Министерство образования и науки Российской Федерации Федеральное государственное автономное образовательное учреждение высшего профессионального образования «Севастопольский государственный университет»

ОТЧЕТ

о выполнении лабораторной работы № 1 по дисциплине

«Методы и системы искусственного интеллекта»

Выполнил:

ст. гр. ИС/б-22-1-о

Гюнтер М. Ю .

Проверил:

доцент кафедры

“Информационные системы”

Бондарев В. Н.

Севастополь,2025

# Цель работы: Изучение технологии подготовки и выполнения программ на языке Python, исследование свойств функций языка Python, используемых при обработке последовательностей, формирование навыков определения классов языка Python.

# Ход работы

## Выполнение задания 1

Продемонстрируем работу метода str.split() (рисунок ‎2.1), возвращающего список слов на основании строки и разделителя с заданным ограничением, по умолчанию равного пробелу, с помощью кода, приведённого в листинге ‎2.1.

Листинг ‎2.1 – Демонстрация метода str.split()

nums\_str = '0 1 2 3 4 5 6 7 8 9'

nums\_int = nums\_str.split()

nums\_inc = [int(num) + 1 for num in nums\_int]

for num in nums\_inc:

print(num, end = " ")



Рисунок ‎2.1 – Результат обработки строки с использованием str.split()

Также продемонстрируем работу аналогичного метода str.join() (рисунок ‎2.2), объединяющего итерируемый объект на основании строки соединителя, для которой вызывается метод, с помощью кода, приведённого в листинге ‎2.2.

Листинг ‎2.2 – Демонстрация метода str.join()

nums\_str\_comma = '0, 1, 2, 3, 4, 5, 6, 7, 8, 9'

print(f'Оригинальная строка: {nums\_str\_comma}')

nums\_int = nums\_str\_comma.split(', ')

nums\_str\_space = " ".join(nums\_int)

print('Строка с изменённым разделителем: {}'.format(nums\_str\_space))



Рисунок ‎2.2 – Изменение разделителя строки с использованием str.join()

Продемонстрируем работу метода str.count() (рисунок ‎2.3), возвращающего количество неперекрывающихся повторений заданной подстроки в строке, с помощью кода, приведённого в листинге ‎2.3.

Листинг ‎2.3 – Демонстрация метода str.count()

haystack = 'Хороший код — это читаемый код.'

needle = 'код'

count = haystack.count(needle)

print(f'''Сколько раз слово "{needle}" содержится в предложении "{haystack}"?\n{count}''')



Рисунок ‎2.3 – Результат поиска подстроки с использованием str.count()

Продемонстрируем работу метода str.startswith() (рисунок ‎2.4), возвращающего True, если строка начинается с заданной подстроки, и False, если строка не начинается с заданной подстроки, с помощью кода, приведённого в листинге ‎2.4.

Листинг ‎2.4 – Демонстрация метода str.startswith()

fruits = [

"Apple",

"banana",

"apricot",

"Grape",

"avocado",

"orange",

"kiwi",

"Almond",

"Blueberry",

"artichoke"

]

count = 0

for i in fruits:

if i.lower().startswith('a'):

count += 1

print('''Количество фруктов, название которых начинается с буквы "a": \n%d''' % count)



Рисунок ‎2.4 – Результат поиска фруктов на букву “a” с использованием str.startswith()

Продемонстрируем работу метода str.endswith() (рисунок ‎2.5), возвращающего True, если строка заканчивается заданной подстрокой, и False, если строка не заканчивается заданной подстрокой, с помощью кода, приведённого в листинге ‎2.5.

Листинг ‎2.5 – Демонстрация метода str.endswith()

sentences = [

"Синтаксис Python очень читаемый.",

"Отладка кода требует терпения",

"Алгоритмы — это основа программирования.",

"Компилятор преобразует код в машинный язык",

"Тестирование необходимо для качества продукта.",

"Версионирование кода с помощью Git обязательно",

"Функции помогают избежать повторения кода.",

"Базы данных хранят информацию структурированно",

"Рекурсия может быть сложной для понимания.",

"Веб-разработка включает фронтенд и бэкенд"

]

count = 0

for i in sentences:

if i.endswith('.'):

count += 1

print('Число предложений, где пропущена точка: \n%d' % count)



Рисунок ‎2.5 – Результат поиска предложений, где пропущена точка, с использованием str.endswith()

Продемонстрируем работу оставшихся методов обработки строк str.format(), str.split(), str.lsplit(), str.rsplit(), str.capitalize(), str.title(), str.index(), str.rindex(), str.replace(), str.rsplit(), str.partition() и str.rpartition() (рисунок ‎2.6), с помощью кода, приведённого в листинге ‎2.6.

Листинг ‎2.6 – Демонстрация оставшихся методов обработки строк

# format method

print('Это пример {} текста с помощью метода {}.'.format('форматирования', 'format()'))

# strip method

print(' Это пример удаления пробелов в начале и конце строки '.strip())

# lstrip method

print(' Это пример удаления пробелов в начале строки'.lstrip())

# rstrip method

print('''Это пример удаления символов "\_abc" в конце строки\_abc\_abc\_abc\_abc'''.rstrip('\_abc'))

# capitalize method

print('эТО ПРИМЕР ИЗМЕНЕНИЯ СТРОКИ, ПРИ КОТОРОМ ТОЛЬКО ЕЁ ПЕРВЫЙ СИМВОЛ ПРОПИСНОЙ'.capitalize())

# title method

print('ЭТО ПРИМЕР ИЗМЕНЕНИЯ СТРОКИ, ПРИ КОТОРОМ ТОЛЬКО ПЕРВЫЙ СИМВОЛ КАЖДОГО ЕЁ СЛОВА ПРОПИСНОЙ'.title())

# index method

print("Индекс подстроки \"ab\" в строке \"bc ab ab ab ab ab ab bc\" с начала строки: %d" % "bc ab ab ab ab ab ab bc".index("ab"))

# rindex method

print("Индекс подстроки \"ab\" в строке \"bc ab ab ab ab ab ab bc\" с конца строки: %d" % "bc ab ab ab ab ab ab bc".rindex("ab"))

# replace method

print('Это dummy замены подстроки в строке'.replace('dummy', 'пример'))

# rsplit method

print('''Это пример разделения строки "1, 2, 3, 4, 5, 6" с конца с ограничением 4:''')

print('1, 2, 3, 4, 5, 6'.rsplit(", ", 4))

# partition method

print('''Это пример раделения строки "abc|def|ghi на 3 части слева с разделителем "|":"''')

print('abc|def|ghi'.partition('|'))

# rpartition method

print('''Это пример разделения строки "abc|def|ghi" на 3 части справа с разделителем "|":''')

print('abc|def|ghi'.rpartition('|'))

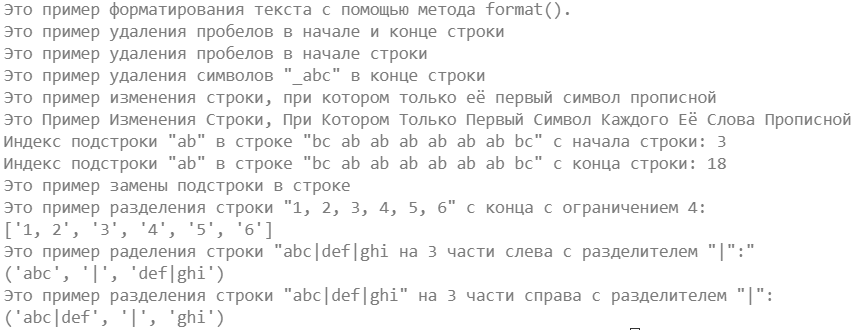


Рисунок ‎2.6 – Результат работы оставшихся методов обработки строк

## Выполнение задания 2

Продемонстрируем работу методов обработки списков list.append(), list.extend(), list.insert(), list.pop(), list.remove(), list.reverse(), list.sort(), list.count(), list.index() (рисунок ‎2.7) с помощью кода, приведенного в листинге ‎2.7.

Листинг ‎2.7 – Демонстрация методов обработки списков

# append method

myList = [1, 2, 3]

print('Исходный список: ', myList)

myList.append(4)

print('Список с единственным добавленным элементом: ', myList, end="\n\n")

# extend method

myList = [1, 2, 3]

print('Исходный список: ', myList)

myList.extend([4, 5, 6])

print('Список с несколькими добавленными элементами: ', myList, end="\n\n")

# insert method

myList = [1, 2, 3, 5, 6, 7]

myNum = 4

print('Исходный список: ', myList)

myList.insert(3, myNum)

print('Список со вставленным элементом %d' % myNum, myList, end="\n\n")

# pop method

myList = [1, 2, 3, 4, 5]

print('Исходный список: ', myList)

myNum = myList.pop()

print('Список без последнего элемента %d: ' % myNum, myList, end="\n\n")

# remove method

myList = [1, 2, 3, 4, 5]

print('Исходный список: ', myList)

myList.remove(2)

print('Список без элемента с индексом 2: ', myList, end="\n\n")

# reverse method

myList = [1, 2, 3, 4, 5]

print('Перевернутый список %s: %s' % (myList, myList.reverse()), end="\n\n")

# sort method

myList = [3, 2, 5, 6, 9, 8, 7, 0, 4, 1]

print('Список до сортировки: ', myList)

myList.sort()

print('Список после прямой сортировки: ', myList)

myList.sort(reverse = True)

print('Список после обратной сортировки: ', myList, end="\n\n")

# count method

haystack = [1, 2, 3, 1, 2, 3, 1, 1, 1, 1, 2]

needle = 1

print('Число повторений элемента %d в списке %s: %d' % (needle, str(haystack), haystack.count(needle)), end="\n\n")

# index method

haystack = [1, 2, 3, 5, 1, 5, 2, 7, 8, 9, 0]

needle = 5

print(f'Индекс первого слева элемента со значением {needle} в списке {str(haystack)}: {haystack.index(needle)}', end="\n\n")

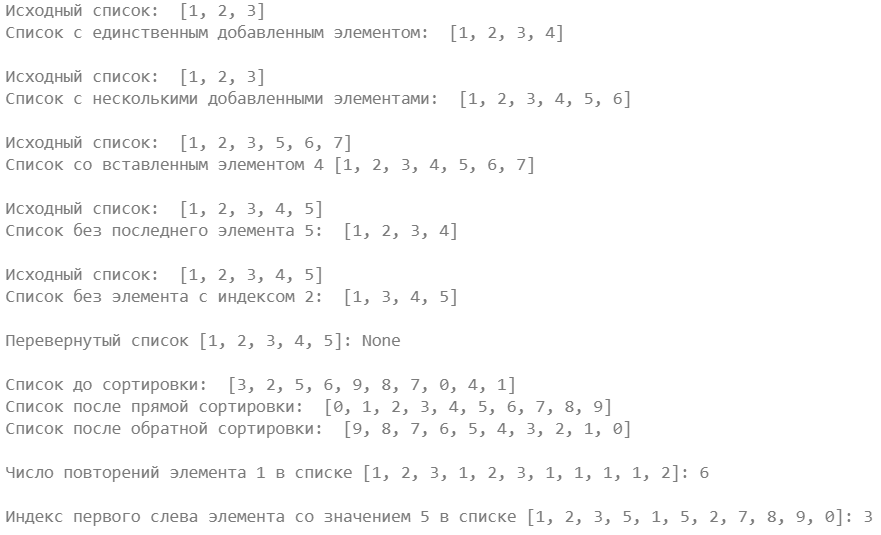


Рисунок ‎2.7 – Результаты работы методов обработки списков

## Выполнение задания 3

Продемонстрируем работу методов обработки словарей dict.clear(), dict.copy(), dict.fromitems(), dict.get(), dict.keys(), dict.values(), dict.items(), dict.pop(), dict.popitem(), dict.setdeafult(), dict.update() (рисунок ‎2.8) с помощью кода, приведенного в листинге ‎2.8.

Листинг ‎2.8 – Демонстрация методов обработки словарей

# clear method

fruit\_prices = { 'apple': 50, 'banana': 30, 'orange': 40, 'grape': 80, 'lemon': 25, 'pear': 45, 'kiwi': 60, 'mango': 70, 'pineapple': 90, 'watermelon': 120 }

print('Оригинальный словарь: ', fruit\_prices)

fruit\_prices.clear()

print('Очищенный словарь: ', fruit\_prices)

print()

# copy method

fruit\_prices\_1 = { 'apple': 50, 'banana': 30, 'orange': 40, 'grape': 80, 'lemon': 25, 'pear': 45, 'kiwi': 60, 'mango': 70, 'pineapple': 90, 'watermelon': 120 }

fruit\_prices\_2 = fruit\_prices\_1.copy()

key = 'apple'

fruit\_prices\_1[key] = 100

print('Цена фрукта %s в словаре 1: %d' % (key, fruit\_prices\_1[key]))

print('Цена фрукта %s в словаре 2: %d' % (key, fruit\_prices\_2[key]))

print()

# fromkeys method

fruits = ['apple', 'banana', 'orange', 'grape', 'lemon', 'pear', 'kiwi', 'mango', 'pineapple', 'watermelon']

fruit\_prices = dict.fromkeys(fruits, 100)

print('Цены фруктов: ', fruit\_prices)

print()

# get method

fruit\_prices = { 'apple': 50, 'banana': 30, 'orange': 40, 'grape': 80, 'lemon': 25, 'pear': 45, 'kiwi': 60, 'mango': 70, 'pineapple': 90, 'watermelon': 120 }

# print(fruit\_prices['grapefruit']) # Приводит к исключению KeyError

print(fruit\_prices.get('grapefruit', 'Нет в продаже'))

print()

# keys method

fruit\_prices = { 'apple': 50, 'banana': 30, 'orange': 40, 'grape': 80, 'lemon': 25, 'pear': 45, 'kiwi': 60, 'mango': 70, 'pineapple': 90, 'watermelon': 120 }

product = 'Фрукты: '

for key in fruit\_prices.keys():

product += f'{key}, '

product = product.rstrip(', ')

print(product)

print()

# values method

fruit\_prices = { 'apple': 50, 'banana': 30, 'orange': 40, 'grape': 80, 'lemon': 25, 'pear': 45, 'kiwi': 60, 'mango': 70, 'pineapple': 90, 'watermelon': 120 }

product = 'Цены фруктов: '

for value in fruit\_prices.values():

product += f'{value}, '

product = product.rstrip(', ')

print(product)

print()

# items method

fruit\_prices = { 'apple': 50, 'banana': 30, 'orange': 40, 'grape': 80, 'lemon': 25, 'pear': 45, 'kiwi': 60, 'mango': 70, 'pineapple': 90, 'watermelon': 120 }

product = 'Фрукты и их цены: '

for key, value in fruit\_prices.items():

product += f'{key}: {value}, '

product = product.rstrip(', ')

print(product)

print()

# pop method

fruit\_prices = { 'apple': 50, 'banana': 30, 'orange': 40 }

print('Словарь до удаления ключа: ', fruit\_prices)

fruit\_prices.pop('apple')

print('Словарь после удаления пары ключ-значение по ключу: ', fruit\_prices)

print()

# popitem method

fruit\_prices = { 'apple': 50, 'banana': 50, 'orange': 40 }

fruit\_prices['grape'] = 80

fruit\_prices['lemon'] = 25

print('Словарь до удаления последней добавленной пары ключ-значение: ', fruit\_prices)

fruit\_prices.popitem()

print('Словарь после удаления последней пары ключ-значение: ', fruit\_prices)

print()

# setdefault method

fruit\_prices = { 'apple': 50, 'banana': 50, 'orange': 40 }

print('Словарь до вызова setdefault() для несуществующего ключа: ', fruit\_prices)

print('Значение по умолчанию для нового ключа: ', fruit\_prices.setdefault('pear', 60))

print('Словарь после вызова setdefault(): ', fruit\_prices)

print()

# update method

fruit\_prices = { 'apple': 50, 'banana': 50, 'orange': 40 }

fruit\_prices\_1 = { 'grape': 80, 'lemon': 25, 'pear': 45, 'kiwi': 60, 'mango': 70 }

print('Оригинальный словарь до обновления: ', fruit\_prices)

fruit\_prices.update(fruit\_prices\_2)

print('Оригинальный словарь после обновления: ', fruit\_prices)

Изображение выглядит как текст, письмо, Шрифт, документ

Содержимое, созданное искусственным интеллектом, может быть неверным.

Рисунок ‎2.8 – Результаты работы методов обработки словарей

## Выполнение задания 4

Был написан код, позволяющий с помощью списковых включений получить из существующего списка новый список, содержащий только те строки исходного, длина которых превышает 5 символов, и которые записаны в нижнем регистре (листинг ‎2.9). Результат работы кода приведён на рисунке ‎2.9.

Листинг ‎2.9 – Код, демонстрирующий работу списковых включений

myList = [ "Hello", "world", "PYTHON", "programming", "TEST", "string", "AaBbCc", "lowercase", "UPPERCASE", "mixedCASE", "short", "verylongword", "123456", "abc", "ABCDEFGH" ]

resList = [x for x in myList if len(x) > 5 and x == x.lower()]

print(resList)

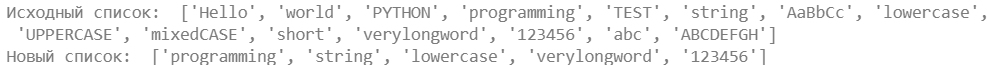


Рисунок ‎2.9 – Результаты работы кода со списковыми включениями

Для проверки созданной модели построим матрицу неточностей (рисунок ‎2.10) с помощью команд R, приведённых в листинге ‎2.10.

Листинг ‎2.10 – Команды для построения матрицы неточностей

table (dataset.ldap, Dataset.unknow[,7])

Изображение выглядит как текст, Шрифт, снимок экрана, рукописный текст

Контент, сгенерированный ИИ, может содержать ошибки.

Рисунок ‎2.10 – Результаты построения матрицы неточностей

По данной матрице видно, что тренировочная выборка привела к построению гипотезы, по которой ни один объект не попал не свой класс, следовательно, ошибка в данной модели составляет 0%.

## Проведение шаговой процедуры выбора переменных для построения дискриминантной модели.

Для найденного набора данных была проведена шаговая процедура выбора переменных для построения дискриминантной модели (листинг ‎2.11). В результате выполнения процедуры (рисунок ‎2.11).

Листинг ‎2.11 – Команды для проведения пошаговой процедуры выбора переменных

stepclass(Dataset[,1:6], Dataset[,7], method="lda")

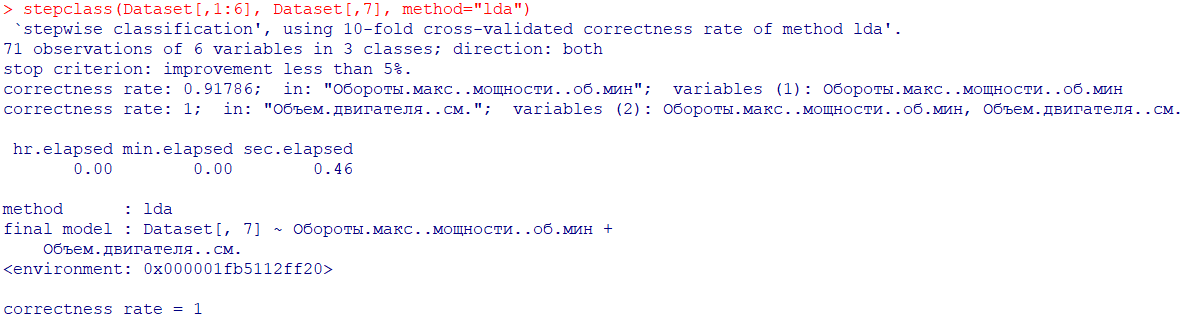


Рисунок ‎2.11 – Результаты проведения пошаговой процедуры выбора переменных

Исходя из результатов выбора переменных для построения дискриминантной модели будут использоваться только две переменные «Обороты максимальной мощности» и «Объем двигателя».

## Построение дискриминантной модели на основании выбранных переменных

Был проведён дискриминантный анализ тренировочных данных с учётом только выбранных на предыдущем этапе независимых переменных (рисунок ‎2.12) с помощью команд R, приведённых в листинге ‎2.12.

Листинг ‎2.12 – Команды для проведения дискриминантного анализа

dataset.lda <- lda (Dataset.train[, 2:3], Dataset.train[, 7])

dataset.lda

Изображение выглядит как текст, снимок экрана, Шрифт, число

Контент, сгенерированный ИИ, может содержать ошибки.

Рисунок ‎2.12 – Результаты проведения дискриминантного анализа тренировочных данных

В соответствии с результатами дискриминантного анализа были получены коэффициенты линейных дискриминантов (рисунок ‎2.13).

Изображение выглядит как текст, снимок экрана, Шрифт, линия

Контент, сгенерированный ИИ, может содержать ошибки.

Рисунок ‎2.13 – Коэффициенты линейных дискриминантов

На основании этих коэффициентов построим дискриминантные функции:

z(x) = –0.0025x1 – 0.0019x2

z(x) = 0.0005x1 – 0.0024x2

Была произведена классификация и проверка оставшихся данных с учётом выбранных независимых переменных (рисунок ‎2.14) с помощью команд R, приведённых в листинге ‎2.13.

Листинг ‎2.13 – Команды для классификации и проверки оставшихся данных

dataset.ldap <- predict(dataset.lda, Dataset.unknow [,2:3])$class

dataset.ldap

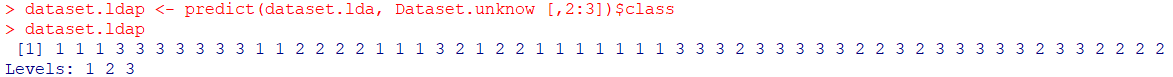


Рисунок ‎2.14 – Результаты классификации и проверки оставшихся данных с учётом выбранных столбцов

Для проверки созданной модели построим матрицу неточностей (рисунок ‎2.15) с помощью команд R, приведённых в листинге ‎2.14.

Листинг ‎2.14 – Команды для построения матрицы неточностей

table (dataset.ldap, Dataset.unknow[,7])

Изображение выглядит как текст, Шрифт, рукописный текст, снимок экрана

Контент, сгенерированный ИИ, может содержать ошибки.

Рисунок ‎2.15 – Результаты построения матрицы неточностей

Также для построенной матрицы посчитаем ошибку распознавания (рисунок ‎2.16) с помощью команд R, приведённых в листинге ‎2.15.

Листинг ‎2.15 – Команды для расчёта ошибки распознавания

Err\_S <- mean (Dataset.unknow[,7] != dataset.ldap)

Err\_S

Изображение выглядит как текст, Шрифт, снимок экрана

Контент, сгенерированный ИИ, может содержать ошибки.

Рисунок ‎2.16 – Ошибка распознавания

По результатам подсчёта ошибки распознавания, и в целом по полученной матрице, видно, что тренировочная выборка привела к построению гипотезы, по которой один объект попал не свой класс, следовательно, ошибка в данной модели составляет 0.017857%. Получаем, что ошибка распознавания исходной модели без удаления переменной меньше ошибки распознавания модели с удалёнными переменными.

Теперь рассчитаем расстояние Махалонобиса (рисунок ‎2.17) с помощью команд R, приведённых в листинге ‎2.16.

Листинг ‎2.16 – Команды для расчёта расстояния Махалонобиса

mahDist <- dist(dataset.lda$means %\*% dataset.lda$scaling)

mahDist

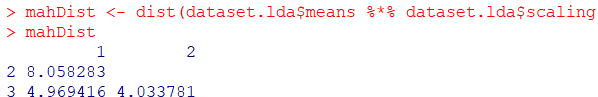


Рисунок ‎2.17 – Рассчитанное расстояние Махалонобиса

## Построение дискриминантной модели на основании выборки с записями без классификации

Добавим в выборку 4 записи без классификации, то есть со значением столбца кластера 0 (рисунок ‎2.18)

Изображение выглядит как текст, снимок экрана, Параллельный, число

Контент, сгенерированный ИИ, может содержать ошибки.

Рисунок ‎2.18 – Выборка с 4 добавленными записями без классификации

Создадим тренировочную выборку строк от первой до последней проклассифицированной (то есть не включая последние 4 строки) с шагом 5 (рисунок ‎2.19) с помощью команд R, приведённых в листинге ‎2.17.

Листинг ‎2.17 – Команды для создания и выведения на экран тренировочной выборки без последних 4-х строк

Dataset.train <- Dataset [seq (1, nrow(Dataset) - 4, 5),]

Dataset.train

Изображение выглядит как текст, снимок экрана, линия, число

Контент, сгенерированный ИИ, может содержать ошибки.

Рисунок ‎2.19 – Тренировочная выборка для дискриминантного анализа без не проклассифицированных строк

Аналогично создадим проверочную выборку из оставшихся строк вместе с не проклассифицированными (рисунок ‎2.20) с помощью команд R, приведённых в листинге ‎2.18.

Листинг ‎2.18 – Команды для создания и выведения на экран проверочной выборки вместе с не проклассифицированными строками

Dataset.unknow <- Dataset [-seq (1, nrow(Dataset) - 4, 5),]

Dataset.unknow

Изображение выглядит как текст, снимок экрана

Контент, сгенерированный ИИ, может содержать ошибки.

Рисунок ‎2.20 – Конец проверочной выборки для дискриминантного анализа, в которую добавлены не проклассифицированные строки

Был проведён дискриминантный анализ тренировочных данных (рисунок ‎2.21) с помощью команд R, приведённых в листинге ‎2.19.

Листинг ‎2.19 – Команды для проведения дискриминантного анализа

dataset.lda <- lda (Dataset.train[, 1:6], Dataset.train[, 7])

dataset.lda

Изображение выглядит как текст, снимок экрана, Шрифт

Контент, сгенерированный ИИ, может содержать ошибки.

Рисунок ‎2.21 – Результаты проведения дискриминантного анализа тренировочных данных

В соответствии с результатами дискриминантного анализа были получены коэффициенты линейных дискриминантов (рисунок ‎2.22).

Изображение выглядит как текст, снимок экрана, Шрифт, число

Контент, сгенерированный ИИ, может содержать ошибки.

Рисунок ‎2.22 – Коэффициенты линейных дискриминантов

На основании этих коэффициентов построим дискриминантные функции:

z(x) = 0.0185x1 – 0.0031x2 – 0.0031x3 + 0.5x4 + 0.003x5 + 0.5x6

z(x) = – 0.0134x1 + 0.0003x2 – 0.0014x3 + 0.21x4 – 0.0068x5 + 0.208x6

Отметим, что в результате проведения этого дискриминантного анализа, по сравнению с предыдущим случаем, поменялись на противоположные знаки коэффициентов для второй дискриминантной функции.

Была произведена классификация и проверка оставшихся данных с не проклассифицированными строками (рисунок ‎2.23) с помощью команд R, приведённых в листинге ‎2.20.

Листинг ‎2.20 – Команды для классификации и проверки оставшихся данных с не проклассифицированными строками

dataset.ldap <- predict(dataset.lda, Dataset.unknow [,1:6])$class

dataset.ldap

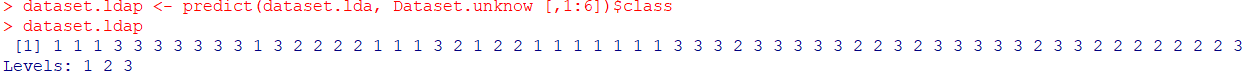


Рисунок ‎2.23 – Результаты классификации и проверки оставшихся данных с не проклассифицированными строками

В результате классификации первые три строки были отнесены ко 2-му кластеру, а последняя к 3-му.

Для проверки созданной модели построим матрицу неточностей (рисунок ‎2.24) с помощью команд R, приведённых в листинге ‎2.21.

Листинг ‎2.21 – Команды для построения матрицы неточностей

table (dataset.ldap, Dataset.unknow[,7])

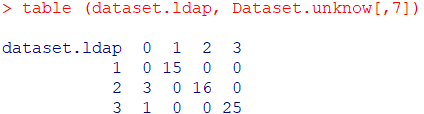


Рисунок ‎2.24 – Результаты построения матрицы неточностей

По построенной матрице убеждаемся в том, что 3 их добавленных строк попали во 2-й кластер, а одна – в 3-й.

# Вывод:

В результате выполнения лабораторной работы был проведён дискриминантный анализ найденных данных при количестве кластеров 3. В результате проведения дискриминантного анализа на тренировочных данных были получены коэффициенты линейных дискриминантов, на основании которых были построены дискриминантные функции. Полученные в ходе дискриминантного анализа данные были проверены на оставшихся данных, и при построении матрицы неточностей выяснилось, что ошибка в построенной модели составляет 0%. Далее была проведена пошаговая процедура выбора переменных для построения дискриминантной модели, в ходе которой было выделено 2 переменные «Обороты максимальной мощности» и «Объем двигателя». При проведении повторного дискриминантного анализа на основе только этих 2-х переменных была получена модель с ошибкой классификации 0.017857%. Также, дополнительно в исходную выборку было добавлено 4 строки с произвольными данными, схожими с таковыми в строках оригинальной выборки. По результатам проведения дискриминантного анализа этой выборки 3 из добавленных строк попали во 2-й кластер, а одна – в 3-й.