Load Dataset

In [1]:
```
import pandas as pd
import numpy as np
from sklearn.model_selection import train_test_split
from sklearn.preprocessing import MinMaxScaler
from sklearn.metrics import accuracy_score, balanced_accuracy_score
import time
```

In [2]:
```
df = pd.read_csv("https://github.com/propublica/compas-analysis/raw/master/compas-scores-two-years.csv")
```

In [3]:
```
data = df[df['race'].isin(['Caucasian', 'African-American'])].copy()
data.loc[:, 'race_binary'] = data['race'].apply(lambda x: 1 if x == 'Caucasian' else 0)
data = data.drop('race', axis=1)
data = data.rename(columns={'race_binary': 'race'})
```

In [4]:
```
data
```

Out[4]:

	id	name	first	last	compas_screening_date	sex	dob	age	age_cat	juv_fel_count	...	v_score_text	v_screening_date	in_custody	out_custody	priors_count.1
1	3	kevon dixon	kevon	dixon	2013-01-27	Male	1982-01-22	34	25 - 45	0	...	Low	2013-01-27	2013-01-26	2013-02-05	0
2	4	ed philo	ed	philo	2013-04-14	Male	1991-05-14	24	Less than 25	0	...	Low	2013-04-14	2013-06-16	2013-06-16	4
3	5	marcu brown	marcu	brown	2013-01-13	Male	1993-01-21	23	Less than 25	0	...	Medium	2013-01-13	NaN	NaN	1
4	6	edward riddle	edward	riddle	2014-02-19	Male	1973-07-23	41	25 - 45	0	...	Low	2014-02-19	2014-03-31	2014-04-18	14
8	10	elizabeth thieme	elizabeth	thieme	2014-03-16	Female	1976-06-03	39	25 - 45	0	...	Low	2014-03-16	2014-03-15	2014-03-18	0
...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...
7207	10994	jarred payne	jarred	payne	2014-05-10	Male	1985-07-31	30	25 - 45	0	...	Low	2014-05-10	2015-10-22	2015-10-22	0
7208	10995	raheem smith	raheem	smith	2013-10-20	Male	1995-06-28	20	Less than 25	0	...	High	2013-10-20	2014-04-07	2014-04-27	0
7209	10996	steven butler	steven	butler	2013-11-23	Male	1992-07-17	23	Less than 25	0	...	Medium	2013-11-23	2013-11-22	2013-11-24	0
7210	10997	malcolm simmons	malcolm	simmons	2014-02-01	Male	1993-03-25	23	Less than 25	0	...	Medium	2014-02-01	2014-01-31	2014-02-02	0
7212	11000	farrah jean	farrah	jean	2014-03-09	Female	1982-11-17	33	25 - 45	0	...	Low	2014-03-09	2014-03-08	2014-03-09	3

6150 rows × 53 columns

Generate X Y and S

chosen X:

1. age
2. sex, 1 for Male and 0 for Female
3. juv_fel_count: The number of juvenile felony charges can indicate a history of serious criminal behavior during adolescence, which could be a predictor of recidivism.
4. juv_misd_count: Similarly, the number of juvenile misdemeanor charges can provide information about an individual's early criminal history.
5. juv_other_count: This represents the number of other juvenile offenses that do not fall under the categories of felony or misdemeanor.
6. priors_count: The number of prior offenses an individual has can be a strong indicator of their likelihood to reoffend.
7. c_charge_degree, 1 for M and 0 for F: The degree of the current charge (M for misdemeanor, F for felony) can help determine the seriousness of the current offense and potentially indicate the risk of recidivism.
8. days_b_screening_arrest: The number of days between the arrest and the screening for the current charge can provide context about the individual's recent involvement with the criminal justice system.
9. *is_recid*: This binary feature indicates whether the individual has a history of recidivism, which can be an important predictor of future recidivism.
10. *is_violent_recid*: indicates whether the individual has a history of violent recidivism. A history of violent reoffending can be a strong predictor of future violent recidivism.

In [5]:
```
X = data[['age', 'sex', 'juv_fel_count', 'juv_misd_count', 'juv_other_count', 'priors_count', 'c_charge_degree', 'days_b_screening_arrest', 'is_recid', 'is_violent_recid']]
```

In [6]:
```
# apply categorical feature transformation to numerical
X[X.columns[X.columns.get_loc('sex')]] = X['sex'].apply(lambda x: 1 if x == 'Male' else 0)
X[X.columns[X.columns.get_loc('c_charge_degree')]] = X['c_charge_degree'].apply(lambda x: 1 if x == 'F' else 0)
```

In [7]:
```
# scale features
scaler = MinMaxScaler()
X[['age', 'juv_fel_count', 'juv_misd_count', 'juv_other_count', 'priors_count', 'days_b_screening_arrest']] = scaler.fit_transform(X[['age', 'juv_fel_count', 'juv_misd_count', 'juv_other_count', 'priors_count', 'days_b_screening_arrest']])
```

In [8]:
```
# fill any NA with 0
X = X.fillna(0)
```

In [9]:
```
Y = data['two_year_recid']
S = data['race']
```

In [10]:
```
X
```

Out[10]:

	age	sex	juv_fel_count	juv_misd_count	juv_other_count	priors_count	c_charge_degree	days_b_screening_arrest	is_recid	is_violent_recid
1	0.246154	1	0.0	0.000000	0.000000	0.000000	1	0.280761	1	1
2	0.092308	1	0.0	0.000000	0.000000	0.058824	0.105263	0.280761	1	0
3	0.076923	1	0.0	0.076923	0.000000	0.029316	1	0.000000	0	0
4	0.353846	1	0.0	0.000000	0.000000	0.368421	1	0.280761	1	0
8	0.323077	0	0.0	0.000000	0.000000	0.000000	0	0.280761	0	0
...	...	...	...	...	...	...	...	...	...	...
7207	0.184615	1	0.0	0.000000	0.000000	0.000000	0	0.280761	1	0
7208	0.030769	1	0.0	0.000000	0.000000	0.000000	1	0.280761	0	0
7209	0.076923	1	0.0	0.000000	0.000000	0.000000	1	0.280761	0	0
7210	0.076923	1	0.0	0.000000	0.000000	0.000000	1	0.280761	0	0
7212	0.230769	0	0.0	0.000000	0.000000	0.078947	0	0.280761	1	0

6150 rows × 10 columns

In [11]:
```
Y
```

Out[11]:
```
1       1
2       1
3       0
6       1
8       0
       ..
7207    1
7208    0
7209    0
7210    0
7212    0
Name: two_year_recid, Length: 6150, dtype: int64
```

In [12]:
```
S
```

Out[12]:
```
1       0
2       0
3       0
6       1
8       1
       ..
7207    0
7208    0
7209    0
7210    0
7212    0
Name: race, Length: 6150, dtype: int64
```

In [13]:
```
# Generate train/test set with 80%-20%
```

In [14]:
```
random_state = 42
```

In [15]:
```
X_train, X_test, Y_train, Y_test, S_train, S_test = train_test_split(X, Y, S, test_size=0.2, random_state=random_state, stratify=Y)
```

In [16]:
```
X_train = X_train.reset_index(drop=True)
X_test = X_test.reset_index(drop=True)
Y_train = Y_train.reset_index(drop=True)
Y_test = Y_test.reset_index(drop=True)
S_train = S_train.reset_index(drop=True)
S_test = S_test.reset_index(drop=True)
```

Fairness-aware Classifier with Prejudice Remover Regularizer

1. Define model M(Y|X,S;theta), using sigmoid funciton

In [17]:
```
def sigmoid(x):
    return 1 / (1 + np.exp(-x))
```

In [18]:
```
def M_y_given_x_s(X_S, theta):
    # using sigmoid model
    M = sigmoid(np.dot(X_S, theta))
    return M
```

2. Define the prejudice index R(D; theta)

In [19]:
```
def get_PI(X_S, S, Y, theta):
    M = M_y_given_x_s(X_S, theta)
    # P_y_s[0] save Phat(Y,S): Phat(0, 0), Phat(0, 1),Phat(1, 0),Phat(1, 1)
    P_y_s = np.zeros((2, 2))
    # Phat(Y): Phat(0), Phat(1)
    P_y = np.zeros(2)
    # compute Phats
    for s in [0, 1]:
        s_indx = np.where(S == s)
        P_y_s[:, s] = np.mean(M[s_indx], axis=0)
        P_y += np.mean(M[s_indx], axis=0)
    P_y /= 2

    # PI = sum P(Y,S)log((P(Y,S)/P(S)P(Y)))
    # using P(S) = 0.5
    PI = 0
    for y in [0, 1]:
        for s in [0, 1]:
            PI += P_y_s[y, s] * np.log(P_y_s[y, s] / (P_y[y] * (0.5)))
    return PI
```

3. Define Loss function L(D; theta)

In [20]:
```
def loss(X_S, S, Y, theta, eta, lamb):
    M = M_y_given_x_s(X_S, theta)
    L = -np.sum(Y * np.log(M) + (1 - Y) * np.log(1 - M))
    PI = get_PI(X_S, S, Y, theta)
    regularization = (lamb / 2) * np.sum(theta ** 2)
    loss = L + eta * PI + regularization
    return loss
```

4. Implement optimization to minimize the objective function using gradient method

In [21]:
```
def get_gradients_of_objective_function(X_S, Y, S, theta, eta, lamb):
    # this function calculate derivative of each term in the loss function
    M = M_y_given_x_s(X_S, theta)
    # calculate partial derivative of theta on -L(D;theta)
    # L(D;theta) = sum(log(M(yi|xi,si;theta))) = np.sum(Y * np.log(sigmoid(np.dot(X_S, theta))) + (1 - Y) * np.log(1 - sigmoid(np.dot(X_S, theta))));
    # then dL(theta)/d(theta) = X_S.T * (sigmoid(np.dot(X_S, theta)) - Y)
    dLdtheta = np.dot(X_S.T, M - Y)
    PI = get_PI(X_S, S, Y, theta)
    # calculate partial derivative of theta on eta*R(D,theta)
    detaR_dtheta = np.zeros_like(theta)
    for y in [0, 1]:
        for s in [0, 1]:
            idx_s = np.where(S == s)
            M_y_s = np.mean(M[idx_s], axis=0)
            if np.isscalar(M_y_s):
                continue
            detaR_dtheta += np.dot(X_S[idx_s].T, M_y_s) * np.log(M_y_s[y] / (0.5 * np.sum(M_y_s)))
    detaR_dtheta /= X_shape[0]
    # calculate partial derivative of theta on lambda/2 ||theta||_2^2
    dRegular_dtheta = lamb * theta
    gradients = dLdtheta + eta * detaR_dtheta + dRegular_dtheta
    return gradients
```

In [22]:
```
def fit(X, S, Y, eta, lamb, learning_rate, num_iterations):
    X_S = np.column_stack((X, S))
    # initialized theta with 0
    theta = np.zeros(X_S.shape[1])
    for i in range(num_iterations):
        gradients = get_gradients_of_objective_function(X_S, Y, S, theta, eta, lamb)
        theta -= learning_rate * gradients
    return theta
```

In [23]:
```
def predict(X, S, theta):
    X_S = np.column_stack((X, S))
    M = M_y_given_x_s(X_S, theta)
    return (M > 0.5).astype(int)
```

In [24]:
```
# Train the fairness-aware classifier
start_time_fit = time.time()
theta = fit(X_train.values, S_train.values, Y_train.values, eta=1.0, lamb=1.0, learning_rate=0.01, num_iterations=100)
end_time_fit = time.time()

# Make predictions on the test set
start_time_predict = time.time()
Y_pred = predict(X_test.values, S_test.values, theta)
end_time_predict = time.time()

# Calculate accuracy
print("Time fit:", end_time_fit-start_time_fit)
print("Time Predict:", end_time_predict-start_time_predict)
accuracy = accuracy_score(Y_test, Y_pred)
print("Accuracy:", accuracy)
```

```
Time fit: 0.04455141702297633
Time Predict: 0.0002751350402832031
Accuracy: 0.9390243902439024
```

In [25]:
```
balanced_accuracy = balanced_accuracy_score(Y_test, Y_pred)
print("Balanced Accuracy:", accuracy)
```

```
Balanced Accuracy: 0.9390243902439024
```

In [26]:
```
unique_S = np.unique(S_test)
calibration_difference = {}
for s in unique_S:
    indices = S_test == s
    Y_pred_s = Y_pred[indices]
    Y_test_s = Y_test[indices]
    calibration_s = np.abs(np.mean(Y_pred_s) - np.mean(Y_test_s))
    calibration_difference[s] = calibration_s

print("Calibration difference:")
for s, diff in calibration_difference.items():
    print(f"Class {s}: {diff}")
```

```
Calibration difference:
Class 0: 0.004143644088039796
Class 1: 0.023715415019762875
```

In [27]:
```
def equalized_odds(Y_true, Y_pred, S):
    race = np.unique(S)
    tp = np.zeros(len(race))
    fp = np.zeros(len(race))
    tn = np.zeros(len(race))
    fn = np.zeros(len(race))
    for i, s in enumerate(race):
        idx = np.where(S == s)[0]
        tp[i] = np.sum((Y_true[idx] == 1) & (Y_pred[idx] == 1))
        fp[i] = np.sum((Y_true[idx] == 0) & (Y_pred[idx] == 1))
        tn[i] = np.sum((Y_true[idx] == 0) & (Y_pred[idx] == 0))
        fn[i] = np.sum((Y_true[idx] == 1) & (Y_pred[idx] == 0))
    tpr = tp / (tp + fn)
    fpr = fp / (fp + tn)
    tnr = tn / (tn + fp)
    fnr = fn / (fp + fn)
    return tpr, fpr, tnr, fnr
```

In [28]:
```
tpr, fpr, tnr, fnr = equalized_odds(Y_test, Y_pred, S_test)
print("True positive rate for group 0:", tpr[0])
print("False positive rate for group 0:", fpr[0])
print("True negative rate for group 0:", tnr[0])
print("False negative rate for group 0:", fnr[0])
print("True positive rate for group 1:", tpr[1])
print("False positive rate for group 1:", fpr[1])
print("True negative rate for group 1:", tnr[1])
print("False negative rate for group 1:", fnr[1])
```

```
True positive rate for group 0: 0.9486486486486486
False positive rate for group 0: 0.06214689265536723
True negative rate for group 0: 0.9378531073446328
False negative rate for group 0: 0.05135135135135154
True positive rate for group 1: 0.8866995073891626
False positive rate for group 1: 0.036303630363036306
True negative rate for group 1: 0.9636963696369637
False negative rate for group 1: 0.11330049261083744
```

In [ ]:
```
```

In [ ]:
```
```

In [ ]:
```
```

In [ ]:
```
```

In [ ]:
```
```

In [ ]:
```
```

In [ ]:
```
```

In [ ]:
```
```