Lab2 实验报告

PB20111689 蓝俊玮　　实验环境为 Goggle Colab

SVM1

我的 `SVM1` 的目标优化函数是 `hinge` 损失函数加上正则化项。即优化的目标函数是

$$\min_{\boldsymbol{w},b} = \frac{1}{2}C_1||\boldsymbol{w}||^2 + C_2\sum_{i=1}^{n}\max(0, 1-y_i(\boldsymbol{w}^T\boldsymbol{x}_i+b))$$

因为 2 - 范数和 max 函数都是凸函数，因此这个目标函数是凸函数，是可以进行梯度下降法进行求解的。

在这里，为了方便训练，我们像逻辑斯蒂回归那样，令

$$\hat{\boldsymbol{w}} = (\boldsymbol{w}; b) \quad \hat{\boldsymbol{x}} = (\boldsymbol{x}; 1)$$

所以只需要对 $\hat{\boldsymbol{w}}$ 进行优化求解即可。

则对其进行求导，可以得到：

$$\frac{\partial L(\boldsymbol{x}_i)}{\partial \hat{\boldsymbol{w}}} = C_1\boldsymbol{w} + C_2\sum_{i=1}^{n}\frac{\partial \max(0, 1-y_i(\hat{\boldsymbol{w}}^T\boldsymbol{x}_i))}{\partial \hat{\boldsymbol{w}}}$$

对其进行分类讨论，当 $1-y_i(\hat{\boldsymbol{w}}^T\boldsymbol{x}_i) \le 0$ 的时候，$\max(0, 1-y_i(\hat{\boldsymbol{w}}^T\boldsymbol{x}_i)) = 0$，则此时的 $\frac{\partial \max(0,1-y_i(\hat{\boldsymbol{w}}^T\boldsymbol{x}_i))}{\partial \hat{\boldsymbol{w}}} = 0$；当 $1-y_i(\hat{\boldsymbol{w}}^T\boldsymbol{x}_i) > 0$ 的时候，$\frac{\partial \max(0,1-y_i(\hat{\boldsymbol{w}}^T\boldsymbol{x}_i))}{\partial \hat{\boldsymbol{w}}} = \frac{\partial (1-y_i(\hat{\boldsymbol{w}}^T\boldsymbol{x}_i))}{\partial \hat{\boldsymbol{w}}} = -y_i\boldsymbol{x}_i$，因此最后可以得到：

$$\frac{\partial L(\boldsymbol{x}_i)}{\partial \hat{\boldsymbol{w}}} = C_1\boldsymbol{w} + C_2\sum_{i=1}^{n}-y_i\boldsymbol{x}_i$$

因此可以据此得到 `SVM1` 模型的梯度下降法：

```
class SVM1:
    def fit(self, X, y, C1=1e-2, C2=1e-5, lr=0.01, tol=1e-3, max_iter=1e5):
        early_stop = False
        for iteration in range(int(max_iter)):
            z = y * np.dot(X, self.weights)
            hinge = np.maximum(0, 1 - z)

            # 求解 hinge 函数的梯度
            hinge_grad = -y * X * C2
            hinge_grad[np.where(hinge == 0)[0]] = 0
            hinge_grad = np.sum(hinge_grad, axis=0)

            grad = hinge_grad.reshape((-1, 1)) + C1 * self.weights

            hinge_loss = np.sum(hinge) * C2
            loss = C1 * 0.5 * np.linalg.norm(self.weights) + hinge_loss
            self.train_loss.append(loss)

            if (np.absolute(grad) < tol).all():
                early_stop = True
```

```
                print("early stop at iteration {}.".format(iteration))
                break
            self.weights = self.weights - lr * grad
        if not early_stop:
            print("stop at iteration {}.".format(int(max_iter)))
```

SVM2

完整的 SMO 算法包含许多优化，旨在大型数据集上加速算法并确保算法即使在退化的条件下也能收敛。而我的 `SVM2` 是基于《统计学习方法》中描述的 SMO 算法实现的，并且为了能够加快 SMO 算法的运行速度，我同时对该算法进行了一定程度的简化和优化。

SMO 算法的核心其实就是三个：选择两个 α 参数，优化这两个 α 参数并且更新计算阈值 b。

第一个变量的选择需要先遍历所有满足条件 $0 < \alpha_i < C$ 的样本点，即在间隔边界上的支持向量点，检查它们是否满足 KKT 条件。如果这些样本点都满足 KKT 条件，那么遍历整个训练集，检验它们是否都满足 KKT 条件。而第二个变量的选择希望能够让 $|E_1 - E_2|$ 最大，因此需要根据 α_1 的值来选择。如果 $E_1 > 0$ 则选择最小的 E_i 作为 E_2；如果 $E_1 < 0$ 则选择最大的 E_i 作为 E_2。选择完之后，用

$$
\begin{gathered}
\alpha_2^{new,unc} = \alpha_2^{old} + \frac{y_2(E_1 - E_2)}{\eta} \\
\eta = K_{11} + K_{22} - 2K_{12} \\
\alpha_2^{new} = \begin{pmatrix} H, & \alpha_2^{new,unc} > H \\ \alpha_2^{new,unc}, & L \le \alpha_2^{new,unc} \le H \\ L, & \alpha_2^{new,unc} < L \end{pmatrix} \\
\alpha_1^{new} = \alpha_1^{old} + y_1 y_2 (\alpha_2^{old} - \alpha_1^{old})
\end{gathered}
$$

更新 α_1 和 α_2。接着再更新 b 的值：

$$
\begin{gathered}
b_1^{new} = -E_1 - y_1 K_{11}(\alpha_1^{new} - \alpha_1^{old}) - y_2 K_{21}(\alpha_2^{new} - \alpha_2^{old}) + b^{old} \\
b_2^{new} = -E_2 - y_1 K_{12}(\alpha_1^{new} - \alpha_1^{old}) - y_2 K_{22}(\alpha_2^{new} - \alpha_2^{old}) + b^{old} \\
b = \begin{pmatrix} b_1 & 0 < \alpha_1 < C \\ b_2 & 0 < \alpha_2 < C \\ \frac{b_1 + b_2}{2} & otherwise \end{pmatrix}
\end{gathered}
$$

然后重新计算 E_1 和 E_2 进行下一轮循环。

如上述所述，SMO 算法选择两个 α 参数，α_i 和 α_j，并联合优化这两个 α 的目标值。 最后调整 b 参数基于新的 α。 重复这个过程，直到 α 收敛。大部分完整的 SMO 算法都致力于启发式方法来选择哪些 α_i 和 α_j 优化以尽可能地最大化目标函数。而在本次实验中，对于 10000 数据量的数据集来说，这种启发式的方式速度太慢，因此我采用了更简单的启发式方法。我们简单地遍历所有 α_i，如果 α_i 不满足 KKT 条件，我们从剩余的 α 中**随机选择** α_j 并尝试联合优化 α_i 和 α_j 。如果在对所有 α_i 进行遍历后发现它们都满足 KKT 条件，那么算法终止。通过采用这种简化，算法不会用更长时间保证收敛到全局最优。

同时对于这个二分类的 SVM，可以将其中点乘的结果存储起来，成为一个权重 `alpha_y_k`，这样可以极大的加快算法的运行。

所以可以得到 `SVM2` 如下：

```
# you can do anything necessary about the model
class SVM2:
    def __init__(self, dim, num):
        """
```

```
        You can add some other parameters, which I think is not necessary
        """
        self.dim = dim
        self.num = num
        self.alpha = np.ones((self.num, 1))
        self.C = 1.0
        self.b = 0
        self.alpha_y_k = np.zeros(self.dim)

    def fit(self, X, y, max_iter=1e5):
        """
        Fit the coefficients via your methods
        """
        self.X = X
        self.y = y
        self.weight()

        for iteration in range(int(max_iter)):
            i, j = self.selectAlpha()
            if i == -1 and j == -1:
                print("early stop at iteration {}".format(iteration))
                break
            E_i = self.e(i)
            E_j = self.e(j)
            X_i = self.X[i]
            X_j = self.X[j]
            alpha_i_old, alpha_j_old = self.alpha[i], self.alpha[j]

            L, H = self.LH(i, j)
            eta = self.kernel(X_i, X_i) + self.kernel(X_j, X_j) - 2 *
self.kernel(X_i, X_j)
            if eta <= 0:
                continue

            alpha_j = alpha_j_old + self.y[j] * (E_i - E_j) / eta

            if alpha_j > H:
                alpha_j = H
            elif alpha_j < L:
                alpha_j = L

            alpha_i = alpha_i_old + self.y[i] * self.y[j] * (alpha_j_old -
alpha_j)

            b_1 = self.b - E_i - self.y[i] * (alpha_i - alpha_i_old) *
self.kernel(X_i, X_i) - self.y[j] * (alpha_j - alpha_j_old) * self.kernel(X_j,
X_i)
            b_2 = self.b - E_j - self.y[i] * (alpha_i - alpha_i_old) *
self.kernel(X_i, X_j) - self.y[j] * (alpha_j - alpha_j_old) * self.kernel(X_j,
X_j)
            if 0 < alpha_i < self.C:
                self.b = b_1
            elif 0 < alpha_j < self.C:
                self.b = b_2
            else:
                self.b = (b_1 + b_2) / 2

            self.alpha[i] = alpha_i
```

```
            self.alpha[j] = alpha_j
            self.weight()
        return

    def predict(self, X):
        """
        Use the trained model to generate prediction probabilities on a new
        collection of data points.
        """
        pred = X @ self.alpha_y_k
        pred[pred > 0] = 1
        pred[pred <= 0] = -1
        return pred

    def selectAlpha(self):
        alpha_index = [i for i in range(self.num) if 0 < self.alpha[i] < self.C]
        index = [i for i in range(self.num) if i not in alpha_index]
        alpha_index.extend(index)

        for i in alpha_index:
            if self.kkt(i):
                continue
            j = np.random.randint(self.num)
            while i == j:
                print("re-select")
                j = np.random.randint(self.num)
            return i, j
        return -1, -1

    def kkt(self, i):
        product = self.y[i] * self.g(i)
        if self.alpha[i] == 0:
            return product >= 1
        elif 0 < self.alpha[i] < self.C:
            return product == 1
        else:
            return product <= 1

    def LH(self, i, j):
        if self.y[i] != self.y[j]:
            L = max(0, self.alpha[j] - self.alpha[i])
            H = min(self.C, self.C + self.alpha[j] - self.alpha[i])
        else:
            L = max(0, self.alpha[i] + self.alpha[j] - self.C)
            H = min(self.C, self.alpha[i] + self.alpha[j])
        return L, H

    def e(self, i):
        return self.g(i) - self.y[i]

    def g(self, i):
        val = self.X[i] @ self.alpha_y_k
        return val

    def kernel(self, x1, x2):
        return np.dot(x1, x2)

    def weight(self):
```

```
        val = self.b
        for i in range(self.num):
            val = val + self.alpha[i] * self.y[i] * self.X[i].T
        self.alpha_y_k = val
        return
```

SVM3

SVM3 在 SVM2 的基础上做了进一步改动，即无需考虑 `b` 这个参数，成为 Fixed-b SVM 模型。因此，我们只需要考虑一个拉格朗日因子 α ，线性 SVM 的 Fixed-b SMO 在概念上类似于感知器松弛规则，只要出现错误，就会调整感知器的输出，以使输出恰好位于边缘。所以，在本次实验中发现，通过使用 Fixed-b SMO 算法，可以大大提高准确率。

所以可以得到 `SVM3` 如下：

```
class SVM3:
    def __init__(self, dim, num):
        """
        You can add some other parameters, which I think is not necessary
        """
        self.dim = dim
        self.num = num
        self.alpha = np.ones((self.num, 1))
        self.C = 1.0
        self.alpha_y_k = np.zeros(self.dim)

    def fit(self, X, y, max_iter=1e5):
        """
        Fit the coefficients via your methods
        """
        self.X = X
        self.y = y
        self.weight()

        for iteration in range(int(max_iter)):
            i, j = self.selectAlpha()
            if i == -1 and j == -1:
                print("early stop at iteration {}".format(iteration))
                break
            E_i = self.e(i)
            E_j = self.e(j)
            X_i = self.X[i]
            X_j = self.X[j]
            alpha_i_old, alpha_j_old = self.alpha[i], self.alpha[j]

            L, H = self.LH(i, j)
            eta = self.kernel(X_i, X_i) + self.kernel(X_j, X_j) - 2 *
self.kernel(X_i, X_j)
            if eta <= 0:
                continue

            alpha_j = alpha_j_old + self.y[j] * (E_i - E_j) / eta

            if alpha_j > H:
                alpha_j = H
            elif alpha_j < L:
                alpha_j = L
```

```
            alpha_i = alpha_i_old + self.y[i] * self.y[j] * (alpha_j_old -
alpha_j)

            self.alpha[i] = alpha_i
            self.alpha[j] = alpha_j
            self.weight()
        return

    def predict(self, X):
        """
        Use the trained model to generate prediction probabilities on a new
        collection of data points.
        """
        pred = X @ self.alpha_y_k
        pred[pred > 0] = 1
        pred[pred <= 0] = -1
        return pred

    def selectAlpha(self):
        alpha_index = [i for i in range(self.num) if 0 < self.alpha[i] < self.C]
        index = [i for i in range(self.num) if i not in alpha_index]
        alpha_index.extend(index)

        for i in alpha_index:
            if self.kkt(i):
                continue
            j = np.random.randint(self.num)
            while i == j:
                print("re-select")
                j = np.random.randint(self.num)
            return i, j
        return -1, -1

    def kkt(self, i):
        product = self.y[i] * self.g(i)
        if self.alpha[i] == 0:
            return product >= 1
        elif 0 < self.alpha[i] < self.C:
            return product == 1
        else:
            return product <= 1

    def LH(self, i, j):
        if self.y[i] != self.y[j]:
            L = max(0, self.alpha[j] - self.alpha[i])
            H = min(self.C, self.C + self.alpha[j] - self.alpha[i])
        else:
            L = max(0, self.alpha[i] + self.alpha[j] - self.C)
            H = min(self.C, self.alpha[i] + self.alpha[j])
        return L, H

    def e(self, i):
        return self.g(i) - self.y[i]

    def g(self, i):
        val = self.X[i] @ self.alpha_y_k
        return val
```

```
    def kernel(self, x1, x2):
        return np.dot(x1, x2)

    def weight(self):
        for i in range(self.num):
            self.alpha_y_k = self.alpha_y_k + self.alpha[i] * self.y[i] *
self.X[i].T
        return
```

模型结果与比较

所有模型都满足 `dim = 20, num = 10000`

SVM1	训练时间	准确率
C1 = 0.01, C2 = 1/num, lr = 0.01, seed = 3407	7.08s	0.902
C1 = 1, C2 = 1/num, lr = 0.01, seed = 3407	2.24s	0.856
C1 = 0.01, C2 = 1e-3, lr = 0.01, seed = 3407	18.76s	0.906
C1 = 0.01, C2 = 1/num, lr = 0.1, seed = 3407	18.76s	0.906
C1 = 0.01, C2 = 1/num, lr = 0.01, seed = 1234	6.06s	0.907
C1 = 0.01, C2 = 1/num, lr = 0.01, seed = 6666	6.01s	0.902
C1 = 0.01, C2 = 1/num, lr = 0.01, seed = 2333	6.92s	0.900

SVM2	训练时间	准确率
seed = 3407	9.14s	0.401
seed = 1234	9.19s	0.448
seed = 6666	9.16s	0.607
seed = 2333	9.21s	0.605

SVM3	训练时间	准确率
seed = 3407	9.39s	0.944
seed = 1234	9.31s	0.949
seed = 6666	9.29s	0.957
seed = 2333	10.64s	0.954

sklearn	准确率
seed = 3407	0.954
seed = 1234	0.959
seed = 6666	0.962
seed = 2333	0.960

可以从上述结果看出，在这里对 `SVM1` 调参对其总体的影响不是很大，与第一次实验中模型对参数的敏感性不同。

运行时间之间的比较实际上需要根据 `tol` 的值而定。因为 `tol` 可以让 `SVM1` 的梯度下降法提前终止计算。但是从迭代循环次数来看，SMO 算法的迭代次数是远小于梯度下降法的。但是原始 SMO 算法的计算量很大，它需要在每次循环中做多次 $O(n)$ 的操作，因此我采用了简化版的 SMO 算法，目的就是减小计算的时间。

最终预测准确率的模型性能是 `SVM3` > `SVM1` > `SVM2`。其中 `SVM3` 采用 Fixed-b SMO 算法和 `sklearn` 库中的准确率很接近，而 `SVM1` 采用梯度下降法可以得到一个较好的结果，但 `SVM2` 的效果比较差，原因在于这个 `b` 值无法较好的收敛到一个正确的值。