Responsible AI - Advanced Handout

Machine Learning for Smarter Innovation

1 Responsible AI - Advanced Handout

Target Audience: Data scientists and ML engineers **Duration**: 90 minutes reading **Level**: Advanced (mathematical foundations and production systems)

1.1 Mathematical Foundations of Fairness

1.1.1 Probabilistic Framework

Let X be features, A be protected attribute, Y be true label, and D be model decision. Fairness constraints are probabilistic conditions:

Demographic Parity: P(D = 1 | A = a) = P(D = 1 | A = b) for all groups a, b

Equal Opportunity: P(D = 1 | Y = 1, A = a) = P(D = 1 | Y = 1, A = b)

Equivalently: True Positive Rate equal across groups.

Equalized Odds: P(D = 1 | Y = y, A = a) = P(D = 1 | Y = y, A = b) for all y

Both TPR and FPR equal across groups.

Calibration: P(Y = 1 | S(X) = s, A = a) = s for all groups a

A score of s means probability s of positive outcome, regardless of group.

1.1.2 Relaxed Fairness Metrics

In practice, perfect equality is impossible. Use epsilon-relaxed versions:

epsilon-Demographic Parity: |P(D = 1 | A = a) - P(D = 1 | A = b)| <= epsilon

The 80% rule from disparate impact law corresponds to: P(D = 1 | A = a) / P(D = 1 | A = b) >= 0.8

Disparate Impact Ratio: DIR = min_a,b [P(D = 1 | A = a) / P(D = 1 | A = b)]

Values below 0.8 may indicate illegal discrimination.

1.2 Impossibility Theorems

1.2.1 Chouldechova's Theorem (2017)

If base rates differ across groups (P(Y = 1 | A = a) != P(Y = 1 | A = b)), then it is impossible to simultaneously achieve: 1. Calibration (equal PPV across groups) 2. Equal False Positive Rates 3. Equal False Negative Rates

Proof sketch: PPV = TP / (TP + FP) = (TPR * Prevalence) / (TPR * Prevalence + FPR * (1 - Prevalence))

If PPV_a = PPV_b but Prevalence_a != Prevalence_b, then the relationship between TPR and FPR must differ by group. Equal TPR implies unequal FPR, and vice versa.

1.2.2 Kleinberg-Mullainathan-Raghavan (2016)

Cannot simultaneously achieve: 1. Calibration 2. Balance for positive class (equal TPR) 3. Balance for negative class (equal FPR)

Exception: Perfect prediction or equal base rates.

Implication: Must choose which fairness notion to prioritize. Choice depends on context and stakeholder values.

1.2.3 Practical Consequence

When base rates differ, optimization involves tradeoffs. Consider a hiring model where: - Group A: 60% qualified - Group B: 40% qualified

Achieving equal TPR requires accepting more unqualified candidates from Group B (higher FPR for B) or rejecting more qualified candidates from Group A (lower TPR for A).

1.3 Individual Fairness

1.3.1 Lipschitz Condition

Individual fairness requires similar individuals receive similar treatment:

d_Y(M(x_i), M(x_j)) <= L * d_X(x_i, x_j)

where d_X is distance in feature space, d_Y is distance in outcome space, and L is the Lipschitz constant.

Challenge: Defining appropriate distance metric d_X requires domain knowledge. Features may have different relevance to "similarity."

1.3.2 Metric Learning for Fairness

Learn a fair metric that captures legitimate similarity:

d_X(x_i, x_j) = sqrt((x_i - x_j)ˆT M (x_i - x_j))

where M is positive semi-definite matrix learned from data or expert input.

Constraints: - M should not weight protected attributes - M should reflect task-relevant similarity

---

1.4 Causal Fairness

1.4.1 Counterfactual Fairness

A decision is counterfactually fair if:

P(Y_{A<-a}(U) = y | X = x, A = a) = P(Y_{A<-a'}(U) = y | X = x, A = a)

where Y_{A<-a}(U) is the outcome in a counterfactual world where A is set to a.

Intuition: Changing only the protected attribute should not change the prediction.

1.4.2 Path-Specific Effects

Distinguish between: - **Direct discrimination:** A affects Y directly - **Indirect discrimination:** A affects Y through mediators - **Proxy discrimination:** A affects X which affects Y

Causal graphs formalize these distinctions. A model may be unfair if it uses paths from A to Y that society deems inappropriate.

1.4.3 Structural Causal Model

Define causal relationships:

X = f_X(A, U_X) Y = f_Y(A, X, U_Y)

where U are unobserved factors. Counterfactual fairness requires Y not depend on A after controlling for U.

1.5 Interpretability Methods

1.5.1 SHAP (SHapley Additive exPlanations)

SHAP values decompose predictions into feature contributions based on coalitional game theory.

Shapley value for feature i: phi_i = sum_{S subset F {i}} [|S|! (|F| - |S| - 1)! / |F|!] * [f(S union {i}) - f(S)]

where F is feature set, S is a subset without feature i.

Properties: - Efficiency: sum_i phi_i = f(x) - E[f(x)] - Symmetry: Equal features get equal attribution - Linearity: Additive for linear models - Null: Zero contribution for unused features

SHAP for Fairness:

```
import shap

explainer = shap.Explainer(model, X_train)
shap_values = explainer(X_test)

# Check if protected attribute has high importance
feature_importance = np.abs(shap_values.values).mean(axis=0)
protected_idx = feature_names.index('gender')
print(f"Protected attribute importance: {feature_importance[protected_idx]}")

# Group-wise SHAP analysis
group_a_shap = shap_values[X_test['gender'] == 0].values.mean(axis=0)
group_b_shap = shap_values[X_test['gender'] == 1].values.mean(axis=0)
```

1.5.2 LIME (Local Interpretable Model-agnostic Explanations)

Approximate complex model locally with interpretable model:

xi(x) = argmin_{g in G} L(f, g, pi_x) + Omega(g)

where G is class of interpretable models (e.g., linear), pi_x weights samples by proximity to x, and Omega measures complexity.

1.6 Implementation

1.6.1 Setup

```
import numpy as np
import pandas as pd
from sklearn.model_selection import train_test_split
from sklearn.linear_model import LogisticRegression
from sklearn.ensemble import RandomForestClassifier
from sklearn.metrics import accuracy_score, confusion_matrix
import warnings
warnings.filterwarnings('ignore')
```

1.6.2 Fairness Metrics Calculator

```
class FairnessMetrics:
    """Comprehensive fairness metrics calculator."""

    def __init__(self, y_true, y_pred, y_prob, sensitive_features):
        self.y_true = np.array(y_true)
        self.y_pred = np.array(y_pred)
        self.y_prob = np.array(y_prob) if y_prob is not None else None
        self.sensitive = np.array(sensitive_features)
        self.groups = np.unique(self.sensitive)

    def demographic_parity_difference(self):
        """Difference in positive prediction rates."""
        rates = []
        for group in self.groups:
            mask = self.sensitive == group
            rate = self.y_pred[mask].mean()
            rates.append(rate)
        return max(rates) - min(rates)

    def demographic_parity_ratio(self):
        """Ratio of positive prediction rates (disparate impact)."""
        rates = []
        for group in self.groups:
            mask = self.sensitive == group
            rate = self.y_pred[mask].mean()
            rates.append(rate)
        return min(rates) / max(rates) if max(rates) > 0 else 0

    def equal_opportunity_difference(self):
        """Difference in true positive rates."""
        tprs = []
        for group in self.groups:
            mask = (self.sensitive == group) & (self.y_true == 1)
            if mask.sum() > 0:
                tpr = self.y_pred[mask].mean()
                tprs.append(tpr)
        return max(tprs) - min(tprs) if len(tprs) >= 2 else 0

    def equalized_odds_difference(self):
        """Maximum of TPR and FPR differences."""
        tprs, fprs = [], []
        for group in self.groups:
            # TPR
            mask_pos = (self.sensitive == group) & (self.y_true == 1)
            if mask_pos.sum() > 0:
```

```
                tprs.append(self.y_pred[mask_pos].mean())

            # FPR
            mask_neg = (self.sensitive == group) & (self.y_true == 0)
            if mask_neg.sum() > 0:
                fprs.append(self.y_pred[mask_neg].mean())

        tpr_diff = max(tprs) - min(tprs) if len(tprs) >= 2 else 0
        fpr_diff = max(fprs) - min(fprs) if len(fprs) >= 2 else 0
        return max(tpr_diff, fpr_diff)

    def calibration_difference(self, n_bins=10):
        """Difference in calibration error across groups."""
        if self.y_prob is None:
            return None

        from sklearn.calibration import calibration_curve

        errors = []
        for group in self.groups:
            mask = self.sensitive == group
            if mask.sum() > n_bins:
                prob_true, prob_pred = calibration_curve(
                    self.y_true[mask], self.y_prob[mask],
                    n_bins=n_bins, strategy='uniform'
                )
                error = np.mean(np.abs(prob_true - prob_pred))
                errors.append(error)

        return max(errors) - min(errors) if len(errors) >= 2 else 0

    def get_all_metrics(self):
        """Return all fairness metrics."""
        return {
            'demographic_parity_difference': self.
    demographic_parity_difference(),
            'demographic_parity_ratio': self.demographic_parity_ratio(),
            'equal_opportunity_difference': self.equal_opportunity_difference
    (),
            'equalized_odds_difference': self.equalized_odds_difference(),
            'calibration_difference': self.calibration_difference()
        }
```

1.6.3 Bias Mitigation Techniques

```
class BiasMinigator:
    """Pre-processing, in-processing, and post-processing mitigation."""

    @staticmethod
    def reweight_samples(X, y, sensitive):
        """Pre-processing: Compute sample weights for balanced training."""
        groups = np.unique(sensitive)
        labels = np.unique(y)

        weights = np.ones(len(y))
        total = len(y)

        for group in groups:
            for label in labels:
                mask = (sensitive == group) & (y == label)
                expected = total / (len(groups) * len(labels))
```

```
                actual = mask.sum()
                if actual > 0:
                    weights[mask] = expected / actual

        return weights / weights.sum() * len(weights)

    @staticmethod
    def threshold_optimizer(y_true, y_prob, sensitive, objective='
    equalized_odds'):
        """Post-processing: Find optimal thresholds per group."""
        groups = np.unique(sensitive)
        thresholds = {}

        for group in groups:
            mask = sensitive == group
            best_threshold = 0.5
            best_score = float('inf')

            for thresh in np.linspace(0.1, 0.9, 81):
                y_pred = (y_prob >= thresh).astype(int)

                if objective == 'equalized_odds':
                    # Compute TPR and FPR at this threshold
                    pos_mask = (sensitive == group) & (y_true == 1)
                    neg_mask = (sensitive == group) & (y_true == 0)

                    tpr = y_pred[pos_mask].mean() if pos_mask.sum() > 0 else 0
                    fpr = y_pred[neg_mask].mean() if neg_mask.sum() > 0 else 0

                    # Minimize deviation from target (0.5, 0.5)
                    score = abs(tpr - 0.5) + abs(fpr - 0.5)

                    if score < best_score:
                        best_score = score
                        best_threshold = thresh

            thresholds[group] = best_threshold

        return thresholds

    @staticmethod
    def apply_thresholds(y_prob, sensitive, thresholds):
        """Apply group-specific thresholds."""
        y_pred = np.zeros(len(y_prob), dtype=int)
        for group, thresh in thresholds.items():
            mask = sensitive == group
            y_pred[mask] = (y_prob[mask] >= thresh).astype(int)
        return y_pred
```

1.6.4 Adversarial Debiasing

```
import torch
import torch.nn as nn
import torch.optim as optim

class AdversarialDebiasingModel(nn.Module):
    """Neural network with adversarial debiasing."""

    def __init__(self, input_dim, hidden_dim=64):
        super().__init__()
```

```
        # Main predictor
        self.predictor = nn.Sequential(
            nn.Linear(input_dim, hidden_dim),
            nn.ReLU(),
            nn.Dropout(0.3),
            nn.Linear(hidden_dim, hidden_dim // 2),
            nn.ReLU(),
            nn.Linear(hidden_dim // 2, 1),
            nn.Sigmoid()
        )

        # Adversary (predicts protected attribute from predictor output)
        self.adversary = nn.Sequential(
            nn.Linear(1, 32),
            nn.ReLU(),
            nn.Linear(32, 1),
            nn.Sigmoid()
        )

    def forward(self, x):
        return self.predictor(x)

    def adversary_forward(self, pred):
        return self.adversary(pred)


class AdversarialTrainer:
    """Training loop for adversarial debiasing."""

    def __init__(self, model, lambda_adv=1.0, lr=0.001):
        self.model = model
        self.lambda_adv = lambda_adv

        # Separate optimizers
        self.pred_optimizer = optim.Adam(model.predictor.parameters(), lr=lr)
        self.adv_optimizer = optim.Adam(model.adversary.parameters(), lr=lr)

        self.criterion = nn.BCELoss()

    def train_epoch(self, X, y, a, batch_size=32):
        """Train one epoch."""
        n_samples = len(X)
        indices = np.random.permutation(n_samples)

        total_pred_loss = 0
        total_adv_loss = 0

        for i in range(0, n_samples, batch_size):
            batch_idx = indices[i:i+batch_size]
            x_batch = torch.FloatTensor(X[batch_idx])
            y_batch = torch.FloatTensor(y[batch_idx]).unsqueeze(1)
            a_batch = torch.FloatTensor(a[batch_idx]).unsqueeze(1)

            # Train adversary
            self.adv_optimizer.zero_grad()
            with torch.no_grad():
                pred = self.model(x_batch)
            adv_pred = self.model.adversary_forward(pred)
            adv_loss = self.criterion(adv_pred, a_batch)
            adv_loss.backward()
            self.adv_optimizer.step()
```

```
            # Train predictor (minimize prediction loss, maximize adversary
    loss)
            self.pred_optimizer.zero_grad()
            pred = self.model(x_batch)
            pred_loss = self.criterion(pred, y_batch)
            adv_pred = self.model.adversary_forward(pred)
            adv_loss = self.criterion(adv_pred, a_batch)

            combined_loss = pred_loss - self.lambda_adv * adv_loss
            combined_loss.backward()
            self.pred_optimizer.step()

            total_pred_loss += pred_loss.item()
            total_adv_loss += adv_loss.item()

        return total_pred_loss, total_adv_loss
```

1.6.5 Production Fairness Monitor

```
class FairnessMonitor:
    """Monitor fairness metrics in production."""

    def __init__(self, thresholds=None):
        self.thresholds = thresholds or {
            'demographic_parity_difference': 0.1,
            'equal_opportunity_difference': 0.1,
            'equalized_odds_difference': 0.15
        }
        self.history = []
        self.alerts = []

    def check(self, y_true, y_pred, y_prob, sensitive, timestamp=None):
        """Check fairness and log results."""
        import datetime
        timestamp = timestamp or datetime.datetime.now()

        metrics = FairnessMetrics(y_true, y_pred, y_prob, sensitive)
        results = metrics.get_all_metrics()
        results['timestamp'] = timestamp

        # Check thresholds
        for metric, threshold in self.thresholds.items():
            if results.get(metric, 0) > threshold:
                self.alerts.append({
                    'timestamp': timestamp,
                    'metric': metric,
                    'value': results[metric],
                    'threshold': threshold
                })

        self.history.append(results)
        return results

    def get_trend(self, metric, window=10):
        """Get recent trend for a metric."""
        if len(self.history) < 2:
            return None

        recent = [h[metric] for h in self.history[-window:] if metric in h]
        if len(recent) < 2:
            return None
```

```
        # Simple linear regression for trend
        x = np.arange(len(recent))
        slope = np.polyfit(x, recent, 1)[0]
        return slope

    def report(self):
        """Generate fairness report."""
        if not self.history:
            return "No data collected yet."

        latest = self.history[-1]
        report_lines = ["Fairness Report", "=" * 50]

        for metric, value in latest.items():
            if metric != 'timestamp':
                threshold = self.thresholds.get(metric, 'N/A')
                status = "PASS" if value <= threshold else "FAIL"
                report_lines.append(f"{metric}: {value:.4f} (threshold: {
threshold}) [{status}]")

        if self.alerts:
            report_lines.append("\nRecent Alerts:")
            for alert in self.alerts[-5:]:
                report_lines.append(f"  - {alert['metric']}: {alert['value
']:.4f} > {alert['threshold']}")

        return "\n".join(report_lines)
```

1.7 Privacy-Preserving ML

1.7.1 Differential Privacy

A mechanism M satisfies (epsilon, delta)-differential privacy if:

P(M(D) in S) <= exp(epsilon) * P(M(D') in S) + delta

for all datasets D, D' differing in one record, and all output sets S.

Interpretation: The presence or absence of any individual's data has limited impact on the output.

Gaussian Mechanism: Add noise N(0, sigma^2) where sigma = sqrt(2 * ln(1.25/delta)) * sensitivity / epsilon.

1.7.2 Federated Learning

Train models on distributed data without centralizing:

1. Server sends model to clients
2. Clients train locally on private data
3. Clients send gradients/updates to server
4. Server aggregates updates

Privacy enhanced by: - Secure aggregation (server only sees sum) - Differential privacy on updates - Local training iterations

---

1.8 Common Parameters

() *	() *	() *	() *
0.25	5	25	Typical 0.3409 Range
() *	() *	() *	() *
0.25	5	25	Ac- ness- thresh- old 0.15 ept- able dis- par- ity
() *	() *	() *	() *
0.25	5	25	Le- par- ity im- pact 0.8 gal thresh- old
() *	() *	() *	() *
0.25	5	25	Fairness- ver- sar- ial de- bi- as- ing 2.0 accuracy trade- off
() *	() *	() *	() *
0.25	5	25	Weights ing com- puted from data
() *	() *	() *	() *
0.25	5	25	Per- old op- ti- miza- tion 0.0 group thresh- olds
() *	() *	() *	() *
0.25	5	25	Pri- fer- en- tial pri- vacy 1.0 vacy bud- get

() *	() *	() *	() *
0.25	0.3409	Typical Range	

() *	() *	() *	() *
0.25	0.3409	Fail-	

fer- en- tial pri- vacy	5 to 1e-3	ure prob- a- bil- ity

1.9 Practice Problems

1. **Impossibility Verification:** Create synthetic data with different base rates. Train a calibrated model and verify that equal TPR and FPR cannot both be achieved. Plot the tradeoff.
2. **Mitigation Comparison:** Apply reweighting, threshold optimization, and adversarial debiasing to the same dataset. Compare resulting fairness metrics and accuracy.
3. **SHAP Analysis:** Train a model and compute SHAP values. Identify whether protected attribute has high importance. Analyze whether proxy features correlate with the protected attribute.
4. **Monitoring System:** Implement a production monitoring system that tracks fairness over time. Simulate data drift and detect when fairness metrics exceed thresholds.
5. **Causal Analysis:** Draw a causal graph for a hiring scenario. Identify direct, indirect, and proxy discrimination paths. Propose interventions to block unfair paths.

1.10 References

1. Barocas, S., Hardt, M., & Narayanan, A. (2019). "Fairness and Machine Learning." fairmlbook.org.
2. Chouldechova, A. (2017). "Fair Prediction with Disparate Impact: A Study of Bias in Recidivism Prediction Instruments." Big Data.
3. Kleinberg, J., Mullainathan, S., & Raghavan, M. (2016). "Inherent Trade-Offs in the Fair Determination of Risk Scores." ITCS.
4. Dwork, C., et al. (2012). "Fairness Through Awareness." ITCS.
5. Lundberg, S. M., & Lee, S. I. (2017). "A Unified Approach to Interpreting Model Predictions." NeurIPS.
6. Zhang, B. H., Lemoine, B., & Mitchell, M. (2018). "Mitigating Unwanted Biases with Adversarial Learning." AIES.
7. Dwork, C. (2006). "Differential Privacy." ICALP.

Responsible AI requires both mathematical rigor and ethical reasoning. Fairness metrics provide quantitative measures, but choosing which metrics matter requires understanding the social context and stakeholder values. No algorithm can resolve value conflicts – human judgment remains essential.