



# 统计机器学习实验报告

实验名称：多种模型的葡萄酒品种分类任务

姓名：	王恒
学院：	数学与统计学院
专业：	计算数学
学号：	220220934161

2023 年 6 月 11 日

### 摘要

本文中, 我们实现了感知机的原型算法和对偶算法, 并完成了三点数据分类的题目. 同时, 我们使用对偶算法对 MNIST 数据集进行分类. 其中, 我们首先考虑将以  $\{y : y = 0, 1, \dots, 9\}$  为标签的多分类问题转化为以

$$\hat{y} = \begin{cases} 1, & y < 5, \\ -1, & y \geq 5. \end{cases}$$

为标签的二分类问题, 之后应用对偶算法分类. 其次, 我们实现了 k-近邻的 kd-树的构造算法及搜索算法, 并在 MNIST 数据集上进行实验. 其中, 我们对原始 MNIST 图片 (0-255 的图片上) 添加均匀分布  $U(0, 10)$  的整数随机噪声 (消除数据稀疏性, 不破坏图片的主要特征), 基于添加噪声后的  $28 * 28$  维数据构造 kd-树, 以保证 kd-树是好的. 最后我们对实验结果进行了分析, 实验结果证明我们复现的算法是正确的, 并达到了预期效果. 我们将本实验报告的所有内容开源:

<https://github.com/WANGH950/Statistical-Machine-Learning/tree/main/2ND>.

**关键词:** 感知机, k-近邻, kd-树, MNIST

# 目录

<b>1</b>	<b>实验代码</b>	<b>1</b>
1.1	感知机学习算法 . . . . .	1
1.2	k-近邻算法 . . . . .	3
<b>2</b>	<b>实验结果分析</b>	<b>6</b>
2.1	感知机学习算法结果分析 . . . . .	7
2.2	k-近邻算法结果分析 . . . . .	8

# 1 实验代码

## 1.1 感知机学习算法

Listing 1: 感知机原型算法实现

```
1 class Perception():
2     def __init__(self, dim) -> None:
3         # 构造函数
4         # dim: 特征维度
5         # w: 权重
6         # b: 偏置项
7
8         self.dim = dim
9         self.w = np.zeros([dim])
10        self.b = 0
11
12    def train(self, data_set, epoch, learning_rate):
13        # 训练模型
14
15        for i in range(epoch):
16            for (x,y) in data_set:
17                if self.predict(x)*y <= 0:
18                    self.w = self.w + learning_rate*x*y
19                    self.b = self.b + learning_rate*y
20
21            # 计算准确率
22            acc = self.accuracy(data_set)
23            print('epoch: ',i+1, 'accuracy: ', acc)
24            # 早停条件
25            if acc == 1:
26                break
27            print('Trining complete.')
28
29    def predict(self, x):
30        # 预测
31
32        return np.sign(np.dot(self.w,x) + self.b)
```

```
33     def accuracy(self, data_set):
34         # 计算精度
35
36         acc = 0
37         for (x,y) in data_set:
38             if self.predict(x)*y > 0:
39                 acc += 1
40         return acc/len(data_set)
```

Listing 2: 感知机对偶算法实现

```
1 class PerceptionDual():
2     def __init__(self, data) -> None:
3         # 构造函数
4         # data:      训练数据
5
6         self.data = data
7         self.N = len(data)
8         self.x = np.array([xx for (xx,_) in data])
9         self.y = np.array([yy for (_,yy) in data])
10        self.alpha = np.zeros([self.N])
11        self.b = 0
12
13    def train(self, epoch, learning_rate):
14        # 训练模型
15
16        for i in range(epoch):
17            for j in range(len(self.data)):
18                if self.predict(self.data[j][0])*self.data[j]
19                    ][1] <= 0:
20                    self.alpha[j] = self.alpha[j] +
21                        learning_rate
22                    self.b = self.b + learning_rate*self.data[j]
23                        ][1]
24
25                # 计算准确率
26                acc = self.accuracy(self.data)
27                print('epoch: ',i+1, 'accuracy: ', acc)
28
29                # 早停条件
```

```
25         if acc == 1:
26             break
27         print('Trining complete.')
28
29     def predict(self, x):
30         # 预测
31         return np.sign(np.dot(np.dot(self.x,x),self.alpha*self.
32                               y) + self.b)
33
34     def accuracy(self, data_set):
35         # 计算精度
36
37         acc = 0
38         for (x,y) in data_set:
39             if self.predict(x)*y > 0:
40                 acc += 1
41         return acc/len(data_set)
```

## 1.2 k-近邻算法

Listing 3: kd-树构造算法和搜索算法实现

```
1     class Node():
2     def __init__(self, value, data, label) -> None:
3         # 构造函数
4         # value:         节点的划分超平面参数
5         # data:         落在超平面上的数据点
6         # label:        落在超平面上的数据点对应的标签
7
8         self.value = value
9         self.data = data
10        self.label = label
11        self.left = None
12        self.right = None
13
14    def set_left(self, node):
15        # 设置左子节点
16
```

```
17         if node != None:
18             self.left = node
19
20     def set_right(self, node):
21         # 设置右子节点
22
23         if node != None:
24             self.right = node
25
26 class KDTree():
27     def __init__(self) -> None:
28         # 构造函数
29         # 用于存储KDTree, 支持直接实例化对象时直接输入一个kd-树
30         self.root = None
31
32     def create(self, data, label, j = 0):
33         # 递归构造平衡KD树
34
35         num, k = data.shape
36         if num == 0:
37             return None
38         else:
39             l = j % k
40             ind_sorted = np.argsort(data[:,l])
41             ind_median = ind_sorted[num//2]
42             value_ = int(np.median(data[ind_median,l]))
43             data_ = data[data[:,l]==value_]
44             label_ = label[data[:,l]==value_]
45             node = Node(
46                 value=value_,
47                 data=data_,
48                 label=label_
49             )
50             node.set_left(
51                 self.create(
52                     data=data[data[:,l]<value_],
53                     label=label[data[:,l]<value_],
```

```
54         j=j+1
55     )
56 )
57 node.set_right(
58     self.create(
59         data=data[data[:,1]>value_],
60         label=label[data[:,1]>value_],
61         j=j+1
62     )
63 )
64 if j == 0:
65     self.root = node
66 else:
67     return node
68
69 def search(self, x, j = 0, node = None):
70     # 递归搜索KD树
71
72     if self.root == None:
73         print("You haven't created a KDTree yet.")
74         return None
75     if j == 0:
76         node = self.root
77     k = x.shape[0]
78     l = j % k
79     # 叶子节点停止条件
80     if self.is_leaf(node):
81         distance = np.linalg.norm(x-node.data,2,1)
82         index = np.argmin(distance)
83         return node.data[index], node.label[index]
84     else:
85         # 计算当前节点中的最近数据点
86         distance = np.linalg.norm(x-node.data,2,1)
87         min_distance = np.min(distance)
88         index = np.argmin(distance)
89         nearest = node.data[index]
90         label = node.label[index]
```



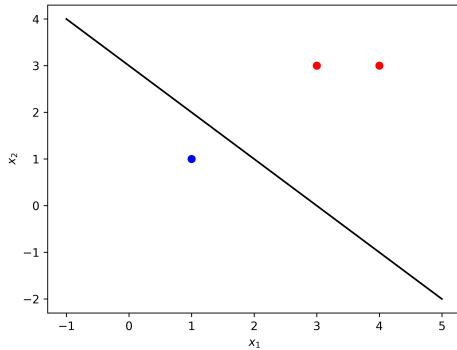
```
91         # 递归计算子节点的最近数据点，并比较
92         if x[l] < node.value and node.left != None:
93             nearest_, label_ = self.search(
94                 x = x,
95                 j = j+1,
96                 node = node.left
97             )
98             if np.linalg.norm(x-nearest_,2) < min_distance:
99                 nearest = nearest_
100                 label = label_
101         elif x[l] > node.value and node.right != None:
102             nearest_, label_ = self.search(
103                 x = x,
104                 j = j+1,
105                 node = node.right
106             )
107             if np.linalg.norm(x-nearest_,2) < min_distance:
108                 nearest = nearest_
109                 label = label_
110         return nearest, label
111
112     def is_leaf(self, node: Node):
113         # 判断是否是叶子节点
114
115         if node.left != None or node.right != None:
116             return False
117         else:
118             return True
```

## 2 实验结果分析

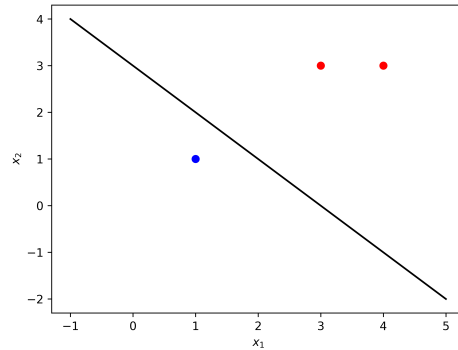
MNIST 数据集通过 torchvision 加载,所有算法基于 python 的 numpy 实现,数据预处理的一部分地方借助 pytorch 实现. 细节请参考:

<https://github.com/WANGH950/Statistical-Machine-Learning/tree/main/1ST>

## 2.1 感知机学习算法结果分析



(a) 感知机原型算法训练结果



(b) 感知机对偶算法训练结果

图 1: 感知机模型训练结果, 其中红点表示正类, 蓝点表示负类, 黑色实线表示感知机分割超平面.

以 0.01 的学习率, 感知机原型算法在题目给定的数据上, 经过 5 次训练后收敛, 得到结果. 以 0.01 的学习率, 感知机对偶算法在题目给定的数据上, 经过 5 次训练后收敛, 得到结果. (图 1)

Listing 4: MNIST 数据加载

```
1 import torchvision
2 data=torchvision.datasets.MNIST(
3     root='MNIST',
4     train=True,
5     transform=torchvision.transforms.ToTensor(),
6     download=True
7 )
```

Listing 5: MNIST 数据预处理

```
1 train_data = data.train_data
2 train_label = data.train_labels
3 # 转化为二分类
4 data_set = []
5 for i in range(train_data.shape[0]):
6     if train_label[i] < 5:
7         y = 1
8     else:
```

```
9         y = -1
10     data_set.append(
11         (train_data[i].reshape([28*2]).numpy()/255,y)
12     )
```

Listing 6: 使用对偶感知机算法处理 MNIST 数据

```
1 model = PerceptionDual(data_set[:1000])
2 model.train(
3     epoch=30,
4     learning_rate=0.001
5 )
```

如 Listing 6 所示, 我们使用 1,000 条数据, 在 0.001 的学习率下训练了 30 次, 最后在训练数据上得到了 0.923 的准确率.

## 2.2 k-近邻算法结果分析

Listing 7: MNIST 数据预处理

```
1 data_num,dimx,dimy = train_data.shape
2 data = train_data.reshape([data_num,dimx*dimy])+torch.randint
   (0,10,[data_num,dimx*dimy])
3 label = train_label.unsqueeze(-1)
```

如 Listing 7 所示, 由于手写数字是稀疏矩阵, 我们通过对其添加随机噪声以保证构造的 kd-树是好的 (不加噪声会导致数据都分布在一个节点上). 这里, 我们添加  $U(0, 10)$  的均匀整数噪声.

Listing 8: 构造 kd-树

```
1 model = KDTree()
2 tree = model.create(
3     data=data[:30000],
4     label=label[:30000]
5 )
```

如 Listing 8 所示, 我们使用前 30,000 条数据构造 kd-树. 我们选取第 40,000 条数据进行预测, 图 1 展示了预测结果和真实结果.

同时, 我们还使用后 30,000 条数据作为测试集进行测试 (listing 9), 得到了 93.223% 的准确率. 结果表明, 我们的算法实现准确.

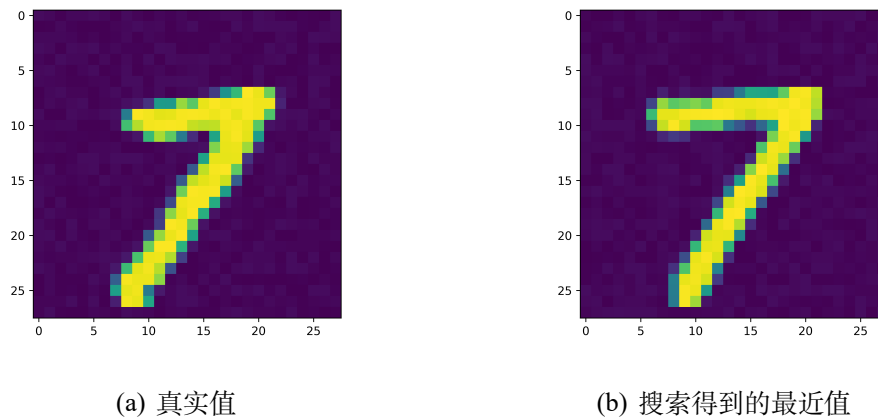


图 2: kd-树搜索结果.

Listing 9: 使用 kd-树分类测试数据

```
1 k = 30000
2 acc = 0
3 data_test = data[-k:]
4 label_test = label[-k:]
5 for i in range(k):
6     data_rel = data_test[i]
7     label_rel = label_test[i]
8     data_pre, label_pre = model.search(data_rel)
9     if label_pre == label_rel:
10         acc += 1
11 acc /= k
```

所有实验结果和运行效率以源码为准:

<https://github.com/WANGH950/Statistical-Machine-Learning/tree/main/1ST>