batch_idx]
            y_pred_batch, _ = self.forward(X_batch, is_train=False)
            y_pred_batch = np.argmax(y_pred_batch, axis=1)
            y_batch = np.argmax(y_batch, axis=1)
            acc += np.sum(y_pred_batch == y_batch)
        return acc / X_test.shape[0]

    @property
```

```
    def hyperparams(self):
        return {
            "init_w": self.init_w,
            "loss": str(self.loss),
            "optimizer": self.optimizer,
            "hidden_dims_1": self.hidden_dims_1,
            "hidden_dims_2": self.hidden_dims_2,
            "dropout keep ratio": self.p,
            "components": {k: v.hyperparams for k, v in self.layers.items()}
        }
```

[25]:
```
"""
引入 dropout
"""
model = DFN(hidden_dims_1=200, hidden_dims_2=10, p=0.5)
model.fit(X_train, y_train, n_epochs=20, batch_size=64)
```

```
[Epoch 1] Avg. loss: 2.286  Delta: inf (0.02m/epoch)
[Epoch 2] Avg. loss: 2.215  Delta: 0.071 (0.02m/epoch)
[Epoch 3] Avg. loss: 2.017  Delta: 0.198 (0.02m/epoch)
[Epoch 4] Avg. loss: 1.681  Delta: 0.335 (0.02m/epoch)
[Epoch 5] Avg. loss: 1.356  Delta: 0.326 (0.02m/epoch)
[Epoch 6] Avg. loss: 1.124  Delta: 0.231 (0.02m/epoch)
[Epoch 7] Avg. loss: 0.965  Delta: 0.160 (0.02m/epoch)
[Epoch 8] Avg. loss: 0.851  Delta: 0.114 (0.02m/epoch)
[Epoch 9] Avg. loss: 0.779  Delta: 0.072 (0.02m/epoch)
[Epoch 10] Avg. loss: 0.718  Delta: 0.061 (0.02m/epoch)
[Epoch 11] Avg. loss: 0.677  Delta: 0.041 (0.02m/epoch)
[Epoch 12] Avg. loss: 0.643  Delta: 0.034 (0.02m/epoch)
[Epoch 13] Avg. loss: 0.615  Delta: 0.027 (0.02m/epoch)
[Epoch 14] Avg. loss: 0.591  Delta: 0.024 (0.02m/epoch)
[Epoch 15] Avg. loss: 0.573  Delta: 0.018 (0.02m/epoch)
[Epoch 16] Avg. loss: 0.554  Delta: 0.020 (0.02m/epoch)
[Epoch 17] Avg. loss: 0.541  Delta: 0.013 (0.02m/epoch)
[Epoch 18] Avg. loss: 0.530  Delta: 0.011 (0.02m/epoch)
[Epoch 19] Avg. loss: 0.525  Delta: 0.005 (0.02m/epoch)
[Epoch 20] Avg. loss: 0.516  Delta: 0.009 (0.02m/epoch)
```

[26]:
```
print("accuracy:{}".format(model.evaluate(X_test, y_test)))
```

```
accuracy:0.8889
```

5 样本测试

在正则化背景下，通过对抗训练可以减少原有独立同分布的测试集的错误率——在对抗扰动的训练集样本上训练网络。

主要原因之一是高度线性，神经网络主要是基于线性模块构建的。输入改变 ϵ ，则权重为 $\boldsymbol{w}$ 的线性函数将改变 $\epsilon\|\boldsymbol{w}\|_1$ ，对于高维的 $\boldsymbol{w}$ 这是一个非常大的数。对抗训练通过鼓励网络在训练数据附近的局部区域恒定来限制这一个高度敏感的局部线性行为。

6 补充材料

6.1 Boosting

在前面介绍的 Bagging 方法中，主要通过**对训练数据集进行随机采样**，以重新组合成不同的数据集，利用弱学习算法对不同的新数据集进行学习，得到一系列的预测结果，对这些预测结果做平均或者投票得到最终的预测。

而 Boosting 方法的主要目标是**将弱分类器“提升”为强分类器**，根据前一个弱分类器的训练效果对样本分布进行调整，再根据新的样本分布训练下一个弱分类器，如此迭代，最后将一系列弱分类器组合成一个强分类器。

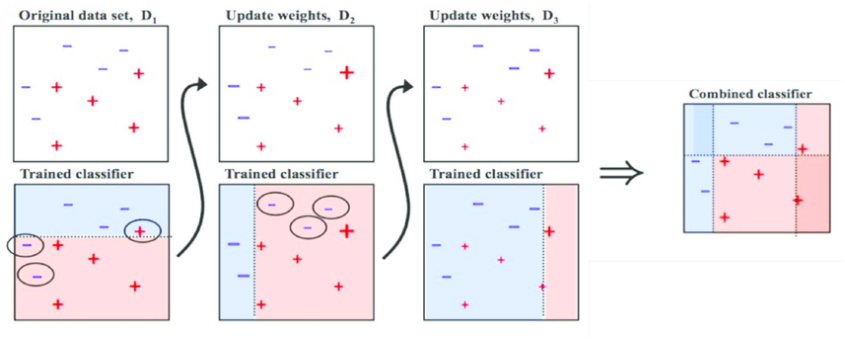


图 6. Boosting 方法示意图。

这里便存在几个问题：弱分类器选什么？如何调整样本分布？如何进行组合？针对这三个问题的不同回答就可以得到不同的 Boosting 方法。不过在具体介绍各种 Boosting 方法，我们需要先介绍前向分步加法模型。

6.1.1 前向分步加法模型

首先是加法模型 (`Addtive Model`)：

$$f(x) = \sum_{k=1}^{K} \beta_k \cdot b(\boldsymbol{x}; \gamma_k) \tag{28}$$

其中 $b(\boldsymbol{x}; \gamma_k)$ 是基函数，γ_k 是基函数的参数，β_k 是基函数的系数。

前向分步算法 (`Forward Stagewise Algorithm`)

在给定训练数据和损失函数 $\mathcal{L}(y, f(\boldsymbol{x}))$ 的情况下，学习加法模型 $f(\boldsymbol{x})$ 成为经验风险最小化即损失函数最小化的问题：

$$\min_{\beta_k, \gamma_k} \quad \sum_{i=1}^{m} \mathcal{L}\left(y^{(i)}, \sum_{k=1}^{K} \beta_k \cdot b(\boldsymbol{x}^{(i)}; \gamma_k)\right) \tag{29}$$

通常这是一个复杂的优化问题。前向分步算法求解这一优化问题的思路是：因为学习的是加法模型，如果能够从前向后，每一步只学习一个基函数及其系数，逐步逼近优化目标函数式，那么就可以简化优化的复杂度。具体地，每步只需优化如下损失函数：

$$\min_{\beta, \gamma} \quad \sum_{i=1}^{M} \mathcal{L}(y^{(i)}, \beta \cdot b(\boldsymbol{x}^{(i)}; \gamma)) \tag{30}$$

给定训练数据 $D = \{(\boldsymbol{x}^{(1)}, y^{(1)}), \cdots, (\boldsymbol{x}^{(m)}, y^{(m)})\}, \boldsymbol{x}^{(i)} \in \mathbb{R}^n, y^{(i)} \in \{-1, +1\}$，损失函数 $\mathcal{L}(y, f(\boldsymbol{x}))$ 和基函数集 $\{b(\boldsymbol{x}; \gamma)\}$。前向分步算法的步骤如下：

1. 初始化 $f_0(\boldsymbol{x}) = 0$。
2. 极小化损失函数 $(\beta_k, \gamma_k) = \arg\min_{\beta, \gamma} \sum_{i=1}^{m} \mathcal{L}(y^{(i)}, f_{k-1}(\boldsymbol{x}^{(i)}) + \beta \cdot b(\boldsymbol{x}; \gamma))$ 得到参数 β_k 和 γ_k。
3. 更新 $f_k(\boldsymbol{x}) = f_{k-1}(\boldsymbol{x}) + \beta_k \cdot b(\boldsymbol{x}; \gamma_k)$。
4. 重复步骤 2-3，$k = 1, 2, \cdots, K$。
5. 得到加法模型 $f(x) = f_K(x) = \sum_{k=1}^{K} \beta_k b(x; \gamma_k)$。

这样，前向分步算法将同时求解从 $k = 1$ 到 K 的所有参数 β_k, γ_k 的优化问题简化为逐次求解各个 β_k, γ_k 的优化问题。

6.1.2 AdaBoost 算法

回到三个问题，AdaBoost 的解决方案是什么？

- 弱分类器 (基分类器) 选什么？一般选法是决策树桩 (就是只有一层的决策树)。
- 如何调整样本分布？可以先赋予每个训练样本相同的权重；然后用弱分类器进行训练，每次训练后，对分类错误的样本加大权重 (重采样，具体做法后面再述)，使得在下一次的迭代中更加关注这些样本，如图 6 所示。
- 如何进行组合？组合方式即为加法模型 $f(\boldsymbol{x}) = sgn\left(\sum_{k=1}^{K} \alpha_k h_k(\boldsymbol{x})\right)$。其中 AdaBoost 的损失函数为指数损失 $\mathcal{L}(y, f(\boldsymbol{x})) = \exp(-yf(\boldsymbol{x}))$。

下面先分析为什么选择指数损失作为损失函数

假设 $f(\boldsymbol{x})$ 能使损失达到最小，对其求偏导：

$$\frac{\partial \exp(-yf(\boldsymbol{x}))}{\partial f(\boldsymbol{x})} = -\exp(-f(\boldsymbol{x}))P(y = 1|\boldsymbol{x}) + \exp(f(\boldsymbol{x}))P(y = -1|\boldsymbol{x}) \tag{31}$$

然后我们令导数为 0，可以求得：

$$f(\boldsymbol{x}) = \frac{1}{2} \ln \frac{P(y = 1 \mid \boldsymbol{x})}{P(y = -1 \mid \boldsymbol{x})} \tag{32}$$

接下来我们可以得到输出：

$$\begin{aligned} sgn(f(\boldsymbol{x})) &= sgn\left(\frac{1}{2}\ln\frac{P(y=1\mid\boldsymbol{x})}{P(y=-1\mid\boldsymbol{x})}\right) \\ &= \begin{cases} 1, P(y=1\mid x) > P(y=-1\mid x) \\ -1, P(y=1\mid x) < P(y=-1\mid x) \end{cases} \end{aligned} \tag{33}$$

这意味着 $sgn(f(\boldsymbol{x}))$ 达到了贝叶斯最优错误率。换言之，如果指数损失函数最小化，则分类错误率也将最小化。这说明指数损失函数是分类任务 0-1 损失函数的一致性替代函数。由于这个替代函数是单调连续可微函数，因此用它代替 0-1 损失函数作为优化目标。

基分类器权重 α 的更新

回到迭代过程，第一个基分类器 h_1 是通过直接将基学习算法用于初始训练数据 (初始数据分布) D_1 而得；此后迭代地生成 h_k 和 α_k。当基分类器 h_k 基于分布 D_k 产生后，该基分类器的权重 α_k 应当使得 $\alpha_k C_k$ 最小化指数损失函数

$$\begin{aligned} \min_{\alpha_k}\mathcal{L}(y, f_{k-1}(\boldsymbol{x}) + \alpha_k\cdot h_k(\boldsymbol{x})) &= \min_{\alpha_k}\mathbb{E}_{\boldsymbol{x}\sim D_k}[\exp\left(-y\left(f_{k-1}(\boldsymbol{x}) + \alpha_k\cdot h_k(\boldsymbol{x})\right)\right)] \\ &= \min_{\alpha_k}\mathbb{E}_{\boldsymbol{x}\sim D_k}[\exp\left(-y\left(\alpha_k\cdot h_k\left(\boldsymbol{x}\right)\right)\right)] \\ &= \exp(-\alpha_k)P_{\boldsymbol{x}\sim D_k}(y = h_k(\boldsymbol{x})) + \exp(\alpha_k)P_{\boldsymbol{x}\sim D_k}(y \neq h_k(\boldsymbol{x})) \\ &= \exp(-\alpha_k)(1-\varepsilon_k) + \exp(\alpha_k)\varepsilon_k \end{aligned} \tag{34}$$

其中，$\varepsilon_k = P_{\boldsymbol{x}\sim D_k}(h_k(\boldsymbol{x}) \neq y)$，表示错误率。我们接下来对该式求导，可以得到：

$$\frac{\partial\mathcal{L}(y, f_{k-1}(\boldsymbol{x}) + \alpha_k\cdot h_k\left(\boldsymbol{x}\right))}{\partial\alpha_k} = -\exp(-\alpha_k)(1-\varepsilon_k) + \exp(\alpha_k)\varepsilon_k \tag{35}$$

令导数为 0，有：

$$\alpha_k = \frac{1}{2}\ln\left(\frac{1-\varepsilon_k}{\varepsilon_k}\right) \tag{36}$$

这样我们便得到了在每次迭代，训练完基分类器后，加法模型中基函数的系数。

样本分布的更新与迭代过程中训练样本的生成

现在回到第二个问题的解答，我们如何调整样本分布。

在获得 f_{k-1} 之后样本的分布将进行调整，使下一轮基分类器 h_k 能纠正 f_{k-1} 的一些错误。理想的 h_k 能纠正 f_{k-1} 的全部错误，即最小化：

$$\begin{aligned} \mathcal{L}(y, f_{k-1}(\boldsymbol{x}) + h_k\left(\boldsymbol{x}\right)) &= \mathbb{E}_{\boldsymbol{x}\sim D}[\exp\left(-yf_{k-1}(\boldsymbol{x}) + h_k(\boldsymbol{x})\right)] \\ &= \mathbb{E}_{\boldsymbol{x}\sim D}[\exp\left(-yf_{k-1}(\boldsymbol{x})\right)\exp\left(-yh_k(\boldsymbol{x})\right)] \end{aligned} \tag{37}$$

注意 $y^2 = h_k^2(\boldsymbol{x}) = 1$ (因为输出为 1 或 -1)，上式可将 $\exp\left(-yh_k(\boldsymbol{x})\right)$ 泰勒展开：

$$\begin{aligned} \mathcal{L}(y, f_{k-1}(\boldsymbol{x}) + h_k(\boldsymbol{x})) &\simeq \mathbb{E}_{\boldsymbol{x}\sim D}[\exp\left(-yf_{k-1}(\boldsymbol{x})\right)\left(1 - yh_k(\boldsymbol{x}) + \frac{y^2h_k^2(\boldsymbol{x})}{2}\right)] \\ &= \mathbb{E}_{\boldsymbol{x}\sim D}[\exp\left(-yf_{k-1}(\boldsymbol{x})\right)\left(1 - f(x)h_t(x) + \frac{1}{2}\right)] \end{aligned} \tag{38}$$

于是，理想的基分类器应满足：

$$\begin{aligned} h_k\left(\boldsymbol{x}\right) &= \arg\min_h \mathcal{L}(y, f_{k-1}(\boldsymbol{x}) + h_k(\boldsymbol{x})) \\ &= \arg\min_h \mathbb{E}_{\boldsymbol{x}\sim D}[\exp\left(-yf_{k-1}(\boldsymbol{x})\right)yh_k(\boldsymbol{x})] \\ &= \arg\min_h \mathbb{E}_{\boldsymbol{x}\sim D}\left[\frac{(-yf_{k-1}(\boldsymbol{x}))}{\mathbb{E}_{\boldsymbol{x}\sim D}[\exp\left(-yf_{k-1}(\boldsymbol{x})\right)]}yh_k(\boldsymbol{x})\right] \end{aligned} \tag{39}$$

这里的 $\mathbb{E}_{\boldsymbol{x}\sim D}[\exp\left(-yf_{k-1}(\boldsymbol{x})\right)]$ 表示一个常数。令 D_k 表示一个分布：

$$D_k\left(\boldsymbol{x}\right) = \frac{\exp\left(-yf_{k-1}(\boldsymbol{x})\right)}{\mathbb{E}_{\boldsymbol{x}\sim D}[\exp\left(-yf_{k-1}(\boldsymbol{x})\right)]}D(\boldsymbol{x}) \tag{40}$$

因为根据数学期望的定义，这里等价于令：

$$\begin{aligned} h_k(\boldsymbol{x}) &= \arg\min_h \mathbb{E}_{\boldsymbol{x}\sim D}\left[\frac{\exp\left(-yf_{k-1}(\boldsymbol{x})\right)}{\mathbb{E}_{\boldsymbol{x}\sim D}[\exp\left(-yf_{k-1}(\boldsymbol{x})\right)]}yh_k(\boldsymbol{x})\right] \\ &= \arg\min_h \mathbb{E}_{\boldsymbol{x}\sim D_k}[yG_k(\boldsymbol{x})] \end{aligned} \tag{41}$$

由于 $y, h_k \in \{+1, -1\}$，所以有：$y\,h_k = 1 - 2I\left(y \neq h_k\left(\boldsymbol{x}\right)\right)$。

则理想的基分类器：

$$h_k\left(\boldsymbol{x}\right) = \arg\min_h \mathbb{E}_{\boldsymbol{x}\sim D_k}[I\left(y \neq h_k\left(\boldsymbol{x}\right)\right)] \tag{42}$$

可见，理想的 $h_k(\boldsymbol{x})$ 在分布 D_k 下最小化分类误差。因此，基分类器将基于分布 D_k 来训练，且针对 D_k 的分类误差应当小于 0.5。考虑到 D_k 和 D_{k+1} 的关系有：

$$\begin{aligned} D_{k+1}(\boldsymbol{x}) &= \frac{\exp(-yf_k(\boldsymbol{x}))}{\mathbb{E}_{\boldsymbol{x}\sim D}[\exp(-yf_k(\boldsymbol{x}))]} D(\boldsymbol{x}) \\ &= D(\boldsymbol{x}) \exp\left(-yf_{k-1}(\boldsymbol{x})\right) \frac{\exp\left(-y\alpha_k \cdot h_k(\boldsymbol{x})\right)}{\mathbb{E}_{\boldsymbol{x}\sim D}[\exp\left(-yf_k\left(\boldsymbol{x}\right)\right)]} \\ &= D_k(\boldsymbol{x}) \frac{\exp\left(-y\alpha_k \cdot h_k(\boldsymbol{x})\right) \mathbb{E}_{\boldsymbol{x}\sim D}[\exp\left(-yf_{k-1}(\boldsymbol{x})\right)]}{\mathbb{E}_{\boldsymbol{x}\sim D}[\exp\left(-yf_k(\boldsymbol{x})\right)]} \\ &= \frac{D_k(\boldsymbol{x}) \exp\left(-y\alpha_k \cdot h_k(\boldsymbol{x})\right)}{\mathbb{E}_{\boldsymbol{x}\sim D}\left[\frac{-yf_{k-1} - y\alpha_k h_k}{\mathbb{E}_{\boldsymbol{x}\sim D}\left[\exp(-yf_{k-1}\left(\boldsymbol{x}\right))\right]}\right]} \\ &= \frac{D_k(\boldsymbol{x}) \exp\left(-y\alpha_k \cdot h_k(\boldsymbol{x})\right)}{\mathbb{E}_{\boldsymbol{x}\sim D}[D_k(\boldsymbol{x}) \exp\left(-y\alpha_k \cdot h_k(\boldsymbol{x})\right)]} \end{aligned} \tag{43}$$

于是，我们便得到了样本分布的迭代更新公式。

自定义实现

[27]:
```
# 进度条
bar_widgets = [
    'Training: ', progressbar.Percentage(), ' ', progressbar.Bar(marker="-", left="[", right="]"),
    ' ', progressbar.ETA()
]
```

[28]:
```
# 决策树桩，作为 Adaboost 算法的弱分类器 (基分类器)
class DecisionStump():

    def __init__(self):
        self.polarity = 1            # 表示决策树桩默认输出的类别为 1 或是 -1
        self.feature_index = None    # 用于分类的特征索引
        self.threshold = None        # 特征的阈值
        self.alpha = None            # 表示分类器准确性的值

class Adaboost():
    """
    Adaboost 算法。
    """
    def __init__(self, n_estimators=5):
        self.n_estimators = n_estimators    # 将使用的弱分类器的数量
        self.progressbar = progressbar.ProgressBar(widgets=bar_widgets)

    def fit(self, X, y):
        n_samples, n_features = np.shape(X)
        # 初始化权重 (上文中的 D)，均为 1/N
        w = np.full(n_samples, (1 / n_samples))
        self.trees = []
        # 迭代过程
        for _ in self.progressbar(range(self.n_estimators)):
            tree = DecisionStump()
            min_error = float('inf')    # 使用某一特征值的阈值预测样本的最小误差
            # 迭代遍历每个 (不重复的) 特征值，查找预测 y 的最佳阈值
            for feature_i in range(n_features):
                feature_values = np.expand_dims(X[:, feature_i], axis=1)
                unique_values = np.unique(feature_values)
                # 将该特征的每个特征值作为阈值
                for threshold in unique_values:
                    p = 1
                    # 将所有样本预测默认值可以设置为 1
                    prediction = np.ones(np.shape(y))
                    # 低于特征值阈值的预测改为 -1
                    prediction[X[:, feature_i] < threshold] = -1
                    # 计算错误率
                    error = sum(w[y != prediction])
                    # 如果错误率超过 50%，我们反转决策树桩默认输出的类别
```

```
                # 比如 error = 0.8 => (1 - error) = 0.2,
                # 原来计算的是输出到类别 1 的概率，类别 1 作为默认类别。反转后类别 0 作为默认类别
                if error > 0.5:
                    error = 1 - error
                    p = -1
                # 如果这个阈值导致最小的错误率，则保存
                if error < min_error:
                    tree.polarity = p
                    tree.threshold = threshold
                    tree.feature_index = feature_i
                    min_error = error

            # 计算用于更新样本权值的 alpha 值，也是作为基分类器的系数。
            tree.alpha = 0.5 * math.log((1.0 - min_error) / (min_error + 1e-10))
            # 将所有样本预测默认值设置为 1
            predictions = np.ones(np.shape(y))
            # 如果特征值低于阈值，则修改预测结果，这里还需要考虑弱分类器的默认输出类别
            negative_idx = (tree.polarity * X[:, tree.feature_index] < tree.polarity * tree.threshold)
            predictions[negative_idx] = -1
            # 计算新权值，未正确分类样本的权值增大，正确分类样本的权值减小
            w *= np.exp(-tree.alpha * y * predictions)
            w /= np.sum(w)
            # 保存分类器
            self.trees.append(tree)

    def predict(self, X):
        n_samples = np.shape(X)[0]
        y_pred = np.zeros((n_samples, 1))
        # 用每一个基分类器预测样本
        for tree in self.trees:
            # 将所有样本预测默认值设置为 1
            predictions = np.ones(np.shape(y_pred))
            negative_idx = (tree.polarity * X[:, tree.feature_index] < tree.polarity * tree.threshold)
            predictions[negative_idx] = -1
            # 对基分类器加权求和，权重 alpha
            y_pred += tree.alpha * predictions
        # 返回预测结果 1 或 -1
        y_pred = np.sign(y_pred).flatten()
        return y_pred

    def score(self, X, y):
        y_pred = self.predict(X)
        accuracy = np.sum(y == y_pred, axis=0) / len(y)
        return accuracy
```

用自定义的 Adaboost，乳腺癌数据集测试

[29]:
```
column_names = ['Sample code number','Clump Thickness',
                'Uniformity of Cell Size','Uniformity of Cell Shape',
                'Marginal Adhesion','Single Epithelial Cell Size',
                'Bare Nuclei','Bland Chromatin','Normal Nucleoli','Mitoses','Class']
data = pd.read_csv('../data/cancer/breast-cancer-wisconsin.data',names=column_names)
data = data.replace(to_replace='?',value=np.nan)  # 非法字符替代
data = data.dropna(how='any')  # 去掉空值，any；出现空值行则删除
print(data.shape)
# 随机采样 25% 的数据用于测试，剩下 75% 用于构建训练集
X_train,X_test,y_train,y_test = train_test_split(data[column_names[1:10]], data[column_names[10]],
                                                 test_size=0.25, random_state=1111)
# 再查看训练样本的数量和类别分布
print(y_train.value_counts())
```

```
# 修改标签为 -1 和 1
print(y_train.shape)
y_train[y_train==2] = -1
y_train[y_train==4] = 1
y_test[y_test==2] = -1
y_test[y_test==4] = 1
# 再查看训练样本的数量和类别分布
print(y_train.value_counts())
# 数据标准化预处理，保证每个维度特征均值为 0，方差为 1
ss = StandardScaler()
X_train = ss.fit_transform(X_train)
X_test = ss.transform(X_test)
y_train = y_train.as_matrix()
y_test = y_test.as_matrix()
```

```
(683, 11)
2    328
4    184
Name: Class, dtype: int64
(512,)
-1    328
 1    184
Name: Class, dtype: int64
```

[30]:
```
model = Adaboost(n_estimators=20)
model.fit(X_train, y_train)
print(model.score(X_test, y_test))
```

```
Training: 100% [------------------------------------------------] Time: 0:00:00
0.935672514619883
```

用 sklearn 的 Adaboost，乳腺癌数据集测试

[31]:
```
from sklearn.ensemble import AdaBoostClassifier
skl_model = AdaBoostClassifier()
skl_model.fit(X_train, y_train)
print(skl_model.score(X_test, y_test))
```

```
0.9532163742690059
```

6.1.3 GBDT 算法

Boosting Tree 算法

回到三个问题，Boosting Tree 的解决方案是什么?

- 弱分类器 (基分类器) 选什么? 选法是回归树 (CART，见第五章)。
- 如何调整样本分布? 样本不做修改，但每一次迭代的样本标签为真实结果和当前模型预测结果的残差。
- 如何进行组合? 组合方式即为加法模型 $f(\boldsymbol{x}) = \sum_{k=1}^{K} h_k(\boldsymbol{x})$，其中基学习器的系数为 1。如果用于回归 GBDT 的损失函数为平方损失 (`Square loss`) $\mathcal{L}(y, f(\boldsymbol{x})) = (y - f(\boldsymbol{x}))^2$，如果用于分类 GBDT 的损失函数为交叉熵 (`Cross-entropy`)。

同样分析损失函数的选择

我们先从回归问题来看，假设在第 k 次迭代时我们已有的可加模型为 $f_{k-1}(\boldsymbol{x})$。那么在新一轮迭代中最好的学习目标什么? 学习真实结果和当前模型预测结果的残差 (`residuals`) r:

$$r(\boldsymbol{x}) = y - f(\boldsymbol{x}) \tag{44}$$

我们构造新一轮迭代过程中的数据集 $D = \{(\boldsymbol{x}^{(1)}, y^{(1)} - f(\boldsymbol{x}^{(1)})), \cdots, (\boldsymbol{x}^{(m)}, y^{(m)} - f(\boldsymbol{x}^{(m)}))\}, \boldsymbol{x}^{(i)} \in \mathbb{R}^n, y^{(i)} \in \mathbb{R}$。经过新一轮迭代训练得到的基学

习器 $C_k(\boldsymbol{x})$ 如果是完美学习的，则等于当前的残差项 $r_{k-1}(\boldsymbol{x})$。这样一来，我们的可加模型就变为：

$$\begin{aligned} f_k(\boldsymbol{x}) &= f_{k-1}(\boldsymbol{x}) + h_k(\boldsymbol{x}) \\ &= f_{k-1}(\boldsymbol{x}) + r_{k-1}(\boldsymbol{x}) \\ &= y \end{aligned} \tag{45}$$

这样一来，我们就可以得到完美的可加模型。那为什么选择平方损失，但平方损失有个很大的问题，对离群点 (`Outliers`) 呢？首先，看一下在第 k 次迭代时的损失函数 (优化目标)：

$$\begin{aligned} \mathcal{L}(y, f_k(\boldsymbol{x})) &= (y - f_{k-1}(\boldsymbol{x}) - h_k(\boldsymbol{x}))^2 \\ &= (r_{k-1}(\boldsymbol{x}) - h_k(\boldsymbol{x}))^2 \end{aligned} \tag{46}$$

所以，对于回归问题来说，如果是在平方损失函数的前提下，每一步确实只需要拟合当前模型的残差就可以了。至于分类问题求解，后面会进行介绍。

GBDT 算法

GBDT 是 Boosting Tree 的改进方法。回到三个问题，GBDT 的解决方案是什么？

- 弱分类器 (基分类器) 选什么？选法是回归树 (CART)。
- 如何调整样本分布？样本不做修改，但每一次迭代的样本标签为残差的负梯度。
- 如何进行组合？组合方式即为加法模型 $f(\boldsymbol{x}) = \sum_{k=1}^{K} h_k(\boldsymbol{x})$，其中基学习器的系数为 1。如果用于回归 GBDT 的损失函数为平方损失 (`Square loss`) $\mathcal{L}(y, f(\boldsymbol{x})) = (y - f(\boldsymbol{x}))^2$，绝对损失 (`Absolute loss`)，Huber 损失 (`Huber loss`)。如果用于分类 GBDT 的损失函数为交叉熵 (`Cross-entropy`)。

学习目标：残差的负梯度

先考虑第二个问题，残差的负梯度是什么？我们回到平方损失，假设在第 k 次迭代时我们已有的可加模型为 $f_{k-1}(\boldsymbol{x})$，于是会有：

$$\begin{aligned} \frac{\partial \mathcal{L}}{\partial f_{k-1}(\boldsymbol{x})} &= \frac{\partial (y - f_{k-1}(\boldsymbol{x}))^2}{\partial f_{k-1}(\boldsymbol{x})} \\ &= -(y - f_{k-1}(\boldsymbol{x})) \\ &= -r_k(\boldsymbol{x}) \end{aligned} \tag{47}$$

所以，我们在 Boosting Tree 算法中要学习的目标残差就等于负的梯度项 $-\frac{\partial \mathcal{L}}{\partial f_{k-1}(\boldsymbol{x})}$。所以我们可以考虑将学习目标从残差到残差的梯度项。但为什么要这么做？

现在，我们从泰勒展开的角度来分析。首先，回顾一阶泰勒展开公式：

$$f(x) \approx f(x_0) + f'(x_0)\Delta x \tag{48}$$

我们可以将在第 k 次迭代中已有的可加模型 $f_{k-1}(\boldsymbol{x})$ 视作 x_0，将需要学习的基学习器 $h_k(\boldsymbol{x})$ 视作 Δx。那么我们的损失函数可以写作：

$$\mathcal{L}(y, f_{k-1}(\boldsymbol{x}) + h_k(\boldsymbol{x})) \approx \mathcal{L}(y, f_{k-1}(\boldsymbol{x})) + \frac{\partial \mathcal{L}}{\partial f_{k-1}(\boldsymbol{x})} h_k(\boldsymbol{x}) \tag{49}$$

而在第 $k-1$ 次迭代后的损失为 $\mathcal{L}(y, f_{k-1}(\boldsymbol{x}))$。所以，经过第 k 次迭代后，损失的变化为 $\frac{\partial \mathcal{L}}{\partial f_{k-1}(\boldsymbol{x})} h_k(\boldsymbol{x})$。我们是希望损失越来越小的，而如果令 $h_k(\boldsymbol{x}) = -\frac{\partial \mathcal{L}}{\partial f_{k-1}(\boldsymbol{x})}$，则损失变化量为 $-(\frac{\partial \mathcal{L}}{\partial f_{k-1}(\boldsymbol{x})})^2$，这样一定是使损失递减。也就是说损失会向着下降的方向移动，这也是梯度下降的思想。它和 Boosting Tree 中使用的残差的区别在于，通过残差是在寻找全局最优值 (每一步都试图让结果达到最好)，而使用残差的梯度是在搜寻局部最优值 (每一步都试图让结果更好一些)。既然如此，使用残差的梯度优势在哪？这就要考虑损失函数了。

损失函数的使用

在前面的描述中，我们对回归问题默认的损失函数是平方损失，但平方损失有个很大的问题，对异常点 (`Outliers`) 很是敏感。比如下面这个例子，可以看出最后一个样本 (异常点) 会在下一次迭代中占据主要影响，下一个基学习器会过多关注于最后一个的异常点。而 Boosting Tree 使用平方损失正是因为这个损失可以帮助获得残差，可如果换个对异常点鲁棒的损失函数呢？这个时候 Boosting Tree 就无计可施了。

y	0.5	1.2	2	5
$f(\boldsymbol{x})$	0.6	1.4	1.5	1.7
$\mathcal{L} = (y - f(\boldsymbol{x}))^2$	0.005	0.02	0.125	5.445

但 GBDT 却可以做到。如果我们将损失函数换做绝对损失：

$$\mathcal{L}(y, f(\boldsymbol{x})) = |y - f(\boldsymbol{x})| \tag{50}$$

其负梯度为：$-\frac{\partial \mathcal{L}}{\partial f(\boldsymbol{x})} = sgn(y - f(\boldsymbol{x}))$。

或者我们将其换做 Huber 损失：

$$\mathcal{L}(y, f(\boldsymbol{x})) = \begin{cases} y - f(\boldsymbol{x}), & |y - f(\boldsymbol{x})| \leq \delta \\ \delta sgn(y - f(\boldsymbol{x})), & |y - f(\boldsymbol{x})| > \delta \end{cases} \tag{51}$$

其负梯度为：$-\frac{\partial \mathcal{L}}{\partial f(\boldsymbol{x})} = \begin{cases} \frac{1}{2}(y - f(\boldsymbol{x}))^2, & |y - f(\boldsymbol{x})| \leq \delta \\ \delta(|y - f(\boldsymbol{x})| - \frac{\delta}{2}), & |y - f(\boldsymbol{x})| > \delta \end{cases}$。

我们可以看一下采用绝对损失和 Huber 损失时的示例：

y	0.5	1.2	2	5
$f(\boldsymbol{x})$	0.6	1.4	1.5	1.7
Square loss	0.005	0.02	0.125	5.445
Absolute loss	0.1	0.2	0.5	3.3
Huber loss($\delta = 0.5$)	0.005	0.02	0.125	1.525

可以看到后两种损失对异常点要鲁棒些，当然使用这两种损失后梯度就不等于残差，但梯度仍可以作为一种近似。

Shrinkage 收缩

Shrinkage 收缩指，每次走一小步逐渐逼近结果的效果，要比每次迈一大步很快逼近结果的方式更容易得到精确值。就是说它不完全信任每一棵残差树，认为每棵树只学到了真实的一部分，于是累加的时候只累加一小部分再多学几棵树来弥补不足。这个技巧类似于梯度下降里的学习率。

实现 Shrinkage 收缩的一种简略方法是固定每一个基学习器的步长 (学习率) η：$f_k(\boldsymbol{x}) = f_{k-1}(\boldsymbol{x}) + \eta h_k(\boldsymbol{x})$。这个形式就回到了最初前向分步加法模型的描述，只是这里的基学习器的参数预先给定。

另一种方法是 Line Search，去寻找最优的步长：

$$\eta = \arg\min_{\eta} \sum_{i=1}^{m} \mathcal{L}(y^{(i)}, f_{k-1}(\boldsymbol{x}^{(i)}) + \eta h_k(\boldsymbol{x}^{(i)})) \tag{52}$$

当损失函数为平方误差时，显然有：

$$\begin{aligned} \eta &= \arg\min_{\eta} \sum_{i=1}^{m} \left((y^{(i)} - f_{k-1}(\boldsymbol{x}^{(i)})) - \eta h_k(\boldsymbol{x}^{(i)}) \right) \\ &= \arg\min_{\eta} \sum_{i=1}^{m} \left(-2\eta(y^{(i)} - f_{k-1}(\boldsymbol{x}^{(i)})) + (\eta h_k(\boldsymbol{x}^{(i)}))^2 \right) \end{aligned} \tag{53}$$

对其求导并令导数为 0，可以得到此时的最优 η：

$$\eta^* = \frac{\sum_{i=1}^{m} 2(y^{(i)} - f_{k-1}(\boldsymbol{x}^{(i)}))h_k(\boldsymbol{x}^{(i)})}{\sum_{i=1}^{m} h_k^2(\boldsymbol{x}^{(i)})} \tag{54}$$

分类问题下的损失函数

以上我们讨论完了回归问题，现在我们讨论分类问题。首先是**二分类问题**，在第五章我们介绍过概率监督学习 (逻辑回归)，逻辑回归实质上是用线性模型去拟合对数几率 (`log odds`)，$\log\left(\frac{p}{1-p}\right)$。如果说回归问题是用线性模型 (线性可加模型) 直接学习目标结果，那分类问题就是用线性模型 (线性可加模型) 去学习对数几率。所以，分类模型可以表达为：

$$\hat{y} = P(y = 1 \mid \boldsymbol{x}) = \frac{1}{1 + e^{-\sum\limits_{k} h_k(\boldsymbol{x})}} = \frac{1}{1 + e^{-f(\boldsymbol{x})}} \tag{55}$$

其中 $\hat{y}$ 表示分类模型的预测。所以，损失函数就可以表达为交叉熵：

$$\mathcal{L}(y, f(\boldsymbol{x})) = -y \log \hat{y} - (1 - y) \log(1 - \hat{y}) \tag{56}$$

在第六章我们介绍过二分类交叉熵的求导，所以在这里我们有：

$$-\frac{\partial \mathcal{L}(y, f(\boldsymbol{x}))}{\partial f(\boldsymbol{x})} = y - \hat{y} \tag{57}$$

可以看到，与回归问题类似，下一棵决策树的训练样本为：$\{(\boldsymbol{x}, y - \hat{y})\}$，其所需要拟合的残差为真实标签与预测概率之差。

再看一下多分类问题，假设一个有 C 个类别，每一次迭代的训练实际上是训练了 C 棵树去拟合每一个类别。如果一共再进行 K 次迭代的话，那么训练完之后总共有 $C \times K$ 棵树。

$$\begin{aligned} \mathcal{L}(\boldsymbol{y}, f(\boldsymbol{x})) &= -\boldsymbol{y} \log \hat{\boldsymbol{y}} \\ &= -\sum_{c=1}^{C} y_c \log \hat{y}_c \end{aligned} \tag{58}$$

其中 $\hat{y}_c = \frac{e^{-f^c(\boldsymbol{x})}}{\sum_{l=1}^{C} e^{-f^l(\boldsymbol{x})}}$。同样在第六章我们介绍过多分类交叉熵的求导：

$$-\frac{\partial \mathcal{L}(y, f^c(\boldsymbol{x}))}{\partial f^c(\boldsymbol{x})} = y_c - \hat{y}_c \tag{59}$$

下一批基学习器 (C 棵决策树) 的训练样本为：$\{(\boldsymbol{x}, \boldsymbol{y} - \hat{\boldsymbol{y}})\}$，其所需要拟合的残差为真实标签与预测概率之差。

自定义实现

[32]:
```
from chapter5 import RegressionTree
```

[33]:
```
class Loss(ABC):

    def __init__(self):
        super().__init__()

    @abstractmethod
    def loss(self, y_true, y_pred):
        return NotImplementedError()

    @abstractmethod
    def grad(self, y, y_pred):
        raise NotImplementedError()

class SquareLoss(Loss):

    def __init__(self):
        pass

    def loss(self, y, y_pred):
        pass

    def grad(self, y, y_pred):
        return -(y - y_pred)

    def hess(self, y, y_pred):
        return 1

class CrossEntropyLoss(Loss):

    def __init__(self):
        pass

    def loss(self, y, y_pred):
        pass

    def grad(self, y, y_pred):
        return - (y - y_pred)

    def hess(self, y, y_pred):
        return y_pred * (1-y_pred)
```

[34]:
```
def softmax(x):
    e_x = np.exp(x - np.max(x, axis=-1, keepdims=True))
    return e_x / e_x.sum(axis=-1, keepdims=True)


def line_search(self, y, y_pred, h_pred):
    Lp = 2 * np.sum((y - y_pred) * h_pred)
    Lpp = np.sum(h_pred * h_pred)
    return 1 if np.sum(Lpp) == 0 else Lp / Lpp


def to_categorical(x, n_classes=None):
    """
    One-hot 编码
    """
    if not n_classes:
        n_classes = np.amax(x) + 1
```

```
    one_hot = np.zeros((x.shape[0], n_classes))
    one_hot[np.arange(x.shape[0]), x] = 1
    return one_hot


class GradientBoostingDecisionTree(object):
    """
    GBDT 算法。用一组基学习器 (回归树) 学习损失函数的梯度。
    """
    def __init__(self, n_estimators, learning_rate=1, min_samples_split=2,
                 min_impurity=1e-7, max_depth=float("inf"), is_regression=False, line_search=False):
        self.n_estimators = n_estimators          # 迭代的次数
        self.learning_rate = learning_rate        # 训练过程中沿着负梯度走的步长，也就是学习率
        self.min_samples_split = min_samples_split    # 分割所需的最小样本数
        self.min_impurity = min_impurity          # 分割所需的最小纯度
        self.max_depth = max_depth                # 树的最大深度
        self.is_regression = is_regression        # 分类问题或回归问题
        self.line_search = line_search            # 是否使用 line search
        self.progressbar = progressbar.ProgressBar(widgets=bar_widgets)
        # 回归问题采用基础的平方损失，分类问题采用交叉熵损失
        self.loss = SquareLoss()
        if not self.is_regression:
            self.loss = CrossEntropyLoss()

    def fit(self, X, Y):
        # 分类问题将 Y 转化为 one-hot 编码
        if not self.is_regression:
            Y = to_categorical(Y.flatten())
        else:
            Y = Y.reshape(-1, 1) if len(Y.shape) == 1 else Y
        self.out_dims = Y.shape[1]
        self.trees = np.empty((self.n_estimators, self.out_dims), dtype=object)
        Y_pred = np.full(np.shape(Y), np.mean(Y, axis=0))
        self.weights = np.ones((self.n_estimators, self.out_dims))
        self.weights[1:, :] *= self.learning_rate
        # 迭代过程
        for i in self.progressbar(range(self.n_estimators)):
            for c in range(self.out_dims):
                tree = RegressionTree(
                        min_samples_split=self.min_samples_split,
                        min_impurity=self.min_impurity,
                        max_depth=self.max_depth)
                # 计算损失的梯度，并用梯度进行训练
                if not self.is_regression:
                    Y_hat = softmax(Y_pred)
                    y, y_pred = Y[:, c], Y_hat[:, c]
                else:
                    y, y_pred = Y[:, c], Y_pred[:, c]
                neg_grad = -1 * self.loss.grad(y, y_pred)
                tree.fit(X, neg_grad)
                # 用新的基学习器进行预测
                h_pred = tree.predict(X)
                # line search
                if self.line_search == True:
                    self.weights[i, c] *= line_search(y, y_pred, h_pred)
                # 加法模型中添加基学习器的预测，得到最新迭代下的加法模型预测
                Y_pred[:, c] += np.multiply(self.weights[i, c], h_pred)
                self.trees[i, c] = tree
```

```
    def predict(self, X):
        Y_pred = np.zeros((X.shape[0], self.out_dims))
        # 生成预测
        for c in range(self.out_dims):
            y_pred = np.array([])
            for i in range(self.n_estimators):
                update = np.multiply(self.weights[i, c], self.trees[i, c].predict(X))
                y_pred = update if not y_pred.any() else y_pred + update
            Y_pred[:, c] = y_pred
        if not self.is_regression:
            # 分类问题输出最可能类别
            Y_pred = Y_pred.argmax(axis=1)
        return Y_pred

    def score(self, X, y):
        y_pred = self.predict(X)
        accuracy = np.sum(y == y_pred, axis=0) / len(y)
        return accuracy


class GradientBoostingRegressor(GradientBoostingDecisionTree):

    def __init__(self, n_estimators=200, learning_rate=1, min_samples_split=2,
                 min_impurity=1e-7, max_depth=float("inf"), is_regression=True, line_search=False):
        super(GradientBoostingRegressor, self).__init__(n_estimators=n_estimators,
            learning_rate=learning_rate,
            min_samples_split=min_samples_split,
            min_impurity=min_impurity,
            max_depth=max_depth,
            is_regression=is_regression,
            line_search=line_search)


class GradientBoostingClassifier(GradientBoostingDecisionTree):

    def __init__(self, n_estimators=200, learning_rate=1, min_samples_split=2,
                 min_impurity=1e-7, max_depth=float("inf"), is_regression=False, line_search=False):
        super(GradientBoostingClassifier, self).__init__(n_estimators=n_estimators,
            learning_rate=learning_rate,
            min_samples_split=min_samples_split,
            min_impurity=min_impurity,
            max_depth=max_depth,
            is_regression=is_regression,
            line_search=line_search)
```

用自定义的 GBDT，乳腺癌数据集测试

[35]:
```
column_names = ['Sample code number','Clump Thickness',
                'Uniformity of Cell Size','Uniformity of Cell Shape',
                'Marginal Adhesion','Single Epithelial Cell Size',
                'Bare Nuclei','Bland Chromatin','Normal Nucleoli','Mitoses','Class']
data = pd.read_csv('../data/cancer/breast-cancer-wisconsin.data',names=column_names)
data = data.replace(to_replace='?',value=np.nan)  # 非法字符替代
data = data.dropna(how='any')  # 去掉空值，any；出现空值行则删除
print(data.shape)
# 随机采样 25% 的数据用于测试，剩下 75% 用于构建训练集
X_train,X_test,y_train,y_test = train_test_split(data[column_names[1:10]], data[column_names[10]],
                                                 test_size=0.25, random_state=1111)
# 再查看训练样本的数量和类别分布
print(y_train.value_counts())
```

```
# 修改标签为 0 和 1
print(y_train.shape)
y_train[y_train==2] = 0
y_train[y_train==4] = 1
y_test[y_test==2] = 0
y_test[y_test==4] = 1
# 再查看训练样本的数量和类别分布
print(y_train.value_counts())
# 数据标准化预处理，保证每个维度特征均值为 0，方差为 1
ss = StandardScaler()
X_train = ss.fit_transform(X_train)
X_test = ss.transform(X_test)
y_train = y_train.as_matrix()
y_test = y_test.as_matrix()
```

```
(683, 11)
2    328
4    184
Name: Class, dtype: int64
(512,)
0    328
1    184
Name: Class, dtype: int64
```

[36]:
```
model = GradientBoostingClassifier(n_estimators=20)
model.fit(X_train, y_train)
print(model.score(X_test, y_test))
```

```
Training: 100% [----------------------------------------------] Time: 0:00:12

0.9532163742690059
```

用 sklearn 的 GBDT，乳腺癌数据集测试

[37]:
```
from sklearn.ensemble import GradientBoostingClassifier
skl_model= GradientBoostingClassifier()
skl_model.fit(X_train, y_train)
print(skl_model.score(X_test, y_test))
```

```
0.9473684210526315
```

6.1.4 XGBoost 算法

XGBoost 是 GBDT 的改进。再次回到三个问题，XGBoost 的解决方案是什么?

- 弱分类器 (基分类器) 选什么？选法是 XGBoost 回归树 (见后文)。
- 如何调整样本分布？不做更改。
- 如何进行组合？组合方式即为加法模型 $f(\boldsymbol{x}) = \sum_{k=1}^{K} h_k(\boldsymbol{x})$，其中基学习器的系数为 1。损失函数同 GBDT。但是，在 XGBoost 中，显式地将树模型的复杂度作为正则项加在优化目标里。

XGBoost 回归树的学习策略

XGBoost 的损失函数中显示地添加了树模型的复杂度作为正则项 $\Omega(h_k)$:

$$\Omega(h_k) = \gamma T + \frac{\lambda}{2} \sum_{j=1}^{T} w_j^2 \tag{60}$$

其中 T 是基学习器叶子节点的数目，w_j 是每个叶子节点的权重。对叶子节点个数进行惩罚，相当于在训练过程中做了剪枝。对于样本 $\boldsymbol{x}$，我们用 $q(\boldsymbol{x})$ 表示将样本 $\boldsymbol{x}$ 分到了某个叶节点上，于是 $w_{q(\boldsymbol{x})}$ 表示回归树对样本的预测值。因此基学习器 $h_k = w_{q(\boldsymbol{x})}$。考虑正则项条件下，目标函数为:

$$J(y, f(\boldsymbol{x})) = \mathcal{L}(y, f(\boldsymbol{x})) + \sum_{k=1}^{K} \Omega(h_k) \tag{61}$$

然后我们再考虑学习目标，第 k 次迭代时：

$$\begin{aligned} J(y, f(\boldsymbol{x})) &= \mathcal{L}(y, f_{k-1}(\boldsymbol{x}) + h_k(\boldsymbol{x})) + \sum_{j=1}^{k-1} \Omega(h_j) + \Omega(h_k) \\ &\approx \mathcal{L}(y, f_{k-1}(\boldsymbol{x})) + \frac{\partial \mathcal{L}}{\partial f_{k-1}(\boldsymbol{x})} h_k(\boldsymbol{x}) + \frac{\partial^2 \mathcal{L}}{2\partial f_{k-1}^2(\boldsymbol{x})} h_k^2(\boldsymbol{x}) + \sum_{j=1}^{k-1} \Omega(h_j) + \Omega(h_k) \\ &= \mathcal{L}(y, f_{k-1}(\boldsymbol{x})) + \frac{\partial \mathcal{L}}{\partial f_{k-1}(\boldsymbol{x})} h_k(\boldsymbol{x}) + \frac{\partial^2 \mathcal{L}}{2\partial f_{k-1}^2(\boldsymbol{x})} h_k^2(\boldsymbol{x}) + \Omega(h_k) + Const \end{aligned} \tag{62}$$

其中，从第一步到第二步用到了泰勒二阶展开：

$$f(x) \approx f(x_0) + f'(x_0)(x - x_0) + \frac{f''(x_0)}{2!}(x - x_0) \tag{63}$$

简化一下书写，我们令第 i 个样本的：

$$\begin{aligned} g_i' &:= \frac{\partial \mathcal{L}}{\partial f_{k-1}(\boldsymbol{x}_i)} \\ h_i' &:= \frac{\partial^2 \mathcal{L}}{2\partial f_{k-1}^2(\boldsymbol{x}_i)} \end{aligned} \tag{64}$$

于是，损失函数为 (把样本编号信息也标注上)：

$$\begin{aligned} J(y, f(\boldsymbol{x})) &= \sum_{i=1}^{m} \left(\mathcal{L}(y, f_{k-1}(\boldsymbol{x})) + g_i' h_k(\boldsymbol{x}_i) + \frac{1}{2} h_i' h_k^2(\boldsymbol{x}_i) \right) + \Omega(h_k(\boldsymbol{x})) + Constant \\ &= \sum_{i=1}^{m} \left(\mathcal{L}(y, f_{k-1}(\boldsymbol{x})) + g_i' w_{q(\boldsymbol{x}_i)} + \frac{1}{2} h_i' w_{q(\boldsymbol{x}_i)}^2 \right) + \gamma T + \frac{\lambda}{2} \sum_{j=1}^{T} w_j^2 + Constant \\ &= \sum_{i=1}^{m} \left(g_i' w_{q(\boldsymbol{x}_i)} + \frac{1}{2} h_i' w_{q(\boldsymbol{x}_i)}^2 \right) + \gamma T + \frac{\lambda}{2} \sum_{j=1}^{T} w_j^2 + Constant' \end{aligned} \tag{65}$$

红色项是对样本的累加，蓝色项是对叶节点的累加。怎么统一起来？定义每个叶节点 j 上的样本集合为：$I_j = \{i \mid q(\boldsymbol{x}_i) = j\}$。则目标函数可以写成按叶节点累加的形式 (优化问题中常数项不影响结果，我们不再考虑)：

$$\begin{aligned} J(y, f(\boldsymbol{x})) &= \sum_{j=1}^{T} \left[(\sum_{i \in I_j} g_i') w_j + \frac{1}{2} (\sum_{i \in I_j} h_i' + \lambda) w_j^2 \right] + \gamma T \\ &= \sum_{j=1}^{T} \left[(G_j w_j + \frac{1}{2} (H_j + \lambda) w_j^2 \right] + \gamma T \end{aligned} \tag{66}$$

如果确定了树的结构 (即 $q(\boldsymbol{x})$ 确定)，为了使目标函数最小，可以令其导数为 0，解得每个叶节点的最优预测分数为：

$$w_j^* = -\frac{G_j}{H_j + \lambda}, \quad j = 1, 2, \cdots, T \tag{67}$$

代入目标函数，得到最小损失为：

$$J^* = -\frac{1}{2} \sum_{j=1}^{T} \frac{G_j^2}{H_j + \lambda} + \gamma T \tag{68}$$

当回归树的结构确定时，我们前面已经推导出其最优的叶节点分数以及对应的最小损失值，问题是怎么确定树的结构？也就是说我们的基学习器是什么样的？一种是暴力枚举所有可能的树结构，选择损失值最小的，但这是 NP 难问题。另一种方法是贪心法，每次尝试分裂一个叶节点，计算分裂前后的增益，选择增益最大的 (信息增益见第五章)。

$$J^* = -\frac{1}{2} \sum_{j=1}^{T} \frac{G_j^2}{H_j + \lambda} + \gamma T \tag{69}$$

标红部分衡量了每个叶子节点对总体损失的的贡献，我们希望损失越小越好，则标红部分的值越大越好。

因此，对一个叶子节点进行分裂，分裂前后的增益定义为：

$$\begin{aligned} Gain &= J^* - (J_L^* + J_R^*) \\ &= \left(-\frac{1}{2} \frac{(G_L + G_R)^2}{H_L + H_G + \lambda} + \gamma \right) - \left(-\frac{1}{2} \frac{G_L^2}{H_L + \lambda} + \gamma \right) - \left(-\frac{1}{2} \frac{G_R^2}{H_R + \lambda} + \gamma \right) \\ &= \frac{1}{2} \left(\frac{G_L^2}{H_L + \lambda} + \frac{G_R^2}{H_R + \lambda} - \frac{(G_L + G_R)^2}{H_L + H_G + \lambda} \right) - \gamma \end{aligned} \tag{70}$$

这个结果就可以用于在实践中评估候选分裂节点是不是应该分裂的划分依据，我们尽量找到使之最大的特征值分裂点。

在树学习中，另外一个关键问题是如何找到每一个特征上的分裂点。直觉的想法是枚举特征上所有可能的分裂点，然后计算上述的增益。这种算法称为 Exact Greedy Algorithm。当然为了有效率的找到最佳分裂节点，算法可以先将该特征的所有取值进行排序，之后按顺序取分裂值计算。但是当数据量很大时，数据不可能一次性的全部读入到内存中，或者在分布式计算中，也不可能事先对所有值进行排序，且无法使用所有数据来计算分裂节点之后的树结构的增益分数。

XGBoost 回归树的节点分裂算法

正如上面所说，尽管我们找到了寻找最佳分裂点的指标，但使用贪婪算法逐个特征值计算的计算量过高。为解决这个问题，有几种解决方案。

一种是**近似算法** (`Approximate Algo for Split Finding`)，近似算法首先按照特征取值中统计分布的百分位点确定一些候选分裂点，然后算法将连续的值映射到桶 (`buckets`) 中，接着汇总统计数据，并根据聚合统计数据在候选节点中找到最佳节点。XGBoost 采用的近似算法对于每个特征，只考察分位点，减少复杂度。例如，以三分位点举例：

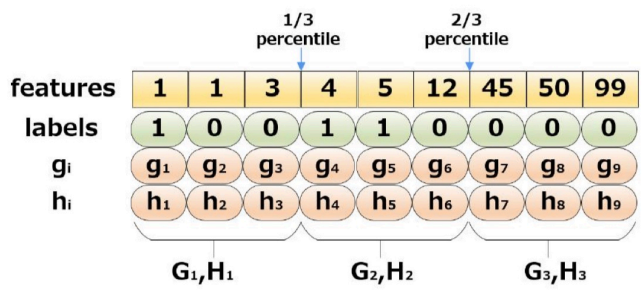


图 7. 只考察分位点的近似算法。

于是，我们找到其中最大的信息增益的划分方法：

$$Gain = \max\left(Gain, \frac{1}{2}\left(\frac{G_1^2}{H_1+\lambda}+\frac{G_{23}^2}{H_{23}+\lambda}-\frac{(G_{123})^2}{H_{123}+\lambda}\right)-\gamma, \frac{1}{2}\left(\frac{G_{12}^2}{H_{12}+\lambda}+\frac{G_3^2}{H_3+\lambda}-\frac{(G_{123})^2}{H_{123}+\lambda}\right)-\gamma\right) \tag{71}$$

然而，这种划分分位点的方法在实际中可能效果不是很好。

另一种是加权分位数 (`Weighted Quantile Sketch`)，我们需先构造一个集合：$\mathcal{D}_j = \left\{\left(x_{ij}, h'_{ij}\right)_{i=1,\dots m}\right\}$，其中 x_{ij} 表示第 i 个样本的第 j 个特征值，h'_{ij} 是第 i 个样本的第 j 个特征的二阶梯度统计。我们现在定义一个排序函数 r_j：

$$r_j(z) = \frac{1}{\sum_{(x,h')\in\mathcal{D}_j} h'} \sum_{(x,h')\in\mathcal{D}_j, x<z} h' \tag{72}$$

表示特征 j 所有可取值中小于 z 的特征值的总权重占总的所有可取值的总权重和的比例，用 h' 加权。目标就是寻找候选分裂点集 $\{s_{j1}, s_{j2}, \dots s_{jl}\}$，有

$$\left|r_j\left(s_{j,t}\right) - r_j\left(s_{j,t+1}\right)\right| < \epsilon, \quad s_{j1} = \min_i x_{ij}, s_{jl} = \max_i x_{ij} \tag{73}$$

ϵ 是近似因子或者说是扫描步幅，按照步幅 ϵ 挑选出特征 j 的取值候选点，组成候选点集，这意味着有大概 $\frac{1}{\epsilon}$ 个候选点。但是，我们为什么要用 h' 加权呢？我们把目标函数整理成以下形式：

$$\begin{aligned} J(y, f(\boldsymbol{x})) &\simeq \sum_{i=1}^m \left[g'_i h_k(\boldsymbol{x}_i) + \frac{1}{2}h'_i h_k^2(\boldsymbol{x}_i)\right] + \Omega\left(h_k\right) + Constant \\ &= \sum_{i=1}^m \left[g'_i h_k(\boldsymbol{x}_i) + \frac{1}{2}h'_i h_k^2(\boldsymbol{x}_i) + \underbrace{\frac{1}{2}\frac{g_i'^2}{h'_i}}_{\text{添加常数项}}\right] + \Omega\left(h_k\right) + Constant' \\ &= \sum_{i=1}^m \frac{1}{2}h'_i\left[h_k(\boldsymbol{x}_i) - \left(-\frac{g'_i}{h'_i}\right)\right]^2 + \Omega\left(h_k\right) + Constant' \end{aligned} \tag{74}$$

最后的代价函数就是一个加权平方误差，权值为 h'_i，标签为 $-\frac{g'_i}{h'_i}$，所以可以将特征 j 的取值权重看成对应的 h'_i。

XGBoost 的损失函数

最后，如何定义 $\mathcal{L}(y, f(\boldsymbol{x}))$？如果是回归模型，可以采用平方损失；如果是分类模型，可以采用交叉熵损失。具体原理同 GBDT。有了 $\mathcal{L}(y, f(\boldsymbol{x}))$ 的具体形式，我们便可以得到 H 和 G。(注：平方损失和交叉熵损失的 G 和 H 的推导可以参考第六章，在第六章我们推导了一阶导数的获得，而二阶导数只需要在一阶导数上再简单求导一次便可。)

XGBoost 的其它设置

除了上述描述的以外，XGBoost 在实现过程中，还使用了包括：

1. **缺失值处理**。当有缺失值时，系统将样本分到默认方向的叶子节点。每个分支都有两个默认方向，最佳的默认方向可以从训练数据中学习到。
2. **系统优化设计**。包括分块并行、缓存优化等。关于并行问题我们会在十二章介绍。
3. **列抽样**。这里借鉴了随机森林的做法，支持对特征采样，不仅能降低过拟合，还能减少计算。
4. Shrinkage。相当于学习率，这里同 GBDT 的做法，主要是为了削弱每棵树的影响，让后面有更大的学习空间。

[38]:
```
from chapter5 import DecisionTree
```

[39]:
```
class XGBoostRegressionTree(DecisionTree):
    """
    XGBoost 回归树。此处基于第五章介绍的决策树，故采用贪心算法找到特征上分裂点 (枚举特征上所有可能的分裂点)。
    """
    def __init__(self, min_samples_split=2, min_impurity=1e-7,
                 max_depth=float("inf"), loss=None, gamma=0., lambd=0.):
```

```
        super(XGBoostRegressionTree, self).__init__(min_impurity=min_impurity,
            min_samples_split=min_samples_split,
            max_depth=max_depth)
        self.gamma = gamma    # 叶子节点的数目的惩罚系数
        self.lambd = lambd    # 叶子节点的权重的惩罚系数
        self.loss = loss      # 损失函数

    def _split(self, y):
        # y 包含 y_true 在左半列，y_pred 在右半列
        col = int(np.shape(y)[1]/2)
        y, y_pred = y[:, :col], y[:, col:]
        return y, y_pred

    def _gain(self, y, y_pred):
        # 计算信息
        nominator = np.power((y * self.loss.grad(y, y_pred)).sum(), 2)
        denominator = self.loss.hess(y, y_pred).sum()
        return nominator / (denominator + self.lambd)

    def _gain_by_taylor(self, y, y1, y2):
        # 分割为左子树和右子树
        y, y_pred = self._split(y)
        y1, y1_pred = self._split(y1)
        y2, y2_pred = self._split(y2)
        true_gain = self._gain(y1, y1_pred)
        false_gain = self._gain(y2, y2_pred)
        gain = self._gain(y, y_pred)
        # 计算信息增益
        return 0.5 * (true_gain + false_gain - gain) - self.gamma

    def _approximate_update(self, y):
        y, y_pred = self._split(y)
        # 计算叶节点权重
        gradient = self.loss.grad(y, y_pred).sum()
        hessian = self.loss.hess(y, y_pred).sum()
        leaf_approximation = -gradient / (hessian + self.lambd)
        return leaf_approximation

    def fit(self, X, y):
        self._impurity_calculation = self._gain_by_taylor
        self._leaf_value_calculation = self._approximate_update
        super(XGBoostRegressionTree, self).fit(X, y)


class XGBoost(object):
    """
    XGBoost 学习器。
    """
    def __init__(self, n_estimators=200, learning_rate=0.001, min_samples_split=2,
                 min_impurity=1e-7, max_depth=2, is_regression=False, gamma=0., lambd=0.):
        self.n_estimators = n_estimators            # 树的数目
        self.learning_rate = learning_rate          # 训练过程中沿着负梯度走的步长，也就是学习率
        self.min_samples_split = min_samples_split  # 分割所需的最小样本数
        self.min_impurity = min_impurity            # 分割所需的最小纯度
        self.max_depth = max_depth                  # 树的最大深度
        self.gamma = gamma                          # 叶子节点的数目的惩罚系数
        self.lambd = lambd                          # 叶子节点的权重的惩罚系数
        self.is_regression = is_regression          # 分类或回归问题
        self.progressbar = progressbar.ProgressBar(widgets=bar_widgets)
```

```
        # 回归问题采用基础的平方损失，分类问题采用交叉熵损失
        self.loss = SquareLoss()
        if not self.is_regression:
            self.loss = CrossEntropyLoss()

    def fit(self, X, Y):
        # 分类问题将 Y 转化为 one-hot 编码
        if not self.is_regression:
            Y = to_categorical(Y.flatten())
        else:
            Y = Y.reshape(-1, 1) if len(Y.shape) == 1 else Y
        self.out_dims = Y.shape[1]
        self.trees = np.empty((self.n_estimators, self.out_dims), dtype=object)
        Y_pred = np.zeros(np.shape(Y))
        self.weights = np.ones((self.n_estimators, self.out_dims))
        self.weights[1:, :] *= self.learning_rate
        # 迭代过程
        for i in self.progressbar(range(self.n_estimators)):
            for c in range(self.out_dims):
                tree = XGBoostRegressionTree(
                        min_samples_split=self.min_samples_split,
                        min_impurity=self.min_impurity,
                        max_depth=self.max_depth,
                        loss=self.loss,
                        gamma=self.gamma,
                        lambd=self.lambd)
                # 计算损失的梯度，并用梯度进行训练
                if not self.is_regression:
                    Y_hat = softmax(Y_pred)
                    y, y_pred = Y[:, c], Y_hat[:, c]
                else:
                    y, y_pred = Y[:, c], Y_pred[:, c]

                y, y_pred = y.reshape(-1, 1), y_pred.reshape(-1, 1)
                y_and_ypred = np.concatenate((y, y_pred), axis=1)
                tree.fit(X, y_and_ypred)
                # 用新的基学习器进行预测
                h_pred = tree.predict(X)
                # 加法模型中添加基学习器的预测，得到最新迭代下的加法模型预测
                Y_pred[:, c] += np.multiply(self.weights[i, c], h_pred)
                self.trees[i, c] = tree

    def predict(self, X):
        Y_pred = np.zeros((X.shape[0], self.out_dims))
        # 生成预测
        for c in range(self.out_dims):
            y_pred = np.array([])
            for i in range(self.n_estimators):
                update = np.multiply(self.weights[i, c], self.trees[i, c].predict(X))
                y_pred = update if not y_pred.any() else y_pred + update
            Y_pred[:, c] = y_pred
        if not self.is_regression:
            # 分类问题输出最可能类别
            Y_pred = Y_pred.argmax(axis=1)
        return Y_pred

    def score(self, X, y):
        y_pred = self.predict(X)
        accuracy = np.sum(y == y_pred, axis=0) / len(y)
```

```
        return accuracy


class XGBRegressor(XGBoost):

    def __init__(self, n_estimators=200, learning_rate=1, min_samples_split=2,
                 min_impurity=1e-7, max_depth=float("inf"), is_regression=True,
                 gamma=0., lambd=0.):
        super(XGBRegressor, self).__init__(n_estimators=n_estimators,
            learning_rate=learning_rate,
            min_samples_split=min_samples_split,
            min_impurity=min_impurity,
            max_depth=max_depth,
            is_regression=is_regression,
            gamma=gamma,
            lambd=lambd)
```

```
class XGBClassifier(XGBoost):

    def __init__(self, n_estimators=200, learning_rate=1, min_samples_split=2,
                 min_impurity=1e-7, max_depth=float("inf"), is_regression=False,
                 gamma=0., lambd=0.):
        super(XGBClassifier, self).__init__(n_estimators=n_estimators,
            learning_rate=learning_rate,
            min_samples_split=min_samples_split,
            min_impurity=min_impurity,
            max_depth=max_depth,
            is_regression=is_regression,
            gamma=gamma,
            lambd=lambd)
```

用自定义的 XGBoost，乳腺癌数据集测试

[40]:
```
column_names = ['Sample code number','Clump Thickness',
                'Uniformity of Cell Size','Uniformity of Cell Shape',
                'Marginal Adhesion','Single Epithelial Cell Size',
                'Bare Nuclei','Bland Chromatin','Normal Nucleoli','Mitoses','Class']
data = pd.read_csv('../data/cancer/breast-cancer-wisconsin.data',names=column_names)
data = data.replace(to_replace='?',value=np.nan)  # 非法字符替代
data = data.dropna(how='any')  # 去掉空值，any；出现空值行则删除
print(data.shape)
# 随机采样 25% 的数据用于测试，剩下 75% 用于构建训练集
X_train,X_test,y_train,y_test = train_test_split(data[column_names[1:10]], data[column_names[10]],
                                                 test_size=0.25, random_state=1111)
# 再查看训练样本的数量和类别分布
print(y_train.value_counts())
# 修改标签为 0 和 1
print(y_train.shape)
y_train[y_train==2] = 0
y_train[y_train==4] = 1
y_test[y_test==2] = 0
y_test[y_test==4] = 1
# 再查看训练样本的数量和类别分布
print(y_train.value_counts())
# 数据标准化预处理，保证每个维度特征均值为 0，方差为 1
ss = StandardScaler()
X_train = ss.fit_transform(X_train)
X_test = ss.transform(X_test)
y_train = y_train.as_matrix()
```

```
y_test = y_test.as_matrix()
```

```
(683, 11)
2    328
4    184
Name: Class, dtype: int64
(512,)
0    328
1    184
Name: Class, dtype: int64
```

[41]:
```
model = XGBClassifier(n_estimators=20)
model.fit(X_train, y_train)
print(model.score(X_test, y_test))
```

```
Training: 100% [----------------------------------------------] Time: 0:00:06

0.9590643274853801
```

[42]:
```
import numpy
import PIL
import matplotlib
import re
import pandas
import progressbar
import sklearn

print("numpy:", numpy.__version__)
print("PIL:", PIL.__version__)
print("matplotlib:", matplotlib.__version__)
print("re:", re.__version__)
print("pandas:", pandas.__version__)
print("progressbar:", progressbar.__version__)
print("sklearn:", sklearn.__version__)
```

```
numpy: 1.14.5
PIL: 6.2.1
matplotlib: 3.1.1
re: 2.2.1
pandas: 0.25.1
progressbar: 2.5
sklearn: 0.21.3
```