6-2021

Who Pays? New York State Political Donor Matching with Machine Learning

Annalisa Wilde
The Graduate Center, City University of New York

WHO PAYS?

NEW YORK STATE POLITICAL DONOR MATCHING WITH MACHINE LEARNING

by

ANNALISA WILDE

A master's capstone project submitted to the Graduate Faculty in Data Visualization and Analysis in partial fulfillment of the requirements for the degree of Master of Science, The City University of New York

2021

Who Pays?

New York State Political Donor Matching with Machine Learning

by

Annalisa Wilde

This manuscript has been read and accepted for the Graduate Faculty in Data Analysis and Visualization in satisfaction of the capstone project requirement for the degree of Master of Science.

_____	_____
Date	Michelle McSweeney Capstone Project Advisor
_____	_____
Date	Matthew Gold Executive Officer

THE CITY UNIVERSITY OF NEW YORK

ABSTRACT

Who Pays?

New York State Political Donor Matching with Machine Learning

by

Annalisa Wilde

Advisor: Dr. Michelle McSweeney

Starting with the publicly available data from the New York State Board of Elections, this project first explored the best data processing and algorithmic parameters by which to match the donors. Once an optimal algorithm was generated, the donors were matched in two separate groups: organizations and individuals. The database that stores the matched donors is a product also of this project, with the hope that it will be used by local reporters and advocacy organizations.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

DIGITAL MANIFEST

I. PDF of Whitepaper
II. Zip file containing the contents of the git repository at submission date (https://github.com/rawild/ny-campaign-finance-dedupe)
III. Tar file(s) containing the contents of the PostgreSQL server at submission date with a max size of 1 GBss

TECHNICAL SPECIFICATIONS

This project runs Python 3 modules that manipulate data downloaded from the New York State Board of Elections (NYSBOE) and insert it into a PostgreSQL database. The data from the NYSBOE at publication comes in a .out file type. This can be converted by modules into .txt and .csv files for loading into the PostgreSQL database.

This project has been tested against PostgreSQL v11 and 12. In order to load data into the database the user will need to create a PostgreSQL server and database. The Python module will create the necessary tables.

In addition to Python 3, users will need to install pip, the Python package manager, in order to secure the necessary Python packages. The Python packages needed in order to use all the modules are listed below:

dj_database_url
numpy >= 1.9
pandas
psycopg2
psycopg2.extras
Requests
sklearn
unidecode

Note: this project uses a forked copy of the dedupe package, but for many of the functions you could also use the maintained dedupe package. Installing dedupe with pip will automatically supersede the locally saved version of the dedupe package.

Tuning New York State Donor Matching

Overview

My goal in this project was to match donors between separate filings with the New York State Board of Elections (NYSBOE). Candidates and political action committees (PACs) have unique identifiers when filing with the state, but the donors do not. The absence of unique identifiers means it is difficult to quickly ascertain which donors donate the most across all filings. It's also difficult to understand how those kinds of donors change their behavior overtime.

In order to connect the donors between the filings, I used a machine learning algorithm to create clusters of donors that appear to be the same entity. In this data set, there are actually two distinct types of donor that are best treated as two data sets: individual donors and organization donors. These different data sets require different algorithms to best cluster the donors.

The outcome of this project is a database that has representations of all the filers who reported the donations, the donation filings themselves, the donors, and the clusters for both the individual and the organization donors. Additionally, the project includes a github repository of the code used for the data cleaning before matching, the matching code and some of the jupyter notebooks I used for evaluating the code after matching.

Foundations in Coursework

This project sprang out of coursework for the courses in the Data Analysis and Visualization program, as well as some CUNY Digital Initiatives extracurricular programs. In my first semester of the program I took the Introduction to Data Visualization class and simultaneously participated in the Python Users Group. After learning the basics of python early on in the semester I was able to begin to use python and jupyter notebooks for data analysis for my final project for the Introduction to Data Visualization course.

The following semester I took the Advanced Data Analysis course as well as the Interactive Data Visualization course. In the data analysis course we extensively used python libraries to clean data and process it for machine learning algorithms. In the data visualization course we were iterating through data visualization types with open-ended options for data. I quickly applied the skills I had learned in the analysis class to write the scripts to open and clean New York state political donation data to use in my interactive visualizations. These scripts would later be repurposed for this project.

The machine learning algorithms inspired me to look for libraries of algorithms that I could use to match political donor data for my end-of-semester projects in the Interactive Data Visualization class. This is how I found the dedupe library which is the foundation of the matching in this thesis. Because I was still very new machine learning, in my first usage of the library I did not efficiently approach identifying the best matching algorithms. I ended up doing a lot of manual cleanup to the data in order to create my year-end data visualizations. In that project I limited that matching to just the 86 state-level politicians that represent the New York City boroughs and just the data from January 2010 to January 2020.

In the following semester I worked simultaneously on this thesis and with that smaller dataset in the Advanced Interactive Data Visualization studio class to create even more advanced and complex visualizations of the data. While this thesis is focused on the data and the matching, I am hopeful that I will be able to use this data to power similar data visualizations to the ones I have already made for the Interactive Data Visualization classes.

The Inbound Data

The starting data is the filer and filing data that was available from the NYSBOE on December 30, 2020. Filers include candidates, their authorized campaign committees, PACs, state parties and party committees at the town, village, and county levels. The filer file includes 46,672 different filers with thirteen attributes about the filer, including id, name, type, status, treasurer, and address information. A list is provided in the Data Storage section. The filing data file includes 13,770,589 entries with thirty attributes that describe the transactions. A list of attributes is provided in the Data Storage section. These include the filer that filed the entry, the type of entry, the name and address of the person or organization involved, the amount transacted, and the date of the transaction. The earliest filings are from 1999 and they stretch until filings made in December of 2020

The filings file includes many transaction types listed in Figure 1. Of those, types A-E were parsed out as the “income” transactions to be matched. After filtering for just those transaction types, the number of transactions is reduced to 9,510,546 transactions. I will refer to this file as the donations file.

Figure 1: List of Transaction Types in the Board of Elections Filings

A - Monetary Contributions/Individual & Partnerships
B - Monetary Contributions/Corporate
C - Monetary Contributions/All Other
D - In-Kind Contributions
E - Other Receipts
F - Expenditure/Payments
G - Transfers In
H - Transfers Out
I - Loans Received
J - Loan Repayments
K - Liabilities/Loans Forgiven
L - Expenditure Refunds
M - Contributions Refunded
N - Outstanding Liabilities
O - Partners / Subcontracts
P - Non Campaign Housekeeping Receipts
Q - Non Campaign Housekeeping Expenses
X - A No Activity Statement Was Submitted
Y - A In-Lieu-Of Statement Was Submitted

Of the more than 46,000 filers in the filer file, only 18,989 had transactions listed in the donations file. I will refer to these filers as "recipients." The average number of donations for a recipient was 501, but the median is 63 and the maximum is 517,081. Of these 18,989 recipients, 79.5% of them are candidates or their authorized campaign committees. Eight percent are PACs. See Table 1.

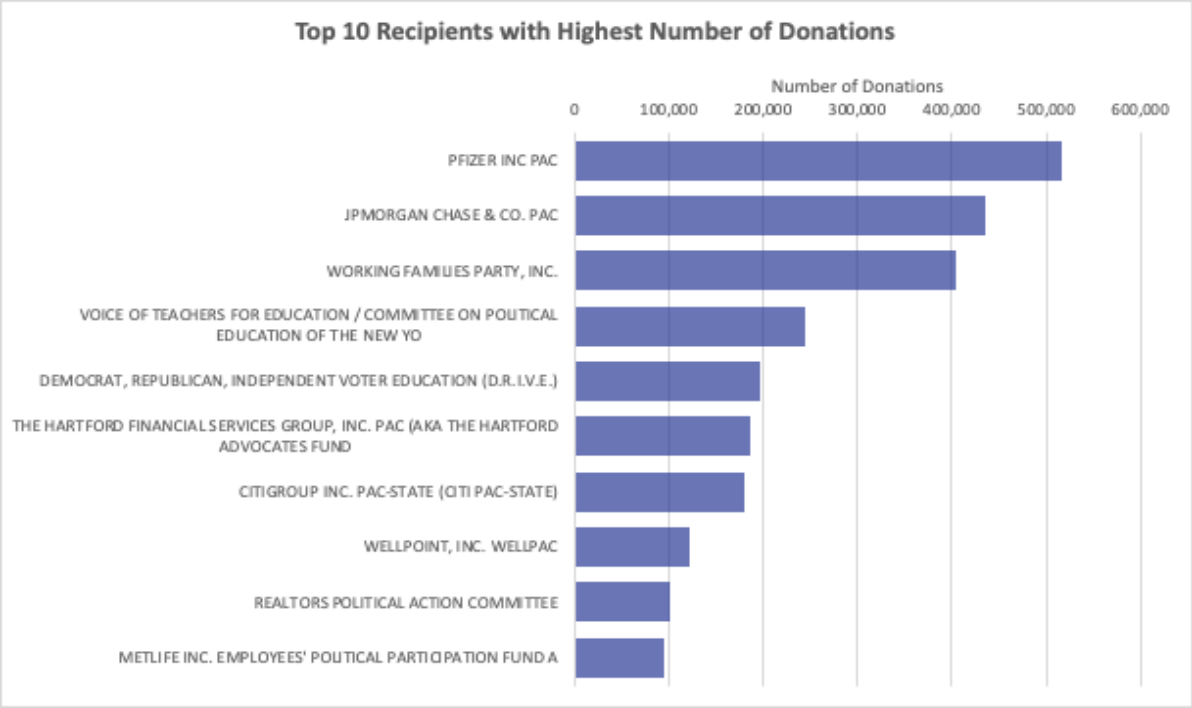
Table 1: Proportion of Recipients by Type of Recipient

Proportion of Recipients by Type of Recipient		
Number of Recipients	Code	Type
1206	0	Candidate
13881	1	Authorized campaign committee
1523	2	PAC
276	3	Constituted County
25	3H	Constituted County House Keeping
157	4	Party County

11	4H	Party County House Keeping
13	5	Constituted State
7	5H	Constituted State House Keeping
16	6	Party State
6	6H	Party State House Keeping
94	7	Duly Constituted Sub-committee of a County Committee
80	7C	Duly Constituted Sub-committee of a County Committee - City
6	7H	Duly Constituted Sub-committee of a County Committee - HouseKeeping
5	7HT	Duly Constituted Sub-committee of a County Committee - HouseKeeping-Town
463	7T	Duly Constituted Sub-committee of a County Committee - Town
22	7V	Duly Constituted Sub-committee of a County Committee - Village
125	8	Unknown
928	9	Others
115	9B	Ballot Issue
30	9U	Undeclared

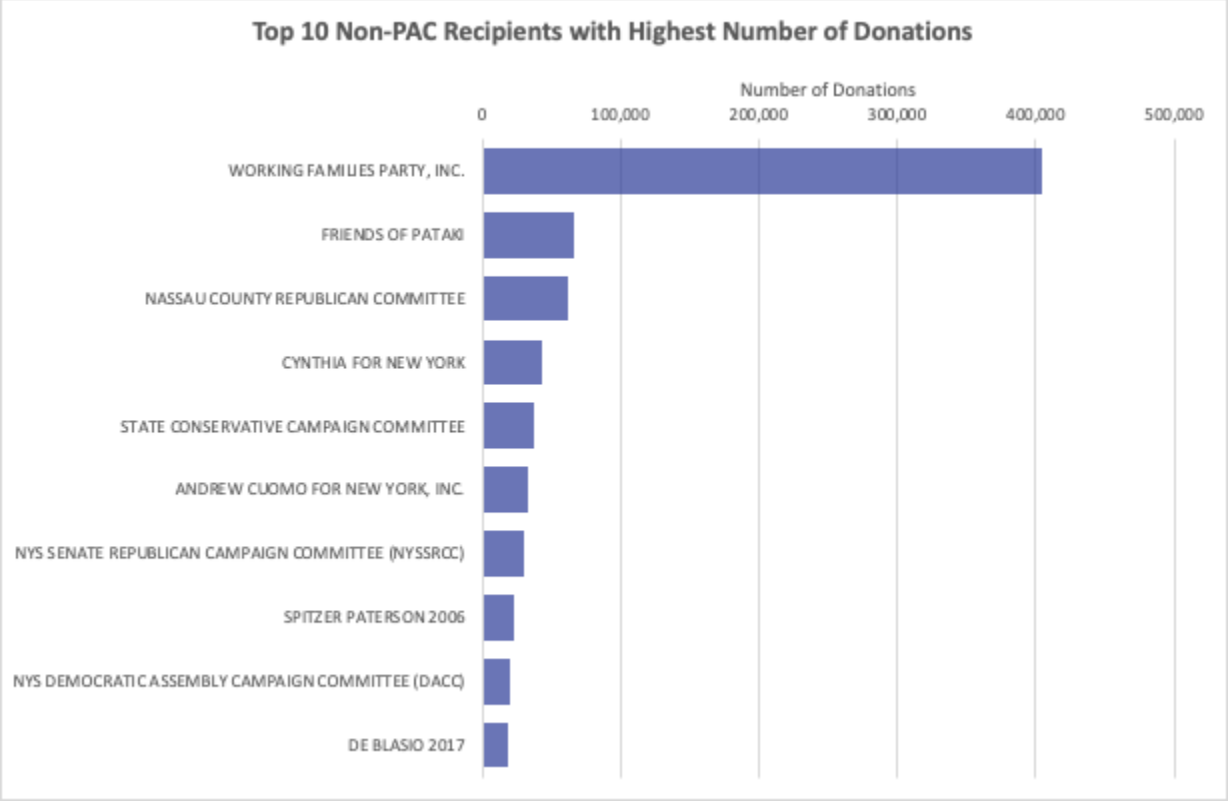
The top ten recipients with the highest number of donations are below. With the exception of the Working Families Party, they are all PACs.

Figure 2: Top 10 Recipients with Highest Number of Donations



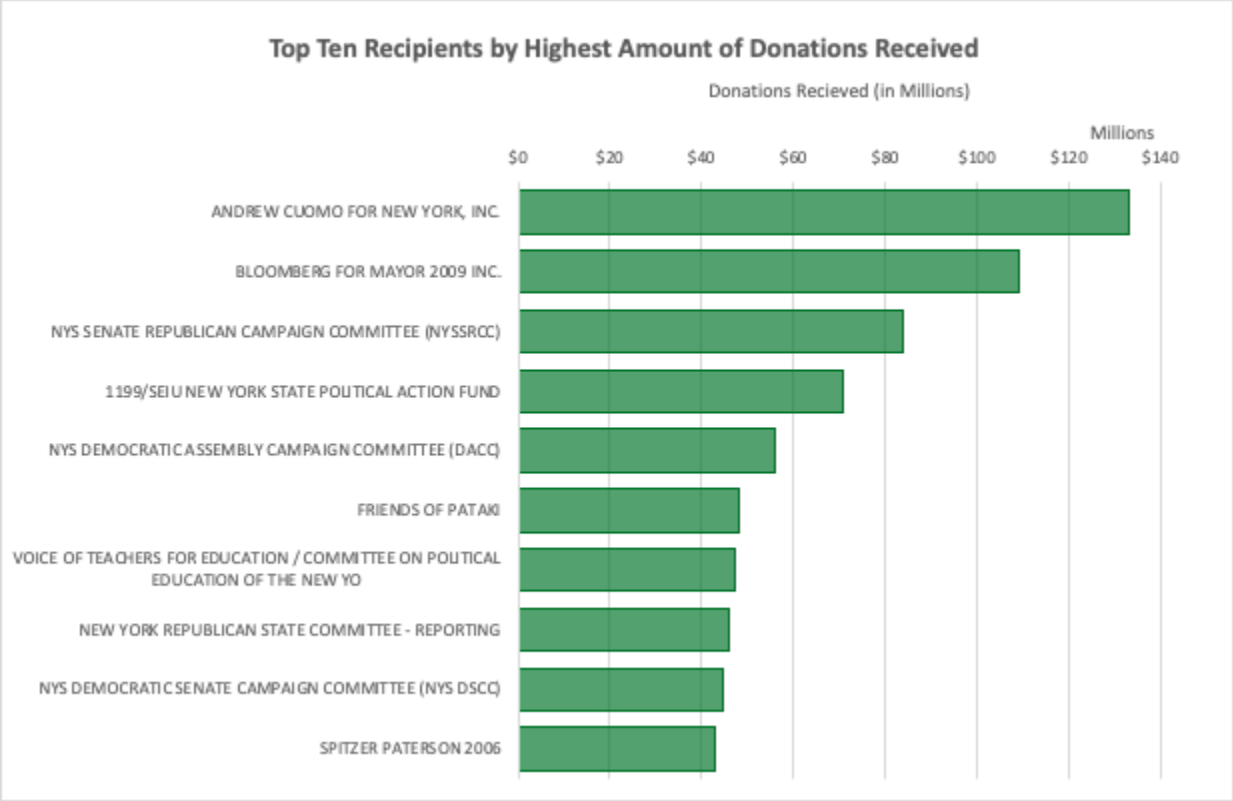
The top ten recipients with the highest number of donations that are not PACs are below.

Figure 3: Top 10 Non-PAC Recipients with Highest Number of Donations



The top ten recipients in terms of dollar amount for both PACs and others are listed below.

Figure 4: Top 10 Recipients by Highest Amount of Donations Received



These lists and statistics about the recipients of political donations are easy to calculate and find with minimal processing. The goal of my project was to enable easy calculation of similar lists and statistics for the donors, not just the recipients.

The Matching Library

The matching library that I am using is a python library called dedupe. It is documented here and the code is available on github here (Forest 2019). Two developers, Forest Gregg and Derek Eder, created the library using the steps laid out in this dissertation by Mikhail Yuryevich Bilenko. Eder and Gregg went on to found a small data analysis firm called DataMade that

maintains the dedupe library. In my code, I have cloned the repository so that I could make tweaks to some of the code that is not exposed by the API.

The Data Storage

The data from the NYSBOE has errors for parsing and is not well-structured for non-proprietary software use. There are a series of steps that need to be done to transform the files into files that can be uploaded into a SQL server from a csv.

The data from the NYSBOE is available to download as two headless comma delimited files, one for the filers and one for the filings. Presumably, they are exported from an Oracle database. Additionally, there are files that include a list of columns and descriptions of some of the codes used in the tables. Below are figures that describe the attributes included for each of the files from the NYSBOE.

Figure 5: List of Attributes in the Board of Elections Filers File

FIELD	LOCATION	TYPE
FILER_ID	01	CHAR
FILER_NAME	02	CHAR
FILER_TYPE	03	CHAR
STATUS	04	CHAR
COMMITTEE_TYPE	05	CHAR
OFFICE	06	INTEGER
DISTRICT	07	INTEGER
TREAS_FIRST_NAME	08	CHAR
TREAS_LAST_NAME	09	CHAR
ADDRESS	10	CHAR
CITY	11	CHAR
STATE	12	CHAR
ZIP	13	CHAR

Figure 6: List of Attributes in the Board of Elections Filings File

FIELD	LOCATION	TYPE	FORMAT
FILER_ID	01	CHAR	
FREPORT_ID	02	CHAR	
TRANSACTION_CODE	03	CHAR	
E_YEAR	04	CHAR	
T3_TRID	05	INTEGER	
DATE1_10	06	DATE	'MM/DD/YYYY'
DATE2_12	07	DATE	'MM/DD/YYYY'
CONTRIB_CODE_20	08	CHAR	
CONTRIB_TYPE_CODE_25	09	CHAR	
CORP_30	10	CHAR	
FIRST_NAME_40	11	CHAR	
MID_INIT_42	12	CHAR	
LAST_NAME_44	13	CHAR	
ADDR_1_50	14	CHAR	
CITY_52	15	CHAR	
STATE_54	16	CHAR	
ZIP_56	17	CHAR	
CHECK_NO_60	18	CHAR	
CHECK_DATE_62	19	DATE	'MM/DD/YYYY'
AMOUNT_70	20	FLOAT	
AMOUNT2_72	21	FLOAT	
DESCRIPTION_80	22	CHAR	
OTHER_RECPT_CODE_90	23	CHAR	
PURPOSE_CODE1_100	24	CHAR	
PURPOSE_CODE2_102	25	CHAR	
EXPLANATION_110	26	CHAR	
XFER_TYPE_120	27	CHAR	
CHKBOX_130	28	CHAR	
CREREC_UID	29	CHAR	
CREREC_DATE	30	DATE	'MM/DD/YYYY HH24:MI:SS'

From the NYSBOE Raw Data to a csv

The files from the NYSBOE are processed with a line by line editor in fix_all_reports.py to try to clean out instances of mismatched quotation marks and things that corrupt the data load into the database. The output of this module is a txt file version of the filings file from the NYSBOE. Any lines that still are identified to be the wrong length by this process are written out to a separate txt file. In the last iteration of processing, about 3,400 of the 13 million transactions were written out to the separate file.

In all_txt_to_csv.py, there is further cleaning of the data values to strip white space and clear out illegal values. In this processing, the full filings file is cut down to the five transaction codes that represent income. The output of this module is two csv files, one for the recipients and one for the donations. These files are ready to be loaded into the database.

Loading the Data into the SQL Server

The module that does the data load into the database is init_postgres_db.py. This code is based heavily on code that is provided in the dedupe examples using dedupe with a PostgreSQL database. First, all the donations data is uploaded directly into the raw_data table. The recipients file is loaded directly into the recipients table.

From the raw_data table, all entries that are exact matches between all the demographic fields are condensed into a donors table. The demographics that must be exact matches for this first round of matching are: first_name, last_name, corp, street, type, city, state and zip.

From this new donors table and the recipients and raw_data tables, the contributions table is created. This table has a donor_id from the donor table, the recipient_id from the recipients, and then all the information associated with the donation such as the amount, date, and type. A final step of the data loading is to create the processed_donors file. This reduces the first_name, last_name, and corp fields to a single field and it generates a flag indicating if the donor is an individual or not. All of the later matching runs are done on the data in the processed_donors table.

After the data is loaded into the database, there is a final cleaning module that is run that normalizes addresses and some common acronyms like “NYC.” The address information is cleaned because review of the clusters after some early matching runs found that many large clusters were splitting on differences in address like “Ave” and “Avenue.” Normalizing the address meant that the clusters were more likely to be combined.

Figure 7: List of SQL Tables that the Data is Loaded Into

raw_table (filer_id, freport_id, transaction_code, e_year, t3_trid, date1, date2, contrib_code, contrib_type, corp, first_name, mid_init, last_name, addr_1, city, state, zip, check_no, check_date, amount, amount2, description, other_recpt_code, purpose_code1, purpose_code2, explanation, xfer_type, chkbox, crerec_uid, crerec_date)

donors (id, name, type, street, city, state, zip)

contributions (donor_id, filer_id, date, type, amount, contributer_type, receipt_type, election_cycle, timing, uuid) Note: generated uuid from filer_id, transaction_code, t3_trid, and date1

recipients (id, name, type, status, committee_type, office, district, treas_first_name, treas_last_name, street, city, state, zip, candidate_id)

processed_donors [All fields of donors with cleaning and normalization done.]

blocking_map_(IND|CORP)(blocking_key, donor_id) [Generated from a selected number of predicates. There is one for the individual clusters and one for the organization clusters.]

enitity_map_(IND|CORP) (cluster_id, donor_id, score) [Generated from hierarchical clustering of the records after they are scored pair-wise. There is one for the individual clusters and one for the organization clusters.]

The Data Matching

The data is matched via three high-level steps. First is blocking, which is used to quickly generate tokens from all of the records and then group together the records that have the same tokens. This is done in order to speed up the matching process and to avoid comparing records that are totally dissimilar from each other. These tokens are generated via similarity predicates that reduce the value of a field to the most meaningful parts for quick comparison. The records and their associated tokens are saved to a table called the blocking_map.

Once the records are grouped together based on these tokens, pairs of records in the same groups are run through a logistic regression classifier to score them as a match. This is the first step of the matching.

After pairs are matched together, the scores are used as a measure of distance between pairs and the pairs are clustered via the hierarchical clustering Python library: fastcluster. The hierarchical clusters are flattened and disagreements in cluster overlap are resolved via a Python library: hcluster. The IDs of these clusters are saved to an entity_map table.

Selection of the correct predicates and classifier for the data set need to be done with training data, which is a sample of the larger data set. In the training data, the correct pairs need to be identified manually. This can be done via a command line interface or a file load. When the training data is loaded, the regularized logistic regression classifier is then fit to match the pairs. The paired training data is also used to calculate which blocking predicates have the highest percent coverage of possible pairs. The classifier and the predicates are saved to a binary settings file so that they can be used for future matching runs.

To identify which predicates and classifiers were best for the individual and organization data sets, I generated a series of them based on different random samples. I also tried using different data model representations of a donor. I added the data types 'Address,' which uses the usaddress Python package to help parse the street address field into tokens, and 'Name,' which uses the probablepeople Python package to help parse the name field into tokens. The efficacy of these representations is explored in the next three sections.

Matching Evaluation

The metrics used to evaluate a clustering run are the total number of clusters, the donor to cluster ratio, the average cluster size, and the biggest cluster size. They are plotted in the sections below. Additionally, I looked at the top ten biggest clusters for each of the runs and evaluated the constituent donors that were a part of those clusters. This second, manual step was the most helpful in identifying which classifier and predicates were actually the best fit for the data set.

The Individual Data Set

The table below lists out the tuples of predicates that were generated for the individual data set. There were two data models used: one with regular data types, and one with the Address and Name data type.

Table 2: Individual Blocking Predicates

Individual Blocking Predicates		
Data Model	Name	Predicates

Regular	settings_IND_0	(LevenshteinCanopyPredicate: (1, name), SimplePredicate: (sameThreeCharStartPredicate, street)), (SimplePredicate: (commonThreeTokens, name), TfidfNGramCanopyPredicate: (0.6, zip)), (SimplePredicate: (commonSixGram, zip), SimplePredicate: (fingerprint, name))
Regular	settings_IND_1	(LevenshteinCanopyPredicate: (2, name), LevenshteinCanopyPredicate: (3, street)), (SimplePredicate: (commonTwoTokens, city), SimplePredicate: (commonTwoTokens, name)), (SimplePredicate: (suffixArray, street), TfidfTextCanopyPredicate: (0.8, name)), (SimplePredicate: (commonTwoTokens, name), TfidfNGramCanopyPredicate: (0.8, name)), (TfidfNGramCanopyPredicate: (0.6, name), TfidfNGramCanopyPredicate: (0.8, street)), (SimplePredicate: (commonThreeTokens, name), SimplePredicate: (oneGramFingerprint, state)), (LevenshteinCanopyPredicate: (2, name), SimplePredicate: (doubleMetaphone, street))
Regular	settings_IND_2	(SimplePredicate: (commonTwoTokens, name), SimplePredicate: (commonTwoTokens, street)), (SimplePredicate: (commonThreeTokens, name), TfidfNGramCanopyPredicate: (0.4, zip)) (ExistsPredicate: (Exists, state), SimplePredicate: (commonIntegerPredicate, name)) (LevenshteinCanopyPredicate: (1, state), SimplePredicate: (commonThreeTokens, name)) (SimplePredicate: (fingerprint, name), TfidfNGramCanopyPredicate: (0.8, street))
Regular	settings_IND_3	(LevenshteinCanopyPredicate: (1, name), SimplePredicate: (sortedAcronym, street)) (SimplePredicate: (commonThreeTokens, name), TfidfNGramCanopyPredicate: (0.6, state)) (TfidfNGramCanopyPredicate: (0.6, name), TfidfNGramCanopyPredicate: (0.8, street)) (SimplePredicate: (hundredIntegersOddPredicate, name), TfidfTextCanopyPredicate: (0.6, name)) (TfidfNGramCanopyPredicate: (0.6, street), TfidfNGramCanopyPredicate: (0.8, name))
Regular	settings_IND_4	(SimplePredicate: (doubleMetaphone, name), SimplePredicate: (oneGramFingerprint, zip)), (TfidfNGramCanopyPredicate: (0.6, name), TfidfNGramCanopyPredicate: (0.6, street)) (SimplePredicate: (commonThreeTokens, name), TfidfNGramCanopyPredicate: (0.8, name))
Special Name and Address	settings_IND_ext_0	((SimplePredicate: (doubleMetaphone, street), TfidfTextCanopyPredicate: (0.8, name)), (SimplePredicate: (fingerprint, name), TfidfNGramCanopyPredicate: (0.8, name)), (SimplePredicate: (commonThreeTokens, name), TfidfTextCanopyPredicate: (0.4, street)), (PartialIndexTfidfNGramCanopyPredicate: (0.8, name, Surname), SimplePredicate: (commonThreeTokens, name)), (LevenshteinCanopyPredicate: (1, state), SimplePredicate: (commonThreeTokens, name)),

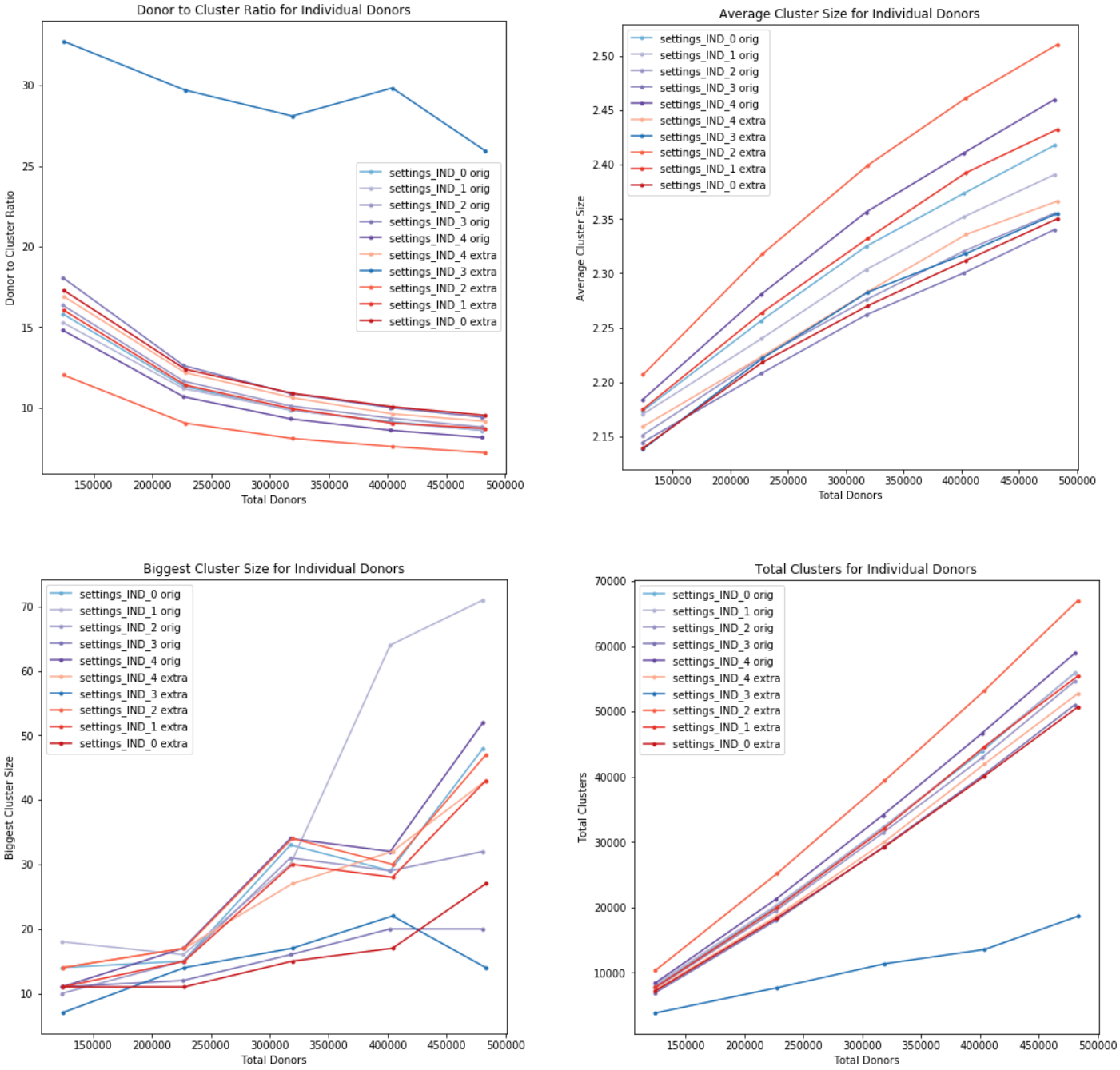
		(SimplePredicate: (commonTwoTokens, name), TfidfNGramCanopyPredicate: (0.8, street)))
Special Name and Address	settings_IND_ext_1	((SimplePredicate: (commonTwoTokens, name), SimplePredicate: (nearIntegersPredicate, zip)), (PartialIndexLevenshteinCanopyPredicate: (2, name, Surname), TfidfNGramCanopyPredicate: (0.8, street)), (PartialIndexTfidfNGramCanopyPredicate: (0.4, name, Surname), TfidfNGramCanopyPredicate: (0.8, name)), (SimplePredicate: (alphaNumericPredicate, name), SimplePredicate: (hundredIntegersOddPredicate, name)), (TfidfNGramCanopyPredicate: (0.8, name), TfidfTextCanopyPredicate: (0.8, street)), (LevenshteinCanopyPredicate: (3, zip), SimplePredicate: (commonThreeTokens, name)), (PartialIndexTfidfTextCanopyPredicate: (0.4, city, StreetName), TfidfNGramCanopyPredicate: (0.6, name)), (PartialIndexTfidfNGramCanopyPredicate: (0.6, name, Surname), TfidfNGramCanopyPredicate: (0.6, street)), (SimplePredicate: (commonTwoTokens, city), TfidfNGramCanopyPredicate: (0.8, name)))
Special Name and Address	settings_IND_ext_2	((SimplePredicate: (suffixArray, street), TfidfTextCanopyPredicate: (0.8, name)), (TfidfNGramCanopyPredicate: (0.6, street), TfidfNGramCanopyPredicate: (0.8, name)), (SimplePredicate: (commonThreeTokens, name), TfidfNGramCanopyPredicate: (0.2, street)), (PartialPredicate: (commonSixGram, city, StreetName), SimplePredicate: (oneGramFingerprint, name)), (PartialIndexTfidfNGramCanopyPredicate: (0.4, name, Surname), SimplePredicate: (commonThreeTokens, street)), (TfidfNGramCanopyPredicate: (0.8, street), TfidfTextCanopyPredicate: (0.4, name)), (PartialIndexLevenshteinCanopyPredicate: (1, name, Surname), SimplePredicate: (commonTwoTokens, name)), (SimplePredicate: (commonTwoTokens, city), TfidfNGramCanopyPredicate: (0.8, name)), (PartialIndexTfidfNGramCanopyPredicate: (0.6, name, Surname), SimplePredicate: (sameSevenCharStartPredicate, street)))
Special Name and Address	settings_IND_ext_3	((SimplePredicate: (fingerprint, name), TfidfNGramCanopyPredicate: (0.8, zip)), (SimplePredicate: (commonThreeTokens, name), TfidfNGramCanopyPredicate: (0.6, name)), (PartialIndexTfidfTextCanopyPredicate: (0.6, name, CorporationName), PartialPredicate: (sameSevenCharStartPredicate, name, CorporationName)), (PartialPredicate: (commonSixGram, name, Surname), TfidfNGramCanopyPredicate: (0.8, street)))

Special Name and Address	settings_IND_ext_4	((LevenshteinCanopyPredicate: (1, name), LevenshteinCanopyPredicate: (2, street)), (SimplePredicate: (nearIntegersPredicate, zip), TfidfNGramCanopyPredicate: (0.6, name)), (SimplePredicate: (commonTwoTokens, name), TfidfNGramCanopyPredicate: (0.6, street)), (SimplePredicate: (commonThreeTokens, name), SimplePredicate: (sameSevenCharStartPredicate, name)), (SimplePredicate: (hundredIntegersOddPredicate, zip), SimplePredicate: (nearIntegersPredicate, name)), (LevenshteinCanopyPredicate: (1, name), PartialPredicate: (suffixArray, city, StreetName)), (PartialPredicate: (sameFiveCharStartPredicate, name, CorporationName), TfidfTextCanopyPredicate: (0.6, name)), (PartialIndexLevenshteinCanopyPredicate: (1, name, CorporationName), TfidfNGramCanopyPredicate: (0.8, name)))

Final Predicates and Classifier

Unfortunately, a review of these predicates and the clusters they generated showed that some of the predicates and classifiers with the regular data model, and most of the predicates and classifiers with the extended data types, return results that included individuals at the same address but with very different names. This was particularly a problem with addresses that were Pfizer workplaces. This is perhaps not surprising given that the Pfizer PAC was the recipient with the highest number of donations with over 500,000. IND_0 and IND_ext_3 had the most diverse results that did not include bad values. I used the classifier from IND_0 because it created larger groups than IND_ext_3, but I added the predicates from IND_ext_3 to those used in IND_0 for the final matching run. These settings are shown in blue in the graphs. The settings with the plain data model are shown in purple and the settings with the extended data model are shown in orange.

Figure 8: Evaluation of Predicates and Classifiers for Individual Donors



The classifier that was the best fit was settings_IND_0 orig, but it performs in an average zone for all of these metrics. Theset of predicates in settings_IND_3 extra, clustered many fewer donor records together, but it caught different clusters than most of the other predicates

did. The data in these charts was generated from running the matching on a series of random samples of the data of increasing size.

The Individual Donor Matching Results

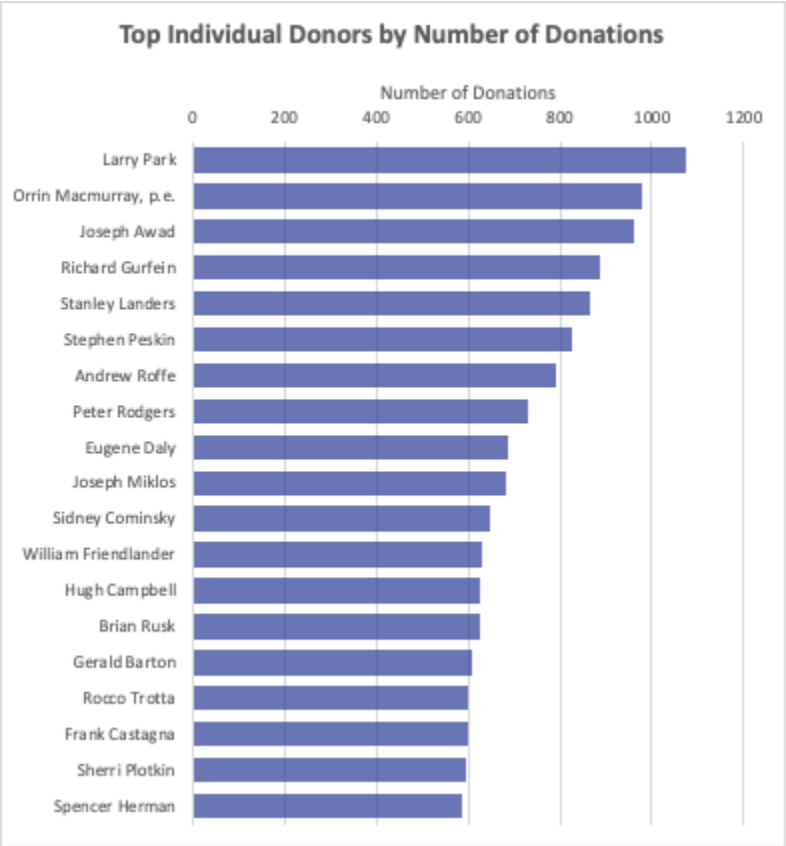
The universe of the individual donors started at 2,813,273. Based on the matching algorithm it has been reduced to 1,900,223 donors who donated $1,648,348,574. Of these donors, 739,295 have been clustered into clusters larger than one. As in the sample data sets, there were a few donors that donated to a large number of candidates and spent a lot of money, but a majority of donors only donated to one campaign. The median donor is one who donated $100 to one campaign one time.

The biggest cluster was formed from 201 separate donor entries that were matched to a person named Lloyd Douglas. All 201 entries appear to be the same person with variations on the same address in Central Harlem. Interestingly, Lloyd Douglas does not appear among the donors who have given the most times or among the donors who have given the most money. The average cluster was just 2.97 donor entries. This reflects the fact that in this data there are a few large clusters, but the majority of the clusters are very small or non-clusters, that is, of size one.

The individual donors with the highest number of contributions are shown below. The individual donor with the highest number of donations is Larry Park, the executive director of the New York State Trial Lawyers Association, which donates under the LawPAC acronym. He had 1,075 donations associated with him. Many other individuals on this list are trial lawyers, or, like Orrin MacMurray, own companies that have contracts with the state. Despite the high

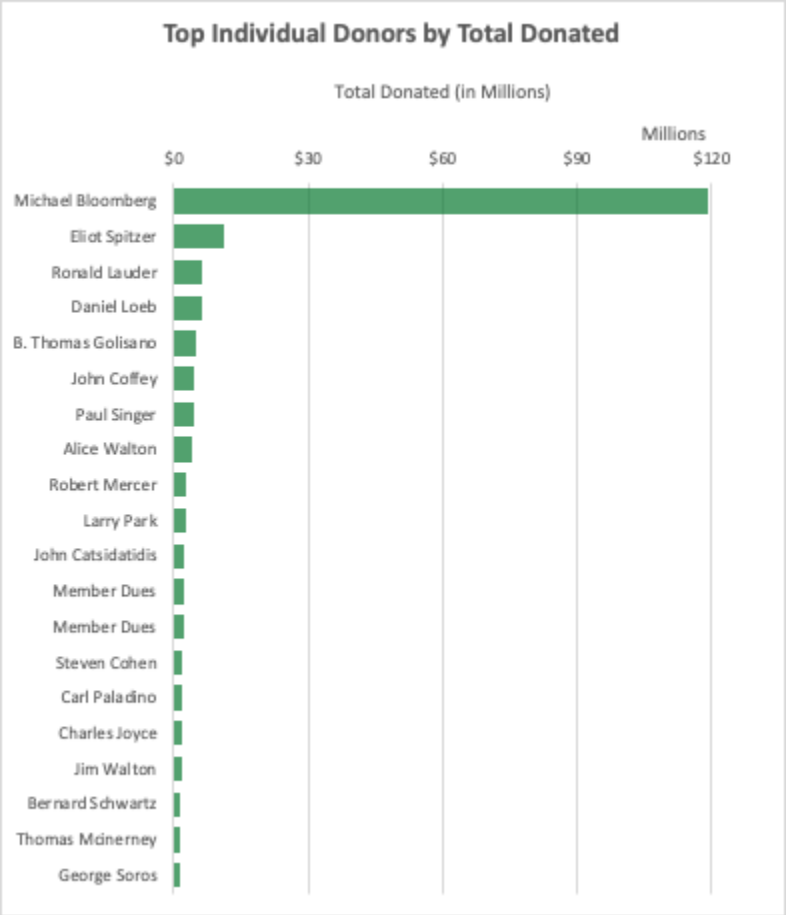
number of donations by top donors, the average number of donations by an individual is four and the median is one. That is, most donors donate only once.

Figure 9: Top Individual Donors by Number of Donations



The individual donors that donated the most amount of money in the time period covered by the data set are listed below. The biggest donor will not surprise New Yorkers. Michael Bloomberg donated to his own mayoral campaign and to many other campaigns, as well. In total in this data set he spent more than $119 million dollars on New York State politics. That is much much more than the average individual donor total which was $895.11. The median individual donation was $100.

Figure 10: Top Individual Donors by Total Donated



The Organization Data Set

The table below lists out the tuples of predicates that were generated for the organization data set. There were two data models used: one with regular data types, and one with the Address and Name data type. The predicates used are in a table below, where blue indicates the predicates that were finally selected, purple indicates the predicates with the regular data model and orange indicates the predicates with the special data model.

Table 3: Organization Blocking Predicates

Organization Blocking Predicates		
Data	Name	Predicates

Model		
Regular	settings_0	(ExistsPredicate: (Exists, zip), SimplePredicate: (sameSevenCharStartPredicate, name)) (SimplePredicate: (commonTwoTokens, name), TfidfTextCanopyPredicate: (0.8, street)) (SimplePredicate: (alphaNumericPredicate, name), TfidfNGramCanopyPredicate: (0.2, street)) (LevenshteinCanopyPredicate: (3, name), SimplePredicate: (firstIntegerPredicate, name)) (SimplePredicate: (wholeFieldPredicate, street), TfidfNGramCanopyPredicate: (0.8, name)) (LevenshteinCanopyPredicate: (4, name), TfidfNGramCanopyPredicate: (0.8, street))
Regular	settings_1	(SimplePredicate: (sameFiveCharStartPredicate, name), TfidfNGramCanopyPredicate: (0.4, name)) (SimplePredicate: (sameThreeCharStartPredicate, name), SimplePredicate: (wholeFieldPredicate, street)) (SimplePredicate: (twoGramFingerprint, street), TfidfNGramCanopyPredicate: (0.4, name)) (SimplePredicate: (commonThreeTokens, name), TfidfNGramCanopyPredicate: (0.8, street))
Regular	settings_2	(SimplePredicate: (doubleMetaphone, state), SimplePredicate: (sameSevenCharStartPredicate, name)) (SimplePredicate: (alphaNumericPredicate, name), SimplePredicate: (firstIntegerPredicate, zip)) (SimplePredicate: (doubleMetaphone, name), TfidfNGramCanopyPredicate: (0.6, name)) (SimplePredicate: (commonFourGram, name), SimplePredicate: (wholeFieldPredicate, street)) (SimplePredicate: (firstIntegerPredicate, name), TfidfNGramCanopyPredicate: (0.4, name))
Regular	settings_3	(ExistsPredicate: (Exists, city), SimplePredicate: (sameSevenCharStartPredicate, name)) (SimplePredicate: (commonFourGram, name), SimplePredicate: (wholeFieldPredicate, street)) (SimplePredicate: (alphaNumericPredicate, name), SimplePredicate: (sortedAcronym, city)) (SimplePredicate: (sortedAcronym, street), TfidfNGramCanopyPredicate: (0.8, name))
Regular	settings_4	(SimplePredicate: (sameSevenCharStartPredicate, name), SimplePredicate: (tokenFieldPredicate, name)) (SimplePredicate: (commonIntegerPredicate, name), TfidfNGramCanopyPredicate: (0.2, street)) (LevenshteinCanopyPredicate: (4, street), SimplePredicate: (commonIntegerPredicate, city)) (TfidfNGramCanopyPredicate: (0.6, name), TfidfNGramCanopyPredicate: (0.6, street)) (SimplePredicate: (suffixArray, name), TfidfNGramCanopyPredicate: (0.6, street))
Special Name and Address	settings_CORP_ext_0	((PartialIndexLevenshteinCanopyPredicate: (3, city, StreetName), SimplePredicate: (sameSevenCharStartPredicate, name)),

		(SimplePredicate: (alphaNumericPredicate, name), SimplePredicate: (sameSevenCharStartPredicate, street)), (PartialIndexTfidfNGramCanopyPredicate: (0.6, name, CorporationName), SimplePredicate: (commonIntegerPredicate, name)), (PartialPredicate: (sameSevenCharStartPredicate, city, StreetName), SimplePredicate: (sortedAcronym, name)), (SimplePredicate: (suffixArray, name), TfidfNGramCanopyPredicate: (0.6, street)), (SimplePredicate: (wholeFieldPredicate, street), TfidfTextCanopyPredicate: (0.8, name)))
Special Name and Address	settings_CORP_ext_1	((SimplePredicate: (commonFourGram, name), SimplePredicate: (sameSevenCharStartPredicate, name)), (SimplePredicate: (alphaNumericPredicate, name), SimplePredicate: (commonTwoTokens, city)), (SimplePredicate: (commonIntegerPredicate, name), TfidfNGramCanopyPredicate: (0.2, name)), (SimplePredicate: (suffixArray, name), SimplePredicate: (wholeFieldPredicate, street)), (SimplePredicate: (firstIntegerPredicate, name), TfidfNGramCanopyPredicate: (0.2, street)), (SimplePredicate: (wholeFieldPredicate, street), TfidfNGramCanopyPredicate: (0.6, name)))
Special Name and Address	settings_CORP_ext_2	((SimplePredicate: (commonSixGram, name), SimplePredicate: (sameThreeCharStartPredicate, name)), (PartialIndexLevenshteinCanopyPredicate: (3, name, CorporationName), SimplePredicate: (alphaNumericPredicate, name)), (PartialPredicate: (alphaNumericPredicate, name, CorporationName), TfidfTextCanopyPredicate: (0.6, street)), (SimplePredicate: (commonThreeTokens, city), SimplePredicate: (hundredIntegersOddPredicate, name)), (SimplePredicate: (fingerprint, street), TfidfNGramCanopyPredicate: (0.6, name)), (TfidfNGramCanopyPredicate: (0.8, street), TfidfTextCanopyPredicate: (0.6, name)), (SimplePredicate: (firstIntegerPredicate, name), SimplePredicate: (firstIntegerPredicate, street)), (PartialIndexTfidfTextCanopyPredicate: (0.4, name, CorporationName), TfidfNGramCanopyPredicate: (0.8, street)), (PartialIndexLevenshteinCanopyPredicate: (2, name, CorporationName), TfidfNGramCanopyPredicate: (0.8, name)))
Special Name and Address	settings_CORP_ext_3	((SimplePredicate: (commonSixGram, name), SimplePredicate: (metaphoneToken, city)), (PartialPredicate: (commonThreeTokens, name, CorporationName), SimplePredicate: (sortedAcronym, name)), (SimplePredicate: (firstIntegerPredicate, name), SimplePredicate: (tokenFieldPredicate, state)), (PartialIndexTfidfTextCanopyPredicate: (0.6, name, Surname), TfidfTextCanopyPredicate: (0.8, name)), (PartialIndexLevenshteinCanopyPredicate: (1, name, Surname), PartialPredicate: (commonSixGram, name, Surname)), (SimplePredicate: (commonIntegerPredicate, street), TfidfNGramCanopyPredicate: (0.8, name)), (SimplePredicate: (fingerprint, name), SimplePredicate: (nearIntegersPredicate, name)), (SimplePredicate: (fingerprint, name), TfidfNGramCanopyPredicate: (0.6, street)),

		(PartialIndexTfidfNGramCanopyPredicate: (0.6, name, CorporationName), TfidfNGramCanopyPredicate: (0.6, name)))
Special Name and Address	settings_CORP_ext_4	((PartialPredicate: (sameThreeCharStartPredicate, name, CorporationName), SimplePredicate: (commonSixGram, name)), (LevenshteinCanopyPredicate: (1, zip), SimplePredicate: (alphaNumericPredicate, name)), (PartialIndexTfidfNGramCanopyPredicate: (0.2, name, Surname), SimplePredicate: (sortedAcronym, name)), (SimplePredicate: (commonFourGram, name), SimplePredicate: (firstIntegerPredicate, name)), (LevenshteinCanopyPredicate: (1, name), TfidfNGramCanopyPredicate: (0.8, name)), (PartialPredicate: (suffixArray, city, StreetName), SimplePredicate: (twoGramFingerprint, name)))

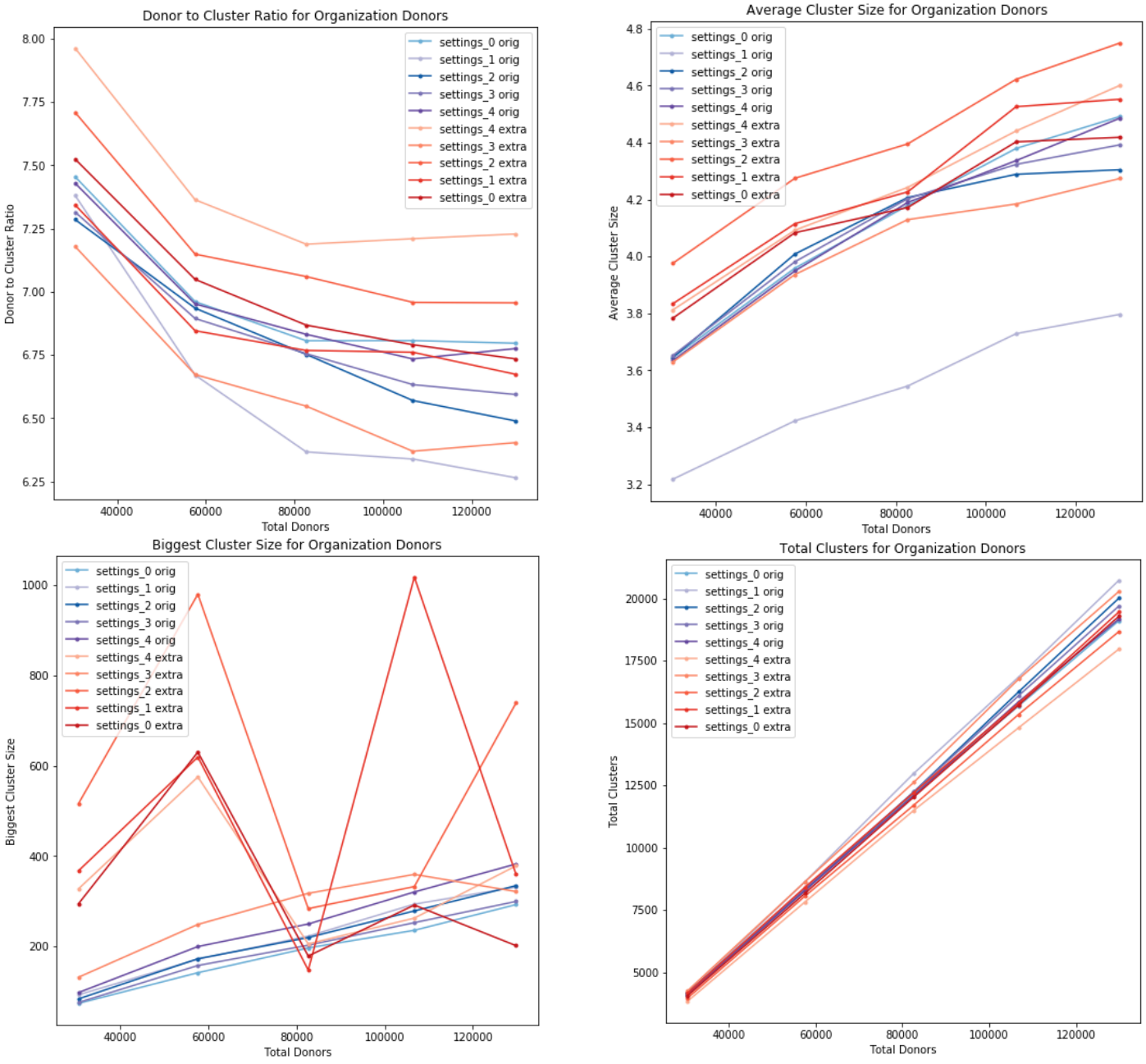
Final Predicates and Classifier

As with the findings of the individual data set, many of the classifiers using the data model that included the extended parsing of Name and Address types grouped clusters that included “New York state” in the name but were different organizations (e.g. New York State Laborers’ Association and New York State Patrolmen’s Benevolent Association). These errant clusters were identified by reviewing the top clusters from sample data.

The top classifier was generated in settings_0, and the two sets of predicates that included diverse results without including errant clusters were from settings_0 and settings_2. These predicates were combined for the final matching run. They are shown in blue in the charts. The settings with the plain data model are shown in purple and the settings with the extended data model are shown in orange.

Both of these predicate groups generate average results according to these metrics. In those with higher average cluster size and bigger maximum cluster size, the biggest clusters included bad matches.

Figure 11: Evaluation of Predicates and Classifiers for Organization Donors



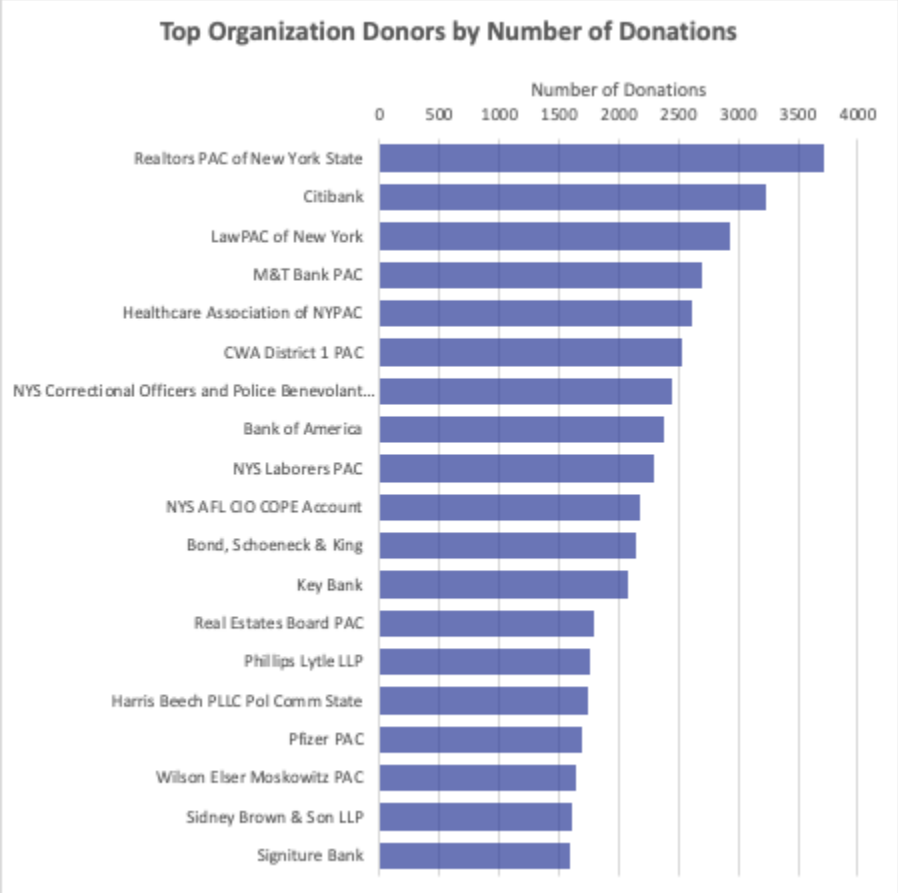
The Organization Donor Matching Results

The universe of organization donors started at 864,664 entries. The matching algorithm reduced the number of organization donors to 281,904 that donated $1,916,663,704. Of these,

133,259 have been clustered in clusters larger than one. The largest cluster has 760 matched donor entries in it. It is for Clough, Harbour & Associates LLP (now known as CHA Consulting), is an Albany-based engineering and design firm that has secured many New York State contracts through the years. The average matched number of organization donor entries was much higher than individual donors at 5.8.

The organization donor that donated the most number of times was the Realtors PAC of New York State, which contributed 3,722 transactions within this data set. Similar to the individual donors, the average number of donations is much much smaller at 6, and the median is 1. This means the majority of organization donors only donated one time. There are a small handful of donors who donate a lot. Below is a list of the top 10 organization donors.

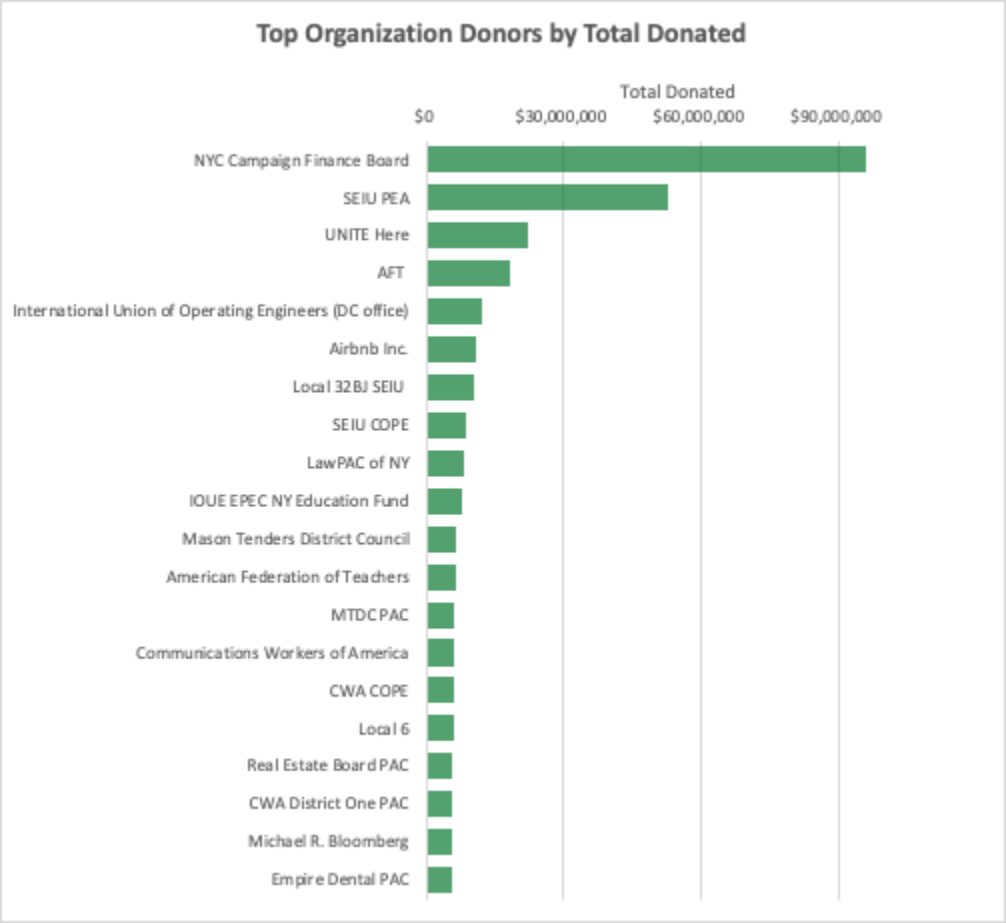
Figure 12: Top Organization Donors by Number of Donations



The biggest organization donor by far in this data set with more than $96 million is perhaps surprising: the New York City Board of Elections. This money is matching funds that people who ran for city office, like public advocate, received as part of New York City's public financing of elections program. The next highest organization donor is SEIU's Political Education and Action Committee with almost $53 million. The totals drop off steeply after that, with many of the entries appearing to be different parts or different acronyms that represent the same larger unions for SEIU, AFT, and CWA. That these organizations are not clustered demonstrates the limits of the conservative matching that I chose. These clusters could be condensed even further by manual review. Perhaps notably the top organization donor list also contains Michael Bloomberg.

The average total amount donated by an organization is $7,202.72 while the median is a mere $500. This means that half of the organizations gave $500 or less.

Figure 13: Top Organization Donors by Total Donated



Summary

The matching of this donor data was required because there is no unique identifier for donors within the NYSBOE data. Each recipient files separate filings for their donors each time they give. Across recipient filings, there are many entries that represent the same individuals or organizations, but they are not connected in the data kept by the NYSBOE. Complicating the clustering of donors, typos and varying abbreviations abound in the data, making it very hard to find perfect matches between donors. Through tuning the algorithm to match well, but not too well, I have made a first attempt at matching the donors given the data available. There is

certainly more matching that could be done, but it is still useful to analyze the data at this stage.

My analysis of this data reveals donations are skewed. There are a few organizations and wealthy individuals that give a lot of money, including unions, professional associations, and famous billionaires. Evidence seems to indicate that some groups get more influence for their donations that others. A 2014 study measured who influenced federal policy from 1989 to 2002. The different groups measured were average citizens, economic elites, mass-based interest groups (including everything from unions to the National Rifle Association), and business interest groups. The study found that average citizens have "essentially zero" impact on policy changes at the national level. Economic elites had by far the most influence, followed by business interest groups, which had 55% of the influence of economic elites. Mass-based groups had about half of the influence of business groups and about 30% of the influence of economic elites (Gilens 2014).

Organization donors gave more than $1.9 billion to recipients and more than half of the money came from just 784 organizations. 0.28% of the organizations contributed more than half of the total. The top quarter of the money came from just 66 organizations, while the bottom 35% of the organizations gave $250 or less.

On the individual donor side it is slightly less skewed, but still more than one half of the money donated came from less than 1% of the donors. Of the more than $1.6 billion dollars given by individual donors, one half of that came from 15,559 donors, only 0.82% of the total number of individual donors. One quarter of the money came from 967 individual donors: 0.05% contributed $412 million in this data set, while the bottom 50% donated $100 or less.

This project was aimed at building out this data set for further use and evaluation. Now that the algorithm is well-tuned and the data fully matched, I intend to take this data to fellow activists, friends, and organizations and encourage them to question it, consider it, and hopefully find meaning and answers within it.

In gathering a network of people who are interested in this data, I hope to build a small collective, modeled off of the Housing Data Coalition: https://www.housingdatanyc.org/, that can identify new ways to use this data for advocacy and research. My hope is that people in this network will build on this initial project with their own expertise and ideas and improve it and grow it.

Appendix A: List of Variables by Module

Universal Variable for most modules:

DATABASE_URL	The url including user credentials to connect to the database where the data will be loaded and retrieved from

data_load.all_txt_to_csv.py

filers_dir	route to the filers file directory (from the board of elections)
filings_dir	route to the filings file directory (from the board of elections)
filings_file	the cleaned txt file in the filings directory that is ready to be converted into a csv for loading into the database

data_load.clean_donors.py

No parameters

data_load.cluster_data_load.py

processed_donors	csv file that is dump of a processed_donors table from a previous matching run
entity_map	csv file that is a dump of an entity_map table from a previous matching run

data_load.fix_all_reports.py

filings_dir	route to the filings file directory (from the board of elections)
infile_name	the cleaned txt file in the filings directory that is ready to be converted into a csv for loading into the database
outfile_name	desired name of the output csv file

data_load.get_samples.py

filers_dir	route to the filers file directory (from the board of elections)
filings_dir	route to the filings file directory (from the board of elections) (note this module assumes the name of the filings file to use is:

	ALL_REPORTS_fixed.txt
sample_size	percent of the total starting number of records that you would like to sample
random_num	The seed for the selection of the random sample

data_load.init_postgres_db.py

recipients_file	the cleaned recipients csv file (filers) from the board of elections
contributions_file	The cleaned donations csv file (filings) from the board of elections

data_load.pre_init_db.py
No parameters

dedupe_extension.campaign_finanace_dedupe.py

-s settings_file	file to load previously generated settings from (data model, classifier, predicates)
-t type	type of the donor (IND or CORP)
-v	increase the verbosity of logging, 2 levels so can use twice

dedupe_extension.start_at_clustering.py

-s settings_file	file to load previously generated settings from (data model, classifier, predicates)
-t type	type of the donor (IND or CORP)
-v	increase the verbosity of logging, 2 levels so can use twice

matching_evaluation.combine_predicates.py

first_settings	file to load previously generated settings from (data model, classifier, predicates). This is the file that retains the data_model and classifier
second_settings	file to load previously generated settings from (data model, classifier, predicates). This is the file from which the predicates are cherry-picked by the indexes and added to the first_settings file

indexes	a comma delimited list of the indexes of the predicates from the second_settings file that should be added to the first settings file

run_stats_only.py

settings_file	the settings file used for the run
-t type	type the donor (IND or CORP)

run_tests_comb.py

-t type	type the donor (IND or CORP)

run_tests.py

-t type	type the donor (IND or CORP)

Appendix B: Glossary of Functions

data_load.all_txt_to_csv.py: transform the headless cleaned files from the board of elections into csvs with headers and proper data types for loading into the database

params: filers_dir, filings_dir,filings_file

data_load.clean_donors.py: cleans the data from the processed_donors table to expand the acronyms within the street names and remove floating periods

params: none

data_load.cluster_data_load.py: loads previously saved files for clusters into a database

params: processed_donors, entity_map

data_load.fix_all_reports.py: clean bad entries in the filings file from the board of elections, including non-matching quotation marks and other issues

params: filings_dir, infile_name, outfile_name

data_load.get_samples.py: get a stratified sample from the larger files of filings and filers. It is stratified by transaction type.

params: filers_dir, filings_dir, sample_size, random_num

data_load.init_postgres_db.py: loading of the cleaned and prepped csvs into the database. This is the main module for creating the database after cleaning before the matching

params: recipients_file, donations_file

data_load.pre_init_db.py: create match_runs table - only needs to be run once when setting up the database for the first time

params: none

dedupe_extension.campaign_finanace_dedupe.py: this is the main module for matching the data from the database

params: (all optional) -s settings_file, -t type, -v verbosity

dedupe_extension.start_at_clustering.py: there is an issue with large data sets (> 2-3 million filing entries) where the database connections would time out at the end of the blocking phase. The blocking maps are saved in the database however, so they can be leveraged for clustering. This restarts the matching process at the clustering phase, assuming the blocking map has been saved in the database.

params: (-v and -t optional) -s settings_file, -t type, -v verbosity

matching_evaluation.combine_predicates.py: takes two settings files and combines the predicates of the second with the first based on the supplied indexes

params: first_settings, secode_settings, indexes

run_stats_only.py: generate the statistics for a match_runs table entry based on the current state of the database

params: settings_file, (optional) -t type

run_tests_comb.py: run a series of tests of a single settings file

params: (optional) -t type

run_tests.py: cross validate a series of settings files across a series of samples of filings and filers

params: (optional) -t type

REFERENCES

Bilenko, Mikhail Yuryevich. 2006. Learnable Similarity Functions and Their Applications in Record Linkage and Clustering.https://www.cs.utexas.edu/~ml/papers/marlin-dissertation-06.pdf

Forest, Gregg and Derek Eder. 2019. Dedupe. https://github.com/dedupeio/dedupe.

Gilens, Martin, and Benjamin I. Page. "Testing Theories of American Politics: Elites, Interest Groups, and Average Citizens." *Perspectives on Politics*, vol. 12, no. 3, Sept. 2014, pp. 564–81. *DOI.org (Crossref)*, doi:10.1017/S1537592714001595.