Классификация — один из разделов машинного обучения, посвящённый решению следующей задачи.

Имеется множество объектов (или ситуаций), разделяемых на определённые классы.
Задано конечное множество объектов, для которых известно, к каким классам они относятся.
Это множество называется **обучающей выборкой**.

Классовая принадлежность остальных объектов неизвестна.
Требуется построить алгоритм, способный **классифицировать** произвольный объект из исходного множества.

Определения:

- **Классифицировать объект** — значит указать номер (или наименование) класса, к которому относится данный объект.
- **Классификация объекта** — это номер или наименование класса, выдаваемый алгоритмом классификации в результате его применения к конкретному объекту.

1.1. Теоретический материал – Функции Python

Функция (или метод) в Python — это объект, который принимает аргументы и возвращает значение.
Она определяется с помощью ключевого слова `def`.

Определение простой функции

```
def add(x, y):
    return x + y
```

Инструкция return говорит, что нужно вернуть значение. В нашем случае функция возвращает сумму x и y. Теперь мы ее можем вызвать:

```
add(1, 10)         # Вывод: 11
add('abc', 'def') # Вывод: 'abcdef'
```

Функция может быть любой сложности и возвращать любые объекты (списки, кортежи, и даже функции):

```
def newfunc(n):
    def myfunc(x):
```

```
        return x + n
    return myfunc

new = newfunc(100)  # new — это теперь функция!
print(new(200))     # Вывод: 300
```

Функция может и не заканчиваться инструкцией return, при этом функция вернет значение None:

```
def func():
 pass
print(func()) # None
```

Функция может принимать произвольное количество аргументов или не принимать их вовсе. Также распространены функции с произвольным числом аргументов, функции с позиционными и именованными аргументами, обязательными и необязательными.

```
def func(*args):
 return args
func(1, 2, 3, 'abc')
# (1, 2, 3, 'abc')
func()
# ()
func(1)
# (1,)
```

Как видно из примера, args - это кортеж из всех переданных аргументов функции, и с переменной можно работать также, как и с кортежем. Функция может принимать и произвольное число именованных аргументов, тогда перед именем ставится **:

```
def func(**kwargs):
 return kwargs
func(a=1, b=2, c=3)
# {'a': 1, 'c': 3, 'b': 2}
func()
# {}
func(a='python')
# {'a': 'python'}
```

1.2.1. Пример

Задача:

Напишите функцию `sum_range(start, end)`, которая суммирует все целые числа от значения `start` до величины `end` **включительно**.

> Если пользователь задаст первое число больше второго, просто поменяйте их местами.

Решение:

```python
def sum_range(start, end):
    # Проверка и обмен аргументов, если start > end
    if start > end:
        end, start = start, end  # меняем местами

    # Суммирование чисел в диапазоне [start, end]
    return sum(range(start, end + 1))

# Тесты
print(sum_range(2, 12))   # 77
print(sum_range(-4, 4))   # 0
print(sum_range(3, 2))    # 5
```

Ответ:

```
77
0
5
```

1.2.2. Пример

Задача:

Напишите рекурсивную функцию вычисления факториала на языке Python.

> 🔹 Факториал числа $ n $ — это произведение всех натуральных чисел от 1 до $ n $:
> $$ n! = n \times (n-1) \times (n-2) \times \ldots \times 1 $$
> По определению: $ 0! = 1 $

Решение:

```python
def fact(num):
    if num == 0:
        return 1
    else:
        return num * fact(num - 1)
```

```
print(fact(5))
```

Ответ:

```
120
```

1.1. Теоретический материал – Расстояние между объектами

Определение расстояния между объектами класса

Сходство или различие между объектами классификации устанавливается в зависимости от выбранного метрического расстояния между ними.

Если каждый объект описывается $ p $ свойствами (признаками), то он может быть представлен как точка в $ p $-мерном пространстве.
Сходство с другими объектами будет определяться как соответствующее расстояние.

При классификации используются различные меры расстояния между объектами.

---

1. Евклидово расстояние

Это, пожалуй, наиболее часто используемая мера расстояния.
Она является геометрическим расстоянием в многомерном пространстве и вычисляется по формуле:

$$ r = \sqrt{\sum_{i=1}^{p} (A_i - B_i)^2} $$

где:

- $ r $ — расстояние между объектами A и B,
- $ A_i $ — значение i-го свойства объекта A,
- $ B_i $ — значение i-го свойства объекта B.

---

2. Квадрат евклидова расстояния

Данная мера используется в тех случаях, когда требуется придать большее значение более отдалённым друг от друга объектам.
Она вычисляется без извлечения корня:

$$ r = \sum_{i=1}^{p} (A_i - B_i)^2 $$

---

3. Взвешенное евклидово расстояние

Применяется, когда каждому i-му признаку можно присвоить "вес" $ w_i $, пропорциональный степени важности признака в задаче классификации:

$$ r = \sqrt{\sum_{i=1}^{p} w_i (A_i - B_i)^2} $$

---

4. Хеммингово расстояние

Также называется **манхэттенским**, **сити-блок расстоянием** или **расстоянием по координатам**.
Это расстояние является суммой абсолютных разностей по каждой координате:

$$ r = \sum_{i=1}^{p} |A_i - B_i| $$

---

5. Расстояние Чебышева

Принимает значение наибольшего модуля разности между значениями соответствующих свойств (признаков) объектов:

$$ r = \max_{i} |A_i - B_i| $$

Выбор меры расстояния и весов для классифицирующих свойств – очень важный этап, так как от этих процедур зависят состав и количество формируемых классов, а также степень сходства объектов внутри классов.

1.2.3. Пример

Задача:

Напишите функцию на Python, которая вычисляет **евклидово расстояние** между двумя массивами NumPy.

Решение:

```python
import numpy as np

def euclidean_distance(v1, v2):
    # Вычисление суммы квадратов разностей
    return sum((x - y) ** 2 for x, y in zip(v1, v2)) ** 0.5

x = np.array([0, 0, 0])
```

```
y = np.array([3, 3, 3])

print(euclidean_distance(x, y))
```

Ответ:

```
5.196152422706632
```

1.2.4. Пример

Задача:

Напишите 4 функции на Python, которые рассчитывают:

1. **Квадрат евклидова расстояния**
2. **Взвешенное евклидово расстояние**
3. **Хеммингово (манхэттенское) расстояние**
4. **Расстояние Чебышева**

между двумя массивами NumPy.

Решение:

```
import numpy as np

def sqr_euclidean_distance(v1, v2):
    return sum((x - y) ** 2 for x, y in zip(v1, v2))

def weighted_euclidean_distance(v1, v2, w):
    return sum((x - y) ** 2 * s for x, y, s in zip(v1, v2, w)) ** 0.5

def manhattan_distance(v1, v2):
    return sum(abs(x - y) for x, y in zip(v1, v2))

def chebyshev_distance(v1, v2):
    return max(abs(x - y) for x, y in zip(v1, v2))

# Тесты
x = np.array([0, 0, 0])
y = np.array([3, 3, 3])
w = np.array([0, 0, 1])

print(sqr_euclidean_distance(x, y))
print(weighted_euclidean_distance(x, y, w))
print(manhattan_distance(x, y))
print(chebyshev_distance(x, y))
```

Ответ:

```
27
3.0
9
3
```

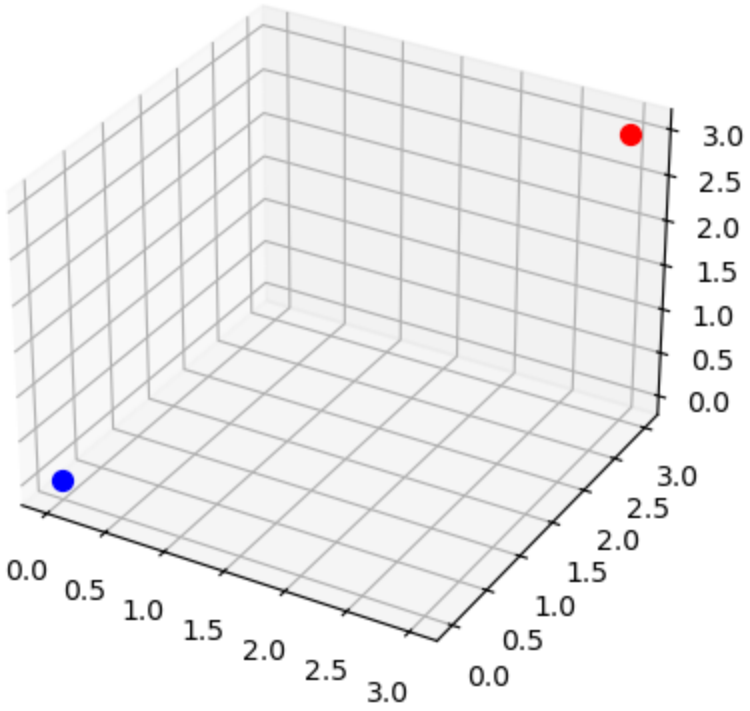
1.2.5. Пример

Задача:

В Python есть встроенные функции для вычисления расстояний между векторами.
Мы будем использовать NumPy для расчёта расстояния между двумя точками в 3D-пространстве.

Решение:

In [3]:
```python
import matplotlib.pyplot as plt
from mpl_toolkits.mplot3d import Axes3D

fig = plt.figure()
ax = fig.add_subplot(111, projection='3d')

ax.scatter(0, 0, 0, c='blue', s=50, label='Точка A')
ax.scatter(3, 3, 3, c='red', s=50, label='Точка B')
plt.show()
```



1.2.6. Пример

Задача:

Рассчитать расстояния между двумя точками с использованием методов из библиотеки NumPy, определённых выше.

Решение:

```
import numpy as np

p1 = np.array([0, 0, 0])
p2 = np.array([3, 3, 3])
print(np.linalg.norm(p1 - p2))
print(np.linalg.norm(p1 - p2) ** 2)
print(np.linalg.norm(p1 - p2, ord=np.inf))
print(np.linalg.norm(p1 - p2, ord=1))
```

Ответ:

```
5.196152422706632
27.0
3.0
9.0
```

1.3.1. Задание

Задача:

Задайте 4 точки в трёхмерном пространстве, рассчитайте расстояния между ними по следующим метрикам:

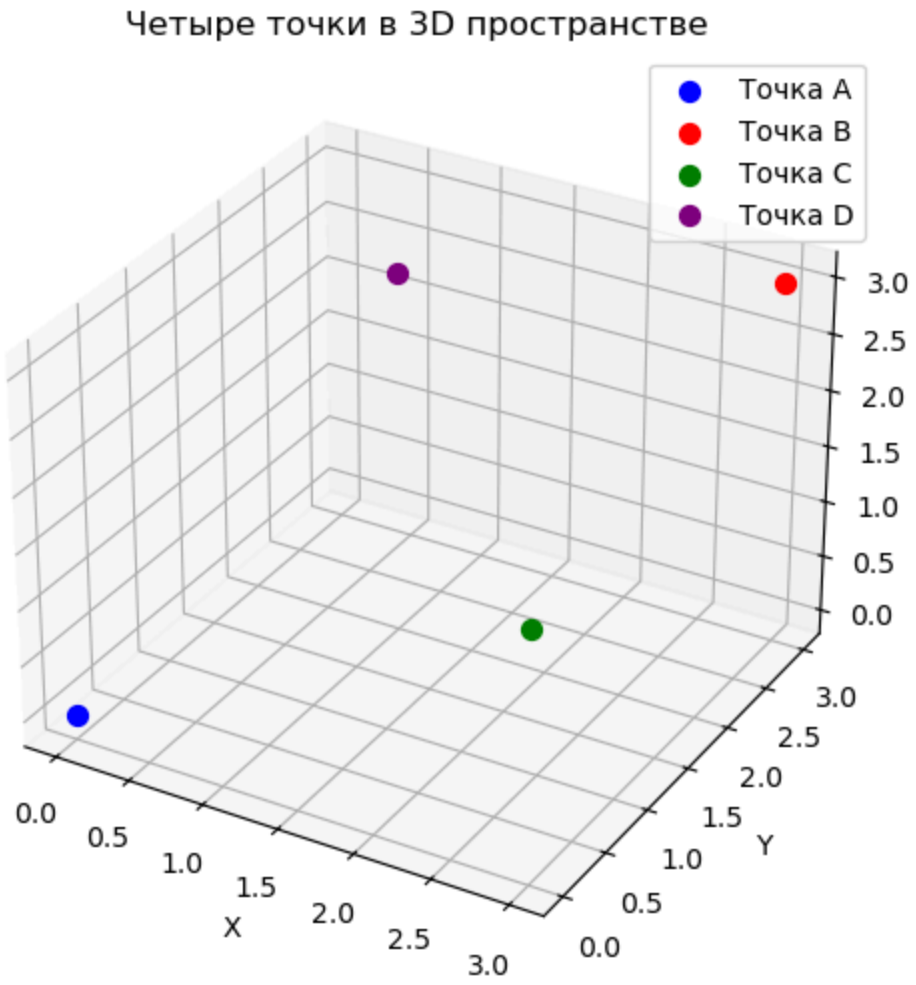
- Евклидово расстояние
- Квадрат евклидова расстояния
- Хеммингово (манхэттенское) расстояние
- Расстояние Чебышева

Решение:

In [8]:
```
import numpy as np
import matplotlib.pyplot as plt
from mpl_toolkits.mplot3d import Axes3D

A = np.array([0, 0, 0])
B = np.array([3, 3, 3])
C = np.array([3, 0, 2])
D = np.array([1, 2, 3])

# Функция для вычисления всех видов расстояний
def compute_distances(p1, p2):
    euclidean = np.linalg.norm(p1 - p2)
    sq_euclidean = euclidean ** 2
    manhattan = np.linalg.norm(p1 - p2, ord=1)
    chebyshev = np.linalg.norm(p1 - p2, ord=np.inf)

    return {
        'euclidean': euclidean,
        'sq_euclidean': sq_euclidean,
        'manhattan': manhattan,
        'chebyshev': chebyshev
    }

points = [A, B, C, D]
labels = ['A', 'B', 'C', 'D']

print("Расстояния между точками:")
for i in range(len(points)):
    for j in range(i+1, len(points)):
        dist = compute_distances(points[i], points[j])
        print(f"{labels[i]}-{labels[j]}:")
        print(f"  Евклидово: {dist['euclidean']:.4f}")
        print(f"  Квадрат евклида: {dist['sq_euclidean']:.4f}")
        print(f"  Хеммингово: {dist['manhattan']:.4f}")
        print(f"  Чебышева: {dist['chebyshev']:.4f}")
        print()
```

```
fig = plt.figure(figsize=(8, 6))
ax = fig.add_subplot(111, projection='3d')
colors = ['blue', 'red', 'green', 'purple']
for i, point in enumerate(points):
    ax.scatter(point[0], point[1], point[2], c=colors[i], s=50, label=f'Точка
ax.set_xlabel('X')
ax.set_ylabel('Y')
ax.set_zlabel('Z')
ax.set_title('Четыре точки в 3D пространстве')
ax.legend()
plt.show()
```

```
Расстояния между точками:
A-B:
  Евклидово: 5.1962
  Квадрат евклида: 27.0000
  Хеммингово: 9.0000
  Чебышева: 3.0000

A-C:
  Евклидово: 3.6056
  Квадрат евклида: 13.0000
  Хеммингово: 5.0000
  Чебышева: 3.0000

A-D:
  Евклидово: 3.7417
  Квадрат евклида: 14.0000
  Хеммингово: 6.0000
  Чебышева: 3.0000

B-C:
  Евклидово: 3.1623
  Квадрат евклида: 10.0000
  Хеммингово: 4.0000
  Чебышева: 3.0000

B-D:
  Евклидово: 2.2361
  Квадрат евклида: 5.0000
  Хеммингово: 3.0000
  Чебышева: 2.0000

C-D:
  Евклидово: 3.0000
  Квадрат евклида: 9.0000
  Хеммингово: 5.0000
  Чебышева: 2.0000
```



1.3.2. Задание

Задача:

Создать матрицу размером 5×5, где значения в каждой строке изменяются от 0 до 4 (включительно).
Для создания необходимо использовать функцию `np.arange()`.

Решение:

In [9]:
```
import numpy as np

Z = np.zeros((5, 5))
Z += np.arange(5)
print(Z)
```

```
[[0. 1. 2. 3. 4.]
 [0. 1. 2. 3. 4.]
 [0. 1. 2. 3. 4.]
 [0. 1. 2. 3. 4.]
 [0. 1. 2. 3. 4.]]
```

2.1. Теоретический материал – Задачи классификации

Решение задачи классификации методом k ближайших соседей

Метод k-ближайших соседей используется для решения задачи классификации. Он относит объекты к классу, которому принадлежит большинство из k его ближайших соседей в многомерном пространстве признаков. Это один из простейших алгоритмов обучения классификационных моделей. Число k – это количество соседних объектов в пространстве признаков, которые сравниваются с классифицируемым объектом. Иными словами, если k=10, то каждый объект сравнивается с 10-ю соседями. В процессе обучения алгоритм просто запоминает все векторы признаков и соответствующие им метки классов. При работе с реальными данными, т.е. наблюдениями, метки класса которых неизвестны, вычисляется расстояние между вектором нового наблюдения и ранее запомненными. Затем выбирается k ближайших к нему векторов, и новый объект относится к классу, которому принадлежит большинство из них.

Приведем алгоритм метода.

1. Выберите значение **K** соседей (скажем, k = 5)
2. Найдите ближайшую точку данных **K** (5) для нашей новой точки данных на основе евклидова расстояния (которое мы обсудим позже)
3. Среди этих **K** точек данных подсчитайте точки данных в каждой категории.
4. Назначьте новую точку данных категории, которая имеет наибольшее количество соседей с новой точкой данных

Модуль библиотеки sklearn - sklearn.neighbors предоставляет функциональные возможности для контролируемого обучения на основе соседей. Обучение на основе контролируемых соседей бывает двух видов: классификация данных с дискретными метками и регрессия для данных с непрерывными метками. В данном разделе рассмотрим несколько примеров с использованием названного метода.

2.2.1. Пример

Задача:

В примере показано создание 2D-массива со значениями $ x $ и $ y $. Список `target` содержит возможные выходные классы (часто называемые

метками).
Далее происходит обучение классификатора k-ближайших соседей по исходным данным, а затем — прогноз принадлежности к классам для двух точек данных.

Решение:

```
from sklearn.neighbors import KNeighborsClassifier
import numpy as np

X = np.array([[-1, -1], [-2, -1], [-3, -2], [1, 1], [2, 1], [3, 2]])
target = [0, 0, 0, 1, 1, 1]
K = 3
model = KNeighborsClassifier(n_neighbors=K)
model.fit(X, target)
print("(-2,-2) is class:", model.predict([[-2, -2]]))
print("(1,3) is class:", model.predict([[1, 3]]))
```

Ответ:

```
KNeighborsClassifier(n_neighbors=3)
(-2,-2) is class
[0]
(1,3) is class
[1]
```

2.2.2. Пример

Задача:

Далее приведем более наглядный пример. Будет построена граница решения для каждого класса. В качестве данных будем использовать уже знакомый нам и встроенный в библиотеку `sklearn` набор данных ирисов Фишера. Этот набор данных стал уже классическим, и часто используется в литературе для иллюстрации работы различных статистических алгоритмов. Датасет содержит наблюдения за 150 разными цветками ирисов, данные по каждому цветку расположены в строках. В стобцах записаны длина и ширина чашелистика, длина и ширина лепестка, вид ириса.

Решение:

```
from sklearn.neighbors import KNeighborsClassifier
import seaborn as sns
import matplotlib.pyplot as plt

iris = sns.load_dataset('iris')
iris
```

Ответ:

```
   sepal_length  sepal_width  petal_length  petal_width species
0           5.1          3.5           1.4          0.2  setosa
1           4.9          3.0           1.4          0.2  setosa
2           4.7          3.2           1.3          0.2  setosa
3           4.6          3.1           1.5          0.2  setosa
4           5.0          3.6           1.4          0.2  setosa

...

   sepal_length  sepal_width  petal_length  petal_width species
145           6.7          3.0           5.2          2.3  virginica
146           6.3          2.5           5.0          1.9  virginica
147           6.5          3.0           5.2          2.0  virginica
148           6.2          3.4           5.4          2.3  virginica
149           5.9          3.0           5.1          1.8  virginica

150 rows × 5 columns
```

2.2.3. Пример

Задача:

Покажем на графиках зависимости:

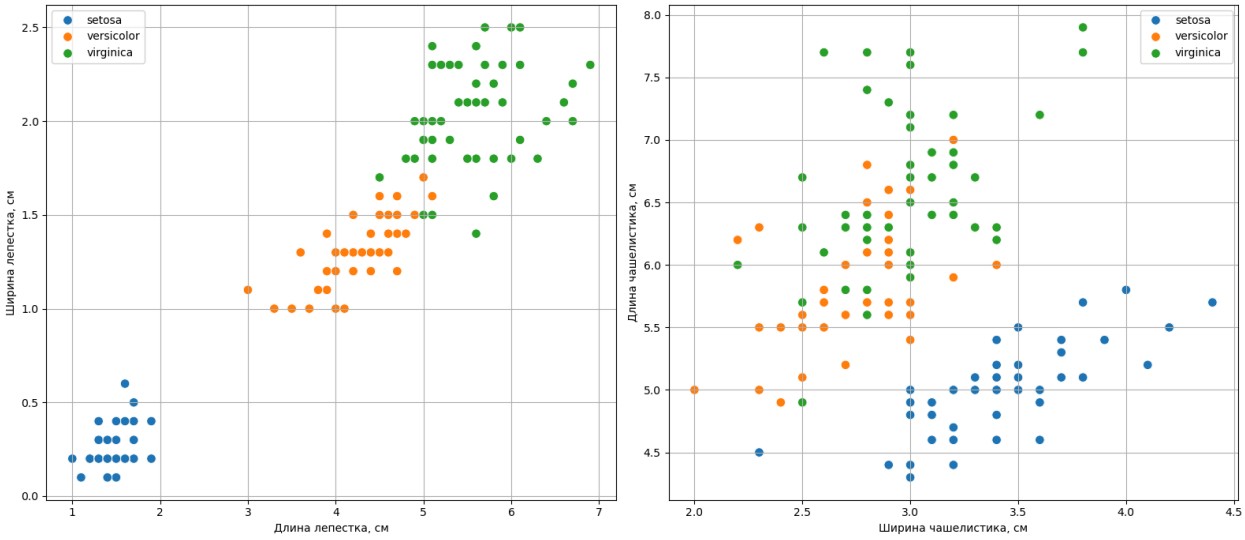
1. ширины лепестка от его длины,
2. длины чашелистика от его ширины.

Разные виды цветков (species) отмечены разными цветами.

Решение:

In [2]:
```python
import matplotlib.pyplot as plt
import seaborn as sns

# Загрузка данных
iris = sns.load_dataset('iris')

# Создание фигуры с двумя подграфиками
plt.figure(figsize=(16, 7))

# Левый график: зависимость ширины лепестка от длины
plt.subplot(121)
sns.scatterplot(
    data=iris,
    x='petal_length',  # длина лепестка
    y='petal_width',   # ширина лепестка
    hue='species',     # цвет по виду цветка
```

```
    s=70              # размер точек
)
plt.xlabel('Длина лепестка, см')
plt.ylabel('Ширина лепестка, см')
plt.legend()
plt.grid()

# Правый график: зависимость длины чашелистика от ширины
plt.subplot(122)
sns.scatterplot(
    data=iris,
    x='sepal_width',   # ширина чашелистика
    y='sepal_length',  # длина чашелистика
    hue='species',
    s=70
)
plt.xlabel('Ширина чашелистика, см')
plt.ylabel('Длина чашелистика, см')
plt.legend()
plt.grid()

# Отображение графиков
plt.tight_layout()
plt.show()
```



2.2.4. Пример

Задача:

Из графиков видно, что в первом случае классы визуально хорошо отделимы друг от друга, хотя два класса имеют небольшое пересечение.
Во втором случае разделить два класса между собой уже намного труднее.

Далее разделим датасет на обучающую и тестовую выборки в соотношении

80:20:

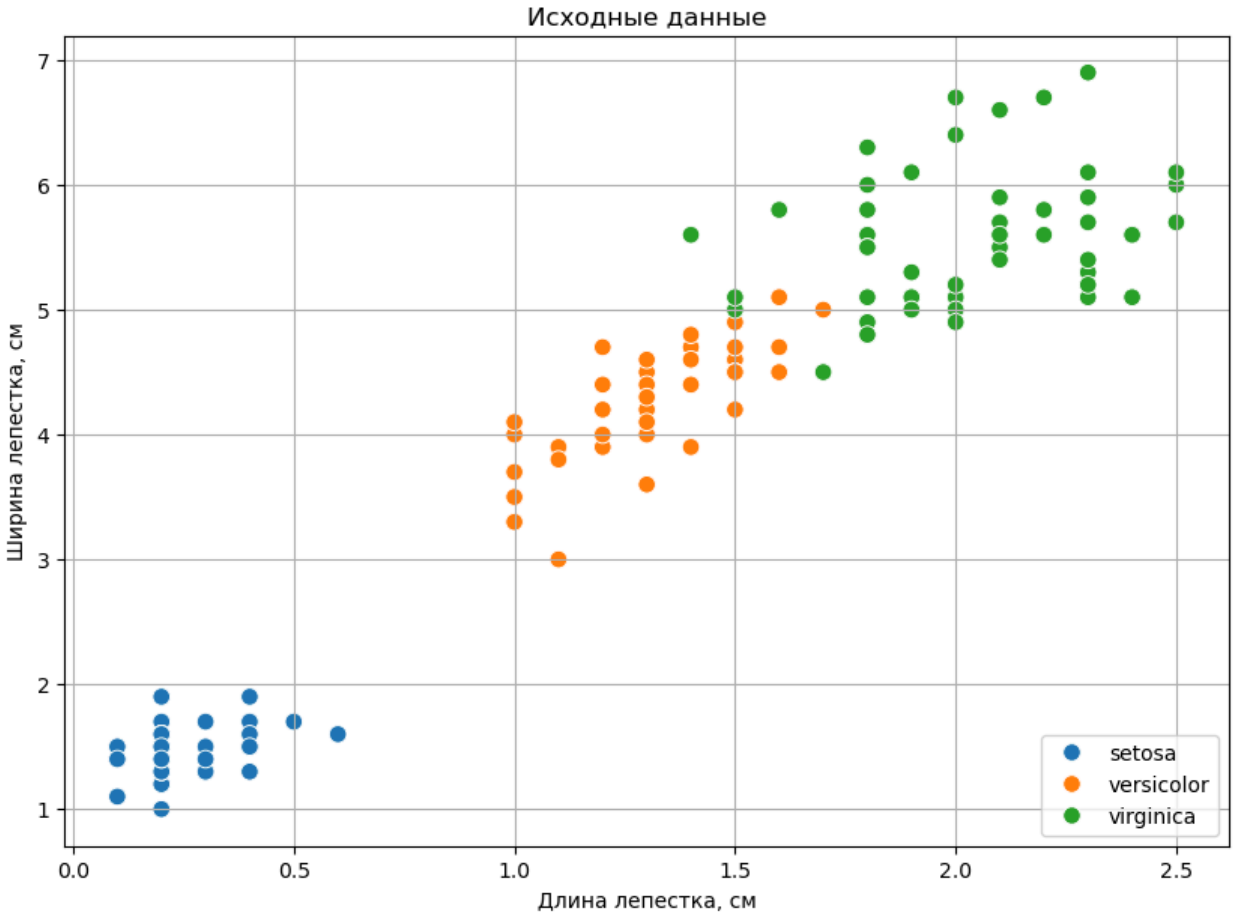
- **Обучающая выборка (training sample)** — используется для обучения модели (настройки параметров),
- **Тестовая (или контрольная) выборка (test sample)** — используется для оценки качества построенной модели.

Решение:

In [3]:
```
from sklearn.model_selection import train_test_split
from sklearn.neighbors import KNeighborsClassifier
from sklearn.metrics import accuracy_score
import matplotlib.pyplot as plt
import seaborn as sns
import numpy as np

# Загрузка данных
iris = sns.load_dataset('iris')

# Разделение признаков и целевой переменной
X = iris.iloc[:, :-1]  # Все столбцы кроме последнего (признаки)
y = iris.iloc[:, -1]   # Последний столбец — класс (вид цветка)

# Разделение на обучающую и тестовую выборки (20% для теста)
X_train, X_test, y_train, y_test = train_test_split(
    X, y, test_size=0.2, random_state=42
)

# Проверка размеров
print("Размеры выборок:")
print(f"X_train: {X_train.shape}")
print(f"X_test: {X_test.shape}")
print(f"y_train: {y_train.shape}")
print(f"y_test: {y_test.shape}")

# Вывод первых строк
print("\nПервые строки X_train:")
print(X_train.head())
print("\nПервые значения y_train:")
print(y_train.head())

# Обучение модели KNN (3 ближайших соседа)
model = KNeighborsClassifier(n_neighbors=3)
model.fit(X_train, y_train)

# Предсказание на тестовой выборке
y_pred = model.predict(X_test)

# Визуализация результатов
plt.figure(figsize=(10, 7))
sns.scatterplot(
    x='petal_width', y='petal_length',
```

```
    data=iris, hue='species', s=70
)
plt.xlabel('Длина лепестка, см')
plt.ylabel('Ширина лепестка, см')
plt.legend(loc='lower right')
plt.grid()
plt.title('Исходные данные')

# Отметим неправильно классифицированные точки красным
for i in range(len(y_test)):
    if np.array(y_test)[i] != y_pred[i]:
        plt.scatter(X_test.iloc[i, 3], X_test.iloc[i, 2], color='red', s=150,

plt.show()

# Оценка качества модели
accuracy = accuracy_score(y_test, y_pred)
print(f"\nТочность модели: {accuracy:.3f}")
```

```
Размеры выборок:
X_train: (120, 4)
X_test: (30, 4)
y_train: (120,)
y_test: (30,)

Первые строки X_train:
    sepal_length  sepal_width  petal_length  petal_width
22           4.6          3.6           1.0          0.2
15           5.7          4.4           1.5          0.4
65           6.7          3.1           4.4          1.4
11           4.8          3.4           1.6          0.2
42           4.4          3.2           1.3          0.2

Первые значения y_train:
22        setosa
15        setosa
65    versicolor
11        setosa
42        setosa
Name: species, dtype: object
```




```
Точность модели: 1.000
```

2.3.1. Задание

Задача:

Для предыдущего примера (классификация цветков ириса) поэкспериментируйте с параметрами классификатора:

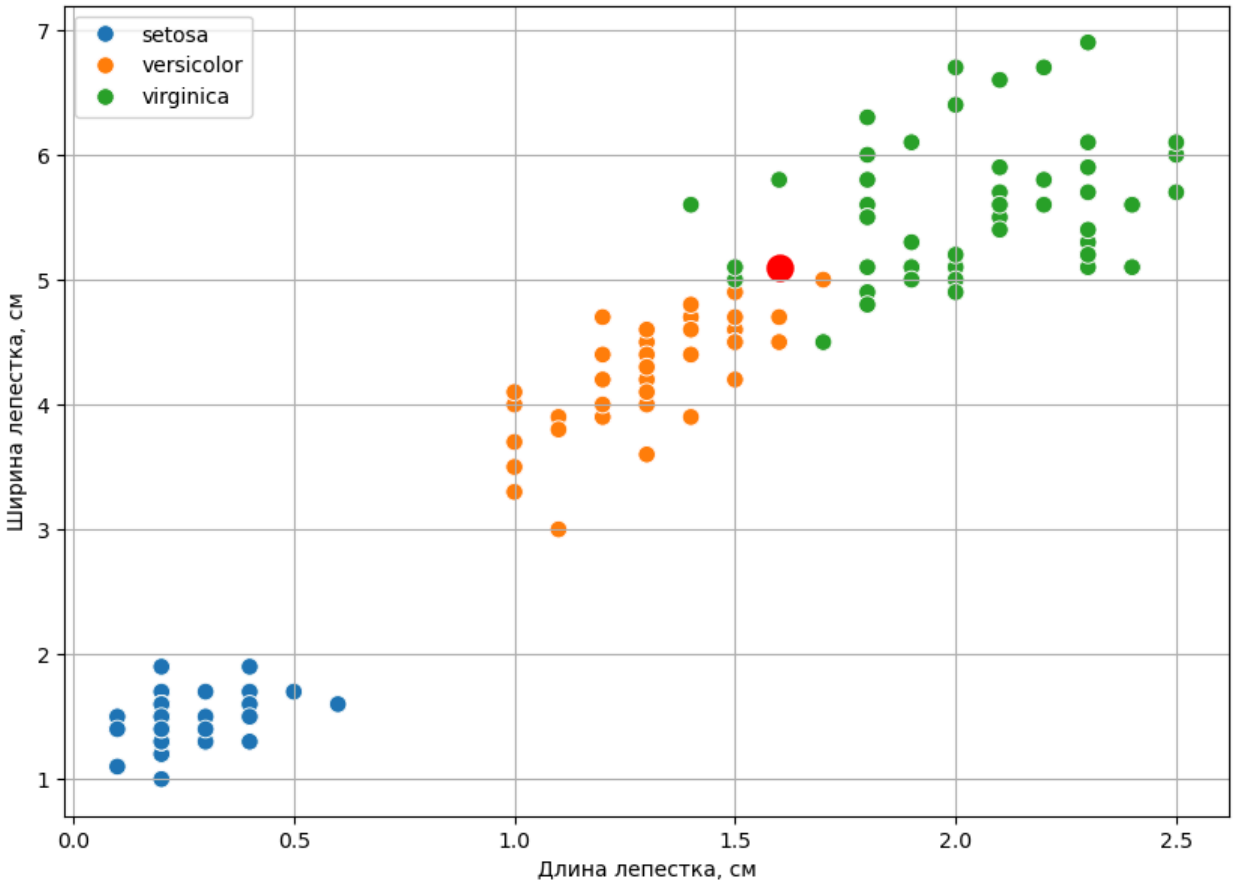
1. Установите другое количество ближайших соседей: $ k = 1, 5, 10 $
2. Установите размер тестовой выборки **15%** от всего датасета
3. Постройте графики и оцените качество моделей, проанализируйте результаты

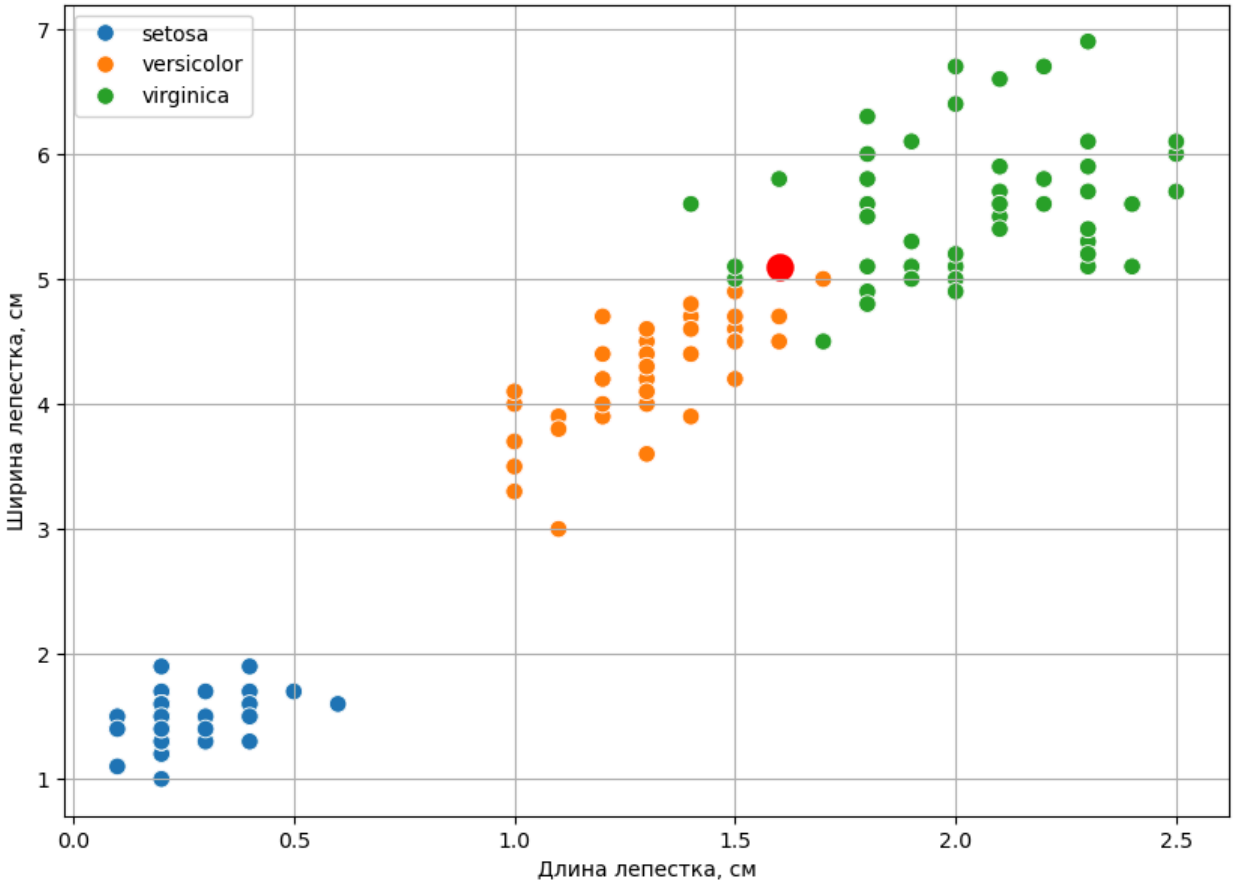
Решение:

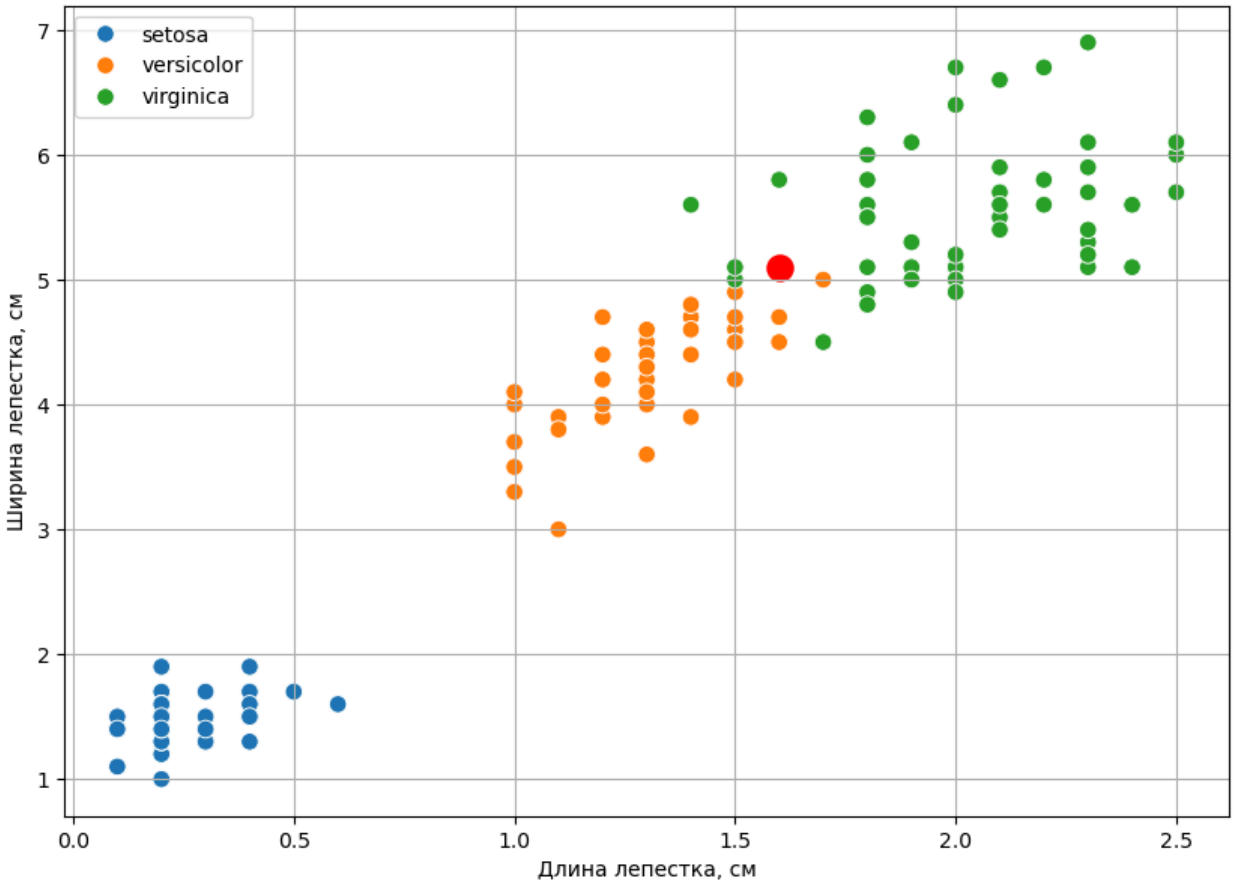
In [4]:
```
from sklearn.neighbors import KNeighborsClassifier
from sklearn.model_selection import train_test_split
import numpy as np
import seaborn as sns

iris = sns.load_dataset('iris')
```

```
X_train, X_test, y_train, y_test = train_test_split(
    iris.iloc[:, :-1],
    iris.iloc[:, -1],
    test_size=0.15
)
X_train.shape, X_test.shape, y_train.shape, y_test.shape


def init_model(k, X_train, y_train, X_test):
    model = KNeighborsClassifier(n_neighbors=k)
    model.fit(X_train, y_train)
    print(model)
    y_pred = model.predict(X_test)
    return y_pred


def graph(y_test, X_test, y_pred):
    plt.figure(figsize=(10, 7))
    sns.scatterplot(x='petal_width', y='petal_length', data=iris, hue='species
    plt.xlabel('Длина лепестка, см')
    plt.ylabel('Ширина лепестка, см')
    plt.legend(loc=2)
    plt.grid()

    for i in range(len(y_test)):
        if np.array(y_test)[i] != y_pred[i]:
            plt.scatter(X_test.iloc[i, 3], X_test.iloc[i, 2], color='red', s=1


y_pred = init_model(1, X_train, y_train, X_test)
graph(y_test, X_test, y_pred)
y_pred = init_model(5, X_train, y_train, X_test)
graph(y_test, X_test, y_pred)
y_pred = init_model(10, X_train, y_train, X_test)
graph(y_test, X_test, y_pred)
```

```
KNeighborsClassifier(n_neighbors=1)
KNeighborsClassifier()
KNeighborsClassifier(n_neighbors=10)
```


setosa
versicolor
virginica
Ширина лепестка, см
Длина лепестка, см
0.0
0.5
1.0
1.5
2.0
2.5
1
2
3
4
5
6
7


setosa
versicolor
virginica
Ширина лепестка, см
Длина лепестка, см
0.0
0.5
1.0
1.5
2.0
2.5
1
2
3
4
5
6
7



3.1. Теоретический материал – Работа с категориальными данными

Категориальные данные — это информация, которая описывает качественные характеристики объектов (например, пол, цвет, марка автомобиля).
Часто бывает полезно разбивать объекты не по количеству, а по **категориям**.

> 🔹 Качественная информация редко представляется числами, но большинство алгоритмов машинного обучения требуют числовые признаки.

Типы категориальных данных

Категории могут быть:

- **Номинальными** — без естественного порядка:
  - синий, красный, зелёный;
  - мужчина, женщина;
  - банан, клубника, яблоко.

- **Порядковыми** — с естественным упорядочением:
    - низкий, средний, высокий;
    - молодые, старые;
    - согласен, нейтрален, не согласен.

> 📌 Пример:
> Столбец с городами: "Москва", "Санкт-Петербург", "Казань" — это **номинальные** данные, так как нет естественного порядка.

Проблема обработки категориальных данных

Алгоритмы, такие как k-ближайших соседей, часто используют вычисление расстояний между наблюдениями.
Однако строковые значения (например, "Москва") **не могут быть использованы напрямую** в математических операциях.

> ❌ Проблема:
> Если просто присвоить числам: Москва=1, Санкт-Петербург=2, Казань=3, то возникнет ложное представление о порядке и расстоянии между городами.

> ✅ Решение:
> Необходимо преобразовать категориальные данные в числовой формат таким образом, чтобы **сохранялась их семантика**.

Методы кодирования категориальных данных

Существует множество способов преобразования категориальных признаков:

- **One-Hot Encoding** — для номинальных данных
- **Label Encoding** — для порядковых данных
- **Target Encoding** — для больших наборов категорий
- **Binary Encoding** — для экономии памяти

> Выбор метода зависит от:
>
> - типа данных (номинальные или порядковые),
> - количества категорий,
> - мощности модели,
> - алгоритма машинного обучения.

> В следующих разделах рассмотрим основные методы

> преобразования.

3.2.1. Пример

Задача:

Дан порядковый категориальный признак (например, "низкий", "средний", "высокий").
Выполнить его кодировку — преобразовать в числовой эквивалент.

Решение:

```
import pandas as pd

# Создание фрейма данных с порядковым признаком
dataframe = pd.DataFrame({
    'оценка': ['низкая', 'низкая', 'средняя', 'средняя', 'высокая']
})

# Создание словаря для преобразования значений
scale_mapper = {
    'низкая': 1,
    'средняя': 2,
    'высокая': 3
}

# Замена строковых значений на числовые
dataframe['оценка'] = dataframe['оценка'].replace(scale_mapper)
print(dataframe)
```

Ответ:

```
0       1
1       1
2       2
3       2
4       3
Name: оценка, dtype: int64
```

3.2.2. Пример

Задача:

Дан словарь, и требуется его конвертировать в матрицу признаков. Для решения задачи можно задействовать класс-векторизатор словаря Dictvectorizer:

Решение:

```
from sklearn.feature_extraction import DictVectorizer

# Создание списка словарей
data_dict = [
    {"красный": 2, "синий": 4},
    {"красный": 4, "синий": 3},
    {"красный": 1, "желтый": 2},
    {"красный": 2, "желтый": 2}
]

# Создание векторизатора
dictvectorizer = DictVectorizer(sparse=False)

# Преобразование словаря в матрицу признаков
features = dictvectorizer.fit_transform(data_dict)

print(features)
```

Ответ:

```
array([[0. 2. 4.]
     [0. 4. 3.]
     [2. 1. 0.]
     [2. 2. 0.]])
```

3.3.2. Пример

Задача:

Определите набор признаков человека, по аналогии из `PT 1`, – например, цвет глаз и конвертируйте его в матрицу признаков.

Решение:

In [6]:
```
from sklearn.feature_extraction import DictVectorizer


hair_color = [
    {'blond': 1, 'brunet': 0, 'ginger': 0, 'brown hair': 0, 'gray-haired': 0},
    {'blond': 0, 'brunet': 0, 'ginger': 1, 'brown hair': 0, 'gray-haired': 0},
    {'blond': 1, 'brunet': 0, 'ginger': 0, 'brown hair': 0, 'gray-haired': 0},
    {'blond': 0, 'brunet': 1, 'ginger': 0, 'brown hair': 0, 'gray-haired': 0},
    {'blond': 0, 'brunet': 1, 'ginger': 0, 'brown hair': 0, 'gray-haired': 0},
    {'blond': 0, 'brunet': 0, 'ginger': 0, 'brown hair': 1, 'gray-haired': 0},
    {'blond': 0, 'brunet': 0, 'ginger': 0, 'brown hair': 0, 'gray-haired': 1},
    {'blond': 0, 'brunet': 0, 'ginger': 0, 'brown hair': 0, 'gray-haired': 0},
    {'blond': 0, 'brunet': 1, 'ginger': 0, 'brown hair': 0, 'gray-haired': 0},
```

```
    {'blond': 1, 'brunet': 0, 'ginger': 0, 'brown hair': 0, 'gray-haired': 0},
    {'blond': 0, 'brunet': 0, 'ginger': 1, 'brown hair': 0, 'gray-haired': 0},
]



dictvectorizer = DictVectorizer(sparse=False)
features = dictvectorizer.fit_transform(hair_color)
features
```

Out[6]:
```
array([[1., 0., 0., 0., 0.],
       [0., 0., 0., 1., 0.],
       [1., 0., 0., 0., 0.],
       [0., 0., 1., 0., 0.],
       [0., 0., 1., 0., 0.],
       [0., 1., 0., 0., 0.],
       [0., 0., 0., 0., 1.],
       [0., 0., 0., 0., 0.],
       [0., 0., 1., 0., 0.],
       [1., 0., 0., 0., 0.],
       [0., 0., 0., 1., 0.]])
```

In [ ]:
```

```