**Импорты и зачем они нужны**

В начале подключаются библиотеки для работы с данными, визуализацией и моделями. Кратко:

* pandas — для чтения CSV и манипуляций с таблицей;
* numpy — для числовых операций (иногда используется неявно);
* matplotlib и seaborn — для построения графиков и визуальной проверки распределений;
* из sklearn берутся инструменты для разбиения данных, масштабирования, сами модели (логистическая регрессия, KNN, случайный лес, SVM) и метрики (accuracy).  
  Каждая импортируемая сущность используется в соответствующем блоке: EDA, предобработка, обучение, оценка.

### Creation and testing different models for prediction of cancer, their evaluation.

### General-purpose libraries

import **numpy** as np

* **NumPy**: Core Python library for numerical computing.
* **Provides arrays, mathematical operations (linear algebra, stats, etc.).**
* Very fast and efficient with large datasets.

import **matplotlib.pyplot** as plt

* **Matplotlib**: Standard Python plotting library.
* **pyplot provides functions to create charts (line, bar, scatter, image plots, etc.).**
* Often used for low-level control of plots.

import **seaborn** as sns

* **Seaborn**: **Builts on top of Matplotlib.**
* **Makes statistical graphics easier and prettier (heatmaps, countplots, boxplots, etc.).**
* Great for Exploratory Data Analysis (EDA).

### 📦 Scikit-learn (Machine Learning)

from sklearn.model\_selection import **train\_test\_split**

* **Splits dataset into training and testing sets.**
* Important to evaluate model performance on unseen data.

from sklearn.preprocessing import **StandardScaler**

* **Used to standardize features: subtract mean and divide by standard deviation.**
* **Ensures all features are on the same scale** (important for models like KNN and SVM).

from sklearn.linear\_model import **LogisticRegression**

* **Logistic Regression model: A linear classifier, good for binary classification problems.**

from sklearn.neighbors import **KNeighborsClassifier**

* **K-Nearest Neighbors (KNN)**: **A simple model that classifies a point based on the majority class of its nearest neighbors.**

from sklearn.ensemble import **RandomForestClassifier**

* **Random Forest**: **An ensemble of decision trees.**
* Robust, works well for many datasets, **reduces overfitting**.

from sklearn.svm import **SVC**

* **Support Vector Classifier (SVC)**: A powerful classifier that **finds the best boundary (hyperplane) to separate classes.**
* **Works well with both linear and non-linear data (using kernels).**

If the data is separated by a complex curve → the kernel is connected (for example, RBF, polynomial). It transfers the data to another space where it is already separable.

The kernel is a special function that transforms data into another (often higher-dimensional) space. The kernel allows you to "lift" data to a higher dimension, where you can already draw a hyperplane that will separate them. The Kernel does exactly this transformation **implicitly**, so as not to manually calculate coordinates in new dimensions.

from sklearn.metrics import **accuracy\_score**

**Scikit-learn**

* Used to measure model performance.
* accuracy\_score(y\_true, y\_pred) → **percentage of correct predictions.**

**Загрузка и первичный осмотр данных**

Сначала файл загружается в DataFrame. Дальше выполняется быстрый осмотр:

* вывод первых строк для понимания структуры и типов столбцов;
* информация о столбцах и количестве ненулевых значений (показывает типы данных и пропуски);
* подсчёт пропусков по столбцам, чтобы понять, есть ли отсутствующие значения.

На основании этой информации принимаются решения о чистке данных: какие столбцы — идентификаторы или полностью пустые — можно удалить, какие требуют заполнения/обработки.

### 1. ****Check for missing values****

print("\nMissing values per column:")

print(df.isnull().sum())

* This prints out how many missing (NaN) values there are in each column of the DataFrame df.
* It helps you quickly see if some columns have incomplete data.
* **Удаление ненужных столбцов**
* Из данных удаляют явные «идентификаторы» (например, id) и столбец, содержащий только NaN (Unnamed: 32). Причина: id не несёт полезной информации для обучения модели (это уникальный ключ), а пустой столбец — мусор. Удаление таких столбцов уменьшает размерность и уменьшает риск, что модель начнёт «учитывать» бессмысленные признаки.

### 2. ****Drop unnecessary columns****

cols\_to\_drop = ["id", "Unnamed: 32"]

existing\_cols\_to\_drop = [col for col in cols\_to\_drop if col in df.columns]

if existing\_cols\_to\_drop:

df = df.drop(existing\_cols\_to\_drop, axis=1)

* The dataset has useless columns, e.g.:
  + id → just an identifier, not useful for analysis.
  + Unnamed: 32 → an empty column with all NaN values.
* The code first **checks if these columns exist** in the DataFrame before trying to remove them (to avoid errors).
* If they exist, they are dropped from the dataset.
* **Визуализация распределения классов**
* Построение countplot (гистограммы по категориям) для столбца diagnosis даёт представление о числе объектов в каждом классе (доброкачественные/злокачественные). Это важно, чтобы увидеть, сбалансированы ли классы. При визуализации учитывается оформление (палитра/легенда), но в новых версиях seaborn надо следить за аргументами palette и hue, чтобы не возникало предупреждений.

### 3. ****Plot the distribution of diagnosis values (with** pyplot**)

plt.figure(figsize=(6,4))

sns.countplot(x="diagnosis", data=df, hue="diagnosis", palette="magma", legend=False)

plt.title("Distribution of Diagnosis")

plt.show()

* This creates a **count plot** (bar chart) showing how many samples are classified as:
  + **Benign (B)** – not cancer
  + **Malignant (M)** – cancer
* hue="diagnosis" ensures each bar is colored differently.
* palette="magma" applies a nice color scheme.
* legend=False hides the extra legend because it’s not needed (the x-axis already shows the categories).

✅ **Summary:**  
This block of code:

1. Checks for missing values.
2. Cleans the dataset by removing useless columns.
3. Visualizes how many benign vs malignant cases there are.

**Предобработка целевой переменной**

### 1. ****Check unique values in the target column****

print("\nCounts of unique values in 'diagnosis':")

print(df["diagnosis"].value\_counts())

* Prints how many samples belong to each category in the column **diagnosis**.
* For example:
  + **M (Malignant = cancer)**
  + **B (Benign = not cancer)**
* This helps you see if the dataset is balanced or skewed (e.g., more benign than malignant cases).

Далее считается распределение классов (value\_counts) и категориальная переменная diagnosis переводится в числовой формат (mapping:, например, M→1, B→0). Это обязательно: большинство алгоритмов машинного обучения ожидают числовую целевую переменную для задачи классификации.

### 2. ****Convert categorical labels into numbers****

df["diagnosis"] = df["diagnosis"].map({"M":1, "B":0})

* Machine learning models need numbers, not letters.
* Here:
  + **M → 1** (Malignant)
  + **B → 0** (Benign)

### 3. ****Split dataset into features and target****

X = df.drop("diagnosis", axis=1) #leave everything except diagnosis

y = df["diagnosis"]

* **X (features)** → all the columns except diagnosis (e.g., measurements like radius, texture, etc.).
* **y (target)** → the diagnosis column (0 or 1, after mapping).

**Масштабирование признаков (StandardScaler)**

Стандартизация переводит признаки в пространство с нулевым средним и единичным стандартным отклонением. Причина: некоторые алгоритмы (KNN, SVM) чувствительны к масштабу признаков — без масштабирования признаки с большими численными значениями будут доминировать. Стандартизация делает обучение более стабильным и сравнимым между признаками.

Важно заметить (практическая оговорка): масштабирование лучше подгонять *только по обучающей выборке* (fit на train, transform на test), чтобы избежать утечки информации из теста в тренировку. В первоначальном коде масштабирование делается до разбиения — это можно улучшить.

### 4. ****Standardize features****

scaler = StandardScaler()

X\_scaled = scaler.fit\_transform(X)

* Some models (like **KNN** and **SVM**) are very sensitive to the **scale** of the data.
* Example: if one feature is in millimeters (0–10) and another is in micrometers (0–1000), **the bigger scale dominates the model**.
* StandardScaler transforms features so they all have:
  + **Mean = 0**
  + **Standard deviation = 1**

This makes training fairer.

**Формирование матрицы признаков и целевой переменной**

Отдельно формируются:

* X — матрица признаков (все столбцы кроме diagnosis);
* y — вектор целей (diagnosis).  
  Такой раздел позволяет четко понимать, что мы передаём моделям на вход и что они должны предсказывать.
* **Разбиение на обучающую и тестовую выборки**
* Применяется разбиение (train\_test\_split) с параметром test\_size=0.2 и фиксированным random\_state. Это означает, что 20% данных откладываются для проверки качества, а 80% идут на обучение. random\_state нужен для воспроизводимости: при одинаковом значении разбиение будет одинаковым при каждом запуске.
* При разбивании полезно учитывать стратификацию (stratify) по целевой переменной, чтобы соотношение классов в train и test было одинаковым — это особенно важно при несбалансированных классах.

### 5. ****Split into training and testing sets****

X\_train, X\_test, y\_train, y\_test = train\_test\_split(

X\_scaled, y, test\_size=0.2, random\_state=42

)

* Splits the dataset into two parts:
  + **80% for training** (model learns patterns).
  + **20% for testing** (model is evaluated).
* random\_state=42 ensures the split is **reproducible** (same results every time).

✅ **Summary:**  
This preprocessing step:

1. Checks class distribution.
2. Converts labels (M/B) into numbers.
3. Separates features from the target.
4. Standardizes the feature values.
5. Splits the dataset into training and test sets.

**Построение и обучение моделей**

В коде последовательно создаются четыре модели и оцениваются:

* Логистическая регрессия: линейный классификатор, полезен как базовая модель. Параметр max\_iter указывается, чтобы обеспечить сходимость алгоритма при большом числе признаков.
* K-Nearest Neighbours (KNN): не модель на основе расстояний; параметр n\_neighbors (по умолчанию 5) задаёт число ближайших соседей для голосования.
* Random Forest (случайный лес): ансамбль решающих деревьев; n\_estimators — число деревьев в ансамбле, random\_state — для воспроизводимости.
* SVM (Support Vector Machine): в коде используется линейное ядро (kernel="linear"), хорошо работает для разделимых векторов и в сочетании со стандартизацией.

Для каждой модели выполняется метод обучения (fit) на train и предсказание (predict) на test.

**Оценка: accuracy**

Качество каждой модели измеряется метрикой accuracy\_score — доля верных предсказаний на тестовой выборке. Полученные значения собираются в словарь model\_accuracies, затем определяется «лучший» модель по максимальной точности (функция max с ключом, возвращающим значение словаря).

### ****Setup****

model\_accuracies = {}

* Creates an **empty dictionary** to store the accuracy of each model.
* Example later: {"Logistic Regression": 0.95, "KNN": 0.93, ...}

### ****1. Logistic Regression****

**For classification problems**

* Unlike **Linear Regression**, which predicts continuous numbers, **Logistic Regression predicts categories (classes)**.
* **Simple, fast, and intErpretable**
  + It’s often the **first model to try** because:
    - It trains quickly.
    - You can easily interpret the coefficients to understand which features are important.
    - It works well if the relationship between features and the outcome is roughly linear.
* **Arguments and functions:**
* max\_iter=10000 → ensures the algorithm has enough iterations to converge.
* fit() function trains the model on the training data.
* predict() function makes predictions on the test set.
* accuracy\_score() function compares predictions with the real answers (y\_test).
* Save the accuracy in the dictionary and print it.

### ****2. K-Nearest Neighbors (KNN)****

We need KNN because it’s a **simple, intuitive, and flexible algorithm** that works well when decision boundaries are complex or non-linear. It’s especially useful as a baseline model for small to medium datasets.

**If it looks like a duck, swims like a duck, and quacks like a duck, then it is a duck**

* Uses **5 neighbors** to classify each point.
* Trains with training data, predicts on test data.
* Stores and prints accuracy.

### ****3. Random Forest****

We need Random Forest because it’s a **robust, flexible, and powerful model** that reduces overfitting, handles complex data, and provides feature importance. It’s ideal when you want high accuracy without heavy assumptions about data structure.

* n\_estimators=100 → builds a forest of **100 decision trees**.
* Each tree votes, and the majority wins.
* More robust and less likely to overfit than a single decision tree.
* Accuracy is calculated, stored, and printed.

### ****4. Support Vector Machine (SVM)****

We use SVM because it’s a **precise and flexible classifier** that finds the best separation between classes, works with non-linear data using kernels, handles high-dimensional datasets, and is robust to outliers and noise.

* Uses a **linear kernel** → tries to separate the two classes with a straight line (or hyperplane in higher dimensions).
* Train, predict, evaluate, store, and print accuracy.

✅ **Summary:**  
This block:

Finds the key (model name) with the highest accuracy value.

Save result to best\_model

Stores the name of the model with the highest accuracy.

Print result

This code picks the most accurate model out of all tested ones and prints its name and accuracy score.

**Ограничения и тонкости оценки**

* Accuracy — простая метрика, но может вводить в заблуждение при несбалансированных классах. В таких случаях важно смотреть precision/recall, F1-score, confusion matrix и ROC-AUC.
* Одна единственная разбиение train/test даёт оценку, но может быть шумной. Для более надёжной оценки используют кросс-валидацию (cross-validation).
* В коде отсутствует подбор гиперпараметров (grid search / randomized search). Для реального улучшения моделей стоит оптимизировать параметры.
* Масштабирование до разбиения — риск утечки информации; правильнее выполнять fit scaler на обучающей выборке и потом применять transform к тестовой.
* Также полезно фиксировать seed для всех стохастических операций для воспроизводимости (random\_state в моделях, в train\_test\_split и т.д.).
* Для интерпретации модели (особенно логрег и случайного леса) можно посмотреть важности признаков / коэффициенты.

**Итог**

Код реализует стандартный ML-пайплайн: загрузка → очистка → визуализация → преобразование целевой переменной → масштабирование → разбиение → обучение четырёх моделей → оценка по accuracy → выбор победителя по максимальной точности. Основные места, где стоит улучшить процесс: корректная последовательность масштабирования, использование стратификации при разбиении, расширенная оценка качества (confusion matrix, precision/recall/F1, ROC-AUC), и подбор гиперпараметров с кросс-валидацией.

Imports and why they are needed

At the beginning, libraries for working with data, visualization, and models are connected. Briefly:

• pandas — for reading CSV and manipulating the table;

• numpy — for numeric operations (sometimes used implicitly);

• matplotlib and seaborn — for plotting and visual verification of distributions;

• sklearn provides tools for data partitioning, scaling, the models themselves (logistic regression, KNN, random forest, SVM) and metrics (accuracy).

Each imported entity is used in the corresponding block: EDA, preprocessing, training, evaluation.

Loading and initial inspection of data

First, the file is uploaded to the DataFrame. Next, a quick inspection is performed:

• output of the first rows to understand the structure and types of columns;

• information about columns and the number of non-zero values (shows data types and omissions);

• Count the gaps in the columns to see if there are missing values.

Based on this information, decisions are made about data cleaning: which columns — identifiers or completely empty ones — can be deleted, which ones require filling in/processing.

Deleting unnecessary columns

Explicit "identifiers" (for example, id) and a column containing only NaN (Unnamed: 32) are removed from the data. Reason: the id does not carry useful information for training the model (it is a unique key), and an empty column is garbage. Removing such columns reduces the dimension and reduces the risk that the model will start to "take into account" meaningless features.

Visualization of class distribution

Constructing a countplot (histogram by category) for the diagnosis column gives an idea of the number of objects in each class (benign/malignant). This is important to see if the classes are balanced. When rendering, the design (palette/legend) is taken into account, but in new versions of seaborn, you need to monitor the palette and hue arguments so that warnings do not occur.

Preprocessing the target variable

Next, the class distribution (value\_counts) is calculated and the categorical variable diagnosis is converted to a numeric format (mapping:, for example, M→1, B→0). This is mandatory: most machine learning algorithms expect a numerical target variable for a classification task.

Formation of a matrix of features and a target variable

They are formed separately:

• X — feature matrix (all columns except diagnosis);

• y is the vector of goals (diagnosis).

This section allows us to clearly understand what we are passing to the models as input and what they should predict.

Feature Scaling (StandardScaler)

Standardization translates features into a space with zero mean and a single standard deviation. The reason: some algorithms (KNN, SVM) are sensitive to the scale of features — without scaling, features with large numerical values will dominate. Standardization makes learning more stable and comparable between features.

It is important to note (practical caveat): it is better to adjust scaling only for the training sample (fit for train, transform for test) in order to avoid leakage of information from the test into the training. In the initial code, scaling is done before partitioning — this can be improved.

Splitting into training and test samples

Splitting is applied (train\_test\_split) with the parameter test\_size=0.2 and a fixed random\_state. This means that 20% of the data is set aside for quality control, and 80% goes to training. random\_state is needed for reproducibility: with the same value, the split will be the same every time you run.

When splitting, it is useful to consider stratification by the target variable so that the ratio of classes in train and test is the same — this is especially important for unbalanced classes.

Building and training models

Four models are created sequentially in the code and evaluated:

• Logistic regression: a linear classifier, useful as a basic model. The max\_iter parameter is specified to ensure the convergence of the algorithm with a large number of features.

• K-Nearest Neighbors (KNN): A distance-based non-model; the n\_neighbors parameter (default 5) sets the number of nearest neighbors to vote on.

• Random Forest: an ensemble of decision trees; n\_estimators — the number of trees in the ensemble, random\_state — for reproducibility.

• SVM (Support Vector Machine): the code uses a linear kernel (kernel="linear"), which works well for separable vectors and in combination with standardization.

For each model, a training method (fit) is performed on train and a prediction (predict) on test.

Rating: accuracy

The quality of each model is measured by the accuracy\_score metric, which is the percentage of correct predictions in the test sample. The resulting values are collected in the model\_accuracies dictionary, then the "best" model is determined for maximum accuracy (the max function with a key that returns the dictionary value).

Limitations and subtleties of evaluation

• Accuracy is a simple metric, but it can be misleading if classes are unbalanced. In such cases, it is important to watch precision/recall, F1-score, confusion matrix and ROC-AUC.

• A single train/test split gives an estimate, but it can be noisy. For a more reliable assessment, cross-validation is used.

• There is no selection of hyperparameters in the code (grid search / randomized search). To really improve the models, it is worth optimizing the parameters.

• Scaling to partitioning is a risk of information leakage; it is more correct to perform fit scaler on the training sample and then apply transform to the test sample.

• It is also useful to fix the seed for all stochastic operations for reproducibility (random\_state in models, in train\_test\_split, etc.).

• To interpret the model (especially logreg and random forest), you can look at the importance of features/coefficients.

Result

The code implements a standard ML pipeline: loading → cleaning → visualization → transformation of the target variable → scaling → partitioning → training four models → accuracy estimation → selection of the winner based on maximum accuracy. The main places where it is worth improving the process are: the correct scaling sequence, the use of stratification in partitioning, advanced quality assessment (confusion matrix, precision/recall/F1, ROC-AUC), and the selection of hyperparameters with cross-validation.