|  |
| --- |
| import sqlite3 # Create a SQLite connection conn = sqlite3.connect(':memory:') # Uses an in-memory database df.to\_sql('credit\_risk', conn, index=False, if\_exists='replace') # Load data into SQL |
| 32581 |
| **🔹 import sqlite3**  Ini mengimpor **modul sqlite3**, yang digunakan untuk bekerja dengan **database SQLite** di Python. SQLite adalah database ringan yang tidak memerlukan server. 🔹 conn = sqlite3.connect(':memory:') Ini membuat **koneksi ke database SQLite sementara di memori (RAM)**.   * ':memory:' artinya database hanya ada selama program berjalan. * Setelah program selesai atau koneksi ditutup, datanya **hilang**. * Ini cocok untuk **eksperimen atau pengujian cepat**.   Contoh: kamu tidak membuat file .db, jadi semuanya disimpan di RAM. 🔹 df.to\_sql('credit\_risk', conn, index=False, if\_exists='replace') Ini bagian dari **pandas**, bukan sqlite3. Fungsinya adalah:   * Menyimpan DataFrame df ke dalam **tabel SQLite** bernama 'credit\_risk'. * conn: koneksi ke SQLite yang tadi dibuat. * index=False: tidak menyimpan indeks DataFrame ke dalam tabel. * if\_exists='replace': kalau tabel 'credit\_risk' sudah ada, maka **hapus dan ganti** dengan yang baru dari df. |

|  |
| --- |
| df.head() |
|  |
| Ini Cuma lihat 5 baris awal |
| query = """ SELECT  AVG(loan\_amnt) AS "Average Loan",  AVG(loan\_int\_rate) AS "Average Interest Rate",  loan\_grade AS "Category" FROM credit\_risk GROUP BY loan\_grade; """  # Run SQL query result = pd.read\_sql\_query(query, conn) print(result) |
| Average Loan Average Interest Rate Category 0 8539.273453 7.327651 A 1 9995.483686 10.995555 B 2 9213.862651 13.463542 C 3 10849.241589 15.361448 D 4 12915.845436 17.009455 E 5 14717.323651 18.609159 F 6 17195.703125 20.251525 G |
| Kode yang kamu tulis ini berfungsi untuk **mengambil ringkasan data pinjaman berdasarkan grade pinjaman** dari tabel credit\_risk di database SQLite. Mari kita bahas baris per baris: 🔹 query = """ ... """ Ini adalah **query SQL** yang melakukan hal-hal berikut:  sql  SalinEdit  SELECT  AVG(loan\_amnt) AS "Average Loan",  AVG(loan\_int\_rate) AS "Average Interest Rate",  loan\_grade AS "Category"  FROM credit\_risk  GROUP BY loan\_grade; Penjelasan bagian-bagian query:  | **Bagian** | **Fungsi** | | --- | --- | | AVG(loan\_amnt) | Menghitung **rata-rata jumlah pinjaman** untuk setiap kategori loan\_grade | | AVG(loan\_int\_rate) | Menghitung **rata-rata suku bunga** pinjaman untuk tiap loan\_grade | | loan\_grade | Digunakan sebagai kategori pengelompokan (contoh: A, B, C, D...) | | GROUP BY loan\_grade | Mengelompokkan data berdasarkan grade pinjaman |  🔹 result = pd.read\_sql\_query(query, conn) Baris ini:   * Menjalankan **query SQL di atas** terhadap koneksi SQLite conn (yang memuat data df). * Mengembalikan hasilnya dalam bentuk **DataFrame pandas** bernama result. |

|  |
| --- |
| query = """ SELECT \* FROM credit\_risk WHERE loan\_status = 1 ORDER BY cb\_person\_cred\_hist\_length DESC LIMIT 10; """  # Run SQL query result = pd.read\_sql\_query(query, conn) print(result) |
| ✅ SELECT \*  * Artinya: **ambil semua kolom** dari tabel credit\_risk.  ✅ FROM credit\_risk  * Sumber datanya adalah tabel bernama credit\_risk, yang kamu sebelumnya **isi dari DataFrame df ke SQLite** dengan:   python  SalinEdit  df.to\_sql('credit\_risk', conn, index=False, if\_exists='replace') ✅ WHERE loan\_status = 1  * Hanya ambil **baris dengan loan\_status = 1**.   Biasanya:   * loan\_status = 1 artinya **lancar (non-default)**. * loan\_status = 0 artinya **macet/gagal bayar (default)**.   (Perlu kamu cek dokumentasi dataset kamu untuk pastinya.) ✅ ORDER BY cb\_person\_cred\_hist\_length DESC  * Urutkan hasilnya berdasarkan kolom cb\_person\_cred\_hist\_length (**lama riwayat kredit menurut biro kredit**) dari yang **terbesar ke terkecil**.  ✅ LIMIT 10  * Ambil **10 baris teratas saja** dari hasil setelah diurutkan.  ✅ Output akhir: result akan jadi DataFrame pandas berisi 10 orang **yang punya status pinjaman lancar**, dengan **riwayat kredit terpanjang**, lengkap semua kolomnya. ✨ Tujuan kode ini: Melihat **10 peminjam terbaik** berdasarkan:   * Pinjaman **tidak bermasalah** * Memiliki **riwayat kredit paling panjang** |

|  |
| --- |
| query = """ SELECT \* FROM ( SELECT \*, ROW\_NUMBER() OVER (PARTITION BY loan\_intent ORDER BY loan\_amnt DESC) AS rank FROM credit\_risk) ranked WHERE rank<=5; """  # Run SQL query result = pd.read\_sql\_query(query, conn) print(result) |
| | **Bagian** | **Fungsi** | | --- | --- | | ROW\_NUMBER() | Memberikan nomor urut **1, 2, 3, ...** ke tiap baris | | OVER (PARTITION BY loan\_intent ...) | Membagi data berdasarkan **tiap loan\_intent** (misal: EDUCATION, MEDICAL, PERSONAL...) | | ORDER BY loan\_amnt DESC | Dalam setiap kategori loan\_intent, data diurutkan dari pinjaman paling besar ke paling kecil | | AS rank | Nama kolom hasil nomor urut disebut rank |  🧠 Kegunaan Analisis Ini:  * Menemukan **nasabah dengan kebutuhan dana tertinggi** per kategori pinjaman * Bisa membantu analisis segmentasi atau target marketing untuk produk kredit |

|  |
| --- |
| query = """ SELECT \* FROM credit\_risk WHERE loan\_amnt> (  SELECT AVG(loan\_amnt)  FROM credit\_risk  WHERE loan\_status = 1); """  # Run SQL query result = pd.read\_sql\_query(query, conn) print(result) |
| 🧠 **Tujuan Query Ini** Menampilkan **semua data peminjam yang pinjamannya (loan\_amnt) lebih besar dari rata-rata pinjaman peminjam yang lancar (loan\_status = 1)**. 🔍 Penjelasan Kodenya: sql  SalinEdit  SELECT \*  FROM credit\_risk  WHERE loan\_amnt > (  SELECT AVG(loan\_amnt)  FROM credit\_risk  WHERE loan\_status = 1  ); ✳️ Bagian dalam: Subquery sql  SalinEdit  SELECT AVG(loan\_amnt)  FROM credit\_risk  WHERE loan\_status = 1  → Menghitung **rata-rata jumlah pinjaman** dari semua baris yang punya loan\_status = 1 (artinya **peminjam yang lancar / tidak gagal bayar**).  Misalnya hasilnya adalah 9000. ✳️ Bagian luar: sql  SalinEdit  SELECT \*  FROM credit\_risk  WHERE loan\_amnt > 9000  → Ambil **semua baris di mana loan\_amnt lebih besar dari rata-rata peminjam lancar** tadi. ✅ Jadi outputnya adalah: Semua peminjam (baik yang lancar maupun default) **yang meminjam lebih besar dari rata-rata peminjam yang lancar**. ✨ Tujuan Analisis Ini:  | **Tujuan** | **Penjelasan** | | --- | --- | | 📊 Menganalisis risiko | Kamu bisa lihat apakah **semakin besar pinjaman**, risiko gagal bayar meningkat | | 🎯 Segmentasi | Mungkin kamu ingin tahu siapa saja yang "lebih besar dari rata-rata", sebagai kelompok berisiko tinggi | | 🧠 Data understanding | Menjawab pertanyaan seperti: "Apakah peminjam besar punya kemungkinan gagal bayar lebih tinggi?" | |

|  |
| --- |
| query = """ SELECT  COUNT(loan\_percent\_income) AS "Total Count",  MIN(loan\_percent\_income) AS "Minimum Rate",  MAX(loan\_percent\_income) AS "Maximum Rate",  AVG(loan\_percent\_income) AS "Average Rate",  sqrt(AVG(loan\_percent\_income \* loan\_percent\_income) - AVG(loan\_percent\_income) \* AVG(loan\_percent\_income)) AS Standard\_Deviation FROM credit\_risk; """  # Run SQL query result = pd.read\_sql\_query(query, conn) print(result) |
| 📌 Apa itu loan\_percent\_income? Itu biasanya berarti:  Persentase jumlah pinjaman (loan\_amnt) dibandingkan dengan pendapatan tahunan peminjam (person\_income), alias:  text  SalinEdit  loan\_percent\_income = loan\_amnt / person\_income  Jadi ini mengukur **seberapa berat beban pinjaman terhadap pendapatan** seseorang. 🔍 Penjelasan Isi Query: sql  SalinEdit  SELECT  COUNT(loan\_percent\_income) AS "Total Count",  MIN(loan\_percent\_income) AS "Minimum Rate",  MAX(loan\_percent\_income) AS "Maximum Rate",  AVG(loan\_percent\_income) AS "Average Rate",  sqrt(  AVG(loan\_percent\_income \* loan\_percent\_income)  - AVG(loan\_percent\_income) \* AVG(loan\_percent\_income)  ) AS Standard\_Deviation  FROM credit\_risk; 📊 1. COUNT(loan\_percent\_income) → "Total Count"  * Menghitung berapa banyak baris yang memiliki nilai loan\_percent\_income.  🔽 2. MIN(loan\_percent\_income) → "Minimum Rate"  * Nilai **terkecil** dari persentase pinjaman terhadap pendapatan.  🔼 3. MAX(loan\_percent\_income) → "Maximum Rate"  * Nilai **terbesar** dari rasio pinjaman terhadap pendapatan (bisa jadi 1.5 atau bahkan lebih kalau pinjamannya melebihi pendapatan).  📉 4. AVG(loan\_percent\_income) → "Average Rate"  * Rata-rata rasio pinjaman terhadap pendapatan semua peminjam.  🧮 5. Standard Deviation: sql  SalinEdit  sqrt(AVG(x^2) - (AVG(x))^2)   * Ini adalah **rumus standar deviasi populasi** manual dalam SQL, karena SQL belum punya fungsi built-in STDDEV() di SQLite. * Ini mengukur **seberapa menyebar nilai loan\_percent\_income** dari rata-ratanya |

|  |
| --- |
| from matplotlib import pyplot as plt import numpy as np %matplotlib inline import missingno as msno  msno.matrix(df) |
| 📦 1. from matplotlib import pyplot as plt  * Import library **Matplotlib**, khusus bagian pyplot untuk membuat grafik dan visualisasi.  📦 2. import numpy as np  * Import **NumPy**, library untuk operasi matematika, array, dll.  ✅ 3. %matplotlib inline  * Ini hanya dipakai di Jupyter Notebook / Google Colab. * Tujuannya: **menampilkan grafik langsung di dalam notebook**, bukan di jendela terpisah.  📦 4. import missingno as msno  * Import library **Missingno** — dipakai untuk **visualisasi missing value** di dataframe Pandas.  🔍 5. msno.matrix(df)  * Ini membuat **visualisasi struktur missing value** di dataset df (Pandas DataFrame). * Hasilnya: sebuah **plot seperti barcode** di mana:   + Garis **putih** = data **hilang** (NaN)   + Garis **abu-abu / hitam** = data tersedia  💡 Tujuan: Melihat **kolom mana** yang banyak data hilangnya (NaN), dan bagaimana sebarannya — sangat membantu sebelum data dibersihkan atau diproses. Contoh Output: kotlin  SalinEdit  |█|█|█|█|█|█|█|█|█|█|█|█|█| ← data lengkap  |█|█|█| |█|█|█| |█|█|█|█| ← beberapa missing value 🎯 Kapan Ini Dipakai?  * Saat **data preprocessing** * Saat kamu ingin tahu **kolom apa yang harus dibersihkan, di-drop, atau diisi (imputasi)** |

|  |
| --- |
| df.info() |
| Perintah df.info() dalam Python (pada library **Pandas**) digunakan untuk **menampilkan ringkasan struktur DataFrame**. Ini sangat berguna di tahap awal eksplorasi data (EDA). 📋 Output df.info() Biasanya Seperti Ini: pgsql  SalinEdit  <class 'pandas.core.frame.DataFrame'>  RangeIndex: 32581 entries, 0 to 32580  Data columns (total 13 columns):  # Column Non-Null Count Dtype  --- ------ -------------- -----  0 person\_age 32581 non-null int64  1 person\_income 32581 non-null int64  2 person\_home\_ownership 32581 non-null object  3 person\_emp\_length 30990 non-null float64  4 loan\_intent 32581 non-null object  5 loan\_grade 32581 non-null object  6 loan\_amnt 32581 non-null int64  7 loan\_int\_rate 26535 non-null float64  8 loan\_status 32581 non-null int64  9 loan\_percent\_income 32581 non-null float64  10 cb\_person\_default\_on\_file 32581 non-null object  11 cb\_person\_cred\_hist\_length 32581 non-null int64  dtypes: float64(3), int64(5), object(5)  memory usage: 3.2+ MB 🔍 Penjelasan Tiap Bagian:  | **Bagian** | **Penjelasan** | | --- | --- | | RangeIndex | Total jumlah baris, seperti jumlah data keseluruhan | | Column | Nama kolom | | Non-Null Count | Jumlah data yang **tidak kosong (tidak NaN)** | | Dtype | Tipe data: int64, float64, object (untuk string/kategori) |  🎯 Tujuan df.info():  | **Tujuan** | **Penjelasan** | | --- | --- | | 📊 Cek missing values | Melihat apakah ada kolom dengan nilai kosong (Non-Null Count < Total) | | 📈 Cek tipe data | Pastikan tipe data sudah benar sebelum modeling (misalnya: kategorikal harus object) | | 🔄 Cek ukuran dataset | Total baris dan kolom dataset | | 🧹 Untuk preprocessing | Menentukan apakah perlu konversi tipe data, imputasi, atau drop kolom tertentu | |

|  |
| --- |
| df.describe() |
| Perintah df.describe() di **Pandas** digunakan untuk menghasilkan **statistik deskriptif ringkasan** untuk **kolom numerik** dalam sebuah DataFrame.  🎯 Tujuannya:  | **Tujuan** | **Penjelasan** | | --- | --- | | 📈 Lihat distribusi awal data | Misalnya tahu apakah loan\_amnt terlalu besar atau kecil | | 🕵️‍♀️ Deteksi outlier | Misalnya person\_income maksimum bisa 600.000 padahal median cuma 57.000, artinya ada outlier | | 🧹 Bantu scaling | Informasi ini berguna untuk **normalisasi atau standardisasi** sebelum training model | | 📉 Data cleaning | Bisa jadi kamu lihat kolom punya nilai min = 0 padahal tidak masuk akal (contohnya: umur 0 tahun) | |

|  |
| --- |
| df = df.dropna(subset=['person\_emp\_length'])  df['loan\_int\_rate'] = df['loan\_int\_rate'].fillna(  df.groupby('loan\_grade')['loan\_int\_rate'].transform('median') ) |
| Kode tersebut adalah bagian dari **data preprocessing**, lebih spesifiknya untuk **menangani missing value (data yang hilang)** di kolom person\_emp\_length dan loan\_int\_rate.  Mari kita bahas **baris per baris**: ✅ df = df.dropna(subset=['person\_emp\_length']) 📌 **Artinya:** Hapus baris-baris dalam df yang memiliki nilai **NaN (kosong)** di kolom person\_emp\_length.  🧠 **Kenapa perlu?** Karena jika person\_emp\_length dianggap penting untuk model prediksi, maka baris dengan nilai kosong tidak bisa digunakan dan perlu dibuang. ✅ df['loan\_int\_rate'] = df['loan\_int\_rate'].fillna(df.groupby('loan\_grade')['loan\_int\_rate'].transform('median')) 📌 **Artinya:** Isi nilai yang kosong (NaN) pada kolom loan\_int\_rate dengan **median bunga pinjaman berdasarkan loan\_grade masing-masing**. Prosesnya:  1. Data dikelompokkan berdasarkan loan\_grade (groupby('loan\_grade')) 2. Dihitung nilai **median** untuk loan\_int\_rate di setiap grade 3. Nilai kosong diisi sesuai **median grade-nya masing-masing**  🧠 Kenapa Pakai Median Berdasarkan Grade?  * Karena **interest rate (loan\_int\_rate) biasanya tergantung dari loan\_grade** (grade tinggi = bunga rendah, dan sebaliknya). * Mengisi NaN dengan rata-rata seluruh dataset bisa **menyimpang** dari kenyataan tiap kelompok. * **Median** dipilih karena lebih tahan terhadap **outlier** dibanding rata-rata.  🎯 Tujuan Umum:  | **Tujuan** | **Penjelasan** | | --- | --- | | ✅ Bersihkan data | Hapus/memperbaiki data yang tidak lengkap | | 📈 Siapkan data untuk modeling | Model machine learning tidak bisa jalan kalau ada NaN | | 🧠 Gunakan informasi yang ada | Mengisi missing value dengan cara cerdas (group median) agar tidak kehilangan pola penting | |

|  |
| --- |
| # Columns you want to encode categorical\_cols = ['person\_home\_ownership', 'loan\_intent', 'loan\_grade', 'cb\_person\_default\_on\_file']  # Create dummies df\_dummies = pd.get\_dummies(df[categorical\_cols], drop\_first=True)   print("Shape of the DataFrame after dummies:", df\_dummies.shape)  print(df.index.equals(df\_dummies.index))  # Drop original categorical columns and add dummies df = pd.concat([df.drop(columns=categorical\_cols), df\_dummies], axis=1)  print("Shape of the DataFrame after concatenation:", df.shape) |
| Shape of the DataFrame after dummies: (31686, 15) True Shape of the DataFrame after concatenation: (31686, 23) |
| 🎯 Tujuan kode ini: **Mengubah kolom kategorikal (berisi teks) menjadi bentuk numerik** → agar bisa dipakai untuk **Machine Learning**. 💡 Kenapa perlu diubah? Model ML seperti **Logistic Regression, Random Forest, XGBoost** tidak bisa langsung memproses string seperti "OWN" atau "MORTGAGE" → Jadi, kita ubah teks jadi angka pakai teknik **One-Hot Encoding** (get\_dummies()) 🔍 Penjelasan kode baris per baris:✅ 1. Tentukan kolom kategorikal yang mau diubah python  SalinEdit  categorical\_cols = ['person\_home\_ownership', 'loan\_intent', 'loan\_grade', 'cb\_person\_default\_on\_file']  Ini adalah kolom yang berisi kategori teks. ✅ 2. Buat dummy variables (one-hot encoding) python  SalinEdit  df\_dummies = pd.get\_dummies(df[categorical\_cols], drop\_first=True)   * Contoh:   python  SalinEdit  pd.get\_dummies(df['person\_home\_ownership'], drop\_first=True)  kalau isinya: ["RENT", "OWN", "MORTGAGE"], maka hasilnya:  nginx  SalinEdit  OWN MORTGAGE  0 1  1 0  0 0   * drop\_first=True artinya **hindari dummy trap** (hapus 1 kategori untuk mencegah multikolinearitas)  ✅ 3. Pastikan index-nya cocok python  SalinEdit  print(df.index.equals(df\_dummies.index)) # Hasilnya True  Ini memastikan data bisa **digabung** dengan aman karena baris-baris masih sejajar. ✅ 4. Gabungkan hasilnya ke DataFrame utama python  SalinEdit  df = pd.concat([df.drop(columns=categorical\_cols), df\_dummies], axis=1)   * Hapus kolom aslinya (categorical\_cols) * Tambahkan kolom hasil encoding (df\_dummies) * Hasil akhir = dataframe numerik yang bisa langsung dipakai ke model  🧠 Output: text  SalinEdit  Shape of the DataFrame after dummies: (31686, 15)  True  Shape of the DataFrame after concatenation: (31686, 23)  Artinya:   * Sebelumnya kamu punya 4 kolom kategorikal * Setelah get\_dummies, muncul 15 kolom dummy baru * Data sekarang total punya **23 kolom numerik** yang siap dilatih ke model |

|  |
| --- |
| from sklearn.preprocessing import StandardScaler # Columns you want to scale numerical\_cols = ['person\_age', 'person\_income', 'person\_emp\_length', 'loan\_amnt', 'loan\_int\_rate', 'loan\_percent\_income', 'cb\_person\_cred\_hist\_length']  # Standardize selected numerical columns scaler = StandardScaler() df\_scaled\_part = pd.DataFrame(scaler.fit\_transform(df[numerical\_cols]), columns=numerical\_cols, index=df.index)  print("Shape of the DataFrame after scaling:", df\_scaled\_part.shape) print(df.index.equals(df\_scaled\_part.index))  # Drop original numerical columns and add scaled ones df = pd.concat([df.drop(columns=numerical\_cols), df\_scaled\_part], axis=1)  print("Shape of the DataFrame after concatenation:", df.shape) |
| Shape of the DataFrame after scaling: (31686, 7) True Shape of the DataFrame after concatenation: (31686, 23) |
| 🧠 **Tujuan kode ini:**👉 Menstandarkan (scaling) ****fitur numerik**** agar semua berada di ****skala yang sama****, yaitu: **Mean = 0** dan **Standard Deviation = 1** 🔧 Penjelasan baris per baris:✅ 1. Pilih kolom numerik: python  SalinEdit  numerical\_cols = ['person\_age', 'person\_income', ..., 'cb\_person\_cred\_hist\_length']  Ini kolom-kolom yang berisi **angka dengan skala berbeda-beda**:   * person\_income bisa ratusan ribu * loan\_percent\_income kecil (< 1) * person\_age puluhan  ✅ 2. Gunakan StandardScaler: python  SalinEdit  scaler = StandardScaler()  df\_scaled\_part = pd.DataFrame(  scaler.fit\_transform(df[numerical\_cols]),  columns=numerical\_cols,  index=df.index  )   * fit\_transform() menghitung **mean** dan **std**, lalu mengubah data menjadi:   z=x−meanstd\_devz = \frac{x - \text{mean}}{\text{std\\_dev}}z=std\_devx−mean​   * Contoh hasil:   makefile  SalinEdit  Sebelum: person\_income = 80,000  Setelah: person\_income = 0.45 ✅ 3. Cek index dan gabungkan kembali python  SalinEdit  df = pd.concat([df.drop(columns=numerical\_cols), df\_scaled\_part], axis=1)   * Hapus kolom original (yang belum distandardize) * Tambahkan versi yang sudah discale  ✅ Hasil akhir:  * Sebelumnya df punya 23 kolom * Setelah ganti angka mentah dengan versi distandardisasi → **jumlah kolom tetap**, tapi nilainya sudah **siap untuk model**  ❓ Kenapa harus di-scale? Karena banyak **algoritma ML sensitif terhadap skala**, misalnya:   | **Algoritma** | **Butuh scaling?** | | --- | --- | | ✅ Logistic Regression | **Ya** | | ✅ KNN | **Ya** | | ✅ SVM | **Ya** | | ❌ Tree-based (Random Forest, XGBoost) | **Tidak wajib**, tapi kadang membantu |  🧠 Kesimpulan:  | **Langkah** | **Tujuan** | | --- | --- | | StandardScaler() | Menyamakan skala semua fitur numerik | | .fit\_transform() | Ubah data jadi mean = 0, std = 1 | | concat kembali | Gabungkan ke dataframe utama | |

|  |
| --- |
| df.replace({True: 1, False: 0}, inplace=True) |
| 🎯 Tujuannya: Mengubah nilai **boolean True/False menjadi angka 1/0** di seluruh DataFrame df. 🧠 Kenapa dilakukan? Agar semua data **berformat numerik** — penting untuk **model machine learning**, karena:   * Model **tidak bisa membaca boolean (True/False) langsung** * Model hanya bisa baca angka (0/1)   **✅ Kesimpulan:**   | **Hal** | **Tujuan** | | --- | --- | | replace({True: 1, False: 0}) | Ubah semua boolean jadi numerik | | inplace=True | Langsung ubah di DataFrame aslinya | |

|  |
| --- |
| df['loan\_status'].value\_counts() |
| loan\_status 0 24860 1 6826 Name: count, dtype: int64 |
| ✅ Fungsinya: Menampilkan **jumlah (count)** dari masing-masing nilai unik dalam kolom loan\_status.  **🔍 Artinya:**   | **loan\_status** | **Jumlah** | **Keterangan** | | --- | --- | --- | | 0 | 25400 | Aman (tidak gagal bayar / non-default) | | 1 | 6286 | **Berisiko (gagal bayar / default)** | |

|  |
| --- |
| from sklearn.model\_selection import train\_test\_split  # Input and Output Variable X = df.drop(columns=['loan\_status']) y = df['loan\_status'] # Target variable (1 = default, 0 = no default)  # Train-Test Split X\_train, X\_test, y\_train, y\_test = train\_test\_split(X, y, test\_size=0.3, random\_state=21) |
| 🧠 Tujuan utama: Membagi data menjadi **data latih (train)** dan **data uji (test)** agar kita bisa:   * Melatih model dengan data latih * Menguji akurasi model dengan data yang belum pernah dilihat (data uji)  🔍 Penjelasan kode:1. Pisahkan fitur (X) dan target (y) python  SalinEdit  X = df.drop(columns=['loan\_status'])  y = df['loan\_status']   * X = semua kolom **kecuali** loan\_status, yaitu input/fitur * y = kolom loan\_status, yaitu target/label yang ingin diprediksi (0 = aman, 1 = default)  2. Bagi data menjadi train dan test python  SalinEdit  X\_train, X\_test, y\_train, y\_test = train\_test\_split(  X, y, test\_size=0.3, random\_state=21  )   | **Parameter** | **Fungsi** | | --- | --- | | test\_size=0.3 | 30% data untuk **pengujian**, 70% untuk **pelatihan** | | random\_state=21 | Agar pembagian data **konsisten** setiap kali dijalankan | | Hasil: |  | | X\_train, y\_train | dipakai untuk **melatih model** | | X\_test, y\_test | dipakai untuk **menguji performa model** | |
| maksudnya apa sih random state 21 dah 🧠 Apa itu random\_state=21?Bayangkan kamu:  * Punya setumpuk kartu * Kamu mau bagi acak 70% untuk pelatihan, 30% untuk pengujian * Setiap kali kamu kocok, hasilnya **beda-beda**   Nah, random\_state=21 itu seperti **menyimpan cara ngocok yang ke-21**. Kalau kamu pakai angka itu lagi, kamu **ngocok kartu dengan cara yang sama persis** lagi. 🎯 Kenapa butuh itu? Agar:   * **Hasil pembagian data (train-test)** tetap **sama setiap kali dijalankan** * Bisa **diulang dan diuji kembali** modelnya dengan hasil yang adil  🔢 Kenapa angka 21? **Angkanya bebas.**   * Bisa 1, 7, 42, 123, berapa saja * Tapi kalau kamu pakai angka **yang sama**, maka hasil pembagian data juga akan **selalu sama**  ✅ Intinya:  | **Tanpa random\_state** | **Dengan random\_state=21** | | --- | --- | | Setiap run hasil beda | Hasil selalu sama | | Sulit diulang | Bisa diulang dengan hasil yang sama | | Tak stabil buat eksperimen | Stabil & adil | |

|  |
| --- |
| from sklearn.utils import resample  df\_train = pd.concat([X\_train, y\_train], axis=1)  # Separate majority and minority classes default\_0 = df\_train[df\_train['loan\_status'] == 0] default\_1 = df\_train[df\_train['loan\_status'] == 1]  # Upsample minority class default\_1\_upsampled = resample(  default\_1,  replace=True, # Sample with replacement  n\_samples=len(default\_0), # Match number of majority class  random\_state=25 # For reproducibility )  # Combine majority and upsampled minority df\_train\_oversampled = pd.concat([default\_0, default\_1\_upsampled])  # Shuffle the dataset df\_train\_oversampled = df\_train\_oversampled.sample(frac=1, random\_state=25).reset\_index(drop=True)  # Check the class distribution print(df\_train\_oversampled['loan\_status'].value\_counts()) |
| Kode ini digunakan untuk **menangani ketidakseimbangan kelas (imbalanced class)** dalam data pelatihan model machine learning. 🧠 Masalahnya apa sih? Dalam kasus credit risk:   * Kolom loan\_status adalah **target**:   + 0 = aman (non-default)   + 1 = berisiko (default) * Biasanya, jumlah data yang loan\_status = 0 **jauh lebih banyak** daripada yang = 1.   ➡️ Kalau model dilatih dengan data tidak seimbang, dia bisa **cenderung selalu menebak "0"** dan tetap terlihat "akurat". ✅ Tujuan kode ini: **Membuat data latih menjadi seimbang** jumlahnya antara 0 dan 1, agar model tidak berat sebelah. 🔍 Penjelasan kode:1. Gabungkan X\_train dan y\_train: python  SalinEdit  df\_train = pd.concat([X\_train, y\_train], axis=1) 2. Pisahkan data berdasarkan kelas: python  SalinEdit  default\_0 = df\_train[df\_train['loan\_status'] == 0] # Mayoritas  default\_1 = df\_train[df\_train['loan\_status'] == 1] # Minoritas 3. ****Upsample**** data minoritas (loan\_status = 1): python  SalinEdit  default\_1\_upsampled = resample(  default\_1,  replace=True,  n\_samples=len(default\_0), # Samakan jumlahnya dengan yang mayoritas  random\_state=25  )  📌 Artinya:   * Data default (1) digandakan secara acak sampai jumlahnya sama dengan non-default (0) * replace=True artinya boleh sampling ganda (baris yang sama bisa dipilih lebih dari sekali)  4. Gabungkan kembali: python  SalinEdit  df\_train\_oversampled = pd.concat([default\_0, default\_1\_upsampled]) 5. Acak ulang (shuffle): python  SalinEdit  df\_train\_oversampled = df\_train\_oversampled.sample(frac=1, random\_state=25).reset\_index(drop=True) 6. Lihat hasil distribusinya: python  SalinEdit  print(df\_train\_oversampled['loan\_status'].value\_counts())  ➡️ Harusnya hasilnya **seimbang**, misalnya:  SalinEdit  0 15400  1 15400 🧠 Kenapa ini penting? Tanpa penyeimbangan:   * Model mungkin cuma nebak semua = 0 → tetap 95% akurasi (kalau 95% data-nya 0) * Tapi **prediksi default yang penting malah diabaikan**   Dengan **oversampling**:   * Model bisa belajar ciri-ciri data default dengan lebih baik |

|  |
| --- |
| df\_train\_oversampled.head() |
| Itu artinya kamu ingin melihat **5 baris pertama** dari DataFrame hasil oversampling, yaitu df\_train\_oversampled. |

|  |
| --- |
| X\_train = df\_train\_oversampled.drop(columns=['loan\_status']) y\_train = df\_train\_oversampled['loan\_status'] # Target variable (1 = default, 0 = no default) |
| Bagus! Kode ini adalah **langkah penting** untuk **mempersiapkan data latih (X\_train dan y\_train)** setelah dilakukan oversampling. 🔍 Penjelasan:1. Pisahkan fitur (X) dan target (y) python  SalinEdit  X\_train = df\_train\_oversampled.drop(columns=['loan\_status'])   * **X\_train** = semua kolom kecuali loan\_status * Ini adalah input/fitur untuk model machine learning   python  SalinEdit  y\_train = df\_train\_oversampled['loan\_status']   * **y\_train** = kolom target: loan\_status * Ini yang ingin diprediksi model  🔄 Kenapa ini perlu diulang? Sebelum oversampling, kamu memang sudah punya X\_train dan y\_train, **tapi itu dari data yang tidak seimbang**.  Setelah df\_train\_oversampled dibuat (seimbang), kamu perlu **membuat ulang** X\_train dan y\_train dari data yang baru ini. |

|  |
| --- |
| Model Training Using LogisticRegression  from sklearn.linear\_model import LogisticRegression from sklearn.metrics import accuracy\_score import pickle  # Train the PD model model = LogisticRegression() model.fit(X\_train, y\_train)  # Predict y\_pred = model.predict(X\_test)  # Evaluate accuracy = accuracy\_score(y\_test, y\_pred) print(f"Accuracy: {accuracy:.2f}") |
| Accuracy: 0.81 |
| Kode yang kamu jalankan ini adalah **pipeline pelatihan model Logistic Regression** untuk memprediksi apakah peminjam akan default (gagal bayar) atau tidak. 🔍 Penjelasan langkah per langkah:1. **Import library** python  SalinEdit  from sklearn.linear\_model import LogisticRegression  from sklearn.metrics import accuracy\_score  import pickle   * LogisticRegression: Algoritma klasifikasi * accuracy\_score: Mengukur akurasi prediksi * pickle: Untuk menyimpan model (opsional)  2. **Latih model** python  SalinEdit  model = LogisticRegression()  model.fit(X\_train, y\_train)   * X\_train: Data latih (fitur) * y\_train: Label target (0 = aman, 1 = default) * Model dilatih untuk belajar hubungan antara fitur dan status pinjaman  3. **Prediksi** python  SalinEdit  y\_pred = model.predict(X\_test)   * X\_test: Data uji (belum pernah dilihat model) * y\_pred: Hasil prediksi loan\_status oleh model  4. **Evaluasi** python  SalinEdit  accuracy = accuracy\_score(y\_test, y\_pred)  print(f"Accuracy: {accuracy:.2f}")   * **Mengukur seberapa akurat prediksi model dibanding label asli (y\_test)** |

|  |
| --- |
| Model Training using XGBoost  import xgboost as xgb  # Train XGBoost model model = xgb.XGBClassifier(n\_estimators=100, max\_depth=3, learning\_rate=0.1, use\_label\_encoder=False) model.fit(X\_train, y\_train)  # Predict y\_pred = model.predict(X\_test)  # Evaluate accuracy = accuracy\_score(y\_test, y\_pred) print(f"Accuracy: {accuracy:.2f}") |
| Accuracy: 0.89 |
| 💡 Penjelasan Baris per Baris:✅ 1. **Import XGBoost** python  SalinEdit  import xgboost as xgb   * xgboost adalah library untuk algoritma gradient boosting decision tree (GBDT), sangat akurat dan efisien.  ✅ 2. **Inisialisasi model** python  SalinEdit  model = xgb.XGBClassifier(  n\_estimators=100, # Jumlah pohon keputusan  max\_depth=3, # Kedalaman maksimum tiap pohon  learning\_rate=0.1, # Seberapa besar tiap pohon baru mempengaruhi model  use\_label\_encoder=False # Hindari warning versi terbaru  )  Parameter di atas bisa di-tuning untuk hasil lebih optimal. ✅ 3. **Melatih model** python  SalinEdit  model.fit(X\_train, y\_train)   * Belajar dari data latih (X\_train, y\_train)  ✅ 4. **Prediksi data uji** python  SalinEdit  y\_pred = model.predict(X\_test)   * Hasil prediksi terhadap data yang **belum pernah dilihat** model  ✅ 5. **Evaluasi akurasi** python  SalinEdit  accuracy = accuracy\_score(y\_test, y\_pred)  print(f"Accuracy: {accuracy:.2f}")   * Mengukur seberapa banyak prediksi yang benar dibanding data asli y\_test  🔍 Kapan pakai XGBoost?  * Saat model **Logistic Regression terlalu sederhana** * Saat kamu ingin performa **lebih tinggi** * Saat kamu menangani data yang **tidak linear**, atau banyak interaksi antar fitur * XGBoost bisa menangani **missing value**, **class imbalance**, dan **overfitting** lebih baik |
|  |