人工智能基础编程作业 2

监督学习——学生表现预测

数据集

[Student Performance Data Set]

属性说明

数据集是包含了学生的各种属性，详见数据集说明，学习目标是根据其他属性（包含成绩`G1` `G2`与否）预测成绩`G3`

学习算法

- KNN, k-NearestNeighbor k-近邻算法
- SVM, Support Vector Machine 支持向量机
- LR, Logistic Regression 逻辑斯蒂回归

文件结构

```
supervise
|   README.md
├──data
|   └──student
|           student-mat.csv
|           student-merge.R
|           student-por.csv
|           student.txt
|
└──src
        KNN.py
        LR.py
        main.py
        SVM.py
```

依赖

程序依赖于`numpy`及`sklearn`（仅`sklearn.preprocessing`），并使用`rich`包显示了训练与预测过程的进度条

运行

在`src`目录下运行`main.py`文件，使用`-h`或`--help`的参数获得如下的帮助

```
> python main.py -h
usage: main.py [-h] [-d {math,portuguese}] [-G] {KNN,SVM,LR} ...

Simple machine learning test

optional arguments:
```

```
  -h, --help            show this help message and exit
  -d {math,portuguese}, --dataset {math,portuguese}
                        Dataset of the learning task (default: None)
  -G                    Use G1 & G2 as in attributes or not (default:
False)

Learning Algorithms:
  {KNN,SVM,LR}
    KNN                 k-Nearest Neighbors
    SVM                 Support Vector Machine
    LR                  Logistic Regression

PB17000297 罗晏宸 AI Programming Assignment 2
```

其中数据集选项`-d`是必选的，为学习任务指定使用数学`math`或葡萄牙语`portuguese`数据集，使用`-G`选项与否决定了是否使用数据集中的`G1`与`G2`属性。学习算法也是必需的选项，可以从 k-近邻`KNN`、支持向量机`SVM`、 Logistic 回归`LR`中选择。可以在选择参数后使用`-h`或`--help`的参数获得更多帮助，下面以支持向量机为例

```
> python main.py -d math -G SVM -h
usage: main.py SVM [-h] [-C penalty] [-t xi] {Gaussian,Linear,Polynomial}
...

Support Vector Machine

optional arguments:
  -h, --help            show this help message and exit
  -C penalty            Soft margin penalty hyperparameter for support
vector machine (default: 200)
  -t xi, --toler xi     Slack variable (toler) for support vector machine
(default: 0.0001)

Kernel Functions:
  {Gaussian,Linear,Polynomial}
    Gaussian            Gaussian kernel function(default)
    Linear              Linear kernel function
    Polynomial          Polynomial kernel function
```

进一步选择高斯核函数后，可以获得更多关于核参数的帮助与默认值信息

```
> python main.py -d math -G SVM Gaussian -h
usage: main.py SVM Gaussian [-h] [-s sigma]

Gaussian kernel function

optional arguments:
  -h, --help            show this help message and exit
  -s sigma, --sigma sigma
                        Parameter of gaussian kernel function for support
vector machine (default: 10)
```

其他核函数乃至算法的命令是类似的。

结果

程序成功运行后可以在终端看到训练及预测过程的进度，预测完成后会输出如下的模型评价信息

```
Elapsed time: 1.869s
TP = 262  TN =  69
FP =  61  FN =   3
F1 score: 89.115646%
```

算法、实现与效果

数据读取与预处理

在`main.py`中使用`loadData(file, Normalize, Methods)`函数从文件`file`中读取数据，后两个参数分别指示了是否对数据进行标准化以及对类别标签的处理：

- `'NearestNeighbors'` k-近邻算法，类别标签为 `0`-不及格与 `1`-及格
- `'LogisticRegression'` Logistic 回归算法，类别标签为 `0`-不及格与 `1`-及格
- `'SupportVectorMachine'` 支持向量机算法，类别标签为 `-1`-不及格与 `1`-及格

根据数据特征，对类型分别为`binary`或`nominal` `numeric`的数据分别处理，利用`sklearn.preprocessing.LabelEncoder().fot_transform`将非整数数据编码为整数

```
for line in lines:
    for i in [0, 1, 3, 4, 5, 8, 9, 10, 11, 15, 16, 17, 18, 19, 20, 21, 22]:
 # binary or nominal
        Attributes[i].append(line[i])
    for i in [2, 6, 7, 12, 13, 14, 23, 24, 25, 26, 27, 28, 29]:  # numeric
        Attributes[i].append(int(line[i]))
    Grades.append([int(num) for num in line[30:32]])  # 31:G1 32:G2
    # G3 >= 10 为及格
    Label.append(1 if int(line[-1]) >= 10 else (-1 if Methods ==
'SupportVectorMachine' else 0))
for i in [0, 1, 3, 4, 5, 8, 9, 10, 11, 15, 16, 17, 18, 19, 20, 21, 22]:
    Attributes[i] = labelEncoder.fit_transform(Attributes[i])  # 编码为整数
```

并将`G3`维度的成绩数据转化为是否及格的标签

可选的标准化过程将各维度数据归化到标准区间中

```
if Normalize == True:
    upperBounds = [1, 1, 22, 1, 1, 1, 4, 4, 4, 4, 3, 2, 4,
                   4, 3, 1, 1, 1, 1, 1, 1, 1, 1, 5, 5, 5, 5, 5, 5, 93]  #
属性上界
    lowerBounds = [0, 0, 15, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1,
                   1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 1, 1, 1, 1, 1, 0]  # 属
性下界

    Data_withoutG = Data_withoutG.astype(np.float64)  # 转换为浮点类型

    for i in range(Data_withoutG.shape[0]):
        for j in range(30):
            Data_withoutG[i, j] = (
                Data_withoutG[i, j] - lowerBounds[j]) / (upperBounds[j] -
lowerBounds[j])
    Grades = np.array(Grades, dtype=float) / 20
```

函数最终返回值包括：

- `Data_withG1G2` 包含属性 G1 G2 的数据集
- `Data_withoutG1G2` 不包含属性 G1 G2 的数据集
- `Label` 指示是否及格的标签集

数据划分

在`main.py`中使用`dataSplit(data, label, train_size, test_size, shuffle)`函数对原始数据`data`和`label`进行训练集与测试集的划分，参数指定了训练集与测试集的比例以及是否对数据进行打乱，默认的划分是$\text{训练集}:\text{测试集}=7:3$

函数中使用相同的随机种子打乱原始数据`data`和`label`

```
if shuffle:
    randomSeed = random.randint(0, 100)
    random.seed(randomSeed)
    random.shuffle(data)
    random.seed(randomSeed)
    random.shuffle(label)
```

并根据比例计算训练集与测试集的大小

```
trainStripIndex = math.floor(train_size * len(data))
testStripIndex = math.ceil(test_size * len(data))

trainData = data[0: trainStripIndex]
testData = data[trainStripIndex: trainStripIndex + testStripIndex]
trainLabel = label[0: trainStripIndex]
testLabel = label[trainStripIndex: trainStripIndex + testStripIndex]
```

模型测试

在`main.py`中使用函数`modelTest(testLabel, predictLabel)`根据测试集标签与预测标签对模型的学习结果做评价，评价标准为 F1 值

$$\text{F1 Score}=\frac{2\times\text{Precision}\times\text{Recall}}{\text{Precision}+\text{Recall}}$$

其中准确率 Precision 与召回率 Recall 分别为

$$\text{Precision}=\frac{\text{TP}}{\text{TP}+\text{FP}}$$
$$\text{Recall}=\frac{\text{TP}}{\text{TP}+\text{FN}}$$

三种学习算法分别位于`KNN.py` `SVM.py` `LR.py`中，在`main.py`中导入，并且均提供相同的`predict(trainData, trainLabel, testData, ...)`的函数接口（可选参数对学习算法进行调参），使用训练集`trainData`及训练集标签`trainLabel`，预测测试数据集`testData`的分类标签。

KNN, k-NearestNeighbor k-近邻算法

算法思想

KNN 是一种基于实例的学习，或者是局部近似和将所有计算推迟到分类之后的惰性学习。所以算法中没有实质上的训练过程，而是根据输入空间中的所有观测，由一个数据点附近的 k 个邻居对其进行多数表决而确定其分类，在本次学习任务中，即指某位学生是否及格。

算法中对于两个对象间距离的计算使用了 Manhattan 距离，这是基于数据集属性均为离散的整数值的考虑。通常 KNN 算法中的距离计算会使用欧式距离(Euclidean metric)、海明距离、或 Manhattan 距离，分别适用于连续值、文本分类与离散值属性数据，故做此选择。

代码实现

距离计算

Manhattan 距离是各分量差值绝对值的线性和，可直接计算如下

```
def distanceBetween(j, q):
    return np.sum(np.abs(j - q))
```

近邻预测

通过 k-最近邻确定测试数据的标签的函数`NearestNeighbor(trainData, trainLabel, testDatum, K)`，函数根据由`trainData`与`trainLabel`组成的输入空间预测单个测试数据`testDatum`的标签，参数`K`指定了最近邻样本的采样数目

```
def NearestNeighbor(trainData, trainLabel, testDatum, K):
    Distances = np.array([distanceBetween(trainDatum, testDatum)
                          for trainDatum in trainData])  # 向量间距离

    topK_Neighbors = np.argpartition(Distances, K)[:K]  # k-近邻

    labelList = [0, 0]  # 0 : failed  1 : passed
    for x in topK_Neighbors:
        labelList[int(trainLabel[x])] += 1  # 统计标签为对应类别的近邻数
    return labelList.index(max(labelList))  # 返回具有最多相同近邻数的标签
```

函数首先计算了测试数据`testDatum`与样本空间`trainData`的距离，并利用`numpy`包中的`argpartition`方法，按距离得到k-最近邻，使用一个长度为 2 的列表`labelList`对这些邻居进行唱票，计票最多的下标对应了测试数据的预测分类。

接口函数

在函数`KNN.predict(trainData, trainLabel, testData, K)`中，
默认的参数`K`取 27，依次对所有的测试数据使用`NearestNeighbor`函数得到预测分类。

```
def predict(trainData, trainLabel, testData, K=27):
    predictLabel = []

    for x in testData:
        predictLabel.append(NearestNeighbor(
            trainData, trainLabel, x, K))  # 预测标签分类

    return predictLabel
```

实验结果

对于数学成绩，经过反复调参尝试，确定$k = 11$时有比较好的效果，结果如下

```
python .\main.py -d math -G SVM -K 11
python .\main.py -d math SVM -K 11
```

math	耗时	TP	TN	FP	FN	Precision	Recall	F1
使用 G1 & G2	0.288s	81	30	6	2	93.103%	97.590%	95.294%
不使用 G1 & G2	0.338s	84	3	27	5	75.676%	94.382%	84.000%

对于葡萄牙语成绩，经过反复调参尝试，确定$k=11$时有比较好的效果，结果如下

```
python .\main.py -d portuguese -G SVM -K 11
python .\main.py -d portuguese SVM -K 11
```

portuguese	耗时	TP	TN	FP	FN	Precision	Recall	F1
使用 G1 & G2	0.741s	172	11	11	1	93.989%	99.422%	96.629%
不使用 G1 & G2	0.892s	172	0	22	1	88.660%	99.422%	93.733%

从 0 开始增加 k ，各指标基本增加，但由于数据集并不大，当k继续增大时，各指标出现显著降低，学习效果较差。并且由于数据离散程度较小，在没有其他连续化处理的情况下，过多的近邻对于预测水平的提升没有什么帮助。

SVM, Support Vector Machine 支持向量机

算法思想（实验及以下内容部分参考维基百科）

支持向量机是在分类与回归分析中分析数据的监督式学习模型与相关的学习算法。给定一组训练观测，每个训练实例被标记为属于两个类别中的一个，SVM 训练算法创建一个将新的实例分配给两个类别之一的模型，使其成为非概率二元线性分类器。SVM 模型是将实例表示为高维度空间中的点，这样映射就使得单独类别的实例被尽可能宽的明显的间隔分开。然后，将新的实例映射到同一空间，并基于它们落在间隔的哪一侧来预测所属类别。

更正式地来说，支持向量机在高维或无限维空间中构造超平面或超平面集合，其可以用于分类、回归或其他任务。直观来说，分类边界距离最近的训练数据点越远越好，因为这样可以缩小分类器的泛化误差。即学习的目的是找到一个最大间隔超平面

$$\vec{w}\cdot\vec{x}-b=0$$

使得超平面与最近的点 $\vec{x}_i$ 之间的距离最大化。为了解决这一问题即求解超平面参数，通常需要对 SVM 产生的参数最值的线性规划问题（或其对偶问题）求解二次规划(Quadratic programming, QP)。有几种专门的算法可用于快速解决由 SVM 产生的 QP 问题，它们主要依靠启发式算法将问题分解成更小、更易于处理的子问题。在本任务中，我们选用**序列最小优化算法(Sequential minimal optimization, SMO)**进行解决。

考虑数据集$(\mathbf{x_1},y_1),\ldots,(\mathbf{x_n},y_n)$的二分类问题，其中$\mathbf{x_i}$ 是输入向量，$y_i\in\{-1,1\}$是向量的类别标签，只允许取两个值。一个软间隔支持向量机的目标函数最优化等价于求解以下二次规划问题的最大值：

$$W = \max_{\alpha} \sum_{i=1}^{n} \alpha_i - \frac{1}{2} \sum_{i=1}^{n} \sum_{j=1}^{n} y_i y_j K(\mathbf{x}_i, \mathbf{x}_j) \alpha_i \alpha_j,$$

满足：

$$0 \leq \alpha_i \leq C, \quad i = 1, 2, \ldots, n,$$
$$\sum_{i=1}^{n} y_i \alpha_i = 0,$$

其中，C 是软间隔惩罚参数，而$K(\mathbf{x}_i, \mathbf{x}_j)$是核函数。这两个参数都需要使用者制定。在本任务中，有 3 个核函数得到了具体实现，分别为

- 线性核函数（无参数）

$$K(\mathbf{x}_i, \mathbf{x}_j) = \mathbf{x}_i \cdot \mathbf{x}_j$$

- 高斯核函数（即径向基核函数，参数为σ）

$$K(\mathbf{x}_i, \mathbf{x}_j) = e^{\frac{\|\mathbf{x}_i - \mathbf{x}_j\|^2}{2\sigma^2}}$$

- 多项式核函数（参数为p）

$$K(\mathbf{x}_i, \mathbf{x}_j) = (\mathbf{x}_i \cdot \mathbf{x}_j + 1)^p$$

SMO 是一种解决此类优化问题的迭代算法。由于目标函数为凸函数，通常的算法都通过梯度方法每次迭代优化一个变量求解二次规划问题的最大值，但是，对于以上问题，由于限制条件$\sum_{i=1}^{n} y_i \alpha_i = 0$存在，当某个$\alpha_i$ 从α_i^{old}更新到α_i^{new}时，上述限制条件随即破坏，而 SMO 采用一次更新一对变量的方法来解决这个问题。

在每一次迭代中，算法首先选取两个旧向量α_1^{old}和α_2^{old}，前者是不满足 **Karush-Kuhn-Tucker, KKT 条件**

$$y_i f(\mathbf{x}_i) \begin{cases} > 1 & \alpha_i = 0 \\ = 1 & 0 < \alpha_1 < C \\ < 1 & \alpha_i = C \end{cases}$$

的变量，而后者是使得$|E_1 - E_2|$最大的向量。则其余变量都可以视为常量，记

$$K_{ij} = K(\mathbf{x_i}, \mathbf{x_j}), f(\mathbf{x_i}) = \sum_{j=1}^{n} y_j \alpha_j K_{ij} + b,$$
$$v_i = f(\mathbf{x_i}) - \sum_{j=1}^{2} y_j \alpha_j K_{ij} - b$$

则二次规划目标值可以写成

$$\begin{aligned} W(\alpha_1, \alpha_2) &= \textstyle\sum_{i=1}^{n} \alpha_i - \frac{1}{2} \sum_{i=1}^{n} \sum_{j=1}^{n} y_i y_j K(x_i, x_j) \alpha_i \alpha_j \\ &= \alpha_1 + \alpha_2 - \tfrac{1}{2} K_{11} \alpha_1^2 - \tfrac{1}{2} K_{22} \alpha_2^2 - y_1 y_2 K_{12} \alpha_1 \alpha_2 - y_1 \alpha_1 v_1 - y_2 \alpha_2 v_2 + \mathrm{C}' \end{aligned}$$

由于限制条件$\sum_{i=1}^{n} y_i \alpha_i = 0$存在，将$\alpha_3, \ldots, \alpha_n, y_3, \ldots, y_n$看作常数，则有$\alpha_1 y_1 + \alpha_2 y_2 = \mathrm{C}$成立（C为常数）。由于$y_i \in \{-1, 1\}$，从而$\alpha_1 = \gamma - s\alpha_2$，其中

$$\gamma = \frac{\mathrm{C}}{y_1} = y_1\mathrm{C} = \mathrm{C}''$$
$$s = \frac{y_2}{y_1} = y_2 y_1$$

取α_2为优化变量，则上式又可写成

$$\begin{aligned} W(\alpha_2) &= \gamma - s\alpha_2 + \alpha_2 - \tfrac{1}{2}K_{11}(\gamma - s\alpha_2)^2 - \tfrac{1}{2}K_{22}\alpha_2^2 - \\ &\quad sK_{12}(\gamma - s\alpha_2)\alpha_2 - y_1(\gamma - s\alpha_2)v_1 - y_2\alpha_2 v_2 + \mathrm{C}''' \end{aligned}$$

对α_2求偏导以求得最大值，有

$$\begin{aligned} \frac{\partial W(\alpha_2)}{\partial \alpha_2} &= -s + 1 + sK_{11}\gamma - K_{11}\alpha_2 - K_{22}\alpha_2 + 2K_{12}\alpha_2 - sK_{12}\gamma + y_2 v_1 - y_2 v_2 \\ &= 0 \end{aligned}$$

因此，可以得到

$$\alpha_2^{\text{new}} = \frac{y_2(y_2 - y_1 + y_1\gamma(K_{11} - K_{12}) + v_1 - v_2)}{K_{11} + K_{22} - 2K_{12}}$$

规定误差项$E_i = f(\mathbf{x}_i) - y_i$，取$\gamma = \alpha_1^{\text{old}} + s\alpha_2^{\text{old}}$，并记$K = K_{11} + K_{22} - 2K_{12}$，上述结果可以化简为

$$\alpha_2^{\text{new}} = \alpha_2^{\text{old}} + \frac{y_2(E_1 - E_2)}{K}$$

再考虑限制条件$0 \leqslant \alpha_i \leqslant C$，$(\alpha_1, \alpha_2)$的取值只能为直线$\alpha_1 y_1 + \alpha_2 y_2 = \gamma$落在矩形$[0, C] \times [0, C]$中的部分。因此，具体的还需要检查$\alpha_2^{\text{new}}$的值以确认其落在约束区间之内。计算$\alpha_2^{\text{new}}$的下界$U$与上界$V$，有

$$U = \begin{cases} \max\left\{0, \alpha_2^{\text{old}} - \alpha_1^{\text{old}}\right\} & y_1 y_2 = -1 \\ \max\left\{0, \alpha_1^{\text{old}} + \alpha_2^{\text{old}} - C\right\} & y_1 y_2 = 1 \end{cases}$$
$$V = \begin{cases} \min\left\{C, C + \alpha_2^{\text{old}} - \alpha_1^{\text{old}}\right\} & y_1 y_2 = -1 \\ \min\left\{C, \alpha_2^{\text{old}} + \alpha_1^{\text{old}}\right\} & y_1 y_2 = 1 \end{cases}$$

最后再根据 SVM 的定义计算出偏移量b。

SMO 算法在所有变量均满足 KKT 条件时终止。

代码实现

与 KNN 不同，使用 SVM 进行学习任务有一定的训练过程，训练得到的模型即由参数描述的最大间隔超平面，因此在`SVM.py`中除了统一的接口函数`predict`，通过一个名为`SupportVectorMachine`的类实现支持向量机模型的初始化、训练、预测。

类构造函数

除了对于软间隔惩罚参数C与松弛变量ε进行初始化，例化一个支持向量机对象时还需要对其使用的核函数以及相应参数进行指定。

```
    def __init__(self, kernel='Gaussian', C=200, epsilon=0.0001, sigma=10,
p=2):

        self.__C = C
        self.__epsilon = epsilon

        if kernel == 'Gaussian':
            self.Kernel = self.__GaussianKernel
            self.__sigma = sigma
        elif kernel == 'Linear':
            self.Kernel = self.__LinearKernel
        elif kernel == 'Polynomial':
            self.Kernel = self.__PolynomialKernel
            self.__p = p
```

核函数

不同的核函数均实现为类方法，在初始化时作为对象拷贝为通用方法 `SupportVectorMachine.Kernel`

```
    def __LinearKernel(self, j, k):

        return np.dot(j, k)
```

```
    def __GaussianKernel(self, j, k, sigma=None):

        if sigma == None:
            sigma = self.__sigma

        return np.exp(-np.sum(np.square(j - k)) / (2 * sigma**2))
```

```
    def __PolynomialKernel(self, j, k, p=None):

        if p == None:
            p = self.__p

        return np.power(np.dot(j, k) + 1, p)
```

KKT 条件判定

一个返回值为布尔类型的方法 `__ifSatisfyKKT`，用于判断训练样本点(x_i, y_i)是否违反 KKT 条件，以挑选 SMO 算法每次迭代的首个变量

```
    def __ifSatisfyKKT(self, i, C=None, epsilon=None):

        if C == None:
            C = self.__C
        if epsilon == None:
            epsilon = self.__epsilon

        z = self.__y[i] * (np.sum([self.__alpha[j] * self.__y[j] *
self.__K[i, j]
                                   for j in range(self.__alpha.size)]) +
self.__b)  # y_i * g(x_i)

```

```
        if ((-epsilon < self.__alpha[i] < epsilon) and (z >= 1 - epsilon))
or \
            ((C - epsilon < self.__alpha[i] < C + epsilon) and (z <= 1 +
epsilon)) or \
                ((-epsilon < self.__alpha[i] < C + epsilon) and (1 -
epsilon <= z <= 1 + epsilon)):
            return True
        return False
```

训练过程—— SMO 算法的实现

通过一个过程化类方法，实现前述的迭代步骤，对类成员变量即相应模型参数进行修正

```
    def train(self, trainData, trainLabel):
        self.__x, self.__y = np.array(trainData), np.array(trainLabel)
        self.__alpha = np.zeros(self.__x.shape[0])
        self.__b = 0
        self.__K = np.zeros(
            [self.__x.shape[0], self.__x.shape[0]], dtype=float)  # 训练数据
核函数表

        for i in range(self.__x.shape[0]):
            for j in range(i, self.__x.shape[0]):
                self.__K[i, j] = self.__K[j, i] = self.Kernel(
                    self.__x[i], self.__x[j])  # 计算核函数表


        allSatisfied = False  # 全部满足 KKT 条件
        iteration = 1  # 迭代次数
        while not allSatisfied:
            allSatisfied = True
            iteration += 1
            for i in range(self.__x.shape[0]):  # 外层循环
                if not (self.__ifSatisfyKKT(i)):  # 选择第一个变量
                    E1 = self.__Error(i)
                    maximum = -1
                    for k in range(self.__x.shape[0]):  # 内层循环
                        tempE = self.__Error(k)
                        tempE_difference = np.fabs(E1 - self.__Error(k))
                        if tempE_difference > maximum:  # 选择第二个变量
                            maximum = tempE_difference
                            E2 = tempE
                            j = k
                    if maximum == -1:
                        continue

                    U = max(0, (self.__alpha[i] + self.__alpha[j] -
self.__C) if self.__y[i]
                               == self.__y[j] else (self.__alpha[j] -
self.__alpha[i]))  # alpha^2_new 的下界
                    V = min(self.__C, (self.__alpha[i] + self.__alpha[j])
if self.__y[i]
                               == self.__y[j] else (self.__alpha[j] -
self.__alpha[i] + self.__C))  # alpha^new_2 的上界
                    alpha_2_new = self.__alpha[j] + self.__y[j] * (E1 - E2)
/ (
```

```
                        self.__K[i, i] + self.__K[j, j] - 2 * self.__K[i,
j])

                    # alpha^2_new 越界
                    if alpha_2_new > V:
                        alpha_2_new = V
                    elif alpha_2_new < U:
                        alpha_2_new = U

                    alpha_1_new = self.__alpha[i] + self.__y[i] * \
                        self.__y[j] * (self.__alpha[j] - alpha_2_new)

                    # 更新偏置
                    b_1_new = -E1 - self.__y[i] * self.__K[i, i] * (
                        alpha_1_new - self.__alpha[i]) - self.__y[j] *
self.__K[j, i] * (alpha_2_new - self.__alpha[j]) + self.__b
                    b_2_new = -E2 - self.__y[i] * self.__K[i, j] * (
                        alpha_1_new - self.__alpha[i]) - self.__y[j] *
self.__K[j, j] * (alpha_2_new - self.__alpha[j]) + self.__b

                    # 实装更新
                    if (np.fabs(self.__alpha[i] - alpha_1_new) < 0.0000001)
and (np.fabs(self.__alpha[j] - alpha_2_new) < 0.0000001):
                        continue
                    else:
                        allSatisfied = False

                    self.__alpha[i] = alpha_1_new
                    self.__alpha[j] = alpha_2_new

                    if 0 < alpha_1_new < self.__C:
                        self.__b = b_1_new
                    elif 0 < alpha_2_new < self.__C:
                        self.__b = b_2_new
                    else:
                        self.__b = (b_1_new + b_2_new) / 2

        return
```

其中误差项的计算是另一个类方法，其计算了当前模型参数下第i个数据的误差

```
    def __Error(self, i):

        return np.sum([self.__alpha[j] * self.__y[j] * self.__K[i, j] for j
in range(self.__alpha.size)]) + self.__b - self.__y[i]
```

分类预测

作为`SupportVectorMachine`类对外的方法接口，`classify`实现了对单个测试数据`testDatum`的类别标签进行预测

```
    def classify(self, testDatum):

        distance = np.sum([self.__alpha[i] * self.__y[i] *
self.Kernel(testDatum, self.__x[i])
                           for i in range(self.__x.shape[0]) if
self.__alpha[i] > 0]) + self.__b  # 支持向量 alpha > 0
        return np.sign(distance)
```

接口函数

在函数 `SVM.predict(trainData, trainLabel, testData, kernel, C, epsilon, sigma, p)` 中，默认的核函数参数 `kernel` 为 `'Gaussian'` 即高斯核函数，核参数$\sigma = 10$，默认的软间隔惩罚参数$C = 200$，松弛变量$\varepsilon = 0.0001$，通过例化一个支持向量机类对象 `machine`，并对其进行初始化、训练，随即调用其 `classify` 方法对测试数据进行分类预测

```
def predict(trainData, trainLabel, testData, kernel='Gaussian', C=200,
epsilon=0.0001, sigma=10, p=2):

    predictLabel = []
    machine = SupportVectorMachine(
        kernel=kernel, C=C, epsilon=epsilon, sigma=sigma) # 初始化模型对象
    machine.train(trainData, trainLabel) # 训练模型

    for testDatum in testData:
        predictLabel.append(machine.classify(testDatum))  # 预测标签分类

    return predictLabel
```

实验结果

在列出实验结果之前，由于支持向量机超参较多，先对于任务的参数选择做一个讨论。

SVM 本身的两个重要参数——软间隔惩罚 `C` 与核参数在很大程度上决定了其有效性，包括松弛变量 `\varepsilon` 在内的各参数的最佳组合通常使用指数增长序列下的网格搜索来选取，下面是 `main.py` 中的网格搜索函数

```
def gridSearch_Gaussian(trainData, trainLabel, testData, testLabel):
    C = [np.power(2.0, i) for i in range(-5, 16, 2)]
    Sigma = [np.power(2.0, i) for i in range(-3, 8)]
    Epsilon = [np.power(10.0, i) for i in range(-6, 0)]
    subRange = 100

    maximumArguments = (0, 0, 0)
    maximumF1Score = 0
    for c, sigma, epsilon in [(c, sigma, epsilon) for c in C for sigma in
Sigma for epsilon in Epsilon]:
        print(c, sigma, epsilon)
        predictLabel = SVM.predict(
            trainData[:subRange], trainLabel[:subRange], testData, C=c,
sigma=sigma, epsilon=epsilon)
        f1Score = modelTest(testLabel, predictLabel)
        if f1Score > maximumF1Score:
            maximumF1Score = f1Score
            maximumArguments = (c, sigma, epsilon)
    print(maximumF1Score, maximumArguments)
```

最终选定并在代码中设为默认的 SVM 超参数为$C = 200$，$\varepsilon = 0.0001$

定性的来说，对于本次任务默认使用的高斯核函数，如果 σ 选得很大的话，高次特征上的权重实际上衰减得非常快，所以实际上相当于一个低维的子空间；相对的，如果 σ 很小，则可以将任意的数据映射为线性可分，可能导致非常严重的过拟合问题。

下面是 SVM 在各核函数下的实验结果

高斯核函数

对于数学成绩，经过反复调参尝试，确定$\sigma = 10$时有比较好的效果，结果如下

```
python .\main.py -d math -G SVM
python .\main.py -d math SVM
```

math	耗时	TP	TN	FP	FN	Precision	Recall	F1
使用 G1 & G2	171.5s	67	29	11	12	85.897%	84.810%	85.350%
不使用 G1 & G2	213.9s	57	26	18	18	76.000%	76.000%	76.000%

对于葡萄牙语成绩，经过反复调参尝试，确定$\sigma = 10$时有比较好的效果，结果如下

```
python .\main.py -d portuguese -G SVM
python .\main.py -d portuguese SVM
```

portuguese	耗时	TP	TN	FP	FN	Precision	Recall	F1
使用 G1 & G2	487.2s	136	40	6	13	95.775%	91.275%	93.470%
不使用 G1 & G2	390.0s	112	54	18	11	86.154%	91.057%	88.538%

线性核函数

对于数学成绩，结果如下

```
python .\main.py -d math -G SVM Linear
python .\main.py -d math SVM Linear
```

math	耗时	TP	TN	FP	FN	Precision	Recall	F1
使用 G1 & G2	112.8s	80	20	19	0	80.810%	100.00%	89.385%
不使用 G1 & G2	89.32s	60	24	25	10	70.588%	85.714%	77.419%

对于葡萄牙语成绩，结果如下

```
python .\main.py -d portuguese -G SVM Linear
python .\main.py -d portuguese SVM Linear
```

portuguese	耗时	TP	TN	FP	FN	Precision	Recall	F1
使用 G1 & G2	592.1s	112	60	13	10	89.600%	91.803%	90.688%
不使用 G1 & G2	413.6s	89	70	16	20	84.762%	81.651%	83.178%

多项式核函数

对于数学成绩，经过反复调参尝试，确定$p = 2$时有比较好的效果，结果如下

```
python .\main.py -d math -G Polynomial
python .\main.py -d math Polynomial
```

math	耗时	TP	TN	FP	FN	Precision	Recall	F1
使用 G1 & G2	298.6s	80	0	39	0	67.227%	100.00%	80.402%
不使用 G1 & G2	513.9s	69	8	28	14	71.134%	83.132%	76.667%

对于葡萄牙语成绩，经过反复调参尝试，确定$p = 2$时有比较好的效果，结果如下

```
python .\main.py -d portuguese -G Polynomial
python .\main.py -d portuguese Polynomial
```

portuguese	耗时	TP	TN	FP	FN	Precision	Recall	F1
使用 G1 & G2	396.2s	160	0	31	4	83.770%	97.561%	90.561%
不使用 G1 & G2	271.8s	87	62	29	17	75.000%	83.564%	79.090%

可以看到，相比之下高斯核函数有着更好的学习表现，由于高斯核函数的值随距离增大而减小，并介于 0 和 1 之间，所以它是一种相似性度量表示法。在各种使用 SVM 的学习任务中也有很好的泛用性。

但与其他两种学习算法比较可见，SVM 在小规模的数据集上效果并不明显，同时耗时较长，任务效率较低。

LR, Logistic Regression 逻辑斯蒂回归

为了便于横向比较，基于学生成绩数据集的 Logistic 回归任务目标仍是预测`G3`成绩。

算法思想

用 Logistic 函数

$$f(z) = \frac{1}{1 + e^{-z}}$$

代替线性回归中的阈值函数，拟合该模型权重以使得数据集上的损耗极小化的过程。即求解参数向量$\mathbf{w}$，使得分类器

$$h_{\mathbf{w}}(\mathbf{x}) = \frac{1}{1 + e^{-\mathbf{w}\cdot\mathbf{x}}}$$

适用。其权重的更新为

$$w_i \leftarrow w_i + \alpha(y - h_{\mathbf{w}}(\mathbf{x})) \times h_{\mathbf{w}}(\mathbf{x})(1 - h_{\mathbf{w}}(\mathbf{x})) \times x_i$$

代码实现

使用 Logistic 回归进行学习任务也有着较为完整的训练过程，训练得到的模型即由参数描述的阈值分类器，因此在`LR.py`中除了统一的接口函数`predict`，通过一个名为`LogisticRegressionClassifier`的类实现分类器的初始化、训练、预测。

类构造函数

对回归的最大迭代次数与学习速率进行初始化

```
    def __init__(self, iteration=200, learning_rate=0.0001):

        self.__alpha = learning_rate
        self.__iteration = iteration
```

训练过程

通过一个过程化类方法，实现回归的权重更新

```
    def train(self, trainData, trainLabel):

         self.__x = np.insert(np.array(trainData).astype(
             float), 0, values=1.0, axis=1)  # 训练数据，增加哑变量
         self.__y = np.array(trainLabel)  # 训练样本
         self.__weights = np.zeros(self.__x.shape[1], dtype=float)  # 初始化分类器权重

         for iter in range(self.__iteration):
             for i in range(self.__x.shape[0]):
                 h = self.__sigmoid(np.dot(self.__x[i], self.__weights))
                 self.__weights += self.__alpha * \
                     (self.__y[i] - h) * h * (1 - h) * self.__x[i]  # 更新权重

         return
```

其中 sigmoid 函数由另一个类方法实现

```
    def __sigmoid(self, x):
        return 1 / (1 + np.exp(-x))
```

分类预测

作为`LogisticRegressionClassifier`类对外的方法接口，`classify`实现了对单个测试数据`testDatum`的类别标签进行预测，根据阈值对数据分类做出预测

```
    def classify(self, testDatum):

        h = self.__sigmoid(
            np.dot(self.__weights, np.insert(testDatum, 0, values=1.0)))  # 增加哑变量
        return 1 if h >= 0.5 else 0  # 二分类
```

接口函数

在函数`LR.predict(trainData, trainLabel, testData, iteration, learning_rate)`中，默认的迭代次数为 200，学习速率为 0.0001，通过例化一个 Logistic 回归分类器对象`classifier`，并对其进行初始化、训练，随即调用其`classify`方法对测试数据进行分类预测

```
def predict(trainData, trainLabel, testData, iteration=200, learning_rate=0.0001):

    predictLabel = []
    classifier = LogisticRegressionClassifier(iteration, learning_rate)
    classifier.train(trainData, trainLabel)

    for testDatum in testData:
        predictLabel.append(classifier.classify(testDatum))  # 预测标签分类

    return predictLabel
```

实验结果

在默认的参数设置下，学习任务有如下结果

对于数学成绩，结果如下

```
python .\main.py -d math -G LR
python .\main.py -d math LR
```

math	耗时	TP	TN	FP	FN	Precision	Recall	F1
使用 G1 & G2	1.174s	63	50	0	6	100.00%	91.304%	95.454%
不使用 G1 & G2	1.634s	72	36	8	3	90.000%	96.000%	92.903%

对于葡萄牙语成绩，结果如下

```
python .\main.py -d portuguese -G LR
python .\main.py -d portuguese LR
```

portuguese	耗时	TP	TN	FP	FN	Precision	Recall	F1
使用 G1 & G2	2.197s	158	35	1	1	99.371%	99.371%	99.371%
不使用 G1 & G2	1.96s	157	0	38	0	80.513%	100.00%	89.205%

可以在小规模数据集上看到回归分类器的效果是非常好的，同时也将训练时间控制在一个不错的范围内。

注

所有结果展示的运行质量均列在相应的表格之前，在`src`文件夹下运行即可复现工作，但由于数据集划分的随机性，以及算法中的部分随机过程，复现结果可能与前述有所出入。