Московский Авиационный Институт
(Национальный Исследовательский Университет)

Факультет: «Компьютерных наук и прикладной математики»
Кафедра: 806 «Вычислительная математика и программирование»

Лабораторная работа № 1

по курсу «Машинное обучение»

Студент:	Королев И.М.
Группа:	М8О-308Б-19
Преподаватель:	Ахмед С.Х.
Дата:	
Оценка:	

Москва 2022

1. Постановка задачи

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 машинным обучением классическим 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, и 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 Jupyter Notebook ваших экспериментов.

2. Реализация алгоритмов обучения

Для ранее проанализированного датасета (**Heart Failure Prediction Dataset**) в прошлой лабораторной работе, выполним преобразования для дальнейшей работы с ним. Для категориальных признаков необходимо выполнить One Hot Encoding, чтобы использовать значения категориальных признаков при обучении.

После выполняется нормализация признаков модели. Также выполняется разбиение данных на тренировочную (80%) и тестовую (20%) выборки.

Далее были реализованы алгоритмы машинного обучения.

Logistic Regression

Логистическая модель – статистическая модель обучения, которая используется для прогнозирования вероятности принадлежности объекта с заданными признаками к определённому классу путём сравнения.

Классы Network, Linear и Sigmoid были взяты из лабораторной работы по искусственному интеллекту по созданию персептрона.

Линейный слой сети

```
class Linear():
    def __init__(self, num_in, num_out): # num_in - число входных сигналов, num_out - число сигналов на выходе
        self.W = np.random.normal(0, 1.0 / np.sqrt(num_in), (num_out, num_in)) # Нормальное распределение с центром в 0, и отклонением 1.0 / np.sqrt(num_in)
        self.b = np.zeros((1, num_out))                                        # Чтобы у всех сигналов были с одинаковыми характеристиками Сред = 0, Отклонение ~ 1
        self.dW = np.zeros((num_out, num_in))
        self.db = np.zeros((1, num_out))

    # Выполнение вычисления (z = x * w_T + 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, learning_rate):
        self.W -= learning_rate * self.dW
        self.b -= learning_rate * self.db
```

Сеть

```
# Класс сети
class Network():
    def __init__(self, loss_function):
        self.layers = [] # Инициализация слоёв
        self.loss_function = loss_function()

    # Добавление слоя в сеть
    def add(self, layer):
        self.layers.append(layer)

    # Проход по всем слоям
    def forward(self, x):
        for layer in self.layers:
            x = layer.forward(x)
        return x

    # Обратный проход по всем слоям
    def backward(self, z):
        for layer in self.layers[::-1]:
            z = layer.backward(z)
        return z

    # Вычисление функции потерь
    def loss_forward(self, x, y):
        p = self.forward(x)
        return self.loss_function.forward(p, y)

    def loss_backward(self, l):
        dp = None
        dp = self.loss_function.backward(l)
        return self.backward(dp)

    # Обновление для всех слоёв, у которых есть 'update'
    def update(self, learning_rate):
        for layer in self.layers:
            if 'update' in layer.__dir__():
                layer.update(learning_rate)

    def train_epoch(self, x, y, batch_size=10, lr=0.001):
        for i in range(0, len(x), batch_size):
            x_batch = x[i:i+batch_size]
            y_batch = y[i:i+batch_size]
            loss = self.loss_forward(x_batch, y_batch)
            dx = self.loss_backward(loss)
            self.update(lr)
```

Функция потерь

```
# Функция потерь перекрёстной энтропии
class CrossEntropyLoss():
    def forward(self,p,y):
        self.p = p
        y = y.reshape((y.shape[0], 1))
        self.y = y
        result = y * np.log(p) + (1 - y) * np.log(1 - p)
        np.log
        return -np.mean(result)

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

Логистическая функция сигмоиды

```
# Функция сигмоиды
class Sigmoid():
    def forward(self, x):
        self.y = 1.0 / (1 + np.exp(-x))
        return self.y

    def backward(self, dy):
        return (1.0 - self.y**2) * dy
```

Реализация логистической регрессии

```
# Logistic regression
class LogisticRegression(BaseEstimator, ClassifierMixin):
    def __init__(self, epochs=1, batch_size=15, learning_rate=0.01):
        self.epochs = epochs
        self.batch_size = batch_size
        self.learning_rate = learning_rate
        self.Network = Network(CrossEntropyLoss)
        self.Network.add(Linear(num_in=18, num_out=1))
        self.Network.add(Sigmoid())

    def fit(self, X, y):
        # Проверяет X и y на одинаковую длину
        # Принудительно делает X двумерным, y - одномерным
        X, y = check_X_y(X, y)
        self.X = X
        self.y = y
        for epoch in range(self.epochs):
            self.Network.train_epoch(X, y, self.batch_size, self.learning_rate)
        return self

    def predict(self, X):
        check_is_fitted(self, ['X', 'y'])
        prediction = self.Network.forward(X)
        result = np.where(prediction < 0.5, 0, 1)
        return result
```

SVM

SVM – метод опорных векторов. Этот метод можно считать расширением персептрона. С применением алгоритма персептрона сводятся к минимуму ошибки неправильной классификации. В методе опорных векторов целью оптимизации является доведение до максимума зазора. Зазор определяется как расстояние между разделяющей гиперплоскостью (границей решений) и ближайшими к этой гиперплоскости обучающими образцами, которые называются опорными векторами.

Реализация SVM

```
class SVM(BaseEstimator, ClassifierMixin):
    def __init__(self, epochs=1, batch_size=15, lr=0.01, alpha=0.001):
        self.epochs = epochs
        self.batch_size = batch_size
        self.lr = lr
        self.alpha = alpha

    def fit(self, X, y):
        self.W = np.random.normal(0, 1, (X.shape[1]+1,))
        y = y * 2 - 1
        X = np.concatenate((X, np.ones((X.shape[0],1))), axis=1)
        for epoch in range(self.epochs):
            for i in range(self.batch_size, len(X), self.batch_size):
                X_batch = X[i:i+self.batch_size]
                y_batch = y[i:i+self.batch_size]
                gradient = 2 * self.alpha * self.W
                for i, x in enumerate(X_batch):
                    if 1 - x.dot(self.W) * y_batch[i] > 0:
                        gradient -= x * y_batch[i]
                self.W -= self.lr * gradient
        return self

    def predict(self, data):
        return    (np.sign(np.concatenate((data,    np.ones((data.shape[0],1))),
axis=1).dot(self.W)) + 1) / 2
```

KNN

KNN – метод k-ближайших соседей. Этот алгоритм не узнаёт различающую функцию из обучающих данных, а взамен запоминает обучающий набор данных. В этом алгоритме выбирается число k и метрика расстояния. После, находятся k ближайших соседей образца, который нужно классифицировать, и назначается метка класса по большинству голосов.

Реализация KNN

```
class KNN(BaseEstimator, ClassifierMixin):
    def __init__(self, k=1):
        self.k = k # Количество ближайших точек

    def fit(self, X, y):
        X, y = check_X_y(X, y)
        self.X = X
        self.y = y
        return self

    def predict(self, X):
        # Проверяет, что ранее был вызван fit
        check_is_fitted(self, ['X', 'y'])
        predictions = np.ndarray((X.shape[0],))
        for (num, elem) in enumerate(X):
            distances = euclidean_distances([elem], self.X)[0]
            neighbors = np.argpartition(distances, kth=self.k-1) # Косвенное
разбиение
            k_nearest_neighbors = neighbors[:self.k] # k ближайших соседей
            # Различные метки и количество для каждой метки
            labels, counts = np.unique(self.y[k_nearest_neighbors],
return_counts=True)
            predictions[num] = labels[counts.argmax()] # Наиболее вероянтая
метка
        return predictions
```

Naive Bayes

Naive Bayes – наивный байесовский классификатор, являющийся алгоритмом классификации, основанный на применении теоремы Байеса со строгими (наивными) предположениями о независимости.

Реализация Naive Bayes

```
class NaiveBayes(BaseEstimator, ClassifierMixin):
    def __init__(self):
        pass

    # Нормальное Гауссово распределение
    def normal_gauss(self, x, mu, sigma): # mu - математическое ожидание, sigma
- стандартное отклонение
        # return (np.exp(-(x - mu)**2 / (2 * sigma**2))) / np.float32(sigma *
np.sqrt(2 * np.pi))
        return (np.exp(-((x - mu) / sigma)**2 / 2)) / np.float32(sigma *
np.sqrt(2 * np.pi))

    def fit(self, X, y):
        X, y = check_X_y(X, y)
        self.X = X
        self.y = y
```

```
        # Различные метки и количество для каждой метки
        labels, counts = np.unique(self.y, return_counts=True)
        self.standard_deviations = np.array([self.X[self.y == label].std(axis=0)
for label in labels])
        self.means = np.array([self.X[self.y == label].mean(axis=0) for label in
labels])
        self.y_pred = np.array([count / self.y.shape[0] for count in counts])
        self.labels = labels
        return self

    def predict(self, X):
        check_is_fitted(self, ['X', 'y'])
        result = np.ndarray(X.shape[0])
        for (num_x, x) in enumerate(X):
            predictions = np.array(self.y_pred)
            for (num_label, label) in enumerate(self.labels):
                predictions[num_label]                                          *=
np.prod(np.array([self.normal_gauss(x[i],              self.means[num_label][i],
self.standard_deviations[num_label][i]) for i in range(X.shape[1])]))
            result[num_x] = np.argmax(predictions)
        return result
```

3. Подбор гиперпараметров и обучение

Чтобы для заданного алгоритма обучающей модели подобрать такие параметры, которые давали бы наилучший результат (среди указанных параметров), используется кросс-валидация *GridSearchCV*.

Подбор гиперпараметров

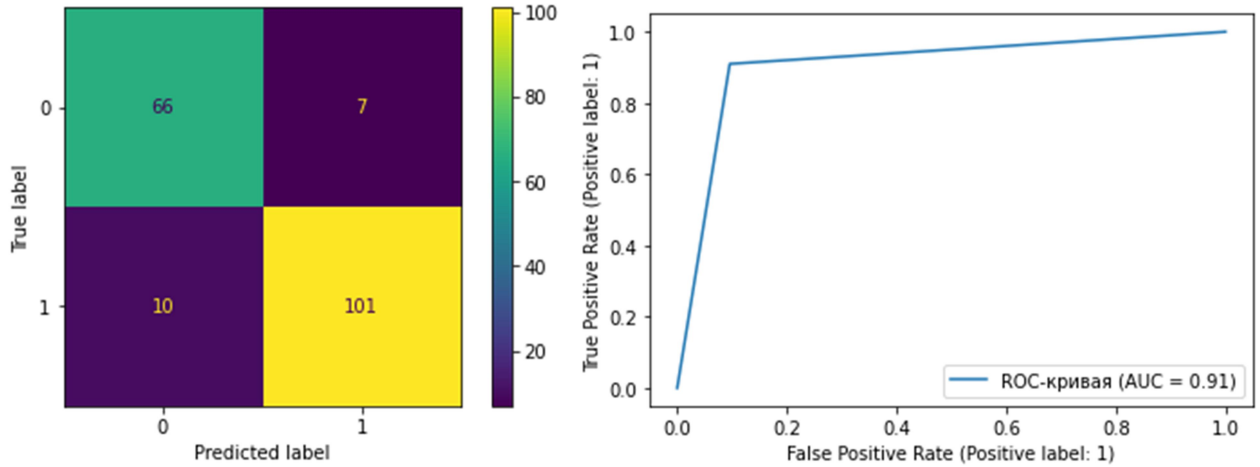
```
log_regression = GridSearchCV(Pipeline([("Logistic_regression",
LogisticRegression())]),
    {"Logistic_regression__epochs" : [1, 5, 10, 15, 20],
    "Logistic_regression__batch_size" : [5, 10, 15, 20, 25],
    "Logistic_regression__learning_rate" : [0.01, 0.05, 0.25, 0.5, 0.1]})

log_regression.fit(X_train, y_train)
log_regression_best = log_regression.best_estimator_
print('Лучший оценщик: ', log_regression_best)
```

Результаты обучения

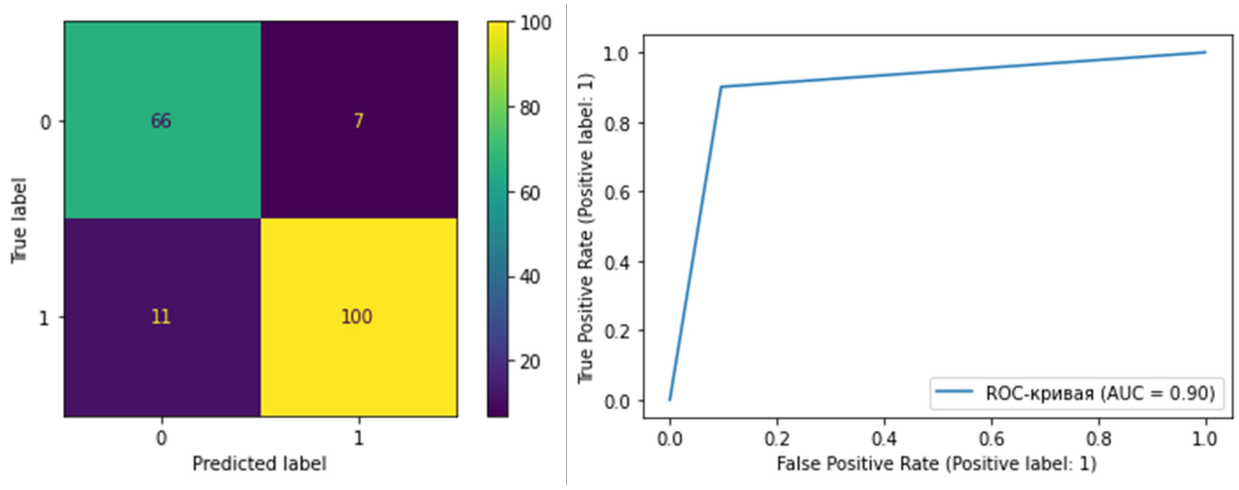
Logistic Regression

```
Accuracy score:  0.907608695652174
Recall score:  0.9099099099099099
Precision score:  0.9351851851851852
ROC AUC score:  0.907009749475503
```



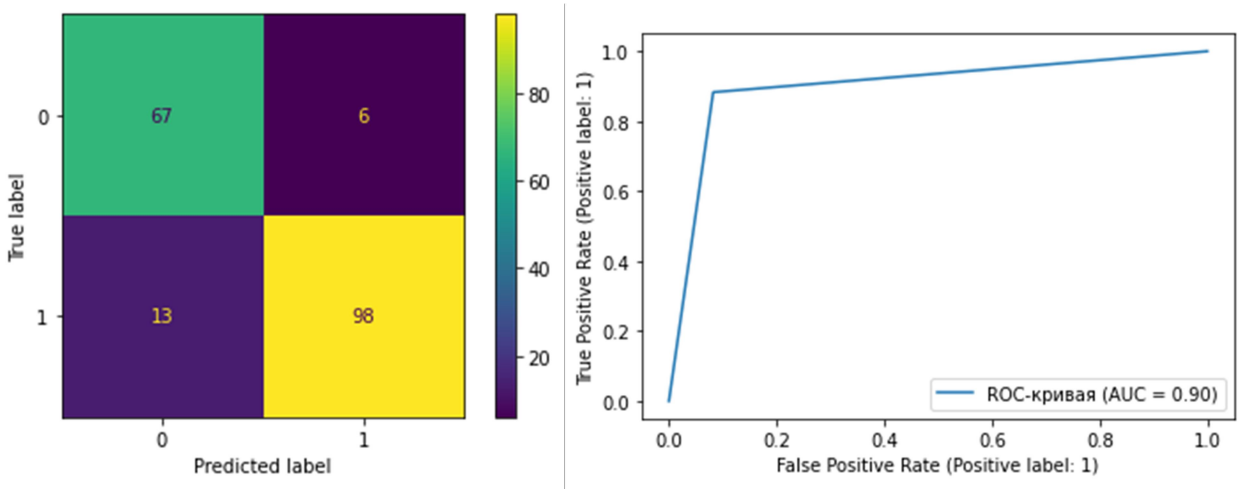
Результат коробочной реализации Logistic Regression

```
Accuracy score:  0.9021739130434783
Recall score:  0.9009009009009009
Precision score:  0.9345794392523364
ROC AUC score:  0.9025052449709985
```



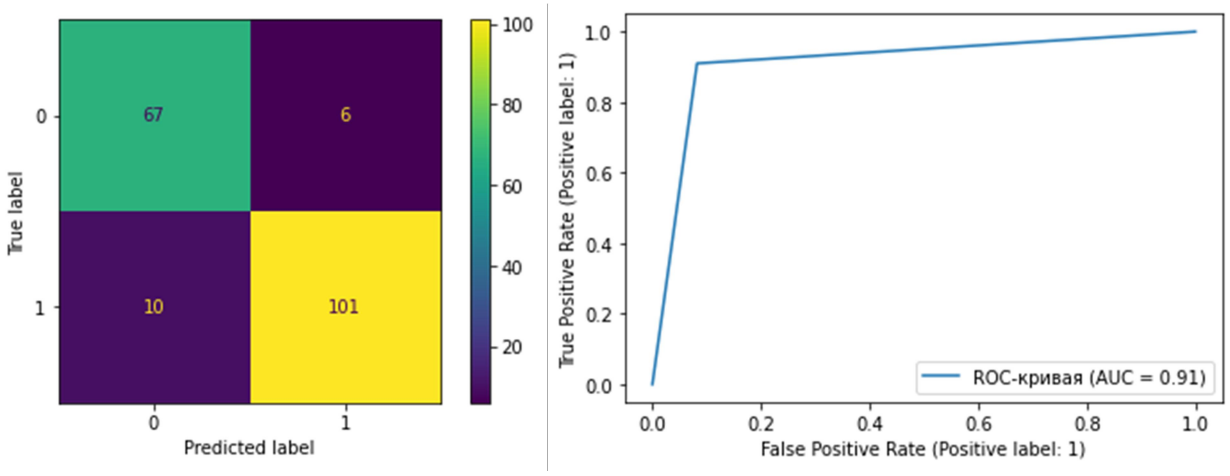
SVM

```
Accuracy score:  0.8967391304347826
Recall score:  0.8828828828828829
Precision score:  0.9423076923076923
ROC AUC score:  0.9003455510304825
```



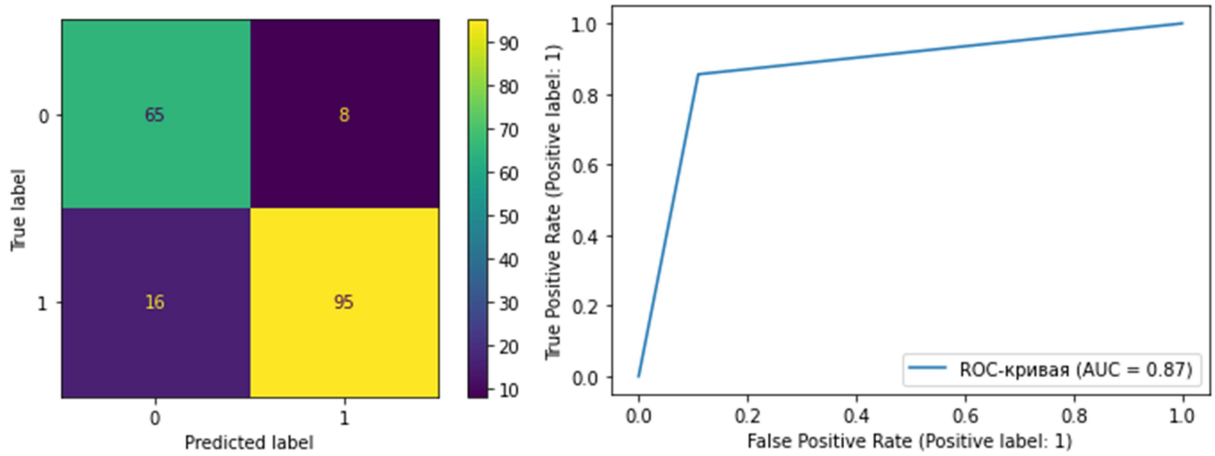
Результат коробочной реализации SVM

```
Accuracy score:  0.9130434782608695
Recall score:  0.9099099099099099
Precision score:  0.9439252336448598
ROC AUC score:  0.913859064543996
```



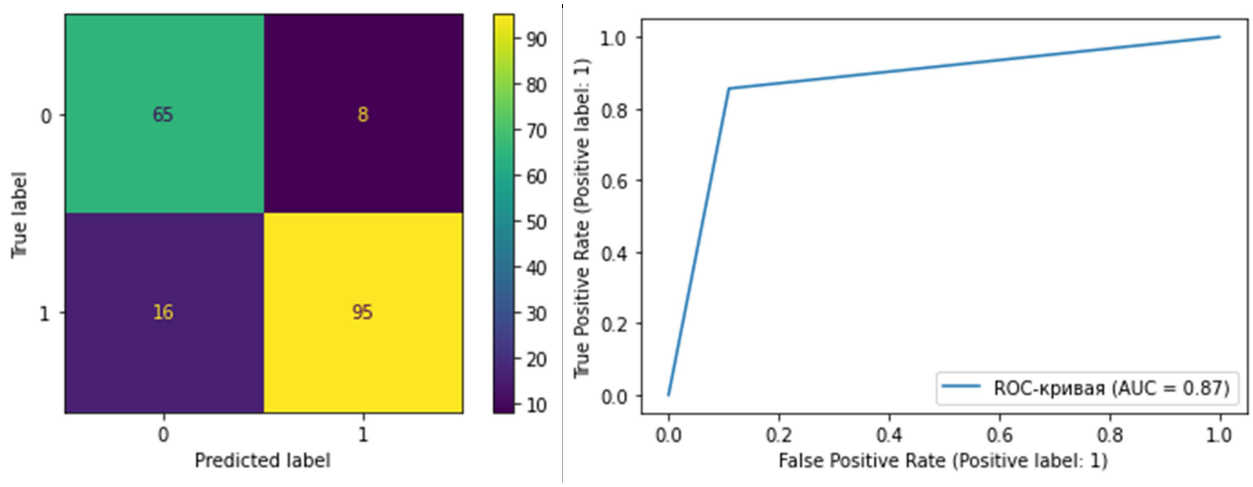
KNN

```
Accuracy score:  0.8695652173913043
Recall score:  0.8558558558558559
Precision score:  0.9223300970873787
ROC AUC score:  0.8731334073799828
```



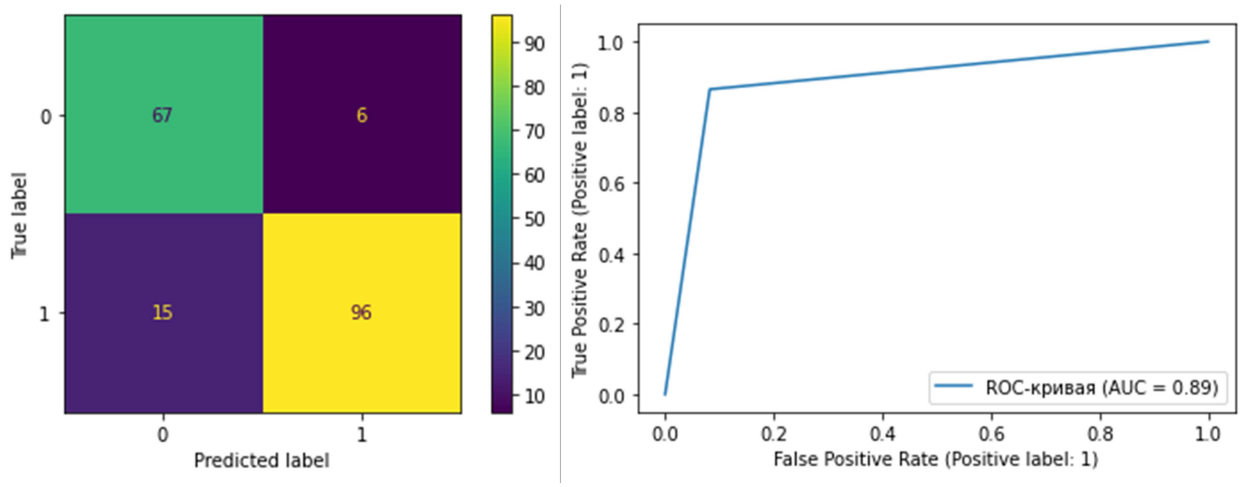
Результат коробочной реализации KNN

```
Accuracy score:  0.8695652173913043
Recall score:  0.8558558558558559
Precision score:  0.9223300970873787
ROC AUC score:  0.8731334073799828
```



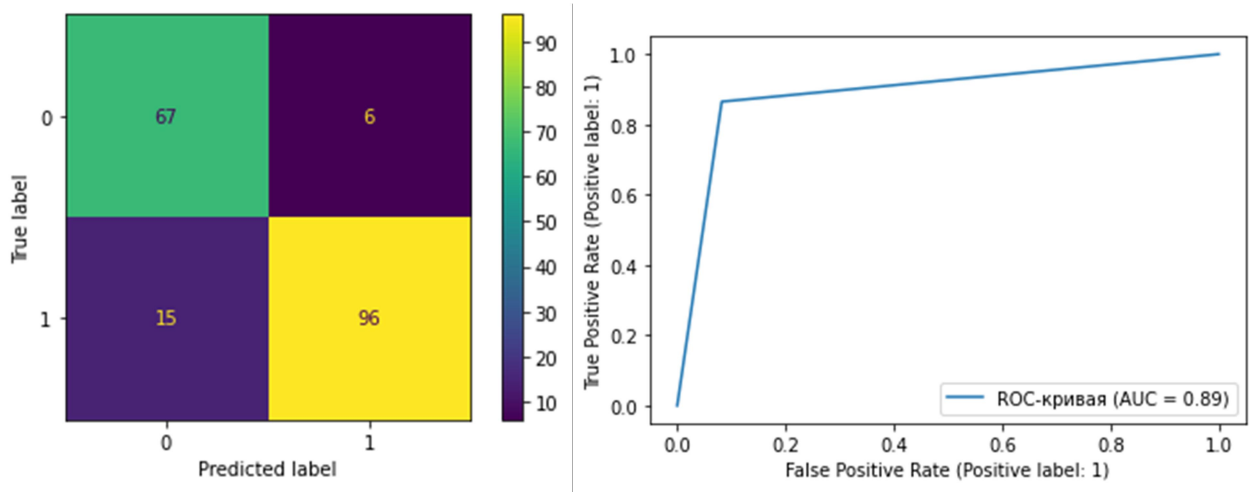
Naive Bayes

```
Accuracy score:  0.8858695652173914
Recall score:  0.8648648648648649
Precision score:  0.9411764705882353
ROC AUC score:  0.8913365420214736
```



Результат коробочной реализации Naive Bayes

```
Accuracy score:  0.8858695652173914
Recall score:  0.8648648648648649
Precision score:  0.9411764705882353
ROC AUC score:  0.8913365420214736
```



Вывод

Были выбран датасет (классификации наличия болезни сердца), на котором в предыдущей лабораторной работе был проведён анализ данных для дальнейшего обучения. С помощью One Hot Encoding категориальные данные были преобразованы для дальнейшего их использования при обучении моделей. Во время выполнения лабораторной работы были получены знания об алгоритмах машинного обучения: логистической регрессии, метода опорных векторов, метода k-ближайших соседей и наивного байесовского классификатора. Для этих алгоритмов была выполнена своя реализация. Для каждого алгоритма были найдены параметры среди заданных, при которых он наиболее хорошо выполняет предсказания. Среди реализованных алгоритмов вручную наилучшим оказался алгоритм логистической регрессии (Logistic Regression), где точность получается около 90%. Худшей моделью оказалась модель с алгоритмом KNN. Для всех моделей выполненных вручную были проведены сравнения с их коробочными реализациями. И для каждой модели точности обоих вариантов были очень близки. Также обученные модели были сохранены в формате pickle.