Лабораторная работа №1

Задача: научиться определять пациента с сердечными заболеваниями по его показателям

Текущий шаг: реализовать линейные моделы классификации(LG, SVM, BN, KNN), проанализировать их результаты

In [150]:

```
import numpy as np
import pandas as pd
from sklearn.base import BaseEstimator, ClassifierMixin
from sklearn.model_selection import train_test_split
from sklearn.neighbors import KNeighborsClassifier
from sklearn.linear_model import LogisticRegression
from sklearn.naive_bayes import GaussianNB
from sklearn import svm
from collections import Counter
from sklearn.model_selection import GridSearchCV
from sklearn.metrics import accuracy_score
from sklearn.metrics import precision_score
from sklearn.metrics import recall_score
from sklearn.metrics import confusion_matrix
```

In [62]:

```
def logit(x, w):
    return np.dot(x, w)

def sigmoid(h):
    return 1. / (1. + np.exp(-h))

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

In [55]:

```
class MyLogisticRegression(BaseEstimator, ClassifierMixin):
    def __init__(self):
        self.w = None

    def fit(self, X, y, max_iter=1000, lr=0.1, activation_function = sigmoid):

        n, k = X.shape

        if self.w is None:
            self.w = np.random.randn(k + 1)

        X_train = np.concatenate((np.ones((n, 1)), X), axis=1)

        losses = []

        for iter_num in range(max_iter):
            z = activation_function(logit(X_train, self.w))
            grad = np.dot(X_train.T, (z - y)) / len(y)

            self.w -= grad * lr

            losses.append(self.__loss(y, z))

        return losses

    def predict_proba(self, X, activation_function = sigmoid):
        n, k = X.shape
        X_ = np.concatenate((np.ones((n, 1)), X), axis=1)
        return np.array(activation_function(logit(X_, self.w)), dtype = "int")

    def predict(self, X, threshold=0.5):
        return self.predict_proba(X) >= threshold

    def get_weights(self):
        return self.w

    def __loss(self, y, p):
        p = np.clip(p, 1e-10, 1 - 1e-10)
        return np.mean(y * np.log(p) + (1 - y) * np.log(1 - p))
```

In [82]:

```
class MyKNN(BaseEstimator, ClassifierMixin):
    def __init__(self, nn=5):
        self.nn = nn

    def evklid(self, X):
        num_test = X.shape[0]
        num_train = self.X.shape[0]

        t = np.dot(X, self.X.T)
        dists = np.sqrt(-2 * t + np.square(self.X).sum(1) +
                        np.matrix(np.square(X).sum(1)).T)
        return dists

    def fit(self, X, y):
        self.X = X.to_numpy()
        self.y = y.to_numpy()

    def predict(self, X):
        X = X.to_numpy()
        dists = self.evklid(X)
        preds = np.zeros(dists.shape[0])
        for i in range(dists.shape[0]):
            labels = self.y[np.argsort(dists[i, :])].flatten()
            top_nn_y = labels[:self.nn]
            preds[i] = Counter(top_nn_y).most_common(1)[0][0]
        return preds
```

In [133]:

```
class MySVM(BaseEstimator, ClassifierMixin):
    def __init__(self, etha=0.01, alpha=0.1, epochs=200):
        self._epochs = epochs
        self._etha = etha
        self._alpha = alpha
        self._w = None
        self.history_w = []
        self.train_errors = None
        self.val_errors = None
        self.train_loss = None
        self.val_loss = None

    def fit(self, X_train, Y_train, X_val, Y_val, verbose=False):
        X_train, Y_train, X_val, Y_val = X_train.to_numpy(), Y_train.to_numpy(), X_val.
to_numpy(), Y_val.to_numpy()


        if len(set(Y_train)) != 2 or len(set(Y_val)) != 2:
            raise ValueError("Number of classes in Y is not equal 2!")

        X_train = add_bias_feature(X_train)
        X_val = add_bias_feature(X_val)
        self._w = np.random.normal(loc=0, scale=0.05, size=X_train.shape[1])
        self.history_w.append(self._w)
        train_errors = []
        val_errors = []
        train_loss_epoch = []
        val_loss_epoch = []

        for epoch in range(self._epochs):
            tr_err = 0
            val_err = 0
            tr_loss = 0
            val_loss = 0
            for i,x in enumerate(X_train):
                margin = Y_train[i]*np.dot(self._w,X_train[i])
                if margin >= 1:
                    self._w = self._w - self._etha*self._alpha*self._w/self._epochs
                    tr_loss += self.soft_margin_loss(X_train[i],Y_train[i])
                else:
                    self._w = self._w +\
                    self._etha*(Y_train[i]*X_train[i] - self._alpha*self._w/self._epoch
s)
                    tr_err += 1
                    tr_loss += self.soft_margin_loss(X_train[i],Y_train[i])
                self.history_w.append(self._w)
            for i,x in enumerate(X_val):
                val_loss += self.soft_margin_loss(X_val[i], Y_val[i])
                val_err += (Y_val[i]*np.dot(self._w,X_val[i])<1).astype(int)
            train_errors.append(tr_err)
            val_errors.append(val_err)
            train_loss_epoch.append(tr_loss)
            val_loss_epoch.append(val_loss)
        self.history_w = np.array(self.history_w)
        self.train_errors = np.array(train_errors)
        self.val_errors = np.array(val_errors)
        self.train_loss = np.array(train_loss_epoch)
        self.val_loss = np.array(val_loss_epoch)
    def predict(self, X:np.array) -> np.array:
        y_pred = []
        X_extended = add_bias_feature(X)
        for i in range(len(X_extended)):
            y_pred.append(np.sign(np.dot(self._w,X_extended[i])))
        return np.array(y_pred)

    def hinge_loss(self, x, y):
        return max(0,1 - y*np.dot(x, self._w))

    def soft_margin_loss(self, x, y):
        return self.hinge_loss(x,y)+self._alpha*np.dot(self._w, self._w)
```

In [54]:

```
class MyNaive_Bayes(BaseEstimator, ClassifierMixin):
    def fit(self, X_train, y_train):
        self.classes = np.unique(y_train)
        self.n_classes = len(self.classes)
        self.prior = np.array(X_train.groupby(y_train).apply(lambda col: len(col)) / le
n(y_train))
        self.mean = np.array(X_train.groupby(y_train).apply(np.mean))
        self.var = np.array(X_train.groupby(y_train).apply(np.var))

    def gauss_distribution(self, class_idx, x):
        mean = self.mean[class_idx]
        var = self.var[class_idx]
        return np.exp((-1/2) * ((x-mean)**2) / (2 * var)) / np.sqrt(2 * np.pi * var)

    def predict(self, X_test):
        y_pred = []
        for x in np.array(X_test):
            posteriors = []
            for class_idx in range(self.n_classes):
                prior = np.log(self.prior[class_idx])
                conditional = np.sum(np.log(self.gauss_distribution(class_idx, x)))
                posterior = prior + conditional
                posteriors.append(posterior)
            y_pred.append(self.classes[np.argmax(posteriors)])
        return y_pred
```

Обработка данных, разбиение выборки

In [76]:

```
heart = pd.read_csv("heart.csv")
names = heart.columns
x_train, x_test, y_train, y_test = train_test_split(heart[names[:13]], heart[names[13
]])
```

Реализация Pipeline

In [141]:

```
def pipeline(model, x_train, y_train, x_test, y_test):
    if isinstance(model, MySVM):
        model.fit(x_train, y_train, x_test, y_test)
        y_pred = model.predict(x_test)
    else:
        model.fit(x_train, y_train)
        y_pred = model.predict(x_test)
    print("Confusion matrix:\n", confusion_matrix(y_test, y_pred), "\n")
    print("Acuracy score: {}".format(accuracy_score(y_test, y_pred)))
    print("Precision score: {}".format(precision_score(y_test, y_pred)))
    print("Recall score: {}".format(recall_score(y_test, y_pred)))
```

Тестирование своей реализации линейных моделей классификации

In [142]:

```
regressor = MyLogisticRegression()
pipeline(regressor, x_train, y_train, x_test, y_test)
```

```
/usr/local/lib/python3.7/dist-packages/ipykernel_launcher.py:5: RuntimeWar
ning: overflow encountered in exp
  """

Confusion matrix:
 [[ 7 34]
 [ 2 33]]

Acuracy score: 0.5263157894736842
Precision score: 0.4925373134328358
Recall score: 0.9428571428571428
```

In [138]:

```
knn = MyKNN()
pipeline(knn, x_train, y_train, x_test, y_test)
```

```
[[27 14]
 [12 23]]

Acuracy score: 0.6578947368421053
Precision score: 0.6216216216216216
Recall score: 0.6571428571428571
```

In [139]:

```
naive_bayes = MyNaive_Bayes()
pipeline(naive_bayes, x_train, y_train, x_test, y_test)
```

```
[[33  8]
 [ 3 32]]

Acuracy score: 0.8552631578947368
Precision score: 0.8
Recall score: 0.9142857142857143
```

In [140]:

```
svm = MySVM()
pipeline(svm, x_train, y_train, x_test, y_test)
```

```
[[ 0 41]
 [ 0 35]]

Acuracy score: 0.4605263157894737
Precision score: 0.4605263157894737
Recall score: 1.0
```

Тестирование коробочный методов

In [164]:

```
import warnings
warnings.filterwarnings('ignore')

model = LogisticRegression()
pipeline(model, x_train, y_train, x_test, y_test)
```

```
Confusion matrix:
 [[34  7]
 [ 4 31]]

Acuracy score: 0.8552631578947368
Precision score: 0.8157894736842105
Recall score: 0.8857142857142857
```

In [144]:

```
model = KNeighborsClassifier(n_neighbors=5)
pipeline(model, x_train, y_train, x_test, y_test)
```

```
Confusion matrix:
 [[27 14]
 [12 23]]

Acuracy score: 0.6578947368421053
Precision score: 0.6216216216216216
Recall score: 0.6571428571428571
```

In [146]:

```
model = GaussianNB()
pipeline(model, x_train, y_train, x_test, y_test)
```

```
Confusion matrix:
 [[37  4]
 [ 6 29]]

Acuracy score: 0.868421052631579
Precision score: 0.8787878787878788
Recall score: 0.8285714285714286
```

In [149]:

```
model = svm.SVC()
pipeline(model, x_train, y_train, x_test, y_test)
```

```
Confusion matrix:
 [[17 24]
 [ 7 28]]

Acuracy score: 0.5921052631578947
Precision score: 0.5384615384615384
Recall score: 0.8
```

Анализ гиперпараметров

In [163]:

```
import warnings
warnings.filterwarnings('ignore')

model = LogisticRegression()
hyperparams = {
    "solver": ['newton-cg', 'lbfgs'],
    "penalty": ['l2', 'none'],
    "C": [100, 10, 1.0, 0.1, 0.01]
}
grid_search = GridSearchCV(model, hyperparams, cv = 5)
grid_result = grid_search.fit(x_train.to_numpy(), y_train.to_numpy())
print("Best: %f using %s" % (grid_result.best_score_, grid_result.best_params_))
means = grid_result.cv_results_['mean_test_score']
stds = grid_result.cv_results_['std_test_score']
params = grid_result.cv_results_['params']
for mean, stdev, param in zip(means, stds, params):
    print("%f (%f) with: %r" % (mean, stdev, param))
```

```
Best: 0.836908 using {'C': 10, 'penalty': 'l2', 'solver': 'newton-cg'}
0.832560 (0.017987) with: {'C': 100, 'penalty': 'l2', 'solver': 'newton-c
g'}
0.832657 (0.046808) with: {'C': 100, 'penalty': 'l2', 'solver': 'lbfgs'}
0.832560 (0.017987) with: {'C': 100, 'penalty': 'none', 'solver': 'newton-
cg'}
0.828213 (0.044359) with: {'C': 100, 'penalty': 'none', 'solver': 'lbfgs'}
0.836908 (0.023062) with: {'C': 10, 'penalty': 'l2', 'solver': 'newton-c
g'}
0.828213 (0.044359) with: {'C': 10, 'penalty': 'l2', 'solver': 'lbfgs'}
0.832560 (0.017987) with: {'C': 10, 'penalty': 'none', 'solver': 'newton-c
g'}
0.828213 (0.044359) with: {'C': 10, 'penalty': 'none', 'solver': 'lbfgs'}
0.836812 (0.030694) with: {'C': 1.0, 'penalty': 'l2', 'solver': 'newton-c
g'}
0.823768 (0.043617) with: {'C': 1.0, 'penalty': 'l2', 'solver': 'lbfgs'}
0.832560 (0.017987) with: {'C': 1.0, 'penalty': 'none', 'solver': 'newton-
cg'}
0.828213 (0.044359) with: {'C': 1.0, 'penalty': 'none', 'solver': 'lbfgs'}
0.832464 (0.023322) with: {'C': 0.1, 'penalty': 'l2', 'solver': 'newton-c
g'}
0.832367 (0.031149) with: {'C': 0.1, 'penalty': 'l2', 'solver': 'lbfgs'}
0.832560 (0.017987) with: {'C': 0.1, 'penalty': 'none', 'solver': 'newton-
cg'}
0.828213 (0.044359) with: {'C': 0.1, 'penalty': 'none', 'solver': 'lbfgs'}
0.748792 (0.018765) with: {'C': 0.01, 'penalty': 'l2', 'solver': 'newton-c
g'}
0.731401 (0.043566) with: {'C': 0.01, 'penalty': 'l2', 'solver': 'lbfgs'}
0.832560 (0.017987) with: {'C': 0.01, 'penalty': 'none', 'solver': 'newton
-cg'}
0.828213 (0.044359) with: {'C': 0.01, 'penalty': 'none', 'solver': 'lbfg
s'}
```

In [172]:

```
import warnings
warnings.filterwarnings('ignore')

model = LogisticRegression(C = 10, penalty = "l2", solver = "newton-cg")
pipeline(model, x_train, y_train, x_test, y_test)
```

```
Confusion matrix:
 [[35  6]
 [ 4 31]]

Acuracy score: 0.868421052631579
Precision score: 0.8378378378378378
Recall score: 0.8857142857142857
```

In [171]:

```
import warnings
warnings.filterwarnings('ignore')

model = model = KNeighborsClassifier()
hyperparams = {
    "leaf_size": list(range(1, 5)),
    "n_neighbors": list(range(1, 8)),
    "p": [1, 2]
}
grid_search = GridSearchCV(model, hyperparams, cv = 5)
grid_result = grid_search.fit(x_train.to_numpy(), y_train.to_numpy())
print("Best: %f using %s" % (grid_result.best_score_, grid_result.best_params_))
means = grid_result.cv_results_['mean_test_score']
stds = grid_result.cv_results_['std_test_score']
params = grid_result.cv_results_['params']
for mean, stdev, param in zip(means, stds, params):
    print("%f (%f) with: %r" % (mean, stdev, param))
```

```
Best: 0.727150 using {'leaf_size': 1, 'n_neighbors': 6, 'p': 1}
0.643188 (0.025005) with: {'leaf_size': 1, 'n_neighbors': 1, 'p': 1}
0.621159 (0.025329) with: {'leaf_size': 1, 'n_neighbors': 1, 'p': 2}
0.617198 (0.045439) with: {'leaf_size': 1, 'n_neighbors': 2, 'p': 1}
0.599614 (0.048535) with: {'leaf_size': 1, 'n_neighbors': 2, 'p': 2}
0.696425 (0.054014) with: {'leaf_size': 1, 'n_neighbors': 3, 'p': 1}
0.647633 (0.011824) with: {'leaf_size': 1, 'n_neighbors': 3, 'p': 2}
0.682995 (0.041764) with: {'leaf_size': 1, 'n_neighbors': 4, 'p': 1}
0.660966 (0.043590) with: {'leaf_size': 1, 'n_neighbors': 4, 'p': 2}
0.727053 (0.039526) with: {'leaf_size': 1, 'n_neighbors': 5, 'p': 1}
0.656425 (0.032512) with: {'leaf_size': 1, 'n_neighbors': 5, 'p': 2}
0.727150 (0.041309) with: {'leaf_size': 1, 'n_neighbors': 6, 'p': 1}
0.674106 (0.036454) with: {'leaf_size': 1, 'n_neighbors': 6, 'p': 2}
0.674589 (0.057854) with: {'leaf_size': 1, 'n_neighbors': 7, 'p': 1}
0.652174 (0.047052) with: {'leaf_size': 1, 'n_neighbors': 7, 'p': 2}
0.638744 (0.022261) with: {'leaf_size': 2, 'n_neighbors': 1, 'p': 1}
0.621159 (0.025329) with: {'leaf_size': 2, 'n_neighbors': 1, 'p': 2}
0.617198 (0.045439) with: {'leaf_size': 2, 'n_neighbors': 2, 'p': 1}
0.599614 (0.048535) with: {'leaf_size': 2, 'n_neighbors': 2, 'p': 2}
0.696425 (0.054014) with: {'leaf_size': 2, 'n_neighbors': 3, 'p': 1}
0.647633 (0.011824) with: {'leaf_size': 2, 'n_neighbors': 3, 'p': 2}
0.682995 (0.041764) with: {'leaf_size': 2, 'n_neighbors': 4, 'p': 1}
0.660966 (0.043590) with: {'leaf_size': 2, 'n_neighbors': 4, 'p': 2}
0.727053 (0.039526) with: {'leaf_size': 2, 'n_neighbors': 5, 'p': 1}
0.656425 (0.032512) with: {'leaf_size': 2, 'n_neighbors': 5, 'p': 2}
0.727150 (0.041309) with: {'leaf_size': 2, 'n_neighbors': 6, 'p': 1}
0.674106 (0.036454) with: {'leaf_size': 2, 'n_neighbors': 6, 'p': 2}
0.674589 (0.057854) with: {'leaf_size': 2, 'n_neighbors': 7, 'p': 1}
0.652174 (0.047052) with: {'leaf_size': 2, 'n_neighbors': 7, 'p': 2}
0.638744 (0.022261) with: {'leaf_size': 3, 'n_neighbors': 1, 'p': 1}
0.621159 (0.025329) with: {'leaf_size': 3, 'n_neighbors': 1, 'p': 2}
0.621546 (0.041089) with: {'leaf_size': 3, 'n_neighbors': 2, 'p': 1}
0.599614 (0.048535) with: {'leaf_size': 3, 'n_neighbors': 2, 'p': 2}
0.696425 (0.054014) with: {'leaf_size': 3, 'n_neighbors': 3, 'p': 1}
0.647633 (0.011824) with: {'leaf_size': 3, 'n_neighbors': 3, 'p': 2}
0.687440 (0.040964) with: {'leaf_size': 3, 'n_neighbors': 4, 'p': 1}
0.660966 (0.043590) with: {'leaf_size': 3, 'n_neighbors': 4, 'p': 2}
0.727053 (0.039526) with: {'leaf_size': 3, 'n_neighbors': 5, 'p': 1}
0.656425 (0.032512) with: {'leaf_size': 3, 'n_neighbors': 5, 'p': 2}
0.727150 (0.041309) with: {'leaf_size': 3, 'n_neighbors': 6, 'p': 1}
0.674106 (0.036454) with: {'leaf_size': 3, 'n_neighbors': 6, 'p': 2}
0.674589 (0.057854) with: {'leaf_size': 3, 'n_neighbors': 7, 'p': 1}
0.652174 (0.047052) with: {'leaf_size': 3, 'n_neighbors': 7, 'p': 2}
0.638744 (0.022261) with: {'leaf_size': 4, 'n_neighbors': 1, 'p': 1}
0.621159 (0.025329) with: {'leaf_size': 4, 'n_neighbors': 1, 'p': 2}
0.621546 (0.041089) with: {'leaf_size': 4, 'n_neighbors': 2, 'p': 1}
0.599614 (0.048535) with: {'leaf_size': 4, 'n_neighbors': 2, 'p': 2}
0.696425 (0.054014) with: {'leaf_size': 4, 'n_neighbors': 3, 'p': 1}
0.647633 (0.011824) with: {'leaf_size': 4, 'n_neighbors': 3, 'p': 2}
0.687440 (0.040964) with: {'leaf_size': 4, 'n_neighbors': 4, 'p': 1}
0.660966 (0.043590) with: {'leaf_size': 4, 'n_neighbors': 4, 'p': 2}
0.727053 (0.039526) with: {'leaf_size': 4, 'n_neighbors': 5, 'p': 1}
0.656425 (0.032512) with: {'leaf_size': 4, 'n_neighbors': 5, 'p': 2}
0.727150 (0.041309) with: {'leaf_size': 4, 'n_neighbors': 6, 'p': 1}
0.674106 (0.036454) with: {'leaf_size': 4, 'n_neighbors': 6, 'p': 2}
0.674589 (0.057854) with: {'leaf_size': 4, 'n_neighbors': 7, 'p': 1}
0.652174 (0.047052) with: {'leaf_size': 4, 'n_neighbors': 7, 'p': 2}
```

In [173]:

```
model = KNeighborsClassifier(n_neighbors = 6, p = 1, leaf_size = 1)
pipeline(model, x_train, y_train, x_test, y_test)
```

```
Confusion matrix:
 [[32  9]
 [18 17]]

Acuracy score: 0.6447368421052632
Precision score: 0.6538461538461539
Recall score: 0.4857142857142857
```

Выводы:

При решении моей задачи лучше всего себя показал метод наивной байесовской классификации выдав точность на моей задачи 85%+. Он также показал наивысшую точность среди коробочный методов. После анализа изменений гиперпараметров у ЛогРег и КНН я смог убедидиться, что выбор лучших параметров для задачи требует длительного подсчёта и полностью зависит от того, как программист задаёт параметры(они могут улучшить результат как в ЛГ, так и из-за неправильной подачи или неиспользования каких-то важных параметров при анализе, ухудшить его)