机器学习实验2报告

PB21000039 陈骆鑫

实验内容

使用两种不同的方法实现 SVM（支持向量机）算法，并对训练时间、准确率等参数进行比较。

实验原理

支持向量机是一种二分类算法。它的主要思路是找到一个超平面 $\boldsymbol{w}^T\boldsymbol{x}+b=0$，将不同类的样本分开，即满足：（样本量为 N）

$$y_i(\boldsymbol{w}^T\boldsymbol{x}_i+b)\geq 1, 1\leq i\leq N$$

SVM 使用的最优化目标是“间隔”，即到样本点到平面间的最近距离。在使用前面的约束后，可以使用最大化目标 $2/||\boldsymbol{w}||$，这与最小化 $\frac{1}{2}||\boldsymbol{w}||^2$ 等价。

使用拉格朗日乘子法，可以得到该问题的对偶问题：

$$\max_{\boldsymbol{\alpha}}\sum_{i=1}^{N}\alpha_i-\frac{1}{2}\sum_{i=1}^{N}\sum_{j=1}^{N}\alpha_i\alpha_j y_i y_j \boldsymbol{x}_i^T\boldsymbol{x}_j$$

$$\text{s.t.}\sum_{i=1}^{N}\alpha_i y_i=0, \alpha_i\geq 0$$

软间隔

前面的算法要求所有样本严格被划分超平面分开，而实际情况下，由于噪声等原因，这样的策略不一定能得到最优解——数据甚至不一定线性可分。针对这样的问题，可以使用“软间隔”的方法缓解。软间隔不要求，而使用惩罚函数惩罚不正确分类的样本。具体而言，最优化任务改写如下：

$$\min_{\boldsymbol{w},b}\frac{1}{2}||\boldsymbol{w}||^2+C\sum_{i=1}^{N}l(y_i(\boldsymbol{w}^T\boldsymbol{x}_i+b))$$

其中 l 为选择的损失函数，C 为选择的惩罚参数。若使用 hinge 损失函数 $l_{hinge}(z)=\max(0,1-z)$，改写后的对偶问题如下：

$$\max_{\boldsymbol{\alpha}}\sum_{i=1}^{N}\alpha_i-\frac{1}{2}\sum_{i=1}^{N}\sum_{j=1}^{N}\alpha_i\alpha_j y_i y_j \boldsymbol{x}_i^T\boldsymbol{x}_j$$

$$\text{s.t.}\sum_{i=1}^{N}\alpha_i y_i=0, 0\leq\alpha_i\leq C$$

从对偶问题的解得到原问题的解

对得到的最优化参数 $\boldsymbol{\alpha}$，计算 $\boldsymbol{w}$：

$$\boldsymbol{w}=\sum_{i=1}^{N}\alpha_i y_i \boldsymbol{x}_i$$

令 j 为任一个 $\alpha_i>0$ 的下标，计算 b:

$$b=y_j-\sum_{i=1}^{N}y_i\alpha_i K_{ij}$$

，则分离超平面即为 $\boldsymbol{w}^T\boldsymbol{x}+b=0$，决策函数即为 $\operatorname{sgn}(\boldsymbol{w}^T\boldsymbol{x}+b)$。为了更好的鲁棒性，可以选择 b 为所有满足条件的 j 计算出的值的平均。

SMO 算法

针对这个特定问题，可以使用 SMO 算法求解。该算法的基本思路是每次只取两个参数 α_i,α_j 更新，而固定其余参数不变。而优化这两个参数的过程十分高效，有闭式解。具体而言，两个变量的问题如下：

$$\max_{\alpha_1,\alpha_2}\frac{1}{2}K_{11}\alpha_1^2+\frac{1}{2}K_{22}\alpha_2^2+y_1y_2K_{12}\alpha_1\alpha_2-(\alpha_1+\alpha_2)+y_1\alpha_1\sum_{i=3}^{N}y_i\alpha_iK_{i1}+y_2\alpha_2\sum_{i=3}^{N}y_i\alpha_iK_{i2}$$

$$\text{s.t. }\alpha_1y_1+\alpha_2y_2=-\sum_{i=3}^{N}\alpha_iy_i=\zeta,$$

$$0\le\alpha_1,\alpha_2\le C$$

子问题求解

令：

$$\begin{cases}L=\max(0,\alpha_2-\alpha_1), & H=\min(C,C+\alpha_2-\alpha_1), & y_1\ne y_2\\ L=\max(0,\alpha_2+\alpha_1-C), & H=\min(C,\alpha_2+\alpha_1), & y_1=y_2\end{cases}$$

令 $g(\boldsymbol{x})=\sum_{i=1}^{N}y_i\alpha_iK(\boldsymbol{x}_i,\boldsymbol{x})$，其中 b 是学习的参数之一；损失 $E_i=g(\boldsymbol{x}_i)-y_i$，则问题的未经剪辑的解是（参考《统计学习方法》）

$$\alpha_2^{new,unc}=\alpha_2+\frac{y_2(E_1-E_2)}{\eta}$$

，剪辑后的解是

$$\alpha_2^{new}=\begin{cases}H, & \alpha_2^{new,unc}>H\\ \alpha_2^{new,unc}, & L\le\alpha_2^{new,unc}\le H,\\ L, & \alpha_2^{new,unc}<L\end{cases}$$

$$\alpha_1^{new}=\alpha_1+y_1y_2(\alpha_2^{new}-\alpha_2)$$

而对于 b 的更新，计算：

$$\begin{aligned}b_1&=-E_1-y_1K_{11}(\alpha_1^{new}-\alpha_1)-y_2K_{21}(\alpha_2^{new}-\alpha_2)+b\\ b_2&=-E_2-y_2K_{12}(\alpha_1^{new}-\alpha_1)-y_2K_{22}(\alpha_2^{new}-\alpha_2)+b\end{aligned}$$

可以令 $b^{new}=\dfrac{b_1+b_2}{2}$。

优化变量选取

对这样一对 α_1, α_2 的选取，SMO 采用启发式的方法。对偶问题的 KKT 条件如下：

$$\begin{aligned} \alpha_i = 0 &\Leftrightarrow y_i g(x_i) \geq 1 \\ 0 < \alpha < C &\Leftrightarrow y_i g(x_i) = 1 \\ \alpha_i = C &\Leftrightarrow y_i g(x_i) \leq 1 \end{aligned}$$

SMO 算法首先选取违反 KKT 条件最严重的样本点，将对应变量作为第一个变量。而使用 $|E_1 - E_2|$ 最大的样本作为第二个变量。若这样的选择不能造成足够大的优化，则可以重新随机选取第二个变量。

本代码具体采用如下略有不同的方法：多次遍历所有样本，若其对应参数违反 KKT 条件，则尝试以其作为第一个变量，按上面的规则选取第二个变量求解子问题。

梯度下降法

对软间隔部分给出的问题（采用 hinge 损失）：

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

（与线性回归相同，将 b 吸收入 w 中）

发现损失函数是凸的，并且没有对参数的限制，因此可以简单地采用梯度下降法。梯度如下：

$$\frac{\partial L}{\partial \boldsymbol{w}} = \boldsymbol{w} + C\sum_{i=1}^{N} \boldsymbol{p}_i$$

其中：

$$\boldsymbol{p}_i = \begin{cases} 0, & y_i \boldsymbol{w}^T \boldsymbol{x}_i > 1 \\ -y_i \boldsymbol{x}_i, & y_i \boldsymbol{w}^T \boldsymbol{x}_i \leq 1 \end{cases}$$

代码实现

下面解释核心代码的实现。

SMO算法

训练函数如下：

```
def fit(self, X: np.ndarray, y: np.ndarray, eps=1e-4, max_round=5):
    N = X.shape[0]
    alpha = np.zeros((N, 1))
    b = 0.0
    K = X @ (X.transpose())
    E = np.zeros((N, 1))
```

其中 K 作为预计算出的核矩阵（线性核），即 $K_{ij} = \boldsymbol{x}_i^T \boldsymbol{x}_j$。

写出计算 $g(\boldsymbol{x}_i)$ 与 E_i 的函数：

```
def gi(i):
    return K[i] @ (alpha * y) + b

def calcE():
    nonlocal E
    for i in range(N):
        E[i] = gi(i) - y[i]
```

外循环每轮检查所有变量是否满足 KKT 条件，若不满足即进入内循环。其中检验的条件是对于上面的条件的改写，因为 $y_iE_i = y_i(g_i - y_i) = y_ig_i - y_i^2 = y_ig_i - 1$，因此可以比较 y_iE_i 与 0 的大小，而无需额外计算 y_ig_i。

```
for round in range(max_round):
    tried, changed = 0, 0
    for i in range(N):
        if (y[i] * E[i] < -self.tol and alpha[i] < self.C - eps) or (
            y[i] * E[i] > self.tol and alpha[i] > 0 + eps
        ):
            tried += 1
            changed += innerLoop(i)
    if tried == 0:
        break
    print(tried, changed, round)
```

内循环中，已知 α_1, α_2，尝试更新参数的函数如下。它是代码最长的函数，但完全是“子问题求解”一节结果的转写：

```
def run(a2: int):
    nonlocal b
    # calc new alpha1 and alpha2
    L, H = 0, 0
    if y[a1] != y[a2]:
        L = max(0, alpha[a2] - alpha[a1])
        H = min(self.C, self.C + alpha[a2] - alpha[a1])
    else:
        L = max(0, alpha[a2] + alpha[a1] - self.C)
        H = min(self.C, alpha[a2] + alpha[a1])
    if L >= H:
        return 0

    def trunc(x):
        return H if x > H else (L if x < L else x)

    K11, K22, K12 = K[a1][a1], K[a2][a2], K[a1][a2]
    eta = K11 + K22 - 2 * K12

    alpha2 = trunc(alpha[a2] + y[a2] * (E[a1] - E[a2]) / eta)
    alpha1 = alpha[a1] + y[a1] * y[a2] * (alpha[a2] - alpha2)

    # calc b
    b1new = (
        -E[a1]
        - y[a1] * K11 * (alpha1 - alpha[a1])
        - y[a2] * K12 * (alpha2 - alpha[a2])
        + b
    )
    b2new = (
```

```
            -E[a2]
            - y[a1] * K12 * (alpha1 - alpha[a1])
            - y[a2] * K22 * (alpha2 - alpha[a2])
            + b
        )
        if abs(alpha[a2] - alpha2) < eps:
            return 0
        alpha[a1], alpha[a2] = alpha1, alpha2
        if 0 + eps < alpha[a1] < self.C - eps:
            b = b1new
        elif 0 + eps < alpha[a2] < self.C - eps:
            b = b2new
        else:
            b = (b1new + b2new) / 2.0
        calcE()
        return 1
```

若成功更新并且参数在精度范围内有差异，函数返回 1，否则返回 0。

内循环首先选取 $|E_1 - E_2|$ 最大的参数 α_2 进行求解，若结果不显著（函数返回 0），则另取一个随机参数作为 α_2 尝试求解。

```
    def innerLoop(a1: int):
        nonlocal b

        def randa2():
            a2 = randint(0, N - 1)
            while a2 == a1:
                a2 = randint(0, N - 1)
            return a2

        a2 = a1
        if E[a1] >= 0:
            a2 = E.argmin()
        else:
            a2 = E.argmax()
        a2 = int(a2)
        if run(a2) == 0:
            a2 = randa2()
            return run(a2)
        else:
            return 1
```

训练完成后，根据所有 $\alpha_j > 0$ 的 j 计算 b。

```
    # generate model
    self.w = np.zeros(self.dim)
    for i in range(N):
        self.w += alpha[i] * y[i] * X[i]

    js = np.where(alpha > eps)[0]
    for j in js:
        self.b += y[j]
        for i in range(N):
            self.b -= alpha[i] * y[i] * K[i][j]
    self.b /= len(js)
```

计算出 w 和 b 之后，只需按照 $\operatorname{sgn}(\boldsymbol{w}^T\boldsymbol{x} + b)$ 预测即可。

```python
def predict(self, X):
    return np.sign(np.dot(X, self.w) + self.b)
```

梯度下降

梯度下降法与实验一使用大体相同的方式。

按照前面推出的表达式容易写出计算梯度与损失的函数（为归一化，这里将后一部分求和改为平均，即除以数据集大小）。

```python
def gradient(self, X: np.ndarray, y: np.ndarray):
    g = np.zeros(self.dim)
    cond = np.dot(X, self.w) * y
    for i in range(X.shape[0]):
        if cond[i] <= 1:
            g -= self.C * y[i] * X[i]
    g /= X.shape[0]
    g += self.w
    return g

def loss(self, X: np.ndarray, y: np.ndarray):
    res = 0.0
    l = 1 - y * np.dot(X, self.w)
    res += self.C * np.sum(np.maximum(0, l))
    res /= X.shape[0]
    res += np.linalg.norm(self.w) / 2.0
    return res
```

则训练时只需根据学习率和计算出的梯度更新当前参数即可。

```python
def fit(self, X: np.ndarray, y: np.ndarray, lr=0.005, round=2000):
    X = np.c_[np.ones(X.shape[0]), X]
    y = y.reshape(X.shape[0])
    ls = []
    for epoch in range(round):
        self.w -= lr * self.gradient(X, y)
        ls.append(self.loss(X, y))
    print(ls[-10:])
```

预测函数与 SMO 算法类似，但注意这里将 b 吸收入了 w 中。

```python
def predict(self, X):
    X = np.c_[np.ones(X.shape[0]), X]
    return np.sign(np.dot(X, self.w))
```

测试

使用给定的生成随机数据函数，采用推荐的参数 `dim = 20, N = 10000` 进行测试，其中 20% 作为测试集。测试环境为 `Python 3.9.12`，CPU `i5-11400H`。

SMO 算法

使用 VS Code 插件的计时，若选择遍历所有样本 5 轮，训练在 5-7 分钟内可以完成，准确率在 $88\% - 93\%$ 之间。若选择遍历所有样本 10 轮，则需要 10 分钟以上的时间，准确率可以稳定在到 90% 以上。

梯度下降

选择较小的学习率 `0.005`，迭代轮数 `2000`。在本程序中，梯度下降有很好的表现，只需 10-15 秒即可完成计算，并且准确率稳定在 95% 以上。也可以适当减小迭代轮数、增大学习率以减小训练时间到 5 秒内，准确率不会有显著的降低。

sklearn

直接使用 `sklearn.SVC()` 类，在 1s 内即可完成训练，准确率在 93% 左右。但若增大数据量，它也无法在短时间内完成训练。

若改用 `sklearn.LinearSVC()`，使用默认参数会提醒模型不收敛，加入 `dual=False` 可解决该问题。其训练时间极短（小于一秒），准确率接近于上面手动实现的梯度下降。事实上，它的底层使用 `SGD`（随机梯度下降）方法，一般来说表现优于梯度下降法。

若再增大数据维度与样本个数，则只有梯度下降和 `LinearSVC` 可以在较短时间内完成训练。由此可见，在这一简单的问题下，使用简单的梯度下降方法可能更好，在复杂度、准确率表现上都要更优秀。