Matching Exercise

> Daniela Jiménez Lara
>
> Gavin Li

In this exercise, we'll be evaluating how getting a college degree impacts earnings in the US using matching.

Matching Packages: Python v. R

Just as the best tools for machine learning tend to be in Python since they're developed by CS people (who prefer Python), most of the best tools for causal inference are implemented in R since innovation in causal inference tends to be lead by social scientists using R. As a result, the most well developed matching package is called MatchIt, and is only available in R (though you can always call it from Python using `rpy2`).

In the last couple years, though, a group of computer scientists and statisticians here at Duke have made some great advancements in matching (especially the computational side of things), and they recently released a set of matching packages in both R and Python that we'll be using today. They have some great algorithms we'll use today, but be aware these packages aren't as mature, and aren't general purpose packages yet. So if you ever get deep into matching, be aware you will probably still want to make at least partial use of the R package MatchIt, as well as some other R packages for new innovative techniques (like Matching Frontier estimation), or Adaptive Hyper-Box Matching.

Installing dame-flame.

For this lesson, begin by installing `dame-flame` with `pip install dame-flame` (it's not on conda yet).

DAME is an algorithm that we can use for a version of coarse exact matching. The package only accepts a list of categorical variables, and then attempts to match pairs that match exactly on those variables. That means that if you want to match on, say, age, you have to break it up into categories (say, under 18, 18-29, 30-39, etc. etc.).

(NOTE: As of 2024, their documentation site is weird: click the dropdowns next to headings to see the content, otherwise the documentation looks deserted)

Of course, one cannot always find exact matches on all variables, so what DAME does is:

1. Find all observations that match on *all* matching variables.
2. Figure out which matching variable is least useful in predicting the outcome of interest Y and drops that, then tries to match the remaining observations on the narrowed set of matching variables.
3. This repeats until you run out of variables, all observations are matched, or you hit a stopping run (namely: quality of matches falls below a threshold).

In addition, the lab has also created FLAME, which does the same thing, but employs some tricks to make it *massively* more computationally efficient, meaning it can be used on datasets with millions of observations (which most matching algorithms cannot). It's a little less accurate, but an amazing contribution never the less.

Data Setup

To save you some time and let you focus on matching, I've *pre-cleaned* about one month worth of of data from the US Current Population Survey data we used for our gender discrimination analysis. You can download the data from here, or read it directly with:

```
cps = pd.read_stata(
    "https://github.com/nickeubank/MIDS_Data/blob/master"
    "/Current_Population_Survey/cps_for_matching.dta?raw=true"
)
```

Load the data and quickly familiarize yourself with its contents.

In [ ]:
```
import pandas as pd
from scipy.stats import ttest_ind
import statsmodels.formula.api as smf
import matplotlib.pyplot as plt


cps = pd.read_stata(
    "https://github.com/nickeubank/MIDS_Data/blob/master"
    "/Current_Population_Survey/cps_for_matching.dta?raw=true"
)
```

Getting To Know Your Data

Before you start matching, it is important to examine your data to ensure that matching is feasible (you have some overlap the the features of people in the treated and untreated groups), and also that there is a reason *to* match: either you're unsure about some of the functional forms at play, or your have some imbalance between the two groups.

In [ ]:
```
cps.info()
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 11150 entries, 0 to 11149
Data columns (total 8 columns):
 #   Column           Non-Null Count  Dtype
---  ------           --------------  -----
 0   index            11150 non-null  int32
 1   annual_earnings  5515 non-null   float64
 2   female           11150 non-null  int32
 3   simplified_race  11150 non-null  float64
 4   has_college      11150 non-null  int32
 5   age              11150 non-null  int8
 6   county           11150 non-null  object
 7   class94          11150 non-null  object
dtypes: float64(2), int32(3), int8(1), object(2)
memory usage: 490.1+ KB
```

In [ ]:
```
cps.describe()
```

Out[ ]:

	index	annual_earnings	female	simplified_race	has_college	age
count	11150.000000	5515.000000	11150.000000	11150.000000	11150.000000	11150.000000
mean	152259.915426	41818.028976	0.434439	0.630673	0.397937	43.296951
std	87584.443311	22930.578764	0.495705	1.019898	0.489494	13.327370
min	22.000000	2100.800000	0.000000	0.000000	0.000000	16.000000
25%	77102.000000	27040.000000	0.000000	0.000000	0.000000	32.000000
50%	152730.500000	35360.000000	0.000000	0.000000	0.000000	43.000000
75%	228309.250000	49400.000000	1.000000	1.000000	1.000000	54.000000
max	302307.000000	207979.200000	1.000000	3.000000	1.000000	85.000000

Exercise 1

Show the raw difference of `annual_earnings` between those with and without a college degree (`has_college`). Is the difference statistically significant?

In [ ]:
```
earnings_no_college = cps[cps["has_college"] == 0]["annual_earnings"]

earnings_college = cps[cps["has_college"] == 1]["annual_earnings"]

raw_dif = round(earnings_college.mean() - earnings_no_college.mean())

t_statistic, p_value = ttest_ind(
    earnings_college.dropna().values, earnings_no_college.dropna().values
)
```

```
p_value = round(p_value, 2)
print(
    f"The mean annual earnings for people with college is ${earnings_college.mean():,.0f}\nThe mean annual earn
)

print("The difference is significant as the p-value from the t-test is", p_value)
raw_dif
```

```
The mean annual earnings for people with college is $53,024
The mean annual earnings for people without college is $38,866
The raw difference is $14,158
The difference is significant as the p-value from the t-test is 0.0
```

Out[ ]: 14158

Exercise 2

Next we can check for balance. Check the share of people in different racial groups who have college degrees. Are those differences statistically significant? (Remember how to check for difference in distributions of categorical variables).

Race is coded as White Non-Hispanic (0), Black Non-Hispanic (1), Hispanic (2), Other (3).

Does the data seem balanced?

In [ ]:
```
from scipy.stats import chi2_contingency

cross_t = pd.crosstab(cps["has_college"], cps["simplified_race"])
stat_chi, p_value, d_o_f, exp = chi2_contingency(cross_t.values)

print(cross_t)
print(
    f"Per the p-value of {round(p_value,2)} the differences are statistically significant"
)
```

```
simplified_race   0.0  1.0   2.0  3.0
has_college
0                4282  696  1212  523
1                3340  324   300  473
Per the p-value of 0.0 the differences are statistically significant
```

The data does not seem balanced as the differeces are statistically significant per the Chi2 result.

Exercise 3

One of the other advantages of matching is that even when you have balanced data, you don't have to go through the process of testing out different functional forms to see what fits the data base.

In our last exercise, we looked at the relationship between gender and earnings "controlling for age", where we just put in age as a linear control. Plot a non-linear regression of `annual_earnings` on age (`PolyFit(order=3)` is fine.)

In [ ]:
```
import numpy as np

cps_no_na = cps.dropna(subset=["annual_earnings"])

age = cps_no_na["age"]
earnings = cps_no_na["annual_earnings"]

coefs = np.polyfit(age, earnings, 3)
pol_transform = np.poly1d(coefs)

age_val = np.linspace(age.min(), age.max(), 100)
earning_hat = pol_transform(age_val)


plt.figure(figsize=(12, 6))
plt.scatter(age, earnings, alpha=0.3, label="Actual Data")
plt.plot(age_val, earning_hat, color="purple", label="3 order polynomial fit")
plt.grid(True)
plt.title("Annual earnings per Age")
plt.xlabel("Age")
```

```
plt.ylabel("Annual earnings")
plt.legend()

plt.show()
```

Does the relationship look linear?

Does this speak to why it's nice to not have to think about functional forms with matching as much?

> The relationshio looks polynomial, not linear. Matching can help us find groups with similar behaviors and relationships, thus, hopefuly, allowing us to work with linear relationships.

Matching!

Because DAME is an implementation of exact matching, we have to discretize all of our continuous variables. Thankfully, in this case we only have `age`, so this shouldn't be too hard!

Exercise 4

Create a new variable that discretizes age into a single value for each decade of age.

Because CPS only has employment data on people 18 or over, though, include people who are 18 or 19 with the 20 year olds so that group isn't too small, and if you see any other really small groups, please merge those too.

In [ ]:
```
cps["age"].value_counts()
cps["age"].min()
cps = cps[cps["age"] >= 18]
cps["age_cat"] = cps["age"].apply(lambda i: 20 if i < 20 else i)
cps["age_cat"] = cps["age_cat"] // 10 * 10
cps["age_cat"].value_counts()
```

Out[ ]:
```
age_cat
30    2760
40    2551
50    2397
20    1976
60    1236
70     173
80      43
Name: count, dtype: int64
```

Exercise 5

We also have to covert our string variables into numeric variables for DAME, so convert `county` and `class94` to a numeric vector of intergers.

(Note: it's not clear whether `class94` belongs: if it reflects people choosing fields based on passion, it belongs; if people choose certain jobs because of their degrees, its not something we'd actually want in our regression.

Hint: if you use `pd.Categorical` to convert you var to a categorical, you can pull the underlying integer codes with `.codes`.

In [ ]:
```
cps["class94_cat"] = pd.Categorical(cps["class94"]).codes
cps["county_cat"] = pd.Categorical(cps["county"]).codes
cps
```

Out[ ]:

	index	annual_earnings	female	simplified_race	has_college	age	county	class94	age_cat	class94_cat	co
0	151404	NaN	1	3.0	1	30	0-WV	Private, For Profit	30	3	
1	123453	NaN	0	0.0	0	21	251-TX	Private, For Profit	20	3	
2	187982	NaN	0	0.0	0	40	5-MA	Self-Employed, Unincorporated	40	6	
3	122356	NaN	1	0.0	1	27	0-TN	Private, Nonprofit	20	4	
4	210750	42900.0	1	0.0	0	52	0-IA	Private, For Profit	50	3	
...	...	...	...	...	...	...	...	...	...	...	
11145	178199	37440.0	0	0.0	0	29	13-AZ	Private, For Profit	20	3	
11146	40843	NaN	1	0.0	1	52	35-NJ	Private, For Profit	50	3	
11147	164534	26000.0	0	1.0	0	53	0-MS	Government - State	50	2	
11148	106816	NaN	0	0.0	1	35	1-DC	Private, For Profit	30	3	
11149	40371	NaN	0	0.0	1	58	15-NH	Private, Nonprofit	50	4	

11136 rows × 11 columns

Let's Do Matching with DAME

Exercise 6

First, drop all the variables you *don't* want in matching (e.g. your original `age` variable), and any observations for which `annual_earnings` is missing.

You will probably also have to drop a column named `index`: DAME will try and match on ANY included variables, and so because there was a column called `index` in the data we imported, if we leave it in DAME will try (and obviously fail) to match on index.

Also, it's best to reset your index, as `dame_flame` using index labels (e.g., the values in `df.index`) to identify matches. So you want to be sure those are unique.

In [ ]:
```
cps = cps.dropna(subset=["annual_earnings"])
cps = cps.drop(columns=["index", "age", "county", "class94"])
```

In [ ]:
```
cps.reset_index(drop=True, inplace=True)
```

Exercise 7

The syntax of `dame_flame` is similar to the syntax of `sklearn`. If you start with a dataset called `my_data` with a `treat` variable with treatment assignment and an `outcome` variable for my outcome of interest (Y), the syntax to do basic matching would be:

```
import dame_flame
model = dame_flame.matching.DAME(
    repeats=False,
    verbose=3,
    want_pe=True,
    stop_unmatched_t=True,
)
model.fit(
    for_matching,
    treatment_column_name="has_college",
    outcome_column_name="annual_earnings",
)
result = model.predict(for_matching)
```

Where the arguments:

- `repeats=False` says that I only want each observation to get matched once. We'll talk about what happens if we use `repeats=True` below.
- `verbose=3` tells dame to report everything it's doing as it goes.
- `want_pe` says "please include the predictive error in your printout at each step". This is a measure of match quality.
- `stop_unmatched_t` says "once you've matched all the treatment units, you can stop."

So run DAME on your data!

In [ ]:
```
import dame_flame

model = dame_flame.matching.DAME(
    repeats=False,
    verbose=3,
    want_pe=True,
    stop_unmatched_t=True,
)
model.fit(
    cps,
    treatment_column_name="has_college",
    outcome_column_name="annual_earnings",
)
result = model.predict(cps)
```

```
Completed iteration 0 of matching
        Number of matched groups formed in total:  369
        Unmatched treated units:  645 out of a total of  1150 treated units
        Unmatched control units:  3180 out of a total of  4355 control units
        Number of matches made this iteration:  1680
        Number of matches made so far:  1680
        Covariates dropped so far:  set()
        Predictive error of covariate set used to match:  1199886642.049121
Completed iteration 1 of matching
        Number of matched groups formed in total:  494
        Unmatched treated units:  26 out of a total of  1150 treated units
        Unmatched control units:  185 out of a total of  4355 control units
        Number of matches made this iteration:  3614
        Number of matches made so far:  5294
        Covariates dropped so far:  frozenset({'county_cat'})
        Predictive error of covariate set used to match:  1200005739.113411
Completed iteration 2 of matching
        Number of matched groups formed in total:  494
        Unmatched treated units:  26 out of a total of  1150 treated units
        Unmatched control units:  185 out of a total of  4355 control units
        Number of matches made this iteration:  0
        Number of matches made so far:  5294
        Covariates dropped so far:  frozenset({'simplified_race'})
        Predictive error of covariate set used to match:  1205378823.3674634
Completed iteration 3 of matching
        Number of matched groups formed in total:  506
        Unmatched treated units:  8 out of a total of  1150 treated units
        Unmatched control units:  132 out of a total of  4355 control units
        Number of matches made this iteration:  71
        Number of matches made so far:  5365
        Covariates dropped so far:  frozenset({'county_cat', 'simplified_race'})
        Predictive error of covariate set used to match:  1205391583.8579605
Completed iteration 4 of matching
        Number of matched groups formed in total:  506
        Unmatched treated units:  8 out of a total of  1150 treated units
        Unmatched control units:  132 out of a total of  4355 control units
        Number of matches made this iteration:  0
        Number of matches made so far:  5365
        Covariates dropped so far:  frozenset({'class94_cat'})
        Predictive error of covariate set used to match:  1205712999.3178656
Completed iteration 5 of matching
        Number of matched groups formed in total:  509
        Unmatched treated units:  5 out of a total of  1150 treated units
        Unmatched control units:  124 out of a total of  4355 control units
        Number of matches made this iteration:  11
        Number of matches made so far:  5376
        Covariates dropped so far:  frozenset({'class94_cat', 'county_cat'})
        Predictive error of covariate set used to match:  1205820721.6675985
Completed iteration 6 of matching
        Number of matched groups formed in total:  510
        Unmatched treated units:  4 out of a total of  1150 treated units
        Unmatched control units:  123 out of a total of  4355 control units
        Number of matches made this iteration:  2
        Number of matches made so far:  5378
        Covariates dropped so far:  frozenset({'class94_cat', 'simplified_race'})
        Predictive error of covariate set used to match:  1211243995.3580637
Completed iteration 7 of matching
        Number of matched groups formed in total:  512
        Unmatched treated units:  0 out of a total of  1150 treated units
        Unmatched control units:  114 out of a total of  4355 control units
        Number of matches made this iteration:  13
        Number of matches made so far:  5391
        Covariates dropped so far:  frozenset({'class94_cat', 'simplified_race', 'county_cat'})
        Predictive error of covariate set used to match:  1211256886.777822
5391 units matched. We finished with no more treated units to match
```

Interpreting DAME output

The output you get from doing this *should* be reports from about 8 iterations of matching. In each iteration, you'll see a description of the number of matches made in the iteration, the number of treatment units still unmatched, and the number of control units unmatched.

In the first iteration, the algorithm tries to match observations that match on *all* the variables in your data. That's why in the first iteration, you see the set of variables being dropped is an empty set (`Covariates dropped so far: set()`) — it *hasn't* dropped any variables:

```
Completed iteration 0 of matching
        Number of matched groups formed in total:  370
        Unmatched treated units:  644 out of a total of  1150 treated units
        Unmatched control units:  3187 out of a total of  4365 control units
        Number of matches made this iteration:  1684
        Number of matches made so far:  1684
        Covariates dropped so far:  set()
        Predictive error of covariate set used to match:  1199312680.0957854
```

(Note depending on how you binned ages, you may get slightly different results than this)

But as we can see from this output, the algorithm found 1,684 perfect matches—pairs of observations (one treated, one untreated) that had *exactly* the same value of all the variables we included. But we also see we still have 644 *unmatched* treated units, so what do we do?

The answer is that if we want to match more of our treatment variables, we have to try and match on a subset of our variables.

But what variable should we drop? This is the secret sauce of DAME. DAME picks the variables to drop by trying to predict our outcome Y using all our variables (by default using a ridge regression), then it drops the matching variable that is contributing the least to that prediction. Since our goal in matching is to eliminate baseline differences ($E(Y_0|D=1) - E(Y_1|D=0)$), dropping the covariates least related to Y makes sense.

As a result, in the second iteration (called iteration 1, since it uses 0-based indexing), we see that the variable it drops first is `county`, and it's subsequently able to make another 3,626 new matches on the remaining variables!

```
Completed iteration 1 of matching
        Number of matched groups formed in total:  494
        Unmatched treated units:  25 out of a total of  1150 treated units
        Unmatched control units:  180 out of a total of  4365 control units
        Number of matches made this iteration:  3626
        Number of matches made so far:  5310
        Covariates dropped so far:  frozenset({'county'})
        Predictive error of covariate set used to match:  1199421883.1095908
```

And so DAME continues until its matched all treated observations, and even then it keeps going to evaluate different covariates it might exclude.

Exercise 8

Congratulations! You just on your first one-to-many matching!

The next step is to think about which of the matches that DAME generated are good enough for inclusion in our analysis. As you may recall, one of the choices you have to make as a researcher when doing matching is how "good" a match has to be in order to be included in your final data set. By default, DAME will keep dropping matching variables until it has been able to match all the treated observations or runs out of variables. It will do this no matter how bad the matches start to become -- if it ends up with the treated observation and a control observation that can only be matched on gender, it will match them just on gender, even though we probably don't think that that's a "good" match.

The way to control this behavior is to tell DAME when to stop manually using the `early_stop_iterations` argument to tell the matching algorithm when to stop.

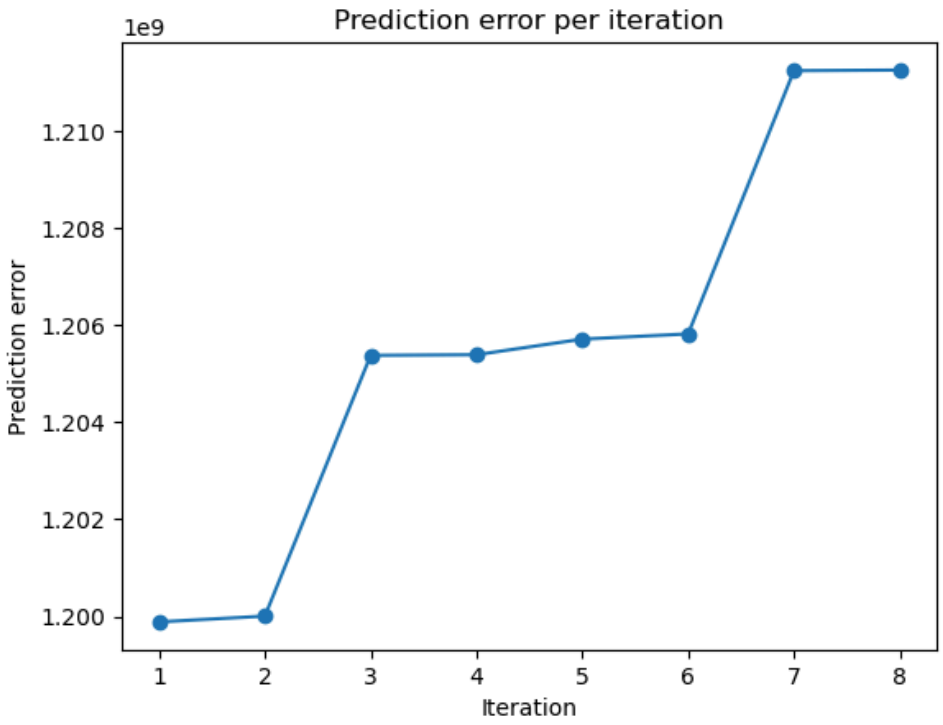
So when is a good time to stop? There's no objective or "right" answer to that question. It fundamentally comes down to a trade-off between bias (which gets higher is you allow more low quality matches into your data) and variance (which will go down as you increase the number of matches you keep).

But one way to start the process of picking a cut point is to examine how the quality of matches evolves over iterations. DAME keeps this information in `model.pe_each_iter`. This shows, for each iteration, the "prediction error" resulting from dropping the variables excluded in each step. This "prediction error" is the difference in the mean-squared error of regressing Y on our matching variables (by default in a ridge regression) with all variables versus with the subset being used for matching in a given iteration. By design, of course, this is always increasing.

To see how this evolves, plot your `pe` against iteration numbers. You can also see the `pe` values for each iteration reported in the output from when DAME ran above if you want to make your you're lining up the errors with iterations right.

Are there any points where the match quality seems to fall off dramatically?

In [ ]:
```
pe = model.pe_each_iter
iter = range(1, 9)
plt.plot(iter, pe, marker="o", linestyle="-")
plt.xlabel("Iteration")
plt.ylabel("Prediction error")
plt.title("Prediction error per iteration")
plt.show()
```



> The match quality falls dramatically at two points:
>
> - On the second iteration and then stablizes
> - On the sixth interation where it drops even more dramatically than before

Exercise 9

Suppose we want to ensure we have at least 5,000 observations in our matched data—where might you cut off the data to get a sample size of at least that but before a big quality falloff?

> The cutoff to ensure at least 5,000 observations before a big quality falloff can be done between the second and sixth iteration, as they have over 5000 matches. However the second one has the lowest error among them (even when the error differences in this range of iteration is very low). The second iteration consists of 5294 matches.

Exercise 10

Re-run your matching, stopping at the point you picked above using `early_stop_iterations`.

In [ ]:
```
model_d = dame_flame.matching.DAME(
    repeats=False,
    verbose=3,
    want_pe=True,
    stop_unmatched_t=True,
    early_stop_iterations=1,
)
model_d.fit(
    cps,
    treatment_column_name="has_college",
    outcome_column_name="annual_earnings",
)
result_d = model_d.predict(cps)
```

```
Completed iteration 0 of matching
        Number of matched groups formed in total:  369
        Unmatched treated units:  645 out of a total of  1150 treated units
        Unmatched control units:  3180 out of a total of  4355 control units
        Number of matches made this iteration:  1680
        Number of matches made so far:  1680
        Covariates dropped so far:  set()
        Predictive error of covariate set used to match:  1199886642.049121
Completed iteration 1 of matching
        Number of matched groups formed in total:  494
        Unmatched treated units:  26 out of a total of  1150 treated units
        Unmatched control units:  185 out of a total of  4355 control units
        Number of matches made this iteration:  3614
        Number of matches made so far:  5294
        Covariates dropped so far:  frozenset({'county_cat'})
        Predictive error of covariate set used to match:  1200005739.113411
5294 units matched. We stopped after iteration 1
Completed iteration 1 of matching
        Number of matched groups formed in total:  494
        Unmatched treated units:  26 out of a total of  1150 treated units
        Unmatched control units:  185 out of a total of  4355 control units
        Number of matches made this iteration:  3614
        Number of matches made so far:  5294
        Covariates dropped so far:  frozenset({'county_cat'})
        Predictive error of covariate set used to match:  1200005739.113411
5294 units matched. We stopped after iteration 1
```

Getting Back a Dataset

OK, my one current complaint with DAME is that it doesn't just give you back a nice dataset of your matches for analysis. If we look at our results — `matches` — it's *almost* what we want, except it has dropped our treatment and outcome columns, and put a string `*` in any entry where a value *wasn't* used for matching:

	female	simplified_race	county	class94	discretized_age
0	1.0	0.0	10.0	3.0	5.0
1	0.0	2.0	*	3.0	3.0
2	0.0	0.0	8.0	3.0	6.0
3	0.0	0.0	*	1.0	4.0
4	0.0	0.0	24.0	3.0	3.0

So for now (though I think this will get updated in the package), we'll have to do it ourselves! Just copy-paste this:

```
def get_dataframe(model, result_of_fit):

    # Get original data
    better = model.input_data.loc[result_of_fit.index]
    if not better.index.is_unique:
        raise ValueError("Need index values in input data to be unique")

    # Get match groups for clustering
```

```
    better["match_group"] = np.nan
    better["match_group_size"] = np.nan
    for idx, group in enumerate(model.units_per_group):
        better.loc[group, "match_group"] = idx
        better.loc[group, "match_group_size"] = len(group)

    # Get weights. I THINK this is right?! At least for with repeat=False?
    t = model.treatment_column_name
    better["t_in_group"] = better.groupby("match_group")[t].transform(np.sum)

    # Make weights
    better["weights"] = np.nan
    better.loc[better[t] == 1, "weights"] = 1  # treaments are 1

    # Controls start as proportional to num of treatments
    # each observation is matched to.
    better.loc[better[t] == 0, "weights"] = better["t_in_group"] / (
        better["match_group_size"] - better["t_in_group"]
    )

    # Then re-normalize for num unique control observations.
    control_weights = better[better[t] == 0]["weights"].sum()

    num_control_obs = len(better[better[t] == 0].index.drop_duplicates())
    renormalization = num_control_obs / control_weights
    better.loc[better[t] == 0, "weights"] = (
        better.loc[better[t] == 0, "weights"] * renormalization
    )
    assert better.weights.notnull().all()

    better = better.drop(["t_in_group"], axis="columns")

    # Make sure right length and values!
    assert len(result_of_fit) == len(better)
    assert better.loc[better[t] == 0, "weights"].sum() == num_control_obs

    return better
```

Exercise 11

Copy-paste that code and run it with your original data, your (fit) model, and what you got back when you ran `result_of_fit`. Then we'll work with the output of that. You should get back a single dataframe of the same length as your original model.

In [ ]:

```
def get_dataframe(model, result_of_fit):
    # Get original data
    better = model.input_data.loc[result_of_fit.index]
    if not better.index.is_unique:
        raise ValueError("Need index values in input data to be unique")

    # Get match groups for clustering
    better["match_group"] = np.nan
    better["match_group_size"] = np.nan
    for idx, group in enumerate(model.units_per_group):
        better.loc[group, "match_group"] = idx
        better.loc[group, "match_group_size"] = len(group)

    # Get weights. I THINK this is right?! At least for with repeat=False?
    t = model.treatment_column_name
    better["t_in_group"] = better.groupby("match_group")[t].transform(np.sum)

    # Make weights
    better["weights"] = np.nan
    better.loc[better[t] == 1, "weights"] = 1  # treaments are 1

    # Controls start as proportional to num of treatments
    # each observation is matched to.
    better.loc[better[t] == 0, "weights"] = better["t_in_group"] / (
        better["match_group_size"] - better["t_in_group"]
```

```
    )

    # Then re-normalize for num unique control observations.
    control_weights = better[better[t] == 0]["weights"].sum()

    num_control_obs = len(better[better[t] == 0].index.drop_duplicates())
    renormalization = num_control_obs / control_weights
    better.loc[better[t] == 0, "weights"] = (
        better.loc[better[t] == 0, "weights"] * renormalization
    )
    assert better.weights.notnull().all()

    better = better.drop(["t_in_group"], axis="columns")

    # Make sure right length and values!
    assert len(result_of_fit) == len(better)
    assert better.loc[better[t] == 0, "weights"].sum() == num_control_obs

    return better


cps_m = get_dataframe(model_d, result_d)
cps_m
```

```
/var/folders/cx/sln5wm7x7bnglq3_q93tj1yw0000gn/T/ipykernel_72301/2765204515.py:16: FutureWarning: The provided c
allable <function sum at 0x10a0b91c0> is currently using SeriesGroupBy.sum. In a future version of pandas, the p
rovided callable will be used directly. To keep current behavior pass 'sum' instead.
  better["t_in_group"] = better.groupby("match_group")[t].transform(np.sum)
```

Out[ ]:

	annual_earnings	female	simplified_race	has_college	age_cat	class94_cat	county_cat	match_group	match_group_s
0	42900.0	1	0.0	0	50	3	10	58.0	
1	31200.0	0	2.0	0	30	3	31	410.0	10
2	20020.0	0	0.0	1	60	3	8	51.0	
3	22859.2	0	0.0	0	40	1	44	423.0	2
4	73860.8	0	0.0	1	30	3	24	105.0	
...	...	...	...	...	...	...	...	...	
5499	22505.6	0	0.0	0	60	3	232	477.0	11
5500	33800.0	1	3.0	0	30	3	247	350.0	
5501	23920.0	0	3.0	0	50	3	272	463.0	3
5502	31200.0	0	2.0	0	20	3	246	347.0	
5503	37440.0	0	0.0	0	20	3	99	277.0	

5294 rows × 10 columns

Check Your Matches and Analyze

Exercise 12

We previously tested balance on `simplified_race` and `county`. Check those again. Are there still statistically significant differences in college education by `simplified_race`?

Note that when you test for this, you'll need to take into account the `weights` column you got back from `get_dataframe`. What DAME does is not actually the 1-to-1 matching described in our readings — instead, however many observations that exact match it finds it puts in the same "group". (These groups are identified in the dataframe you got from `get_dataframe` by the column `match_group`, and the size of each group is in `match_group_size`.)

So to analyze the data, you need to use the `wls` (weighted least squares) function in `statsmodels`. For example, if your data is called `matched_data`, you might run:

```
smf.wls(
    "has_college ~ C(simplified_race)", matched_data, weights=matched_data["weights"]
).fit().summary()
```

In [ ]:
```
smf.wls(
    "has_college ~ C(simplified_race)", cps_m, weights=cps_m["weights"]
).fit().summary()
```

Out[ ]:

WLS Regression Results

Dep. Variable:	has_college	**R-squared:**	0.000
Model:	WLS	**Adj. R-squared:**	-0.001
Method:	Least Squares	**F-statistic:**	4.076e-12
Date:	Fri, 19 Apr 2024	**Prob (F-statistic):**	1.00
Time:	14:16:26	**Log-Likelihood:**	-3726.1
No. Observations:	5294	**AIC:**	7460.
Df Residuals:	5290	**BIC:**	7487.
Df Model:	3		
Covariance Type:	nonrobust		

	coef	std err	t	P>|t|	[0.025	0.975]
Intercept	0.2123	0.007	31.617	0.000	0.199	0.225
C(simplified_race)[T.1.0]	1.665e-16	0.018	9.18e-15	1.000	-0.036	0.036
C(simplified_race)[T.2.0]	1.425e-16	0.019	7.57e-15	1.000	-0.037	0.037
C(simplified_race)[T.3.0]	1.431e-16	0.020	7.02e-15	1.000	-0.040	0.040

Omnibus:	855.409	**Durbin-Watson:**	2.000
Prob(Omnibus):	0.000	**Jarque-Bera (JB):**	1342.687
Skew:	1.231	**Prob(JB):**	2.75e-292
Kurtosis:	2.843	**Cond. No.**	3.96

Notes:
[1] Standard Errors assume that the covariance matrix of the errors is correctly specified.

> As per the P-values of the coefficients, there are no significant differences in college education by race.

Exercise 13

Now use a weighted least squares regression on your matched data to regress annual earnings on *just* having a college eduction. What is the apparent effect of a BA? How does that compare to our initial estimate using the raw CPS data (before matching)?

In [ ]:
```
smf.wls(
    "annual_earnings ~ has_college", cps_m, weights=cps_m["weights"]
).fit().summary()
```

Out[ ]:

WLS Regression Results

Dep. Variable:	annual_earnings	**R-squared:**	0.058
Model:	WLS	**Adj. R-squared:**	0.058
Method:	Least Squares	**F-statistic:**	324.6
Date:	Fri, 19 Apr 2024	**Prob (F-statistic):**	1.79e-70
Time:	14:16:26	**Log-Likelihood:**	-61562.
No. Observations:	5294	**AIC:**	1.231e+05
Df Residuals:	5292	**BIC:**	1.231e+05
Df Model:	1		
Covariance Type:	nonrobust		

	coef	std err	t	P>|t|	[0.025	0.975]
Intercept	3.909e+04	351.762	111.129	0.000	3.84e+04	3.98e+04
has_college	1.375e+04	763.409	18.016	0.000	1.23e+04	1.53e+04

Omnibus:	2917.634	**Durbin-Watson:**	2.004
Prob(Omnibus):	0.000	**Jarque-Bera (JB):**	32782.395
Skew:	2.417	**Prob(JB):**	0.00
Kurtosis:	14.192	**Cond. No.**	2.58

Notes:
[1] Standard Errors assume that the covariance matrix of the errors is correctly specified.

In [ ]:
```
result_match = smf.wls(
    "annual_earnings ~has_college+ female+simplified_race+ age_cat+class94_cat+ county_cat",
    cps_m,
    weights=cps_m["weights"],
).fit()
match_dif = result_match.params["has_college"]
```

In [ ]:
```
# raw_dif
print(
    f"With the matched data, the effect of having a BA in annual earnings is +${match_dif:,.0f}; before matchin
)
```

```
With the matched data, the effect of having a BA in annual earnings is +$13,651; before matching, the difference
was +$14,158
```

Exercise 14

Now include our other matching variables as controls (e.g. all the coefficients you gave to DAME to use). Does the coefficient change?

In [ ]:
```
smf.wls(
    "annual_earnings ~has_college+ female+simplified_race+ age_cat+class94_cat+ county_cat",
    cps_m,
    weights=cps_m["weights"],
).fit().summary()
```

Out[ ]:

WLS Regression Results

Dep. Variable:	annual_earnings	**R-squared:**	0.132
Model:	WLS	**Adj. R-squared:**	0.131
Method:	Least Squares	**F-statistic:**	134.0
Date:	Fri, 19 Apr 2024	**Prob (F-statistic):**	1.67e-158
Time:	14:16:26	**Log-Likelihood:**	-61345.
No. Observations:	5294	**AIC:**	1.227e+05
Df Residuals:	5287	**BIC:**	1.227e+05
Df Model:	6		
Covariance Type:	nonrobust		

	coef	std err	t	P>|t|	[0.025	0.975]
Intercept	4.034e+04	1434.693	28.121	0.000	3.75e+04	4.32e+04
has_college	1.365e+04	735.357	18.563	0.000	1.22e+04	1.51e+04
female	-8490.2550	606.343	-14.002	0.000	-9678.937	-7301.573
simplified_race	-1450.0737	313.639	-4.623	0.000	-2064.935	-835.212
age_cat	262.1644	23.045	11.376	0.000	216.987	307.342
class94_cat	-2144.9617	343.710	-6.241	0.000	-2818.775	-1471.148
county_cat	6.0925	3.436	1.773	0.076	-0.644	12.829

Omnibus:	3038.600	**Durbin-Watson:**	1.989
Prob(Omnibus):	0.000	**Jarque-Bera (JB):**	41725.213
Skew:	2.474	**Prob(JB):**	0.00
Kurtosis:	15.833	**Cond. No.**	610.

Notes:
[1] Standard Errors assume that the covariance matrix of the errors is correctly specified.

In [ ]:
```
result_controls = smf.wls(
    "annual_earnings ~has_college+ female+simplified_race+ age_cat+class94_cat+ county_cat",
    cps_m,
    weights=cps_m["weights"],
).fit()
controls_dif = result_controls.params["has_college"]

print(f"Before matching, the difference was +${raw_dif:,.0f}")
print(
    f"With the matched data, the effect of having a BA in annual earnings is +${match_dif:,.0f};"
)
print(
    f"With the matched data and adding control, the effect of having a BA in annual earnings is +${controls_dif
)
```

```
Before matching, the difference was +$14,158
With the matched data, the effect of having a BA in annual earnings is +$13,651;
With the matched data and adding control, the effect of having a BA in annual earnings is +$13,651;
```

> After adding the control variables, the cofficient does not change.

Exercise 15

If you stopped matching after the second iteration (Iteration 1) back in Exercise 10, you may be wondering if that was a good choice! Let's check by restricting our attention to ONLY exact matches (`iteration = 0`). Run that match.

In [ ]:
```
model_0 = dame_flame.matching.DAME(
    repeats=False,
    verbose=3,
    want_pe=True,
    stop_unmatched_t=True,
    early_stop_iterations=0,
)
model_0.fit(
    cps,
    treatment_column_name="has_college",
    outcome_column_name="annual_earnings",
)
result_0 = model_0.predict(cps)

cps_0 = get_dataframe(model_0, result_0)
```

```
Completed iteration 0 of matching
       Number of matched groups formed in total:  369
       Unmatched treated units:  645 out of a total of  1150 treated units
       Unmatched control units:  3180 out of a total of  4355 control units
       Number of matches made this iteration:  1680
       Number of matches made so far:  1680
       Covariates dropped so far:  set()
       Predictive error of covariate set used to match:  1199886642.049121
1680 units matched. We stopped after iteration 0
```

```
/var/folders/cx/sln5wm7x7bnglq3_q93tj1yw0000gn/T/ipykernel_72301/2765204515.py:16: FutureWarning: The provided c
allable <function sum at 0x10a0b91c0> is currently using SeriesGroupBy.sum. In a future version of pandas, the p
rovided callable will be used directly. To keep current behavior pass 'sum' instead.
  better["t_in_group"] = better.groupby("match_group")[t].transform(np.sum)
```

Exercise 16

Now use a weighted linear regression on your matched data to regress annual earnings on *just* having a college eduction. Is that different from what you had when you allowed more low quality matches?

In [ ]:
```
smf.wls(
    "annual_earnings ~ has_college", cps_0, weights=cps_0["weights"]
).fit().summary()
```

Out[ ]:

WLS Regression Results

Dep. Variable:	annual_earnings	**R-squared:**	0.050
Model:	WLS	**Adj. R-squared:**	0.049
Method:	Least Squares	**F-statistic:**	88.22
Date:	Fri, 19 Apr 2024	**Prob (F-statistic):**	1.84e-20
Time:	14:16:26	**Log-Likelihood:**	-19461.
No. Observations:	1680	**AIC:**	3.893e+04
Df Residuals:	1678	**BIC:**	3.894e+04
Df Model:	1		
Covariance Type:	nonrobust		

	coef	std err	t	P>|t|	[0.025	0.975]
Intercept	3.904e+04	663.363	58.857	0.000	3.77e+04	4.03e+04
has_college	1.136e+04	1209.931	9.393	0.000	8991.472	1.37e+04

Omnibus:	856.094	**Durbin-Watson:**	2.042
Prob(Omnibus):	0.000	**Jarque-Bera (JB):**	6743.610
Skew:	2.261	**Prob(JB):**	0.00
Kurtosis:	11.712	**Cond. No.**	2.42

Notes:
[1] Standard Errors assume that the covariance matrix of the errors is correctly specified.

In [ ]:
```
result_match_0 = smf.wls(
    "annual_earnings ~ has_college", cps_0, weights=cps_0["weights"]
).fit()
match_0_dif = result_match_0.params["has_college"]

print(f"Before matching, the difference was +${raw_dif:,.0f}")
print(
    f"With the matched data (iteration 2), the effect of having a BA in annual earnings is +${match_dif:,.0f};"
)
print(
    f"With the matched data and adding controls, the effect of having a BA in annual earnings is +${controls_di
)
print(
    f"With the matched data (iteration 1), the effect of having a BA in annual earnings is +${match_0_dif :,.0f
)
```

```
Before matching, the difference was +$14,158
With the matched data (iteration 2), the effect of having a BA in annual earnings is +$13,651;
With the matched data and adding controls, the effect of having a BA in annual earnings is +$13,651;
With the matched data (iteration 1), the effect of having a BA in annual earnings is +$11,365;
```

> The difference of annual earnings with and without college is lower when using data from the first iteration of matching than when we used data from the second iteration of matching.

Other Forms of Matching

OK, hopefully this gives you a taste of matching! There are, of course, *many* other permutations to be aware of though.

- Matching with replacement. In this exercise, we set `repeat=False`, so each observation could only end up in our final dataset once. However, if we use `repeat=True`, if an untreated observation is the closest observation to multiple treated observations, it may get put in the dataset multiple times. We can still use this dataset in *almost* the same way, though, except

we have to make use of weights so that if an observation appears, say, twice, each observation has a weight that's 1/2 the weight of an observation only appearing once.

- Matching with continuous variables: DAME is used for exact matching, but if you have lots of continuous variables, you can also match on those. In fact, the Almost Exact Matching Lab also has a library called MALTS that will do matching with continuous variables. That package does something *like* Mahalanobis Distance matching, but ulike Mahalanobis, which calculates the distance between observations in terms of the difference in all the matching variables normalized by each matching variable's standard deviation, MALTS does something much more clever. (Here's the paper describing the technique if you want all the details). Basically, it figures out how well each matching variable predicts our outcome Y, then weights the different variables by their predictive power instead of just normalizing by something arbitrary like their standard deviation. As a result, final matches will prioritize matching more closely on variables that are outcome-relevant. In addition, when it sees a categorical variable, it recognizes that and only pairs observations when they are an exact match on that categorical variable.
- If you're dataset is huge, use `FLAME`: this dataset is small, but if you have lots of observations and lots of matching variable, the computational complexity of this task explodes, so the AEML created FLAME, which works with millions of observations at only a small cost to match quality.