# Automated Insight Engine - Complete Project Explanation

## 📋 Project Overview

The **Automated Insight Engine (AIE)** is a web-based machine learning application built with **Streamlit** that automatically analyzes datasets and builds baseline ML models without requiring any coding. It's designed for data scientists and analysts who want to quickly understand their data and test different ML models.

### Key Purpose

- Upload a CSV file
- Automatically understand the data
- Clean the data
- Test multiple ML models
- Get performance comparisons
- Make predictions

## 🏗️ Project Architecture - Layered Design

The project is organized into **7 main layers**, each handling a specific phase of the ML workflow:

```
┌─────────────────────────────────────────────────────────┐
│              USER INTERFACE (Streamlit)            │
│                 (app.py)                    │
└────────────────────────┬────────────────────────────────┘
```

```
            │
   ┌────────────────┴──────────────┐
   │                │
┌───▼──────────────┐  ┌─────▼───────────┐
│ Layer 1: UNDERSTANDING  │ Layer 2: DIAGNOSIS │
│ (understanding.py)    │ (diagnostics.py)  │
│          │         │
│ - Detect data types   │ - Missing values   │
│ - Find target variable │ - Outliers      │
│ - Identify ML task   │ - Duplicates     │
│ - Check imbalance    │ - Correlations    │
│ - Flag data leakage  │          │
└───┬───────────────┘ └─────┬──────────┘
   │               │
┌───▼───────────────────────────────▼──┐
│    Layer 3: CLEANING          │
│    (cleaning.py)           │
│                 │
│ - Remove duplicates          │
│ - Fill missing values         │
│ - Remove sparse columns          │
│ - Cap outliers            │
│ - Create missing indicators        │
└───┬───────────────────────────────┘
   │
┌───▼───────────────────────────────┐
│    Layer 4: FEATURE SELECTION      │
│    (selection.py)           │
│                        │
│ - Filter methods (Variance, Mutual Info) │
│ - Embedded methods (Random Forest, L1)   │
│ - Wrapper methods (RFE)                  │
└─────┬───────────────────────────────────┘
      │
┌─────▼───────────────────────────────────┐
│    Layer 5: MODEL SELECTION              │
│    (models.py)                           │
│                        │
│ - Logistic Regression                    │
│ - Decision Trees                         │
│ - Random Forest                          │
│ - XGBoost & LightGBM                     │
│ - KNN, SVM, Neural Networks              │
└─────┬───────────────────────────────────┘
      │
┌─────▼───────────────────────────────────┐
│    Layer 6: ORCHESTRATION                │
│    (runner.py)                           │
│                        │
│ - Prepare features for training          │
│ - Encode targets                         │
│ - Coordinate pipeline                    │
└─────┬───────────────────────────────────┘
      │
┌─────▼───────────────────────────────────┐
│    Layer 7: EVALUATION                   │
│      (evaluation.py)            │
│                          │
│ - Cross-validation              │
│ - Performance metrics              │
│ - Leaderboard ranking              │
└─────────────────────────────────┘
```

## 📁 File-by-File Explanation

1️⃣ **understanding.py** - Data Understanding Layer

What it does: Analyzes the structure of your data to understand what you're working with.

Key Functions:

#### `infer_schema(df)`

- Detects column data types: numeric, categorical, datetime, boolean
- Separates different types of data for appropriate handling
- Identifies text columns that are actually numbers

Example:

```python
# Input: DataFrame with mixed columns
# Output:
```

```
# - numeric: ['age', 'salary', 'score']

# - categorical: ['color', 'department', 'city']

# - datetime: ['birth_date', 'join_date']
```

#### `infer_target(df)` & `infer_task(df, target)`

- Automatically finds which column is the target variable

- Determines if it's a **classification** task (predicting categories) or **regression** task (predicting numbers)

- Examples:

  - Classification: Predicting "Yes/No", "Cat/Dog", "A/B/C"

  - Regression: Predicting house price, temperature, salary

#### `detect_imbalance(y, task, threshold=0.8)`

- Checks if one class has way more samples than others

- Important because models struggle when classes are imbalanced

- Example: If 95% samples are "No" and 5% are "Yes", that's imbalanced

#### `detect_leakage(X, y, task)`

- Flags features that are too correlated with the target

- These can cause overfitting (model learns coincidences, not patterns)

- Example: If predicting house price and a feature is "appraised_value", it's data leakage

Output Structure: `UnderstandingSummary` - contains all findings about your data

---

2️⃣ **diagnostics.py** - Data Diagnosis Layer

What it does: Creates a "health report" of your data before cleaning.

Key Functions:

#### `missing_values_report(df)`

- Shows which columns have missing values (NaN)
- Reports the percentage missing
- Creates a visual heatmap of missing patterns
- Helps identify if missingness is random or systematic

Example Output:

```
Column       | Missing Count | Missing %
────────────────────────────────────────
age          | 5             | 2.5%
email        | 45            | 22.5%
phone        | 180           | 90%
```

#### `duplicate_report(df)`

- Counts how many exact duplicate rows exist
- Calculates duplicate percentage
- Important: Duplicates can skew model performance

#### `outlier_report(df, numeric_cols)`

- Detects unusual values in numeric columns using IQR (Interquartile Range) method

- IQR = distance between 25th and 75th percentile

- Values beyond 1.5 × IQR are flagged as outliers

- Examples: A salary of $10M when average is $100K

#### `correlation_analysis(df)`

- Finds numeric columns that are highly correlated

- Pearson correlation: measures linear relationship (-1 to +1)

- High correlation means two features have redundant information

- Example: "Height in cm" and "Height in inches" are perfectly correlated

#### `categorical_correlation(df, cat_cols)`

- Finds categorical columns that are related using Cramer's V test

- Similar to correlation but for category data

- Example: "Country" and "Language" might be correlated

Output Structure: `DiagnosisSummary` - contains all data health findings

3️⃣ **cleaning.py** - Data Cleaning Layer

What it does: Automatically fixes data quality issues. Works like sklearn transformers (can be chained in pipelines).

Key Classes:

#### `DuplicateRemover`

- Removes exact duplicate rows

- Keeps first occurrence, deletes rest

#### `NumericImputer`

- Fills missing values in numeric columns
- Strategies:
  - **Median**: Takes middle value (robust to outliers)
  - **Mean**: Takes average (affected by outliers)
  - **KNN**: Uses K nearest neighbors' values (sophisticated)

Example:

```
Original: [1, 2, NaN, 4, 5]
After imputation (median): [1, 2, 3, 4, 5]
```

#### `CategoricalImputer`

- Fills missing values in text/category columns
- Default: Replaces with "Unknown"
- Can also use mode (most common value)

#### `OutlierCapper`

- Reduces impact of outliers by capping them
- Example: Values > 99th percentile set to 99th percentile value
- Prevents extreme values from distorting models

#### `SparsityDropper`

- Removes columns with too many missing values
- Default: Drop if >50% missing

- These columns have little usable information

#### `MissingIndicator`

- Creates new binary columns marking where data was missing

- Useful because "missingness itself" can be predictive

- Example: If a customer didn't provide phone number, they might be less engaged

Example Cleaning Pipeline:

```
Raw Data

 ↓

Remove Duplicates

 ↓

Drop Sparse Columns (>50% missing)

 ↓

Fill Missing Numbers (median)

 ↓

Fill Missing Categories (Unknown)

 ↓

Cap Outliers

 ↓

Create Missing Indicators

 ↓

Clean Data (ready for models)
```

4️⃣ **selection.py** - Feature Selection Layer

What it does: Identifies which features (columns) are actually useful for predictions.

Why it matters: Using unnecessary features:

- Slows down training

- Makes models overfit (memorize noise)

- Makes models harder to understand

Three Main Approaches:

#### A. **Filter Methods** - Fast statistical approach

- **Variance Threshold**: Removes features with very low variance

- **Mutual Information**: Measures how much a feature tells about the target

- **Chi-Square**: For categorical features, tests relationship with target

Simple analogy: If a feature is nearly constant (always same value), it doesn't help prediction.

#### B. **Embedded Methods** - Built into the model

- **L1 Regularization (Lasso)**: Linear model that naturally removes weak features

- **Random Forest Importance**: Ranks features by how often they split the data

How it works: Random Forest trains and measures how much each feature improves decisions

#### C. **Wrapper Methods** - Test subsets

- **RFE (Recursive Feature Elimination)**:

1. Train model with all features
2. Remove weakest feature
3. Repeat until desired number remains

Trade-off: Wrapper is slowest but most accurate

Output: List of selected features + importance scores

5️⃣ **models.py** - Model Zoo Layer

What it does: Provides a library of 9 classification and 8 regression models with tuned parameters.

Classification Models (predict categories):

Model	When to Use	Pros	Cons
Logistic Regression	Baseline, interpretable	Fast, simple	Limited for complex patterns
Decision Tree	Interpretable	Handles non-linearity, no scaling needed	Overfits easily
Random Forest	Most robust	Powerful, handles nonlinearity	Less interpretable
KNN	Small datasets	Simple, adapts well	Slow on large data
SVM	High-dimensional data	Powerful, memory efficient	Slow to train
XGBoost	Best performance	Extremely powerful, fast	Complex, needs tuning
LightGBM	Huge datasets	Very fast	Less stable than XGBoost

Neural Network (MLP)	Complex patterns	Universal approximator	Black box, needs lots of data
Naive Bayes	Text/spam	Simple, fast	Assumes independence

Regression Models (predict numbers):

Model	Purpose
Linear Regression	Baseline
Decision Tree Regressor	Non-linear baseline
Random Forest Regressor	Robust predictions
KNN Regressor	Local average
SVM Regressor	Non-linear relationships
XGBoost Regressor	High-precision predictions
LightGBM Regressor	Fast large-scale
Neural Network Regressor	Complex patterns

Model Selection Logic:

```
if dataset is small (< 1000 rows):
  use simpler models (Logistic Regression, Decision Tree)
else if dataset is medium (1000-100K rows):
  use balanced models (Random Forest, XGBoost)
else if dataset is huge (>100K rows):
  use fast models (LightGBM, Linear Regression)

if many categorical features:
  prefer tree-based models (Random Forest, XGBoost)
```

else:

  can use any model

```

---

### 6️⃣ **runner.py** - Orchestration Layer

**What it does:** Coordinates the entire ML pipeline. Prepares data for training.

**Key Functions:**

#### `encode_classification_target(y)`

- Converts text labels to numbers (required by models)

- Example: "Yes" → 1, "No" → 0, "Maybe" → 2

- Stores the mapping for later decoding

#### `prepare_features(df, target)`

- Separates features (X) from target (y)

- Removes ID columns (id, identifier, employee_id) - they don't help

- Removes near-unique columns (99% unique values = essentially IDs)

- Converts text to numbers

**Example:**

```

Input DataFrame:

┌────┬──────────┬──────┬──────────┐

│ ID │ Age   │ City │ Purchased │

```
├────┼──────────┼──────┼───────────┤
│ 1 │ 25     │ NYC │ Yes     │
│ 2 │ 30     │ LA │ No      │
└────┴──────────┴──────┴───────────┘

Output:
X (Features):        y (Target):
┌──────────┬──────┐  ┌───────────┐
│ Age    │ City │  │ Purchased │
├──────────┼──────┤  ├───────────┤
│ 25    │ 0  │  │ 1     │
│ 30    │ 1  │  │ 0     │
└──────────┴──────┘  └───────────┘

(ID dropped, City converted to numbers)
```

7️⃣ **evaluation.py** - Evaluation & Leaderboard Layer

What it does: Tests models and creates a ranking (leaderboard) of their performance.

Key Metrics:

#### Classification Metrics (for Yes/No type predictions):

- **Accuracy**: % of correct predictions
  - Formula: Correct Predictions / Total Predictions
  - Example: 95% accuracy = 95 out of 100 correct

- **Precision**: Of positive predictions, how many are actually positive
  - Formula: True Positives / (True Positives + False Positives)
  - Use when: False positives are expensive
  - Example: Email spam filter - precision = "of emails marked spam, how many really are spam"

- **Recall**: Of actual positives, how many did we find
  - Formula: True Positives / (True Positives + False Negatives)
  - Use when: False negatives are expensive
  - Example: Cancer detection - recall = "of actual cancer cases, how many did we catch"

- **F1-Score**: Harmonic mean of precision & recall
  - Good for imbalanced datasets
  - Balances both concerns

- **ROC-AUC**: Area under ROC curve
  - Measures discrimination ability across all thresholds
  - Range: 0.5 (random) to 1.0 (perfect)

#### Regression Metrics (for number predictions):

- **RMSE** (Root Mean Square Error): Average prediction error
  - Example: RMSE $5000 means predictions off by ~$5000 on average

- **MAE** (Mean Absolute Error): Average absolute difference

  - More interpretable than RMSE

- **R^2**: How much variance is explained (0 to 1)

  - R^2 = 0.8 means model explains 80% of variation

- **MAPE**: Mean Absolute Percentage Error

  - Good for comparing datasets with different scales

  - Example: 15% MAPE means 15% off on average

#### `cross_validate_model(model, X, y, cv_folds=5)`

- Tests model 5 times on different data subsets

- **K-Fold Cross-Validation:**

  1. Split data into 5 equal parts

  2. Use 4 parts to train, 1 to test

  3. Repeat 5 times, each part as test set once

  4. Average the scores

- **Why?** Gives realistic estimate of performance on new data

```

Fold 1: Train on [2,3,4,5]  Test on [1]

Fold 2: Train on [1,3,4,5]  Test on [2]

Fold 3: Train on [1,2,4,5]  Test on [3]

Fold 4: Train on [1,2,3,5]  Test on [4]

Fold 5: Train on [1,2,3,4]  Test on [5]

Final Score = Average of all 5 test scores
```

```

#### `create_leaderboard(models, X, y, task)`

- Trains all models and ranks them

- Shows which model performed best

- Creates bar chart visualization

---

### 8️⃣ **app.py** - User Interface Layer (Streamlit)

**What it does:** Web interface where users interact with everything above.

**Architecture:**

```

User Uploads CSV

↓

[Tab 1: Data Understanding]

↓ Displays schema, target, task type

↓ Shows imbalance, skewness, leakage

↓

[Tab 2: Diagnosis]

↓ Missing values heatmap

↓ Outliers, duplicates

↓ Correlation analysis

↓

[Tab 3: Cleaning]

↓ Apply cleaning transformations

```
  ↓ Choose strategies
  ↓ Preview cleaned data
  ↓
[Tab 4: Features]
  ↓ Feature importance
  ↓ Select important features
  ↓
[Tab 5: Models]
  ↓ Available models for this task
  ↓
[Tab 6: Leaderboard]
  ↓ Cross-validate all models
  ↓ Show rankings
  ↓
[Tab 7: Train & Test]
  ↓ Train selected model
  ↓ Show detailed metrics
  ↓ Performance summary
```

****Key Functions in app.py:****

#### `prepare_features(df, target_col)`

- Converts categorical to numbers using one-hot encoding

- Sanitizes column names for LightGBM compatibility

- Returns X (features) and y (target)

#### `get_available_models(task, n_rows, n_features, n_categorical)`

- Selects models based on:
  - Dataset size
  - Number of features
  - Number of categorical columns
- Returns appropriate model candidates

#### `generate_prediction_summary(model_name, task, metrics)`

- Creates human-readable interpretation of model results
- Example: "Model achieved 95% accuracy meaning..."

## 🔄 Complete Data Pipeline Flow

### Step 1: User Uploads Data

```
CSV File → Streamlit App → Loaded as DataFrame
```

### Step 2: Data Understanding

```
DataFrame
 ↓
infer_schema() → Detect types
 ↓
infer_target() → Find target column
 ↓
infer_task() → Classification or Regression?
```

↓

detect_imbalance() → Check class balance

↓

detect_skewness() → Check numeric distribution

↓

detect_leakage() → Flag suspicious correlations

↓

Display: Schema, Task, Issues

` ` `

### Step 3: Data Diagnosis

` ` `

DataFrame

↓

missing_values_report() → Show NaN patterns

↓

duplicate_report() → Count duplicates

↓

outlier_report() → Find unusual values

↓

correlation_analysis() → Numeric relationships

↓

categorical_correlation() → Categorical relationships

↓

Display: Heatmaps, Statistics, Visualizations

` ` `

### Step 4: Data Cleaning

```
DataFrame
 ↓
Remove Duplicates
 ↓
Drop Sparse Columns (>50% missing)
 ↓
Impute Missing Values
 │   ├─ Numeric: Median strategy
 │  └─ Categorical: "Unknown" strategy
 ↓
Cap Outliers (limit extreme values)
 ↓
Create Missing Indicators (track what was missing)
 ↓
Clean DataFrame
```

### Step 5: Feature Engineering

```
Clean DataFrame
 ↓
Create Interaction Features (optional)
 ↓
Feature Selection:
  ├─ Variance Threshold
  ├─ Mutual Information
  ├─ Chi-Square Test
```

  ├─ Random Forest Importance

 └─ L1 Regularization

 ↓

Selected Features List

```


### Step 6: Data Preparation for Models

```

Clean DataFrame + Selected Features

 ↓

Separate X (features) and y (target)

 ↓

Remove ID columns

 ↓

Encode Categorical Variables

 │ ├─ One-hot encoding (in app)

 │ └─ Factorization (in runner)

 ↓

Encode Target (for classification)

 ↓

Ready for Model Training

```


### Step 7: Model Training & Cross-Validation

```

X, y (prepared data)

 ↓

For Each Model in Registry:

```
 ├– Split into K-Folds (5 or 10)
 ├– Train on fold 1,2,3,4
 ├– Test on fold 5
 ├– Calculate metrics
 ├– Repeat for all folds
 └─ Average metrics
 ↓
Model Performance Scores
```

### Step 8: Leaderboard Generation

```
All Model Scores
 ↓
Sort by Primary Metric
 ├– Classification: F1-Score
 └─ Regression: $R^2$
 ↓
Create Ranking
 ↓
Visualize Bar Charts
 ↓
Display Leaderboard
```

### Step 9: Final Training & Testing

```
Selected Model + Prepared Data
```

↓

Split: 80% train, 20% test

↓

Train on training set

↓

Predict on test set

↓

Calculate detailed metrics

↓

Generate performance summary

↓

Display:

├─ Metrics table

├─ Confusion matrix (classification)

├─ Feature importance

└─ Human-readable summary

` ` `

## 🎯 Example Workflow: Predicting Customer Churn

### User's Goal

Build a model to predict which customers will leave (churn).

### Step-by-Step Execution

1. Upload Data

```
Dataset: 1000 customers, 15 columns

Columns: age, tenure, monthly_bill, customer_service_calls, churn (Yes/No)
```

2. Understanding

```
Schema:

  - Numeric: age, tenure, monthly_bill, customer_service_calls

  - Categorical: gender, contract_type, internet_service

  - Target: churn (Classification task)


Findings:

  ✓ 70% No, 30% Yes (imbalanced but manageable)

  ✓ Some skewness in tenure (many new customers)

  ✗ monthly_bill highly correlated with tenure (expected)
```

3. Diagnosis

```
Issues Found:

  - 2% missing values in customer_service_calls

  - 15 duplicate customer records

  - 3 outliers in monthly_bill (customers paying $5000+)

  - High correlation between contract_type and monthly_bill
```

4. Cleaning

```
Actions Taken:
  ✓ Removed 15 duplicate records
  ✓ Filled 2% missing values with median
  ✓ Capped extreme bills at 95th percentile
  ✓ Created "has_missing_calls" indicator

Result: 985 clean records
```

5. Feature Selection

```
All Features Score:
  - tenure: 0.95 ★★★★★ (most important)
  - monthly_bill: 0.87 ★★★★
  - contract_type: 0.82 ★★★★
  - customer_service_calls: 0.78 ★★★
  - internet_service: 0.65 ★★★
  - age: 0.45 ★★ (less important)

Selected: Top 5 features
```

6. Model Leaderboard

```
Rank | Model         | F1-Score | Accuracy
─────┼───────────────┼──────────┼──────────
```

1 | XGBoost | 0.85 | 0.89 ✓

2 | Random Forest | 0.82 | 0.87

3 | LightGBM | 0.81 | 0.86

4 | Logistic Reg | 0.76 | 0.82

5 | Decision Tree | 0.72 | 0.79

` ` `

7. Final Training

` ` `

Selected: XGBoost (best F1-score)

Train/Test Split: 80% train (788), 20% test (197)

Test Results:

- Accuracy: 89% (correctly identified 175/197 cases)
- Precision: 87% (when we predict churn, 87% actually churn)
- Recall: 83% (we catch 83% of churners)
- F1-Score: 0.85 (excellent balance)

Top Predictors:

1. tenure (most important)
2. monthly_bill
3. contract_type

` ` `

8. Insights Generated

` ` `

📊 Business Insights:

✓ Tenure is strongest churn indicator

→ Focus retention on first 6 months

✓ High bills predict churn

→ Consider price-lock programs

✓ Contract type matters

→ Month-to-month customers churn 5x more

✓ Service calls correlate with churn

→ Better first-contact resolution needed

Model ready for:

- Scoring new customers

- Identifying at-risk segments

- Measuring retention program impact

## 🚀 **How to Run the Application**

### 1. Install Dependencies

```bash

pip install -r requirements.txt

```

### 2. Run the App

```bash
# Windows
.\run.bat

# macOS/Linux
./run.sh

# Or directly
streamlit run ui/app.py
```

### 3. In Browser

```
Open: http://localhost:8501
```

### 4. Upload CSV

- Click "Browse files"
- Select your CSV
- Wait for processing
- Explore tabs

## 🔑 Key Technologies

| Technology | Purpose |

Streamlit	Web interface (no HTML/CSS needed)
Pandas	Data manipulation & analysis
Scikit-learn	ML algorithms & metrics
XGBoost	Powerful gradient boosting
LightGBM	Fast gradient boosting
Plotly	Interactive visualizations
Matplotlib	Static visualizations

## 💡 Summary Table of Each Module

Module	Layer	Input	Output	Key Algorithm
understanding.py	1	Raw DataFrame	Schema, Target, Task	Type inference
diagnostics.py	2	DataFrame	Missing, Outliers, Corr	Statistical analysis
cleaning.py	3	DataFrame	Clean DataFrame	Imputation, capping
selection.py	4	X, y	Feature list	MI, Random Forest
models.py	5	-	Model registry	Algorithm selection
runner.py	6	DataFrame	X, y prepared	Encoding, preprocessing
evaluation.py	7	Model, X, y	Scores, Leaderboard	K-Fold CV
app.py	8	User input	Web interface	Streamlit framework

## 🎓 Learning Path

To understand the codebase:

1. **Start with app.py** - See what users do

2. **Read understanding.py** - How data is analyzed

3. **Read diagnostics.py** - What problems are found

4. **Read cleaning.py** - How problems are fixed

5. **Read selection.py** - How features are selected

6. **Read models.py** - What models are available

7. **Read evaluation.py** - How performance is measured

8. **Read runner.py** - How it all connects

## 🏆 Why This Architecture?

✅ **Modular**: Each layer is independent, easy to modify

✅ **Reusable**: Layers can be used in other projects

✅ **Testable**: Each module has unit tests

✅ **Transparent**: Easy to understand what's happening

✅ **Extensible**: Easy to add new models, metrics, cleaning strategies

✅ **Production-Ready**: Can be deployed as API

This comprehensive explanation should help you understand every aspect of the Automated Insight Engine!