Московский авиационный институт
(Национальный исследовательский университет)
Факультет прикладной математики и информатики
Кафедра вычислительной математики и программирования

Лабораторная работа № 1,2

по курсу «Машинное обучение»

Преподаватель: Ахмед Самир Халид
Студент: Чурсина Н.А.
Группа: 80-308Б
Оценка:

Москва, 2021

Задание

1 ЛР:

Найти себе набор данных (датасет), для следующей лабораторной работы, и проанализировать его. Выявить проблемы набора данных, устранить их. Визуализировать зависимости, показать распределения некоторых признаков. Реализовать алгоритмы К ближайших соседа с использованием весов и Наивный Байесовский классификатор и сравнить с реализацией библиотеки sklearn.

2 ЛР:

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.

Вариант: Random Forest

Описание алгоритмов

1. Алгоритм n-ближайших соседей (KNN)

Пусть имеется набор данных, состоящий из k наблюдений $Xi(i=1,...,k)$, для каждого из которых задан класс $Cj(j=1,...,m)$. Тогда на его основе может быть сформировано обучающее множество из пар Xi,Cj.

Алгоритм KNN можно разделить на два этапа:

- Обучение: алгоритм запоминает векторы признаков наблюдений и их метки классов. Также задаётся параметр алгоритма n, который задаёт число «соседей», которые будут использоваться при классификации.

- Классификация: предъявляется новый объект, для которого метка класса не задана. Для него определяются n ближайших (в смысле некоторой метрики) предварительно классифицированных наблюдений. Затем выбирается класс, которому принадлежит большинство из n ближайших примеров-соседей, и к этому же классу относится классифицируемый объект.

В моей реализации используется стандартный алгоритм с использованием Евклидовой нормы.

2. Гауссовский Наивный Байесовский классификатор

Основная идея — построить классификатор в предположении того, что функция *p(Xi, Cj)* известна для каждого класса и равна плотности многомерного нормального (гауссовского) распределения:

$$p(x_i \mid c_j) = \frac{1}{\sqrt{2\pi\sigma_{i,j}^2}} e^{-\frac{1}{2}\left(\frac{x_i - \mu_{i,j}}{\sigma_{i,j}}\right)^2} \text{ for } i = 1, 2 \text{ and } j = 1, 2, 3$$

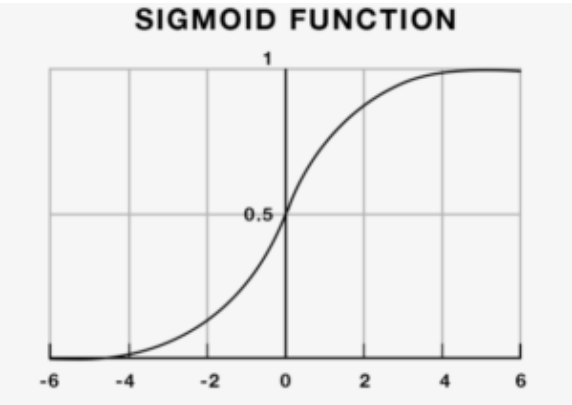
где $\mu_{i,j}$ - среднее значение, а $\sigma_{i,j}$ - стандартное отклонение, которое мы должны оценить по данным. Это означает, что мы получаем одно среднее значение для каждого признака i в паре с классом c_j.

3. Логистическая регрессия

Целью линейной регрессии является оценка значений для коэффициентов модели c, w1, w2, w3 ... .wn и подгонки обучающих данных с минимальной квадратичной ошибкой и прогнозирования вывода y.

Модель логистической регрессии вычисляет взвешенную сумму входных переменных, аналогичную линейной регрессии, но она вычисляет результат с помощью специальной нелинейной функции, логистической или сигмоидной функции для получения значения y.

Сигмоидальная/логистическая функция задается уравнением: y = 1 / 1+ e^{-x}



Таким образом, если результат больше 0,5, мы можем классифицировать результат как 1 (или положительный), а если он меньше 0,5, мы можем классифицировать его как 0 (или отрицательный).

4. Дерево решений

Целью построения такой структуры, как дерево решений, является классификация больших массивов данных.

Для каждого атрибута в наборе данных алгоритм дерева решений формирует узел, в котором наиболее важный атрибут помещается в корневой узел. Для оценки продвигаемся вниз по дереву (начиная с корня), следуя за соответствующим узлом, который соответствует нашему условию. Этот процесс продолжается до тех пор, пока не будет достигнут конечный узел, содержащий прогноз или результат дерева решений.

Для построения дерева и оценки используются следующие понятия:

Энтропия Шеннона определяется для системы с возможными состояниями следующим образом:

$$S = -\sum_{i=1}^{N} p_i \log_2 p_i,$$

где *pi* – вероятности нахождения системы в *i*-ом состоянии. Чем выше энтропия, тем менее упорядочена система и наоборот.

Неопределенность Джини (Gini impurity): $$G = 1 - \sum_k (p_k)^2$$

Максимизацию этого критерия можно интерпретировать как максимизацию числа пар объектов одного класса, оказавшихся в одном поддереве

5. Random Forest

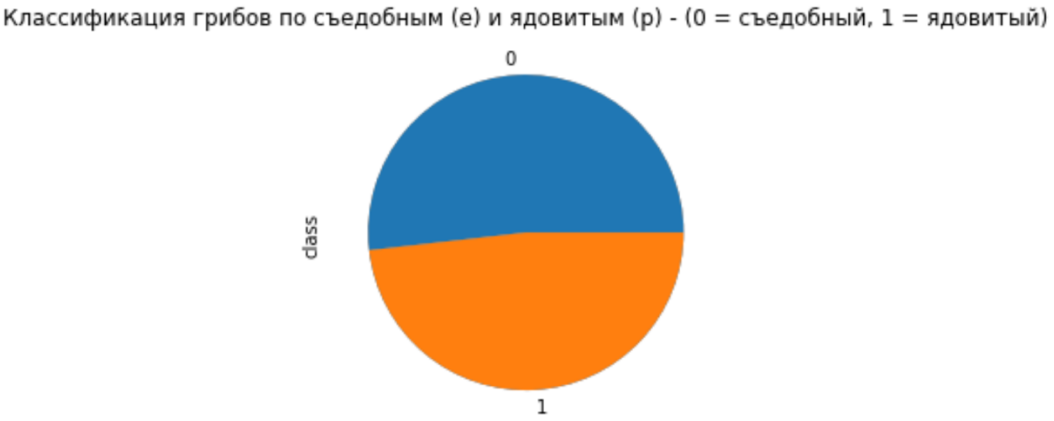
Random forest — это метод, использующий ансамбль деревьев решений, созданных на случайно разделенном датасете. Набор таких деревьев-классификаторов образует лес. Каждое отдельное дерево решений генерируется с использованием метрик отбора показателей, таких как критерий прироста информации, отношение прироста и индекс Джини для каждого признака.

Случайный лес считается высокоточным и надежным методом, поскольку в процессе прогнозирования участвует множество деревьев решений.

Используемые метрики

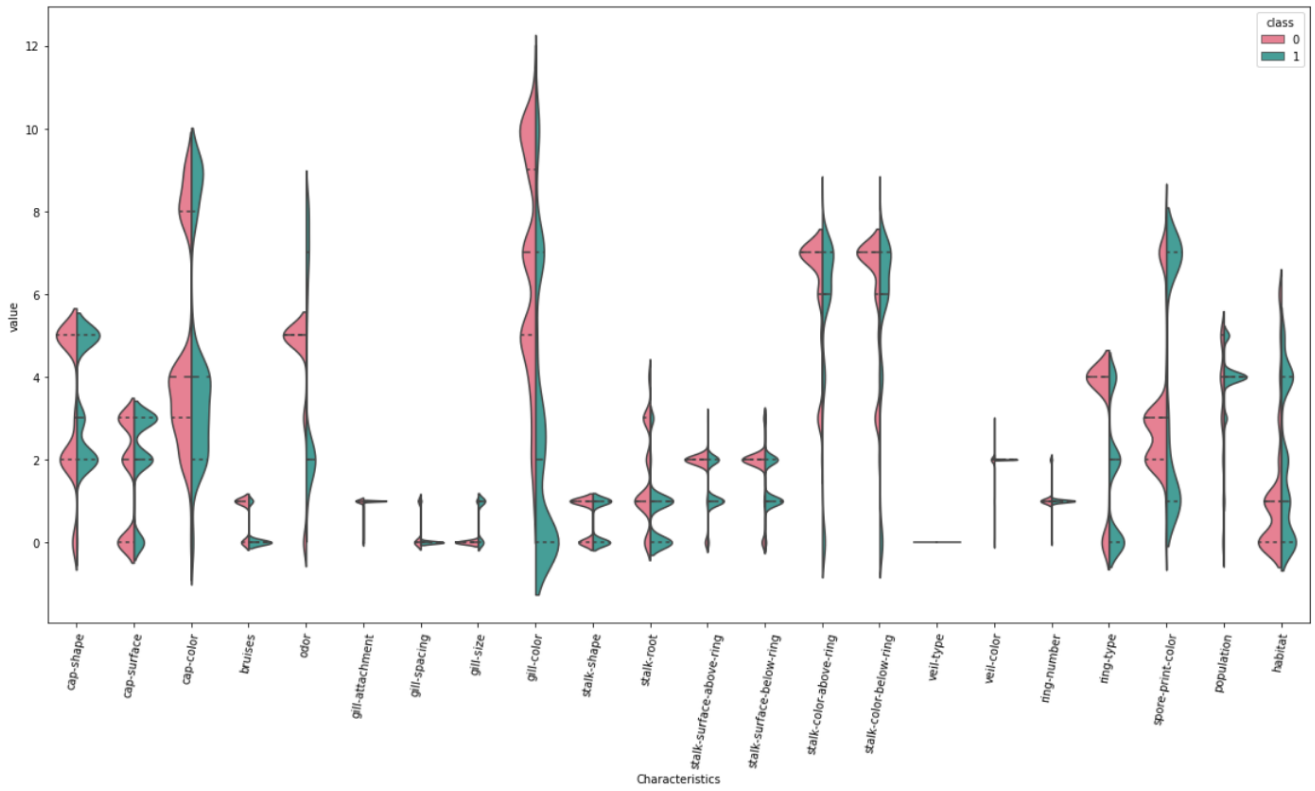
Для оценки качества классификации использованы метрика accuracy, так как в обеих задачах отсутсвует дисбаланс. Валидирование проводилось посредством кросс-валидации.

Мною был выбран [mushroom датасет]. Данные были проверены на баланс по классу:



Из построенной зависимости видно, что класс не нуждается в балансировке.

Распределение по признакам:



Результаты

Все алгоритмы оказались достаточно медленнее в собственной реализации по сравнению с алгоритмами библиотеки scikit-learn, поскольку в моих реализациях нет оптимизации.

In [15]:
```
from sklearn.neighbors import KNeighborsClassifier
from sklearn.linear_model import LogisticRegression
from sklearn.naive_bayes import GaussianNB
from sklearn.ensemble import RandomForestClassifier
from sklearn.tree import DecisionTreeClassifier
import ML as algo

from sklearn.model_selection import KFold
from sklearn.metrics import accuracy_score
from sklearn.metrics import precision_score
from sklearn.metrics import recall_score
import warnings
warnings.filterwarnings('ignore')
```

In [16]:
```
def cv(model, X, y, k_folds=5):
    kf = KFold(n_splits=k_folds, random_state=16, shuffle=True)
    scores = np.zeros(k_folds)
    precisions = np.zeros(k_folds)
    recalls = np.zeros(k_folds)
    for i, (train_index, val_index) in enumerate(kf.split(X, y)):
        X_train, y_train = X.loc[train_index].to_numpy(), y.loc[train_index
        X_val, y_val = X.loc[val_index].to_numpy(), y.loc[val_index].to_num
        model.fit(X_train, y_train)
        y_pred = model.predict(X_val)
        scores[i] = accuracy_score(y_val, y_pred)
        precisions[i] = precision_score(y_val, y_pred)
        recalls[i] = recall_score(y_val, y_pred)
    return (scores, precisions, recalls)
```

In [17]:
```
%%time
model = LogisticRegression()
values = cv(model, X, y)
print("Accuracy: ", values[0].mean())
print("Precision: ", values[1].mean())
print("Recall: ", values[2].mean())
```

```
Accuracy:  0.9497783251231526
Precision:  0.9546315700623771
Recall:  0.9404389431928346
CPU times: user 1.09 s, sys: 94 ms, total: 1.18 s
Wall time: 628 ms
```

In [18]:
```
%%time
model = algo.LR()
values = cv(model, X, y)
print("Accuracy: ", values[0].mean())
print("Precision: ", values[1].mean())
print("Recall: ", values[2].mean())
```

```
Accuracy:  0.9164206138688897
Precision:  0.9455713392470543
Recall:  0.8772857046305106
CPU times: user 8.35 s, sys: 519 ms, total: 8.87 s
Wall time: 4.56 s
```

In [19]:
```
%%time
model = KNeighborsClassifier(n_neighbors=5)
values = cv(model, X, y)
print("Accuracy: ", values[0].mean())
print("Precision: ", values[1].mean())
print("Recall: ", values[2].mean())
```

```
Accuracy:  0.998645850701023
Precision:  0.9979719236072025
Recall:  0.9992546583850931
CPU times: user 2.26 s, sys: 300 ms, total: 2.56 s
Wall time: 1.75 s
```

In [20]:
```
%%time
model = algo.KNN(nn=5)
values = cv(model, X, y)
print("Accuracy: ", values[0].mean())
print("Precision: ", values[1].mean())
print("Recall: ", values[2].mean())
```

```
Accuracy:  0.9986459264873059
Precision:  0.9977097671068123
Recall:  0.9995031055900622
CPU times: user 5.52 s, sys: 257 ms, total: 5.77 s
Wall time: 5.89 s
```

In [21]:
```
%%time
model = GaussianNB()
values = cv(model, X, y)
print("Accuracy: ", values[0].mean())
print("Precision: ", values[1].mean())
print("Recall: ", values[2].mean())
```

```
Accuracy:  0.9217145888594166
Precision:  0.9152733655421035
Recall:  0.9228448064380543
CPU times: user 80.4 ms, sys: 7.19 ms, total: 87.6 ms
Wall time: 88.6 ms
```

In [22]:
```
%%time
model = algo.GaussianNaiveBayesClassifier()
values = cv(model, X, y)
print("Accuracy: ", values[0].mean())
print("Precision: ", values[1].mean())
print("Recall: ", values[2].mean())
```

```
Accuracy:  0.9217145888594166
Precision:  0.9152733655421035
Recall:  0.9228448064380543
CPU times: user 692 ms, sys: 17.4 ms, total: 710 ms
Wall time: 736 ms
```

In [23]:
```
%%time
model = DecisionTreeClassifier(max_depth=3)
values = cv(model, X, y)
print("Accuracy: ", values[0].mean())
print("Precision: ", values[1].mean())
print("Recall: ", values[2].mean())
```

```
Accuracy:  0.9586414550966275
Precision:  0.9433898271980151
Recall:  0.9722973135250184
CPU times: user 84.1 ms, sys: 5.58 ms, total: 89.7 ms
Wall time: 93.7 ms
```

In [24]:
```
%%time
model = algo.DTC()
values = cv(model, X, y)
print("Accuracy: ", values[0].mean())
print("Precision: ", values[1].mean())
print("Recall: ", values[2].mean())
```

```
Accuracy:  0.7902507010231148
Precision:  0.8343044863880655
Recall:  0.704640176834282
CPU times: user 4.63 s, sys: 47.4 ms, total: 4.68 s
Wall time: 4.72 s
```

In [25]:
```
%%time
model = algo.RFC()
values = cv(model, X, y)
print("Accuracy: ", values[0].mean())
print("Precision: ", values[1].mean())
print("Recall: ", values[2].mean())
```

```
Accuracy:  0.8878710875331566
Precision:  0.8763113498415525
Recall:  0.8741603085014997
CPU times: user 350 ms, sys: 5.82 ms, total: 355 ms
Wall time: 357 ms
```

In [26]:
```
%%time
model = RandomForestClassifier()
values = cv(model, X, y)
print("Accuracy: ", values[0].mean())
print("Precision: ", values[1].mean())
print("Recall: ", values[2].mean())
```

```
Accuracy:  1.0
Precision:  1.0
Recall:  1.0
CPU times: user 1.69 s, sys: 17.8 ms, total: 1.71 s
Wall time: 1.73 s
```

При выбранных параметрах алгоритмов видно, что модель не переобучалась, поскольку разница в точности классификации на обучающей и тестовой выборках в основном достаточно мала.

Что касается полученной точности – по результатам Байесовского классификатора можно сделать предположение, что дата сет не совсем подходит для этого алгоритма. Поскольку реализация была улучшена и очень близка к библиотечной.

Выводы

При “ручной” реализации алгоритмов машинного обучения нужно использовать максимально эффективные алгоритмы, чтобы достичь таких же высоких результатов, какие предоставляют нам библиотечные реализации.