Московский авиационный институт

(национальный исследовательский университет)

Институт №8 «Информационные технологии и прикладная математика»

Лабораторная работа №2 по искусственному интеллекту

6 семестр

Студент: Катермин Всеволод Сергеевич

Группа: М8О-308Б-18

Руководитель: Ахмед Самир Халид

Дата: 20.06.2021

Москва, 2021

Оглавление

Постановка задачи:

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. Задачи со звездочкой бьются по вариантам:

N по списку % 2 + 1.

1) **Логистическая регрессия**
2) ***SVM - Первый вариант**
3) **Дерево решений**
4) ***Random Forest - второй вариант**

Логистическая регрессия

Алгоритм

В отличие от обычной регрессии, в методе логистической регрессии не производится предсказание значения числовой переменной исходя из выборки исходных значений. Вместо этого, значением функции является вероятность того, что данное

исходное значение принадлежит к определенному классу. Для простоты, давайте предположим, что у нас есть только два класса и вероятность, которую мы будем определять, P_+ вероятности того, что некоторое значение принадлежит классу "+". И конечно $P_- = 1 - P_+$. Таким образом, результат логистической регрессии всегда находится в интервале [0, 1].

Основная идея логистической регрессии заключается в том, что пространство исходных значений может быть разделено линейной границей (т.е. прямой) на две соответствующих классам области. Итак, что же имеется ввиду под линейной границей? В случае двух измерений — это просто прямая линия без изгибов. В случае

трех — плоскость, и так далее. Эта граница задается в зависимости от имеющихся исходных данных и обучающего алгоритма. Чтобы все работало, точки исходных данных должны разделяться линейной границей на две вышеупомянутых области. Если точки исходных данных удовлетворяют этому требованию, то их можно назвать линейно разделяемыми.

Реализация

Я реализовал логистическую регрессию в виде класса с 2 публичными методами. fit - для обучения, predict - для предсказания

```
class LogReg:
    #задаем количество итераций при инициализации класса 100000
    def __init__(self,num_iter = 100000):
        self.num_iter=num_iter
        self.beta=1

     #метод обучающий модель
    def fit(self,x,y):
        #задаем матрицу весов в виде единичной матрицы
        self.beta = np.ones(x.shape[1])
        for i in range(self.num_iter):
            h = self._sigmoid(x, self.beta)#считаем сигмойду
            gradient = self._gradient_spusk(x, h, y)#спускаемся по градиенту
            self.beta =self._weight_update(self.beta, 0.1, gradient)#обновляем веса

    #приватный метод, считающий сигмойду
    def _sigmoid(self,X, weight):
        z = np.dot(X, weight)
        return 1 / (1 + np.exp(-z))

    #приватная функция для градиентного шага
    def _gradient_spusk(self,X, H, Y):
        return np.dot(X.T, (H - Y)) / Y.shape[0]

    #приватная функция для обноления весов
    def _weight_update(self,weight, learning_rate, gradient):
        return weight - learning_rate * gradient


    def predict(self,test):
        final_result=[]

        #приминяем сигмойду к тестовым данным
        result = self._sigmoid(test, self.beta)

        #выбираем  метки для теста
        for i in result:
            final_result.append(self._onepred(i))


        return final_result



    #приватная функция для одного предсказания
    def _onepred(self,x):
        if x < 0.5:
            return 0
        else:
            return 1
```

Обучение и метрики

```
#обучаю свою модель
my_lg=LogReg()
my_lg.fit(X_train,y_train)
#делаем предсказания на трейне и на тесте и смотрим метрики
print('Метрики на обучающей выборки ')
metrics(my_lg.predict(X_train),y_train)
print('Метрики на тестовой выборки ')
metrics(my_lg.predict(X_test),y_test)
```

```
<ipython-input-5-f7a648a8a967>:20: RuntimeWarning: overflow encountered in exp
  return 1 / (1 + np.exp(-z))
```

```
Метрики на обучающей выборки
Accuracy:  0.7624398073836276
Pprecision:  0.7624398073836276
Recall:  0.7624398073836276
F1:  0.7624398073836277
Метрики на тестовой выборки
Accuracy:  0.8097014925373134
Pprecision:  0.8097014925373134
Recall:  0.8097014925373134
F1:  0.8097014925373133
Wall time: 1min 27s
```

In [7]:
```
#обучаю модель из sklearn
sk_lg=LogisticRegression( max_iter=1000000)
sk_lg.fit(X_train,y_train)
#делаем предсказания на трейне и на тесте и смотрим метрики
print('Метрики на обучающей выборки ')
metrics(sk_lg.predict(X_train),y_train)
print('Метрики на тестовой выборки ')
metrics(sk_lg.predict(X_test),y_test)
```

```
Метрики на обучающей выборки
Accuracy:  0.7945425361155698
Pprecision:  0.7945425361155698
Recall:  0.7945425361155698
F1:  0.7945425361155698
Метрики на тестовой выборки
Accuracy:  0.8246268656716418
Pprecision:  0.8246268656716418
Recall:  0.8246268656716418
F1:  0.8246268656716418
```

Выводы о моделях по метрикам

- Модель из sklearn показала чуть лучшие метрики на трейне и тесте чем моя модель
- Обе модели классифицируют объекты с долей верный ответов примерно 80%
- В обоих моделях наблюдается странный эффект связанный с тем, метрики на тесте почему -то выше чем на трейне

Метод опорных векторов

Алгоритм

Алгоритм Главная цель SVM как классификатора — найти уравнение разделяющей гиперплоскости в пространстве, которая бы разделила два класса неким оптимальным образом. После настройки весов алгоритма (обучения), все объекты, попадающие по одну сторону от построенной гиперплоскости, будут предсказываться как первый класс, а объекты, попадающие по другую сторону — второй класс.

Реализация

Я реализовал SVM в виде класса MYSVM с двумя публичными методами fit- для обучения, predict - для предсказания

```
class MYSVM(object):
    # при инициализации класса задается сразу _etha -шаг градиентного спуска,_alpha - коэффициент быстроты
    #пропорционального уменьшения весов, _epochs - количество эпох обучения
    def __init__(self, etha=0.1, alpha=0.2, epochs=990):
        self._epochs = epochs
        self._etha = etha
        self._alpha = alpha
        self._w = None


    #метод для обучения модели
    def fit(self, X_train, Y_train):


        for i in range(len(Y_train)):
            if Y_train.iloc[i] == 0:
                Y_train.iloc[i] = -1

        #добавляем в конец каждого вектора число 1
        X_train = self._add_bias_feature(X_train)
        self._w = np.random.normal(loc=0, scale=0.05, size=X_train.shape[1])#задаем первые веса



        for epoch in range(self._epochs):

            for i,x in enumerate(X_train):
                margin = Y_train.iloc[i]*np.dot(self._w,X_train[i])
                if margin >= 1: # классифицируем верно
                    self._w = self._w - self._etha*self._alpha*self._w/self._epochs

                else: # классифицируем неверно или попадаем на полосу разделения при 0<m<1
                    self._w = self._w +\
                    self._etha*(Y_train.iloc[i]*X_train[i] - self._alpha*self._w/self._epochs)


        for i in range(len(Y_train)):
            if Y_train.iloc[i]==-1:
                Y_train.iloc[i]=0

    #Приватный метод , добовляющей в конец каждого вектора чисор 1
    def _add_bias_feature(self,a):

        a_extended = np.zeros((a.shape[0],a.shape[1]+1))
        a_extended[:,:-1] = a
        a_extended[:,-1] = int(1)
        return a_extended


     #метод для предсказания
    def predict(self, X):
        y_pred = []
        #X_extended = self._add_bias_feature(X)
        for i in range(len(X)):
            y_pred.append(np.sign(1+np.dot(self._w[1:],X.iloc[i])))
        for i in range(len(y_pred)):
            if y_pred[i]==-1:
                y_pred[i]=0

        return y_pred
```

Обучение и метрики

In [9]:
```
my_svm=MYSVM()
my_svm.fit(X_train,y_train)
print('метрики на обучении')
metrics(my_svm.predict(X_train),y_train)
print('метрики на тесте')
metrics(my_svm.predict(X_test),y_test)
```

```
метрики на обучении
Accuracy:  0.6789727126805778
Pprecision:  0.6789727126805778
Recall:  0.6789727126805778
F1:  0.6789727126805778
метрики на тесте
Accuracy:  0.6716417910447762
Pprecision:  0.6716417910447762
Recall:  0.6716417910447762
F1:  0.6716417910447762
```

In [10]:
```
sk_svm = svm.SVC()
sk_svm.fit(X_train, y_train)
print('метрики на обучении')
metrics(sk_svm.predict(X_train),y_train)
print('метрики на тесте')
metrics(sk_svm.predict(X_test),y_test)
```

```
метрики на обучении
Accuracy:  0.6741573033707865
Pprecision:  0.6741573033707865
Recall:  0.6741573033707865
F1:  0.6741573033707865
метрики на тесте
Accuracy:  0.6604477611940298
Pprecision:  0.6604477611940298
Recall:  0.6604477611940298
F1:  0.6604477611940298
```

Выводы

- Разница метрик моей модели и модели из sklearn минимальна.
- Переобучения у обоих моделей не наблюдается
- Данный алгоритм показывает самые низкие метрики из всех, это вызывает сомнения относительно его использования в отношении классификации в датасете Титаник

Дерево принятия решений

Алгоритм

Дерево решений представляет собой иерархическую древовидную структуру, состоящую из правила вида «Если …, то ...». За счет обучающего множества правила генерируются автоматически в процессе обучения. Правила генерируются за счет обобщения множества отдельных наблюдений (обучающих примеров), описывающих предметную область. Поэтому их называют индуктивными 10 правилами, а сам процесс обучения — индукцией деревьев решений. В обучающем множестве для примеров должно быть задано целевое значение, так как деревья решений — модели, создаваемые на основе обучения с учителем

Реализация

Я реализовать дерево решений в виде класса MyDT с двуми публичными методами , fit - для обучения, predict - для предсказания остальные методы приватные и используются в публичных

```
class MyDT():

    # объявляем характеристики класса
    def __init__(self, max_depth=3, min_size=10):

        self.max_depth = max_depth
        self.min_size = min_size
        self.value = 0
        self.feature_idx = -1
        self.feature_threshold = 0
        self.left = None
        self.right = None
```

Реализация метода fit

```
# процедура обучения - сюда передается обучающая выборка
def fit(self, X, y):



    for i in range(len(y)):
        if y.iloc[i] == 0:
            y.iloc[i] = -1


    # начальное значение - среднее значение y
    self.value = y.mean()
    # начальная ошибка - mse между значением в листе (пока нет
    # разбиения, это среднее по всем объектам) и объектами
    base_error = ((y - self.value) ** 2).sum()
    error = base_error
    flag = 0

    # пришли в максимальную глубину
    if self.max_depth <= 1:
        return

    dim_shape = X.shape[1]

    left_value, right_value = 0, 0

    for feat in range(dim_shape):

        prev_error1, prev_error2 = base_error, 0
        if feat==0:
            idxs = np.argsort(X[:, feat])

        # переменные для быстрого переброса суммы
        mean1, mean2 = y.mean(), 0
        sm1, sm2 = y.sum(), 0

        N = X.shape[0]
        N1, N2 = N, 0
        thres = 1

       while thres < N - 1:
            N1 -= 1
            N2 += 1

            idx = idxs[thres]
            x = X[int(idx), feat]

            # вычисляем дельты - по ним в основном будет делаться переброс
            delta1 = (sm1 - y.iloc[idx]) * 1.0 / N1 - mean1
            delta2 = (sm2 + y.iloc[idx]) * 1.0 / N2 - mean2

            # увеличиваем суммы
            sm1 -= y.iloc[idx]
            sm2 += y.iloc[idx]

            # пересчитываем ошибки за O(1)
            prev_error1 += (delta1**2) * N1
            prev_error1 -= (y.iloc[idx] - mean1)**2
            prev_error1 -= 2 * delta1 * (sm1 - mean1 * N1)
            mean1 = sm1/N1

            prev_error2 += (delta2**2) * N2
            prev_error2 += (y.iloc[idx] - mean2)**2
            prev_error2 -= 2 * delta2 * (sm2 - mean2 * N2)
            mean2 = sm2/N2

            # пропускаем близкие друг к другу значения
            if thres < N - 1 and np.abs(x - X[idxs[thres + 1], feat]) < 1e-5:
                thres += 1
                continue

            # 2 условия, чтобы осуществить сплит - уменьшение ошибки
            # и минимальное кол-о эл-в в каждом листе
            if (prev_error1 + prev_error2 < error):
                if (min(N1,N2) > self.min_size):

                    # переопределяем самый лучший признак и границу по нему
                    self.feature_idx, self.feature_threshold = feat, x
                    # переопределяем значения в листах
                    left_value, right_value = mean1, mean2

                    # флаг - значит сделали хороший сплит
                    flag = 1
                    error = prev_error1 + prev_error2

            thres += 1

  # ничего не разделили, выходим
  if self.feature_idx == -1:
      return

  self.left = MyDT(self.max_depth - 1)
  # print ("Левое поддерево с глубиной %d"%(self.max_depth - 1))
  self.left.value = left_value
  self.right = MyDT(self.max_depth - 1)
  # print ("Правое поддерево с глубиной %d"%(self.max_depth - 1))
  self.right.value = right_value

  idxs_l = (X[:, self.feature_idx] > self.feature_threshold)
  idxs_r = (X[:, self.feature_idx] <= self.feature_threshold)

  self.left.fit(X[idxs_l, :], y[idxs_l])
  self.right.fit(X[idxs_r, :], y[idxs_r])

  for i in range(len(y)):
      if y.iloc[i]==-1:
          y.iloc[i]=0
```

Реализация метода predict и приватный методов _predict , _prediction , используемых в predict

```
def __predict(self, x):
    if self.feature_idx == -1:
        return self.value

    if x[self.feature_idx] > self.feature_threshold:
        return self.left.__predict(x)
    else:
        return self.right.__predict(x)

#метод для финального расставления меток
def _prediction(self,x):
    if x < 0:

        return 0
    else:
        return 1

#Метод для предсказания
def predict(self, X):
    y = np.zeros(X.shape[0])

    for i in range(X.shape[0]):
        y[i] = self.__predict(X[i])

    for i in range(len(y)):
        y[i]=self._prediction(y[i])
    return y
```

Обучение и метрики

In [12]:
```
my_dt=MyDT()
my_dt.fit(X_train.values,y_train)
print('метрики на обучении')
metrics(my_dt.predict(X_train.values),y_train)
print('метрики на тесте')
metrics(my_dt.predict(X_test.values),y_test)
```

```
метрики на обучении
Accuracy:  0.7929373996789727
Pprecision:  0.7929373996789727
Recall:  0.7929373996789727
F1:  0.7929373996789727
метрики на тесте
Accuracy:  0.7723880597014925
Pprecision:  0.7723880597014925
Recall:  0.7723880597014925
F1:  0.7723880597014926
```

In [13]:
```
dt = DecisionTreeClassifier(max_depth=3, min_samples_leaf=10)
dt.fit(X_train,y_train)
print('метрики на обучении')
metrics(dt.predict(X_train.values),y_train)
print('метрики на тесте')
metrics(dt.predict(X_test.values),y_test)
```

```
метрики на обучении
Accuracy:  0.826645264847512
Pprecision:  0.826645264847512
Recall:  0.826645264847512
F1:  0.826645264847512
метрики на тесте
Accuracy:  0.8059701492537313
Pprecision:  0.8059701492537313
Recall:  0.8059701492537313
F1:  0.8059701492537313
```

Выводы

- Разница метрик между моей моей моделью и моделью из sklearn минимальна.
- Переобучения у обоих моделей не наблюдается
- Модели неплохо классифицируют датасет с долей верных ответов примерно 80%