A6-LM

December 8, 2021

1 Project 4 Group 7 Handling Conditional Discrimination Algorithm

```python
[14]: # Import required packages
      import numpy as np
      import pandas as pd

      import time
      import cv2
      import matplotlib.pyplot as plt
      from sklearn.metrics import classification_report
      from sklearn.linear_model import LogisticRegression

      import tensorflow as tf
      from tensorflow.keras.callbacks import ModelCheckpoint, ReduceLROnPlateau
      from tensorflow.keras.layers import Input, Dense, BatchNormalization, Flatten,␣
      ↪MaxPooling2D, Activation, GlobalMaxPool2D, GlobalAvgPool2D, Concatenate,␣
      ↪Multiply, Dropout, Subtract
      from tensorflow.keras.models import Model, Sequential
      from tensorflow.keras.layers import Conv2D, MaxPooling2D
      from tensorflow.keras.layers import Activation, Dropout, Flatten, Dense
      from tensorflow.keras.preprocessing.image import ImageDataGenerator,␣
      ↪array_to_img, img_to_array, load_img
      from tensorflow.keras.optimizers import SGD, Adam, RMSprop, Nadam
      from sklearn.utils import shuffle
      from sklearn.model_selection import train_test_split

      from functools import partial
      from pprint import pprint
      from hyperopt import fmin, hp, space_eval, tpe, STATUS_OK, Trials
      from hyperopt.pyll import scope, stochastic
      from plotly import express as px
      from plotly import graph_objects as go
      from plotly import offline as pyo
      from sklearn.datasets import load_boston
      #from sklearn.ensemble import GradientBoostingRegressor, RandomForestRegressor
      from sklearn.ensemble import GradientBoostingClassifier, RandomForestClassifier
```

```
from sklearn.metrics import make_scorer, mean_squared_error, log_loss
from sklearn.model_selection import cross_val_score, KFold
from sklearn.utils import check_random_state
pyo.init_notebook_mode()
```

[2]:
```
print(f"This notebook uses TensorFlow Version {tf.__version__}")
print("And Python Version:")
!python --version
print("Num GPUs Available: ", len(tf.config.list_physical_devices('GPU')))
```

```
This notebook uses TensorFlow Version 2.5.0
And Python Version:
Python 3.8.8
Num GPUs Available:  1
```

1.0.1 After importing the data, we can remove unnceccary columns such as dates and focus on the person's more easily quantifiable features

[3]:
```
#Import the data and filter out unneccessary columns and rows that aren't of␣
↪interest (race not A-A or Cau)
df = pd.read_csv('../data/compas-scores-two-years.csv')
df = df.filter(items=['sex','age_cat','race','juv_fel_count','decile_score',
                    ␣
↪'juv_misd_count','juv_other_count','priors_count','c_charge_degree','two_year_recid'])
df = df[(df.race=='African-American') | (df.race=='Caucasian')]
df
```

[3]:

	sex	age_cat	race	juv_fel_count	decile_score	\
1	Male	25 - 45	African-American	0	3	
2	Male	Less than 25	African-American	0	4	
3	Male	Less than 25	African-American	0	8	
6	Male	25 - 45	Caucasian	0	6	
8	Female	25 - 45	Caucasian	0	1	
...	...	...	...	...	...	
7207	Male	25 - 45	African-American	0	2	
7208	Male	Less than 25	African-American	0	9	
7209	Male	Less than 25	African-American	0	7	
7210	Male	Less than 25	African-American	0	3	
7212	Female	25 - 45	African-American	0	2	

	juv_misd_count	juv_other_count	priors_count	c_charge_degree	\
1	0	0	0	F	
2	0	1	4	F	
3	1	0	1	F	
6	0	0	14	F	
8	0	0	0	M	
...	...	...	...	...	

```
7207                0               0               0               M
7208                0               0               0               F
7209                0               0               0               F
7210                0               0               0               F
7212                0               0               3               M


        two_year_recid
1                    1
2                    1
3                    0
6                    1
8                    0
...                ...
7207                 1
7208                 0
7209                 0
7210                 0
7212                 0

[6150 rows x 10 columns]
```

`[4]:`
```python
print(df.race.value_counts())
df.isnull().sum().sum()
```

```
African-American    3696
Caucasian           2454
Name: race, dtype: int64
```

`[4]:` 0

`[5]:`
```python
df.nunique()
```

`[5]:`
```
sex                  2
age_cat              3
race                 2
juv_fel_count       10
decile_score        10
juv_misd_count      10
juv_other_count      9
priors_count        37
c_charge_degree      2
two_year_recid       2
dtype: int64
```

1.0.2 Dummy variable creation

The previous dataframe with categorical data is not acceptable when optimizing hyperparameter with fmin later on, so we create dummy variables for the age categories, gender, race, and charge

degree next.

```
[6]: df_dummy = df.copy()
     df_dummy['sex'] = (df['sex'].values == 'Female').astype(int)
     df_dummy['race'] = (df['race'].values == 'African-American').astype(int)
     df_dummy = pd.concat([df_dummy, pd.get_dummies(df.age_cat, drop_first=True)],␣
      →axis=1)
     df_dummy = df_dummy.drop('age_cat', axis=1)
     df_dummy = df_dummy.rename(columns={"Greater than 45": "gt45", "Less than 25":␣
      →"lt25"})
     df_dummy['c_charge_degree'] = (df['c_charge_degree'].values == 'F').astype(int)
     df_dummy
```

```
[6]:        sex  race  juv_fel_count  decile_score  juv_misd_count  juv_other_count  \
     1        0     1              0             3               0                0
     2        0     1              0             4               0                1
     3        0     1              0             8               1                0
     6        0     0              0             6               0                0
     8        1     0              0             1               0                0
     ...     ...   ...            ...           ...             ...              ...
     7207     0     1              0             2               0                0
     7208     0     1              0             9               0                0
     7209     0     1              0             7               0                0
     7210     0     1              0             3               0                0
     7212     1     1              0             2               0                0

            priors_count  c_charge_degree  two_year_recid  gt45  lt25
     1                  0                1               1     0     0
     2                  4                1               1     0     1
     3                  1                1               0     0     1
     6                 14                1               1     0     0
     8                  0                0               0     0     0
     ...              ...              ...             ...   ...   ...
     7207               0                0               1     0     0
     7208               0                1               0     0     1
     7209               0                1               0     0     1
     7210               0                1               0     0     1
     7212               3                0               0     0     0

     [6150 rows x 11 columns]
```

```
[50]: #Our X, s, e, and y parameters for the fairness algorithims plus all the␣
      →features for the initial model
      FEATURES =␣
       →['sex','gt45','lt25','juv_fel_count','decile_score','juv_misd_count','juv_other_count','pri
      RACE = "race"
      DEGREE = "c_charge_degree"
```

4

```
TWO_YEAR_RECID = "two_year_recid"
ALL_FEATURES =␣
 ↪['sex','gt45','lt25','juv_fel_count','decile_score','juv_misd_count','juv_other_count',
             'priors_count','race','c_charge_degree']
```

1.1 Choose one boosting and one averaging algorithm to optimize parameters

By diversifying the learning algorithm, number of estimators, learning rate, and max depth(for Gradient Boosting), we can make sure we choose a set of parameters that net us the best performance.

```python
[19]: # Define constant strings that we will use as keys in the "search space"␣
      ↪dictionary below.
      # This helps reduce typos when spelling out the same string repeatedly
      GRADIENT_BOOSTING_CLASSIFIER = "gradient_boosting_classifier"
      KWARGS = "kwargs"
      LEARNING_RATE = "learning_rate"
      #LINEAR_REGRESSION = "linear_regression"
      MAX_DEPTH = "max_depth"
      MODEL = "model"
      MODEL_CHOICE = "model_choice"
      NORMALIZE = "normalize"
      N_ESTIMATORS = "n_estimators"
      RANDOM_FOREST_CLASSIFIER = "random_forest_classifier"
      RANDOM_STATE = "random_state"

      # Declare the search space for the random forest classifier model.
      random_forest_classifier = {
          MODEL: RANDOM_FOREST_CLASSIFIER,
          # The model parameters are a separate dictionary so that we can feed the␣
      ↪parameters to the model
          # via dictionary unpacking which can be seen in the sample_to_model function
          KWARGS: {
              N_ESTIMATORS: scope.int(
                  hp.quniform(f"{RANDOM_FOREST_CLASSIFIER}__{N_ESTIMATORS}", 30, 80,␣
      ↪1)
              ),
              MAX_DEPTH: scope.int(
                  hp.quniform(f"{RANDOM_FOREST_CLASSIFIER}__{MAX_DEPTH}", 3, 10, 1)
              ),
              RANDOM_STATE: 0,
          },
      }

      # Declare the search space for the gradient boosting classifier model,␣
      ↪following the same structure
      # as the random forest classifier search space.
```

```python
gradient_boosting_classifier = {
    MODEL: GRADIENT_BOOSTING_CLASSIFIER,
    KWARGS: {
        LEARNING_RATE: scope.float(
            hp.uniform(
                f"{GRADIENT_BOOSTING_CLASSIFIER}__{LEARNING_RATE}",
                0.01,
                0.15,
            )
        ),  # lower learning rate
        N_ESTIMATORS: scope.int(
            hp.quniform(f"{GRADIENT_BOOSTING_CLASSIFIER}__{N_ESTIMATORS}", 30,
 80, 1)
        ),
        MAX_DEPTH: scope.int(
            hp.quniform(f"{GRADIENT_BOOSTING_CLASSIFIER}__{MAX_DEPTH}", 3, 10,
 1)
        ),
        RANDOM_STATE: 0,
    },
}

# Combine both model search spaces with a top level "choice" between the two
 models to get the final
# search space.
space = {
    MODEL_CHOICE: hp.choice(
        MODEL_CHOICE,
        [
            random_forest_classifier,
            gradient_boosting_classifier,
        ],
    )
}

# Define a few additional variables to represent strings. Note that this code
 expects that we have
# access to all variables that we previously defined in the "search space" code
 snippet.
LOSS = "loss"
STATUS = "status"

# Mapping from string name to model class deinition object that we'll use to
 create an initialized
# version of a model from a sample generated from the search space by hyperopt.
MODELS = {
```

```python
    GRADIENT_BOOSTING_CLASSIFIER: GradientBoostingClassifier,
    RANDOM_FOREST_CLASSIFIER: RandomForestClassifier,
}

# Helper function that converts from a sample generated by hyperopt to an
 ↪initialized model. Note
# that because we split the model type and model keyword-arguments into
 ↪separate key-value pairs in
# the search space declaration we are able to use dictionary unpacking to
 ↪create an initialized
# version of the model.
def sample_to_model(sample):
    kwargs = sample[MODEL_CHOICE][KWARGS]
    return MODELS[sample[MODEL_CHOICE][MODEL]](**kwargs)

# Create a scoring function that we'll use in our objective
cross_ent_scorer = make_scorer(log_loss)

# Define the objective function for hyperopt. We'll fix the `dataset`,
 ↪`features`, and `target`
# arguments with `functools.partial` to create that version of this function
 ↪that we will supply as
# an argument to `fmin`
def objective(sample, dataset_df, features, target):
    model = sample_to_model(sample)
    rng = check_random_state(0)
    # Handle randomization by shuffling when creating folds. In reality, we
 ↪probably want a better
    # strategy for managing randomization than the fixed `RandomState` instance
 ↪generated above.
    cv = KFold(n_splits=10, random_state=rng, shuffle=True)

    # Calculate average cross entropy log loss for each fold. Since `n_splits`
 ↪is 10, `bce` will is an
    # array of size 10 with each element representing the average cross entropy
 ↪log loss for a fold.
    bce = cross_val_score(
        model,
        dataset_df.loc[:, features],
        dataset_df.loc[:, target],
        scoring=cross_ent_scorer,
        cv=cv,
    )

    # Return average of cross entropy log loss across all folds.
    return {LOSS: np.mean(bce), STATUS: STATUS_OK}
```

```
[88]: #Do not run this cell again, it takes forever and is a regression model which␣
      ↪we don't need
      # Since we defined our objective function to be generic in terms of the␣
      ↪dataset, we need to use
      # `partial` from the `functools` module to "fix" the `dataset_df`, `features`,␣
      ↪and `target`
      # arguments to the values that we want for this example so that we have an␣
      ↪objective function that
      # takes in only one argument as assumed by the `hyperopt` interface.
      #compas_objective = partial(
      #    objective, dataset_df=df_dummy, features=FEATURES, target=TWO_YEAR_RECID
      #)
      # `hyperopt` tracks the results of each iteration in this `Trials` object.␣
      ↪We'll be collecting the
      # data that we will use for visualization from this object.
      #trials = Trials()
      #rng = check_random_state(0)  # reproducibility!
      # `fmin` searches for hyperparameters that "minimize" our object, mean squared␣
      ↪error and returns the
      # "best" set of hyperparameters.
      #best = fmin(compas_objective, space, tpe.suggest, 1000, trials=trials,␣
      ↪rstate=rng)
```

```
100%|                                          | 1000/1000
[1:06:14<00:00,  3.97s/trial, best loss: 0.2076017026609608]
```

```
[15]: trialsReg = trials
      pprint([t for t in trialsReg][:5])
```

```
[{'book_time': datetime.datetime(2021, 12, 8, 15, 48, 38, 951000),
  'exp_key': None,
  'misc': {'cmd': ('domain_attachment', 'FMinIter_Domain'),
          'idxs': {'gradient_boosting_regressor__learning_rate': [],
                   'gradient_boosting_regressor__max_depth': [],
                   'gradient_boosting_regressor__n_estimators': [],
                   'model_choice': [0],
                   'random_forest_regressor__max_depth': [0],
                   'random_forest_regressor__n_estimators': [0]},
          'tid': 0,
          'vals': {'gradient_boosting_regressor__learning_rate': [],
                   'gradient_boosting_regressor__max_depth': [],
                   'gradient_boosting_regressor__n_estimators': [],
                   'model_choice': [0],
                   'random_forest_regressor__max_depth': [5.0],
                   'random_forest_regressor__n_estimators': [90.0]},
          'workdir': None},
  'owner': None,
  'refresh_time': datetime.datetime(2021, 12, 8, 15, 48, 41, 667000),
```

```
 'result': {'loss': 0.20818754127918968, 'status': 'ok'},
 'spec': None,
 'state': 2,
 'tid': 0,
 'version': 0},
{'book_time': datetime.datetime(2021, 12, 8, 15, 48, 41, 674000),
 'exp_key': None,
 'misc': {'cmd': ('domain_attachment', 'FMinIter_Domain'),
         'idxs': {'gradient_boosting_regressor__learning_rate': [1],
                  'gradient_boosting_regressor__max_depth': [1],
                  'gradient_boosting_regressor__n_estimators': [1],
                  'model_choice': [1],
                  'random_forest_regressor__max_depth': [],
                  'random_forest_regressor__n_estimators': []},
         'tid': 1,
         'vals': {'gradient_boosting_regressor__learning_rate':
[0.03819110609989756],
                  'gradient_boosting_regressor__max_depth': [8.0],
                  'gradient_boosting_regressor__n_estimators': [137.0],
                  'model_choice': [1],
                  'random_forest_regressor__max_depth': [],
                  'random_forest_regressor__n_estimators': []},
         'workdir': None},
 'owner': None,
 'refresh_time': datetime.datetime(2021, 12, 8, 15, 48, 47, 330000),
 'result': {'loss': 0.21938301350280082, 'status': 'ok'},
 'spec': None,
 'state': 2,
 'tid': 1,
 'version': 0},
{'book_time': datetime.datetime(2021, 12, 8, 15, 48, 47, 340000),
 'exp_key': None,
 'misc': {'cmd': ('domain_attachment', 'FMinIter_Domain'),
         'idxs': {'gradient_boosting_regressor__learning_rate': [2],
                  'gradient_boosting_regressor__max_depth': [2],
                  'gradient_boosting_regressor__n_estimators': [2],
                  'model_choice': [2],
                  'random_forest_regressor__max_depth': [],
                  'random_forest_regressor__n_estimators': []},
         'tid': 2,
         'vals': {'gradient_boosting_regressor__learning_rate':
[0.08587985607913044],
                  'gradient_boosting_regressor__max_depth': [12.0],
                  'gradient_boosting_regressor__n_estimators': [95.0],
                  'model_choice': [1],
                  'random_forest_regressor__max_depth': [],
                  'random_forest_regressor__n_estimators': []},
         'workdir': None},
```

```
  'owner': None,
  'refresh_time': datetime.datetime(2021, 12, 8, 15, 48, 52, 673000),
  'result': {'loss': 0.2453977611590717, 'status': 'ok'},
  'spec': None,
  'state': 2,
  'tid': 2,
  'version': 0},
 {'book_time': datetime.datetime(2021, 12, 8, 15, 48, 52, 679000),
  'exp_key': None,
  'misc': {'cmd': ('domain_attachment', 'FMinIter_Domain'),
           'idxs': {'gradient_boosting_regressor__learning_rate': [],
                    'gradient_boosting_regressor__max_depth': [],
                    'gradient_boosting_regressor__n_estimators': [],
                    'model_choice': [3],
                    'random_forest_regressor__max_depth': [3],
                    'random_forest_regressor__n_estimators': [3]},
           'tid': 3,
           'vals': {'gradient_boosting_regressor__learning_rate': [],
                    'gradient_boosting_regressor__max_depth': [],
                    'gradient_boosting_regressor__n_estimators': [],
                    'model_choice': [0],
                    'random_forest_regressor__max_depth': [2.0],
                    'random_forest_regressor__n_estimators': [93.0]},
           'workdir': None},
  'owner': None,
  'refresh_time': datetime.datetime(2021, 12, 8, 15, 48, 54, 767000),
  'result': {'loss': 0.21586693773448123, 'status': 'ok'},
  'spec': None,
  'state': 2,
  'tid': 3,
  'version': 0},
 {'book_time': datetime.datetime(2021, 12, 8, 15, 48, 54, 774000),
  'exp_key': None,
  'misc': {'cmd': ('domain_attachment', 'FMinIter_Domain'),
           'idxs': {'gradient_boosting_regressor__learning_rate': [4],
                    'gradient_boosting_regressor__max_depth': [4],
                    'gradient_boosting_regressor__n_estimators': [4],
                    'model_choice': [4],
                    'random_forest_regressor__max_depth': [],
                    'random_forest_regressor__n_estimators': []},
           'tid': 4,
           'vals': {'gradient_boosting_regressor__learning_rate':
[0.0638511443414372],
                    'gradient_boosting_regressor__max_depth': [5.0],
                    'gradient_boosting_regressor__n_estimators': [72.0],
                    'model_choice': [1],
                    'random_forest_regressor__max_depth': [],
                    'random_forest_regressor__n_estimators': []},
```

```
            'workdir': None},
  'owner': None,
  'refresh_time': datetime.datetime(2021, 12, 8, 15, 48, 56, 780000),
  'result': {'loss': 0.20907878867489157, 'status': 'ok'},
  'spec': None,
  'state': 2,
  'tid': 4,
  'version': 0}]
```

[10]:
```python
# This is a simple helper function that allows us to fill in `np.nan` when a
 ↪particular
# hyperparameter is not relevant to a particular trial.
def unpack(x):
    if x:
        return x[0]
    return np.nan


# We'll first turn each trial into a series and then stack those series
 ↪together as a dataframe.
trialsReg_df = pd.DataFrame([pd.Series(t["misc"]["vals"]).apply(unpack) for t
 ↪in trialsReg])
# Then we'll add other relevant bits of information to the correct rows and
 ↪perform a couple of
# mappings for convenience
trialsReg_df["loss"] = [t["result"]["loss"] for t in trialsReg]
trialsReg_df["trial_number"] = trialsReg_df.index
trialsReg_df[MODEL_CHOICE] = trialsReg_df[MODEL_CHOICE].apply(
    lambda x: RANDOM_FOREST_REGRESSOR if x == 0 else GRADIENT_BOOSTING_REGRESSOR
)
trialsReg_df
```

[10]:
```
     gradient_boosting_regressor__learning_rate  \
0                                           NaN
1                                      0.038191
2                                      0.085880
3                                           NaN
4                                      0.063851
..                                          ...
995                                         NaN
996                                         NaN
997                                    0.064233
998                                         NaN
999                                         NaN

     gradient_boosting_regressor__max_depth  \
0                                       NaN
```

```
1                                     8.0
2                                    12.0
3                                     NaN
4                                     5.0
..                                    …
995                                   NaN
996                                   NaN
997                                   6.0
998                                   NaN
999                                   NaN

     gradient_boosting_regressor__n_estimators                   model_choice  \
0                                          NaN       random_forest_regressor
1                                        137.0   gradient_boosting_regressor
2                                         95.0   gradient_boosting_regressor
3                                          NaN       random_forest_regressor
4                                         72.0   gradient_boosting_regressor
..                                           …                             …
995                                        NaN       random_forest_regressor
996                                        NaN       random_forest_regressor
997                                       63.0   gradient_boosting_regressor
998                                        NaN       random_forest_regressor
999                                        NaN       random_forest_regressor

     random_forest_regressor__max_depth  \
0                                   5.0
1                                   NaN
2                                   NaN
3                                   2.0
4                                   NaN
..                                    …
995                                 6.0
996                                 6.0
997                                 NaN
998                                 7.0
999                                 5.0

     random_forest_regressor__n_estimators      loss  trial_number
0                                     90.0  0.208188             0
1                                      NaN  0.219383             1
2                                      NaN  0.245398             2
3                                     93.0  0.215867             3
4                                      NaN  0.209079             4
..                                      …         …             …
995                                  125.0  0.207602           995
996                                  124.0  0.207622           996
997                                    NaN  0.210222           997
```

```
998                         131.0  0.208061        998
999                         136.0  0.208064        999

[1000 rows x 8 columns]
```

[104]:
```
#import dill
#dill.dump_session('notebook_env.db')
```

[9]:
```
#import dill
#dill.load_session('notebook_env.db')
```

[11]:
```python
def add_hover_data(fig, df, model_choice):
    # Filter to only columns that are relevant to the current model choice.
 →Note that this relies on
    # the convention of including the model name in the hyperparameter name
 →when we declare the
    # search space.
    cols = [col for col in trialsReg_df.columns if model_choice in col]
    fig.update_traces(
        # This specifies the data that we want to plot for the current model
 →choice.
        customdata=trialsReg_df.loc[
            trialsReg_df[MODEL_CHOICE] == model_choice, cols + [MODEL_CHOICE]
        ],
        hovertemplate="<br>".join(
            [
                f"{col.split('__')[1]}: %{{customdata[{i}]}}"
                for i, col in enumerate(cols)
            ]
        )
        + "<extra></extra>",
        # We only apply the hover data for the current model choice.
        selector={"name": model_choice},
    )
    return fig

# px is an alias for "express" that's created by following the convention of
 →importing "express" by
# running `from plotly import express as px`
fig = px.scatter(
    trialsReg_df,
    x="trial_number",
    y="loss",
    color=MODEL_CHOICE,
)
# We call the `add_hover_data` function once for each model type so that we can
 →add different sets
```

```
# of hyperparameters as hover data for each model type.
fig = add_hover_data(fig, trialsReg_df, RANDOM_FOREST_REGRESSOR)
fig = add_hover_data(fig, trialsReg_df, GRADIENT_BOOSTING_REGRESSOR)
fig.show()
```

[26]:
```
# Since max_depth == 6 outperforms other settings, we'll filter to only look at␣
↪that slice. This
# creates a boolean array that we will use to filter down to relevant rows in␣
↪the `trialsReg_df`
# dataframe.
max_depth_filter = (trialsReg_df[MODEL_CHOICE] == GRADIENT_BOOSTING_REGRESSOR)␣
↪& (
    trialsReg_df["gradient_boosting_regressor__max_depth"] == 6
)

# plotly express does not support contour plots so we will use `graph_objects`␣
↪instead. `go.Contour`
# automatically interpolates "z" values for our loss.
fig = go.Figure(
    data=go.Contour(
        z=trialsReg_df.loc[max_depth_filter, "loss"],
        x=trialsReg_df.loc[max_depth_filter,␣
↪"gradient_boosting_regressor__learning_rate"],
        y=trialsReg_df.loc[max_depth_filter,␣
↪"gradient_boosting_regressor__n_estimators"],
        contours=dict(
            showlabels=True,  # show labels on contours
            labelfont=dict(
                size=12,
                color="white",
            ),  # label font properties
        ),
        colorbar=dict(
            title="loss",
            titleside="right",
        ),
        hovertemplate="loss: %{z}<br>learning_rate: %{x}<br>n_estimators:␣
↪%{y}<extra></extra>",
    )
)

fig.update_layout(
    xaxis_title="learning_rate",
    yaxis_title="n_estimators",
    title={
        "text": "learning_rate vs. n_estimators | max_depth == 6",
```

```
        "xanchor": "center",
        "yanchor": "top",
        "x": 0.5,
    },
)

fig.show()
```

[28]:
```
# sample from the prepped df to split the data into training/validation set and␣
 ↪testing set (0.85 and 0.15, repectively)
train_df, test_df = train_test_split(df_dummy, test_size=0.15)
# samples from the train_df to create a validation dataframe ~10% the size of␣
 ↪the original dataset
train_df, val_df = train_test_split(train_df, test_size=0.12)
```

[20]:
```
# Since we defined our objective function to be generic in terms of the␣
 ↪dataset, we need to use
# `partial` from the `functools` module to "fix" the `train_df`, `features`,␣
 ↪and `target`
# arguments to the values that we want for this example so that we have an␣
 ↪objective function that
# takes in only one argument as assumed by the `hyperopt` interface.
compas_objective = partial(
    objective, dataset_df=train_df, features=FEATURES, target=TWO_YEAR_RECID
)
# `hyperopt` tracks the results of each iteration in this `Trials` object.␣
 ↪We'll be collecting the
# data that we will use for visualization from this object.
trials = Trials()
rng = check_random_state(0)  # reproducibility!
# `fmin` searches for hyperparameters that "minimize" our object, cross entropy␣
 ↪log loss and returns the
# "best" set of hyperparameters.
best = fmin(compas_objective, space, tpe.suggest, 1000, trials=trials,␣
 ↪rstate=rng)
```

```
100%|                                    | 1000/1000
[32:54<00:00,  1.97s/trial, best loss: 11.024427992526189]
```

[21]:
```
pprint([t for t in trials][:5])
```

```
[{'book_time': datetime.datetime(2021, 12, 8, 17, 37, 18, 443000),
  'exp_key': None,
  'misc': {'cmd': ('domain_attachment', 'FMinIter_Domain'),
           'idxs': {'gradient_boosting_classifier__learning_rate': [],
                    'gradient_boosting_classifier__max_depth': [],
                    'gradient_boosting_classifier__n_estimators': [],
                    'model_choice': [0],
```

```
                    'random_forest_classifier__max_depth': [0],
                    'random_forest_classifier__n_estimators': [0]},
            'tid': 0,
            'vals': {'gradient_boosting_classifier__learning_rate': [],
                    'gradient_boosting_classifier__max_depth': [],
                    'gradient_boosting_classifier__n_estimators': [],
                    'model_choice': [0],
                    'random_forest_classifier__max_depth': [5.0],
                    'random_forest_classifier__n_estimators': [50.0]},
            'workdir': None},
  'owner': None,
  'refresh_time': datetime.datetime(2021, 12, 8, 17, 37, 19, 780000),
  'result': {'loss': 11.232224337269372, 'status': 'ok'},
  'spec': None,
  'state': 2,
  'tid': 0,
  'version': 0},
 {'book_time': datetime.datetime(2021, 12, 8, 17, 37, 19, 787000),
  'exp_key': None,
  'misc': {'cmd': ('domain_attachment', 'FMinIter_Domain'),
            'idxs': {'gradient_boosting_classifier__learning_rate': [1],
                    'gradient_boosting_classifier__max_depth': [1],
                    'gradient_boosting_classifier__n_estimators': [1],
                    'model_choice': [1],
                    'random_forest_classifier__max_depth': [],
                    'random_forest_classifier__n_estimators': []},
            'tid': 1,
            'vals': {'gradient_boosting_classifier__learning_rate':
[0.03819110609989756],
                    'gradient_boosting_classifier__max_depth': [7.0],
                    'gradient_boosting_classifier__n_estimators': [73.0],
                    'model_choice': [1],
                    'random_forest_classifier__max_depth': [],
                    'random_forest_classifier__n_estimators': []},
            'workdir': None},
  'owner': None,
  'refresh_time': datetime.datetime(2021, 12, 8, 17, 37, 25, 944000),
  'result': {'loss': 11.344549591808974, 'status': 'ok'},
  'spec': None,
  'state': 2,
  'tid': 1,
  'version': 0},
 {'book_time': datetime.datetime(2021, 12, 8, 17, 37, 25, 951000),
  'exp_key': None,
  'misc': {'cmd': ('domain_attachment', 'FMinIter_Domain'),
            'idxs': {'gradient_boosting_classifier__learning_rate': [2],
                    'gradient_boosting_classifier__max_depth': [2],
                    'gradient_boosting_classifier__n_estimators': [2],
```

```
                    'model_choice': [2],
                    'random_forest_classifier__max_depth': [],
                    'random_forest_classifier__n_estimators': []},
            'tid': 2,
            'vals': {'gradient_boosting_classifier__learning_rate':
[0.08587985607913044],
                    'gradient_boosting_classifier__max_depth': [10.0],
                    'gradient_boosting_classifier__n_estimators': [52.0],
                    'model_choice': [1],
                    'random_forest_classifier__max_depth': [],
                    'random_forest_classifier__n_estimators': []},
            'workdir': None},
  'owner': None,
  'refresh_time': datetime.datetime(2021, 12, 8, 17, 37, 37, 353000),
  'result': {'loss': 11.74329149737122, 'status': 'ok'},
  'spec': None,
  'state': 2,
  'tid': 2,
  'version': 0},
 {'book_time': datetime.datetime(2021, 12, 8, 17, 37, 37, 360000),
  'exp_key': None,
  'misc': {'cmd': ('domain_attachment', 'FMinIter_Domain'),
            'idxs': {'gradient_boosting_classifier__learning_rate': [],
                    'gradient_boosting_classifier__max_depth': [],
                    'gradient_boosting_classifier__n_estimators': [],
                    'model_choice': [3],
                    'random_forest_classifier__max_depth': [3],
                    'random_forest_classifier__n_estimators': [3]},
            'tid': 3,
            'vals': {'gradient_boosting_classifier__learning_rate': [],
                    'gradient_boosting_classifier__max_depth': [],
                    'gradient_boosting_classifier__n_estimators': [],
                    'model_choice': [0],
                    'random_forest_classifier__max_depth': [3.0],
                    'random_forest_classifier__n_estimators': [52.0]},
            'workdir': None},
  'owner': None,
  'refresh_time': datetime.datetime(2021, 12, 8, 17, 37, 38, 808000),
  'result': {'loss': 11.400702012835154, 'status': 'ok'},
  'spec': None,
  'state': 2,
  'tid': 3,
  'version': 0},
 {'book_time': datetime.datetime(2021, 12, 8, 17, 37, 38, 814000),
  'exp_key': None,
  'misc': {'cmd': ('domain_attachment', 'FMinIter_Domain'),
            'idxs': {'gradient_boosting_classifier__learning_rate': [4],
                    'gradient_boosting_classifier__max_depth': [4],
```

```
                    'gradient_boosting_classifier__n_estimators': [4],
                    'model_choice': [4],
                    'random_forest_classifier__max_depth': [],
                    'random_forest_classifier__n_estimators': []},
            'tid': 4,
            'vals': {'gradient_boosting_classifier__learning_rate':
    [0.0638511443414372],
                    'gradient_boosting_classifier__max_depth': [5.0],
                    'gradient_boosting_classifier__n_estimators': [41.0],
                    'model_choice': [1],
                    'random_forest_classifier__max_depth': [],
                    'random_forest_classifier__n_estimators': []},
            'workdir': None},
    'owner': None,
    'refresh_time': datetime.datetime(2021, 12, 8, 17, 37, 40, 880000),
    'result': {'loss': 11.220994295117999, 'status': 'ok'},
    'spec': None,
    'state': 2,
    'tid': 4,
    'version': 0}]
```

```python
[22]:  # This is a simple helper function that allows us to fill in `np.nan` when a
       # particular
       # hyperparameter is not relevant to a particular trial.
       def unpack(x):
           if x:
               return x[0]
           return np.nan


       # We'll first turn each trial into a series and then stack those series
       # together as a dataframe.
       trials_df = pd.DataFrame([pd.Series(t["misc"]["vals"]).apply(unpack) for t in
       trials])
       # Then we'll add other relevant bits of information to the correct rows and
       # perform a couple of
       # mappings for convenience
       trials_df["loss"] = [t["result"]["loss"] for t in trials]
       trials_df["trial_number"] = trials_df.index
       trials_df[MODEL_CHOICE] = trials_df[MODEL_CHOICE].apply(
           lambda x: RANDOM_FOREST_CLASSIFIER if x == 0 else
       GRADIENT_BOOSTING_CLASSIFIER
       )
       trials_df
```

```
[22]:     gradient_boosting_classifier__learning_rate  \
       0                                          NaN
```

```
1                                 0.038191
2                                 0.085880
3                                      NaN
4                                 0.063851
..                                      …
995                                    NaN
996                                    NaN
997                                    NaN
998                                    NaN
999                                    NaN


     gradient_boosting_classifier__max_depth  \
0                                        NaN
1                                        7.0
2                                       10.0
3                                        NaN
4                                        5.0
..                                        …
995                                      NaN
996                                      NaN
997                                      NaN
998                                      NaN
999                                      NaN


     gradient_boosting_classifier__n_estimators                    model_choice  \
0                                            NaN      random_forest_classifier
1                                           73.0  gradient_boosting_classifier
2                                           52.0  gradient_boosting_classifier
3                                            NaN      random_forest_classifier
4                                           41.0  gradient_boosting_classifier
..                                             …                             …
995                                          NaN      random_forest_classifier
996                                          NaN      random_forest_classifier
997                                          NaN      random_forest_classifier
998                                          NaN      random_forest_classifier
999                                          NaN      random_forest_classifier


     random_forest_classifier__max_depth  \
0                                    5.0
1                                    NaN
2                                    NaN
3                                    3.0
4                                    NaN
..                                     …
995                                  7.0
996                                  5.0
997                                  6.0
```

```
998                                      6.0
999                                      5.0

     random_forest_classifier__n_estimators      loss  trial_number
0                                       50.0  11.232224             0
1                                        NaN  11.344550             1
2                                        NaN  11.743291             2
3                                       52.0  11.400702             3
4                                        NaN  11.220994             4
..                                       ...       ...           ...
995                                     30.0  11.215378           995
996                                     51.0  11.220992           996
997                                     45.0  11.080588           997
998                                     59.0  11.052508           998
999                                     52.0  11.220991           999

[1000 rows x 8 columns]
```

```python
[23]: def add_hover_data(fig, df, model_choice):
          # Filter to only columns that are relevant to the current model choice.␣
      ↪Note that this relies on
          # the convention of including the model name in the hyperparameter name␣
      ↪when we declare the
          # search space.
          cols = [col for col in trials_df.columns if model_choice in col]
          fig.update_traces(
              # This specifies the data that we want to plot for the current model␣
      ↪choice.
              customdata=trials_df.loc[
                  trials_df[MODEL_CHOICE] == model_choice, cols + [MODEL_CHOICE]
              ],
              hovertemplate="<br>".join(
                  [
                      f"{col.split('__')[1]}: %{{customdata[{i}]}}"
                      for i, col in enumerate(cols)
                  ]
              )
              + "<extra></extra>",
              # We only apply the hover data for the current model choice.
              selector={"name": model_choice},
          )
          return fig

      # px is an alias for "express" that's created by following the convention of␣
      ↪importing "express" by
      # running `from plotly import express as px`
      fig = px.scatter(
```

```python
    trials_df,
    x="trial_number",
    y="loss",
    color=MODEL_CHOICE,
)
# We call the `add_hover_data` function once for each model type so that we can
 ↪add different sets
# of hyperparameters as hover data for each model type.
fig = add_hover_data(fig, trials_df, RANDOM_FOREST_CLASSIFIER)
fig = add_hover_data(fig, trials_df, GRADIENT_BOOSTING_CLASSIFIER)
fig.show()
```

```python
[27]: # Since max_depth == 6 outperforms other settings, we'll filter to only look at
       ↪that slice. This
      # creates a boolean array that we will use to filter down to relevant rows in
       ↪the `trials_df`
      # dataframe.
      max_depth_filter = (trials_df[MODEL_CHOICE] == GRADIENT_BOOSTING_CLASSIFIER) & (
          trials_df["gradient_boosting_classifier__max_depth"] == 6
      )

      # plotly express does not support contour plots so we will use `graph_objects`
       ↪instead. `go.Contour`
      # automatically interpolates "z" values for our loss.
      fig = go.Figure(
          data=go.Contour(
              z=trials_df.loc[max_depth_filter, "loss"],
              x=trials_df.loc[max_depth_filter,
       ↪"gradient_boosting_classifier__learning_rate"],
              y=trials_df.loc[max_depth_filter,
       ↪"gradient_boosting_classifier__n_estimators"],
              contours=dict(
                  showlabels=True,  # show labels on contours
                  labelfont=dict(
                      size=12,
                      color="white",
                  ),  # label font properties
              ),
              colorbar=dict(
                  title="loss",
                  titleside="right",
              ),
              hovertemplate="loss: %{z}<br>learning_rate: %{x}<br>n_estimators:
       ↪%{y}<extra></extra>",
          )
      )
```

```
fig.update_layout(
    xaxis_title="learning_rate",
    yaxis_title="n_estimators",
    title={
        "text": "learning_rate vs. n_estimators | max_depth == 6",
        "xanchor": "center",
        "yanchor": "top",
        "x": 0.5,
    },
)

fig.show()
```

1.2 Optimized model parameters:

We end up using the Gradient Boosting Classifier with 71 estimators, a max depth of 6, and a learning rate of 0.02973046

With these optimized model parameters we can now implement the Local Massaging and Local Preferential Sampling algorithms

```
[131]: from sklearn.datasets import make_classification
       from sklearn.metrics import confusion_matrix, ConfusionMatrixDisplay,
        ↪precision_score, recall_score, accuracy_score
       from sklearn.svm import SVC

       X_train = train_df[ALL_FEATURES]
       L_train = train_df[FEATURES]
       s_train = train_df[RACE]
       e_train = train_df[DEGREE]
       y_train = train_df[TWO_YEAR_RECID]

       X_val = val_df[ALL_FEATURES]
       L_val = val_df[FEATURES]
       s_val = val_df[RACE]
       e_val = val_df[DEGREE]
       y_val = val_df[TWO_YEAR_RECID]

       X_test = test_df[ALL_FEATURES]
       L_test = test_df[FEATURES]
       s_test = test_df[RACE]
       e_test = test_df[DEGREE]
       y_test = test_df[TWO_YEAR_RECID]
```

```
[132]: GBC = GradientBoostingClassifier(n_estimators=71, learning_rate=0.02973046,
        ↪max_depth=6, random_state=0)
       classifier = GBC.fit(X_train, y_train)
```

```
classifier.score(X_val, y_val)
```

[132]: 0.678343949044586

[133]:
```python
fig, axs = plt.subplots(2, 2, figsize=(12,10))
fig.suptitle('Confusion Matricies of Caucasians and African Americans')

CaucVal = X_val.race==0
CVpredictions = classifier.predict(X_val[CaucVal])
CVcm = confusion_matrix(y_val[CaucVal], CVpredictions, labels=classifier.
 ↪classes_)
CVdisp = ConfusionMatrixDisplay(confusion_matrix=CVcm,
                                display_labels=classifier.classes_)
CVdisp.plot(ax=axs[0,0])

cmTot = np.sum(CVcm[:,:]) + np.sum(AAVcm[:,:])
CVcmF = CVcm/cmTot
CVdispF = ConfusionMatrixDisplay(confusion_matrix=CVcmF,
                                 display_labels=classifier.classes_)
CVdispF.plot(ax=axs[1,0])

AAVal = X_val.race==1
AAVpredictions = classifier.predict(X_val[AAVal])
AAVcm = confusion_matrix(y_val[AAVal], AAVpredictions, labels=classifier.
 ↪classes_)
AAVdisp = ConfusionMatrixDisplay(confusion_matrix=AAVcm,
                                 display_labels=classifier.classes_)
AAVdisp.plot(ax=axs[0,1])

AAVcmF = AAVcm/cmTot
AAVdispF = ConfusionMatrixDisplay(confusion_matrix=AAVcmF,
                                  display_labels=classifier.classes_)
AAVdispF.plot(ax=axs[1,1])

fig.tight_layout()
plt.show()
```

Confusion Matricies of Caucasians and African Americans
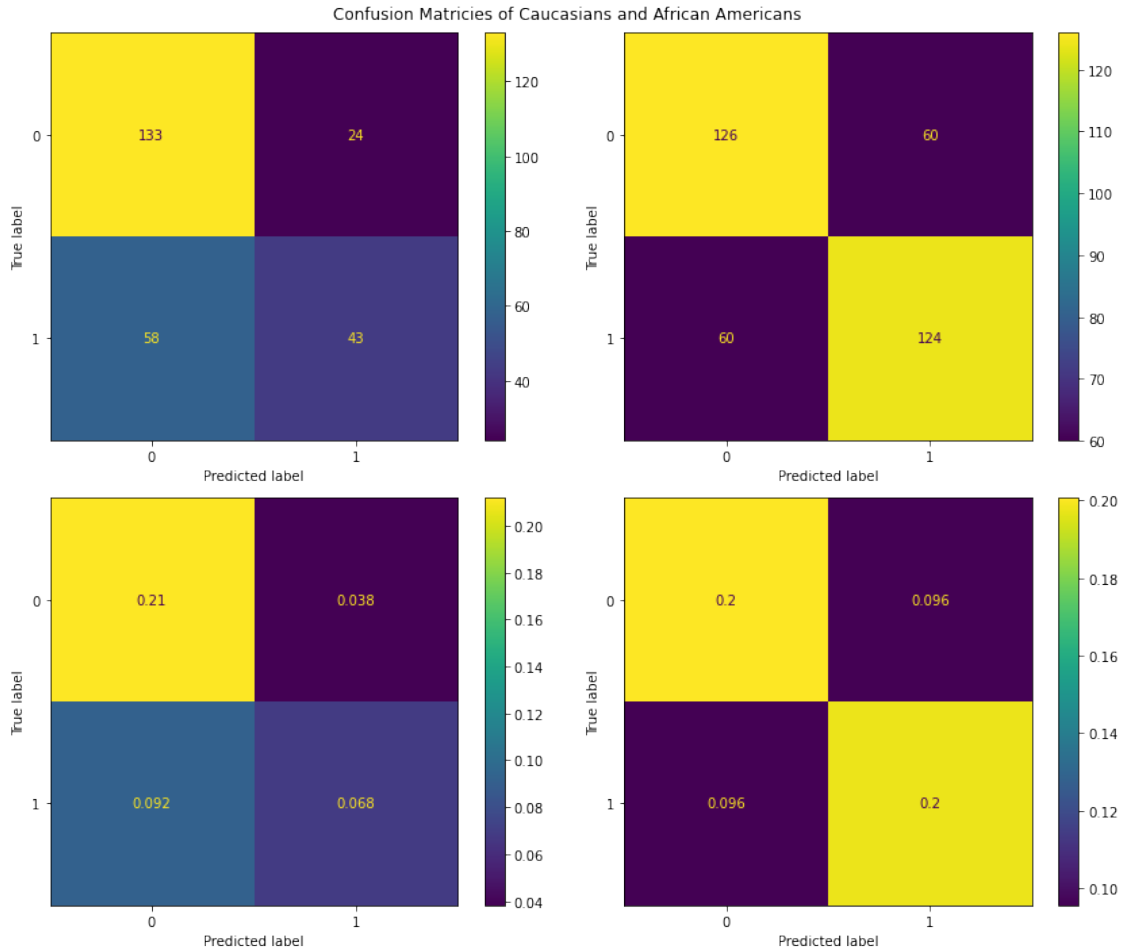
```python
def train_model(model, prediction_function, X_train, y_train, X_test, y_test):
    model.fit(X_train, y_train)

    y_train_pred = prediction_function(model, X_train)
    print('train precision: ' + str(precision_score(y_train, y_train_pred)))
    print('train recall: ' + str(recall_score(y_train, y_train_pred)))
    print('train accuracy: ' + str(accuracy_score(y_train, y_train_pred)))

    y_test_pred = prediction_function(model, X_test)
    print('test precision: ' + str(precision_score(y_test, y_test_pred)))
    print('test recall: ' + str(recall_score(y_test, y_test_pred)))
    print('test accuracy: ' + str(accuracy_score(y_test, y_test_pred)))

    return model

def get_predicted_outcome(model, data):
    return np.argmax(model.predict_proba(data), axis=1).astype(np.float32)
```

```
[135]: GBCmodel = train_model(GBC, get_predicted_outcome, L_train, y_train, L_val,␣
        ↪y_val)
```

```
train precision: 0.7167310877967974
train recall: 0.6048462255358807
train accuracy: 0.7040661013263753
test precision: 0.6627906976744186
test recall: 0.6
test accuracy: 0.6799363057324841
```

```
[136]: GBCmodel = train_model(GBC, get_predicted_outcome, X_train, y_train, X_val,␣
        ↪y_val)
```

```
train precision: 0.7232905982905983
train recall: 0.6309412861136999
train accuracy: 0.7151554685801261
test precision: 0.6653386454183267
test recall: 0.5859649122807018
test accuracy: 0.678343949044586
```

2 Inferences from using the baseline Gradient Boosting Classifier on the validation set

The cross validation statistics improving when using all parameters to predict recidivism and the imbalanced confusion matricies suggest the model might not be fair, which is something we will check next

2.1 Evaluating the 3 fairness definitions - Parity, Calibration, and Equality of Odds - we get the following results:

2.1.1 Parity:

The algorithm should satisfy P(G=1|D=0)=P(G=1|D=1) to achieve fairness definition 1. This suggests that the probability of a positive recividism guess should be the same regardless of demographic (race in our case). This is similar to

However, it's apparent that this is far from true (difference of 0.2376).

```
[137]: print('P(G=1|race=caucasian)=', sum(CVcmF[:,1])/np.sum(CVcmF))
       print('P(G=1|race=african american)=', sum(AAVcmF[:,1])/np.sum(AAVcmF))
       print('P(G=1|race=african american)!=P(G=1|race=caucasian)')
```

```
P(G=1|race=caucasian)= 0.2596899224806202
P(G=1|race=african american)= 0.4972972972972974
P(G=1|race=african american)!=P(G=1|race=caucasian)
```

2.1.2 Calibration:

The algorithm should satisfy P(G=T|D=0)=P(G=T|D=1) to achieve fairness definition 2. This suggests that the probability of an accurate recividism guess should be the same regardless of

demographic (race in our case). This is similar to Accuracy.

In this case, the algorithim almost satisfies calibration (difference of only 0.0065).

```
[138]: print('P(G=T|race=caucasian)=', (CVcmF[0,0]+CVcmF[1,1])/np.sum(CVcmF))
       print('P(G=T|race=african american)=', (AAVcmF[0,0]+AAVcmF[1,1])/np.sum(AAVcmF))
       print('P(G=T|race=african american)~=P(G=T|race=caucasian)')
```

```
P(G=T|race=caucasian)= 0.6821705426356589
P(G=T|race=african american)= 0.6756756756756758
P(G=T|race=african american)~=P(G=T|race=caucasian)
```

2.1.3   Equality of Odds:

The algorithm should satisfy P(G=T|D=0,T=1)=P(G=T|D=1,T=1) to achieve fairness definition 3. This suggests that the probability of an accurate positive recidivism guess should be the same regardless of demographic (race in our case). This is similar to Recall.

In this case, the algorithim is even further off than definition 1 (difference of 0.2482)

```
[139]: print('P(G=T|race=caucasian,recidivism=true)=', (CVcmF[1,1])/sum(CVcmF[1,:]))
       print('P(G=T|race=african american,recidivism=true)=', (AAVcmF[1,1])/
       ↪sum(AAVcmF[1,:]))
       print('P(G=T|race=african american,recidivism=true)!
       ↪=P(G=T|race=caucasian,recidivism=true)')
```

```
P(G=T|race=caucasian,recidivism=true)= 0.42574257425742573
P(G=T|race=african american,recidivism=true)= 0.6739130434782608
P(G=T|race=african
american,recidivism=true)!=P(G=T|race=caucasian,recidivism=true)
```

3   Local Massaging (Handling Conditional Discrimination Alg1)

```
[163]: #Helper functions
       def get_predicted_outcome(model, data):
           return np.argmax(model.predict_proba(data), axis=1).astype(np.float32)

       def get_predicted_rank(model, data):
           return model.predict_proba(data)[:, 1]

       #full_df is the dataframe with all columns to be partitioned
       #e is the column/list of values of the different crime degrees
       def PARTITION(full_df, e):
           ret_dfs = list()
           uniques = np.unique(e)

           for u in uniques:
               ret_dfs.append(full_df[full_df[DEGREE]==u])
```

```python
        return ret_dfs

#full_dfi is the dataframe with all columns, but partitioned to one crime degree
#si is the current sensitive parameter value
def DELTA(full_df, full_dfi, si):
    raceSub = full_dfi[RACE]==si
    Gi = sum(raceSub)
    num = full_dfi[raceSub][TWO_YEAR_RECID]==1
    denom = sum(raceSub)/len(full_df)
    P1 = num/denom

    raceSub = full_dfi[RACE]!=si
    num = full_dfi[raceSub][TWO_YEAR_RECID]==1
    denom = sum(raceSub)/len(full_df)
    Ps = 0.5*(P1 + num/denom)

    return Gi*abs(P1-Ps)
```

```python
[186]:  #Local Massaging algorithm
        for part in PARTITION(train_df, train_df[DEGREE]):
            X_part = part.drop(TWO_YEAR_RECID, axis=1)
            y_part = part[TWO_YEAR_RECID]
            model = GBC.fit(X_part, y_part)

            rank1 = get_predicted_rank(model, X_part[X_part[RACE]==1])
            delta1 = DELTA(train_df, part, 1)

            rank0 = get_predicted_rank(model, X_part[X_part[RACE]==0])
            delta0 = DELTA(train_df, part, 0)

            #relabel closest DELTA ranks from the partition
```

```
[0 0 0 0 0 1 0 0 1 1 0 0 0 0 0 1 0 0 0 0 0 0 0 1 1 0 0 0 1 0 1 0 0 0 0 0 1 1
 0 0 1 0 1 0 0 1 0 1 0 1 0 1 0 1 0 0 0 1 0 0 0 0 0 0 1 0 0 1 0 0 0 0 0 1 1 0
 0 1 0 1 0 0 1 0 0 0 1 0 0 0 0 0 1 1 1 0 1 0 0 0 0 0 0 1 0 0 0 1 0 0 0
 0 0 0 1 0 0 0 0 0 1 0 1 0 0 0 0 1 1 0 1 0 0 0 0 0 0 0 0 0 0 0 0 1 0 1
 0 0 1 0 0 0 0 0 0 1 0 1 1 0 0 0 0 1 1 0 0 1 0 1 1 0 0 0 0 0 1 1 0 0 0 1
 1 0 0 0 0 0 0 0 1 0 0 0 1 0 0 1 0 0 0 1 0 0 1 0 0 1 0 0 0 0 0 1 1 0 1 1 0 1 0
 1 0 0 1 0 0 0 0 0 0 1 0 0 0 0 0 0 0 0 1 0 1 0 0 0 0 0 1 0 0 1 1 0 1
 0 0 0 0 1 0 0 0 0 1 1 0 0 0 0 1 0 0 0 0 0 0 0 0 0 1 0 0 0 1 1 1 0 0 1 0 0
 0 1 1 0 0 0 0 0 0 1 1 0 1 0 1 0 0 0 1 0 0 0 0 0 0 0 0 0 0 0 0 1 0 1 0 0 1
 0 0 0 1 0 0 0 0 1 0 1 0 1 0 1 0 0 1 0 1 0 1 0 1 0 0 0 0 0 1 1 0 1 0 1 0 0 1
 0 0 0 0 1 0 1 0 0 0 0 1 0 0 0 0 0 0 1 0 0 1 0 0 0 1 0 1 0 1 0 1 0 0 0 1 0 0
 0 1 0 0 0 0 1 1 0 1 0 1 0 0 1 1 1 1 0 0 0 1 1 0 0 0 0 0 1 0 1 1 0 0 0 1 0
 0 0 0 1 0 0 0 0 0 0 1 0 0 1 0 0 1 1 1 0 1 0 1 0 1 0 0 0 1 0 1 0 0 0 1 0 1 0
 1 0 0 0 0 0 1 1 0 0 0 0 0 0 0 1 0 0 0 0 0 1 0 0 0 0 1 1 1 1 1 0 0 0 0 0 0 0
 0 0 0 0 1 1 0 0 0 0 0 0 0 0 0 0 0 0 0 1 0 0 1 0 0 0 1 1 0 0 1 0 0 1 0 1
 1 0 0 1 0 0 0 0 0 1 1 0 1 0 1 0 1 0 0 0 1 0 0 0 0 0 0 0 1 0 0 0 0 0 0 0 0 0
```

```
 0 1 0 0 0 0 1 0 1 0 1 0 1 0 1 0 1 0 1 0 0 0 0 0 1 0 1 0 0 0 0 0 0 0 0 0 0 1 0
 0 1 0 0 1 0 0 1 0 0 0 1 1 0 0 0 0 0 1 0 0 0 1 1 1 0 0 0 1 0 0 0 0 1 1 0 0
 0 0 1 1 0 1 0 0 0 1 1 1 0 1 0 1 1 0 1 0 0 1 0 0 1 0 1 0 0 0 0 0 0 0 0 0 0
 1 0 1 1 1 1 0 1 1 1 1 0 0 1 0 0 0 0 1 1 0 1 1 1 1 0 1 0 1 0 1 0 1 0 1 0 0
 1 0 1 0 1 0 1 0 1 0 0 0 0 0 0 1 0 0 0 0 0 0 1 0 0 0 0 1 1 1 1 0 0 0 0 0 0
 1 1 0 1 0 1 0 0 0 0 1 0 0 1 0 0 0 0 0 1 1 0 1 0 0 1 0 1 0 0 1 1 0 1 1 1 0
 0 0 0 0 1 0 0 0 0 0 1 1 0 1 1 0 1 0 0 0 0 0 0 1 0 0 0 1 0 0 1 0 0 0 1 1 1
 1 1 1 0 0 0 0 0 0 1 0 0 0 0 0 1 0 0 1]
[1 0 0 … 1 1 0]
```

4 Local Preferential Sampling (Handling Conditional Discrimination Alg2)

```python
#Local Preferential Sampling algorithm
for part in PARTITION(train_df, train_df[DEGREE]):
    X_part = part.drop(TWO_YEAR_RECID, axis=1)
    y_part = part[TWO_YEAR_RECID]
    model = GBC.fit(X_part, y_part)

    rank1 = get_predicted_rank(model, X_part[X_part[RACE]==1])
    delta1 = DELTA(train_df, part, 1)

    rank0 = get_predicted_rank(model, X_part[X_part[RACE]==0])
    delta0 = DELTA(train_df, part, 0)

    #remove closest DELTA/2 ranks from the partition
```