KNIME PROJECT DOCUMENTATION

Telecom Complaint Prediction (Logistic Regression)

Objective: Build a classification model in KNIME to predict whether a telecom customer will raise a complaint using a pre-cleaned dataset.

Index

Section 0 — Project Overview

This section defines the project scope, dataset, target variable, and evaluation outputs.

0.1 Project Title

KNIME Mini Project — Telecom Complaint Prediction (Logistic Regression)

0.2 Objective

Build a classification model in **KNIME** to predict whether a telecom customer will raise a complaint using a **pre-cleaned telecom dataset**.

0.3 Input Dataset

File: `telecom_data_clean.csv`
Type: Tabular dataset (customer-level records)

0.4 Target Column

Original target: `complaint_flag` (Yes/No)
Converted target: `complaint_flag_num` (1/0)

0.5 Final Model Used

Logistic Regression with **L2 regularization**
(Uniform Gauss / Ridge)

0.6 Evaluation Outputs

Model performance is validated using:

- Confusion Matrix
- Accuracy Statistics:
  - Accuracy
  - Precision
  - Recall
  - F1 Score
  - Cohen's Kappa

Section 1 — Business Understanding

1.1 What is Telecom Complaint Prediction?

In telecom, customer complaints occur due to:

- Network outage / weak signal
- Payment or recharge failures
- Poor customer support handling
- Billing disputes
- Device/network compatibility issues

Complaint prediction helps telecom companies:

- Detect unhappy customers before they complain
- Improve proactive support strategies
- Reduce complaint load on call centers
- Improve customer satisfaction and trust

1.2 Why this is a Classification Problem?

This is a binary classification problem because output is one of two classes:

- 1 = Complaint (Yes)
- 0 = No Complaint (No)

Section 2 — Dataset Overview

In this section, you understand the input dataset and ensure it is safe for ML training (no leakage).

2.1 Dataset Information

Dataset Used
`telecom_data_clean.csv`

Status
Dataset is cleaned and ready for ML modeling.

What it Represents
Telecom customer behavior including:

- usage patterns
- network/outage issues
- recharge/payment behavior
- plan and device details

2.2 Leakage Risk (Critical)

What is leakage?
Leakage happens when the dataset contains fields that directly reveal the complaint outcome.

Why it is dangerous
If leakage columns remain:

- the model learns shortcuts
- accuracy becomes unrealistically high
- results become invalid for real-world use

So leakage columns must be removed before training.

2.3 Columns Removed (Leakage-Safe Filtering)

A) IDs / High-cardinality

- `session_id`, `user_id`, `tower_id`
- `recharge_id`, `usage_id`, `complaint_id`
- `name`

B) Dates / timestamps

- `call_timestamp`, `join_date`
- `recharge_date`, `next_recharge_date`
- `complaint_date`

C) Complaint detail leakage fields

- `complaint`, `issue`, `severity`
- `resolution_status`, `resolution_time_hours`

D) Duplicate region

- Removed: `region_y`
- Kept: `region_x`

E) Out of scope

- `churn`

2.4 Predictive Features Kept

Kept only valid predictive features such as:

- plan-based features

- network/outage behavior
- usage category fields
- recharge/payment behavior
- region/device/gender predictors

These features support realistic complaint prediction.

---

0) Project Setup

Section 3 — KNIME Workflow Setup

In this section, you create a new KNIME workflow and set up a clean, ML-correct pipeline structure.

3.1 Create the Workflow

Start a new KNIME project workspace for the Telecom Complaint Prediction model.

Steps

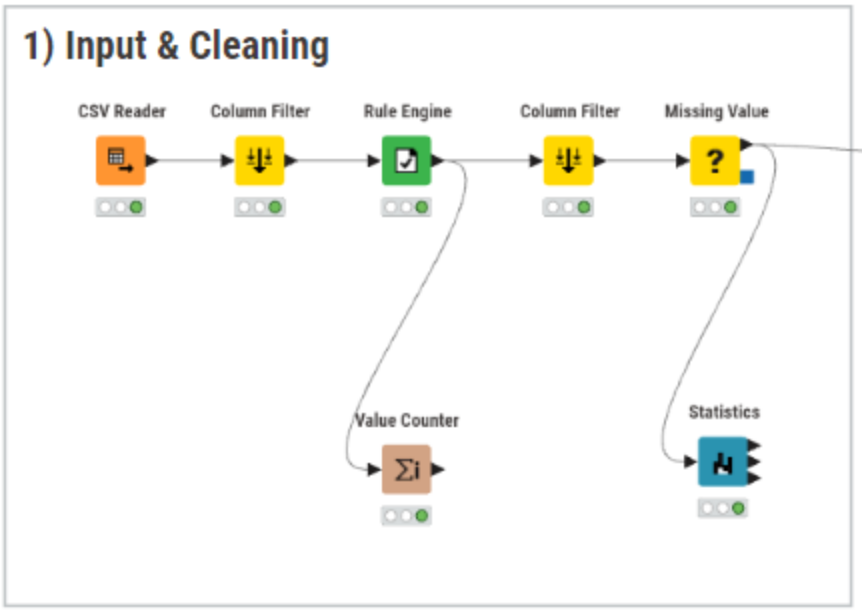
1. Create a **New Workflow**
2. Rename it clearly (example: `Telecom_Complaint_Prediction_LR`)
3. Build the pipeline in a **left** → **right** flow (standard readability)

3.2 Recommended Workflow Best Practices Followed

- Use **clear node labels** (students can understand flow quickly)
- Fix **random seed** wherever available (reproducibility)
- Maintain separate **train vs test branches** (ML correctness)
- Apply **SMOTE only on training branch** (avoid test data leakage)

---

1) Input & Cleaning Block



Section 4 — Data Loading + Leakage-Safe Feature Selection

Node 1: CSV Reader

Purpose

Load cleaned telecom dataset into KNIME workflow.

Actions to Perform

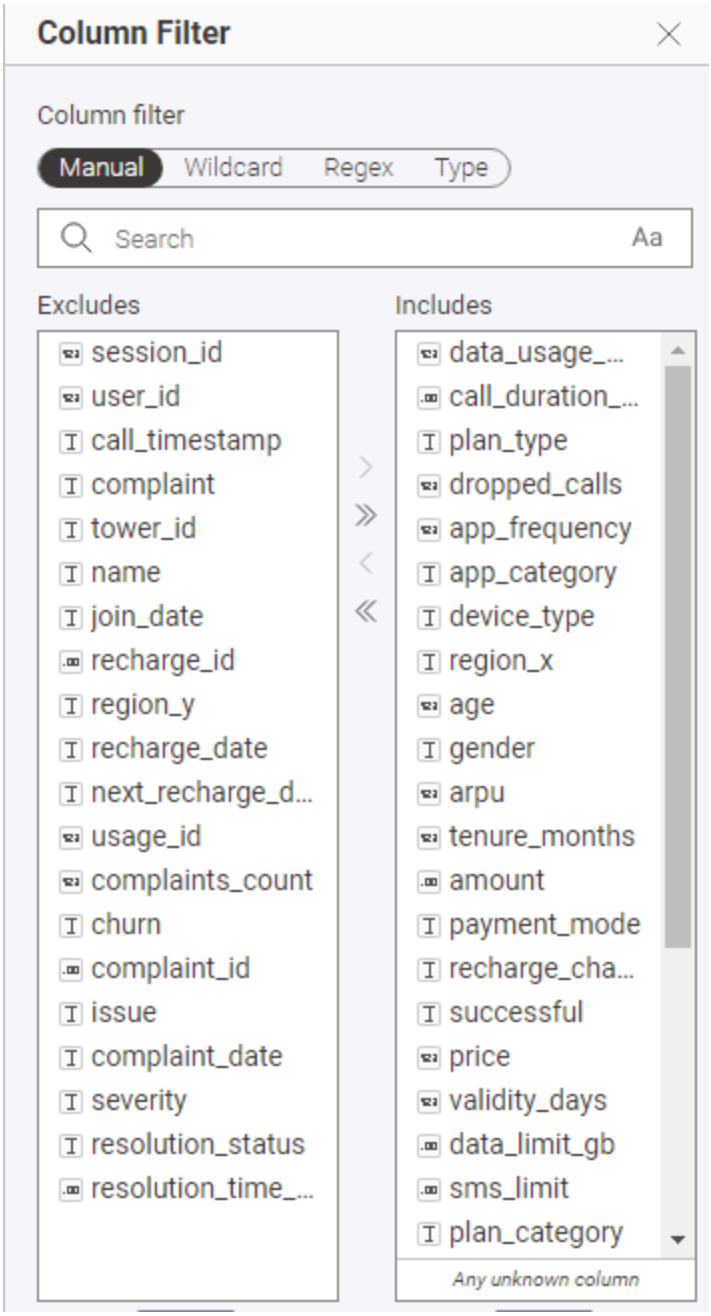
- Select file: telecom_data_clean.csv
- Verify columns loaded correctly

Node 2: Column Filter (Step 1 Filter)

Purpose

Remove fields that:

- are identifiers
- cause leakage
- are timestamps
- are non-predictive



Result

Only relevant ML features are kept.

Section 5 — Target Creation (Complaint Flag → 0/1)

In this section, you convert the target column into a numeric ML-ready format and validate class imbalance.

Node 3: Rule Engine

Purpose

Convert Yes/No class into numeric labels suitable for ML.

Rules Used

```
$complaint_flag$ = "Yes" => 1
$complaint_flag$ = "No"  => 0
TRUE => 0
```

Append Column Created

`complaint_flag_num`

Node 4: Column Filter

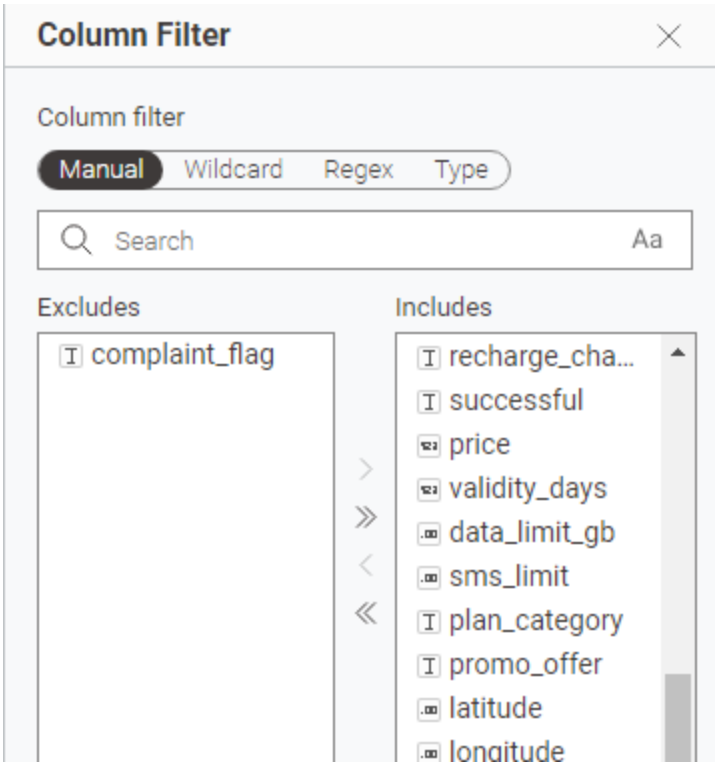
Purpose

Remove string target column and keep numeric target for modeling.

Action

Dropped: `complaint_flag`

Kept: `complaint_flag_num`



Node 5: Value Counter (Validation)

Purpose

Validate class distribution for imbalance awareness.

Column `complaint_flag_num`

Distribution Found
1 = 127
0 = 23

Confirms dataset is imbalanced; SMOTE justified later.

Section 6 — Missing Value Handling

In this section, you handle missing values to ensure the dataset is clean and model-ready.

Node 6: Missing Value

Purpose
Prevent model errors and ensure stable preprocessing pipeline.

Strategy Used

- Numeric columns → replace with **Mean**
- String columns → replace with **Most frequent value**

Output

- cleaned dataset with no blanks/nulls passed forward

Node 7: Statistics (Validation)

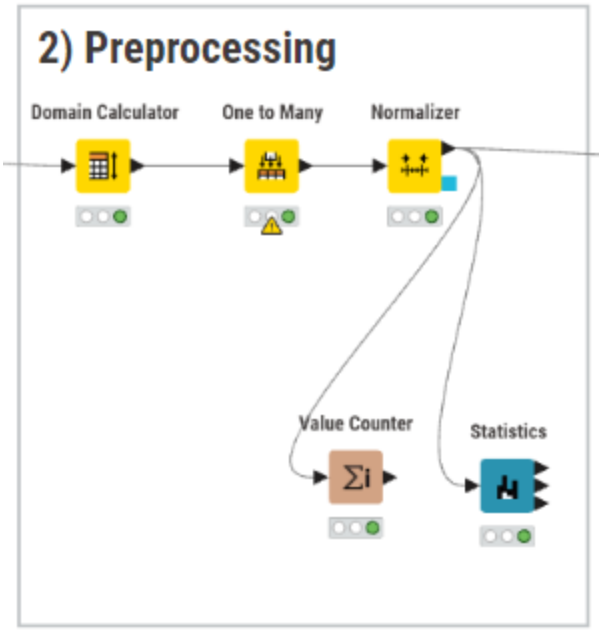
Purpose
Confirm missing value handling is successful.

Validation Checks
`Missing` = 0
`NaNs` = 0

Dataset ready for encoding/scaling.

---

2) Preprocessing Block



Section 7 — Domain / Data Type Preparation

In this section, you ensure KNIME correctly recognizes column types and category domains before encoding.

Node 8: Domain Calculator

Purpose
KNIME workflows depend heavily on domain metadata.

Domain Calculator ensures:

- categorical values are properly recognized
- encoding becomes stable and consistent
- domain values remain consistent across workflow nodes

Outcome

- domain consistency is established before One Hot Encoding
- reduces risk of category mismatch during partitioning and model training

---

Section 8 — Categorical Encoding (One Hot Encoding)

In this section, you convert categorical (string) columns into numeric dummy variables so the model can train on them.

Node 9: One to Many

Purpose
Convert categorical columns into numeric dummy/indicator columns.

Columns Encoded

- `plan_type`
- `app_category`
- `device_type`
- `region_x`
- `gender`
- `payment_mode`
- `recharge_channel`
- `successful`
- `plan_category`
- `promo_offer`
- `network_type`
- `outage_flag`
- `usage_category`

Observed Warning

- Message: **"duplicate possible values found..."**

Reason

- Multiple categorical columns contain repeated labels such as:
  - Yes / No
  - High / Medium / Low
  - True / False

KNIME Handling

- KNIME automatically **prefixes dummy column names** with the original column name to avoid name collisions.

Section 9 — Feature Scaling

In this section, you scale numeric features so Logistic Regression can train effectively.

Node 10: Normalizer

Purpose
Logistic Regression performs best when numeric features are standardized.

Scaling Method
Z-score normalization (standardization)

Critical Fix

- **Exclude** target column: `complaint_flag_num`
- Only feature columns must be normalized

Validation

- Target remains unchanged and valid:
  - Min = `0`
  - Max = `1`.

Node 10.1: Value Counter

Purpose
After applying preprocessing steps (One-Hot Encoding + Normalization), this node is used as a sanity
validation checkpoint to ensure that the target column remains correct and unchanged.

Column `complaint_flag_num`

Expected Output
The Value Counter should show only two classes:

- 1 → Complaint Raised
- 0 → No Complaint

Why this validation is needed
This step confirms that:

- the target column is still present after preprocessing
- the target values remain strictly binary (0 and 1)
- the target column was not accidentally included in normalization/scaling
- no missing / unexpected label values were introduced
- Confirms dataset is imbalanced; SMOTE justified later.

Node 10.2: Statistics (Post-Scaling Validation)

Purpose
Validate preprocessing output quality after normalization.

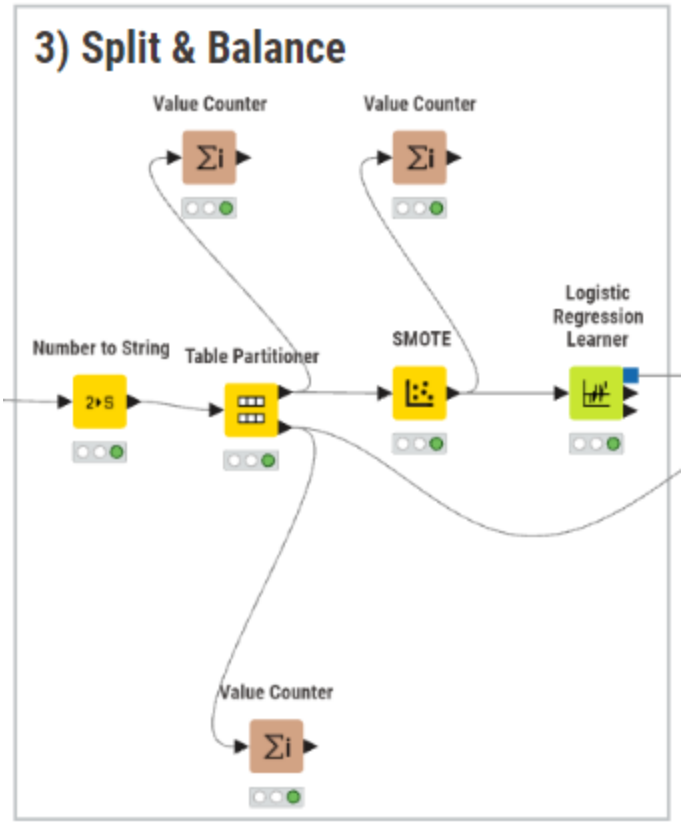
Column `complaint_flag_num`

Checks Performed

- missing values = 0
- NaNs/Inf = 0
- numerical columns standardized
- target column remains binary (Min=0, Max=1)

---

3) Split and Balance Block



Section 10 — Train/Test Split

In this section, you split the dataset into **training** and **testing** partitions using a **stratified split** to preserve class distribution.

Node 11: Number to String

Purpose
Convert target column into **nominal (categorical)** format for stratified partitioning.

Action

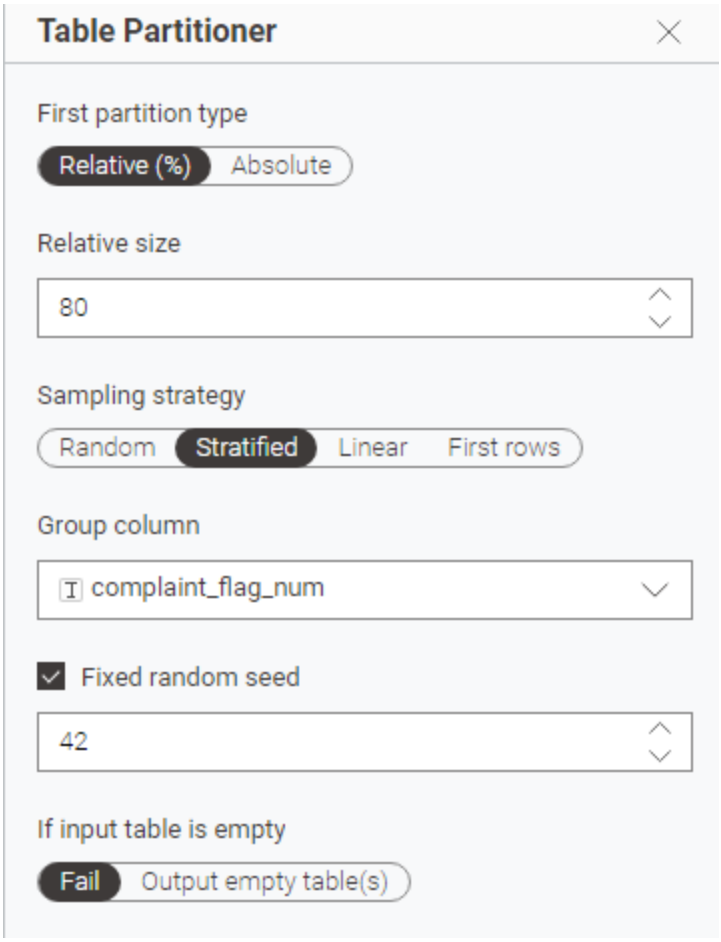
- Convert `complaint_flag_num` → temporary **string** version
- Use this only for split grouping (model target remains numeric later)

Node 12: Table Partitioner

Purpose
Split dataset into **Train (80%)** and **Test (20%)** correctly.

Configuration

- Sampling method: **Stratified sampling**
- Group column: `complaint_flag_num` (string/nominal)
- Train ratio: `80%`
- Test ratio: `20%`
- Seed: `42` (reproducibility)



Node 12.1: Value Counters

Purpose Ensure **both classes (0 and 1)** are present in both partitions.

Check

- Train partition: class 0 and class 1 present
- Test partition: class 0 and class 1 present

Section 11 — Class Imbalance Handling

In this section, you handle **class imbalance** using SMOTE so the model learns minority-class patterns properly.

Node 13: SMOTE

Purpose
Balance minority class (0 = No Complaint) only in training set.

Why only training branch?
Because test set must remain real-world / untouched.

SMOTE Configuration

- Class column: `complaint_flag_num`
- Nearest Neighbour = 5
- Oversampling Method = Miority classes
- Random seed = 42

Output

- A **balanced training dataset** (oversampled) used for model training

Node 13.1: Value Counter

Purpose Validate that SMOTE successfully balanced the training dataset.

Column `complaint_flag_num`

Distribution Found (Post-SMOTE)

- 1 = 102
- 0 = 102

Confirms training data is now balanced (50:50) for stable Logistic Regression learning.

Section 12 — Model Training

In this section, you train a **Logistic Regression classification model** using the SMOTE-balanced training dataset.

Node 14: Logistic Regression Learner

Purpose
Train Logistic Regression classifier for predicting telecom complaints.

Input

- SMOTE-balanced training data (from **SMOTE** node)

Configuration

- Target column (class): `complaint_flag_num`
- Enable **Regularization** to handle multicollinearity
  - Method: **Uniform Gauss (L2 / Ridge)**
- Max iterations / Epochs: **1000** (for stable convergence)

Important Note (Common Issue + Fix)

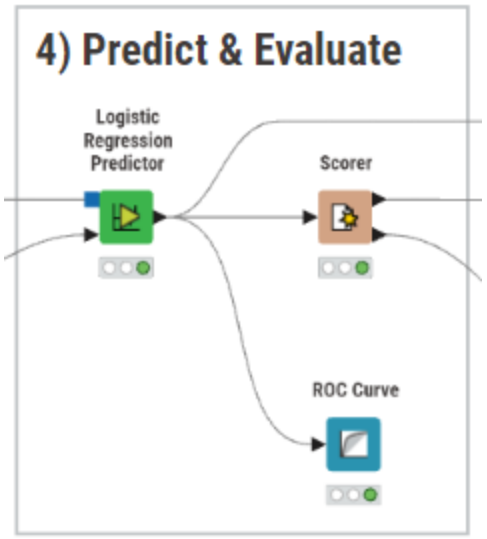
- Problem: **Non-convergence** / **Singular matrix warning**
- Cause: **Multicollinearity** created by one-hot encoded dummy variables
- Fix: Use **L2 regularization + higher epochs**

Output

- Trained Logistic Regression **model port** (used in Predictor)

---

4) Predict & Evaluate Block



Section 13 — Prediction

In this section, you apply the trained Logistic Regression model on the **test dataset** and generate prediction outputs.

Node 15: Logistic Regression Predictor

Purpose
Generate predictions on test data.

Inputs

- Model port output from **Logistic Regression Learner**
- Test partition output from **Table Partitioner**

Configuration

- Target / class column: `complaint_flag_num`
- Keep prediction + probability columns enabled (recommended)

Output Created

- `Prediction (complaint_flag_num)`
  (Optional: probability/confidence columns for ROC Curve)

Section 14 — Evaluation

In this section, you evaluate the model performance using **Scorer** and **ROC Curve**.

Node 16: Scorer

Purpose
Evaluate classification results using confusion matrix and metrics.

Configuration

- Actual column: `complaint_flag_num`
- Predicted column: `Prediction (complaint_flag_num)`

Outputs

- Output Port 1 → Confusion Matrix table
- Output Port 2 → Accuracy Statistics table

Output Results

1. Confusion Matrix Results

Actual / Predicted	1	0
1	22	3
0	1	4

TP=22, FN=3, FP=1, TN=4

2. Accuracy Statistics (Overall)

- Accuracy = 0.867
- Cohen's Kappa = 0.586

3. Class-wise Metrics

Class	Precision	Recall	F1
1	0.957	0.88	0.917
0	0.571	0.80	0.667

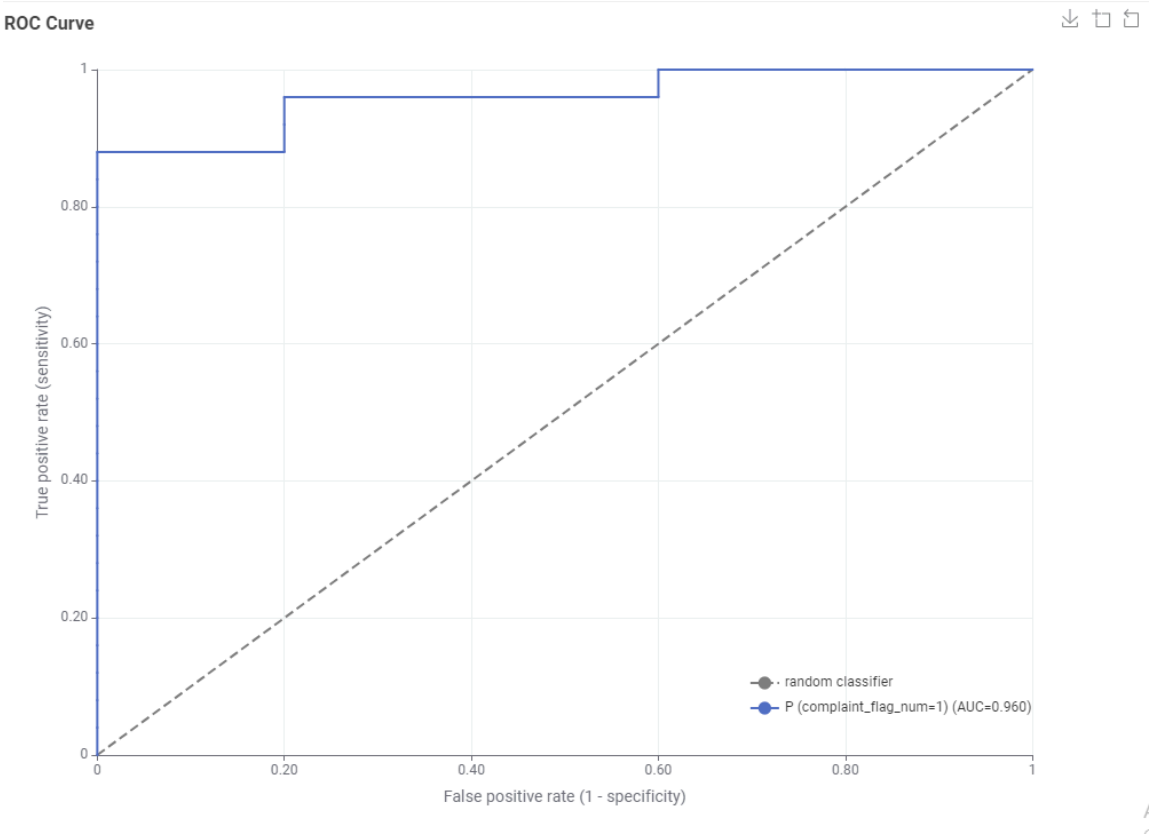
Node 17: ROC Curve

Purpose
Visualize classification threshold performance and check model separability.

Steps

1. Add **ROC Curve** node after **Logistic Regression Predictor**

2. Configure class column: `complaint_flag_num`
3. Select positive class: `1`
4. Ensure probability column is selected (class probability output)
5. Execute node and open ROC view



Expected Output

- ROC curve graph
- AUC value displayed in ROC view

---

5) Exports Block

Section 15 — Exports / Deliverables

In this section, you export final outputs using **CSV Writer** nodes.

- **Predictions CSV:** knime_predictions.csv
- **Confusion Matrix CSV:** knime_confusion_matrix.csv
- **Model Metrics CSV:** knime_model_metrics.csv

15.1 Predictions Export

Purpose: Save predicted results for test data
Output file: `knime_predictions.csv`

Steps

1. Add **CSV Writer** after **Logistic Regression Predictor**
2. Set file name: `knime_predictions.csv`
3. Execute and verify prediction column exists

15.2 Confusion Matrix Export

Purpose: Save confusion matrix table
Output file: `knime_confusion_matrix.csv`

Steps

1. Add **CSV Writer** from **Scorer** → **Output Port 1**
2. Set file name: `knime_confusion_matrix.csv`
3. Execute and verify matrix values

15.3 Model Metrics Export

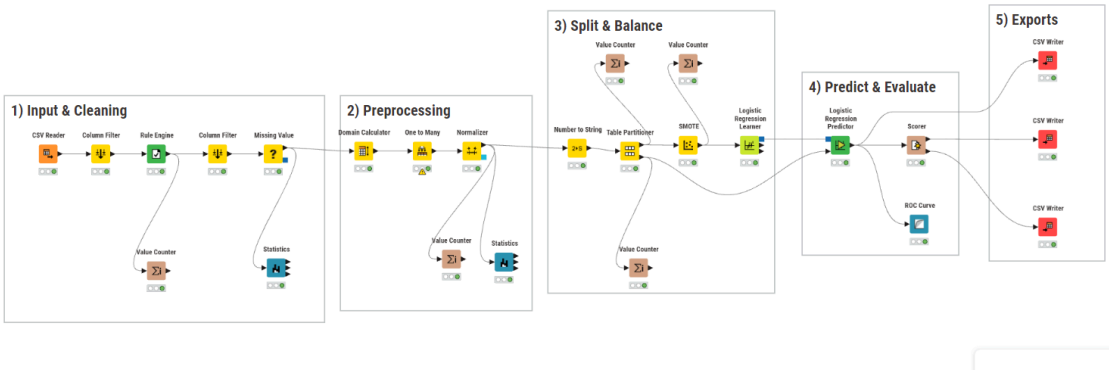
Purpose: Save accuracy + precision/recall/F1 metrics
Output file: `knime_model_metrics.csv`

Steps

1. Add **CSV Writer** from **Scorer** → **Output Port 2**
2. Set file name: `knime_model_metrics.csv`
3. Execute and verify metrics table

Section 16 — Final KNIME Workflow Summary

Final Node Flow (Left → Right)



What You Achieved

- Built end-to-end ML pipeline
- Leakage-safe modeling
- Proper scaling and encoding
- Balanced training data
- Regularization for convergence

- Exported all model outputs

Suggested Improvements

- Add ROC Curve node after Scorer
- Add Feature Importance
- Use Logistic Regression coefficients
- Add Parameter Optimization Loop
- Try Random Forest / XGBoost for comparison

🎉 End of Project

Section 17 — Student Practice Tasks

17.1 Change Train/Test Split Ratio

Modify Table Partitioner to 70/30. Compare Accuracy, Precision, Recall, F1.

17.2 Try a Different Normalization Method

Change Normalizer to Min-Max. Observe impact on Logistic Regression.

17.3 Improve Target Validation

Add Value Counter after Rule Engine. Confirm 0 vs 1 balance.

17.4 Compare Model Results Without SMOTE

Bypass SMOTE. Compare Confusion Matrix & F1. Analyze Recall change.

17.5 Export More Outputs

Export preprocessed dataset & encoded dataset as CSV for documentation.

17.6 Reflection Questions

- Which step prevented data leakage the most?
- Why is SMOTE applied only on training data?
- Why was regularization needed in this project?

End