Московский авиационный институт (национальный исследовательский университет)

Факультет информационных технологий и прикладной математики

Кафедра вычислительной математики и программирования

Лабораторная работа №2 по курсу «Искусственный интеллект»

Студент: Я. А. Графчикова
Преподаватель: С. Х. Ахмед
Группа: М8О-408Б

Дата:
Оценка:
Подпись:

Москва, 2019

Лабораторная работа №2

Задача: Требуется реализовать класс на выбранном языке программирования, который реализует один из алгоритмов машинного обучения. Обязательным является наличия в классе двух методов fit, predict. Необходимо проверить работу вашего алгоритма на ваших данных (на таблице и на текстовых данных), произведя необходимую подготовку данных. Также необходимо реализовать алгоритм полиномиальной регрессии, для предсказания значений в таблице. Сравнить результаты с стандартной реализацией sklearn, определить в чем сходство и различие ваших алгоритмов. Замерить время работы алгоритмов.

Номер по списку: 3
Номер варианта: 4

1 Описание

Случайный лес — алгоритм машинного обучения, предложенный Лео Брейманом и Адель Катлер, заключающийся в использовании комитета (ансамбля) решающих деревьев. Алгоритм применяется для задач классификации, регрессии и кластеризации. Основная идея заключается в использовании большого ансамбля решающих деревьев, каждое из которых само по себе даёт очень невысокое качество классификации, но за счёт их большого количества результат получается хорошим.

Алгоритм построения случайного леса:

1. Сгенерируем случайную подвыборку размером N из обучающей выборки
2. Построим дерево решений, строящее регрессию по образцам данной подвыборки, причём в ходе создания очередного узла дерева будем выбирать наилучший признак, относительно которого будет происходить разбиение. Критерием может выступать среднеквадратическая ошибка.
3. Дерево строится до полного исчерпания подвыборки или до заданной максимальной высоты.

Предсказание значений: каждое дерево из леса предсказывает своё значение. В качестве результата регрессии берётся среднее значение по всем деревьям.
Оптимальное количество деревьев для конкретной задачи подбирается таким образом, чтобы минимизировать ошибку на тестовой выборке. Чем выше глубина дерева, тем более точно алгоритм может предсказывать значения по выборке, но при неограниченном росте глубины появляется склонность к переобучению.

2 Случайный лес

Исходный код:

```
class RandomForest():
    def __init__(self, max_depth=5, n_trees=3, sample_size=0.1):
        self.trees = []
        self.n_trees = n_trees
        self.max_depth = max_depth
        self.sample_size = sample_size

    def fit(self, X, y):
        self.trees = []
        for i in range(self.n_trees):
            X_sample, y_sample = self.subsample(X, y, self.sample_size)
            tree = DecisionTree(max_depth=self.max_depth)
            tree.fit(X_sample, y_sample)
            self.trees.append(tree)

    def predict(self, X):
        predictions = np.zeros(len(X))

        for tree in self.trees:
            pred = tree.predict(X)
            predictions += pred
        predictions /= len(self.trees)

        return predictions

    def subsample(self, X, y, ratio):
        sample = []
        sample_y = []
        n_sample = round(len(X) * ratio)
        while len(sample) < n_sample:
            index = random.randrange(len(X))
            sample.append(X[index])
            sample_y.append(y[index])
        return np.array(sample), np.array(sample_y)


class DecisionTree():
    def __init__(self, max_depth=5):
        self.feature = None
        self.label = None
        self.n_samples = None
        self.gain = None
        self.root = None
        self.left = None
```

```
        self.right = None
        self.threshold = None
        self.depth = 0
        self.max_depth = max_depth

    def fit(self, features, target):
        self.root = DecisionTree()
        self.root.build(features, target)
        self.root.prune(self.max_depth, self.root.n_samples)

    def predict(self, features):
        return np.array([self.root.predict_feature(feature) for feature in features])

    def predict_feature(self, feature):
        if self.feature != None:
            if feature[self.feature] <= self.threshold:
                return self.left.predict_feature(feature)
            else:
                return self.right.predict_feature(feature)
        else:
            return self.label

    def build(self, features, target):
        self.n_samples = features.shape[0]

        if len(np.unique(target)) == 1:
            self.label = target[0]
            return

        best_gain = 0.0
        best_feature = None
        best_threshold = None

        self.label = np.mean(target)

        impurity_node = self.mse(target)

        for col in range(features.shape[1]):
            feature_level = np.unique(features[:, col])
            thresholds = (feature_level[:-1] + feature_level[1:]) / 2.0

            for threshold in thresholds:
                target_l = target[features[:, col] <= threshold]
                impurity_l = self.mse(target_l)
                n_l = float(target_l.shape[0]) / self.n_samples

                target_r = target[features[:, col] > threshold]
                impurity_r = self.mse(target_r)
                n_r = float(target_r.shape[0]) / self.n_samples
```

```
                impurity_gain = impurity_node - (n_l * impurity_l + n_r * impurity_r)
                if impurity_gain > best_gain:
                    best_gain = impurity_gain
                    best_feature = col
                    best_threshold = threshold

        self.feature = best_feature
        self.gain = best_gain
        self.threshold = best_threshold
        self.split_node(features, target)

    def split_node(self, features, target):
        features_l = features[features[:, self.feature] <= self.threshold]
        target_l = target[features[:, self.feature] <= self.threshold]
        self.left = DecisionTree()
        self.left.depth = self.depth + 1
        self.left.build(features_l, target_l)

        features_r = features[features[:, self.feature] > self.threshold]
        target_r = target[features[:, self.feature] > self.threshold]
        self.right = DecisionTree()
        self.right.depth = self.depth + 1
        self.right.build(features_r, target_r)

    def mse(self, target):
        return np.mean((target - np.mean(target)) ** 2)

    def prune(self, max_depth, n_samples):
        if self.feature is None:
            return

        self.left.prune(max_depth, n_samples)
        self.right.prune(max_depth, n_samples)

        if self.depth >= max_depth:
            self.left = None
            self.right = None
            self.feature = None
```

В качестве датасета будем использовать акции Tesla. По значениям цены открытия, минимальной цены, максимальной цены и обороте за день попробуем предсказать цену на момент закрытия.

Сравним результаты работы данного алгоритма с алгоритмом, реализованным в модуле sklearn. Также сравним время работы данных алгоритмов. Будем использовать разное количество деревьев в лесу. Отношение размера тестовой выборки к обучающей 3:7.

2 trees mine sklearn
time: 0.7253 0.0065
Explained_variance_score: 0.9986 0.9996
Mean Absolute Error: 2.2144 1.255
Mean Squared Error: 12.198 3.2924

5 trees mine sklearn
time: 1.8963 0.0147
Explained_variance_score: 0.9987 0.9997
Mean Absolute Error: 2.1077 1.1565
Mean Squared Error: 11.8165 2.513

10 trees mine sklearn
time: 3.6303 0.0284
Explained_variance_score: 0.999 0.9998
Mean Absolute Error: 1.8714 1.0403
Mean Squared Error: 9.1174 2.1644

20 trees mine sklearn
time: 7.1685 0.0592
Explained_variance_score: 0.9991 0.9998
Mean Absolute Error: 1.818 0.9306
Mean Squared Error: 8.2133 1.8306

50 trees mine sklearn
time: 18.1883 0.1382
Explained_variance_score: 0.9991 0.9998
Mean Absolute Error: 1.7232 0.866
Mean Squared Error: 7.9108 1.5736

3 Полиномиальная регрессия

В алгоритме полиномиальной регрессии как и в линейной строится многочлен по входным параметрам, однако порядок многочлена может быть выше единицы. При обучении ищутся такие коэффициенты, при которых результат вычисления многочлена был как можно ближе к заданному значению. Для этого используется решение СЛАУ методом наименьших квадратов. При это для решения я пользуюсь PolynomialFeatures и np.linalg.lstsq (решение системы).

Исходный код:

```
class PolynomialRegression:
    def __init__(self, order):
        self.poly = PolynomialFeatures(order)

    def fit(self, X, Y):
        transformed_X = self.poly.fit_transform(X)
        self.coef = np.linalg.lstsq(transformed_X, Y, rcond=None)[0]

    def predict(self, X):
        return np.dot(self.poly.fit_transform(X), self.coef)
```

Ниже представлены результаты тестирования.

Polynomial Regression of 1 order
Time: 0.005145377
Mean Absolute Error: 0.9886351031675271
Mean Squared Error: 1.4690826810203232

Polynomial Regression of 2 order
Time: 0.004508586
Mean Absolute Error: 1.8775020325108662
Mean Squared Error: 2.347439481496505

Polynomial Regression of 3 order
Time: 0.00853866
Mean Absolute Error: 3.0270986572266287
Mean Squared Error: 3.813938258740262

4 Вывод

Выполнив лабораторную работу, я ознакомилась с библиотекой sklearn, изучила и разработала два алгоритма машинного обучения – случайный лес и полиномиальную регрессию, сравнила их работу с библиотечными функциями sklearn.