Note for Regularization Paths for Generalized Linear Models

Tomoyo

November 10, 2019

Instruction: Derivation for two-class classification.

1 Linear regression:

- 1.1 basis
- 1.2 derivation

1.1 definition

$$probability : Pr(y = 1|x) = \frac{1}{1 + e^{-(\beta_0 + \sum\limits_{i=1}^{m} \beta_i x_i)}}$$

$$likelihood : L = \prod_{i=0}^{n} p(x_i)^{y_i} * [1 - p(x_i)]^{1-y_i}$$

$$P_\alpha(\beta) = \sum_{j=1}^{m} [\frac{1}{2}(1 - \alpha)\beta_j^2 + \alpha|\beta_j|]$$

1.2 derivation

Based on definition, penalized log-likelihood can be presented as follow.

$$L = \sum_{i}^{n} \{y_i log(p(x_i)) + (1 - y_i) log(1 - p(x_i))\} - \sum_{j=1}^{m} [\frac{1}{2}(1 - \alpha)\beta_j^2 + \alpha|\beta_j|]$$

Note that the first part can be written as follow.

$$
\begin{aligned}
L &= \sum_{i=1}^{n}[log(1-p(x_i)) + y_i log\frac{p(x_i)}{1-p(x_i)}] \\
&= \sum_{i=1}^{n}[y_i(\beta_0 + \sum_{i=1}^{m}\beta_i x_i) + log(\frac{e^{-(\beta_0+\sum\limits_{i=1}^{m}\beta_i x_i)}}{1+e^{-(\beta_0+\sum\limits_{i=1}^{m}\beta_i x_i)}})] \\
&= \sum_{i=1}^{n}[y_i(\beta_0 + \sum_{i=1}^{m}\beta_i x_i) + log(1+e^{(\beta_0+\sum\limits_{i=1}^{m}\beta_i x_i)})]
\end{aligned}
$$

Using quadratic approximation(taylor expansion), The log-likelihood can be written as follow.

$$
\begin{aligned}
&l_Q = -\frac{1}{2}\sum_{i=1}^{N} w_i(z_i - \beta_0 - \sum_{i=1}^{m}\beta_i x_i)^2 + C(\hat{\beta}_0, \hat{\beta})^2 \\
&z_i = \hat{\beta}_0 + \sum_{i=1}^{m}\hat{\beta}_i x_i + \frac{y_i - \hat{p}(x_i)}{\hat{p}(x_i)(1-\hat{p}(x_i))} \\
&w_i = \hat{p}(x_i)(1-\hat{p}(x_i))
\end{aligned}
$$

Using coordinate descent to solve the penalized weighted least-squares problem, which is

$$
\min_{\beta_0,\beta}\{-l_q(\beta_0,\beta) + \lambda P_\beta)\}
$$

derive derivative to solve this problem, for $\beta_j, j \neq 0$:

$$
\begin{aligned}
part_1 = \frac{\partial L}{\partial \beta_j} &= \sum_{i=1}^{N} w_i[z_i - (\beta_0 + \sum_{j=1}^{m}\beta_j x_j)]x_{ij} \\
&= \sum_{i=1}^{N} w_i[z_i - (\beta_0 + \sum_{k\neq j}^{m}\beta_k x_k)]x_{ij} - \sum_{i=1}^{N} w_i x_{ij}^2 \beta_j \\
&= \sum_{i=1}^{N} w_i[\hat{\beta}_j x_{ij} + \frac{y_i - \hat{p}(x_i)}{\hat{p}(x_i)(1-\hat{p}(x_i))}]x_{ij} - \sum_{i=1}^{N} w_i x_{ij}^2 \beta_j \\
&= \sum_{i=1}^{N}[w_i x_{ij}^2 \hat{\beta}_j + (y_i - \hat{p}(x_i))x_{ij}] - \sum_{i=1}^{N} w_i x_{ij}^2 \beta_j \\
&\text{denotation}, P = \sum_{i=1}^{N} w_i x_{ij}^2, Q = \sum_{i=1}^{N} x_{ij}(y_i - \hat{p}(x_i)) \\
&= P * \hat{\beta}_j + Q - P * \beta_j
\end{aligned}
$$

Now,derive the penalty part.

$$
\frac{\partial P_\alpha(\beta)}{\partial \beta_j} = \sum_{j=1}^{m}[(1-\alpha)\beta_j + \alpha\partial|\beta_j|]
$$

Using Subderivative, the equation can be written as follow.

$$
\begin{aligned}
part_2 = \frac{\partial P_\alpha(\beta)}{\partial \beta_j} &= [(1-\alpha)\beta_j + \alpha\partial|\beta_j|] \\
&= \begin{cases} (1-\alpha)\beta_j + \alpha, \quad \beta_j > 0 \\ [(1-\alpha)\beta_j - \alpha, (1-\alpha)\beta_j + \alpha], \quad \beta_j = 0 \\ (1-\alpha)\beta_j - \alpha, \quad \beta_j < 0 \end{cases}
\end{aligned}
$$

to find the minimum point, $-part_1 + \lambda part_2 = 0$, which is

$$
0 = \begin{cases} \lambda[(1-\alpha)\beta_j + \alpha] - (P\hat{\beta}_j + Q - P\beta_j), \quad \beta_j > 0 \\ \{\lambda[(1-\alpha)\beta_j - \alpha] - (P\hat{\beta}_j + Q - P\beta_j), \lambda[(1-\alpha)\beta_j + \alpha] - (P\beta_j\} + \hat{Q} - P\beta_j), \quad \beta_j = 0 \\ \lambda[(1-\alpha)\beta_j - \alpha] - (P\hat{\beta}_j + Q - P\beta_j), \quad \beta_j < 0 \end{cases}
$$

Take $\beta_j > 0$ as an example.

$$
\begin{aligned}
&\lambda\beta_j - (P\hat{\beta}_j + Q - P\beta_j) = 0 \\
&\beta_j = \frac{Q - \lambda\alpha + P\hat{\beta}_j}{P + \lambda(1-\alpha)} \\
&condition : Q + P\hat{\beta}_j > \lambda\alpha
\end{aligned}
$$

Implmentation as follow

```python
from joblib import Parallel, delayed
import gc
import numpy as np
import pandas as pd


class Lasso_CR(object):
    def __init__(self, **kwargs):
        self.beta = 0
        self.intercept = 0
        self.lambda_ = 0
        self.max_iter = 500
        self.lambda_ = kwargs.get('lambda_')

    def standardlize(self, data):
        return (data - data.mean(axis = 0))/data.std(axis = 0)

    def soft_thresholding(self, beta_old, P, Q, threshold):
        if P * beta_old + Q > threshold:
            beta_new = beta_old + (Q - threshold)/P
        elif P * beta_old + Q < - threshold:
```

```
            beta_new = beta_old + (Q + threshold)/P
        else:
            beta_new = 0
        return beta_new
    def coordinate_descent(self, x, y, beta, intercept, lambda_):
        def process_helper(x):
            x = np.where(x > 1 - 10**-5, 1 - 10**-5, x)
            x = np.where(x < 10**-5, 10**-5, x)
            return x
        for j in range(len(beta)):
            linear = intercept + np.sum(beta * x, axis = 1)
            prob = 1/(1 + np.exp(-linear))
            prob = process_helper(prob)
            w = prob * (1 - prob)
            P = np.mean(w * (x[:, j] ** 2))
            Q = np.mean((y - prob) * x[:, j])
            beta[j] = self.soft_thresholding(beta[j], P, Q, lambda_)

        linear = intercept + np.sum(beta * x, axis = 1)
        prob = process_helper(1/(1 + np.exp(-linear)))
        w_ = prob * (1 - prob)
        dir_ = y - prob
        intercept_new = intercept + np.mean(dir_)/np.mean(w)
        return beta, intercept_new

    def fit(self, X, y):
        X_std = self.standardlize(X)
        max_iter = 500
        self.beta = np.ones(X.shape[1])/X_std.shape[1]
        self.intercept = 0
        for iteration in range(max_iter):
            self.beta, self.intercept = self.coordinate_descent(X_std, y,
            self.beta,self.intercept, self.lambda_)
```