梯度下降和 MCMC 实现逻辑回归的 LASSO 形式

目录

1 LASSO 算法实现逻辑回归模型

逻辑回归模型：

$$\mathbb{P}(Y=1)=\frac{\exp\left(\boldsymbol{X}^{\top}\boldsymbol{\beta}\right)}{1+\exp\left(\boldsymbol{X}^{\top}\boldsymbol{\beta}\right)}$$

最小化带有 L1 正则项的损失函数，求得最优的参数 $\boldsymbol{\beta}$：

$$\hat{\boldsymbol{\beta}}=\underset{\boldsymbol{\beta}}{\operatorname{argmin}}\left\{\frac{1}{n}\sum_{i=1}^{n}\ell\left(Y_i,\boldsymbol{X}_i^{\top}\boldsymbol{\beta}\right)+\lambda\sum_{j=1}^{p}|\beta_j|\right\}$$

其中，损失函数为：

$$\begin{aligned}f(\boldsymbol{\beta})&=\frac{1}{n}\sum_{i=1}^{n}l\left(Y_i,\boldsymbol{X}_i^{\top}\boldsymbol{\beta}\right)+\lambda\sum_{j=1}^{p}|\beta_j|\\&=\frac{1}{n}\sum_{i=1}^{n}\left(\log\left(1+\exp\left(\boldsymbol{X}_i^{\top}\boldsymbol{\beta}\right)\right)-Y_i\boldsymbol{X}_i^{\top}\boldsymbol{\beta}\right)+\lambda\sum_{j=1}^{p}|\beta_j|\\&=\frac{1}{n}\sum_{i=1}^{n}\left(\log\left(1+\exp\left(\sum_{j=1}^{p}\boldsymbol{X}_{ij}\beta_j\right)\right)-Y_i\sum_{j=1}^{p}\boldsymbol{X}_{ij}\beta_j\right)+\lambda\sum_{j=1}^{p}|\beta_j|\end{aligned}$$

则梯度为：

$$\begin{aligned}\nabla f(\boldsymbol{\beta})&=\frac{1}{n}\sum_{i=1}^{n}\left(\frac{\exp\left(\boldsymbol{X}_i^{\top}\boldsymbol{\beta}\right)}{1+\exp\left(\boldsymbol{X}_i^{\top}\boldsymbol{\beta}\right)}-Y_i\right)\boldsymbol{X}_i+\lambda\frac{\partial|\boldsymbol{\beta}|}{\partial\boldsymbol{\beta}}\\&=\frac{1}{n}\sum_{i=1}^{n}\left(\frac{1}{1+\exp\left(-\boldsymbol{X}_i^{T}\boldsymbol{\beta}\right)}-Y_i\right)\boldsymbol{X}_i+\lambda\frac{\partial|\boldsymbol{\beta}|}{\partial\boldsymbol{\beta}}\end{aligned}$$

梯度的第 j 个分量为：

$$\frac{\partial f(\beta)}{\partial\beta_j}=\frac{1}{n}\sum_{i=1}^{n}\left(\frac{1}{1+\exp\left(-\boldsymbol{X}_i^{T}\boldsymbol{\beta}\right)}-Y_i\right)\boldsymbol{X}_{ij}+\lambda\frac{\partial|\beta_j|}{\partial\beta_j}$$

其中，$\frac{\partial|\beta_j|}{\partial\beta_j}$ 的值为：

$$\frac{\partial|\beta_j|}{\partial\beta_j}=\begin{cases}1 & \text{if } \beta_j>0\\ [-1,1] & \text{if } \beta_j=0\\ -1 & \text{if } \beta_j<0\end{cases}$$

1.1 迭代公式

1.1.1 梯度下降法

梯度下降法使用所有样本的梯度来更新参数，迭代算法如下：

Algorithm 1 梯度下降法

Input 回溯线性搜索的参数 α 和 c, 最大迭代次数 K, 迭代初始值 $\boldsymbol{\beta}^0$

Output 迭代 K 次后得到的 $\boldsymbol{\beta}^k$

function GRADIENT_DESCENT($\alpha, c, K, \boldsymbol{\beta}^0$)
  $t_0 \leftarrow 1$
  for $k \leftarrow 0$ to $(K-1)$ **do**
    while $f\left(\boldsymbol{\beta}^k - t_k \nabla f\left(\boldsymbol{\beta}^k\right)\right) \geqslant f\left(\boldsymbol{\beta}^k\right) - \alpha t_k \left\|\nabla f\left(\boldsymbol{\beta}^k\right)\right\|^2$ **do**
      $t_k \leftarrow c t_k$ ▷ 回溯线性搜索选择合适的步长
    end while
    $\boldsymbol{\beta}^{k+1} \leftarrow \boldsymbol{\beta}^k - t^k \nabla f\left(\boldsymbol{\beta}^k\right)$
  end for
  return $\boldsymbol{\beta}^{k+1}$
end function

1.1.2 随机梯度下降法

随机梯度下降法使用部分小批量样本的梯度来更新参数，迭代算法如下：

Algorithm 2 随机梯度下降法

Input 批大小 b, 步长减小的参数 $\alpha(\alpha > 0)$, 最大迭代次数 K, 迭代初始值 $\boldsymbol{\beta}^0$

Output 迭代 K 次后得到的 $\boldsymbol{\beta}^k$

function MINI_BATCH_STOCHASTIC_GRADIENT_DESCENT($b, \alpha, K, \boldsymbol{\beta}^0$)
  $t_0 \leftarrow 1$
  for $k \leftarrow 0$ to $(K-1)$ **do**
    $t_k \leftarrow (k+1)^{-\alpha}$
    从训练集中随机抽取 b 个样本，构成小批量样本集合 $\mathcal{B}$
    $\boldsymbol{\beta^{k+1}} \leftarrow \boldsymbol{\beta^k} - t_k \frac{1}{b} \sum_{i \in \mathcal{B}} \nabla f_i(\boldsymbol{\beta})$
  end for
  return $\boldsymbol{\beta}^{k+1}$
end function

1.1.3 坐标下降法

坐标下降法所有样本的梯度来更新参数，每次只更新一个分量，迭代算法如下：

Algorithm 3 坐标下降法

Input 步长减小的参数 $\alpha(\alpha > 0)$, 最大迭代次数 K, 迭代初始值 $\boldsymbol{\beta}^0$

Output 迭代 K 次后得到的 $\boldsymbol{\beta}$

function COORDINATE_DESCENT($\alpha, K, \boldsymbol{\beta}^0$)
  $t_0 \leftarrow 1$, $\boldsymbol{\beta} \leftarrow \boldsymbol{\beta}^0$
  for $k \leftarrow 0$ to $(K-1)$ **do**
    $t_k \leftarrow (k+1)^{-\alpha}$
    $j = k\%\beta.shape$ ▷ 循环更新每个分量
    $\boldsymbol{\beta}_j \leftarrow \boldsymbol{\beta}_j - t_k \nabla f_j(\boldsymbol{\beta})$
  end for
  return $\boldsymbol{\beta}$
end function

1.2 求解最优参数的 Python 实现

1.2.1 base 类

首先定义 base 类，它可以基于 $\boldsymbol{\beta}$ 和自变量 $\boldsymbol{X}$ 预测标签，计算最优的判别 Y_i 为 0 或 1 的概率门槛，并打印模型的 Accuracy、Precision、Recall 和 F1-Score 评价指标。

由于页面有限，在正文中只展示函数名称，详细的代码见附录C。

```
class base:
    def __init__(self, beta=None):
        """

        Args:
            beta: 待优化的参数

        """
        self.beta = beta
        self.best_threshold = None

    def predict_proba(self, x):


    def get_best_threshold(self, x, y):


    def predict(self, x, threshold=None):


    def accuracy(self, y_pred, y):


    def precision(self, y_pred, y):


    def recall(self, y_pred, y):


    def f1_score(self, y_pred, y):
```

```
    def print_performance(self, y_pred, y):
```

1.2.2 LogisticRegression 类

定义 LogisticRegression 类，它继承自 base 类。

在初始化 LogisticRegression 类的实例时，可以定义 L1 惩罚项的系数 λ 和最大迭代次数 K。

```
class LogisticRegression(base):
    def __init__(self, lambda_=0.01, K=1000):
        """

        Args:
            lambda_: 正则化调节参数
            K: 最大迭代次数

        """
        super().__init__()
        self.lambda_ = lambda_
        self.K = K
```

1.2.3 逻辑回归的损失函数与梯度

在 LogisticRegression 类中，计算：

1. 单个样本的损失；
2. 最优化问题的目标函数，即所有样本的损失加上 L1 惩罚项；
3. 基于所有样本的梯度向量。

```
# 逻辑回归的损失函数
def l(self, x_i, y_i, beta):
    """

    Args:
        x_i: 样本 i 的特征
        y_i: 样本 i 的标签
        beta: 待优化的参数

    Returns:
        损失函数的值

    """
    return -y_i * x_i.dot(beta) + np.log(1 + np.exp(x_i.dot(beta)))

# Lasso 最优化问题的目标函数
def f(self, x, y, beta, lambda_):
    """

    Args:
        x: 样本 i 的特征
        y: 样本 i 的标签
        beta: 待优化的参数
        lambda_: 正则化调节参数

    Returns:
        目标函数的值
```

```
    """
    n = len(y)
    part1 = np.sum(np.log(1 + np.exp(x.dot(beta))))
    part2 = - y.T.dot(x.dot(beta))
    part3 = lambda_ * np.sum(np.abs(beta))
    return (part1 + part2) / n + part3

# 梯度下降法，基于所有样本求梯度向量
def gradient(self, x, y, beta, lambda_):
    """

    Args:
        x: 所有样本的特征
        y: 所有样本的标签
        beta: 待优化的参数
        lambda_: 正则化调节参数

    Returns:
        基于所有样本求得的梯度向量

    """
    n = len(y)
    part1 = np.sum((1 / (1 + np.exp(-x.dot(beta)))) * x.T, axis=1)
    part2 = - np.sum(y * (x.T), axis=1)
    part3 = lambda_ * np.sign(beta)
    return (part1 + part2) / n + part3
```

1.2.4 梯度下降法求解最优参数

基于所有样本求梯度向量，根据回溯线性搜索确定的步长，迭代 K 次得到最优的参数 β。

```
def gradient_descent(self, x, y, alpha=0.5, c=0.5, beta_initial=None):
    """

    Args:
        x: 所有样本的特征
        y: 所有样本的标签
        alpha: 回溯线性搜索的参数
        c: 回溯线性搜索的参数
        beta_initial: 待优化的参数的初始值，默认为全零向量

    Returns:
        beta: 迭代后的最优参数

    """
    # 初始化损失函数值
    self.loss = []
    # 初始化待优化的参数
    if beta_initial is None:
        beta = np.zeros(x.shape[1])
    else:
        beta = beta_initial
    # 迭代求解
    for k in range(self.K):
        # 计算梯度
        gradient_beta = self.gradient(x, y, beta, self.lambda_)
        # 计算回溯线性搜索的步长
        t = 1
```

```
        while self.f(x, y, beta - t * gradient_beta, self.lambda_) > self.f(x, y, beta, self.lambda_
                                                                             ) - alpha * t * gradient_beta.T.dot(
                                                                             gradient_beta):
            t = c * t
        # 更新参数
        beta = beta - t * gradient_beta
        # 记录损失函数的值
        self.loss.append(self.f(x, y, beta, self.lambda_))
        # 改变当前值的条件：至少迭代了100轮，且损失函数的值在最近2轮的迭代中变化小于1e-6
        if k > 100 and abs(self.loss[-2] - self.loss[-1]) < 1e-6:
            # 为每个分量加上噪音N(0, 1e-2)
            beta = beta + np.random.normal(0, 1e-2, beta.shape)
    # 将参数值小于 1e-3 的置为 0
    beta[np.abs(beta) < 1e-3] = 0

    return beta
```

1.2.5 随机梯度下降法求解最优参数

基于部分样本求梯度向量，根据 $t_k = (k+1)^{-\alpha}$ 确定步长 $(\alpha = 0.1)$，迭代 K 次得到最优的参数 β。

```
def mini_batch_stochastic_gradient_descent(self, x, y, batch_size=None, beta_initial=None):
    """

    Args:
        x: 所有样本的特征
        y: 所有样本的标签
        batch_size: 小批量梯度下降的批大小，默认为总样本量的 5/10
        beta_initial: 待优化的参数的初始值，默认为全零向量

    Returns:
        beta: 迭代后的最优参数

    """
    # 初始化损失函数值
    self.loss = []
    # 初始化批大小
    if batch_size is None:
        batch_size = int(x.shape[0] / 10 * 5)
    # 初始化待优化的参数
    if beta_initial is None:
        beta = np.zeros(x.shape[1])
    else:
        beta = beta_initial
    # 迭代求解
    for k in range(self.K):
        # 随机选择 batch_size 个样本
        batch = np.random.choice(
            x.shape[0], size=batch_size, replace=False)
        # 计算梯度
        gradient_beta = self.gradient(
            x[batch], y[batch], beta, self.lambda_)
        # 计算步长
        t = np.power(k + 1, -0.1)
        # 更新参数
        beta = beta - t * gradient_beta
        # 记录损失函数的值
```

```
        self.loss.append(self.f(x, y, beta, self.lambda_))
    # 将参数值小于 1e-3 的置为 0
    beta[np.abs(beta) < 1e-3] = 0

    return beta
```

1.2.6 坐标下降法求解最优参数

基于所有样本求梯度向量的一个分量，根据 $t_k = (k+1)^{-\alpha}$ 确定步长 $(\alpha = 0.1)$，迭代 K 次得到最优的参数 β。

```
# 坐标下降法，基于所有样本求梯度向量的第 j 个分量
def coordinate_descent_gradient_j(self, x, y, beta, lambda_, j):
      """

      Args:
          x: 所有样本的特征
          y: 所有样本的标签
          beta: 待优化的参数
          lambda_: 正则化调节参数
          j: 待优化的参数的索引

      Returns:
          基于所有样本求得的梯度向量的第 j 个分量

      """
      n = len(y)
      x_j = x[:, j]
      part1 = np.sum((1 / (1 + np.exp(-x.dot(beta)))) * x_j.T)
      part2 = - np.sum(y * (x_j.T))
      part3 = lambda_ * np.sign(beta[j])
      return (part1 + part2) / n + part3

def coordinate_descent(self, x, y, beta_initial=None):
    """

    Args:
        x: 所有样本的特征
        y: 所有样本的标签
        beta: 待优化的参数的初始值，默认为全零向量

    Returns:
        beta: 迭代后的最优参数

    """
    # 初始化损失函数值
    self.loss = []
    # 初始化待优化的参数
    if beta_initial is None:
        beta = np.zeros(x.shape[1])
    else:
        beta = beta_initial
    # 迭代求解
    for k in range(self.K):
        # 循环更新每个分量
        j = k % x.shape[1]
        # 计算梯度
        gradient_beta_j = self.coordinate_descent_gradient_j(
```

```
            x, y, beta, self.lambda_, j)
        # 计算步长
        t = np.power(k + 1, -0.1)
        # 更新参数，只更新第 j 个分量
        beta[j] = beta[j] - t * gradient_beta_j
        # 记录损失函数的值
        self.loss.append(self.f(x, y, beta, self.lambda_))
    # 将参数值小于 1e-3 的置为 0
    beta[np.abs(beta) < 1e-3] = 0

    return beta
```

1.3 交叉验证选择最优 λ

拟合数据，得到最优参数 β，并确定最优的判别 Y_i 为 0 或 1 的概率门槛。

```
def fit(self, x, y, method='gradient_descent'):
    """

    Args:
        x: 所有样本的特征
        y: 所有样本的标签
        method: 优化方法，默认为梯度下降

    Returns:
        beta: 迭代后的最优参数

    """
    if method == 'gradient_descent':
        self.beta = self.gradient_descent(x, y)
    elif method == 'mini_batch_stochastic_gradient_descent':
        self.beta = self.mini_batch_stochastic_gradient_descent(x, y)
    elif method == 'coordinate_descent':
        self.beta = self.coordinate_descent(x, y)
    else:
        raise ValueError(
            'method must be one of gradient_descent, mini_batch_stochastic_gradient_descent,
                                                                coordinate_descent')
    self.best_thredhold = self.get_best_threshold(x, y)
```

将数据集划分为 K 折，循环检验每一个超参数下的模型评价指标，以 K 个验证集上的 F1-Score 均值作为评判最优 λ 的依据。

```
def cv(self, x, y, lambda_s, method='gradient_descent', k=5):
    """

    Args:
        x: 所有样本的特征
        y: 所有样本的标签
        lambda_s: 超参数
        method: 优化方法，默认为梯度下降
        k: 交叉验证的折数

    Returns:
        best_lambda_: 最优的超参数

    """
    # 将数据集分成k份
```

```
    x_folds = np.array_split(x, k)
    y_folds = np.array_split(y, k)
    best_lambda_ = None
    best_f1_score = 0
    # 循环超参数
    for lambda_ in lambda_s:
        # 设定超参数
        self.lambda_ = lambda_
        # 计算每一折的性能度量指标
        accuracys = []
        precisions = []
        recalls = []
        f1_scores = []
        pbar = tqdm(range(k))
        for i in pbar:
            x_train = np.vstack(x_folds[:i] + x_folds[i+1:])
            y_train = np.hstack(y_folds[:i] + y_folds[i+1:])
            x_val = x_folds[i]
            y_val = y_folds[i]
            self.fit(x_train, y_train, method=method)
            y_pred = self.predict(x_val)
            # 计算性能度量指标
            accuracy = self.accuracy(y_pred, y_val)
            presision = self.precision(y_pred, y_val)
            recall = self.recall(y_pred, y_val)
            f1_score = self.f1_score(y_pred, y_val)
            # 保存性能度量指标
            accuracys.append(accuracy)
            precisions.append(presision)
            recalls.append(recall)
            f1_scores.append(f1_score)
            # 输出进度条
            pbar.set_description(f"lambda={lambda_}, fold={i}")
            pbar.set_postfix(dict(accucary=f'{np.mean(accuracys):.2%}',
                                  presision=f'{np.mean(precisions):.2%}',
                                  recall=f'{np.mean(recalls):.2%}',
                                  f1_score=f'{np.mean(f1_scores):.2%}'))
        # 计算平均f1_score
        mean_f1_score = np.mean(f1_scores)
        if mean_f1_score > best_f1_score:
            best_f1_score = mean_f1_score
            best_lambda_ = lambda_
    # 设定最优超参数
    self.lambda_ = best_lambda_
    # 返回最优超参数
    return best_lambda_
```

```python
# 寻找最优的正则化参数
lambda_s= [0.001, 0.005, 0.01, 0.05, 0.1]
for method in ['gradient_descent', 'mini_batch_stochastic_gradient_descent', 'coordinate_descent']:
    print(method.center(80, '-'))
    globals()['best_lambda_'+method] = lr.cv(x_train, y_train, lambda_s= lambda_s, method=method)
```

3m 20.9s

```
--------------------------------gradient_descent--------------------------------

lambda=0.001, fold=4: 100%|████████| 5/5 [00:12<00:00,  2.41s/it, accucary=89.20%, presision=63.49%, recall=66.90%, f1_score=64.77%]
lambda=0.005, fold=4: 100%|████████| 5/5 [00:27<00:00,  5.57s/it, accucary=90.50%, presision=68.91%, recall=65.60%, f1_score=67.06%]
lambda=0.01, fold=4: 100%|████████| 5/5 [00:31<00:00,  6.25s/it, accucary=90.20%, presision=68.51%, recall=64.48%, f1_score=66.03%]
lambda=0.05, fold=4: 100%|████████| 5/5 [00:52<00:00, 10.44s/it, accucary=84.70%, presision=51.84%, recall=58.78%, f1_score=52.23%]
lambda=0.1, fold=4: 100%|████████| 5/5 [01:00<00:00, 12.19s/it, accucary=73.50%, presision=31.26%, recall=57.39%, f1_score=39.64%]

--------------------mini_batch_stochastic_gradient_descent--------------------

lambda=0.001, fold=4: 100%|████████| 5/5 [00:11<00:00,  2.39s/it, accucary=90.00%, presision=66.53%, recall=66.66%, f1_score=66.31%]
lambda=0.005, fold=1:  40%|███     | 2/5 [00:04<00:06,  2.14s/it, accucary=90.00%, presision=62.59%, recall=67.20%, f1_score=64.81%]
```

图 1: 逻辑回归交叉验证程序运行截图

2 MCMC 算法实现贝叶斯逻辑回归模型

2.1 L1 正则化与拉普拉斯先验的等价性

若对参数 $\boldsymbol{\beta}$ 施加拉普拉斯先验：

$$P(\beta_j) = f(\beta_j|\mu, b) = \frac{1}{2b}\exp(-\frac{|\beta_j - \mu|}{b})$$

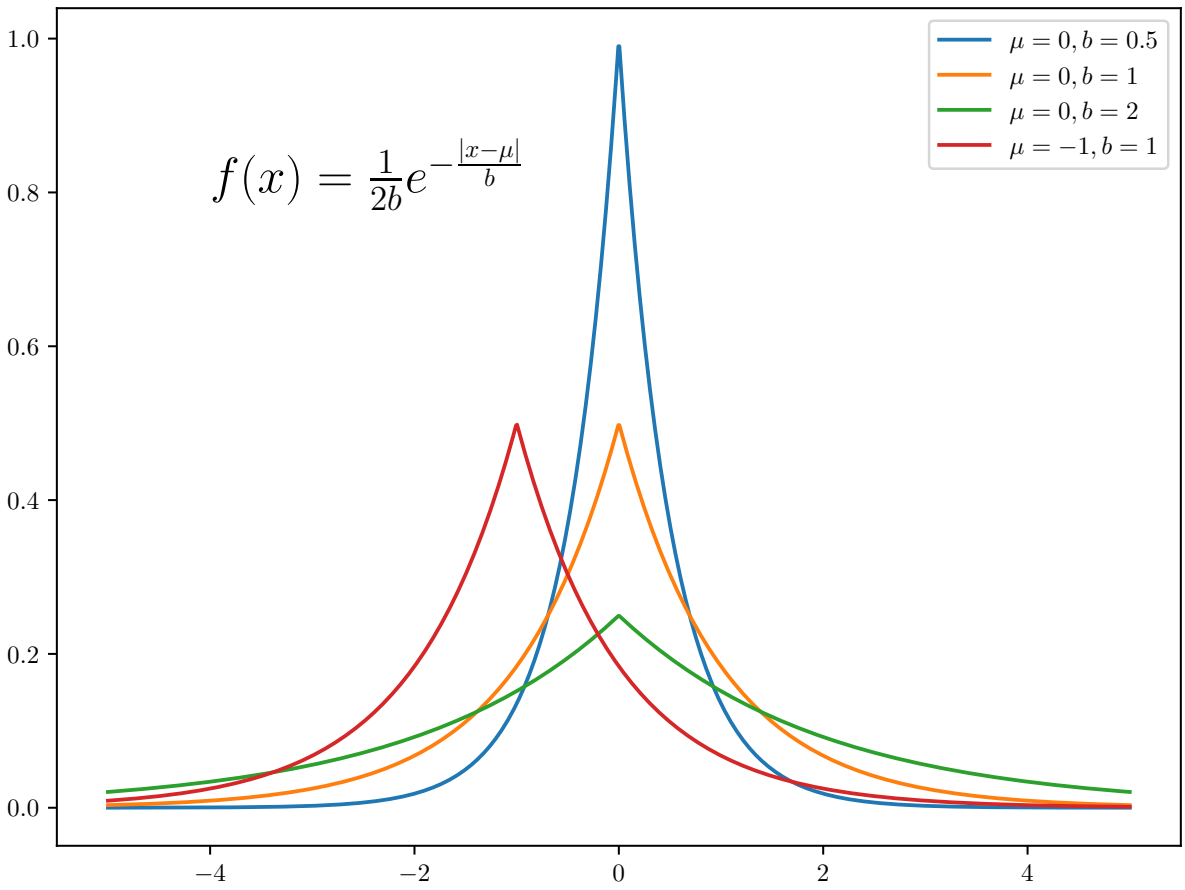

图 2: 拉普拉斯分布的概率密度函数

$\boldsymbol{\beta}$ 的后验分布为：

$$P(\boldsymbol{\beta}|\boldsymbol{X},\boldsymbol{Y}) \propto P(\boldsymbol{Y}|\boldsymbol{X},\boldsymbol{\beta})P(\boldsymbol{\beta})$$

最大化后验分布，求得最优的 $\boldsymbol{\beta}$，即为贝叶斯逻辑回归模型的参数。

$$\begin{aligned}
\boldsymbol{\beta}^* &= \underset{\boldsymbol{\beta}}{\operatorname{argmax}} \left(\prod_i P\left(Y_i \mid \boldsymbol{X}_i, \boldsymbol{\beta}\right) \prod_j P\left(\boldsymbol{\beta}_j\right) \right) \\
&= \underset{\boldsymbol{\beta}}{\operatorname{argmax}} \prod_{i=1}^{n} \left(\frac{\exp\left(\boldsymbol{X}_i^\top \boldsymbol{\beta}\right)}{1+\exp\left(\boldsymbol{X}_i^\top \boldsymbol{\beta}\right)} \right)^{I(Y_i=1)} \left(\frac{1}{1+\exp\left(\boldsymbol{X}_i^\top \boldsymbol{\beta}\right)} \right)^{I(Y_i=0)} \prod_{j=1}^{p} \frac{1}{2b} e^{-\frac{|\boldsymbol{\beta}_j-\mu)}{b}} \\
&= \underset{\boldsymbol{\beta}}{\operatorname{argmax}} \sum_{i=1}^{n} \left(Y_i \exp\left(\boldsymbol{X}_i^\top \boldsymbol{\beta}\right) - \ln\left(1+\exp\left(\boldsymbol{X}_i^\top \boldsymbol{\beta}\right)\right) \right) - \sum_{j=1}^{p} \ln(2b) - \sum_{j=1}^{p} \frac{|\boldsymbol{\beta}_j - \mu|}{b} \\
&= \underset{\boldsymbol{\beta}}{\operatorname{argmin}} \sum_{i=1}^{n} l\left(Y_i, \boldsymbol{X}_i^\top \boldsymbol{\beta}\right) + \sum_{j=1}^{p} \ln(2b) + \sum_{j=1}^{p} \frac{|\boldsymbol{\beta}_j - \mu|}{b} \\
&= \underset{\boldsymbol{\beta}}{\operatorname{argmin}} \sum_{i=1}^{n} l\left(Y_i, \boldsymbol{X}_i^\top \boldsymbol{\beta}\right) + \frac{1}{b} \sum_{j=1}^{p} |\boldsymbol{\beta}_j - \mu| \\
&= \underset{\boldsymbol{\beta}}{\operatorname{argmin}} \frac{1}{n} \sum_{i=1}^{n} l\left(Y_i, \boldsymbol{X}_i^\top \boldsymbol{\beta}\right) + \lambda \sum_{j=1}^{p} |\boldsymbol{\beta}_j|
\end{aligned}$$

最后一步中，令 $\mu = 0, b = \frac{1}{n\lambda}$，即可得到 L1 正则化的贝叶斯逻辑回归模型。

2.2 求解最优参数的 MCMC 算法实现

2.2.1 MCMC 类

定义 MCMC 类，它继承自 base 类。初始化时可以指定拉普拉斯先验的参数 b，以及最大迭代次数 K。

```
class MCMC(base):
    def __init__(self, b=0.5, K=1000):
    """

    Args:
    b: 拉普拉斯先验分布的参数，默认为0.5
    K: 最大迭代次数，默认为1000

    """
    super().__init__()
    self.b = b
    self.K = K
```

2.2.2 先验、似然和后验概率

定义先验概率、似然概率和后验概率的计算函数。

```
# 先验分布的概率密度函数，即拉普拉斯分布，其中mu=0
def prior(self, beta, mu=0):
    """

    Args:
        beta: 参数

    Returns:
        prior_value: 先验分布的概率密度函数
    """
    b = self.b
    prior_value = 1
    for beta_i in beta:
        prior_value *= 1/(2*b) * np.exp(-np.abs(beta_i-mu)/b)
    return prior_value

# 似然函数
def likelihood(self, beta, x, y):
    """

    Args:
        beta: 参数
        x: 所有样本的特征
        y: 所有样本的标签

    Returns:
        likelihood_value: 似然函数

    """
    likelihood_value = 1
    for i in range(len(y)):
        if y[i] == 1:
```

```
            likelihood_value *= 1/(1+np.exp(-x[i].dot(beta)))
        else:
            likelihood_value *= (1-1/(1+np.exp(-x[i].dot(beta))))
    return likelihood_value

# 后验分布的概率密度函数
def posterior(self, beta, x, y):
    """

    Args:
        beta: 参数
        x: 所有样本的特征
        y: 所有样本的标签

    Returns:
        posterior_value: 后验分布的概率密度函数

    """
    posterior_value = self.prior(beta) * self.likelihood(beta, x, y)
    return posterior_value
```

2.2.3 MCMC 采样

定义接受率函数和采样函数。采用逐分量 MH 算法进行采样，即每次只对一个参数进行采样，其他参数保持不变。

目标函数为 $\boldsymbol{\beta}$ 的后验概率。采用随机游走算法，每次采样的新值为上一次采样的值加上一个随机数，即 $\beta_j^{(t+1)} = \beta_j^{(t)} + \epsilon$，其中 $\epsilon \sim N(0,1)$。

Algorithm 4 逐分量 MH 算法

Input 所有样本的特征 x, 所有样本的标签 y, 迭代初始值 $\boldsymbol{\beta}$(默认为零向量)

Output K 次采样得到的样本 samples

function SAMPLE($x, y, \boldsymbol{\beta}$)
  accept $\leftarrow 0$
  if 未提供 $\boldsymbol{\beta}$ 的初始值 **then**
    $\boldsymbol{\beta} \leftarrow \mathbf{0}$
  end if
  for $k \leftarrow 0$ to $(K-1)$ **do**
    for $i \leftarrow 0$ to $p-1$ **do**
      $\boldsymbol{\beta}' \leftarrow \boldsymbol{\beta}$
      $\epsilon \leftarrow \text{Normal}(0,1)$
      $\boldsymbol{\beta}'_i \leftarrow \boldsymbol{\beta}_i + \epsilon$
      $accept\_rate \leftarrow \text{alpha}(\boldsymbol{\beta}, \boldsymbol{\beta}', x, y)$
      if $accept\_rate > \text{Uniform}(0,1)$ **then**
        $\boldsymbol{\beta} \leftarrow \boldsymbol{\beta}'$
        accept $\leftarrow accept + 1$
      end if
    end for
    samples$[k] \leftarrow \boldsymbol{\beta}$
  end for
  return samples
end function

```python
# 接受率函数
def alpha(self, beta, beta_new, x, y):
    """

    Args:
        beta: 参数
        beta_new: 新的参数
        x: 所有样本的特征
        y: 所有样本的标签

    Returns:
        alpha_value: 接受率

    """
    alpha_value = min(1, self.posterior(
        beta_new, x, y)/self.posterior(beta, x, y))
    return alpha_value

# MCMC采样
def sample(self, x, y, beta_init=None, verbose=True):
    """

    Args:
        x: 所有样本的特征
        y: 所有样本的标签
        beta_init: 初始参数，默认为None且初始化为0
```

```
        verbose: 是否输出进度条，默认为True

    Returns:
        samples: K次采样的参数

    """
    # 初始化接受率
    accept_rate = 0
    accept = 0
    # 初始化参数
    if beta_init is None:
        beta = np.zeros(x.shape[1])
    else:
        beta = beta_init
    # 迭代K次，生成数量为K的样本
    if verbose:
        pbar = tqdm(range(self.K))
    else:
        pbar = range(self.K)
    for k in pbar:
        # 逐个分量进行更新
        for i in range(len(beta)):
            # 生成新的参数，它等于原参数加上随机游走变量，随机游走变量来自正态分布N(0, 1)
            beta_new = beta.copy()
            beta_new[i] = beta_new[i] + np.random.normal(0, 1)
            # 计算接受率
            alpha_value = self.alpha(beta, beta_new, x, y)
            # 以接受率的概率接受新的参数
            if np.random.uniform(0, 1) < alpha_value:
                beta = beta_new
                accept += 1
        # 保存样本
        # 第一次迭代时，初始化样本
        if k == 0:
            samples = beta.copy()
        # 后续迭代时，将样本添加到样本集中
        else:
            samples = np.vstack((samples, beta))
    # 记录接受率
    accept_rate = accept/(self.K*len(beta))
    self.accept_rate = accept_rate
    # 记录样本
    self.samples = samples
    # 记录参数均值
    self.beta = np.mean(samples, axis=0)
    # 记录最优门槛
    self.best_thredhold = self.get_best_threshold(x, y)
    return samples
```

2.3 交叉验证选择最优 b

拉普拉斯分布先验的参数 b 决定了 L1 惩罚项的系数。对于这一模型超参数，我们同样可以采用交叉验证的方式选择最优的 b。与梯度下降法中的交叉验证过程类似，我们将数据集分为训练集和验证集，对于每一个 b，我们在训练集上训练模型，然后在验证集上评估模型的性能，最终选择验证集上平均 F1-Score 最好的 b 作为最终模型的超参数。

```
def cv(self, x, y, b_list, k=5):
```

```
    """

    Args:
        x: 所有样本的特征
        y: 所有样本的标签
        b_list: 超参数
        k: 交叉验证的折数

    Returns:
        best_b: 最优超参数

    """
    # 将数据集分成k份
    x_folds = np.array_split(x, k)
    y_folds = np.array_split(y, k)
    best_lambda_ = None
    best_f1_score = 0
    # 循环超参数
    for b in b_list:
        # 设定超参数
        self.b = b
        # 计算每一折的性能度量指标
        accuracys = []
        precisions = []
        recalls = []
        f1_scores = []
        pbar = tqdm(range(k))
        for i in pbar:
            x_train = np.vstack(x_folds[:i] + x_folds[i+1:])
            y_train = np.hstack(y_folds[:i] + y_folds[i+1:])
            x_val = x_folds[i]
            y_val = y_folds[i]
            self.sample(x_train, y_train, verbose=False)
            y_pred = self.predict(x_val)
            # 计算性能度量指标
            accuracy = self.accuracy(y_pred, y_val)
            presision = self.precision(y_pred, y_val)
            recall = self.recall(y_pred, y_val)
            f1_score = self.f1_score(y_pred, y_val)
            # 保存性能度量指标
            accuracys.append(accuracy)
            precisions.append(presision)
            recalls.append(recall)
            f1_scores.append(f1_score)
            # 输出进度条
            pbar.set_description(f"b={b}, fold={i}")
            pbar.set_postfix(dict(accucary=f'{np.mean(accuracys):.2%}',
                                  presision=f'{np.mean(precisions):.2%}',
                                  recall=f'{np.mean(recalls):.2%}',
                                  f1_score=f'{np.mean(f1_scores):.2%}'))
        # 计算平均f1_score
        mean_f1_score = np.mean(f1_scores)
        if mean_f1_score > best_f1_score:
            best_f1_score = mean_f1_score
            best_b = b
    # 设定最优超参数
    self.b = best_b
    # 返回最优超参数
    return best_b
```

2.4 收敛性诊断

我们将生成的马氏链的累计均值对迭代次数作图，观察遍历均值图是否趋向于一条水平的直线。在训练集上估计参数 β，可以看到，随着迭代次数的增加，参数值的遍历均值已经趋于一条水平的直线，如图3所示，说明算法已经收敛。

```
def plot_mean(self):
    cumsum = self.samples.cumsum(axis=0)
    cum_iteration = np.arange(1, self.K+1).reshape(-1, 1)
    cummean = cumsum/cum_iteration
    plt.figure(figsize=(10, 6))
    plt.plot(cummean)
    plt.legend(list(map(lambda i: r'$\beta_{'+str(i)+r'}$', list(range(self.samples.shape[0])))),
               loc=(1.01, 0.1), fontsize=14)
    plt.xlabel('迭代次数', usetex=False)
    plt.ylabel('遍历均值', usetex=False)
    plt.show()
```

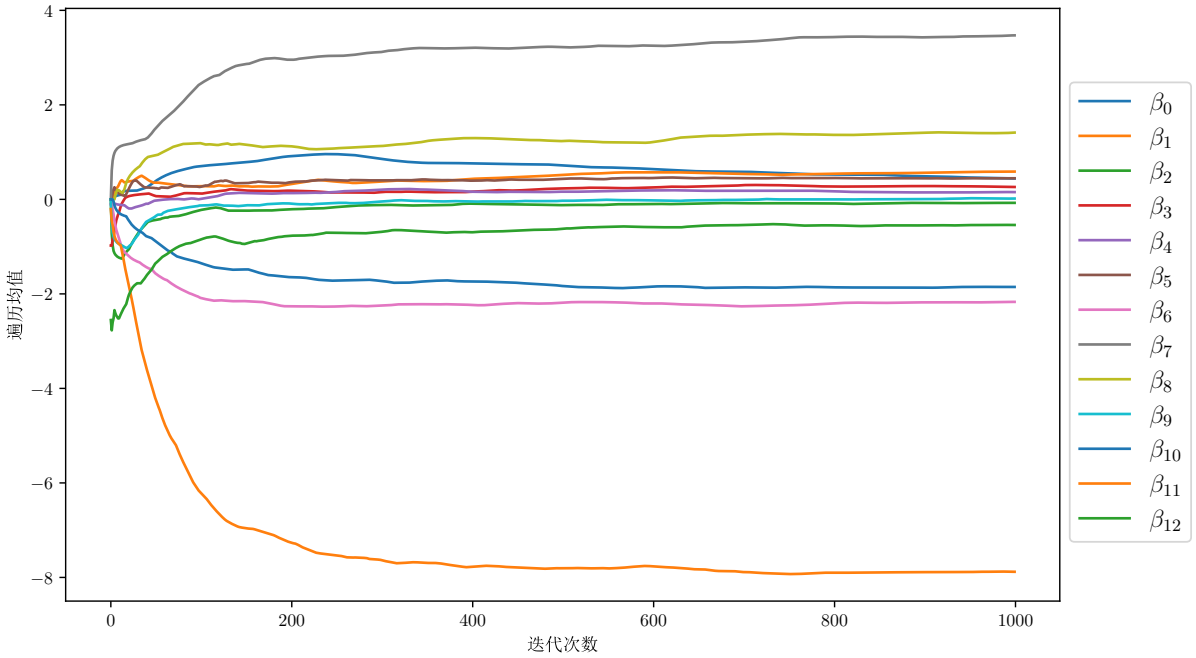

图 3: 遍历均值图

2.5 贝叶斯推断

2.5.1 后验均值

若马氏链收敛，则我们可以将累计均值作为后验均值。经检验，迭代次数大于 200 时马氏链基本收敛，因此我们限制迭代次数必须大于 200。

```
def mean(self):
    """

    Returns:
        mean_value: 后验分布的均值
```

```
    """
    assert self.samples.shape[0] > 200, '请至少迭代200次，以确保样本收敛'
    mean_value = np.mean(self.samples[200:], axis=0)
    return mean_value
```

在训练集上估计参数 β，得到的均值为：

```
[ 0.32,  0.65, -0.04,  0.28,  0.16,  0.47, -2.15,  3.6 ,  1.49, 0.04, -1.9 , -8.04, -0.48]
```

2.5.2 后验置信区间

给定置信水平 α，我们可以根据收敛后的样本，计算后验置信区间。具体地，我们舍去前 200 个样本，根据剩余样本的累计分位数，即可得到后验置信区间。

```
def interval(self, alpha=0.05):
    """

    Args:
        alpha: 置信水平

    Returns:
        interval_value: 后验分布的置信区间

    """
    assert self.samples.shape[0] > 200, '请至少迭代200次，以确保样本收敛'
    interval_value = np.percentile(
        self.samples[200:], [alpha/2*100, (1-alpha/2)*100], axis=0)
    return interval_value
```

在训练集上估计参数 β，得到的后验置信区间为：

```
[[-0.28, -0.11, -0.5 , -0.24, -0.45, -0.24, -2.86,  2.76,  0.47, -0.58, -2.8 , -9.47, -1.48],
 [1.2 ,  1.23,  0.57,  0.95,  0.59,  1.28, -1.36,  4.67,  2.97, 0.82, -1.14, -6.82,  0.42]]
```

3 实证建模与结果分析

3.1 特征工程

对数据进行预处理，主要操作包括：

1. 对文本特征进行 `LabelEncoder()` 编码。
2. 将标签数据转换为 0 或 1。其中，`'Existing Customer'` 设为 0，`'Attrited Customer'` 设为 0。
3. 将特征进行最小最大归一化。
4. 移除低方差（归一化后方差小于 0.02）特征。
5. 移除高相关特征中方差较低的一个特征。
6. 移除与标签相关性绝对值小于 0.02 的特征。

7. 一共筛选得到 12 个原始特征，分别是 Customer_Age、Education_Level、Marital_Status、Income_Category、Card_Category、Total_Relationship_Count、Months_Inactive_12_mon、Contacts_Count_12_mon、Credit_Limit、Total_Revolving_Bal、Total_Trans_Ct 和 Avg_Utilization_Ratio。

8. 为特征数据加上偏置项 1 后，特征维度为 13。

特征工程的代码见附录B。

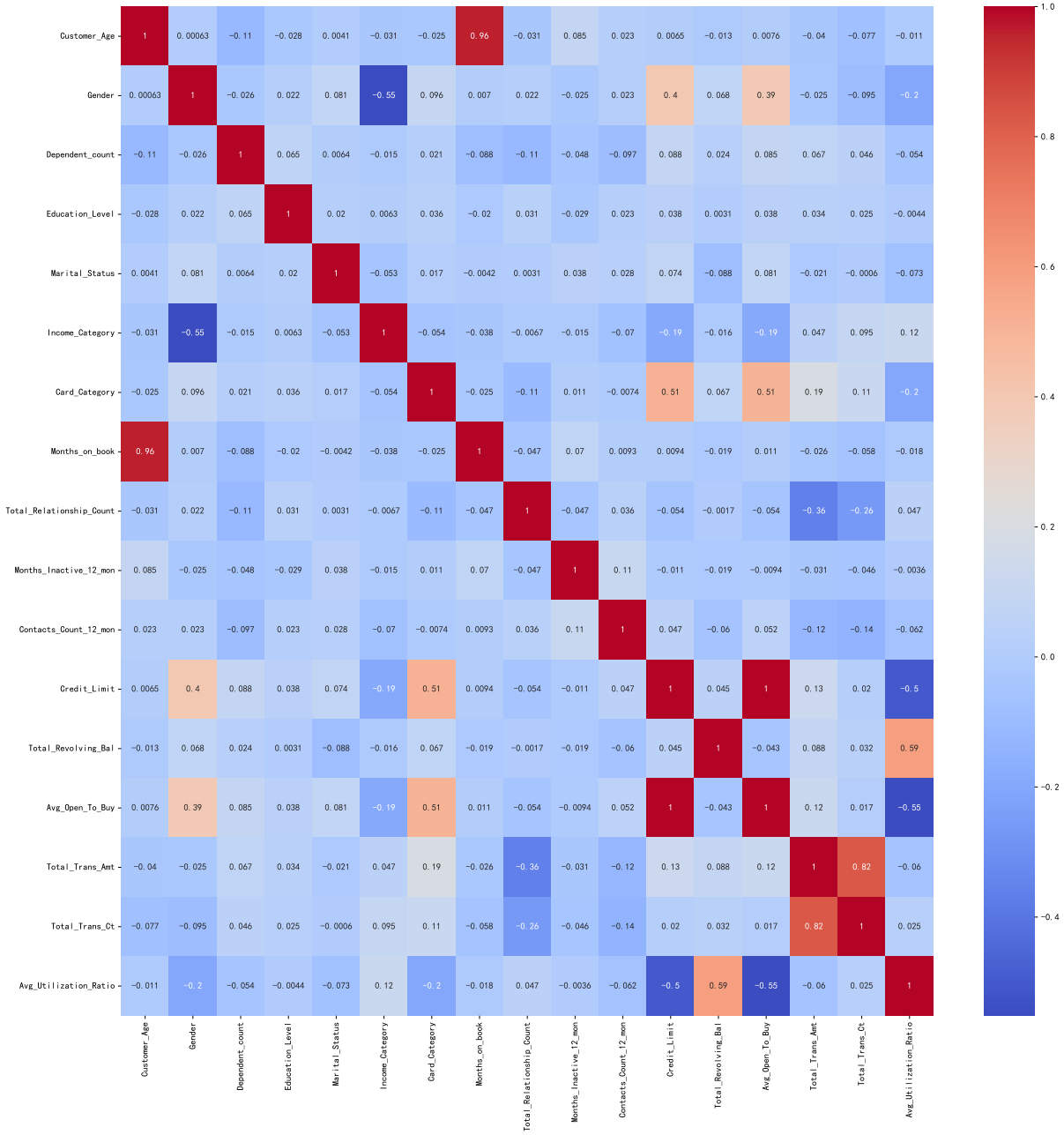

图 4: 相关系数热力图

3.2 交叉验证选择合适的正则化系数 λ

应用上文定义的 LogisticRegression 类，设最大迭代次数 $K = 10000$，使用 5 折交叉验证，得到验证集上的平均 F1-Score 值随 L1 正则项系数 λ 的变化如图5所示。

观察到参数 $\lambda = 0.005$ 时，三种梯度下降法的验证集 F1-Score 值均较高，因此将 $\lambda = 0.005$ 作为最优参数。

对 MCMC 算法，同样使用 5 折交叉验证，取 b =[0.01, 0.05, 0.1, 0.5, 1]，得到验证集上的平均 F1-Score 值随拉普拉斯先验的参数 b 的变化如图6所示。

观察到参数 $b = 0.5$ 时，验证集 F1-Score 值较高，因此将 $b = 0.5$ 作为最优参数。

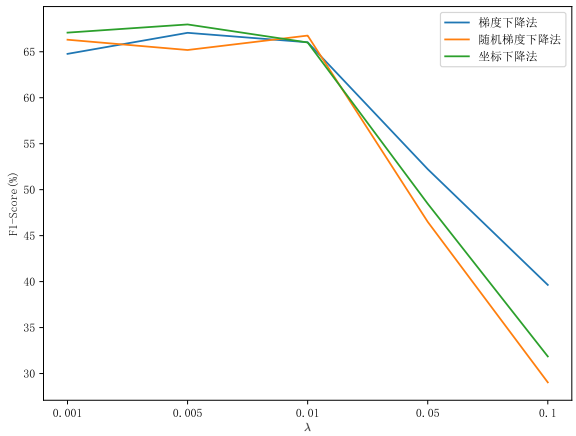

图 5: 梯度下降法的交叉验证

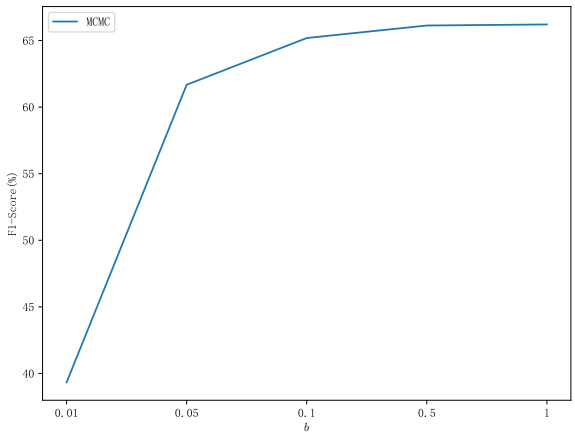

图 6: MCMC 的交叉验证

3.3 跳出局部最优解

在编写代码的过程中，我最初设置了终止条件：至少迭代了 100 轮，且损失函数的值在最近 2 轮的迭代中变化小于 1e-6，则直接停止迭代。

```
# 终止条件：至少迭代了100轮，且损失函数的值在最近2轮的迭代中变化小于1e-6
if k > 100 and abs(self.loss[-2] - self.loss[-1]) < 1e-6:
    break
```

但在绘制三种梯度下降算法的损失值后，我发现梯度下降法的收敛值与其他两种方法不一致，如图7所示。

为了解决这个问题，我在观察到损失函数值不发生变化时，对当前参数加上一个随机扰动项 $N(0, 0.01^2)$，帮助跳出局部最优解。

```
# 改变当前值的条件：至少迭代了100轮，且损失函数的值在最近2轮的迭代中变化小于1e-6
if k > 100 and abs(self.loss[-2] - self.loss[-1]) < 1e-6:
    # 为每个分量加上噪音N(0, 0.01^2)
    beta = beta + np.random.normal(0, 1e-2, beta.shape)
```

对于三种梯度下降算法，绘制损失函数随迭代次数的变化关系如图8所示。

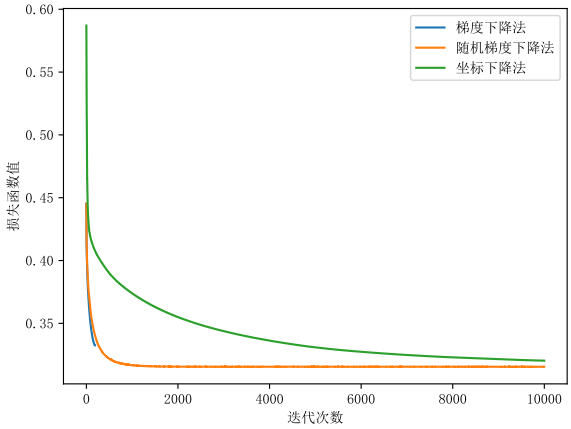

图 7: 局部最优解

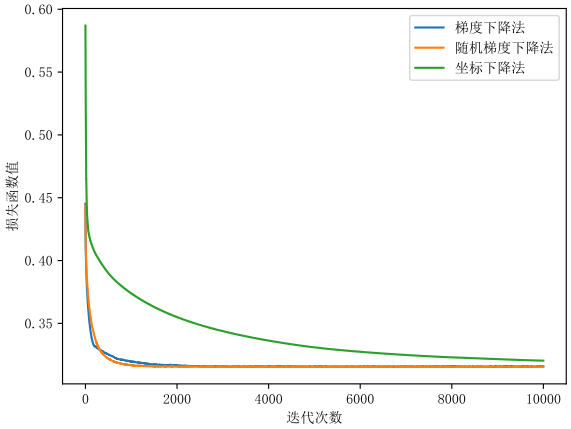

图 8: 跳出局部最优解

3.4 估计 β 及其现实意义

对于 MCMC 算法，我们通过采样得到了 K 个参数值。通过遍历均值图来考察算法的收敛性。可以看到，随着迭代次数的增加，参数值的遍历均值已经趋于一条水平的直线，如图9所示，说明算法已经收敛。我们将遍历均值作为参数的估计值。

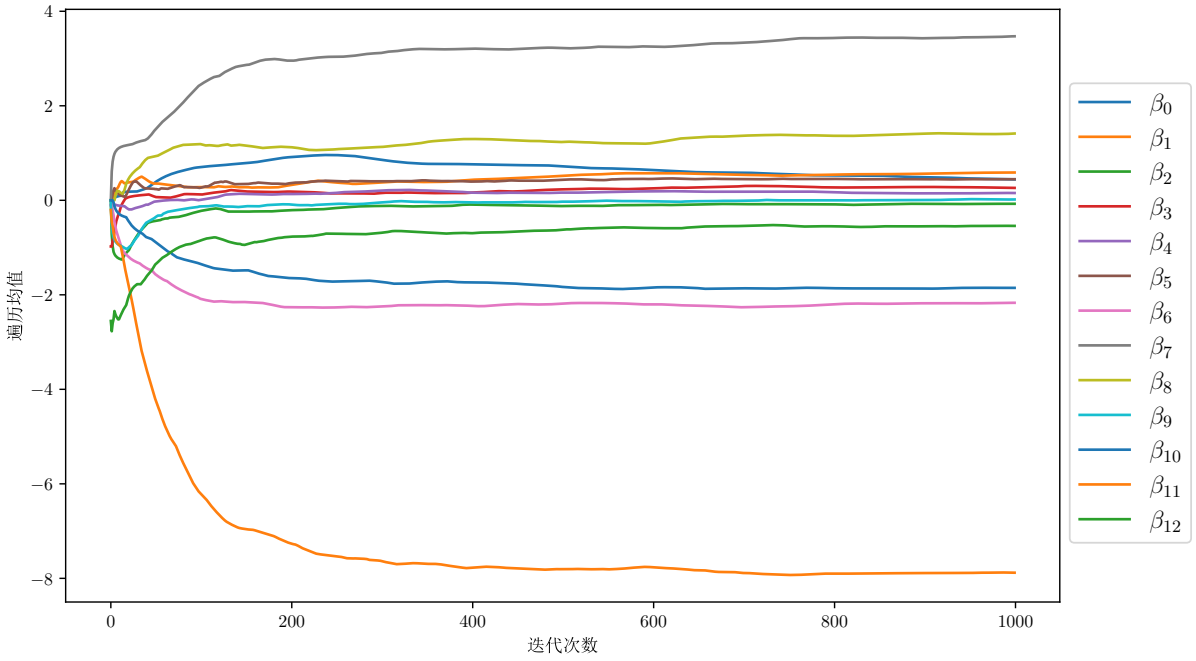

图 9: 遍历均值图

应用梯度下降法（GD）、随机梯度下降法（SGD）、坐标下降法（CD）和 MCMC 算法（MCMC），得到的 β 的估计值为：

Method	beta

```
|   GD   | [0.02, 0.18, 0.00, 0.07, 0.02, 0.00, -1.52, 2.68, 0.97, 0.07, -1.39, -5.47, -0.28]  |
|  SGD   | [0.00, 0.22, 0.00, 0.00, 0.00, 0.00, -1.46, 2.79, 0.99, 0.00, -1.46, -5.59, -0.19]  |
|   CD   | [0.00, 0.18, 0.00, 0.00, -0.00, 0.00, -1.37, 1.9, 0.77, 0.00, -1.29, -4.08, -0.49]  |
|  MCMC  | [0.21, 0.64, -0.04, 0.39, 0.24, 0.38, -2.27, 3.58, 1.59, 0.12, -1.93, -7.94, -0.45] |
+--------+-------------------------------------------------------------------------------------+
```

4 种方法给的各 β 的估计值如图10所示。四种方法对 β 的各分量的估计中，相对大小基本一致，但 MCMC 算法的 L1 惩罚力度较小。这是因为 MCMC 算法中拉普拉斯先验分布的超参数 b 与 LASSO 算法中的 λ 不完全等价。

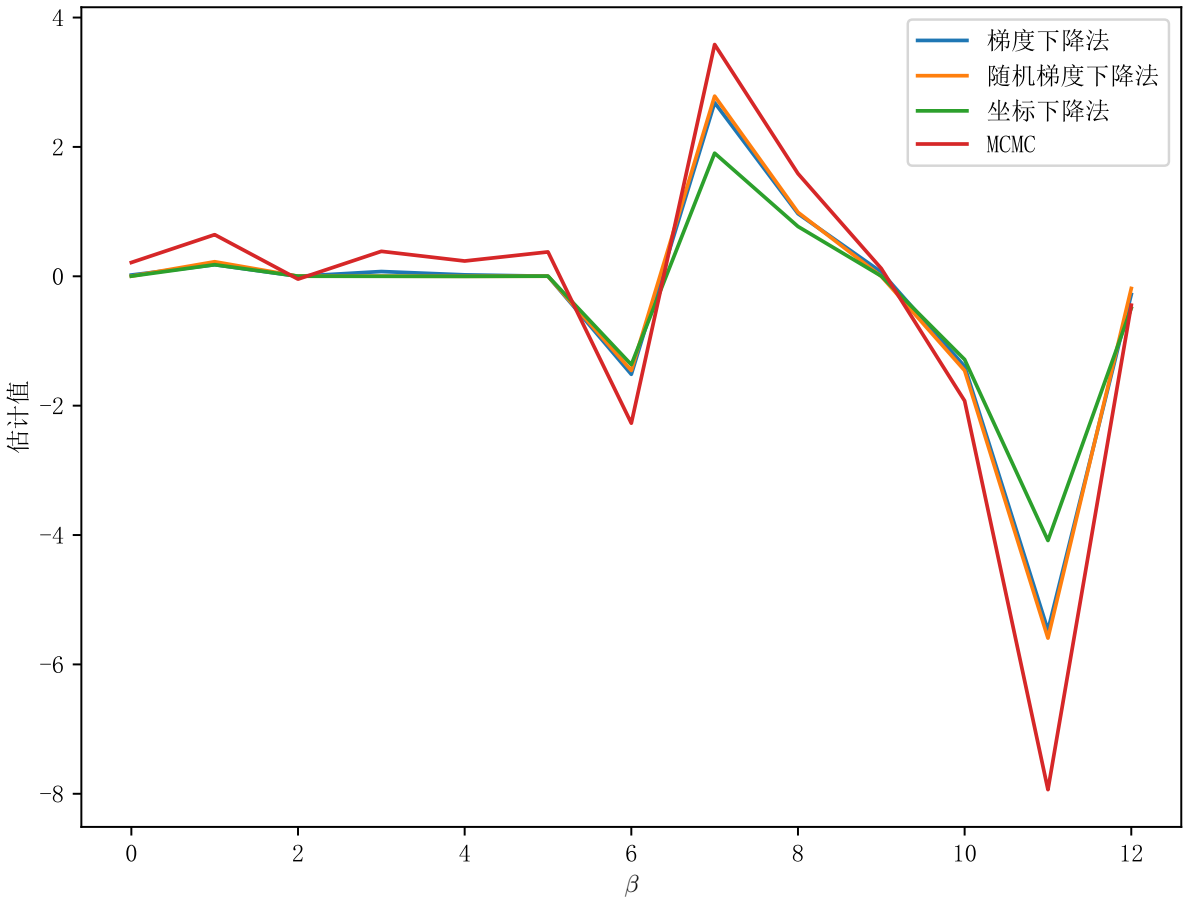

图 10: β 估计值

进一步考察 β 的各分量对应的特征，发现 Total_Relationship_Count、Total_Revolving_Bal、Total_Trans_Ct 和 Avg_Utilization_Ratio 的系数显著为负。在现实意义上，这说明顾客所持有的产品数量越多、信用卡总额度越高、近一年内的平均信用卡使用率越高，则该顾客越不容易流失。

相反，Months_Inactive_12_mon 和 Contacts_Count_12_mon 的系数显著为正。这说明顾客近一年内未使用产品的月数越多、近一年内与银行的沟通次数越多（可能是遇到业务困扰而需要帮助），则该顾客越容易流失。

3.5 测试集上的模型评价

在测试集上，使用 4 种方法得到的模型评价指标如表1所示。

表 1: 测试集上的模型评价指标

Method	Accuracy	Precision	Recall	F1
梯度下降法	0.86	0.79	0.45	0.58
随机梯度下降法	0.86	0.72	0.5	0.59
坐标下降法	0.85	0.69	0.52	0.59
MCMC	0.85	0.71	0.48	0.57

在测试集上，使用 4 种方法得到的模型评价指标如图11所示。可以看出，准确率高于 79%，即优于基于样本均值的随机猜测（测试集中的正例比例为 79%）。

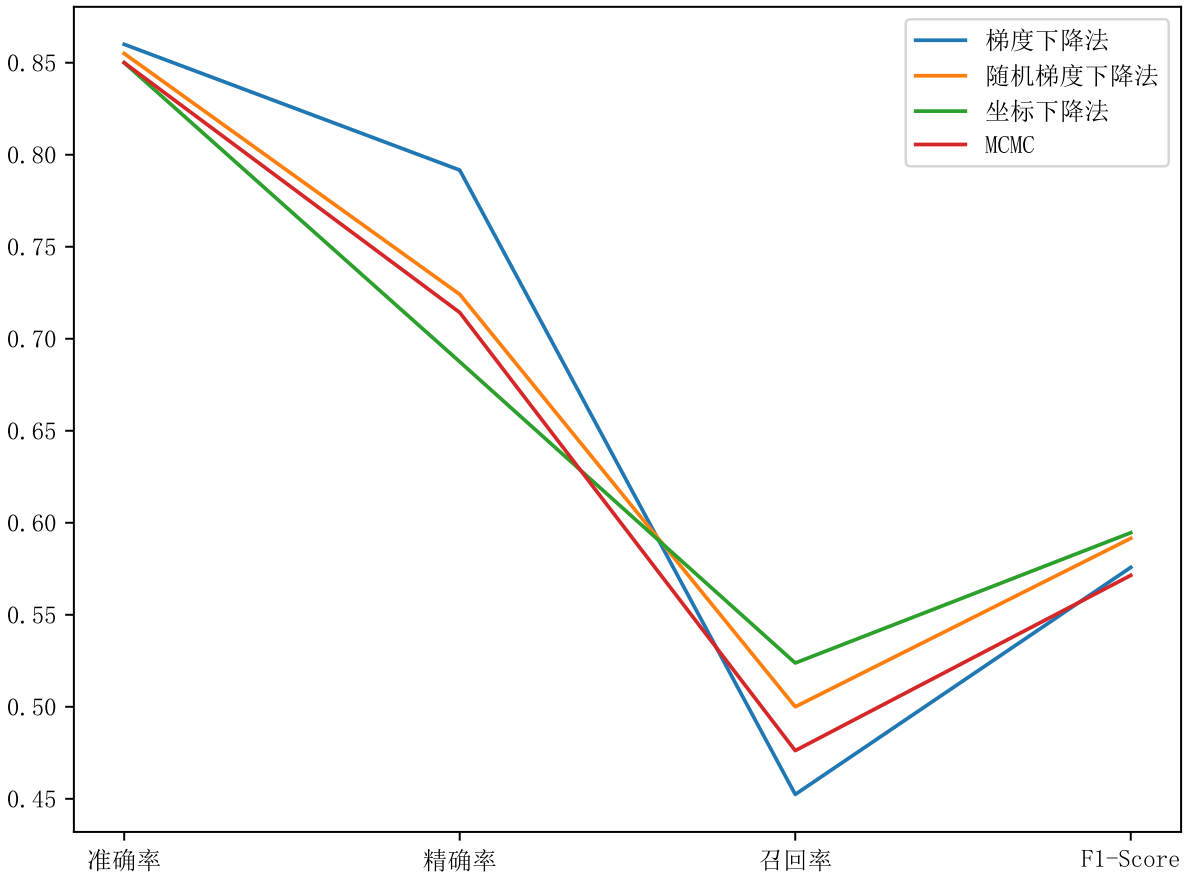

图 11: 测试集上的模型评价指标

在测试集上，4 种方法的 ROC 曲线如图12所示。

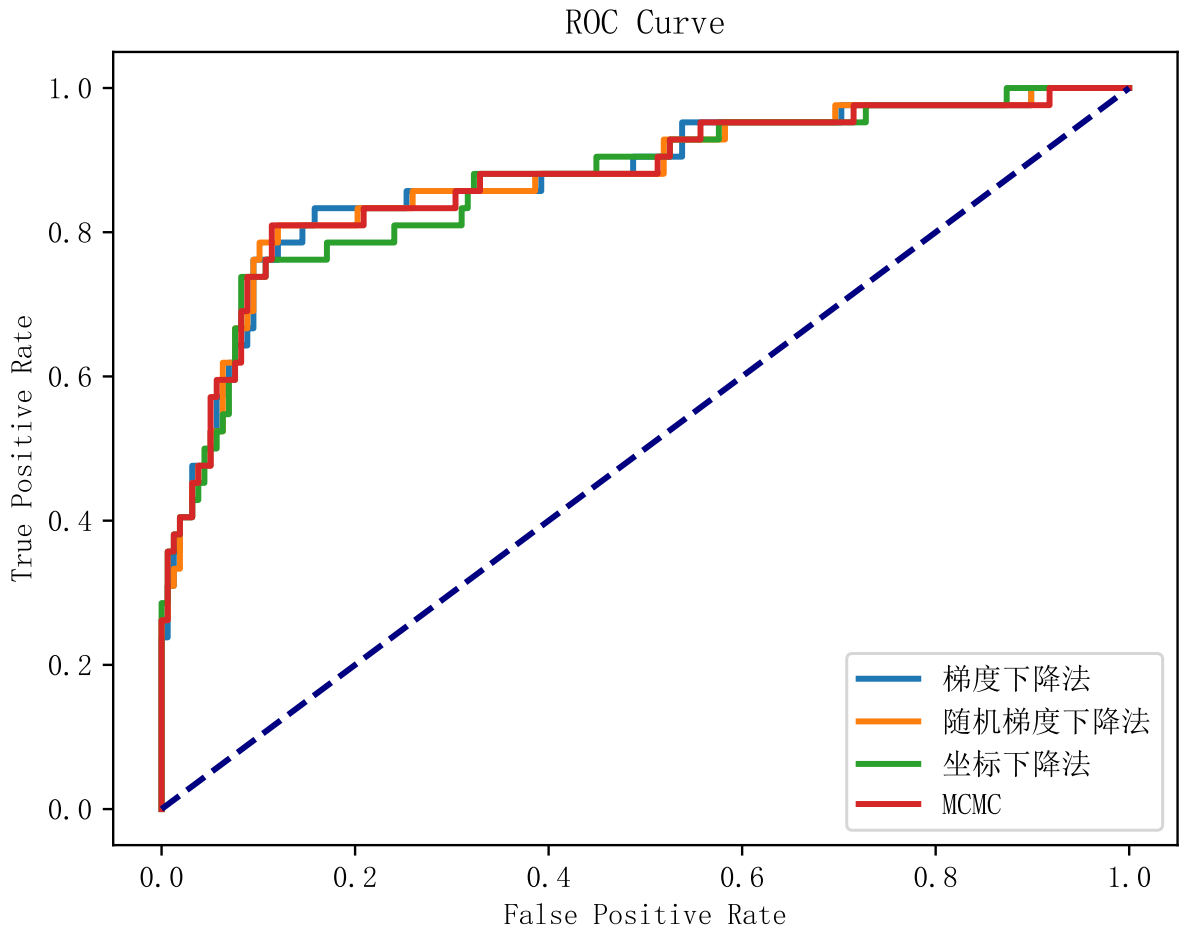

图 12: ROC 曲线

4 种方法的预测误差基本相近，ROC 曲线也基本一致，这也说明了 LASSO 回归和施加拉普拉斯先验的贝叶斯模型具有等价性。

A 绘制拉普拉斯分布的概率密度函数

```
# 绘制拉普拉斯分布的概率密度函数
def laplace(mu, b, x):
    return 1/(2*b) * np.exp(-abs(x-mu)/b)

x = np.linspace(-5, 5, 1000)

plt.rcParams['text.usetex'] = True
fig = plt.figure(figsize=(8, 6))

mu = 0
for b in [0.5, 1, 2]:
    plt.plot(x, laplace(mu, b, x), label=r'$\mu={}, b={}$'.format(mu, b))

mu = -1
b = 1
plt.plot(x, laplace(mu, b, x), label=r'$\mu={}, b={}$'.format(mu, b))
```

```
# 在图中添加文字，显示概率密度函数
plt.text(-4, 0.8, r'$f(x)=\frac{1}{2b}e^{-\frac{|x-\mu|}{b}}$', fontsize=20)
plt.legend()
plt.savefig("拉普拉斯分布的概率密度函数.pdf", format="pdf", bbox_inches="tight")
plt.show()
```

B 特征工程代码

```
# 导入包
import pandas as pd
import numpy as np
import seaborn as sns
import matplotlib.pyplot as plt
plt.rcParams['font.sans-serif'] = ['SimHei']
plt.rcParams['axes.unicode_minus'] = False
from scipy.stats import pearsonr
from sklearn.preprocessing import LabelEncoder
from sklearn.preprocessing import MinMaxScaler
from sklearn.feature_selection import VarianceThreshold
# 读取数据
train = pd.read_excel('BankChurners.xlsx', sheet_name='training data')
test = pd.read_excel('BankChurners.xlsx', sheet_name='testing data')
x_train = train.drop(['Attrition_Flag'], axis=1)
y_train = train['Attrition_Flag']
x_test = test.drop(['Attrition_Flag'], axis=1)
y_test = test['Attrition_Flag']
# 变量编码
# 提取文本类型的特征
text_features = [column for column in x_train.columns if x_train[column].dtype == 'object']
# 对文本类型的特征进行编码
le = LabelEncoder()
for column in text_features:
    x_train[column] = le.fit_transform(train[column])
    x_test[column] = le.transform(test[column])
# 对标签进行编码
y_train.replace({'Existing Customer': 0, 'Attrited Customer': 1}, inplace=True)
y_test.replace({'Existing Customer': 0, 'Attrited Customer': 1}, inplace=True)
# 特征选择
## 移除低方差特征
# 删除无意义的ID列
x_train.drop(['ID'], axis=1, inplace=True)
x_test.drop(['CLIENTNUM'], axis=1, inplace=True)
# 将特征进行最小最大归一化
scaler = MinMaxScaler()
x_train = pd.DataFrame(scaler.fit_transform(x_train), index=x_train.index, columns=x_train.columns)
x_test = pd.DataFrame(scaler.transform(x_test), index=x_test.index, columns=x_test.columns)
# 移除低方差（归一化后方差小于0.02）特征
selector = VarianceThreshold(threshold=0.02)
selector.fit(x_train)
# 打印因为低方差被移除的特征
print('因为低方差被移除特征：', x_train.columns[~selector.get_support()])
x_train = x_train.loc[:, selector.get_support()]
x_test = x_test.loc[:, selector.get_support()]
## 移除高相关特征中方差较低的那一个
# 绘制特征之间的相关性热力图
plt.figure(figsize=(20, 20))
```

```
corr = x_train.corr()
sns.heatmap(corr, annot=True, cmap='coolwarm')
plt.show()
# 打印相关性绝对值大于0.8的特征
corr = corr.abs()
corr = corr.where(np.triu(np.ones(corr.shape), k=1).astype(bool))
corr = corr.stack()
corr = corr[corr > 0.8]
print('相关性绝对值大于0.8的特征: ', [i for i in corr.index])
# 从数据集中移除相关性绝对值大于0.8的特征中方差较小的特征
feature_to_remove = []
corr = corr.reset_index()
corr.columns = ['feature1', 'feature2', 'corr']
for row in corr.itertuples():
    if x_train[row.feature1].var() < x_train[row.feature2].var():
        feature_to_remove.append(row.feature1)
    else:
        feature_to_remove.append(row.feature2)
feature_to_remove = list(set(feature_to_remove))
print('相关性绝对值大于0.8的特征中方差较小的特征: ', feature_to_remove)
x_train.drop(feature_to_remove, axis=1, inplace=True)
x_test.drop(feature_to_remove, axis=1, inplace=True)
## 移除与标签相关性较低的特征
# 移除与标签相关性绝对值小于0.02的特征
feature_to_remove = []
for i in x_train.columns:
    pearson_corr = pearsonr(x_train[i], y_train)[0]
    if abs(pearson_corr)<0.02:
        feature_to_remove.append(i)
print('与标签相关性绝对值小于0.02的特征: ', feature_to_remove)
x_train.drop(feature_to_remove, axis=1, inplace=True)
x_test.drop(feature_to_remove, axis=1, inplace=True)
# 打印最终选择的特征
print('最终选择的特征', x_train.columns.values)
# 打印最终选择的特征
print('最终选择的特征', x_train.columns.values)
# 移除与标签相关性绝对值小于0.02的特征
feature_to_remove = []
for i in x_train.columns:
    pearson_corr = pearsonr(x_train[i], y_train)[0]
    if abs(pearson_corr)<0.02:
# 导出特征选择后的数据集
x_train.to_csv('x_train.csv', index=False)
x_test.to_csv('x_test.csv', index=False)
y_train.to_csv('y_train.csv', index=False)
y_test.to_csv('y_test.csv', index=False)
# 导出特征选择后的数据集
x_train.to_csv('x_train.csv', index=False)
x_test.to_csv('x_test.csv', index=False)
y_train.to_csv('y_train.csv', index=False)
y_test.to_csv('y_test.csv', index=False)
        feature_to_remove.append(i)
print('与标签相关性绝对值小于0.02的特征: ', feature_to_remove)
x_train.drop(feature_to_remove, axis=1, inplace=True)
x_test.drop(feature_to_remove, axis=1, inplace=True)
```

C base 类代码

```python
class base:
    def __init__(self, beta=None):
        """

        Args:
            beta: 待优化的参数

        """
        self.beta = beta
        self.best_threshold = None

    def predict_proba(self, x):
        """

        Args:
            x: 待预测样本的特征

        Returns:
            y: 预测的标签

        """
        y_prob = 1 / (1 + np.exp(-x.dot(self.beta)))
        return y_prob

    def get_best_threshold(self, x, y):
        """

        Args:
            x: 所有样本的特征
            y: 所有样本的标签

        Returns:
            best_thredhold: 最优的阈值

        """
        # 计算预测的标签
        y_prob = self.predict_proba(x)
        # 计算最优的阈值，依据F1值的大小
        best_thredhold = 0
        best_f1 = 0
        for thredhold in np.linspace(0, 1, 1001):
            y_pred = (y_prob >= thredhold).astype(int)
            presision = np.mean(y[y_pred == 1] == 1) if np.mean(
                y_pred == 1) else 1  # 如果预测的样本中没有正样本，则精度为1
            recall = np.mean(y_pred[y == 1] == 1)
            f1 = 2 * presision * recall / (presision + recall)
            if f1 > best_f1:
                best_f1 = f1
                best_thredhold = thredhold
        return best_thredhold

    def predict(self, x, threshold=None):
        """

        Args:
            x: 待预测样本的特征
            threshold: 阈值，默认为使准确率达到最优的阈值

        Returns:
```

```
        y: 预测的标签

    """
    if threshold is None:
        assert self.best_thredhold is not None
        threshold = self.best_thredhold
    y_prob = self.predict_proba(x)
    y = np.where(y_prob > threshold, 1, 0)
    return y

def accuracy(self, y_pred, y):
    """

    Args:
        y_pred: 所有样本的预测标签
        y: 所有样本的标签

    Returns:
        accuracy: 预测的准确率

    """
    accuracy = np.mean(y_pred == y)
    return accuracy

def precision(self, y_pred, y):
    """

    Args:
        y_pred: 所有样本的预测标签
        y: 所有样本的标签

    Returns:
        precision: 预测的精确率

    """
    precision = np.mean(y[y_pred == 1] == 1)
    return precision

def recall(self, y_pred, y):
    """

    Args:
        y_pred: 所有样本的预测标签
        y: 所有样本的标签

    Returns:
        recall: 预测的召回率

    """
    recall = np.mean(y_pred[y == 1] == 1)
    return recall

def f1_score(self, y_pred, y):
    """

    Args:
        y_pred: 所有样本的预测标签
        y: 所有样本的标签
```

```
        Returns:
            f1_score: 预测的 f1_score

        """
        precision = self.precision(y_pred, y)
        recall = self.recall(y_pred, y)
        f1_score = 2 * precision * recall / (precision + recall)
        return f1_score

    def print_performance(self, y_pred, y):
        """

        Args:
            y_pred: 所有样本的预测标签
            y: 所有样本的标签

        Returns:
            None

        """
        accuracy = self.accuracy(y_pred, y)
        precision = self.precision(y_pred, y)
        recall = self.recall(y_pred, y)
        f1_score = self.f1_score(y_pred, y)
        # 打印性能指标，以2位小数显示
        tb = pt.PrettyTable()
        tb.field_names = ["Measurement", "Value"]
        tb.add_row(["Accuracy", "{:.2f}".format(accuracy)])
        tb.add_row(["Precision", "{:.2f}".format(precision)])
        tb.add_row(["Recall", "{:.2f}".format(recall)])
        tb.add_row(["F1 Score", "{:.2f}".format(f1_score)])
        print(tb)
```

D 主程序

```
from functions import *
# 读取数据
x_train = pd.read_csv('x_train.csv')
y_train = pd.read_csv('y_train.csv')
x_test = pd.read_csv('x_test.csv')
y_test = pd.read_csv('y_test.csv')
# 将数据转换为矩阵形式
x_train = x_train.values
y_train = y_train.values.squeeze()
x_test = x_test.values
y_test = y_test.values.squeeze()
# 对特征数据加上截距项
x_train = np.hstack([np.ones((x_train.shape[0], 1)), x_train])
x_test = np.hstack([np.ones((x_test.shape[0], 1)), x_test])
# 应用梯度下降实现LASSO方法
lr = LogisticRegression(lambda_=0.005, K=10000)
# 寻找最优的正则化参数
lambda_s= [0.001, 0.005, 0.01, 0.05, 0.1]
for method in ['gradient_descent', 'mini_batch_stochastic_gradient_descent', 'coordinate_descent']:
    print(method.center(80, '-'))
    globals()['best_lambda_'+method] = lr.cv(x_train, y_train, lambda_s= lambda_s, method=method)
# 在测试集上进行预测
```

```
# 梯度下降法
lr.fit(x_train, y_train, method='gradient_descent')
pred_prob_gd = lr.predict_proba(x_test)
pred_gd = lr.predict(x_test)
acc_gd = lr.accuracy(pred_gd, y_test)
precision_gd = lr.precision(pred_gd, y_test)
recall_gd = lr.recall(pred_gd, y_test)
f1_gd = lr.f1_score(pred_gd, y_test)
# 随机梯度下降法
lr.fit(x_train, y_train, method='mini_batch_stochastic_gradient_descent')
pred_prob_sgd = lr.predict_proba(x_test)
pred_sgd = lr.predict(x_test)
acc_sgd = lr.accuracy(pred_sgd, y_test)
precision_sgd = lr.precision(pred_sgd, y_test)
recall_sgd = lr.recall(pred_sgd, y_test)
f1_sgd = lr.f1_score(pred_sgd, y_test)
# 坐标下降法
lr.fit(x_train, y_train, method='coordinate_descent')
pred_prob_cd = lr.predict_proba(x_test)
pred_cd = lr.predict(x_test)
acc_cd = lr.accuracy(pred_cd, y_test)
precision_cd = lr.precision(pred_cd, y_test)
recall_cd = lr.recall(pred_cd, y_test)
f1_cd = lr.f1_score(pred_cd, y_test)
# 应用MCMC方法
mcmc = MCMC()
# 交叉验证
mcmc.cv(x_train, y_train, b_list=[0.01, 0.05, 0.1, 0.5, 1])
# MCMC采样
samples = mcmc.sample(x_train, y_train)
# 绘制遍历均值图
mcmc.plot_mean()
mean = mcmc.mean()
interval = mcmc.interval()
# 训练集上的预测
y_pred = mcmc.predict(x_train)
mcmc.print_performance(y_pred, y_train)
# 测试集上的预测
y_pred = mcmc.predict(x_test)
mcmc.print_performance(y_pred, y_test)
```