

**Московский авиационный институт
(национальный исследовательский университет)**

**Институт №8 «Информационные технологии и прикладная
математика»**

**Кафедра 806 «Вычислительная математика и
программирование»**

Лабораторная работа №1 по курсу «Искусственный интеллект»

Студент: М. А. Волков
Преподаватели: Д. В. Сошников
С. Х. Ахмед
Группа: М8О-307Б-19
Дата:
Оценка:
Подпись:

Москва, 2022

Лабораторная работа №1

Задача: Вы собрали данные и их проанализировали, визуализировали и представили отчет своим партнерам и спонсорам. Они согласились, что ваша задача имеет перспективу и продемонстрировали заинтересованность в вашем проекте. Самое время реализовать прототип! Вы считаете, что нейронные сети переоценены (просто боитесь признаться, что у вас не хватает ресурсов и данных), и считаете что за классическим машинным обучением будущее и потому собираетесь использовать классические модели. Вашим первым предположением является предположение, что данные и все в этом мире имеет линейную зависимость, ведь не зря же в конце каждой нейронной сети есть линейный слой классификации. В качестве первых моделей вы выбрали линейную/логистическую регрессию и SVM. Так как вы очень осторожны и боитесь ошибиться, вы хотите реализовать случай, когда все таки мы не делаем никаких предположений о данных и взяли за основу идею "близкие объекты дают близкий ответ" и идею, что теорема Байеса имеет ранг королевской теоремы. Так как вы не доверяете другим людям, вы хотите реализовать алгоритмы сами с нуля без использования scikit-learn (почти). Вы хотите узнать насколько хорошо ваши модели работают на выбранных вам данных и хотите замерить метрики качества. Ведь вам нужно еще отчитаться спонсорам!

Формально говоря вам предстоит сделать следующее:

1. Реализовать следующие алгоритмы машинного обучения: Linear/Logistic Regression, SVM, KNN, Naïve Bayes в отдельных классах;
2. Данные классы должны наследоваться от BaseEstimator и ClassifierMixin, иметь методы fit и predict;
3. Вы должны организовать весь процесс предобработки, обучения и тестирования с помощью Pipeline;
4. Вы должны настроить гиперпараметры моделей с помощью кросс валидации, вывести и сохранить эти гиперпараметры в файл, вместе с обученными моделями;
5. Прodelать аналогично с коробочными решениями;
6. Для каждой модели получить оценки метрик: Confusion Matrix, Accuracy, Recall, Precision, ROC_AUC curve;
7. Проанализировать полученные результаты и сделать выводы о применимости моделей;
8. Загрузить полученные гиперпараметры модели и обученные модели в формате pickle на гит вместе с Jupyter Notebook ваших экспериментов.

1 Описание

Для корректной работы моделей признаки нужно нормализовать. Сначала разделим данные на тренировочную и тестовую выборку, после чего нормализуем их. Делаем это именно в таком порядке, чтобы у нас было гарантировано, что и в тренировочной, и в тестовой выборке все значения будут от 0 до 1. Данные разделим в пропорциях (80 к 20), используя `train_test_split` из `scikit-learn` [2]. `scikit-learn` позволяет сделать это с помощью `normalize` [3], я использую метрику «max».

В задании требуется сделать все модели совместимыми с `scikit-learn` [4], поэтому получение оценок модели [1] можно сделать методами из этой же библиотеки:

```
1 def scores(model, X, y_true):
2     y_pred = model.predict(X)
3     print("Accuracy:", accuracy_score(y_true, y_pred))
4     print("Recall:", recall_score(y_true, y_pred))
5     print("Precision:", precision_score(y_true, y_pred))
6     figure = plt.figure(figsize = (20, 5))
7     matr = confusion_matrix(y_true, y_pred)
8     ax = plt.subplot(1, 2, 1)
9     ConfusionMatrixDisplay(matr).plot(ax = ax)
10    ax = plt.subplot(1, 2, 2)
11    RocCurveDisplay.from_predictions(y_true = y_true, y_pred = y_pred, name = "ROC-
12    curve", ax = ax)
13    plt.show()
```

При реализации моделей я использовал шаблон [5] из `scikit-learn`, в котором учтены все тонкости реализации: наследование от нужных классов (`ClassifierMixin`, `BaseEstimator`) и необходимые методы (`fit`, `predict`).

1 Метод k-ближайших соседей

Среди всех объектов обучающей выборки ищем k ближайших, среди них классифицируем объект тем классом, которого больше всего среди соседей:

```
1 class kNN(BaseEstimator, ClassifierMixin):
2
3     def __init__(self, k=1):
4         self.k = k
5
6     def fit(self, X, y):
7         X, y = check_X_y(X, y)
8         # Store the classes seen during fit
9         self.classes_ = unique_labels(y)
10
11         self.X_ = X
12         self.y_ = y
13         self.is_fitted_ = True
14         # 'fit' should always return 'self'
15         return self
16
17     def predict(self, X):
18         # Check if fit had been called
19         check_is_fitted(self, ['X_', 'y_'])
20
21         # Input validation
22         X = check_array(X)
23
24         y = np.ndarray((X.shape[0],))
25         for (i, elem) in enumerate(X):
26             distances = euclidean_distances([elem], self.X_)[0]
27             indexes = np.argsort(distances, kind='heapsort')[:self.k]
28
29             labels, cnts = np.unique(self.y_[indexes], return_counts=True)
30             y[i] = labels[cnts.argmax()]
31         return y
```

Использую Евклидову метрику из scikit-learn для вычисления расстояний и метод argsort из numpy [9] для индексной сортировки. Также в качестве алгоритма сортировки использую heapsort, т.к. по умолчанию используется quicksort, который, исходя из документации, имеет оценку сложности $O(n^2)$, в то время как heapsort сортирует массив за $O(n \cdot \log(n))$, при этом не используя дополнительную память (исходя из документации numpy)

Notes

The various sorting algorithms are characterized by their average speed, worst case performance, work space size, and whether they are stable. A stable sort keeps items with the same key in the same relative order. The four algorithms implemented in NumPy have the following properties:

kind	speed	worst case	work space	stable
'quicksort'	1	$O(n^2)$	0	no
'heapsort'	3	$O(n \log(n))$	0	no
'mergesort'	2	$O(n \log(n))$	$\sim n/2$	yes
'timsort'	2	$O(n \log(n))$	$\sim n/2$	yes

Из отсортированных индексов получаю k ближайших соседей.

2 Логистическая регрессия

Логистическая регрессия по сути является однослойной нейросетью. Использую результаты из ЛР по реализации собственного фреймворка, чтобы реализовать модель. Описываю класс сети, которую можно строить из разных слоёв:

```
1 import matplotlib.pyplot as plt
2 class Net:
3     def __init__(self, loss_function=CrossEntropyLoss()):
4         self.layers = []
5         self.loss_func = loss_function
6
7     # ----Net's standart methods----
8     def add(self,l):
9         self.layers.append(l)
10
11     def forward(self,x):
12         for l in self.layers:
13             x = l.forward(x)
14         return x
15
16     def backward(self,z):
17         for l in self.layers[::-1]:
18             z = l.backward(z)
19         return z
20
21     def update(self,lr):
22         for l in self.layers:
23             if 'update' in l.__dir__():
24                 l.update(lr)
25 # ----end Net's standart methods----
26
27 # ----loss functions----
28     def forward_loss(self, x, y):
29         p = self.forward(x)
30         return self.loss_func.forward(p, y)
31
32     def backward_loss(self, l):
33         dp = self.loss_func.backward(l)
34         return self.backward(dp)
35
36     def _update_dry(self, x, y, step):
37         self.update(step)
38         loss = self.forward_loss(x, y)
39         self.update(-step)
40         return loss
41 # ----end loss functions----
42
43 # ----Net train----
```

```

44     def _is_less_update_dry(self, x_dry, y_dry, step, l_dry, r_dry):
45         lhs = (2.0 * l_dry + r_dry) / 3.0
46         rhs = (l_dry + 2.0 * r_dry) / 3.0
47
48         loss_lhs = self._update_dry(x_dry, y_dry, step * lhs)
49         loss_rhs = self._update_dry(x_dry, y_dry, step * rhs)
50
51         return loss_lhs < loss_rhs
52
53     def train_epoch(self, epoch_train_x, train_labels, batch_size=100, step = 1e-7):
54         for index in range(0, len(epoch_train_x), batch_size):
55             xb = epoch_train_x[index:index + batch_size]
56             yb = train_labels[index:index + batch_size]
57
58             loss = self.forward_loss(xb, yb)
59             self.backward_loss(loss)
60
61             l = 1.0
62             r = 5e2
63             while r - l < 0.01:
64                 lhs = (2.0 * l + r) / 3.0
65                 rhs = (l + 2.0 * r) / 3.0
66                 if self._is_less_update_dry(xb,yb,step, lhs, rhs):
67                     r = rhs
68                 else:
69                     l = lhs
70             self.update(r * step)

```

Линейный слой сети с возможностью обновления весов (изначально матрица весов заполняется случайными числами, а вектор смещения нулями) так же вынесен в отдельный класс:

```
1 class Linear(Layer):
2     def __init__(self, nin, nout):
3         sigma = 1.0 / np.sqrt(2.0 * nin)
4         self.W = np.random.normal(0, sigma, (nout, nin))
5         self.b = np.zeros((1, nout))
6         self.dW = np.zeros_like(self.W)
7         self.db = np.zeros_like(self.b)
8
9     def forward(self, x):
10        self.x = x
11        return np.dot(x, self.W.T) + self.b
12
13    def backward(self, dz):
14        dx = np.dot(dz, self.W)
15        dW = np.dot(dz.T, self.x)
16        db = dz.sum(axis=0)
17        self.dW = dW
18        self.db = db
19        return dx
20
21    def update(self, lr):
22        self.W -= lr * self.dW
23        self.b -= lr * self.db
```

В логистической регрессии используется функция активации сигмоида, которая так же описана в отдельном классе:

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

```
1 class Sigmoid(Layer):
2     def forward(self, x):
3         self.y = 1.0 / (1.0 + np.exp(-x))
4         return self.y
5
6     def backward(self, dy):
7         return self.y * (1.0 - self.y) * dy
```


В качестве функции потерь использую binary cross entropy loss, так как стоит задача бинарной классификации:

```
1 class BinaryCrossEntropy(Layer):
2     def forward(self, p, y):
3         y = y.reshape((y.shape[0], 1))
4         self.p = p
5         self.y = y
6         res = y * np.log(p) + (1 - y) * np.log(1 - p)
7         return -np.mean(res)
8
9     def backward(self, loss):
10        res = (self.p - self.y) / (self.p * (1 - self.p))
11        return res / self.p.shape[0]
```

Логистическая регрессия содержит нейросеть, состоящую из линейного слоя, сигмиды и описанной выше функции потерь. Также есть настраиваемый параметр, который характеризует кол-во входных признаков. Алгоритм обучения сети — стохастический градиентный спуск с постоянным шагом:

```
1 class LogisticRegression(ClassifierMixin, BaseEstimator):
2     def __init__(self, epochs=1, batch_size=10, SGD_step=0.001, nin=10):
3         self.epochs = epochs
4         self.batch_size = batch_size
5         self.SGD_step = SGD_step
6         self.nin = nin
7         self.Net = Net(BinaryCrossEntropy())
8         self.Net.add(Linear(nin, 1))
9         self.Net.add(Sigmoid())
10
11    def fit(self, X, y):
12        # Check that X and y have correct shape
13        X, y = check_X_y(X, y)
14        # Store the classes seen during fit
15        self.classes_ = unique_labels(y)
16
17        self.X_ = X
18        self.y_ = y
19        for _ in range(self.epochs):
20            self.Net.train_epoch(X, y, self.batch_size, self.SGD_step)
21        # Return the classifier
22        return self
23
24    def predict(self, X):
25        # Check is fit had been called
26        check_is_fitted(self, ['X_', 'y_'])
27
28        # Input validation
29        X = check_array(X)
30
```

```
31         y = self.Net.forward(X)
32         res = np.where(y < 0.5, 0, 1)
33         return res
34
35     def getW(self):
36         return self.Net.layers[0].W
37
38     def getb(self):
39         return self.Net.layers[0].b
```

3 Метод опорных векторов

Метод похож на предыдущий, но функция ошибки требует сами данные, на которых происходит обучение, поэтому встроить в класс сети не получилось. Однако, из-за схожести архитектуры реализации было принято решение создать класс, который наследуется от Net, написанного в ЛР по персептронам. На основе этого класса отдельно описываю модель опорных векторов с мягким зазором:

```
1 class SoftMarginSVM(Net):
2     def __init__(self, nin, alpha):
3         super().__init__()
4         self.alpha = alpha
5         sigma = 1.0 / np.sqrt(nin)
6         self.W = np.random.normal(0., sigma, (1, nin + 1))
7
8     def forward(self, x):
9         z = np.dot(x, self.W.T)
10        return z
11
12    def add_ones(self, x):
13        ones = np.ones((x.shape[0], 1))
14        return np.hstack((x, ones))
15
16    def predict(self, x):
17        res = self.forward(self.add_ones(x))
18        return np.where(res < 0, 0, 1)
19
20    def train_epoch(self, x, y, batch_size=100, step=1e-7):
21        x = self.add_ones(x)
22        y = np.where(y > 0, 1, -1)
23        for i in range(0, len(x), batch_size):
24            xb = x[i:i + batch_size]
25            yb = y[i:i + batch_size]
26
27            pred = self.forward(xb)
28            grad = self.alpha * self.W
29            for i in range(len(xb)):
30                if (yb[i] * pred[i] < 1):
31                    grad -= yb[i] * xb[i]
32            self.W -= step * grad
```

Класс получился простой, но не очень универсальный. Чтобы отбросить вектор смещения b , я ввёл искусственный признак, который всегда равен единице [14].

Эта модель затем встраивается в классификатор. Параметры почти такие же, как и в случае с логистической регрессией:

```
1 class SVM(ClassifierMixin, BaseEstimator):
2     def __init__(self, epoches=1, batch_size=10, SGD_step=0.001, alpha=0.1, nin=10):
3         self.epoches = epoches
4         self.batch_size = batch_size
5         self.SGD_step = SGD_step
6         self.nin = nin
7         self.alpha = alpha
8         self.Net = SoftMarginSVM(nin, alpha)
9
10    def fit(self, X, y):
11        # Check that X and y have correct shape
12        X, y = check_X_y(X, y)
13        # Store the classes seen during fit
14        self.classes_ = unique_labels(y)
15
16        self.X_ = X
17        self.y_ = y
18        for _ in range(self.epoches):
19            self.Net.train_epoch(X, y, self.batch_size, self.SGD_step)
20        # Return the classifier
21        return self
22
23    def predict(self, X):
24        y = self.Net.predict(X)
25        return y
26
27    def getW(self):
28        return self.Net.W
```

4 Наивный байесовский классификатор

Идея модели в наивном предположении о независимости параметров. Так же часто используется модель с нормальным распределением признаков.

```
1 class NaiveBayes(ClassifierMixin, BaseEstimator):
2     def __init__(self):
3         None
4
5     def fit(self, X, y):
6         # Check that X and y have correct shape
7         X, y = check_X_y(X, y)
8
9         self.X_ = X
10        self.y_ = y
11
12        labels, cnts = np.unique(self.y_, return_counts = True)
13        self.labels = labels
14        self.p_of_y = np.array([elem / self.y_.shape[0] for elem in cnts])
15        self.means = np.array([self.X_[self.y_ == elem].mean(axis = 0) for elem in
16                               labels])
17        self.stds = np.array([self.X_[self.y_ == elem].std(axis = 0) for elem in labels
18                               ])
19        # Return the classifier
20        return self
21
22    def gaussian(self, mu, sigma, x0):
23        return np.exp(-(x0 - mu) ** 2 / (2 * sigma)) / np.sqrt(2.0 * pi * sigma)
24
25    def predict(self, X):
26        # Check is fit had been called
27        check_is_fitted(self, ['X_', 'y_'])
28
29        # Input validation
30        X = check_array(X)
31
32        res = np.zeros(X.shape[0])
33        for (i, elem) in enumerate(X):
34            p = np.array(self.p_of_y)
35            for (j, label) in enumerate(self.labels):
36                p_x_cond_y = np.array([self.gaussian(self.means[j][k], self.stds[j][k],
37                                                       elem[k]) for k in range(X.shape[1])])
38                p[j] *= np.prod(p_x_cond_y)
39            res[i] = np.argmax(p)
40        return res
```

5 Подбор гиперпараметров

Для подбора гиперпараметров используются кросс-валидации GridSearchCV [7] и RandomizedSearchCV [8]. Приведу пример использования с SVM:

```
1 | gscv = GridSearchCV(Pipeline([("SVM", SVM(nin=train_X.shape[1]))]),
2 |                     {"SVM__epoches" : [1, 2, 4],
3 |                      "SVM__batch_size" : [5, 10, 20],
4 |                      "SVM__SGD_step" : [0.01, 0.05, 0.1],
5 |                      "SVM__alpha" : [1.0, 0.1, 0.01, 0.0]})
6 | gscv.fit(train_X, train_y)
7 | best(gscv)

1 | rscv = RandomizedSearchCV(Pipeline([("SVM", SVM(nin=train_X.shape[1]))]),
2 |                           {"SVM__epoches" : [1, 2, 4],
3 |                            "SVM__batch_size" : [5, 10, 20],
4 |                            "SVM__SGD_step" : [0.01, 0.05, 0.1],
5 |                            "SVM__alpha" : [1.0, 0.1, 0.01, 0.0]})
6 | rscv.fit(train_X, train_y)
7 | best(rscv)
```

Наивный байесовский классификатор не имеет параметров, поэтому для него поиск гиперпараметров не осуществляется.

2 Результаты моделей

В датасете данные можно разделить линейно, но из-за низкого количества данных точность классических алгоритмов держится в районе 80-90 процентов.

1 Метод k-ближайших соседей

1.1 kNN

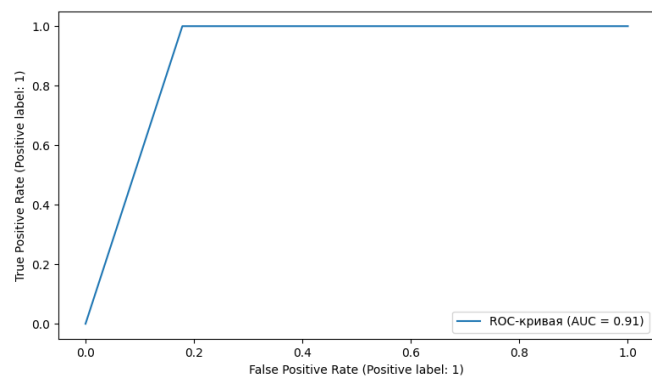
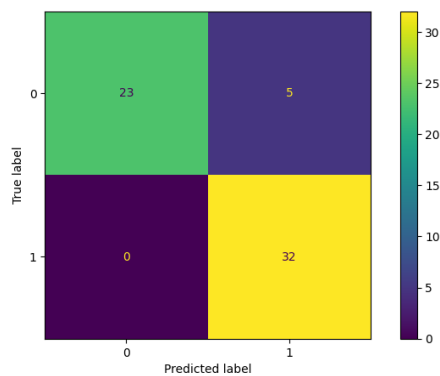
Best params: 'knn__k': 5

Best acc: 0.9282801418439716

Accuracy: 0.9166666666666666

Recall: 1.0

Precision: 0.8648648648648649



1.2 sklearn.neighbors.KNeighborsClassifier

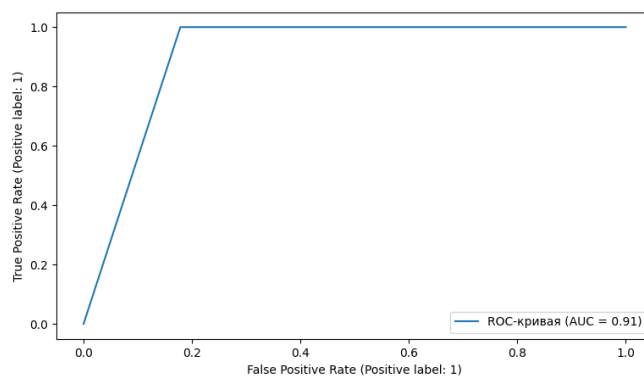
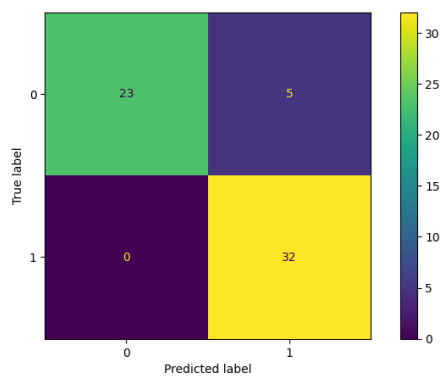
Best params: 'knn__n_neighbors': 5

Best acc: 0.9282801418439716

Accuracy: 0.9166666666666666

Recall: 1.0

Precision: 0.8648648648648649



2 Логистическая регрессия

2.1 LogisticRegression

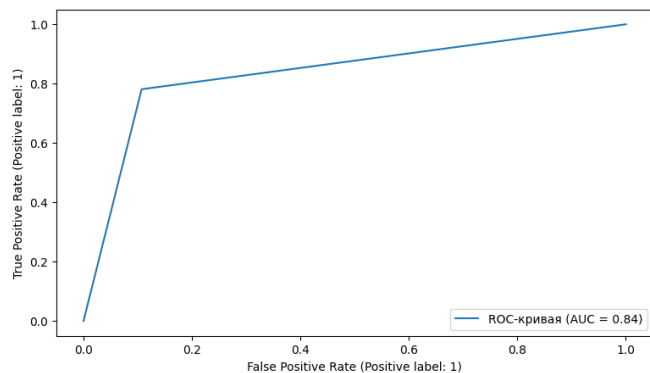
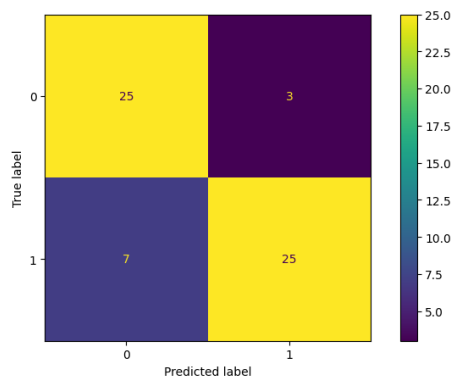
Best params: 'logreg__SGD_step': 0.01, 'logreg__batch_size': 5, 'logreg__epoches': 1

Best acc: 0.8479609929078015

Accuracy: 0.8333333333333334

Recall: 0.78125

Precision: 0.8928571428571429



2.2 sklearn.linear_model.LogisticRegression

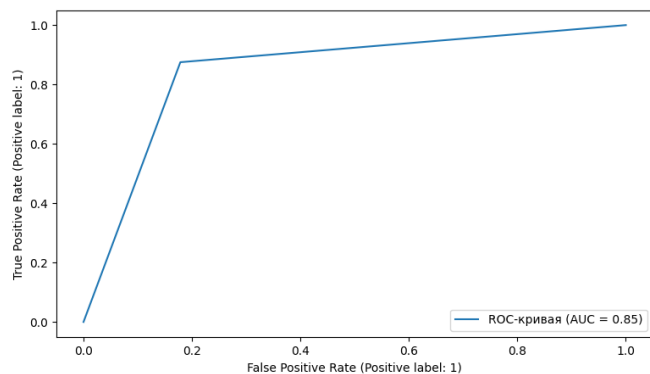
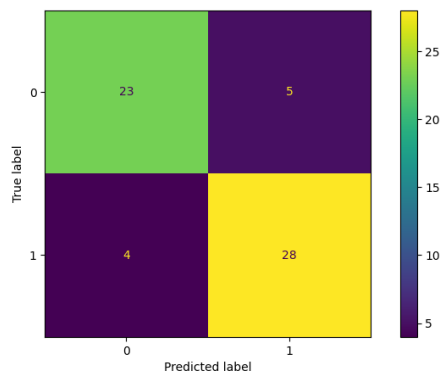
Best params: 'logreg__max_iter': 1000, 'logreg__penalty': 'l2', 'logreg__solver': 'newton-cg'

Best acc: 0.8820035460992907

Accuracy: 0.85

Recall: 0.875

Precision: 0.8484848484848485



3 Метод опорных векторов

3.1 SVM

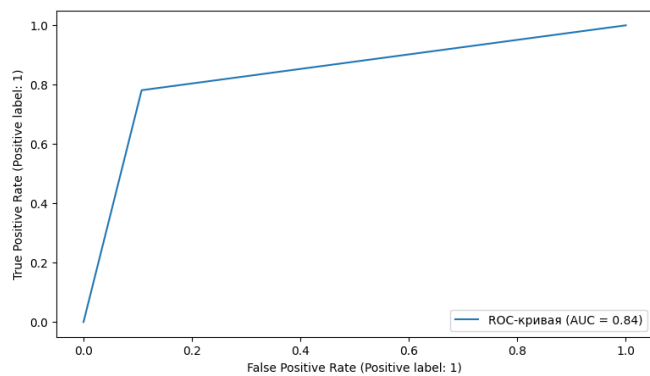
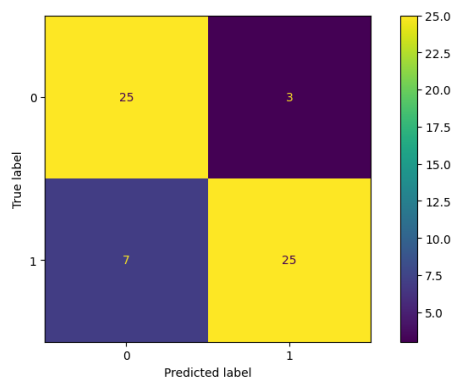
Best params: 'SVM__SGD_step': 0.05, 'SVM__alpha': 0.01, 'SVM__batch_size': 20, 'SVM__epochs': 4

Best acc: 0.8649822695035461

Accuracy: 0.8333333333333334

Recall: 0.78125

Precision: 0.8928571428571429



3.2 sklearn.svm.LinearSVC

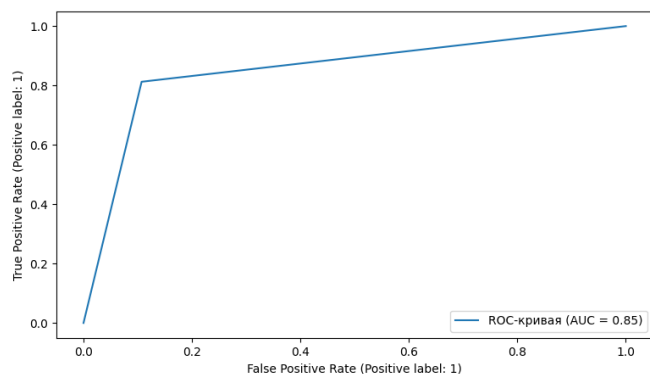
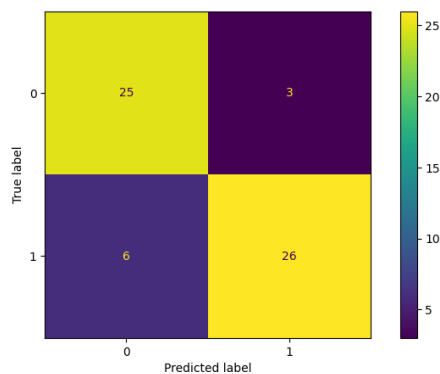
Best params: 'svc__loss': 'hinge', 'svc__max_iter': 100000.0

Best acc: 0.8906028368794325

Accuracy: 0.85

Recall: 0.8125

Precision: 0.896551724137931



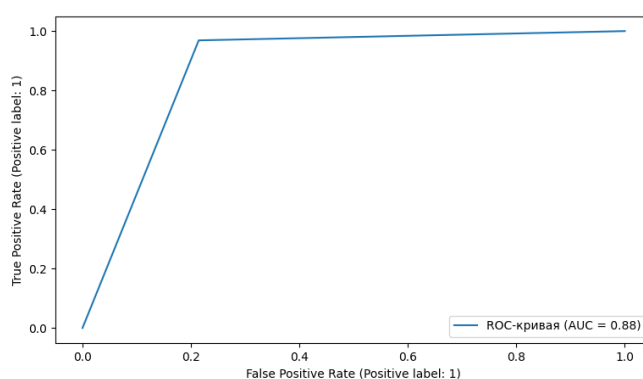
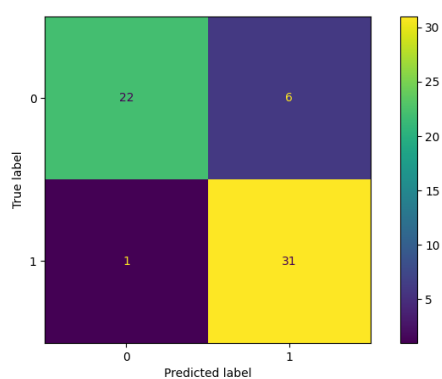
4 Наивный байесовский классификатор

4.1 NaiveBayes

Accuracy: 0.8833333333333333

Recall: 0.96875

Precision: 0.8378378378378378

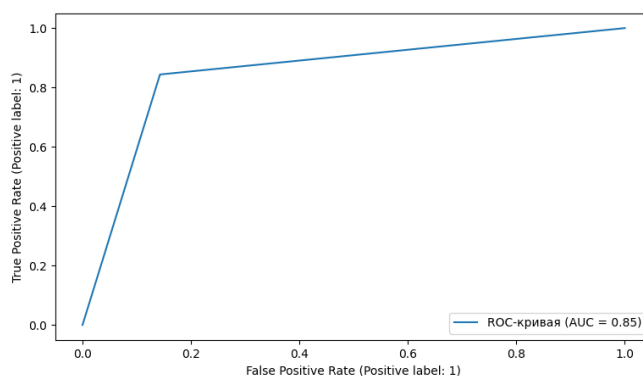
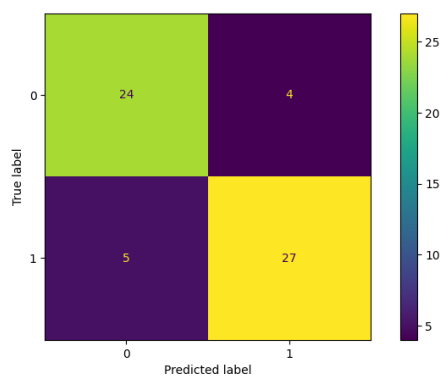


4.2 sklearn.naive_bayes.GaussianNB

Accuracy: 0.85

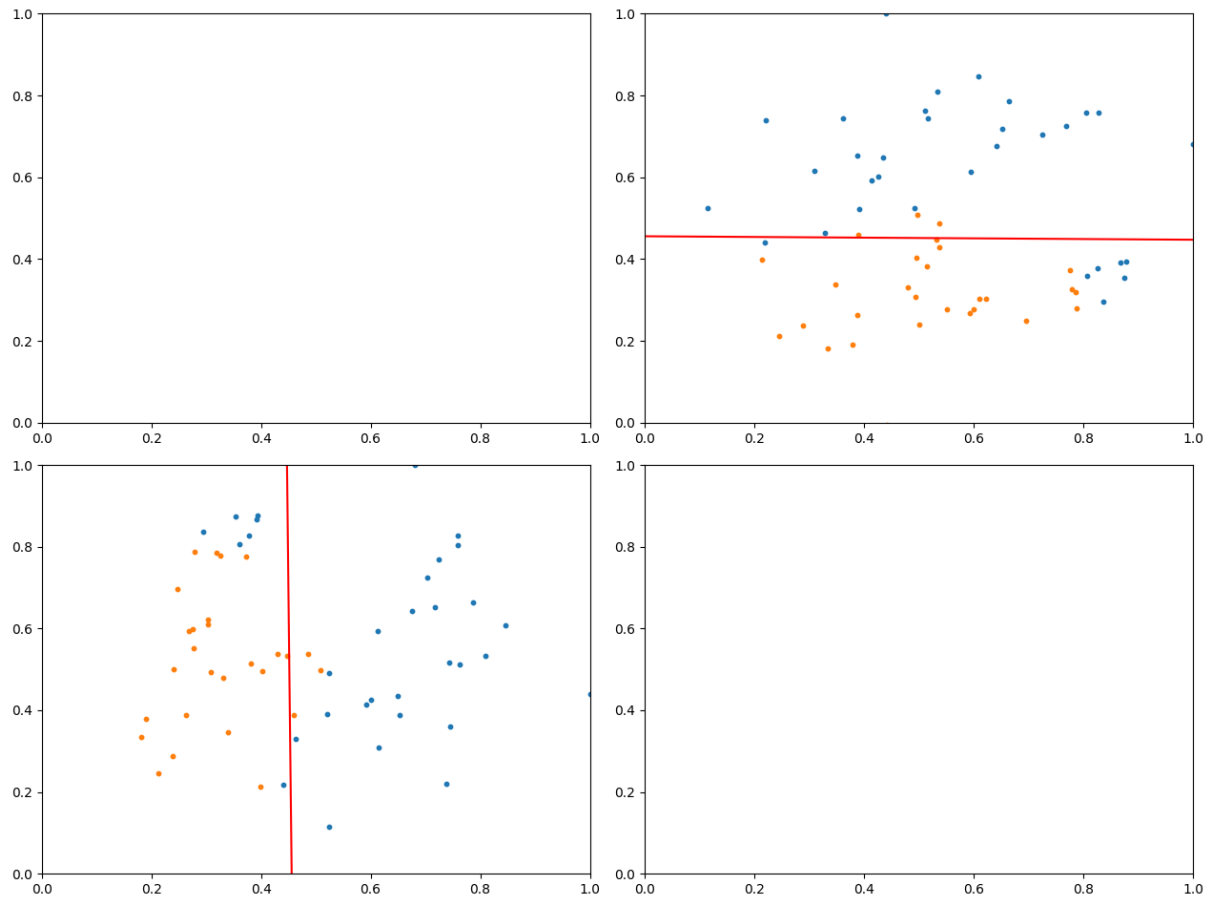
Recall: 0.84375

Precision: 0.8709677419354839

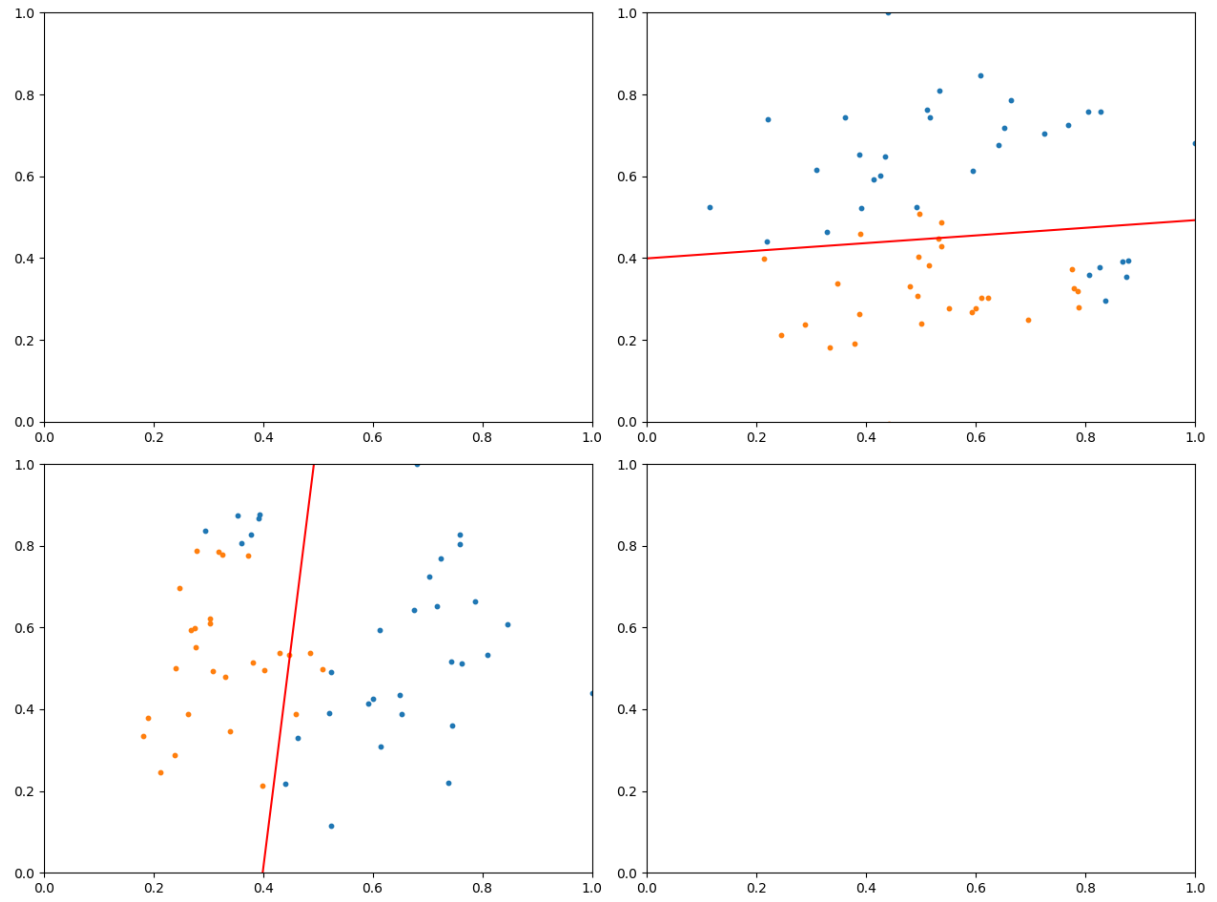


5 Разделяющая прямая для метода опорных векторов

5.1 Моя реализация



5.2 sklearn.svm.LinearSVC



3 Выводы

В ходе выполнения лабораторной работы я познакомился с линейными моделями классического машинного обучения: логистической регрессией, методом опорных векторов, наивным байесовским классификатором и методом k -ближайших соседей. Больше всего заинтересовал алгоритм k -ближайших соседей для классификации, так как он кажется интуитивно лучшим, ведь позволяет разделять классы не прямой, как в случае SVM, а проводить границу, фактически, по точкам. Однако его недостаток в большой вычислительной сложности и в том, что если нет большой корреляции между признаком и результатом, то алгоритм банально будет часто ошибаться.

Основная сложность в работе — реализация каждого метода вручную, пришлось искать очень много методов из библиотек `numpy` и `scikit-learn`, чтобы легко работать с данными. Пожалуй, это заняло большую часть времени.

В результате набор данных `Rice type` получилось разделить линейными моделями с поразительной точностью 98-99%. Точность такая высокая, потому что изначально данные очень хорошо делились, к тому же были удалены выбросы, которые могли мешать обучению.

Особенно хочется отметить, что такой точности получается добиться далеко не всегда, потому что часто бывает так, что между признаками и результатом нет такой большой корреляции, как в выбранном датасете. Предположу, что вместо применения алгоритмов машинного обучения можно было с таким же успехом описать алгоритм, который относил бы рис к одному или другому типу на основе жестко заданных границ признаков.

Список литературы

- [1] *Оценка качества моделей — Учебник по ML от ШАД*
URL: https://ml-handbook.ru/chapters/model_evaluation/intro
(дата обращения: 30.08.2022).
- [2] *sklearn.model_selection.train_test_split*
URL: https://scikit-learn.org/stable/modules/generated/sklearn.model_selection.train_test_split.html
(дата обращения: 30.08.2022).
- [3] *sklearn.preprocessing.normalize*
URL: <https://scikit-learn.org/stable/modules/generated/sklearn.preprocessing.normalize.html>
(дата обращения: 30.08.2022).
- [4] *Developing scikit-learn estimators*
URL: <https://scikit-learn.org/stable/developers/develop.html>
(дата обращения: 30.08.2022).
- [5] *project-template/_template.py at master · scikit-learn-contrib*
URL: https://github.com/scikit-learn-contrib/project-template/blob/master/skltemplate/_template.py
(дата обращения: 30.08.2022).
- [6] *sklearn.pipeline.Pipeline — scikit-learn 1.0.2 documentation*
URL: <https://scikit-learn.org/stable/modules/generated/sklearn.pipeline.Pipeline.html>
(дата обращения: 30.08.2022).
- [7] *sklearn.model_selection.GridSearchCV*
URL: https://scikit-learn.org/stable/modules/generated/sklearn.model_selection.GridSearchCV.html
(дата обращения: 30.08.2022).
- [8] *sklearn.model_selection.RandomizedSearchCV*
URL: https://scikit-learn.org/stable/modules/generated/sklearn.model_selection.RandomizedSearchCV.html
(дата обращения: 30.08.2022).
- [9] *numpy.argsort — NumPy v1.22 Manual*
URL: <https://numpy.org/doc/stable/reference/generated/numpy.argpartition.html>
(дата обращения: 30.08.2022).

- [10] *numpy.unique — NumPy v1.22 Manual*
URL: <https://numpy.org/doc/stable/reference/generated/numpy.unique.html>
(дата обращения: 30.08.2022).
- [11] *sklearn.neighbors.KNeighborsClassifier*
URL: <https://scikit-learn.org/stable/modules/generated/sklearn.neighbors.KNeighborsClassifier.html>
(дата обращения: 30.08.2022).
- [12] *numpy.where — NumPy v1.22 Manual*
URL: <https://numpy.org/doc/stable/reference/generated/numpy.where.html>
(дата обращения: 30.08.2022).
- [13] *sklearn.linear_model.LogisticRegression*
URL: https://scikit-learn.org/stable/modules/generated/sklearn.linear_model.LogisticRegression.html
(дата обращения: 30.08.2022).
- [14] *SVM. Объяснение с нуля и реализация на Python.*
Подробный разбор метода опорных векторов
URL: <https://habr.com/ru/company/ods/blog/484148/>
(дата обращения: 30.08.2022).
- [15] *sklearn.svm.SVC — scikit-learn 1.0.2 documentation*
URL: <https://scikit-learn.org/stable/modules/generated/sklearn.svm.SVC.html#sklearn.svm.SVC>
(дата обращения: 30.08.2022).
- [16] *Машинное обучение. Байесовская классификация.*
К.В. Воронцов, Школа анализа данных, Яндекс.
URL: <https://www.youtube.com/watch?v=qMndsltzNGA&t>
(дата обращения: 30.08.2022).
- [17] *numpy.mean — NumPy v1.22 Manual*
URL: <https://numpy.org/doc/stable/reference/generated/numpy.mean.html>
(дата обращения: 30.08.2022).
- [18] *numpy.std — NumPy v1.22 Manual*
URL: <https://numpy.org/doc/stable/reference/generated/numpy.std.html>
(дата обращения: 30.08.2022).
- [19] *sklearn.naive_bayes.GaussianNB*
URL: https://scikit-learn.org/stable/modules/generated/sklearn.naive_bayes.GaussianNB.html
(дата обращения: 30.08.2022).