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: Д. А. Тарпанов
Преподаватели: Д. В. Сошников
С. Х. Ахмед
Группа: М8О-407Б-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, Naive 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. Проделать 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 тренировучную 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 пропоциях (80 к 20), используя используя train_test_split из scikit-learn [?]. scikit-learn позволяет сделать это с помощью normalize [?], я использую метрику «max».

В задании требуется сделать все модели совместимыми с scikit-learn [?], поэтому получение оценок модели [?] можно сделать методами из этой же библотеки:

```
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-
           curve", ax = ax)
12     plt.show()
```

При реализации моделей я использовал шаблон [?] 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:

```
class kNN(BaseEstimator, ClassifierMixin):

    def __init__(self, k=1):
        self.k = k

    def fit(self, X, y):
        X, y = check_X_y(X, y)
        # Store the classes seen during fit
        self.classes_ = unique_labels(y)

        self.X_ = X
        self.y_ = y
        self.is_fitted_ = True
        # `fit` should always return `self`
        return self

    def predict(self, X):
        # Check is fit had been called
        check_is_fitted(self, ['X_', 'y_'])

        # Input validation
        X = check_array(X)

        y = np.ndarray((X.shape[0],))
        for (i, elem) in enumerate(X):
            distances = euclidean_distances([elem], self.X_)[0]
            indexes = np.argsort(distances, kind='heapsort')[:self.k]

            labels, cnts = np.unique(self.y_[indexes], return_counts=True)
            y[i] = labels[cnts.argmax()]
        return y
```

Использую Евклидову метрику из scikit-learn для вычисления расстояний и метод argpsort из numpy [?] 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)

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:

```
class Net:
    def __init__(self, loss=CrossEntropyLoss()):
        self.layers = []
        self.loss_ = loss

    def add(self, l: Layer):
        self.layers.append(l)

    def forward(self, x):
        for l in self.layers:
            x = l.forward(x)
        return x

    def backward(self, z):
        for l in self.layers[::-1]:
            z = l.backward(z)
        return z

    def update(self, lr):
        for l in self.layers:
            if 'update' in l.__dir__():
                l.update(lr)

    def get_loss_acc(self, x, y):
        p = self.forward(x)
        l = self.loss_.forward(p, y)
        pred = np.argmax(p, axis=1)
        acc = (pred == y).mean()
        return l, acc

    def train_epoch(self, train_x, train_labels, batch_size=4, lr=0.1):
        for i in range(0, len(train_x), batch_size):
            xb = train_x[i:i + batch_size]
            yb = train_labels[i:i + batch_size]

            p = self.forward(xb)
            l = self.loss_.forward(p=p, y=yb)
            dp = self.loss_.backward(loss=l)
            dx = self.backward(z=dp)
            self.update(lr)
```

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

```
class Linear(Layer):
    def __init__(self, nin, nout):
        sigma = 1.0 / np.sqrt(2.0 * nin)
        self.W = np.random.normal(0, sigma, (nout, nin))
        self.b = np.zeros((1, nout))
        self.dW = np.zeros_like(self.W)
        self.db = np.zeros_like(self.b)

    def forward(self, x):
        self.x = x
        return np.dot(x, self.W.T) + self.b

    def backward(self, dz):
        dx = np.dot(dz, self.W)
        dW = np.dot(dz.T, self.x)
        db = dz.sum(axis=0)
        self.dW = dW
        self.db = db
        return dx

    def update(self, lr):
        self.W -= lr * self.dW
        self.b -= lr * self.db
```

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

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

```
class Sigmoid(Layer):
    def forward(self, x):
        self.y = 1.0 / (1.0 + np.exp(-x))
        return self.y

    def backward(self, dy):
        return self.y * (1.0 - self.y) * dy
```

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

```
class BinaryCrossEntropy(Layer):
    def forward(self, p, y):
        y = y.reshape((y.shape[0], 1))
        self.p = p
        self.y = y
        res = y * np.log(p) + (1 - y) * np.log(1 - p)
        return -np.mean(res)

    def backward(self, loss):
        res = (self.p - self.y) / (self.p * (1 - self.p))
        return res / self.p.shape[0]
```

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

```
class LogisticRegression(ClassifierMixin, BaseEstimator):
    def __init__(self, epoches=1, batch_size=10, SGD_step=0.001, nin=10):
        self.epoches = epoches
        self.batch_size = batch_size
        self.SGD_step = SGD_step
        self.nin = nin
        self.Net = Net(BinaryCrossEntropy())
        self.Net.add(Linear(nin, 1))
        self.Net.add(Sigmoid())

    def fit(self, X, y):
        # Check that X and y have correct shape
        X, y = check_X_y(X, y)
        # Store the classes seen during fit
        self.classes_ = unique_labels(y)

        self.X_ = X
        self.y_ = y
        for _ in range(self.epoches):
            self.Net.train_epoch(X, y, self.batch_size, self.SGD_step)
        # Return the classifier
        return self

    def predict(self, X):
        # Check is fit had been called
        check_is_fitted(self, ['X_', 'y_'])

        # Input validation
        X = check_array(X)

```

```
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:

```
class SoftMarginSVM(Net):
    def __init__(self, nin, alpha):
        super().__init__()
        self.alpha = alpha
        sigma = 1.0 / np.sqrt(nin)
        self.W = np.random.normal(0., sigma, (1, nin + 1))

    def forward(self, x):
        z = np.dot(x, self.W.T)
        return z

    def add_ones(self, x):
        ones = np.ones((x.shape[0], 1))
        return np.hstack((x, ones))

    def predict(self, x):
        res = self.forward(self.add_ones(x))
        return np.where(res < 0, 0, 1)

    def train_epoch(self, x, y, batch_size=100, step=1e-7):
        x = self.add_ones(x)
        y = np.where(y > 0, 1, -1)
        for i in range(0, len(x), batch_size):
            xb = x[i:i + batch_size]
            yb = y[i:i + batch_size]

            pred = self.forward(xb)
            grad = self.alpha * self.W
            for i in range(len(xb)):
                if (yb[i] * pred[i] < 1):
                    grad -= yb[i] * xb[i]
            self.W -= step * grad
```

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

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

```
class SVM(ClassifierMixin, BaseEstimator):
    def __init__(self, epoches=1, batch_size=10, SGD_step=0.001, alpha=0.1, nin=10):
        self.epoches = epoches
        self.batch_size = batch_size
        self.SGD_step = SGD_step
        self.nin = nin
        self.alpha = alpha
        self.Net = SoftMarginSVM(nin, alpha)

    def fit(self, X, y):
        # Check that X and y have correct shape
        X, y = check_X_y(X, y)
        # Store the classes seen during fit
        self.classes_ = unique_labels(y)

        self.X_ = X
        self.y_ = y
        for _ in range(self.epoches):
            self.Net.train_epoch(X, y, self.batch_size, self.SGD_step)
        # Return the classifier
        return self

    def predict(self, X):
        y = self.Net.predict(X)
        return y

    def getW(self):
        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. В прошлой работе гистограммы 2 из 8 распределены нормально, поэтому попробую применить такой метод:

```
class NaiveBayes(ClassifierMixin, BaseEstimator):
    def __init__(self):
        None

    def fit(self, X, y):
        # Check that X and y have correct shape
        X, y = check_X_y(X, y)

        self.X_ = X
        self.y_ = y

        labels, cnts = np.unique(self.y_, return_counts = True)
        self.labels = labels
        self.p_of_y = np.array([elem / self.y_.shape[0] for elem in cnts])
        self.means = np.array([self.X_[self.y_ == elem].mean(axis = 0) for elem in
            labels])
        self.stds = np.array([self.X_[self.y_ == elem].std(axis = 0) for elem in labels
            ])
        # Return the classifier
        return self

    def gaussian(self, mu, sigma, x0):
        return np.exp(-(x0 - mu) ** 2 / (2 * sigma)) / np.sqrt(2.0 * pi * sigma)

    def predict(self, X):
        # Check is fit had been called
        check_is_fitted(self, ['X_', 'y_'])

        # Input validation
        X = check_array(X)

        res = np.zeros(X.shape[0])
        for (i, elem) in enumerate(X):
            p = np.array(self.p_of_y)
            for (j, label) in enumerate(self.labels):
                p_x_cond_y = np.array([self.gaussian(self.means[j][k], self.stds[j][k],
                    elem[k]) for k in range(X.shape[1])])
                p[j] *= np.prod(p_x_cond_y)
            res[i] = np.argmax(p)
        return res
```

5 Подбор гиперпараметров

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

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

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

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

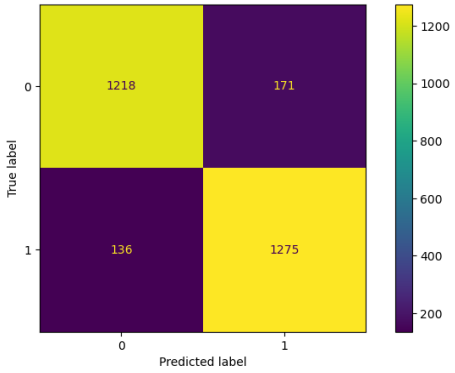
2 Результаты моделей

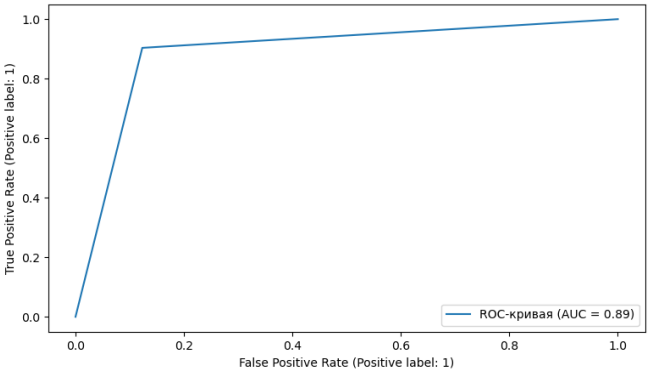
Так как данные очень хорошо линейно разделимы, явные выбросы были удалены, все модели дают одинаковый результат с поразительной точностью 98-88%. Ниже приведены результаты всех моделей. Сначала модели, реализованные вручную, затем из библиотеки.

1 Метод k-ближайших соседей

1.1 kNN

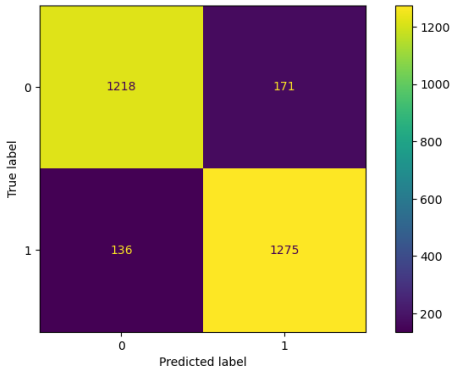
```
Best params: 'knn__k': 5
Best acc: 0.8883529397690296
Accuracy: 0.8903571428571428
Recall: 0.9036144578313253
Precision: 0.8817427385892116
```

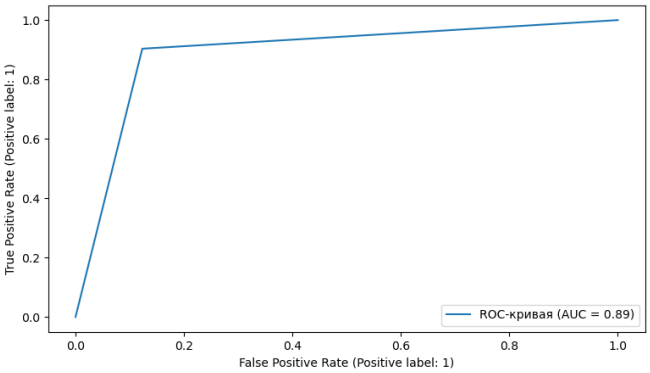




1.2 sklearn.neighbors.KNeighborsClassifier

```
Best params: 'knn__n_neighbors': 5
Best acc: 0.8883529397690296
Accuracy: 0.8903571428571428
Recall: 0.9036144578313253
Precision: 0.8817427385892116
```

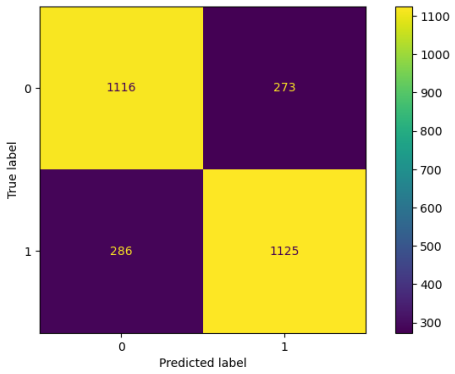


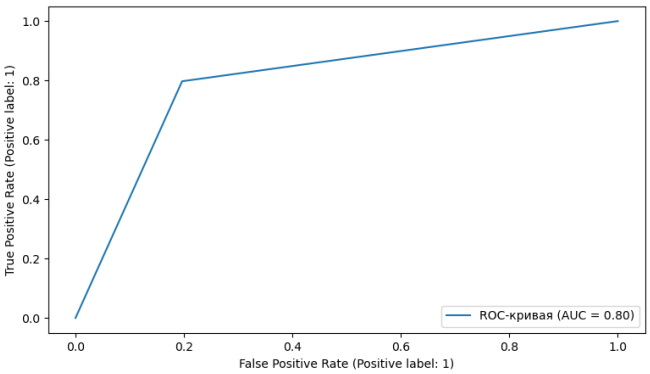


2 Логистическая регрессия

2.1 LogisticRegression

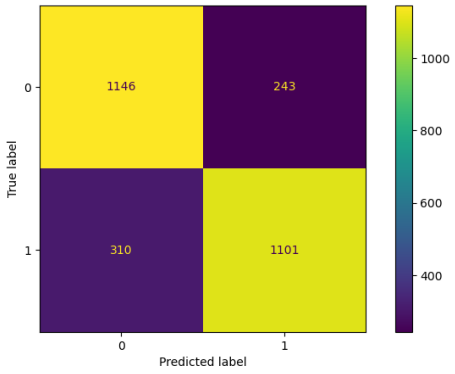
```
Best params: 'logreg__SGD_step': 0.05,'logreg__batch_size': 10,'logreg__epoches':
4
Best acc: 0.8060917262170613
Accuracy: 0.8003571428571429
Recall: 0.7973068745570517
Precision: 0.8047210300429185
```

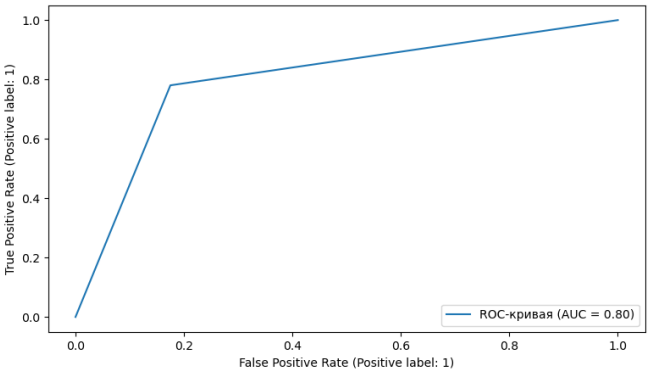




2.2 sklearn.linear_model.LogisticRegression

```
Best params: 'logreg__max_iter': 1000,'logreg__penalty': 'l2','logreg__solver':
'newton-cg'
Best acc: 0.8034123572385632
Accuracy: 0.8025
Recall: 0.780297661233168
Precision: 0.8191964285714286
```

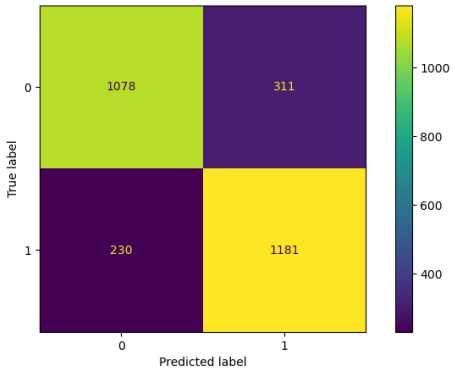


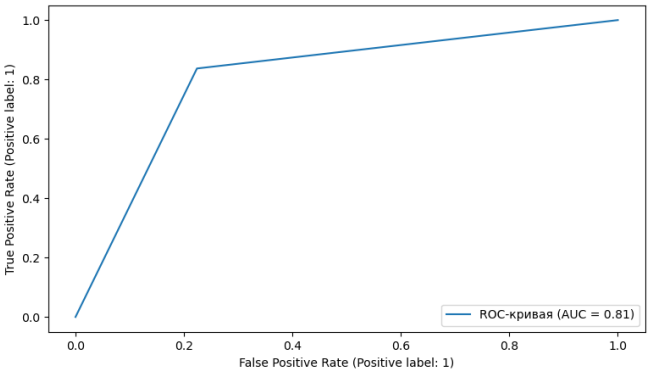


3 Метод опорных векторов

3.1 SVM

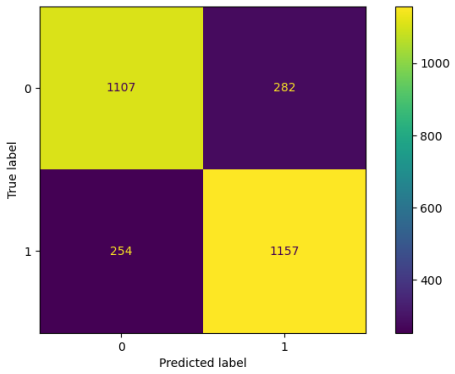
```
Best params: 'SVM__SGD_step': 0.01,'SVM__alpha': 0.0,'SVM__batch_size': 10,'SVM__epoch
1
Best acc: 0.8048408090346456
Accuracy: 0.8067857142857143
Recall: 0.8369950389794472
Precision: 0.7915549597855228
```

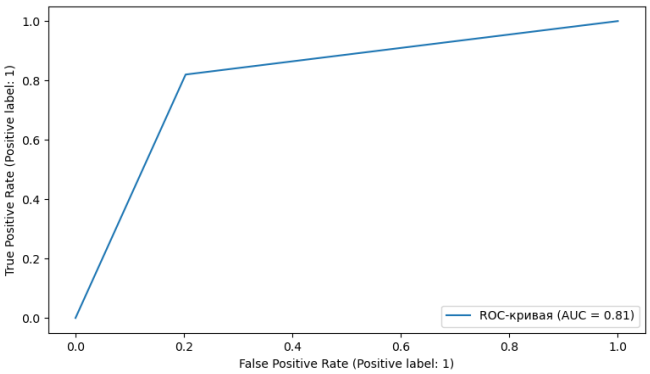




3.2 sklearn.svm.LinearSVC

```
Best params: 'svc__loss': 'hinge','svc__max_iter': 100000.0
Best acc: 0.8068059720538505
Accuracy: 0.8085714285714286
Recall: 0.8199858256555634
Precision: 0.8040305767894371
```

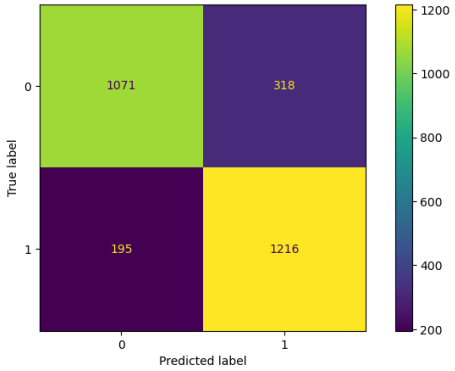


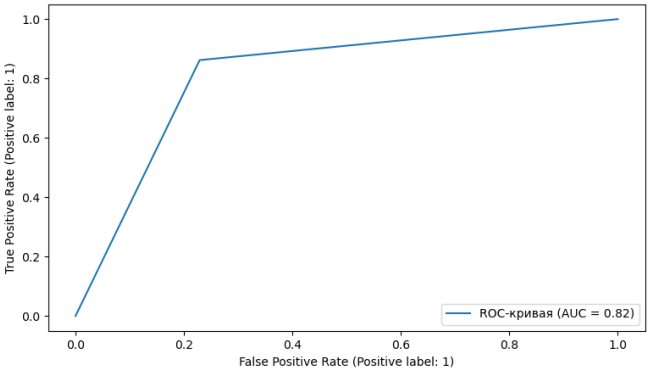


4 Наивный байесовский классификатор

4.1 NaiveBayes

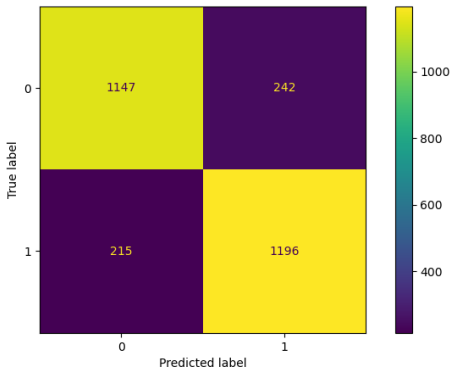
```
Accuracy: 0.8167857142857143
Recall: 0.8618001417434443
Precision: 0.7926988265971316
```

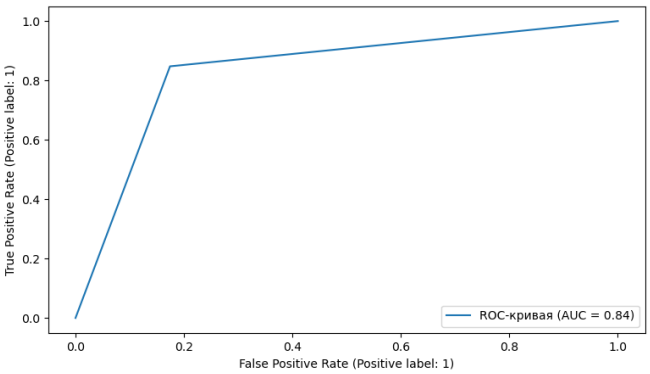




4.2 sklearn.naive_bayes.GaussianNB

```
Accuracy: 0.8367857142857142
Recall: 0.8476257973068746
Precision: 0.8317107093184979
```





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 Water quality получилось разделить линейными моделями с точностью 80-90%. Точность не самая высокая, так как данные распределены не лучшим образом и значительно улучшить их у меня не получилось. Тем не менее, до аугментации данных recall в среднем составлял 30%. Поэтому, я считаю, что результат все равно неплохой.

Список литературы

[1] *Water quality | Kaggle*
URL: `https://www.kaggle.com/datasets/mssmartypants/water-quality`
(дата обращения: 10.11.2022).

[2] *Exploratory data analysis with Pandas — mlcourse.ai*
URL: `https://mlcourse.ai/book/topic01/topic01_pandas_data_analysis.html`
(дата обращения: 10.11.2022).