МИНОБРНАУКИ РОССИИ

ФЕДЕРАЛЬНОЕ ГОСУДАРСТВЕННОЕ АВТОНОМНОЕ ОБРАЗОВАТЕЛЬНОЕ УЧРЕЖДЕНИЕ ВЫСШЕГО ОБРАЗОВАНИЯ

«САНКТ-ПЕТЕРБУРГСКИЙ ПОЛИТЕХНИЧЕСКИЙ УНИВЕРСИТЕТ ПЕТРА ВЕЛИКОГО»

Институт Компьютерных наук и технологий
Высшая школа искусственного интеллекта
Направление 02.03.01 Математика и Компьютерные науки

Отчёт по дисциплине «Системы искусственного интеллекта »

Лабораторная работа №1 «Наивный Байесовский классификатор»

Студент: _____ Федотов Станислав Юрьевич

Преподаватель: _____ Уткин Лев Владимирович

«____»_____ 20__ г.

Санкт-Петербург – 2024

Содержание

Постановка задачи

1. Исследовать, как объем обучающей выборки и количество тестовых данных влияет на точность классификации или на вероятность ошибочной классификации в примере крестики-нолики и примере о спаме e-mail сообщений.

2. Сгенерировать 100 точек с двумя признаками X1 и X2 в соответствии с нормальным распределением так, что первые 50 точек (class -1) имеют параметры: мат. ожидание X1 равно 10, мат. ожидание X2 равно 14, среднеквадратические отклонения для обеих переменных равны 4. Вторые 50 точек (class +1) имеют параметры: мат. ожидание X1 равно 20, мат. ожидание X2 равно 18, среднеквадратические отклонения для обеих переменных равны 3. Построить соответствующие диаграммы, иллюстрирующие данные. Построить байесовский классификатор и оценить качество классификации.

3. . Разработать байесовский классификатор для данных Титаник (Titanic dataset) - `https://www.kaggle.com/c/titanic`
   Классы:
   survival Выжил (0 = No; 1 = Yes)
   Признаки:
   pclass Класс каюты (1 = 1st; 2 = 2nd; 3 = 3rd)
   name Имя
   sex Пол
   age Возраст
   sibsp Число братьев-сестер/муж-жена на борту
   parch Число родителей/детей на борту
   ticket Номер билета
   fare Стоимость билета
   cabin Каюта
   embarked Порт посадки (C = Cherbourg; Q = Queenstown; S = Southampton)
   Специальные отметки:
   Pclass: 1st Верхний; 2nd Средний; 3rd Нижний
   Age – в годах; дробный, если возраст меньше одного года

1 Реализация наивного байесовского классификатора на Python

Для выполнения лабораторной работы был разработан наивный байесовский классификатор с использованием языка Python. Для реализации классификатора был создан класс **NaiveBayesClassifier**. Класс имеет три метода:

- __init__(target_index, alpha=1, window_size=0.5) - конструктор класса, создающий объект модели с заданными параметрами:

  target_index - индекс класса в векторе параметров,

  alpha - параметр α, используемый в формуле для нахождения условной вероятности:

  $$P\left(f_i \mid y = c\right) = \frac{M_i(c) + \alpha}{\sum_{j=1}^{m}\left(M_j(c) + \alpha\right)}$$

  window_size - параметр h, используемый в формуле для нахождения оценки плотности для непрерывного случая:

  $$p_h(x) = \frac{1}{2nh}\sum_{i=1}^{n}\left[|x - x_i| < h\right]$$

- train(data) - метод обучения модели. Вычисление априорных вероятностей каждого класса, вычисление условных вероятностей для каждого значения для каждого признака из обучающей выборки.

  data - обучающая выборка.

- predicate(raw) - метод классификации входного объекта. На основе априорных и условных вероятностей вычисляется вероятность принадлежности объекта к каждому классу. Метод возвращает класс, имеющий наибольшую вероятность.

  raw - вектор признаков входящего объекта.

  Метод predicate выполняется двумя способами:

  1. С помощью перемножения всех условных вероятностей - predicate_mult:

  $$c_{\text{opt}} = \arg\max_{c \in C}\left(P(y = c)\prod_{i=1}^{m} P\left(f_i \mid y = c\right)\right)$$

  2. С помощью суммирования логарифмов условных вероятностей - predicate_log:

  $$c_{\text{opt}} = \arg\max_{c \in C}\left(\log P(y = c) + \sum_{i=1}^{m}\log P\left(f_i \mid y = c\right)\right)$$

Для всех последующих примеров точность модели будет определяться с использование метода *Shuffle-Split Cross-Validation*: поступающая на вход выборка случайным образом перемешивается и разделяется на обучающую и тестовую наборы в соответствии с заданным параметром разбиения. Этот процесс повторяется K раз, и в конце вычисляется среднее значение метрик производительности модели, в контексте выполнения первой лабораторной работы - точность классификации.

Код Shuffle-Split Cross-Validation на Python

```
def cross_validation_accuracy(model, data, split_ratio=0.8, k=10):
    accuracy = 0

    for i in range(k):

        # Случаное перемешивание данных в датасете
        random.shuffle(data)

        # Разделение датасета на обучающую и тестирующую выборку
        split_index = int(len(data) * split_ratio)
        train_data = data[:split_index]
        test_data = data[split_index:]

        # Тренировка модели
        model.train(train_data)

        # Тестирование модели
        count = 0
        for test in test_data:
            if model.predicate_log(test[:model.target_index] +
            test[model.target_index + 1:]) == test[model.target_index]:
                count += 1

        accuracy += count / len(test_data)

    return accuracy / k
```

2 Исследование зависимости точности классификатора от объема обучающей выборки

2.1 Реализация на языке R

2.1.1 Пример обучающего множества Tic-Tac-Toe

```
library(e1071)
# Импортир данных в R.
A_raw <- read.table("R_files\\lab1\\Tic_tac_toe.txt", sep = ",",
stringsAsFactors = TRUE)
# Число строк в датасете n = 958.
n <- dim(A_raw)[1]
# Создание обучающей и тестирующей выборки.
# Для примера используем 80% для обучения и оставшиеся - для тестирования.
# Установка базы генерации случайных чисел и рандомизация выборки.
set.seed(12345)
A_rand <- A_raw[ order(runif(n)), ]
# Разделение данных на обучающие и тестирующие.
nt <- as.integer(n*0.8)
A_train <- A_rand[1:nt, ]
A_test <- A_rand[(nt+1):n, ]
# Можно убедиться, какой имеется процент каждого
# класса V2 в обучающей и тестирующей выборке
prop.table(table(A_train$V10))

 negative  positive
0.3524804 0.6475196

prop.table(table(A_test$V10))

 negative  positive
0.3229167 0.6770833

# Использование Наивного Байесовского классификатора из пакета e1071
A_classifier <- naiveBayes(A_train[,-10], A_train$V10)

# Оценка полученной модели:
A_predicted <- predict(A_classifier, A_test)

# Используем table для сравнения прогнозируемых значений с тем, что есть
table(A_predicted, A_test$V10)

A_predicted negative positive
   negative       22       20
   positive       40      110

#Вычислим точность
> result <- table(A_predicted, A_test$V10)
> accuracy <- sum(diag(result)) / sum(result)
```

```
> accuracy
[1] 0.6875
```

На диаграмме, представленной на рис. 1, отображается зависимость точности классификатора от объема обучающей выборки. Из анализа графика можно выделить следующие наблюдения: наихудшие результаты точности классификации достигаются при доле обучающей выборки менее 0.3 и в интервале [0.8, 0.85]. В промежутке [0.3, 0.8] точность классификации находится на среднем уровне. Лучшие результаты точности достигаются при доле обучающей выборки в пределах [0.85, 0.9]. Среднее значения точности классификатора - 0.693. Максимальное значение - 0.739 достигается при доле обучающей выборке равной 0.9, минимальное - 0.664 доистигается при доле обучающей выборки равной 0.19.

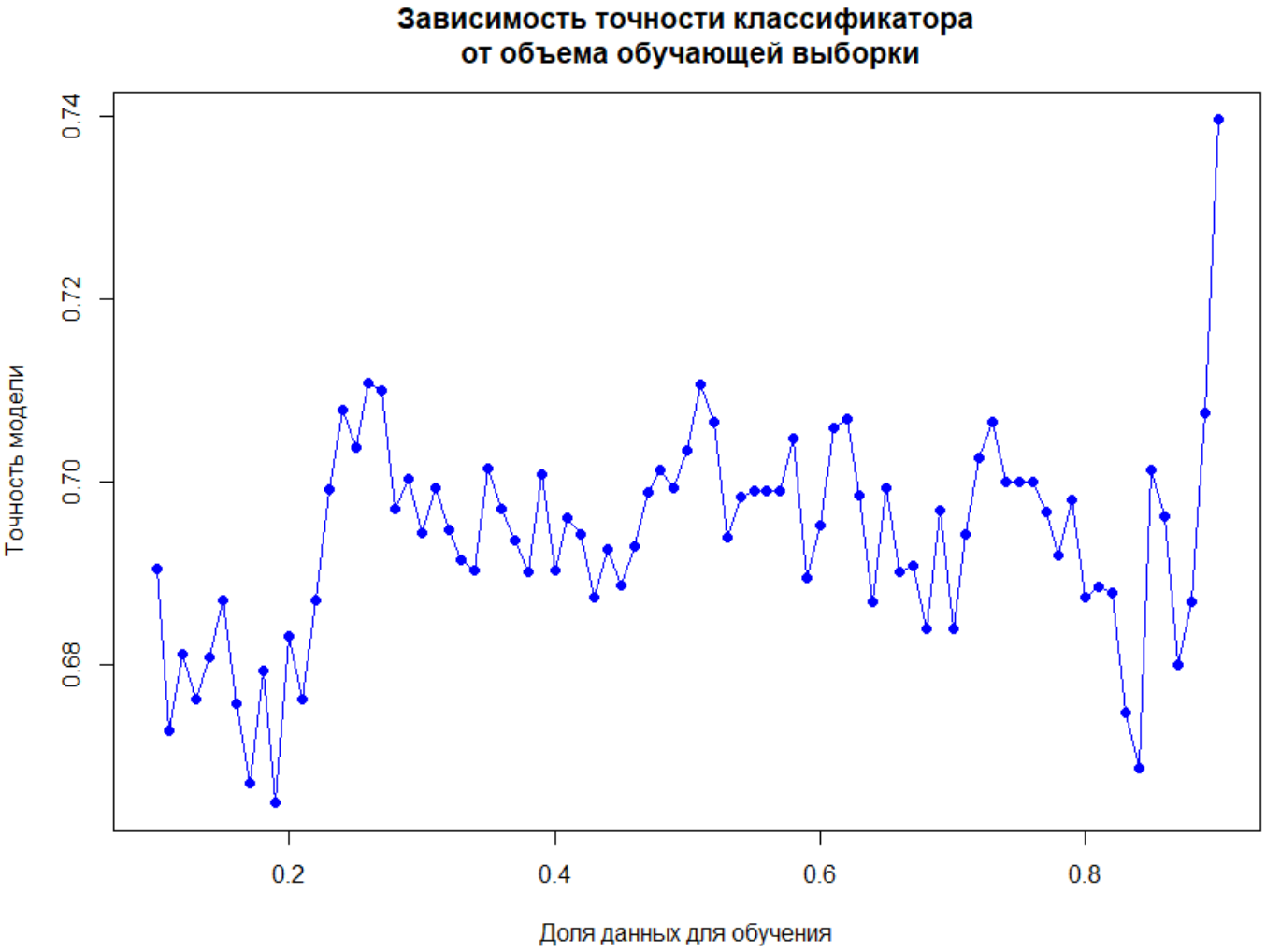

Рис. 1 – Зависимость точности классификатора от объема обучающей выборки

2.1.2 Пример обучающего множества о спаме e-mail

```
library(e1071)
library(kernlab)
library(e1071)
data(spam)
#Общее количество сообщений n=4601
n<-dim(spam)[1]
#Количество обучающих данных
x <- c(x, 0.9)
nt <- as.integer(n*(0.9))
#Случайным образом выбираем сообщения для тестирования
idx <- sample(1:dim(spam)[1], n-nt)
spamtrain <- spam[-idx, ]
spamtest <- spam[idx, ]

# Построение классификатора
model <- naiveBayes(type ~ ., data = spamtrain)

result <- table(predict(model, spamtest), spamtest$type)

# Точность модели
accuracy <- sum(diag(result)) / sum(result)
accuracy
```

```
[1] 0.7245119
```

На диаграмме, представленной на рис. 2, отображается зависимость точности классификатора от объема обучающей выборки. Из анализа графика можно выделить, что для данной модели объем обучающей выборки на промежутке [0.1, 0.9] не коррелирует с точностью классификации. Среднее значение точности классификации - 0.707, минимальное значение - 0.665 достигается при доле обучающей выборки 0.11, максимальное значение - 0.748 достигается при доле обучающей выборке 0.58.

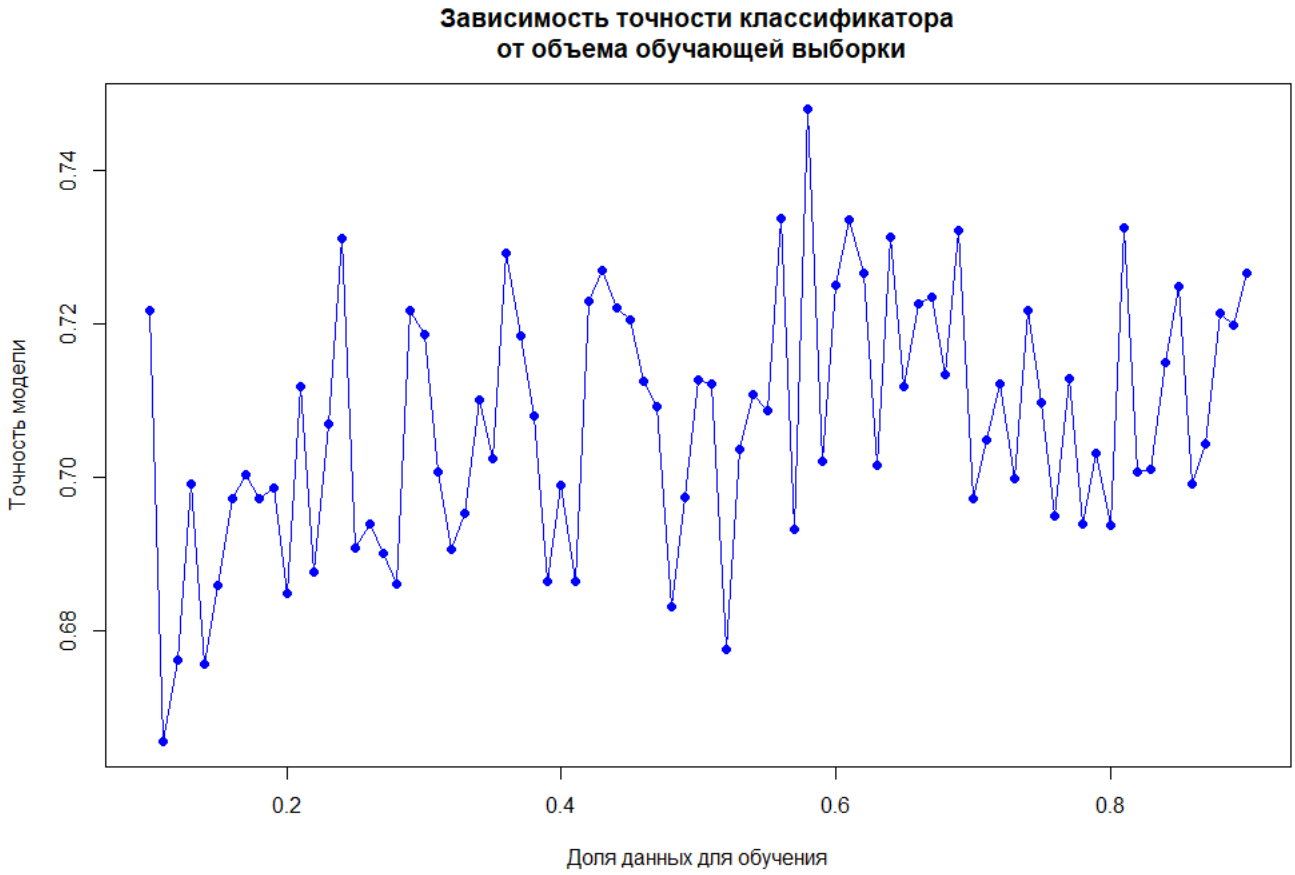

Рис. 2 – Зависимость точности классификатора от объема обучающей выборки

2.2 Реализация на Python

2.2.1 Пример обучающего множества Tic-Tac-Toe

На диаграмме, представленной на рис. 3, отображается зависимость точности классификатора от объема обучающей выборки. Из анализа графика сложно выделить зависимости точности от объема обучающей выборки. Среднее значение точности - 0.654732, минимальное значение - 0.641973 достигается при доле обучающей выборки 0.86, максимальное значение - 0.661783. достигается при доле обучающей выборке 0.92.

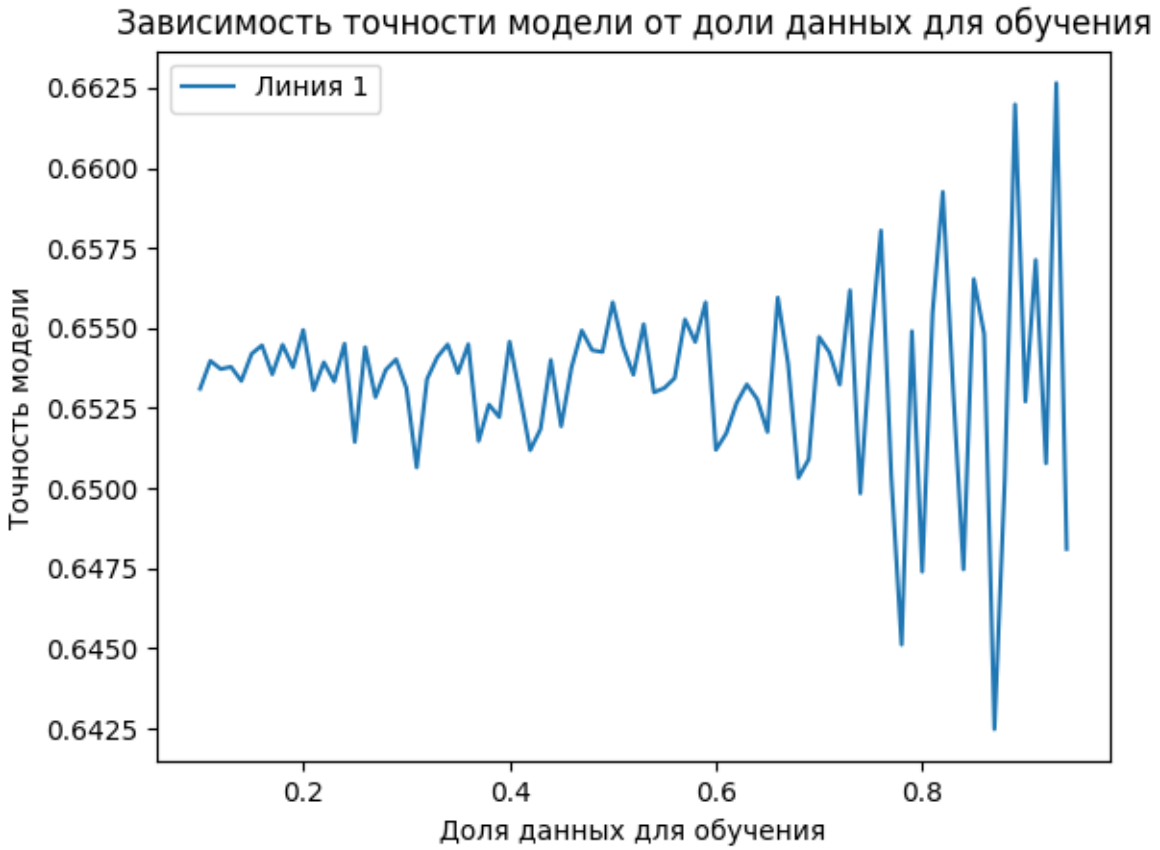

Рис. 3 – Зависимость точности классификатора от объема обучающей выборки

2.2.2 Пример обучающего множества MNIST

Так как предыдущие примеры не смогли отразить зависимость точности классификатора от объема обучающей выборки, было принято решение исследовать более сложную выборку с большим количеством обучающих данных - 60000. MNIST (Modified National Institute of Standards and Technology database) - это набор данных, который состоит из изображений, представляющих цифры от 0 до 9, написанных от руки.

На диаграмме, представленной на рис. 4, отображается зависимость точности классификатора от объема обучающей выборки. Из анализа графика можно выделить линейную зависимость точности классификации выборки от объема обучающей выборки. Максимальное значение точности классификации - 0.6805 достигается при доле данных для обучения - 0.95.

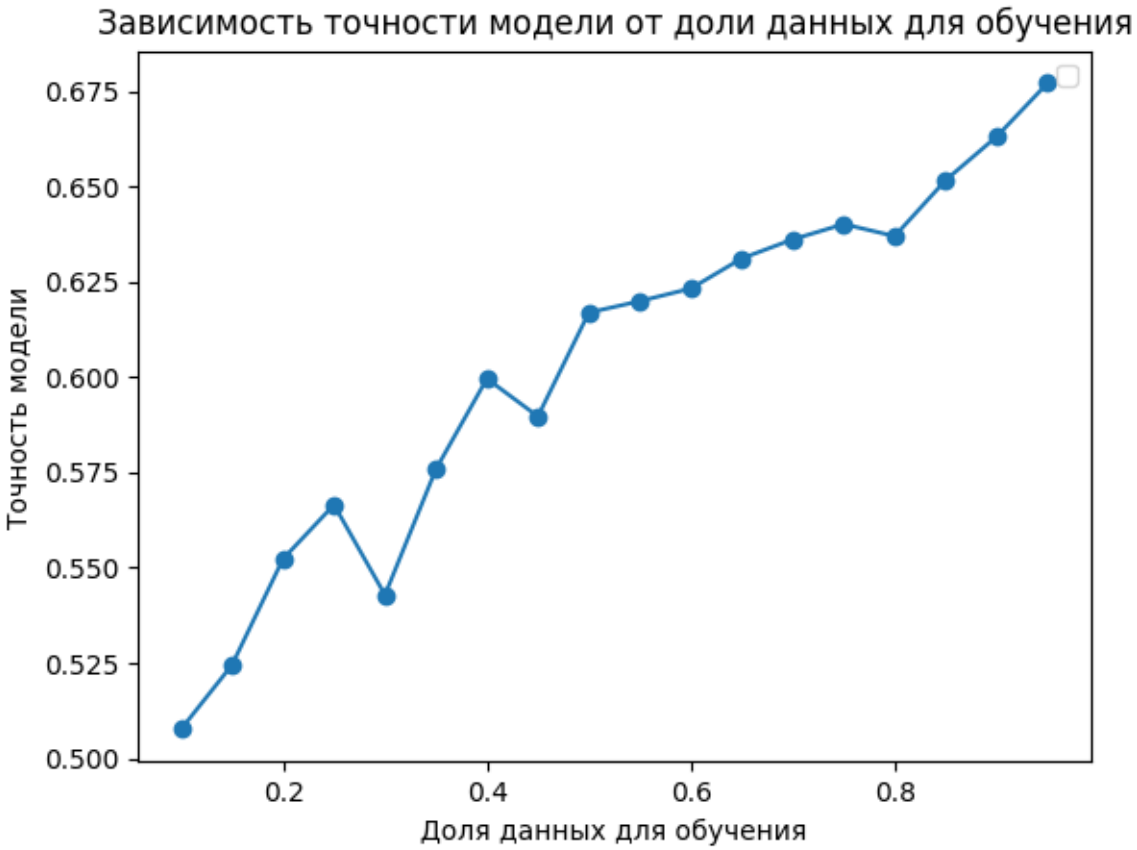

Рис. 4 – Зависимость точности классификатора от объема обучающей выборки

3 Генерация точек

3.1 Реализация на R

```
#Генерация точек для класса 1
random_X1 <- rnorm(50, mean = 10, sd = 4)
random_X2 <- rnorm(50, mean = 14, sd = 4)

data_class1 <- cbind(random_X1, random_X2)

data_class1 <- cbind(data_class1, -1)

#Генерация точек для класса 2
random_X1 <- rnorm(50, mean = 20, sd = 3)
random_X2 <- rnorm(50, mean = 18, sd = 3)

data_class2 <- cbind(random_X1, random_X2)

data_class2 <- cbind(data_class2, 1)

#Формирование обучающей выборки
train_data <- rbind(data_class1, data_class2)

train_data <- train_data[sample.int(nrow(train_data)), ]

library(e1071)

#Обучение модели
model <- naiveBayes(train_data[,-3], train_data[,3])

#Формирование тестирующей выборки
random_X1 <- rnorm(50, mean = 10, sd = 4)
random_X2 <- rnorm(50, mean = 14, sd = 4)

data_class1 <- cbind(random_X1, random_X2)

data_class1 <- cbind(data_class1, -1)

random_X1 <- rnorm(50, mean = 20, sd = 3)
random_X2 <- rnorm(50, mean = 18, sd = 3)

data_class2 <- cbind(random_X1, random_X2)

data_class2 <- cbind(data_class2, 1)

test_data <- rbind(data_class1, data_class2)

test_data <- test_data[sample.int(nrow(test_data)), ]

# Классификация
```

```
predicted <- predict(model, test_data[,-3])

result <- table(predicted, test_data[,3])

# Точность модели
accuracy <- sum(diag(result)) / sum(result)

accuracy

[1] 0.92
```

После нескольких запусков алгоритма была вычислена средняя точность классификатора - 0.94.

На рис. 5 отображен пример распределения сгенерированных точек для обучающей выборки.

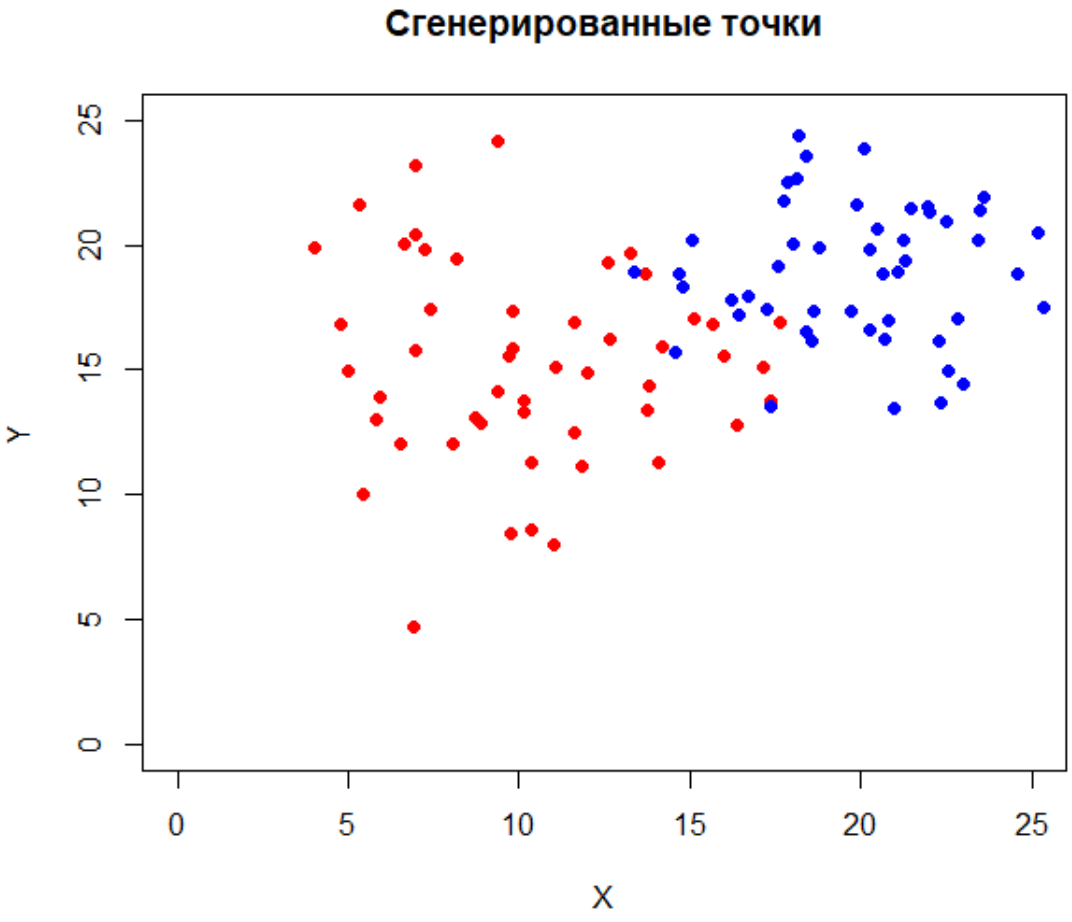

Рис. 5 – Распределение сгенерированных точек для обучающей выборки

На рис. 6 отображен пример распределения сгенерированных точек для тестовой выборки.

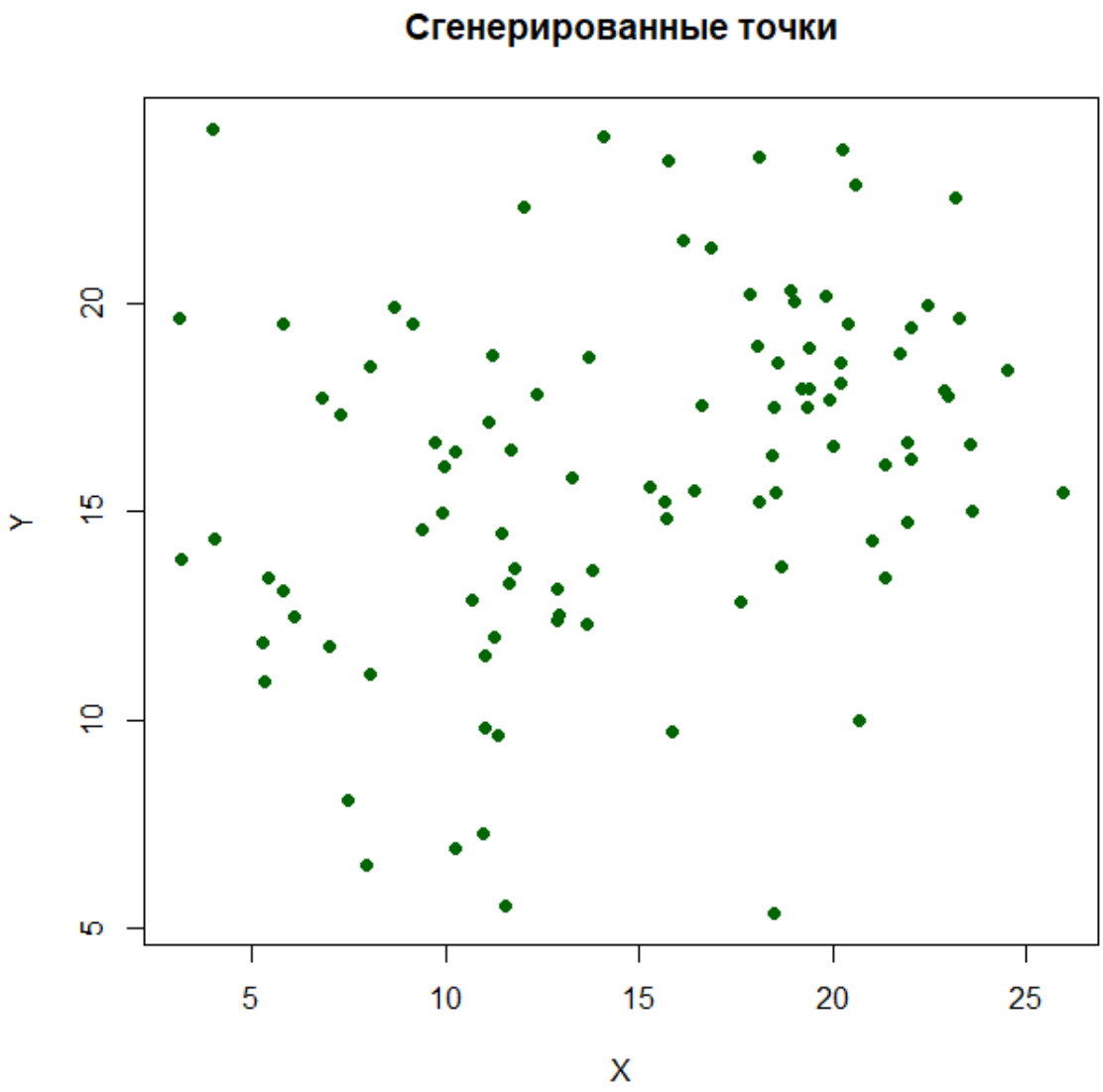

Рис. 6 – Распределение сгенерированных точек для тестовой выборки

3.2 Реализация на Python

```
points_train = []

# Генерация точек для класса 1
X1 = norm(loc=10, scale=4)
X2 = norm(loc=14, scale=4)

for i in range(100):
    point = [X1.rvs(), X2.rvs(), -1]
    points_train.append(point)

X1 = norm(loc=20, scale=3)
X2 = norm(loc=18, scale=3)

# Генерация точек для класса 2
```

```
for i in range(100):
    point = [X1.rvs(), X2.rvs(), 1]
    points_train.append(point)

# Определние модели модели
model = NaiveBayesClassifier(2, window_size=3)

# Алгоритм кросс-валидации
acc = cross_validation_accuracy(model, points_train, split_ratio=0.5, k=100)

print(acc)    # 0.937
```

После нескольких запусков алгоритма была вычислена средняя точность классификатора - 0.92. Снижение точности, по сравнению с реализацией классификатора на языке R, можно объяснить использованием алгоритма кросс-валидации для оценки точности классификации модели.

На рис. 7 отображен пример распределения сгенерированных точек для обучающей выборки.

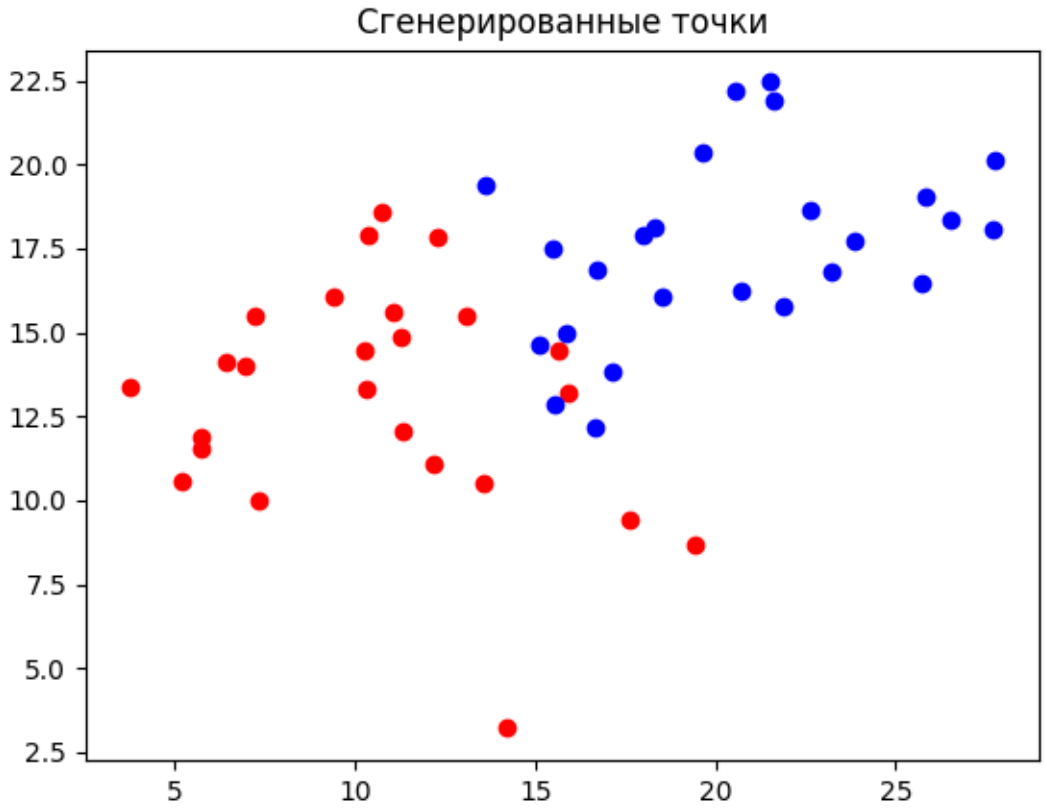

Рис. 7 – Распределение сгенерированных точек для обучающей выборки

На рис. 8 отображен пример распределения сгенерированных точек для тестовой выборки.

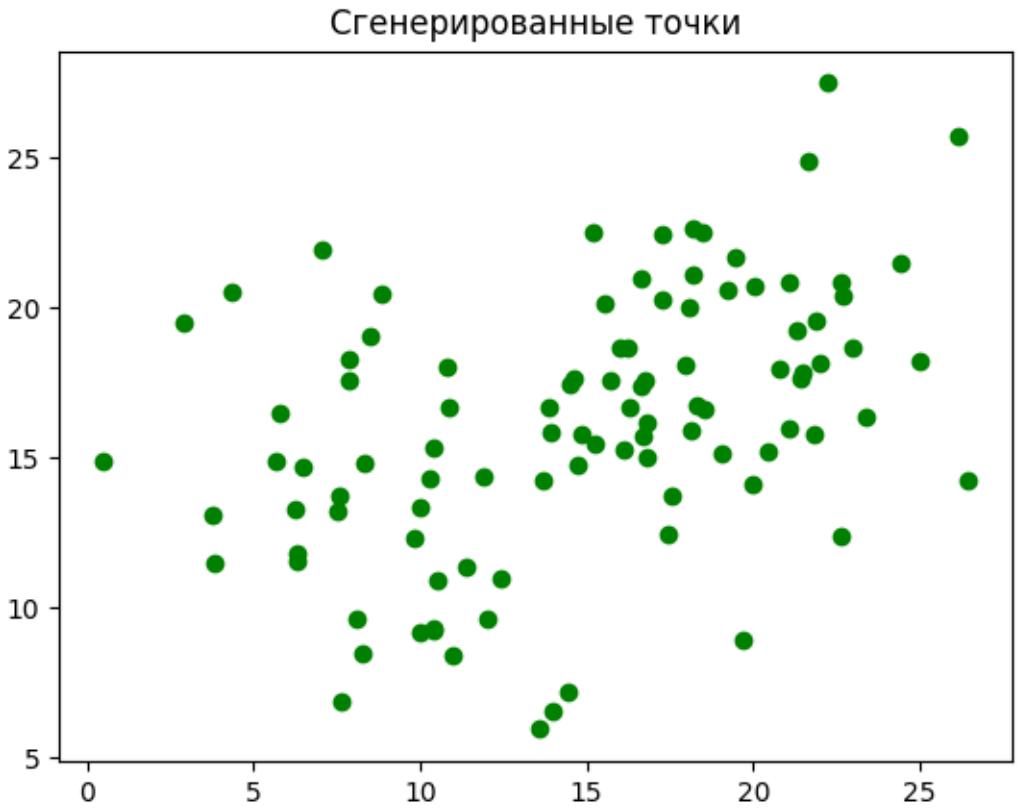

Рис. 8 – Распределение сгенерированных точек для тестовой выборки

4 Классификатор для обучающего множества «Титаник»

4.1 Реализация на R

```
# Импорт обучающей выборки
train <- read.csv("R_files\\lab1\\Titanic_train.csv", header = TRUE)

# Объединяем имя и фамилию в один столбец
train <- data.frame(do.call(rbind, strsplit(as.character(train$PassengerId), ",")))
train$Name <- paste(train$X4, train$X5, sep = " ")

# Удаляем столбцы X4 и X5
train <- train[, !(names(train) %in% c("X4", "X5"))]
colnames(train) <- c("PassengerId","Survived", "Pclass", "Sex", "Age",
"SibSp", "Parch", "Ticket", "Fare", "Cabin", "Embarked", "Name")

# Импорт тестирующей выборки
test <- read.csv("R_files\\lab1\\Titanic_test.csv", header = TRUE)
test <- data.frame(do.call(rbind, strsplit(as.character(test$PassengerId),
",")))
test$Name <- paste(test$X4, test$X5, sep = " ")
test <- test[, !(names(test) %in% c("X4", "X5"))]
colnames(test) <- c("PassengerId","Pclass", "Sex", "Age", "SibSp", "Parch",
"Ticket", "Fare", "Cabin", "Embarked", "Name")

library(e1071)

# Построение модели
model <- naiveBayes(train, train$Survived)

# Классификация
result <- predict(model, test)

data <- data.frame(PassengerId = 892:1309, Survived = result)

# Вывод данных в файл
write.csv(data, "R_files\\Titanic_result.csv", row.names = FALSE)
```

Полученные результаты были загружены на сайт `https://www.kaggle.com/c/titanic`, точность классификатора была оценена на 69.61%.

4.2 Реализация на Python

```
train_file_path = 'Titanic_train.csv'
test_file_path = 'Titanic_test.csv'
result_file_path = 'Titanic_result.csv'

# Импорт обучающей выборки с типизацией
titanic_train = CSVReader.read_csv(train_file_path, [int, int, int, str, str,
float, int, int, str, float, str, str], head=False)

# Определение модели
model = NaiveBayesClassifier(1)

# Обучение модели
model.train(titanic_train)

# Импорт тестирующей выборки выборки с типизацией
titanic_test = CSVReader.read_csv(test_file_path,
                                  [int, int, int, str, str, float, int, int,
                                  str, float, str, str], head=False)

# Формирование результата
results = [["PassengerId", "Survived"]]

for test in titanic_test:
    id = test[0]
    result = model.predicate_mult(test)
    results.append([id, result])

# Запись результата в файл
with open(result_file_path, 'w', newline='') as csvfile:
    csv_writer = csv.writer(csvfile)

    for row in results:
        csv_writer.writerow(row)
```

Полученные результаты были загружены на сайт `https://www.kaggle.com/c/titanic`, точность классификатора была оценена на 71.052%

Заключение

В данной лабораторной работе был реализован наивный байевоский классификатор на языке Python, также был реализован алгоритм Shuffle-Split Cross-Validation для измерения точности классификации модели.

Реализованный на Python классификатор был исследован на 4 наборах данных: Tic Tac Toe, MNIST, сгенерированное согласно нормальному распределению множество точек, Titanic. Показатели точности классификации соответственно:

- Tic Tac Toe - 66.17%
- MNIST - 68.05
- Сгенерированное согласно нормальному распределению множество точек - 92%
- Titanic - 71.052%

Также было проведено исследование классификатора, реализованного на языке R, на 4 наборах данных: Tic Tac Toe, spam, сгенерированное согласно нормальному распределению множество точек, Titanic. Показатели точности классификации соответственно:

- Tic Tac Toe - 73.9%
- spam - 74.8
- Сгенерированное согласно нормальному распределению множество точек - 94%
- Titanic - 69.61%

Различие в точности между двумя вариантами классификатора можно объяснить разными способами измерения точности классификации.

Было проведено исследование влияния размера обучающей выборки на точность классификации с использованием трех наборов данных: Tic Tac Toe, spam и MNIST. Для Tic Tac Toe и spam из-за ограниченного объема обучающих данных не удалось выявить общей тенденции в зависимости точности от размера выборки. В случае датасета MNIST была обнаружена линейная зависимость точности классификации от объема обучающей выборки.

ПРИЛОЖЕНИЕ 1 (Код класса NaiveBayesClassifier)

```
class NaiveBayesClassifier:

    def __init__(self, target_index, alpha=1, window_size=0.5):
        self.alpha = alpha
        self.window_size = window_size
        self.target_index = target_index

    def train(self, data):
        self.sample_size = len(data)
        self.feature_size = len(data[0])
        self.conditional_probabilities = [{} for _ in range(self.feature_size)]
        self.prior_probabilities = {}

        for i in range(self.feature_size):
            for j in range(self.sample_size):
                if i != self.target_index:

                    if isinstance(data[j][i], (int, float)):
                        data[j][i] =
                        self._approximate_value(self.conditional_probabilities
                        [i], data[j][i])

                    if data[j][i] not in self.conditional_probabilities[i]:
                        self.conditional_probabilities[i][data[j][i]] = {}

                    target_data = data[j][self.target_index]

                    if target_data in self.conditional_probabilities[i]
                    [data[j][i]]:
                        self.conditional_probabilities[i][data[j][i]]
                        [target_data] += 1
                    else:
                        self.conditional_probabilities[i][data[j][i]]
                        [target_data] = 1
                else:
                    target_data = data[j][self.target_index]

                    if target_data in self.prior_probabilities:
                        self.prior_probabilities[target_data] += 1
                    else:
                        self.prior_probabilities[target_data] = 1

        del self.conditional_probabilities[self.target_index]

        for feature in self.conditional_probabilities:
            for target_key in feature.keys():
                for target_value in feature[target_key].keys():
```

```
                feature[target_key][target_value] = \
                    (feature[target_key][target_value] + self.alpha) /
                    (self.sample_size + self.alpha)

    self.prior_probabilities = {key: value / len(data) for key, value in
    self.prior_probabilities.items()}

def _approximate_value(self, map, value):

    close_value = [key for key in map.keys() if value - self.window_size
    <= key <= value + self.window_size]
    if close_value:
        value = close_value[0]
    return value

def predicate_mult(self, raw):
    final_probabilities = dict(self.prior_probabilities)

    for target_value in self.prior_probabilities.keys():
        p = self.prior_probabilities[target_value]

        for i in range(len(raw)):
            try:
                if isinstance(raw[i], (int, float)):
                    raw[i] =
                    self._approximate_value(self.conditional_probabilities
                    [i], raw[i])
                p *= self.conditional_probabilities[i][raw[i]][target_value]
            except KeyError:
                p *= self.alpha / (self.sample_size + self.alpha)

        final_probabilities[target_value] = p

    return max(final_probabilities, key=final_probabilities.get)

def predicate_log(self, raw):
    final_probabilities = dict(self.prior_probabilities)

    for target_value in self.prior_probabilities.keys():
        p = log(self.prior_probabilities[target_value])

        for i in range(len(raw)):
            try:
                if isinstance(raw[i], (int, float)):
                    raw[i] =
                    self._approximate_value(self.conditional_probabilities
                    [i], raw[i])
                p += log(self.conditional_probabilities[i][raw[i]][target_value])
            except KeyError:
                p += log(self.alpha / (self.sample_size + self.alpha))
```

```
        final_probabilities[target_value] = p

    return max(final_probabilities, key=final_probabilities.get)
```