Logistic Supplement

Tomoyo

November 21, 2019

Instruction: Derivation for two-class classification.

1 Linear regression:

- 1.1 basis
- 1.2 derivation

1.1 definition

$$probability : P(x_i) = Pr(y = 1|x_i) = \frac{1}{1 + e^{-(\beta_0 + \sum\limits_{j=1}^{m} \beta_j x_{ji})}}$$

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

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

$$NLL = -\sum_{i=1}^{n} [y_i log(p(x_i) + (1-y_i) log(1 - p(x_i))]$$

1.2 derivation

NLL can be written as follow.

$$L = -\sum_{i=1}^{n} [y_i(\beta_0 + \sum_{j=1}^{m} \beta_j x_{ij}) - log(1 + e^{\beta_0 + \sum\limits_{j=1}^{m} \beta_j x_{ij}})]$$

calculation of grad, hessian

$$g_j = \frac{\partial L}{\partial \beta_j} = -\sum_{i=1}^{n} [y_i - p(x_i)] x_{ij}$$

$$h_j = \frac{\partial g}{\partial \beta_j} = \sum_{i=1}^{n} x_{ij}^2 p(x_i)(1 - p(x_i))$$

2 gradient descent

Using g as direction, optimize function based on g, when $beta_j$ in k iteration:

$$beta_j^{k+1} = beta_j^k - \eta g_j$$

for intercept:

$$intercept^{(k+1)} = intercept^{(k)} - \eta[-(y_i - p_i)]$$

3 Newton's method

Taylor's expansion for L in $\beta_j^{(k)}$

$$\begin{aligned}
L(\beta_j) &= L(\beta_j^{(k)}) + (\beta_j - \beta_j^{(k)})g_j + \frac{1}{2}(\beta_j - \beta_j^{(k)})^2 h_j + o(n^3) \\
&= \beta_j g_j + \frac{1}{2}\beta_j^2 h_j - \beta_j \beta_j^{(k)} h_j + \frac{(\beta_j^{(k)})^2}{2} h_j + L - \beta_j^{(k)} g_j \\
&= (\frac{1}{2}h_j)\beta_j^2 + (g_j - \beta_j^{(k)} hj)\beta_j + L - \beta_j^{(k)} g_j + \frac{(\beta_j^{(k)})^2}{2} h_j \\
& donation : M = (\frac{1}{2}h_j, N = g_j - \beta_j^{(k)} hj), C = L - \beta_j^{(k)} g_j + \frac{(\beta_j^{(k)})^2}{2} h_j \\
&= M\beta_j^2 + N\beta_j + C, \text{so that} \\
L &= M\beta_j^2 + N\beta_j + C
\end{aligned}$$

The solution is:

$$\beta_j = -\frac{N}{2M} = -\frac{g_j - \beta_j^{(k)} hj}{h_j} = \beta_j^{(k)} - \frac{g_j}{h_j}$$

4 IRLS(Iterative Reweighted Least Squares)

The Newton's solution can be rewritten as follow.

$$\begin{aligned}
\beta_j &= \frac{h_j\beta_j^{(k)} - g_j}{h_j} \\
&= \frac{\sum_{i=1}^{n} x_{ij}^2 p(x_i)(1-p(x_i))\beta_j^{(k)} + \sum_{i=1}^{n}[y_i - p(x_i)]x_{ij}}{h_j} \\
&= \frac{\sum_{i=1}^{n}\{x_{ij}^2 p(x_i)(1-p(x_i))\beta_j^{(k)} + [y_i - p(x_i)]x_{ij}\}}{h_j} \\
&= \frac{\sum_{i=1}^{n} x_{ij} p(x_i)(1-p(x_i))\{x_{ij}\beta_j^{(k)} + \frac{[y_i - p(x_i)]}{p(x_i)(1-p(x_i))}\}}{h_j} \\
&donation : w_i = p(x_i)(1-p(x_i)), z_i = \frac{[y_i - p(x_i)]}{p(x_i)(1-p(x_i))} + x_{ij}\beta_j^{(k)} \\
&= \frac{\sum_{i=1}^{n} x_{ij} w_i z_i}{h_j}
\end{aligned}$$

the solution is equal to

$$\sum_{i=1}^{n}[x_{ij}^2 p(x_i)(1-p(x_i))]\beta_j = \sum_{i=1}^{n} x_{ij} w_i z_i$$

$$\sum_{i=1}^{n} x_{ij} w_i (x_{ij}\beta_j - z_i) = 0$$

equals to the minimize the problem:

$$\arg\min_{\beta_j} \sum_{i=1}^{n} w_i (x_{ij}\beta_j - z_i)^2$$

equals to:

$$\arg\min_{\beta_j} \sum_{i=1}^{n} w_i (\sum_{j=1}^{m} x_{ij}\beta_j + \beta_0 - z_i)^2$$

$$\text{where} \quad z_i = \frac{[y_i - p(x_i)]}{p(x_i)(1-p(x_i))} + \sum_{k=1}^{m} x_{ik}\beta_k^{(k)}$$

So the minimize the NLL is equal to

$$\arg\min_{\beta} \sum_{i=1}^{n} w_i (\sum_{j=1}^{m} x_{ij}\beta_j + \beta_0 - z_i)^2$$

5 implementation

```
from sklearn.datasets import load_breast_cancer
from joblib import Parallel, delayed
import gc
import numpy as np
import pandas as pd

class Logistic_Regression(object):
    def __init__(self, eta = 0.005, tol = 10e-4):
        self.eta = eta
        self.beta = None
        self.intercept = 0
        self.tol = tol
    def process_helper(self, x):
        x = np.where(x > 1 - 10**-4, 1 - 10**-4, x)
        x = np.where(x < 10**-4, 10**-4, x)
        return x
    def standard_function(self, data):
        return (data - data.mean(axis = 0))/data.std(axis = 0)
    def gradient_descent(self, X, y, beta, intercept, parallel = False):
        linear = intercept + np.sum(beta * X, axis = 1)
        prob = 1/(1 + np.exp(-linear))
        new_beta = np.zeros(beta.shape)
        if parallel:
            def cal_grad(y, prob, x, beta, eta):
                return beta - eta * np.sum(-(y - prob) * x)
            new_beta = Parallel(n_jobs = -1, verbose = 0)\
            (delayed(cal_grad)(target, prob, X[:,j], beta[j], eta) for j in range(data_
            new_beta = np.array(new_beta)
        else:
            for j in range(X.shape[1]):
                grad = -np.sum((y - prob) * X[:, j])
                new_beta[j] = beta[j] - self.eta * grad
        new_intercept = intercept - self.eta * np.sum(-(y - prob))
        return new_beta, new_intercept
    def newton_descent(self, X, y, beta, intercept):
        linear = intercept + np.sum(beta * X, axis = 1)
        prob = 1/(1 + np.exp(-linear))
        new_beta = np.zeros(beta.shape)
        prob = self.process_helper(prob)
        for j in range(X.shape[1]):
            grad = -np.sum((y - prob) * X[:, j])
            hess = np.sum(prob * (1 - prob) * X[:, j] ** 2)
            new_beta[j] = beta[j] - self.eta * grad/hess
        grad_intercept = -np.sum(y - prob)
```

```
        hess_intercept = np.sum(prob * (1 - prob))
        new_intercept = intercept - self.eta * grad_intercept/hess_intercept
        return new_beta, new_intercept
    def IRLS(self, X, y, beta, intercept):
        new_beta = np.zeros(beta.shape)
        linear = intercept + np.sum(beta * X, axis = 1)
        prob = 1/(1 + np.exp(-linear))
        prob = self.process_helper(prob)
        w = prob * (1 - prob)
        for j in range(X.shape[1]):
            h = np.sum(w * X[:, j] ** 2)
            z = (y - prob)/w + beta[j] * X[:, j]
            beta[j] = np.sum(X[:, j] * w * z)/h
        new_intercept = np.sum(w * (intercept + (y - prob)/w))/np.sum(w)
        return beta, new_intercept
    def fit(self, X, y, method = 'newton'):
        self.beta = np.zeros(X.shape[1])
        for i in range(1000):
            converge = 1
            beta_p, intercept_p = self.beta, self.intercept
            if method == 'newton':
                self.beta, self.intercept = self.newton_descent(data_std, target, self.
            elif method == 'gradient':
                self.beta, self.intercept = self.gradient_descent(data_std, target, sel
            elif method == 'IRLS':
                self.beta, self.intercept = self.IRLS(data_std, target, self.beta, self
            else:
                raise Exception("only support ['gradient', 'newton', 'IRLS']")
            converge = np.sum((beta - beta_p) ** 2) + (intercept - intercept_p) ** 2
            if converge < self.tol:
                break
    def predict(self, X):
        linear = self.intercept + np.sum(self.beta * X, axis = 1)
        prob = 1/(1 + np.exp(-linear))
        return np.where(prob > 0.5, 1, 0)
```