# 4 Bias and fairness: Modeling recidivism

This chapter covers

- Recognizing and mitigating bias in our data and ML models
- Quantifying fairness through various metrics
- Applying feature engineering techniques to remove bias from our model without sacrificing model performance

In our last chapter, we focused on building a feature engineering pipeline that would maximize our model's performance on our dataset. This is generally the stated goal of an ML problem. Our goal in this chapter will be to not only monitor and measure model performance but also to keep track of how our model treats different groups of data because sometimes *data are people*.

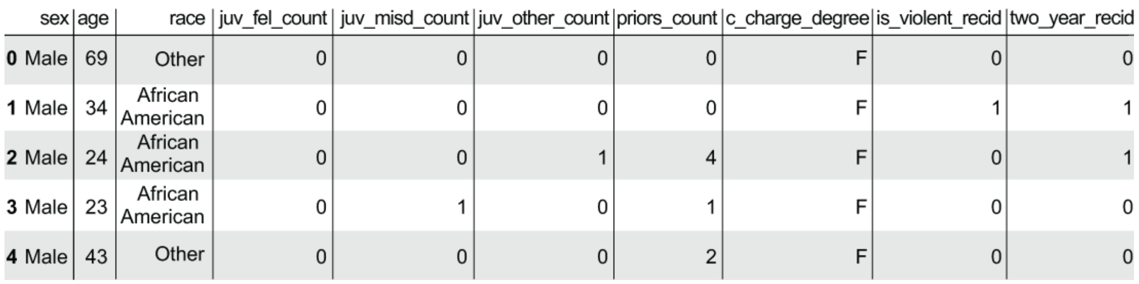
In our case study today, data are people whose lives are on the line. Data are people who simply want to have the best life they can possibly have. As we navigate the waters around bias and discrimination, around systemic privilege and racial discrepancies, *we urge you to keep in mind that when we are talking about rows of data, we are talking about people, and when we are talking about features, we are talking about aggregating years—if not decades—of life experiences into a single number, class, or Boolean*. We must be respectful of our data and of the people our data represents. Let's get started.

## 1 The COMPAS dataset

The dataset for this case study is the *Correctional Offender Management Profiling for Alternative Sanctions* (COMPAS) dataset, which is a collection of criminal offenders screened in Broward County, Florida, in the years of 2013-2014. In particular, we are looking at a subset of this data that corresponds to a binary classification problem of predicting recidivism (whether or not a person will reoffend), given certain characteristics about an individual. A link to the dataset can be found here: **https://www.kaggle.com/danofer/compass**.

On its face, the problem is pretty simple—binary classification, no missing data—let's go! The problem arises when our ML models have very real downstream effects on people's lives and well being. As ML engineers and data scientists, much of this burden is on us to create models that are not only performing well but are also generating predictions that can be considered fair.

As we go through this chapter, we will define and quantify *fair* in many ways, and ultimately, a decision must be rendered on what fairness criteria is best for a particular problem domain. It will be our goal in this chapter to introduce various definitions of fairness and give examples throughout on how each one is meant to be interpreted. Let's jump in and start by ingesting our data and taking a look around in the listing 4.1 (figure 4.1).



| | sex | age | race | juv_fel_count | juv_misd_count | juv_other_count | priors_count | c_charge_degree | is_violent_recid | two_year_recid |
|---|---|---|---|---|---|---|---|---|---|---|
| **0** | Male | 69 | Other | 0 | 0 | 0 | 0 | F | 0 | 0 |
| **1** | Male | 34 | African American | 0 | 0 | 0 | 0 | F | 1 | 1 |
| **2** | Male | 24 | African American | 0 | 0 | 1 | 4 | F | 0 | 1 |
| **3** | Male | 23 | African American | 0 | 1 | 0 | 1 | F | 0 | 0 |
| **4** | Male | 43 | Other | 0 | 0 | 0 | 2 | F | 0 | 0 |

Figure 4.1 The first five rows of our COMPAS dataset, which shows some sensitive information about people who have been incarcerated in Broward County, Florida. Our response label here is `two_year_recid`, which represents an answer to the binary question, "Did this person return to incarceration within 2 years of being released?"

**NOTE** This case study does not represent a statistical study, nor should it be used to make any generalizations about America's criminal justice system. Our goal is to highlight instances of bias in data and promote tools to maximize fairness in our ML systems.

Listing 4.1 Ingesting the data

```
import pandas as pd                                         ❶
import numpy as np                                          ❶
compas_df = pd.read_csv('../data/compas-scores-two-years.csv')  ❷
compas_df.head()                                            ❷
```

❶ Import packages.

❷ Show the first five rows.

In the original ProPublica study from 2016 that looked into the fairness of the COMPAS algorithm, software, and underlying data, they focused on the *decile score* given to each person. A decile score is a score from 1-10 that scales data into buckets of 10%. If this word looks somewhat familiar, it's because it is closely related to the idea of a percentile. The idea is that a person can be given a score between 1-10, where each score represents a bucket of a population in which a certain percentage of people above and below rank higher in a metric. For example, if we give someone a decile score of 3, that means 70% of people should have a higher risk of recidivism (people with scores of 4, 5, 6, 7, 8, 9, and 10), and 20% of people have a lower risk of recidivism (people with scores of 1 and 2). Likewise, a score of 7 means 30% of people have a higher rate of recidivism (people with scores of 8, 9, and 10), where 60% of people have a lower rate of recidivism (people with scores of 1, 2, 3, 4, 5, and 6).

The study went further to show the disparities between how decile scores are used and how they don't always look fair. For example, if we look at how scores are distributed, we can see that scores are given out differently by race. The following snippet will plot a histogram of decile scores by race (figure 4.2) and highlight a few things:

- African American decile scores are spread out relatively evenly, with about 10% of the population residing in each decile score. By definition of a decile score, this on its face is appropriate. In theory, 10% of the population should live in each decile score.
- The Asian, Caucasian, Hispanic, and other categories seem to have a right skew on decile scores with a larger-than-expected portion of the category having a decile score of 1 or 2.

```
compas_df.groupby('race')['decile_score'].value_counts(
    normalize=True
).unstack().plot(
    kind='bar', figsize=(20, 7),
    title='Decile Score Histogram by Race', ylabel='% with Decile Score'
)
```

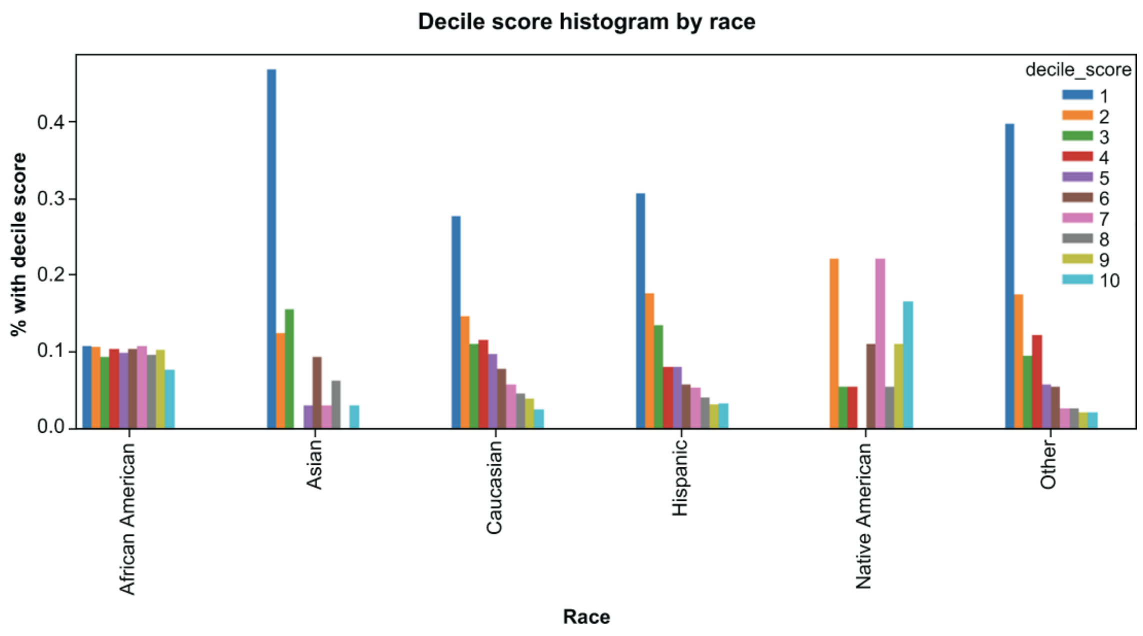
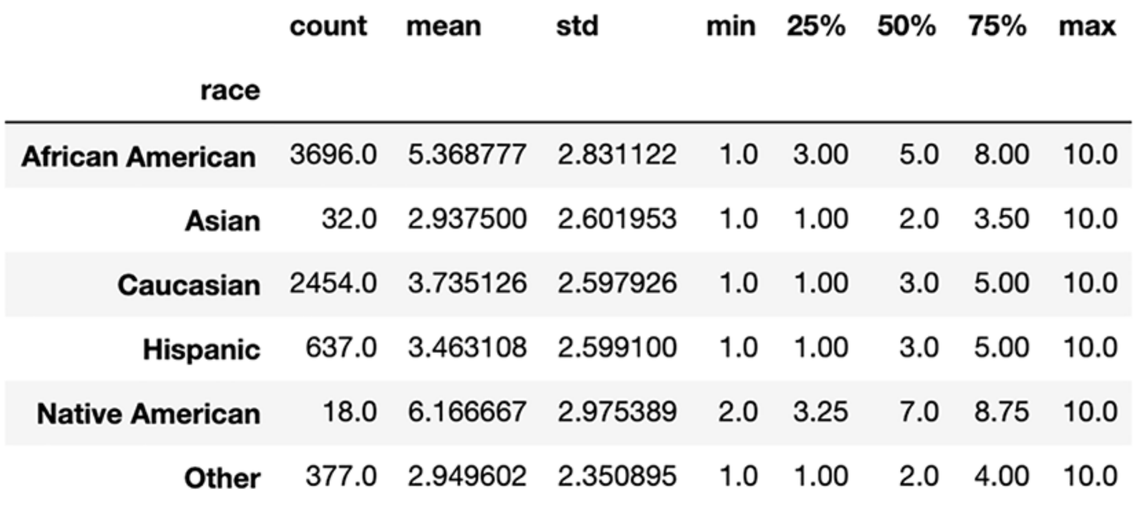


Figure 4.2 We can see clear differences in how decile scores are distributed when broken down by race.

We can see this more clearly by inspecting some basic statistics around decile scores by race, as shown in figure 4.3.

```
compas_df.groupby('race')['decile_score'].describe()
```



| | count | mean | std | min | 25% | 50% | 75% | max |
|---|---|---|---|---|---|---|---|---|
| **race** | | | | | | | | |
| **African American** | 3696.0 | 5.368777 | 2.831122 | 1.0 | 3.00 | 5.0 | 8.00 | 10.0 |
| **Asian** | 32.0 | 2.937500 | 2.601953 | 1.0 | 1.00 | 2.0 | 3.50 | 10.0 |
| **Caucasian** | 2454.0 | 3.735126 | 2.597926 | 1.0 | 1.00 | 3.0 | 5.00 | 10.0 |
| **Hispanic** | 637.0 | 3.463108 | 2.599100 | 1.0 | 1.00 | 3.0 | 5.00 | 10.0 |
| **Native American** | 18.0 | 6.166667 | 2.975389 | 2.0 | 3.25 | 7.0 | 8.75 | 10.0 |
| **Other** | 377.0 | 2.949602 | 2.350895 | 1.0 | 1.00 | 2.0 | 4.00 | 10.0 |

Figure 4.3 Taking a look at the means and medians of decile scores by race, we can see, for example, that the median decile score for African Americans is 5 (which is expected), but for Caucasians and Hispanics it is 3.

We could go on looking at how the ProPublica study interpreted this data, but rather than attempting to recreate these results, our approach to this dataset will be focused on building a binary classifier with the data, ignoring the decile score already given to people.

## .1.1 The problem statement and defining success

As mentioned in the previous section, the ML problem here is one of binary classification. The goal of our model can be summarized by this question: "Given certain aspects about a person, can we predict recidivism both accurately *and fairly*?"

The term *accurately* should be easy enough. We have plenty of metrics to measure model performance, including accuracy, precision, and AUC.

When it comes to the term *fairly*, however, we will need to learn a few new terms and metrics. Before we get into how to quantify bias and fairness, let's first do some EDA knowing the problem at hand.

## 2 Exploratory data analysis

Our goal is to directly model our response label `two_year_recid`, based on features about people in this dataset. Specifically, we have the following features:

- `sex`—Qualitative (binary `Male` or `Female`)
- `age`—Quantitative ratio (in years)
- `race`—Qualitative nominal
- `juv_fel_count`—Quantitative (the number of prior juvenile felonies this person has)
- `juv_misd_count`—Quantitative (the number of prior juvenile misdemeanors this person has)
- `juv_other_count`—Quantitative (the number of juvenile convictions that are neither a felony nor a misdemeanor)
- `priors_count`—Quantitative (the number of prior crimes committed)
- `c_charge_degree`—Qualitative, binary (`F` for felony and `M` for misdemeanor)

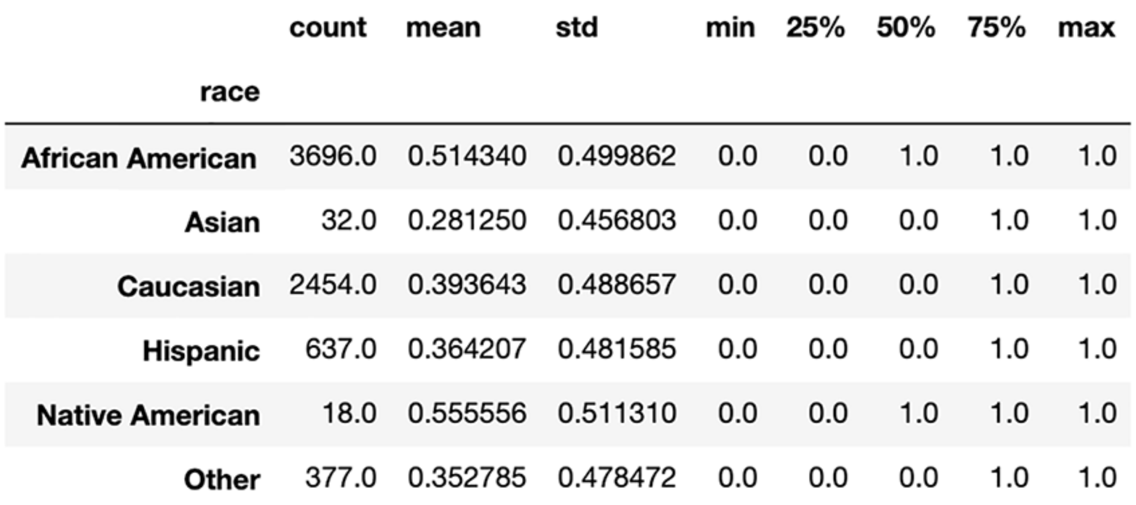
And our response label is

- `two_year_recid`—Qualitative, binary (did this person recidivate, commit another crime, within 2 years)

Note that we have three separate columns to count juvenile offenses. We should note that for our models, we may want to combine these into a single column that simply counts the number of juvenile offenses this person had.

Given our problem statement of creating an accurate and fair model, let's look at the breakdown of recidivism, by race (figure 4.4). When we group our dataset by race and look at the rate of recidivism, it becomes clear that there are differences in the base rates of recidivism. Without breaking down further (e.g., by age, criminal history, etc.) there are pretty big differences in recidivism rates between different race categories.

```
compas_df.groupby('race')['two_year_recid'].describe()
```



| | count | mean | std | min | 25% | 50% | 75% | max |
|---|---|---|---|---|---|---|---|---|
| **race** | | | | | | | | |
| **African American** | 3696.0 | 0.514340 | 0.499862 | 0.0 | 0.0 | 1.0 | 1.0 | 1.0 |
| **Asian** | 32.0 | 0.281250 | 0.456803 | 0.0 | 0.0 | 0.0 | 1.0 | 1.0 |
| **Caucasian** | 2454.0 | 0.393643 | 0.488657 | 0.0 | 0.0 | 0.0 | 1.0 | 1.0 |
| **Hispanic** | 637.0 | 0.364207 | 0.481585 | 0.0 | 0.0 | 0.0 | 1.0 | 1.0 |
| **Native American** | 18.0 | 0.555556 | 0.511310 | 0.0 | 0.0 | 1.0 | 1.0 | 1.0 |
| **Other** | 377.0 | 0.352785 | 0.478472 | 0.0 | 0.0 | 0.0 | 1.0 | 1.0 |

Figure 4.4 Descriptive statistics of recidivism by race. We can see clear difference in recidivism rates between our different racial groups.

We should also note that we have two race categories (`Asian` and `Native American`) with extremely small representation in our data. This is an example of a *sample bias*, where the population may not be represented appropriately. These data are taken from Broward County, Florida, where, according to the US census, those identifying as Asian, for example, make up about 4% of the population, whereas in this dataset, they make up about .44% of the data.

For our purposes in this book, we will relabel data points with race as either `Asian` or `Native American` as `Other` to avoid any misconceptions in our metrics in relation to having two categories of race being so underrepresented. Our main reason for doing this relabeling is to make the resulting classes more balanced. In our last figure, it's clear that the counts of people in the `Asian` and `Native American` classes are vastly underrepresented, and therefore, it would be inappropriate to try and use this dataset to make meaningful predictions about them. Once we relabel these data points, let's then plot the actual 2-year recidivism rates for our, now, four considered race categories, as shown in the following listing and the resulting figure 4.5.

Listing 4.2 Relabeling underrepresented races

```
compas_df.loc[compas_df['race'].isin(['Native American', 'Asian']), 'race'] =
➥ 'Other'                                                              ❶

compas_df.groupby('race')['two_year_recid'].value_counts(
    normalize=True
).unstack().plot(
    kind='bar', figsize=(10, 5), title='Actual Recidivism Rates by Race'
)                                                                          ❷
```

❶ Relabel rows with Asian/Native American races as Other.

❷ Plot recidivism rates for the four races we are considering.

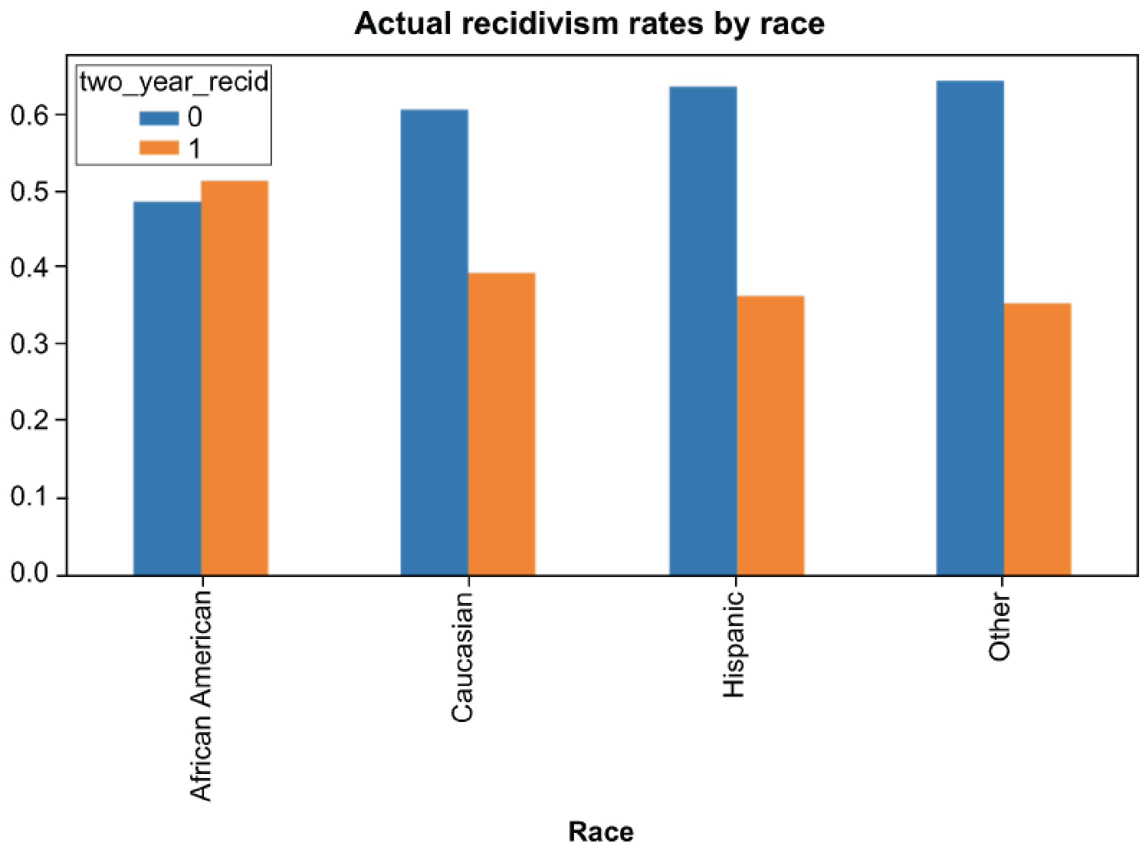


Figure 4.5 Bar chart showing recidivism rates by group

Again, we can see that our data are showing that African American peo-ple recidivate at a higher rate than Caucasian, Hispanic, or other. This is true due to numerous systemic reasons that we cannot begin to touch on in this book. For now, let’s note that, even though recidivism rates are different between groups, the difference between a near 50/50 split for African Americans and the 60/40 split for Caucasians is not radically different rates.

**NOTE** We also could have chosen to look at bias in sex, as there are definitely disparities between those identified as males and females in this dataset. For the purposes of this case study, we chose to focus on the racial biases present in the data.

Let’s continue on by looking at our other features a bit more. We have a binary charge degree feature that we will have to encode as a Boolean but otherwise looks usable in its current form (figure 4.6):

```
compas_df[‘c_charge_degree’].value_counts(normalize=True).plot(
   kind=’bar’, title=’% of Charge Degree’, ylabel=’%’, xlabel=’Charge Degree’
)
```

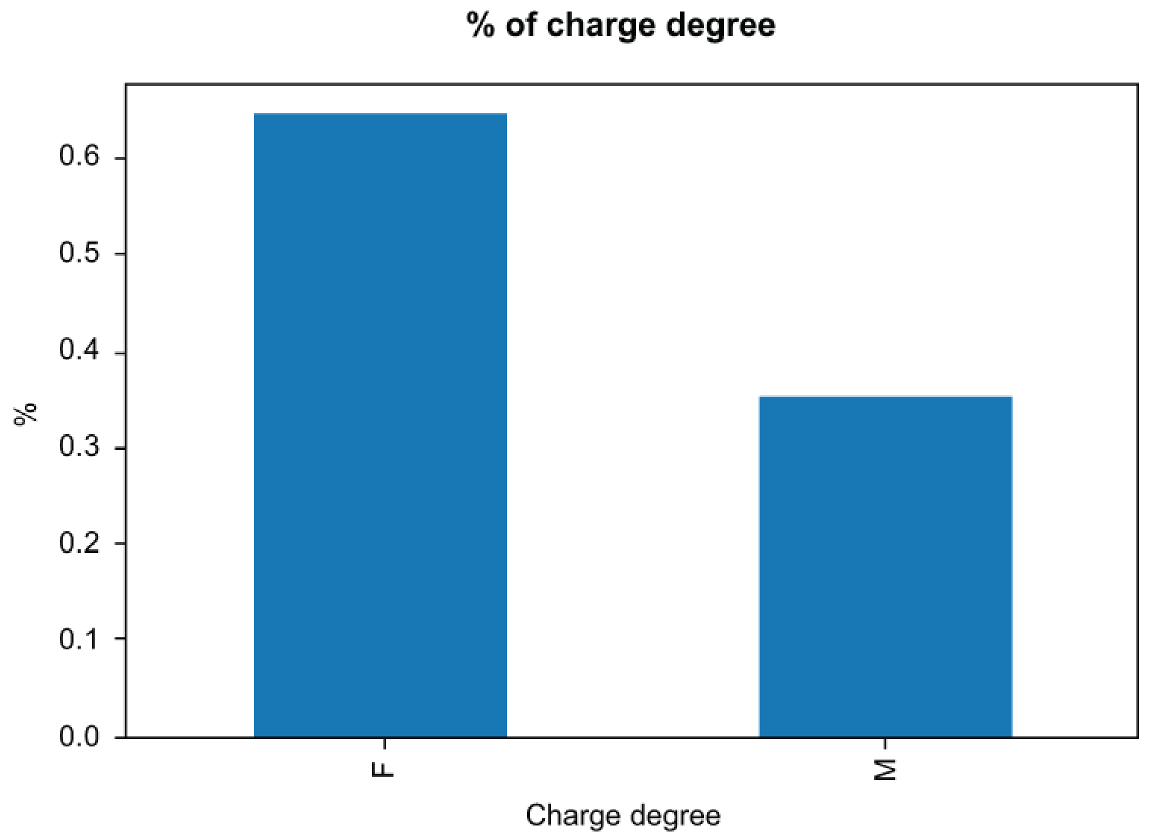


Figure 4.6 Distribution of felonies and misdemeanors in our dataset by degree. We have about 65% of our charges as *F* for felonies, and the rest are *M* for misdemeanors.

Let’s wrap up our EDA by looking at a histogram of our remaining quantitative features: `age` and `priors_count`. Both of these variables are showing a pretty clear right skew and would benefit from some standardization to reel in those outliers a bit, as shown in the following listing with resulting graphs in figure 4.7.

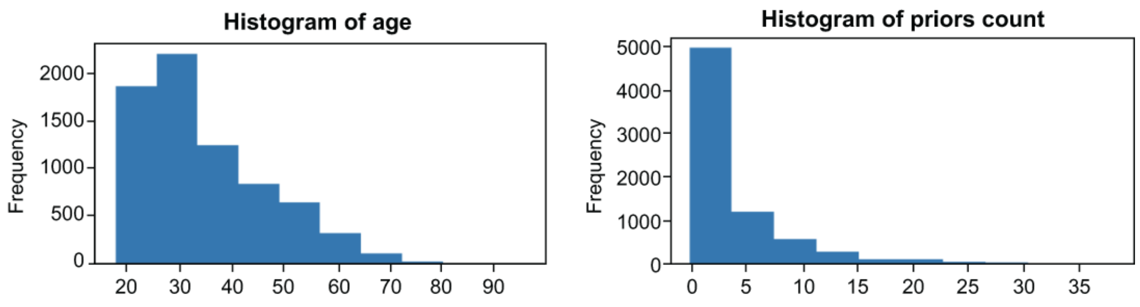


Figure 4.7 `Age` and `Priors Count` are showing a right skew in the data. It's showing that most of the people in our dataset are on the young side, but we do have some outliers pulling the average to the right. This will come up again when we investigate model fairness.

Listing 4.3 Plotting histograms of our quantitative variables

```
compas_df['age'].plot(
    title='Histogram of Age', kind='hist', xlabel='Age', figsize=(10, 5)
)                                              ❶

compas_df['priors_count'].plot(
    title='Histogram of Priors Count', kind='hist', xlabel='Priors',
    ➥ figsize=(10, 5)
)                                              ❷
```

❶ Right skew on Age

❷ Right skew on Priors, as well

With our EDA giving us some initial insight, let's move on to discussing and measuring the bias and fairness of our models.

## 3 Measuring bias and fairness

When tasked with making model predictions fair and as unbiased as possible, we need to look at a few different ways of formulating and quantifying fairness. Doing so will help us quantify how well our ML models are doing.

### .3.1 Disparate treatment vs. disparate impact

In general, a model—or really, any predictive or decision-making process—can suffer from two forms of bias: disparate treatment and disparate impact. A model is considered to suffer from *disparate treatment* if predictions are in some way based on a sensitive attribute (e.g., sex or race). A model could also have a *disparate impact* if the predictions or downstream outcomes of the predictions disproportionately hurt or benefit people with specific sensitive features, which can look like predicting higher rates of recidivism for one race over another.

### .3.2 Definitions of fairness

There are at least dozens of ways of defining fairness in a model, but let’s focus on three for now. When we are building our baseline model, we will see these again and more.

#### *UNAWARENESS*

*Unawareness* is likely the easiest definition of fairness. It states that a model should not include sensitive attributes as a feature in the training data. This way our model will not have access to the sensitive values when training. This definition aligns well with the idea of disparate treatment in that we are literally not allowing the model to see the sensitive values of our data.

The surface-level pro of using unawareness as a definition is that it is very easy to explain to someone that we simply did not use a feature in our model; therefore, how could it have obtained any bias? The counter-argument to this statement and the major flaw of relying on unawareness to define fairness is that, more often than not, the model will be able to reconstruct sensitive values by relying on other features that are highly correlated to the original sensitive feature we were trying to be unaware of.

For example, if a recruiter is deciding whether or not to hire a candidate and we wish for them to be sensitive to the sex of the candidate, we could simply blind the recruiter to the candidate's sex; however, if the recruiter also notices that the candidate listed *fraternities* as a prior volunteer or leadership experience, the recruiter may reasonably put together that the candidate is likely a male.

## *STATISTICAL PARITY*

*Statistical parity,* also known as demographic parity or disparate impact, is a very common definition for fairness. Simply put, it states that our model's prediction of being in a certain class (whether they will recidivate or not) is independent of the sensitive feature. Put as a formula

P(recidivism | race = African American) = P(recidivism | race = Caucasian) = P(recidivism | race = Hispanic) = P(recidivism | race =

Other)

Put yet another way, to achieve good statistical parity, our model should predict equal rates of recidivism for every racial category. The above formula is pretty strict, and to relax this we can lean on the *four-fifths rule*, which states that if the selection rate (the rate at which we predict recidivism) of a certain group is either less than 80% or greater than 125% of the selection rate of another group, then we are possibly looking at disparate impact. As a formula, this looks like the following:

0.8 < P(recidivism | race = disadvantaged) / P(recidivism | race = advantaged) < 1 / .8 (1.25)

The pros of using statistical parity as a definition of fairness is that it is relatively easy to explain the metric, and there is evidence that using statistical parity as a definition can lead to both short-term and long-term benefits of the disadvantaged groups (see Hu and Chen, https://arxiv.org/pdf/1712.00064.pdf).

One caveat of relying on statistical parity is that it ignores any possible relationship between our label and our sensitive attribute. In our case, this is actually a good thing because we want to ignore any correlation between our response (will this person recidivate) and our sensitive attribute in question (race), as that correlation is driven by much bigger factors than our case study can deal with. For any use case you may consider in the future, this may not be desired, so please take this into consideration!

Another caveat of relying solely on statistical parity is that our ML model, in theory, could just be *lazy* and select random people from each group, and we would still technically achieve statistical parity. Obviously, our ML metrics should prevent our models from doing this, but it's always something to look out for.

### *EQUALIZED ODDS*

Also known as positive rate parity, the *equalized odds* definition of fairness states that our model's prediction of our response should be independent of our sensitive feature, conditional on our response value. In the context of our example, equalized odds would mean the following two conditions are met:

- P(recidivism | race = Hispanic, actually recidivated = True) = P(recidivism | race = Caucasian, actually recidivated = True) = P(recidivism | race = African American, actually recidivated = True) = P(recidivism | race = Other, actually recidivated = True)
- P(recidivism | race = Hispanic, actually recidivated = False) = P(recidivism | race = Caucasian, actually recidivated = False) = P(recidivism | race = African American, actually recidivated = False) = P(recidivism | race = Other, actually recidivated = False)

Another way to view this would be to say our model has equalized odds if

- Independent of race, our model predicted recidivism rates equally for people who did actually recidivate.
- Independent of race, our model predicted recidivism rates equally for people who did not actually recidivate.

The pros of using equalized odds for our definition are that it penalizes the same *laziness* we talked about with statistical parity; and it encourages the model to become more accurate in all groups, rather than allowing the model to simply randomly predict recidivism to achieve similar rates of prediction between groups.

The biggest flaw is those equalized odds are sensitive to different underlying base rates of the response. In our data, we saw that African Americans recidivated at a higher rate than the other three racial categories. If this was a scenario wherein we believed there were some natural differences between racial groups and recidivism rates, then equalized odds would not be a good metric for us. In our case, this will not be an issue, as we reject the idea that these base rates relating to race and recidivism reflect natural recidivism rates.

**NOTE** There are dozens of established metrics for measuring fairness and bias. Our case study will touch on a few of them, but we recommend looking at other texts that focus exclusively on bias and fairness for a comprehensive treatment.

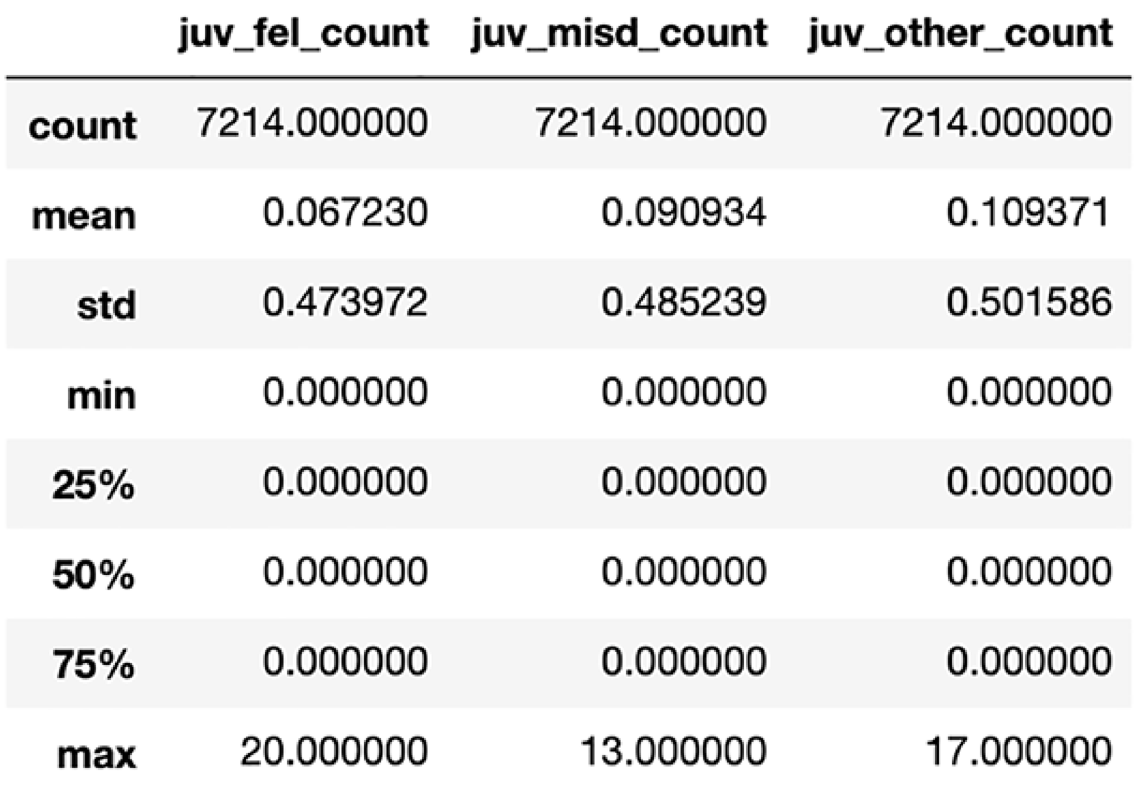
## 4 Building a baseline model

It's time to build our baseline ML model. For our first pass at our model, we will apply a bit of feature engineering to ensure our model interprets all of our data correctly and spend time analyzing the fairness/performance results of our model.

## .4.1 Feature construction

As we saw in our EDA, we have three features that each count the number of juvenile offenses of the person in question. Let’s take another look at our three juvenile features (figure 4.8).

```
compas_df[["juv_fel_count", "juv_misd_count", "juv_other_count"]].describe()
```



| | juv_fel_count | juv_misd_count | juv_other_count |
|---|---|---|---|
| **count** | 7214.000000 | 7214.000000 | 7214.000000 |
| **mean** | 0.067230 | 0.090934 | 0.109371 |
| **std** | 0.473972 | 0.485239 | 0.501586 |
| **min** | 0.000000 | 0.000000 | 0.000000 |
| **25%** | 0.000000 | 0.000000 | 0.000000 |
| **50%** | 0.000000 | 0.000000 | 0.000000 |
| **75%** | 0.000000 | 0.000000 | 0.000000 |
| **max** | 20.000000 | 13.000000 | 17.000000 |

Figure 4.8 We have three different features that each count a subset of prior juvenile offenses. Our goal will be to combine them together into one feature.

Let’s combine these features into one by adding them all up into a new column called `juv_count` that should be a bit easier to interpret, as shown in the following listing.

Listing 4.4 Constructing a new juvenile offense count feature

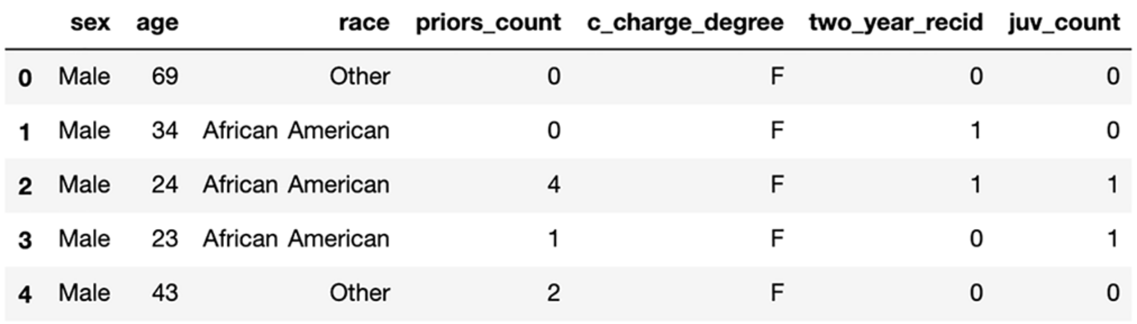
```
compas_df['juv_count'] = compas_df[["juv_fel_count", "juv_misd_count",
➥ "juv_other_count"]].sum(axis=1)                                    ❶
compas_df = compas_df.drop(["juv_fel_count", "juv_misd_count",
➥ "juv_other_count"], axis=1)                                        ❷
```

❶ Construct our new total juvenile offense count.

❷ Remove the original juvenile features.

**EXERCISE 4.1** Plot the distribution of the new `juv_count` feature. What is the arithmetic mean and standard deviation?

We now have one new feature, and we have removed three features as a result. Figure 4.9 shows the current state of our training data.



| | sex | age | race | priors_count | c_charge_degree | two_year_recid | juv_count |
|---|---|---|---|---|---|---|---|
| 0 | Male | 69 | Other | 0 | F | 0 | 0 |
| 1 | Male | 34 | African American | 0 | F | 1 | 0 |
| 2 | Male | 24 | African American | 4 | F | 1 | 1 |
| 3 | Male | 23 | African American | 1 | F | 0 | 1 |
| 4 | Male | 43 | Other | 2 | F | 0 | 0 |

Figure 4.9 The current state of our training data with our combined juvenile offense count

## .4.2 Building our baseline pipeline

Let’s start putting our pipeline together to create our baseline ML model. Let’s begin by splitting up our data into training and testing sets, and let’s also instantiate a static random forest classifier. We have chosen a random forest model here because random forest models have the useful feature of calculating feature importances. This will end up being very useful for us. We could have chosen a decision tree or even a logistic regression, as they both also have representations of feature importances, but for now, we will go with a random forest. Remember that our goal is to manipulate our features and not our model, so we will use the same model with the same parameters for all of our iterations. In addition to splitting up our *x* and our *y*, we will also split the race column, so we have an easy way to split our test set by race.

Listing 4.5 Splitting up our data into training and testing sets

```
from sklearn.model_selection import train_test_split
from sklearn.ensemble import RandomForestClassifier
```

```
X_train, X_test, y_train, y_test, race_train, race_test = train_test_split(
    compas_df.drop('two_year_recid', axis=1),                                  ❶

compas_df['two_year_recid'],
                                                    compas_df['race'],

stratify=compas_df['two_year_recid'],
                                                    test_size=0.3,
                                                    random_state=0)

classifier = RandomForestClassifier(
    max_depth=10, n_estimators=20, random_state=0)                             ❷
```

❶ Split up our data.

❷ Our static classifier

Now that we have our data split up and our classifier ready to go, let's start creating our feature pipelines just like we did in our last chapter. First up is our categorical data. Let's create a pipeline that will one-hot encode our categorical columns and only drop the second dummy column if the categorical feature is binary, as shown in the following listing.

Listing 4.6 Creating our qualitative pipeline

```
from sklearn.compose import ColumnTransformer
from sklearn.pipeline import Pipeline, FeatureUnion
from sklearn.preprocessing import OneHotEncoder, StandardScaler

categorical_features = ['race', 'sex', 'c_charge_degree']
```

```
categorical_transformer = Pipeline(steps=[
    ('onehot', OneHotEncoder(drop='if_binary'))
])
```

And for our numerical data, we will scale our data to bring down those outliers that we saw in our EDA.

Listing 4.7 Creating our quantitative pipeline

```
numerical_features = ["age", "priors_count"]
numerical_transformer = Pipeline(steps=[
    ('scale', StandardScaler())
])
```

Let's introduce the `ColumnTransformer` object from scikit-learn that will help us quickly apply our two pipelines to our specific columns with minimal code, as shown in the following listing.

Listing 4.8 Putting our pipelines together to create our feature preprocessor

```
preprocessor = ColumnTransformer(transformers=[
        ('cat', categorical_transformer, categorical_features),
        ('num', numerical_transformer, numerical_features)
])

clf_tree = Pipeline(steps=[
    ('preprocessor', preprocessor),
        ('classifier', classifier)
    ])
```

With our pipeline set up, we can train it on our training set and run it on our test set.

Listing 4.9 Running our bias-unaware model on our test set

```
clf_tree.fit(X_train, y_train)
unaware_y_preds = clf_tree.predict(X_test)
```

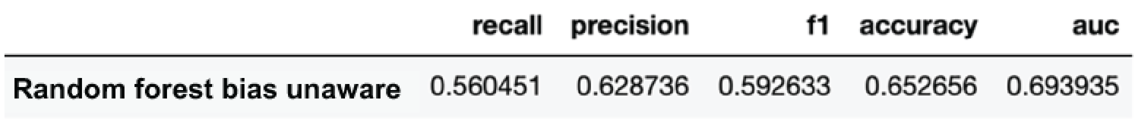
`Unaware_y_preds` will be an array of 0s and 1s, where 0 represents our model predicting that this person will not recidivate, and a 1 represents our model, predicting that this person will recidivate. Now that we have our predictions of our model predicting on our test set, it's time to start investigating how fair our ML model truly is.

## .4.3 Measuring bias in our baseline model

To help us dive into our fairness metrics, we are going to be using a module called *Dalex*. Dalex has some excellent features that help visualize different kinds of bias and fairness metrics. Our base object is the `Explainer` object, and with our explainer object we can obtain some basic model performance, as shown in the following listing with results in figure 4.10.

Listing 4.10 Using Dalex to explain our model

```
import dalex as dx

exp_tree = dx.Explainer(
    clf_tree, X_test, y_test,
    label='Random Forest Bias Unaware', verbose=True)
exp_tree.model_performance()
```



| | recall | precision | f1 | accuracy | auc |
|---|---|---|---|---|---|
| **Random forest bias unaware** | 0.560451 | 0.628736 | 0.592633 | 0.652656 | 0.693935 |

Figure 4.10 Baseline model performance with our bias-unaware model

Our metrics are not amazing, but we are concerned with both performance and fairness, so let's dig into fairness a bit. Our first question is, "How much did our model rely on race as a way to predict recidivism?" This question goes hand in hand with our model's disparate treatment. Dalex has a very handy plot that can be used with tree-based models and linear models to help visualize the features our model is learning the most from (figure 4.11).

```
exp_tree.model_parts().plot()
```

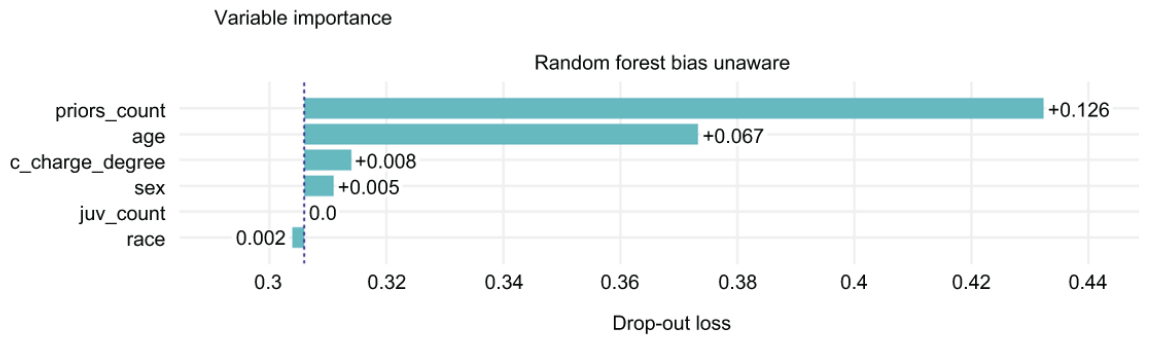


Figure 4.11 Feature importance of our bias-unaware model as reported by Dalex. This visualization is taking feature importances directly from the random forest's feature importance attribute and is showing that `pri-ors_count` and `age` are our most important features.

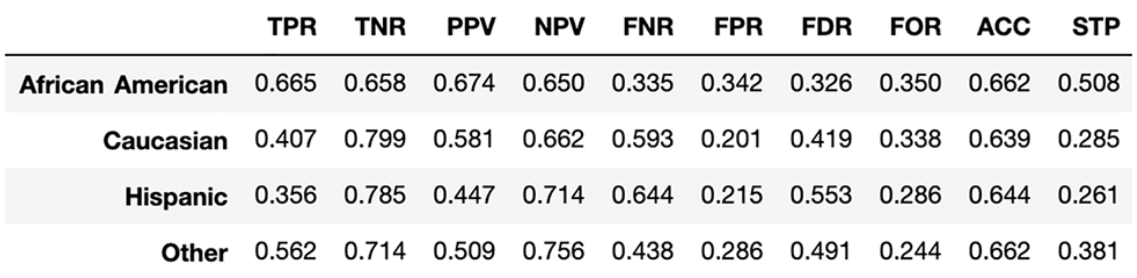
Dalex is reporting importance in terms of *drop-out loss*, or how much the overall *fit* of our model would decrease if the feature in question were entirely removed. According to this chart, our model would lose a lot of information if we lost `priors_count`, but in theory, it would have been better if we dropped `race`. It would seem that our model isn't even learning from the race at all! This speaks to the model's unawareness of sensitive features.

Before we begin our no-bias-here dance, we should look at a few more metrics. Dalex also has a `model_fairness` object we can look at that will calculate several metrics for each of our racial categories, as shown in the following listing with results in figure 4.12.

Listing 4.11 Outputting model fairness

```
mf_tree = exp_tree.model_fairness(
    protected=race_test, privileged = "Caucasian")
```

```
mf_tree.metric_scores
```



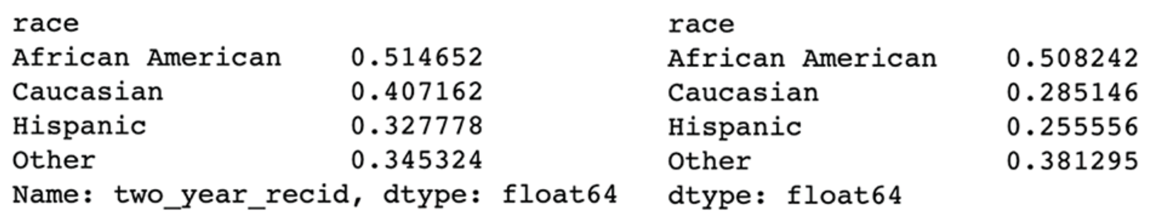
| | TPR | TNR | PPV | NPV | FNR | FPR | FDR | FOR | ACC | STP |
|---|---|---|---|---|---|---|---|---|---|---|
| **African American** | 0.665 | 0.658 | 0.674 | 0.650 | 0.335 | 0.342 | 0.326 | 0.350 | 0.662 | 0.508 |
| **Caucasian** | 0.407 | 0.799 | 0.581 | 0.662 | 0.593 | 0.201 | 0.419 | 0.338 | 0.639 | 0.285 |
| **Hispanic** | 0.356 | 0.785 | 0.447 | 0.714 | 0.644 | 0.215 | 0.553 | 0.286 | 0.644 | 0.261 |
| **Other** | 0.562 | 0.714 | 0.509 | 0.756 | 0.438 | 0.286 | 0.491 | 0.244 | 0.662 | 0.381 |

Figure 4.12 A breakdown of 10 fairness metrics for our bias-unaware model

This package gives us 10 metrics here by default; let's break down how to calculate each one in terms of true positives (TP), false positives (FP), false negatives (FN), actual positives (AP), actual negatives (AN), predicted positives (PP), and predicted negatives (PN). Keep in mind that we can calculate each of these metrics by race:

1. TPR(r) = TP / AP (aka sensitivity)
2. TNR(r) = TN / AN (aka specificity)
3. PPV(r) = TP / (PP) (aka precision)
4. NPV(r) = TN / (PN)
5. FNR(r) = FN / AP *or* 1 - TPR
6. FPR(r) = FP / AN *or* 1 - TNR
7. FDR(r) = FP / (PP) *or* 1 - PPV
8. FOR(r) = FN / (PN) *or* 1 - NPV
9. ACC(r) = TP + TN / (TP + TN + FP + FN) (overall accuracy by race)
10. STP(r) = TP + FP / (TP + FP + FP + FN) (aka P[recidivism predicted | Race = r])

These numbers on their own will not be very helpful, so let’s perform a fairness check by comparing our values to the privileged group of people: Caucasians. Why are we choosing Caucasians as our privileged group? Well, among a lot of other reasons, if we look at how often our baseline model predicted recidivism between our groups, we will notice that the model is vastly underpredicting Caucasian recidivism compared to actual rates in our test set (listing 4.12 and figure 4.13).



```
race                                         race
African American    0.514652                 African American    0.508242
Caucasian           0.407162                 Caucasian           0.285146
Hispanic            0.327778                 Hispanic            0.255556
Other               0.345324                 Other               0.381295
Name: two_year_recid, dtype: float64         dtype: float64
```

Figure 4.13 On the left we have actual recidivism rates by group in our test set, and the right has the rates of recidivism predicted by our baseline bias-unaware model. Our model is vastly underpredicting Caucasian recidivism. Nearly 41% of Caucasian folk recidivated; meanwhile, our model only thought 28% of them would. That means that our underpredicted recidivism for Caucasians by over 30%,

For our purposes, we will focus on TPR, ACC, PPV, FPR, and STP as our main metrics. The reason we are choosing these metrics is that

- TPR relates to how well our model captures actual recidivism. Of all the times people recidivate, did our model predict them as positive? We want this to be higher.
- ACC is our overall accuracy. It is a fairly well-rounded way to judge our model but will not be taken into consideration in a vacuum. We want this to be higher.

- PPV is our precision. It measures how much we can trust our model's positive predictions. Of the times our model predicts recidivism, how often was the model correct in that positive prediction? We want this to be higher.
- FPR relates to our model's rate of predicting recidivism when someone will not actually recidivate. We want this to be lower.
- STP is statistical parity per group. We want this to be roughly equal to each other by race, meaning our model should be able to reliably predict recidivism based on nondemographic information.

Listing 4.12 Highlighting Caucasian privilege

```
y_test.groupby(race_test).mean()                                    ❶

pd.Series(unaware_y_preds, index=y_test.index).groupby(
    race_test).mean()                                               ❷
```

❶ Recidivism by race in our test set

❷ Predicted recidivism by race in our bias-unaware model

The rates of recidivism predicted among African American people are very similar, while Caucasians seem to only get a recidivism prediction less than 29% of the time, even though the actual rate is almost 41%. The fact that our model is underpredicting the Caucasian group is an indicator that Caucasians are privileged by our model. Part of the reason this is happening is that the data are representative of an unfair justice system. Thinking back to the fact that African Americans have a higher priors count and that the priors count was the most important feature in our model, and it is still unable to accurately predict Caucasian recidivism, our model is clearly unable to reliably predict recidivism based on the raw data.

Let’s run that fairness check now to see how our bias-unaware model is doing across our five bias metrics:

```
mf_tree = exp_tree.model_fairness(protected=race_test, privileged =
➥ “Caucasian”)
mf_tree.fairness_check()
```

Our output is outlined in the following table, and at first glance, it is a lot! We’ve highlighted the main areas to focus on. We want each of the values to be between (0.8 and 1.25), and the boldface values are those that are outside of that range and, therefore, being called out as being evidence of bias.

```
Bias detected in 4 metrics: TPR, PPV, FPR, STP
Conclusion: your model is not fair because 2 or more criteria exceeded
acceptable limits set by epsilon.
Ratios of metrics, based on ‘Caucasian’. Parameter ‘epsilon’ was set to 0.8
and, therefore, metrics should be within (0.8, 1.25)
                       TPR       ACC       PPV       FPR       STP
African American  1.633907  1.035994  1.160069  1.701493  1.782456
Hispanic          0.874693  1.007825  0.769363  1.069652  0.915789
Other             1.380835  1.035994  0.876076  1.422886  1.336842
```

Each value in the table above is the value from the `metric_scores` table divided by the Caucasian value (our privileged group). For example, the

African American TPR value of 1.633907 is equal to the TPR(African American) / TPR(Caucasian), which is calculated as 0.665 / 0.407.

These ratios are then checked against a four-fifth range of (0.8, 1.25), and if our metric falls outside of that range, we consider that ratio unfair. The ideal value is 1, which indicates that the specified metric for that race is equal to the value of that metric for our privileged group. If we count up the number of ratios outside of that range, we come up with 7 (they are in boldface). We can plot the numbers in the previous table using Dalex as well (figure 4.14):

```
mf_tree.plot()  # Same numbers from the fairness_check in a plot
```

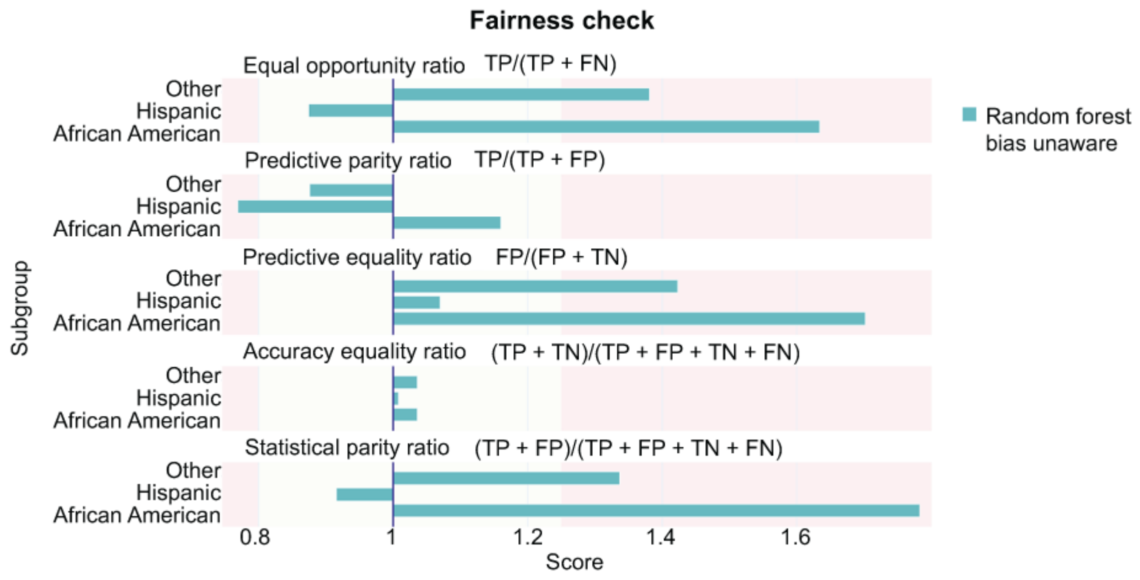


Figure 4.14 Dalex offers a visual breakdown of the five main ratios we will focus on, broken down by subgroup. The lighter areas of the bars are meant to convey acceptable ranges for fairness. Any bar in the darker re-

gions are out of range of (.8, 1.25) and are considered unfair. We can see that we have some work to do!

To make things a bit simpler, let’s focus on the parity loss of each of the five metrics from our fairness check. *Parity loss* represents a total score across our disadvantaged groups. Dalex calculates parity loss for a metric as being the sum of the absolute value of the log of the metric ratios in our fairness checks.

$$metric_{parity\_loss} = \sum_{i \in \{a,b,\ldots z\}} \left| \log\left(\frac{metric_i}{metric_{privileged}}\right) \right|$$

For example, if we look at the statistical parities of our groups (STP), we have the following:

STP(African American) = 0.508

STP(Hispanic) = 0.261

STP(Other) = 0.381

STP(Caucasian) = 0.285

We can calculate our parity loss for STP for our bias-unaware model should be 0.956. Luckily, Dalex gives us an easier way to calculate parity loss for all five metrics and stack them together in a chart. Figure 4.15 is the one we will use to compare across our models, and the five stacks represent the values for each of our five bias metrics. They are stacked up together to represent the overall bias of the model. We want to see the overall stacked length to *decrease* as we become more bias aware. We will be pairing this stacked parity loss graph with classic ML metrics, like accuracy, precision, and recall as seen in figure 4.15.

**EXERCISE 4.2** Write Python code to calculate the STP parity loss.

```
mf_tree.plot(type = 'stacked')    ❶
```

❶ Plot of parity loss of each metric

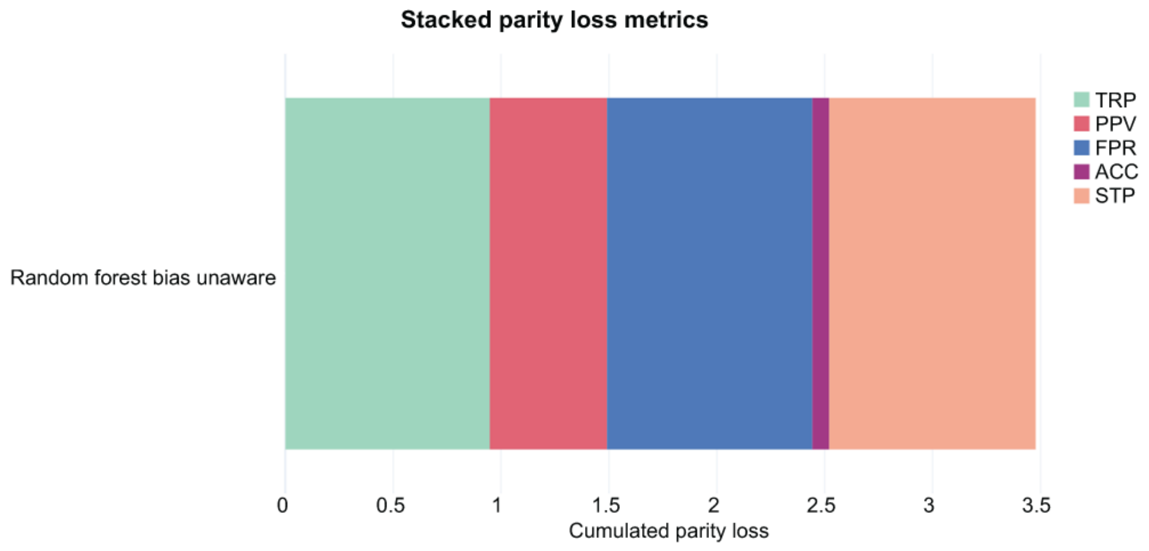


Figure 4.15 Cumulative parity loss. In this case, smaller is better, meaning less bias. For example, the far-right section represents the 0.956 we previously calculated by hand. Overall, our bias-unaware model is scoring around 3.5, which is our number to beat on the bias front.

We now have both a baseline for model performance (from our model performance summary) and a baseline for fairness given by our stacked parity loss chart. Let’s move on now to how we can actively use feature engineering to mitigate bias in our data.

## 5 Mitigating bias

When it comes to mitigating bias and promoting fairness in our models, we have three main opportunities to do so:

1. *Preprocessing*—Bias mitigation, as applied to the training data (i.e., before the model has had a chance to train on the training data)
2. *In-processing*—Bias mitigation applied to a model during the training phase
3. *Postprocessing*—Bias mitigation applied to the predicted labels after the model has been fit to the training data

Each phase of bias mitigation has pros and cons, and preprocessing directly refers to feature engineering techniques and, therefore, will be the main focus of this chapter.

### 4.5.1 Preprocessing

Preprocessing bias mitigation takes place in the training data before modeling takes place. Preprocessing is useful when we don’t have access to the model itself or the downstream predictions, but we do have access to the initial training data.

Two examples of preprocessing bias mitigation techniques that we will implement in this chapter are

- *Disparate impact removal*—Editing feature values to improve group fairness
- *Learning fair representations*—Extracting a new feature set by obfuscating the original information regarding protected attributes

By implementing these two techniques, we will be hoping to reduce the overall bias that our model is exhibiting, while also trying to enhance our ML pipeline's performance.

### .5.2 In-processing

In-processing techniques are applied during training time. They usually come in the form of some regularization term or an alternative objective function. In-processing techniques are only possible when we have access to the actual learning algorithm. Otherwise, we'd have to rely on pre- or postprocessing.

Some examples of in-processing bias mitigation techniques include

- *Meta fair classifier*—Uses fairness as an input to optimize a classifier for fairness
- *Prejudice remover*—Implementing a privilege-aware regularization term to our learning objective

### .5.3 Postprocessing

Postprocessing techniques, as the name implies, are applied after training time and are most useful when we need to treat the ML model as a black box, and we don’t have access to the original training data. Some examples of postprocessing bias mitigation techniques include

- *Equalized odds*—Modifying predicted labels, using a separate optimization objective to make the predictions fairer
- *Calibrated equalized odds*—Modifying the classifier’s scores to make for fairer results

## 6 Building a bias-aware model

Let’s begin to construct a more bias-aware model, using two feature engineering techniques. We will begin by applying a familiar transformation to construct a new less-biased column and then move on to new feature extraction methods. Our goal is to minimize the bias of our model without sacrificing a great deal of model performance.

### .6.1 Feature construction: Using the Yeo-Johnson transformer to treat the disparate impact

In the last chapter, we used the Box-Cox transformation to transform some of our features to make them appear more normal. We will want to do something similar here. We have to investigate why our model is underpredicting recidivism for non-African American people. One approach would be to remove race entirely from our dataset and expect the ML model to remove all bias. This rarely is the answer.

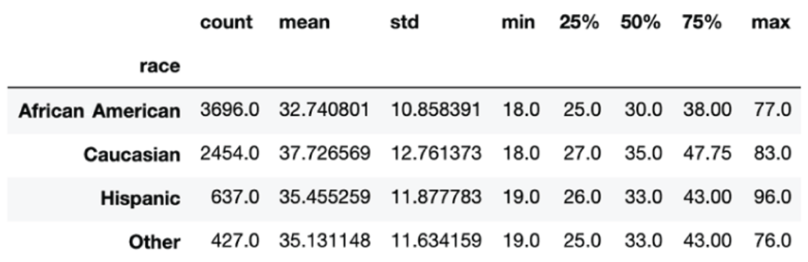
Unprivileged and privileged groups of people experience different opportunities, and this likely presents itself in the data through correlated features. The most likely cause for our model’s bias is that at least one of our features is highly correlated with race, and our model is able to reconstruct someone's racial identity through this feature. To find this feature, let's start by finding the correlation coefficient of our numerical features and being African American:

```
compas_df.corrwith(compas_df['race'] == 'African-American').sort_values()
age              -0.179095
juv_count         0.111835
priors_count      0.202897
```

Both `age` and `priors_count` are highly correlated with our Boolean label of simply being African American, so let's take a closer look at each of those. Let's start by looking at age. We can plot a histogram and print out some basic statistics (figure 4.16), and we will see that across our four racial categories, age seems to be relatively similar with a similar mean, standard deviation, and median. This signals to us that even though age is negatively correlated to being African American, this relationship is likely not a huge contributing factor to our model's bias.

```
compas_df.groupby('race')['age'].plot(
    figsize=(20,5),
    kind='hist', xlabel='Age', title='Histogram of Age'      ❶
)
compas_df.groupby('race')['age'].describe()
```

❶ Age is not very skewed.



| | count | mean | std | min | 25% | 50% | 75% | max |
|---|---|---|---|---|---|---|---|---|
| **race** | | | | | | | | |
| **African American** | 3696.0 | 32.740801 | 10.858391 | 18.0 | 25.0 | 30.0 | 38.00 | 77.0 |
| **Caucasian** | 2454.0 | 37.726569 | 12.761373 | 18.0 | 27.0 | 35.0 | 47.75 | 83.0 |
| **Hispanic** | 637.0 | 35.455259 | 11.877783 | 19.0 | 26.0 | 33.0 | 43.00 | 96.0 |
| **Other** | 427.0 | 35.131148 | 11.634159 | 19.0 | 25.0 | 33.0 | 43.00 | 76.0 |

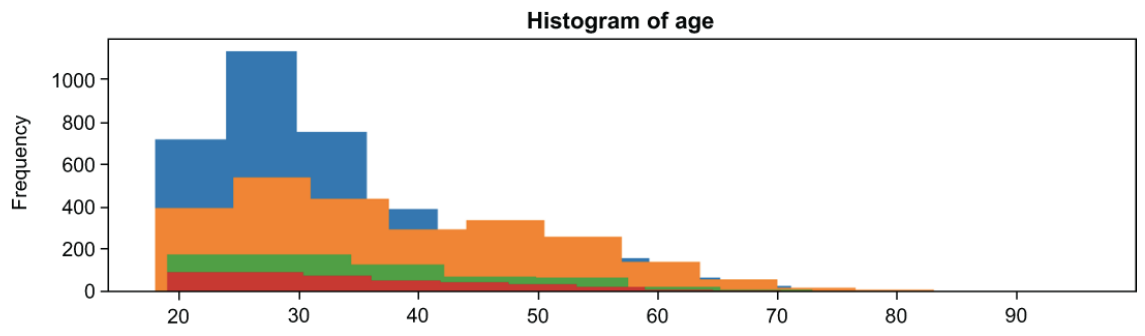


Figure 4.16 Distribution of age by group. The table on top implies that age distribution is not drastically different across groups, thereby implying less impact to disparate treatment and impact. It is worth noting that the average and the median age of African Americans is about 10%-15% younger than those identified in the other categories, which is probably why we are seeing a strong correlation between our age column and our African American identifying column.

Let's turn our attention to `priors_count` and do the same printout. When we do, we will see some stark contrasts from age, as seen in figure 4.17.

```
compas_df.groupby('race')['priors_count'].plot(
    figsize=(20,5),
    kind='hist', xlabel='Count of Priors',
title='Histogram of Priors'
)                                                                   ❶
compas_df.groupby('race')['priors_count'].describe()
```

❶ Priors is extremely skewed by looking at the differences in mean/median/std across the racial categories.

| race | count | mean | std | min | 25% | 50% | 75% | max |
|---|---|---|---|---|---|---|---|---|
| **African American** | 3696.0 | 4.438853 | 5.579835 | 0.0 | 1.0 | 2.0 | 6.0 | 38.0 |
| **Caucasian** | 2454.0 | 2.586797 | 3.798803 | 0.0 | 0.0 | 1.0 | 3.0 | 36.0 |
| **Hispanic** | 637.0 | 2.252747 | 3.647673 | 0.0 | 0.0 | 1.0 | 2.0 | 26.0 |
| **Other** | 427.0 | 2.016393 | 3.695856 | 0.0 | 0.0 | 1.0 | 2.5 | 31.0 |

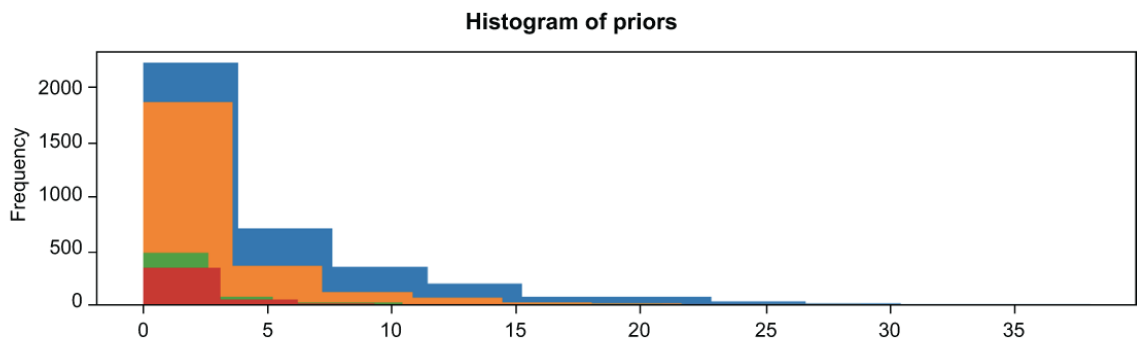


Figure 4.17 At first glance, it may seem like the patterns of priors for all races are similar. Distributions of priors count show a similar right skew between racial groups. However, for reasons out of many people's control, the median and mean priors counts for African Americans are nearly twice that of the other groups.

There are two things to note:

- African American priors are hugely right skewed, as evidenced by the mean being over twice the median.

- African American priors are nearly twice as high as the other racial groups combined, due to a long history of systemic criminal justice issues.

The facts that `priors_count` is so correlated to race, and it is skewed differently for the different racial categories, are huge problems mainly because the ML model can likely pick up on these and bias itself against certain races, simply by looking at the `priors_count` column.

To remedy this, we will create a custom transformer that will modify a column in place by applying the *Yeo-Johnson transformation*—as discussed in our previous chapter—to each racial category's subset of values. This will help to remove the disparate impact that this column would have on our group fairness.

As pseudocode, it would look like this:

```
For each group label:
    Get the subset of priors_count values for that group
    Apply the yeo-johnson transformation to the subset
    Modify the column in place for that group label with the new values
```

By applying the transformation on each subset of values, rather than on the column as a whole, we are forcing each group's set of values to be normal with a mean of 0 and a standard deviation of 1, making it harder for the model to reconstruct a particular group label from a given `priors_count` value. Let's construct a custom scikit-learn transformer to perform this operation, as shown in the following listing.

Listing 4.13 Disparate treatment mitigation through Yeo-Johnson

```
from sklearn.preprocessing import PowerTransformer          ❶
from sklearn.base import BaseEstimator, TransformerMixin    ❶

class NormalizeColumnByLabel(BaseEstimator, TransformerMixin):
    def __init__(self, col, label):
        self.col = col
        self.label = label
        self.transformers = {}

    def fit(self, X, y=None):                                ❷
        for group in X[self.label].unique():
            self.transformers[group] = PowerTransformer(
                method='yeo-johnson', standardize=True
            )
            self.transformers[group].fit(
                X.loc[X[self.label]==group][self.col].values.reshape(-1, 1)
            )
        return self

    def transform(self, X, y=None):                          ❸
        C = X.copy()
        for group in X[self.label].unique():
            C.loc[
                X[self.label]==group, self.col
            ] = self.transformers[group].transform(
                X.loc[X[self.label]==group][self.col].values.reshape(-1, 1)
            )
        return C
```

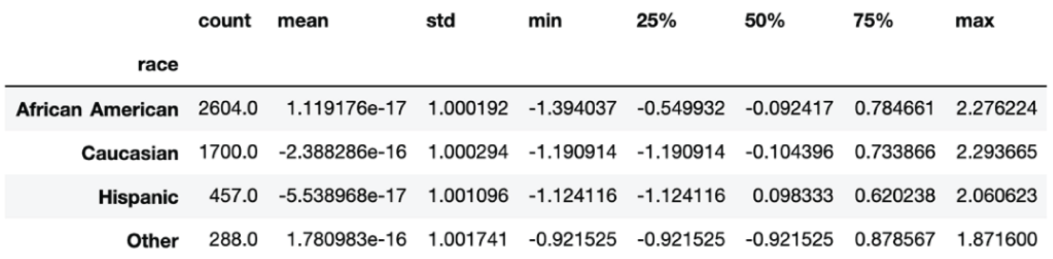
❶ Imports from scikit-learn for our yeo-johnson transformation and base transformer classes

❷ Fit a PowerTransformer for each group label.

❸ When transforming a new DataFrame, we use the transform method of our already fit transformers and modify the DataFrame in place.

Let’s apply our new transformer to our training data to see our priors counts modified (figure 4.18), so each group label has a mean priors count of 0 and a standard deviation of 1:

```
n = NormalizeColumnByLabel(col='priors_count', label='race')

X_train_normalized = n.fit_transform(X_train, y_train)

X_train_normalized.groupby('race')['priors_count'].hist(figsize=(20,5))
X_train_normalized.groupby('race')['priors_count'].describe()
```



| race | count | mean | std | min | 25% | 50% | 75% | max |
|---|---|---|---|---|---|---|---|---|
| African American | 2604.0 | 1.119176e-17 | 1.000192 | -1.394037 | -0.549932 | -0.092417 | 0.784661 | 2.276224 |
| Caucasian | 1700.0 | -2.388286e-16 | 1.000294 | -1.190914 | -1.190914 | -0.104396 | 0.733866 | 2.293665 |
| Hispanic | 457.0 | -5.538968e-17 | 1.001096 | -1.124116 | -1.124116 | 0.098333 | 0.620238 | 2.060623 |
| Other | 288.0 | 1.780983e-16 | 1.001741 | -0.921525 | -0.921525 | -0.921525 | 0.878567 | 1.871600 |

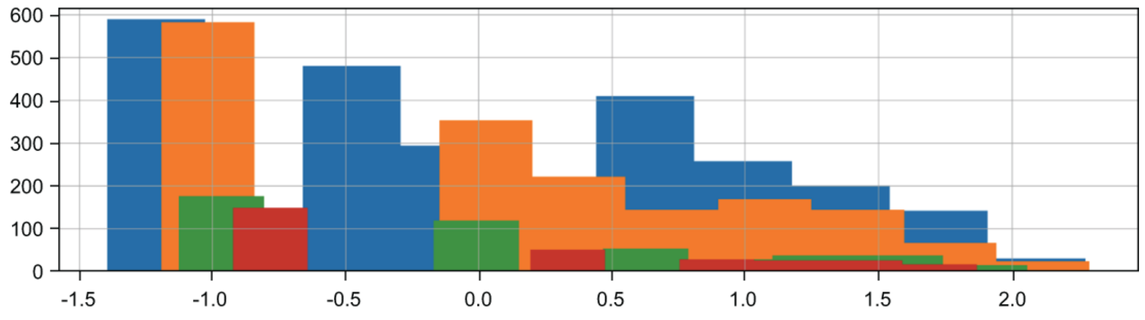


Figure 4.18 After applying the Yeo-Johnson transformation on each subgroup's subset of priors counts, the distributions begin to look much less skewed and different from one another. This will make it difficult for the ML model to reconstruct race from this feature.

Listing 4.14 Our first bias-aware model

```
clf_tree_aware = Pipeline(steps=[
    ('normalize_priors', NormalizeColumnByLabel(
                         col='priors_count', label='race')),    ❶
    ('preprocessor', preprocessor),
    ('classifier', classifier)
])

clf_tree_aware.fit(X_train, y_train)
aware_y_preds = clf_tree_aware.predict(X_test)
```

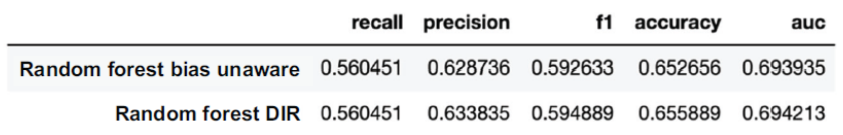
```
exp_tree_aware = dx.Explainer(
    clf_tree_aware, X_test, y_test,
    label='Random Forest DIR', verbose=False)                  ❷
mf_tree_aware = exp_tree_aware.model_fairness(
    protected=race_test, privileged = "Caucasian")

# performance is virtually unchanged overall
pd.concat(
  [exp.model_performance().result for exp in [exp_tree, exp_tree_aware]])

# We can see a small drop in parity loss
mf_tree.plot(objects=[mf_tree_aware], type='stacked')          ❸
```

❶ Add in our new transformer before our preprocessor to fix the priors_-count before doing anything else.

❷ Check out our model performance.

❸ Investigate the change in parity loss.

Our new bias-aware model (listing 4.14) with disparate impact removal is working quite well! We can actually see a small boost in model performance and a small decrease in cumulative parity loss (figure 4.19).



| | recall | precision | f1 | accuracy | auc |
|---|---|---|---|---|---|
| **Random forest bias unaware** | 0.560451 | 0.628736 | 0.592633 | 0.652656 | 0.693935 |
| **Random forest DIR** | 0.560451 | 0.633835 | 0.594889 | 0.655889 | 0.694213 |

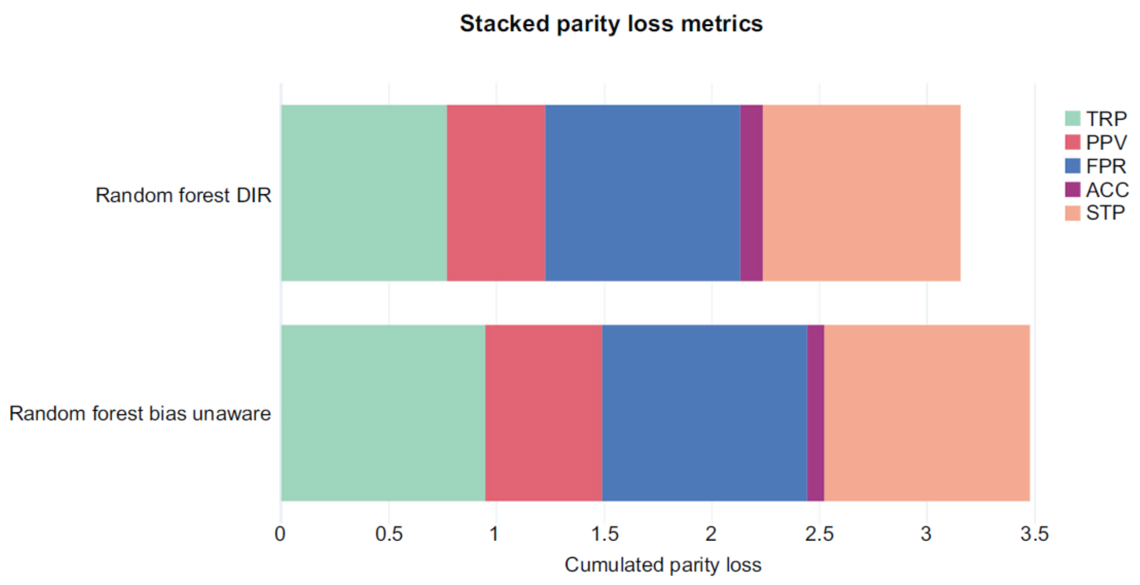


Figure 4.19 The top bar represents the sum of our bias metrics for our bias-aware model, which is seeing a minor boost in model performance (noted in the metric table) in all metrics, except recall, where it is unchanged. The bottom bar shows the original bias-unaware stacked plot we saw earlier. Overall, our new bias-unaware model is performing better in some ML metrics and is showing a decrease in bias based on our parity loss bar chart. We are on the right track!

### 4.6.2 Feature extraction: Learning fair representation implementation using AIF360

Up until now, we haven't done anything to address our model's unawareness of sensitive features. Rather than remove race completely, we are going to use *AI Fairness 360* (AIF360), which is an open source toolkit developed by IBM to help data scientists get access to preprocessing, in-processing, and postprocessing bias mitigation techniques, to apply our first feature extraction technique, called *learning fair representation (LFR)*. The idea of LFR is to map our data *x* onto a new set of features that represent a more fair representation with respect to sensitive variables, including gender and race.

For our use case, we are going to attempt to map our categorical variables (4 / 6 of them representing race) into a new *fairer* vector space that preserves statistical parity and retains as much information as possible from our original *x*.

AIF360 can be a bit tricky to use, as it forces you to use its own version of a DataFrame called the `BinaryLabelDataset`. Listing 4.15 is a custom scikit-learn transformer that will

1. Take in *x*, a DataFrame of binary values, which are created from our categorical preprocessor
2. Convert the DataFrame into a `BinaryLabelDataset`
3. Fit the LFR module from the AIF360 package
4. Transform any new dataset, using the now-fit LFR to map it onto our new fair representation

Listing 4.15 Custom LFR transformer

```
from aif360.algorithms.preprocessing.lfr import LFR
from aif360.datasets import BinaryLabelDataset

class LFRCustom(BaseEstimator, TransformerMixin):
```

```
def __init__(
    self, col, protected_col,
    unprivileged_groups, privileged_groups
):
    self.col = col
    self.protected_col = protected_col
    self.TR = None
    self.unprivileged_groups = unprivileged_groups
    self.privileged_groups = privileged_groups

def fit(self, X, y=None):
    d = pd.DataFrame(X, columns=self.col)
    d['response'] = list(y)

    binary_df = BinaryLabelDataset(                                  ❶
        df=d,
        protected_attribute_names=self.protected_col,
        label_names=['response']
    )

    self.TR = LFR(unprivileged_groups=self.unprivileged_groups,
             privileged_groups=self.privileged_groups, seed=0,
             k=2, Ax=0.5, Ay=0.2, Az=0.2,                            ❷
             verbose=1
            )
    self.TR.fit(binary_df, maxiter=5000, maxfun=5000)
    return self


def transform(self, X, y=None):
    d = pd.DataFrame(X, columns=self.col)
    if y:
        d['response'] = list(y)
        else:
            d['response'] = False

        binary_df = BinaryLabelDataset(
            df=d,
            protected_attribute_names=self.protected_col,
            label_names=['response']
        )
        return self.TR.transform(
            binary_df).convert_to_dataframe()[0].drop(
                ['response'], axis=1)  #B
```

❶ Conversion to and from the AIF360 BinaryLabelDataset object

❷ These parameters can be found on the AIF360 website and were discovered through offline grid searching.

To use our new transformer, we will need to modify our pipeline slightly and make use of the `FeatureUnion` object, as shown in the following listing.

Listing 4.16 Model with disparate impact removal and LFR

```
categorical_preprocessor = ColumnTransformer(transformers=[
    ('cat', categorical_transformer, categorical_features)
])                                                              ❶

#
privileged_groups = [{'Caucasian': 1}]                          ❷
unprivileged_groups = [{'Caucasian': 0}]                        ❷
```

```
lfr = LFRCustom(
    col=[‘African-American’, ‘Caucasian’, ‘Hispanic’, ‘Other’, ‘Male’, ‘M’],
    protected_col=sorted(X_train[‘race’].unique()) ,
    privileged_groups=privileged_groups,
    unprivileged_groups=unprivileged_groups
)

categorical_pipeline = Pipeline([
    (‘transform’, categorical_preprocessor),
    (‘LFR’, lfr),
])

numerical_features = [“age”, “priors_count”]
numerical_transformer = Pipeline(steps=[
    (‘scale’, StandardScaler())
])

numerical_preprocessor = ColumnTransformer(transformers=[
        (‘num’, numerical_transformer, numerical_features)
])                                                              ❶

preprocessor = FeatureUnion([                                   ❸
    (‘numerical_preprocessor’, numerical_preprocessor),
    (‘categorical_pipeline’, categorical_pipeline)
])

clf_tree_more_aware = Pipeline(
    steps=[                                                     ❹
        (‘normalize_priors’, NormalizeColumnByLabel(
            col=’priors_count’, label=’race’)),
        (‘preprocessor’, preprocessor),
        (‘classifier’, classifier)
])
```

```
clf_tree_more_aware.fit(X_train, y_train)

more_aware_y_preds = clf_tree_more_aware.predict(X_test)
```

❶ Isolate the numerical and categorical preprocessor, so we can fit the LFR to the categorical data separately.

❷ Tell AIF360 that rows with a Caucasian label of 1 are privileged, and rows with a Caucasian label of 0 are unprivileged. Right now, the AIF360 package can only support one privileged and one unprivileged group.

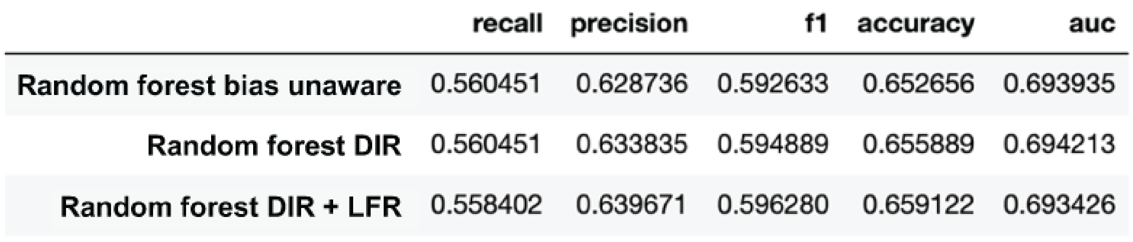
❸ Use FeatureUnion to combine our categorical data and our numerical data.

❹ Our new pipeline will remove disparate impact/treatment via Yeo-Johnson and will apply LFR to our categorical data to address model unawareness.

That was a lot of code to simply apply an LFR module to our DataFrame. Truly, the only reason it was so much was the need to transform our pandas DataFrame into AIF360's custom data object and back. Now that we have fit our model, let's take a final look at our model's fairness:

```
exp_tree_more_aware = dx.Explainer(
    clf_tree_more_aware, X_test, y_test,
    label='Random Forest DIR + LFR', verbose=False)

mf_tree_more_aware = exp_tree_more_aware.model_fairness(
    protected=race_test, privileged="Caucasian")

pd.concat(
    [exp.model_performance().result for exp in [exp_tree,
        exp_tree_aware, exp_tree_more_aware]
])
```

We can see that our final model with disparate impact removal and LFR applied has arguably better model performance than our original baseline model (figure 4.20).



| | recall | precision | f1 | accuracy | auc |
|---|---|---|---|---|---|
| Random forest bias unaware | 0.560451 | 0.628736 | 0.592633 | 0.652656 | 0.693935 |
| Random forest DIR | 0.560451 | 0.633835 | 0.594889 | 0.655889 | 0.694213 |
| Random forest DIR + LFR | 0.558402 | 0.639671 | 0.596280 | 0.659122 | 0.693426 |

Figure 4.20 Our final bias-aware model has improved accuracy, f1, and precision and is seeing only a minor drop in recall and AUC. This is wonderful because it shows that by reducing bias, we have gotten our ML model to perform better in more *classical* metrics, like accuracy, at the same time. Win-win!

We also want to check in on our cumulative parity loss to make sure we are heading in the right direction:

```
Mf_tree.plot(objects=[mf_tree_aware, mf_tree_more_aware], type='stacked')
```

When we check our plot, we can see that our fairness metrics are decreasing as well! This is all-around great news. Our model is not suffering performance-wise from our baseline model, and our model is also acting much more fairly (figure 4.21).

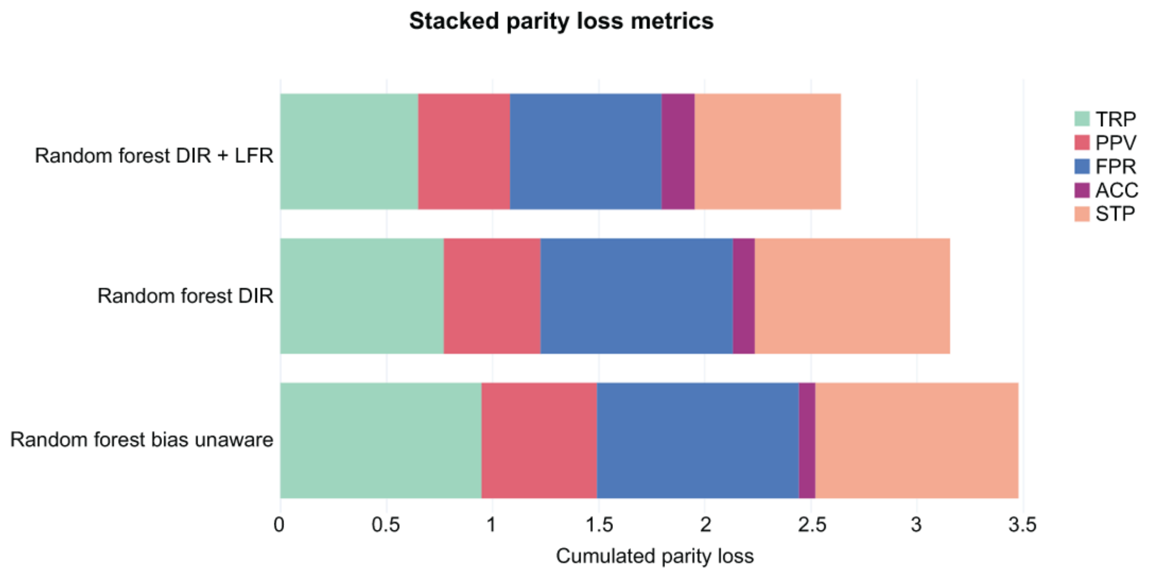


Figure 4.21 Our final, bias-aware model that has disparate impact removal and LFR is the fairest model yet. Again, keep in mind that smaller means less bias, which is generally better for us. We are definitely making some right moves here to see such a drop in bias and an increase in model performance after doing some pretty simple transformations to our data!

Let’s take a look at our Dalex model fairness check one last time. Recall for our unaware model, we had seven numbers outside of our range of (0.8, 1.25), and we had bias detected in four of five metrics.

```
mf_tree_more_aware.fairness_check()          ❶
Bias detected in 3 metrics: TPR, FPR, STP

Conclusion: your model is not fair because 2 or more criteria exceeded
➥ acceptable limits set by epsilon.

Ratios of metrics, based on 'Caucasian'. Parameter 'epsilon' was set to 0.8
➥ and therefore metrics should be within (0.8, 1.25)
                       TPR       ACC       PPV       FPR       STP
African-American  1.626829  1.058268  1.198953  1.538095  1.712329
Hispanic          1.075610  1.102362  0.965096  0.828571  0.893836
Other             0.914634  0.996850  0.806283  1.100000  0.962329
```

❶ 3 / 15 numbers out of the range of (08, 1.25)

We now only have three metrics out of our range, as opposed to the seven previously, and bias is now only being detected in three metrics, rather than four. All in all, our work has seemed to improve our model performance slightly and reduced our cumulative parity loss at the same time.

We've done a lot of work on this data, but would we be comfortable submitting this model as is to be considered an accurate and fair recidivism predictor? *Absolutely not!* Our work in this chapter barely scratched the surface of bias and fairness awareness and only focused on a few preprocessing techniques, not even beginning to discuss, in depth, the other forms of bias mitigation available to us.

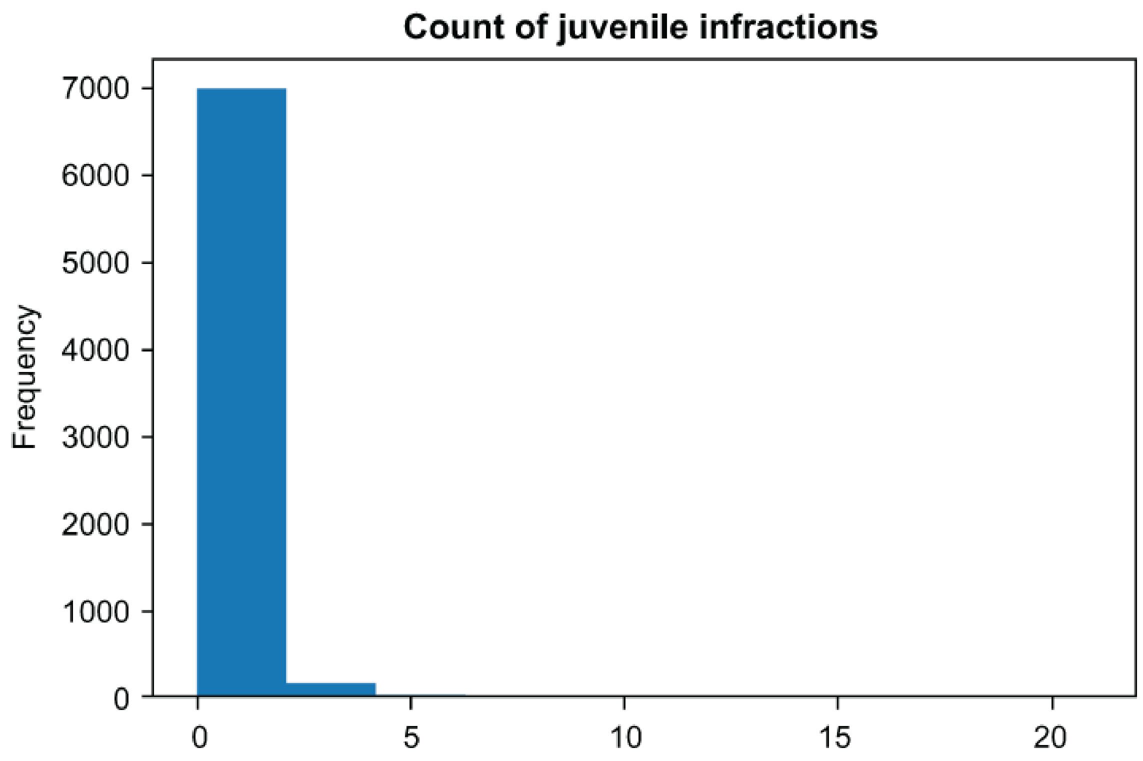
## 7 Answers to exercises

### Exercise 4.1

Plot the distribution of the new `juv_count` feature. What is the arithmetic mean and standard deviation?

Answer:

```
compas_df['juv_count'].plot(
    title='Count of Juvenile Infractions', kind='hist', xlabel='Count'
)
```



To calculate the mean and standard deviation, we could run the following code:

```
compas_df['juv_count'].mean(), compas_df['juv_count'].std()
```

The mean is 0.2675, and the standard deviation is 0.9527.

**Exercise 4.2**

Write Python code to calculate the STP parity loss.

Answer:

```
unpriv_stp = [0.508, 0.261, 0.381]                          ❶
caucasian_stp = 0.285                                       ❷
sum([abs(np.log(u / caucasian_stp)) for u in unpriv_stp])   ❸
```

❶ STP metrics for unprivileged groups

❷ STP metrics for privileged group

❸ 0.956

## ummary

- Model fairness is as important as, if not more important than, model performance alone.

- There are multiple ways of defining fairness in our model, each with its pros and cons:
  - Relying on the unawareness of a model is almost always not enough because of correlating factors in our data.
  - Statistical parity and equalized odds are two common definitions of fairness but can sometimes contradict one another.
- We can mitigate bias before, during, and after training a model.
- Disparate impact removal and learning of fair representation helped our model become more fair and also led to a small bump in model performance.
- Preprocessing alone is *not enough* to mitigate bias. We would also have to work on in-processing and postprocessing methods to further minimize our bias.