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, 我们实现了朴素贝叶斯、逻辑斯谛回归、线性支持向量机、基于高斯核函数的非线性支持向量机、多层非线性感知机 (神经网络) 模型及其相应的学习算法, 并用以上模型对葡萄酒品种进行分类, 六种模型分别得到了 95.83%, 97.22%, 94.44%, 95.83%, 97.22% 的准确率.

具体的, 我们将数据集按照 $6:4$ 的比例划分为训练集和测试集, 并进行归一化. 其中, 由于某些特征是连续型数据, 我们在实现朴素贝叶斯模型时, 将所有特征归一化到 $[-1,1]$, 并将其划分为固定的小区间, 模型中的概率均为特征值所处区间的概率. 对于其他四种回归模型, 我们在输入数据后先采用了批量归一化技术, 自动学习归一化参数, 相应的, 均采用小批量随机梯度下降算法在训练集上进行优化, 在测试集上测试准确率. 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: 朴素贝叶斯, 逻辑斯谛回归, 支持向量机, 神经网络, 葡萄酒品种分类

目录

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

```
class Perception():
    def __init__(self, dim) -> None:
        # 构造函数
        # dim:   特征维度
        # w:     权重
        # b:     偏置项

        self.dim = dim
        self.w = np.zeros([dim])
        self.b = 0

    def train(self, data_set, epoch, learning_rate):
        # 训练模型

        for i in range(epoch):
            for (x,y) in data_set:
                if self.predict(x)*y <= 0:
                    self.w = self.w + learning_rate*x*y
                    self.b = self.b + learning_rate*y
            # 计算准确率
            acc = self.accuracy(data_set)
            print('epoch: ',i+1, 'accuracy: ', acc)
            # 早停条件
            if acc == 1:
                break
        print('Trining complete.')

    def predict(self, x):
        # 预测

        return np.sign(np.dot(self.w,x) + self.b)

```

```
    def accuracy(self, data_set):
        # 计算精度

        acc = 0
        for (x,y) in data_set:
            if self.predict(x)*y > 0:
                acc += 1
        return acc/len(data_set)
```

Listing 2: 感知机对偶算法实现

```
class PerceptionDual():
    def __init__(self, data) -> None:
        # 构造函数
        # data:     训练数据

        self.data = data
        self.N = len(data)
        self.x = np.array([xx for (xx,_) in data])
        self.y = np.array([yy for (_,yy) in data])
        self.alpha = np.zeros([self.N])
        self.b = 0

    def train(self, epoch, learning_rate):
        # 训练模型

        for i in range(epoch):
            for j in range(len(self.data)):
                if self.predict(self.data[j][0])*self.data[j][1] <= 0:
                    self.alpha[j] = self.alpha[j] + learning_rate
                    self.b = self.b + learning_rate*self.data[j][1]
            # 计算准确率
            acc = self.accuracy(self.data)
            print('epoch: ',i+1, 'accuracy: ', acc)
            # 早停条件
```

```
            if acc == 1:
                break
        print('Trining complete.')

    def predict(self, x):
        # 预测
        return np.sign(np.dot(np.dot(self.x,x),self.alpha*self.y) + self.b)

    def accuracy(self, data_set):
        # 计算精度

        acc = 0
        for (x,y) in data_set:
            if self.predict(x)*y > 0:
                acc += 1
        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

```
        class Node():
    def __init__(self, value, data, label) -> None:
        # 构造函数
        # value:        节点的划分超平面参数
        # data:         落在超平面上的数据点
        # label:        落在超平面上的数据点对应的标签

        self.value = value
        self.data = data
        self.label = label
        self.left = None
        self.right = None

    def set_left(self, node):
        # 设置左子节点

```

```
        if node != None:
            self.left = node

    def set_right(self, node):
        # 设置右子节点

        if node != None:
            self.right = node

class KDTree():
    def __init__(self) -> None:
        # 构造函数
        # 用于存储KDTree，支持直接实例化对象时直接输入一个kd-树
        self.root = None

    def create(self, data, label, j = 0):
        # 递归构造平衡KD树

        num, k = data.shape
        if num == 0:
            return None
        else:
            l = j % k
            ind_sorted = np.argsort(data[:,l])
            ind_median = ind_sorted[num//2]
            value_ = int(np.median(data[ind_median,l]))
            data_ = data[data[:,l]==value_]
            label_ = label[data[:,l]==value_]
            node = Node(
                value=value_,
                data=data_,
                label=label_
            )
            node.set_left(
                self.create(
                    data=data[data[:,l]<value_],
                    label=label[data[:,l]<value_],
```

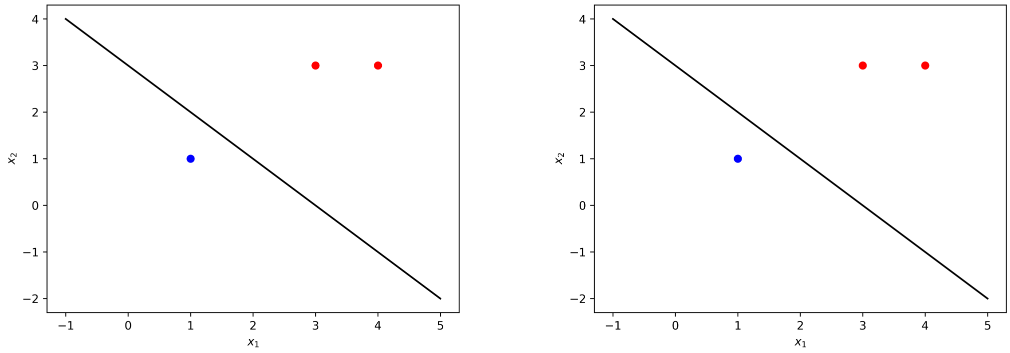
```
                    j=j+1
                )
            )
            node.set_right(
                self.create(
                    data=data[data[:,l]>value_],
                    label=label[data[:,l]>value_],
                    j=j+1
                )
            )
            if j == 0:
                self.root = node
            else:
                return node

    def search(self, x, j = 0, node = None):
        # 递归搜索KD树

        if self.root == None:
            print("You haven't created a KDTree yet.")
            return None
        if j == 0:
            node = self.root
        k = x.shape[0]
        l = j % k
        # 叶子节点停止条件
        if self.is_leaf(node):
            distance = np.linalg.norm(x-node.data,2,1)
            index = np.argmin(distance)
            return node.data[index], node.label[index]
        else:
            # 计算当前节点中的最近数据点
            distance = np.linalg.norm(x-node.data,2,1)
            min_distance = np.min(distance)
            index = np.argmin(distance)
            nearest = node.data[index]
            label = node.label[index]
```

```
            # 递归计算子节点的最近数据点，并比较
            if x[l] < node.value and node.left != None:
                nearest_, label_ = self.search(
                    x = x,
                    j = j+1,
                    node = node.left
                )
                if np.linalg.norm(x-nearest_,2) < min_distance:
                    nearest = nearest_
                    label = label_
            elif x[l] > node.value and node.right != None:
                nearest_, label_ = self.search(
                    x = x,
                    j = j+1,
                    node = node.right
                )
                if np.linalg.norm(x-nearest_,2) < min_distance:
                    nearest = nearest_
                    label = label_
            return nearest, label

    def is_leaf(self, node: Node):
        # 判断是否是叶子节点

        if node.left != None or node.right != None:
            return False
        else:
            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

```
import torchvision
data=torchvision.datasets.MNIST(
    root='MNIST',
    train=True,
    transform=torchvision.transforms.ToTensor(),
    download=True
)
```

Listing 5: MNIST 数据预处理

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

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

Listing 6: 使用对偶感知机算法处理 MNIST 数据

```
model = PerceptionDual(data_set[:1000])
model.train(
    epoch=30,
    learning_rate=0.001
)
```

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

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

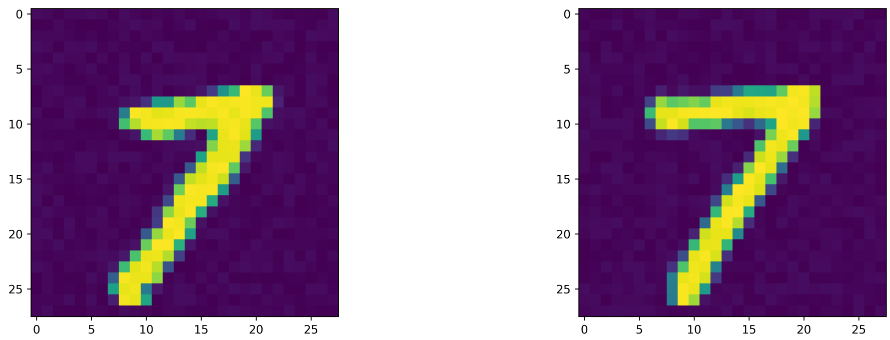
如 Listing 7 所示, 由于手写数字是稀疏矩阵, 我们通过对其添加随机噪声以保证构造的 kd-树是好的 (不加噪声会导致数据都分布在一个节点上). 这里, 我们添加 $U(0,10)$ 的均匀整数噪声.

Listing 8: 构造 kd-树

```
model = KDTree()
tree = model.create(
    data=data[:30000],
    label=label[:30000]
)
```

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.



(a) 真实值 (b) 搜索得到的最近值

图 2: kd-树搜索结果.

Listing 9: 使用 kd-树分类测试数据

```
k = 30000
acc = 0
data_test = data[-k:]
label_test = label[-k:]
for i in range(k):
    data_rel = data_test[i]
    label_rel = label_test[i]
    data_pre,label_pre = model.search(data_rel)
    if label_pre == label_rel:
        acc += 1
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

A 模型及算法源码

A.1 朴素贝叶斯模型及其训练算法

Listing 10: 朴素贝叶斯模型及其训练算法实现

```
# 定义模型
class NaiveBayes(nn.Module):
    def __init__(self, n, full_labels, S, lamb) -> None:
        super(NaiveBayes,self).__init__()
        # 归一化参数
        self.max = None
        self.min = None
        self.n = n # 特征数量
        self.full_labels = full_labels # 所有标签
        self.K = len(full_labels) # 标签数量
        self.lamb = lamb # 贝叶斯估计参数lambda
        self.S = S # 每个特征分划区间数, 这里默认都为S
        self.cond_prob = torch.zeros([self.K,self.n,S]) # 条件
           概率
        self.pre_prob = torch.zeros([self.K]) # 先验概率

    def forward(self, features):
        B,n = features.shape
        assert n == self.n
        post_prob = torch.ones([B,1])*self.pre_prob
        # 归一化
        features = (features - self.min) / (self.max - self.min
           ) * 2 - 1
        delta_x = 2 / (self.S - 2)
        for i in range(B):
            for j in range(self.K):
                for k in range(n):
                    # (-\infty,-1)和(1,\infty)的概率
                    if features[i,k] < -1: post_prob[i,j] *=
                       self.cond_prob[j,k,0]
                    elif features[i,k] >= 1: post_prob[i,j] *=
                       self.cond_prob[j,k,-1]
```

```
                    else:
                        for h in range(self.S-2):
                            l = -1 + h * delta_x
                            r = l + delta_x
                            if features[i,k] >= l and features[i,k] < r:
                                post_prob[i,j] *= self.cond_prob[j,k,h+1]
                                break
        return self.full_labels[torch.argmax(post_prob,dim=1)]

    def fit(self, train_data):
        # 计算先验概率
        N,_ = train_data.shape
        self.max = torch.max(train_data[:,:-1],dim=0).values
        self.min = torch.min(train_data[:,:-1],dim=0).values
        train_data[:,:-1] = (train_data[:,:-1] - self.min) / (self.max - self.min) * 2 - 1
        features = train_data[:,:-1] # 特征
        labels = train_data[:,-1:].int() # 标签
        delta_x = 2 / (self.S - 2)
        for i in range(self.K):
            labels_i = labels == self.full_labels[i]
            self.pre_prob[i] = (labels_i.sum() + self.lamb) / (N + self.K*self.lamb)
            for j in range(self.n):
                self.cond_prob[i,j,0] = 1 / self.S
                for k in range(self.S-1):
                    l = -1 + k * delta_x
                    r = l + delta_x
                    features_ij = features[labels_i[:,0],j]
                    features_ijk = features_ij[(features_ij>=l)*(features_ij<r)]
                    self.cond_prob[i,j,k+1] = (features_ijk.shape[0] + self.lamb) / (labels_i.sum() + self.S*self.lamb)
        return self.pre_prob, self.cond_prob
```

A.2 Logistic 模型及相应学习算法

Listing 11: 逻辑斯谛回归模型及损失函数和训练方法

```python
# 定义模型
class Logistic(nn.Module):
    def __init__(self,feature_num,class_num) -> None:
        super(Logistic,self).__init__()
        self.feature_num = feature_num
        self.class_num = class_num
        self.linear = nn.Sequential(
            nn.BatchNorm1d(feature_num), # 批量归一化
            nn.Linear(feature_num,class_num-1,bias=False)
        )

    def forward(self, x):
        B,d = x.shape
        assert d == self.feature_num and B > 0
        y = torch.cat([torch.exp(self.linear(x)),torch.ones([B,1])],dim=1)
        y = y / torch.sum(y,dim=1,keepdim=True)
        return y

# 负对数似然损失函数
class NLLLoss(nn.Module):
    def __init__(self) -> None:
        super().__init__()

    def forward(self,y_pre,y_rel):
        assert y_pre.shape == y_rel.shape
        loss = -torch.log(y_pre)*y_rel
        return torch.sum(loss)

# 定义模型训练函数
def train(model, data_set, batch_size, epoch = 1000, learning_rate = 1e-3):
    N,_ = data_set.shape
    criterion = NLLLoss()
```

```
    optim = torch.optim.SGD(model.parameters(),learning_rate, momentum=0) # 动量设置为0

    start = time.time()
    loss_values = torch.zeros(epoch)
    for i in range(epoch):
        model.train()
        optim.zero_grad()
        index = torch.randint(0,N,[batch_size]) # 随机选取batch条数据
        data_i = data_set[index,:]
        x_i = data_i[:,:-1].to(torch.float32)
        y_i = nn.functional.one_hot(data_i[:,-1].to(torch.int64),num_classes=model.class_num)
        outputs = model(x_i)
        loss = criterion(outputs,y_i)
        loss.backward()
        optim.step()

        model.eval()
        loss_values[i] = loss.item()

        print('\r%5d/{}|{}{}|{:.2f}s  [Loss: %e]'.format(
            epoch,
            "#"*int((i+1)/epoch*50),
            " "*(50-int((i+1)/epoch*50)),
            time.time() - start) %
            (i+1,
            loss_values[i]), end = ' ', flush=True)
    print("\nTraining has been completed.")
    return loss_values
```

A.3 支持向量机

Listing 12: 线性支持向量机

```
# 定义模型
class SVMLinear(nn.Module):
```

```
    def __init__(self,feature_num) -> None:
        super(SVMLinear,self).__init__()
        self.feature_num = feature_num
        self.batch_norm = nn.BatchNorm1d(feature_num) # 特征批量归一化，学习归一化参数
        self.linear = nn.Linear(feature_num,1)

    def forward(self, x, signed = True):
        B,d = x.shape
        assert B > 0 and d == self.feature_num
        normed = self.batch_norm(x)
        outputs = self.linear(normed)
        if signed:
            return torch.sign(outputs)
        else:
            return outputs

# 合页损失函数
class HingeLoss(nn.Module):
    def __init__(self,lamb) -> None:
        super(HingeLoss,self).__init__()
        self.lamb = lamb

    def forward(self,res_pre_linear_values,res_rel,parameters):
        return torch.sum(torch.relu(1 - res_rel*res_pre_linear_values)) + self.lamb*torch.norm(parameters)**2

# 定义训练函数
def train_svm(model, data_set, batch_size, lamb=1, epoch = 1000, learning_rate = 1e-3):
    N,_ = data_set.shape
    criterion = HingeLoss(lamb)
    optim = torch.optim.SGD(model.parameters(),learning_rate, momentum=0) # 动量设置为0

    start = time.time()
```

```
    loss_values = torch.zeros(epoch)
    for i in range(epoch):
        model.train()
        optim.zero_grad()
        index = torch.randint(0,N,[batch_size]) # 随机选取batch条数据
        data_i = data_set[index,:]
        x_i = data_i[:,:-1].to(torch.float32)
        y_i = data_i[:,-1:].to(torch.int32)
        outputs = model(x_i,signed=False)
        loss = criterion(outputs,y_i,model.linear.weight)
        loss.backward()
        optim.step()

        model.eval()
        loss_values[i] = loss.item()

        print('\r%5d/{}|{}{}|{:.2f}s  [Loss: %e]'.format(
            epoch,
            "#"*int((i+1)/epoch*50),
            " "*(50-int((i+1)/epoch*50)),
            time.time() - start) %
            (i+1,
            loss_values[i]), end = ' ', flush=True)
    print("\nTraining has been completed.")
    return loss_values
```

Listing 13: 非线性支持向量机

```
# 高斯核非线性SVM
class SVMNonLinear(nn.Module):
    def __init__(self,feature_num,kernel_data) -> None:
        super(SVMNonLinear,self).__init__()
        self.feature_num = feature_num
        self.kernel_features = kernel_data[:,:-1].to(torch.float32)
        self.kernel_labels = kernel_data[:,-1].to(torch.int32)
        self.batch_norm = nn.BatchNorm1d(feature_num)
```

```
        self.linear = nn.Linear(feature_num+kernel_data.shape
           [0],1)
        self.kernel_params = nn.Parameter(torch.rand(1)+5,
           requires_grad=True) # 核函数标准差

    def forward(self, x, signed = True):
        B,d = x.shape
        assert B > 0 and d == self.feature_num
        # 归一化特征
        normed_data = self.batch_norm(x)
        normed_kernel_data = self.batch_norm(self.
           kernel_features)
        kernel_features = torch.exp(-torch.mm(normed_data,
           normed_kernel_data.T) / self.kernel_params**2) *
           self.kernel_labels
        outputs = self.linear(torch.cat([normed_data,
           kernel_features],dim=1)) # 核方法特征和原特征结合
        if signed:
            return torch.sign(outputs)
        else:
            return outputs
```

A.4 全连接神经网络

Listing 14: 全连接神经网络模型实现

```
class MLP(nn.Module):
def __init__(self,feature_num,class_num,hidden_dim=20,layer_num
   =2) -> None:
    super(MLP,self).__init__()
    self.feature_num = feature_num
    self.class_num = class_num
    self.hidden_dim = hidden_dim
    self.layer_num = layer_num
    self.model = nn.Sequential(
        OrderedDict(
            [("input_layer",
                nn.Sequential(
```

```
                    nn.BatchNorm1d(feature_num),
                    nn.Linear(feature_num,hidden_dim),
                    nn.Tanh()
                ))] +
            [("hidden_layer_"+str(i+1),
                nn.Sequential(
                    nn.Linear(hidden_dim,hidden_dim),
                    nn.Tanh()
                )) for i in range(layer_num-1)] +
            [("output_layer",
                nn.Sequential(
                    nn.Linear(hidden_dim,class_num),
                    nn.Softmax(dim=1)
                ))]
        )
    )

# 前向传播
def forward(self,x):
    return self.model(x)
```