

## Python Code

```
1 import pandas as pd
2 import numpy as np
3 from collections import defaultdict
4 from fastapi import FastAPI, UploadFile, File, Form, HTTPException
5 import tempfile, os
6 from scipy.stats import skew, kurtosis, entropy, normaltest, jarque_bera
7 from sklearn.feature_selection import mutual_info_classif, mutual_info_regression
8
9 app = FastAPI(
10     title="Scalable Statistical EDA Engine | Phase 1-3",
11     debug=True
12 )
13
14 MAX_SAMPLE_ROWS = 50_000
15 MAX_MI_ROWS = 20_000
16 DISTANCE_WIN_MARGIN = 0.25
17 RF_PRIOR_BONUS = 0.20
18 BOOSTING_PRIOR_BONUS = 0.18
19
20
21 #----- Safety net for JSON Conversion & Policy Helpers -----
22 def distance_model_eligible(signals):
23     return (
24         signals.get("categorical_ratio", 0.0) < 0.3
25         and signals.get("sparsity_ratio", 1.0) < 0.35
26         and signals.get("missing_ratio", 0.0) < 0.05
27         and signals.get("p_over_n", 1.0) < 1.0
28         and signals.get("n_features", 0) <= 25
29         and signals.get("n_samples", 0) <= 50_000
30     )
31
32 def random_forest_preferred(signals):
33     return (
34         signals.get("nonlinear_signal_strength", 0.0) >= 0.25
35         and signals.get("n_samples", 0) >= 300
36         and signals.get("n_features", 0) <= 200
37         and signals.get("sparsity_ratio", 0.0) <= 0.5
38         and signals.get("categorical_ratio", 0.0) <= 0.4
39     )
40
41
42 def _safe_float(val):
43     if val is None:
44         return None
45     if not isinstance(val, (int, float, np.integer, np.floating)):
46         return None
47     if np.isnan(val) or np.isinf(val):
48         return None
49     return float(val)
50
```

```
51
52
53     def to_json_safe(obj):
54         if isinstance(obj, (np.integer,)):
55             return int(obj)
56         if isinstance(obj, (np.floating,)):
57             return _safe_float(float(obj))
58         if isinstance(obj, (np.bool_,)):
59             return bool(obj)
60         if isinstance(obj, float):
61             return _safe_float(obj)
62         if isinstance(obj, dict):
63             return {k: to_json_safe(v) for k, v in obj.items()}
64         if isinstance(obj, list):
65             return [to_json_safe(v) for v in obj]
66         return obj
67
68
69     # ----- Utilities -----
70
71     def save_temp_file(uploaded_file: UploadFile) -> str:
72         suffix = os.path.splitext(uploaded_file.filename)[1]
73         with tempfile.NamedTemporaryFile(delete=False, suffix=suffix) as tmp:
74             tmp.write(uploaded_file.file.read())
75         return tmp.name
76
77
78
79
80     def smart_sample(df, max_rows):
81         sampled = df if len(df) <= max_rows else df.sample(max_rows, random_state=42)
82         return smart_fillna(sampled)
83
84     def smart_fillna(df: pd.DataFrame) -> pd.DataFrame:
85         df = df.copy()
86
87         # Drop columns that are entirely NaN
88         df = df.dropna(axis=1, how="all")
89
90         for col in df.columns:
91             if pd.api.types.is_numeric_dtype(df[col]):
92                 median = df[col].median()
93                 if np.isnan(median):
94                     df[col] = df[col].fillna(0.0)
95                 else:
96                     df[col] = df[col].fillna(median)
97             else:
98                 mode = df[col].mode()
99                 if len(mode) > 0:
100                     df[col] = df[col].fillna(mode.iloc[0])
101                 else:
102                     df[col] = df[col].fillna("UNKNOWN")
103
104     return df
```

```
105
106
107 # ----- Step 1 -----
108
109 def step1_dataset_overview(csv_path):
110     df = pd.read_csv(csv_path, nrows=MAX_SAMPLE_ROWS)
111
112     return {
113         "rows_sampled": len(df),
114         "columns": int(df.shape[1]),
115         "memory_mb_estimate": round(df.memory_usage(deep=True).sum() / (1024 ** 2),
116                                     2),
116         "column_types": {c: str(t) for c, t in df.dtypes.items()},
117         "sample_rows": df.head(5).to_dict(orient="records")
118     }, df
119
120
121 # ----- Step 2 -----
122
123 def step2_target_statistics(series):
124     s = series.dropna()
125
126     if len(s) == 0:
127         raise ValueError("Target column contains only missing values")
128
129     stats = {
130         "unique_values": int(s.nunique()),
131         "unique_ratio": round(s.nunique() / max(len(s), 1), 4),
132         "entropy": _safe_float(float(entropy(s.value_counts(normalize=True))))
133     }
134
135     if pd.api.types.is_numeric_dtype(s):
136         stats.update({
137             "skewness": _safe_float(float(skew(s))),
138             "kurtosis": _safe_float(float(kurtosis(s))),
139             "outlier_ratio_proxy": _safe_float(float(
140                 ((s < s.quantile(0.01)) | (s > s.quantile(0.99))).mean()
141             ))
142         })
143     else:
144         stats["max_class_ratio"] = float(s.value_counts(normalize=True).max())
145
146     return stats
147
148
149 # ----- Step 3 -----
150
151 def step3_problem_inference(target_stats):
152     if target_stats["unique_ratio"] <= 0.05 or target_stats["unique_values"] <= 20:
153         return {"problem_type": "classification", "confidence": 0.95}
154     return {"problem_type": "regression", "confidence": 0.95}
155
156
157 # ----- Step 4 (MOST IMPORTANT FIXES) -----
```

```
158
159     def step4_feature_target_signals(df, target_col, task_type):
160         df = smart_sample(df, MAX_MI_ROWS)
161
162         X = df.drop(columns=[target_col])
163         y = df[target_col].dropna()
164         X = X.loc[y.index]
165         n_samples = int(len(X))
166         n_features = int(X.shape[1])
167
168         numeric_X = X.select_dtypes(include="number")
169         categorical_X = X.select_dtypes(exclude="number")
170         missing_ratio = float(X.isna().mean().mean()) if X.shape[1] > 0 else 0.0
171         p_over_n = float(X.shape[1] / max(len(X), 1))
172
173         # No usable numeric features
174         if numeric_X.shape[1] == 0 and categorical_X.shape[1] == 0:
175             return {
176                 "linear_signal_strength": 0.0,
177                 "nonlinear_signal_strength": 0.0,
178                 "sparsity_ratio": 1.0,
179                 "categorical_ratio": 0.0,
180                 "note": "no_usable_features"
181             }
182
183         y_enc = y if task_type == "regression" else y.astype("category").cat.codes
184
185         # Constant target → MI & corr invalid
186         if y_enc.nunique() <= 1:
187             return {
188                 "linear_signal_strength": 0.0,
189                 "nonlinear_signal_strength": 0.0,
190                 "sparsity_ratio": float((numeric_X == 0).mean().mean()),
191                 "categorical_ratio": float(categorical_X.shape[1] / max(X.shape[1], 1)),
192                 "note": "constant_target"
193             }
194
195         if numeric_X.shape[1] > 0:
196             corr = numeric_X.corrwith(y_enc).abs()
197             linear_signal = _safe_float(float(corr.mean()))
198             if linear_signal is None:
199                 linear_signal = 0.0
200             skew_mean = _safe_float(float(numeric_X.skew(numeric_only=True).abs().mean()))
201             kurt_mean =
202             _safe_float(float(numeric_X.kurtosis(numeric_only=True).abs().mean()))
203             skew_kurtosis_score = float((skew_mean or 0.0) + (kurt_mean or 0.0))
204         else:
205             linear_signal = 0.0
206             skew_kurtosis_score = 0.0
207
208         # Build mixed feature matrix for MI
209         if categorical_X.shape[1] > 0:
210             cat_encoded = categorical_X.apply(lambda col: pd.factorize(col)[0])
211             X_mi = pd.concat([numeric_X, cat_encoded], axis=1)
```

```

211         discrete_mask = [False] * numeric_X.shape[1] + [True] * cat_encoded.shape[1]
212     else:
213         X_mi = numeric_X
214         discrete_mask = [False] * numeric_X.shape[1]
215
216     if X_mi.shape[1] > 0:
217         mi = (
218             mutual_info_regression(X_mi, y_enc, discrete_features=discrete_mask)
219             if task_type == "regression"
220             else mutual_info_classif(X_mi, y_enc, discrete_features=discrete_mask)
221         )
222         mi_mean = _safe_float(float(np.mean(mi)))
223         if mi_mean is None:
224             mi_mean = 0.0
225         else:
226             mi_mean = 0.0
227
228     nonlinearity_index = (
229         float(mi_mean / max(linear_signal, 1e-6))
230         if linear_signal > 0.0
231         else float(mi_mean)
232     )
233
234     return {
235         "linear_signal_strength": linear_signal,
236         "nonlinear_signal_strength": mi_mean,
237         "sparsity_ratio": float((numeric_X == 0).mean().mean()) if numeric_X.shape[1]
238         > 0 else 1.0,
239         "categorical_ratio": float(categorical_X.shape[1] / max(X.shape[1], 1)),
240         "nonlinearity_index": _safe_float(float(nonlinearity_index)),
241         "missing_ratio": missing_ratio,
242         "p_over_n": p_over_n,
243         "skew_kurtosis_score": _safe_float(float(skew_kurtosis_score)),
244         "n_samples": n_samples,
245         "n_features": n_features
246     }
247
248
249 # ----- Step 5 -----
250
251 def step5_model_family_scoring(task_type, signals):
252     cat_ratio = signals.get("categorical_ratio", 0.0)
253     nonlin_idx = signals.get("nonlinearity_index", 0.0) or 0.0
254     sparsity = signals.get("sparsity_ratio", 0.0)
255     linear = signals.get("linear_signal_strength", 0.0)
256     nonlinear = signals.get("nonlinear_signal_strength", 0.0)
257     missing_ratio = signals.get("missing_ratio", 0.0)
258     p_over_n = signals.get("p_over_n", 0.0)
259     n_samples = signals.get("n_samples", 0)
260     skew_kurtosis = signals.get("skew_kurtosis_score", 0.0)
261     # Distance models are sensitive to high p/n, sparsity, missingness, and large n.
262     size_penalty = 0.0
263     if n_samples >= 100_000:

```

```
264         size_penalty = 0.4
265     elif n_samples >= 50_000:
266         size_penalty = 0.2
267
268     distance_score = (
269         (1.0 - sparsity)
270         - cat_ratio * 0.25
271         - missing_ratio * 0.2
272         - max(p_over_n - 1.0, 0.0) * 0.2
273         - size_penalty
274     )
275     if not distance_model_eligible(signals):
276         distance_score = -1.0
277
278     tree_boost = 0.0
279     if nonlin_idx and nonlin_idx > 1.2:
280         tree_boost += 0.1
281     if missing_ratio >= 0.1:
282         tree_boost += 0.05
283     if cat_ratio >= 0.3:
284         tree_boost += 0.05
285
286     scores = {
287         "linear_models": (
288             linear
289             - sparsity * 0.3
290             - cat_ratio * 0.2
291             - max(p_over_n - 1.0, 0.0) * 0.1
292             - skew_kurtosis * 0.1
293         ),
294         "tree_based": (
295             nonlinear
296             + cat_ratio * 0.2
297             + missing_ratio * 0.2
298             + max(nonline_idx - 1.0, 0.0) * 0.1
299             + tree_boost
300         ),
301         "distance_based": distance_score
302     }
303
304     if task_type == "classification":
305         scores["probabilistic"] = 0.7 + cat_ratio * 0.2 + max(p_over_n - 1.0, 0.0) *
306         0.1
307
308         for k in list(scores.keys()):
309             scores[k] = max(min(float(scores[k]), 1.0), -1.0)
310
311     return scores
312
313 # ----- Step 6 -----
314
315 def step6_final_family_selection(scores):
316     if not scores:
```

```
317         return {"chosen_family": None, "reason": "no_valid_models"}
```

```
318
```

```
319     best = max(scores, key=scores.get)
```

```
320     return {
```

```
321         "chosen_family": best,
```

```
322         "generalization_score": round(max(min(float(scores[best])), 1.0), -1.0), 4),
```

```
323         "rule": "highest_statistical_signal"
```

```
324     }
```

```
325
```

```
326
```

```
327 def step6b_generalization_selection(scores, signals):
```

```
328     if not scores:
```

```
329         return {"chosen_family": None, "reason": "no_valid_models"}
```

```
330
```

```
331     n_samples = signals.get("n_samples", 0)
```

```
332     n_features = signals.get("n_features", 0)
```

```
333     sample_ratio = float(n_samples / max(n_features, 1))
```

```
334     tail_score = float(signals.get("skew_kurtosis_score") or 0.0)
```

```
335
```

```
336     variance_risk = {
```

```
337         "linear_models": 0.05,
```

```
338         "tree_based": 0.25,
```

```
339         "distance_based": 0.55,
```

```
340         "probabilistic": 0.10
```

```
341     }
```

```
342
```

```
343     def sample_stress(family):
```

```
344         if family == "distance_based":
```

```
345             return max(0.0, 1.0 - np.log(max(sample_ratio, 1e-6)) / 4.0)
```

```
346         if family == "tree_based":
```

```
347             return max(0.0, 1.0 - np.log(max(sample_ratio, 1e-6)) / 6.0)
```

```
348         return 0.0
```

```
349
```

```
350     def instability(family):
```

```
351         if family == "distance_based":
```

```
352             return min(0.3, tail_score / 20.0)
```

```
353         if family == "tree_based":
```

```
354             return min(0.15, tail_score / 30.0)
```

```
355         if family == "linear_models":
```

```
356             return min(0.1, tail_score / 40.0)
```

```
357         return min(0.1, tail_score / 40.0)
```

```
358
```

```
359     def distance_reliability(signals):
```

```
360         score = 1.0
```

```
361         nonlin = signals.get("nonlinearity_index", 0.0) or 0.0
```

```
362         score -= min(0.4, nonlin / 2)
```

```
363         score -= min(0.3, (signals.get("skew_kurtosis_score", 0.0) or 0.0) / 20)
```

```
364         score -= min(0.2, np.log(signals.get("n_features", 1) + 1) / 4)
```

```
365         if nonlin > 1.2:
```

```
366             score -= 0.25
```

```
367         elif nonlin > 0.8:
```

```
368             score -= 0.15
```

```
369         n = signals.get("n_samples", 0)
```

```
370         if n > 10_000:
```

```
371         score -= 0.20
372     if n > 25_000:
373         score -= 0.35
374     return max(0.0, score)
375
376     family_prior = {
377         "tree_based": 0.10,
378         "linear_models": 0.05,
379         "distance_based": -0.15
380     }
381
382     robustness_bonus = {
383         "linear_models": 0.15,
384         "tree_based": 0.10,
385         "distance_based": 0.00,
386         "probabilistic": 0.05
387     }
388
389     generalization_scores = {}
390     for family, s_signal in scores.items():
391         s_adj = float(s_signal)
392
393         if family == "distance_based":
394             s_adj *= distance_reliability(signals)
395
396         g = (
397             0.45 * s_adj
398             - 0.20 * variance_risk.get(family, 0.15)
399             - 0.15 * sample_stress(family)
400             - 0.10 * instability(family)
401             + 0.10 * robustness_bonus.get(family, 0.05)
402             + family_prior.get(family, 0.0)
403         )
404
405         # ✅ FIX 1: probabilistic models collapse under nonlinear interactions
406         if family == "probabilistic":
407             if signals.get("nonlinearity_index", 0.0) >= 0.8:
408                 g -= 0.25
409
410         # existing RF preference
411         if family == "tree_based" and random_forest_preferred(signals):
412             g += RF_PRIOR_BONUS
413
414         # ✅ FIX 2: reward trees for real interaction capture
415         if family == "tree_based" and signals.get("nonlinearity_index", 0.0) >= 1.0:
416             g += 0.20
417
418         generalization_scores[family] = max(min(float(g), 1.0), -1.0)
419
420         distance = generalization_scores.get("distance_based", -1.0)
421         tree = generalization_scores.get("tree_based", -1.0)
422         linear = generalization_scores.get("linear_models", -1.0)
423
424         if distance >= 0:
```

```

425         if not (
426             distance > tree + DISTANCE_WIN_MARGIN
427             and distance > linear + DISTANCE_WIN_MARGIN
428         ):
429             generalization_scores["distance_based"] = max(
430                 min(distance - 0.3, 1.0), -1.0
431             )
432
433     if (
434         "distance_based" in generalization_scores
435         and "tree_based" in generalization_scores
436     ):
437         if (
438             generalization_scores["distance_based"]
439             - generalization_scores["tree_based"]
440             < 0.15
441         ):
442             best = "tree_based"
443         else:
444             best = max(generalization_scores, key=generalization_scores.get)
445     else:
446         best = max(generalization_scores, key=generalization_scores.get)
447
448     return {
449         "chosen_family": best,
450         "generalization_score": round(float(generalization_scores[best]), 4),
451         "rule": "multi_signal_generalization_score",
452         "generalization_scores": {
453             k: round(float(v), 4) for k, v in generalization_scores.items()
454         }
455     }
456
457
458
459 # ----- Step 7 -----
460
461 def step7_family_risks(signals):
462     return {
463         "linear_models": "risk" if signals["linear_signal_strength"] <
464             signals["nonlinear_signal_strength"] else "low",
465         "tree_based": "risk" if signals["linear_signal_strength"] > 0.7 else "low",
466         "distance_based": "risk" if signals["sparsity_ratio"] > 0.4 else "low"
467     }
468
469 # ----- Step 8 -----
470
471 def step8_hyperparameter_hints(df):
472     rows, cols = df.shape
473     hints = {}
474
475     if rows > 100_000:
476         hints["tree_depth"] = "limit"
477     if cols > 100:

```

```
478         hints["feature_sampling"] = "enable"
479
480     return hints
481
482
483 # ----- Step 9 -----
484
485 def step9_family_algorithms(task_type):
486     families = {
487         "linear_models": [
488             "linear_regression",
489             "ridge",
490             "lasso",
491             "elastic_net",
492             "logistic_regression",
493             "linear_svm"
494         ],
495         "tree_based": [
496             "decision_tree",
497             "random_forest",
498             "extra_trees",
499             "gradient_boosting",
500             "xgboost_or_lightgbm"
501         ],
502         "distance_based": [
503             "knn",
504             "svm_rbf"
505         ],
506         "probabilistic": [
507             "naive_bayes",
508             "gaussian_process"
509         ]
510     }
511
512     if task_type == "regression":
513         families["probabilistic"] = ["bayesian_ridge", "gaussian_process"]
514     if task_type != "classification":
515         families["linear_models"] = [
516             "linear_regression",
517             "ridge",
518             "lasso",
519             "elastic_net"
520         ]
521
522     return families
523
524
525 # ----- Step 10 -----
526
527 def step10_algorithm_tests(df, target_col, task_type, signals):
528     df = smart_sample(df, MAX_MI_ROWS)
529     y = df[target_col].dropna()
530     X = df.drop(columns=[target_col]).loc[y.index]
531     numeric_X = X.select_dtypes(include="number")
```

```

532     categorical_X = X.select_dtypes(exclude="number")
533
534     tests = {}
535
536     # Target distribution tests (for linear assumptions)
537     if pd.api.types.is_numeric_dtype(y):
538         y_vals = y.astype(float).values
539         if len(y_vals) >= 8:
540             tests["target_normality_pvalue"] =
541                 _safe_float(float(normaltest(y_vals).pvalue))
542             tests["target_jarque_bera_pvalue"] =
543                 _safe_float(float(jarque_bera(y_vals).pvalue))
544         else:
545             tests["target_normality_pvalue"] = None
546             tests["target_jarque_bera_pvalue"] = None
547
548     # Feature-level diagnostics
549     if numeric_X.shape[1] > 0:
550         corr = numeric_X.corrwith(y).abs()
551         tests["top_linear_corr"] = _safe_float(float(corr.max())) if len(corr) else
552             None
553         tests["mean_linear_corr"] = _safe_float(float(corr.mean())) if len(corr) else
554             None
555         tests["zero_variance_features"] = int((numeric_X.nunique() <= 1).sum())
556         tests["high_sparsity_features"] = int(((numeric_X == 0).mean() > 0.8).sum())
557     else:
558         tests["top_linear_corr"] = None
559         tests["mean_linear_corr"] = None
560         tests["zero_variance_features"] = 0
561         tests["high_sparsity_features"] = 0
562         tests["categorical_features"] = int(categorical_X.shape[1])
563
564     # Algo-specific guidance
565     tests["knn_scale_required"] = True if numeric_X.shape[1] > 0 else False
566     tests["svm_rbf_recommended"] = True if signals["nonlinear_signal_strength"] >
567         signals["linear_signal_strength"] else False
568     tests["linear_models_recommended"] = True if signals["linear_signal_strength"] >
569         0.2 else False
570     tests["tree_models_recommended"] = True if signals["nonlinear_signal_strength"] >
571         0.2 else False
572     tests["distance_models_recommended"] = bool(
573         signals.get("categorical_ratio", 0.0) < 0.3
574         and signals.get("sparsity_ratio", 1.0) < 0.4
575         and signals.get("p_over_n", 1.0) < 1.0
576         and signals.get("n_samples", 0) <= 50_000
577     )
578
579     if task_type == "classification":
580         tests["class_imbalance_ratio"] =
581             _safe_float(float(y.value_counts(normalize=True).max()))
582
583         # ----- Tree / Boosting Diagnostics -----
584
585         # Interaction proxy: MI vs correlation gap

```

```
578     interaction_strength = None
579     if (
580         tests.get("mean_linear_corr") is not None
581         and signals.get("nonlinear_signal_strength") is not None
582     ):
583         interaction_strength = max(
584             0.0,
585             signals["nonlinear_signal_strength"] - tests["mean_linear_corr"]
586         )
587
588     # Noise proxy (lower is better for boosting)
589     noise_proxy = None
590     if tests.get("top_linear_corr") is not None:
591         noise_proxy = 1.0 - tests["top_linear_corr"]
592
593     tests["tree_interaction_strength"] = _safe_float(interaction_strength)
594     tests["noise_proxy"] = _safe_float(noise_proxy)
595
596     # Random Forest suitability
597     tests["random_forest_suitable"] = bool(
598         interaction_strength is not None
599         and interaction_strength >= 0.15
600         and signals.get("n_samples", 0) >= 300
601         and signals.get("sparsity_ratio", 0.0) <= 0.5
602     )
603
604     # Gradient Boosting suitability
605     tests["gradient_boosting_suitable"] = bool(
606         interaction_strength is not None
607         and interaction_strength >= 0.20
608         and noise_proxy is not None
609         and noise_proxy <= 0.6
610         and signals.get("n_samples", 0) >= 1_000
611         and signals.get("sparsity_ratio", 0.0) <= 0.4
612     )
613
614     # XGBoost / LightGBM suitability
615     tests["xgboost_lightgbm_suitable"] = bool(
616         signals.get("nonlinear_signal_strength", 0.0) >= 0.30
617         and (
618             signals.get("skew_kurtosis_score", 0.0) >= 1.5
619             or signals.get("missing_ratio", 0.0) >= 0.1
620         )
621         and signals.get("n_samples", 0) >= 2_000
622     )
623
624     return tests
625
626
627 # ----- Step 11 -----
628 def step11_boosting_needed(df, target_col, task_type, signals, generalization_scores):
629     rows, cols = df.shape
630
631     nonlin = signals.get("nonlinear_signal_strength", 0.0)
```

```

632     linear = signals.get("linear_signal_strength", 0.0)
633     sparsity = signals.get("sparsity_ratio", 0.0)
634     skew_kurt = signals.get("skew_kurtosis_score", 0.0)
635     n = signals.get("n_samples", 0)
636
637     tree_score = generalization_scores.get("tree_based", -1.0)
638     linear_score = generalization_scores.get("linear_models", -1.0)
639
640     boosting_recommended = (
641         tree_score >= 0.15
642         and tree_score > linear_score + 0.10
643         and nonlin >= 0.3
644         and sparsity <= 0.4
645         and skew_kurt >= 1.5
646         and n >= 1_000
647         and n <= 200_000
648     )
649
650     return {
651         "boosting_recommended": bool(boosting_recommended),
652         "preferred_boosting_model": (
653             "xgboost_or_lightgbm" if boosting_recommended else None
654         ),
655         "rule": "tree_generalization_high_and_structured_nonlinearity_and_sufficient_data",
656         "signals": {
657             "tree_generalization_score": tree_score,
658             "linear_generalization_score": linear_score,
659             "nonlinear_signal_strength": nonlin,
660             "skew_kurtosis_score": skew_kurt,
661             "rows": n,
662             "cols": cols
663         }
664     }
665
666
667 # ----- Step 12 -----
668
669 def step12_top_algorithms_min_latency(task_type, signals, scores):
670     """
671     Exposes candidate algorithms per family with a latency-aware bias,
672     but does NOT prematurely suppress Random Forest or Boosting when
673     the data clearly supports them.
674     """
675
676     top = {}
677
678     # ----- Linear family (always cheap, always baseline) -----
679     if task_type == "classification":
680         top["linear_models"] = ["logistic_regression", "linear_svm"]
681     else:
682         top["linear_models"] = ["linear_regression", "ridge"]
683
684     # ----- Tree family (progressive exposure) -----

```

```

685     tree_algos = ["decision_tree", "extra_trees"]
686
687     # Prefer Random Forest when structure + data size justify it
688     if random_forest_preferred(signals):
689         tree_algos.append("random_forest")
690
691     # Allow boosting only when signal + scale justify it
692     if (
693         signals.get("nonlinear_signal_strength", 0.0) >= 0.30
694         and signals.get("n_samples", 0) >= 1_000
695         and signals.get("sparsity_ratio", 0.0) <= 0.4
696     ):
697         tree_algos.extend([
698             "gradient_boosting",
699             "xgboost_or_lightgbm"
700         ])
701
702     top["tree_based"] = tree_algos
703
704     # ----- Distance family (strict gating) -----
705     distance_ok = (
706         scores.get("distance_based", 0.0) >= 0.25
707         and signals.get("categorical_ratio", 0.0) < 0.3
708         and signals.get("sparsity_ratio", 1.0) < 0.5
709         and signals.get("p_over_n", 1.0) < 1.0
710         and signals.get("n_samples", 0) <= 50_000
711         and signals.get("nonlinearity_index", 0.0) <= 1.0
712     )
713
714     top["distance_based"] = ["knn", "svm_rbf"] if distance_ok else []
715
716     # ----- Probabilistic (baseline, not dominant) -----
717     if task_type == "classification":
718         top["probabilistic"] = ["naive_bayes", "gaussian_process"]
719     else:
720         top["probabilistic"] = ["bayesian_ridge", "gaussian_process"]
721
722     # ----- Family ranking (used later by Step 13 / 15) -----
723     family_rank = sorted(scores.items(), key=lambda kv: kv[1], reverse=True)
724     ordered_families = [f for f, _ in family_rank]
725
726     return {
727         "ordered_families": ordered_families,
728         "top_algorithms_by_family": top,
729         "latency_bias": "favor_simple_models_before_boosting"
730     }
731
732
733
734     # ----- Step 13 -----
735
736     def step13_top5_algorithms(step12, best_family):
737         MAX_PER_FAMILY = 2
738         ordered = step12["ordered_families"]

```

```

739     by_family = step12["top_algorithms_by_family"]
740     top5 = []
741     family_counts = defaultdict(int)
742     for fam in ordered:
743         for algo in by_family.get(fam, []):
744             if family_counts[fam] >= MAX_PER_FAMILY:
745                 continue
746             if algo not in top5:
747                 top5.append(algo)
748                 family_counts[fam] += 1
749             if len(top5) >= 5:
750                 break
751             if len(top5) >= 5:
752                 break
753
754     if best_family == "distance_based":
755         tree_algos = by_family.get("tree_based", [])
756         has_tree = any(a in tree_algos for a in top5)
757         if not has_tree and tree_algos:
758             # Ensure at least one tree-based model is present
759             if len(top5) >= 5:
760                 top5.pop()
761             top5.append(tree_algos[0])
762
763     return {
764         "top_5_algorithms": top5,
765         "selection_rule": "family_rank_then_latency_bias"
766     }
767
768
769 # ----- Step 15 -----
770
771 def step15_overall_best5_models(step12, generalization_scores, algo_tests, signals,
772                                 boosting_info):
773     by_family = step12["top_algorithms_by_family"]
774
775     ranked = []
776
777     # ---- Build scored candidate list ----
778     for fam, fam_score in generalization_scores.items():
779         if fam_score < -0.1:
780             continue
781
782         for algo in by_family.get(fam, []):
783             bonus = 0.0
784
785             # Model-specific suitability bonuses
786             if algo == "random_forest" and random_forest_preferred(signals):
787                 bonus += 0.20
788
789             if algo == "gradient_boosting" and
790             algo_tests.get("gradient_boosting_suitable"):
791                 bonus += 0.25
792
793

```

```

791         if algo == "xgboost_or_lightgbm" and
792             algo_tests.get("xgboost_lightgbm_suitable"):
793                 bonus += 0.30
794
795             ranked.append((algo, fam, fam_score + bonus))
796
797             # Sort by adjusted score
798             ranked.sort(key=lambda x: x[2], reverse=True)
799
800             # ---- Enforce family diversity (THIS WAS MISSING) ----
801             MAX_PER_FAMILY = 2
802             family_counts = {}
803             final = []
804
805             for algo, fam, score in ranked:
806                 family_counts.setdefault(fam, 0)
807
808                 if family_counts[fam] >= MAX_PER_FAMILY:
809                     continue
810
811                 final.append((algo, fam, score))
812                 family_counts[fam] += 1
813
814                 if len(final) >= 5:
815                     break
816
817             return {
818                 "overall_best_5_models": [
819                     {
820                         "algorithm": a,
821                         "family": f,
822                         "generalization_score": round(float(s), 4),
823                         "selection_reason": (
824                             "preferred_tree_model"
825                             if a == "random_forest" and random_forest_preferred(signals)
826                             else "boosting_recommended"
827                             if a in ["gradient_boosting", "xgboost_or_lightgbm"]
828                             and boosting_info.get("boosting_recommended")
829                             else "baseline_candidate"
830                         )
831                     }
832                     for a, f, s in final
833                 ],
834                 "selection_rule": "generalization_score_rank_with_family_diversity"
835             }
836
837
838             # ----- Step 14 -----
839
840             def step14_condition_based_recommendations(df, target_col, task_type, signals,
841                 algo_tests):
842                 rows, cols = df.shape
843                 n = rows

```

```
843     p = max(cols - 1, 0)
844     cat_ratio = signals.get("categorical_ratio", 0.0)
845     sparsity = signals.get("sparsity_ratio", 0.0)
846     linear = signals.get("linear_signal_strength", 0.0)
847     nonlinear = signals.get("nonlinear_signal_strength", 0.0)
848
849     recs = []
850
851     def add_rule(condition, models, rationale):
852         if condition:
853             recs.append({
854                 "recommended_models": models,
855                 "rationale": rationale
856             })
857
858         add_rule(
859             n < 1_000 and p <= 50,
860             ["logistic_regression", "linear_regression", "naive_bayes",
861             "decision_tree_small", "knn"],
862             "Very small dataset; prefer simple models to reduce overfitting."
863         )
864
865         add_rule(
866             n <= 200_000 and p <= 200,
867             ["random_forest", "xgboost_or_lightgbm", "ridge"],
868             "Moderate size; trees capture interactions, GLM as baseline."
869         )
870
871         add_rule(
872             n > 200_000,
873             ["sgd_linear", "linear_svm", "lightgbm_hist"],
874             "Very large dataset; prioritize scalable models."
875         )
876
877         add_rule(
878             p > n * 2,
879             ["lasso", "elastic_net", "feature_selection_plus_glm"],
880             "High-dimensional (p >> n); regularized linear models or feature selection."
881         )
882
883         add_rule(
884             sparsity >= 0.6,
885             ["linear_svm_sparse", "logistic_regression_sparse", "naive_bayes"],
886             "High sparsity; linear or NB models are robust."
887         )
888
889         add_rule(
890             cat_ratio >= 0.4,
891             ["catboost", "lightgbm", "random_forest", "target_encoding_plus_glm"],
892             "Many categorical features; tree boosting handles categorical splits."
893         )
894         add_rule(
895             signals.get("missing_ratio", 0.0) >= 0.2,
896             ["lightgbm", "xgboost_or_lightgbm", "random_forest"],
```

```
896         "High missingness; tree-based models handle missing values better."
897     )
898     add_rule(
899         signals.get("skew_kurtosis_score", 0.0) >= 2.0,
900         ["random_forest", "xgboost_or_lightgbm", "robust_regression_huber"],
901         "Heavy tails/outliers; prefer trees or robust regression."
902     )
903
904     add_rule(
905         task_type == "regression" and linear >= 0.3 and nonlinear <= 0.2,
906         ["ridge", "lasso", "elastic_net"],
907         "Continuous target with linear signal; linear models are sufficient."
908     )
909
910     add_rule(
911         task_type == "regression" and nonlinear > linear + 0.1,
912         ["random_forest", "xgboost_or_lightgbm", "svm_rbf"],
913         "Continuous target with nonlinear signal; tree ensembles or kernels perform
better."
914     )
915
916     add_rule(
917         task_type == "classification",
918         ["logistic_regression", "random_forest", "xgboost_or_lightgbm"],
919         "Classification: start with GLM + trees + SVM."
920     )
921
922     if task_type == "classification":
923         imb = algo_tests.get("class_imbalance_ratio")
924         add_rule(
925             imb is not None and imb >= 0.8,
926             ["xgboost_weighted", "random_forest_weighted",
927             "weighted_logistic_regression"],
928             "Imbalanced classes; use class-weighted models or boosting with weights."
929         )
930
931         add_rule(
932             algo_tests.get("high_sparsity_features", 0) >= max(1, int(p * 0.3)),
933             ["linear_svm_sparse", "logistic_regression_sparse"],
934             "Many sparse features; sparse linear models perform well."
935         )
936         add_rule(
937             algo_tests.get("distance_models_recommended", False),
938             ["knn", "svm_linear"],
939             "Low dimensional, mostly numeric, and low sparsity; distance models can work
well."
940         )
941
942         return {
943             "rules_triggered": recs,
944             "note": "Rules are heuristics; use as guidance with validation."
945         }
946
```

```
947     # ----- API -----
948
949     @app.post("/eda/phase1_3")
950     def run_optimized_eda(
951         file: UploadFile = File(...),
952         target_column: str = Form(...)
953     ):
954         csv_path = None
955
956         try:
957             csv_path = save_temp_file(file)
958             step1, df = step1_dataset_overview(csv_path)
959
960             if target_column not in df.columns:
961                 raise HTTPException(
962                     status_code=400,
963                     detail=f"Target column '{target_column}' not found"
964                 )
965
966             step2 = step2_target_statistics(df[target_column])
967             step3 = step3_problem_inference(step2)
968             step4 = step4_feature_target_signals(df, target_column, step3["problem_type"])
969             step5 = step5_model_family_scoring(step3["problem_type"], step4)
970             step6 = step6b_generalization_selection(step5, step4)
971             step7 = step7_family_risks(step4)
972             step8 = step8_hyperparameter_hints(df)
973             step9 = step9_family_algorithms(step3["problem_type"])
974             step10 = step10_algorithm_tests(df, target_column, step3["problem_type"],
975                 step4)
976             step11 = step11_boosting_needed(
977                 df,
978                 target_column,
979                 step3["problem_type"],
980                 step4,
981                 step6["generalization_scores"]
982             )
983
984             step12 = step12_top_algorithms_min_latency(step3["problem_type"], step4,
985                 step5)
986             step13 = step13_top5_algorithms(step12, step6["chosen_family"])
987             step14 = step14_condition_based_recommendations(df, target_column,
988                 step3["problem_type"], step4, step10)
989             step15 = step15_overall_best5_models(
990                 step12=step12,
991                 generalization_scores=step6["generalization_scores"],
992                 algo_tests=step10,
993                 signals=step4,
994                 boosting_info=step11
995             )
996
997             response = {
```

```
998     "STEP_1_DATASET_OVERVIEW": step1,
999     "STEP_2_TARGET_STATISTICS": step2,
1000    "STEP_3_PROBLEM_INFERENCE": step3,
1001    "STEP_4_FEATURE_TARGET_SIGNALS": step4,
1002    "STEP_5_MODEL_FAMILY_SCORING": step5,
1003    "STEP_6_FAMILY_RISK_DIAGNOSTICS": step7,
1004    "STEP_7_HYPERPARAMETER_HINTS": step8,
1005    "STEP_8_FINAL_MODEL_SELECTION": step6,
1006    "distance_models_status": {
1007        "eligible": distance_model_eligible(step4),
1008        "confidence": "conditional",
1009        "conditions": [
1010            "low dimensional",
1011            "mostly numeric",
1012            "low sparsity",
1013            "no heavy skew/kurtosis"
1014        ]
1015    },
1016    "STEP_9_FAMILY_ALGORITHMS": step9,
1017    "STEP_10_ALGO_SPECIFIC_TESTS": step10,
1018    "STEP_11_BOOSTING_RECOMMENDATION": step11,
1019    "STEP_12_TOP_ALGOS_MIN_LATENCY": step12,
1020    "STEP_13_TOP_5_ALGOS": step13,
1021    "STEP_14_CONDITION_BASED_RECOMMENDATIONS": step14,
1022    "STEP_15_OVERALL_BEST_5_MODELS": step15
1023}
1024
1025    return to_json_safe(response)
1026
1027 finally:
1028     if csv_path and os.path.exists(csv_path):
1029         os.remove(csv_path)
1030
```