Linear Regression

```
#encoding=utf8
import numpy as np
#mse
def mse_score(y_predict,y_test):
    mse = np.mean((y_predict-y_test)**2)
    return mse
#r2
def r2_score(y_predict,y_test):
    '''
    input:y_predict(ndarray):预测值
          y_test(ndarray):真实值
    output:r2(float):r2值
    '''
    #********* Begin *********#

    sst = np.sum((y_test - np.mean(y_test)) ** 2)
    ssr = np.sum((y_predict - y_test) ** 2)

    r2 = 1 - ssr / sst

    #********* End *********#
    return r2

class LinearRegression :
    def __init__(self):
        '''初始化线性回归模型'''
        self.theta = None

    def fit_normal(self,train_data,train_label):
        '''
        input:train_data(ndarray):训练样本
              train_label(ndarray):训练标签
        '''
        #********* Begin *********#
        m,n = train_data.shape
        bias = np.ones(m).reshape(-1, 1)
        new_train_data = np.concatenate((train_data, bias), axis=1)

        self.theta = np.linalg.inv(new_train_data.T @ new_train_data) @ new_train_data.T @ trai
```

```python
        #********* End *********#
        return self.theta

    def predict(self,test_data):
        '''
        input:test_data(ndarray):测试样本
        '''
        #********* Begin *********#
        m,n = test_data.shape
        bias = np.ones(m).reshape(-1,1)
        new_test_data = np.concatenate((test_data, bias), axis=1)

        return (new_test_data @ self.theta).T
        #********* End *********#
```

Logistic Regression

```python
# -*- coding: utf-8 -*-

import numpy as np
import warnings
warnings.filterwarnings("ignore")

def sigmoid(x):
    '''
    sigmoid函数
    :param x: 转换前的输入
    :return: 转换后的概率
    '''
    return 1/(1+np.exp(-x))


def fit(x,y,eta=1e-3,n_iters=10000):
    '''
    训练逻辑回归模型
    :param x: 训练集特征数据，类型为ndarray
    :param y: 训练集标签，类型为ndarray
    :param eta: 学习率，类型为float
    :param n_iters: 训练轮数，类型为int
    :return: 模型参数，类型为ndarray
    '''
    #   请在此添加实现代码   #

    # grad = sum((pi - yi) * xi.T)
    #********** Begin *********#

    m,n = x.shape
    w = np.ones(n)

    for i in range(100):
        y_pred = sigmoid(x @ w)
        grad = (y_pred - y) @ x
        w -= eta * grad

    return w

    #********** End **********#
```

LDA

```
#encoding=utf8
import numpy as np
from numpy.linalg import inv
def lda(X, y):
    '''
    input:X(ndarray):待处理数据
          y(ndarray):待处理数据标签，标签分别为0和1
    output:X_new(ndarray):处理后的数据
    '''
    #********* Begin *********#
    #划分出第一类样本与第二类样本

    X_class1 = X[y == 1]
    X_class2 = X[y == 0]

    #获取第一类样本与第二类样本中心点

    mu_class1 = np.mean(X_class1, axis=0)
    mu_class2 = np.mean(X_class2, axis=0)

    #计算第一类样本与第二类样本协方差矩阵

    Sigma_class1 = (X_class1 - mu_class1).T @ (X_class1 - mu_class1)
    Sigma_class2 = (X_class2 - mu_class2).T @ (X_class2 - mu_class2)

    #计算类内散度矩阵

    Sw = Sigma_class1 + Sigma_class2

    #计算w

    w = inv(Sw) @ (mu_class1 - mu_class2).T
    w = w.T

    #计算新样本集

    X_new = X @ w

    #********* End *********#
    return X_new.reshape(-1, 1)
```

KNN

```python
#encoding=utf8
import numpy as np

class kNNClassifier(object):
    def __init__(self, k):
        '''
        初始化函数
        :param k:kNN算法中的k
        '''
        self.k = k
        # 用来存放训练数据，类型为ndarray
        self.train_feature = None
        # 用来存放训练标签，类型为ndarray
        self.train_label = None


    def fit(self, feature, label):
        '''
        kNN算法的训练过程
        :param feature: 训练集数据，类型为ndarray
        :param label: 训练集标签，类型为ndarray
        :return: 无返回
        '''

        #********* Begin *********#
        self.train_feature = feature
        self.train_label = label
        #********* End *********#


    def predict(self, feature):
        '''
        kNN算法的预测过程
        :param feature: 测试集数据，类型为ndarray
        :return: 预测结果，类型为ndarray或list
        '''
        #********* Begin *********#
        def _predict(test_data):
            distances = [np.sqrt(np.sum((test_data - vec) ** 2)) for vec in self.train_feature]
            nearest = np.argsort(distances)
```

```python
    topK = [self.train_label[i] for i in nearest[:self.k]]
    votes = {}
    result = None
    max_count = 0
    for label in topK:
        if label in votes.keys():
            votes[label] += 1
            if votes[label] > max_count:
                max_count = votes[label]
                result = label
        else:
            votes[label] = 1
            if votes[label] > max_count:
                max_count = votes[label]
                result = label
    return result
predict_result = [_predict(test_data) for test_data in feature]
return predict_result
#********* End *********#
```

Random Forest

```
import numpy as np

#建议代码，也算是Begin-End中的一部分
from collections import  Counter
from sklearn.tree import DecisionTreeClassifier

class RandomForestClassifier():
    def __init__(self, n_model=10):
        '''
        初始化函数
        '''
        #分类器的数量，默认为10
        self.n_model = n_model
        #用于保存模型的列表，训练好分类器后将对象append进去即可
        self.models = []
        #用于保存决策树训练时随机选取的列的索引
        self.col_indexs = []


    def fit(self, feature, label):
        '''
        训练模型
        :param feature: 训练集数据，类型为ndarray
        :param label: 训练集标签，类型为ndarray
        :return: None
        '''

        #************* Begin ************#
        n_samples, n_features = feature.shape
        for i in range(self.n_model):
            sample_indices = np.random.choice(n_samples, int(n_samples/2), replace=True)
            feature_indices = np.random.choice(n_features, int(np.log2(n_features)), replace=Tru
            samples_features = feature[sample_indices, :]
            samples_features = samples_features[:, feature_indices]
            samples_labels = label[sample_indices]

            model = DecisionTreeClassifier()
            model.fit(samples_features, samples_labels)

            self.models.append(model)
```

```
            self.col_indexs.append(feature_indices)
        #************* End **************#


    def predict(self, feature):
        '''
        :param feature:测试集数据，类型为ndarray
        :return:预测结果，类型为ndarray，如np.array([0, 1, 2, 2, 1, 0])
        '''
        #************* Begin ************#
        vote = []

        for i in range(self.n_model):
            model = self.models[i]
            col_indices = self.col_indexs[i]
            sample = feature[:, col_indices]
            vote.append(model.predict(sample))

        y_pred = np.apply_along_axis(lambda x: Counter(x).most_common(1)[0][0], axis=0, arr=np.a

        return y_pred
        #************* End **************#
```

Decision Tree

Gini 系数

```python
import numpy as np

def calcGini(feature, label, index):
    '''
    计算基尼系数
    :param feature:测试用例中字典里的feature，类型为ndarray
    :param label:测试用例中字典里的label，类型为ndarray
    :param index:测试用例中字典里的index，即feature部分特征列的索引。该索引指的是feature中第几个特征，
    :return:基尼系数，类型float
    '''

    #********* Begin *********#
    feature_given_index = feature[:, index].reshape(-1)
    n = feature.shape[0]
    values, counts = np.unique(feature_given_index, return_counts=True)

    gini = 0
    for value in values:
        label_given_value = label[feature_given_index == value]
        _, y_counts = np.unique(label_given_value, return_counts=True)
        y_counts = y_counts / label_given_value.shape[0]
        gini += (1 - np.sum(y_counts * y_counts)) * label_given_value.shape[0] / n

    return gini

    #********* End *********#
```

Information Gain

```python
import numpy as np

def calcInfoGain(feature, label, index):
    '''
    计算信息增益
    :param feature:测试用例中字典里的feature，类型为ndarray
    :param label:测试用例中字典里的label，类型为ndarray
    :param index:测试用例中字典里的index，即feature部分特征列的索引。该索引指的是feature中第几个特征，
    :return:信息增益，类型float
    '''
    # 计算熵
    def calcInfoEntropy(label):
        '''
        计算信息熵
        :param label:数据集中的标签，类型为ndarray
        :return:信息熵，类型float
        '''


        label_set = set(label)
        result = 0
        for l in label_set:
            count = 0
            for j in range(len(label)):
                if label[j] == l:
                    count += 1
            # 计算标签在数据集中出现的概率
            p = count / len(label)
            # 计算熵
            result -= p * np.log2(p)
        return result

    # 计算条件熵
    def calcHDA(feature, label, index, value):
        '''
        计算信息熵
        :param feature:数据集中的特征，类型为ndarray
        :param label:数据集中的标签，类型为ndarray
        :param index:需要使用的特征列索引，类型为int
        :param value:index所表示的特征列中需要考察的特征值，类型为int
        :return:信息熵，类型float
        '''
        count = 0
        # sub_label表示根据特征列和特征值分割出的子数据集中的标签
```

```python
        sub_label = []
        for i in range(len(feature)):
            if feature[i][index] == value:
                count += 1
                sub_label.append(label[i])
        pHA = count / len(feature)
        e = calcInfoEntropy(sub_label)
        return pHA * e

    base_e = calcInfoEntropy(label)
    f = np.array(feature)
    # 得到指定特征列的值的集合
    f_set = set(f[:, index])
    sum_HDA = 0
    # 计算条件熵
    for value in f_set:
        sum_HDA += calcHDA(feature, label, index, value)
    # 计算信息增益
    return base_e - sum_HDA


def calcInfoGainRatio(feature, label, index):
    '''
    计算信息增益率
    :param feature:测试用例中字典里的feature, 类型为ndarray
    :param label:测试用例中字典里的label, 类型为ndarray
    :param index:测试用例中字典里的index, 即feature部分特征列的索引。该索引指的是feature中第几个特征,
    :return:信息增益率, 类型float
    '''

    #********* Begin *********#

    feature_given_index = feature[:, index].reshape(-1)
    _, counts = np.unique(feature_given_index, return_counts=True)
    n = feature.shape[0]
    counts = counts / n
    HA = np.sum(-1 * counts * np.log2(counts))

    IG = calcInfoGain(feature, label, index)

    return IG / HA
```

#********* End *********#