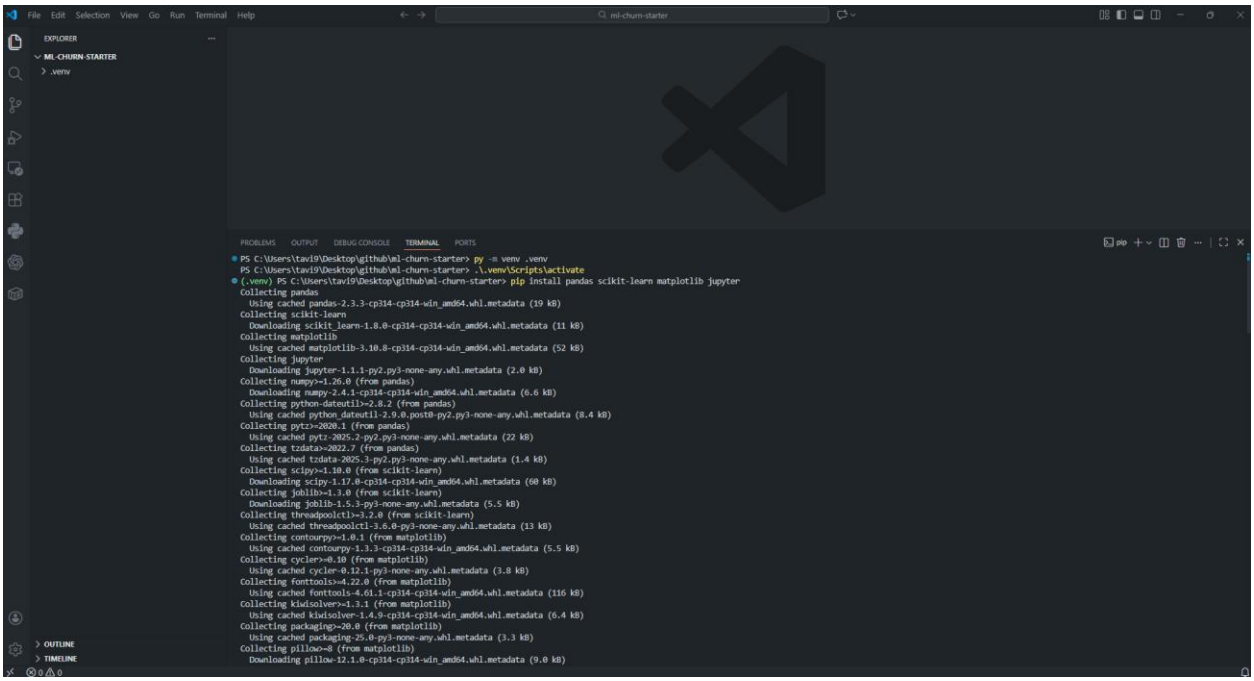
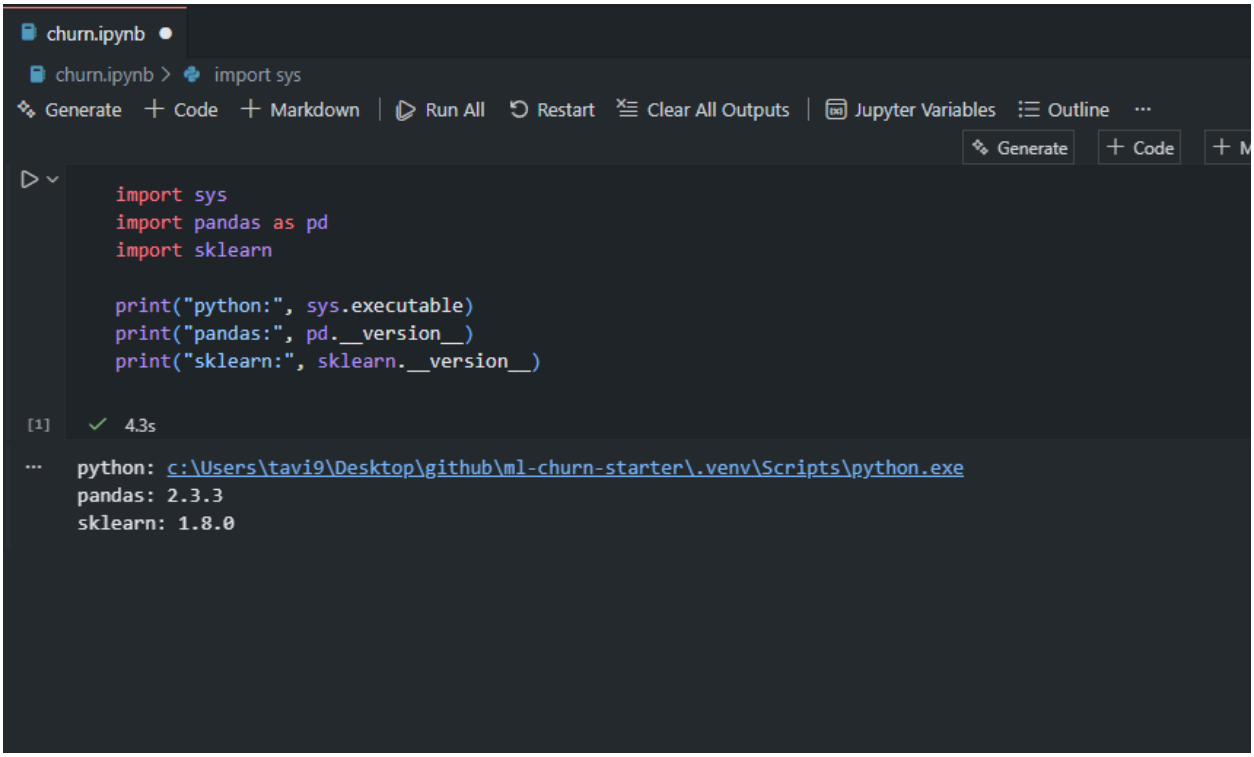
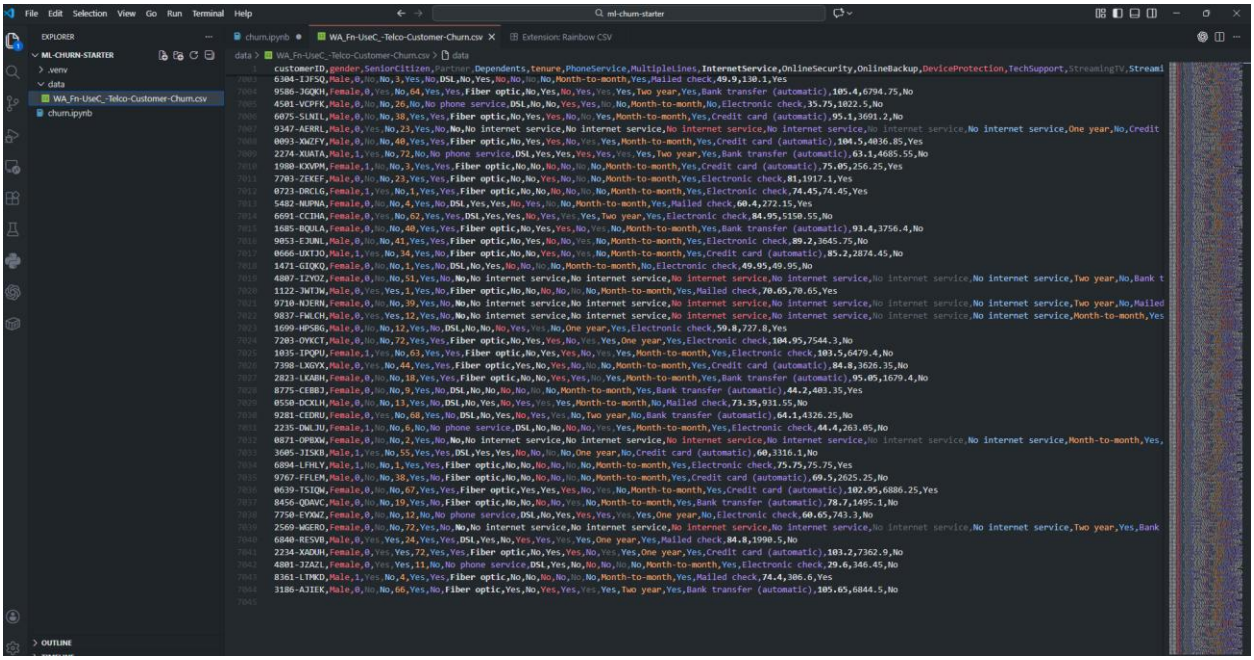


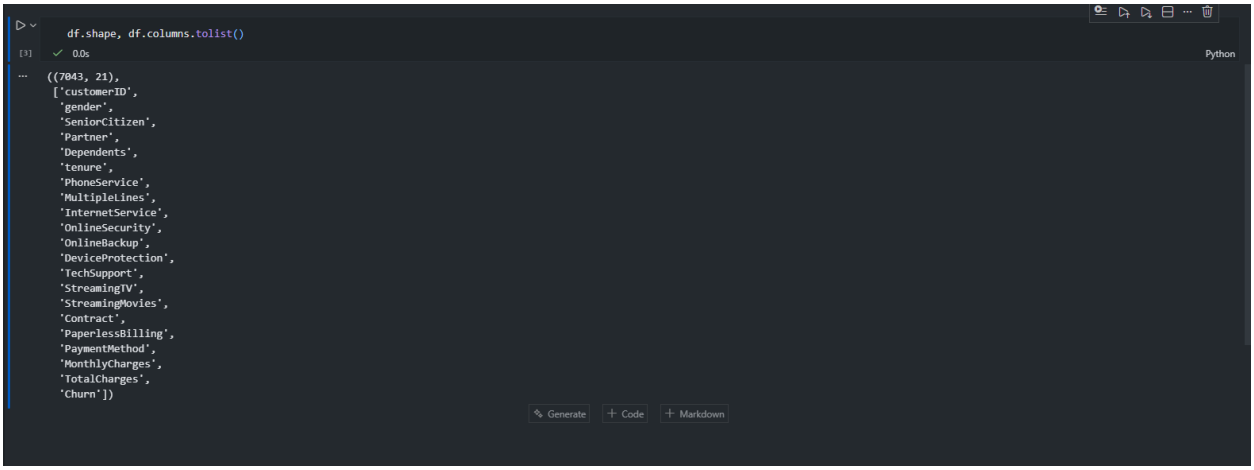
Jupyter -> https://www.youtube.com/watch?v=jZ952vChhuI



Facui cod acum si facui un folder pt date



Apartment sunt multe date incep sa inteleg dc as avea nevoie de un program ca sa analizeze date 😔 👍



7043 linii de date

21 de coloane de tipuri de date

Churn = clientul pleacă.

În dataset-ul tău:

- `Churn = Yes` → clientul **a renunțat** la serviciu
- `Churn = No` → clientul **a rămas**

Deci problema de ML este:

> „Pe baza informațiilor despre client, pot prezice dacă **va pleca** sau **va rămâne?**"

De ce e important churn-ul (în viața reală)

Pentru o firmă:

- e **mai scump** să aduci un client nou decât să-l păstrezi
- dacă știi **din timp** cine are risc mare de churn:
  - îi dai discount
  - îi schimbi planul
  - îl contactezi proactiv

Asta face proiectul **foarte realist și valoros** pentru CV.

În termeni ML (foarte clar)

- **Features (X)**: coloanele gen `tenure`, `Contract`, `MonthlyCharges`, `InternetService`, etc.
- **Label (y)**: `Churn`
- **Tip problemă: Clasificare binară**
  - 2 clase: Yes / No

```
df["Churn"].value_counts()
```

[4] ✓ 0.0s

```
Churn
No     5174
Yes    1869
Name: count, dtype: int64
```

```
x = df.drop(columns=["Churn", "customerID"])
y = df["Churn"]
```

[5] ✓ 0.0s

```
x.shape, y.shape
```

[6] ✓ 0.0s

```
((7043, 19), (7043,))
```

In x am pus tabelul far colenele alea si in y doar coloanal churn

Fara a modifica tabelul initial

```python
from sklearn.model_selection import train_test_split
x_train, x_test, y_train, y_test = train_test_split(
    x, y,
    test_size=0.2,
    random_state=42,
    stratify=y
)

x_train.shape, x_test.shape, y_train.shape, y_test.shape
```

[7] ✓ 0.3s

```
((5634, 19), (1409, 19), (5634,), (1409,))
```

1) Ce sunt `train` și `test`

- **Train set** (`X_train`, `y_train`): datele pe care modelul "învață" (își ajustează parametrii).
- **Test set** (`X_test`, `y_test`): date ținute deoparte, pe care modelul **nu le vede în timpul antrenării**. Le folosim doar la final ca să evaluăm corect.

Dacă ai evalua pe aceleași date pe care ai antrenat, ai obține scoruri prea optimiste (modelul poate "ține minte" detalii).

---

2) Ce face funcția `train_test_split(...)`

Linia asta:

python Copy code

```python
X_train, X_test, y_train, y_test = train_test_split(...)
```

îți împarte aleator rândurile din `X` și `y` în două părți, păstrând alinierea:

- rândul 123 din `X` merge împreună cu eticheta rândului 123 din `y`
- nu se amestecă greșit input-urile cu label-uri diferite

3) Ce înseamnă parametrii (foarte important)

`test_size=0.2`

- 20% din date merg în **test**
- 80% rămân în **train**
  La tine: 7043 rânduri → ~5634 train și ~1409 test (aprox).

`random_state=42`

- fixează "seed"-ul aleator, ca să obții **aceeași împărțire** de fiecare dată.
  Asta e vital pentru reproducibilitate (și pentru proiect/CV).

`stratify=y`

- păstrează **aceeași proporție de Yes/No** în train și în test.
  Pentru că dataset-ul e dezechilibrat (≈73% No / 27% Yes), fără `stratify` ai putea nimeri un test set "stricat" (prea puține Yes), și evaluarea ar deveni înșelătoare.

---

4) Ce verifici cu `shape`

Linia:

```python
X_train.shape, X_test.shape, y_train.shape, y_test.shape
```

doar confirmă că:

- numărul de rânduri se potrivește (train + test = total)
- numărul de coloane din X rămâne identic în train/test
- y are același număr de elemente ca X pe fiecare subset

```python
from sklearn.compose import ColumnTransformer
from sklearn.preprocessing import OneHotEncoder, StandardScaler
from sklearn.impute import SimpleImputer
from sklearn.pipeline import Pipeline

# separam coloanele după tip
num_cols = x.select_dtypes(include=["number"]).columns
cat_cols = x.select_dtypes(exclude=["number"]).columns

numeric_transformer = ColumnTransformer(
    transformers=[
        ("num",
         Pipeline(steps=[
             ("imputer", SimpleImputer(strategy="median")),
             ("scaler", StandardScaler())
         ]), num_cols)
    ],
    remainder="drop"
)

categorical_transformer = Pipeline(steps=[
    ("imputer", SimpleImputer(strategy="most_frequent")),
    ("onehot", OneHotEncoder(handle_unknown="ignore"))
])

preprocessor = ColumnTransformer(
    transformers=[
        ("num", numeric_transformer, num_cols),
        ("cat", categorical_transformer, cat_cols)
    ]
)
```

[9] ✓ 0.2s

1) De ce ai nevoie de preprocesare

Modelele din scikit-learn (Logistic Regression, RandomForest etc.) lucrează bine cu **numere**.
Dar în dataset ai multe coloane categorice (ex. `Contract`, `PaymentMethod`, `InternetService`) care sunt **text**.

În plus, pot exista:

- valori lipsă (missing)
- tipuri "greșite" (ex. `TotalCharges` uneori e text)

Preprocesarea rezolvă asta standardizat.

---

2) `num_cols` și `cat_cols`

python Copy code

```python
num_cols = X.select_dtypes(include=["number"]).columns
cat_cols = X.select_dtypes(exclude=["number"]).columns
```

Aici împarți coloanele în două liste:

- `num_cols`: coloane numerice (ex. `tenure`, `MonthlyCharges`)
- `cat_cols`: coloane text/categorice (ex. `Contract`, `gender`)

E important pentru că fiecare tip de coloană se tratează diferit.

---

3) Ce se întâmplă pe coloanele numerice

În `numeric_transformer` ai:

- **Imputer (median):**

  python Copy code

  ```python
  SimpleImputer(strategy="median")
  ```

  Dacă lipsesc valori, le completează cu **mediana** coloanei.
  (Mediană e robustă la outliers.)
- **Scaler (StandardScaler):**

  python Copy code

  ```python
  StandardScaler()
  ```

  Transformă numerele ca să aibă aproximativ:
  - medie 0
  - deviație standard 1

De ce? Pentru modele precum Logistic Regression, scalarea ajută mult.

---

4) Ce se întâmplă pe coloanele categorice (text)

În `categorical_transformer` ai:

- **Imputer (most_frequent):**
  Completează valorile lipsă cu cea mai frecventă categorie.
- **OneHotEncoder(handle_unknown="ignore"):**
  Transformă o coloană text în mai multe coloane 0/1.

Exemplu simplu:
`Contract` = {Month-to-month, One year, Two year}

devine:

- `Contract_Month-to-month` (0/1)
- `Contract_One year` (0/1)
- `Contract_Two year` (0/1)

`handle_unknown="ignore"` e important: dacă în test apare o categorie nouă, nu crapă.

5) Ce face `ColumnTransformer` (preprocessor)

```python
preprocessor = ColumnTransformer(
    transformers=[
        ("num", numeric_transformer, num_cols),
        ("cat", categorical_transformer, cat_cols)
    ]
)
```

Acesta este "motorul" care:

- aplică pipeline-ul numeric pe `num_cols`
- aplică pipeline-ul categoric pe `cat_cols`
- combină rezultatele într-o singură matrice numerică finală

---

6) Ce ai câștigat practic

Acum ai un obiect (`preprocessor`) pe care îl poți "lipi" direct de un model într-un **Pipeline** complet:

- preprocesare
- antrenare model
- predict

totul corect, fără leakage.

```python
from sklearn.pipeline import Pipeline
from sklearn.linear_model import LogisticRegression
from sklearn.metrics import classification_report, confusion_matrix

clf = Pipeline(steps=[
    ("preprocess", preprocessor),
    ("model", LogisticRegression(max_iter=1000))
])

clf.fit(x_train, y_train)

y_pred = clf.predict(x_test)

print(confusion_matrix(y_test, y_pred))
print(classification_report(y_test, y_pred))
```

[10] ✓ 0.0s

```
[[917 118]
 [172 202]]
              precision    recall  f1-score   support

          No       0.84      0.89      0.86      1035
         Yes       0.63      0.54      0.58       374

    accuracy                           0.79      1409
   macro avg       0.74      0.71      0.72      1409
weighted avg       0.79      0.79      0.79      1409
```

1) Confusion Matrix (prima parte)

Output:

```lua
[[917 118]
 [172 202]]
```

Structura este mereu:

```yaml
            Pred No   Pred Yes
Actual No      TN        FP
Actual Yes     FN        TP
```

Aplicat la tine:

- **TN = 917** → clienți care **NU au plecat** și modelul a zis corect „No”
- **FP = 118** → clienți care **NU au plecat**, dar modelul a zis greșit „Yes”
- **FN = 172** → clienți care **AU plecat**, dar modelul a zis greșit „No” ❗
- **TP = 202** → clienți care **AU plecat** și modelul a zis corect „Yes”

👉 **FN (172)** sunt cei mai periculoși în churn: clienți care pleacă și tu NU îi detectezi.

TN = True Negative

FP = False Positive

FN = False Negative

TP = True Pozitive

2) Precision / Recall / F1 (partea importantă)

Pentru clasa Yes (Churn) — asta te interesează cel mai mult

Metric	Valoare	Ce înseamnă
Precision = 0.63	Din toți clienții marcați ca „pleacă”, **63% chiar pleacă**	
Recall = 0.54	Din toți clienții care pleacă, **54% sunt detectați**	
F1 = 0.58	Compromis între precision și recall	

Interpretare simplă:

- Modelul **prinde cam jumătate** din cei care pleacă
- Când spune “pleacă”, de cele mai multe ori are dreptate

Pentru clasa No

- Precision / Recall mari (0.84 / 0.89)
- Normal, pentru că dataset-ul e dominat de „No”

3) Accuracy (de ce nu e suficient)

ini

Copy code

```ini
accuracy = 0.79
```

Pare bun (79%), **dar**:

- Un model care spune mereu „No” ar avea ~73% accuracy
- Deci îmbunătățirea reală pe churn este modestă, dar reală

👉 De asta ne uităm la **recall pe Yes**, nu doar la accuracy.

---

4) Ce ai făcut, pe scurt (ca explicație de interviu)

> „Am construit un pipeline ML pentru churn prediction folosind Logistic Regression.
> Am gestionat date mixte (numerice + categorice) cu imputare, scalare și one-hot encoding.
> Modelul obține un recall de ~54% pe churn, detectând peste jumătate dintre clienții care pleacă, cu un F1-score de ~0.58.”

Asta sună **corect și profesionist**.

Ce este un pipeline ML (definiție corectă)

Un **pipeline ML** este:

> un **lanț fix de pași**, executați **în aceeași ordine**, care transformă date brute → predicții, **fără scurtături și fără scurgeri de informație (data leakage)**.

La tine, pipeline-ul este acest obiect:

python Copy code

```python
Pipeline(steps=[
    ("preprocess", preprocessor),
    ("model", LogisticRegression(...))
])
```

Ăsta este pipeline-ul. Nu notebook-ul.

Ce face pipeline-ul tău, concret (pas cu pas)

Când scrii:

```python
clf.fit(X_train, y_train)
```

Pipeline-ul face **automat**:

1️⃣ Preprocesare (DOAR pe train)

- împarte coloanele numeric / categoric
- completează valori lipsă
- scalează numerele
- face one-hot encoding la texte

⚠️ **fără să vadă test-ul**

2️⃣ Antrenează modelul

- Logistic Regression învață coeficienții DOAR din datele preprocesate din train

Când scrii:

```python
clf.predict(X_test)
```

Pipeline-ul:

1. aplică **exact aceleași transformări** învățate din train
2. apoi face predicția

👉 Tu **nu refaci nimic manual**. Asta e puterea pipeline-ului.

De ce NU e suficient "am făcut train/test split"

Dacă ai fi făcut manual:

- one-hot pe tot dataset-ul
- scalare pe tot dataset-ul
- apoi split

❌ ar fi fost **greșit** (data leakage).

Pipeline-ul te forțează să faci corect.

Ce este Logistic Regression

Logistic Regression este un **model de clasificare**, nu de regresie (numele e înșelător).

👉 Tu l-ai folosit ca să răspunzi la întrebarea:

> „Care este probabilitatea ca acest client să plece?”

Ce face modelul, intuitiv

Modelul:

1. ia **toate caracteristicile clientului** (tenure, contract, preț, servicii etc.)
2. le combină într-o **sumă ponderată** (fiecare feature are un coeficient)
3. trece rezultatul printr-o funcție specială (sigmoid)
4. obține un număr între **0 și 1** → probabilitatea de churn

Exemplu:

- 0.82 → risc mare de churn
- 0.15 → risc mic

Apoi:

- peste un prag (implicit 0.5) → `Yes`
- sub prag → `No`

Ce înseamnă "a învățat" Logistic Regression

În timpul `fit()`:

- modelul învață **coeficienți** (greutăți) pentru fiecare feature
- acești coeficienți spun:
  - dacă feature-ul **crește** riscul de churn
  - sau îl **scade**
  - și **cât de mult**

Exemplu conceptual:

- `Contract = Month-to-month` → coeficient pozitiv → crește churn
- `tenure mare` → coeficient negativ → scade churn

De ce e Logistic Regression bun ca prim model

✔ simplu
✔ rapid
✔ stabil
✔ ușor de interpretat
✔ foarte folosit în industrie pentru probleme de tip churn

În practică, **foarte multe sisteme reale pornesc cu Logistic Regression.**

Ce NU face Logistic Regression

❌ nu prinde relații foarte complexe (non-lineare)
❌ nu e cel mai "puternic" model
❌ dar e un **baseline excelent**

De asta, orice proiect serios începe cu el.

Ce ai făcut TU concret (tradus simplu)

- ai ales un **model probabilistic**
- ai estimat **riscul de churn**
- ai obținut coeficienți care pot fi interpretați business
- ai un baseline solid

1) Imputare (Imputation)

Problema:

Datele reale au **valori lipsă** (goale).

Exemplu:

- `TotalCharges` = lipsă
- `OnlineSecurity` = lipsă

Un model ML **nu știe ce să facă cu goluri.**

Ce înseamnă imputare

👉 **Înlocuiești valorile lipsă cu ceva calculat.**

La tine:

- **numeric** → cu **mediană**
- **categoric** → cu **cea mai frecventă valoare**

De ce mediană, nu media?

Mediană:

- nu e afectată de valori extreme (outliers)
- e mai stabilă

Exemplu simplu

Coloană `tenure`:

```csharp
[1, 2, 3, ?, 100]
```

- media ≈ 26 (prost)
- **mediana = 3** (bun)

2) Scalare (StandardScaler)

Problema:

Unele numere sunt mici, altele mari.

Exemplu:

- `tenure` → 1–72
- `MonthlyCharges` → 18–120
- `TotalCharges` → 0–9000

Modelul ar putea crede că:

> TotalCharges e „mai important” doar pentru că are valori mai mari

Ce face scalarea

👉 Aduci toate coloanele numerice pe **aceeași scară**.

StandardScaler face:

- medie = 0
- deviație standard = 1

Exemplu:

makefile

Copy code

```
tenure:  72   →  1.2
tenure:   1   → -1.1
```

De ce e important pentru Logistic Regression

- modelul se bazează pe **coeficienți**
- fără scalare, coeficienții devin greu de interpretat și instabili

3) One-Hot Encoding

Problema:

Modelele ML **nu înțeleg text.**

Exemplu:

```ini
Contract = "Month-to-month"
Contract = "One year"
Contract = "Two year"
```

Modelul nu știe ce înseamnă "One year".

Ce face One-Hot Encoding

👉 Transformă textul în **coloane binare (0/1).**

Exemplu:

```css
Contract_Month-to-month  Contract_One_year  Contract_Two_year
1                        0                  0
0                        1                  0
```

De ce NU folosim 1,2,3

Dacă ai face:

```sql
Month-to-month = 1
One year = 2
Two year = 3
```

modelul ar crede greșit că:

> Two year > One year > Month-to-month

One-hot evită această capcană.

Cum lucrează toate împreună (foarte important)

Când rulezi:

```python
clf.fit(X_train, y_train)
```

Pipeline-ul tău face:

1. imputare → completează golurile
2. scalare → normalizează numerele
3. one-hot → transformă textele
4. antrenează Logistic Regression

Totul **automat**, **corect**, **reproductibil**.

Ce este recall (definiție corectă)

Recall răspunde la întrebarea:

> „Dintre toți clienții care **chiar au plecat**, câți au fost **detectați de model**?"

Formula (conceptual):

```ini
recall = TP / (TP + FN)
```

unde:

- **TP** = clienți care au plecat și modelul a zis „Yes"
- **FN** = clienți care au plecat, dar modelul a zis „No" (cei rataţi ❗)

Ce înseamnă recall = 54% pe churn

La tine:

- clienți care AU plecat în setul de test: **374**
- detectați corect de model: **202**
- rataţi de model: **172**

Calcul:

```ini
recall = 202 / (202 + 172) ≈ 0.54
```

Tradus în română simplă

👉 **Din 100 de clienți care chiar pleacă, modelul tău reușește să identifice ~54.**

Restul (~46) pleacă **fără să fie semnalați.**

De ce “recall pe churn” e mai important decât accuracy

În churn:

- **False Negative (FN)** = client pleacă și tu NU știi
  → pierdere de bani
- **False Positive (FP)** = crezi că pleacă, dar nu pleacă
  → cel mult dai un discount degeaba

Deci:

> Este mai grav să **nu vezi** un client care pleacă decât să alarmezi fals.

👉 De asta ne interesează **recall pe clasa “Yes” (churn)**.

De ce 54% e “ok” pentru un prim model

- Logistic Regression = baseline
- date dezechilibrate
- fără tuning

În industrie:

- 50–60% recall pe churn **nu e rău** ca prim pas
- apoi îl îmbunătățești (70%+ e deja foarte bine)

Ce este F1-score (definiție corectă)

F1-score este un **echilibru** între:

- **precision** și
- **recall**

Formula (conceptual):

```ini
F1 = 2 * (precision * recall) / (precision + recall)
```

Nu e media simplă, ci o **medie armonică** → penalizează valorile dezechilibrate.

Ce înseamnă precision și recall, foarte pe scurt

Pentru clasa **Churn = Yes:**

- **Precision:**
  „Când modelul spune *pleacă*, cât de des are dreptate?”
- **Recall:**
  „Din toți cei care *pleacă*, câți sunt detectați?”

Ce înseamnă precision și recall, foarte pe scurt

Pentru clasa **Churn = Yes:**

- **Precision:**
  „Când modelul spune *pleacă*, cât de des are dreptate?"
- **Recall:**
  „Din toți cei care *pleacă*, câți sunt detectați?"

Ce înseamnă F1 ≈ 0.58 la tine

La tine:

- precision ≈ **0.63**
- recall ≈ **0.54**

F1 ≈ **0.58** înseamnă:

👉 Modelul are un **compromis rezonabil:**

- nu alarmează aiurea prea mult
- dar nici nu ratează masiv clienți care pleacă

Este un scor **echilibrat**, nu extrem într-o direcție.

De ce NU folosim doar accuracy

Ai deja:

- accuracy ≈ 79%
- dar churn = doar 27% din date

Accuracy mare poate veni din:

> „zic mereu No"

F1 te forțează să fii bun **exact pe problema grea:** churn.

Cum interpretezi F1 ca inginer

- **F1 mic (<0.4)** → model slab
- **F1 ~0.5–0.6** → baseline solid
- **F1 ~0.7+** → model bun
- **F1 ~0.8+** → foarte bun (rar fără date foarte bune)

👉 Pentru **primul tău proiect ML, 0.58 este chiar bine.**

```python
import pandas as pd

model = clf.named_steps["model"]
feature_names = clf.named_steps["preprocess"].get_feature_names_out()
coefs = model.coef_[0]

coef_df = pd.DataFrame({
    "feature": feature_names,
    "coef": coefs
}).sort_values("coef", ascending=False)

coef_df.head(15)
```

[11] ✓ 0.0s

	feature	coef
1321	cat__TotalCharges_20.15	1.335326
1230	cat__TotalCharges_19.9	1.330952
1328	cat__TotalCharges_20.5	1.311257
1320	cat__TotalCharges_20.1	1.181960
2014	cat__TotalCharges_288.05	1.128285
1224	cat__TotalCharges_19.6	1.123792
4650	cat__TotalCharges_740.3	1.029200
1231	cat__TotalCharges_19.95	1.022892
1334	cat__TotalCharges_20.9	1.022670
3113	cat__TotalCharges_45.7	1.015341
1221	cat__TotalCharges_19.45	1.007067
1322	cat__TotalCharges_20.2	1.005200
125	cat__TotalCharges_1052.35	0.961899
203	cat__TotalCharges_1099.6	0.959884
3281	cat__TotalCharges_4759.75	0.959046

1) Ce reprezintă tabelul pe care îl vezi

Fiecare rând = **un feature real** folosit de model **după preprocesare.**

Coloane:

- **feature** → numele feature-ului (după one-hot)
- **coef** → coeficientul Logistic Regression

Regulă generală:

- **coeficient > 0** → crește șansa de churn
- **coeficient < 0** → scade șansa de churn
- **|coef| mare** → impact puternic

2) De ce vezi `cat_TotalCharges_20.15`, `cat_TotalCharges_740.3`, etc.

Asta e partea IMPORTANTĂ 👇

Ce înseamnă `cat_`

Prefixul `cat_` vine de la **categorical transformer.**

👉 Asta înseamnă că `TotalCharges` **a fost tratată ca variabilă categorică (text)**, nu numerică.

De ce s-a întâmplat asta

În dataset-ul Telco:

- `TotalCharges` **pare numeric**
- dar conține valori lipsă sub formă de string gol `" "`
- pandas o citește ca **object (string)**

Consecință:

- pipeline-ul a pus `TotalCharges` în **categorice**
- one-hot encoding a creat **o coloană pentru fiecare valoare distinctă**
- de aici sute de `cat_TotalCharges_xxx`

👉 Modelul a învățat:
„Dacă TotalCharges = exact 20.15 → churn crește mult"
ceea ce **nu e ce ne dorim conceptual.**

3) Interpretarea coeficienților pe care îi vezi ACUM

Chiar dacă nu e ideal, ce înseamnă acești coeficienți:

Exemplu:

```
cat_TotalCharges_20.15 → coef ≈ 1.33
```

Înseamnă:

> Clienții cu **TotalCharges ≈ 20** au risc mare de churn
> (are sens: sunt clienți noi, tenure mic)

Deci **semnalul e real**, dar reprezentarea e proastă.

---

4) Concluzie importantă (foarte matură ML-wise)

❌ **Nu e o greșeală de cod**
❌ **Nu e o greșeală de model**
✔ Este o **problemă de tip de date**

Și e foarte comună în proiecte reale.

5) Ce ar trebui să vedem IDEAL

După corectare, ne dorim coeficienți gen:

- `num__tenure` (negativ)
- `num__MonthlyCharges` (pozitiv)
- `cat__Contract_Month-to-month` (pozitiv)
- `cat__Contract_Two_year` (negativ)

Adică **factori interpretabili business**, nu valori exacte.

```python
df["TotalCharges"] = pd.to_numeric(df["TotalCharges"], errors="coerce")

df["TotalCharges"].isna().sum(), df["TotalCharges"].dtype
```

[12] ✓ 0.0s

```
(np.int64(11), dtype('float64'))
```

- `TotalCharges` este acum **numeric (float64)**
- ai **11 valori lipsă** (NaN) — foarte puține, și le rezolvăm prin imputare (mediană) în pipeline.

```python
from sklearn.model_selection import train_test_split
from sklearn.compose import ColumnTransformer
from sklearn.preprocessing import OneHotEncoder, StandardScaler
from sklearn.impute import SimpleImputer
from sklearn.pipeline import Pipeline

# refacem x si y DUPA conversia TotalCharges
x = df.drop(columns=["Churn", "customerID"])
y = df["Churn"]

x_train, x_test, y_train, y_test = train_test_split(
    x, y, test_size=0.2, random_state=42, stratify=y
)

num_cols = x.select_dtypes(include=["number"]).columns
cat_cols = x.select_dtypes(exclude=["number"]).columns

numeric_transformer = Pipeline(steps=[
    ("imputer", SimpleImputer(strategy="median")),
    ("scaler", StandardScaler())
])

categorical_transformer = Pipeline(steps=[
    ("imputer", SimpleImputer(strategy="most_frequent")),
    ("onehot", OneHotEncoder(handle_unknown="ignore"))
])

preprocessor = ColumnTransformer(
    transformers=[
        ("num", numeric_transformer, num_cols),
        ("cat", categorical_transformer, cat_cols)
    ]
)

num_cols, cat_cols
```

[13] ✓ 0.0s

```
(Index(['SeniorCitizen', 'tenure', 'MonthlyCharges', 'TotalCharges'], dtype='object'),
 Index(['gender', 'Partner', 'Dependents', 'PhoneService', 'MultipleLines',
        'InternetService', 'OnlineSecurity', 'OnlineBackup', 'DeviceProtection',
        'TechSupport', 'StreamingTV', 'StreamingMovies', 'Contract',
        'PaperlessBilling', 'PaymentMethod'],
       dtype='object'))
```

Coloane numerice (tratate corect):

```css
['SeniorCitizen', 'tenure', 'MonthlyCharges', 'TotalCharges']
```

- ✔ `TotalCharges` este acum numeric
- ✔ va primi **imputare + scalare**
- ✔ coeficienții vor fi **interpretabili**

Coloane categorice:

```bash
['gender', 'Partner', 'Dependents', 'PhoneService', 'MultipleLines',
 'InternetService', 'OnlineSecurity', 'OnlineBackup', 'DeviceProtection',
 'TechSupport', 'StreamingTV', 'StreamingMovies', 'Contract',
 'PaperlessBilling', 'PaymentMethod']
```

- ✔ vor primi **imputare + one-hot encoding**

👉 Asta înseamnă că problema de mai devreme (`cat_TotalCharges_20.15`) este **rezolvată**.

```python
from sklearn.linear_model import LogisticRegression
from sklearn.metrics import classification_report, confusion_matrix

clf = Pipeline(steps=[
    ("preprocess", preprocessor),
    ("model", LogisticRegression(max_iter=1000))
])

clf.fit(x_train, y_train)

y_pred = clf.predict(x_test)

print(confusion_matrix(y_test, y_pred))
print(classification_report(y_test, y_pred))
```

[14] ✓ 0.0s

```
[[926 109]
 [165 209]]
              precision    recall  f1-score   support

          No       0.85      0.89      0.87      1035
         Yes       0.66      0.56      0.60       374

    accuracy                           0.81      1409
   macro avg       0.75      0.73      0.74      1409
weighted avg       0.80      0.81      0.80      1409
```

1) Ce s-a îmbunătățit față de înainte

Metrici pentru Churn = Yes

- **Recall: 0.54 → 0.56**
- **F1-score: 0.58 → 0.60**
- **Precision: 0.63 → 0.66**

👉 Asta înseamnă:

- prin simplul fapt că ai tratat `TotalCharges` corect (numeric),
- modelul **detectează mai mulți clienți care pleacă**,
- și face asta **mai curat** (mai puține alarme false).

Este exact ce ne așteptam. Ai reparat o problemă de date, nu ai „păcălit" modelul.

2) Interpretarea confusion matrix (rapid)

lua

Copy code

```
[[926 109]
 [165 209]]
```

- **TP (Yes corect):** 209 ↑ (era 202)
- **FN (Yes ratat):** 165 ↓ (era 172)

👉 Mai puțini clienți care pleacă sunt rataţi.

Asta este **business value real.**

---

3) Concluzie intermediară (importantă)

Acum ai:

- un **baseline Logistic Regression solid**
- date curate
- metrici corecte
- și un model **interpretabil**

Abia **acum** are sens să răspundem la întrebarea:

> **De ce pleacă clienții?**

```python
import pandas as pd

model = clf.named_steps["model"]
feature_names = clf.named_steps["preprocess"].get_feature_names_out()
coefs = model.coef_[0]

coef_df = pd.DataFrame({
    "feature": feature_names,
    "coef": coefs
}).sort_values("coef", ascending=False)

# Top factori care CRESC churn
coef_df.head(10)
```

[15] ✓ 0.0s

	feature	coef
16	cat__InternetService_Fiber optic	0.640026
36	cat__Contract_Month-to-month	0.579304
3	num__TotalCharges	0.516708
35	cat__StreamingMovies_Yes	0.204014
32	cat__StreamingTV_Yes	0.203626
43	cat__PaymentMethod_Electronic check	0.198412
18	cat__OnlineSecurity_No	0.157023
27	cat__TechSupport_No	0.132115
14	cat__MultipleLines_Yes	0.105954
0	num__SeniorCitizen	0.053929

De ce PLEACĂ clienții (coeficienți pozitivi)

Coeficient pozitiv ⇒ **crește riscul de churn.**

1) `InternetService_Fiber optic` (+0.64)

👉 Clienții cu **Fiber optic** pleacă mai des.

Interpretare business:

- servicii mai scumpe
- așteptări mai mari
- dacă apar probleme → pleacă rapid

---

2) `Contract_Month-to-month` (+0.58)

👉 **Cel mai clasic factor de churn.**

- fără angajament
- pot pleca oricând
- risc mare implicit

---

3) `TotalCharges` (+0.52)

👉 Clienții care au plătit **mult în total** au risc mai mare.

Interpretare:

- au stat suficient timp ca să acumuleze cost
- pot fi nemulțumiți de raport preț/valoare

4) `StreamingTV_Yes` / `StreamingMovies_Yes` (~+0.20)

👉 Clienți cu **servicii extra** pleacă mai des.

Motiv probabil:

- factură mai mare
- servicii alternative (Netflix, etc.)

---

5) `PaymentMethod_Electronic check` (+0.19)

👉 Electronic check = **risc mai mare.**

Este un pattern foarte des întâlnit în churn datasets.

---

6) `OnlineSecurity_No` / `TechSupport_No`

👉 Fără suport / securitate = risc crescut.

---

7) `SeniorCitizen`

👉 Seniorii au risc ușor mai mare (efect mic, dar real).

```python
# Top factori care SCAD churn
coef_df.tail(10)
```

[16] ✓ 0.0s

	feature	coef
19	cat__OnlineSecurity_No internet service	-0.301082
17	cat__InternetService_No	-0.301082
31	cat__StreamingTV_No internet service	-0.301082
25	cat__DeviceProtection_No internet service	-0.301082
22	cat__OnlineBackup_No internet service	-0.301082
39	cat__PaperlessBilling_No	-0.343260
2	num__MonthlyCharges	-0.596392
15	cat__InternetService_DSL	-0.652406
38	cat__Contract_Two year	-0.772261
1	num__tenure	-1.241319

De ce RĂMÂN clienții (coeficienți negativi)

Coeficient negativ ⇒ **scade riscul de churn.**

1) `tenure` (−1.24)

👉 **Cel mai puternic factor de retenție.**

- cu cât clientul stă mai mult
- cu atât șansa să plece scade drastic

---

2) `Contract_Two year` (−0.77)

👉 Contractele lungi **protejează puternic** împotriva churn.

---

3) `InternetService_DSL` (−0.65)

👉 DSL = mai stabil, mai ieftin → mai puțin churn.

---

4) `MonthlyCharges` (−0.60)

👉 Interesant: **facturi lunare mai mici** → clienți mai stabili.

---

5) `PaperlessBilling_No`

👉 Clienții mai „clasici" (fără paperless) sunt mai stabili.

Cum putem crește recall (fără model nou)

Avem 2 pârghii clasice:

1️⃣ `class_weight='balanced'`

Spui modelului:

> „Greșelile pe churn (Yes) sunt mai grave."

Modelul:

- penalizează mai tare FN
- încearcă să prindă mai mulți „Yes"

2️⃣ Scădem pragul de decizie

Implicit:

- probabilitate ≥ 0.5 → Yes

Putem face:

- ≥ 0.4 → Yes
  → recall ↑

```python
from sklearn.linear_model import LogisticRegression
from sklearn.metrics import classification_report, confusion_matrix
from sklearn.pipeline import Pipeline

clf_recall = Pipeline(steps=[
    ("preprocess", preprocessor),
    ("model", LogisticRegression(
        max_iter=1000,
        class_weight="balanced"
    ))
])

clf_recall.fit(x_train, y_train)

y_pred_recall = clf_recall.predict(x_test)

print(confusion_matrix(y_test, y_pred_recall))
print(classification_report(y_test, y_pred_recall))
```

[17] ✓ 0.0s

```
[[747 288]
 [ 81 293]]
              precision    recall  f1-score   support

          No       0.90      0.72      0.80      1035
         Yes       0.50      0.78      0.61       374

    accuracy                           0.74      1409
   macro avg       0.70      0.75      0.71      1409
weighted avg       0.80      0.74      0.75      1409
```

1) Ce s-a schimbat față de modelul anterior

Înainte (model normal):

- Recall (Yes) ≈ 0.56
- Precision (Yes) ≈ 0.66
- F1 (Yes) ≈ 0.60
- Accuracy ≈ 0.81

Acum (`class_weight="balanced"`):

- Recall (Yes) ≈ 0.78 ⬆️ ⬆️ ⬆️
- Precision (Yes) ≈ 0.50 ⬇️
- F1 (Yes) ≈ 0.61 ⬆️ ușor
- Accuracy ≈ 0.74 ⬇️

👉 **Ai câștigat exact ce voiai:**
prinzi mult mai mulți clienți care pleacă.

---

2) Confusion matrix explicată (diferența reală)

lua Copy code

```
[[747 288]
 [ 81 293]]
```

Pentru churn = Yes:

- **TP = 293** (înainte era 209)
- **FN = 81** (înainte era 165)

👉 Ai **salvat încă ~84 de clienți** care înainte plecau neobservați.

Ce ai pierdut:

- **FP = 288** (mai multe alarme false)
- adică mai mulți clienți care NU pleacă, dar îi semnalezi

În churn:

- FP = discount inutil
- FN = client pierdut

👉 FP < FN ca impact.

3) De ce a scăzut accuracy (și nu ne pasă)

Accuracy a scăzut pentru că:

- ai etichetat mai mulți „Yes”
- dataset-ul e dominat de „No”

Dar:

> **Accuracy NU este metricul corect pentru churn.**

Recall + F1 sunt mult mai relevante.

5) Ce variantă alegi în practică

Ai acum **două modele valide:**

Model A – echilibrat

- recall ≈ 56%
- precision mai bună
- potrivit dacă resursele sunt limitate

Model B – orientat pe retenție

- recall ≈ 78%
- prinde majoritatea clienților care pleacă
- potrivit pentru campanii proactive

👉 În churn, **Model B e adesea preferat.**

Ce este threshold (pragul de decizie)

Modelul tău (Logistic Regression) **nu spune direct „Yes / No”.**
El calculează mai întâi o **probabilitate:**

```powershell
P(churn = Yes) = un număr între 0 și 1
```

Exemple:

- 0.12 → risc mic
- 0.47 → risc mediu
- 0.81 → risc mare

Threshold = regula după care decizi eticheta finală

Implicit, scikit-learn folosește:

```yaml
dacă P >= 0.5 → Yes
dacă P <  0.5 → No
```

👉 **0.5 este threshold-ul implicit.**

De ce 0.5 NU e obligatoriu bun

Pragul 0.5 este:

- o **convenție**
- nu o regulă universală

El e bun când:

- clasele sunt echilibrate
- costurile greșelilor sunt egale

❌ **NU este cazul la churn.**

De ce 0.5 NU e obligatoriu bun

Pragul 0.5 este:

- o **convenție**
- nu o regulă universală

El e bun când:

- clasele sunt echilibrate
- costurile greșelilor sunt egale

❌ **NU este cazul la churn.**

Ce legătură are threshold-ul cu recall și precision

Dacă scazi threshold-ul (ex: 0.5 → 0.4):

- mai mulți clienți sunt marcați „Yes"
- **recall** ↑ (prinzi mai mulți care pleacă)
- **precision** ↓ (mai multe alarme false)

Dacă crești threshold-ul (ex: 0.5 → 0.6):

- mai puțini „Yes"
- **precision** ↑
- **recall** ↓

👉 Threshold-ul controlează **trade-off-ul.**

De ce „trebuie” să îl reglezi

Pentru că tu **nu vrei același lucru ca modelul default**.

În churn:

- FN (ratezi un client care pleacă) = **foarte scump**
- FP (alertezi degeaba) = **acceptabil**

Deci vrei:

> **recall mare**, chiar dacă precision scade.

Threshold-ul îți permite asta **fără să schimbi modelul**.

Diferența față de `class_weight`

- `class_weight` → influențează **antrenarea**
- `threshold` → influențează **decizia finală**

De multe ori:

- antrenezi normal
- **reglezi threshold-ul la final** în funcție de business

```python
# probabilitatea pentru clasa "Yes"
proba_yes = clf.predict_proba(x_test)[:, 1]

proba_yes[:10], proba_yes.min(), proba_yes.max()
```

[18] ✓ 0.0s

```
(array([0.04529387, 0.68441126, 0.05872074, 0.40170424, 0.02154772,
        0.60393536, 0.44922498, 0.13005522, 0.00288077, 0.39517899]),
 np.float64(0.0017387470659074764),
 np.float64(0.8554356722033837))
```

1) `clf.predict_proba(x_test)`

Asta cere modelului să returneze **probabilități**, nu clase.

Pentru fiecare client din `x_test`, `predict_proba` întoarce două numere:

- `P(No)`
- `P(Yes)`

Deci rezultatul are formă:

- **(număr_rânduri_test, 2)**
  la tine test-ul are 1409 rânduri → (1409, 2)

Exemplu pentru un client:

```csharp
[0.62, 0.38]
```

adică:

- 62% șanse să NU plece
- 38% șanse să plece

2) `[:, 1]`

Asta selectează **doar coloana 1**, adică **probabilitatea pentru clasa "Yes"** (churn).

Deci:

```python
proba_yes = clf.predict_proba(x_test)[:, 1]
```

Copy code

înseamnă:

> "pentru fiecare client din setul de test, păstrează doar P(churn=Yes)".

Rezultatul `proba_yes` este un vector cu 1409 valori, fiecare între 0 și 1.

---

3) `proba_yes[:10]`

Îți arată **primele 10 probabilități**, doar ca să verifici că arată normal.

Exemplu din output-ul tău:

- 0.045 → risc mic
- 0.684 → risc mare
- 0.401 → risc mediu

4) `proba_yes.min(), proba_yes.max()`

Îți arată intervalul de probabilități:

- min ≈ 0.0017 (aproape sigur "No")
- max ≈ 0.855 (destul de sigur "Yes")

```python
import numpy as np

thresholds = [0.5, 0.4, 0.3]

for t in thresholds:
    y_pred_t = np.where(proba_yes >= t, "Yes", "No")

    print(f"\n=== Threshold = {t} ===")
    print(confusion_matrix(y_test, y_pred_t))
    print(classification_report(y_test, y_pred_t))
```

✓ 0.0s

```
=== Threshold = 0.5 ===
[[926 109]
 [165 209]]
              precision    recall  f1-score   support

          No       0.85      0.89      0.87      1035
         Yes       0.66      0.56      0.60       374

    accuracy                           0.81      1409
   macro avg       0.75      0.73      0.74      1409
weighted avg       0.80      0.81      0.80      1409


=== Threshold = 0.4 ===
[[846 189]
 [124 250]]
              precision    recall  f1-score   support

          No       0.87      0.82      0.84      1035
         Yes       0.57      0.67      0.62       374

    accuracy                           0.78      1409
   macro avg       0.72      0.74      0.73      1409
weighted avg       0.79      0.78      0.78      1409


=== Threshold = 0.3 ===
[[774 261]
 [ 92 282]]
              precision    recall  f1-score   support

          No       0.89      0.75      0.81      1035
         Yes       0.52      0.75      0.62       374

    accuracy                           0.75      1409
   macro avg       0.71      0.75      0.71      1409
weighted avg       0.79      0.75      0.76      1409
```

Compararea pragurilor (pe clasa Yes = churn)

Threshold 0.5 (default)

- Precision **0.66**
- Recall **0.56**
- F1 **0.60**
- Confusion (Yes): TP=209, FN=165

👉 Ratezi destui clienți care pleacă (FN mare).

Threshold 0.4

- Precision **0.57**
- Recall **0.67**
- F1 **0.62**
- Confusion (Yes): TP=250, FN=124

👉 **Câștig major** la recall: prinzi cu **41** clienți în plus care pleacă (250 vs 209), și reduci FN cu **41** (124 vs 165). Trade-off: mai multe alarme false.

Threshold 0.3

- Precision **0.52**
- Recall **0.75**
- F1 **0.62**
- Confusion (Yes): TP=282, FN=92

👉 Recall și mai mare: prinzi cu **73** clienți în plus față de 0.5 (282 vs 209), dar crește și numărul de alarme false (precision scade).

```python
THRESHOLD = 0.3

def predict_with_threshold(model, X, threshold=THRESHOLD):
    proba_yes = model.predict_proba(X)[:, 1]
    return np.where(proba_yes >= threshold, "Yes", "No")
```

[21] ✓ 0.0s

```python
y_pred_prod = predict_with_threshold(clf, x_test)
print(confusion_matrix(y_test, y_pred_prod))
print(classification_report(y_test, y_pred_prod))
```

[22] ✓ 0.0s

```
[[774 261]
 [ 92 282]]
              precision    recall  f1-score   support

          No       0.89      0.75      0.81      1035
         Yes       0.52      0.75      0.62       374

    accuracy                           0.75      1409
   macro avg       0.71      0.75      0.71      1409
weighted avg       0.79      0.75      0.76      1409
```