曹瑞升-5140219073 秦格华-5140219335

25/05/2017

Project5 实验报告

一. kNN 实现细节

我们首先利用 Python 实现了 kNN 分类算法，简要实现方法如下：

(a) 首先通过 Numpy 库的 `loadtxt` 方法将训练集读入，并保存在数组变量 `train_matrix` 中。同样将测试集保存在变量 `test_matrix` 中；

(b) 作为监督学习的算法，我们将 `train_matrix` 作为已知标签数据，利用其对每个 `test_matrix` 中的样本和我们在 `methods.py` 中写好的计算方法 `calculateknn()` 进行 kNN 的分析和预测，给出相应的标签。

二. kNN 实验结果

我们首先测试了在 $k = 5$ 的情况下测试集的输出结果，可视化后结果如图 1。

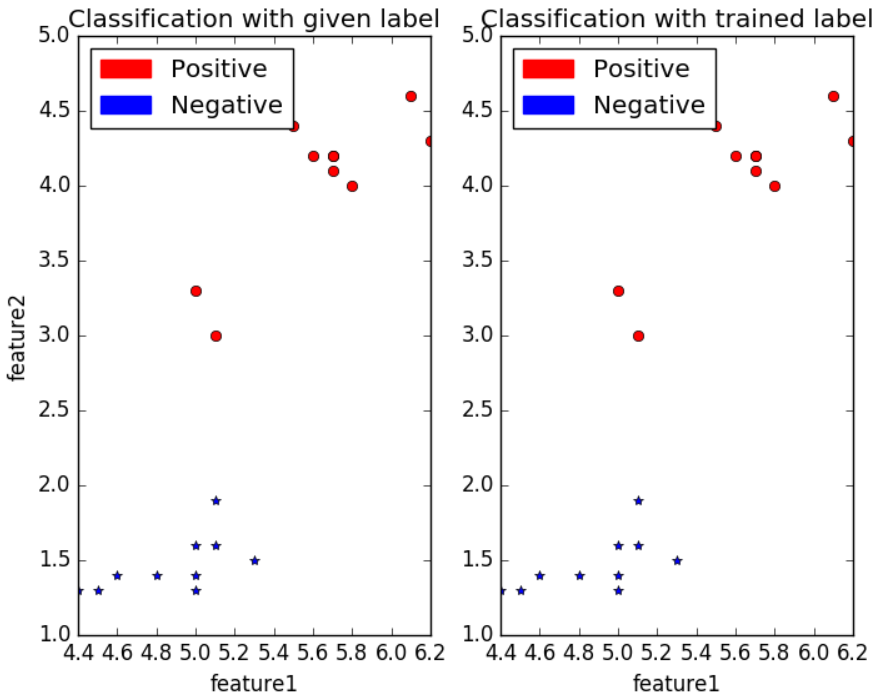

图 1: kNN 实验结果

从图中抑或数值计算上我们得到的测试集上的 Accuracy 都等于 100%，所以可以得到结论我们的 kNN 是有效的。

意外地，我们发现在已给的测试集条件下，k 即使小到 1 得出的 Accuracy 也仍然是 100%。经过分析后我们并没有发现代码中的问题，应该是测试集本身线性可分性太过明显，并且数量台下，导致分类器出现了与 k 的取值无关的现象。而且对于训练集来说，本身的线性可分性也非常明显，没有出现奇异点，即使我们把训练集和测试集交换位置仍然得到了 Accuracy=100% 的结果。所以我们这里就暂不讨论 k 的取值与结果的关系。

三. Perceptron 实现细节

我们同样利用 Python 实现了 Perceptron 算法，简要实现方法如下：

(a) 首先通过 Numpy 库的 `loadtxt` 方法将训练集读入，并保存在数组变量 `train_matrix` 中。同样将测试集保存在变量 `test_matrix` 中；

(b) 定义学习权重参数向量 `weight`，和学习率 `learning_rate`，利用 `train_matrix` 中数据对权重进行学习；

(c) 利用学习到的向量 `weight` 对测试集中数据进行分类，并给出相应标签。

四. Perceptron 实验结果

我们利用给出的训练集训练模型后，在测试集上进行了实验，可视化后结果如图 2。

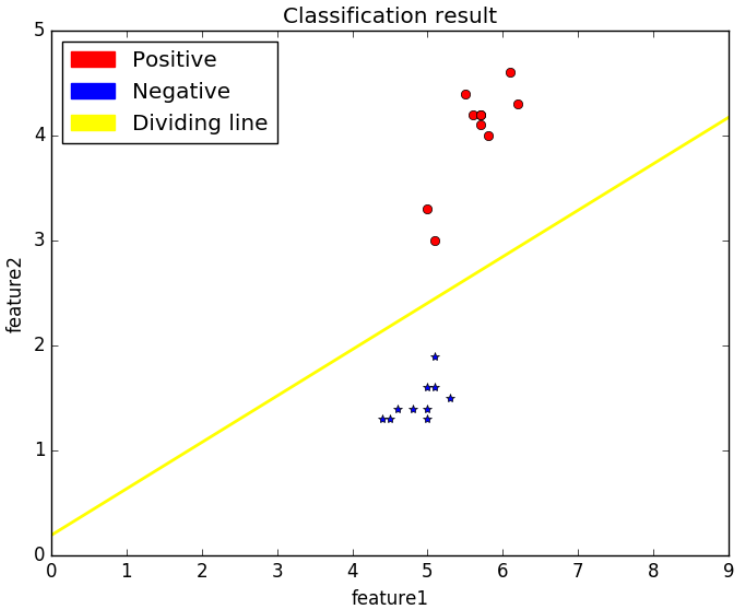

图 2: Perceptron 实验结果

从结果可以看出，生成的权重向量很好地分开了两类点，并且从数值上也可以得到在给出的测试集上的 Accuracy 达到了 100%。同样因为测试集过小、训练集线性可分性较强等原因，测试集的数据分类效果过好。我们发现学习率的变化虽然会对 w_0、w_1 和 w_2 数值产生影响，但是基本可以保证 Accuracy 稳定在 100%。

表 1: 权重、Accuracy 与准确率的关系

学习率	w_0	w_1	w_2	$\lvert w_1/w_2 \rvert$	Accuracy
0	0	0	0	null	0.5
0.001	-0.001	-0.0023	0.0052	0.4423	1.0
0.01	-0.01	-0.023	0.052	0.4423	1.0
0.1	-0.1	-0.23	0.52	0.4423	1.0
0.2	-0.2	-0.46	1.04	0.4423	1.0
0.3	-0.3	-0.69	1.56	0.4423	1.0
0.4	-0.4	-0.92	2.08	0.4423	1.0
0.5	-0.5	-1.15	2.6	0.4423	1.0
0.6	-0.6	-1.38	3.12	0.4423	1.0
0.7	-0.7	-1.61	3.64	0.4423	1.0

从表 1 基本可以看出一个有趣的现象：虽然学习率在变化的同时会影响 w_0、w_1 和 w_2 数值上的变化，但是 $w_0 : w_1 : w_2$ 的数值却稳定在一个固定的数值。看起来这并不直观，但是实际上，因为 $y = w_0 + w_1x_1 + w_2x_2$，且更新函数中 `weight`$= \alpha \cdot y \cdot x$，更重要的是，我们设置的 `weight` 初始值为 `[0,0,0]`，所以一旦训练集中数据顺序固定，w_0、w_1 和 w_2 的值的变化都是同时成比例地变化，这也就导致了实际上学习率 α 的具体取值对 $w_0 : w_1 : w_2$ 的值并无影响。

但是需要注意的是，在给出的原始训练集中 `label` 是 1 和-1 聚集为两类出现的，这样会导致训练效果失败，因为如果先训练 `label` 为 1 的样本，然后再训练 `label` 为-1 的样本，最后的 `weight` 向量会更偏向分类为-1 的样本，从而导致对 1 的样本分类效果变差。所以我们的处理办法是通过打乱训练集中数据的顺序，使得训练时不会因为样本的出现顺序导致 `weight` 向量出现对某一个 `label` 过拟合而另一个 `label` 高偏差的情况。

五. kNN 和 Perceptron 比较

kNN 算法适用场景

- 主要用于非参数的分类问题；

- 易于扩展，无论是增加类别数量还是训练数据，不需要重新训练；
- 对于噪声和异常点容忍性较高，离测试点较远的训练数据对分类结果影响很小；
- 不仅适合线性分类，甚至对于非线性分类也适用，只要样本向量满足 contiguity hypothesis。

kNN 算法缺点

- 属于 Lazy learning，测试时才进行学习，所以需要一直存储训练集，耗费存储空间；
- 对于样本不平衡问题较敏感 (即有些类别的样本数量很多，而其它样本的数量很少)；
- 测试时需要求出测试数据与每一个训练样本的距离并排序找出距离最小的 k 个，时间复杂度是 O(kN)，对内存要求较高；
- 在求解测试数据与样本之间距离时，假定每一个属性的权重相同 (实际上不同特征可能贡献度不同)；
- k 值选择不当对结果影响也较大，比如如果 k 只取 1, 则对异常点和噪声极为敏感，而当 k 过大时，又会使得分类更偏向于分类为样本中数量占优的类别。

Perceptron 算法适用场景

- 主要用于线性可分的分类问题；
- 实现容易且训练方便，并且可以接受新的数据进行不断训练，适合在线学习的场景；
- 训练完成后只存储一组权值，进行对为标签样本的分类预测时计算复杂度低。

Perceptron 算法缺点

- 模型过于简单，仅适合线性可分的分类场景；
- 训练效果还会受到训练集本身分布特征影响，已标记样本的混合熵越大明显 Perceptron 算法的效果会越好,具体原因我们上面也已经提到过,具体现象我们可以从图 3中看出，因为后半部分训练集都是标签为 1 的数据，所以即使前面-1 的样本对权重已经进行了训练，训练进入后半段后权重又会迅速转向将样本分类为 1 的情况，导致最后的分类效果十分不理想。然而对于 kNN 来说则没有这种问题；
- 模型需要进行对学习率调参，在较大规模训练集的情况下，调参以避免无法收敛会带来较大的计算代价

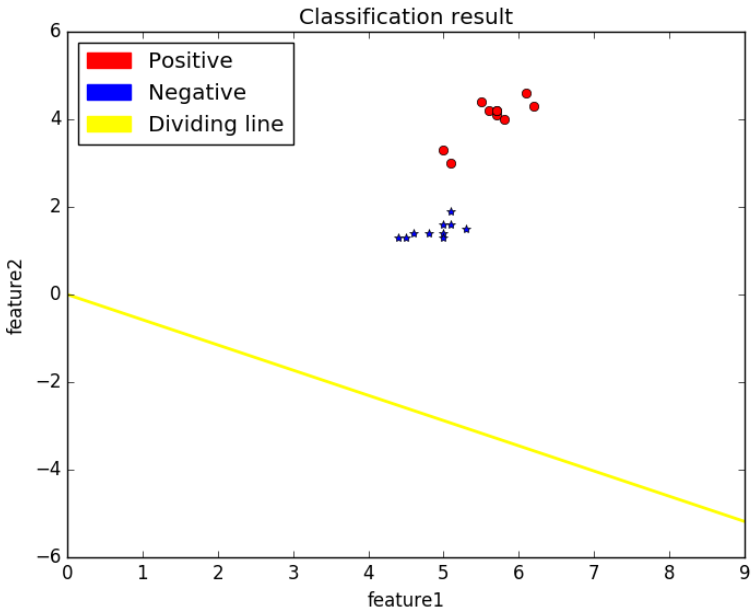

图 3: 当训练集不做随机混合时 Perceptron 实验结果