Московский авиационный институт
(национальный исследовательский университет)

Институт №8 «Информационные технологии и прикладная математика»

Кафедра 806 «Вычислительная математика и программирование»

Курсовой проект по курсу «Методы, средства и технологии мультимедия»

Студент: В. В. Бирюков
Преподаватель: Б. В. Вишняков
Группа: М8О-407Б-19

Дата:
Оценка:
Подпись:

Москва, 2022

Курсовой проект

Основные этапы работы:

1. Выбрать задачу (классификация или регрессия)
2. Выбрать датасет
3. Сделать описание датасета
4. Сделать и описать предпроцессинг над датасетом
5. Выбрать алгоритм
6. Реализовать алгоритм
7. Сделать описание алгоритма
8. Выбрать метрики качества
9. Продемонстрировать полученные результаты
10. Сравнить результат вашего алгоритма с алгоритмом из sklearn
11. Сделать выводы

1 Описание данных

В качестве источника данных выбран датасет Concrete Compressive Strength. Данные содержат информацию о прочности бетона для определенного состава и возраста.

Описание столбцов:

1. Cement – содержание цемента (кг/м^3)
2. Blast Furnace Slag – содержание доменного шлака (кг/м^3)
3. Fly Ash – содержание летучей золы (кг/м^3)
4. Water – содержание воды (кг/м^3)
5. Superplasticizer – содержание пластификатора (кг/м^3)
6. Coarse Aggregate – содержание крупного заполнителя (кг/м^3)
7. Fine Aggregate – содержание мелкого заполнителя (кг/м^3)
8. Age – возраст бетона (дни)
9. Concrete compressive strength – прочность бетона на сжатие (МПа)

2 Описание алгоритма

Градиентный бустинг

Градиентный бустинг — ансамблевая модель машинного обучения, в котором базовые алгоритмы строятся последовательно, причем каждый следующий алгоритм старается уменьшить ошибку текущего ансамбля.

Рассмотрим обоснование градиентного бустинга для задачи регрессии с некоторой функцией потерь $\mathcal{L}(y_i, \hat{y_i})$. Пусть имеется обучающая выборка $X = \{x_i\},\ i = 1...N$ и целевая переменная $Y = \{y_i\},\ i = 1...N$. Для решения задачи будем строить композицию $a(x)$ из базовых алгоритмов $b(x)$.

$$a(x) = a_k(x) = b_1(x) + b_2(x) + \cdots + b_k(x)$$

Композиция строится последовательно – $a_k(x) = a_{k-1}(x) + b_k(x)$ – добавляемый базовый алгоритм b_k обучается так, чтобы улучшить предсказание текущего ансамбля.

$$b_k = \underset{b \in \mathcal{B}}{\operatorname{argmin}} \sum_{i=1}^{N} \mathcal{L}(y_i, a_{k-1}(x_i) + b(x_i))$$

Рассмотрим разложение функции потерь $\mathcal{L}$ в точке $(y_i, a_{k-1}(x) + b_k(x))$ в ряд Тейлора до первого члена в окрестности точки $(y_i, a_{k-1}(x))$

$$\mathcal{L}(y_i, a_{k-1}(x_i)+b(x_i)) \approx \mathcal{L}(y_i, a_{k-1}(x_i))+b(x_i)\frac{\partial \mathcal{L}(y_i, z)}{\partial z}\bigg|_{z=a_{k-1}(x_i)} = \mathcal{L}(y_i, a_{k-1}(x_i))+b(x_i)g_i^{k-1}$$

Таким образом, получаем следующую задачу оптимизации:

$$b_k \approx \underset{b \in \mathcal{B}}{\operatorname{argmin}} \sum_{i=1}^{N} b(x_i) g_i^{k-1}$$

Выражение, которое необходимо минимизировать, есть ни что иное, как скалярное произведение векторов $b(x)$ и g^{k-1}, поэтому его минимизируют значения $b(x_i)$ пропорциональные $-g_i^{k-1}$. То есть на каждой итерации базовый алгоритм $b_i(x)$ надо обучать так, чтобы он приближал значение антиградиента функции потерь для текущего ансамбля, что по сути является применением идеи градиентного спуска.

Аналогично с градиентным спуском, перемещение на значение антиградиента целиком может привести к проскакиванию минимума и потенциально вести к расхождению метода. Поэтому введем скорость обучения $\eta \in (0, 1]$, которая будет использовать для определения вклада базового алгоритма в композицию.

$$a_{k+1}(x) = a_k(x) + \eta b_{k+1}(x)$$

Потенциально, градиентный бустинг можно использовать с любым алгоритмом машинного обучения в качестве базового, но, например, для линейной модели результирующий ансамбль будет представлять из себя линейную комбинацию линейных моделей — линейную модель. Поэтому традиционный базовый алгоритм для градиентного бустинга — решающее дерево.

3 Предобработка данных

Данные не потребовали дополнительной предобработки. Они не содержат пропусков. Распределения признаков выглядят нормально, за исключением большого количества нулей у Furnace Slag, Fly Ash и Superplasticizer, которых однако слишком много, чтобы считать выбросами.

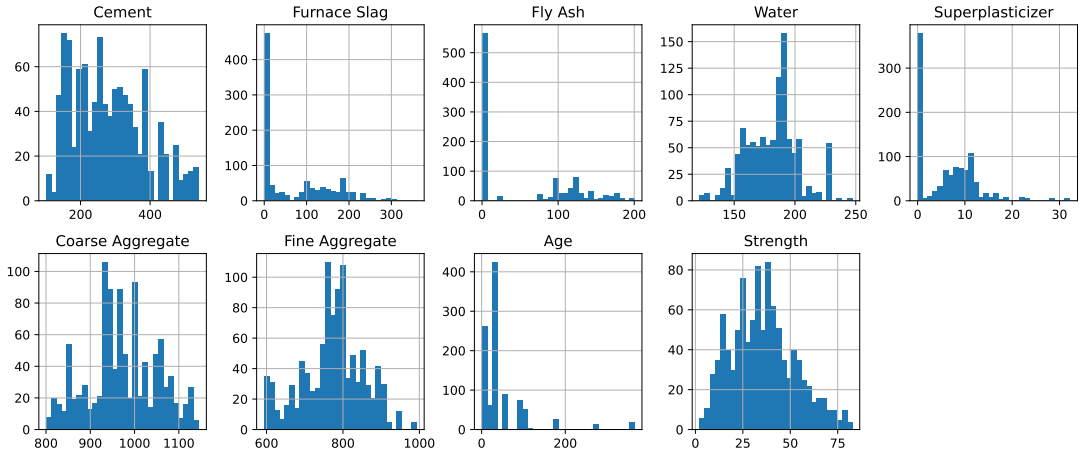


Рис. 1: Гистограммы на основе значений признаков

Мультиколлинеарность также не наблюдается, поэтому в удалении каких-то признаков нет необходимости.

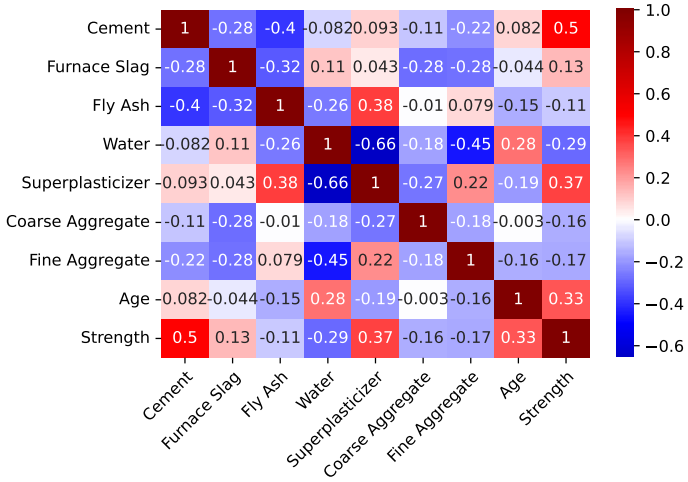


Рис. 2: Матрица корреляции признаков

Единственная обработка данных, которая используется — нормализация при помощи

`sklearn.preprocessing.Normalizer` — встроена в пайплайн каждой модели первым этапом.

4 Реализация алгоритма

Реализация решающего дерева не изменилась по сравнению с лабораторными работами. Настраиваемые гиперпараметры: глубина дерева, минимальный размер листа, способ вычисления ответа (среднее значение или медиана целевой переменной в листе) и критерий построения дерева. Реализованные критерии: среднеквадратичная ошибка, средняя абсолютная ошибка и среднее отклонение Пуассона.

```
class DecisionTree(BaseEstimator, RegressorMixin):
    class Node:
        def __init__(self):
            self.feature = -1
            self.value = None
            self.left = None
            self.right = None

    def __init__(self, min_leaf_size=5, max_depth=None, criterion='mse', answer='mean'
    , features=None):
        self.min_leaf_size = min_leaf_size
        self.max_depth = max_depth
        self.criterion = criterion
        self.answer = answer
        self.features = features

    def fit(self, data, target):
        self.root = self.Node()
        self.process_node(data, target, self.root, np.arange(len(target)), 0)
        return self

    def crit(self, y_sum, y_sq_sum, y_med_sum, n):
        if self.criterion == 'mse':
            return y_sq_sum / n - y_sum ** 2 / n ** 2
        elif self.criterion == 'mae':
            return y_med_sum / n
        elif self.criterion == 'poisson':
            return -y_sum / n * np.log(y_sum / n)

    def process_node(self, data, target, node, ids, depth):
        X = data[ids]
        Y = target[ids]
        n = len(X)
        y_sum = np.sum(Y)
        y_sq_sum = np.sum(Y ** 2)
        y_med = np.median(Y)
```

```
36          y_med_sum = np.sum(np.abs(Y - y_med))
37
38          if (self.max_depth is not None) and depth == self.max_depth or \
39             (self.min_leaf_size is not None) and n <= self.min_leaf_size:
40              if self.answer == 'mean':
41                  node.value = y_sum / n
42              elif self.answer == 'median':
43                  node.value = y_med
44              return
45
46          h = self.crit(y_sum, y_sq_sum, y_med_sum, n)
47          max_value = None
48          max_f = None
49          max_gain = -1
50          best_left_ids = None
51          best_right_ids = None
52          for f in (self.features if self.features is not None else range(data.shape[1])
    ):
53              sort_ids = X[:, f].argsort()
54              left = 1
55              left_sum = Y[sort_ids[0]]
56              left_sq_sum = Y[sort_ids[0]] ** 2
57              left_med_sum = np.abs(Y[sort_ids[0]] - y_med)
58
59              while left < n:
60                  while left < n and X[sort_ids[left-1]][f] == X[sort_ids[left-2]][f]:
61                      left += 1
62                      left_sum += Y[sort_ids[left-1]]
63                      left_sq_sum += Y[sort_ids[left-1]] ** 2
64                      left_med_sum += np.abs(Y[sort_ids[left-1]] - y_med)
65                  if left == n:
66                      break
67
68                  left_h = self.crit(left_sum, left_sq_sum, left_med_sum, left)
69                  right_h = self.crit(y_sum - left_sum, y_sq_sum - left_sq_sum,
    y_med_sum - left_med_sum, n - left)
70
71                  gain = h - (left * left_h + (n - left) * right_h) / n
72                  if gain > max_gain:
73                      max_gain = gain
74                      max_value = X[sort_ids[left-1]][f]
75                      max_f = f
76                      best_left_ids = sort_ids[:left]
77                      best_right_ids = sort_ids[left:]
78
79                  left += 1
80                  left_sum += Y[sort_ids[left-1]]
81                  left_sq_sum += Y[sort_ids[left-1]] ** 2
82                  left_med_sum += np.abs(Y[sort_ids[left-1]] - y_med)
```

```
83
84          if max_value is None:
85              if self.answer == 'mean':
86                  node.value = y_sum / n
87              elif self.answer == 'median':
88                  node.value = y_med
89              return
90
91          node.feature = max_f
92          node.value = max_value
93          node.left = self.Node()
94          node.right = self.Node()
95
96          self.process_node(X, Y, node.left, best_left_ids, depth+1)
97          self.process_node(X, Y, node.right, best_right_ids, depth+1)
98
99      def predict(self, data):
100         res = np.ndarray(data.shape[0])
101         for i, obj in enumerate(data):
102             node = self.root
103             while node.feature != -1:
104                 if obj[node.feature] > node.value:
105                     node = node.right
106                 else:
107                     node = node.left
108             res[i] = node.value
109         return res
```

Градиентный бустинг реализован с гиперпараметрами: количество деревьев, скорость обучения и функция потерь. Из функций потерь реализована только среднеквадратичная ошибка. Присутствует возможность обучать каждое дерево на подмножестве признаков и данных, но такой вариант показал себя плохо на данной задаче.

```
1  class GradientBoosting(BaseEstimator, RegressorMixin):
2      def __init__(self, loss='mse', n_estimators=100, lr=0.1, max_features=1, subsample
       =1, **estimator_params):
3          self.loss = loss
4          self.n_estimators = n_estimators
5          self.lr = lr
6          self.max_features = max_features
7          self.subsample = subsample
8          self.estimator_params = estimator_params
9
10     def negative_gradient(self, target, pred):
11         if self.loss == 'mse':
12             return target - pred
13
14     def fit(self, data, target):
15         features = np.arange(data.shape[1])
16         if self.max_features == 'sqrt':
```

```
            max_features = int(np.floor(np.sqrt(len(features))))
        else:
            max_features = int(np.floor(len(features) * self.max_features))
        indexes = np.arange(len(data))
        samples = int(np.floor(self.subsample * len(data)))

        self.estimators = []
        pred = np.zeros(data.shape[0])
        for _ in range(self.n_estimators):
            if (self.max_features < 1):
                np.random.shuffle(features)

            self.estimators.append(DecisionTree(features=features[:max_features], **
    self.estimator_params))
            cur_target = self.negative_gradient(target, pred)
            if self.subsample < 1:
                idx = np.random.choice(indexes, (samples, ), replace=False)
                self.estimators[-1].fit(data[idx], cur_target[idx])
            else:
                self.estimators[-1].fit(data, cur_target)
            pred += self.lr * self.estimators[-1].predict(data)
        return self

    def predict(self, data):
        pred = np.zeros(data.shape[0])
        for tree in self.estimators:
            pred += self.lr * tree.predict(data)
        return pred
```

5 Результаты работы

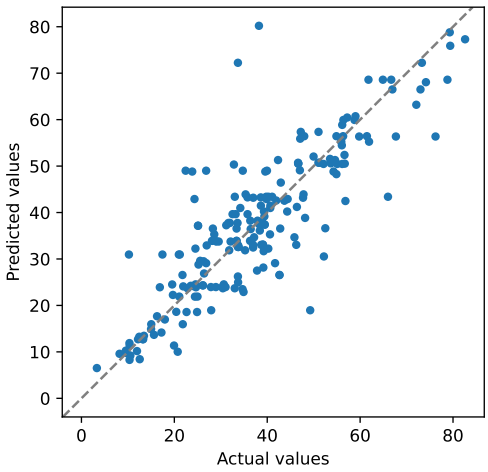
В качестве метрик качества выбраны стандартные метрики для регрессии — среднеквадратичная и средняя абсолютная ошибки, а также максимальная ошибка и коэффициент детерминации R^2.

В отличие от случайного леса, которому необходимы деревья максимальной глубины, градиентному бустингу нужны не переобученные деревья. Поэтому сначала определим оптимальные гиперпараметры одиночного дерева при помощи кросс-валидации.

Оптимальная глубина дерева: 20, минимальный размер листа: 5, критерий построения дерева: среднеквадратичная ошибка, способ вычисления ответа: среднее значение.

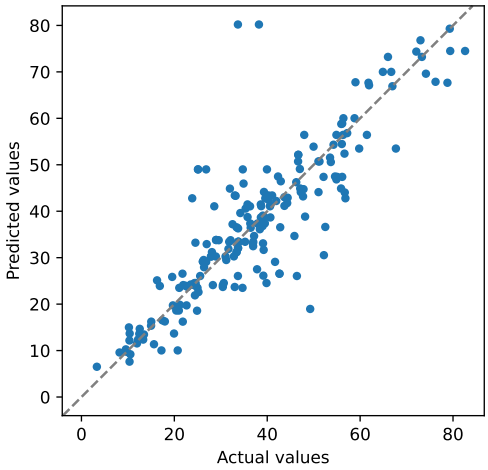
Одиночное дерево показывает следующие результаты:

```
Max error: 41.984262068
MAE: 5.772610978436216
MSE: 73.80822161902006
R^2: 0.7168804103896775
```



Сравним его с деревом из sklearn с теми же параметрами:

```
Max error: 46.51205096
MAE: 5.184229691927071
MSE: 67.95003686449105
R^2: 0.7393517127348934
```



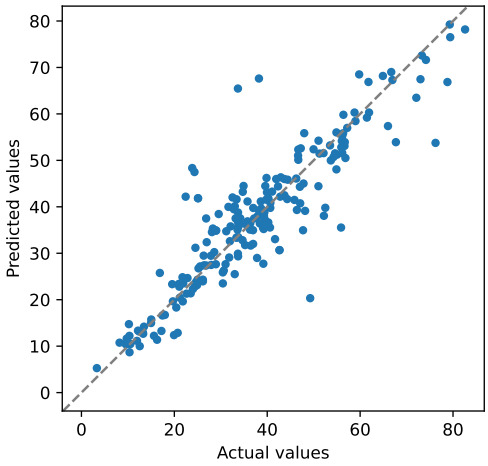
Результаты довольно похожи.

Будем использовать получившиеся параметры дерева для градиентного бустинга, гиперпараметры же самого бустинга снова определим через кросс-валидацию.

Оптимальное число деревьев: 80, скорость обучения: 0.1.

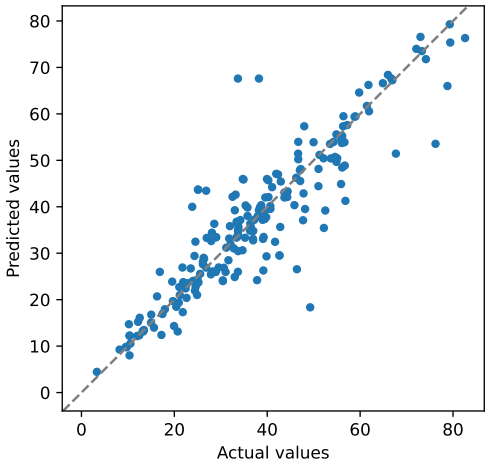
Градиентный бустинг показывает следующие результаты:

```
Max error: 31.724549658115357
MAE: 4.5016877121038945
MSE: 50.426762544929396
R^2: 0.8065689159834937
```



Сравним с градиентным бустингом из sklearn с теми же параметрами:

```
Max error: 31.774143301871298
MAE: 4.478631496353804
MSE: 48.81379503539908
R^2: 0.8127560681643196
```



Результаты очень похожи, разница в ошибках порядка десятых, R^2 отличается меньше чем на 0.01.

6 Выводы

В ходе выполнения курсового проекта я познакомился с алгоритмом машинного обучения градиентный бустинг. До того как я узнал принцип его работы, он казался очень сложным. Но оказалось, что в его основе лежит простая идея дообучения с учетом ошибок предыдущей модели. На выбранном датасете градиентный бустинг показывает довольно хорошие результаты. Возможно при более тонкой настройке их можно улучшить еще больше.