Great! Here's a consolidated, high-context knowledge base (within ~1000 lines) for training your Smart Data Analyzer chatbot. This includes memory, formulas, code snippets, and how n8n integrates into the system.

### **🧠 Smart Data Analyzer — Master Context for AI Chatbot**

#### **🚀 Core Concept**

A universal data quality and insights platform powered by AI agents (n8n) + deep learning (e.g., TabNet), capable of:

* Evaluating CSV datasets of any type (numerical, categorical, date, text).
* Computing 20+ data quality metrics.
* Generating actionable insights via a connected AI agent using n8n.
* Frontend hosted on Netlify, backend API (Flask or FastAPI) hosted on Render or Railway.

#### **🧪 20+ Data Quality Metrics (Examples)**

| **Metric** | **Description** |
| --- | --- |
| Data\_Quality\_Score | Weighted score (0–100) from all other metrics. |
| Anomaly\_Count | Count of outlier values using IQR/Z-score. |
| Cardinality\_Categorical | Count of unique values in categorical columns. |
| Class\_Overlap\_Score | Score (0–1) measuring classification ambiguity. |
| Data\_Density\_Completeness | Fraction of non-null entries. |
| Data\_Freshness | Time since last update (in days/hours). |
| Data\_Type\_Mismatch\_Rate | Percentage of mismatched types (e.g., strings in float column). |
| Domain\_Constraint\_Violations | Rows violating custom business rules. |
| Duplicate\_Rows\_Count | Total duplicates in the dataset. |
| Null\_Value\_Count | Total missing values. |
| Outlier\_Score | Composite score using multiple methods. |
| Skewness\_Score | Skewness per numerical feature. |
| Imbalance\_Score | Target class imbalance ratio. |
| High\_Correlation\_Columns | Features with >0.9 Pearson correlation. |
| Leakage\_Detection\_Score | Detects features that leak target info. |

#### **🧮 Formulas / Logic**

**Outlier Detection (IQR Method)**:  
  
 Q1 = df[col].quantile(0.25)

Q3 = df[col].quantile(0.75)

IQR = Q3 - Q1

outliers = df[(df[col] < Q1 - 1.5\*IQR) | (df[col] > Q3 + 1.5\*IQR)]

**Data Freshness**:  
  
 df['timestamp'] = pd.to\_datetime(df['timestamp'])

freshness = (datetime.now() - df['timestamp'].max()).total\_seconds() / 3600

**Missing Value %**:  
  
 missing\_rate = df.isnull().mean().mean()

**Categorical Cardinality**:  
  
 cardinality = df.select\_dtypes(include='object').nunique()

#### **🤖 n8n Agent Integration (Flow)**

1. **Frontend Button:**
   * Button labeled: Get n8n Insights.
   * Triggers JSON POST request to local/production n8n webhook.

**Payload Example:** {

"category": "Model Recommendation",

"metrics": {

"Data\_Quality\_Score": 92.1,

"Imbalance\_Score": 0.67,

"Class\_Overlap\_Score": 0.53

},

"problem\_type": "classification"

}

1. **Agent Flow (n8n):**
   * **Trigger**: HTTP Webhook.
   * **AI Model**: OpenAI node (GPT-4 or Claude).
   * **Memory**: Optional — use chat history node.
   * **Return Insight**: As text, to be displayed under chart or table.

**Frontend Render Insight**:  
  
 createChat({

webhookUrl: 'https://n8n-n91d.onrender.com/webhook/xyz',

// Styles are included

});

#### **🧑‍💻 Synthetic Data Generation Snippet**

from sklearn.datasets import make\_classification

import pandas as pd

X, y = make\_classification(n\_samples=100000, n\_features=27, random\_state=42)

df = pd.DataFrame(X, columns=[f"feature\_{i}" for i in range(27)])

df["target\_num"] = y

df["target\_text"] = df["target\_num"].map({0: "Class A", 1: "Class B"})

#### **⚙️ Model - TabNet (Deep Learning for Tabular)**

* Benefits:  
  + Handles missing values internally.
  + Works well with mixed feature types.
  + Explains decisions using feature masks.
* Preprocessing Needed:  
  + Encode categorical features (LabelEncoder or embeddings).
  + Normalize numerical features.

#### **🧩 Missing Value Handling Strategy**

| **Feature Type** | **Strategy** |
| --- | --- |
| Numeric | Mean/Median or KNNImputer |
| Categorical | Mode or "Unknown" |
| Date | Fill with most recent |
| Text | Fill with "" or "missing" |

#### **📦 Project Hosting Stack**

| **Layer** | **Technology** |
| --- | --- |
| Frontend | Next.js (Netlify) |
| Backend API | Flask or FastAPI (Render or Railway) |
| ML Model | TabNet, sklearn |
| Workflow Logic | n8n with OpenAI agent |
| Database | Supabase (optional for logs/chat history) |

#### **💬 Chatbot Training Tips**

* Inject all metric definitions, preprocessing rules, example inputs.
* Allow chatbot to analyze metric JSON and generate:  
  + Model recommendations
  + Preprocessing advice
  + Visual explanation of anomalies
  + Fixes for imbalance, missing values, or duplicate data
* Provide deep memory of TabNet, n8n, and pipeline logic.

#### **🧠 Chatbot Prompt Examples**

**User:** My Imbalance\_Score is 0.9 and Data\_Quality\_Score is 60. What should I do?

**Bot:** 🔍 Your dataset has a severe class imbalance (score 0.9). Consider using techniques like SMOTE, class weights, or undersampling.  
 📊 Also, your Data Quality Score is low—check for nulls, outliers, and type mismatches.

This is a highly condensed knowledge block your chatbot can reference or be trained on. Let me know if you'd like this turned into a JSON/Markdown/embedding-friendly format or broken into modular docs.

Here’s a **100-line context summary** for your chatbot project, derived from your three uploaded workflow files: ai\_insights.json, data\_chatbot\_side\_one.json, and n8n\_chatbot\_workflow.json. This context can be used to describe the bot’s architecture, data flow, integration logic, and design choices.

### **🧠 Chatbot Context (100 Lines)**

1. This chatbot project is built using **n8n** workflows.
2. The workflows integrate **LangChain-compatible** agents and memory modules.
3. It supports **multiple language models**, including **Groq** and **Mistral Cloud**.
4. **Trigger Mechanism:** Each chatbot initiates using a chatTrigger node that listens to new chat messages.
5. The primary trigger receives HTTP requests via a public webhook.
6. This webhook can be used to connect platforms like Telegram, web frontends, or custom UIs.
7. Each chat session may maintain **conversational context** using a memory buffer (memoryBufferWindow).
8. The memory buffer helps retain short-term conversation state across user inputs.
9. AI agent nodes (agent) handle orchestrating logic between the LLM and memory.
10. The system message for the agents enforces clarity, conciseness, and excludes JSON metadata in replies.
11. The **Groq-based** model is used in one flow (ai\_insights.json) with ID PmEVccOCzIuTTFba.
12. The **Mistral Cloud** model is used in the other two flows (data\_chatbot\_side\_one.json, n8n\_chatbot\_workflow.json).
13. API credentials are securely referenced via credential objects (not hardcoded).
14. All workflows use the **LangChain lmChat nodes** to connect with the respective model providers.
15. After processing the AI response, a code node formats the output text.
16. This node ensures a clean string is returned—whether the AI responds with plain text or nested object fields.
17. Fallback logic provides default error messages when parsing fails.
18. For user interaction monitoring, workflows log each Q&A interaction.
19. Logging is handled through **Google Sheets integration** via the googleSheets node.
20. The spreadsheet used is: dataviz-ai-questions.
21. Columns include Question, Answer, Timestamp, and SessionId.
22. The bot identifies sessions using a chat ID or generated session ID.
23. A separate node (Process Chat Data) consolidates question/answer pairs and timestamps them.
24. Another node (Filter Errors) ensures only successful interactions are stored.
25. Error conditions include empty, missing, or fallback answers.
26. Messages are also forwarded to **Telegram** using the telegram node.
27. Telegram messages include Markdown formatting for clarity.
28. This provides real-time visibility into chatbot usage.
29. Workflows include detailed JS logic for safely extracting, formatting, and cleaning chat data.
30. Message responses are routed through nodes like Format Chat Response and Process Chat Data.
31. The use of await ensures AI responses are received before continuing.
32. These workflows are designed to support **low-latency response time**.
33. The agents are stateless unless memory context is explicitly enabled.
34. One workflow (data\_chatbot\_side\_one) uses a memory buffer named Simple Memory1.
35. Another workflow (ai\_insights) uses a separate Simple Memory node.
36. All memory nodes utilize the memoryBufferWindow type from LangChain integration.
37. The workflows are modular and support parallel or chained execution paths.
38. When chat message received is always the entry point node.
39. Agent nodes are connected directly to both memory and model components.
40. One agent workflow includes a system prompt guiding the bot behavior.
41. The system prompt is: “You are a helpful AI assistant...”
42. This ensures the agent focuses on helpful, non-verbose answers.
43. Responses are consistently returned in plain text form, stripped of code or JSON.
44. Model parameters such as temperature, top\_p, and maxTokens are configurable.
45. One Mistral instance uses temperature: 0.7 and maxTokens: 1000.
46. Origins are not restricted (allowedOrigins: \*) in chat trigger setup.
47. Chatbot sessions can load previous messages when loadPreviousSession: true.
48. The workflows can run in **multi-user** environments using session keys.
49. Unique webhook IDs are assigned per workflow for message routing.
50. All workflows are actively deployed (except the n8n\_chatbot\_workflow.json marked inactive).
51. The data\_chatbot\_side\_one and ai\_insights flows are marked as active.
52. Each chatbot flow is tagged with a versionId and a meta.instanceId.
53. Sheets API integration uses OAuth2 and references shared credentials.
54. Telegram integrations use registered bot tokens referenced by name.
55. Telegram chat ID used: 5352707272.
56. Workflow architecture separates logic, memory, and transport layers.
57. The bot is modular and easy to scale by switching the model provider node.
58. The Telegram message includes: question, answer, session ID, timestamp.
59. The sheet appends new rows after each valid interaction.
60. Fallback and error messages are human-readable and informative.
61. The bot can support visualization bots by adding chart/image nodes.
62. Output formatting handles edge cases such as undefined fields or nested formats.
63. Workflows are visual-first (n8n style) but support raw JS code for flexibility.
64. Model credentials are securely managed and reused across nodes.
65. Session IDs help in future personalization, summaries, or analytics.
66. The sheet used for data tracking is publicly accessible (based on URLs).
67. JavaScript nodes return only one JSON object with key text.
68. The model response selection prioritizes output.text, then output.content.
69. Errors during parsing or formatting are logged as message errors.
70. Each AI model node is plug-and-play with the agent node.
71. The AI agent always sits between the trigger and the formatter.
72. There’s no frontend UI defined—but a web frontend could easily POST to the webhook.
73. Memory can be enhanced with vector memory or persistent storage later.
74. All nodes are version-controlled (e.g., typeVersion: 1.3, 2, etc.).
75. The workflows are future-ready with placeholders for expansion.
76. Clean separation of business logic, memory, formatting, and output.
77. Telegram node uses Markdown parse mode for better formatting.
78. Credentials are linked via friendly IDs like Groq account, Mistral Cloud account.
79. Data flow: Trigger → Agent → Formatter → Sheet/Telegram.
80. Memory nodes can be scaled with session scoping for larger conversations.
81. The fallback message is: "I'm sorry, I couldn't process that request properly."
82. Each response code node has error-handling and default message fallbacks.
83. Questions are dynamically captured from chatInput field.
84. Responses are mapped via output from AI Agent or formatter node.
85. Timestamp uses ISO format for compatibility.
86. Session ID defaults to session\_ + timestamp when missing.
87. Workflows can integrate analytics or dashboards based on sheet data.
88. Sheet schema is predefined with types and fields marked optional.
89. Each row in the sheet can be linked back to a specific session/chat ID.
90. System messages for agents are customizable per workflow.
91. Workflows can send real-time logs, alerts, or updates via Telegram.
92. Workflows can be cloned or extended with new branches (e.g., image generation).
93. Telegram feedback can help in monitoring model behavior and errors.
94. Each code node is isolated and reusable across workflows.
95. n8n allows visually connecting complex logic without losing transparency.
96. The chatbot is API-first and does not require a frontend to operate.
97. Each step in the workflow ensures consistency and fallback handling.
98. Workflows could be converted into reusable templates.
99. The system supports multiple platforms via webhook compatibility.
100. Overall, this chatbot architecture is robust, extendable, and modular.

Let me know if you'd like a shorter executive summary, a diagram of the architecture, or how to deploy this on your preferred hosting environment.

Thank you for uploading your files. Here’s a detailed breakdown and in-depth technical analysis of the project composed of the three main components:

## **🔍 Project Overview**

This project is a **web-based data quality analysis platform** built with:

* **Frontend**: React, Next.js, Chart.js
* **Backend integrations**: Flask (for metric computation), MongoDB (for cache/persistence), n8n (as a conversational webhook system)
* **UX Components**: Chatbot UI, data insight dashboards, AI-driven recommendations

It enables users to:

* Upload datasets
* View automatically computed data quality metrics
* Interact with an AI assistant to interpret insights
* Trigger model-driven suggestions (ML/DL use-cases)
* Download insights and conversations

## **📁 FILE: chatbot.tsx — AI Data Assistant**

This file defines a **fully interactive chatbot widget** that serves as a user-facing interface for exploring dataset insights.

### **🔧 Key Features**

* **Toggleable UI widget** fixed at bottom-right
* Fetches and displays user datasets from MongoDB
* Allows selection of datasets to fetch cached metrics
* Sends user questions to an n8n webhook
* Displays bot responses based on n8n-generated answers
* Handles failures, retries, and downloadable transcripts

### **🧠 Chat Memory**

interface ChatMessage {

type: 'user' | 'bot';

content: string;

timestamp: Date;

}

Chat history is maintained in state and persisted for the session. Messages are timestamped and distinguish between user and bot.

### **📊 Metrics Structure**

interface Metrics {

basic: { nullCounts, uniqueCounts, datatypes }

statistical: { mean, median, std, min, max }

advanced: { correlations, outliers }

}

All metrics are structured and passed to the AI assistant during interaction. This rich metadata enables contextual conversations.

### **🔗 n8n Integration**

The chatbot communicates with an n8n webhook at:

https://n8n-n91d.onrender.com/webhook/...

It sends structured JSON payloads including:

* action: always "sendMessage"
* chatInput: user message
* sessionId: a composite of userId and datasetId
* context: JSON stringified metrics and metadata

### **📤 Downloadable Transcript**

Provides a button to **download chat as .txt** including timestamps, dataset name, and dialogue.

## **📁 FILE: n8n.tsx — AI Insight View**

This file is a **dedicated AI-driven insight renderer** using the same n8n backend but optimized for analytics display.

### **🔄 Prompt Generator**

The formatAllMetrics() function dynamically composes a structured prompt from available metrics:

Please analyze my dataset and provide insights on:

1. Preprocessing Suggestions

2. Model Recommendations

3. Production Readiness

4. Monitoring & Alerts

Dataset Metrics: ...

Top Issues: ...

Overall Score: ...

### **🤖 sendToN8n() Logic**

Sends payload to webhook with full metrics + fallback retry logic for non-standard responses. Handles malformed or incomplete n8n responses gracefully.

### **📄 Insight Rendering**

* Extracts section headers like 1. Preprocessing Suggestions
* Formats each section into **gradient cards**
* Converts bullet points and paragraphs into stylized readable blocks

### **📥 Downloads**

Users can download insight summaries (generated by AI) as plain .txt files after cleaning formatting symbols like \*\*.

## **📁 FILE: page.tsx — Quality Metrics Dashboard**

This is the **core dashboard page** providing:

* Dataset selection
* Metric loading from MongoDB (or computed via Flask API)
* Metric visualization and scoring
* Deep learning prediction via API
* Integrated chatbot and insights

### **📈 Visuals with Chart.js**

Charts are created dynamically using:

BarElement, CategoryScale, LinearScale

It compares **top 5 best vs worst metrics**, auto-scales based on scores, and adapts to resize events.

### **🧮 Metric Categories**

Metrics are grouped into:

* data\_structure
* data\_quality
* statistical
* advanced

They are color-coded and scored using defined thresholds and reverse-logic (e.g., lower missing values = better score).

### **🧠 Deep Learning Integration**

Sends feature metrics to:

https://data-viz-ai-model.onrender.com/predict

Returns:

* overall\_score
* quality\_label
* top\_issues
* metric\_scores

These are then visualized and explained.

### **🛠️ Backend Fallback Logic**

* Checks MongoDB cache for existing metrics
* If missing, converts dataset into CSV and sends to Flask API
* Returns computed metrics for immediate use (unsaved)
* User can then save these metrics to MongoDB manually

### **🔁 Metrics Save API**

POST /api/metrics/save

{

userId, datasetId, metrics, timestamp

}

## **📐 System Flow**

graph TD

A[User Opens Dashboard] --> B[Select Dataset]

B --> C{Is Dataset Cached?}

C -- Yes --> D[Load Metrics from MongoDB]

C -- No --> E[Generate Metrics via Flask]

D --> F[Display Dashboard & Charts]

E --> F

F --> G[User opens Chatbot]

G --> H[Chatbot Loads Metrics & Dataset]

H --> I[User Asks Question]

I --> J[n8n Webhook Request]

J --> K[n8n Response]

K --> L[Bot Responds]

F --> M[User Asks for AI Insights]

M --> N[n8n Request with Full Prompt]

N --> O[AI Insights Rendered]

## **🔐 Authentication & Session**

All three components use:

const { data: session } = useSession();

Ensures user-scoped dataset access and personalized chat.

## **🧩 Component Composition**

* Chatbot: Embedded on every page, allows conversational interface
* N8nInsights: Rich static AI results based on metrics
* QualityMetrics (page.tsx): Central dashboard with full control

## **✅ Strengths**

* Modular and scalable design
* Handles failure states and fallback gracefully
* Rich UX and visual feedback
* AI integration via n8n allows custom workflow logic

## **🔁 Suggestions for Scale-Up**

* 🔐 Role-based access: Admin vs Analyst
* 🗂️ Dataset upload feature with file parsing
* 📊 Downloadable reports in PDF
* 🧠 Add memory to chatbot across sessions
* 🗺️ Embed lineage/feature impact graphs
* 🕵️ Drift detection across time

## **📌 Next Steps**

If you’d like, I can:

* Generate **full documentation** in .md or .docx
* Break down each component’s code line-by-line
* Convert this into an **engineering spec**
* Help create test cases or API mocks

Let me know how you'd like to proceed.

Would you also like this entire summary exported into a .txt or .docx format for sharing?

Perfect — here is **Version 2** of your Smart Data Analyzer chatbot training context, focused entirely on **advanced material not already given in Version 1**. This includes deeper technical architecture, error handling, optimization logic, advanced n8n workflows, memory design, and UI enhancement strategies.

### **💡 Smart Data Analyzer — Chatbot Training Context v2 (Advanced/Extended)**

#### **🏗️ Expanded Architecture Overview (Data → Insight)**

graph TD

A[CSV Upload] --> B[Data Ingestion Module]

B --> C[Data Cleaning Engine]

C --> D[Metric Calculator (20+ metrics)]

D --> E[Score Aggregator + Analyzer]

E --> F[n8n Agent API Call]

F --> G[Insight Display Component]

* **Data Ingestion Module**: Parses CSV, auto-detects schema, types, and size.
* **Metric Calculator**: Modular function set to allow metric extension.
* **n8n Agent Call**: Includes HTTP retry logic, timeouts, and fallback handling.

#### **⚠️ Error Handling Strategies**

| **Error Type** | **Detection Method** | **Resolution** |
| --- | --- | --- |
| CORS error from n8n | fetch fails, CORS header missing | Show retry button or use proxy server |
| Memory Overflow (JS) | JS heap out of memory in browser logs | Use Web Workers or batch processing |
| n8n Timeout (Status 200 with no output) | Empty response body | Add timeout fallback text like “n8n agent is thinking...” |
| API Down (Render) | 500/503 | Auto-switch to backup n8n URL (Netlify if deployed) |

#### **🧠 n8n Agent Memory + Chat Context Logic**

You can integrate **contextual memory** inside n8n using:

{

"type": "memory",

"context": [

{"role": "user", "content": "My dataset has too many nulls."},

{"role": "assistant", "content": "Try imputing with mean or using KNN."},

{"role": "user", "content": "Now my Imbalance Score is high."}

]

}

* Store context in Supabase or n8n storage module.
* Restore in OpenAI node as Previous messages.

#### **🧪 New Metric Ideas (Version 2)**

| **Metric Name** | **Description** |
| --- | --- |
| Multicollinearity\_Score | Variance Inflation Factor (VIF)-based redundancy measure |
| Data\_Drift\_Detection | Measures if a dataset has distribution shift from training |
| Feature\_Importance\_Leak | Detect if a feature too strongly predicts the target (potential leakage) |
| Rare\_Label\_Frequency | % of infrequent labels in categorical variables |
| Shapley\_Stability\_Score | Checks if Shapley explanations are consistent over similar rows |

#### **📊 Advanced Visualization UI Enhancements**

* 🧱 **Metric Cards Grid**:  
  + Each card shows metric name, score, impact severity badge (⚠️, 🟡, ✅)
  + Color-coded (red <60, orange <80, green ≥80)
* 📈 **Anomaly Timeline Plot**:  
  + Plot time-series data and overlay anomaly count spikes.
* 🧩 **Feature Dependency Graph**:  
  + Use recharts or cytoscape.js to visualize relationships between columns.

#### **🧠 TabNet Model Deployment Hints**

# Convert PyTorch TabNet model to ONNX for inference API

python export\_onnx.py --model tabnet\_model.pt --output tabnet\_model.onnx

* Use onnxruntime in production Flask app for faster inference:

import onnxruntime as rt

sess = rt.InferenceSession("tabnet\_model.onnx")

#### **🔄 Chatbot Fallback & Auto-Retry Logic**

try {

const response = await fetch(n8nURL, { method: 'POST', body: JSON.stringify(data) });

if (!response.ok) throw new Error("Agent unavailable");

const json = await response.json();

displayInsight(json.output);

} catch (e) {

displayInsight("⚠️ Insight unavailable. Retrying...");

setTimeout(() => retryInsightFetch(data), 3000);

}

#### **🧠 Chatbot Advanced Prompts (Training Set)**

**Prompt 1:**

Dataset has high skewness and a few dominant categorical values. What preprocessing steps do you suggest?

**Bot Output:** Apply log or Box-Cox transform to skewed numeric fields. For categorical dominance, group rare categories into an "Other" bucket or apply frequency encoding.

**Prompt 2:**

Why is my Class Overlap Score high, even with clean data?

**Bot Output:** A high Class Overlap Score (e.g., >0.6) indicates poor feature separability. Try dimensionality reduction (PCA/t-SNE) or add engineered features to improve signal.

#### **🧩 JSON Prompt Structure to n8n Agent (Advanced)**

{

"project": "Smart Data Analyzer",

"metrics": {

"Data\_Quality\_Score": 88.9,

"Class\_Overlap\_Score": 0.65,

"Skewness\_Score": 0.72,

"Outlier\_Score": 0.4

},

"dataset\_info": {

"row\_count": 5000,

"column\_count": 12,

"feature\_types": ["numeric", "categorical", "text"]

},

"problem\_type": "regression",

"action\_type": "insight\_request"

}

#### **🛠️ Chatbot Personality / Tone Recommendations**

* Use helpful, non-technical summaries by default.
* Allow toggling "🔍 Expert View" for advanced insights.
* Example:  
  + **Default**: “Your data seems balanced and well-distributed. Great job!”
  + **Expert**: “Class distribution ratio is 1.2:1. No action needed under current thresholds.”

Would you like me to combine both v1 and v2 into a full training\_data.jsonl or embed-ready format?

Here’s a comprehensive **technical breakdown of the API endpoints and their usage** within your platform based on the source code analysis. This includes coding logic, data flows, formulas, integration points, and error handling across the three main files (chatbot.tsx, n8n.tsx, page.tsx).

## **📡 ENDPOINT MAP**

| **Endpoint** | **Method** | **Used In** | **Purpose** |
| --- | --- | --- | --- |
| /api/metrics/list-datasets | GET | chatbot.tsx | Fetch all datasets for current user |
| /api/metrics/get | GET | chatbot.tsx, page.tsx | Retrieve cached metrics for dataset |
| /api/metrics/save | POST | page.tsx | Save generated metrics to MongoDB |
| /api/datasets | GET | page.tsx | List datasets for dashboard UI |
| https://metric-models-dataviz.onrender.com/analyze | POST | page.tsx | Flask backend to compute metrics from CSV |
| https://data-viz-ai-model.onrender.com/predict | POST | page.tsx | ML model to compute AI score + top issues |
| https://n8n-n91d.onrender.com/webhook/... | POST | chatbot.tsx, n8n.tsx | AI assistant webhook (via n8n) |

## **🧠 CHATBOT ENDPOINT FLOW (chatbot.tsx)**

### **🟢 Fetch Datasets**

const response = await fetch('/api/metrics/list-datasets');

* Fetches all user-specific datasets from MongoDB.
* Expects a JSON list of objects: { \_id, name }
* On error: renders an AI error message and shows a toast.

### **🟡 Load Metrics**

await fetch(`/api/metrics/get?userId=${userId}&datasetId=${datasetId}`)

* Checks for cached metrics in MongoDB.
* If successful, loads into currentMetrics state.
* If empty: alerts the user that metrics need to be generated first.

### **🔵 Send Message to n8n Webhook**

await fetch('https://n8n-n91d.onrender.com/webhook/<uuid>/chat', {

method: 'POST',

body: JSON.stringify({

action: 'sendMessage',

chatInput: userMessage,

sessionId: `${userId}\_${datasetId}`,

context: JSON.stringify({

datasetName,

metrics,

userId,

timestamp: new Date().toISOString()

})

})

})

* Sends the message to a workflow in n8n which returns a response like:

{

"response": "Here's what I found...",

"text": "...",

"output": "...",

}

* Fallbacks handle missing or malformed JSON by parsing responseText manually.
* Timeout: AbortSignal.timeout(30000) to prevent hangs.

## **📊 INSIGHTS ENDPOINT FLOW (n8n.tsx)**

### **✍️ Send Full Dataset Prompt**

const WEBHOOK\_URL = 'https://n8n-n91d.onrender.com/webhook/<uuid>/chat';

await fetch(WEBHOOK\_URL, {

method: 'POST',

body: JSON.stringify({

action: 'sendMessage',

sessionId: `session\_${Date.now()}`,

chatInput: fullPrompt

})

});

* Sends a long-form, structured prompt including:  
  + All metrics
  + Overall score
  + Top issues
* AI response is returned and formatted as insight sections.

## **📈 METRIC FLOW (page.tsx)**

### **🟤 Check MongoDB Cache**

await fetch(`/api/metrics/get?userId=${session.user.id}&datasetId=${dataset.\_id}`);

* If success → use cached metrics
* If not → fallback to Flask backend

### **🟤 Call Flask Model (CSV Input → Metrics)**

await fetch('https://metric-models-dataviz.onrender.com/analyze', {

method: 'POST',

body: JSON.stringify({

csvData,

datasetId,

datasetName

})

})

* Converts dataset into raw CSV.
* Sends to Flask ML API that returns result.metrics object.
* Example computed metrics:  
  + Missing\_Values\_Pct
  + Feature\_Correlation\_Mean
  + Data\_Density\_Completeness

### **🔵 Save Metrics**

await fetch('/api/metrics/save', {

method: 'POST',

body: JSON.stringify({

userId,

datasetId,

metrics,

timestamp

})

})

* Stores metrics to MongoDB with a user- and dataset-scoped ID.
* Shows success or error based on HTTP status.

### **🟣 Deep Learning Analysis**

await fetch('https://data-viz-ai-model.onrender.com/predict', {

method: 'POST',

body: JSON.stringify({ features })

})

* features: numerical representation of ~25 metrics
* Backend returns:  
  + overall\_score (0-100)
  + label\_probabilities (High, Medium, Low)
  + top\_issues (e.g., Missing\_Values\_Pct: 48)
* Results are visualized in a chart

## **📐 KEY FORMULAS**

### **📘 Metric Score Interpretation**

const rule = {

great: 0.95,

good: 0.85,

average: 0.70,

reverse: true

};

* reverse = true means lower is better (e.g., for Missing\_Values\_Pct)
* Grading: >= great → Great, >= good → Good, etc.

Color-coded via:  
  
 'text-green-600 bg-green-100' // for Great

## **🧪 Summary of Data Flow**

1. **Frontend** sends CSV/raw metrics → Flask or n8n
2. **Flask** processes raw data → returns metrics
3. **MongoDB** caches or fetches metrics
4. **n8n** handles AI-driven interpretation (both simple and full prompts)
5. **ML Server** uses metrics → returns score + label + insights

Would you like this entire section expanded into a .docx document or a diagram format? I can also walk you through one endpoint’s request/response cycle step-by-step with code.

Here’s a deep, structured breakdown (500+ lines of rich context) of how the uploaded model-train.py script works—including **how data flows**, **what models are trained**, and **formulas used internally**. This is suitable for full documentation or technical reporting.

## **🧠 Overview: What the Script Does**

This script:

* Loads the latest scored\_data\_quality\_\*.csv file from a folder
* Preprocesses data (scaling, encoding, splitting)
* Trains **three different models using TabNet**:  
  1. **Regressor** to predict an overall data quality score
  2. **Classifier** to assign a quality label (like "Good", "Poor")
  3. **Regressor** for predicting individual metric scores (like Missing\_Values\_Score)
* Evaluates all models using multiple performance metrics
* Saves models, test data, predictions, and a metadata file

## **📁 Step-by-Step Breakdown**

### **1. Data Loading**

scored\_files = [f for f in os.listdir(data\_path) if f.startswith("scored\_data\_quality\_")]

latest\_file = max(scored\_files)

df = pd.read\_csv(os.path.join(data\_path, latest\_file))

* Pulls the **latest CSV** with prefix scored\_data\_quality\_ to ensure model is always trained on newest version.

### **2. Feature Definition**

There are **26 primary input features** including:

| **Feature** | **Description** |
| --- | --- |
| Row\_Count | Number of rows in the dataset |
| File\_Size\_MB | Dataset file size in megabytes |
| Missing\_Values\_Pct | % of missing cells |
| Outlier\_Rate | % of outlier records |
| Data\_Type\_Mismatch\_Rate | Incorrect data types |
| Target\_Imbalance | Label distribution skew |
| *(+20 others...)* | Data integrity, density, freshness, domain constraint checks |

Also, for each of the 26 features, a **\_Score** variant exists, e.g.:

"Missing\_Values\_Pct" ➝ "Missing\_Values\_Pct\_Score"

### **3. Targets**

overall\_score\_target = "Calculated\_Data\_Quality\_Score"

label\_target = "Data\_Quality\_Label"

* **Regression Target**: A float between 0–100
* **Classification Target**: Label like "Excellent", "Good", "Bad", etc.

### **4. Preprocessing**

df.replace([np.inf, -np.inf], np.nan, inplace=True)

df.fillna(0, inplace=True)

scaler = StandardScaler()

df\_scaled[input\_features] = scaler.fit\_transform(df[input\_features])

* Handles infinities
* Imputes missing values with 0
* Scales features using **Z-score normalization**:

z=x−μσz = \frac{x - \mu}{\sigma}

Then, label is **encoded** into integers:

LabelEncoder().fit\_transform(df[label\_target])

### **5. Train/Test Split**

X\_train, X\_test, y\_score\_train, y\_score\_test, y\_label\_train, y\_label\_test = train\_test\_split(...)

Y\_metrics = df[metric\_score\_features].values

* 80/20 split
* Creates X/y sets for:  
  + Overall Score (Regression)
  + Label (Classification)
  + 26 Individual Metric Scores (Multivariate Regression)

## **🤖 Model Architecture**

All 3 models use **TabNet**, a deep learning architecture for tabular data.

TabNet characteristics:

* Uses **sequential attention** to choose features at each decision step
* Better interpretability and performance on tabular data

### **TabNet Hyperparameters**

TabNetRegressor/Classifier(

n\_d=64, n\_a=64, n\_steps=5,

gamma=1.5, n\_independent=2, n\_shared=2,

optimizer\_fn=torch.optim.Adam,

optimizer\_params=dict(lr=2e-2),

scheduler\_fn=torch.optim.lr\_scheduler.StepLR,

scheduler\_params={"step\_size": 10, "gamma": 0.9},

mask\_type='entmax'

)

* n\_d, n\_a: width of decision and attention layers
* n\_steps: number of decision steps
* gamma: relaxation factor
* optimizer: Adam
* scheduler: StepLR → multiplies LR by γ every 10 epochs

### **🧪 Training Logic**

Each model is trained using:

.fit(X\_train, y\_train, eval\_set=[(X\_test, y\_test)], max\_epochs=100, patience=10)

* Early stopping after 10 epochs without improvement
* Virtual batch size of 128 (for stability)
* Actual batch size: 1024

## **📈 Evaluation**

### **A. Overall Score Model (Regressor)**

y\_score\_pred = regressor.predict(X\_test)

Metrics:

MAE=1n∑∣y−y^∣\text{MAE} = \frac{1}{n} \sum |y - \hat{y}| MSE=1n∑(y−y^)2\text{MSE} = \frac{1}{n} \sum (y - \hat{y})^2 RMSE=MSE\text{RMSE} = \sqrt{\text{MSE}} R2=1−∑(y−y^)2∑(y−yˉ)2R^2 = 1 - \frac{\sum (y - \hat{y})^2}{\sum (y - \bar{y})^2}

### **B. Label Model (Classifier)**

y\_label\_pred = classifier.predict(X\_test)

Metrics:

* Accuracy
* Balanced Accuracy
* Precision
* Recall
* F1
* Cohen’s Kappa
* ROC AUC (multi-class OvR)
* Log Loss
* Hinge Loss
* Gini Coefficient:

Gini=2×AUC−1\text{Gini} = 2 \times \text{AUC} - 1

### **C. Metric Score Model (Multi-Output Regressor)**

Y\_metrics\_pred = metric\_regressor.predict(X\_test)

* For each of the 26 metric score features (e.g. Missing\_Values\_Pct\_Score)
* Evaluated with: MAE, MSE, RMSE, R²

## **💾 Outputs & Saving**

The script creates and saves:

| **File Type** | **Description** |
| --- | --- |
| .pkl | Trained models (3 total) |
| .csv | Predictions for scores, labels, metric scores |
| model\_performance\_metrics\_\*.csv | All performance metrics |
| model\_metadata\_\*.json | Input features, targets, model files, label classes, sample counts |

### **✅ Saved Artifacts**

* tabnet\_score\_model\_<timestamp>.pkl
* tabnet\_label\_model\_<timestamp>.pkl
* tabnet\_metric\_scores\_model\_<timestamp>.pkl
* feature\_scaler\_<timestamp>.pkl
* label\_encoder\_<timestamp>.pkl
* CSVs of X\_test, y\_test, predictions
* model\_metadata\_<timestamp>.json

## **📌 Summary: Model Roles**

| **Model** | **Type** | **Output** | **Purpose** |
| --- | --- | --- | --- |
| regressor | TabNetRegressor | Float [0–100] | Predicts overall quality score |
| classifier | TabNetClassifier | Label (str) | Categorizes dataset quality |
| metric\_regressor | TabNetRegressor | Vector of 26 floats | Predicts each metric’s individual score |

## **🔍 Suggestions for Extension**

* **SHAP analysis** on TabNet outputs for explainability
* **Model versioning** using MLflow or DVC
* Add support for **dataset drift monitoring**
* Auto-trigger retraining when new files appear
* Visualize **attention masks** to show which features matter per prediction

Let me know if you’d like this written as a PDF report or split into modular documentation blocks for a README, Wiki, or API doc.

Here is an **in-depth, real-time development-oriented breakdown** of how the **formulas, code logic, MongoDB**, and **n8n AI agent** all work together in your system. This includes code-level rationale, architectural purpose, and real-time usage insights.

## **📁 1. MongoDB — Caching + Persistence**

### **🔹 Where It’s Used:**

* /api/metrics/list-datasets
* /api/metrics/get?userId=X&datasetId=Y
* /api/metrics/save

### **🔹 Why MongoDB?**

* Fast NoSQL storage — ideal for schema-less metrics objects.
* Caching metrics reduces computation (no need to recalculate for each view).
* Enables user-specific scoped data (via userId, datasetId keys).

### **🔹 How It’s Used:**

In **page.tsx and chatbot.tsx**:

const mongoResponse = await fetch(`/api/metrics/get?userId=${userId}&datasetId=${datasetId}`);

### **📄 Example Document in MongoDB:**

{

"userId": "abc123",

"datasetId": "xyz456",

"metrics": {

"Row\_Count": 10000,

"Missing\_Values\_Pct": 4.2,

...

},

"timestamp": "2025-06-15T18:44:23Z"

}

### **🧠 Real-Time Benefit:**

When a user opens the dashboard or chatbot:

* First check: MongoDB cache (fast, no re-processing)
* If not found: trigger Flask backend to generate metrics (slower)

## **🤖 2. n8n AI Agent — Chat Workflow Logic**

### **🔹 What is n8n?**

* n8n is a **low-code workflow automation platform**.
* Think of it as a "customizable AI backend" via visual nodes.
* It hosts your AI chat logic without needing your own backend code.

### **🔹 Why Use n8n?**

* Custom workflows (preprocessing, agents, filters)
* No server maintenance
* Works well with OpenAI, webhooks, HTTP nodes

### **🔹 How It’s Triggered:**

const response = await fetch('https://n8n-n91d.onrender.com/webhook/<workflow-id>/chat', {

method: 'POST',

body: JSON.stringify({

action: 'sendMessage',

sessionId: `user\_dataset\_id`,

chatInput: "What are the main issues in my dataset?",

context: JSON.stringify({

datasetName,

metrics,

userId,

timestamp

})

})

})

### **🧠 Workflow Logic Inside n8n:**

1. Trigger: Webhook (receives user question + dataset context)
2. Run GPT-4: via OpenAI node, using prompt like:  
     
     
    "Based on the following metrics, suggest quality issues and modeling strategies:\n{metrics}"
3. Return structured response: response, message, output, text

### **🔄 Real-Time Response Handling:**

if (responseJson.output) {

botResponse = responseJson.output;

} else if (responseJson.text) {

botResponse = responseJson.text;

} else {

botResponse = JSON.stringify(responseJson, null, 2);

}

## **📐 3. Real-Time Metric Formulas & Interpretation**

### **🔹 Example: Missing Values**

Missing\_Values\_Pct = (Total Nulls in dataset / Total Cells) \* 100

If:

* 4 columns
* 1000 rows → 4000 total cells
* 168 cells are null → (168 / 4000) × 100 = 4.2%

This is auto-computed in Flask backend and stored.

### **🔹 Metric Scoring Formula (Frontend)**

const scoringRules = {

'Missing\_Values\_Pct': { great: 2, good: 5, average: 15, reverse: true }

};

const getScore = (value) => {

if (value <= 2) return 'Great';

else if (value <= 5) return 'Good';

else if (value <= 15) return 'Average';

return 'Poor';

};

reverse: true means lower = better (e.g. missing values, outliers, label noise)

### **🔹 Quality Labels from Model (/predict)**

Returned JSON from deep learning API:

{

"overall\_score": 72.5,

"quality\_label": "Good",

"label\_probabilities": {

"Low": 0.05,

"Medium": 0.22,

"High": 0.73

},

"top\_issues": {

"Outlier\_Rate": 0.15,

"Target\_Imbalance": 0.61,

"Missing\_Values\_Pct": 4.9

}

}

Used in visualizations and AI interpretation.

## **🔁 4. Workflow: Real-Time User Journey**

### **Step-by-Step:**

1. User opens dashboard
2. App fetches datasets from MongoDB (/api/metrics/list-datasets)
3. On dataset selection:  
   * Check MongoDB for metrics (/api/metrics/get)
   * If not found → send CSV to Flask (/analyze)
4. Metrics appear in UI + Chart.js
5. User clicks “AI Insights”:  
   * Send metrics + top issues to n8n (/webhook/...)
   * n8n returns GPT-4 response formatted into cards
6. Chatbot also available — passes messages to same n8n system with richer metadata

## **🧠 Summary: Component Logic by Role**

| **Component** | **Role** | **Technologies** | **Real-Time Use** |
| --- | --- | --- | --- |
| MongoDB | Cache + storage | NoSQL | Speeds up dashboard & chatbot |
| Flask API | Metrics Engine | Python, Pandas | Computes metric values |
| n8n | AI Agent | GPT-4, HTTP | Interprets metrics, answers questions |
| React UI | Frontend | Next.js, Chart.js | User controls, visual insights |
| ML Model API | Deep Scoring | TabNet or DNN | Predicts quality labels |

## **📤 Export Option**

Would you like me to compile this into:

* ✅ A .docx or .pdf?
* ✅ An architecture diagram (system workflow)?
* ✅ Full documentation for each endpoint with schemas and examples?

Let me know how you'd like to continue — I can generate any of the above instantly.

To double the depth of insight from your current setup, we need to **combine the model training pipeline (model-train.py) with the detailed metric extraction logic (wind.py)**, creating a unified, deeply integrated explanation of both **how the data is enriched** and **how the enriched data is used in predictive modeling**.

## **🧠 Complete Integrated System: 2× Deep Context Breakdown**

### **PART A — 📊 Data Enrichment via wind.py: Metric Engineering**

This script is responsible for computing **over 26 nuanced data quality metrics** from any raw dataset. These metrics are engineered features that later feed the machine learning models in model-train.py.

### **🔍 What Does wind.py Do?**

* Takes in one or more CSVs
* For each dataset:  
  + Detects column types (numeric, categorical, date)
  + Calculates metrics like:  
    - **Missing data %**
    - **Outlier rate** (via Z-score or Isolation Forest)
    - **Data type mismatch**
    - **Class overlap** (via silhouette score)
    - **Mean/Median drift**
    - **Anomaly count** (via Isolation Forest)
    - **Freshness** of date columns
    - **Label noise** (via KMeans clustering disagreement)
    - **Domain constraint violations**
  + Assigns a **Data Quality Score (0–100)** using weighted normalization

These metrics form the features like: Outlier\_Rate, Mean\_Median\_Drift, Encoding\_Coverage\_Rate, etc., which are then used by TabNet models.

### **📐 Example Formula: Outlier Rate**

Two approaches are used:

#### **A. Z-score for univariate:**

Z=x−μσ,Outliers: ∣Z∣>3Z = \frac{x - \mu}{\sigma},\quad \text{Outliers: } |Z| > 3

#### **B. Isolation Forest for multivariate:**

clf = IsolationForest()

outliers = clf.fit\_predict(df[numeric\_cols])

rate = (outliers == -1).sum() / len(df)

### **📈 Feature Importance Consistency**

Uses mutual information to compare stability of top features in two splits of data:

mutual\_info\_1 = mutual\_info\_classif(X1, y1)

mutual\_info\_2 = mutual\_info\_classif(X2, y2)

spearman\_corr = spearmanr(mutual\_info\_1, mutual\_info\_2)

### **🔗 Output of wind.py**

The result is a **single-row CSV file** with 30–35 enriched fields:

* Raw stats: row/col count, size
* Type counts: numeric, categorical, date
* 20+ metric columns
* A final "Calculated\_Data\_Quality\_Score" field

### **PART B — 🤖 Predictive Modeling via model-train.py**

Once the metrics are ready, they are passed into **three separate TabNet models**.

#### **Model Inputs**

X = df[input\_features] # 26 engineered metrics

y\_score = df["Calculated\_Data\_Quality\_Score"]

y\_label = df["Data\_Quality\_Label"]

Y\_metrics = df[[f"{col}\_Score" for col in input\_features]]

#### **Models:**

| **Model** | **Task** | **Output** |
| --- | --- | --- |
| regressor | Regression | Predicts the overall data quality score |
| classifier | Classification | Predicts the label (Good/Bad etc.) |
| metric\_regressor | Multi-target Regression | Predicts individual metric scores |

### **💡 Key Integrations & Enhancements**

| **Metric** | **Source** | **Use in Model** |
| --- | --- | --- |
| Outlier\_Rate | Z-Score / IsolationForest | Predicts Outlier\_Rate\_Score |
| Anomaly\_Count | IsolationForest | Score target |
| Target\_Imbalance | Label distribution entropy | Classification target feature |
| Encoding\_Coverage\_Rate | Categorical cardinality | Feature for label prediction |
| Label\_Noise\_Rate | KMeans Clustering | Measures label quality for Data\_Quality\_Label |
| Domain\_Constraint\_Violations | Rule-based | Input for Data\_Quality\_Score prediction |
| Data\_Freshness | Timestamp difference | Key for time-series degradation tracking |

## **🧬 Combined Formula for Final Quality Score**

Inside wind.py, the final Data\_Quality\_Score is computed via weighted average:

Final Score=∑metricwm⋅normalizedm\text{Final Score} = \sum\_{\text{metric}} w\_m \cdot \text{normalized}\_m

Example:

normalized['Missing\_Values\_Pct'] = 1 - (pct\_missing / 100)

normalized['Duplicate\_Records\_Count'] = 1 - (dupes / 100)

...

Final Score = ∑ normalized\_m × weight\_m

## **📁 What the Full Pipeline Looks Like**

Raw CSV

│

▼

wind.py ➜ calculate 26 metrics + score

│

▼

scored\_data\_quality\_YYYY.csv

│

▼

model-train.py ➜ train 3 TabNet models:

┌───────────────┬──────────────┬───────────────────────────────┐

│ Score Model │ Label Model │ Individual Metrics Model │

│ (Regression) │ (Classifier) │ (Multi-regression of scores) │

└───────────────┴──────────────┴───────────────────────────────┘

│

▼

y\_score\_pred.csv

y\_label\_pred.csv

y\_metrics\_pred.csv

## **📌 Final Result: Why This is “2× Data Deep”**

* **wind.py handles domain-aware metric engineering**, not just feature extraction
* **model-train.py applies deep neural tabular learning**, not traditional ML
* Both sides use **robust statistical techniques + explainability**
* Adds **label integrity detection, domain constraint logic, multivariate outlier detection**, etc.

Let me know if you want:

* Full merged code that feeds wind.py output directly into model-train.py
* Auto-pipeline with n8n webhooks or cron
* Explanation in PDF/Markdown for documentation or a video demo script

The provided Python script main-model-test.py is designed to test a data quality assessment model. It loads pre-trained models, makes predictions on sample input data, and generates detailed reports and visualizations of the results.

Here are 100 insights from the code, including explanations and relevant formulas where applicable:

1. **Purpose of the Script:** The script's primary goal is to perform a comprehensive test of a data quality model, evaluating its performance on a sample dataset and generating insightful outputs.
2. **Modular Design:** The code is well-structured into functions (e.g., find\_latest\_model\_files, load\_models, predict\_quality, create\_results\_table, create\_visualizations, save\_results\_to\_csv), promoting reusability and readability.
3. **Path Management:** It uses os.path for robust path handling, making it cross-platform compatible for file operations (e.g., os.path.join, os.makedirs).
4. **Dynamic Output Directory:** The script creates a time-stamped output directory (test\_output\_path) to store results, preventing overwriting previous test runs.  
   * test\_output\_path = r"c:\Users\ATTELLI SANJAY KUMAR\Desktop\New Version\Model-Test"
   * timestamp = datetime.now().strftime("%Y%m%d\_%H%M%S")
   * os.makedirs(test\_output\_path, exist\_ok=True)
5. **Timestamping:** All output files (CSV, PNG) are named with a timestamp to ensure uniqueness and traceability of test runs. This is crucial for logging and historical analysis.
6. **Warning Suppression:** warnings.filterwarnings('ignore') is used to suppress warnings, which can be useful in production environments but should be used with caution during development to avoid missing critical issues.
7. **Sample Input Features:** A dictionary input\_features defines a "High Quality Dataset" example with various data quality metrics. This serves as the input to the trained models.
8. **Comprehensive Feature Set:** The input\_features dictionary includes a wide array of data quality dimensions, such as:  
   * Row\_Count
   * Column\_Count
   * File\_Size\_MB
   * Numeric\_Columns\_Count
   * Categorical\_Columns\_Count
   * Date\_Columns\_Count
   * Missing\_Values\_Pct
   * Duplicate\_Records\_Count
   * Outlier\_Rate
   * Inconsistency\_Rate
   * Data\_Type\_Mismatch\_Rate
   * Null\_vs\_NaN\_Distribution
   * Cardinality\_Categorical
   * Target\_Imbalance
   * Feature\_Importance\_Consistency
   * Class\_Overlap\_Score
   * Label\_Noise\_Rate
   * Feature\_Correlation\_Mean
   * Range\_Violation\_Rate
   * Mean\_Median\_Drift
   * Data\_Freshness
   * Anomaly\_Count
   * Encoding\_Coverage\_Rate
   * Variance\_Threshold\_Check
   * Data\_Density\_Completeness
   * Domain\_Constraint\_Violations
9. **Single Test Case:** The test\_cases list currently contains only one test case: "High Quality Dataset". This implies the script is set up for evaluating one specific scenario at a time, though it could be extended to multiple.
10. **Model File Naming Convention:** The script expects model files (e.g., tabnet\_score\_model\_, tabnet\_label\_model\_) and metadata files (model\_metadata\_) to follow a specific naming convention that includes a timestamp. This allows the script to identify the latest trained models.
11. **Finding Latest Models (find\_latest\_model\_files):** This function iterates through the model\_path to find the most recent .pkl (for models and preprocessors) and .json (for metadata) files based on their timestamps.
12. **Metadata-Driven Model Loading:** The script first identifies the latest model\_metadata\_\*.json file to extract the associated timestamp, ensuring that it loads all model components (score model, label model, metric model, scaler, label encoder) from the same training run.
13. **Robust File Handling:** try-except blocks are used extensively for error handling during file operations (e.g., FileNotFoundError if models are not found).
14. **Pickle for Model Serialization:** pickle is used to load the trained machine learning models (.pkl files), which is a common practice for serializing Python objects.
15. **JSON for Metadata:** json is used to load the model\_metadata\_\*.json file, which likely contains crucial information such as the order of input features (input\_features), metric score features (metric\_score\_features), and quality label classes (label\_classes).
16. **Loading Multiple Models (load\_models):** The load\_models function is responsible for loading all required components:  
    * score\_model: Predicts the overall quality score.
    * label\_model: Predicts the quality label (e.g., Excellent, Good, Moderate, Poor).
    * metric\_model: Predicts individual metric scores.
    * scaler: For feature scaling (e.g., StandardScaler, MinMaxScaler).
    * label\_encoder: For inverse transforming predicted labels back to human-readable strings.
    * metadata: For understanding the model's structure and expected inputs.
17. **TabNet Models:** The comments and file names suggest that the underlying machine learning models are likely TabNet models, suitable for tabular data.
18. **Feature Order from Metadata:** Crucially, the predict\_quality function retrieves the expected feature\_order from the loaded metadata. This ensures that the input features are provided to the model in the exact order it was trained on.
19. **Input Feature Validation:** The predict\_quality function includes a check to ensure that all required features (as defined in the model's metadata) are present in the input features\_dict.
20. **Feature Scaling:** Before making predictions, the input features\_array is transformed using the loaded scaler. This is a critical preprocessing step for many machine learning models to normalize feature ranges.  
    * features\_scaled = models['scaler'].transform(features\_array)
21. **Prediction Types:** The predict\_quality function generates three main types of predictions:  
    * overall\_score: A continuous numerical score (0-100).
    * quality\_label: A categorical label (e.g., "Excellent", "Good").
    * metric\_scores: Individual scores for each data quality metric.
22. **Label Encoding and Decoding:** The label\_model predicts encoded labels, which are then converted back to human-readable labels using the label\_encoder.inverse\_transform().
23. **Probability Scores for Labels:** The label\_model.predict\_proba() method is used to obtain probability distributions for each quality label, indicating the model's confidence in its label prediction.  
    * P(labeli​∣features)
24. **Metric Score Naming:** The metric\_scores are processed to remove the \_Score suffix for cleaner presentation (e.g., Completeness\_Score becomes Completeness).
25. **Prediction Timestamp:** Each prediction result includes a prediction\_time for tracking when the prediction was made.
26. **Tabulate for Results Display:** The tabulate library is used to create well-formatted, ASCII-art tables for displaying the test results summary in the console, enhancing readability.
27. **Identifying Top Metric Issues:** The create\_results\_table function identifies the top 3 "worst" performing metrics by sorting them in ascending order of their scores.
28. **Confidence Metric:** The "Confidence" in the results table is derived from the highest probability among the predicted quality labels.  
    * Confidence=max(P(labeli​∣features))
29. **Matplotlib and Seaborn for Visualizations:** matplotlib.pyplot and seaborn are used to create informative visualizations of the test results, providing a graphical overview.
30. **Comprehensive Visualization Layout:** The create\_visualizations function generates a 2x2 subplot layout to present different aspects of the results:  
    * Overall Quality Score
    * Quality Label Probabilities (Pie Chart)
    * Top 10 Worst Performing Metrics (Horizontal Bar Chart)
    * Bottom 15 Metric Scores (Heatmap)
31. **Color Coding for Quality:** Visualizations use distinct colors (e.g., green for Excellent, red for Poor) to intuitively represent different quality levels.
32. **Quality Score Thresholds:** The "Overall Quality Score" plot includes horizontal dashed lines at specific thresholds (85, 70, 50) to visually represent "Excellent," "Good," and "Moderate" quality ranges.  
    * Excellent: Score ≥85
    * Good: 70≤Score<85
    * Moderate: 50≤Score<70
    * Poor: Score <50
33. **Pie Chart for Label Distribution:** A pie chart effectively visualizes the probabilistic distribution across different quality labels.
34. **Horizontal Bar Chart for Metrics:** A horizontal bar chart is used to display the scores of the worst-performing metrics, making it easy to compare and identify areas for improvement.
35. **Heatmap for Metric Overview:** A heatmap provides a compact visual representation of multiple metric scores, with color intensity indicating the score value. The RdYlGn colormap (Red-Yellow-Green) is typically used to indicate bad-to-good ranges.
36. **Saving Visualizations:** The generated plot is saved as a PNG image with a timestamp, ensuring a visual record of each test run.  
    * plt.savefig(os.path.join(output\_path, f'model\_test\_results\_{timestamp}.png'), dpi=300, bbox\_inches='tight')
37. **Detailed CSV Output:** The save\_results\_to\_csv function generates three separate CSV files for detailed results:  
    * test\_summary\_\*.csv: High-level summary of the test case.
    * detailed\_metrics\_\*.csv: Scores for all individual metrics, sorted by score.
    * input\_features\_\*.csv: The exact input features used for the test.
38. **DataFrames for CSV Output:** pandas.DataFrame is used to structure the data before saving it to CSV, which is a standard practice for tabular data manipulation.
39. **Sorting Detailed Metrics:** The detailed\_metrics\_df is sorted by Score in ascending order, making it easy to immediately identify the metrics with the lowest scores.
40. **Quality Rating for Each Metric:** Each individual metric in the detailed\_metrics.csv is assigned a "Quality\_Rating" based on the same thresholds used for the overall score, providing granular quality assessment.
41. **Main Execution Block:** The if \_\_name\_\_ == "\_\_main\_\_": block ensures that the main() function is called only when the script is executed directly.
42. **Start and End Time Tracking:** The main function tracks the start\_time and end\_time to calculate the total execution\_time of the testing process.
43. **Pre-Run Checks:** Before proceeding with predictions, the main function performs crucial checks:  
    * Are model files found?
    * Are models loaded successfully?
    * Are all required input features present in the test case?
44. **Informative Print Statements:** The script uses numerous print statements to provide clear feedback on the progress, status, and results of the testing process.
45. **Clarity in Console Output:** The use of print("="\*80) and print("-" \* 50) creates visual separators in the console output, making it easier to read and parse the information.
46. **Detailed Analysis Section:** The "DETAILED ANALYSIS" section in the console output provides a concise breakdown of the overall score, quality label, label probabilities, and top/bottom performing metrics.
47. **Top 5 Worst/Best Metrics:** The script explicitly identifies and prints the top 5 worst and best performing metrics, highlighting areas that need most attention or are performing exceptionally well.
48. **Model Agnosticism (to some extent):** While the models are referred to as "TabNet," the predict\_quality function itself is quite generic. As long as the loaded models have predict and predict\_proba methods and the scaler has a transform method, the core prediction logic would work with other model types as well.
49. **Extensibility:** The test\_cases list can be easily extended to include more test scenarios (e.g., "Poor Quality Dataset", "Moderate Quality Dataset") to thoroughly evaluate the model's performance under different conditions.
50. **Dependency Management:** The script clearly lists its dependencies at the top: pandas, numpy, matplotlib, seaborn, os, pickle, json, datetime, tabulate.
51. **Relative Paths vs. Absolute Paths (Consideration):** While absolute paths are used (r"c:\Users\..."), in a more production-ready setup, relative paths or environment variables might be preferred for better portability.
52. **Error Handling for Missing Models:** The find\_latest\_model\_files function explicitly raises a FileNotFoundError if no model or metadata files are found, preventing further execution with missing components.
53. **Error Logging:** Although not explicitly written to a log file, the print(f"❌ Error...") statements serve as basic error logging to the console. In a production system, these errors would typically be routed to a dedicated logging framework.
54. **Scalability for Multiple Test Cases:** While currently configured for a single test case, the loop for test\_case in test\_cases: in the main function is designed to iterate through multiple test cases, allowing for future expansion.
55. **Clear Variable Naming:** Variable names are descriptive and contribute to code readability (e.g., overall\_score\_model, label\_encoder, test\_results).
56. **Data Freshness Metric:** The Data\_Freshness feature suggests that the model can account for the recency or up-to-dateness of the data. This is crucial for real-time or near real-time data quality monitoring.  
    * Potentially, Data Freshness=1−Max acceptable lagTime since last update​.
57. **Anomaly Count Feature:** The Anomaly\_Count feature implies that the model considers the presence of outliers or unusual data points as a factor in data quality.
58. **Encoding Coverage Rate:** This metric suggests the model considers how well categorical features are encoded, possibly indicating the completeness of mapping or the presence of unmapped values.  
    * Encoding Coverage Rate=Total number of valuesNumber of mapped values​.
59. **Variance Threshold Check:** This feature (Variance\_Threshold\_Check) might relate to feature engineering or selection, ensuring that features have sufficient variance to be informative. High variance usually indicates better information.  
    * Variance=n−1∑i=1n​(xi​−xˉ)2​ for a sample. A high value is generally better for model input.
60. **Data Density Completeness:** This metric likely refers to how densely populated the feature space is, or the inverse of sparseness. Higher density usually means more complete data.
61. **Domain Constraint Violations:** This indicates that the model can identify instances where data violates predefined business rules or domain-specific constraints (e.g., age must be positive, prices within a certain range).
62. **Mean-Median Drift:** This metric can indicate changes in the distribution of a feature over time, which is a key aspect of data drift detection.  
    * Drift=∣Mean1​−Mean2​∣+∣Median1​−Median2​∣ (or a more sophisticated statistical test). A lower drift is better.
63. **Feature Correlation Mean:** This metric suggests that the model assesses the average correlation between features. Lower average correlation often implies less multicollinearity and more independent information from features.  
    * Mean Correlation=N1​∑i=j​∣Corr(Xi​,Xj​)∣
64. **Null vs. NaN Distribution:** This could be a metric to differentiate between explicitly Null values and NaN (Not a Number) representations of missing data, potentially indicating data cleaning consistency.  
    * Perhaps a ratio: Count of Null+Count of NaNCount of Null​ or a similar measure.
65. **Cardinality Categorical:** This refers to the number of unique values in categorical columns. High cardinality can sometimes indicate issues or require specific handling. The desired value depends on context.
66. **Target Imbalance:** This metric is crucial for classification problems, indicating if one class significantly outnumbers others in the target variable. A value close to 0.5 (for binary classification) or a balanced distribution for multi-class is generally better.  
    * For binary, Imbalance=min(proportion of class 0,proportion of class 1).
67. **Feature Importance Consistency:** This metric suggests that the model might be evaluating how consistently feature importances behave, potentially across different subsets of data or model runs. Higher consistency is better.
68. **Class Overlap Score:** This metric is relevant for classification tasks, indicating how much the feature distributions of different classes overlap. Lower overlap generally means better separability and easier classification.
69. **Label Noise Rate:** This metric assesses the presence of incorrect or erroneous labels in the training data, which can significantly impact model performance. Lower noise is better.
70. **Range Violation Rate:** This metric quantifies how many data points fall outside expected or defined ranges for numerical features.  
    * Range Violation Rate=Total count of valuesCount of values outside valid range​. Lower is better.
71. **Outlier Rate:** The proportion of data points identified as outliers.  
    * Outlier Rate=Total Number of RecordsNumber of Outliers​. Lower is better.
72. **Inconsistency Rate:** This could refer to logical inconsistencies within the data (e.g., a person's age is less than their years of experience).  
    * Inconsistency Rate=Total RecordsNumber of Inconsistent Records​. Lower is better.
73. **Data Type Mismatch Rate:** This measures the percentage of values that do not conform to their expected data type.  
    * Data Type Mismatch Rate=Total ValuesNumber of Type Mismatches​. Lower is better.
74. **Missing Values Percentage:**
    * Missing Values Pct=Total Number of CellsCount of Missing Values​×100%. Lower is better.
75. **Duplicate Records Count:**
    * Duplicate Records Count=Number of identical rows. Lower is better.
76. **File Size (MB) as a Feature:** Including File\_Size\_MB as a feature might be a proxy for data volume, which can sometimes correlate with data quality challenges.
77. **Column Count as a Feature:** The number of columns can influence data complexity and potential for quality issues.
78. **Row Count as a Feature:** Similar to column count, higher row counts might introduce more opportunities for data quality problems if not managed well.
79. **Numeric, Categorical, Date Column Counts:** These features provide a structural understanding of the dataset, which can be relevant for data quality assessment.
80. **Performance Metric Display:** The execution\_time calculation provides a quick performance indicator for the testing script itself.
81. **Clear Output Summary:** The final print statements clearly list all generated output files, making it easy for the user to locate the results.
82. **Assumed Pre-trained Models:** The script assumes that the models and preprocessors (.pkl and .json files) already exist in the specified model\_path. It does not include any training logic.
83. **Standard Machine Learning Workflow:** The overall flow follows a typical machine learning inference pipeline: load models -> preprocess input -> make predictions -> interpret results.
84. **Data Quality Model (Implied):** The script operates on the premise that a "Data Quality Model" has been trained to predict an overall score, a quality label, and individual metric scores based on the provided data characteristics (features).
85. **Input Feature Engineering:** The input\_features dictionary represents the "engineered" features that describe the quality characteristics of a dataset. This implies a preceding step where these features are extracted from raw data.
86. **Potential for Integration:** This script could be integrated into a larger data pipeline for automated data quality checks (e.g., as part of a CI/CD process for data).
87. **Interactive vs. Scripted:** The script is designed for batch execution and reporting, rather than interactive analysis.
88. **Hardcoded Paths (Area for Improvement):** The model\_path and test\_output\_path are hardcoded. For deployment, these would typically be configurable via environment variables or command-line arguments.
89. **Single Test Case Limitation (Current):** While designed for multiple, its current configuration tests only one input\_features set. This could be expanded for more rigorous testing.
90. **No Data Loading Logic:** The script does not load actual raw data. It relies on pre-computed data quality metrics provided as input\_features.
91. **Model Explainability (Limited):** While it identifies worst-performing metrics, it doesn't provide deeper explanations on *why* a specific data point led to a low score for a metric (e.g., specific missing values causing low completeness). This would require more advanced explainability techniques.
92. **No Model Retraining:** This is purely an inference/testing script; it does not contain any code to retrain or update the models based on new data or feedback.
93. **Feature Scaling Importance:** The presence of feature\_scaler.pkl and its usage emphasizes the importance of consistent data scaling between training and inference.
94. **Inverse Transformation:** The label\_encoder.inverse\_transform() is essential for converting numerical predictions back into meaningful category names.
95. **Model Metadata Importance:** The model\_metadata\_\*.json file is critical as it defines the expected structure of the input features and output metrics, ensuring compatibility between the training and testing phases.
96. **Readability of Numeric Output:** The use of :.2f and :.1f in f-strings ensures that numerical scores are formatted to a readable number of decimal places.
97. **Robustness of find\_latest\_model\_files:** The function relies on finding model\_metadata\_ files first to ensure it's picking up a consistent set of models from the same training run, which is a good practice.
98. **Clear Start/End Messages:** The print statements "📊 Data Quality Model Testing..." at the start and "✅ Model testing completed successfully..." at the end provide clear markers for the execution.
99. **Potential for Threshold Customization:** While thresholds for quality labels (85, 70, 50) are hardcoded for visualization, in a more flexible system, these could be configurable parameters.
100. **Comprehensive Reporting:** The combination of console output, tabular summaries, detailed CSV files, and visualizations provides a very comprehensive reporting mechanism for data quality model testing.  
     Perfect! Let’s dive deep into **each data quality metric** from wind.py, explain its **definition, formula, statistical reasoning, and logic**, and describe **how it contributes to the overall data quality score**.

# **🧠 In-Depth Metric-by-Metric Explanation (with Formulas)**

## **1. Missing\_Values\_Pct**

**What it means:** Percentage of missing (null/NaN) values in the dataset.

**Formula:**

Missing\_Values\_Pct=Total Missing CellsTotal Cells×100\text{Missing\\_Values\\_Pct} = \frac{\text{Total Missing Cells}}{\text{Total Cells}} \times 100

**Example:** If 100 out of 10,000 cells are missing:

10010,000×100=1%\frac{100}{10,000} \times 100 = 1\%

**Impact:** More missing values → lower data completeness → lower quality.

## **2. Duplicate\_Records\_Count**

**What it means:** Number of rows that are exact duplicates.

**Formula:**

Duplicate\_Records\_Count=Number of rows x where df.duplicated(x)=True\text{Duplicate\\_Records\\_Count} = \text{Number of rows } x \text{ where } df.duplicated(x) = \text{True}

**Impact:** Redundant data inflates dataset, corrupts analysis.

## **3. Outlier\_Rate**

**What it means:** Fraction of rows with extreme/unusual values.

**Approach A: Z-score (for 1D)**

z=x−μσ,Outliers: ∣z∣>3z = \frac{x - \mu}{\sigma}, \quad \text{Outliers: } |z| > 3

**Approach B: IsolationForest (for multi-feature)** IsolationForest detects isolated samples in multidimensional space.

**Formula:**

Outlier\_Rate=Count of OutliersTotal Records\text{Outlier\\_Rate} = \frac{\text{Count of Outliers}}{\text{Total Records}}

## **4. Inconsistency\_Rate**

**What it means:** Proportion of values violating logical assumptions.

**Logic-based Checks:**

* Age < 0 or > 120
* Percentage > 100
* Score < 0 or > 100

**Formula (simplified):**

Inconsistency\_Rate=ViolationsTotal Checked Values\text{Inconsistency\\_Rate} = \frac{\text{Violations}}{\text{Total Checked Values}}

## **5. Data\_Type\_Mismatch\_Rate**

**What it means:** Fraction of values that don’t match expected type.

**Example:**

* A string in a numeric column
* A malformed date in a datetime column

**Formula:**

Mismatch\_Rate=Non-conformant ValuesTotal Checked Values\text{Mismatch\\_Rate} = \frac{\text{Non-conformant Values}}{\text{Total Checked Values}}

## **6. Null\_vs\_NaN\_Distribution**

**What it means:** Proportion of missing values that are NaN (vs empty string "").

**Formula:**

NaN CountNaN Count + Empty String Count\frac{\text{NaN Count}}{\text{NaN Count + Empty String Count}}

## **7. Cardinality\_Categorical**

**What it means:** Average number of unique values per categorical column.

**Formula:**

Avg. Cardinality=1n∑i=1nunique(coli)\text{Avg. Cardinality} = \frac{1}{n} \sum\_{i=1}^{n} \text{unique}(col\_i)

**Impact:**

* Too high → overfitting, encoding issues
* Too low → not informative

## **8. Target\_Imbalance**

**What it means:** Measures label imbalance in classification.

**For binary target:**

Imbalance=2×min(class count)total samples\text{Imbalance} = 2 \times \frac{\text{min(class count)}}{\text{total samples}}

**For multi-class (Normalized Entropy):**

Entropy=−∑pilog⁡2(pi),Normalized: Entropylog⁡2(K)\text{Entropy} = -\sum p\_i \log\_2(p\_i),\quad \text{Normalized: } \frac{\text{Entropy}}{\log\_2(K)}

## **9. Feature\_Correlation\_Mean**

**What it means:** Average absolute pairwise Pearson correlation.

**Formula:**

Mean Corr=∑i<j∣corr(Xi,Xj)∣(n2)\text{Mean Corr} = \frac{\sum\_{i<j} |\text{corr}(X\_i, X\_j)|}{\binom{n}{2}}

**Impact:** High correlation → multicollinearity → redundant features.

## **10. Range\_Violation\_Rate**

**What it means:** Proportion of numeric values outside 3-sigma bounds.

**Formula:**

Violation Rate=Count(x<μ−3σ or x>μ+3σ)Total Values\text{Violation Rate} = \frac{\text{Count}(x < \mu - 3\sigma \text{ or } x > \mu + 3\sigma)}{\text{Total Values}}

## **11. Mean\_Median\_Drift**

**What it means:** Indicates skewness in numeric distributions.

**Formula:**

Drift=∣mean−median∣∣median∣+ϵ\text{Drift} = \frac{|mean - median|}{|median| + \epsilon}

**Avg Drift:**

1n∑i=1nDrifti\frac{1}{n} \sum\_{i=1}^n \text{Drift}\_i

## **12. Class\_Overlap\_Score**

**What it means:** Measures how separable the classes are.

**Technique:** Silhouette Score

Silhouette=b−amax⁡(a,b)\text{Silhouette} = \frac{b - a}{\max(a, b)} Normalized Score=Silhouette+12\text{Normalized Score} = \frac{\text{Silhouette} + 1}{2}

Where:

* a: intra-cluster distance
* b: nearest-cluster distance

## **13. Data\_Freshness**

**What it means:** Days since the most recent timestamp in dataset.

**Formula:**

Freshness=today−max⁡(date columns)\text{Freshness} = \text{today} - \max(\text{date columns})

## **14. Feature\_Importance\_Consistency**

**What it means:** Measures stability of feature importance.

**Formula:**

* Split data in half
* Use mutual\_info\_classif() on both
* Compute **Spearman Rank Correlation**

Normalized Score=Spearman Correlation+12\text{Normalized Score} = \frac{\text{Spearman Correlation} + 1}{2}

## **15. Anomaly\_Count**

**What it means:** Total anomalies detected via IsolationForest.

**Formula:**

\text{Anomaly\\_Count} = \text{# of rows with prediction = -1}

## **16. Encoding\_Coverage\_Rate**

**What it means:** Fraction of categories covered by top 80% of values.

**Formula:**

\text{Coverage} = \frac{\text{Top categories covering 80%}}{\text{Total unique categories}}

## **17. Variance\_Threshold\_Check**

**What it means:** Proportion of features with near-zero variance.

**Formula:**

Low-Variance Proportion=#vars with σ2<ϵTotal Numeric Columns\text{Low-Variance Proportion} = \frac{\# \text{vars with } \sigma^2 < \epsilon}{\text{Total Numeric Columns}}

## **18. Data\_Density\_Completeness**

**What it means:** Average % of non-null cells per row and per column.

**Formula:**

Density=12(mean row completeness+mean column completeness)\text{Density} = \frac{1}{2} \left( \text{mean row completeness} + \text{mean column completeness} \right)

## **19. Label\_Noise\_Rate**

**What it means:** Approximates mislabeled examples via clustering.

**Steps:**

* Run KMeans on features
* Compare cluster vs true labels
* Estimate disagreement:

Noise Rate=1−∑max(class count per cluster)Total Samples\text{Noise Rate} = 1 - \frac{\sum \text{max(class count per cluster)}}{\text{Total Samples}}

## **20. Domain\_Constraint\_Violations**

**What it means:** Violations of business rules.

**Examples:**

* Age < 0
* Email without @
* Phone format incorrect
* Future dates in date fields

**Formula:**

Violation Rate=ViolationsTotal Checked Fields\text{Violation Rate} = \frac{\text{Violations}}{\text{Total Checked Fields}}

## **🎯 Final: Data\_Quality\_Score**

**What it means:** A weighted score from all metrics (scaled to 0–100).

### **Formula:**

Score=∑metricwm⋅normalizedm\text{Score} = \sum\_{\text{metric}} w\_m \cdot \text{normalized}\_m

Where each metric is scaled:

* Good → 1
* Bad → 0

Then:

Final Score (0–100)=Score×100\text{Final Score (0–100)} = \text{Score} \times 100

Would you like this:

* In Markdown for documentation?
* Exported as CSV or Excel with formulas?
* Visualized as a PDF or infographic with icons and graphs?