

```

1 import numpy as np
2 import pandas as pd
3 import argparse
4 import matplotlib.pyplot as plt
5 from sklearn.linear_model import LogisticRegression
6 from sklearn.tree import DecisionTreeClassifier
7 from sklearn.ensemble import RandomForestClassifier
8 from sklearn.svm import SVC
9 from sklearn.model_selection import train_test_split
10 from sklearn.metrics import accuracy_score
11 from sklearn.base import BaseEstimator, ClassifierMixin
12 from tabulate import tabulate
13 import warnings
14 warnings.filterwarnings('ignore')
15
16
17 # Constraint Enforcer Classes
18 class ConstraintEnforcer:
19     def __init__(self, threshold=0.3):
20         self.threshold = threshold
21         self.mutex_pairs = []
22         self.implications = []
23
24     def add_mutex(self, a, b):
25         a_key = f'class_{a}' if isinstance(a, int) else a
26         b_key = f'class_{b}' if isinstance(b, int) else b
27         self.mutex_pairs.append((a_key, b_key))
28
29     def add_implication(self, a, b):
30         a_key = f'class_{a}' if isinstance(a, int) else a
31         b_key = f'class_{b}' if isinstance(b, int) else b
32         self.implications.append((a_key, b_key))
33
34     def _single_pass(self, preds):
35         # mutual exclusion pruning
36         for a, b in self.mutex_pairs:
37             if preds[a] > self.threshold and preds[b] > self.threshold:
38                 if preds[a] > preds[b]:
39                     preds[b] *= 0.3
40                 else:
41                     preds[a] *= 0.3
42
43         # implication enforcement
44         for a, b in self.implications:
45             if preds[a] > self.threshold and preds[b] < self.threshold:
46                 deficit = self.threshold - preds[b]
47                 available = preds[a] - self.threshold
48                 max_transfer = min(available * 0.3, deficit, preds[a] * 0.3)
49                 if max_transfer > 0:
50                     preds[a] = max(0.0, preds[a] - max_transfer)
51                     preds[b] = min(1.0, preds[b] + max_transfer)
52
53     def enforce(self, probs):
54         preds = {f'class_{j}': float(probs[j]) for j in range(len(probs))}
55         for _ in range(2):
56             self._single_pass(preds)
57         for k in preds:
58             if preds[k] < 0.0:
59                 preds[k] = 0.0
60         total = sum(preds.values())
61         if total > 1e-12:
62             inv = 1.0 / total
63             for k in preds:
64                 preds[k] *= inv
65         return np.array([preds[f'class_{j}'] for j in range(len(probs))])
66
67 # Fundamental PhIML Approaches
68 class ConstraintAwareLossClassifier(BaseEstimator, ClassifierMixin):
69     """Simulates constraint-aware loss by reweighting training samples."""
70     def __init__(self, base_classifier, constraints, constraint_weight=0.8, iterations=10, threshold=0.3):
71         self.base_classifier = base_classifier
72         self.constraints = constraints
73         self.constraint_weight = constraint_weight
74         self.iterations = iterations
75         self.threshold = threshold

```

```

76         self.fitted_classifiers = []
77
78     def fit(self, X, y):
79         from sklearn.base import clone
80
81         supports_weight = True
82         try:
83             probe = clone(self.base_classifier)
84             probe.fit(X[:,2], y[:,2], sample_weight=np.ones(2))
85         except Exception:
86             supports_weight = False
87         self.fitted_classifiers = []
88         if not supports_weight:
89             probs = None
90             for iteration in range(self.iterations):
91                 clf = clone(self.base_classifier)
92                 if iteration == 0 or probs is None:
93                     clf.fit(X, y)
94                 else:
95                     violation_scores = np.zeros(len(X), dtype=float)
96                     for idx, prob in enumerate(probs):
97                         vs = 0.0
98                         for c_type, i, j in self.constraints:
99                             if c_type == 'mutex' and prob[i] > self.threshold and prob[j] > self.threshold:
100                                 vs += min(prob[i], prob[j])
101                             elif c_type == 'implies' and prob[i] > self.threshold and prob[j] < self.threshold:
102                                 vs += prob[i] * (1.0 - prob[j])
103                     violation_scores[idx] = vs
104                     scale = 10.0 * self.constraint_weight
105                     w = np.exp(-scale * violation_scores)
106                     w_sum = w.sum()
107                     if w_sum <= 0:
108                         w = np.ones_like(w) / len(w)
109                     else:
110                         w /= w_sum
111                     idxs = np.random.choice(len(X), size=len(X), replace=True, p=w)
112                     clf.fit(X[idxs], y[idxs])
113                 self.fitted_classifiers.append(clf)
114                 probs = clf.predict_proba(X)
115             else:
116                 sample_weights = np.ones(len(X), dtype=float)
117                 for iteration in range(self.iterations):
118                     clf = clone(self.base_classifier)
119                     if iteration == 0:
120                         clf.fit(X, y)
121                     else:
122                         clf.fit(X, y, sample_weight=sample_weights)
123                     self.fitted_classifiers.append(clf)
124                     if iteration == self.iterations - 1:
125                         continue
126                     probs = clf.predict_proba(X)
127                     violation_scores = np.zeros(len(X), dtype=float)
128                     for idx, prob in enumerate(probs):
129                         vs = 0.0
130                         for c_type, i, j in self.constraints:
131                             if c_type == 'mutex' and prob[i] > self.threshold and prob[j] > self.threshold:
132                                 vs += min(prob[i], prob[j])
133                             elif c_type == 'implies' and prob[i] > self.threshold and prob[j] < self.threshold:
134                                 vs += prob[i] * (1.0 - prob[j])
135                     violation_scores[idx] = vs
136
137                     wf = self.constraint_weight * (iteration + 1) / self.iterations
138                     arg = -wf * violation_scores * 10.0
139                     arg = np.clip(arg, -50, 50)
140                     sample_weights = np.exp(arg)
141                     mean_sw = sample_weights.mean()
142                     if mean_sw <= 0:
143                         sample_weights = np.ones_like(sample_weights)
144                     else:
145                         sample_weights /= mean_sw
146
147         self.classes_ = np.unique(y)
148         return self
149
150     def predict(self, X):
151         return self.classes_[np.argmax(self.predict_proba(X), axis=1)]
152
153     def predict_proba(self, X):

```

```

154         if self.fitted_classifiers:
155             all_probs = np.array([clf.predict_proba(X) for clf in self.fitted_classifiers])
156             return all_probs.mean(axis=0)
157         else:
158             return self.base_classifier.predict_proba(X)
159
160 class LogicGuidedClassifier(BaseEstimator, ClassifierMixin):
161     """Implements a logic layer that guides predictions"""
162     def __init__(self, base_classifier, constraints, threshold=0.3, strength=1.2):
163         self.base_classifier = base_classifier
164         self.constraints = constraints
165         self.threshold = threshold
166         self.strength = strength
167
168     def fit(self, X, y):
169         self.base_classifier.fit(X, y)
170         self.classes_ = self.base_classifier.classes_
171         return self
172
173     def _apply_logic_layer(self, probs):
174         adjusted_probs = probs.copy()
175
176         for c_type, i, j in self.constraints:
177             if c_type == 'mutex':
178                 mask = (adjusted_probs[:, i] > self.threshold) & (adjusted_probs[:, j] > self.threshold)
179                 if np.any(mask):
180                     violation = np.minimum(adjusted_probs[mask, i], adjusted_probs[mask, j])
181                     reduction = violation * self.strength
182                     adjusted_probs[mask, i] = np.maximum(0.0, adjusted_probs[mask, i] - reduction)
183                     adjusted_probs[mask, j] = np.maximum(0.0, adjusted_probs[mask, j] - reduction)
184             elif c_type == 'implies':
185                 mask = (adjusted_probs[:, i] > self.threshold) & (adjusted_probs[:, j] < self.threshold)
186                 if np.any(mask):
187                     deficit = self.threshold - adjusted_probs[mask, j]
188                     transfer = np.minimum(adjusted_probs[mask, i] * 0.5, deficit)
189                     adjusted_probs[mask, i] = np.maximum(0.0, adjusted_probs[mask, i] - transfer)
190                     adjusted_probs[mask, j] = np.minimum(1.0, adjusted_probs[mask, j] + transfer)
191
192         row_sums = adjusted_probs.sum(axis=1, keepdims=True)
193         row_sums[row_sums == 0] = 1.0
194         adjusted_probs = adjusted_probs / row_sums
195         return adjusted_probs
196
197     def predict_proba(self, X):
198         base_probs = self.base_classifier.predict_proba(X)
199         return self._apply_logic_layer(base_probs)
200
201     def predict(self, X):
202         probs = self.predict_proba(X)
203         return self.classes_[np.argmax(probs, axis=1)]
204
205 # Utility: common classifier factory
206 def _get_classifier(alg):
207     return {
208         'logistic': LogisticRegression(multi_class='multinomial', solver='lbfgs',
209                                         max_iter=500, random_state=42),
210         'decision_tree': DecisionTreeClassifier(random_state=42),
211         'random_forest': RandomForestClassifier(random_state=42),
212         'svm': SVC(probability=True, random_state=42),
213     }[alg]
214
215 # Scenario 1: Document Classification
216 def scenario_document_classification(alg, threshold=0.3):
217     print("\n=== Scenario 1: Legal Document Classification ===")
218     print("Description: Classify documents as Contract, Patent, or General Legal.")
219     print("Constraint: Contract and Patent are mutually exclusive document types.")
220     print("Violations: fraction where both p(Contract) and p(Patent) > threshold.\n")
221
222     np.random.seed(42)
223     n_per_class = 150
224
225     # Feature extraction simulation (e.g., word frequencies, document structure)
226     # General legal documents - broad vocabulary
227     general = np.random.randn(n_per_class, 5) * [0.5, 0.5, 0.4, 0.3, 0.6] + [0, 0, 0, 0, 0]
228
229     # Contracts - specific terminology, structured sections
230     contracts = np.random.randn(n_per_class, 5) * [0.3, 0.4, 0.3, 0.5, 0.3] + [2, 1.5, -1, 0.5, 1]

```

```

232
233     # Patents - technical language, claims structure
234     patents = np.random.randn(n_per_class, 5) * [0.3, 0.3, 0.5, 0.3, 0.4] + [-1, 2, 1.5, -0.5, 1.5]
235
236     # Ambiguous documents (e.g., licensing agreements with technical specs)
237     n_ambiguous = 100
238     ambiguous = np.random.randn(n_ambiguous, 5) * [0.6, 0.6, 0.6, 0.6, 0.6] + [0.5, 1.75, 0.25, 0, 1.25]
239     # Half lean toward contracts, half toward patents
240     ambiguous_labels = np.concatenate([np.ones(n_ambiguous//2, dtype=int),
241                                       np.full(n_ambiguous//2, 2, dtype=int)])
242     np.random.shuffle(ambiguous_labels)
243
244     X = np.vstack([general, contracts, patents, ambiguous])
245     y = np.concatenate([
246         np.zeros(n_per_class, dtype=int),
247         np.ones(n_per_class, dtype=int),
248         np.full(n_per_class, 2, dtype=int),
249         ambiguous_labels
250     ])
251
252     clf = _get_classifier(alg)
253     print(f"--- Algorithm: {alg.replace('_', ' ').title()} ---")
254
255     X_train, X_test, y_train, y_test = train_test_split(
256         X, y, test_size=0.3, random_state=42, stratify=y)
257     clf.fit(X_train, y_train)
258     probs = clf.predict_proba(X_test)
259     base_preds = clf.predict(X_test)
260
261     vio_pre = np.mean((probs[:,1] > threshold) & (probs[:,2] > threshold))
262     acc_pre = accuracy_score(y_test, base_preds)
263
264     enf = ConstraintEnforcer(threshold)
265     enf.add_mutex('class_1', 'class_2')
266     adjusted = np.array([enf.enforce(p) for p in probs])
267     post_preds = np.argmax(adjusted, axis=1)
268
269     vio_post = np.mean((adjusted[:,1] > threshold) & (adjusted[:,2] > threshold))
270     acc_post = accuracy_score(y_test, post_preds)
271
272     print(f"Without PhIML: Violation = {vio_pre:.3f}, Accuracy = {acc_pre:.3f}")
273     print(f"With PhIML: Violation = {vio_post:.3f}, Accuracy = {acc_post:.3f}")
274
275     return {
276         "Scenario": "Document Classification",
277         "Algorithm": alg,
278         "Threshold": threshold,
279         "Violation Without PhIML": vio_pre,
280         "Violation With PhIML": vio_post,
281         "Accuracy Without PhIML": acc_pre,
282         "Accuracy With PhIML": acc_post
283     }
284
285 # Scenario 2: Disease Severity Hierarchy
286 def scenario_severity_hierarchy(alg, threshold=0.3):
287     print("\n=== Scenario 2: Disease Severity Hierarchy ===")
288     print("Description: Severity progression where severe conditions imply milder ones.")
289     print("Hierarchy: Pneumonia → Flu → Mild Symptoms.")
290     print("Violations: Cases where severe condition present but milder condition absent.\n")
291
292     np.random.seed(42)
293     n_samples = 500
294
295     # Generate severity scores (0-3)
296     severity = np.random.beta(2, 5, n_samples) * 3
297
298     # Features based on severity
299     X = np.column_stack([
300         severity + np.random.randn(n_samples) * 0.4,      # Primary symptom intensity
301         severity * 0.8 + np.random.randn(n_samples) * 0.5, # Secondary symptoms
302     ])
303
304     # Labels based on severity thresholds
305     y = np.zeros(n_samples, dtype=int)
306     y[severity > 0.8] = 1 # Flu
307     y[severity > 1.8] = 2 # Pneumonia
308
309     clf = _get_classifier(alg)

```

```

310     print(f"--- Algorithm: {alg.replace('_', ' ').title()} ---")
311
312     X_train, X_test, y_train, y_test = train_test_split(
313         X, y, test_size=0.3, random_state=42, stratify=y)
314     clf.fit(X_train, y_train)
315     probs = clf.predict_proba(X_test)
316     base_preds = clf.predict(X_test)
317
318     # Check hierarchy violations
319     vio_pre = np.mean(
320         ((probs[:,2] > threshold) & (probs[:,1] < threshold)) | # Pneumonia without flu
321         ((probs[:,1] > threshold) & (probs[:,0] < threshold))    # Flu without mild symptoms
322     )
323     acc_pre = accuracy_score(y_test, base_preds)
324
325     # Apply hierarchical constraints
326     enf = ConstraintEnforcer(threshold)
327     enf.add_implication('class_2', 'class_1') # Pneumonia implies flu
328     enf.add_implication('class_1', 'class_0') # Flu implies mild symptoms
329     adjusted = np.array([enf.enforce(p) for p in probs])
330     post_preds = np.argmax(adjusted, axis=1)
331
332     vio_post = np.mean(
333         ((adjusted[:,2] > threshold) & (adjusted[:,1] < threshold)) |
334         ((adjusted[:,1] > threshold) & (adjusted[:,0] < threshold))
335     )
336     acc_post = accuracy_score(y_test, post_preds)
337
338     print(f"Without PhIML: Violations = {vio_pre:.3f}, Accuracy = {acc_pre:.3f}")
339     print(f"With      PhIML: Violations = {vio_post:.3f}, Accuracy = {acc_post:.3f}")
340
341     return {
342         "Scenario": "Severity Hierarchy",
343         "Algorithm": alg,
344         "Threshold": threshold,
345         "Violation Without PhIML": vio_pre,
346         "Violation With PhIML":     vio_post,
347         "Accuracy Without PhIML":   acc_pre,
348         "Accuracy With PhIML":     acc_post
349     }
350
351
352 # Scenario 3: Constraint-Aware Loss Function
353 def scenario_constraint_aware_loss(alg, threshold=0.3):
354     print("\n=== Scenario 3: Constraint-Aware Loss Function ===")
355     print("Description: Train classifier with constraint violations influencing sample weights.")
356     print("Constraint: Flu and Pneumonia are mutually exclusive.\n")
357
358     np.random.seed(42)
359     n_samples = 600
360     healthy   = np.random.randn(n_samples//3, 4) * 0.3 + [0.0, 0.0, 0.0, 0.0]
361     flu       = np.random.randn(n_samples//3, 4) * 0.5 + [1.2, 1.0, 0.6, 1.2]
362     pneumonia = np.random.randn(n_samples//3, 4) * 0.5 + [1.4, 1.2, 0.9, 1.0]
363
364     X = np.vstack([healthy, flu, pneumonia])
365     y = np.concatenate([
366         np.zeros(n_samples//3, dtype=int),
367         np.ones(n_samples//3, dtype=int),
368         np.full(n_samples//3, 2, dtype=int)
369     ])
370
371     X_train, X_test, y_train, y_test = train_test_split(
372         X, y, test_size=0.3, random_state=42, stratify=y)
373
374     constraints = [('mutex', 1, 2)] # Flu and Pneumonia are mutually exclusive
375
376     print(f"--- Algorithm: {alg.replace('_', ' ').title()} with Constraint-Aware Loss ---")
377     base_clf = _get_classifier(alg)
378     clf = ConstraintAwareLossClassifier(
379         base_classifier=base_clf,
380         constraints=constraints,
381         constraint_weight=2,
382         threshold=threshold # Added
383     )
384     clf.fit(X_train, y_train)
385
386     # Evaluate PhIML
387     probs = clf.predict_proba(X_test)

```



```

388     preds = clf.predict(X_test)
389     violations = np.mean((probs[:,1] > threshold) & (probs[:,2] > threshold))
390     accuracy = accuracy_score(y_test, preds)
391     print(f"Constraint-Aware Training: Violations = {violations:.3f}, Accuracy = {accuracy:.3f}")
392
393     # Compare with standard
394     std_clf = _get_classifier(alg)
395     std_clf.fit(X_train, y_train)
396     std_probs = std_clf.predict_proba(X_test)
397     std_violations = np.mean((std_probs[:,1] > threshold) & (std_probs[:,2] > threshold))
398     std_accuracy = accuracy_score(y_test, std_clf.predict(X_test))
399     print(f"Standard Training: Violations = {std_violations:.3f}, Accuracy = {std_accuracy:.3f}")
400
401     return {
402         "Scenario": "Constraint-Aware Loss",
403         "Algorithm": alg,
404         "Threshold": threshold,
405         "Violation Without PhIML": std_violations,
406         "Violation With PhIML": violations,
407         "Accuracy Without PhIML": std_accuracy,
408         "Accuracy With PhIML": accuracy
409     }
410
411
412 # Scenario 4: Logic-Guided Architecture
413 def scenario_logic_guided(alg, threshold=0.3):
414     print("\n=== Scenario 4: Logic-Guided Architecture ===")
415     print("Description: Base learner + probability logic layer that enforces constraints.")
416     print("Constraints: Flu  $\perp$  Pneumonia; *Pneumonia implies Flu-like symptoms* (class_2  $\Rightarrow$  class_1).\n")
417
418     np.random.seed(42)
419     n_samples = 600
420     X = np.random.randn(n_samples, 5)
421     y = np.zeros(n_samples, dtype=int)
422     mask_healthy = (X[:,0] < -0.5) & (X[:,1] < -0.5)
423     mask_flu = (X[:,0] > -0.5) & (X[:,0] < 0.5) & (X[:,2] > 0)
424     mask_pneumonia = (X[:,0] > 0.5) & (X[:,3] > 0)
425     y[mask_healthy] = 0
426     y[mask_flu] = 1
427     y[mask_pneumonia] = 2
428
429     X_train, X_test, y_train, y_test = train_test_split(
430         X, y, test_size=0.3, random_state=42, stratify=y)
431     constraints = [('mutex',1,2), ('implies',2,1)]
432
433     print(f"--- Algorithm: {alg.replace('_', ' ').title()} with Logic-Guided Architecture ---")
434     base_clf = _get_classifier(alg)
435     clf = LogicGuidedClassifier(
436         base_classifier=base_clf,
437         constraints=constraints,
438         threshold=threshold,
439         strength=0.8
440     )
441     clf.fit(X_train, y_train)
442
443     # Evaluate PhIML
444     probs = clf.predict_proba(X_test)
445     preds = clf.predict(X_test)
446     mutex_vio = np.mean((probs[:,1] > threshold) & (probs[:,2] > threshold))
447     impl_vio = np.mean((probs[:,2] > threshold) & (probs[:,1] < threshold))
448     total_vio = (mutex_vio + impl_vio) / 2.0
449     accuracy = accuracy_score(y_test, preds)
450     print(f"Logic-Guided: Violations = {total_vio:.3f}, Accuracy = {accuracy:.3f}")
451
452     # Compare with standard
453     std_clf = _get_classifier(alg)
454     std_clf.fit(X_train, y_train)
455     std_probs = std_clf.predict_proba(X_test)
456     std_mutex = np.mean((std_probs[:,1] > threshold) & (std_probs[:,2] > threshold))
457     std_impl = np.mean((std_probs[:,2] > threshold) & (std_probs[:,1] < threshold))
458     std_vio = (std_mutex + std_impl) / 2.0
459     std_acc = accuracy_score(y_test, std_clf.predict(X_test))
460     print(f"Standard: Violations = {std_vio:.3f}, Accuracy = {std_acc:.3f}")
461
462     return {
463         "Scenario": "Logic-Guided Architecture",
464         "Algorithm": alg,
465         "Threshold": threshold,

```

```

466         "Violation Without PhIML": std_vio,
467         "Violation With PhIML": total_vio,
468         "Accuracy Without PhIML": std_acc,
469         "Accuracy With PhIML": accuracy
470     }
471
472 # Visualization Functions
473 def create_paired_bar_chart(df,
474                             post_hoc_scenarios=('Document Classification', 'Severity Hierarchy'),
475                             fundamental_scenarios=('Constraint-Aware Loss', 'Logic-Guided Architecture')):
476     """
477     Create paired before/after bar charts per algorithm with two columns (Violation, Accuracy).
478     A light vertical separator is drawn where post-hoc scenarios end and fundamental scenarios begin
479     *within each algorithm row* if both groups are present.
480     """
481     algs = df['Algorithm'].unique().tolist()
482     n = len(algs)
483
484     fig, axes = plt.subplots(n, 2, figsize=(14, 4 * n), sharex='col')
485     if n == 1:
486         axes = np.array([axes])
487
488     without_color = '#1f77b4'
489     with_color = '#ff7f0e'
490     alpha = 0.8
491
492     for i, alg in enumerate(algs):
493         df_alg = df[df['Algorithm'] == alg].copy()
494
495         # order scenarios: post-hoc first, then fundamental
496         order = list(post_hoc_scenarios) + list(fundamental_scenarios)
497         df_alg['__ord__'] = df_alg['Scenario'].apply(lambda s: order.index(s) if s in order else 999)
498         df_alg = df_alg.sort_values('__ord__')
499         scenarios = df_alg['Scenario'].tolist()
500         x = np.arange(len(scenarios))
501         width = 0.35
502
503         ax_v = axes[i, 0]
504         vio_before = df_alg['Violation Without PhIML'].values * 100
505         vio_after = df_alg['Violation With PhIML'].values * 100
506         bars1 = ax_v.bar(x - width/2, vio_before, width, label='Without PhIML',
507                         color=without_color, alpha=alpha)
508         bars2 = ax_v.bar(x + width/2, vio_after, width, label='With PhIML',
509                         color=with_color, alpha=alpha)
510         for bar in bars1 + bars2:
511             h = bar.get_height()
512             ax_v.text(bar.get_x() + bar.get_width()/2., h, f'{h:.1f}%', ha='center', va='bottom', fontsize=9)
513         ax_v.set_ylabel('Violation Rate (%)')
514         ax_v.set_ylim(0, 120)
515         ax_v.set_xticks(x)
516         ax_v.set_xticklabels(scenarios, rotation=35, ha='right')
517         ax_v.tick_params(axis='x', labelsize=9)
518         if i == 0:
519             ax_v.legend(fontsize=9, loc='upper left')
520         ax_v.grid(True, axis='y', alpha=0.3)
521
522         # separator: index of last post-hoc present
523         n_post = sum(s in post_hoc_scenarios for s in scenarios)
524         if 0 < n_post < len(scenarios):
525             ax_v.axvline(n_post - 0.5, color='k', linestyle='--', linewidth=0.5)
526
527         ax_a = axes[i, 1]
528         acc_before = df_alg['Accuracy Without PhIML'].values * 100
529         acc_after = df_alg['Accuracy With PhIML'].values * 100
530         bars3 = ax_a.bar(x - width/2, acc_before, width, color=without_color, alpha=alpha)
531         bars4 = ax_a.bar(x + width/2, acc_after, width, color=with_color, alpha=alpha)
532         for bar in bars3 + bars4:
533             h = bar.get_height()
534             ax_a.text(bar.get_x() + bar.get_width()/2., h, f'{h:.1f}%', ha='center', va='bottom', fontsize=9)
535         ax_a.set_ylabel('Accuracy (%)')
536         ax_a.set_ylim(0, 120)
537         ax_a.set_xticks(x)
538         ax_a.set_xticklabels(scenarios, rotation=35, ha='right')
539         ax_a.tick_params(axis='x', labelsize=9)
540         ax_a.grid(True, axis='y', alpha=0.3)
541
542         ax_v.set_title(alg, fontsize=10, pad=10)
543         ax_a.set_title(alg, fontsize=10, pad=10)

```

```

544
545     if 0 < n_post < len(scenarios):
546         ax_a.axvline(n_post - 0.5, color='k', linestyle='--', linewidth=0.5)
547
548 plt.tight_layout()
549 plt.savefig('phiml_results.png', dpi=900, bbox_inches='tight')
550 plt.show()
551
552 # Main execution
553 if __name__ == "__main__":
554     parser = argparse.ArgumentParser(description="Run PhIML scenarios per algorithm")
555     parser.add_argument(
556         "--algs", "-a",
557         nargs="+",
558         choices=["logistic", "decision_tree", "random_forest", "svm"],
559         default=["random_forest", "svm"],
560         help="Which classifier(s) to use"
561     )
562     args, _ = parser.parse_known_args()
563
564     all_results = []
565     for alg in args.algs:
566         # Post-hoc enforcement scenarios
567         all_results.append(scenario_document_classification(alg)) # Replaced medical
568         all_results.append(scenario_severity_hierarchy(alg))
569
570         # Fundamental PhIML approaches
571         all_results.append(scenario_constraint_aware_loss(alg))
572         all_results.append(scenario_logic_guided(alg))
573
574     # Update visualization categories
575     post_hoc_scenarios = ['Document Classification', 'Severity Hierarchy']
576     fundamental_scenarios = ['Constraint-Aware Loss', 'Logic-Guided Architecture']
577
578     # Summary Table
579     df = pd.DataFrame(all_results)
580     print("\n=== Summary of Results ===")
581     print(tabulate(df, headers='keys', tablefmt='grid', showindex=False))
582
583     # Compare approaches
584     print("\n=== Comparison: Post-hoc vs Fundamental Approaches ===")
585     post_hoc_scenarios = ['Document Classification', 'Severity Hierarchy']
586     fundamental_scenarios = ['Constraint-Aware Loss', 'Logic-Guided Architecture']
587
588     for alg in args.algs:
589         print(f"\n{alg.replace('_', ' ').title()}:")
590         # Post-hoc stats
591         post_hoc_data = df[(df['Algorithm'] == alg) & (df['Scenario'].isin(post_hoc_scenarios))]
592         if not post_hoc_data.empty:
593             post_vio_imp = (post_hoc_data['Violation Without PhIML'] - post_hoc_data['Violation With PhIML']).mean()
594             post_acc_change = (post_hoc_data['Accuracy With PhIML'] - post_hoc_data['Accuracy Without PhIML']).mean()
595             print(f"  Post-hoc: Avg violation reduction = {post_vio_imp:.3f}, Avg accuracy change = {post_acc_change:.3f}")
596
597         # Fundamental stats
598         fund_data = df[(df['Algorithm'] == alg) & (df['Scenario'].isin(fundamental_scenarios))]
599         if not fund_data.empty:
600             fund_vio_imp = (fund_data['Violation Without PhIML'] - fund_data['Violation With PhIML']).mean()
601             fund_acc_change = (fund_data['Accuracy With PhIML'] - fund_data['Accuracy Without PhIML']).mean()
602             print(f"  Fundamental: Avg violation reduction = {fund_vio_imp:.3f}, Avg accuracy change = {fund_acc_change:.3f}")
603
604     # Create Visualizations
605     print("\nGenerating visualizations...")
606     create_paired_bar_chart(df)
607     print("Visualization saved as 'phiml_results.png'")
608

```



```

1 import numpy as np
2 import pandas as pd
3 import argparse
4 import matplotlib.pyplot as plt
5 from sklearn.linear_model import LinearRegression
6 from sklearn.ensemble import RandomForestRegressor
7 from sklearn.model_selection import train_test_split
8 from sklearn.metrics import mean_squared_error
9 import warnings
10 warnings.filterwarnings("ignore")
11
12 # PhIML Functions
13 class CounterfactualConsistencyEnforcer:
14     def __init__(self, threshold=1.0):
15         self.threshold = threshold
16
17     def enforce(self, factual, counterfactual):
18         factual, counterfactual = float(factual), float(counterfactual)
19         delta = counterfactual - factual
20         if abs(delta) > self.threshold:
21             return factual + np.sign(delta) * self.threshold
22         return counterfactual
23
24 def scenario_counterfactual_consistency(alg="random_forest"):
25     np.random.seed(42)
26     n = 1000
27     age = np.random.uniform(20, 80, n)
28     sev = np.random.beta(2, 5, n) * 10
29     com = np.random.poisson(1, n)
30     frailty = 0.01*age + 0.2*sev + 0.1*com + np.random.normal(0, 0.5, n)
31
32     treat = np.zeros(n, dtype=int)
33     treat[frailty < 0] = np.random.choice([1, 2], (frailty < 0).sum(), p=[0.3, 0.7])
34     mid = (frailty >= 0) & (frailty < 2)
35     treat[mid] = np.random.choice([0, 1, 2], mid.sum(), p=[0.3, 0.5, 0.2])
36     treat[frailty >= 2] = np.random.choice([0, 1], (frailty >= 2).sum(), p=[0.7, 0.3])
37
38     te = np.where(frailty < 0, 2.0*treat,
39                  np.where(frailty < 2, 0.5*treat, -0.5*treat))
40     y = 5 - 0.05*age - 0.5*sev - 0.3*com + te + np.random.normal(0, 0.5, n)
41     X = np.column_stack([age, sev, com])
42
43     X_tr, X_te, y_tr, y_te, t_tr, t_te = train_test_split(
44         X, y, treat, test_size=0.3, random_state=42)
45
46     model = (RandomForestRegressor(n_estimators=100, random_state=42)
47             if alg == "random_forest" else LinearRegression())
48     model.fit(np.column_stack([X_tr, t_tr]), y_tr)
49
50     enforcer = CounterfactualConsistencyEnforcer(threshold=1.0)
51
52     pred_without = model.predict(np.column_stack([X_te, t_te]))
53     pred_with = np.empty_like(pred_without)
54
55     vio_without, vio_with = [], []
56
57     for i in range(len(X_te)):
58         preds = {t: model.predict([X_te[i], t])[0] for t in (0, 1, 2)}
59         factual = preds[t_te[i]]
60
61         # Baseline violation
62         vio_without.append(
63             any(abs(preds[t] - factual) > enforcer.threshold for t in (0, 1, 2) if t != t_te[i])
64         )
65
66         # Single-pass counterfactual clipping
67         preds_adj = {t: (factual if t == t_te[i]
68                        else enforcer.enforce(factual, preds[t]))
69                     for t in (0, 1, 2)}
70
71         pred_with[i] = preds_adj[t_te[i]]
72
73         # Post-repair violation
74         vio_with.append(
75             any(abs(preds_adj[t] - preds_adj[t_te[i]]) > enforcer.threshold for t in (0, 1, 2))

```

```

76         if t != t_te[i])
77     )
78
79     return dict(
80         Scenario          = "Counterfactual Consistency",
81         Algorithm          = alg,
82         **{
83             "Violation Without": float(np.mean(vio_without)),
84             "Violation With":    float(np.mean(vio_with)),
85             "MSE Without":       float(mean_squared_error(y_te, pred_without)),
86             "MSE With":         float(mean_squared_error(y_te, pred_with)),
87         }
88     )
89
90
91 def scenario_environment_ensemble(alg="random_forest"):
92     np.random.seed(42)
93     envs_data = {}
94     for e in range(4):
95         X_e = np.column_stack([
96             np.random.normal(40 + 10*e, 10, 200),
97             np.random.exponential(2, 200),
98             np.random.binomial(1, 0.3 + 0.1*e, 200),
99         ])
100         y_e = (0.1*X_e[:, 0] + e*0.05*X_e[:, 1] + e*2*X_e[:, 2]
101             + np.random.normal(0, 1, 200))
102         envs_data[e] = (X_e, y_e)
103
104     train_envs, test_envs = [0, 1], [2, 3]
105
106     def Reg(seed=None):
107         if alg == "random_forest":
108             return RandomForestRegressor(n_estimators=100, random_state=seed)
109         return LinearRegression()
110
111     # ----- baseline pooled -----
112     X_pool = np.vstack([envs_data[e][0] for e in train_envs])
113     y_pool = np.hstack([envs_data[e][1] for e in train_envs])
114     pooled = Reg(seed=0).fit(X_pool, y_pool)
115
116     mse_pool = [mean_squared_error(envs_data[e][1], pooled.predict(envs_data[e][0]))
117         for e in test_envs]
118     vio_pool    = float(np.var(mse_pool))
119     mse_pool_mean = float(np.mean(mse_pool))
120
121     # ----- PhIML : experts + linear meta with numeric env-ID -----
122     experts = [(e, Reg(seed=43+e).fit(*envs_data[e])) for e in train_envs]
123
124     meta_X, meta_y = [], []
125     for e_src in train_envs:
126         Xs_, ys_ = envs_data[e_src]
127         preds_src = np.column_stack([m.predict(Xs_) for _, m in experts])
128         env_feat  = np.full((len(Xs_), 1), e_src) # numeric ID (0 or 1)
129         meta_X.append(np.hstack([preds_src, env_feat]))
130         meta_y.append(ys_)
131     meta_X = np.vstack(meta_X); meta_y = np.hstack(meta_y)
132
133     meta = LinearRegression().fit(meta_X, meta_y)
134
135     mse_meta = []
136     for e_t in test_envs:
137         X_t, y_t = envs_data[e_t]
138         base_preds = np.column_stack([m.predict(X_t) for _, m in experts])
139         env_feat    = np.full((len(X_t), 1), e_t) # unseen IDs 2 or 3
140         y_hat       = meta.predict(np.hstack([base_preds, env_feat]))
141         mse_meta.append(mean_squared_error(y_t, y_hat))
142
143     vio_meta    = float(np.var(mse_meta))
144     mse_meta_mean = float(np.mean(mse_meta))
145
146     return dict(
147         Scenario          = "Environment Ensemble",
148         Algorithm          = alg,
149         **{
150             "Violation Without": vio_pool,
151             "Violation With":    vio_meta,
152             "MSE Without":       mse_pool_mean,
153             "MSE With":         mse_meta_mean,

```

```

154     }
155 )
156
157
158 # Scenario-wise comparison: one row per scenario, two columns (Violation | MSE)
159 def create_scenario_paired_chart(
160     df,
161     order=('Counterfactual Consistency', 'Environment Ensemble')
162 ):
163     """
164     Plot violation-and-MSE bars with 'Without' vs 'With' for every algorithm,
165     but arrange the figure by *scenario* instead of by algorithm.
166
167     Parameters
168     -----
169     df : pd.DataFrame
170         Must have columns
171         ['Scenario', 'Algorithm', 'Violation Without', 'Violation With',
172          'MSE Without', 'MSE With'].
173     order : tuple
174         Display order of scenarios (top to bottom).
175     """
176     import numpy as np
177     import matplotlib.pyplot as plt
178
179     scenarios = [s for s in order if s in df['Scenario'].unique()]
180     algs       = df['Algorithm'].unique().tolist()
181     n_scen     = len(scenarios)
182     width      = 0.35
183     palette    = {'Without': '#1f77b4',
184                  'With'   : '#ff7f0e'}
185
186     fig, axes = plt.subplots(
187         n_scen, 2, figsize=(14, 4 * n_scen), sharex=False
188     )
189     if n_scen == 1:
190         axes = np.array([axes])
191
192     for i, scen in enumerate(scenarios):
193         ax_vio, ax_mse = axes[i]
194
195         df_s = df[df['Scenario'] == scen].copy()
196         x = np.arange(len(algs))
197
198         # --- Violation subplot -----
199         vio_w = df_s['Violation Without'].values
200         vio_p = df_s['Violation With'].values
201         ax_vio.bar(x - width/2, vio_w, width,
202                  label='Without PhIML', color=palette['Without'], alpha=0.85)
203         ax_vio.bar(x + width/2, vio_p, width,
204                  label='With PhIML', color=palette['With'], alpha=0.85)
205         ax_vio.set_title(f"{scen} - Violation metric")
206         ax_vio.set_ylabel('Rate' if scen.startswith('Counterfactual')
207                          else 'Variance')
208         ax_vio.set_xticks(x)
209         ax_vio.set_xticklabels(algs, rotation=15)
210
211         # --- MSE subplot -----
212         mse_w = df_s['MSE Without'].values
213         mse_p = df_s['MSE With'].values
214         ax_mse.bar(x - width/2, mse_w, width,
215                  label='Without PhIML', color=palette['Without'], alpha=0.85)
216         ax_mse.bar(x + width/2, mse_p, width,
217                  label='With PhIML', color=palette['With'], alpha=0.85)
218         ax_mse.set_title(f"{scen} - MSE")
219         ax_mse.set_ylabel('MSE')
220         ax_mse.set_xticks(x)
221         ax_mse.set_xticklabels(algs, rotation=15)
222
223         if i == 0:
224             ax_vio.legend(loc='upper left', frameon=True, edgecolor='black', facecolor='white')
225
226     fig.tight_layout()
227     return fig
228
229 # Main
230 if __name__ == "__main__":
231     parser = argparse.ArgumentParser()

```

```
232     parser.add_argument("-a", "--algs", nargs='+',
233                         choices=["random_forest", "linear"],
234                         default=["random_forest", "linear"])
235     args, _ = parser.parse_known_args()
236
237     records = []
238     for alg in args.algs:
239         records.append(scenario_counterfactual_consistency(alg))
240         records.append(scenario_environment_ensemble(alg))
241
242     df = pd.DataFrame(records)
243     print("\n=== Summary of Results ===")
244     print(df.to_string(index=False))
245
246     fig = create_scenario_paired_chart(df)
247     fig.savefig("my_plot.png", dpi=900)
248
```

```

1 import numpy as np
2 import pandas as pd
3 import matplotlib.pyplot as plt
4 import seaborn as sns
5 from typing import Dict, Tuple, Any
6 from sklearn.ensemble import RandomForestClassifier
7 from sklearn.model_selection import train_test_split
8 from sklearn.metrics import accuracy_score
9 from sklearn.preprocessing import StandardScaler
10 from sklearn.base import BaseEstimator, ClassifierMixin
11 import torch
12 import torch.nn as nn
13 import torch.optim as optim
14
15 # ----- RF (post-hoc) -----
16 class StandardRandomForest:
17     """Standard Random Forest without philosophical considerations."""
18
19     def __init__(self, n_estimators=100, max_depth=10, random_state=42, **rf_kwargs):
20         self.model = RandomForestClassifier(n_estimators=n_estimators, max_depth=max_depth, random_state=random_state, **rf_kwargs)
21         self.name = "Standard Random Forest"
22
23     def fit(self, X_train, y_train):
24         self.model.fit(X_train, y_train)
25
26     def predict(self, X):
27         return self.model.predict(X)
28
29     def predict_proba(self, X):
30         return self.model.predict_proba(X)
31
32 # ----- RawlsianRandomForest (post-hoc) -----
33 class RawlsianRandomForest:
34     """
35     Random Forest enhanced with Rawlsian principles:
36     1. Veil of Ignorance: Equal consideration for all groups
37     2. Difference Principle: Prioritise improving outcomes for worst-off
38     3. Minimax: Maximise minimum group performance
39     """
40
41     # ----- GLOBAL CONSTANTS -----
42     MIN_GROUP_SIZE = 20 # Minimum samples required for a group
43     VALIDATION_SPLIT = 0.20 # Fraction of training data for threshold tuning
44     MIN_ACCURACY_RETENTION = 0.90 # Keep ≥ this fraction of baseline accuracy
45     WORST_OFF_FRACTION = 1/3 # Fraction of groups deemed "worst-off"
46     MAX_WORST_OFF_GROUPS = 5 # Upper bound on # worst-off groups
47     THRESHOLD_STEP = 0.02 # Grid step for threshold search
48     MINIMAX_WEIGHT = 0.70 # Objective weight on minimax accuracy
49     AVG_WEIGHT = 0.30 # Objective weight on average accuracy
50
51     USE_PER_GROUP_THRESHOLD = False
52     # -----
53
54     def __init__(self, n_estimators=100, max_depth=10, random_state=42, **rf_kwargs):
55         self.random_state = random_state
56         self.model = RandomForestClassifier(n_estimators=n_estimators, max_depth=max_depth, random_state=random_state, **rf_kwargs)
57         self.name = "Rawlsian-Enhanced Random Forest"
58         self.group_thresholds = {}
59         self.worst_off_groups = []
60
61     def fit(self, X_train, y_train, X_train_sensitive):
62         """
63         Train with Rawlsian principles using a held-out validation split:
64         1. Split numeric + sensitive into base and validation sets
65         2. Train on base set
66         3. Identify worst-off groups on validation set
67         4. Optimize thresholds on validation set
68         """
69         # 1. Split for threshold tuning (preserving label distribution)
70         X_base, X_val, sens_base, sens_val, y_base, y_val = train_test_split(
71             X_train,
72             X_train_sensitive,
73             y_train,
74             test_size=self.VALIDATION_SPLIT,
75             random_state=self.random_state,

```



```

76         stratify=y_train
77     )
78
79     # 2. Standard training on base set
80     self.model.fit(X_base, y_base)
81
82     # 3. Identify group performance on the validation set
83     val_probs = self.model.predict_proba(X_val)[:, 1]
84     group_performances = self._analyze_group_performance(
85         sens_val,
86         y_val,
87         val_probs,
88         sens_val.columns
89     )
90
91     # Select worst-off groups
92     valid_groups = [(g, p) for g, p in group_performances.items() if p['size'] >= self.MIN_GROUP_SIZE]
93
94     # Handle empty valid_groups scenario
95     if not valid_groups:
96         print(f"\nWARNING: No groups meet minimum size threshold of {self.MIN_GROUP_SIZE}")
97         print("Falling back to standard prediction without group-specific thresholds")
98         self.worst_off_groups = []
99         self.group_thresholds = {}
100     # Store the groups we found for diagnostic purposes
101     print("\nGroups found (with sizes):")
102     for g, p in group_performances.items():
103         print(f"    {g}: size={p['size']}, accuracy={p['accuracy']:.3f}")
104     return
105
106     sorted_groups = sorted(valid_groups, key=lambda x: x[1]['accuracy'])
107     n_worst = max(1, min(int(len(sorted_groups) * self.WORST_OFF_FRACTION), self.MAX_WORST_OFF_GROUPS))
108     self.worst_off_groups = [g for g, _ in sorted_groups[:n_worst]]
109
110     print(f"\nIdentified {len(self.worst_off_groups)} worst-off groups:")
111     for group in self.worst_off_groups:
112         perf = group_performances[group]
113         print(f"    {group}: accuracy={perf['accuracy']:.3f}, size={perf['size']}")
114
115     # 4. Optimize thresholds using the validation set
116     if self.worst_off_groups: # Only optimize if we have groups
117         self._optimize_rawlsian_thresholds(X_val, y_val, sens_val)
118     else:
119         self.group_thresholds = {}
120
121 def _analyze_group_performance(self, X, y, predictions, sensitive_features):
122     """Analyze performance for each demographic group."""
123     performances = {}
124
125     for feature in sensitive_features:
126         unique_values = X[feature].unique()
127         for value in unique_values:
128             mask = X[feature] == value
129             # FIXED: Use class constant
130             if mask.sum() >= self.MIN_GROUP_SIZE:
131                 group_pred = (predictions[mask] > 0.5).astype(int)
132                 group_acc = accuracy_score(y[mask], group_pred)
133
134                 performances[f"{feature}={value}"] = {
135                     'accuracy': group_acc,
136                     'size': mask.sum(),
137                     'positive_rate': group_pred.mean()
138                 }
139
140     return performances
141
142 def _optimize_rawlsian_thresholds(self, X_val, y_val, X_val_sensitive):
143     """
144     Tune decision thresholds.
145
146     - Default (USE_PER_GROUP_THRESHOLD = False) -
147         • Behaviour identical to the original script: one common
148           threshold applied to all worst-off groups.
149
150     - Per-group mode (set flag to True) -
151         • Derives a separate threshold for each worst-off group,
152           while guaranteeing global accuracy ≥ MIN_ACCURACY_RETENTION
153           of the untuned baseline.

```

```

154     """
155     import numpy as np
156     import pandas as pd
157
158     base_probs      = self.model.predict_proba(X_val)[: , 1]
159     overall_acc_baseline= accuracy_score(y_val, (base_probs > 0.5).astype(int))
160     min_acceptable_acc = overall_acc_baseline * self.MIN_ACCURACY_RETENTION
161     thr_grid         = np.arange(0.05, 0.95, self.THRESHOLD_STEP)
162
163     print(f"Base overall accuracy: {overall_acc_baseline:.3f}")
164     print(f"Minimum acceptable accuracy: {min_acceptable_acc:.3f}")
165
166     # Safe type conversion helper
167     def safe_type_convert(val, col):
168         if pd.api.types.is_object_dtype(col):
169             return str(val)
170         elif pd.api.types.is_integer_dtype(col):
171             try:
172                 return int(float(val))
173             except (ValueError, TypeError):
174                 return None
175         elif pd.api.types.is_float_dtype(col):
176             try:
177                 return float(val)
178             except (ValueError, TypeError):
179                 return None
180         return val
181
182     # UNIFORM-THRESHOLD PATH (current behaviour)
183     if not self.USE_PER_GROUP_THRESHOLD:
184         best_score, best_thresholds = -np.inf, {}
185         for thr in thr_grid:
186             test_preds = (base_probs > 0.5).astype(int).copy()
187             current     = {}
188
189             for group in self.worst_off_groups:
190                 feat, val = group.split('=', 1)
191                 if feat not in X_val_sensitive.columns:
192                     continue
193                 col      = X_val_sensitive[feat]
194                 typed_val = safe_type_convert(val, col)
195                 if typed_val is None:
196                     continue
197                 mask      = (col == typed_val).values
198                 if mask.sum() >= self.MIN_GROUP_SIZE:
199                     test_preds[mask] = (base_probs[mask] > thr).astype(int)
200                     current[group]   = {
201                         'threshold': thr,
202                         'accuracy' : accuracy_score(y_val[mask], test_preds[mask])
203                     }
204
205             new_overall = accuracy_score(y_val, test_preds)
206             if new_overall >= min_acceptable_acc and current:
207                 minimax = min(v['accuracy'] for v in current.values())
208                 avg_acc = np.mean([v['accuracy'] for v in current.values()])
209                 score    = self.MINIMAX_WEIGHT*minimax + self.AVG_WEIGHT*avg_acc
210                 if score > best_score:
211                     best_score, best_thresholds = score, current.copy()
212                     print(f" New best thr {thr:.2f}: minimax={minimax:.3f}, avg={avg_acc:.3f}")
213
214             if not best_thresholds:
215                 print("No improvement found; applying default threshold 0.30.")
216                 best_thresholds = {g: {'threshold': 0.30, 'accuracy': np.nan}
217                                     for g in self.worst_off_groups}
218
219             self.group_thresholds = best_thresholds
220         return
221
222     # PER-GROUP PATH (if enabled)
223     per_group_thresholds = {}
224     working_preds        = (base_probs > 0.5).astype(int).copy()
225
226     for group in self.worst_off_groups:
227         feat, val = group.split('=', 1)
228         if feat not in X_val_sensitive.columns:
229             continue
230         col      = X_val_sensitive[feat]
231         typed_val = safe_type_convert(val, col)

```

```

232         if typed_val is None:
233             continue
234         mask = (col == typed_val).values
235
236         if mask.sum() < self.MIN_GROUP_SIZE:
237             continue
238
239         best_thr, best_acc = 0.5, accuracy_score(y_val[mask], working_preds[mask])
240
241         for thr in thr_grid:
242             trial_preds = working_preds.copy()
243             trial_preds[mask] = (base_probs[mask] > thr).astype(int)
244
245             if accuracy_score(y_val, trial_preds) < min_acceptable_acc:
246                 continue
247
248             group_acc = accuracy_score(y_val[mask], trial_preds[mask])
249             if group_acc > best_acc:
250                 best_thr, best_acc = thr, group_acc
251
252         per_group_thresholds[group] = {'threshold': best_thr, 'accuracy': best_acc}
253         working_preds[mask] = (base_probs[mask] > best_thr).astype(int)
254
255         print(f" Chosen thr {best_thr:.2f} for {group}: acc={best_acc:.3f}")
256
257     if not per_group_thresholds:
258         print("Per-group optimisation found no valid improvements; reverting to 0.30.")
259         per_group_thresholds = {g: {'threshold': 0.30, 'accuracy': np.nan}
260                                for g in self.worst_off_groups}
261
262     self.group_thresholds = per_group_thresholds
263
264     def predict_proba(self, X):
265         return self.model.predict_proba(X)
266
267     def predict(self, X, X_sensitive=None):
268         """
269         Return binary predictions with group-specific thresholds when
270         sensitive features are supplied.
271         """
272         import pandas as pd
273
274         base_probs = self.model.predict_proba(X)[:, 1]
275         preds = (base_probs > 0.5).astype(int)
276
277         # Safe type conversion helper
278         def safe_type_convert(val, col):
279             if pd.api.types.is_object_dtype(col):
280                 return str(val)
281             elif pd.api.types.is_integer_dtype(col):
282                 try:
283                     return int(float(val))
284                 except (ValueError, TypeError):
285                     return None
286             elif pd.api.types.is_float_dtype(col):
287                 try:
288                     return float(val)
289                 except (ValueError, TypeError):
290                     return None
291             return val
292
293         if X_sensitive is not None and self.group_thresholds:
294             for group, cfg in self.group_thresholds.items():
295                 # Skip if this group wasn't found during training
296                 if 'threshold' not in cfg:
297                     continue
298
299                 feat, val = group.split('=', 1)
300                 if feat not in X_sensitive.columns:
301                     continue
302
303                 col = X_sensitive[feat]
304
305                 # Apply the FIXED type conversion logic
306                 typed_val = safe_type_convert(val, col)
307                 if typed_val is None:
308                     continue
309

```

```

310         mask = (col == typed_val).values
311
312         # Apply threshold to all members of optimized groups
313         if mask.sum() > 0:
314             preds[mask] = (base_probs[mask] > cfg['threshold']).astype(int)
315
316     return preds
317
318 # ===== NEW IN-TRAINING / NN MODELS
319 # ----- In-training models -----
320
321 class RawlsianNeuralNetwork(BaseEstimator, ClassifierMixin):
322     """In-training Rawls-aware neural net (minimax + variance)."""
323     def __init__(self, hidden_dim=64, n_epochs=100, learning_rate=1e-3,
324                 rawlsian_weight=0.7, min_group_size=20):
325         self.hidden_dim = hidden_dim
326         self.n_epochs = n_epochs
327         self.learning_rate = learning_rate
328         self.rawlsian_weight = rawlsian_weight
329         self.min_group_size = min_group_size
330         self.name = "Rawlsian NN"
331
332     # ----- private helpers -----
333     def _create_network(self, input_dim: int):
334         return nn.Sequential(
335             nn.Linear(input_dim, self.hidden_dim), nn.ReLU(),
336             nn.Dropout(0.2),
337             nn.Linear(self.hidden_dim, self.hidden_dim // 2), nn.ReLU(),
338             nn.Dropout(0.2),
339             nn.Linear(self.hidden_dim // 2, 1), nn.Sigmoid()
340         )
341
342     def _compute_rawlsian_loss(self, outputs, targets, groups, standard_loss):
343         unique = torch.unique(groups)
344         gloss = []
345         for g in unique:
346             mask = groups == g
347             if mask.sum().item() >= self.min_group_size:
348                 gloss.append(standard_loss[mask].mean())
349         if not gloss:
350             # fall back to avg loss
351             return standard_loss.mean(), {}
352         gloss = torch.stack(gloss)
353         worst = gloss.max()
354         equal = gloss.mean()
355         var = gloss.var()
356         raw = 0.5 * worst + 0.3 * equal + 0.2 * var
357         total = self.rawlsian_weight * raw + \
358             (1 - self.rawlsian_weight) * standard_loss.mean()
359         return total, {'worst': worst.item(),
360                       'equal': equal.item(),
361                       'var' : var.item()}
362
363     # ----- scikit-style API -----
364     def fit(self, X_train, y_train, X_sens):
365         torch.manual_seed(42)
366         X = torch.FloatTensor(X_train.values)
367         y = torch.FloatTensor(y_train.values)
368
369         gids = [hash('_'.join(f"{c}={X_sens.iloc[i][c]}"
370                               for c in X_sens.columns)) % 1000
371                 for i in range(len(X_sens))]
372         groups = torch.LongTensor(gids)
373
374         net = self._create_network(X.shape[1])
375         opt = optim.Adam(net.parameters(), lr=self.learning_rate)
376         crit = nn.BCELoss(reduction='none')
377
378         for _ in range(self.n_epochs):
379             net.train()
380             opt.zero_grad()
381             out = net(X).squeeze()
382             sloss = crit(out, y)
383             loss, _ = self._compute_rawlsian_loss(out, y, groups, sloss)
384             loss.backward()
385             opt.step()
386
387         self.network = net
388         return self

```

```

388
389     def predict_proba(self, X):
390         Xt = torch.FloatTensor(X.values)
391         self.network.eval()
392         with torch.no_grad():
393             p = self.network(Xt).squeeze().numpy()
394         return np.vstack([1 - p, p]).T
395
396     def predict(self, X):
397         return (self.predict_proba(X)[:, 1] > 0.5).astype(int)
398
399
400 # ----- Standard NN baseline -----
401 class StandardNeuralNetwork(BaseEstimator, ClassifierMixin):
402     """Vanilla NN (no Rawlsian terms)."""
403     def __init__(self, hidden_dim=64, n_epochs=100, learning_rate=1e-3):
404         self.hidden_dim = hidden_dim
405         self.n_epochs = n_epochs
406         self.learning_rate = learning_rate
407         self.name = "Standard NN"
408
409     def _create_network(self, input_dim: int):
410         return nn.Sequential(
411             nn.Linear(input_dim, self.hidden_dim), nn.ReLU(),
412             nn.Dropout(0.2),
413             nn.Linear(self.hidden_dim, self.hidden_dim // 2), nn.ReLU(),
414             nn.Dropout(0.2),
415             nn.Linear(self.hidden_dim // 2, 1), nn.Sigmoid()
416         )
417
418     def fit(self, X_train, y_train, X_sens=None):
419         X = torch.FloatTensor(X_train.values)
420         y = torch.FloatTensor(y_train.values)
421         net = self._create_network(X.shape[1])
422         opt = optim.Adam(net.parameters(), lr=self.learning_rate)
423         crit= nn.BCELoss()
424         for _ in range(self.n_epochs):
425             net.train(); opt.zero_grad()
426             out = net(X).squeeze()
427             loss = crit(out, y)
428             loss.backward(); opt.step()
429         self.network = net
430         return self
431
432     def predict_proba(self, X):
433         Xt = torch.FloatTensor(X.values)
434         self.network.eval()
435         with torch.no_grad():
436             p = self.network(Xt).squeeze().numpy()
437         return np.vstack([1 - p, p]).T
438
439     def predict(self, X):
440         return (self.predict_proba(X)[:, 1] > 0.5).astype(int)
441
442
443 # ----- Rawlsian Random Forest (in-training) -----
444 class RawlsianRandomForestInTraining(BaseEstimator, ClassifierMixin):
445     """Builds trees with Rawls-aware impurity reduction."""
446     def __init__(self, n_estimators=100, max_depth=10,
447                 min_group_size=20, rawlsian_weight=0.3, random_state=42, **rf_kwargs):
448         self.n_estimators = n_estimators
449         self.max_depth = max_depth
450         self.min_group_size = min_group_size
451         self.rawlsian_weight= rawlsian_weight
452         self.random_state = random_state
453         self.name = "Rawls RF In-Training"
454
455 # ----- helper: gini + Rawls mix -----
456     def _compute_rawlsian_impurity(self, y, groups):
457         if len(np.unique(y)) < 2:
458             return 0.0
459         _, counts = np.unique(y, return_counts=True)
460         p = counts / len(y)
461         gini = 1 - np.sum(p ** 2)
462
463         gg = []
464         for gr in np.unique(groups):
465             mask = groups == gr

```



```

466         if mask.sum() >= self.min_group_size:
467             gy = y[mask]
468             pc = np.bincount(gy, minlength=2) / len(gy)
469             gg.append(1 - np.sum(pc ** 2))
470         raw = (0.7 * max(gg) + 0.3 * np.mean(gg)) if gg else gini
471         return (1 - self.rawlsian_weight) * gini + \
472             self.rawlsian_weight * raw
473
474 # ----- helper: recursive tree -----
475 def _build_tree(self, X, y, groups, depth=0):
476     n = len(y)
477     if (depth >= self.max_depth or
478         n < self.min_group_size * 2 or
479         len(np.unique(y)) < 2):
480         return {'type': 'leaf',
481                 'value': np.bincount(y).argmax()}
482
483     best, best_red = None, -np.inf
484     for f in range(X.shape[1]):
485         idx = np.argsort(X[:, f])
486         Xs, ys, gs = X[idx, f], y[idx], groups[idx]
487         for i in range(1, n):
488             if Xs[i] == Xs[i - 1]:
489                 continue
490             left_y, right_y = ys[:i], ys[i:]
491             left_g, right_g = gs[:i], gs[i:]
492             parent = self._compute_rawlsian_impurity(ys, gs)
493             li = self._compute_rawlsian_impurity(left_y, left_g)
494             ri = self._compute_rawlsian_impurity(right_y, right_g)
495             red = parent - (i/n)*li - ((n-i)/n)*ri
496             if red > best_red:
497                 best_red = red
498                 best = {'feature': f,
499                         'threshold': (Xs[i] + Xs[i-1]) / 2,
500                         'idx': idx,
501                         'i': i}
502     if best is None:
503         return {'type': 'leaf',
504                 'value': np.bincount(y).argmax()}
505
506     left_idx = best['idx'][:best['i']]
507     right_idx = best['idx'][best['i']:]
508     return {
509         'type': 'split',
510         'feature': best['feature'],
511         'threshold': best['threshold'],
512         'left': self._build_tree(X[left_idx], y[left_idx],
513                                   groups[left_idx], depth+1),
514         'right': self._build_tree(X[right_idx], y[right_idx],
515                                   groups[right_idx], depth+1)
516     }
517
518 # ----- scikit-style API -----
519 def fit(self, X_train, y_train, X_sens):
520     X_arr, y_arr = X_train.values, y_train.values
521     gids = [hash('_'.join(f"{c}={X_sens.iloc[i][c]}")
522                      for c in X_sens.columns)) % 1000
523             for i in range(len(X_sens))]
524     groups = np.array(gids)
525     np.random.seed(self.random_state)
526
527     self.trees = []
528     for _ in range(self.n_estimators):
529         idxs = np.random.choice(len(y_arr), len(y_arr), replace=True)
530         self.trees.append(
531             self._build_tree(X_arr[idxs], y_arr[idxs], groups[idxs])
532         )
533     return self
534
535 def _predict_tree(self, tree, X):
536     if tree['type'] == 'leaf':
537         return np.full(len(X), tree['value'])
538     feat, thr = tree['feature'], tree['threshold']
539     mask = X[:, feat] <= thr
540     pred = np.zeros(len(X), dtype=int)
541     if mask.any():
542         pred[mask] = self._predict_tree(tree['left'], X[mask])
543     if (~mask).any():

```

```

544         pred[~mask] = self._predict_tree(tree['right'], X[~mask])
545     return pred
546
547     def predict(self, X):
548         X_arr = X.values
549         preds = np.vstack([self._predict_tree(t, X_arr) for t in self.trees]).T
550         return (preds.mean(axis=1) > 0.5).astype(int)
551
552     def predict_proba(self, X):
553         X_arr = X.values
554         preds = np.vstack([self._predict_tree(t, X_arr) for t in self.trees]).T
555         p = pd.DataFrame(preds).mean(axis=1)
556         return np.vstack([1 - p, p]).T
557
558 # ===== DATA GENERATOR
559 def create_biased_hiring_data(n_samples=2000, random_state=42):
560     np.random.seed(random_state)
561     data = pd.DataFrame({
562         'years_experience': np.random.exponential(5, n_samples),
563         'education_level' : np.random.choice([1, 2, 3, 4], n_samples,
564                                             p=[0.1, 0.3, 0.4, 0.2]),
565         'test_score'      : np.random.normal(70, 15, n_samples),
566         'num_skills'      : np.random.poisson(5, n_samples),
567         'has_internship'  : np.random.binomial(1, 0.4, n_samples),
568         'gender'          : np.random.choice(['Male', 'Female', 'Non-binary'],
569                                             n_samples,
570                                             p=[0.45, 0.45, 0.10]),
571         'ethnicity'       : np.random.choice(['Group_A', 'Group_B',
572                                             'Group_C', 'Group_D'],
573                                             n_samples,
574                                             p=[0.4, 0.3, 0.2, 0.1]),
575         'socioeconomic'   : np.random.choice(['Low', 'Medium', 'High'],
576                                             n_samples,
577                                             p=[0.3, 0.5, 0.2])
578     })
579
580     hire_score = (0.3 * data['years_experience'] +
581                 0.2 * data['education_level'] * 5 +
582                 0.2 * data['test_score'] / 20 +
583                 0.2 * data['num_skills'] +
584                 0.1 * data['has_internship'] * 10)
585
586     hire_score[data['gender'] == 'Female'] -= 2.5
587     hire_score[data['gender'] == 'Non-binary'] -= 3.5
588     hire_score[data['ethnicity'] == 'Group_C'] -= 2.0
589     hire_score[data['ethnicity'] == 'Group_D'] -= 3.0
590     hire_score[data['socioeconomic'] == 'Low'] -= 3.5
591     hire_score[(data['gender'] == 'Female') &
592               (data['socioeconomic'] == 'Low')] -= 1.5
593     hire_score[(data['ethnicity'].isin(['Group_C', 'Group_D'])) &
594               (data['socioeconomic'] == 'Low')] -= 1.5
595
596     threshold = np.percentile(hire_score, 70) # top 30 % hired
597     y = (hire_score > threshold).astype(int)
598     return data, y
599
600 # ===== FAIRNESS EVALUATION (unchanged) =
601 def evaluate_fairness(model, X_test, y_test, X_test_sensitive, model_name):
602     if hasattr(model, 'group_thresholds'):
603         preds = model.predict(X_test, X_test_sensitive)
604     else:
605         preds = model.predict(X_test)
606
607     results = {
608         'model_name'      : model_name,
609         'overall_accuracy' : accuracy_score(y_test, preds),
610         'group_performances': {},
611         'fairness_metrics' : {}
612     }
613
614     for feature in X_test_sensitive.columns:
615         unique_vals = X_test_sensitive[feature].unique()
616         accuracies, pos_rates = [], []
617         for val in unique_vals:
618             mask = X_test_sensitive[feature] == val
619             min_size = getattr(model, 'MIN_GROUP_SIZE', 20)
620             if mask.sum() >= min_size:
621                 g_acc = accuracy_score(y_test[mask], preds[mask])

```

```

622         g_pos = preds[mask].mean()
623         results['group_performances'][f"{feature}={val}"] = {
624             'accuracy'      : g_acc,
625             'positive_rate': g_pos,
626             'size'         : mask.sum()
627         }
628         accuracies.append(g_acc)
629         pos_rates.append(g_pos)
630
631     if len(accuracies) > 1:
632         results['fairness_metrics'][feature] = {
633             'accuracy_disparity' : max(accuracies) - min(accuracies),
634             'worst_group_accuracy' : min(accuracies),
635             'positive_rate_disparity': max(pos_rates) - min(pos_rates)
636         }
637     return results
638
639 # ===== PLOTTING (multi-model overview)
640 def plot_fairness_multi(results_list, sensitive_features):
641     """
642     Show: (i) overall vs worst-group accuracy per model,
643           (ii) accuracy disparity per sensitive feature.
644     """
645     models = [r['model_name'] for r in results_list]
646     n_models = len(models)
647     width = 0.15
648
649     fig = plt.figure(figsize=(14, 6), dpi=900)
650     gs = fig.add_gridspec(1, 2, width_ratios=[1.0, 1.2])
651
652     ax1 = fig.add_subplot(gs[0, 0])
653     overall = [r['overall_accuracy'] for r in results_list]
654     worst = [min(g['accuracy']
655                 for g in r['group_performances'].values())
656              for r in results_list]
657
658     x = np.arange(n_models)
659     ax1.bar(x - width/2, overall, width, label='Overall', alpha=0.8)
660     ax1.bar(x + width/2, worst, width, label='Worst Group', alpha=0.8)
661     ax1.set_xticks(x); ax1.set_xticklabels(models, rotation=15, ha='right')
662     ax1.set_ylabel('Accuracy'); ax1.set_ylim(0, 1)
663     ax1.set_title('Overall vs Worst-Group Accuracy'); ax1.legend()
664
665     ax2 = fig.add_subplot(gs[0, 1])
666     bar_positions = np.arange(len(sensitive_features))
667     for i, res in enumerate(results_list):
668         disp = [res['fairness_metrics']
669                 .get(f, {'accuracy_disparity': 0})['accuracy_disparity']
670                 for f in sensitive_features]
671         ax2.bar(bar_positions + (i - n_models/2)*width*0.9,
672                 disp, width*0.9, label=models[i], alpha=0.8)
673     ax2.set_xticks(bar_positions)
674     ax2.set_xticklabels([f.capitalize() for f in sensitive_features])
675     ax2.set_ylabel('Accuracy Disparity')
676     ax2.set_title('Fairness Gap by Feature')
677     ax2.legend(fontsize=8)
678
679     plt.tight_layout()
680     fig.savefig('fairness_comparison_multi.png', dpi=900)
681     plt.show()
682
683 # ===== NEW: Most-Advantaged Groups Hiring-Rate Plot
684 def plot_hiring_rate_advantaged(results_list, sensitive_cols, top_k=8,
685                                 reference_index=0):
686     """
687     For the 'top_k' most advantaged demographic groups (highest accuracy in
688     the *reference* model, default = Standard RF), show hiring-rate bars for
689     *all* models.
690
691     • results_list - list of fairness-result dicts (same order as plotted
692                     elsewhere: Standard RF, Post-hoc Rawls RF, ... Rawls NN)
693     • sensitive_cols - list of sensitive column names (needed only for nicer
694                       label munging)
695     """
696     import numpy as np
697     import matplotlib.pyplot as plt
698
699     ref = results_list[reference_index]

```

```

700     rows = []
701     for g, perf in ref['group_performances'].items():
702         pos_rates = []
703         for res in results_list:
704             pr = res['group_performances'].get(g, {'positive_rate': np.nan}) \
705                 ['positive_rate']
706             pos_rates.append(pr)
707             rows.append((g, perf['accuracy'], pos_rates))
708
709     rows.sort(key=lambda t: t[1], reverse=True)
710     rows = rows[:top_k]
711
712     n_models = len(results_list)
713     width = 0.75 / n_models
714     x = np.arange(top_k)
715     fig, ax = plt.subplots(figsize=(14, 5), dpi=900)
716
717     for i in range(n_models):
718         ax.bar(x + (i - n_models/2)*width + width/2,
719             [r[2][i] for r in rows],
720             width, alpha=0.85,
721             label=results_list[i]['model_name'])
722
723     def nice(lbl):
724         return lbl.replace('gender=', 'Gen:') \
725             .replace('ethnicity=', 'Eth:') \
726             .replace('socioeconomic=', 'SES:')
727     ax.set_xticks(x)
728     ax.set_xticklabels([nice(r[0]) for r in rows],
729                       rotation=45, ha='right', fontsize=8)
730     ax.set_ylabel('Hiring Rate')
731     ax.set_title('Most Advantaged Groups - Hiring Rate (all models)')
732     ax.axhline(0.3, ls='--', alpha=0.4)
733     ax.legend(fontsize=8, ncol=min(3, n_models))
734     fig.tight_layout()
735     fig.savefig('hiring_rate_advantaged.png', dpi=900)
736     plt.show()
737
738 def plot_hiring_rate_worst_off(results_list, sensitive_cols, bottom_k=8, reference_index=0):
739     """
740     Plot hiring rates for the 'bottom_k' WORST-OFF demographic groups
741     (lowest accuracy in the reference model).
742
743     This is the mirror of plot_hiring_rate_advantaged but for disadvantaged groups.
744     """
745     import numpy as np
746     import matplotlib.pyplot as plt
747
748     ref = results_list[reference_index]
749
750     rows = []
751     for g, perf in ref['group_performances'].items():
752         pos_rates = []
753         for res in results_list:
754             pr = res['group_performances'].get(g, {'positive_rate': np.nan})['positive_rate']
755             pos_rates.append(pr)
756             rows.append((g, perf['accuracy'], pos_rates))
757
758     rows.sort(key=lambda t: t[1])
759     rows = rows[:bottom_k]
760
761     n_models = len(results_list)
762     width = 0.75 / n_models
763     x = np.arange(bottom_k)
764     fig, ax = plt.subplots(figsize=(14, 5), dpi=900)
765
766     for i in range(n_models):
767         ax.bar(x + (i - n_models/2)*width + width/2,
768             [r[2][i] for r in rows],
769             width, alpha=0.85,
770             label=results_list[i]['model_name'])
771
772     def nice(lbl):
773         return lbl.replace('gender=', 'Gen:') \
774             .replace('ethnicity=', 'Eth:') \
775             .replace('socioeconomic=', 'SES:')
776
777     ax.set_xticks(x)

```

```

778     ax.set_xticklabels([nice(r[0]) for r in rows],
779                        rotation=45, ha='right', fontsize=8)
780     ax.set_ylabel('Hiring Rate')
781     ax.set_title('Worst-Off Groups - Hiring Rate (all models)')
782     ax.axhline(0.3, ls='--', alpha=0.4, label='30% baseline')
783     ax.legend(fontsize=8, ncol=min(3, n_models))
784     fig.tight_layout()
785     fig.savefig('hiring_rate_worst_off.png', dpi=900)
786     plt.show()
787
788 def plot_hiring_rates_comparison(results_list, sensitive_cols, k=8, reference_index=0):
789     """
790     Create side-by-side plots comparing hiring rates for advantaged vs worst-off groups
791     with consistent scales and clear comparisons.
792     """
793     import numpy as np
794     import matplotlib.pyplot as plt
795
796     ref = results_list[reference_index]
797
798     all_groups = []
799     for g, perf in ref['group_performances'].items():
800         pos_rates = []
801         for res in results_list:
802             pr = res['group_performances'].get(g, {'positive_rate': np.nan})['positive_rate']
803             pos_rates.append(pr)
804         all_groups.append((g, perf['accuracy'], pos_rates))
805
806     all_groups.sort(key=lambda t: t[1])
807
808     worst_off = all_groups[:k]
809     best_off = all_groups[-k:][::-1]
810
811     fig, (ax1, ax2) = plt.subplots(1, 2, figsize=(16, 5), dpi=900)
812
813     n_models = len(results_list)
814     width = 0.75 / n_models
815
816     def nice(lbl):
817         return lbl.replace('gender=', 'Gen:') \
818                .replace('ethnicity=', 'Eth:') \
819                .replace('socioeconomic=', 'SES:')
820
821     # Plot 1: Worst-off groups
822     x1 = np.arange(k)
823     for i in range(n_models):
824         values = [r[2][i] for r in worst_off]
825         ax1.bar(x1 + (i - n_models/2)*width + width/2,
826                values, width, alpha=0.85,
827                label=results_list[i]['model_name'])
828
829     ax1.set_xticks(x1)
830     ax1.set_xticklabels([nice(r[0]) for r in worst_off],
831                        rotation=45, ha='right', fontsize=8)
832     ax1.set_ylabel('Hiring Rate')
833     ax1.set_title('Worst-Off Groups - Hiring Rate Comparison')
834     ax1.axhline(0.3, ls='--', alpha=0.4)
835     ax1.set_ylim(0, 0.6) # Fixed scale for comparison
836     ax1.legend(fontsize=8, ncol=min(3, n_models))
837
838     # Plot 2: Advantaged groups
839     x2 = np.arange(k)
840     for i in range(n_models):
841         values = [r[2][i] for r in best_off]
842         ax2.bar(x2 + (i - n_models/2)*width + width/2,
843                values, width, alpha=0.85,
844                label=results_list[i]['model_name'])
845
846     ax2.set_xticks(x2)
847     ax2.set_xticklabels([nice(r[0]) for r in best_off],
848                        rotation=45, ha='right', fontsize=8)
849     ax2.set_ylabel('Hiring Rate')
850     ax2.set_title('Most Advantaged Groups - Hiring Rate Comparison')
851     ax2.axhline(0.3, ls='--', alpha=0.4)
852     ax2.set_ylim(0, 0.6) # Same scale as left plot
853     ax2.legend(fontsize=8, ncol=min(3, n_models))
854
855     fig.tight_layout()

```



```

856     fig.savefig('hiring_rates_comparison.png', dpi=900)
857     plt.show()
858
859     return worst_off, best_off
860
861
862 def plot_hiring_rate_changes(results_list, sensitive_cols, k=8, reference_index=0):
863     """
864     Plot the CHANGE in hiring rates compared to Standard RF for worst-off groups.
865     This makes it easier to see which models improve fairness most.
866     """
867     import numpy as np
868     import matplotlib.pyplot as plt
869
870     # baseline reference
871     ref = results_list[reference_index]
872
873     # Get worst-off groups
874     all_groups = []
875     for g, perf in ref['group_performances'].items():
876         all_groups.append((g, perf['accuracy'], perf['positive_rate']))
877     all_groups.sort(key=lambda t: t[1])
878     worst_off = all_groups[:k]
879
880     fig, ax = plt.subplots(figsize=(14, 5), dpi=900)
881
882     colors = plt.rcParams['axes.prop_cycle'].by_key()['color']
883
884     n_models = len(results_list) - 1
885     width = 0.75 / n_models
886     x = np.arange(k)
887
888     for i, res in enumerate(results_list[1:]):
889         changes = []
890         for g, _, ref_rate in worst_off:
891             new_rate = res['group_performances'].get(g, {'positive_rate': ref_rate})['positive_rate']
892             change = (new_rate - ref_rate) * 100
893             changes.append(change)
894
895         bars = ax.bar(
896             x + (i - n_models/2) * width + width/2,
897             changes,
898             width,
899             alpha=0.85,
900             label=res['model_name'],
901             color=colors[i+1]
902         )
903
904         for bar, val in zip(bars, changes):
905             if val != 0:
906                 ax.text(
907                     bar.get_x() + bar.get_width()/2,
908                     bar.get_height() + 0.3,
909                     f'{val:+.1f}',
910                     ha='center',
911                     va='bottom',
912                     fontsize=7
913                 )
914
915     def nice(lbl):
916         return lbl.replace('gender=', 'Gen:') \
917             .replace('ethnicity=', 'Eth:') \
918             .replace('socioeconomic=', 'SES:')
919
920     ax.set_xticks(x)
921     ax.set_xticklabels([nice(g) for g, _, _ in worst_off],
922                        rotation=45, ha='right', fontsize=8)
923     ax.set_ylabel('Change in Hiring Rate (percentage points)')
924     ax.set_title('Hiring Rate Changes for Worst-Off Groups (vs Standard RF)')
925     ax.axhline(0, ls='--', alpha=0.4)
926     ax.legend(fontsize=8, ncol=min(3, n_models))
927
928     fig.tight_layout()
929     fig.savefig('hiring_rate_changes.png', dpi=900)
930     plt.show()
931
932
933 def generate_summary_table(results_list, worst_off, best_off):

```

```

934     """Generate a summary table of key metrics"""
935     print("\n" + "="*100)
936     print("HIRING RATE ANALYSIS SUMMARY")
937     print("="*100)
938
939     # Overall metrics
940     print("\n1. OVERALL HIRING RATES BY MODEL")
941     print("-" * 60)
942     print(f"{'Model':<25} {'Overall HR':>15} {'Worst-Off Avg':>15} {'Best-Off Avg':>15}")
943     print("-" * 60)
944
945     for res in results_list:
946         all_hr = [p['positive_rate'] for p in res['group_performances'].values()]
947         overall_hr = np.mean(all_hr)
948
949         worst_hr = []
950         best_hr = []
951
952         for g, _, _ in worst_off:
953             if g in res['group_performances']:
954                 worst_hr.append(res['group_performances'][g]['positive_rate'])
955
956         for g, _, _ in best_off:
957             if g in res['group_performances']:
958                 best_hr.append(res['group_performances'][g]['positive_rate'])
959
960         worst_avg = np.mean(worst_hr) if worst_hr else 0
961         best_avg = np.mean(best_hr) if best_hr else 0
962
963         print(f"{res['model_name']:<25} {overall_hr:>15.3f} {worst_avg:>15.3f} {best_avg:>15.3f}")
964
965     print("\n2. HIRING RATE IMPROVEMENTS (vs Standard RF)")
966     print("-" * 60)
967     print(f"{'Model':<25} {'Worst-Off Improvement':>25} {'Equity Gap Reduction':>25}")
968     print("-" * 60)
969
970     std_worst = []
971     std_best = []
972     for g, _, _ in worst_off:
973         if g in results_list[0]['group_performances']:
974             std_worst.append(results_list[0]['group_performances'][g]['positive_rate'])
975     for g, _, _ in best_off:
976         if g in results_list[0]['group_performances']:
977             std_best.append(results_list[0]['group_performances'][g]['positive_rate'])
978
979     std_worst_avg = np.mean(std_worst)
980     std_best_avg = np.mean(std_best)
981     std_gap = std_best_avg - std_worst_avg
982
983     for res in results_list[1:]:
984         worst_hr = []
985         best_hr = []
986
987         for g, _, _ in worst_off:
988             if g in res['group_performances']:
989                 worst_hr.append(res['group_performances'][g]['positive_rate'])
990
991         for g, _, _ in best_off:
992             if g in res['group_performances']:
993                 best_hr.append(res['group_performances'][g]['positive_rate'])
994
995         new_worst_avg = np.mean(worst_hr) if worst_hr else 0
996         new_best_avg = np.mean(best_hr) if best_hr else 0
997         new_gap = new_best_avg - new_worst_avg
998
999         worst_improvement = ((new_worst_avg - std_worst_avg) / std_worst_avg) * 100
1000        gap_reduction = ((std_gap - new_gap) / std_gap) * 100
1001
1002        print(f"{res['model_name']:<25} {worst_improvement:>24.1f}% {gap_reduction:>24.1f}%")
1003
1004    print("\n3. INTERPRETATION GUIDE")
1005    print("-" * 60)
1006    print("• Hiring Rate (HR): Proportion of group members predicted as hired (0-1 scale)")
1007    print("• Baseline: ~30% overall hiring rate (by design)")
1008    print("• For Worst-Off Groups: HIGHER hiring rates = BETTER (more opportunities)")
1009    print("• For Best-Off Groups: Already high; question is whether to reduce for equity")
1010    print("• Equity Gap: Difference between best-off and worst-off average hiring rates")
1011    print("• Gap Reduction: How much the model reduces the equity gap vs Standard RF")

```

```

1012
1013 def print_detailed_results(results_list, top_k=5, bottom_k=5):
1014     """
1015     Print detailed numerical results from all analyses.
1016     """
1017     print("\n" + "="*70)
1018     print("DETAILED NUMERICAL RESULTS")
1019     print("="*70)
1020
1021     # Overall metrics
1022     print("\n1. OVERALL PERFORMANCE METRICS")
1023     print("-" * 50)
1024     print(f"{'Model':<25} {'Overall Acc':>12} {'Worst Acc':>12} {'Avg Disparity':>15}")
1025     print("-" * 50)
1026
1027     for r in results_list:
1028         overall = r['overall_accuracy']
1029         worst = min(v['accuracy'] for v in r['group_performances'].values())
1030
1031         # Calculate average disparity across all features
1032         disparities = []
1033         for feat, metrics in r['fairness_metrics'].items():
1034             disparities.append(metrics['accuracy_disparity'])
1035         avg_disp = np.mean(disparities) if disparities else 0
1036
1037         print(f"{r['model_name']:<25} {overall:>12.3f} {worst:>12.3f} {avg_disp:>15.3f}")
1038
1039     # Fairness gaps by feature
1040     print("\n2. FAIRNESS GAPS BY FEATURE")
1041     print("-" * 50)
1042
1043     features = list(results_list[0]['fairness_metrics'].keys())
1044     print(f"{'Model':<25}", end="")
1045     for feat in features:
1046         print(f"{feat:>15}", end="")
1047     print()
1048     print("-" * 50)
1049
1050     for r in results_list:
1051         print(f"{r['model_name']:<25}", end="")
1052         for feat in features:
1053             disp = r['fairness_metrics'][feat]['accuracy_disparity']
1054             print(f"{disp:>15.3f}", end="")
1055         print()
1056
1057     # Top performing groups
1058     print(f"\n3. TOP {top_k} BEST-PERFORMING GROUPS (by Standard RF accuracy)")
1059     print("-" * 50)
1060
1061     ref = results_list[0] # Standard RF
1062     groups = [(g, p['accuracy'], p['positive_rate'])
1063               for g, p in ref['group_performances'].items()]
1064     groups.sort(key=lambda x: x[1], reverse=True)
1065
1066     print(f"{'Group':<30} {'Accuracy':>10} {'Hiring Rate':>12}")
1067     print("-" * 50)
1068     for g, acc, hr in groups[:top_k]:
1069         print(f"{g:<30} {acc:>10.3f} {hr:>12.3f}")
1070
1071     # Worst performing groups
1072     print(f"\n4. BOTTOM {bottom_k} WORST-PERFORMING GROUPS (by Standard RF accuracy)")
1073     print("-" * 50)
1074
1075     groups.sort(key=lambda x: x[1]) # ascending
1076     print(f"{'Group':<30} {'Accuracy':>10} {'Hiring Rate':>12}")
1077     print("-" * 50)
1078     for g, acc, hr in groups[:bottom_k]:
1079         print(f"{g:<30} {acc:>10.3f} {hr:>12.3f}")
1080
1081     # Hiring rate changes for worst-off groups
1082     print(f"\n5. HIRING RATE CHANGES FOR WORST-OFF GROUPS")
1083     print("-" * 50)
1084
1085     worst_groups = [g[0] for g in groups[:bottom_k]]
1086
1087     print(f"{'Group':<30}", end="")
1088     for r in results_list:
1089         print(f"{r['model_name']:>20}", end="")

```

```

1090     print()
1091     print("-" * 50)
1092
1093     for g in worst_groups:
1094         print(f"{g:<30}", end="")
1095         for r in results_list:
1096             hr = r['group_performances'].get(g, {'positive_rate': 0})['positive_rate']
1097             print(f"{hr:>20.3f}", end="")
1098         print()
1099
1100     # Calculate improvement ratios
1101     print(f"\n6. HIRING RATE IMPROVEMENT RATIOS (vs Standard RF)")
1102     print("-" * 50)
1103
1104     std_rates = results_list[0]['group_performances']
1105
1106     for i, r in enumerate(results_list[1:], 1): # Skip Standard RF
1107         print(f"\n{i}{r['model_name']}:")
1108         improvements = []
1109
1110         for g in worst_groups:
1111             if g in std_rates and g in r['group_performances']:
1112                 std_hr = std_rates[g]['positive_rate']
1113                 new_hr = r['group_performances'][g]['positive_rate']
1114                 if std_hr > 0:
1115                     ratio = new_hr / std_hr
1116                     improvements.append((g, std_hr, new_hr, ratio))
1117
1118         improvements.sort(key=lambda x: x[3], reverse=True)
1119
1120         print(f"{'Group':<30} {'Std HR':>10} {'New HR':>10} {'Ratio':>10}")
1121         for g, std_hr, new_hr, ratio in improvements[:5]: # Top 5 improvements
1122             print(f"{g:<30} {std_hr:>10.3f} {new_hr:>10.3f} {ratio:>10.2f}x")
1123
1124 # ===== MAIN DEMO
1125 def main():
1126     print("\n=== PHIML Benchmark: RF & NN (Standard vs Rawlsian) ===\n")
1127
1128     # ----- data -----
1129     data, y = create_biased_hiring_data(n_samples=1500)
1130     feature_cols = ['years_experience', 'education_level', 'test_score',
1131                    'num_skills', 'has_internship']
1132     sensitive_cols = ['gender', 'ethnicity', 'socioeconomic']
1133
1134     X_train, X_test, y_train, y_test = train_test_split(
1135         data, y, test_size=0.30, random_state=42, stratify=y)
1136
1137     Xtr_num = X_train[feature_cols]
1138     Xts_num = X_test[feature_cols]
1139     Xtr_sens = X_train[sensitive_cols]
1140     Xts_sens = X_test[sensitive_cols]
1141
1142     # ----- models -----
1143     print("Training Standard Random Forest ...")
1144     std_rf = StandardRandomForest()
1145     std_rf.fit(Xtr_num, y_train)
1146
1147     print("Training Post-hoc Rawlsian Random Forest ...")
1148     rawls_post = RawlsianRandomForest()
1149     rawls_post.fit(Xtr_num, y_train, Xtr_sens)
1150
1151     print("Training Rawlsian RF (in-training) ...")
1152     rawls_rf_in = RawlsianRandomForestInTraining()
1153     rawls_rf_in.fit(Xtr_num, y_train, Xtr_sens)
1154
1155     print("Training Standard Neural Network ...")
1156     std_nn = StandardNeuralNetwork()
1157     std_nn.fit(Xtr_num, y_train)
1158
1159     print("Training Rawlsian Neural Network ...")
1160     rawls_nn = RawlsianNeuralNetwork()
1161     rawls_nn.fit(Xtr_num, y_train, Xtr_sens)
1162
1163     # ----- evaluation -----
1164     results = []
1165     results.append(evaluate_fairness(std_rf, Xts_num, y_test, Xts_sens,
1166                                     "Standard RF"))
1167     results.append(evaluate_fairness(rawls_post, Xts_num, y_test, Xts_sens,

```

```

1168         "Post-hoc Rawls RF"))
1169     results.append(evaluate_fairness(rawls_rf_in, Xts_num, y_test, Xts_sens,
1170         "Rawls RF In-Training"))
1171     results.append(evaluate_fairness(std_nn, Xts_num, y_test, Xts_sens,
1172         "Standard NN"))
1173     results.append(evaluate_fairness(rawls_nn, Xts_num, y_test, Xts_sens,
1174         "Rawlsian NN"))
1175
1176     # ----- tabular summary -----
1177     print("\n=== RESULTS SUMMARY ===")
1178     print(f"{'Model':<25} {'Overall':>8} {'Worst':>8} {'Disp':>8}")
1179     for r in results:
1180         overall = r['overall_accuracy']
1181         worst = min(v['accuracy'] for v in r['group_performances'].values())
1182         disp = max(v['accuracy'] for v in r['group_performances'].values()) - worst
1183         print(f"{r['model_name']:<25} {overall:>8.3f} {worst:>8.3f} {disp:>8.3f}")
1184
1185     # ----- plot -----
1186     print("\nGenerating multi-model fairness plot ...")
1187     plot_fairness_multi(results, sensitive_cols)
1188
1189     print("Generating Most-Advantaged-Groups hiring-rate plot ...")
1190     plot_hiring_rate_advantaged(results, sensitive_cols, top_k=8,
1191         reference_index=0) # 0 = Standard RF
1192
1193     print("\nDone. Figures saved as "
1194         "'fairness_comparison_multi.png' and "
1195         "'hiring_rate_advantaged.png'.")
1196
1197     print("Generating Worst-Off Groups hiring-rate plot...")
1198     plot_hiring_rate_worst_off(results, sensitive_cols, bottom_k=8, reference_index=0)
1199
1200     print("\nPrinting detailed numerical results...")
1201     print_detailed_results(results, top_k=8, bottom_k=8)
1202
1203     print("\nGenerating enhanced comparison plots...")
1204     worst_off, best_off = plot_hiring_rates_comparison(results, sensitive_cols, k=8)
1205
1206     print("\nGenerating hiring rate changes plot...")
1207     plot_hiring_rate_changes(results, sensitive_cols, k=8)
1208
1209     print("\nGenerating summary analysis...")
1210     generate_summary_table(results, worst_off, best_off)
1211
1212     print("\nDone. Figures saved as:")
1213     print(" - 'fairness_comparison_multi.png' (overall fairness metrics)")
1214     print(" - 'hiring_rates_comparison.png' (side-by-side worst vs best groups)")
1215     print(" - 'hiring_rate_changes.png' (improvements for worst-off groups)")
1216
1217     # -----
1218     if __name__ == "__main__":
1219         main()

```