西瓜书机器学习算法

线性回归

```python
import numpy as np
import matplotlib.pyplot as plt

# 构造数据
x = np.array([1, 2, 3, 4, 5])  # 输入特征
y = np.array([3, 4, 5, 6, 7])  # 目标变量

# 添加偏置项
X = np.vstack([x, np.ones(len(x))]).T

# 使用最小二乘法估计参数
w, b = np.linalg.lstsq(X, y, rcond=None)[0]

# 绘制数据点和拟合直线
plt.scatter(x, y, color='blue', label='Data')
plt.plot(x, w*x + b, color='red', label='Linear Regression')
plt.xlabel('Input feature')
plt.ylabel('Target variable')
plt.legend()
plt.show()

```

对数几率回归

```python
import numpy as np

def sigmoid(z):
    # Sigmoid函数
    return 1 / (1 + np.exp(-z))

def logistic_regression(X, y, num_iterations, learning_rate):
    # 初始化参数
    num_samples, num_features = X.shape
    intercept = np.ones((num_samples, 1))
    X = np.hstack((intercept, X))
    theta = np.zeros(num_features + 1)

    # 梯度下降优化
    for i in range(num_iterations):
        z = np.dot(X, theta)
        h = sigmoid(z)
        gradient = np.dot(X.T, (h - y)) / num_samples
        theta -= learning_rate * gradient

    return theta

# 构造数据
X = np.array([[1, 2], [2, 1], [3, 3], [4, 2], [5, 4]])  # 输入特征
y = np.array([0, 0, 1, 1, 1])  # 目标变量

# 执行对数几率回归
num_iterations = 1000
learning_rate = 0.01
theta = logistic_regression(X, y, num_iterations, learning_rate)

print("Theta:", theta)


```

线性判别分析

```python
import numpy as np

def linear_discriminant_analysis(X, y):
    # 计算每个类别的均值向量
    class_mean = []
    for c in np.unique(y):
        class_mean.append(np.mean(X[y == c], axis=0))

    # 计算类内散布矩阵
    within_class_scatter = np.zeros((X.shape[1], X.shape[1]))
    for c, mean_vec in zip(np.unique(y), class_mean):
        class_scatter = np.cov(X[y == c].T)
        within_class_scatter += class_scatter

    # 计算类间散布矩阵
    overall_mean = np.mean(X, axis=0)
    between_class_scatter = np.zeros((X.shape[1], X.shape[1]))
    for mean_vec in class_mean:
        n = X[y == c].shape[0]
        mean_diff = (mean_vec - overall_mean).reshape(X.shape[1], 1)
        between_class_scatter += n * np.dot(mean_diff, mean_diff.T)

    # 计算投影矩阵
    eigenvalues, eigenvectors = np.linalg.eig(
        np.linalg.inv(within_class_scatter).dot(between_class_scatter)
    )
    # 按特征值从大到小排序特征向量
    eig_pairs = [
        (np.abs(eigenvalues[i]), eigenvectors[:, i]) for i in range(len(eigenvalues))
    ]
    eig_pairs.sort(key=lambda k: k[0], reverse=True)

    # 选择k个最大的特征向量
    projection_matrix = np.hstack(
        [eig_pairs[i][1].reshape(X.shape[1], 1) for i in range(X.shape[1])]
    )

    # 将数据投影到新的特征空间
    X_lda = np.dot(X, projection_matrix)

    return X_lda

# 构造数据
X = np.array([[1, 2], [2, 1], [3, 3], [4, 2], [5, 4]])  # 输入特征
y = np.array([0, 0, 1, 1, 1])  # 目标变量

# 执行线性判别分析
X_lda = linear_discriminant_analysis(X, y)

print("Projected Data:")
print(X_lda)

```

多分类学习

1. 一对一

```python
import numpy as np

def one_vs_one(X, y, num_classes):
    num_samples, num_features = X.shape
    classifiers = []

    for i in range(num_classes):
        for j in range(i + 1, num_classes):
            # 准备数据（两个类别之间）
            class_indices = np.logical_or(y == i, y == j)
            binary_X = X[class_indices]
            binary_y = y[class_indices]
            binary_y[binary_y == i] = 0
            binary_y[binary_y == j] = 1

            # 训练二分类器
            classifier = train_binary_classifier(binary_X, binary_y)
            classifiers.append((i, j, classifier))

    return classifiers

def train_binary_classifier(X, y):
    # 训练二分类器（例如：使用逻辑回归）
    # 返回训练好的分类器
    pass

# 构造数据
X = np.array([[1, 2], [2, 1], [3, 3], [4, 2], [5, 4]])  # 输入特征
y = np.array([0, 1, 2, 1, 2])  # 目标变量

# 执行一对一分类
num_classes = 3
classifiers = one_vs_one(X, y, num_classes)

# 使用分类器进行预测
def predict_one_vs_one(classifiers, x):
    scores = np.zeros(num_classes)

    for i, j, classifier in classifiers:
        binary_prediction = classifier.predict(x)
        if binary_prediction == 0:
                scores[i] += 1
            else:
                scores[j] += 1

    return np.argmax(scores)

# 预测新样本
new_x = np.array([2.5, 2.5])
prediction = predict_one_vs_one(classifiers, new_x)
print("Prediction (One-vs-One):", prediction)

```

2. 一对多

```python
import numpy as np

def one_vs_rest(X, y, num_classes):
    num_samples, num_features = X.shape
    classifiers = []

    for i in range(num_classes):
        # 准备数据（一个类与其他类）
        binary_X = X
        binary_y = np.where(y == i, 1, 0)

        # 训练二分类器
        classifier = train_binary_classifier(binary_X, binary_y)
        classifiers.append((i, classifier))

    return classifiers

# 构造数据
X = np.array([[1, 2], [2, 1], [3, 3], [4, 2], [5, 4]])  # 输入特征
y = np.array([0, 1, 2, 1, 2])  # 目标变量

# 执行一对多分类
num_classes = 3
classifiers = one_vs_rest(X, y, num_classes)

# 使用分类器进行预测
def predict_one_vs_rest(classifiers, x):
    scores = np.zeros(num_classes)

    for i, classifier in classifiers:
        binary_prediction = classifier.predict(x)
        scores[i] = binary_prediction

    return np.argmax(scores)

# 预测新样本
new_x = np.array([2.5, 2.5])
prediction = predict_one_vs_rest(classifiers, new_x)
print("Prediction (One-vs-Rest):", prediction)

```

3. 多对多

```python
import numpy as np

def multiclass_classifier(X, y):
    # 训练多分类器（例如：使用逻辑回归、决策树等方法）
    # 返回训练好的多分类器
    pass

# 构造数据
X = np.array([[1, 2], [2, 1], [3, 3], [4, 2], [5, 4]])  # 输入特征
y = np.array([0, 1, 2, 1, 2])  # 目标变量

# 执行多分类学习
classifier = multiclass_classifier(X, y)

# 预测新样本
new_x = np.array([2.5, 2.5])
prediction = classifier.predict(new_x)
print("Prediction (Multiclass Classifier):", prediction)

```

感知机

```python
import numpy as np

```

```python
# 定义多层感知机类
class MLP:
    def __init__(self, input_size, hidden_size, output_size):
        self.input_size = input_size
        self.hidden_size = hidden_size
        self.output_size = output_size

        # 初始化权重和偏置
        self.W1 = np.random.randn(self.input_size, self.hidden_size)
        self.b1 = np.zeros((1, self.hidden_size))
        self.W2 = np.random.randn(self.hidden_size, self.output_size)
        self.b2 = np.zeros((1, self.output_size))

    def forward(self, X):
        # 前向传播
        self.z1 = np.dot(X, self.W1) + self.b1
        self.a1 = self.relu(self.z1)
        self.z2 = np.dot(self.a1, self.W2) + self.b2
        self.a2 = self.softmax(self.z2)
        return self.a2

    def relu(self, x):
        # ReLU 激活函数
        return np.maximum(0, x)

    def softmax(self, x):
        # Softmax 函数
        exp_scores = np.exp(x)
        return exp_scores / np.sum(exp_scores, axis=1, keepdims=True)

    def backward(self, X, y, learning_rate):
        # 反向传播
        num_samples = X.shape[0]

        # 计算误差
        delta3 = self.a2
        delta3[range(num_samples), y] -= 1
        delta3 /= num_samples

        # 更新权重和偏置
        dW2 = np.dot(self.a1.T, delta3)
        db2 = np.sum(delta3, axis=0, keepdims=True)
        delta2 = np.dot(delta3, self.W2.T) * (self.a1 > 0)
        dW1 = np.dot(X.T, delta2)
        db1 = np.sum(delta2, axis=0)

        # 梯度下降更新权重和偏置
        self.W2 -= learning_rate * dW2
        self.b2 -= learning_rate * db2
        self.W1 -= learning_rate * dW1
        self.b1 -= learning_rate * db1

    def train(self, X, y, learning_rate, num_epochs):
        for epoch in range(num_epochs):
            # 前向传播
            output = self.forward(X)

            # 计算损失函数（交叉熵损失）
            loss = self.cross_entropy_loss(output, y)
            if (epoch + 1) % 100 == 0:
                print(f"Epoch [{epoch+1}/{num_epochs}], Loss: {loss}")

            # 反向传播和参数更新
            self.backward(X, y, learning_rate)

    def predict(self, X):
        # 前向传播并预测类别
        output = self.forward(X)
        return np.argmax(output, axis=1)

    def cross_entropy_loss(self, y_pred, y_true):
        # 计算交叉熵损失
        num_samples = y_pred.shape[0]
        log_probs = -np.log(y_pred[range(num_samples), y_true])
        loss = np.sum(log_probs) / num_samples
        return loss

# 构造随机数据
X = np.array([[1, 2], [2, 1], [3, 3], [4, 2], [5, 4]])  # 输入特征
y = np.array([0, 1, 2, 1, 2])  # 目标变量

# 设置超参数
input_size = 2
hidden_size = 100
output_size = 3
learning_rate = 0.001
num_epochs = 1000

# 创建 MLP 对象
model = MLP(input_size, hidden_size, output_size)

# 训练模型
```

```python
model.train(X, y, learning_rate, num_epochs)

# 预测新样本
new_X = np.array([[2.5, 2.5]])
predictions = model.predict(new_X)
print("Prediction:", predictions)

```

梯度下降算法

```python
def gradient_descent(learning_rate, num_epochs):
    x = 5  # 初始值

    for epoch in range(num_epochs):
        gradient = 2 * x  # 目标函数的导数
        step = learning_rate * gradient  # 计算步长
        x -= step  # 更新参数

    return x

learning_rate = 0.1
num_epochs = 100
result = gradient_descent(learning_rate, num_epochs)
print("Result:", result)

```

反向传播算法（BP算法）

```python
import numpy as np

def sigmoid(x):
    return 1 / (1 + np.exp(-x))

def sigmoid_derivative(x):
    return sigmoid(x) * (1 - sigmoid(x))

def initialize_parameters(layer_dims):
    parameters = {}
    num_layers = len(layer_dims)

    for l in range(1, num_layers):
        parameters['W' + str(l)] = np.random.randn(layer_dims[l], layer_dims[l-1]) * 0.01
        parameters['b' + str(l)] = np.zeros((layer_dims[l], 1))

    return parameters

def forward_propagation(X, parameters):
    A = X
    caches = []
    num_layers = len(parameters) // 2

    for l in range(1, num_layers):
        Z = np.dot(parameters['W' + str(l)], A) + parameters['b' + str(l)]
        A = sigmoid(Z)
        cache = (Z, A)
        caches.append(cache)

    Z = np.dot(parameters['W' + str(num_layers)], A) + parameters['b' + str(num_layers)]
    A = sigmoid(Z)
    cache = (Z, A)
    caches.append(cache)

    return A, caches

def compute_cost(A, Y):
    m = Y.shape[1]
    cost = (-1 / m) * np.sum(Y * np.log(A) + (1 - Y) * np.log(1 - A))
    return cost

def backward_propagation(X, Y, caches, parameters):
    m = X.shape[1]
    num_layers = len(parameters) // 2
    grads = {}

    dZ = (1 / m) * (caches[-1][1] - Y)
    grads['dW' + str(num_layers)] = np.dot(dZ, caches[-2][1].T)
    grads['db' + str(num_layers)] = np.sum(dZ, axis=1, keepdims=True)

    for l in range(num_layers-1, 0, -1):
        dA = np.dot(parameters['W' + str(l+1)].T, dZ)
        dZ = dA * sigmoid_derivative(caches[l-1][0])
```

```python
            grads['dW' + str(l)] = np.dot(dZ, caches[l-1][1].T)
            grads['db' + str(l)] = np.sum(dZ, axis=1, keepdims=True)

    return grads

def update_parameters(parameters, grads, learning_rate):
    num_layers = len(parameters) // 2

    for l in range(1, num_layers+1):
        parameters['W' + str(l)] -= learning_rate * grads['dW' + str(l)]
        parameters['b' + str(l)] -= learning_rate * grads['db' + str(l)]

    return parameters

def train(X, Y, layer_dims, learning_rate, num_epochs):
    parameters = initialize_parameters(layer_dims)

    for epoch in range(num_epochs):
        A, caches = forward_propagation(X, parameters)
        cost = compute_cost(A, Y)
        grads = backward_propagation(X, Y, caches, parameters)
        parameters = update_parameters(parameters, grads, learning_rate)

        if epoch % 100 == 0:
            print(f"Cost after epoch {epoch}: {cost}")

    return parameters

# 示例用法
X = np.array([[0, 0, 1, 1], [0, 1, 0, 1]])
Y = np.array([[0, 1, 1, 0]])
layers_dims = [2, 2, 1]  # 输入层、隐藏层、输出层的神经元数量
learning_rate = 0.1
num_epochs = 1000

trained_parameters = train(X, Y, layers_dims, learning_rate, num_epochs)
print(trained_parameters)
```

支持向量分类

```python
import numpy as np

class SVM:
    def __init__(self, learning_rate=0.01, max_iter=1000):
        self.lr = learning_rate
        self.max_iter = max_iter
        self.W = None
        self.b = None

    def fit(self, X, y):
        y = np.where(y <= 0, -1, 1)  # 将标签转换为二分类任务的形式（-1和1)
        self.W = np.zeros(X.shape[1])
        self.b = 0

        for _ in range(self.max_iter):
            for i in range(len(X)):
                if y[i] * (np.dot(X[i], self.W) - self.b) >= 1:
                    self.W -= self.lr * (2 * 1/self.max_iter * self.W)  # 更新权重
                else:
                    self.W -= self.lr * (2 * 1/self.max_iter * self.W - np.dot(X[i], y[i]))  # 更新权重和偏置
                    self.b -= self.lr * y[i]  # 更新偏置

    def predict(self, X):
        y_pred = np.sign(np.dot(X, self.W) - self.b)  # 计算预测值
        return np.where(y_pred <= 0, 0, 1)  # 将连续的预测值转换为分类标签（0和1)

# 示例用法
X = np.array([[1, 2], [-1, -2], [2, 2], [-2, -1]])
y = np.array([1, -1, 1, -1])

svm_classifier = SVM(learning_rate=0.01, max_iter=1000)
svm_classifier.fit(X, y)

X_test = np.array([[3, 3], [-3, -3]])
y_pred = svm_classifier.predict(X_test)

print("Predictions:", y_pred)

```

支持向量回归

```python
import numpy as np
from sklearn.preprocessing import StandardScaler


class SVR:
    def __init__(self, kernel='linear', epsilon=0.1, C=1.0, max_iter=1000, learning_rate=0.01):
        self.kernel = kernel
        self.epsilon = epsilon
        self.C = C
        self.max_iter = max_iter
        self.learning_rate = learning_rate
        self.scaler = StandardScaler()
        self.support_vectors = None
        self.support_vector_labels = None
        self.support_vector_weights = None
        self.b = None

    def fit(self, X, y):
        scaled_X = self.scaler.fit_transform(X)
        n_samples, n_features = scaled_X.shape
        self.support_vectors = []
        self.support_vector_labels = []
        self.support_vector_weights = []
        self.b = 0

        for _ in range(self.max_iter):
            for i in range(n_samples):
                error = np.dot(self.support_vector_weights, self.kernel_func(scaled_X[i], self.support_vectors)) - y[i]

                if (y[i] * error < -self.epsilon and self.support_vector_weights[i] < self.C) or \
                        (y[i] * error > self.epsilon and self.support_vector_weights[i] > 0):
                    self.support_vector_weights[i] += self.learning_rate

            # 更新常数项 b
            self.b = np.mean(y - np.dot(self.support_vector_weights, self.kernel_func(scaled_X, self.support_vectors)))

            # 选择支持向量
            mask = self.support_vector_weights > 0
            self.support_vectors = scaled_X[mask]
            self.support_vector_weights = self.support_vector_weights[mask]
            self.support_vector_labels = y[mask]

    def predict(self, X):
        scaled_X = self.scaler.transform(X)
        y_pred = np.dot(self.support_vector_weights, self.kernel_func(scaled_X, self.support_vectors)) + self.b
        return y_pred

    def kernel_func(self, x1, x2):
        if self.kernel == 'linear':
            return np.dot(x1, np.transpose(x2))
        elif self.kernel == 'rbf':
            gamma = 1.0 / x1.shape[1]  # 默认 gamma 设置为特征数的倒数
            pairwise_sq_dists = np.sum((x1[:, np.newaxis, :] - x2[np.newaxis, :, :]) ** 2, axis=-1)
            return np.exp(-gamma * pairwise_sq_dists)
        else:
            raise ValueError("Unsupported kernel type. Supported kernels: linear, rbf")


# 示例用法
X = np.array([[1, 2], [-1, -2], [2, 2], [-2, -1]])
y = np.array([3, -1, 4, -3])

svr_model = SVR(kernel='linear', epsilon=0.1, C=1.0, max_iter=1000, learning_rate=0.01)
svr_model.fit(X, y)

X_test = np.array([[3, 3], [-3, -3]])
y_pred = svr_model.predict(X_test)

print("Predictions:", y_pred)

```

朴素贝叶斯分类器

```python
import numpy as np


class NaiveBayesClassifier:
    def __init__(self):
        self.classes = None
        self.class_priors = None
        self.feature_probabilities = None

    def fit(self, X, y):
        self.classes = np.unique(y)
        self.class_priors = np.zeros(len(self.classes))
        self.feature_probabilities = []

        for i, c in enumerate(self.classes):
            X_c = X[y == c]
            self.class_priors[i] = X_c.shape[0] / X.shape[0]

            # 计算每个特征在给定类别下的概率
            feature_probs = []
            for j in range(X.shape[1]):
                feature_values = np.unique(X[:, j])
                feature_counts = np.zeros(len(feature_values))
                for k, v in enumerate(feature_values):
                    feature_counts[k] = np.sum(X_c[:, j] == v)

                # 使用拉普拉斯平滑避免概率为零
                feature_probs.append((feature_counts + 1) / (X_c.shape[0] + len(feature_values)))
            self.feature_probabilities.append(feature_probs)

    def predict(self, X):
        y_pred = []
        for xi in X:
            class_scores = []
            for i, c in enumerate(self.classes):
                class_scores.append(np.log(self.class_priors[i]) + \
 np.sum(np.log(self.feature_probabilities[i])[np.arange(len(xi)), xi]))

            y_pred.append(self.classes[np.argmax(class_scores)])
        return np.array(y_pred)


# 示例用法
X = np.array([[1, 2], [1, 3], [2, 2], [3, 2], [3, 4]])
y = np.array(['A', 'A', 'B', 'B', 'B'])

nb_classifier = NaiveBayesClassifier()
nb_classifier.fit(X, y)
```

```python

X_test = np.array([[1, 1], [3, 3]])
y_pred = nb_classifier.predict(X_test)

print("Predictions:", y_pred)

```

半朴素贝叶斯分类器

```python
import numpy as np


class SemiNaiveBayesClassifier:
    def __init__(self, alpha=1.0):
        self.alpha = alpha
        self.classes = None
        self.class_priors = None
        self.feature_probabilities = None
        self.feature_selector = None

    def fit(self, X, y, feature_selector=None):
        self.classes = np.unique(y)
        self.class_priors = np.zeros(len(self.classes))
        self.feature_probabilities = []
        self.feature_selector = feature_selector

        for i, c in enumerate(self.classes):
            X_c = X[y == c]
            self.class_priors[i] = X_c.shape[0] / X.shape[0]

            # 根据特征选择器选择一部分特征
            if feature_selector is None:
                selected_features = np.arange(X.shape[1])
            else:
                selected_features = feature_selector(X_c, y[y == c])

            # 计算每个特征在给定类别下的概率
            feature_probs = []
            for j in selected_features:
                feature_values = np.unique(X_c[:, j])
                feature_counts = np.zeros(len(feature_values))
                for k, v in enumerate(feature_values):
                    feature_counts[k] = np.sum(X_c[:, j] == v)

                # 使用拉普拉斯平滑避免概率为零
                feature_probs.append((feature_counts + self.alpha) / (X_c.shape[0] + self.alpha * len(feature_values)))
            self.feature_probabilities.append(feature_probs)

    def predict(self, X):
        y_pred = []
        for xi in X:
            class_scores = []
            for i, c in enumerate(self.classes):
                class_scores.append(np.log(self.class_priors[i]) + \
                                    np.sum(np.log(self.feature_probabilities[i])[np.arange(len(xi)), xi]))

            y_pred.append(self.classes[np.argmax(class_scores)])
        return np.array(y_pred)


# 示例用法
X = np.array([[1, 2], [1, 3], [2, 2], [3, 2], [3, 4]])
y = np.array(['A', 'A', 'B', 'B', 'B'])

def feature_selector(X, y):
    selected_features = []
    for j in range(X.shape[1]):
        unique_values = np.unique(X[:, j])
        if len(unique_values) > 1:
            selected_features.append(j)
    return selected_features

snb_classifier = SemiNaiveBayesClassifier(alpha=1.0)
snb_classifier.fit(X, y, feature_selector=feature_selector)

X_test = np.array([[1, 1], [3, 3]])
y_pred = snb_classifier.predict(X_test)

print("Predictions:", y_pred)

```

贝叶斯网

```python
import numpy as np
from collections import defaultdict

```

```python

class BayesianNetwork:
    def __init__(self):
        self.variables = []
        self.dependencies = defaultdict(list)  # 变量之间的依赖关系
        self.probabilities = {}  # 条件概率表

    def add_variable(self, variable, dependencies):
        self.variables.append(variable)
        self.dependencies[variable] = dependencies

    def add_probability(self, variable, value, prob):
        self.probabilities[(variable, value)] = prob

    def get_parents(self, variable):
        return self.dependencies[variable]

    def get_probability(self, variable, value, evidence):
        parents = self.get_parents(variable)
        key = (variable, value)

        # 在给定的父节点取值情况下，计算变量取值的概率
        parent_values = tuple(evidence[parent] for parent in parents)
        return self.probabilities.get((variable, value, parent_values), 0.0)

    def infer(self, query_variable, query_value, evidence):
        probabilities = {}
        evidence_copy = evidence.copy()

        for value in self.variables[query_variable]:
            evidence_copy[query_variable] = value
            probability = self.calculate_probability(query_variable, query_value, evidence_copy)
            probabilities[value] = probability

        return probabilities

    def calculate_probability(self, query_variable, query_value, evidence):
        probability = 0.0

        for value in self.variables[query_variable]:
            evidence_copy = evidence.copy()
            evidence_copy[query_variable] = value

            # 检查证据的一致性
            if self.check_evidence_consistency(evidence_copy):
                prob = 1.0
                for variable in self.variables:
                    parents = self.get_parents(variable)
                    parent_values = tuple(evidence_copy[parent] for parent in parents)
                    prob *= self.get_probability(variable, evidence_copy[variable], parent_values)

                probability += prob

        return probability

    def check_evidence_consistency(self, evidence):
        for variable, value in evidence.items():
            if variable not in self.variables[variable]:
                return False

            parents = self.get_parents(variable)
            for parent in parents:
                if parent not in evidence:
                    return False

        return True


# 示例用法
network = BayesianNetwork()

network.add_variable('A', [])  # 添加变量 A，没有依赖关系
network.add_probability('A', 'T', 0.3)  # 添加变量 A 的概率
network.add_probability('A', 'F', 0.7)

network.add_variable('B', ['A'])  # 添加变量 B，依赖于变量 A
network.add_probability('B', 'T', [('T', 0.4), ('F', 0.7)])  # 添加变量 B 的概率

network.add_variable('C', ['A'])  # 添加变量 C，依赖于变量 A
network.add_probability('C', 'T', [('T', 0.2), ('F', 0.6)])  # 添加变量 C 的概率

query_variable = 2  # 查询变量的索引
query_value = 'T'  # 查询变量的取值
evidence = {0: 'F', 1: 'T'}  # 证据的值，对应变量的索引与取值

```

```python
probabilities = network.infer(query_variable, query_value, evidence)  # 进行推断，计算概率

print("Probabilities:", probabilities)

```

EM算法

```python
import numpy as np

def expectation_step(data, means, covariances, weights):
    num_clusters = len(means)
    num_samples = data.shape[0]

    # 计算每个样本在每个簇的概率
    probabilities = np.zeros((num_samples, num_clusters))
    for k in range(num_clusters):
        covariance = covariances[k]
        mean = means[k]
        weight = weights[k]
        # 使用多元高斯分布计算概率密度
        probabilities[:, k] = weight * multivariate_normal.pdf(data, mean, covariance)

    # 标准化概率
    probabilities /= np.sum(probabilities, axis=1, keepdims=True)

    return probabilities

def maximization_step(data, probabilities):
    num_clusters = probabilities.shape[1]
    num_samples, num_features = data.shape

    # 更新每个簇的均值、协方差和权重
    means = np.zeros((num_clusters, num_features))
    covariances = []
    weights = np.mean(probabilities, axis=0)

    for k in range(num_clusters):
        probability = probabilities[:, k]
        weight = weights[k]

        # 更新均值
        means[k] = np.sum(probability.reshape(-1, 1) * data, axis=0) / np.sum(probability)

        # 更新协方差矩阵
        centered_data = data - means[k]
        covariance = np.dot((probability.reshape(-1, 1) * centered_data).T, centered_data) / np.sum(probability)
        covariances.append(covariance)

    return means, covariances, weights

def initialize_parameters(num_clusters, num_features):
    # 初始化均值、协方差和权重
    means = np.random.randn(num_clusters, num_features)
    covariances = [np.eye(num_features)] * num_clusters
    weights = np.ones(num_clusters) / num_clusters

    return means, covariances, weights

def EM_algorithm(data, num_clusters, num_iterations):
    num_samples, num_features = data.shape

    # 初始化参数
    means, covariances, weights = initialize_parameters(num_clusters, num_features)

    # 迭代进行E步和M步
    for _ in range(num_iterations):
        # E步：计算后验概率
        probabilities = expectation_step(data, means, covariances, weights)

        # M步：更新参数
        means, covariances, weights = maximization_step(data, probabilities)

    return means, covariances, weights

```

AdaBoost算法

```python
import numpy as np

def adaboost(X, y, num_classifiers):
    num_samples = X.shape[0]
    weights = np.ones(num_samples) / num_samples
    classifiers = []
    alphas = []

    for _ in range(num_classifiers):
        classifier = build_classifier(X, y, weights)  # 根据样本权重训练分类器
        classifiers.append(classifier)
        predictions = classifier.predict(X)
        incorrect = predictions != y

        error = np.sum(weights[incorrect])  # 计算分类器的错误率
        alpha = 0.5 * np.log((1 - error) / error)  # 分类器的权重

        alphas.append(alpha)
        weights *= np.exp(alpha * incorrect)  # 更新样本权重
        weights /= np.sum(weights)  # 归一化样本权重

    return classifiers, alphas

def build_classifier(X, y, weights):
    # 在这里使用任何机器学习算法来训练分类器，例如决策树、支持向量机等
    # 使用带权重的训练数据来获得分类器

    classifier = YourClassifier()  # 替换为您选择的分类器

    classifier.fit(X, y, sample_weight=weights)

    return classifier

```

Bagging算法

```python
import numpy as np

def bagging(X, y, num_classifiers):
    num_samples = X.shape[0]
    classifiers = []

    for _ in range(num_classifiers):
        sample_indices = np.random.choice(num_samples, num_samples, replace=True)  # 有放回地从样本中选择数据
        X_sampled = X[sample_indices]
        y_sampled = y[sample_indices]

        classifier = build_classifier(X_sampled, y_sampled)  # 构建并训练分类器
        classifiers.append(classifier)

    return classifiers

def build_classifier(X, y):
    # 在这里使用任何机器学习算法来训练分类器，例如决策树、支持向量机等
    # 使用训练数据 X 和 y 来获得分类器

    classifier = YourClassifier()  # 替换为您选择的分类器

    classifier.fit(X, y)

    return classifier

```

随机森林

```python
import numpy as np

class DecisionTree:
    def __init__(self, max_depth=None):
        self.max_depth = max_depth
        self.tree = None

    def fit(self, X, y):
        self.tree = self._build_tree(X, y, depth=0)

    def predict(self, X):
        if self.tree is not None:
            return np.array([self._predict_sample(x, self.tree) for x in X])
        else:
            raise Exception("Tree is not built yet.")

    def _build_tree(self, X, y, depth):
        # 检查终止条件
        if depth == self.max_depth or np.unique(y).size == 1:
            return self._leaf_value(y)

        num_features = X.shape[1]
        # 随机选择特征子集
        feature_indices = np.random.choice(num_features, int(np.sqrt(num_features)), replace=False)
        best_feature = self._select_best_feature(X, y, feature_indices)
        best_threshold = self._find_best_threshold(X[:, best_feature], y)

        node = {
            'feature': best_feature,
            'threshold': best_threshold,
            'left_child': None,
            'right_child': None
        }

        # 递归地构建左右子树
        left_indices = X[:, best_feature] <= best_threshold
        right_indices = X[:, best_feature] > best_threshold

        if np.any(left_indices):
            node['left_child'] = self._build_tree(X[left_indices], y[left_indices], depth + 1)

        if np.any(right_indices):
            node['right_child'] = self._build_tree(X[right_indices], y[right_indices], depth + 1)

        return node

    def _select_best_feature(self, X, y, feature_indices):
        best_gain = -np.inf
        best_feature = None

        for feature in feature_indices:
            thresholds = np.unique(X[:, feature])
            for threshold in thresholds:
                gain = self._information_gain(X[:, feature], y, threshold)
                if gain > best_gain:
                    best_gain = gain
                    best_feature = feature

        return best_feature

    def _find_best_threshold(self, feature_values, y):
        thresholds = np.unique(feature_values)
        best_threshold = None
        best_entropy = np.inf

        for threshold in thresholds:
            left_indices = feature_values <= threshold
            right_indices = feature_values > threshold

            if np.any(left_indices) and np.any(right_indices):
                entropy = (np.sum(left_indices) * self._entropy(y[left_indices]) +
                           np.sum(right_indices) * self._entropy(y[right_indices])) / y.size

                if entropy < best_entropy:
                    best_entropy = entropy
                    best_threshold = threshold

        return best_threshold

    def _information_gain(self, feature_values, y, threshold):
        left_indices = feature_values <= threshold
        right_indices = feature_values > threshold

        parent_entropy = self._entropy(y)
        left_entropy = self._entropy(y[left_indices])
        right_entropy = self._entropy(y[right_indices])

        return parent_entropy - (np.sum(left_indices) * left_entropy + np.sum(right_indices) * right_entropy) / y.size

    def _entropy(self, y):
        _, counts = np.unique(y, return_counts=True)
        probabilities = counts / y.size
        return -np.sum(probabilities * np.log2(probabilities + 1e-10))

    def _leaf_value(self, y):
        unique_labels, counts = np.unique(y, return_counts=True)
        return unique_labels[np.argmax(counts)]


class RandomForest:
    def __init__(self, num_estimators, max_depth=None):
        self.num_estimators = num_estimators
        self.max_depth = max_depth
        self.estimators = []

    def fit(self, X, y):
        num_samples = X.shape[0]
        num_features = X.shape[1]
```

```python

        for _ in range(self.num_estimators):
            indices = np.random.choice(num_samples, num_samples, replace=True)
            X_sampled = X[indices]
            y_sampled = y[indices]

            estimator = DecisionTree(max_depth=self.max_depth)
            estimator.fit(X_sampled, y_sampled)
            self.estimators.append(estimator)

    def predict(self, X):
        if len(self.estimators) > 0:
            predictions = np.array([estimator.predict(X) for estimator in self.estimators])
            return np.mean(predictions, axis=0)
        else:
            raise Exception("Random Forest is not built yet.")

```

Stacking算法

```python
import numpy as np


class DecisionTree:
    def __init__(self, max_depth=None):
        self.max_depth = max_depth

    def fit(self, X, y):
        self.tree = self._build_tree(X, y)

    def _calc_gini(self, labels):
        # 计算Gini指数
        _, counts = np.unique(labels, return_counts=True)
        probabilities = counts / len(labels)
        gini = 1 - np.sum(probabilities ** 2)
        return gini

    def _split_data(self, X, y, feature_index, split_value):
        # 根据特征和分割值将数据集分割成两部分
        left_indices = np.where(X[:, feature_index] <= split_value)[0]
        right_indices = np.where(X[:, feature_index] > split_value)[0]
        left_X, left_y = X[left_indices], y[left_indices]
        right_X, right_y = X[right_indices], y[right_indices]
        return left_X, left_y, right_X, right_y

    def _find_best_split(self, X, y):
        best_gini = float('inf')
        best_feature_index = None
        best_split_value = None

        # 遍历所有特征和特征值，选择最佳的分割点
        for feature_index in range(X.shape[1]):
            unique_values = np.unique(X[:, feature_index])
            for value in unique_values:
                left_X, left_y, right_X, right_y = self._split_data(X, y, feature_index, value)

                # 计算分割后的基尼指数总和
                gini = (len(left_y) * self._calc_gini(left_y) +
                        len(right_y) * self._calc_gini(right_y)) / len(y)

                if gini < best_gini:
                    best_gini = gini
                    best_feature_index = feature_index
                    best_split_value = value

        return best_feature_index, best_split_value

    def _build_tree(self, X, y, depth=0):
        # 递归构建决策树

        # 如果达到最大深度或无法再分割，则返回叶节点
        if self.max_depth is not None and depth >= self.max_depth or len(np.unique(y)) == 1:
            return {'leaf': True, 'class': np.argmax(np.bincount(y))}

        feature_index, split_value = self._find_best_split(X, y)
        if feature_index is None:
            # 如果无法找到最佳分割点，则返回叶节点
            return {'leaf': True, 'class': np.argmax(np.bincount(y))}

        # 分割数据集
        left_X, left_y, right_X, right_y = self._split_data(X, y, feature_index, split_value)

        # 递归构建左右子树
        left_tree = self._build_tree(left_X, left_y, depth + 1)
        right_tree = self._build_tree(right_X, right_y, depth + 1)

        # 返回当前节点的划分信息和子树
        return {'leaf': False, 'feature_index': feature_index, 'split_value': split_value,
                'left': left_tree, 'right': right_tree}

    def _traverse_tree(self, node, x):
        # 遍历决策树，预测样本的类别
        if node['leaf']:
            return node['class']

        if x[node['feature_index']] <= node['split_value']:
            return self._traverse_tree(node['left'], x)
        else:
            return self._traverse_tree(node['right'], x)

    def predict(self, X):
        # 预测样本的类别
        predictions = [self._traverse_tree(self.tree, x) for x in X]
        return np.array(predictions)


class StackingClassifier:
    def __init__(self, base_classifiers, meta_classifier):
        self.base_classifiers = base_classifiers
        self.meta_classifier = meta_classifier

    def fit(self, X, y):
        # 训练基础分类器
        self.base_predictions = []
        for classifier in self.base_classifiers:
            classifier.fit(X, y)

            # 对训练集进行预测
            predictions = classifier.predict(X)
            self.base_predictions.append(predictions)

        # 将基础分类器的预测结果堆叠为特征矩阵
        meta_features = np.column_stack(self.base_predictions)

        # 使用元分类器训练
        self.meta_classifier.fit(meta_features, y)

    def predict(self, X):
        base_predictions = []
        for classifier in self.base_classifiers:
                predictions = classifier.predict(X)
                base_predictions.append(predictions)

            # 将基础分类器在测试集上的预测结果堆叠为特征矩阵
            meta_features = np.column_stack(base_predictions)

            # 使用元分类器进行预测
            return self.meta_classifier.predict(meta_features)


# 创建基础分类器和元分类器
tree1 = DecisionTree(max_depth=3)
tree2 = DecisionTree(max_depth=5)
tree3 = DecisionTree(max_depth=7)
meta_classifier = DecisionTree(max_depth=3)

# 创建Stacking分类器
stacking_classifier = StackingClassifier([tree1, tree2, tree3],
meta_classifier)

# 加载示例数据集
X = np.array([[1, 2], [2, 3], [3, 4], [4, 5]])
y = np.array([0, 0, 1, 1])

# 训练Stacking分类器
stacking_classifier.fit(X, y)

# 预测样本类别
test_X = np.array([[1, 2], [2, 3]])
predictions = stacking_classifier.predict(test_X)

print(predictions)

```

k均值算法

```
import numpy as np


def kmeans(X, k, max_iters=100):
    # 随机初始化聚类中心
    indices = np.random.choice(len(X), size=k, replace=False)
    centers = X[indices]

```

```python
    for _ in range(max_iters):
        # 分配样本到最近的聚类中心
        distances = np.linalg.norm(X[:, np.newaxis] - centers, axis=-1)
        labels = np.argmin(distances, axis=-1)

        # 更新聚类中心为各类别样本的均值
        new_centers = np.array([X[labels == i].mean(axis=0) for i in range(k)])

        # 如果聚类中心变化小于阈值，停止迭代
        if np.allclose(centers, new_centers):
            break

        centers = new_centers

    return centers, labels


# 示例用法
X = np.array([[1, 2], [1, 4], [1, 0], [4, 2], [4, 4], [4, 0]])
k = 2

centers, labels = kmeans(X, k)
print("聚类中心：", centers)
print("类别标签：", labels)

```

学习向量量化算法

```python
import numpy as np


class LVQ:
    def __init__(self, n_prototypes, learning_rate=0.1, max_epochs=100):
        self.n_prototypes = n_prototypes
        self.learning_rate = learning_rate
        self.max_epochs = max_epochs

    def initialize_prototypes(self, X, y):
        unique_labels = np.unique(y)
        indices = np.random.choice(len(X), size=self.n_prototypes, replace=False)
        self.prototypes = X[indices]
        self.prototype_labels = y[indices]

    def train(self, X, y):
        self.initialize_prototypes(X, y)

        for epoch in range(self.max_epochs):
            for i, x in enumerate(X):
                # 寻找最近的原型向量
                distances = np.linalg.norm(x - self.prototypes, axis=1)
                nearest_prototype_idx = np.argmin(distances)

                # 更新原型向量
                if self.prototype_labels[nearest_prototype_idx] == y[i]:
                    self.prototypes[nearest_prototype_idx] += self.learning_rate * (x - self.prototypes[nearest_prototype_idx])
                else:
                    self.prototypes[nearest_prototype_idx] -= self.learning_rate * (x - self.prototypes[nearest_prototype_idx])

    def predict(self, X):
        distances = np.linalg.norm(X[:, np.newaxis] - self.prototypes, axis=-1)
        labels = self.prototype_labels[np.argmin(distances, axis=-1)]
        return labels


# 示例用法
X = np.array([[1, 2], [1, 4], [1, 0], [4, 2], [4, 4], [4, 0]])
y = np.array([0, 0, 0, 1, 1, 1])

lvq = LVQ(n_prototypes=2, learning_rate=0.1, max_epochs=50)
lvq.train(X, y)

test_X = np.array([[0, 1], [5, 3]])
predictions = lvq.predict(test_X)

print("预测结果:", predictions)

```

高斯混合聚类

```
import numpy as np


def initialize_parameters(X, n_clusters):
```

```python
    n_samples, n_features = X.shape

    # 初始化高斯分布的均值、方差和权重
    np.random.seed(0)
    means = np.random.rand(n_clusters, n_features)
    variances = np.ones((n_clusters, n_features))
    weights = np.ones(n_clusters) / n_clusters

    return means, variances, weights


def gaussian_pdf(X, mean, variance):
    exponent = -0.5 * np.sum(((X - mean) / variance) ** 2, axis=1)
    pdf = np.exp(exponent) / (np.sqrt(2 * np.pi) * variance).prod(axis=1)
    return pdf


def expectation_step(X, means, variances, weights):
    n_samples, n_features = X.shape
    n_clusters = means.shape[0]

    likelihoods = np.zeros((n_samples, n_clusters))
    for c in range(n_clusters):
        likelihoods[:, c] = gaussian_pdf(X, means[c], variances[c])

    weighted_likelihoods = likelihoods * weights
    responsibilities = weighted_likelihoods / np.sum(weighted_likelihoods,
axis=1, keepdims=True)

    return responsibilities


def maximization_step(X, responsibilities):
    n_samples, n_features = X.shape
    n_clusters = responsibilities.shape[1]

    Nk = np.sum(responsibilities, axis=0)
    means = np.dot(responsibilities.T, X) / Nk[:, np.newaxis]
    variances = np.zeros((n_clusters, n_features))
    for c in range(n_clusters):
        diff = X - means[c]
        variances[c] = np.dot(responsibilities[:, c] * diff.T, diff) /
Nk[c]

    weights = Nk / n_samples

    return means, variances, weights


def gaussian_mixture_clustering(X, n_clusters, max_iterations=100, tol=1e-4):
    means, variances, weights = initialize_parameters(X, n_clusters)

    for i in range(max_iterations):
        # E-step
        responsibilities = expectation_step(X, means, variances, weights)

        # M-step
        new_means, new_variances, new_weights = maximization_step(X, responsibilities)

        # 计算更新前后的均值变化
        mean_change = np.linalg.norm(new_means - means)

        means = new_means
        variances = new_variances
        weights = new_weights

        # 当均值变化小于阈值时停止迭代
        if mean_change < tol:
            break

    # 预测聚类标签
    labels = np.argmax(responsibilities, axis=1)

    return labels


# 示例用法
X = np.array([[1, 2], [1, 4], [1, 0], [4, 2], [4, 4], [4, 0]])
n_clusters = 2

labels = gaussian_mixture_clustering(X, n_clusters)

print("聚类标签:", labels)
```

密度聚类——DBSCAN算法

```python
import numpy as np


def calculate_distance(point1, point2):
    return np.sqrt(np.sum((point1 - point2) ** 2))


def find_neighbors(X, point_index, eps):
    neighbors = []
    for i in range(len(X)):
        distance = calculate_distance(X[point_index], X[i])
        if distance <= eps:
            neighbors.append(i)
    return neighbors


def dbscan(X, eps, min_samples):
    n_samples = len(X)
    visited = np.zeros(n_samples)  # 已访问标记数组，0表示未访问，1表示已访问
    labels = np.zeros(n_samples)  # 聚类标签，-1表示噪声点，大于等于0表示聚类簇编号
    cluster_id = 0

    for i in range(n_samples):
        if visited[i] == 1:
            continue

        visited[i] = 1
        neighbors = find_neighbors(X, i, eps)

        if len(neighbors) < min_samples:
            labels[i] = -1  # 将当前样本标记为噪声点
        else:
            cluster_id += 1
            labels[i] = cluster_id

            # 扩展当前簇
            j = 0
            while j < len(neighbors):
                neighbor_index = neighbors[j]
                if visited[neighbor_index] == 0:
                    visited[neighbor_index] = 1
                    new_neighbors = find_neighbors(X, neighbor_index, eps)
                    if len(new_neighbors) >= min_samples:
                        neighbors.extend(new_neighbors)
                if labels[neighbor_index] == 0:  # 未分配聚类标签的样本
                    labels[neighbor_index] = cluster_id
                j += 1

    return labels


# 示例用法
X = np.array([[1, 2], [1, 4], [1, 0], [4, 2], [4, 4], [4, 0]])
eps = 0.3
min_samples = 3

labels = dbscan(X, eps, min_samples)

print("聚类标签:", labels)

```

层次聚类——AGNES算法

```python
import numpy as np

def calculate_distance(point1, point2):
    return np.sqrt(np.sum((point1 - point2) ** 2))

def AGNES(X, k):
    n_samples = len(X)
    distances = np.zeros((n_samples, n_samples))
    np.fill_diagonal(distances, np.inf)  # 将距离矩阵对角线元素设为无穷大

    labels = np.arange(n_samples)  # 初始聚类标签，每个样本为一个簇

    for i in range(n_samples):
        for j in range(i + 1, n_samples):
            distances[i, j] = calculate_distance(X[i], X[j])
            distances[j, i] = distances[i, j]

    while len(np.unique(labels)) > k:
        min_distance = np.min(distances)
        min_indices = np.argwhere(distances == min_distance)  # 获取距离最小的索引

        cluster_1 = min_indices[0][0]
        cluster_2 = min_indices[0][1]

        # 合并两个最近的簇
        labels[labels == cluster_2] = cluster_1

        # 更新样本间的距离
        distances = np.delete(distances, cluster_2, axis=0)
        distances = np.delete(distances, cluster_2, axis=1)
        for i in range(n_samples-1):
            if i != cluster_1:
                dist_1 = distances[i, cluster_1]
                dist_2 = distances[i, cluster_2]
                distances[i, cluster_1] = min(dist_1, dist_2)
                distances[cluster_1, i] = distances[i, cluster_1]

    return labels

# 示例用法
X = np.array([[1, 2], [1, 4], [1, 0], [4, 2], [4, 4], [4, 0]])
k = 2

labels = AGNES(X, k)

print("聚类标签:", labels)

```

K近邻学习

```
import numpy as np

def calculate_distance(point1, point2):
    return np.sqrt(np.sum((point1 - point2) ** 2))

def k_nearest_neighbors(X_train, y_train, X_test, k):
    n_samples_train = X_train.shape[0]
    n_samples_test = X_test.shape[0]
    n_features = X_train.shape[1]

    y_pred = np.zeros(n_samples_test)

    for i in range(n_samples_test):
        distances = np.zeros(n_samples_train)
        for j in range(n_samples_train):
            distances[j] = calculate_distance(X_test[i], X_train[j])
```

```python

        sorted_indices = np.argsort(distances)  # 按距离升序排列索引

        k_nearest_labels = y_train[sorted_indices[:k]]
        unique_labels, counts = np.unique(k_nearest_labels, return_counts=True)
        majority_label = unique_labels[np.argmax(counts)]

        y_pred[i] = majority_label

    return y_pred

# 示例用法
X_train = np.array([[1, 2], [1, 4], [1, 0], [4, 2], [4, 4], [4, 0]])
y_train = np.array([0, 0, 0, 1, 1, 1])
X_test = np.array([[2, 3], [3, 1]])
k = 3

y_pred = k_nearest_neighbors(X_train, y_train, X_test, k)

print("预测标签:", y_pred)

```

低维嵌入——MDS算法

```python
import numpy as np
from scipy.spatial.distance import squareform, pdist
import matplotlib.pyplot as plt

def mds(data, n_components=2):
    # 计算距离矩阵
    distances = squareform(pdist(data))

    # 计算Gram矩阵
    n = distances.shape[0]
    J = np.eye(n) - np.ones((n, n)) / n
    B = -0.5 * J.dot(distances ** 2).dot(J)

    # 对Gram矩阵进行特征值分解
    eigvals, eigvecs = np.linalg.eigh(B)
    idx = np.argsort(eigvals)[::-1]  # 按特征值降序排序
    eigvals = eigvals[idx]
    eigvecs = eigvecs[:, idx]

    # 提取前n_components个特征向量
    principal_components = eigvecs[:, :n_components]

    # 返回嵌入表示
    return principal_components * np.sqrt(eigvals[:n_components])

# 加载示例数据集（鸢尾花数据集）
from sklearn.datasets import load_iris
iris = load_iris()
X = iris.data
y = iris.target

# 进行低维嵌入
X_mds = mds(X)

# 绘制结果
plt.scatter(X_mds[:, 0], X_mds[:, 1], c=y)
plt.xlabel('MDS Component 1')
plt.ylabel('MDS Component 2')
plt.title('MDS Embedding')
plt.show()

```

主成分分析——PCA算法

```python
import numpy as np
import matplotlib.pyplot as plt


def pca(data, n_components):
    # 计算数据矩阵的均值
    mean = np.mean(data, axis=0)

    # 数据中心化
    centered_data = data - mean

    # 计算协方差矩阵
    cov_matrix = np.cov(centered_data, rowvar=False)

    # 对协方差矩阵进行特征值分解
    eigenvalues, eigenvectors = np.linalg.eig(cov_matrix)

    # 对特征向量按照特征值大小进行排序
    sorted_indices = np.argsort(eigenvalues)[::-1]
    sorted_eigenvectors = eigenvectors[:, sorted_indices]

    # 提取前n_components个主成分
    principal_components = sorted_eigenvectors[:, :n_components]

    # 转换数据到新的空间
    transformed_data = np.dot(centered_data, principal_components)

    # 返回降维后的数据和主成分
    return transformed_data, principal_components


# 生成示例数据
np.random.seed(42)
mu = [2, 3]
cov = [[3, 1], [1, 2]]
data = np.random.multivariate_normal(mu, cov, size=100)

# 执行PCA
n_components = 2
transformed_data, principal_components = pca(data, n_components)

# 绘制结果
plt.scatter(transformed_data[:, 0], transformed_data[:, 1])
plt.xlabel('Principal Component 1')
plt.ylabel('Principal Component 2')
plt.title('PCA')
plt.show()

```

核化线性降维

```
import numpy as np
from scipy.linalg import eigh
from sklearn.metrics.pairwise import pairwise_kernels


def kernel_pca(data, n_components, kernel_type='rbf', gamma=None):
    # 计算核矩阵
    kernel_matrix = pairwise_kernels(data, metric=kernel_type,
gamma=gamma)

    # 中心化核矩阵
    n = kernel_matrix.shape[0]
```

```python
    one_n = np.ones((n, n)) / n
    centered_kernel_matrix = kernel_matrix - one_n.dot(kernel_matrix) -
kernel_matrix.dot(one_n) + one_n.dot(kernel_matrix).dot(one_n)

    # 对中心化核矩阵进行特征值分解
    eigvals, eigvecs = eigh(centered_kernel_matrix)
    idx = np.argsort(eigvals)[::-1]  # 按特征值降序排序
    eigvals = eigvals[idx]
    eigvecs = eigvecs[:, idx]

    # 提取前n_components个特征向量
    principal_components = eigvecs[:, :n_components]

    # 返回降维后的数据和主成分
    return principal_components


# 生成示例数据
np.random.seed(42)
data = np.random.rand(100, 2)  # 2维数据

# 执行核PCA
n_components = 1
kernel_type = 'rbf'  # RBF核函数
gamma = 1.0  # 核参数
transformed_data = kernel_pca(data, n_components, kernel_type, gamma)

# 打印降维后的数据
print(transformed_data)

```

流形学习——Isomap算法

```python
import numpy as np
import matplotlib.pyplot as plt
from scipy.spatial.distance import cdist
from scipy.sparse.linalg import eigs


def lle(data, n_neighbors, n_components):
    # 计算每个样本之间的距离
    dists = cdist(data, data)

    n_samples = data.shape[0]
    W = np.zeros((n_samples, n_samples))

    for i in range(n_samples):
        # 找到每个样本的k个近邻
        indices = np.argsort(dists[i])[1:n_neighbors+1]
        neighbors = data[indices]

        # 计算局部权重矩阵
        X = neighbors - data[i]
        gram_matrix = np.dot(X, X.T)
        weights = np.linalg.solve(gram_matrix, np.ones(n_neighbors))
        weights /= np.sum(weights)  # 归一化权重

        # 填充关系矩阵
        W[i, indices] = weights

    # 计算降维后的数据矩阵
    M = np.eye(n_samples) - W
    eigenvalues, eigenvectors = eigs(np.dot(M.T, M), n_components + 1, which='SM')
    indices = np.argsort(np.abs(eigenvalues.real))[1:n_components + 1]
    embedding = eigenvectors.real[:, indices]

    return embedding


def isomap(data, n_neighbors, n_components):
    # 计算每个样本之间的距离
    dists = cdist(data, data)

    # 找到每个样本的k个近邻
    knn_indices = np.argsort(dists, axis=1)[:, 1:n_neighbors+1]
    knn_dists = dists[np.arange(dists.shape[0])[:, None], knn_indices]

    # 构建距离矩阵
    D = np.zeros_like(dists)
    for i, indices in enumerate(knn_indices):
        D[i, indices] = knn_dists[i]

    # 使用Floyd算法计算最短距离矩阵
    for k in range(data.shape[0]):
        D = np.minimum(D, D[:, k][:, None] + D[k, :][None, :])

    # 计算中心化的距离矩阵
    J = np.eye(data.shape[0]) - np.ones((data.shape[0], data.shape[0])) / data.shape[0]
    B = -0.5 * J.dot(D**2).dot(J)

    # 对中心化距离矩阵进行特征值分解
    eigenvalues, eigenvectors = eigs(B, n_components + 1, which='SM')
    indices = np.argsort(np.abs(eigenvalues.real))[1:n_components + 1]
    embedding = eigenvectors.real[:, indices]

    return embedding


# 生成瑞士卷数据集
np.random.seed(0)
n_samples = 1000
noise = 0.2
t = np.linspace(0, 10, n_samples)
data = np.empty((n_samples, 3))
data[:, 0] = t * np.cos(t)
data[:, 1] = 25 * np.random.rand(n_samples)
data[:, 2] = t * np.sin(t)
data += noise * np.random.randn(n_samples, 3)

# LLE算法降维结果
n_neighbors = 10
n_components = 2
lle_data = lle(data, n_neighbors, n_components)

# Isomap算法降维结果
isomap_data = isomap(data, n_neighbors, n_components)

# 绘制原始数据和降维结果
plt.figure(figsize=(12, 6))

# 原始数据
plt.subplot(131)
plt.scatter(data[:, 0], data[:, 2], c=t, cmap=plt.cm.Spectral)
plt.title('Original Data')

# LLE算法降维结果
plt.subplot(132)
plt.scatter(lle_data[:, 0], lle_data[:, 1], c=t, cmap=plt.cm.Spectral)
plt.title('LLE')

# Isomap算法降维结果
plt.subplot(133)
plt.scatter(isomap_data[:, 0], isomap_data[:, 1], c=t,
cmap=plt.cm.Spectral)
```

```python
plt.title('Isomap')

plt.show()

```

度量学习

```python
import numpy as np
import matplotlib.pyplot as plt
from sklearn.datasets import make_blobs
from sklearn.neighbors import KNeighborsClassifier
from sklearn.metrics import accuracy_score


def metric_learning(X, y):
    # 训练度量学习模型，这里以K最近邻分类器为例
    knn = KNeighborsClassifier(n_neighbors=3)
    knn.fit(X, y)

    return knn


# 生成示例数据集
n_samples = 200
centers = [[-2, 0], [2, 4], [0, -3]]
X, y = make_blobs(n_samples=n_samples, centers=centers, random_state=0)

# 度量学习
model = metric_learning(X, y)

# 使用度量学习后的模型进行预测
y_pred = model.predict(X)

# 计算准确率
accuracy = accuracy_score(y, y_pred)
print(f"准确率: {accuracy}")

# 绘制原始数据和预测结果
plt.figure(figsize=(8, 4))
plt.subplot(121)
plt.scatter(X[:, 0], X[:, 1], c=y, cmap=plt.cm.Set1)
plt.title('Original Data')

plt.subplot(122)
plt.scatter(X[:, 0], X[:, 1], c=y_pred, cmap=plt.cm.Set1)
plt.title('Predicted Labels')

plt.show()

```