Московский авиационный институт
(национальный исследовательский университет)

Институт №8 «Информационные технологии и прикладная математика»

Лабораторная работа №2 по искусственному интеллекту

«Алгоритмы классификации»

6 семестр

Студент: Стифеев Евгений Михайлович
Группа: М8О-306Б-17
Руководитель: Ахмед Самир Халид
Дата: 30.05.20

Москва, 2020

Оглавление

Условие

Необходимо реализовать алгоритмы машинного обучения. Применить данные алгоритмы на наборы данных, подготовленных в первой лабораторной работе. Провести анализ полученных моделей, вычислить метрики классификатора. Произвести тюнинг параметров в случае необходимости. Сравнить полученные результаты с моделями, реализованными в *scikit-learn*. Аналогично построить метрики классификации. Показать, что полученные модели не переобучились. Также необходимо сделать выводы о применимости данных моделей к вашей задаче.

1. Логистическая регрессия
2. *KNN*
3. *SVM*
4. Дерево решений
5. *Random Forest*

3 или 5 алгоритмы нужно реализовать в соответствии со своим вариантом $N = 24$:

$N \,\%\, 2 + 1 = 24 \,\%\, 2 + 1 = 1$ *(SVM)*

Логистическая регрессия

Алгоритм

Подобно линейной регрессионной модели нужно подсчитать взвешенную сумму входных признаков (плюс член смещения), но взамен выдачи результата напрямую сначала вычисляется *логистика*:

$$\hat{p} = h_\theta(\boldsymbol{x}) = \sigma(\theta^T \cdot \boldsymbol{x})$$

Логистика, также называемая *логитом* (*logit*) и обозначаемая $\sigma(\cdot)$:

$$\sigma(t) = \frac{1}{1 + e^{-t}}$$

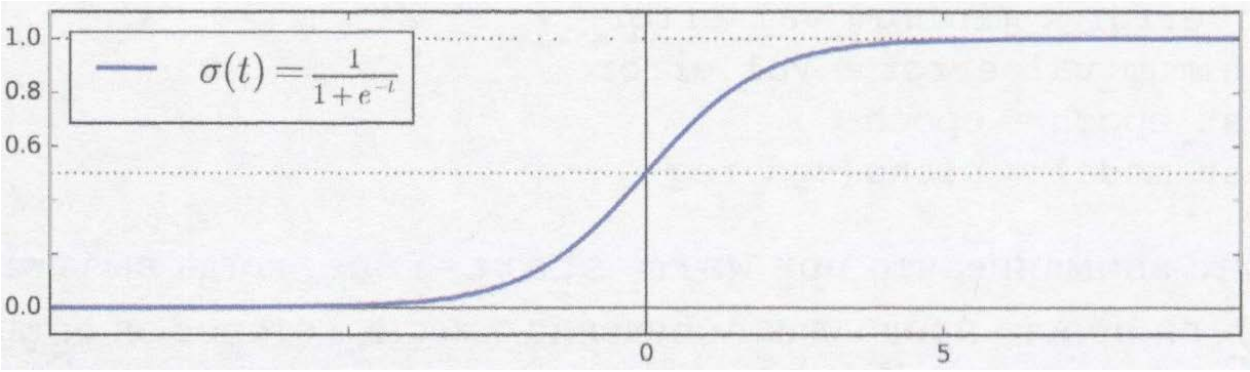

Рис. 1 - Логистическая функция

После этого можно легко выработать прогноз:

$$\hat{y} = \begin{cases} 0, \text{если } \hat{p} < 0.5, \\ 1, \text{если } \hat{p} \geq 0.5. \end{cases}$$

Функция издержек для одиночного образца:

$$c(\theta) = \begin{cases} -\ln(\hat{p}), \text{если } y = 1, \\ -\ln(1 - \hat{p}), \text{если } y = 0. \end{cases}$$

Функция издержек логистической регрессии (логарифмическая потеря):

$$J(\theta) = -\frac{1}{m}\sum_{i=1}^{m}\left[y^{(i)} \ln\left(\hat{p}^{(i)}\right) + \left(1 - y^{(i)}\right) \ln\left(1 - \hat{p}^{(i)}\right)\right].$$

В аналитическом виде пока нет решений для поиска оптимальных значений θ, в отличие от линейной регрессии, но функция издержек является выпуклой, поэтому даже простой градиентный спуск гарантировано отыщет глобальный минимум.

Реализация

Всякого рода МО вычисления, связанные с поиском минимума функции издержек посредством градиентного спуска и подобных алгоритмов безусловной оптимизации, принято проводить на *GPU*, т. к. они связана с большими матричными вычислениями, т. ч. и я не ударил в грязь лицом и реализовал на алгоритм посредством построения графа вычисления в *tensorflow*.

Непосредственно код для построения графа читатель может посмотреть, открыв скрипт *mushrooms.py*, начиная с 300 строки. Я же отмечу, что первоначально веса θ, я инициализировал случайно с помощью усеченного нормального (гауссова) распределения со стандартным отклонением $\sigma = \frac{2}{\sqrt{n_{inputs}+1}}$ (инициализация Ксавье), а $b = 0$. Это немного помогает алгоритму лучше сходиться. Функцию издержек я оптимизировал с помощью *AdamOptimizer*'а при *learning_rate* = 0.01.

Обучение и метрики

Мой первый датасет *mushrooms* имеет атрибут $class = \{e, p\}$ (съедобный, несъедобный) соответственно его значение я и буду прогнозировать.

Здесь и далее в качестве метрики – меры точности – я буду использовать функцию:

$$acc = \frac{1}{m}\sum_{i=1}^{m}\sigma\left(\hat{y}^{(i)}, y^{(i)}\right), \text{где}$$

$$\sigma\left(\hat{y}^{(i)}, y^{i}\right) = \begin{cases} 1, \text{если } \hat{y}^{(i)} = y^{i} \\ 0, \text{если } \hat{y}^{(i)} \neq y^{i} \end{cases}$$

Эту функцию можно легко интерпретировать, как вероятность получить правильный результат при классификации образца.

Метрики для обучения табличного датасета *mushrooms* (8124 образца, 42 атрибута, из них 7311 обучающих и 813 тестовых мини-пакетами (*batch*) по 50 образцов) в течение 4-х эпох:

Эпоха	Мини-пакетов	Средняя точность на обучающем датасете	Средняя точность на тестовом датасете
1	147	0.90912	0.89921
2	147	0.98	0.97495
3	147	0.98857	0.98256

4	147	0.99143	0.98372

Из таблицы легко видеть, что модель не переобучилась.

Сравнение с *LogisticRegression* из *ScikitLearn*:

Модель	**Точность на тренировочных данных**	**Точность на тестовых данных**
Моя	0.99398166	0.98400986
LogisticRegression	1.0	1.0

Метод *k*-ближайших соседей

Алгоритм

Согласно этому методу объект присваивается тому классу, который является наиболее распространённым среди k соседей данного элемента, классы которых уже известны.

Под соседями подразумеваются образцы, которые наиболее близки к тому, для которого требуется определить класс, т.е. расстояние от них до образца является минимальным.

Функция расстояния вообще говоря может быть любой (но, конечно, удовлетворяющая аксиомам метрики). Я, например, использовал классическую евклидову метрику:

$$\rho(\boldsymbol{x}, \boldsymbol{y}) = \sqrt{\sum_{i=1}^{n} (x_i - y_i)^2}$$

Этот алгоритм имеет всякие модификации, которые работают лучше, например, взвешенный способ, но я не стал усложнять.

Понятное дело, что перед применением этого алгоритма, нужно тщательным образом выделить только те атрибуты, которые правильным образом влияют на класс, т.е. способствует группировке образцов в страты, например, данные в моём втором датасете *wine*, как показано на рис. 2.

Особенно важным аспектом в этом алгоритме является масштабирование признаков.

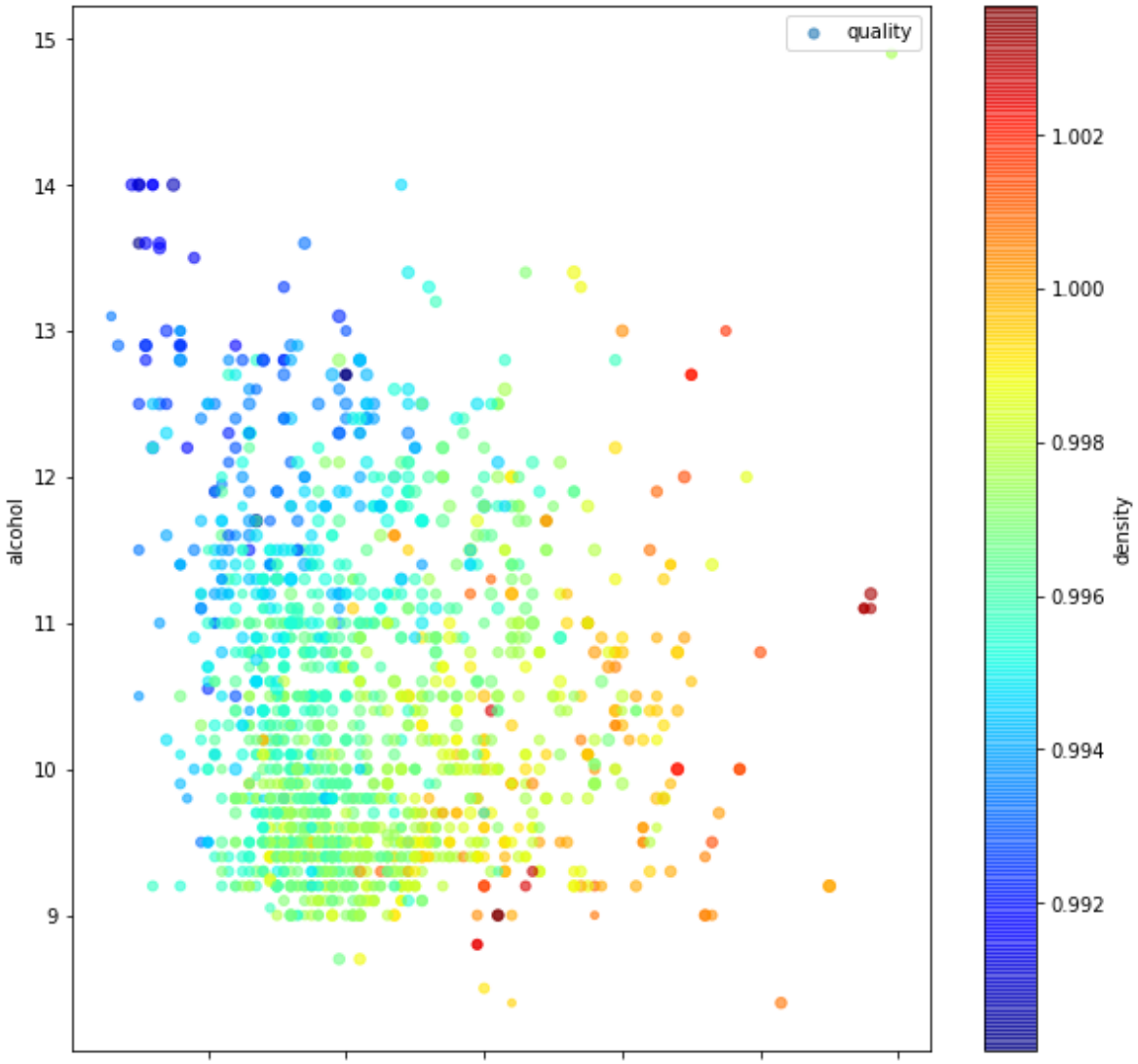

Рис. 2 - Группировка объектов в страты

Реализация

Здесь особо нечего описывать, разве что, этот алгоритм не требует предварительного обучения и весь код является довольно компактным (*wine.py* 107 строка), т. ч. его даже можно привести в настоящем отчёте.

```
k = 7

def KNN_predict(X, y, target, k=1): # предсказание
    dist = []
    for i in range(target.shape[0]):
        dist.append(np.sqrt(np.sum(np.square(X - target[i]), axis=1)))
    distanses = np.stack(dist) # (batch_size, train_samples)
    ans = np.zeros(target.shape[0], dtype=np.int32)
    for i in range(target.shape[0]):
        sort_k_neigbours = sorted(range(distanses.shape[1]), key=lambda x: distanses[i,
x])[:k]
        counts = {i: 0 for i in set(y[sort_k_neigbours])}
        for j in sort_k_neigbours: # проходим по всем таким соседям
```

```
            counts[y[j]] += 1
        m = 0
        l = 0
        for j in counts:
            if counts[j] > m:
                m = counts[j]
                l = j
        ans[i] = l
    return ans
```

Метрики

Данный алгоритм я применял к датасету *wine* (1599 образцов, 12 атрибутов, из них 1439 обучающих и 160 тестовых) для предсказания атрибута $quality = \{3, 4, 5, 6, 7, 8\}$.

Модель	***k***	**Точность на обучающем наборе**	**Точность на тестовом наборе**
Моя	1	1.0	0.7125
KNeighborsClassifier		1.0	0.7125
Моя	3	0.77	0.625
KNeighborsClassifier		0.769	0.63
Моя	5	0.712	0.675
KNeighborsClassifier		0.712	0.675
Моя	7	0.68	0.6875
KNeighborsClassifier		0.68	0.6875
Моя	9	0.656	0.65
KNeighborsClassifier		0.656	0.65

Метод опорных векторов

Алгоритм

Фундаментальная идея, лежащая в основе *SVM* заключается в том, что линейно разделять данные лучше таким образом, чтобы граница решения была как можно дальше от образцов разных классов (рис. 3).

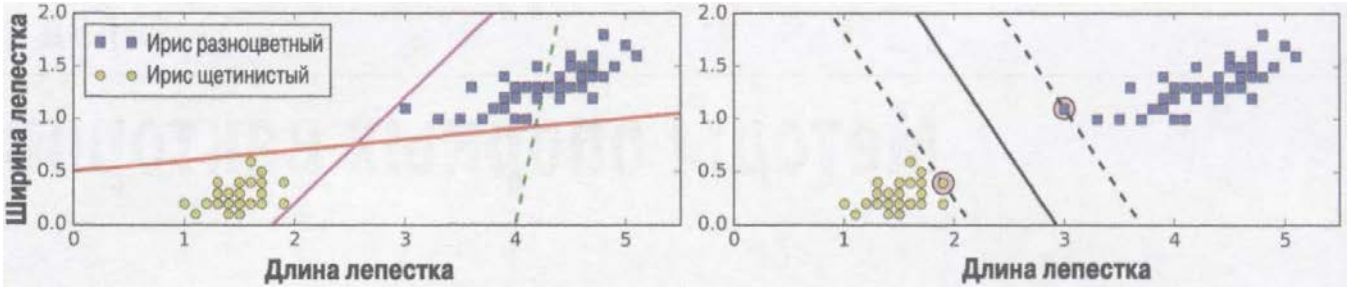

Рис. 3 - Классификация с широким зазором

На графике слева приведены границы решений трех возможных линейных классификаторов. Модель, граница решений которой представлена пунктирной линией, до такой степени плоха, что даже не разделяет классы надлежащим образом. Остальные две модели прекрасно работают на данном обучающем наборе, но их границы решений настолько близки к образцам, что эти модели, вероятно, не будут выполняться так же хорошо на новых образцах.

По контрасту сплошной линией на графике справа обозначена граница решений классификатора *SVM*; линия не только разделяет два класса, но также находится максимально возможно далеко от ближайших обучающих образцов.

Классификатор *SVM* устанавливает самую широкую, какую только возможно, полосу (представленную параллельными пунктирными линиями) между классами. Это называется классификацией с широким зазором (*large margin classification*).

Данная модель имеет очень много модификаций, я остановлюсь на *линейном классификаторе с жёстким зазором*.

Прогноз линейного классификатора *SVM*:

$$\hat{y} = \begin{cases} 0, \text{если } \boldsymbol{w}^T \cdot \boldsymbol{x} + b < 0, \\ 1, \text{если } \boldsymbol{w}^T \cdot \boldsymbol{x} + b \geq 0 \end{cases}$$

Здесь член смещения b не входит в веса $\boldsymbol{w}$ и рассматривается отдельно.

Если обозначить $t^{(i)} = -1$ для всех отрицательных образцов (т. е. когда $y^{(i)} = 0$) и $t^{(i)} = 1$ для всех положительных, то для нахождения нужных весов, достаточно решить задачу квадратичного программирования с линейными ограничениями:

$$\begin{cases} \min\limits_{\boldsymbol{w},b} \frac{1}{2}\boldsymbol{w}^T \cdot \boldsymbol{w}, \\ t^{(i)}\left(\boldsymbol{w}^T \cdot \boldsymbol{x}^{(i)} + b\right) \geq 1 \text{ для } i = 1,2,\dots,m \end{cases}$$

Минимизировать нужно $\frac{1}{2}\boldsymbol{w}^T \cdot \boldsymbol{w} = \frac{1}{2}\|\boldsymbol{w}\|^2$, а не $\|\boldsymbol{w}\|^2$, т. к. первый вариант имеет более простой градиент, а также $\|\boldsymbol{w}\|$ не дифференцируется для $\boldsymbol{w} = 0$.

Реализация

Для решения задачи условной минимизации я воспользовался функцией *minimize* библиотеки *scipy*.

Код довольно прост, т. ч. просто приведу его здесь (*mushrooms.py* 388 строка):

```
1.  n_inputs = X_train.shape[1] # число входных признаков
2.
3.  np.random.seed(42)
4.  stddev = 2 / np.sqrt(n_inputs + 1) # стандартное отклонение
5.  W = np.random.normal(size = n_inputs, scale=stddev) # инициализация весов с
        помощью усечённого гауссова распределения (n_inputs)
6.  b = 0
7.
8.  def SVM_predict(X): # прогноз линейного классификатора SVM
9.      return np.where(np.dot(X, W + b) < 0, np.zeros(X.shape[0]),
        np.ones(X.shape[0]))
10.
11. def f(x): # функция для оптимизации
12.     return np.sum((x[:-1] ** 2)) / 2
13.
14. def grad(x): # её градиент
15.     return np.concatenate((x[:-1], [0]))
16.
17. def hess(x): # её матрица Гесса
18.     return np.diag(np.concatenate((np.ones(len(x) - 1), [0])))
19.
20. def SVM_train(X, y, maxiter=10): # обучение
21.     global W, b
22.     m = X.shape[0] # число образцов
23.     t = np.where(y == 0, -np.ones_like(y), np.ones_like(y)).reshape((-1, 1))
24.     A = np.hstack((X, b * np.ones([m, 1])))
25.     A = t * A
26.     linear_constraint = LinearConstraint(A, np.ones(m), np.ones(m) * np.inf) #
        ограничения на переменные
27.     x0 = np.concatenate((W, [b]))
28.     res = minimize(f, x0, method='trust-constr', jac=grad, hess=hess,
29.                    constraints=[linear_constraint],
30.                    options={'verbose': 2, 'maxiter': maxiter})
```

```
31.     W = res.x[:-1]
32.     b = res.x[-1]
```

Обучение и метрики

Обучать буду на датасете mushrooms при тех же условиях, но не на мини-пакетах (*batch*), а сразу на всём, т. к. способ условной оптимизации, который я выбрал, а именно *trust-constr*, не подходит для пакетного обучения.

Сравнение с *LinearSVC(C=1, loss="hinge")*:

Модель	**Точность на тренировочных данных**	**Точность на тестовых данных**
Моя	1.0	1.0
LinearSVC	1.0	1.0

Вполне достойный результат!

Дерево принятия решений

Алгоритм

Я реализовал алгоритм *CART*, с поддержкой гиперпараметров *min_samples_split* – минимальное число образцов в узле для расщепления, *min_samples_leaf* – минимальное число образцов в листе, *max_depth* – миксимальная глубина дерева.

Если в вкратце, то алгоритм начинает с полного обучающего набора и пытается расщепить его на два поднабора так, чтобы в итоге получилось дерево. Для расщепления нужно подобрать оптимальные параметры: k – номер одного из атрибутов (не только не прогнозируемого класса (!)) и число t_k так, чтобы при спуске по дереву с образцом x для прогноза, можно было руководствовать правилом (рис. 4):

Если $x_k \leq t_k$, то спускаться влево.

Если $x_k > t_k$, то спускаться вправо.

Если мы в листе, то прогнозируемый класс соответствует наиболее часто встречающейся метке среди образцов в листе.

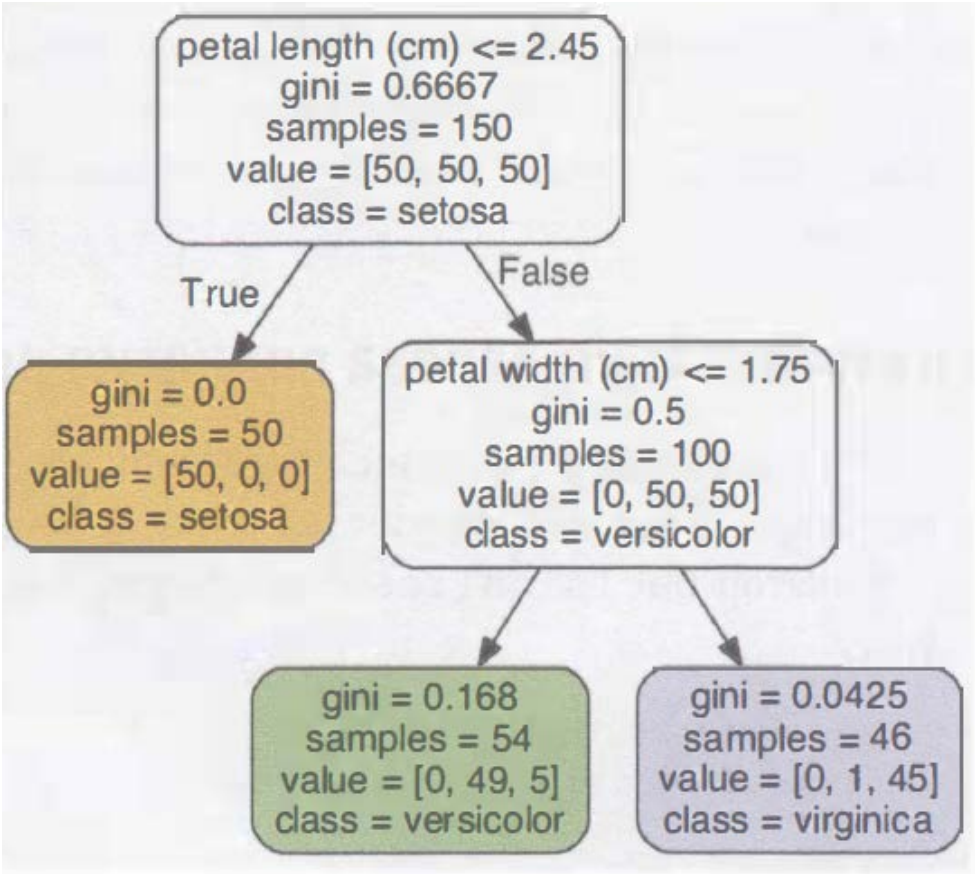

Рис. 4 - Дерево принятия решения iris_tree

Чтобы подобрать параметры k и t_k нужно минимизировать функцию:

$$Gini_{split} = \frac{L}{N} \cdot \left(1 - \sum_{i=1}^{n} \left(\frac{l_i}{L}\right)^2\right) + \frac{R}{N} \cdot \left(1 - \sum_{i=1}^{n} \left(\frac{r_i}{R}\right)^2\right) \to min,$$

или после нехитрых преобразований максимизировать:

$$\tilde{G}_{split} = \frac{1}{L} \cdot \sum_{i=1}^{n} l_i^2 + \frac{1}{R} \cdot \sum_{i=1}^{n} r_i^2 \to max.$$

Реализация

Полный код можно посмотреть в *mushrooms.py* 452 строка или 153 в *wine.py*.

Целевую функцию я максимизировал перебором, но довольно *хитрым*, т. ч. суммарное время на одну максимизацию не превысило $O(n \cdot m \cdot$ число классов). На самом деле можно и быстрее, т. ч. оптимизации тут есть куда стремиться.

Также я храню в узле значения, подобные тем, которые показаны в узлах дерева на рис. 4 для контроля результата.

Обучение и метрики

На датасете *mushrooms* с гиперпараметрами:

$$min\_samples\_leaf = 10$$

$$max\_depth = 30$$

$$min\_samples\_split = 5$$

Модель	**Точность на тренировочных данных**	**Точность на тестовых данных**
Моя	0.896	0.891
DecisionTreeClassifier	0.99	0.998

На датасете *wine* с гиперпараметрами:

$$min\_samples\_leaf = 1$$

$$max\_depth = 50$$

$$min\_samples\_split = 2$$

Модель	Точность на тренировочных данных	Точность на тестовых данных
Моя	0.83	0.60
DecisionTreeClassifier	1.0	0.675

Замечание: собственно, это лучшие результаты, которые мне удалось достичь при различных гиперпараметрах. На мой взгляд, проблема в том, что *wine* – это учебный датасет для задачи линейной регрессии для совсем другого атрибута и он является нерепрезентативным для *quality*, чего стоит только вот эта статистика по значениям *quality* для всего датасета:

Значение	Количество образцов
3	10
4	53
5	681
6	638
7	199
8	18

Но тем не менее было интересно поработать и с такими данными.

Выводы

Для *mushrooms* хорошие результаты показали *SVM* и логистическая регрессия, для *wine*, скорее всего, требуется алгоритм помощнее, такой как, например, случайный лес, но он выходит за рамки данной ЛР.