Introduction

This project aims to create a simple interface to create artificial control groups using propensity score matching. It is intended as a way for researchers with less Python experience to have a simple tool that allows them to quickly generate control groups for their needs. Users can install pypsm by cloning and pip installing this directory. It is easy to use by following the example in the jupyter notebook.
Through a single function call, a user can fully create a Propensity Score Matching model and matched dataset as well as export it as a csv.

Users & Use Cases

User Profile

Our targeted users are those who have less experience in Python and hope to quickly and easily create a control group for their study/research with a single function call. For now, the users will need some basic Python understanding to be able to enter in the function and running some command lines; however, in the future, I could extend this program to have a GUI and automatic data cleaning to provide an even simpler interface for the user. In essence, our targeted users might include researchers, policy makers, and students (in a variety of fields from public policy to economics to medicine).

Use Cases

Create an artificial control group that is matched to be similar to treatment group. This is used mainly for observational studies where it is impossible to perform a double blind controlled experiment. After the groups are created, the user can then run whatever analysis they need as they see fit. Steps: 1. User finds dataset that can be split into treatment/control group 2. User cleans dataset according to README & example specifications 3. User installs and imports *pypsm* library 4. User runs *match* function inputting dataset and treatment column as well as determining desired output 5. Program creates and determines optimal Logistic Regression Model, assigns propensity score to each sample, matches each treatment sample to closest control sample, and returns matched dataframe 6. User continues further analysis with matched dataset

Component Specification

Matcher Class

This class is responsible for all the logic of creating the Logistic Regression model and matching each sample of the treatment group with a sample of the control group. **User Input**: - Data = Pandas Dataframe that is cleaned according to specifications in README and example - Treatment Column = String Name of Binary Column where 1 corresponds to treatment group and 0 control group

Compute Matched Data

The only public/interface function in Matcher is **compute_matched_data**.
It takes no further inputs than Matcher.
Outputs: - matched_data = dataframe of fully matched data (i.e. each treatment sample has matched control sample) with all of the initially inputted columns as well as an additional 'SCORE' column indicating the propensity score of each sample

Because most of the logic of my package is hidden in private functions for to create a simple interface for the intended users, I have included the logic behind private methods in this document.

Create Logistic Regression

This is a private method in Matcher. It takes no inputs and returns no outputs. However, this function uses a randomized search to determine the optimal hyperparameters for the LogisticRegression model, fits the model to the data, and sets this to the hidden _final_model attribute.

Set Scores

This is another private method in Matcher. It takes no inputs and returns no outputs. This function adds a SCORE column to the original dataframe and sets its value to the propensity score for each sample based on the Logistic Regression Model returned in Create Logistic Regression.

Match

This is the final private method in Matcher. It takes no inputs. This function uses the SCORE column and greedily matches each sample in the treatment group to its closest value in the control group. **Outputs**: - matched_data = dataframe of fully matched data (i.e. each treatment sample has matched control sample) with all of the initially inputted columns as well as an additional 'SCORE' column indicating the propensity score of each sample

Match

This wrapper function is what I intend users to call when creating their dataset. It relies on the logic of the Matcher class to perform matching, but it also includes some extra functionality in terms of allowing for the input and output of csv files rather than just dataframes. **User Input**: - Data = Pandas Dataframe that is cleaned according to specifications in README and example or csv file location if is_csv flag set to True - Treatment Column = String Name of Binary Column where 1 corresponds to treatment group and 0 control group - is_csv = boolean flag indicating that is True when data is csv (default False) - ouput_csv = directory where to output matched dataframe, None representing no csv output (default None)

Outputs: - matched_data = dataframe of fully matched data (i.e. each treatment sample has matched control sample) with all of the initially inputted columns as well as an additional 'SCORE' column indicating the propensity score of each sample - Optionally also outputs csv file to location specified in output_csv (if not None)

Design Decisions

1. To create a simple interface, I decided to have only two public, accessible functions. The functions that perform the logic behind matching are hidden in private methods. I chose this for two major reasons. The first is that my intended userbase consists of people who have less experience in Python. Thus, I did

not want my interface to be overly complex and alienate those users. Secondly, I am able to change the logic behind my matching in the future without worrying about keeping the logic of matching backwards compatible. For instance, if I later realize that there is another matching method that performs better than my current greedy matching, I can easily change the match function without worrying that some users rely on the old match function.
2. Another important design decision I made was to separate the match wrapper function from the Matcher class. This is due separation of concerns as the Matcher class's concern is simply matching the treatment to a control group assuming an already cleaned pandas dataframe. On the other hand, the match wrapper function can allow for both a pandas and csv input as well as giving the user options on how to output the matched dataset. As I will discuss in a future section, this greatly increases the extensibility of the code as well as provides a simpler interface for users who may not fully understand Python's OOP systems.

Comparison to Scikit-Learn

To achieve similar functionality with sklearn, it would require a large amount of boilerplate code and an understanding of the logic behind propensity score matching from the user. Where with pypsm the user can enter in a single line of code and get a matched dataset, the same function with sklearn would take at least 30-50 lines of code and require the user to create their own Logistic Regression Model before being able to match data based on the output. While for most users pypsm provides a simple and optimal solution to this problem, one advantage of sklearn compared to pypsm is its increased versatility. Users comfortable with Python and looking to use a specific Matching algorithm or different method for choosing the optimal Logistic Regression may determine sklearn to be the better tool for them in that instance. However, users looking for a quick and simple interface who do not necessarily care about the exact method to create matched datasets will definitely find comfort in the ease of pypsm.

Extensibility Discussion

As mentioned earlier, one design decision that shaped pypsm was the separation of the match wrapper function from the Matcher class. This allows for a better separation of concerns that also increases the extensibility of the project. Two future features that could be added to pypsm are a GUI that allows users to select a csv file and output a matched csv file and an automatic data cleaner that allows users to input less treated data. The match wrapper function would easily accommodate these extensions. For the GUI, I could simply have the GUI call on the match wrapper with is_csv set to True and the desired output location in output_csv. For the automatic data cleaner, I could simply add another class that cleans the data and call this class in the match function before calling initiating Matcher. Thanks to the principle of separation of concerns, pypsm is easily extendible.