第四章 分类

本章介绍的分类器：

- 逻辑斯蒂回归(Logistic regression)
- 线性判别分析(linear discriminant analysis)
- 二次判别分析(quadratic discriminant analysis)
- 朴素贝叶斯(naive Bayes)
- K-近邻(K-nearest neighbors)
- 广义线性模型(generalized linear models)
- 泊松回归(Poisson regression)

4.1 分类概述

分类问题的例子：

- 急诊室病人有几个病症特征，确定其疾病
- 网络银行根据交易的IP、历史记录等判断交易是否欺诈
- 在患病和不患病的动物中提取DNA，确定哪些序列会造成疾病

4.2 为什么不线性回归

在上面的急诊室例子中，若使用线性回归，则需要将可能得情况映射到一些数字中。实际上，不能确定不同情况之间的差距有多大，不同的数字编码会造成完全不一样的判断，因此不能使用线性回归(尤其是多个分类的情况下。二元回归翻转数字结果相同，但在概率上可能超过[0-1]值域)

> 例如将3种疾病作为分类，无法用数字衡量它们之间判断的差距

不使用线性回归的原因：

1. 线性回归不能处理多个取值定性变量的情况
2. 即使是两个取值，线性回归也可能无法解释超过0-1区间的预测值

> 此处的预测值指Y

4.3 逻辑斯蒂回归

用逻辑斯蒂回归，可以将概率控制在[0-1]之间；使用线性回归，概率可能超过这个值域(在二元回归中)

4.3.1 逻辑斯谛模型

用**逻辑斯蒂方程**，可以将概率的值落到[0-1]之间，方程形式如下：

$$p(X)=\frac{e^{\beta_0+\beta_1X}}{1+e^{\beta_0+\beta_1X}}$$

逻辑斯蒂方程总能将概率转化为一个S型曲线。在数学上变换我们有：

$$\frac{p(x)}{1-p(x)}=e^{\beta_0+\beta_1X}$$

其中的$\frac{p(x)}{1-p(x)}$被称为**赔率(*odds*)**，表示的是发生概率与不发生概率的比，取值总在$0-1$之间。

> 一些与赌注有关的活动常常使用赔率而不是概率

取对数后有：

$$\log(\frac{p(x)}{1-p(x)})=\beta_0+\beta_1X$$

其中$\log(\frac{p(x)}{1-p(x)})$被称为**对数赔率(*logit*)**。我们注意到在逻辑斯蒂方程中，对数赔率是X的线性函数

> 虽然概率$p(X)$并不是X的线性函数，我们仍可以发现X在正方向上的变化会引起$p(X)$在正方向上的变化

4.3.2 估计回归参数

我们试图找到估计参数β_0和β_1，使得对于所有状态为*非*的个体，获得一个概率接近0的数；对所有状态为*是*的个体，获得概率接近1的数，用**极大似然法**可以获得一个**近似函数**：

$$l(\beta_0,\beta_1)=\prod_{i:y_i=1}p(x_i)\prod_{i':y_i=0}(1-p(x_{i'})$$

> 最小二乘法是极大似然估计的一种特殊情况

4.3.3 做出预测

对于定性变量，可以考虑哑变量来设置逻辑斯蒂方程。例如将状态是设置为1，状态否设置为0

4.3.4 多元逻辑斯蒂回归

类比线性回归，多元逻辑斯蒂方程可以写为：

$$\log(\frac{p(x)}{1-p(x)})=\beta_0+\beta_1X+\beta_2X_2+\cdots+\beta_pX_p$$

或者用向量表示为：

$$z=\mathbf{w}^T\mathbf{x}+b$$

同样地：

$$p(X)=\frac{e^{\beta_0+\beta_1X+\beta_2X_2+\cdots+\beta_pX_p}}{1+e^{\beta_0+\beta_1X+\beta_2X_2+\cdots+\beta_pX_p}}$$

用极大似然法估计系数参数

用一个预测变量进行逻辑斯蒂回归发生预测错误的风险比多个预测变量时大得多，这种现象被称为**混杂(*confounding*)**

> 混杂变量的出现需要满足混杂变量与X和Y都相关，且会影响他们之间的关系

4.3.5 多项逻辑斯蒂回归

当分类情况超过两类时，逻辑斯蒂回归需要扩展为**多项逻辑斯蒂回归(*multinomia logistic regression*)**

> 此处的分类情况指的是Y的取值；应当与多元逻辑斯蒂回归相区分

此时，我们的概率模型将转换为：

$$Pr(Y=k|X=x)=\frac{e^{\beta_{k0}+\beta_{k1}x_1+\cdots+\beta_{kp}x_p}}{1+\sum_{l=1}^{K-1}e^{\beta_{l0}+\beta_{l1}x_1+\cdots+\beta_{kp}x_p}}$$

其中的$k=1,2,\cdots,K-1$，K表示分类种类数，并有：

$$Pr(Y=K|X=x)=\frac{1}{1+\sum_{l=1}^{K-1}e^{\beta_{l0}+\beta_{l1}x_1+\cdots+\beta_{lp}x_p}}$$

不难发现：

$$\log(\frac{Pr(Y=k|X=x)}{Pr(Y=K|X=x)})=\beta_{k0}+\beta_{k1}x_1+\cdots+\beta_{lp}x_p$$

> 当$X=x$时，$Y=K$的概率被记为基线(*baseline*)，基线的选择对分类结果没有影响；然而，不同基线选择会导致模型的解释情况不同，因为逻辑斯蒂回归产生的结果是相对于基准线的赔率，不同的基准线的解释情况不同

softmax编码是多分类问题中常用的一个函数，它将所有的类别视为对称的而非选择一个基线，每种类别都将输出一个对应的概率值：

$$Pr(Y=k|X=x)=\frac{e^{\beta_{k0}+\beta_{k1}x_1+\cdots+\beta_{kp}x_p}}{\sum_{l=1}^{K}e^{\beta_{l0}+\beta_{l1}x_1+\cdots+\beta_{lp}x_p}}$$

在此情况下，第k类和第k'类的比值赔率相等：

$$\log(\frac{Pr(Y=k|X=x)}{Pr(Y=k'|X=x)})=(\beta_{k0}-\beta_{k'0})+(\beta_{k1}-\beta_{k'1})x_1+\cdots+(\beta_{kp}-\beta_{k'p})x_p$$

4.4 用于分类的生成模型

逻辑斯蒂回归是一种相对直接的分类模型，在某些情况下，它的效果可能不好，例如：

- 两类之间存在明显的分离情况，此时逻辑斯蒂回归模型的参数会异常不稳定
- 预测变量X在每个类别中的分布近似正态分布，且样本量较小。利用非逻辑斯蒂回归更准确
- 部分多分类问题

考虑用**贝叶斯公式**求当$X=x$时，Y属于类别k的概率：

$$Pr(Y=k|X=x)=p_k(x)=\frac{\pi_k f_k(x)}{\sum_{l=1}^{K}\pi_l f_l(k)}$$

其中π_k表示**先验概率**，即$Y=k$的概率(第k类占比$P(Y=k)$；$f_k(x)$表示在第k类中，$X=x$的概率，也即X在$Y=k$上的**密度函数**，满足$f_k(x)=Pr(X|Y=k)$；称$p_k(x)$为**后验概率**

在一个数据集中，估计π_k是很容易的；然而，确定$f_k(x)$却困难得多。如果我们能找到一个近似的方法估计$f_k(x)$，我们就能快速获取后验概率

4.4.1 线性判别分析

4.4.1.1 单特征线性判别分析

首先假设$f_k(x)$是 **正态分布(*normal*)** 的，并在单维度设定下，正态密度：

$$f_k(x)=\frac{1}{\sqrt{2\pi}\sigma_k}\exp(-\frac{1}{2\sigma_k^2}(x-\mu_k)^2)$$

> 特征x按高斯分布

其中，μ_k是第k类的均值，σ_k^2是第k类的方差

> 符号exp表示指数函数e的幂

进一步假设$\sigma_1^2=\sigma_2^2=\cdots=\sigma_K^2$，也就是说$K$类都是共方差的。用$\sigma^2$表示并代入之前的$p_k(x)$的贝叶斯公式中计算，有：

$$p_k(x)=\frac{\pi_k\frac{1}{\sqrt{2\pi}\sigma}exp(-\frac{1}{2\sigma^2}(x-\mu_k)^2)}{\sum_{l=1}^{K}\pi_l\frac{1}{\sqrt{2\pi}\sigma}exp(-\frac{1}{2\sigma^2}(x-\mu_l)^2)}$$

贝叶斯分类器将$X=x$分类到可能概率最大的一个类别中去。对上式取对数，我们得到一个简化的**判别函数**：

$$\delta_k(x)=x\frac{\mu_k}{\sigma^2}-\frac{\mu_k^2}{2\sigma^2}+log(\pi_k)$$

也就是找到最大的$\delta_k(x)$，使得$X=x$获得分类k。例如在两个分类等可能出现时，贝叶斯边界即为$\sigma_1(x)=\sigma_2(x)$处的预测变量x取值，此时：

$$x=\frac{\mu_1^2-\mu_2^2}{2(\mu_1-\mu_2)}=\frac{\mu_1+\mu_2}{2}$$

线性判别分析(*linear discriminant analusis, LDA*) 通过将π_k、μ_k和σ^2的估计值代入简化后的判别函数来进行概率的计算。参数估计的公式为：

$$\hat{\mu}_k = \frac{1}{n_k} \sum_{i:y_i=k} x_i$$

$$\hat{\sigma}^2 = \frac{1}{n-K} \sum_{k=1}^{K} \sum_{i:y_i=k} (x_i - \hat{\mu}_k)^2$$

其中$\sum_{i:y_i=k} x_i$表示对所有属于第k类的样本点求和，n为训练观测数据的总数，n_k表示在训练数据中的k类个数，μ_k表示训练数据中第k类的均值，σ^2表示K个样本变量的加权方差

有时候π_k可以直接获取，否则需要估计：

$$\hat{\pi}_k = \frac{n_k}{n}$$

> 也就是第k特征的出现频率

将所有估计好的参数代入判断函数：

$$\hat{\delta}_k(x) = x\frac{\hat{\mu}_k}{\hat{\sigma}^2} - \frac{\hat{\mu}_k}{2\hat{\sigma}^2} + \log(\hat{\pi}_k)$$

则我们将$X = x$判断为第k类的依据是$\hat{\delta}_k(x)$最大

> 线性判别分析的"线性"即是来源于判别函数是x的线性函数

注意：贝叶斯决策边界就是两种分类在后验概率下相等的区域；训练数据集的决策边界是找到这样的一条线，使得不同的分类分隔开。这二者是不同的

4.4.1.2 多特征线性判别分析

多元高斯分布(*multivariate Gaussian*)：若一个随机向量$\mathbf{X} = (X_1, X_2, \ldots, X_p)$中的每个元素都满足正态分布，且协方差矩阵相同，则这个随机向量满足多元高斯分布。高斯分布的形状称为**钟型**；当预测变量是相关的，或者方差不相等，钟型将被扭曲

> 满足二元高斯分布的图像从X_1或X_2轴切割得到的截面形状具有一维正态分布的形状

为了说明一个p维随机向量是正态分布的，我们记作：

$$\mathbf{X} \backsim N(\mu, \Sigma)$$

其中$\mu = E(X)$是向量X的期望，$\Sigma = Cov(X)$是p个元素的$p \times p$协方差矩阵

多元高斯分布的密度函数：

$$f(x) = \frac{1}{(2\pi)^{p/2}|\Sigma|^{1/2}} \exp(-\frac{1}{2}(x-\mu)^T\Sigma^{-1}(X-\mu))$$

LDA分类器假设第k类的观测值都来自于多元高斯分布$N(\mu_k, \Sigma)$。其中μ_k是这类的均值向量，Σ是所有K类的共有的协方差矩阵

用类似的数学运算，我们可以由贝叶斯公式推到得到多元线性判别的简化判断函数：

$$\delta_k(x) = x^T\Sigma^{-1}\mu_k - \frac{1}{2}\mu_k^T\Sigma^{-1}\mu_k + \log\pi_k$$

> 这可以看做是一个向量/矩阵版的判别公式

当它最大时进行分类，这实际上是单特征判断函数的向量形式。计算贝叶斯决策边界满足$\delta_k(x) = \delta_l(x)$，即：

$$x^T\Sigma^{-1}\mu_k - \frac{1}{2}\mu_k^T\Sigma^{-1}\mu_k = x^T\Sigma^{-1}\mu_l - \frac{1}{2}\mu_l^T\Sigma^{-1}\mu_l$$

通过判断函数我们发现，$\delta_k(x)$是x中元素的线性组合，这也是LDA中线性的意思。

用模型参数p和样本数n的比值$\frac{p}{n}$来表示过拟合程度，容易发现这个值越小说明越不可能出现过拟合

检验一个分类器是否分类得好，不应该只关注它的错误率。事实上，有些分类器的错误率不比一个零分类器好，这是因为数据集本身的“错误率”就很低

> 对于一个**零分类器(*null*)**，即不论什么预测向量X都将其归为一类，这样的分类器导致的错误率跟我们模型的错误率进行比对，若相差不大，我们需要考虑模型的优化或数据的选取问题

通常，一个二值分类器(*Binary Classifier*)会导致两类错误，即**取伪**或**去真**。使用**混淆矩阵(*confusion matrix*)** 直观地显示有关错误的信息，我们通常期望其中一类错误尽可能小，因此可以降低其中一个概率取值的判断阈值

> 在医学或生物学中，**敏感性(*sensitivity*)** 指的是很少错过真正有病的病例，即去真少；**特异性(*specificity*)** 指的是很少将健康人误诊为病例，即取伪少

在实际问题的应用上，由于LDA会将所有的错误率按尽可能低的方式逼近贝叶斯分类器，因此可能对两类错误的取舍不符合实际问题的预期。所以，我们有必要开发更符合实际问题的分类器，例如改变取舍**阈值**。一般来说，增大阈值会减小敏感性，减小阈值会减小特异性

我们用**ROC曲线(*ROC curve*)** 来表示分类器性能的权衡。ROC曲线的横轴表示取伪率，纵轴表示取真率。理想分类器的ROC曲线特别靠近左上角$(0,1)$点， 面积AUC越大表示分类性能越好，当$AUC = 0.5$时相当于随机分类，小于0.5时分类效果不如随机分类

> 比较模型性能，可以通过绘制模型的ROC曲线，然后计算AUC值

4.4.3 二次判别分析

与LDA不同，**二次判别分析(*Quadratic Discriminant Analysis*)** 考虑每一个类都有一个协方差矩阵，每个协方差矩阵不一定相同，也就是类内数据散布不同。来自第k类的观测满足$X \backsim N(\mu_k, \Sigma_k)$，其中$\Sigma_k$表示第$k$类的协方差矩阵。此时判别函数：

$$\begin{aligned}\delta_k(x) &= -\frac{1}{2}(x-\mu_k)^T\Sigma_k^{-1}(x-\mu_k) - \frac{1}{2}\log|\Sigma_k| + \log\pi_k \\ &= -\frac{1}{2}x^T\Sigma_k^{-1}x + x^T\Sigma_k^{-1}\mu_k - \frac{1}{2}\mu_k^T\Sigma_k^{-1}\mu_k - \frac{1}{2}|\Sigma_k| + \log\pi_k\end{aligned}$$

> QDA比LDA更加灵活，适合类别之间数据散布不同的情况

我们需要估计Σ_k、μ_k、π_k

对于不同的数据集，LDA和QDA的适用不同。如果贝叶斯边界为线性的，LDA更能反映实际情况；否则，QDA更加反应实际情况

> 注意δ_k是一个x的二次函数，因此得名QDA

LDA与QDA的选择取决于方差-偏差权衡。LDA拥有更低的方差，因此不如QDA灵活。因此，建议在训练集较小的时候使用LDA，训练集较大时使用QDA，防止过拟合问题

4.4.4 朴素贝叶斯

对比LDA和QDA作的假设：

- LDA假设f_k是一个多元正态随机变量的密度函数，每个类共享μ和协方差矩阵Σ
- QDA假设对于多元正态随机变量的密度函数f_k，每一类有它的μ_k和Σ_k

朴素贝叶斯(*naive Bayes*) 分类器作出了另一个假设：对于每个k类，p个预测变量都是独立的。从数学上说，这表示：

$$f_k(x) = f_{k1}(x_1) \times f_{k2}(x_2) \times \cdots \times f_{kp}(x_p)$$

这里f_{kj}是第k类观测值中第j个预测因子的密度函数

一个观测向量X内部的协变量x_i之间的相互关系通常很难判断，通过朴素贝叶斯的假设我们很快地消除了它们之间的相互关系。尽管在事实上常常并非如此，但是在观测值数n相对于观测变量p很小时，这种假设也很有用

边缘分布(*Marginal Distribution*) 是指在处理多维随机变量时，只关注部分变量的概率分布；**联合分布(*Joint distribution*)** 用于计算随机变量组合取特定值的概率，在两个随机变量独立时有$f(x,y) = f_X(x)f_Y(y)$，朴素贝叶斯使这两类计算变得很容易。特别是在n相对于p不够大时，这个假设可以使我们有效地估计联合分布

由朴素贝叶斯假设：

$$Pr(Y=k|X=x) = \frac{\pi_k \times f_{k1}(x_1) \times f_{k2}(x_2) \times \cdots \times f_{kp}(x_p)}{\sum_{l=1}^K \pi_l \times f_{l1}(x_1) \times f_{k2}(x_2) \times \cdots \times f_{kp}(x_p)}$$

现在，我们的目标是估计每个一维密度函数。下面有几种选择：

- 对于每个类中定量的随机变量X_j，直接认为每个一维密度函数服从各自的正态分布，即$X_j|Y=k \sim N(\mu_{jk}, \sigma_{jk}^2)$

> $f_{kj}(x_j)$表示在k类别中，特征X_j(特征向量中的一个)取值为x_j的概率密度

- 对于每个类中定量的随机变量X_j，考虑用非参数方法估计这些密度函数(分布律)。例如，画出直方图来估计特征取值x_j，或者考虑用**核密度估计(kernel destity esimator)**

> 直方图估计法类似于用频率估计概率，只不过变量的取值是连续的

- 对于每个类中定性的随机变量X_j，我们直接统计每个类中x_j的出现频率，用频率估计概率

朴素贝叶斯取得的分类效果不一定胜过LDA，这取决于朴素贝叶斯减少的方差相对于偏差来说不一定是值得的。选择朴素贝叶斯的情况是在p相对大n相对小的情况下，这时候，减少方差才是比较重要的

4.5 关于分类器的比较

4.5.1 分析比较

通过研究对数赔率：

$$\log(\frac{P(Y=k|X=x)}{P(Y=K|X=x)})$$

来研究不同分类器的特性

对于LDA模型，通过计算可以得到：

$$\log(\frac{P(Y=k|X=x)}{P(Y=K|X=x)})=a_k+\sum_{j=1}^{p}b_{kj}x_j$$

其中，a_k满足：

$$a_k=\log(\frac{\pi_k}{\pi_K})-\frac{1}{2}(\mu_k+\mu_K)^T\Sigma^{-1}(\mu_k-\mu_K)$$

b_{kj}满足：

$$b_{kj}=\Sigma^{-1}(\mu_k-\mu_K)$$

> a_k是一个常数项，b_{kj}可以认为是第j个特征x_j对该对数几率的权重

这说明LDA模型就像逻辑斯蒂回归模型一样，得到的关于后验概率的对数赔率是x的一个线性组合

同样的方法可以推出关于QDA的对数赔率：

$$\log(\frac{P(Y=k|X=x)}{P(Y=K|X=x)})=a_k+\sum_{j=1}^{p}b_{kj}x_j+\sum_{j=1}^{p}\sum_{l=1}^{p}c_{kjl}x_jx_l$$

其中的a_k,b_{kj},c_{kjl}都是关于$\pi_k,\pi_K,\mu_k,\mu_K,\Sigma_k$和$\Sigma_K$的函数。可以看出，QDA的对数赔率是关于$x$的二次函数

推导朴素贝叶斯的对数赔率：

$$\log(\frac{P(Y=k|X=x)}{P(Y=K|X=x)})=a_k+\sum_{j=1}^{p}g_{kj}(x_j)$$

其中的$a_k=\log(\frac{\pi_k}{\pi_K})$，$g_{kj}(x_j)=\log(\frac{f_{kj}(x_j)}{f_{Kj}(x_j)})$。这个形式也被称为**广义相加模型**

下面是一些结论：

- LDA是QDA的特殊版本，即限制了每个分类都有相同的特征分布
- 任何具有线性决策边界的分类器都是朴素贝叶斯的一个特例，这意味着LDA是朴素贝叶斯的一个特例
- 如果在朴素贝叶斯中使用一维高斯分布对每个特征分布进行建模，可以发现朴素贝叶斯是LDA的一个特例，即特征之间的协方差阵时对角型矩阵
- 朴素贝叶斯是一个**加性模型**，在特征独立时优势明显；QDA适用于捕捉特征之间的交互性

> 上面的分析指出，没有一种分类器占据绝对优势，而应该根据实际选取

逻辑斯蒂回归的对数赔率是：

$$\log(\frac{P(Y=k|X=x)}{P(Y=K|X=x)})=\beta_{k0}+\sum_{j=1}^{p}\beta_{kj}x_j$$

可以发现形式与LDA类似。二者之间的差别在于逻辑斯蒂回归的参数选择是为了满足极大似然函数，而LDA的选择是基于贝叶斯公式。我们期望在高斯分布假设成立时，LDA优于逻辑斯蒂回归，否则逻辑斯蒂回归更优

KNN模型相对于上面的模型来说有这样的特征：

- 完全没有参数，拥有非线性决策边界。因此在贝叶斯边界是非线性时表现得比LDA和逻辑斯蒂回归好
- KNN需要大量的训练数据，即$n>>p$。这是因为它倾向于减少偏差，同时产生大量方差
- 样本量小时，或者说n不大p不小时，QDA比KNN更优。
- 不同于逻辑斯蒂回归，KNN是不会说明哪些预测因子是重要的(无参数)

4.5.2 经验比较

为了比较这几种模型的适用情况，书中生成了6组不同的数据集用于检测。其中，前3种拥有线性决策边界，后三种拥有非线性决策边界，下面是比较的结论：

- 没有一种方法在任何情况下都占据绝对优势
- 当真实决策边界是线性的，LDA和逻辑斯蒂回归表现得较好
- 当真实决策边界中度非线性，QDA或朴素贝叶斯表现得较好
- 当真实决策边界非常复杂，KNN等非参数方法更具有优势。尽管如此，也应当正确地选择平滑程度

在线性回归中使用的X^2、X^3等高次交互项也可以在分类器中使用。不过，要注意偏差-方差均衡

4.6 广义线性回归

在实际生活中，有一些预测变量Y的取值既不是定性的也不是定量的，例如比率、集合、高维度数据等。在这种情况下，使用线性回归方法或分类器方法都不能达到良好的效果，我们因此引入**广义线性回归(Generalized Linear Models)**

本部分使用了“BikeShare”数据集

4.6.1 用线性回归拟合Bikeshare

在用线性回归拟合Bikeshare数据集时出现了下面的问题：

- 负数预测值，不符合实际
- 过强的异方差性，即随机误差是关于随机变量的函数，不符合线性回归假设
- 连续的预测区间，不符合实际(离散型)

使用对数变换，即令$\log Y = \sum_{j=1}^{p} X_j\beta_j + \epsilon$可以减小异方差，但是这导致了解释上的困难(每单位的变量改变对应的响应变量的变化不好解释)

4.6.2 泊松回归

泊松分布(Poisson) 是一类关于离散型随机变量的分布，事件X发生k次的概率取值为：

$$P(X=k)=\frac{e^{-\lambda}\lambda^k}{k!}\quad k=0,1,2,\cdots$$

泊松分布的特点是均值和方差相等，即：

$$E(X)=D(X)=\lambda$$

这说明随着$E(X)$的变大，$D(X)$会跟着变大

泊松分布常常用于对**可数量(counts)** 的建模，因为泊松分布取非负值。

我们将一天中的Biker建模为满足某个参数为λ的泊松分布，这就是**泊松回归(Poisson Regression)**。此处，λ是变量$\mathbf{X}=(X_1,X_2,\cdots,X_p)$的函数，记为：

$$\log(\lambda(X_1,X_2,\cdots,X_p))=\beta_0+\beta_1X_1+\beta_2X_2+\cdots+\beta_pX_p$$

等价于：

$$\lambda(X_1,X_2,\cdots,X_p)=e^{\beta_0+\beta_1X_1+\beta_2X_2+\cdots+\beta_pX_p}$$

其中的β_i是需要估计的参数

考虑用极大似然估计法估计这些参数，有：

$$l(\beta_0,\beta_1,\cdots,\beta_p)=\prod_{i=1}^{n}\frac{e^{-\lambda(x_i)}\lambda(x_i)^{y_i}}{y_i!}$$

其中的$\lambda(x_i)=e^{\beta_0+\beta_1x_{i1}+\beta_2x_{i2}+\cdots+\beta_px_{ip}}$。我们找到一组$\beta_i$的取值使得上式最大

下面是关于泊松回归的特点：

- 参数 为了解释泊松回归模型的参数，我们需要关注λ关于特征向量的函数。不同的特征向量变动导致不同的λ取值，使我们可以对概率作出预测
- 均值-方差关系 泊松回归假定均值和方差会同时发生变动，而回归分析则假设均值和方差保持不变。因此，在某些情况下，泊松回归能正确地捕捉均值和方差的变动关系
- 非负值拟合 泊松回归得到的结果都是非负值，而线性回归可能导致负值出现

4.6.3 更具普遍性的广义回归模型

我们讨论过的三种回归模型，即线性回归、逻辑斯蒂回归和泊松回归有一些相同的特性：

1. 响应变量Y都被假设服从某个分布。线性回归假设Y服从正态分布，逻辑斯蒂回归假设Y服从伯努利分布(二项分布)，泊松回归假设Y服从泊松分布
2. 响应变量Y的均值都被建模为了特征向量的一个函数。线性回归中
   $E(Y|X_1,X_2,\cdots X_p)=\beta_0+\beta_1X_1+\cdots+\beta_pX_p$，逻辑斯蒂回归中
   $E(Y=1|X_1,X_2,\cdots X_p)=\frac{e^{\beta_0+\beta_1X_1+\cdots+\beta_pX_p}}{1+e^{\beta_0+\beta_1X_1+\cdots+\beta_pX_p}}$， 泊松回归中
   $E(Y|X_1,\cdots,X_p)=e^{\beta_0+\beta_1X_1+\cdots+\beta_pX_p}$

对于性质2，我们抽象出这些表达式都使用了所谓的**连接函数(link function)** η来建模。通过使用链接函数，能够将$E(Y|X_1,\cdots,X_p)$转换为特征向量的一个线性函数，也就是：

$$\eta(E(Y|X_1,\cdots,X_p))=\beta_0+\beta_1X_1+\cdots+\beta_pX_p$$

上面三种模型的链接函数分别是：

- 线性回归 $\eta(\mu)=\mu$
- 逻辑斯蒂回归 $\eta(\mu)=\log(\mu/(1-\mu))$
- 泊松回归 $\eta(\mu)=\log(\mu)$

高斯分布、伯努利分布和泊松分布都属于**指数分布族(exponential family)**。此外，属于指数分布族的分布还有**指数分布(expoential distribution)**、**Γ分布(Gamma distribution)** 和 **负二项分布（negative binomial distribution)**。总的来说，属于指数分布族的分布都可以用广义回归模型来建模，即找到一个估计函数η来估计$E(Y|X_1,\cdots,X_p)$。我们上面提到的三种模型都是广义回归模型的例子

4.7 实验：逻辑斯蒂回归，LDA，QDA和KNN

本次实验使用的数据集时Sock Market数据集。这是一个关于股票市场的数据集

下面是本次实验用到的头文件，这是之前用过的头文件：

```
import numpy as np
import pandas as pd
from matplotlib.pyplot import subplots
import statsmodels.api as sm
from ISLP import load_data
from ISLP.models import (ModelSpec as MS, summarize)
```

这是新引入的头文件：

```
from ISLP import confusion_table
from ISLP.models import contrast
from sklearn.discriminant_analysis import \
(LinearDiscriminantAnalysis as LDA,
QuadraticDiscriminantAnalysis as QDA)
from sklearn.naive_bayes import GaussianNB
from sklearn.neighbors import KNeighborsClassifier
from sklearn.preprocessing import StandardScaler
from sklearn.model_selection import train_test_split
from sklearn.linear_model import LogisticRegression
```

生成协方差阵，查变量之间的相关性：

```
df.corr() #进行此操作前需要保证df中的变量都是数值型

_df = df.select_dtypes(include=[np.number]) #检索数值型
cov_matrix = _df.cov()
```

> 若要根据协方差阵作图，使用 `pandas` 的 `.plotting` 方法 `pd.plotting.scatter()`

4.7.1 逻辑斯蒂回归

检验逻辑斯蒂回归参数显著性的统计量是**Z值(z-score)**。它的含义是回归系数与其标准误差的比值，即：

$$z = \frac{\hat{\beta}}{\mathrm{err}(\hat{\beta})}$$

表示回归系数相对于标准误差的偏离程度。大的Z值表示该系数相对于其标准误差较大，对模型有显著影响

检验统计量的p值用于检验假设，若p值很小说明我们可以拒绝零假设$H_0 : \beta = 0$，认为这个统计量有效。通常，很小指的是$p \leq 0.05$或≤ 0.01

`statesmodels.api` 中的模型 `sm.GLM` 可用于拟合广义线性回归模型。调用逻辑斯蒂回归时，只需要指定指数族属性为 `Binomial`：

```
allvars = smarket.columns.drop(['Today', 'Direction', 'Year'])
design = MS(allvars)
X = design.fit_transform(smarket)
y = smarket.Direction == 'Up' #返回一个bool序列
glm = sm.GLM(y,
             X,
             family=sm.families.Binomial()) #指定参数族
results = glm.fit()
```

> 构造矩阵方法参见第三章-线性回归

除了调用广义线性回归模型，也可以直接调用逻辑斯蒂回归 `sm.Logit`

查询参数和p统计量信息：

```
results.params
results.pvalues #p统计量信息
```

用 `results` 自带的方法 `.predict()` 预测给定参数的概率取值，返回的结果是一个介于$0-1$之间的概率取值。当不指定参数时，返回的是训练集数据的预测概率：

```
probs = results.predict() #用训练集预测，指定预测集需要exog=参数
print(probs[:10])
```

为了检验模型准确性，我们可以打印混淆矩阵来查看。混淆矩阵来自 `ISLP` 包：

```
#显示混淆矩阵
labels = np.array(['Down']*1250) #生成一个值全为'Down'的数组
labels[probs>0.5] = "Up" #根据预测结果修改labels中的值
table = confusion_table(labels, smarket.Direction) #创建混淆矩阵
print(table)

#计算正确预报率
print(np.mean(labels == smarket.Direction))
```

> 混淆矩阵的对角线表示正确地预测

实际上，全部使用训练集来构建混淆矩阵常常低估错误率。为此，我们将数据集划分为训练集和测试集来评估错误率：

```
#将数据集划分为训练集和预测集，以便于检验模型正确率
train = (smarket.Year < 2005) #此处的train是布尔序列，用于构造索引
smarket_train = smarket.loc[train]
smarket_test = smarket.loc[~train]
print("Shape: ", smarket_test.shape)
```

```
#使用划分的训练集拟合模型
X_train, X_test = X.loc[train], X.loc[~train]
y_train, y_test = y.loc[train], y.loc[~train]
glm_train = sm.GLM(y_train,
                   X_train,
                   family=sm.families.Binomial())
results = glm_train.fit()
probs = results.predict(exog=X_test)
D = smarket.Direction
L_train, L_test = D.loc[train], D.loc[~train]
labels = np.array(['Down']*252)
labels[probs>0.5] = "Up"
table = confusion_table(labels, L_test) #关于测试集的混淆矩阵
print(table)
print(np.mean(labels == L_test), np.mean(labels != L_test))
```

上面的混淆矩阵结果显示，我们的模型甚至不如随机猜测！这可以理解，根据每个参数的p值，有一些参数对模型没有明显贡献，引入这些参数在提升方差的同时没有降低偏差，导致了模型的恶化。下面考虑用筛选后的参数进行预测：

```
#筛选参数后的模型
model = MS(['Lag1', 'Lag2']).fit(smarket) #参数只包含lag1、lag2
X = model.transform(smarket)
X_train, X_test = X.loc[train], X.loc[~train]
glm_train = sm.GLM(y_train,
                   X_train,
                   family=sm.families.Binomial())
results = glm_train.fit()
probs = results.predict(exog=X_test)
labels = np.array(['Down']*252)
labels[probs>0.5] = 'Up'
table = confusion_table(labels, L_test)
print(table)
print(np.mean(labels == L_test))
```

如果要用指定的参数预测，我们需要先构建预测集：

```
#制定预测集
newdata = pd.DataFrame({'Lag1': [1.2, 1.5],
                        'Lag2': [1.1, -0.8]})
newX = model.transform(newdata)
print(results.predict(newX))
```

4.7.2 LDA

在 `sklearn.discriminant_analysis` 中调用 `LinearDiscriminantAnalysis` 接口使用LDA(在示例中被重名为为 `LDA`)

创建LDA模型并拟合：

```
#创建LDA模型
smarket = load_data('sMarket')
lda = LDA(store_covariance=True) #指定共协方差

#预处理2005年之前的数据，将数据进行划分
model = MS(['Lag1', 'Lag2']).fit(smarket)
X = model.transform(smarket)
y = smarket.Direction == 'Up' #获取变量
train = (smarket.Year < 2005)
X_train, X_test = X.loc[train], X.loc[~train]
y_train, y_test = y.loc[train], y.loc[~train]
D = smarket.Direction
L_train, L_test = D.loc[train], D.loc[~train] #响应变量Y的集合
X_train, X_test = [M.drop(columns=['intercept'])
                   for M in [X_train, X_test]]
lda.fit(X_train, L_train) #用标签直接拟合
```

> LDA模型自动添加截距，因此需要把X的截距事先去掉
> LDA模型可以直接用标签拟合，而不必用Y

显示参数：

```
#拟合数据，显示信息
lda.fit(X_train, L_train)
print(lda.means_) #每个特征值的均值，被用作mu_k的估计
print("Classes: ", lda.classes_) #显示各个分类
print("Priors: ", lda.priors_) #每个"分类"的先验概率
print("Discriminant vector: ", lda.scalings_) #显示判别向量(特征值的最优系数)
```

展示混淆矩阵，评估效果，并构建概率数组：

```
#显示混淆矩阵
lda_pred = lda.predict(X_test)
table = confusion_table(lda_pred, L_test)
print(table)
lda_prob = lda.predict_proba(X_test) #用测试集构建一个关于测试数据的概率列表
print(np.all(
    np.where(lda_prob[:, 1] >= 0.5, 'Up', 'Down') == lda_pred
)) #检查概率结果是否与预测结果相等
print(np.all(
    [lda.classes_[i] for i in np.argmax(lda_prob, 1)] == lda_pred
))
print(np.sum(lda_prob[:, 0] > 0.9)) #显示所有的'Up'概率>0.9的天数
```

sklrean库中的LDA等分类器遵循一个共同的结构，简化了交叉检验之类的任务

4.7.3 QDA

QDA模型的语法规则类似于LDA

一些模型信息和准确性估计：

```
qda = QDA(store_covariance=True)
qda.fit(X_train, L_train) #拟合QDA模型

#展示模型信息
print("Params means: \n", qda.means_) #每个类中的特征值均值
print("Classes priors: \n", qda.priors_) #每个类的先验概率
print("The covariance of classes 1: \n", qda.covariance_[0]) #第一类内部各个特
征的协方差

#进行预测
qda_pred = qda.predict(X_test)
table = confusion_table(qda_pred, L_test)
print("Confusion table: \n", table)

#对测试集的准确性进行估计
print(np.mean(qda_pred == L_test))
```

值得注意的是，QDA模型的准确率达到了约60%，这表明它对此数据集的预测比较好

4.7.4 朴素贝叶斯

朴素贝叶斯模型除了一些特殊的参数外，其余操作与之前类似。显示朴素贝叶斯模型的信息：

```
#构建朴素贝叶斯模型
NB = GaussianNB()
NB.fit(X_train, L_train) #拟合朴素贝叶斯模型

#展示朴素贝叶斯模型的信息
print("Classes priors: \n", NB.class_prior_) #类的先验概率
print("Mean of each predictors in all classes: \n", NB.theta_) #显示每个类中,
每个特征的均值. 其中行表示类的数量, 列为每个特征的均值
print("Variance of each predictors in all classes: \n" ,NB.var_) #同上, 显示
方差
print("Test for the mean of predictors: \n", X_train[L_train ==
'Down'].mean())
print("Test for the variance of predictors: \n", X_train[L_train == 'Down'])
```

检验朴素贝叶斯模型的效果：

```
#检验模型效果
nb_labels = NB.predict(X_test)
table = confusion_table(nb_labels, L_test)
print("Confusion table: \n", table) #显示混淆矩阵
print(NB.predict_proba(X_test)[:5]) #显示前5行的测试集预测概率
```

4.7.5 KNN

构建KNN模型并作出预测：

```
#构建KNN模型
knn1 = KNeighborsClassifier(n_neighbors=1)
knn1.fit(X_train, L_train) #用1个邻居拟合KNN模型

#作出预测
knn1_pred = knn1.predict(X_test)
table = confusion_table(knn1_pred, L_test)
print("Confusion table: ", table)

#修改模型邻居为3
knn3 = KNeighborsClassifier(n_neighbors=3)
knn3_pred = knn3.fit(X_train, L_train).predict(X_test)
print(np.mean(knn3_pred == L_test)) #显示模型预测正确率
```

示例代码用KNN模型拟合了stock market数据集，并给出了$K=1$和$K=3$时的预测效果。结果显示KNN的预测效果并不好

下面是KNN对Caravan数据集的预测，这个数据集的特点是特征向量比较大(85个特征)：

```
#获取Caravan数据集
Caravan = load_data('Caravan')
purchase = Caravan.Purchase
print(purchase.value_counts()) #显示购买人数，返回值是pd.Series
print("Rate of purchaser: ", 348 / 5474)

#准备构建模型
feature_df = Caravan.drop(columns=['Purchase']) #获取特征值向量
scaler = StandardScaler(with_mean=True, #是否减去均值
                        with_std=True, #是否除以标准差
                        copy=True) #构建标准化方法，复制数据
scaler.fit(feature_df) #标准化数据
X_std = scaler.transform(feature_df) #获得标准化特征向量
feature_std = pd.DataFrame(
    X_std,
    columns=feature_df.columns
)
print(feature_std.std()) #检验特征向量的方差
```

```
(X_train, X_test, y_train, y_test) = train_test_split(
    feature_std,
    purchase,
    test_size=1000, #指定测试集大小
    random_state=0 #保证每次都采用相同分割方式
) #分割数据集
```

必须注意变量的**尺度**对KNN模型的构建影响很大，因为KNN模型通过衡量欧氏距离决定分类。不同的计量单位甚至会对预测产生影响！通过**标准化(standardize)** 解决这个问题，实验用到的标准化方法是 `StandardScaler`

> 此处的标准化是将变量改为一个均值为0，方差为1的分布。不同的程序接口使用的标准不一样，即有的除以n有的除以$n-1$

拟合$K=1$模型：

```
#拟合K=1的KNN模型
knn1 = KNeighborsClassifier(n_neighbors=1)
knn1_pred = knn1.fit(X_train, y_train).predict(X_test)
print("Rate of error: \n", np.mean(y_test != knn1_pred), #显示模型错误率
      "\nRate of purchase in test set: \n", np.mean(y_test != "No")) #显示测试集上, 用户的实际购买率
```

我们发现预测购买率的错误率为11%然而实际购买率为6%。这说明虽然我们的错误率看似很低，但是用一个**空置率(null rate)** 为6%的分类器达到的效果甚至比我们好(这个分类器认为所有人都购买，正确率为6%)

事实上，总体错误率往往不是让人感兴趣的。因为对某部分分类的预测涉及到决策的成本问题，我们往往希望提高对我们感兴趣分类的正确预测率，而不是整体的预测率以降低决策成本

显示混淆矩阵，以判别正确预测购买者的比率：

```
#显示混淆矩阵
table = confusion_table(knn1_pred, y_test)
print("Confusion table of K = 1: \n", table)
print("Correctly prediction of purchaser: ", 9/(53+9)) #显示正确预测购买者的比率
```

KNN中的参数K被称为**调优参数(tuning parameter)** 或 **超参数(hyperparameter)**，我们无法事先预知它的值，因此可以通过for循环来探测不同值对预测率的影响：

```
#用for循环探测合适的K值
for K in range(1, 8):
    knn = KNeighborsClassifier(n_neighbors=K)
    knn_pred = knn.fit(X_train, y_train).predict(X_test)
```

```
    C = confusion_table(knn_pred, y_test)
    temp1 = (
        'K={0:d}: # predicted to rent: {1:>2},' +
        ' # who did rent {2:d}, accuracy {3:.1%}'
    ) #格式化输出
    pred = C.loc['Yes'].sum() #所有标签为'Yes'的和
    did_rent= C.loc['Yes', 'Yes'] #行列标签都为'Yes'的值
    print(temp1.format(
        K,
        pred,
        did_rent,
        did_rent / pred
    ))
```

`sklearn`包使用逻辑斯蒂回归的岭回归版本来预测，我们添加参数C为一个很大的值使得这个算法收敛到逻辑斯蒂回归的解上

`skleran`包更注重分类而非推理，之前的`summary()`方法在此处不使用。下面构建逻辑斯蒂回归与KNN进行比较：

```
#拟合逻辑斯蒂回归来对比
logit = LogisticRegression(C=1e10, solver='liblinear') #设定参数
logit.fit(X_train, y_train)
logit_pred = logit.predict_proba(X_test)
logit_labels = np.where(logit_pred[:, 1] > 0.25, 'Yes', 'No')
table = confusion_table(logit_labels, y_test)
print("Confusion table of logistic regression: \n", table)
print("Correct prediction of logistic regression: ", 9/(20+9))
```

4.7.7 线性回归与泊松回归在Bikeshare数据集上的应用

构建线性回归模型：

```
#加载数据，并显示简单的信息
Bike = load_data('Bikeshare')
print("Shape of Bike Share: ", Bike.shape)
print("Columns: \n", Bike.columns)

#构建数据集
X = MS(['mnth',
        'hr',
        'workingday',
        'temp',
        'weathersit']).fit_transform(Bike)
Y = Bike['bikers']

#建立线性回归模型
```

```
M_lm = sm.OLS(Y, X).fit()
print(summarize(M_lm)) #没有提供显示的参数被当做了基线，系数被估计为0
```

注意基线的选取影响我们的判断，非基线特征的参数表示与基线进行的对比

采用另一种编码方式进行预测：

```
#构建新的数据集
hr_encode = contrast('hr', 'sum')
mnth_encode = contrast('mnth', 'sum')

#新的预测
X2 = MS([
    mnth_encode,
    hr_encode,
    'workingday',
    'temp',
    'weathersit'
]).fit_transform(Bike)
M2_lm = sm.OLS(Y, X2).fit()
S2 = summarize(M2_lm)
print("Different OLS: \n", S2)
```

这种方式与第一种的区别是，没有显示离散型特征值的最后一项(实际上，这项的值为其他所有项值和的负数)。这意味着这些离散型特征值的系数和为0，因此这些系数可以被解释为相对于均值水平的差(正数表示相对于均值增加，负数相反)

实际上，编码的选择对模型的正确与否无关紧要，影响的系数的解释，我们可以看看两种方法的区别，其中一种方法用到了`np.allclose()`方法：

```
np.sum(( M_lm.fittedvalues - M2_lm.fittedvalues)**2) #作差比较
np.allclose(M_lm.fittedvalues, M2_lm.fittedvalues)
```

我们可以绘制关于月份和小时数系数的图片，来对比不同离散型特征取值对结果的影响：

```
#重构月份系数
coef_month = S2[S2.index.str.contains('mnth')]['coef'] #获取系数-从S2中获取列名包含'mnth'的列表，获取'coef'
print("parameters of each month in Model 2: \n", coef_month)
months = Bike['mnth'].dtype.categories
coef_month = pd.concat([
    coef_month,
    pd.Series([-coef_month.sum()], index=['mnth[Dec]']) #添加12月
]) #重构月份系数，链接两个Series
print("parameters of each month in Model 2 with December: \n", coef_month)

#建立月份系数图表
```

```
fig_month, ax_month = subplots(figsize=(8, 8)) #创建子图
x_month = np.arange(coef_month.shape[0]) #关于月份系数的列表，用于绘图准备
ax_month.plot(x_month, coef_month, marker='o', ms=10) #在子图上绘制月份系数
ax_month.set_xticks(x_month) #设置x轴刻度为月份系数
ax_month.set_xticklabels([l[5] for l in coef_month.index], fontsize=20) #设
置x轴刻度标签
ax_month.set_xlabel('Month', fontsize=20) #设置x轴标签
ax_month.set_ylabel('Coefficient', fontsize=20)

#重建小时系数
coef_hr = S2[S2.index.str.contains('hr')]['coef']
coef_hr = coef_hr.reindex(['hr[{0}]'.format(h) for h in range(23)])
coef_hr = pd.concat([
    coef_hr,
    pd.Series([-coef_hr.sum()], index=['hr[23]'])
])

#绘制关于小时的图片
fig_hr, ax_hr = subplots(figsize=(8, 8))
x_hr = np.arange(coef_hr.shape[0])
ax_hr.plot(x_hr, coef_hr, marker='o', ms=10)
ax_hr.set_xticks(x_hr[::2])
ax_hr.set_xticklabels(range(24)[::2], fontsize=20)
ax_hr.set_xlabel('Hour', fontsize=20)
ax_hr.set_ylabel('Coefficient', fontsize=20)

plt.show()
```

下面是关于泊松回归的构建，同时创建了关于泊松回归的图表：

```
#绘制关于小时的图片
fig_hr, ax_hr = subplots(figsize=(8, 8))
x_hr = np.arange(coef_hr.shape[0])
ax_hr.plot(x_hr, coef_hr, marker='o', ms=10)
ax_hr.set_xticks(x_hr[::2])
ax_hr.set_xticklabels(range(24)[::2], fontsize=20)
ax_hr.set_xlabel('Hour', fontsize=20)
ax_hr.set_ylabel('Coefficient', fontsize=20)

#构建泊松回归
M_pois = sm.GLM(Y, X2, family=sm.families.Poisson()).fit()
S_pois = summarize(M_pois)

#重建泊松回归的时间系数和月份系数
coef_month = S_pois[S_pois.index.str.contains('mnth')]['coef']
coef_month = pd.concat([coef_month,
                        pd.Series([-coef_month.sum()],
                                  index=['mnth[Dec]'])])
coef_hr = S_pois[S_pois.index.str.contains('hr')]['coef']
```

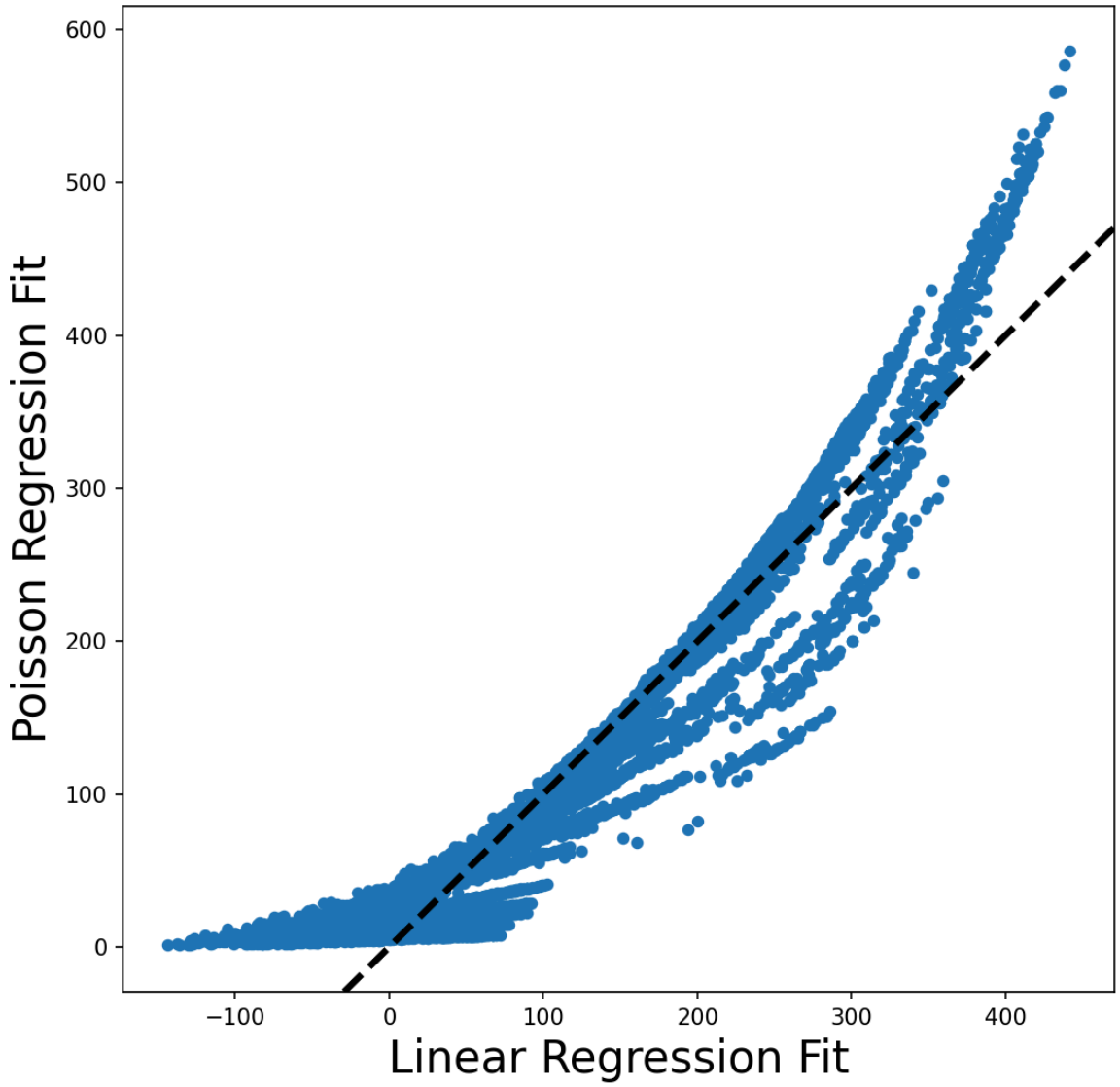
```
coef_hr = pd.concat([coef_hr,
                     pd.Series([-coef_hr.sum()],
                               index=['hr[23]'])])

#绘制泊松分布的时间和月份图
fig_pois , (ax_month , ax_hr) = subplots (1, 2, figsize =(16 ,8))
ax_month.plot(x_month, coef_month , marker='o', ms=10)
ax_month.set_xticks(x_month)
ax_month.set_xticklabels([l[5] for l in coef_month.index], fontsize =20)
ax_month.set_xlabel('Month', fontsize=20)
ax_month.set_ylabel('Coefficient', fontsize=20)
ax_hr.plot(x_hr, coef_hr, marker='o', ms=10)
ax_hr.set_xticklabels(range(24)[::2], fontsize=20)
ax_hr.set_xlabel('Hour', fontsize=20)
ax_hr.set_ylabel('Coefficient', fontsize=20)
```

绘制图片，比较线性回归和泊松回归的关系：

```
#绘制图片，比较泊松回归和线性回归关系
fig, ax = subplots(figsize=(8, 8))
ax.scatter(M2_lm.fittedvalues, M_pois.fittedvalues, s=20)
ax.set_xlabel('Linear Regression Fit', fontsize=20)
ax.set_ylabel('Poisson Regression Fit', fontsize=20)
ax.axline([0, 0], c='black', linewidth=3, linestyle='--', slope=1)

plt.show()
```



这幅图片说明线性回归和泊松回归结果之间存在正相关性。然而，对于两端的值(很高或很低的骑行率)，泊松分布的预测结果往往比线性回归要大(极端)

#CS
#ML