Responsible AI - Intermediate Handout

Machine Learning for Smarter Innovation

1 Responsible AI - Intermediate Handout

Target Audience: Practitioners with Python knowledge **Duration**: 60 minutes reading + coding **Level**: Intermediate (implementation focused)

1.1 Setup

```
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
from sklearn.model_selection import train_test_split
from sklearn.ensemble import RandomForestClassifier,
    GradientBoostingClassifier
from sklearn.linear_model import LogisticRegression
from sklearn.metrics import (
    accuracy_score, precision_score, recall_score,
    confusion_matrix, classification_report
)
from scipy.stats import chi2_contingency
import warnings
warnings.filterwarnings('ignore')

# Fairness tools
from fairlearn.metrics import (
    MetricFrame, demographic_parity_difference,
    equalized_odds_difference, demographic_parity_ratio
)
from fairlearn.reductions import ExponentiatedGradient, EqualizedOdds

# Explainability
import shap

# Visualization settings
plt.rcParams.update({
    'font.size': 10,
    'axes.labelsize': 10,
    'figure.figsize': (10, 6)
})
```

This handout covers practical implementation of responsible AI practices including bias detection, fairness metrics, model explainability, and ongoing monitoring. Responsible AI ensures machine learning systems treat all groups equitably and provide transparent explanations for their decisions. The techniques apply across high-stakes domains including lending, hiring, healthcare, and criminal justice where algorithmic decisions have significant human impact.

1.2 1. Bias Detection in Data

1.2.1 Concept Overview

Bias can enter ML systems through multiple pathways: historical bias in training labels, measurement bias from data collection, representation bias from undersampled groups, and aggregation bias from treating diverse populations uniformly. Before training models, systematic data auditing identifies potential fairness issues. Key checks include demographic representation, label distribution across groups, and proxy variable detection.

1.2.2 Implementation: Data Audit Pipeline

```
class DataBiasAuditor:
    """Audit dataset for potential bias sources."""

    def __init__(self, df, target_col, protected_cols):
        self.df = df
        self.target = target_col
        self.protected = protected_cols
        self.audit_results = {}

    def representation_analysis(self):
        """Check group representation in the dataset."""
        results = {}

        for col in self.protected:
            counts = self.df[col].value_counts()
            proportions = self.df[col].value_counts(normalize=True)

            results[col] = {
                'counts': counts.to_dict(),
                'proportions': proportions.to_dict(),
                'underrepresented': [
                    group for group, prop in proportions.items() if prop <
   0.05
                ]
            }

        self.audit_results['representation'] = results
        return results

    def label_distribution_analysis(self):
        """Check outcome distribution across protected groups."""
        results = {}

        for col in self.protected:
            # Positive rate by group
            positive_rates = self.df.groupby(col)[self.target].mean()

            # Chi-square test for independence
            contingency = pd.crosstab(self.df[col], self.df[self.target])
            chi2, p_value, dof, expected = chi2_contingency(contingency)

            results[col] = {
                'positive_rates': positive_rates.to_dict(),
                'chi2_statistic': chi2,
                'p_value': p_value,
                'significant_difference': p_value < 0.05
```

```
            }

        self.audit_results['label_distribution'] = results
        return results

    def proxy_detection(self, threshold=0.3):
        """Identify features that may act as proxies for protected attributes.
    """
        results = {}

        # Encode protected attributes for correlation
        df_encoded = self.df.copy()
        for col in self.protected:
            if df_encoded[col].dtype == 'object':
                df_encoded[col] = pd.factorize(df_encoded[col])[0]

        numeric_cols = df_encoded.select_dtypes(include=[np.number]).columns
        feature_cols = [c for c in numeric_cols if c not in self.protected + [
    self.target]]

        for protected_col in self.protected:
            correlations = {}
            for feature in feature_cols:
                corr = df_encoded[feature].corr(df_encoded[protected_col])
                if abs(corr) > threshold:
                    correlations[feature] = round(corr, 3)

            results[protected_col] = {
                'high_correlation_features': correlations,
                'potential_proxies': list(correlations.keys())
            }

        self.audit_results['proxies'] = results
        return results

    def generate_report(self):
        """Generate comprehensive bias audit report."""
        if not self.audit_results:
            self.representation_analysis()
            self.label_distribution_analysis()
            self.proxy_detection()

        report = []
        report.append("=" * 60)
        report.append("DATA BIAS AUDIT REPORT")
        report.append("=" * 60)

        # Representation
        report.append("\n1. REPRESENTATION ANALYSIS")
        for col, data in self.audit_results['representation'].items():
            report.append(f"\n  {col}:")
            for group, prop in data['proportions'].items():
                flag = " [UNDERREPRESENTED]" if prop < 0.05 else ""
                report.append(f"    {group}: {prop:.1%}{flag}")

        # Label distribution
        report.append("\n2. LABEL DISTRIBUTION")
        for col, data in self.audit_results['label_distribution'].items():
            report.append(f"\n  {col}:")
            for group, rate in data['positive_rates'].items():
                report.append(f"    {group}: {rate:.1%} positive rate")
            if data['significant_difference']:
```

```
                report.append(f"    [WARNING] Significant difference detected
    (p={data['p_value']:.4f})")

        # Proxy detection
        report.append("\n3. POTENTIAL PROXY VARIABLES")
        for col, data in self.audit_results['proxies'].items():
            if data['potential_proxies']:
                report.append(f"\n  Features correlated with {col}:")
                for feat, corr in data['high_correlation_features'].items():
                    report.append(f"    {feat}: r={corr}")

        return '\n'.join(report)


# Example usage with synthetic data
np.random.seed(42)
n_samples = 2000

data = {
    'age': np.random.randint(22, 65, n_samples),
    'income': np.random.lognormal(10.5, 0.5, n_samples),
    'credit_score': np.random.randint(550, 850, n_samples),
    'gender': np.random.choice(['male', 'female'], n_samples, p=[0.55, 0.45]),
    'zip_code': np.random.choice(['urban', 'suburban', 'rural'], n_samples)
}

# Create biased outcome (women have lower approval rate)
df = pd.DataFrame(data)
base_prob = 0.5 + (df['credit_score'] - 700) / 500 + (df['income'] - 40000) /
    100000
bias_factor = np.where(df['gender'] == 'female', -0.1, 0.05)
df['approved'] = (np.random.random(n_samples) < (base_prob + bias_factor)).
    astype(int)

auditor = DataBiasAuditor(df, 'approved', ['gender'])
print(auditor.generate_report())
```

1.3 2. Fairness Metrics and Evaluation

1.3.1 Concept Overview

Multiple fairness definitions exist, each encoding different ethical positions. Demographic parity requires equal positive prediction rates across groups. Equal opportunity requires equal true positive rates. Equalized odds extends this to also require equal false positive rates. These definitions can conflict, making metric selection a domain-specific decision based on the relative costs of different error types.

1.3.2 Implementation: Comprehensive Fairness Assessment

```
class FairnessEvaluator:
    """Evaluate model fairness across multiple metrics."""

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

```
    def demographic_parity(self):
        """Calculate demographic parity metrics."""
        rates = {}
        for group in self.groups:
            mask = self.sensitive == group
            rates[group] = self.y_pred[mask].mean()

        difference = max(rates.values()) - min(rates.values())
        ratio = min(rates.values()) / max(rates.values()) if max(rates.values
    ()) > 0 else 0

        return {
            'group_rates': rates,
            'difference': difference,
            'ratio': ratio,
            'passes_threshold': difference < 0.1
        }

    def equal_opportunity(self):
        """Calculate equal opportunity (TPR equality)."""
        tpr = {}
        for group in self.groups:
            mask = (self.sensitive == group) & (self.y_true == 1)
            if mask.sum() > 0:
                tpr[group] = self.y_pred[mask].mean()
            else:
                tpr[group] = np.nan

        valid_tpr = [v for v in tpr.values() if not np.isnan(v)]
        difference = max(valid_tpr) - min(valid_tpr) if valid_tpr else np.nan

        return {
            'group_tpr': tpr,
            'difference': difference,
            'passes_threshold': difference < 0.1 if not np.isnan(difference)
    else False
        }

    def equalized_odds(self):
        """Calculate equalized odds (TPR and FPR equality)."""
        tpr = {}
        fpr = {}

        for group in self.groups:
            mask = self.sensitive == group

            # TPR
            pos_mask = mask & (self.y_true == 1)
            if pos_mask.sum() > 0:
                tpr[group] = self.y_pred[pos_mask].mean()
            else:
                tpr[group] = np.nan

            # FPR
            neg_mask = mask & (self.y_true == 0)
            if neg_mask.sum() > 0:
                fpr[group] = self.y_pred[neg_mask].mean()
            else:
                fpr[group] = np.nan

        valid_tpr = [v for v in tpr.values() if not np.isnan(v)]
        valid_fpr = [v for v in fpr.values() if not np.isnan(v)]
```

```
        tpr_diff = max(valid_tpr) - min(valid_tpr) if valid_tpr else np.nan
        fpr_diff = max(valid_fpr) - min(valid_fpr) if valid_fpr else np.nan

        return {
            'group_tpr': tpr,
            'group_fpr': fpr,
            'tpr_difference': tpr_diff,
            'fpr_difference': fpr_diff,
            'passes_threshold': (tpr_diff < 0.1 and fpr_diff < 0.1)
        }

    def calibration_by_group(self, n_bins=10):
        """Check prediction calibration across groups."""
        if self.y_proba is None:
            return None

        calibration = {}
        for group in self.groups:
            mask = self.sensitive == group
            proba = self.y_proba[mask]
            true = self.y_true[mask]

            bins = np.linspace(0, 1, n_bins + 1)
            bin_idx = np.digitize(proba, bins) - 1
            bin_idx = np.clip(bin_idx, 0, n_bins - 1)

            bin_means = []
            bin_true = []
            for i in range(n_bins):
                bin_mask = bin_idx == i
                if bin_mask.sum() > 0:
                    bin_means.append(proba[bin_mask].mean())
                    bin_true.append(true[bin_mask].mean())

            calibration[group] = {
                'predicted': bin_means,
                'actual': bin_true
            }

        return calibration

    def full_report(self):
        """Generate comprehensive fairness report."""
        dp = self.demographic_parity()
        eo = self.equal_opportunity()
        eqo = self.equalized_odds()

        print("=" * 60)
        print("FAIRNESS EVALUATION REPORT")
        print("=" * 60)

        print("\n1. DEMOGRAPHIC PARITY")
        print(f"   Definition: Equal positive prediction rates")
        for group, rate in dp['group_rates'].items():
            print(f"   {group}: {rate:.3f}")
        print(f"   Difference: {dp['difference']:.3f}")
        print(f"   Status: {'PASS' if dp['passes_threshold'] else 'FAIL'}")

        print("\n2. EQUAL OPPORTUNITY")
        print(f"   Definition: Equal true positive rates")
        for group, tpr in eo['group_tpr'].items():
            print(f"   {group} TPR: {tpr:.3f}")
```

```
        print(f"   TPR Difference: {eo['difference']:.3f}")
        print(f"   Status: {'PASS' if eo['passes_threshold'] else 'FAIL'}")

        print("\n3. EQUALIZED ODDS")
        print(f"   Definition: Equal TPR and FPR")
        print(f"   TPR Difference: {eqo['tpr_difference']:.3f}")
        print(f"   FPR Difference: {eqo['fpr_difference']:.3f}")
        print(f"   Status: {'PASS' if eqo['passes_threshold'] else 'FAIL'}")

        return {'demographic_parity': dp, 'equal_opportunity': eo, '
    equalized_odds': eqo}


# Train a model and evaluate fairness
X = df[['age', 'income', 'credit_score']].copy()
X['gender_encoded'] = (df['gender'] == 'female').astype(int)
y = df['approved']

X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2,
    random_state=42)
sensitive_train = df.loc[X_train.index, 'gender'].values
sensitive_test = df.loc[X_test.index, 'gender'].values

model = RandomForestClassifier(n_estimators=100, random_state=42)
model.fit(X_train.drop('gender_encoded', axis=1), y_train)

y_pred = model.predict(X_test.drop('gender_encoded', axis=1))
y_proba = model.predict_proba(X_test.drop('gender_encoded', axis=1))[:, 1]

evaluator = FairnessEvaluator(y_test, y_pred, y_proba, sensitive_test)
results = evaluator.full_report()
```

1.4 3. Model Explainability with SHAP

1.4.1 Concept Overview

SHAP (SHapley Additive exPlanations) provides theoretically-grounded explanations by computing each feature's contribution to individual predictions. Based on game-theoretic Shapley values, SHAP offers both local explanations (why did this individual get this prediction) and global explanations (which features matter most overall). SHAP explanations help identify if protected attributes or their proxies drive predictions.

1.4.2 Implementation: SHAP Analysis

```
class SHAPExplainer:
    """SHAP-based model explanation for fairness analysis."""

    def __init__(self, model, X_train, feature_names=None):
        self.model = model
        self.X_train = X_train
        self.feature_names = feature_names or list(X_train.columns)

        # Create appropriate explainer
        if hasattr(model, 'estimators_'):  # Tree ensemble
            self.explainer = shap.TreeExplainer(model)
        else:
            # Sample background for kernel explainer
```

```
            background = shap.sample(X_train, 100)
            self.explainer = shap.KernelExplainer(model.predict_proba,
    background)

    def explain_instance(self, instance, show_plot=True):
        """Explain a single prediction."""
        if isinstance(instance, pd.Series):
            instance = instance.values.reshape(1, -1)
        elif len(instance.shape) == 1:
            instance = instance.reshape(1, -1)

        shap_values = self.explainer.shap_values(instance)

        # Handle different output formats
        if isinstance(shap_values, list):
            shap_values = shap_values[1]  # For binary classification, take
    positive class

        explanation = dict(zip(self.feature_names, shap_values[0]))

        if show_plot:
            shap.force_plot(
                self.explainer.expected_value if not isinstance(self.explainer
    .expected_value, list)
                else self.explainer.expected_value[1],
                shap_values[0],
                instance,
                feature_names=self.feature_names,
                matplotlib=True,
                show=False
            )
            plt.tight_layout()
            plt.show()

        return explanation

    def global_importance(self, X_sample, show_plot=True):
        """Calculate global feature importance."""
        shap_values = self.explainer.shap_values(X_sample)

        if isinstance(shap_values, list):
            shap_values = shap_values[1]

        importance = np.abs(shap_values).mean(axis=0)
        importance_df = pd.DataFrame({
            'feature': self.feature_names,
            'importance': importance
        }).sort_values('importance', ascending=False)

        if show_plot:
            shap.summary_plot(shap_values, X_sample, feature_names=self.
    feature_names, show=False)
            plt.tight_layout()
            plt.savefig('shap_summary.pdf', bbox_inches='tight')
            plt.show()

        return importance_df

    def fairness_analysis(self, X_test, sensitive_features):
        """Analyze SHAP values by protected group."""
        shap_values = self.explainer.shap_values(X_test)

        if isinstance(shap_values, list):
```

```
            shap_values = shap_values[1]

        groups = np.unique(sensitive_features)
        group_importance = {}

        for group in groups:
            mask = sensitive_features == group
            group_shap = shap_values[mask]
            group_importance[group] = np.abs(group_shap).mean(axis=0)

        # Compare feature importance across groups
        comparison_df = pd.DataFrame(group_importance, index=self.
    feature_names)
        comparison_df['difference'] = abs(
            comparison_df.iloc[:, 0] - comparison_df.iloc[:, 1]
        )

        return comparison_df.sort_values('difference', ascending=False)


# SHAP analysis
X_test_features = X_test.drop('gender_encoded', axis=1)
explainer = SHAPExplainer(model, X_train.drop('gender_encoded', axis=1))

# Global importance
importance = explainer.global_importance(X_test_features[:100], show_plot=True
    )
print("\nGlobal Feature Importance:")
print(importance)

# Fairness-focused SHAP analysis
comparison = explainer.fairness_analysis(X_test_features.values,
    sensitive_test)
print("\nFeature Importance by Group:")
print(comparison)
```

1.5 4. Bias Mitigation Strategies

1.5.1 Concept Overview

Bias mitigation operates at three stages: pre-processing (modifying training data), in-processing (constraining model training), and post-processing (adjusting predictions). Pre-processing techniques include reweighting samples and removing proxy features. In-processing adds fairness constraints to the optimization objective. Post-processing calibrates decision thresholds per group to equalize outcomes.

1.5.2 Implementation: Mitigation Techniques

```
class BiasMitigator:
    """Apply bias mitigation techniques."""

    def __init__(self, X_train, y_train, sensitive_train):
        self.X_train = X_train
        self.y_train = y_train
        self.sensitive = sensitive_train

    def reweight_samples(self):
        """Calculate sample weights to balance representation."""
```

```
        groups = np.unique(self.sensitive)
        weights = np.ones(len(self.y_train))

        for group in groups:
            for label in [0, 1]:
                mask = (self.sensitive == group) & (self.y_train == label)
                if mask.sum() > 0:
                    # Weight inversely proportional to group-label frequency
                    expected = len(self.y_train) / (len(groups) * 2)
                    actual = mask.sum()
                    weights[mask] = expected / actual

        return weights

    def train_with_fairness_constraint(self, base_estimator, constraint_type='
equalized_odds'):
        """Train model with fairness constraints using Fairlearn."""

        if constraint_type == 'equalized_odds':
            constraint = EqualizedOdds()
        else:
            raise ValueError(f"Unknown constraint: {constraint_type}")

        mitigator = ExponentiatedGradient(
            estimator=base_estimator,
            constraints=constraint,
            max_iter=50
        )

        mitigator.fit(self.X_train, self.y_train, sensitive_features=self.
sensitive)
        return mitigator

    def threshold_optimizer(self, model, X_val, y_val, sensitive_val, metric='
equalized_odds'):
        """Find optimal thresholds per group to satisfy fairness constraint.""
"
        y_proba = model.predict_proba(X_val)[:, 1]
        groups = np.unique(sensitive_val)

        best_thresholds = {}
        thresholds_to_try = np.linspace(0.3, 0.7, 41)

        for group in groups:
            mask = sensitive_val == group
            group_proba = y_proba[mask]
            group_true = y_val.values[mask] if hasattr(y_val, 'values') else
y_val[mask]

            best_f1 = 0
            best_thresh = 0.5

            for thresh in thresholds_to_try:
                group_pred = (group_proba >= thresh).astype(int)
                f1 = 2 * precision_score(group_true, group_pred, zero_division
=0) * \
                     recall_score(group_true, group_pred, zero_division=0) / \
                     max(precision_score(group_true, group_pred, zero_division
=0) + \
                         recall_score(group_true, group_pred, zero_division=0)
, 1e-10)

                if f1 > best_f1:
```

```
                    best_f1 = f1
                    best_thresh = thresh

            best_thresholds[group] = best_thresh

        return best_thresholds


# Apply mitigation
mitigator = BiasMitigator(
    X_train.drop('gender_encoded', axis=1),
    y_train,
    sensitive_train
)

# Method 1: Reweighting
weights = mitigator.reweight_samples()
model_weighted = RandomForestClassifier(n_estimators=100, random_state=42)
model_weighted.fit(
    X_train.drop('gender_encoded', axis=1),
    y_train,
    sample_weight=weights
)

# Method 2: Fairness constraint
base_model = LogisticRegression(max_iter=1000)
fair_model = mitigator.train_with_fairness_constraint(base_model)

# Compare results
print("\nOriginal Model:")
y_pred_orig = model.predict(X_test.drop('gender_encoded', axis=1))
eval_orig = FairnessEvaluator(y_test, y_pred_orig, None, sensitive_test)
eval_orig.full_report()

print("\nWeighted Model:")
y_pred_weighted = model_weighted.predict(X_test.drop('gender_encoded', axis=1)
    )
eval_weighted = FairnessEvaluator(y_test, y_pred_weighted, None,
    sensitive_test)
eval_weighted.full_report()

print("\nFairness-Constrained Model:")
y_pred_fair = fair_model.predict(X_test.drop('gender_encoded', axis=1))
eval_fair = FairnessEvaluator(y_test, y_pred_fair, None, sensitive_test)
eval_fair.full_report()
```

1.6 5. Monitoring and Documentation

1.6.1 Concept Overview

Responsible AI requires ongoing monitoring because model fairness can degrade as data distributions shift. Production systems need fairness dashboards that track metrics over time, alert on threshold violations, and log explanations for auditing. Model cards document training data, performance characteristics, and known limitations for downstream users.

1.6.2 Implementation: Fairness Monitoring

```python
class FairnessMonitor:
    """Monitor model fairness over time."""

    def __init__(self, model, fairness_thresholds=None):
        self.model = model
        self.thresholds = fairness_thresholds or {
            'demographic_parity_diff': 0.1,
            'equalized_odds_diff': 0.1,
            'accuracy_diff': 0.05
        }
        self.history = []

    def log_evaluation(self, X, y_true, sensitive_features, timestamp=None):
        """Evaluate and log fairness metrics."""
        import datetime
        timestamp = timestamp or datetime.datetime.now()

        y_pred = self.model.predict(X)

        # Calculate metrics
        metric_frame = MetricFrame(
            metrics={
                'accuracy': accuracy_score,
                'precision': precision_score,
                'recall': recall_score
            },
            y_true=y_true,
            y_pred=y_pred,
            sensitive_features=sensitive_features
        )

        dp_diff = demographic_parity_difference(y_true, y_pred,
    sensitive_features)
        eo_diff = equalized_odds_difference(y_true, y_pred, sensitive_features
    )

        record = {
            'timestamp': timestamp,
            'accuracy': accuracy_score(y_true, y_pred),
            'demographic_parity_diff': dp_diff,
            'equalized_odds_diff': eo_diff,
            'accuracy_by_group': metric_frame.by_group['accuracy'].to_dict(),
            'violations': []
        }

        # Check for violations
        if dp_diff > self.thresholds['demographic_parity_diff']:
            record['violations'].append('demographic_parity')
        if eo_diff > self.thresholds['equalized_odds_diff']:
            record['violations'].append('equalized_odds')

        accuracy_by_group = list(metric_frame.by_group['accuracy'].values())
        if max(accuracy_by_group) - min(accuracy_by_group) > self.thresholds['
    accuracy_diff']:
            record['violations'].append('accuracy_disparity')

        self.history.append(record)
        return record

    def plot_history(self):
        """Visualize fairness metrics over time."""
        if len(self.history) < 2:
            print("Need at least 2 evaluations to plot history")
```

```
            return

        df = pd.DataFrame(self.history)

        fig, axes = plt.subplots(1, 3, figsize=(15, 4))

        # Accuracy over time
        axes[0].plot(range(len(df)), df['accuracy'], marker='o')
        axes[0].set_xlabel('Evaluation')
        axes[0].set_ylabel('Accuracy')
        axes[0].set_title('Overall Accuracy')
        axes[0].grid(True, alpha=0.3)

        # Demographic parity
        axes[1].plot(range(len(df)), df['demographic_parity_diff'], marker='o'
    )
        axes[1].axhline(y=self.thresholds['demographic_parity_diff'],
                        color='r', linestyle='--', label='Threshold')
        axes[1].set_xlabel('Evaluation')
        axes[1].set_ylabel('Difference')
        axes[1].set_title('Demographic Parity Difference')
        axes[1].legend()
        axes[1].grid(True, alpha=0.3)

        # Equalized odds
        axes[2].plot(range(len(df)), df['equalized_odds_diff'], marker='o')
        axes[2].axhline(y=self.thresholds['equalized_odds_diff'],
                        color='r', linestyle='--', label='Threshold')
        axes[2].set_xlabel('Evaluation')
        axes[2].set_ylabel('Difference')
        axes[2].set_title('Equalized Odds Difference')
        axes[2].legend()
        axes[2].grid(True, alpha=0.3)

        plt.tight_layout()
        plt.savefig('fairness_monitoring.pdf', bbox_inches='tight')
        plt.show()

    def generate_model_card(self, model_name, model_description):
        """Generate model card documentation."""
        if not self.history:
            return "No evaluation history available"

        latest = self.history[-1]

        card = f"""
# Model Card: {model_name}

## Model Description
{model_description}

## Performance Metrics
- Overall Accuracy: {latest['accuracy']:.3f}
- Accuracy by Group: {latest['accuracy_by_group']}

## Fairness Metrics
- Demographic Parity Difference: {latest['demographic_parity_diff']:.3f}
  (Threshold: {self.thresholds['demographic_parity_diff']})
- Equalized Odds Difference: {latest['equalized_odds_diff']:.3f}
  (Threshold: {self.thresholds['equalized_odds_diff']})

## Known Limitations
```

```
{', '.join(latest['violations']) if latest['violations'] else 'No violations
    detected'}

## Recommendations
- Monitor fairness metrics monthly
- Retrain if demographic parity difference exceeds 0.15
- Consider reweighting if new groups emerge in data
"""
        return card


# Example monitoring
monitor = FairnessMonitor(model)

# Simulate multiple evaluations
for i in range(5):
    record = monitor.log_evaluation(
        X_test.drop('gender_encoded', axis=1),
        y_test,
        sensitive_test
    )
    print(f"Evaluation {i+1}: Violations = {record['violations']}")

# Generate model card
model_card = monitor.generate_model_card(
    "Loan Approval Classifier v1.0",
    "Random Forest classifier for automated loan approval decisions"
)
print(model_card)
```

1.7 Common Fairness Thresholds

() *	() *	() *	() *
0.1455	Mo	1273	Notes
0.3273	Range		Stakes
			Thresh-
0.4000	old		

() *	() *	() *	() *
0.1	4	0	273
mo	0.1	0.0	5
graphic			i-
Par-			tive
ity			rates
Dif-			across
fer-			groups
ence			

() *	() *	() *	() *
0.1	4	0	273
ized	0.1	0.0	5
Odds			TPR
Dif-			and
fer-			FPR
ence			

() *	() *	() *	() *
0.1455	Mo 1.73	Ac Hi	Notes
	0.3273	Ra Sta	
			Thresh-
	0.4000		old

() *	() *	() *	() *
0.14	1.73	0.27	“80%
mo	0.80	0.90	ule”
graphic Par- ity Ra- tio			from dis- parate im- pact

() *	() *	() *	() *
0.14	1.73	0.27	Equal
i- bra- tion Dif- fer- ence	0.05	0.02	re- dicted vs ac- tual rates

() *	() *	() *	() *
0.14	1.73	0.27	Per-
cu- racy Dis- par- ity	0.05	0.02	or- mance gap across groups

1.8 Practice Projects

1. **Lending Bias Audit**: Audit a loan approval dataset for demographic bias. Identify proxy variables, calculate fairness metrics, and implement reweighting to reduce disparate impact while maintaining predictive accuracy.
2. **Hiring Algorithm Fairness**: Build a resume screening model and evaluate fairness across gender and ethnicity. Use SHAP to identify which features drive disparate outcomes and apply in-processing constraints.
3. **Healthcare Risk Prediction**: Develop a patient risk model and audit for age and socioeconomic bias. Implement threshold optimization to achieve equalized odds across demographic groups.
4. **Model Card Generator**: Create an automated pipeline that generates comprehensive model cards including fairness metrics, SHAP explanations, and performance breakdowns by protected attributes.

1.9 Troubleshooting

()	()	()
*	*	*
0.3	[illegible]	[illegible]ution
()	()	()
*	*	*
0.3	[illegible]	[illegible]s-
ac-	trade-	con-
trade-	off	straints
off	too	se-
vere		
()	()	()
*	*	*
0.3	[illegible]	[illegible]se
tec-	tor	proxy
at-	lec-	de-
tri-	bute	tec-
not	tion,	
avail-	able	col-
lec-	tion	at-
tribute	eth-	
i-	cally	
()	()	()
*	*	*
0.3	[illegible]	[illegible]le-
ter-	at-	tri-
sec-	bute	ana-
lyze	tional	com-
bi-	as	is
missed	da-	
tions	of	
pro-	tected	
at-	tributes	
()	()	()
*	*	*
0.3	[illegible]	[illegible]Choose
rics	fer-	met-
con-	ric	
flict	fair-	aligned
ness	with	
def-	do-	
i-	main	
ni-	ethics	
tions		

[illegible]	[illegible]	[illegible]ution
[illegible] cant in-complete	[illegible]ing eval-u-a-tion data	[illegible] ple-ment com-pre-hen-sive log-ging from start
[illegible] com-pu-ta-tion slow	[illegible] data or com-plex model	[illegible] ple data, use Tree-Ex-plainer for en-sem-bles
[illegible] i-ga-tion re-duces ac-cu-racy too much	[illegible] cor-rec-tion	[illegible] ance fair-ness and util-ity with soft con-straints
[illegible] de-tec-tion fail-ing	[illegible] old too loose	[illegible] thresh-olds, add sta-tis-ti-cal tests

1.10 Next Steps

- Read the advanced handout for causal fairness and intersectionality analysis
- Implement counterfactual explanations for individual decisions
- Explore AIF360 for additional bias mitigation algorithms
- Build automated fairness testing into CI/CD pipelines
- Develop stakeholder communication materials for fairness findings

---

Responsible AI is not a checkbox but an ongoing commitment. Technical fairness tools are necessary but insufficient, requiring integration with domain expertise, stakeholder engagement, and organizational accountability. Build systems that are not just accurate but worthy of the trust placed in them.