In [1]:
```
# Import Required Packages
import numpy as np
import pandas as pd
from sklearn.preprocessing import LabelEncoder
from sklearn.model_selection import train_test_split
from sklearn.metrics import accuracy_score
import math
import operator
import random
import scipy.optimize as optim
```

1. Data Processing

In [2]:
```
# Set a Random Seed
random.seed(1234)
# Import Data
data = pd.read_csv('https://raw.githubusercontent.com/propublica/compas-analy

# Data Cleaning
data = data[(data['race']=='African-American') | (data['race']=='Caucasian')]
data = data[data.columns[data.isna().mean() < 0.5]]
data = data.drop(columns = ['id','name','first', 'last', 'dob','age_cat', 'co
                            'out_custody', 'c_offense_date', 'c_days_from_com
                            'days_b_screening_arrest','screening_date','c_cha
                            'juv_fel_count','juv_misd_count','juv_other_count

# Data Relabel
data['race'].loc[data['race'] == 'African-American'] = 0
data['race'].loc[data['race'] == 'Caucasian'] = 1
data['c_charge_degree'] = LabelEncoder().fit_transform(data['c_charge_degree'
data['score_text'] = LabelEncoder().fit_transform(data['score_text'])
data['v_score_text'] = LabelEncoder().fit_transform(data['v_score_text'])
data['sex'] = LabelEncoder().fit_transform(data['sex'])
```

```
/usr/local/lib/python3.7/dist-packages/pandas/core/indexing.py:1732: SettingWi
thCopyWarning:
A value is trying to be set on a copy of a slice from a DataFrame

See the caveats in the documentation: https://pandas.pydata.org/pandas-docs/st
able/user_guide/indexing.html#returning-a-view-versus-a-copy
  self._setitem_single_block(indexer, value, name)
```

In [3]:
```
# Training & Testing Split
training, testing = train_test_split(data, test_size=0.2, random_state=1234)
training_sensitive = training.loc[data['race'] == 1, data.columns !='two_year
testing_sensitive = testing.loc[data['race'] == 1, data.columns !='two_year_r
training_nonsensitive = training.loc[data['race'] == 0, data.columns !='two_y
testing_nonsensitive = testing.loc[data['race'] == 0, data.columns !='two_yea
ytrain_sensitive = training.loc[data['race'] == 1, data.columns =='two_year_r
ytrain_nonsensitive = training.loc[data['race'] == 0, data.columns =='two_yea
ytesting_sensitive = testing.loc[data['race'] == 1, data.columns =='two_year_
ytesting_nonsensitive = testing.loc[data['race'] == 0, data.columns =='two_ye

# Converting to Array
training_sensitive = np.array(training_sensitive)
training_nonsensitive = np.array(training_nonsensitive)
testing_sensitive = np.array(testing_sensitive)
testing_nonsensitive = np.array(testing_nonsensitive)
ytrain_nonsensitive = np.array(ytrain_nonsensitive)
ytrain_sensitive = np.array(ytrain_sensitive)
ytesting_sensitive = np.array(ytesting_sensitive)
ytesting_nonsensitive = np.array(ytesting_nonsensitive)
```

2. Algorithm Implementation

In [4]:
```
# Helper Function for LFR

def distances(X, v, alpha, N, P, k):
    dists = np.zeros((N, P))
    for i in range(N):
        for p in range(P):
            for j in range(k):
                dists[i, j] += (X[i, p] - v[j, p]) * (X[i, p] - v[j, p]) * al
    return dists

def M_nk(dists, N, k):
    M_nk = np.zeros((N, k))
    exp = np.zeros((N, k))
    denom = np.zeros(N)
    for i in range(N):
        for j in range(k):
            exp[i, j] = np.exp(-1 * dists[i, j])
            denom[i] += exp[i, j]
        for j in range(k):
            if denom[i]:
                M_nk[i, j] = exp[i, j] / denom[i]
            else:
                M_nk[i, j] = exp[i, j] / 1e-6
    return M_nk

def M_k(M_nk, N, k):
```

```
    M_k = np.zeros(k)
    for j in range(k):
        for i in range(N):
            M_k[j] += M_nk[i, j]
        M_k[j] /= N
    return M_k


def x_n_hat(X, M_nk, v, N, P, k):
    x_n_hat = np.zeros((N, P))
    L_x = 0.0
    for i in range(N):
        for p in range(P):
            for j in range(k):
                x_n_hat[i, p] += M_nk[i, j] * v[j, p]
            L_x += (X[i, p] - x_n_hat[i, p]) * (X[i, p] - x_n_hat[i, p])
    return x_n_hat, L_x


def yhat(M_nk, y, w, N, k):
    yhat = np.zeros(N)
    L_y = 0.0
    for i in range(N):
        for j in range(k):
            yhat[i] += M_nk[i, j] * w[j]
        yhat[i] = 1e-6 if yhat[i] <= 0 else yhat[i]
        yhat[i] = 0.999 if yhat[i] >= 1 else yhat[i]
        L_y += -1 * y[i] * np.log(yhat[i]) - (1.0 - y[i]) * np.log(1.0 - yhat
    return yhat, L_y


def LFR(params, data_sensitive, data_nonsensitive, y_sensitive,
        y_nonsensitive,  k=10, A_x = 1e-4, A_y = 0.1, A_z = 1000, results=0):

    LFR.iters += 1
    Ns, P = data_sensitive.shape
    Nns, _ = data_nonsensitive.shape

    alpha0 = params[:P]
    alpha1 = params[P : 2 * P]
    w = params[2 * P : (2 * P) + k]
    v = np.matrix(params[(2 * P) + k:]).reshape((k, P))

    dists_sensitive = distances(data_sensitive, v, alpha1, Ns, P, k)
    dists_nonsensitive = distances(data_nonsensitive, v, alpha0, Nns, P, k)

    M_nk_sensitive = M_nk(dists_sensitive, Ns, k)
    M_nk_nonsensitive = M_nk(dists_nonsensitive, Nns, k)

    M_k_sensitive = M_k(M_nk_sensitive, Ns, k)
    M_k_nonsensitive = M_k(M_nk_nonsensitive, Nns, k)

    L_z = 0.0
```

```
    for j in range(k):
        L_z += abs(M_k_sensitive[j] - M_k_nonsensitive[j])

    x_n_hat_sensitive, L_x1 = x_n_hat(data_sensitive, M_nk_sensitive, v, Ns, 
    x_n_hat_nonsensitive, L_x2 = x_n_hat(data_nonsensitive, M_nk_nonsensitive
    L_x = L_x1 + L_x2

    yhat_sensitive, L_y1 = yhat(M_nk_sensitive, y_sensitive, w, Ns, k)
    yhat_nonsensitive, L_y2 = yhat(M_nk_nonsensitive, y_nonsensitive, w, Nns,
    L_y = L_y1 + L_y2

    criterion = A_x * L_x + A_y * L_y + A_z * L_z

    if LFR.iters % 250 == 0:
        print(LFR.iters, criterion)

    if results:
        return yhat_sensitive, yhat_nonsensitive, M_nk_sensitive, M_nk_nonsen
    else:
        return criterion
LFR.iters = 0
```

In [5]:

```
k = 10
rez = np.random.uniform(size=training_sensitive.shape[1] * 2 + k + training_s

bnd = []
for i, k2 in enumerate(rez):
    if i < training_sensitive.shape[1] * 2 or i >= training_sensitive.shape[1
        bnd.append((None, None))
    else:
        bnd.append((0, 1))
```

In [6]:

```
LFR(rez, training_sensitive, training_nonsensitive, ytrain_sensitive,
    ytrain_nonsensitive, k, 1e-4, 0.1, 1000, 0)
```

Out[6]:

```
array([1831.22692953])
```

In [7]:

```
# Optimization Process
LFR.iters = 0
rez = optim.fmin_l_bfgs_b(LFR, x0=rez,
                          args=(training_sensitive, training_nonsensitive,
                                ytrain_sensitive, ytrain_nonsensitive, k, 1e-
                                0.1, 1000, 0),
                          bounds = bnd, approx_grad=True, maxfun = 1500, maxi
```

```
250 [1939.91176207]
500 [1640.27065909]
750 [1639.83908276]
1000 [1642.92002674]
1250 [1639.74024451]
1500 [1639.96148832]
1750 [1640.11085966]
2000 [1639.65001765]
2250 [1639.64238655]
2500 [1639.63541975]
2750 [1639.55319808]
```

In [13]:

```
# save the optimization result

result=rez[0]
result
```

Out[13]:
```
array([6.70091108e-01, 2.95578493e-01, 4.02644714e-01, 5.84007899e-01,
       8.45709474e-01, 4.53519524e-01, 3.93401377e-01, 2.76994307e-02,
       4.48697452e-01, 4.28644863e-01, 2.24176661e-01, 2.76653586e-01,
       7.64609888e-01, 4.60166201e-02, 3.17763350e-01, 4.65585835e-01,
       7.15328919e-01, 6.71084273e-01, 4.89153797e-01, 3.62430093e-01,
       3.74555654e-01, 2.43800856e-01, 4.91611255e-01, 5.03546225e-01,
       9.26721327e-01, 7.63483876e-01, 9.74476896e-01, 3.98249910e-04,
       6.27381053e-01, 7.67667339e-01, 8.38144640e-02, 3.31638863e-01,
       6.08474306e-01, 4.38187712e-01, 5.90013940e-01, 2.96009793e-01,
       3.85191510e-01, 4.61810466e-01, 8.40535958e-01, 1.28944016e-01,
       2.06800754e-01, 2.02462377e-01, 1.00535352e-01, 3.10745592e-01,
       9.69214048e-01, 5.44370229e-02, 4.37725272e-01, 7.41628138e-01,
       3.15780033e-01, 3.83167708e-01, 7.78999389e-01, 8.66701914e-01,
       1.17426276e-01, 5.09855786e-01, 7.62845655e-01, 3.60444685e-01,
       1.10393794e-03, 1.39422375e-01, 9.37070949e-01, 8.77644054e-01,
       6.46511279e-03, 6.03564172e-02, 5.76188060e-01, 6.67306295e-01,
       4.93775515e-01, 1.10990318e-01, 4.01574507e-02, 7.82897657e-01,
       2.11832602e-01, 8.07801750e-02, 1.67736663e-01, 8.70494854e-02,
       6.23381447e-01, 6.48570914e-01, 7.31638374e-01, 1.25121554e-01,
       4.06627963e-01, 7.25553623e-02, 2.85356262e-01, 5.42595000e-01,
       1.59224695e-01, 2.16676704e-01, 7.75080035e-01, 6.83495356e-01,
       8.93450393e-01, 8.54115158e-02, 1.63615225e-01, 8.24807103e-01,
       4.06209851e-01, 1.54049179e-01, 7.69977945e-02, 6.30832148e-01,
       6.21363849e-01, 6.59155904e-01, 2.15769784e-01, 9.39535564e-01,
       6.10593216e-01, 5.81643261e-02, 8.51675857e-01, 7.84293842e-01,
       1.17028253e-01, 5.03171495e-02, 1.10303550e+00, 3.24087391e-01,
       3.69622675e-01, 8.52089380e-01, 9.93398783e-01, 7.64578443e-03,
       2.49056986e-01, 2.54673644e-01, 3.04365826e-01, 2.26385988e-01,
       8.75704161e-01, 8.32076050e-01, 5.96870455e-02, 4.59827165e-01,
       6.80198768e-01, 4.13128484e-01, 3.31551303e-01, 7.01439576e-01,
       1.05677955e-01, 2.63826916e-01, 8.90145081e-01, 4.29201492e-01,
       7.42756576e-01, 5.16606564e-01, 5.77706960e-01, 2.37015026e-01,
       3.16659472e-02, 1.61388563e-01, 9.72699053e-01, 4.77506457e-01,
       3.24279981e-02, 4.91622043e-01, 3.39817124e-01, 9.32533806e-01,
       3.28923152e-01, 7.32547910e-02, 9.38363166e-03, 3.70017631e-01,
       9.11131699e-01, 2.86810466e-01, 5.76008301e-01, 3.20165460e-01,
       9.89643470e-01, 9.68879949e-01, 5.92104910e-01, 9.76350945e-01,
       1.69921633e-01, 9.70223743e-02, 5.40547584e-01, 2.34766284e-01,
       5.47802041e-01, 8.05173804e-01, 6.92225074e-01, 5.15707512e-01,
       7.55034447e-01, 5.22670029e-02, 2.09619573e-01, 2.06340995e-01,
       9.49715564e-01, 2.29138765e-01, 6.98979371e-01, 4.80926449e-02,
       9.24294670e-01, 1.54337426e-01])
```

3. Evaluation

In [9]:
```
# Euclidean Distance Function
def euclideanDistance(instance1, instance2):
    distance = 0
    for x in range(len(instance1)):
        distance += pow((instance1[x] - instance2[x]), 2)
    return math.sqrt(distance)
```

```
# KNN Function
def getKNeighbors(trainingSet, testInstance, k):
    distances = []
    for x in range(len(trainingSet)):
        dist = euclideanDistance(testInstance, trainingSet[x])
        distances.append(dist)
    distances = np.array(distances)
    neighbors = distances.argsort()[0:k]
    return neighbors

# Individual Fairness Evaluation
def individualfair(data, yhat):
    length = len(yhat)
    result = 0
    for i in range(length):
        neighbors = getKNeighbors(np.concatenate((data[:i],data[i+1:]), axis
        result += abs(yhat[i] - yhat[neighbors[0]] -  yhat[neighbors[1]] -  y

    result = 1 - result/(length * 3)
    return result


# Write A Evaluation Function
def LFR_metric(params, data_sensitive, data_nonsensitive, y_sensitive,
        y_nonsensitive,  k=10, A_x = 1e-4, A_y = 0.1, A_z = 1000):

    Ns, P = data_sensitive.shape
    Nns, _ = data_nonsensitive.shape

    alpha0 = params[:P]
    alpha1 = params[P : 2 * P]
    w = params[2 * P : (2 * P) + k]
    v = np.matrix(params[(2 * P) + k:]).reshape((k, P))

    dists_sensitive = distances(data_sensitive, v, alpha1, Ns, P, k)
    dists_nonsensitive = distances(data_nonsensitive, v, alpha0, Nns, P, k)

    M_nk_sensitive = M_nk(dists_sensitive, Ns, k)
    M_nk_nonsensitive = M_nk(dists_nonsensitive, Nns, k)

    M_k_sensitive = M_k(M_nk_sensitive, Ns, k)
    M_k_nonsensitive = M_k(M_nk_nonsensitive, Nns, k)

    L_z = 0.0
    for j in range(k):
        L_z += abs(M_k_sensitive[j] - M_k_nonsensitive[j])

    x_n_hat_sensitive, L_x1 = x_n_hat(data_sensitive, M_nk_sensitive, v, Ns,
    x_n_hat_nonsensitive, L_x2 = x_n_hat(data_nonsensitive, M_nk_nonsensitive
    L_x = L_x1 + L_x2
```

```
    yhat_sensitive, L_y1 = yhat(M_nk_sensitive, y_sensitive, w, Ns, k)
    yhat_nonsensitive, L_y2 = yhat(M_nk_nonsensitive, y_nonsensitive, w, Nns,
    L_y = L_y1 + L_y2

    data_all = np.concatenate((data_sensitive, data_nonsensitive), axis = 0)
    yhat_all = np.concatenate((yhat_sensitive, yhat_nonsensitive), axis = 0)
    individual_fairness = individualfair(data_all, yhat_all)

    criterion = A_x * L_x + A_y * L_y + A_z * L_z

    sen_acc = accuracy_score(y_sensitive, (yhat_sensitive >= 0.5))
    nonsen_acc = accuracy_score(y_nonsensitive, (yhat_nonsensitive >= 0.5))
    tot_acc = (Ns * sen_acc + Nns * nonsen_acc) / (Ns + Nns)

    calibration = abs(sen_acc - nonsen_acc)

    return criterion, sen_acc, nonsen_acc, tot_acc, calibration, L_x, L_y, L_
```

In [16]:

```
metrics = LFR_metric(result, testing_sensitive, testing_nonsensitive, ytestin
```

In [17]:

```
print("LOSS: {}, Sensitive Accuracy: {}, Nonsensitive Accuracy: {},  Total Ac
            metrics[0], metrics[1], metrics[2], metrics[3], metrics[4], metri
```

```
LOSS: [484.34520659], Sensitive Accuracy: 0.6068548387096774, Nonsensitive Acc
uracy: 0.4891008174386921,  Total Accuracy: 0.5365853658536586,  Clibration: 0
.11775402127098528, Individual Fairness: 0.8118169264545848
```