监督学习实验报告

- PB17111568
- 郭雨轩

写在前面

- 本实验所有的监督学习算法的代码均被封装成一个类，调用`fit(train_x, train_y)`的方法进行训练，调用`predict(test_x)`方法进行预测
- 本实验所有的监督学习算法的代码均被封装成一个类，调用`fit(train_x, train_y)`的方法得到得到算法输出的结果

KNN

算法思路

KNN的是一个“惰性学习”的算法，在训练阶段，KNN算法不做任何操作，只是单纯的保存输入的数据，在预测阶段，对于每个输入的测试集例子，在训练集合中按照某种距离度量找到与其最近的k个邻居，根据这些训练集合中的邻居的标签来确定该输入样例的标签。

代码实现

训练部分：

```
def fit(self, train_x, train_y):
    # 仅需要保存输入的数据集
    self.train_x = train_x.copy()
    self.train_y = train_y.copy()
```

测试部分：

```
def predict(self, test_x):
    '''
    :param test_x: [n_1, m]
    :return: [n_1, 1]
    '''
    assert self.train_x is not None, 'Not init train_x'
    assert self.train_y is not None, 'Not init train_y'
    assert self.train_x.shape[0] == self.train_y.shape[0]
    self.test_x = test_x.copy()
    result = []
```

```
    for t_x in test_x:
        # 对于训练集合中每个样例，先找到距离度量前K小的下标
        # 根据正例和反例的数量来确定预测集合label
        top_k_idx = np.argpartition(self.d_f(t_x, self.train_x), self.K)[:self.K]
        top_k_y = self.train_y[top_k_idx]
        predict_y = [1] if np.sum(top_k_y) / self.K > self.threshold else [0]
        result.append(predict_y)

    return np.array(result)
```

实验结果

在尝试了多组K值之后，我选定了$K = 25$，分别测试使用所有属性，使用所有属性除去G1，G2，和仅使用G1，G2，得到的精度信息如下：

student-por.csv	**P**	**R**	**F1**
使用所有属性	0.91	0.99	0.95
使用除去G1和G2的所有属性	0.89	0.99	0.94
仅使用G1和G2	0.96	0.99	0.98

student-mat.csv	**P**	**R**	**F1**
使用所有属性	0.91	0.93	0.92
使用除去G1和G2的所有属性	0.72	0.99	0.83
仅使用G1和G2	1.00	0.95	0.98

当逐渐增大K值时，各项指标的值先增加后减少，且在我的测试中，不同的距离度量函数并未带来很大的精度差别，推测可能是因为我预先做了一些数据的离散化操作，导致各项指标的差别都不大

SVM

算法思路

SVM的主要思路是解决一个线性划分问题时，如何找到更优的划分超平面的问题，具体是解决凸二次规划：

$$\min_{\omega,b} \frac{1}{2}\|\omega^2\|$$
$$s.t.\ y_i(\omega \cdot x_i + b) \geq 1,\ i = 1 \cdots N$$

由于这是一个凸二次规划问题，我们可以先对其使用拉格朗日乘数法，得到对偶问题：

$$\min_{\alpha} \frac{1}{2}\sum_{i=1}^{N}\sum_{j=1}^{N}\alpha_i\alpha_j y_i y_j(\mathbf{x_i}\cdot\mathbf{x_j^T}) - \sum_{i=1}^{N}\alpha_i$$
$$s.t.\ \sum_{i=1}^{N}\alpha_i y_i = 0,\ \alpha_i \geq 0,\ i = 1, \cdots, N$$

使用smo算法即可快速的求解出α_i，进而解决这个问题，求出：

$$\omega^* = \sum_{i=1}^{N}\alpha_i^* y_i \mathbf{x_i}$$
$$b^* = y_j - \sum_{i=1}^{N}\alpha_i^* y_i(\mathbf{x_i}\cdot\mathbf{x_j^T})$$

但是在很多情况下，输入的数据不会被完美的线性分割，这时候我们可允许一部分数据点不满足分割超平面，并对他们施加惩罚系数C，这就是软间隔SVM。

还有很多情况下，输入的问题并非线性可分，所以我们可以使用核函数的方式，讲输入的数据映射到高维空间中，而在高维空间中是线性可分的，具体的操作是使用核函数代替推导过程中所有的涉及到$\mathbf{x_i}\cdot\mathbf{x_j^T}$的部分，替换为$\mathbb{K}(\mathbf{x_i}, \mathbf{x_j})$，内积的结果直接由核函数给出。

代码实现

在本次实验中，由于不可以调用任何计算优化库，所以就不能使用有关解决凸二次规划的库。因此我主要参考了论文中的伪代码和《统计学习方法》一书中的推导过程，使用python程序对其进行了实现，本部分会一一进行讲解。

训练过程：

```
    def fit(self, train_x, train_y):
        self.train_x = np.mat(train_x.copy())
        self.train_y = train_y.copy()
        self.train_y[self.train_y == 0] = -1  # convert label to -1, +1
        self.train_y = np.mat(self.train_y)

        self.m, self.n = np.shape(self.train_x) # 初始化smo用到的参数
        self.alphas = np.mat(np.zeros((self.m, 1)))
        self.b = 0
        self.smo() # 调用smo计算alpha和b
```

smo：

```
    def smo(self):
    # smo函数的外层循环
        iters = 0
        num_changed = 0
```

```
        examine_all = True
        # 额外增加max iter函数，防止迭代过多次
        while iters < self.max_iter and (num_changed > 0 or examine_all):
            num_changed = 0
            # 若检查所有的alpha，则对所有的alpha迭代
            if examine_all:
                for i1 in range(self.m):
                    num_changed += self.examine(i1)
            else:
              # 否则仅对满足KTT条件的进行迭代
                for i1 in range(len(self.alphas)):
                    if 0 < self.alphas[i1] < self.C:
                        num_changed += self.examine(i1)
            iters += 1
            print(iters)
            if examine_all:
                examine_all = False
            elif num_changed == 0:
                examine_all = True

    def examine(self, i):
        f_X_i = self.f_x(self.alphas, self.train_x, self.train_y,
self.train_x[i], self.b)
        # 计算E_i
        E_i = f_X_i - float(self.train_y[i])

        # 检查是否违反约束条件
        if ((self.train_y[i] * E_i < -self.toler) and (self.alphas[i] <
self.C)) or (
                (self.train_y[i] * E_i > self.toler) and (self.alphas[i] >
0)):
            # 随机选择一个alpha_j
            j = self.select_j(i)
            f_X_j = self.f_x(self.alphas, self.train_x, self.train_y,
self.train_x[j], self.b)
            # 计算E_j
            E_j = f_X_j - float(self.train_y[j])
            alpha_i_old = self.alphas[i].copy()
            alpha_j_old = self.alphas[j].copy()
            # 计算L和H，并据此对alpha进行裁剪
            if self.train_y[i] != self.train_y[j]:
                L = max(0, self.alphas[j] - self.alphas[i])
                H = min(self.C, self.C + self.alphas[j] - self.alphas[i])
            else:
                L = max(0, self.alphas[j] + self.alphas[i] - self.C)
                H = min(self.C, self.alphas[j] + self.alphas[i])
```

```
            if L == H:
                return 0
            # 计算alpha更新所用的eta
            eta = -2.0 * self.kernel_f(self.train_x[i, :], self.train_x[j,
:]) + \
                  self.kernel_f(self.train_x[i, :], self.train_x[i, :]) +
self.kernel_f(self.train_x[j, :],

           self.train_x[j, :])
            if eta <= 0:
                return 0

            self.alphas[j] += self.train_y[j] * (E_i - E_j) / eta
            # 裁剪alpha
            if self.alphas[j] > H:
                self.alphas[j] = H
            elif self.alphas[j] < L:
                self.alphas[j] = L

            if abs(self.alphas[j] - alpha_j_old) < 0.00001:
                return 0

            self.alphas[i] += self.train_y[j] * self.train_y[i] *
(alpha_j_old - self.alphas[j])

            b1 = self.b - E_i - self.train_y[i] * (self.alphas[i] -
alpha_i_old) * self.kernel_f(self.train_x[i, :],

          self.train_x[i, :]) - \
                 self.train_y[j] * (self.alphas[j] - alpha_j_old) *
self.kernel_f(self.train_x[i, :], self.train_x[j, :])

            b2 = self.b - E_j - self.train_y[i] * (self.alphas[i] -
alpha_i_old) * self.kernel_f(self.train_x[i, :],

          self.train_x[j, :]) - \
                 self.train_y[j] * (self.alphas[j] - alpha_j_old) *
self.kernel_f(self.train_x[j, :], self.train_x[j, :])

            if (0 < self.alphas[i]) and (self.C > self.alphas[i]):  # 判断符合
条件的b，并计算
                self.b = b1
            elif (0 < self.alphas[j]) and (self.C > self.alphas[j]):
                self.b = b2
            else:
                self.b = (b1 + b2) / 2.0
```

```
            return 1
        else:
            return 0
```

推断过程：

```
    def predict(self, test_x):
        self.test_x = np.mat(test_x.copy())
        result = []
        for i in range(len(test_x)):
            # 使用计算出的权重计算w并根据wx+b的符号进行分类
            r = np.sign(self.f_x(self.alphas, self.train_x, self.train_y,
self.test_x[i, :], self.b))
            result.append([1] if r == 1 else [0])

        return np.array(result)
```

注：以上的代码中将内积全部替换为核函数

实验结果

本实验中固定C=0.2

线性核

student-por.csv	P	R	F1
使用所有属性	0.86	1	0.92
使用除去G1和G2的所有属性	0.82	1	0.90
仅使用G1和G2	0.84	0.99	0.91

student-mat.csv	P	R	F1
使用所有属性	0.69	1	0.82
使用除去G1和G2的所有属性	0.66	1	0.79
仅使用G1和G2	0.70	1	0.82

高斯核

选取高斯核，当σ非常小的时候，可以保证任何输入的数据集线性可分，进而可能有更好的实验效果。固定sigma=0.1：

student-por.csv	P	R	F1
使用所有属性	0.83	0.99	0.90
使用除去G1和G2的所有属性	0.86	0.99	0.92
仅使用G1和G2	0.83	0.99	0.90

student-mat.csv	P	R	F1
使用所有属性	0.64	0.99	0.78
使用除去G1和G2的所有属性	0.68	0.99	0.81
仅使用G1和G2	0.66	0.99	0.79

分析

SVM在这两个任务中表现不是很好，经常会将几乎所有的label预测为负例或者正例，可能是由于SMO算法实现的不是很好，有一些小错误，或者是因为需要对数据集先进性一定的预处理再使用SVM

ID3 (other supervise learning algorithm)

算法思路

对于离散的分类问题，决策树也是一个非常经典的算法。在对数据进行了适当的离散化之后，我使用信息增益作为属性区分的标准。使用训练数据集构建一颗决策树，对于测试集的每个例子，通过向下遍历决策树最终得到分类信息。具体构建决策树的算法在代码实现部分进行展示。

代码实现

训练部分：

```
def fit(self, train_x, train_y):
    self.train_x = train_x.copy()
    self.train_y = train_y.copy()
    m, n = self.train_x.shape
    self.tree = self.tree_generate(np.array([True] * m), list(range(n))) # 调用递归函数构建决策树
```

构建决策树：

```
def tree_generate(self, d_idx, a_idx):
    # 若当前数据的所有类别都相同，返回这个类别
    if np.max(self.train_y[d_idx, :]) == np.min(self.train_y[d_idx, :]):
```

```
        return int(self.train_y[d_idx][0][0])
    # 若当前所有的属性值都相同，返回这些行中数量最多的类别
    if len(a_idx) == 0 or check_equ(self.train_x[d_idx, :][:,  a_idx]):
        return int(np.argmax(np.bincount(np.squeeze(self.train_y[d_idx, :], 1))))

    node = {}
    # 计算信息增益，选出决策树分支的属性
    select_attr = self.select_attr(a_idx, d_idx)
    # 剩下的属性
    left_attrs = [a for a in a_idx if a != select_attr]
    # 对这个属性值域的每个值进行遍历
    for v in range(np.max(self.train_x[:, select_attr]) + 1):
        # 若这个值不对应任何样本，就返回所有训练集中数量最多的种类
        if len(self.train_y[self.query(d_idx, select_attr, v), :]) == 0:
            node[str(select_attr) + '-' + str(v)] = int(np.argmax(np.bincount(np.squeeze(self.train_y[d_idx, :], 1))))
        else:
            # 否则递归调用
            child = self.tree_generate(self.query(d_idx, select_attr, v), left_attrs)
            node[str(select_attr) + '-' + str(v)] = child
    # 返回决策节点
    return node
```

预测结果：

```
def predict(self, test_x):
    self.test_x = test_x.copy()
    result = []
    for line in self.test_x:
        n = self.tree
        # 若遍历到的对象不是整型（非叶节点），则继续遍历
        while not isinstance(n, int):
            # 根据属性名核属性值，构成查询的key，在字典中找到正确的分支
            attr = int(list(n.keys())[0].split('-')[0])
            val = line[attr]
            n = n[str(attr) + '-' + str(val)]
        result.append([n])
    return np.array(result)
```

实验结果

student-por.csv	P	R	F1
使用所有属性	0.95	0.93	0.94
使用除去G1和G2的所有属性	0.88	0.93	0.91
仅使用G1和G2	0.97	0.96	0.97

student-mat.csv	P	R	F1
使用所有属性	0.91	0.93	0.92
使用除去G1和G2的所有属性	0.69	0.85	0.76
仅使用G1和G2	0.96	0.97	0.97

决策树在分类上总体表现很好，但是在一些情况下会导致过拟合，可以通过对决策树进行剪枝或者对深度进行限制来保证具有泛化性。