西南交通大學

《机器学习 B》

课程设计报告

报告题目：分类算法对比学习研究

年　　级：2021 级

姓　　名：

学　　号：

二〇二四年六月

分类算法对比学习研究

一、模型介绍

朴素贝叶斯：

朴素贝叶斯分类器（naïve Bayes classifier）采用了“属性条件独立性假设”：对已知类别，假设所有属性相互独立，换而言之，假设每个属性独立地对分类结果发生影响。

基于贝叶斯定理，$P(c|\boldsymbol{x})$可写为：

$$P(c|x)=\frac{P(c)P(\boldsymbol{x}|c)}{P(\boldsymbol{x})}$$

基于属性条件独立性假设，可重写为：

$$P(c|\boldsymbol{x})=\frac{P(c)}{P(\boldsymbol{x})}\prod_{i=1}^{d}P(x_i|c)$$

其中，d为属性数目，x_i为x在第i个属性上的取值。

对于所有类别来说，$P(\boldsymbol{x})$相同，因此，基于此的贝叶斯判定准则有

$$h_{nb}(\boldsymbol{x})=\underset{c\in y}{\operatorname{argmax}}\prod_{i=1}^{d}P(x_i|c)$$

这就是朴素贝叶斯分类器的表达式。

令D_c表示训练集D中第c类样本组成的集合，若有充足的独立同分布样本，则可容易地估计出类先验概率

$$P(c)=\frac{|D_c|}{|D|}$$

对离散属性而言，令D_{c,x_i}表示D_c中在第i个属性上取值为x_i的样本组成的集合，则条件概率$P(x_i|c)$可估计为：

$$P(x_i|c)=\frac{|D_{c,x_i}|}{|D_c|}$$

优点：算法原理简单，易于理解和实现。与其他学习算法相比，朴素贝叶斯需要较少的训练数据即可达到较好的分类效果。在特征条件独立假设下，分类性能通常很好，尤其是在特征数量较多的情况下。

缺点：特征独立性的假设在现实中往往不成立，可能会影响分类的准确性。如果训练数据中包含噪声，朴素贝叶斯分类器的性能可能会受到影响。

在本训练代码中不仅给出了朴素贝叶斯分类器的代码，同时也使用了 Scikit-learn GaussianNB，GaussianNB 是 Scikit-learn 库中实现的高斯朴素贝叶斯分类器，用于与实现的朴素贝叶斯分类器进行对比。

神经网络：

神经网络(neural networks)方面的研究很早就已出现，今天“神经网络”已是一个相当大的、多学科交叉的学科领域。目前使用得最广泛的一种定义，是“神

经网络是由具有适应性的简单单元组成的广泛并行互连的网络，它的组织能够模拟生物神经系统对真实世界物体所作出的交互反应”。
神经元接收到来自几个其他神经元传递过来的输入信号，这些输入信号通过带权重的连接进行传递，神经元接收到的总输入值将与神经元的阈值进行比较，然后通过“激活函数”处理以产生神经元的输出。本代码中使用的是$Sigmoid$函数。

$$\mathrm{sigmoid}(x)=\frac{1}{1+e^{-x}}$$

把许多个这样的神经元按一定的层次结构连接起来，就得到了神经网络。
要解决非线性可分问题，需考虑使用多层功能神经元。输出层与输入层之间的一层神经元，被称为隐层或隐含层，隐含层和输出层神经元都是拥有激活函数的功能神经元。
每层神经元与下一层神经元全互连，神经元之间不存在同层连接，也不存在跨层连接. 这样的神经网络结构通常称为“多层前馈神经网络”。其中输入层神经元接收外界输入，隐层与输出层神经元对信号进行加工，最终结果由输出层神经元输出；换言之，输入层神经元仅是接受输入，不进行函数处理，隐层与输出层包含功能神经元。
多层网络的学习能力比单层感知机强得多. 欲训练多层网络，简单感知机学习规则显然不够了，需要更强大的学习算法. 误差逆传播(error BackPropagation，简称 BP）算法就是其中最杰出的代表。
对于给定训练集$D=\{(\boldsymbol{x}_1,\boldsymbol{y}_1),(\boldsymbol{x}_2,\boldsymbol{y}_2),(\boldsymbol{x}_3,\boldsymbol{y}_3),\dots,(\boldsymbol{x}_m,\boldsymbol{y}_m)\},\boldsymbol{x}_i\in\mathbb{R}^d,\boldsymbol{y}_i\in\mathbb{R}^l$，即输入示例由$d$个输入神经元，$l$个输出神经元，$q$个隐层神经元的多层前馈神经网络结构，其中输出层第$j$个神经元的阈值用$\theta_j$表示，隐层第$h$个神经元的阈值用$\gamma_h$表示，输入层第$i$个神经元与隐层第$h$个神经元的连接权为$v_{ih}$，隐层第$h$个神经元与输出层第$j$个神经元之间的连接权为$\omega_{hj}$。记隐层的第$h$个神经元接收到的输入为$\alpha=\sum_{i=1}^{d}v_{ih}x_i$，输出层第$j$个神经元接收到的输入为$\beta_j=\sum_{h=1}^{q}\omega_{hj}b_h$，其中$b_h$为隐层第$h$个神经元的输出。隐层和输出层神经元都使用sigmoid函数。
对于训练例$(\boldsymbol{x}_k,\boldsymbol{y}_k)$，假定神经网络的输出为$\hat{\boldsymbol{y}}_k=\left(\hat{y}_1^k,\hat{y}_2^k,\dots,\hat{y}_l^k\right)$，即：

$$\hat{y}_j^k=f\left(\beta_j-\theta_j\right)$$

则网络$(\boldsymbol{x}_k,\boldsymbol{y}_k)$上的均方误差为：

$$E_k=\frac{1}{2}\sum_{j=1}^{l}\left(\hat{y}_j^k-y_j^k\right)^2$$

网络中有$(d+l+1)q+l$个参数需要确定，输入层到隐层的$d\times q$个权值，隐层到输出层的$q\times l$个权值、q个隐层神经元阈值、l个输出层神经元的阈值，BP是严格迭代学习算法，在迭代的每一论中采用广义的感知机学习规则对参数进行更新估计，任意参数v的更新估计式为

$$v\leftarrow v+\Delta v$$

BP算法基于梯度下降策略，以目标的负梯度方向对于参数进行调整。对误差E_k，给定学习率η，有：

$$\Delta\omega_{hj} = -\eta \frac{\partial E_k}{\partial \omega_{hj}}$$

注意到ω_{hj}先影响到第j个输出层神经元的输入值β_j，再影响到其输出值$\hat{y}_j^k$，再影响到E_k，有：

$$\frac{\partial E_k}{\partial \omega_{hj}} = \frac{\partial E_k}{\partial \hat{y}_j^k} \cdot \frac{\partial \hat{y}_j^k}{\partial \beta_j} \cdot \frac{\partial \beta_j}{\partial \omega_{hj}}$$

根据β_j的定义，显然有：

$$\frac{\partial \beta_j}{\partial \omega_{hj}} = b_h$$

Sigmoid函数有一个很好的性质：

$$f'(x) = f(x)\big(1 - f(x)\big)$$

有：

$$g_j = -\frac{\partial E_k}{\partial \hat{y}_j^k} \cdot \frac{\partial \hat{y}_j^k}{\partial \beta_j} = -\big(\hat{y}_j^k - y_j^k\big) f'\big(\beta_j - \theta_j\big) = \hat{y}_j^k\big(1 - \hat{y}_j^k\big)\big(y_j^k - \hat{y}_j^k\big)$$

就得到了BP算法中关于ω_{hj}的更新公式

$$\Delta\omega_{hj} = \eta g_j b_h$$

类似可得：

$$\Delta\theta_j = -\eta g_j$$
$$\Delta v_{ih} = \eta e_h x_i$$
$$\Delta\gamma_h = -\eta e_h$$

$$e_h = -\frac{\partial E_k}{\partial b_h} \cdot \frac{\partial b_h}{\partial \alpha_h} = -\sum_{j=1}^{l} \frac{\partial E_k}{\partial \beta_j} \cdot \frac{\partial y}{\partial b_h} f'(\alpha_h - \gamma_h) = \sum_{j=1}^{l} \omega_{hj} g_j f'(\alpha_h - \gamma_h)$$

$$= b_h(1 - b_h) \sum_{j=1}^{l} \omega_{hj} g_j$$

学习率$\eta \in (0,1)$控制着算法每一轮迭代中的更新步长，若太大容易振荡，太小容易收敛速度过慢。

神经网络的优点：强大的非线性拟合能力，能够处理高维数据。

缺点：需要大量的数据进行训练，容易过拟合。

随机森林模型

随机森林（$Random\ Forest$,简称RF）是$Bagging$的一个扩展变体。RF在以决策树为基学习器构建$Bagging$集成的基础上，进一步在决策树的训练过程中加入随机属性选择。传统决策树在选择划分属性时是在当前结点的属性集合（假设有d个属性）中选择一个最优属性，而在RF中，对基决策器的每个节点，先从

该节点的属性集合中随机选择一个包含k个属性的子集，然后再从这个子集中选择一个最优属性用于划分，这里的k控制了随机性的引入。随机森林的基学习器的多样性不仅来自于样本扰动，还来自属性扰动，这就使最终集成的泛化性能可以通过个体学习器之间差异度的增加而进一步提升。

二、数据集及其来源

1. 帕尔默群岛（南极洲）企鹅数据

数据文件:data/penguins/penguins_size.csv
来源： https://www.kaggle.com/datasets/parulpandeypalmer-archipelago-antarctica-penguin-data

2. 酒数据集

数据文件:data/wine/wine.data
来源:https://archive.ics.uci.edu/dataset/109/wine

3.电离层数据集

数据文件:data/ionosphere/ionosphere.data
来源:https://archive.ics.uci.edu/dataset/52/ionosphere

4. 鸢尾花数据集

数据文件:data/iris/iris.data
来源:https://archive.ics.uci.edu/dataset/53/iris

5. 皮肤病数据集

数据文件:data/dermatology/dermatology.data
来源:https://archive.ics.uci.edu/dataset/33/dermatology

三、文件说明

data 文件夹：

本文件夹用于存放所有原始数据集文件。每个数据集有其独立的子文件夹
- `data/penguins/` - 存放帕尔默群岛企鹅数据集。
- `data/wine/` - 存放酒数据集。
- `data/ionosphere/` - 存放电离层数据集。
- `data/iris/` - 存放鸢尾花数据集。
- `data/dermatology/` - 存放皮肤病数据集。

model 文件夹：

此文件夹用于存放模型文件
- `model/naive_bayes_classifier.py` - 朴素贝叶斯模型
- `model/network.py` - 神经网络模型

- `random_forest.py` - 随机森林模型

train-code 文件夹：

该部分包含用于训练和测试机器学习模型的代码
- `train-code/penguins/` - 存放帕尔默群岛企鹅训练文件
- `train-code/wine/` - 存放酒训练文件
- `train-code/ionosphere/` - 存放电离层训练文件
- `train-code/iris/` - 存放鸢尾花训练文件
- `train-code/dermatology/` - 存放皮肤病训练文件

四、需求库及实验环境

需求库

numpy==1.24.3
scikit-learn==1.3.0
scipy==1.11.1
pandas==2.0.3

硬件环境

CPU：11th Gen Intel(R) Core(TM) i5-11320H @ 3.20GHz
RAM：16GB

软件环境

操作系统：Windows11
开发工具：PyCharm2023.3.4
虚拟环境：Anaconda
Python 版本：3.11

五、训练过程

在 train-code/下面的以数据集命名的文件夹中，以 dermatology 为例，train-code/dermatology/data_preprocessing.py 中对数据进行预处理。主要使用 pandas 库，然后转化为 numpy 数组，预处理主要工作是需要对于异常值进行处理，以及对于非数字值进行编码。在 train-code/dermatology/dermatology_train.py 的代码中，对于模型进行训练。每个数据集训练六个模型，为朴素贝叶斯、神经网络、随机森林三个模型以及这三个模型的 sklearn 实现，用于对比。model 文件夹中所放的是训练文件，以 model/network.py 为例，class Network 是所实现的神经网络类，用于构建一个神经网络，以及训练、测试等步骤。在 model/network.py 中也提供了函数调用的方式，network_function 函数可以直接一次性输入所有参数，返回预测值。

六、训练结果及分析

	dermatology	ionosphere	iris	penguins	wine
朴素贝叶斯	0.875	0.843	0.867	0.881	0.917
朴素贝叶斯(sklearn)	0.875	0.843	0.867	0.895	0.917
神经网络	0.986	0.871	0.933	0.970	0.861
神经网络(sklearn)	1.000	0.914	0.933	1.000	0.917
随机森林	0.9722	0.871	0.900	1.000	0.889
随机森林(sklearn)	0.9861	0.914	0.900	1.000	0.944

分析：

从我自己实现的模型于 sklearn 的模型比较中可以看出，朴素贝叶斯分类器模型的准确率这两个模型基本上差不多，而神经网络和随机森林上则存在不同程度的差距，在五个数据集上平均分别相差 0.0286 和 0.02238。在一些较小且数据集较为平衡上的相差较小，如 iris 和 penguins，但是在一些数据量较为不均匀的数据集，如 wine 上存在较大差距。主要原因可以归结于 sklearn 中的算法对于参数的优化更好，比如说对于神经网络中隐藏层层数、节点数，神经网络学习率等的设置更为合理，随机森林模型也是同理。

对于 dermatology 数据集，神经网络模型最为优秀，随机森林模型次之，朴素贝叶斯模型较差；

对于 ionosphere 数据集，神经网络模型和随机森林模型基本上相差不大，朴素贝叶斯模型较差；

对于 iris 数据集，神经网络模型最为优秀，随机森林模型次之，朴素贝叶斯模型较差；

对于 penguins 数据集，神经网络模型最为优秀和随机森林模型明显优秀于朴素贝叶斯模型；

对于 wine 数据集，随机森林优秀于朴素贝叶斯模型，最后是神经网络模型。

综合可见，神经网络模型和随机森林的正确率是较高的，而朴素贝叶斯差点，主要也因为朴素贝叶斯分类器的实现是较为简单的，其的优化算法如半朴素贝叶斯算法、贝叶斯网等可能有更好的效果。

七、总结

本报告中采用了神经网络模型、随机森林模型、朴素贝叶斯模型进行分类。其中神经网络算法的表现更好，在各数据集上的表现更为稳定。对于这几个模型都还有优化的地方。可以通过更复杂的模型以及参数的优化等，提升模型的精度。而且对于模型的判定标准也较为单一，可以使用更多样化的评价标准。

参考文献：
[1]周志华.机器学习[M].清华大学出版社,2016.
[1]李航.统计学习方法（第二版）[M].清华大学出版社,2019.

附录：程序源码

model/naïve_bayes_classifier.py

```
import numpy as np


# 计算给定测试数据和标签的多项式概率密度函数
def calculate_pdf(test_data, tag_mean, tag_std, p_tag):
    test_tag = np.zeros(test_data.shape)
    for i in range(test_data.shape[1]):
        test_tag[:, i] = 1 / (np.sqrt(2 * np.pi) * tag_std[i]) * np.exp(
            -(test_data[:, i] - tag_mean[i]) ** 2 / (2 * tag_std[i] ** 2))
    p_test_tag = np.ones(test_tag.shape[0])
    p_test_tag *= p_tag
    for i in range(test_data.shape[1]):
        p_test_tag *= test_tag[:, i]
    return p_test_tag


class NaiveBayesClassifier:
    """
    朴素贝叶斯分类器
    tags_data[]:每种类别的测试数据列表
    p_tags[]:每种类别的先验概率列表
    tags_mean[]:每种类别的均值列表
    tags_std[]:每种类别的标准差列表
    elements:类别标签的唯一值列表
    """

    def __init__(self) -> None:
        self.tags_data = []
```

```
        self.p_tags = []
        self.tags_mean = []
        self.tags_std = []
        self.elements = []

    # 训练模型方法
    def train(self, train_x, train_y):
        self.elements = np.unique(train_y)
        for element in self.elements:
            self.tags_data.append(train_x[train_y == element])
        for tag in self.tags_data:
            self.p_tags.append(tag.shape[0] / train_x.shape[0])
            self.tags_mean.append(np.mean(tag, axis=0))
            tag_std = np.nanstd(tag, axis=0)
            tag_std[tag_std == 0] = 1e-6
            self.tags_std.append(tag_std)

    # 预测函数
    def predict(self, test_x, test_y):
        p_test_tags = []
        for i in range(len(self.elements)):
            p_test_tags.append(calculate_pdf(
                test_x, self.tags_mean[i], self.tags_std[i], self.p_tags[i]
            ))
        # 将每个数据点的所有类别概率合并为一个数组
        concatenated_array_col = None
        for tag in p_test_tags:
            tags = tag.reshape(1, -1)
            if concatenated_array_col is None:
                concatenated_array_col = tags.T
            else:
                concatenated_array_col = np.hstack((concatenated_array_col,
tags.T))
        index_max = np.argmax(concatenated_array_col, axis=1)
        acc = np.sum(index_max == test_y)
        print('Naive bayes classifier test Accuracy', acc / test_y.shape[0])
        return index_max


# 朴素贝叶斯的函数形式调用方法
def naive_bayes_classifier_function(train_x, train_y, test_x, test_y):
    model = NaiveBayesClassifier()
    model.train(train_x, train_y)
    model.predict(test_x, test_y)
```

model/network.py

```
import numpy as np


class Network:
    """
    神经网络类,代表一个带有隐藏层的简单神经网络
    input_size:输入层大小
    hidden_size:隐藏层大小
    output_size:输出层大小
    theta,V,B,w:网络的权重和偏置
    alpha:网络训练的学习率，默认为 0.1
    """

    def __init__(self, input_size, hidden_size, output_size, alpha=0.1) -> None:
        self.input_size = input_size
        self.hidden_size = hidden_size
        self.output_size = output_size
        self.theta = np.random.randn(output_size, 1)
        self.V = np.random.randn(output_size, hidden_size)
        self.b = np.random.randn(hidden_size, 1)
        self.W = np.random.randn(hidden_size, input_size)
        self.alpha = alpha

    # sigmoid 函数,使用静态函数
    @staticmethod
    def sigmoid(z_sigmoid):
        return 1 / (1 + np.exp(-z_sigmoid))

    def train(self, train_x, train_y):
        line = np.linspace(0, train_x.shape[0] - 1, train_x.shape[0])
        loss_array = []
        n = train_x.shape[0]
        train_y = train_y.astype(int)
        one_hot_h = np.zeros((n, self.output_size))
        one_hot_h[np.arange(n), train_y.reshape((n,))] = 1
        for j in range(100):    # epoch=100
            np.random.shuffle(line)   # 随机打乱索引数组
            for i in range(train_x.shape[0]):
                x = train_x[int(line[i]), :]
                x = x.reshape(-1, 1)
                h = one_hot_h[int(line[i]), :]
```

```
                h = h.reshape(-1, 1)
                z = np.dot(self.W, x) + self.b
                a = Network.sigmoid(z)
                t = np.dot(self.V, a) + self.theta
                y = Network.sigmoid(t)
                L = 1 / 2 * np.square(y - h)
                L_theta = (y - h) * y * (1 - y)
                L_V = np.dot(L_theta, a.T)
                L_b = np.dot(self.V.T, L_theta) * a * (1 - a)
                L_W = np.dot(L_b, x.T)
                self.theta = self.theta - self.alpha * L_theta
                self.V = self.V - self.alpha * L_V
                self.b = self.b - self.alpha * L_b
                self.W = self.W - self.alpha * L_W
            Loss = 0
            for i in range(train_x.shape[0]):
                # for i in range(10):
                x = train_x[i, :]
                x = x.reshape(-1, 1)
                h = one_hot_h[i, :]
                h = h.reshape(-1, 1)
                z = np.dot(self.W, x) + self.b
                a = Network.sigmoid(z)
                t = np.dot(self.V, a) + self.theta
                y = Network.sigmoid(t)
                L = 1 / 2 * np.square(y - h)
                loss = np.sum(L)
                Loss += loss
            loss_array.append(Loss)
        return loss_array

    # 网络的预测方法
    def predict(self, test_x, test_y):
        acc = 0
        predictions = []
        for i in range(test_x.shape[0]):
            x = test_x[i, :]
            x = x.reshape(-1, 1)
            h = test_y[i]
            z = np.dot(self.W, x) + self.b
            a = Network.sigmoid(z)
            t = np.dot(self.V, a) + self.theta
            y = Network.sigmoid(t)
            if np.argmax(y) == h:
```

```
                acc = acc + 1
            predictions.append(np.argmax(y))
        print('Network test accuracy', acc / test_x.shape[0])
        return predictions


def network_function(input_size, hidden_size, output_size, train_x, train_y, test_x, test_y,alpha=0.01):
    """
    神经网络的函数调用方法，采用于 Network 类中相同的代码,可以直接通过 network_function 调用
    """
    network = Network(input_size, hidden_size, output_size, alpha=alpha)
    loss = network.train(train_x, train_y)
    predict = network.predict(test_x, test_y)
    return loss, predict
```

model/random_forest.py

```
from sklearn import tree
import numpy as np
from scipy import stats


class RandomForest:
    """
    随机森林模型(以决策树为基学习器)
    model_all:存储所有决策树模型的列表
    y_pred_all:存储所有模型预测结果的列表
    accuracy_all:存储所有模型的准确率的列表
    base_learner_times:基学习器（决策树）的数量(默认为 10)
    """

    def __init__(self, base_learner_times=10):
        self.model_all = []
        self.y_pred_all = []
        self.accuracy_all = []
        self.base_learner_times = base_learner_times

    def train(self, train_x, train_y):
        for i in range(1, self.base_learner_times + 1):
            choice_array = np.random.choice(train_x.shape[0], train_x.shape[0], replace=True)
            train_x_choice = train_x[choice_array, :]
```

```
            train_y_choice = train_y[choice_array]
            model = tree.DecisionTreeClassifier(
                random_state=1,
                criterion='entropy',
                max_features=int(train_x.shape[1] / 3))  # 最大特征数为三分之一的特征总数
            model.fit(train_x_choice, train_y_choice)
            self.model_all.append(model)

    def predict(self, test_x, test_y):
        for i in range(len(self.model_all)):
            y_pred = self.model_all[i].predict(test_x)
            y_pred = y_pred.reshape(-1, 1)
            if i == 0:
                self.y_pred_all = y_pred
            else:
                self.y_pred_all = np.hstack((self.y_pred_all, y_pred))
            # 使用多数投票法作为最终预测结果
            y_pred_f = stats.mode(self.y_pred_all, axis=1).mode
            y_pred_f = y_pred_f.reshape(-1, 1)
            test_y = test_y.reshape(-1, 1)
            accuracy = np.sum((y_pred_f == test_y)) / test_y.size
            self.accuracy_all.append(accuracy)
        # 打印使用全部基学习器时的准确率
        print(f"T:{self.base_learner_times} random forest test accuracy:", self.accuracy_all[-1])
        # 返回所有模型的准确率列表,有需要时进行数据分析
        return self.accuracy_all


# 函数形式调用随机森林模型
def random_forest_function(train_x, train_y, test_x, test_y, base_learner_times=10):
    model = RandomForest(base_learner_times=base_learner_times)
    model.train(train_x, train_y)
    accuracy = model.predict(test_x, test_y)
    return accuracy
```

对于数据集的模型训练部分，代码有些具有相同部分，
仅提供 train-code/dermatology/*中的代码位示例
train-code/dermatology/data_preprocessing.py

```
import pandas as pd
import numpy as np
```

```
def DataPreprocessing():
    """
    该函数针对 dermatology.data 进行预处理
    """
    df = pd.read_csv('../../data/dermatology/dermatology.data')
    df = df[df.iloc[:, -2] != '?']
    last_column = df.columns[-1]
    df[last_column] = df[last_column].replace({6: 0})
    data = df.to_numpy()

    _data_x = data[:, 0:-1]
    _data_y = data[:, -1]
    data_x = _data_x.astype(np.float64)
    data_y = _data_y.astype(np.int32)
    min_vals = data_x[:, -1].min(axis=0)
    max_vals = data_x[:, -1].max(axis=0)
    range_vals = max_vals - min_vals
    data_x[:, -1] = (data_x[:, -1] - min_vals) / range_vals

    return data_x, data_y
```

train-code/dermatology/ dermatology_train.py

```
import numpy as np
from sklearn.model_selection import train_test_split
from sklearn.naive_bayes import GaussianNB
from sklearn.neural_network import MLPClassifier
from sklearn.ensemble import RandomForestClassifier
from data_preprocessing import DataPreprocessing
from model.naive_bayes_classifier import NaiveBayesClassifier
from model.network import Network
from model.random_forest import RandomForest
import warnings

warnings.filterwarnings('ignore')

print("Dataset:preprocessing")
data_x, data_y = DataPreprocessing()
X_train, X_test, y_train, y_test = train_test_split(data_x, data_y, test_size=0.2,
random_state=42)

print("naive bayes classifier:")
model_naive_bayes_classifier = NaiveBayesClassifier()
```

```
model_naive_bayes_classifier.train(X_train, y_train)
model_naive_bayes_classifier.predict(X_test, y_test)

model_sklearn_naive_bayes_classifier = GaussianNB()
model_sklearn_naive_bayes_classifier.fit(X_train, y_train)
y_pred_naive_bayes_classifier =
model_sklearn_naive_bayes_classifier.predict(X_test)
print("sklearn model accuracy:", np.sum(y_pred_naive_bayes_classifier == y_test) /
len(y_test))

print("network:")
model_network = Network(X_train.shape[1], 12, np.unique(y_train).size, alpha=0.1)
model_network.train(X_train, y_train)
model_network.predict(X_test, y_test)

model_sklearn_network = MLPClassifier()
model_sklearn_network.fit(X_train, y_train)
y_pred_network = model_sklearn_network.predict(X_test)
print("sklearn network accuracy:", np.sum(y_pred_network == y_test) / len(y_test))

print("random forest:")
model_random_forest = RandomForest()
model_random_forest.train(X_train, y_train)
y_pred = model_random_forest.predict(X_test, y_test)

model_sklearn_random_forest = RandomForestClassifier()
model_sklearn_random_forest.fit(X_train, y_train)
y_pred_random_forest = model_sklearn_random_forest.predict(X_test)
print("sklearn random forest accuracy:", np.sum(y_pred_random_forest == y_test) /
len(y_test))
```