# Handout 3: Advanced Fairness Metrics

Machine Learning for Smarter Innovation

## 1 Handout 3: Advanced Fairness Metrics

### 1.1 Mathematical Foundations & Implementation

#### 1.1.1 Part 1: Group Fairness Metrics

**1.1 Demographic Parity (Statistical Parity) Definition:**

```
P(D = 1 | A = a) = P(D = 1 | A = b) for all a, b
```

Where: - D: Decision (1 = positive outcome) - A: Protected attribute

**Relaxed Version** (-demographic parity):

```
|P(D = 1 | A = a) - P(D = 1 | A = b)|
```

Typical threshold: = 0.1 (10% disparity allowed)

**Pros:** - Simple to measure and explain - Ensures equal access to opportunities - No need for ground truth labels

**Cons:** - Ignores base rate differences - Can reduce overall accuracy - May require different error rates by group

**Implementation:**

```python
def demographic_parity_difference(y_true, y_pred, sensitive_features):
    groups = np.unique(sensitive_features)
    rates = []

    for group in groups:
        mask = sensitive_features == group
        rate = np.mean(y_pred[mask])
        rates.append(rate)

    return max(rates) - min(rates)

# Alternative: Demographic parity ratio
def demographic_parity_ratio(y_true, y_pred, sensitive_features):
    groups = np.unique(sensitive_features)
    rates = []

    for group in groups:
        mask = sensitive_features == group
        rate = np.mean(y_pred[mask])
        rates.append(rate)

    return min(rates) / max(rates)  # Ideal: 1.0, Acceptable: > 0.8
```

### 1.2 Equal Opportunity Definition:

```
P(D = 1 | Y = 1, A = a) = P(D = 1 | Y = 1, A = b)
```

Equivalently: TPR_a = TPR_b

**Intuition**: Qualified individuals have equal chances regardless of group.

**Mathematical Formulation:**

```
TPR_a = TP_a / (TP_a + FN_a)
TPR_b = TP_b / (TP_b + FN_b)

Equal Opportunity Difference = |TPR_a - TPR_b|
```

**Pros:** - Focuses on qualified individuals - Allows different base rates - Prevents discrimination against deserving

**Cons:** - Ignores false positives - Requires ground truth labels - May not prevent all harms

**Implementation:**

```
def equal_opportunity_difference(y_true, y_pred, sensitive_features):
    groups = np.unique(sensitive_features)
    tprs = []

    for group in groups:
        mask = (sensitive_features == group) & (y_true == 1)
        if mask.sum() > 0:
            tpr = np.mean(y_pred[mask])
            tprs.append(tpr)

    return max(tprs) - min(tprs) if tprs else 0.0
```

### 1.3 Equalized Odds Definition:

```
P(D = 1 | Y = y, A = a) = P(D = 1 | Y = y, A = b) for all y      {0, 1}
```

Equivalently: TPR_a = TPR_b AND FPR_a = FPR_b

**Mathematical Formulation:**

```
TPR: P(D = 1 | Y = 1, A = a) = P(D = 1 | Y = 1, A = b)
FPR: P(D = 1 | Y = 0, A = a) = P(D = 1 | Y = 0, A = b)

Equalized Odds Difference = max(|TPR_a - TPR_b|, |FPR_a - FPR_b|)
```

**Pros:** - Strongest group fairness notion - Balances both types of errors - Prediction independent of group given label

**Cons:** - Most restrictive (hardest to achieve) - May sacrifice accuracy - Requires careful calibration

**Implementation:**

```
def equalized_odds_difference(y_true, y_pred, sensitive_features):
    groups = np.unique(sensitive_features)
    tprs, fprs = [], []
```

```
    for group in groups:
        # TPR
        mask_pos = (sensitive_features == group) & (y_true == 1)
        if mask_pos.sum() > 0:
            tpr = np.mean(y_pred[mask_pos])
            tprs.append(tpr)

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

    tpr_diff = max(tprs) - min(tprs) if tprs else 0.0
    fpr_diff = max(fprs) - min(fprs) if fprs else 0.0

    return max(tpr_diff, fpr_diff)
```

### 1.4 Calibration Definition:

```
P(Y = 1 | S(X) = s, A = a) = P(Y = 1 | S(X) = s, A = b) = s
```

Where S(X) is the model's predicted probability.

**Intuition**: A score of 0.7 means 70% chance of positive outcome, regardless of group.

**Pros:** - Important for decision-making under uncertainty - Allows different acceptance rates - Interpretable probabilities

**Cons:** - Can coexist with other biases - Requires probability predictions - May not ensure equal treatment

**Implementation:**

```
from sklearn.calibration import calibration_curve

def calibration_by_group(y_true, y_pred_proba, sensitive_features, n_bins=10):
    groups = np.unique(sensitive_features)

    for group in groups:
        mask = sensitive_features == group
        prob_true, prob_pred = calibration_curve(
            y_true[mask],
            y_pred_proba[mask],
            n_bins=n_bins
        )

        # Calibration error
        calib_error = np.mean(np.abs(prob_true - prob_pred))
        print(f"Group {group}: Calibration Error = {calib_error:.4f}")

    # Cross-group calibration difference
    # Ideal: All groups have similar calibration curves
```

#### 1.1.2 Part 2: Individual Fairness

### 2.1 Lipschitz Condition Definition:

```
d_Y(M(x_i), M(x_j))     L    d_X(x_i, x_j)
```

Where: - M: Model - d_X: Distance metric in input space - d_Y: Distance metric in output space - L:

Lipschitz constant

**Intuition**: Similar individuals should receive similar predictions.

**Challenge**: Defining "similar" requires domain knowledge.

**Implementation:**

```
def check_individual_fairness(model, X, distance_metric, k=10, threshold=0.1):
    from sklearn.neighbors import NearestNeighbors

    # Find k-nearest neighbors
    nbrs = NearestNeighbors(n_neighbors=k, metric=distance_metric)
    nbrs.fit(X)
    distances, indices = nbrs.kneighbors(X)

    # Get predictions
    y_pred = model.predict_proba(X)[:, 1]

    # Check prediction similarity for neighbors
    violations = []
    for i in range(len(X)):
        for j in range(1, k):  # Skip self (j=0)
            neighbor_idx = indices[i, j]
            input_dist = distances[i, j]
            output_dist = abs(y_pred[i] - y_pred[neighbor_idx])

            # Violation if output distance disproportionate to input distance
            if output_dist > threshold and input_dist < threshold:
                violations.append((i, neighbor_idx, input_dist, output_dist))

    return violations
```

### 1.1.3 Part 3: Causal Fairness

**3.1 Counterfactual Fairness Definition:**

```
P(Y_A a(U) = y | X = x, A = a) = P(Y_A a'(U) = y | X = x, A = a)
```

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

**Requires**: Causal graph and structural equation model.

**Implementation** (conceptual):

```
# Requires causal inference library (e.g., dowhy)
import dowhy

# 1. Define causal model
model = dowhy.CausalModel(
    data=df,
    treatment='protected_attribute',
    outcome='prediction',
    graph=causal_graph
)

# 2. Estimate counterfactual effect
identified_estimand = model.identify_effect()
estimate = model.estimate_effect(identified_estimand)

# 3. Check if effect is near zero
print(f"Causal effect: {estimate.value}")
# Ideal:     0
```

### 1.1.4 Part 4: Impossibility Theorems

**4.1 Chouldechova's Theorem (2017)** **Statement**: If prevalence (base rates) differ across groups, you cannot simultaneously achieve: 1. Calibration by group 2. Equal FPR 3. Equal FNR

**Proof sketch**:

```
Given:
- Prevalence_a     Prevalence_b
- Calibration: PPV_a = PPV_b and NPV_a = NPV_b

Then:
- FPR and FNR must differ to maintain calibration

Mathematical derivation:
PPV = TP / (TP + FP) = TPR    Prev / (TPR    Prev + FPR    (1 - Prev))

If PPV_a = PPV_b and Prev_a     Prev_b,
then TPR and FPR cannot both be equal across groups.
```

**4.2 Kleinberg-Mullainathan-Raghavan (2016)** **Statement**: Cannot simultaneously achieve: 1. Calibration 2. Balance for positive class (equal opportunity) 3. Balance for negative class (equal FPR)

Unless: Perfect prediction OR equal base rates

**Implication**: Must choose which fairness notion to prioritize based on context.

### 1.1.5 Part 5: Practical Trade-offs

**5.1 Accuracy-Fairness Trade-off** **Pareto Frontier**:

```
from sklearn.model_selection import ParameterGrid

# Sweep fairness constraint strength
results = []
for constraint_weight in np.linspace(0, 1, 20):
    model = train_with_fairness_constraint(X_train, y_train, A_train,
    constraint_weight)

    accuracy = accuracy_score(y_test, model.predict(X_test))
    fairness = equalized_odds_difference(y_test, model.predict(X_test), A_test
    )

    results.append({
        'constraint_weight': constraint_weight,
        'accuracy': accuracy,
        'fairness_violation': fairness
    })

# Plot Pareto frontier
plt.scatter([r['fairness_violation'] for r in results],
            [r['accuracy'] for r in results])
plt.xlabel('Fairness Violation')
plt.ylabel('Accuracy')
plt.title('Accuracy-Fairness Trade-off')
```

**5.2 Multi-objective Optimization** **Formulation**:

```
minimize:       L(  ) + (1-  )    F(  )
```

```
Where:
- L(  ): Loss function (e.g., cross-entropy)
- F(  ): Fairness violation
-        [0, 1]: Trade-off parameter
```

**Implementation**:

```python
import torch
import torch.nn as nn

class FairClassifier(nn.Module):
    def __init__(self, input_dim, alpha=0.5):
        super().__init__()
        self.model = nn.Linear(input_dim, 1)
        self.alpha = alpha

    def forward(self, x):
        return torch.sigmoid(self.model(x))

    def compute_loss(self, x, y, a):
        pred = self.forward(x)

        # Standard loss
        bce_loss = nn.BCELoss()(pred, y)

        # Fairness loss (demographic parity)
        groups = torch.unique(a)
        group_preds = [pred[a == g].mean() for g in groups]
        fairness_loss = torch.var(torch.tensor(group_preds))

        # Combined loss
        total_loss = self.alpha * bce_loss + (1 - self.alpha) * fairness_loss

        return total_loss
```

### 1.1.6 Part 6: Advanced Mitigation Techniques

**6.1 Adversarial Debiasing Formulation**:

```
min_ _P  max_ _A L_P( _P ) -    L_A( _A )

Where:
-  _P : Predictor parameters
-  _A : Adversary parameters
- L_P: Prediction loss
- L_A: Adversary loss (predict protected attribute from predictions)
-   : Trade-off parameter
```

**Implementation**:

```python
class AdversarialDebiasing(nn.Module):
    def __init__(self, input_dim):
        super().__init__()
        self.predictor = nn.Sequential(
            nn.Linear(input_dim, 64),
            nn.ReLU(),
            nn.Linear(64, 1),
            nn.Sigmoid()
        )
        self.adversary = nn.Sequential(
```

```
            nn.Linear(1, 32),
            nn.ReLU(),
            nn.Linear(32, 1),
            nn.Sigmoid()
        )

    def train_step(self, x, y, a, lambda_adv=1.0):
        # Train predictor
        pred = self.predictor(x)
        pred_loss = nn.BCELoss()(pred, y)

        # Train adversary (predict protected attribute from predictions)
        adv_pred = self.adversary(pred.detach())
        adv_loss = nn.BCELoss()(adv_pred, a)

        # Combined loss
        total_loss = pred_loss - lambda_adv * adv_loss

        return total_loss
```

**6.2 Fairness Through Unawareness (and why it fails)** **Naive Approach**: Remove protected attribute from features.

**Why it fails**: Proxy variables (correlated features) leak information.

**Example**:

```
# Even without explicit race,these features correlate:
- Zip code     residential segregation
- First name     cultural background
- School name     socioeconomic status

# Correlation analysis
for feature in features:
    corr = np.corrcoef(df[feature], df['race'])[0, 1]
    if abs(corr) > 0.3:
        print(f"{feature}: {corr:.3f} correlation with race")
```

**Proper Approach**: Use fairness-aware methods that account for correlation structure.

### 1.1.7 Key Takeaway

**Mathematics provides precise definitions, but choosing the right fairness metric requires ethical judgment, domain knowledge, and stakeholder input.**

Different contexts require different fairness notions - there is no universal solution.