Content:

1. Information Gain (IG)
2. Fairness Gain (FG)
3. Fair Information Gain (FIG)
4. Python Implementation
5. Experimental Analysis

1) Information Gain (IG)

In Decision Trees our goal is to divide the tree into purer subtrees by selecting the best feature for splitting. By purer we mean that we want to split the tree into subtress that its data points all belong to the same class and are not random. In the other words, we are trying to **reduce impurity.**

So we need to select the best feature that splitting the tree using it, leads to the most reduction in impurity. The metric that indicates this reduction in impurity is called **Information Gain (IG)**. During the process of building a decision tree, our goal is to **maximize** the **IG** at each step.

IG is calculated by the difference of impurity of the original tree with the weighted average impurity of subtrees after splitting:

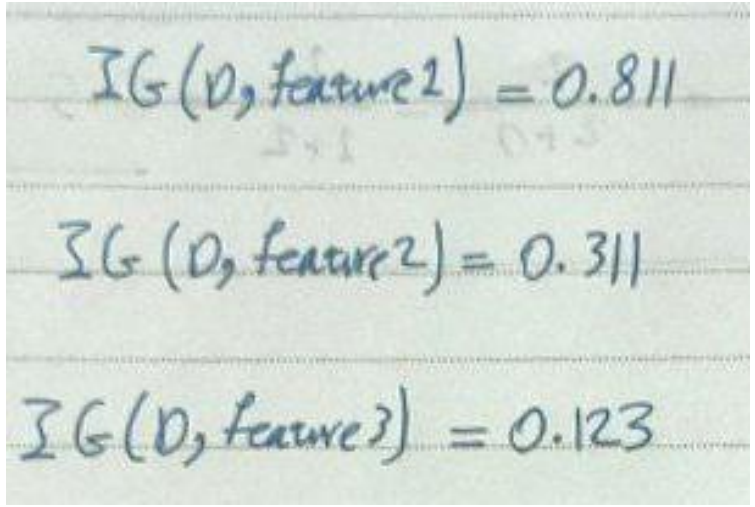
$$\text{Information Gain} = \text{Initial Impurity} - \text{Weighted Impurity}$$

At each step, the feature with the **highest IG** is selected as the best feature to split the tree. For measuring impurity, **Entropy** or **Gini** formulas can be used.

1) Information Gain (IG)

dataIndex	feature1	feature2	feature3	classLabel
1	1	1	1	1
2	1	0	0	1
3	1	0	1	1
4	0	1	1	0

Above is an example dataset. We calculate the IGs for each feature to select the best feature to split the tree:



2) Fairness Gain (FG)

In some datasets we may have features such as race or sex that selecting them as the best feature may result in **Bias** and **Unfairness**. For example, assume that a bank wants to give credits to it's customers based on their information. It will be **unfair** to split the tree based on race of each customer and decide whether to give them the credit or not. Because one group will be favored to grant this benefit and the other group will be deprived and will be rejected the benefit. In order to measure the fairness of selecting a feature to split the tree we use the **Fariness Gain (FG)** metric. It is calculated by the difference of the discrimination in the original tree and the weighted average discrimination of the subtress after splitting:

$$FG(D, A) = |Disc(D)| - \sum_{v \in dom(A)} \frac{|D_v|}{|D|} |Disc(D_v)|$$

$$Disc(D) = \frac{FG}{FG + FR} - \frac{DG}{DG + DR}$$

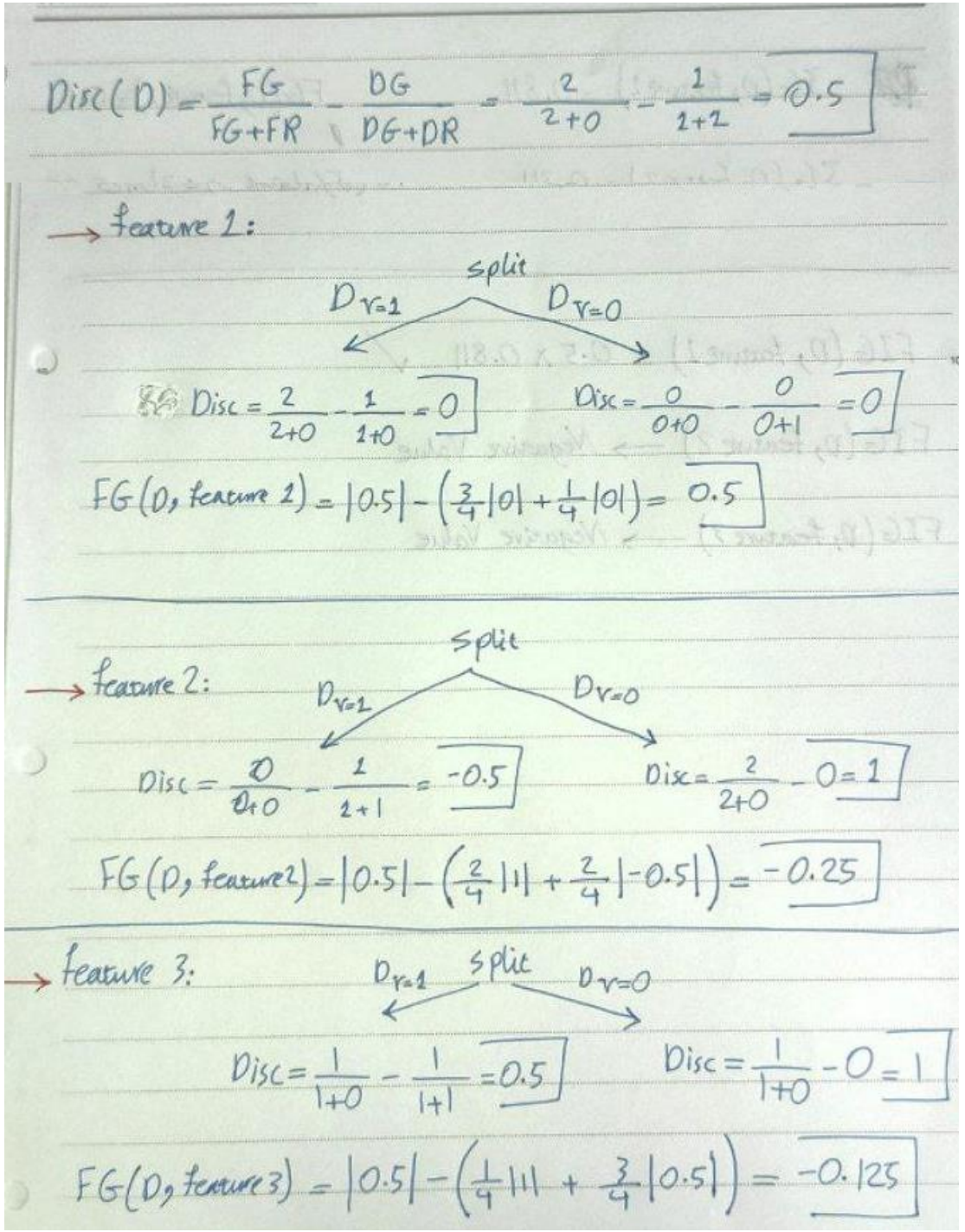
At each step of building a decision tree, we have to **maximize the FG** values by selecting the best feature.

2) Fairness Gain (FG)

dataIndex	feature1	feature2	feature3	classLabel
1	1	1	1	1
2	1	0	0	1
3	1	0	1	1
4	0	1	1	0

Lets calculate the Fairness Gain values of each feature in our toy dataset:

According to the table, the group with feature2 value of 0 has been granted the benefit more than the group with feature2 value of 1. As a result, we consider the group with 0 values as **Favored**, and the group with 1 values as **Deprived**.



3) Fair Information Gain (FIG)

Now assume that we have a tree with sensitive features such as Race or Sex. Like any other decision tree, we have to select the best feature with highest IG at each step. But in order to have fair trees as well, we need to **penalize** the IG value when selecting that feature results in an **increase in unfariness.** To do so, we use the **Fair Information Gain (FIG)** formula below that considers both **impurity** and **unfairness**.

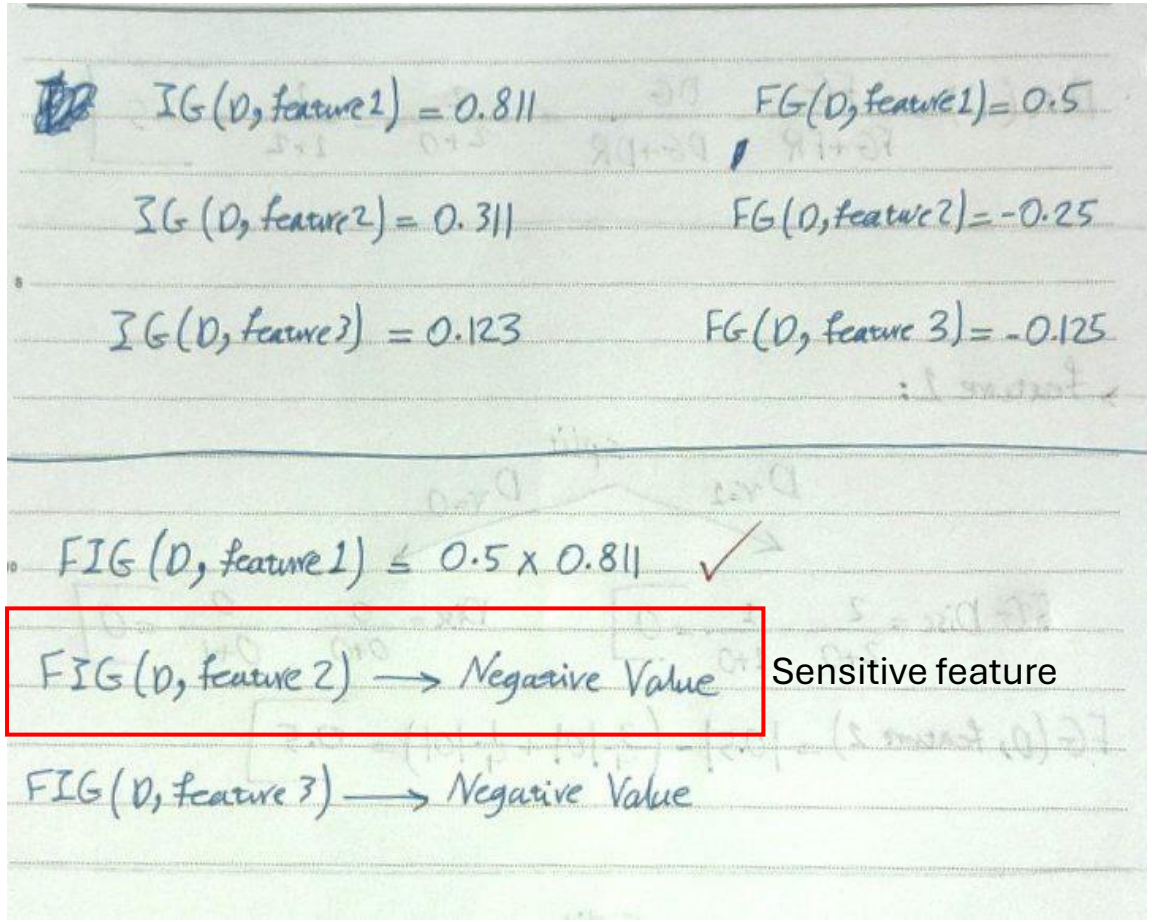
$$FIG(D,A) = \begin{cases} IG(D,A) & ,if FG(D,A) = 0 \\ IG(D,A) \times FG(D,A) & ,otherwise \end{cases}$$

Our goal is to create subtrees that results in a **reduction** in **impurity** and **unfairness**. In the other words, our goal is to **maximize** the **FIG** at each step.

3) Fair Information Gain (FIG)

dataIndex	feature1	feature2	feature3	classLabel
1	1	1	1	1
2	1	0	0	1
3	1	0	1	1
4	0	1	1	0

In the toy dataset, we assume that feature2 is a **sensitive feature.** We use the FIG formula for each feature to select the best feature to split the tree:



Sensitive feature

We select the **feature1** to split the tree at this step, since it has the highest **FIG value.**

4) Python Implementation

At the beginning I tried to implement the tree using scikit-learn library and then override the default splitting criteria of the library and replace it with FIG. It was unsuccessful so I decided to use a custom Decision Tree implementation. The **custom_DT.py** script is a custom implementation that uses FIG metric to split the tree at each step.

```python
class CustomFairDecisionTreeClassifier:
    def __init__(self, max_depth=None):
        self.root = CustomDecisionTreeNode(max_depth=max_depth)

    def fit(self, X, y, protected_indices):
        self.root.fit(X, y, protected_indices)

    def predict(self, X):
        return np.array([self.root.predict(x) for x in X])
```

```python
class CustomDecisionTreeNode:
    def __init__(self, depth=0, max_depth=None):
        self.depth = depth
        self.max_depth = max_depth
        self.feature_index = None
        self.threshold = None
        self.left = None
        self.right = None
        self.is_leaf = False
        self.prediction = None
```

```python
    def predict(self, x):
        if self.is_leaf:
            return self.prediction
        if x[self.feature_index] <=
    self.threshold:
            return self.left.predict(x)
        else:
            return self.right.predict(x)
```

```python
def fit(self, X, y, protected_indices):
    if self.depth == self.max_depth or len(np.unique(y)) == 1:
        self.is_leaf = True
        self.prediction = np.argmax(np.bincount(y))
        return

    best_fig = -np.inf
    best_feature_index = None
    best_threshold = None

    for feature_index in range(X.shape[1]):
        if feature_index in protected_indices:
            continue  # Skip protected attributes

        threshold = np.median(X[:, feature_index])
        fig = fair_information_gain(X, y, feature_index, protected_indices)

        if fig > best_fig:
            best_fig = fig
            best_feature_index = feature_index
            best_threshold = threshold

    if best_feature_index is None:
        self.is_leaf = True
        self.prediction = np.argmax(np.bincount(y))
        return

    self.feature_index = best_feature_index
    self.threshold = best_threshold

    left_indices = X[:, self.feature_index] <= self.threshold
    right_indices = ~left_indices

    self.left = CustomDecisionTreeNode(depth=self.depth + 1, max_depth=self.max_depth)
    self.right = CustomDecisionTreeNode(depth=self.depth + 1, max_depth=self.max_depth)

    self.left.fit(X[left_indices], y[left_indices], protected_indices)
    self.right.fit(X[right_indices], y[right_indices], protected_indices)
```

4) Python Implementation

The main contribution here is the implementation of a function that calculates FIG values of each feature at each step:

```
def fair_information_gain(X, y, feature_index, protected_indices):
    left_indices = X[:, feature_index] <= np.median(X[:, feature_index])
    right_indices = ~left_indices

    y_left, y_right = y[left_indices], y[right_indices]

    # Calculate IG
    ig = information_gain(y, y_left, y_right)

    # If no protected indices are provided, return standard Information Gain
    if not protected_indices:
        return ig

    # Calculate Fairness Gain (FG) if protected indices are provided
    sensitive_left = X[left_indices][:, protected_indices]
    sensitive_right = X[right_indices][:, protected_indices]

    disc_before = np.mean([abs(discrimination_score(y, X[:, i])) for i in protected_indices])
    disc_left = np.mean([abs(discrimination_score(y_left, sensitive_left[:, i])) for i in
range(len(protected_indices))])
    disc_right = np.mean([abs(discrimination_score(y_right, sensitive_right[:, i])) for i in
range(len(protected_indices))])

    fg = disc_before - (len(y_left) / len(y)) * disc_left - (len(y_right) / len(y)) * disc_right

    # Calculate FIG
    if fg == 0:
        return ig
    return ig * fg
```

```
def discrimination_score(y, sensitive_attribute):
    deprived_positive = np.sum((y == 1) & (sensitive_attribute == 0))
    deprived_total = np.sum(sensitive_attribute == 0)
    favored_positive = np.sum((y == 1) & (sensitive_attribute == 1))
    favored_total = np.sum(sensitive_attribute == 1)

    if deprived_total == 0 or favored_total == 0:
        return 0  # Avoid division by zero

    disc_deprived = deprived_positive / deprived_total
    disc_favored = favored_positive / favored_total
    return disc_favored - disc_deprived
```

```
def calculate_entropy(y):
    class_counts = np.bincount(y)
    probabilities = class_counts / len(y)
    return -np.sum([p * np.log2(p) for p in probabilities if p > 0])


def information_gain(y, y_left, y_right):
    entropy_before = calculate_entropy(y)
    entropy_after = (len(y_left) / len(y)) * calculate_entropy(y_left) + (len(y_right) /
len(y)) * calculate_entropy(
        y_right)
    return entropy_before - entropy_after
```

5) Experimental Analysis

I used the adult income dataset, which includes **sensitive features** such as **Sex.** We preprocess the data first:

```
if __name__ == '__main__':
    #
=============================================================================================================
===
    # ==================================== Preprocess the dataset
    #
=============================================================================================================
===
    file_path = "adult.data"
    column_names = [
        "age", "workclass", "fnlwgt", "education", "education-num", "marital-status",
        "occupation", "relationship", "race", "sex", "capital-gain", "capital-loss",
        "hours-per-week", "native-country", "income"
    ]

    adult = pd.read_csv(file_path, names=column_names, na_values=" ?", skipinitialspace=True)
    adult = adult.dropna()

    # Encode categorical variables
    le = LabelEncoder()
    for col in adult.columns:
        if adult[col].dtype == 'object':
            adult[col] = le.fit_transform(adult[col])

    # Split features and target
    X = adult.drop("income", axis=1)
    y = adult["income"]

    # Split the data into training and testing sets
    X_train, X_test, y_train, y_test = train_test_split(X.values, y.values, test_size=0.3, random_state=42)
```

5) Experimental Analysis

Finally we apply the algorithm to the dataset, once using FIG and selecting a sensitive feature, once using the traditional IG:

```
# ============================================================================
# ============================================================ Fair DT With FIG
# ============================================================================
protected_attributes = ["sex"]
protected_indices = [X.columns.get_loc(attr) for attr in protected_attributes]

custom_clf = CustomFairDecisionTreeClassifier(max_depth=5)
custom_clf.fit(X_train, y_train, protected_indices)
y_pred_custom = custom_clf.predict(X_test)

# Metrics
accuracy_custom = accuracy_score(y_test, y_pred_custom)
precision_custom = precision_score(y_test, y_pred_custom)
recall_custom = recall_score(y_test, y_pred_custom)
f1_custom = f1_score(y_test, y_pred_custom)
cm_custom = confusion_matrix(y_test, y_pred_custom)

# Evaluate the model
print("Fair Decision Tree Accuracy:", accuracy_custom)
print("Fair Decision Tree Precision:", precision_custom)
print("Fair Decision Tree Recall:", recall_custom)
print("Fair Decision Tree F1-score:", f1_custom)
print("Fair Decision Tree Confusion Matrix:\n", cm_custom)
```

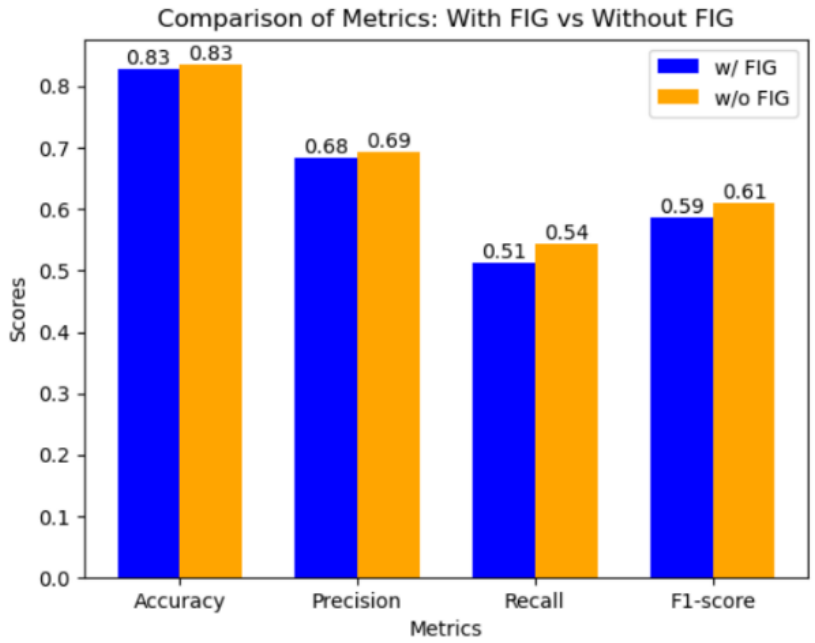
```
# ============================================================================
# ========================================================= Traditional DT Without FIG
# ============================================================================
# index of the protected (sensitive) attribute
protected_attributes = []
protected_indices = [X.columns.get_loc(attr) for attr in protected_attributes]

traditional_clf = CustomFairDecisionTreeClassifier(max_depth=5)
traditional_clf.fit(X_train, y_train, protected_indices)
y_pred = traditional_clf.predict(X_test)

# Metrics
accuracy_traditional = accuracy_score(y_test, y_pred)
precision_traditional = precision_score(y_test, y_pred)
recall_traditional = recall_score(y_test, y_pred)
f1_traditional = f1_score(y_test, y_pred)
cm_traditional = confusion_matrix(y_test, y_pred)

# Evaluate the model
print("Traditional Decision Tree Accuracy:", accuracy_traditional)
print("Traditional Decision Tree Precision:", precision_traditional)
print("Traditional Decision Tree Recall:", recall_traditional)
print("Traditional Decision Tree F1-score:", f1_traditional)
print("Traditional Decision Tree Confusion Matrix:\n", cm_traditional)
```

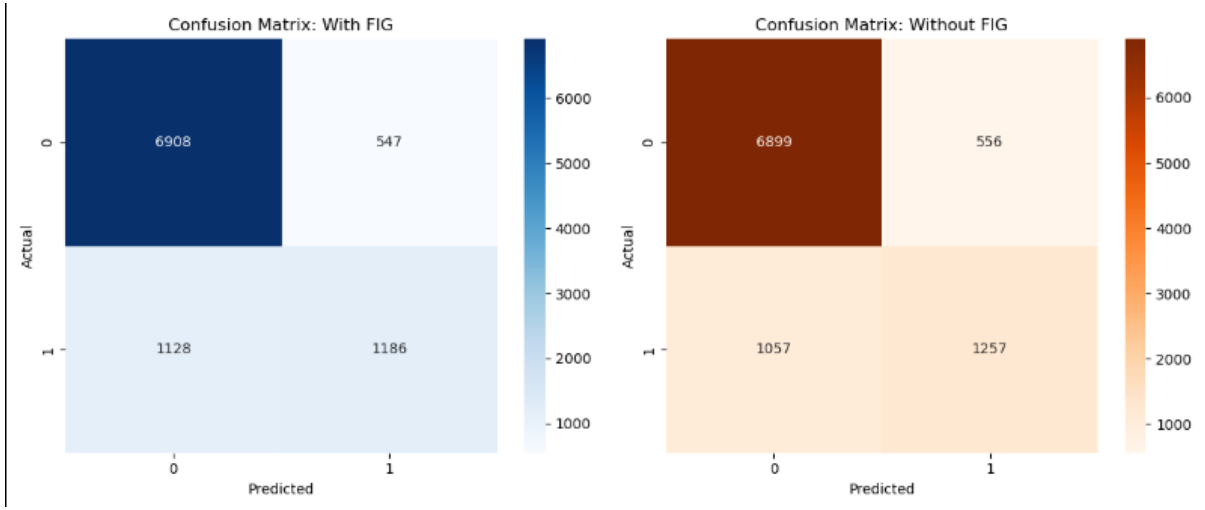
5) Experimental Analysis

These results show that the performance of the model decreases slightly when FIG is used, in comparison with the traditional decision tree implementation.

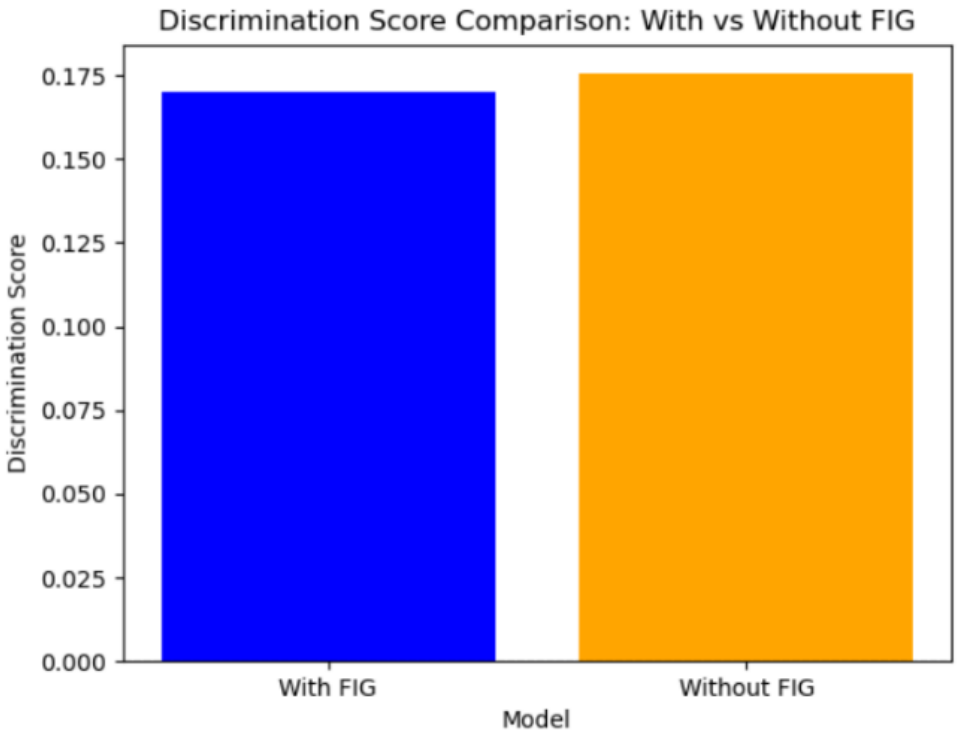
```
Traditional Decision Tree Accuracy: 0.8348858634455932
Traditional Decision Tree Precision: 0.6933259790402647
Traditional Decision Tree Recall: 0.5432152117545376
Traditional Decision Tree F1-score: 0.6091591955415556
Traditional Decision Tree Confusion Matrix:
 [[6899  556]
 [1057 1257]]
```

```
Fair Decision Tree Accuracy: 0.8285392568328386
Fair Decision Tree Precision: 0.6843623773802654
Fair Decision Tree Recall: 0.5125324114088159
Fair Decision Tree F1-score: 0.5861131702495675
Fair Decision Tree Confusion Matrix:
 [[6908  547]
 [1128 1186]]
```





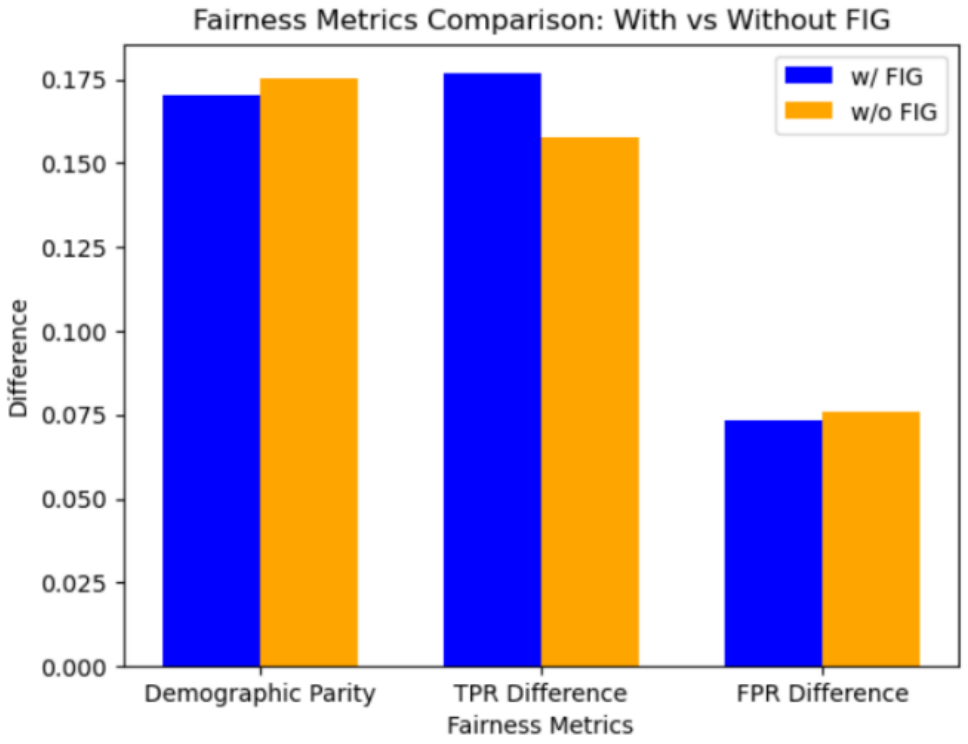
5) Experimental Analysis




```
Discrimination Score for With FIG: 0.17018847442931223
Discrimination Score for Without FIG: 0.17550851930674338
```

A decrease in the discrimination score when using FIG suggests that the implementation of this fairness-enhancing technique has had a positive impact on reducing bias in the model. While the discrimination is still present (as indicated by the positive value), the extent of the bias has been reduced. This decrease indicates that FIG is somewhat effective in narrowing the gap between the probabilities of the favored and deprived groups receiving positive outcomes. It shows progress toward improving fairness in your model's predictions.

5) Experimental Analysis



For fairness evaluation, I considered these metrics:

Demographic Parity:
The differences in demographic parity are quite close for both cases, with FIG showing a slightly lower value. This indicates that incorporating FIG has a marginal impact on ensuring that different demographic groups receive positive outcomes at similar rates. FIG may slightly reduce the disparity in this metric, improving equity.

True Positive Rate (TPR) Difference:
The TPR difference is higher with FIG compared to without. This suggests that FIG might improve true positive predictions for the favored group but could inadvertently increase the disparity between groups in their ability to achieve positive outcomes. A higher TPR difference indicates that fairness may not be perfectly balanced in terms of true positives.

False Positive Rate (FPR) Difference:
For FPR differences, FIG achieves a slightly better (lower) result compared to without FIG. This indicates that the model with FIG better controls disparities in incorrect positive predictions between groups, contributing positively to fairness in this aspect.