AgentPOC - Python script for model work

I am working on loan approval calssififacation model, earlier i was using jupyter notebook, I was pulling scores, fearures, model accuracy score , hyperparameter from cell output.

Now I have to use .py file to be used for model recommendation system.

help me out how i can get the required details.

one option is to store data in text file, is there any other way to handle this problem

Great question. When you move from a notebook to a `.py` pipeline, the key is to produce **structured, machine-readable outputs** that your recommendation system can consume—without you scraping cell text.

Here are four solid patterns (pick one and we'll tailor it):

1. **Return a structured object + (optionally) dump JSON**
   Fast, dependency-free. Your training function returns a Python dict/dataclass with metrics, features, and best params. If run as a script, it also writes `artifacts/metrics.json`, `params.json`, `features.json`, and the serialized model.

```python
# train_loan_model.py
from __future__ import annotations
from dataclasses import dataclass, asdict
from typing import Any, Dict, List
import argparse, json, os, joblib, numpy as np, pandas as pd
from sklearn.compose import ColumnTransformer
from sklearn.preprocessing import OneHotEncoder, StandardScaler
from sklearn.linear_model import LogisticRegression
from sklearn.model_selection import train_test_split, GridSearchCV
from sklearn.pipeline import Pipeline
```

```python
from sklearn.metrics import accuracy_score, roc_auc_score, precision_recall_fscore_support,
confusion_matrix

@dataclass
class RunSummary:
    model_name: str
    best_params: Dict[str, Any]
    metrics: Dict[str, float]
    prfs: Dict[str, Any]
    confusion_matrix: List[List[int]]
    feature_names: List[str]
    n_train: int
    n_test: int

def _json_default(o):
    if isinstance(o, (np.integer,)): return int(o)
    if isinstance(o, (np.floating,)): return float(o)
    if isinstance(o, (np.ndarray,)): return o.tolist()
    return str(o)

def train_and_evaluate(df: pd.DataFrame, target: str) -> RunSummary:
    y = df[target]
    X = df.drop(columns=[target])

    num_cols = X.select_dtypes(include=["int64","float64","int32","float32"]).columns.tolist()
    cat_cols = X.select_dtypes(include=["object","category","bool"]).columns.tolist()

    pre = ColumnTransformer([
        ("num", StandardScaler(), num_cols),
        ("cat", OneHotEncoder(handle_unknown="ignore", sparse_output=False), cat_cols),
    ])

    clf = LogisticRegression(max_iter=500, n_jobs=None)
    pipe = Pipeline([("pre", pre), ("clf", clf)])

    param_grid = {
        "clf__C": [0.1, 1.0, 3.0],
        "clf__penalty": ["l2"],
        "clf__solver": ["lbfgs"]
    }

    X_tr, X_te, y_tr, y_te = train_test_split(X, y, test_size=0.25, stratify=y, random_state=42)
```

```
    gs = GridSearchCV(pipe, param_grid=param_grid, cv=5, scoring="roc_auc", n_jobs=-1,
refit=True)
    gs.fit(X_tr, y_tr)

    y_pred = gs.predict(X_te)
    y_proba = gs.predict_proba(X_te)[:, 1] if hasattr(gs, "predict_proba") else None

    acc = accuracy_score(y_te, y_pred)
    auc = roc_auc_score(y_te, y_proba) if y_proba is not None else float("nan")
    p, r, f1, _ = precision_recall_fscore_support(y_te, y_pred, average="binary", zero_division=0)
    cm = confusion_matrix(y_te, y_pred).tolist()

    # feature names after fit
    pre_fitted = gs.best_estimator_.named_steps["pre"]
    feat_names = []
    if num_cols: feat_names += num_cols
    if cat_cols:
        ohe = pre_fitted.named_transformers_["cat"]
        feat_names += ohe.get_feature_names_out(cat_cols).tolist()

    return RunSummary(
        model_name=gs.best_estimator_.named_steps["clf"].__class__.__name__,
        best_params=gs.best_params_,
        metrics={"accuracy": acc, "roc_auc": auc},
        prfs={"precision": p, "recall": r, "f1": f1},
        confusion_matrix=cm,
        feature_names=feat_names,
        n_train=len(X_tr), n_test=len(X_te)
    )

def main():
    ap = argparse.ArgumentParser()
    ap.add_argument("--csv", required=True)
    ap.add_argument("--target", required=True)
    ap.add_argument("--outdir", default="artifacts")
    args = ap.parse_args()

    os.makedirs(args.outdir, exist_ok=True)
    df = pd.read_csv(args.csv)
    summary = train_and_evaluate(df, target=args.target)

    # dump structured artifacts
```

```
    with open(os.path.join(args.outdir, "metrics.json"), "w") as f:
        json.dump({"metrics": summary.metrics, "prfs": summary.prfs}, f, default=_json_default,
indent=2)
    with open(os.path.join(args.outdir, "params.json"), "w") as f:
        json.dump(summary.best_params, f, default=_json_default, indent=2)
    with open(os.path.join(args.outdir, "features.json"), "w") as f:
        json.dump(summary.feature_names, f, default=_json_default, indent=2)

    joblib.dump(summary, os.path.join(args.outdir, "run_summary.pkl"))
    joblib.dump(df.columns.tolist(), os.path.join(args.outdir, "input_columns.pkl"))
    # also print once to STDOUT so other tools can capture without files
    print(json.dumps(asdict(summary), default=_json_default))

if __name__ == "__main__":
    main()
```

How your recommender consumes it:

- **Programmatically** (preferred): `from train_loan_model import train_and_evaluate` → get a `RunSummary` object directly.
- **From artifacts**: read `metrics.json`, `params.json`, `features.json`.
- **From STDOUT**: call the script and parse the printed JSON.

2. **Use an experiment tracker (MLflow)**

- Pros: automatic time series of runs, comparisons, artifacts, model registry.
- Works offline to a local folder (`mlruns`)—no server required.

```python
import mlflow
mlflow.set_tracking_uri("file:./mlruns")
mlflow.set_experiment("loan_approval")

with mlflow.start_run(run_name="logreg_baseline"):
    # after fitting:
    mlflow.log_params(gs.best_params_)
    mlflow.log_metrics({"accuracy": acc, "roc_auc": auc, "precision": p, "recall": r, "f1": f1})
    mlflow.log_text("\n".join(summary.feature_names), "features.txt")
    mlflow.sklearn.log_model(gs.best_estimator_, artifact_path="model")
```

Your recommender can read the **latest run** and pull params/metrics programmatically via MLflow's Python API or by loading the logged model.

3. **JSON Lines logging (streaming)**

- If you want to tail logs or feed them into another process:

```python
import json, sys
def log_event(kind, payload):
    sys.stdout.write(json.dumps({"kind": kind, **payload}) + "\n"); sys.stdout.flush()
# usage:
log_event("metrics", {"accuracy": acc, "roc_auc": auc})
log_event("params", gs.best_params_)
log_event("features", {"names": summary.feature_names})
```

Downstream, read line-by-line and route by `kind`.

4. **"Sink" interface (clean separation)**

- Define a tiny interface and plug different sinks (JSON file, MLflow, in-memory) without changing your training code.

```python
class Sink:
    def log_params(self, d): ...
    def log_metrics(self, d): ...
    def log_features(self, names): ...

class JsonSink(Sink):
    def __init__(self, outdir): os.makedirs(outdir, exist_ok=True); self.outdir = outdir
    def log_params(self, d): json.dump(d, open(f"{self.outdir}/params.json","w"), indent=2)
    def log_metrics(self, d): json.dump(d, open(f"{self.outdir}/metrics.json","w"), indent=2)
    def log_features(self, names): json.dump(names, open(f"{self.outdir}/features.json","w"),
indent=2)
# later: Sink = MlflowSink() without touching your trainer
```

Why not a plain text file?

Plain text is brittle to parse. **JSON/YAML or a tracker** gives you:

- Type safety (numbers stay numbers)
- Easy parsing & validation
- Stable contracts between training and the recommender

Minimal artifact checklist (good defaults)

- `metrics.json` → accuracy, ROC-AUC, precision, recall, F1

- `params.json` → best hyperparameters
- `features.json` → final feature names (post-OHE)
- `model.pkl` or MLflow model → for reuse
- Optionally: `confusion_matrix.csv` or `report.json`

Your turn (one quick choice):
Will your recommendation system call the trainer **as a Python function** (import and use the returned object) or **as a CLI script** (read JSON or use MLflow)? Pick one and I'll tailor the skeleton to your exact setup.

here, suggest me to tack loan approval model, lwhat kind of information is needs to be tracked.

make sure model should able to explain preporcessing steps fairness, governance, explanibility , able to manage drift if any metrics or detail missing, you can add it

Got it—let's design a **tracking spec** so your loan-approval model can be audited, explained, and monitored end-to-end (preprocessing → fairness → governance → explainability → drift). Use this as your "contract" between the training script and your recommender/monitoring system.

What to track (by section)

1) Data & lineage

- Dataset ID, storage URI, commit/sha or snapshot hash
- Date range, geography/segment filters, sampling method
- Target definition (positive class, label window), leakage checks performed
- Schema (name, type, allowed nulls) and **protected attributes** available (e.g., gender, marital_status, education, region)

2) Preprocessing & feature engineering

- Per-column: imputation (strategy, fill value), scaling (Standard/MinMax/None), encoding (OHE/WOE/target encoding + CV leakage guard), winsorization/outlier rules, bucketing

- Generated features (name, formula/deps), dropped features & reasons
- Train/val/test split details (stratification key, random_seed)
- Class imbalance handling (class_weight/SMOTE params)

3) Training config

- Algorithm, library versions, random_seed
- Hyperparam **search space**, CV scheme, scoring metric(s)
- Best hyperparams + validation fold scores distribution
- Decision threshold selection method (Youden's J, cost curve, business constraint)

4) Core performance & calibration

- ROC-AUC, PR-AUC, Accuracy, Precision/Recall/F1 (at chosen threshold)
- Confusion matrix, KS, Gini, lift @ top k%
- **Calibration**: Brier score, ECE, calibration method (Platt/Isotonic), reliability curve points

5) Fairness (group & intersectional)

For each protected attribute and key intersections (e.g., gender×region):

- Selection/approval rate by group
- **Statistical Parity Difference (SPD)**
- **Disparate Impact (Impact Ratio)** (watch 0.8 "rule of thumb")
- **Equal Opportunity gap** (TPR gap)
- **Average Odds Difference** ((TPR gap + FPR gap)/2)
- **Predictive Parity gap** (PPV gap)
- Mitigation used (reweighing, constraints, post-processing), and effect on utility metrics

6) Explainability

- Global: permutation or model-native importances, SHAP summary stats, top-k features with directionality
- Local: per-decision SHAP top-3 "reason codes" (feature, contribution, sign)
- PDP/ICE for 3–5 business-critical variables
- Business dictionary mapping: model_feature → human label + description

7) Governance & compliance

- **Model Card**: purpose, scope, assumptions, limitations, known risks

- Approvals: owner, reviewer(s), dates, change request IDs
- Regulatory mapping (e.g., ECOA/FCRA/DPDP) and privacy controls (PII handling, anonymization)
- Audit log pointers (who trained, data version, git SHA, environment hash)

8) Monitoring & drift management

- **Data drift** per feature: PSI or JS distance; thresholds & alert rules
- **Concept drift**: rolling AUC/F1, calibration drift, fairness drift
- Volume/mix drift: segment shares vs. baseline
- SLA/ops: latency p95, error rate, fallback path (e.g., manual review)
- **Retrain triggers** (e.g., PSI>0.2 for 2 weeks OR AUC drop >0.05)
- Champion–Challenger/shadow results and cutover criteria

9) Operations & decision logging

- Immutable decision log: request_id, timestamp, model_version, features hash (or redacted), score, decision, threshold, reason_codes, reviewer_override (if any)

Suggested artifact layout (simple & robust)

```pgsql
artifacts/
  run.json              # single summary (see schema below)
  metrics.json          # core metrics at chosen threshold
  params.json           # best hyperparameters
  features.json         # final feature names (after OHE)
  preprocessing.json    # per-column preprocessing plan actually fitted
  fairness.json         # group & intersectional metrics
  calibration.json      # ECE/Brier + curve
  explanations/
    shap_summary.png
    pdp_LoanAmount.png
    reason_codes_sample.csv
  monitoring_baseline.json  # drift baselines & alert thresholds
  model/
    model.pkl           # or MLflow model directory
```

Minimal JSON schema for a single run (copy/paste starter)

json


```json
{
  "run_id": "2025-08-21T12:00:00Z_abc123",
  "context": {
    "git_sha": "beefcafe",
    "env": {"python": "3.11.9", "sklearn": "1.4.2"},
    "random_seed": 42
  },
  "data": {
    "dataset_uri": "s3://bucket/loans/v34.parquet",
    "date_range": "2023-01-01..2024-12-31",
    "schema_hash": "sha256:...",
    "protected_attributes": ["gender", "married", "education"]
  },
  "preprocessing": {
    "split": {"test_size": 0.25, "stratify": "Loan_Status"},
    "imbalance": {"strategy": "SMOTE", "k_neighbors": 5},
    "columns": [
      {"name": "LoanAmount", "impute": "median", "scale": "standard", "winsorize": [0.01, 0.99]},
      {"name": "Property_Area", "encode": "onehot", "handle_unknown": "ignore"}
    ],
    "generated_features": ["DebtToIncome", "LoanToIncome"]
  },
  "training": {
    "algo": "LogisticRegression",
    "search_space": {"C": [0.1, 1.0, 3.0], "solver": ["lbfgs"]},
    "cv": "StratifiedKFold(n_splits=5)",
    "best_params": {"C": 1.0, "solver": "lbfgs", "penalty": "l2"},
    "threshold": 0.42
  },
  "metrics": {
    "roc_auc": 0.812,
    "pr_auc": 0.765,
    "accuracy": 0.781,
    "precision": 0.734,
    "recall": 0.691,
    "f1": 0.712,
```

```
    "ks": 0.36,
    "lift_top_decile": 2.4,
    "brier": 0.183,
    "confusion": {"tn": 820, "fp": 180, "fn": 210, "tp": 790}
  },
  "calibration": {"method": "isotonic", "ece": 0.031},
  "fairness": {
    "gender": {
      "groups": {
        "Male":   {"n": 1200, "approve_rate": 0.62, "tpr": 0.71, "fpr": 0.18, "ppv": 0.75},
        "Female": {"n": 800,  "approve_rate": 0.57, "tpr": 0.67, "fpr": 0.17, "ppv": 0.73}
      },
      "spd": -0.05,
      "impact_ratio": 0.92,
      "tpr_gap": -0.04,
      "avg_odds_diff": 0.02,
      "ppv_gap": -0.02
    }
  },
  "explainability": {
    "global_importance": [{"feature": "Credit_History", "importance": 0.31}, {"feature":
"DebtToIncome", "importance": 0.18}],
    "reason_code_template": "Top 3 drivers: {f1} {s1}, {f2} {s2}, {f3} {s3}"
  },
  "governance": {
    "model_card_path": "artifacts/model_card.md",
    "limitations": ["May underperform on self-employed with sparse history"],
    "approvals": [{"role": "Risk", "name": "A. Sharma", "date": "2025-08-21"}]
  },
  "monitoring": {
    "baselines": {"psi_threshold": 0.2, "auc_drop": 0.05},
    "retrain_rules": [{"if": "psi>0.2 for 2 consecutive weeks OR auc_drop>0.05", "then":
"trigger_retrain"}]
  }
}
```

Tiny helper formulas (so you can compute & log them)

- **SPD** = P(approve | group=A) − P(approve | group=B)
- **Impact Ratio** = P(approve | A) / P(approve | B)

- **Equal Opportunity gap** = TPR(A) − TPR(B)
- **Average Odds Diff** = ½[(TPR(A)−TPR(B)) + (FPR(A)−FPR(B))]
- **PSI** (for drift): sum over bins of (p−q)·ln(p/q) comparing train vs. production feature distributions.

Implementation tip

Wire this spec to **either**:

- **MLflow** (nice UI + registry) and log JSONs as artifacts; or
- Plain **JSON files** (fast, dependency-light) using the "Sink" pattern so you can swap storage later without touching training code.

Want me to turn this into a ready-to-use `Sink` class + MLflow/JSON implementations with method stubs for each section above?

Quick check (one thing to confirm): **Which protected attributes do you want to monitor** (e.g., gender, marital_status, education, region)?

you tell me, in real case scnarios do we track protected variables?

Short answer: **yes—most real deployments do track protected attributes for testing/monitoring fairness**, but you must do it under tight legal and governance controls and never use them to *decide* credit outcomes.

Here's the practical way to handle it:

When and why to track

- **Fairness audits & reporting.** Regulators expect you to demonstrate non-discrimination; that requires group-wise metrics (approval rate, TPR/FPR gaps, impact ratio, etc.). In the US this sits under ECOA/Reg B and adverse-action duties. Consumer Financial Protection Bureau +1
- **EU/UK:** High-risk AI (like credit scoring) must manage dataset bias and document data governance; processing "special category" data may be allowed

for equality monitoring under specific legal bases/conditions. EUR-Lex Artificial Intelligence Act ICO +1

- **India:** The DPDP Act applies to personal data; if you collect sensitive traits, do so with purpose limitation, consent/notice, and strong safeguards. MeitY DLA Piper Data Protection

Two common patterns

1. **Direct collection (preferred if lawful)**
   - Collect protected traits (e.g., sex, age, caste/tribe where legally permitted, etc.) **only for fairness monitoring/self-testing**, not as model inputs.
   - Store in a **separate, access-controlled "fairness table."** Join to decisions via a pseudonymous key to compute **aggregate** metrics; restrict raw attribute access.
   - Record the **legal basis** (consent or other) and retention period in your model card/governance log.
2. **Proxies (when you can't collect)**
   - Use statistical proxies (e.g., **BISG** in the US for race/ethnicity) for *population-level* disparity checks—**never** for individual decisioning or notices. Consumer Finance Files GitHub

Guardrails (do this if you track)

- **Scope control:** Mark attributes as "audit-only"; block them from the training/inference feature store by policy.
- **Role-based access + encryption:** Separate data ownership (risk/compliance) from model developers.
- **Aggregate outputs only:** Dashboards expose gaps (SPD, impact ratio, TPR/FPR gaps); raw rows are locked down.
- **Adverse-action explainability:** Keep reason codes from model features; do **not** cite protected traits. (Reg B requires specific, accurate reasons.) Consumer Financial Protection Bureau
- **Documentation:** Model card: purpose/scope, data sources, fairness tests performed, limitations, approvals, and change history (git SHA, dataset version). (EU AI Act stresses data governance for high-risk systems.) EUR-Lex Artificial Intelligence Act

What to compute (minimal fairness set)

For each protected attribute (and key intersections):

- Approval rate, **Statistical Parity Difference**; **Impact Ratio** (watch the ~0.8 rule of thumb)
- **Equal Opportunity** gap (TPR gap), **Average Odds** difference (½[(TPR gap)+(FPR gap)])
- **Predictive Parity** gap (PPV gap)
  Track trends over time and set alert thresholds.

If you cannot legally/operationally collect

- Run **proxy-based** fairness checks (e.g., BISG) and clearly label them as estimates with confidence limits; many regulators recognize this for portfolio-level analysis, with caveats. Consumer Finance Files
- Prefer **external/third-party audits** or **privacy-preserving enclaves** for sensitive joins.

If you want, I can give you a tiny "fairness data contract" (JSON schema + code hooks) that:

- logs which attributes are tracked (and legal basis),
- keeps them out of the model pipeline, and
- produces the aggregate fairness report automatically.

Quick check: **Which regions will you operate in (India only, or also US/EU/UK)?** That determines exactly *how* we collect and store these fields.

Sources

Its in india:

Great—India context helps. Here's a crisp, **India-ready tracking blueprint** that lets you **audit fairness** without ever using protected traits to decide credit.

Legal framing (India)

- **DPDP Act, 2023**: process only for a **specific purpose with notice/consent**, honor user rights, and add tighter duties if you're designated a **Significant Data Fiduciary**. Keep audit logs and breach-notice readiness. MeitY PRS Legislative Research Latham & Watkins
- **RBI Digital Lending (2025) + earlier guidelines**: transparency (e.g., **Key Fact Statement/APR**), strong governance of digital lending workflows, clear RE–LSP

roles. PDICAI azb FIDC

- **Fair practices / non-discrimination** is expected of lenders; underwriting should be **auditable** and free of prejudice. Bankkeeping.com MEDIANAMA
- **Model-risk governance**: board-approved lifecycle, independent validation, periodic review—applies to credit models (incl. ML). FIDC Reuters

---

What to track (and how)

A) "Audit-only" attributes (kept out of modeling)

Collect (or derive) these **only for fairness testing/monitoring**, store in a **separate, access-controlled table**, and never feed them to the model:

- **Sex/Gender**
- **Age band** (e.g., 18–24, 25–34, …)
- **Marital status**
- **Education level**
- **Occupation type / employment class**
- **Region** (state + urban/rural)
- (Optional) **Disability** *only if you have clear lawful basis and strong governance*

If you can't collect some fields, use **coarse proxies** (e.g., urban/rural, region) for portfolio-level checks—label them as estimates. (Avoid invasive/probabilistic inferences unless cleared by compliance.)

Controls: mark columns as *audit_only*; enforce **RBAC**, encryption at rest, retention limits, and a policy rule that blocks these fields from the feature store used in training/inference. MeitY

B) Preprocessing trace

Log exactly what was *fitted and applied*:

- Per-column: imputation strategy/value, scaler, encoder (with `handle_unknown`), outlier rules, winsorization, bucketing.
- Class imbalance handling: weights/SMOTE params.
- Final **post-OHE feature names** and column lineage.

C) Model & search

- Algo + library versions, random seeds.

- CV scheme, **search space**, chosen **threshold** selection rule.
- Fold-level metrics distribution for the best config.

D) Core performance & calibration

- ROC-AUC, PR-AUC, Accuracy, Precision/Recall/F1, KS, Lift@k%.
- Calibration: **Brier**, **ECE**, and method (Isotonic/Platt) + reliability points.

E) Fairness metrics (per group and key intersections)

For each audit attribute:

- **Approval rate**, **Statistical Parity Difference**, **Impact Ratio** (watch ~0.8 rule of thumb),
- **Equal Opportunity gap** (TPR gap), **Average Odds diff** (½[(TPR gap)+(FPR gap)]),
- **Predictive Parity gap** (PPV gap).
  Record any **mitigation** (reweighing, threshold per segment, post-processing) and its utility trade-offs. (Underwriting must be **auditable** and non-discriminatory.) MEDIANAMA

F) Explainability (for regulators & customers)

- **Global**: permutation or SHAP importances (top-k with sign).
- **Local**: top-3 **reason codes** per decision (feature, contribution, ↑/↓).
- PDP/ICE for 3–5 business-critical variables.
- A **business dictionary**: model_feature → human label + description.
  (Keep adverse-action reasons to **model features**, never protected traits.)
  Reserve Bank of India

G) Governance artifacts

- **Model Card**: purpose/scope, assumptions, known limitations, fairness tests done, approvals (owner, reviewer, dates), **git SHA**, data snapshot.
- **KFS alignment**: ensure APR and key terms logged for disclosure. FIDC
  Vinod Kothari Consultants
- **Validation pack**: challenger/benchmark results, stability analysis.
- **Change log**: every retrain with justification.

H) Monitoring & drift (production)

- **Data drift per feature**: PSI/JS distance vs. training baseline (thresholds & alerting).
- **Concept drift**: rolling AUC/F1, calibration drift; **fairness drift** on key groups.

- Mix drift: segment shares vs. baseline (e.g., age-band, region).
- Ops: latency p95, error rate, fallback (manual review).
- **Retrain rules** (e.g., PSI>0.2 for 2 weeks **or** AUC drop >0.05) + approval workflow.

I) Decision log (immutable)

- request_id, timestamp, **model_version**, feature-hash, score, threshold, decision, **reason codes**, and any reviewer override (who/when/why).

Storage format (clean & auditable)

- Log to **JSON artifacts** or **MLflow** (with JSON blobs as artifacts).
- Keep a single `run.json` "table of contents" pointing to: `preprocessing.json`, `metrics.json`, `fairness.json`, `calibration.json`, `monitoring_baseline.json`, `model/…`. (KFS/loan-terms link stored for transparency.) PDICAI FIDC

Would you like me to generate a **ready-to-drop JSON schema + Python "Sink" stubs** (MLflow + filesystem) for exactly these fields?

Sources

Feature Metadata Unable to display visualization

okay, uploading file before start coading, check the features required for loan approval system are sufficient or not, if not list additional features

Feature Coverage Vs. Recommended Taxonomy

Suggested Engineered Features (Add If Feasible)

Nice—thanks for the file. I pulled your `feature_metadata.csv`, profiled it against a loan-model taxonomy, and generated two handy tables:

- **Feature coverage vs. recommended taxonomy** — what you already cover in each area
- **Suggested engineered features** — high-impact additions you can consider

You can open them here:

- Feature coverage summary (CSV)
- Suggested engineered features (CSV)

Quick read of what you already have (from the preview)

Your columns include: `person_age`, `person_education`, `person_income`, `person_emp_exp`, `person_home_ownership`, `loan_amnt`, `loan_intent`, `loan_int_rate`, `loan_percent_income`, `cb_person_cred_hist_length`, … (and likely more in the file). That's a solid **application + basic bureau** core.

High-value additions I recommend (engineered or raw)

1. **Affordability & burden**
   - **DTI:** `(existing EMIs + proposed EMI) / monthly_income`

- **FOIR**: `fixed obligations / monthly_income`
- **PTI**: `proposed EMI / monthly_income`
- **EMI (recomputed)** from principal–rate–tenure for consistency

2. **Collateral & exposure**
   - **LTV**: `loan_amount / collateral_value` (or property value where applicable)
3. **Bureau depth & risk**
   - **Credit utilization** (revolving balance / limit)
   - **Credit age (months)**, **oldest/newest trade age**
   - **Recent inquiries (3m/6m)**
   - **Delinquency recency** and **worst DPD**
   - **# open accounts**, **secured vs. unsecured ratio**
   - **Write-off/settled flags** if available
4. **Cashflow & stability** (if bank statements exist)
   - **Cashflow volatility** (std dev of monthly net inflow)
   - **Salary stability** (on-time salary credit share)
   - **Employer tenure** and **address tenure** (years)
5. **Fraud & application behavior**
   - Prior **fraud/watchlist** flags, **device/IP risk**, **missing docs count**, **application channel**
6. **Audit-only fairness attributes** (kept out of the model, used only for monitoring)
   - **Age band**, **gender/sex**, **marital status**, **education level**, **region (state + urban/rural)**
   - These enable SPD, impact ratio, TPR/FPR gaps, etc., for governance.

I've already listed these in the "Suggested engineered features" file with brief "why/how" notes.

Want me to wire this into your pipeline next? I can generate:

- a **feature registry** stub (`preprocessing.json` spec) with your current columns + the engineered ones, and
- code hooks to compute them safely (with fallbacks when a source column is missing).

One quick check (so I tailor the engineered features correctly): **Do you have bank statements and a bureau pull for each applicant, or just application form fields?**