Implementation for the Analysis of U.S. Incomes

Conrad Kosowsky

This file documents the code to fit various models to public-use CPS income data and produce associated results and figures. The actual code is contained in seven `py` files:

1. `gen_files.py`. The file to parse the data files. It assumes that the ⟨*year*⟩ CPS data file is contained in a directory "`../data/`⟨*year*⟩ `CPS/`" so that `data` is at the same level as the present working directory. This file will produce one data file in the present working directory called `data_`⟨*year*⟩`.txt` for each year, and the data files will contain four columns: household identifier, state, personal income, and survey weight.
2. `print_edges.py`. This file will read in all the data files and print the smallest and

Files

Table 1: Variables in the Implementation

Purpose	Identifier	Meaning
From the Data	`household`	Household Identifier
	`income`	Individual Income
	`region`	Census Bureau Region Classification
	`state`	State Code
	`weight`	Survey Weight
Parameters (See Table 6)	`alpha`, `p`, `q`	Shape Parameters
	`beta`, `x0`	Scale Parameters
	`c`	Shift Parameter
	`gamma`	Mixture Parameter
	`k`	Cutoff Parameter
	`mu`, `sigma_sq`	Log-normal Parameters
In Binned Data (See Table 4)	`dens`	Density
	`freq`	Frequency
	`left`	Left Endpoint
	`mass`	Total Mass
	`mid`	Bin Midpoint
	`right`	Right Endpoint
	`width`	Bin Width

largest entries in the data. Used for manually determining the cutoff values before analysis.

3. `bin.py`. Code to bin data. It provides a single function that accepts a DataFrame, bins it, and returns the result.
4. `estimate_parameters.py`. The code to analyze the data. This file contains the routines to estimate parameters for all models of interest. It provides a number of estimation functions as well as other useful functions such as density and cumulative distribution. All user-level functions can be accessed through the `distribution`, `density`, `likelihood`, and `estimator` dictionaries.
5. `check_constants.py`. This file contains the code to run the linear regression on inverse-gammma parameters. It prints the results and writes them to a file.
6. `bootstrap.py`. This file contains the code to bootstrap standard errors using the synthetic survey design and estimation functions from `estimate_parameters.py`.
7. `make_figures.py`. This file contains functions that can produce figures using pyplot.
8. `main.py`. The file runs the analysis, produces parameter estimates, and makes the figures. This file loads `estimate_parameters.py`, `check_constants.py`, and `make_figures.py`. Switches at the start of the file determine which code executes when the user runs the file.

File I

gen_files.py

This file documents the code to parse the income information from the Current Population Survey public-use microdata files. We will store the information in a new set of smaller csv files, which reduces computing resources for repeated calculations. We will collect the following information: individual income record, household identifier, state, and survey weight. Table 2 lists the location of these records in the data files. For data files before 2019, the public-use data files encode each respondent's answers as a single string variable, where different responses occupy different positions in the string. For 2019 through 2023, the data files are separate csv files.

We begin by importing Pandas and defining a function to convert strings to integers if possible. We use `try_int` instead of `int` for earlier years of data because a very small number of the entries have errors.

```
import pandas as pd
def try_int(x):
  try:
    return int(x)
  except:
    return float("nan")
```

The dictionary `files` will store the filenames of all the files. The keys are years, and the values are filename strings.

```
files = {
  #1964:"cpsmar64.dat",
  #1966:"cpsmar66.dat",
  1967:"cpsmar67.dat",
  1968:"cpsmar68.dat",
  1969:"cpsmar69.dat",
  1971:"cpsmar71.dat",
```

Table 2: Locations of Variables in the Data Files

Year	Person Identifier	Income	Household Identifier	State	Survey Weight
1964–1967	starts with `"00036"`	`170:180`	`10:17`	`43:45`	`20:30`
1968–1971	first digit 1, 2, or 3	`60:66`	`1:16`	F37–38	`204:216`
1972–1975	first digit 1, 2, or 3	`60:66`	none	F37–38	`204:216`
1976	digits 7,8 are 1–39	`246:253`	`0:6`	H53–54	`117:128`
1977–1979	digits 7,8 are 1–39	`246:253`	`0:6`	H39–40	`117:128`
1980–1987	digits 7,8 are 1–39	`247:254`	`0:6`	H39–40	`117:128`
1988–2010	first digit is 3	`439:447`	`1:6`	H40–41	`65:73`
2011–2018	first digit is 3	`579:587`	`1:6`	H42–43	`154:162`
2019–2023	separate file	`PTOTVAL`	`PH_SEQ`	`GESTFIPS`	`MARSUPWT`

```
1972:"cpsmar72.dat",
1973:"cpsmar73.dat",
1974:"cpsmar74.dat",
1975:"cpsmar75.dat",
1976:"cpsmar76.dat",
1977:"cpsmar77.dat",
1978:"cpsmar78.dat",
1979:"cpsmar79.dat",
1980:"cpsmar80.dat",
1981:"cpsmar81.dat",
1982:"cpsmar82.dat",
1983:"cpsmar83.dat",
1984:"cpsmar84.dat",
1985:"cpsmar85.dat",
1986:"cpsmar86.dat",
1987:"cpsmar87.dat",
1988:"cpsmar88.dat",
1989:"cpsmar89.dat",
1990:"cpsmar90.dat",
1991:"cpsmar91.dat",
1992:"cpsmar92.dat",
1993:"cpsmar93.dat",
1994:"cpsmar94.dat",
1995:"cpsmar95.dat",
1996:"cpsmar96.dat",
1997:"cpsmar97.dat",
1998:"mar98pub.cps",
1999:"mar99pub.cps",
2000:"mar00supp.dat",
2001:"mar01supp.dat",
2002:"mar02supp.dat",
2003:"asec2003.pub",
2004:"asec2004.pub",
2005:"asec2005_pubuse.pub",
2006:"asec2006_pubuse.pub",
2007:"asec2007_pubuse_tax2.dat",
2008:"asec2008_pubuse.dat",
2009:"asec2009_pubuse.dat",
2010:"asec2010_pubuse.dat",
2011:"asec2011_pubuse.dat",
2012:"asec2012_pubuse.dat",
2013:"asec2013_pubuse.dat",
2014:"asec2014_pubuse_3x8_rerun.dat",
2015:"asec2015_pubuse.dat",
2016:"asec2016_pubuse_v3.dat",
2017:"asec2017_pubuse.dat",
2018:"asec2018_pubuse.dat",
```

Table 3: How the Microdata Stores State Identifiers

Year	Stored As	Corresponding Dictionary
1964–1967	(Modified) 1960 State Codes	`st60_to_name1`
1968–1972	(Modified) 1960 State Codes	`st60_to_name2`
1973–1976	(Modified) 1960 State Codes	`st60_to_name3`
1977–2010	(Modified) 1960 State Codes	`st60_to_name4`
2011–2023	Fips (`GESTFIPS`)	N/A

```
  2019:["hhpub19.csv","pppub19.csv"],
  2020:["hhpub20.csv","pppub20.csv"],
  2021:["hhpub21.csv","pppub21.csv"],
  2022:["hhpub22.csv","pppub22.csv"],
  2023:["hhpub23.csv","pppub23.csv"]}
```

The dictionary `regions` stores the region information. It would be simpler to use the region entry from the microdata, but for historical reasons, we convert FIPS state codes to region numbers. The keys in `regions` are integer FIPS codes, and the values are region identifiers. (Northeast is 1; Midwest is 2; South is 3; and West is 4.) Storage of state codes varies from year to year, and Table 2 shows where the microdata contains the state code. Earlier years of microdata encode the state identifier as (a modified verion of) the 1960 Census state codes, and the microdata don't contain explicit state FIPS codes until 2001. (Note that in 2005, the Census Bureau renamed `HG-ST60` to `GESTCEN`.) Table 3 shows how each year of microdata stores the state. For our implementation, we use `HG-ST60` through 2010 and then switch to `GESTFIPS`.

```
regions = {
  1:3,   # Alabama
  2:4,   # Alaska
  # 3 is American Samoa
  4:4,   # Arizona
  5:3,   # Arkansas
  6:4,   # California
  # 7 is Canal Zone
  8:4,   # Colorado
  9:1,   # Connecticut
  10:3,  # Delaware
  11:3,  # DC
  12:3,  # Florida
  13:3,  # Georgia
  # 14 is Guam
  15:4,  # Hawaii
  16:4,  # Idaho
  17:2,  # Illinois
  18:2,  # Indiana
  19:2,  # Iowa
  20:2,  # Kansas
```

```
21:3,  # Kentucky
22:3,  # Louisiana
23:1,  # Maine
24:3,  # Maryland
25:1,  # Massachusetts
26:2,  # Michigan
27:2,  # Minnesota
28:3,  # Mississippi
29:2,  # Missouri
30:4,  # Montana
31:2,  # Nebraska
32:4,  # Nevada
33:1,  # New Hampshire
34:1,  # New Jersey
35:4,  # New Mexico
36:1,  # New York
37:3,  # North Carolina
38:2,  # North Dakota
39:2,  # Ohio
40:3,  # Oklahoma
41:4,  # Oregon
42:1,  # Pennsylvania
# 43 is Puerto Rico
44:1,  # Rhode Island
45:3,  # South Carolina
46:2,  # South Dakota
47:3,  # Tennessee
48:3,  # Texas
49:4,  # Utah
50:1,  # Vermont
51:3,  # Virginia
# 52 is Virgin Islands
53:4,  # Washington
54:3,  # West Virginia
55:2,  # Wisconsin
56:4,  # Wyoming
```

Jolliffe (2003) doesn't specify how to handle fips codes that do not correspond to a state (and hence do not have an official region). We group them geographically with corresponding regions.

```
3:4,   # American Samoa
60:4,  # American Samoa
81:4,  # Baker Island
7:3,   # Canal Zone
64:4,  # Federated States of Micronesia
14:4,  # Guam
66:4,  # Guam
84:4,  # Howland Island
```

```
  86:4,  # Jarvis Island
  67:4,  # Johnston Atoll
  89:4,  # Kingman Reef
  68:4,  # Marshall Islands
  71:4,  # Midway Islands
  74:4,  # Minor Outlying Islands
  76:3,  # Navassa Island
  69:4,  # Northern Mariana Islands
  70:4,  # Palau
  95:4,  # Palmyra Atoll
  43:3,  # Puerto Rico
  72:3,  # Puerto Rico
  52:3,  # Virgin Islands
  78:3,  # Virgin Islands
  79:4}  # Wake Island
```

These non-state fips codes shouldn't appear in the data since the microdata says the FIPS codes run from 1 to 56. However, we code them just to be safe. Some of the earlier CPS datasets store the household state information using 1960 Census state codes (variable `HGST60` instead of `GESTFIPS`), which are different from FIPS codes. Accordingly, we create several dictionaries to convert early-year state codes to FIPS values. These dictionaries use (modified versions of) 1960 state codes as keys and state names as values. See Table 3.

```
st60_to_name1 = {
  # New England
   1: "Maine",
   3: "New Hampshire",
   4: "Vermont",
   2: "Massachusetts",
   5: "Rhode Island",
   6: "Connecticut",
  # Middle Atlantic
  10: "New York",
  11: "New Jersey",
  13: "Pennsylvania",
  # East North Central
  24: "Ohio",
  23: "Indiana",
  25: "Illinois",
  26: "Michigan",
  22: "Wisconsin",
  # West North Central
  31: "Minnesota",
  32: "Iowa",
  33: "Missouri",
  34: "North Dakota",
  35: "South Dakota",
  36: "Nebraska",
  37: "Kansas",
```

```
    # South Atlantic
    41: "Delaware",
    42: "Maryland",
    43: "DC",
    44: "Virginia",
    45: "West Virginia",
    47: "North Carolina",
    46: "South Carolina",
    48: "Georgia",
    49: "Florida",
    # East South Central
    51: "Kentucky",
    52: "Tennessee",
    53: "Alabama",
    54: "Mississippi",
    # West South Central
    65: "Arkansas",
    66: "Louisiana",
    67: "Oklahoma",
    68: "Texas",
    # Mountain
    71: "Montana",
    72: "Idaho",
    73: "Wyoming",
    74: "Colorado",
    75: "New Mexico",
    76: "Arizona",
    77: "Utah",
    78: "Nevada",
    # Pacific
    87: "Washington",
    88: "Oregon",
    89: "California",
    85: "Alaska",
    86: "Hawaii"}
```

Another version. The 1969 microdata combines a number of states into single state codes. Fortunately, it won't affect our region assignment.

```
st60_to_name2 = {
    ## New England
    11: "Connecticut",
    # 19 is also Massachusetts, New Hampshire,
    # Rhode Island, and Vermont
    19: "Maine",
    ## Middle Atlantic
    21: "New York",
    22: "New Jersey",
    23: "Pennsylvania",
```

```
    ## East North Central
    31: "Ohio",
    32: "Indiana",
    33: "Illinois",
    # 39 also contains Wisconsin
    39: "Michigan",
    ## West North Central
    # 41 also contains Minnesota
    41: "Iowa",
    43: "Missouri",
    # 49 also contains Nebraska, Kansas, and South Dakota
    49: "North Dakota",
    ## South Atlantic
    51: "DC",
    52: "Maryland",
    53: "West Virginia",
    54: "Georgia",
    55: "Florida",
    # 57 also contains South Carolina
    57: "North Carolina",
    # 59 also contains Virginia
    59: "Delaware",
    ## East South Central
    61: "Kentucky",
    62: "Tennessee",
    # 69 also contains Mississippi
    69: "Alabama",
    ## West South Central
    71: "Louisiana",
    72: "Texas",
    # 79 also contains Oklahoma
    79: "Arkansas",
    ## Mountain
    # 81 also contains Colorado and New Mexico
    81: "Arizona",
    # 89 also contains Montana, Nevada, Utah, and Wyoming
    89: "Idaho",
    ## Pacific
    91: "Oregon",
    92: "California",
    # 99 also contains Hawaii and Washington
    99: "Alaska"}
```

Another version. Same deal with the 1973 microdata

```
st60_to_name3 = {
    ## New England
    16: "Connecticut",
    14: "Massachusetts",
```

```
    # 19 is also New Hampshire,
    # Rhode Island, and Vermont
    19: "Maine",
    ## Middle Atlantic
    21: "New York",
    22: "New Jersey",
    23: "Pennsylvania",
    ## East North Central
    31: "Ohio",
    32: "Indiana",
    33: "Illinois",
    # 39 also contains Wisconsin
    39: "Michigan",
    ## West North Central
    # 49 also contains Iowa, North Dakota, South Dakota,
    # Nebraska, Kansas, and Missouri
    49: "Minnesota",
    ## South Atlantic
    53: "DC",
    56: "North Carolina",
    # 57 also contains Maryland, Virginia, and West Virginia
    57: "Delaware",
    # 58 also contains Georgia
    58: "South Carolina",
    59: "Florida",
    ## East South Central
    # 67 also contains Tennessee
    67: "Kentucky",
    # 69 also contains Mississippi
    69: "Alabama",
    ## West South Central
    72: "Texas",
    # 79 also contains Oklahoma and Louisiana
    79: "Arkansas",
    ## Mountain
    # 89 also contains Idaho, Wyoming, Colorado, New Mexico,
    # Utah, and Nevada
    89: "Montana",
    ## Pacific
    92: "California",
    # 99 also contains Hawaii, Washington, Oregon, and Alaska
    99: "Alaska"}
```

Main version. This is the verison that we use for most of the years of microdata with 1960 Census state codes.

```
st60_to_name4 = {
    # New England
    11: "Maine",
```

```
12: "New Hampshire",
13: "Vermont",
14: "Massachusetts",
15: "Rhode Island",
16: "Connecticut",
# Middle Atlantic
21: "New York",
22: "New Jersey",
23: "Pennsylvania",
# East North Central
31: "Ohio",
32: "Indiana",
33: "Illinois",
34: "Michigan",
35: "Wisconsin",
# West North Central
41: "Minnesota",
42: "Iowa",
43: "Missouri",
44: "North Dakota",
45: "South Dakota",
46: "Nebraska",
47: "Kansas",
# South Atlantic
51: "Delaware",
52: "Maryland",
53: "DC",
54: "Virginia",
55: "West Virginia",
56: "North Carolina",
57: "South Carolina",
58: "Georgia",
59: "Florida",
# East South Central
61: "Kentucky",
62: "Tennessee",
63: "Alabama",
64: "Mississippi",
# West South Central
71: "Arkansas",
72: "Louisiana",
73: "Oklahoma",
74: "Texas",
# Mountain
81: "Montana",
82: "Idaho",
83: "Wyoming",
```

```
  84: "Colorado",
  85: "New Mexico",
  86: "Arizona",
  87: "Utah",
  88: "Nevada",
  # Pacific
  91: "Washington",
  92: "Oregon",
  93: "California",
  94: "Alaska",
  95: "Hawaii"}
```

The next dictionary uses state (and territory) names as keys and FIPS codes as values.

```
name_to_fips = {
  # states plus DC
  "Alabama":           1,
  "Alaska":            2,
  "Arizona":           4,
  "Arkansas":          5,
  "California":        6,
  "Colorado":          8,
  "Connecticut":       9,
  "Delaware":         10,
  "DC":               11,
  "Florida":          12,
  "Georgia":          13,
  "Hawaii":           15,
  "Idaho":            16,
  "Illinois":         17,
  "Indiana":          18,
  "Iowa":             19,
  "Kansas":           20,
  "Kentucky":         21,
  "Louisiana":        22,
  "Maine":            23,
  "Maryland":         24,
  "Massachusetts":    25,
  "Michigan":         26,
  "Minnesota":        27,
  "Mississippi":      28,
  "Missouri":         29,
  "Montana":          30,
  "Nebraska":         31,
  "Nevada":           32,
  "New Hampshire":    33,
  "New Jersey":       34,
  "New Mexico":       35,
  "New York":         36,
```

```
  "North Carolina":     37,
  "North Dakota":       38,
  "Ohio":               39,
  "Oklahoma":           40,
  "Oregon":             41,
  "Pennsylvania":       42,
  "Rhode Island":       44,
  "South Carolina":     45,
  "South Dakota":       46,
  "Tennessee":          47,
  "Texas":              48,
  "Utah":               49,
  "Vermont":            50,
  "Virginia":           51,
  "Washington":         53,
  "West Virginia":      54,
  "Wisconsin":          55,
  "Wyoming":            56,
  # territories
  "American Samoa":                   3,
  "American Samoa":                  60,
  "Baker Island":                    81,
  "Canal Zone":                       7,
  "Federated States of Micronesia":  64,
  "Guam":                            14,
  "Guam":                            66,
  "Howland Island":                  84,
  "Jarvis Island":                   86,
  "Johnston Atoll":                  67,
  "Kingman Reef":                    89,
  "Marshall Islands":                68,
  "Midway Islands":                  71,
  "Minor Outlying Islands":          74,
  "Navassa Island":                  76,
  "Northern Mariana Islands":        69,
  "Palau":                           70,
  "Palmyra Atoll":                   95,
  "Puerto Rico":                     43,
  "Puerto Rico":                     72,
  "Virgin Islands":                  52,
  "Virgin Islands":                  78,
  "Wake Island":                     79}
```

Check that we spelled all the state names correctly.

```
for i in range(1,5):
  temp = eval("st60_to_name{0}".format(i))
  for k in temp:
    if temp[k] not in name_to_fips:
```

```
      raise KeyError(
"for st60_to_name{0}: {1} not in name_to_fips".format(i, temp[k]))
print("\nIndex check for dicts is good\n")
```

The dictionary `year_keys` stores the year information from Table 2. The keys are years, and the values are the first year with the same variable locations according to the table. However, it is more compact to tell Python to assemble the dictionary for us. We code the list `base_years`, which contains the first year associated with each row in Table 2.

```
base_years = [1964, 1968, 1973, 1976, 1977, 1980, 1988, 2011, 2019]
```

The list does not contain 1972 because we will use the 1968 file reader for the 1972 data files. We loop through years from 1964 to 2023 and populate `year_keys` with elemenets of this range as keys. The pointer `curr_id` keeps track of where we are in `base_years`, and `curr_year` and `next_year` keep track of current and successive entries in `base_years`. On each iteration of the loop, we first check if `i` equals the `next_year`. That means we are outside the target interval and need to update `curr_id`. The `next_year` becomes the `curr_year`, and we take a new `next_year` from `base_years` if possible. Then we set the value in `year_keys` to be `curr_year`.

```
year_keys = {}
curr_id = 0
curr_year = base_years[curr_id]
next_year = base_years[curr_id + 1]
for i in range(1964, 2024):
  if i == next_year:
    curr_year = next_year
    try:
      curr_id = curr_id + 1
      next_year = base_years[curr_id + 1]
    except:
      next_year = False
  year_keys[i] = curr_year
```

Now define the functions to parse the data files. Each function will accept a filename and return a formatted DataFrame. The DataFrame will contain four columns: one for income, one for survey weight, one for state code, and one for household identifier. For most years, we manually parse the file, but for 2019, where the data file becomes csv, we have Pandas parse the file contents. We call each function `read_income_`$\langle$*key*$\rangle$, where the *key* is the first year associated with the current year according to the first column of Table 2.

We begin with 1964 and progress chronologically. For 1964, we make blank lists for income, identifier, state, and weight, and we loop through the lines of the file. For each line, we pull the relevant information if the entry corresponds to a person from the March supplement. We create a `DataFrame` called `temp` and convert the state codes to FIPS values. (We don't add the region until we're ready to write the data extract file.)

```
def read_income_1964(filename):
  household = []
  income    = []
  state     = []
  weight    = []
```

```
  f = open(filename)
  for line in f:
    if line[0:5] == "00036":
      household.append(try_int(line[10:17]))
      income.append(try_int(line[170:180]))
      state.append(try_int(line[43:45]))
      weight.append(try_int(line[20:30]))
  f.close()
  temp = pd.DataFrame(
    {"household": household,
     "income": income,
     "state": state,
     "weight": weight}).dropna()
  temp["state"] = temp["state"].map(st60_to_name1).map(name_to_fips)
  return temp
```

For 1968–1975, we add our own household identifier, which will be an integer starting from 1 that increments at each household record. (The earliest CPS data files do not make as conspicuous a distinction between household and family records as in later iterations.) The first family record in each household has nonzero entry for F47–48, so we increment our household identifier whenever we encounter such a record. Once again, we change the state identifier to a FIPS code.

```
def read_income_1968(filename):
  household = []
  income    = []
  state     = []
  weight    = []
  f = open(filename)
  curr_household = 0
  for line in f:
    if line[0] == "4":  # family record
      if line[46:48] != "00":       # household record
        curr_household = curr_household + 1
        try_int_trace = 1
        curr_state = try_int(line[36:38])
        try_int_trace = 0
    else:               # person record
      household.append(curr_household)
      income.append(try_int(line[60:66]))
      state.append(curr_state)
      weight.append(try_int(line[204:216]))
  f.close()
  temp = pd.DataFrame(
    {"household": household,
     "income": income,
     "state": state,
     "weight": weight}).dropna()
  temp["state"] = temp["state"].map(st60_to_name2).map(name_to_fips)
```

```
  return temp
```

Parsing the 1973–1975 years is exactly the same as 1968 except we use a different dictionary to get the FISP codes.

```
def read_income_1973(filename):
  household = []
  income    = []
  state     = []
  weight    = []
  f = open(filename)
  curr_household = 0
  for line in f:
    if line[0] == "4":  # family record
      if line[46:48] != "00":       # household record
        curr_household = curr_household + 1
        curr_state = try_int(line[36:38])
    else:               # person record
      household.append(curr_household)
      income.append(try_int(line[60:66]))
      state.append(curr_state)
      weight.append(try_int(line[204:216]))
  f.close()
  temp = pd.DataFrame(
    {"household": household,
     "income": income,
     "state": state,
     "weight": weight}).dropna()
  temp["state"] = temp["state"].map(st60_to_name3).map(name_to_fips)
  return temp
```

For 1976–1987, the functions are very similiar. The main difference between 1968 is that we look at digits 7 and 8 to determine household versus family versus person records. (The household records are 0 here.) At this point, the data is good enough that we can use `int` rather than `try_int`. Again, we convert the state codes to FIPS numbers.

```
def read_income_1976(filename):
  household = []
  income    = []
  state     = []
  weight    = []
  f = open(filename)
  for line in f:
    type = int(line[6:8])
    if type == 0:                   # household record
      curr_state = int(line[52:54])
    elif 1 <= type and type <= 39:  # person record
      household.append(int(line[0:6]))
      income.append(int(line[246:253]))
      state.append(curr_state)
```

```
      weight.append(int(line[117:128]))
  f.close()
  temp = pd.DataFrame(
    {"household": household,
     "income": income,
     "state": state,
     "weight": weight})
  temp["state"] = temp["state"].map(st60_to_name3).map(name_to_fips)
  return temp
```

In 1977, the location of the state record is different, and we begin using the main (fourth) `st60_to_name` dictionary to get the FIPS codes. Otherwise the function is the same.

```
def read_income_1977(filename):
  household = []
  income    = []
  state     = []
  weight    = []
  f = open(filename)
  for line in f:
    type = int(line[6:8])
    if type == 0:                   # household record
      curr_state = int(line[38:40])
    elif 1 <= type and type <= 39:  # person record
      household.append(int(line[0:6]))
      income.append(int(line[246:253]))
      state.append(curr_state)
      weight.append(int(line[117:128]))
  f.close()
  temp = pd.DataFrame(
    {"household": household,
     "income": income,
     "state": state,
     "weight": weight})
  temp["state"] = temp["state"].map(st60_to_name4).map(name_to_fips)
  return temp
```

In 1980, the location of the income record changes.

```
def read_income_1980(filename):
  household = []
  income    = []
  state     = []
  weight    = []
  f = open(filename)
  for line in f:
    type = int(line[6:8])
    if type == 0:                   # household record
      curr_state = int(line[38:40])
    elif 1 <= type and type <= 39:  # person record
      household.append(int(line[0:6]))
```

```
      income.append(int(line[247:254]))
      state.append(curr_state)
      weight.append(int(line[117:128]))
  f.close()
  temp = pd.DataFrame(
    {"household": household,
     "income": income,
     "state": state,
     "weight": weight})
  temp["state"] = temp["state"].map(st60_to_name4).map(name_to_fips)
  return temp
```

The structure of the files changes in 1988. Now household records start with a `1`, and person records start with `3`.

```
def read_income_1988(filename):
  household = []
  income    = []
  state     = []
  weight    = []
  f = open(filename)
  for line in f:
```

The 1996 data file has an extra character at the very end of the file, which breaks the parser as written. So we check that the length of each line is greater than 2 before processing it. This problem appears to be confined to the 1996 and 1997 data files.

```
    if len(line) > 2:
      type = int(line[0])
      if type == 1:    # household record
        curr_state = int(line[39:41])
      elif type == 3:  # person record
        household.append(int(line[1:6]))
        income.append(int(line[439:447]))
        state.append(curr_state)
        weight.append(int(line[65:73]))
  f.close()
  temp = pd.DataFrame(
    {"household": household,
     "income": income,
     "state": state,
     "weight": weight})
  temp["state"] = temp["state"].map(st60_to_name4).map(name_to_fips)
  return temp
```

In 2011, the location of the income and weight variables changes. At this point, we can stop manually converting the state codes to FIPS values since we now pull that information directly from the data file.

```
def read_income_2011(filename):
  household = []
  income    = []
```

```
  state     = []
  weight    = []
  f = open(filename)
  for line in f:
    type = int(line[0])
    if type == 1:    # household record
      curr_state = int(line[41:43])
    elif type == 3:  # person record
      household.append(int(line[1:6]))
      income.append(int(line[579:587]))
      state.append(curr_state)
      weight.append(int(line[154:162]))
  f.close()
  return pd.DataFrame(
    {"household": household,
     "income": income,
     "state": state,
     "weight": weight})
```

For 2019 and on, the data files are separate csv files, so we have Pandas parse them.

```
def read_income_2019(h_filename, p_filename):
  household = pd.read_csv(h_filename, header=0)
  person = pd.read_csv(p_filename, header=0)
  temp1 = person[["PH_SEQ", "PTOTVAL", "MARSUPWT"]].rename(
    {"PH_SEQ": "household",
     "PTOTVAL": "income",
     "MARSUPWT": "weight"}, axis=1)
  temp2 = household[["H_SEQ", "GESTFIPS"]].rename(
    {"H_SEQ": "household",
     "GESTFIPS": "state"}, axis=1)
  result = pd.merge(temp1, temp2, how="inner", on="household",
                    sort=False, validate="many_to_one")
  return result
```

Now call the parsing functions and write the new data files.

```
for i in files:
  print("Year {0}".format(i))
  if i < 2019:
    data = eval("read_income_" + str(year_keys[i]))(
      "../data/{0} CPS/{1}".format(i, files[i]))
  else:
    data = read_income_2019("../data/{0} CPS/{1}".format(i, files[i][0]),
                            "../data/{0} CPS/{1}".format(i, files[i][1]))
```

Use the `regions` dictionary to map state fips codes onto region codes. For observations where the fips code does not correspond to a state, this likely indicates some error in the data file itself, so we drop those observations.

```
  data["region"] = data["state"].map(regions)
  temp = data[~((data["region"] >= 1) | (data["region"] <= 4))]
```

```
  if len(temp) > 0:
    print("""
Unable to assign regions for some rows;
will drop them. Portion of data getting
dropped:""")
    print(temp)
    print()
  data = data[data["region"].notna()]
  data["state"] = data["state"].astype(int)
  data["region"] = data["region"].astype(int)
```

Finally, we sort the DataFrame and write it to a txt file.

```
  data.sort_index(axis=1, inplace=True)
  data.sort_values("income", inplace=True)
  data.to_csv("data_{0}.txt".format(i), index=False)
print()
```

File II

print_edges.py

We need code to print the smallest and largest incomes for manual examination. This will allow us to form the correct boundary incomes for the estimation file.

```
import pandas as pd
for i in range(1967, 2024):
  if i != 1970:
    print("YEAR", i)
    d = pd.read_csv("data_{0}.txt".format(i), header=0)["income"]
    print(d.head(30))
    print(d.tail(30))
    print("\n\n\n")
```

File III

bin.py

The file provides a single function, `bin_data`, which accepts a DataFrame and produces a sample density also containing a few extra pieces of information. See Table 4 for a specific list of columns contained in the return value. The general approach here is to first establish bin boundaries, then loop through rows in the income DataFrame. As we loop through incomes, we simultaneously loop through bins and increment the weight on the corresponding bin. Doing both loops at the same time is the smart way to bin the data! Running a loop within a loop—for each income value, loop through bins until we find the correct bin—is hugely inefficient. (It would be much more expedient to use `np.histogram()` here, but I didn't know about that function when I first wrote this code. Live and learn.)

We use linear bins below some cutoff value and logarithmic bins above it. If we implement a combination linear-logarithmic binning scheme, we specify three parameters to determine the bins: linear-logarithmic cutoff, number of linear bins between 0 and the cutoff (which determines linear bin width), and the logarithmic binning factor. See Table 5 for a list of default values. The choice of factor makes the first logarithmic bin lightly larger than the linear bins. After that, we will continue binning using the same factor until we reach the end of the data. We set the parameters as function arguments with default values. That way we can change their values if need be when calling `bin_data` without having to modify the file.

We begin by loading Pandas.

```
import pandas as pd
```

We load the exponential and logarithm functions from Numpy.

```
from numpy import exp
from numpy import log
```

The `bin_data` function will have three arguments. The `data` argument should be a DataFrame, and we check the object type for simple error checking. The `var` argument is the name of a column in `data` that we want to use as the variable for binning, and the `wgt` is an optional argument for the name of a column in `data` that contains survey weights. If the user calls `bin_data` with `wgt`, we will use that column in determining the total bin mass. Otherwise, we weight each observation equally.

Table 4: Columns in Binned Data

Information	Key
Left Endpoint	`left`
Right Endpoint	`right`
Midpoint	`mid`
Length of Bin	`width`
Total Mass in Bin	`mass`
Fraction of Mass in Bin	`freq`
Sample Density	`dens`

```python
def bin_data(data, var, wgt=None, *, cutoff=57000,
             num_linear_from_0=10, factor=1.2):
  if not isinstance(data, pd.DataFrame):
    raise ValueError("First argument of bin_data should be a DataFrame")
  if var not in data:
    raise KeyError("Variable to bin is not a column in the data")
  if isinstance(wgt, type(None)):
    called_with_wgt = False
  else:
    if wgt not in data:
      raise KeyError("Weight variable is not a column in the data")
    called_with_wgt = True
```

To make sure the data is monotonic, we sort it ourselves. To avoid any potential issues with indexing, we also reset the index.

```python
  data_to_use = data.sort_values(var).reset_index(drop=True)
  n = len(data_to_use)
```

We create the linear portion of the bins as a list. For the linear bins, we want to have roughly `num_linear_from_0` linear bins between 0 and the cutoff, and we subtract 0.5 in order to guarantee a bin midpoint at 0. We calculate the linear bin width by dividing the cutoff by the adjusted number of bins. The `min` and `max` are minimum and maximum incomes in the data.

```python
  lin_width = cutoff / (num_linear_from_0 - 0.5)
  min = data_to_use.loc[(0, var)]
  max = data_to_use.loc[(n-1, var)]
```

We calculate the number of linear bins by doing integer division. The variable `remainder` is the distance from the minimum income to the left endpoint of the first full linear bin. Once we know the `remainder`, we can easily calculate the linear bin endpoints.

```python
  num_linear_bins = int((cutoff - min) // lin_width)
  remainder = (cutoff - min) % lin_width
  bins = [min]
  temp = min + remainder
  while temp < max and temp <= cutoff:
    bins.append(temp)
    temp = temp + lin_width
  temp = cutoff * factor
  while temp < max:
    bins.append(temp)
    temp = temp * factor
```

Table 5: Default Bin Specifications

Specification	Keyword	Default Value
Linear-logarithmic Cutoff	`cutoff`	\$57,000
Linear Bins from 0 to Cutoff	`num_lin_from_0`	$10 - 0.5 = 9.5$ (width \$6000)
Logarithmic Factor	`factor`	1.2

We copy bins, remove the zeroth entry, and append max to the copy. We create a DataFrame where bins becomes the left column, and the updated next_bins becomes the right column. We use binned to refer to our eventual return value.

```
next_bins = bins.copy()
next_bins.pop(0)
next_bins.append(max)
binned = pd.DataFrame({"left":bins, "right":next_bins})
binned["mid"] = 0.5 * (binned.left + binned.right)
binned["width"] = binned.right - binned.left
```

Now we do the actual binning. We create a list of total weights in each bin. We keep track of:

- i: the current row in data
- j: the current index in the bin list
- k: the current total weight for bin j

First we make the list. We loop through range(len(bins)), and on each iteration, we set k to be 0, then progressively increase i until we reach either a row where the income entry equals the next entry in bins or the final row in the data. The last bin in the data is a bit odd in terms of the loop, so we code it manually instead.

```
mass_vals = [0 for _ in bins]
i = 0
for j in range(len(bins)):
  k = 0
  while data_to_use.loc[(i, var)] < next_bins[j]:
    if called_with_wgt:  # if weight was specified, use survey weights
      k = k + data_to_use.loc[(i, wgt)]
    else:                # otherwise, each point adds mass of 1
      k = k + 1
    i = i+1
  mass_vals[j] = k
```

Now add the last data points. We could have multiple observations where the variable takes value max, so we can't simply add the final row information to mass_vals. Instead, we count backwards to the first data entry that is different from max. (This approach assumes max $\neq$ min, which should not be a problem.)

```
k = 0
i = n-1
while data_to_use.loc[(i, var)] == max:
  if called_with_wgt:
    k = k + data_to_use.loc[(i, wgt)]
  else:
    k = k + 1
  i = i-1
mass_vals[-1] = mass_vals[-1] + k
```

Finally, we add the mass_vals information to binned, calculate the sample density, and return the result.

```
binned["mass"] = mass_vals
binned["freq"] = binned.mass / binned.mass.sum()
binned["dens"] = binned.freq / binned.width
return binned
```

File IV

estimate_parameters.py

This file contains the code for estimating parameters. Big picture, this file is where the heavy lifting happens. Contents include: (1) the code for calculating the Kolmogorov-Smirnov statistic/objective function, (2) functions to estimate parameters for the models of interest, and (3) several other functions associated with different models. The setup will be as follows:

1. General setup such as loading modules and defining pointers.
2. Define the general functions we need. These are the method-of-moment estimators for the Fisk distribution and the Kolmogorov-Smirnov objective function calculation.
3. Define the estimation functions and other functions associated with specific models. This will be where the numerics happen and is the bulk of the code in this file.

In this section, we outline the structure of the file and functions encountered. See also Table 6 for a list of models used in this section.

We use a variety of functions in the estimation process. In general, the function argument `data` should be a DataFrame, and `var` should be a column name in the DataFrame that serves as the variable of interest. We set `wgt` to be `None` by default, but if present, it should be the name of a column of survey weights. Previous versions of this file assumed that `var` was `"income"` and `wgt` was `"weight"`, but this approach provides a more portable and general interface that we can use for other datasets if need be.

Calculating the Kolmogorov-Smirnov statistic can be computationally intensive if we aren't careful. We will write all `cdf_` functions such that they accept a set of x-values in the form of a numpy array. Then we calculate the distribution using vectorized operations, which is much faster than using `Series.map()`. The `len_data` argument will store the length of the data array.

For each model, we estimate parameters in a multi-step process. Most parameters are easier to find if we know c, some much easier, so we estimate as follows:

Table 6: Candidate Distributions

Distribution (Shifted)	Parameters	How Many?	String Key
Generalized Beta, Type II	α, β, p, q, c	5	`GB2`
Dagum	α, β, p, c	4	`Dagum`
Burr	α, β, q, c	4	`Burr`
Fisk	α, β, c	3	`Fisk`
Inverse Gamma	α, β, c	3	`InvG`
Davis	α, β, and c	3	`Davis`
Constant-Shift Inverse Gamma	α, β	2	`CS_InvG`
Constant-Shift-Shape Inverse Gamma	α	smallest $\rightarrow$ 1	`CSS_InvG`
Log-Normal/Pareto Cutoff	μ, σ^2, α, x_0, k, c	6	`LogN_P_cut`
Log-Normal/Pareto Mixture	μ, σ^2, α, x_0, γ, c	6	`LogN_P_mix`

General functions:

- `validate_var_wgt(`$\langle$*data*$\rangle$`,`$\langle$*var*$\rangle$`,`$\langle$*wgt*$\rangle$`)`—checks that *data* is a DataFrame, if *var* is a column in *data*, and whether *wgt* is a column in *data* if present.
- `make_ecdf(`$\langle$*data*$\rangle$`,`$\langle$*var*$\rangle$`,`$\langle$*wgt*$\rangle$`=None)`—creates a DataFrame containing the empirical cumulative distribution of the *data*.
- `kolmogorov_smirnov(`$\langle$*cdf*$\rangle$`, *, `$\langle$*ecdf*$\rangle$`=None, `$\langle$*x*$\rangle$`=None, `$\langle$*y*$\rangle$`=None, `$\langle$*data*$\rangle$`=None, `$\langle$*var*$\rangle$`=None, `$\langle$*wgt*$\rangle$`=None)`—calculates the value of the objective function based on the Kolmogorov-Smirnov statistic, i.e. $\sup|F(x) - G(x)|$. Here *cdf* should be a (symbolc) cumulative distribution function. To evaluate the fit to data, the functiona accepts either an empirical cumulative distribution ecdf or a DataFrame *data*. In the *ecdf* case, the arguments x and y should be column names in *ecdf* corresponding to x and y-values, and in the *data* case, the var and *wgt* arguments work the same way as in other places. Specifying *ecdf* takes precedence over *data*.
- `fisk_moments(`$\langle$*data*$\rangle$`,`$\langle$*var*$\rangle$`,`$\langle$*wgt*$\rangle$`=None)`—function that accepts data and returns the moment estimators for the Fisk distribution. Used as the initial guess for various estimators.

1. Assume some value of c. Transform incomes according to this test value of c (subtract c from all incomes), and calculate the other parameters conditional on this test value for c. In other words, apply the estimator for unshifted data to the set of transformed incomes.
2. In all cases, this calculation will involve some sort of maximum likelihood for the remaining parameters. For `Burr`, `Dagum`, and `Fisk`, we maximize the likelihood purely

Model-specific functions:

- `cdf_`$\langle$*dist*$\rangle$`(`$\langle$*x*$\rangle$`,`$\langle$*params*$\rangle$`,`$\langle$*len_data*$\rangle$`=1)`—function that accepts a value of x and a list of parameters and returns a cumulative distribution value. This function should treat *data* as a numpy array and vectorize the operations to find the cumulative distribution.
- `density_`$\langle$*dist*$\rangle$`(`$\langle$*x*$\rangle$`,`$\langle$*params*$\rangle$`)`—function that accepts a value of x and a list of parameters and returns a density value.
- `L_`$\langle$*dist*$\rangle$`(`$\langle$*data*$\rangle$`,`$\langle$*params*$\rangle$`,`$\langle$*var*$\rangle$`,`$\langle$*wgt*$\rangle$`=None)`—function that accepts a DataFrame and a list of parameters and returns a log-likelihood value.
- `estimate_`$\langle$*dist*$\rangle$`(`$\langle$*data*$\rangle$`,`$\langle$*var*$\rangle$`,`$\langle$*wgt*$\rangle$`=None)`—for each distribution, the helper function that estimates parameters; accepts a DataFrame with 2 columns where one column is income and one column is normalized weights. THERE WILL also be other functions involved internally in the estimation for each distribution. For example, each model has a corresponding `estimate_`$\langle$*dist*$\rangle$`_fit_for_c` function that calculates the value of the objective function.

numerically. For `Inv_G` and `LogN_P_cut`, we numerically solve a single equation that arises from algebraic calculation. For `GB2`, `Davis`, and `LogN_P_mix`, the estimation is more complicated.

3. Evaluate the test c and other parameters using the objective function derived from the Kolmogorov-Smirnov statistic. Numerically find the value of c that minimizes the objective function.
4. We may have to calculate other parameters in the same step as c. For example, the `LogN_P_cut` distribution has a cutoff k, and we simultaneously fix test values of c and k. Then we calculate the optimal c-k pair that minimizes the objective function.

The estimation functions in this file return dictionary containing the value of the objective function under the key `fit` and a list of parameter estimates under the key `parameters`.

Setup

First import Pandas, Numpy, and two Scipy submodules.

```
import numpy          as np
import pandas         as pd
import scipy.optimize as opt
import scipy.special  as spec
import time
```

Some debugging stuff. In previous versions of this code, numpy was giving a VisibleDeprecationWarning. We also disable the SettingWithCopyWarning from Pandas.

```
#np.warnings.filterwarnings('error', category=np.VisibleDeprecationWarning)
pd.options.mode.chained_assignment = None
def the_time():
  print("The time is", time.asctime())
```

New pointers to important Numpy and Scipy variables.

```
pi    = np.pi
e     = np.e
exp   = np.exp
floor = np.floor
log   = np.log
sqrt  = np.sqrt
min   = opt.minimize
min_s = opt.minimize_scalar
root  = opt.root_scalar
B     = spec.beta
erf   = spec.erf
G     = spec.gamma
I     = spec.betainc
Phi   = spec.ndtr
Phinv = spec.ndtri
psi   = spec.digamma
Q     = spec.gammaincc
def psi1(x):
```

```
  return spec.polygamma(1,x)
def Phi_prime(x):
  return exp(-x**2 / 2) / sqrt(2 * pi)
def zeta(x):
  return 1 + sp.zetac(x)
def zeta(x):
  return 1 + spec.zetac(x)
```

We use the the approximation for $\log\Gamma(z)$ from Chen (2016) when z is large to avoid any numerical problems.[1] The approximation is

$$\log\Gamma(x+1) \approx \log\sqrt{2\pi x} + x(\log x - 1) + \left(x^2 + \frac{53}{210}\right)\log\left(1 + \frac{1}{12x^3 + \frac{24}{7}x - \frac{1}{2}}\right)$$

When $x \geq 10$, this approximation is accurate to one part in a billion. It's probably easier (better?) to use the built-in gamma function in Scipy, but I didn't think about that when I wrote the code. I am confident that for practical purposes, the improvement in accuracy is at best tiny, so it wouldn't gain much to rewrite the code. Plus this way we get to use a fun approximation theorem.

```
Gamma_frac1 = 53/210
Gamma_frac2 = 24/7
def log_G(z):
  return log(G(z))
def log_G_approx(z):
  x = z - 1
  term1 = 0.5 * log(2 * pi * x)
  term2 = x * (log(x) - 1)
  term3 = (x * x + Gamma_frac1)
  term4 = log(1 + 1 / (12 * (x * x * x) + Gamma_frac2 * (x) - 0.5))
  return term1 + term2 + term3 * term4
```

Now create several empty dictionaries. We will fill them up as we get to them.

```
distribution = {}  # cumulative distribution functions
density      = {}  # densities
likelihood   = {}  # likelihoods
estimator    = {}  # estimators
```

General Functions

The first function we code is to validate input. This function does the same thing as the error checking at the beginning of `bin.py`.

```
def validate_var_wgt(data, var, wgt):
  if not isinstance(data, pd.DataFrame):
    raise ValueError("First argument of function should be a DataFrame")
  if var not in data:
```

[1] Chen, Chao-Ping. 2016. "A More Accurate Approximation for the Gamma Function." *Journal of Number Theory* 164: 417–428.

```
    raise KeyError("Variable to analyze is not a column in the data")
  if wgt and wgt not in data:
    raise KeyError("Weight variable is not a column in the data")
```

We will call `validate_var_wgt` at the start of each estimating function.

We need a function to make the empirical cumulative distribution. The result is a DataFrame with columns `x` and `y` containing x and y-values respectively. Because of the discrete nature of data, we will repeat each x-value twice, and the final emprical cumulative distribution DataFrame will contain two rows for each x-value. For a given x, the first corresponding y-value will be $F(x^-)$, and the second y-value will be $F(x^+)$, where F denotes the empirical cumulative distribution. This means the return `temp` DataFrame value will be twice as long as the original data.

```
def make_ecdf(data, var, wgt=None):
  validate_var_wgt(data, var, wgt)
  temp = data.copy()
  if not wgt:
    wgt = "y"
    if var == "y":
      wgt = "y1"
    temp[wgt] = 1
```

Make sure everything is sorted before we do stuff with it.

```
  if not temp[var].is_monotonic_increasing:
    temp = temp.sort_values(var).reset_index(drop=True)
```

We have to calculate the total weight given to unique x-values. (In other words, account for repeated x-values.) The way to do this is to group by `var` and then reset the index. Note that the contatenation uses `axis=0`, i.e. is vertical concatenation.

```
  temp = temp[[var, wgt]].groupby(var).sum()  # get unique var values
  temp.index.set_names(None, inplace=True)
  temp = temp.cumsum() / temp[wgt].sum()      # turn wgt into cdf
  temp[var] = temp.index                      # add var vals back into data
  temp = pd.concat([temp, temp]).sort_values(var).reset_index(drop=True)
```

Now shift the weight column, manually add the weight in the zeroth row, and return.

```
  temp[wgt] = temp[wgt].rename(lambda x: x + 1)
  temp.loc[(0, wgt)] = 0
  return temp
```

The resulting x-column will have name `var`, and the y column will have `wgt` as its name if specified in `make_ecdf`. Otherwise, it will be called `"y"` or `"y1"`.

We code the objective function to calculate the Kolmogorov-Smirnov statistic. For two distribution functions F and G, the statistic is

$$\sup |F(x) - G(x)| = \|F - G\|_\infty$$

In our case, F is the cumulative distribution under the model of interest, and G is the empirical cumulative distribution. We begin by validating the arguments and making sure the user specified the required information.

```
def kolmogorov_smirnov(cdf, *, ecdf=None, x=None, y=None,
```

```
                               data=None, var=None, wgt=None):
  if isinstance(ecdf, type(None)) and isinstance(data, type(None)):
    raise ValueError("Please specify ecdf or data to use Kolmogorov-Smirnov")
  elif isinstance(ecdf, type(None)) and not isinstance(data, type(None)):
    validate_var_wgt(data, var, wgt)
    ecdf = make_ecdf(data, var, wgt)
    x = var
    if not wgt and var != "y":
      y = "y"
    elif not wgt and var == "y":
      y = "y1"
    elif wgt:
      y = wgt
  elif not isinstance(ecdf, type(None)):
    validate_var_wgt(ecdf, x, None)
    validate_var_wgt(ecdf, y, None)
    ecdf = ecdf.copy()
  else:
    raise RuntimeError("Something weird happened")
```

Now we calculate and return the statistic.

```
  #ecdf = ecdf[(ecdf[y] > 0) & (ecdf[y] < 1)]
  vals = cdf(np.array(ecdf[x]))  # model cdf values
  emp_vals = np.array(ecdf[y])   # empitical cdf values
```

We can add weights for a weighted Kolmogorov-Smirnov statistic.

```
  #avg = (vals + ecdf[y]) / 2
  #weights = (1 / (avg * (1-avg))) ** (1/2)
  #weights = 1
  return np.max(abs(vals - emp_vals))
```

Before getting into individual distributions, we also code the Fisk moment estimators. The function `fisk_moments` accepts a DataFrame containing columns `income` and `weights` and calculates the moment estimators for the Fisk distribution as suggested by Graf and Nedyalkova (2014). The Fisk distribution has shape parameter α and scale parameter β, and the estimators are

$$\alpha = \frac{\pi}{\sqrt{3v}}$$
$$\beta = e^m$$

where

$$m = \frac{1}{n}\sum_{i=1}^{n} w_i \log(x_i) \qquad\qquad v = \frac{1}{n}\sum_{i=1}^{n} w_i(\log(x_i) - m)^2$$

Here n is the total normalized survey weights (total number of observations), and w_i is the (normalized) survey weight on respondent i. The function stores the length of the data in `n`, calculates the intermediate quantities, and returns a 2-list containing the estimates.

```
def fisk_moments(data, var, wgt=None):
  if wgt:
    n = data.weight.sum()
    m = (1/n) * (data[wgt] * log(data[var])).sum()
    v = (1/n) * (data[wgt] * (log(data[var]) - m)**2).sum()
  else:
    n = len(data)
    m = (1/n) * log(data[var]).sum()
    v = (1/n) * ((log(data[var]) - m)**2).sum()
  a = pi / sqrt(3 * v)
  b = exp(m)
  return [a, b]
```

We will use this function in GB2, Burr, Dagum, and Fisk estimation.

Specific Distributions

GB2 Generalized Beta, Type 2. Parameters:

- $\alpha > 0$: shape. It's unclear if this parameter needs to be strictly positive or just non-zero. Same with the analogous parameter in child distribution such as Burr and Dagum. Wikipedia has positive values for α, but McDonald and Ransom (2008) say that α can also be negative. If $\alpha < 0$, we should replace α in the numerator by $|\alpha| = -\alpha$. In any event, testing and inspection suggest that, at least for GB2, $\alpha > 0$, so I think it's probably fine to use this formula.
- $\beta > 0$: scale
- $p > 0$: shape
- $q > 0$: shape
- c: shift (for shifted GB2)

The cumulative distribution is

$$F(x) = I\left(\frac{\xi}{1+\xi}; p, q\right)$$

where I is the regularized incomplete beta function and

$$\xi = \left(\frac{x-c}{\beta}\right)^{\alpha}$$

```
def cdf_GB2(x, params):
  a, b, p, q, c = params
  x = x.astype(np.float64)
  mask = x <= c
  x[mask] = 0
  frac = exp(-a * (log(x[~mask] - c) - log(b)))
  x[~mask] = I(p, q, 1/(1+frac))
  return x
```

The density is

$$y = \frac{\alpha\,(x-c)^{\alpha p - 1}}{\beta^{\alpha p} B(p,q)\left(1 + \left(\frac{x-c}{\beta}\right)^{\alpha}\right)^{p+q}}$$

where B is the beta function. We rewrite the numerator and denominator in terms of a logarithm in case any factors are particularly small or particularly large. The beta function can be difficult for a computer to handle numerically depending on how it is implemented. We use the previous approximation for $\log\Gamma(z)$ to calculate $\log B(p,q)$ (specifically $\log\Gamma(p)$, $\log\Gamma(q)$, and $\log\Gamma(p+q)$) without any risk of numerical complications. Code:

```
def density_GB2(x, params):
  a, b, p, q, c = params
```

We manually check if p, q, and $p+q$ are each less than 10. If yes, we can use the built-in gamma function, and otherwise, we approxmate.

```
  if p < 10:
    log_G_p = log_G(p)
  else:
    log_G_p = log_G_approx(p)
  if q < 10:
    log_G_q = log_G(q)
  else:
    log_G_q = log_G_approx(q)
  if p + q < 10:
    log_G_pq = log_G(p + q)
  else:
    log_G_pq = log_G_approx(p + q)
```

And now calculate the density.

```
       if x <= c:
         return 0
       else:
         num   = log(a) + (a*p-1) * log(x-c)           # numerator
         denom = (    (a*p) * log(b) + log_G_p
                 + log_G_q - log_G_pq
                 + (p+q) * log(1+((x-c)/b)**a)    )    # denominator
         return exp(num - denom)
```

The likelihood is

$$\begin{aligned} L = n\log\alpha + (\alpha p - 1)\sum_{i=1}^{n} w_i \log(x_i - c) - n\alpha p \log\beta \\ - n\log(B(p,q)) - (p+q)\sum_{i=1}^{n} w_i \log\left(1 + \left(\frac{x_i - c}{\beta}\right)^{\alpha}\right) \end{aligned}$$

where w_i is the survey weights. The `L_GB2` calculates the likelihood under `params` with respect to the `data`. We calculate each term separately and return their sum. Code:

```
def L_GB2(data, params, var, wgt=None):
  validate_var_wgt(data, var, wgt)
  a, b, p, q, c = params
```

Again we manually check if p, q, and $p+q$ are each less than 10.

```
  if p < 10:
    log_G_p = log_G(p)
  else:
    log_G_p = log_G_approx(p)
  if q < 10:
    log_G_q = log_G(q)
  else:
    log_G_q = log_G_approx(q)
  if p + q < 10:
    log_G_pq = log_G(p + q)
  else:
    log_G_pq = log_G_approx(p + q)
```

And calculate the likelihood.

```
  if wgt:
    n = data[wgt].sum()
    term1 = n * log(a)
    term2 = (a*p - 1) * (data[wgt] * log(data[var] - c)).sum()
    term3 = -n * a * p * log(b)
    term4 = -n * (log_G_p + log_G_q - log_G_pq)
    term5 = -(p+q) * (data[wgt] * log(1 + ((data[var] - c)/b)**a)).sum()
  else:
    n = len(data)
    term1 = n * log(a)
    term2 = (a*p - 1) * log(data[var] - c).sum()
    term3 = -n * a * p * log(b)
    term4 = -n * (log_G_p + log_G_q - log_G_pq)
    term5 = -(p+q) * log(1 + ((data[var] - c)/b)**a).sum()
  return term1 + term2 + term3 + term4 + term5
```

And the unshifted verison is

```
def L_GB2_unshift(data, params, var, wgt=None):
  a, b, p, q = params
  return L_GB2(data, [a, b, p, q, 0], var, wgt)
```

Differentiating the likelihood function gives us

$$\frac{dL}{d\alpha} = \frac{n}{\alpha} + p\sum_{i=1}^{n} w_i \log(x_i - c) - np\log\beta - (p+q)\sum_{i=1}^{n} w_i \frac{\left(\frac{x_i-c}{\beta}\right)^{\alpha}\log\left(\frac{x_i-c}{\beta}\right)}{1+\left(\frac{x_i-c}{\beta}\right)^{\alpha}}$$

$$= \frac{n}{\alpha} + p\sum_{i=1}^{n} w_i \log\left(\frac{x_i - c}{\beta}\right) - (p+q)\sum_{i=1}^{n} \frac{w_i \log\left(\frac{x_i-c}{\beta}\right)}{1+\left(\frac{\beta}{x_i-c}\right)^{\alpha}}$$

$$\frac{dL}{d\beta} = -\frac{n\alpha p}{\beta} - (p+q)\sum_{i=1}^{n} w_i \frac{(x_i-c)^{\alpha}(-\alpha)\beta^{-\alpha-1}}{1+\left(\frac{x_i-c}{\beta}\right)^{\alpha}}$$

$$= -\frac{n\alpha p}{\beta} - (p+q)\frac{\alpha}{\beta}\sum_{i=1}^{n} \frac{w_i}{1+\left(\frac{\beta}{x_i-c}\right)^{\alpha}}$$

Setting both expressions to 0 and solving for p and q gives us

$$p = \frac{1}{\alpha}\left(\frac{1}{n}\sum_{i=1}^{n}\frac{w_i}{1+\left(\frac{\beta}{x_i-c}\right)^{\alpha}}\right)\left(\frac{1}{n}\sum_{i=1}^{n}\frac{w_i\log\left(\frac{x_i-c}{\beta}\right)}{1+\left(\frac{\beta}{x_i-c}\right)^{\alpha}}\right.$$
$$\left. - \left[\frac{1}{n}\sum_{i=1}^{n}\frac{w_i}{1+\left(\frac{\beta}{x_i-c}\right)^{\alpha}}\right]\left[\frac{1}{n}\sum_{i=1}^{n} w_i \log\left(\frac{x_i-c}{\beta}\right)\right]\right)^{-1}$$

$$q = \frac{1}{\alpha}\left(1-\frac{1}{n}\sum_{i=1}^{n}\frac{w_i}{1+\left(\frac{\beta}{x_i-c}\right)^{\alpha}}\right)\left(\frac{1}{n}\sum_{i=1}^{n}\frac{w_i\log\left(\frac{x_i-c}{\beta}\right)}{1+\left(\frac{\beta}{x_i-c}\right)^{\alpha}}\right.$$
$$\left. - \left[\frac{1}{n}\sum_{i=1}^{n}\frac{w_i}{1+\left(\frac{\beta}{x_i-c}\right)^{\alpha}}\right]\left[\frac{1}{n}\sum_{i=1}^{n} w_i \log\left(\frac{x_i-c}{\beta}\right)\right]\right)^{-1}$$

so if we know α and β, we can find p and q. Ostensibly. Previous versions of this document made use of these equations for p and q, but that does not work. We want the values of p and q that make

$$\frac{dL}{dp} = \frac{dL}{dq} = 0,$$

not the values that cause α and β to be maxima. Going back to the likelihood function and differentiating again gives us

$$\frac{dL}{dp} = \alpha\sum_{i=1}^{n} w_i \log(x_i - c) - n\alpha\log\beta - n[\psi(p) - \psi(p+q)]$$
$$- \sum_{i=1}^{n} w_i \log\left(1+\left(\frac{x_i-c}{\beta}\right)^{\alpha}\right)$$
$$= \alpha\sum_{i=1}^{n} w_i \log\left(\frac{x_i-c}{\beta}\right) - n[\psi(p)-\psi(p+q)] - \sum_{i=1}^{n} w_i \log\left(1+\left(\frac{x_i-c}{\beta}\right)^{\alpha}\right)$$

$$\frac{dL}{dq} = -n[\psi(q) - \psi(p+q)] - \sum_{i=1}^{n} w_i \log\left(1 + \left(\frac{x_i - c}{\beta}\right)^{\alpha}\right),$$

where ψ is the digamma function. It follows that we are looking for p and q such that

$$\begin{aligned}
\psi(p+q) - \psi(p) &= \frac{1}{n}\sum_{i=1}^{n} w_i \log\left(1 + \left(\frac{x_i - c}{\beta}\right)^{\alpha}\right) - \frac{\alpha}{n}\sum_{i=1}^{n} w_i \log\left(\frac{x_i - c}{\beta}\right) \\
\psi(p+q) - \psi(q) &= \frac{1}{n}\sum_{i=1}^{n} w_i \log\left(1 + \left(\frac{x_i - c}{\beta}\right)^{\alpha}\right)
\end{aligned}$$

Numerically solving for p and q is more complicated than using the previous equations that give p and q exactly—hence why previous versions of this document did not do this. The `find_pq` function will perform the numerical root-finding. The `log_term1` is the sum that appears in both equations, and the `log_term2` is the sum from the $\psi(p)$ condition. We assume that the `data` has been standardized, so we can ignore β and c.

```
def estimate_GB2_find_pq(data, alpha, var, wgt):
  if wgt:
    n = data[wgt].sum()
    log_sum1 = (data[wgt] * log(1 + data[var] ** alpha)).sum() / n
    log_sum2 = alpha * (data[wgt] * log(data[var])).sum() / n
  else:
    n = len(data)
    log_sum1 = log(1 + data[var] ** alpha).sum() / n
    log_sum2 = alpha * log(data[var]).sum() / n
```

Now code the function to solve (`mle_foc`) and its Jacobian.

```
  def mle_foc(p_and_q):
    p, q = p_and_q
    p_term = psi(p + q) - psi(p) - log_sum1 + log_sum2
    q_term = psi(p + q) - psi(q) - log_sum1
    return [float(p_term), float(q_term)]
  def d_mle_foc(p_and_q):
    p, q = p_and_q
    return [[psi1(p + q) - psi1(p), psi1(p + q)          ],
            [psi1(p + q)          , psi1(p + q) - psi1(q)]]
```

We find p and q and return. The initial guess $p = 2$ and $q = 0.5$ seems to work well.

```
  return opt.root(mle_foc, x0=[3, 1], jac=d_mle_foc).x
```

The `find_a` function will return the value of α that maximizes the likelihood using the p and q estimates conditional on α. The `test_L` function will be our objective function to maximize.

```
def estimate_GB2_find_a(data, var, wgt):
  def test_L(a):
```

```
    p, q = estimate_GB2_find_pq(data, a, var, wgt)
    return L_GB2(data, [a, 1, p, q, 0], var, wgt)
  alpha = min(lambda x: -test_L(x), x0=1, method="Nelder-Mead",
    bounds=[(0.1,None)]).x[0]
  p, q = estimate_GB2_find_pq(data, alpha, var, wgt)
  return [alpha, p, q]
```

The `fit_for_c` function accepts c and β values. It standardizes the data, finds remaining parameters using `estimate_GB2_find_a`, and returns the objective function values and parameters in a dictionary.

```
def estimate_GB2_fit_for_c(data, c, b, var, wgt, ecdf, x, y):
  stand_data = data.copy()
  stand_data[var] = (stand_data[var] - c) / b
  stand_data = stand_data[stand_data.income > 0]
  #print("finding a, p, q:", time.asctime())
  a, p, q = estimate_GB2_find_a(stand_data, var, wgt)
```

The `fit` is the objective function, and the `parameters` are the estimates.

```
  #print("fitting model:", time.asctime())
  def F(x):
    return cdf_GB2(x, [a, b, p, q, c])
  temp = {
    "fit": kolmogorov_smirnov(F, ecdf=ecdf, x=x, y=y),
    "parameters": [a, b, p, q, c]}
  #print("done:", time.asctime())
  return temp
```

Before we code the final estimating function, we need a housekeeping function to extract β and c values from the return value of the `fit_for_c` function.

```
def estimate_GB2_get_cb(dict):
  return [dict["parameters"][-1], dict["parameters"][1]]
```

Finally we estimate β and c by minimizing the Kolmogorov-Smirnov statistic. Unfortunately, the `fit_for_c` function is not quite well-behaved (i.e. practically identifiable) enough to algorithmically minimize the objective function, at least not using the Nelder-Mead algorithm as coded in Scipy. Accordingly, we use a brute-force estimator to find the optimum (c, β) pair over three iterations. In the first iteration, we use a partition mesh size of \$500, and in the second iteration, we use a partition mesh size of \$100 around the ten best-fitting pairs from the first iteration. In the third iteration, we use a partition mesh size of \$10 around the ten best-fitting pairs from the second iteration.

```
def estimate_GB2(data, var, wgt=None, *, ecdf=None, x=None, y=None):
  validate_var_wgt(data, var, wgt)
  if isinstance(ecdf, type(None)):
    ecdf = make_ecdf(data, var, wgt)
    x = var
```

```
  if wgt:
    y = wgt
  else:
    if var == "y":
      y = "y1"
    else:
      y = "y"
```

First iteration. The ⟨*b or c*⟩`_vals` lists contain the test values of the variables we are using, and the `best_fits` list stores corresponding values of the objective function.

```
b_vals = [10000 + 500*i for i in range(81)]
c_vals = [-15000 + 500*i for i in range(21)]
best_fits = []
the_time()
for b in b_vals:
  for c in c_vals:
    temp = estimate_GB2_fit_for_c(data, c, b, var, wgt, ecdf, x, y)
```

We modify `best_fits` dynamically. If the list has fewer than ten entries, we add the current results. Otherwise, we check whether the current result fits better than the last entry in `best_fits`. If no, we ignore. If yes, we insert it into `best_fits` at the first position where the current result fits worse than previous items in `best_fits`. In this way, we maintain a list of best-fitting parameter combinations ordered by the value of their objective function. The while-loop we're using now is probably not the fastest way to implement the ordered list. Oh well.

```
    if len(best_fits) < 10 and temp["parameters"][0] > 0.1:
      best_fits.append(temp)
    else:
```

We make sure to exclude possibilities where the estimate of α is 0.1.

```
      if temp["fit"] < best_fits[-1]["fit"] and temp["parameters"][0] > 0.1:
        i = 0
        while temp["fit"] > best_fits[i]["fit"]:
          i = i + 1
        best_fits.insert(i, temp)
        best_fits.pop()
```

For the second iteration, we store the (c, β) values in `pairs`. First extract the values from `best_fits`, and then we populate the grid of parameter values to test. We turn `pairs` into a set to avoid duplicate parameter pairs.

```
pairs = [estimate_GB2_get_cb(i) for i in best_fits]
pairs = [(p[0] - 500 + 100*i, p[1]) for p in pairs for i in range(11)]
pairs = [(p[0], p[1] - 500 + 100*i) for p in pairs for i in range(11)]
pairs = set(pairs)
best_fits = []
```

```
the_time()
for p in pairs:
  temp = estimate_GB2_fit_for_c(data, *p, var, wgt, ecdf, x, y)
  if len(best_fits) < 10 and temp["parameters"][0] > 0.1:
    best_fits.append(temp)
  elif temp["parameters"][0] > 0.1:
    if temp["fit"] < best_fits[-1]["fit"] and temp["parameters"][0] > 0.1:
      i = 0
      while temp["fit"] > best_fits[i]["fit"]:
        i = i + 1
      best_fits.insert(i, temp)
      best_fits.pop()
```

Now do this again for the third iteration. This time, we save just the best-fitting entry, not the whole list.

```
pairs = [estimate_GB2_get_cb(i) for i in best_fits]
pairs = [(p[0] - 100 + 10*i, p[1]) for p in pairs for i in range(21)]
pairs = [(p[0], p[1] - 100 + 10*i) for p in pairs for i in range(21)]
pairs = set(pairs)
solution = {"fit":2}
the_time()
for p in pairs:
  temp = estimate_GB2_fit_for_c(data, *p, var, wgt, ecdf, x, y)
  if temp["fit"] < solution["fit"] and temp["parameters"][0] > 0.1:
    solution = temp
the_time()
return solution
```

And add functions to dictionaries.

```
distribution["GB2"] = cdf_GB2
density["GB2"]      = density_GB2
likelihood["GB2"]   = L_GB2
estimator["GB2"]    = estimate_GB2
```

Done with GB2!

Dagum Dagum distribution. Parameters:

- $\alpha > 0$: shape.
- $\beta > 0$: scale.
- $p > 0$: shape.
- c: shift. (For shifted Dagum)

The cumulative distribution is

$$y = \left(1 + \left(\frac{x - c}{\beta}\right)^{-\alpha}\right)^{-p}$$

```
def cdf_Dagum(x, params):
  a, b, p, c = params
  x = x.astype(np.float64)
  mask = x <= c
  x[mask] = 0
  frac = exp(-a * (log(x[~mask] - c) - log(b)))
  x[~mask] = (1 + frac) ** (-p)
  return x
```

The density is

$$y = \frac{\alpha p (x-c)^{\alpha p - 1}}{\beta^{\alpha p}\left(1+\left(\frac{x-c}{\beta}\right)^{\alpha}\right)^{p+1}}$$

As with GB2, we calculate the numerator an denominator in logs to avoid any potential numerical issues.

```
def density_Dagum(x, params):
  a, b, p, c = params
  if x <= c:
    return 0
  else:
    num = log(a) + log(p) + (a*p-1) * log(x-c)
    denom = (a*p) * log(b) + (p+1) * log(1 + ((x-c)/b)**a)
    return exp(num - denom)
```

The likelihood is

$$\begin{aligned} L = n\log\alpha + n\log p + (\alpha p - 1)\sum_{i=1}^{n} w_i \log(x_i - c) \\ - \alpha p \log\beta - (p+1)\sum_{i=1}^{n} w_i \log\left(1+\left(\frac{x-c}{\beta}\right)^{\alpha}\right) \end{aligned}$$

```
def L_Dagum(data, params, var, wgt=None):
  validate_var_wgt(data, var, wgt)
  a, b, p, c = params
  if wgt:
    n = data[wgt].sum()
    term1 = n * log(a)
    term2 = n * log(p)
    term3 = (a*p-1) * (data[wgt] * log(data[var] - c)).sum()
    term4 = -n * a * p * log(b)
    term5 = -(p+1) * (data[wgt] * log(1 + ((data[var] - c)/b)**a)).sum()
  else:
    n = len(data)
    term1 = n * log(a)
    term2 = n * log(p)
    term3 = (a*p-1) * log(data[var] - c).sum()
```

```
    term4 = -n * a * p * log(b)
    term5 = -(p+1) * log(1 + ((data[var] - c)/b)**a).sum()
  return term1 + term2 + term3 + term4 + term5
```

Unshifted version is

```
def L_Dagum_unshift(data, params, var, wgt=None):
  a, b, p = params
  return L_Dagum(data, [a, b, p, 0], var, wgt)
```

Once again, we use the moment estimators of the Fisk distribution as the initial guess for the Dagum likelihood maximization.

```
def estimate_Dagum_initial(data, var, wgt):
  a, b = fisk_moments(data, var, wgt)
  return [a, b, 1]
```

The estimation function is similar to `GB2`.

```
def estimate_Dagum_unshift(data, var, wgt):
  guess = estimate_Dagum_initial(data, var, wgt)
  def neg_L(params):
    return -L_Dagum_unshift(data, params, var, wgt)
  sol = min(neg_L, x0=guess, method="Nelder-Mead")
  a, b, p = sol.x
  return [a, b, p]
```

The `fit_for_c` function takes a vlaue of c, finds the parameter estimates for that c, calculates the Kolmogorov-Smirnov statistic, and returns everything in a dictionary. Same as `GB2`.

```
def estimate_Dagum_fit_for_c(data, c, var, wgt, ecdf, x, y):
  shift_data = data.copy()
  shift_data[var] = shift_data[var] - c
  shift_data = shift_data[shift_data[var] > 0]
  a, b, p = estimate_Dagum_unshift(shift_data, var, wgt)
  def F(x):
    return cdf_Dagum(x, [a, b, p, c])
  temp = {
    "fit": kolmogorov_smirnov(F, ecdf=ecdf, x=x, y=y),
    "parameters": [a, b, p, c]}
  return temp
```

Now we code the main estimation for Dagum distribution. Unlike with `GB2`, we can ask Python to numerically minimize the Kolmogorov-Smirnov value for us. This code uses Brent's method.

```
def estimate_Dagum(data, var, wgt=None, *, ecdf=None, x=None, y=None):
  validate_var_wgt(data, var, wgt)
  if isinstance(ecdf, type(None)):
```

```
    ecdf = make_ecdf(data, var, wgt)
    x = var
    if wgt:
      y = wgt
    else:
      if var == "y":
        y = "y1"
      else:
        y = "y"
  def check_c(c):
    return estimate_Dagum_fit_for_c(data, c, var, wgt, ecdf, x, y)["fit"]
  sol = min_s(check_c, bracket=[-20000,-5000], options={"xtol":1.4e-5})
```

I can't figure out a good/easy way to get the parameters without another function call. Oh well, not a big deal.

```
  return estimate_Dagum_fit_for_c(data, sol.x, var, wgt, ecdf, x, y)
```

And add the functions to the appropriate dictionaries.

```
distribution["Dagum"] = cdf_Dagum
density["Dagum"]      = density_Dagum
likelihood["Dagum"]   = L_Dagum
estimator["Dagum"]    = estimate_Dagum
```

Burr Burr (Singh-Maddala) distribution. Parameters:

- $\alpha > 0$: shape.
- $\beta > 0$: scale.
- $q > 0$: shape.
- c: shift (for shifted Burr)

The cumulative distribution is

```
def cdf_Burr(x, params):
  a, b, q, c = params
  x = x.astype(np.float64)
  mask = x <= c
  x[mask] = 0
  frac = exp(a * (log(x[~mask] - c) - log(b)))
  x[~mask] = 1 - (1 + frac) ** (-q)
  return x
```

Density is

$$y = \frac{\alpha q (x-c)^{\alpha-1}}{\beta^{\alpha}\left(1+\left(\frac{x-c}{\beta}\right)^{\alpha}\right)^{q+1}}$$

Code:

```
def density_Burr(x, params):
```

```
  a, b, q, c = params
  if x <= c:
    return 0
  else:
    num = log(a) + log(q) + (a-1) * log(x-c)
    denom = a * log(b) + (1+q) * log(1 + ((x-c)/b)**a)
    return exp(num - denom)
```

Likelihood is

$$L = n\log\alpha + n\log q + (\alpha-1)\sum_{i=1}^{n} w_i \log(x_i - c) \\ - n\alpha\log\beta - (1+q)\sum_{i=1}^{n} w_i \log\left(1 + \left(\frac{x_i - c}{\beta}\right)^{\alpha}\right)$$

```
def L_Burr(data, params, var, wgt=None):
  validate_var_wgt(data, var, wgt)
  a, b, q, c = params
  if wgt:
    n = data[wgt].sum()
    term1 = n * log(a)
    term2 = n * log(q)
    term3 = (a-1) * (data[wgt] * log(data[var] - c)).sum()
    term4 = -n * a * log(b)
    term5 = -(q+1) * (data[wgt] * log(1 + ((data[var] - c)/b)**a)).sum()
  else:
    n = len(data)
    term1 = n * log(a)
    term2 = n * log(q)
    term3 = (a-1) * log(data[var] - c).sum()
    term4 = -n * a * log(b)
    term5 = -(q+1) * log(1 + ((data[var] - c)/b)**a).sum()
  return term1 + term2 + term3 + term4 + term5
```

Unshifted version:

```
def L_Burr_unshift(data, params, var, wgt=None):
  a, b, q = params
  return L_Burr(data, [a, b, q, 0], var, wgt)
```

Again, use Fisk moment estimators.

```
def estimate_Burr_initial(data, var, wgt):
  a, b = fisk_moments(data, var, wgt)
  return [a, b, 1]
```

Same thing as `Dagum`. The next functions are almost exactly identical to `Burr`.

```
def estimate_Burr_unshift(data, var, wgt):
```

```
  guess = estimate_Burr_initial(data, var, wgt)
  def neg_L(params):
    return -L_Burr_unshift(data, params, var, wgt)
  sol = min(neg_L, x0=guess, method="Nelder-Mead")
  return sol.x
```

Now code `fit_for_c`.

```
def estimate_Burr_fit_for_c(data, c, var, wgt, ecdf, x, y):
  shift_data = data.copy()
  shift_data[var] = shift_data[var] - c
  shift_data = shift_data[shift_data[var] > 0]
  a, b, q = estimate_Burr_unshift(shift_data, var, wgt)
  def F(x):
    return cdf_Burr(x, [a, b, q, c])
  temp = {
    "fit": kolmogorov_smirnov(F, ecdf=ecdf, x=x, y=y),
    "parameters": [a, b, q, c]}
  return temp
```

Main estimation function.

```
def estimate_Burr(data, var, wgt=None, *, ecdf=None, x=None, y=None):
  validate_var_wgt(data, var, wgt)
  if isinstance(ecdf, type(None)):
    ecdf = make_ecdf(data, var, wgt)
    x = var
    if wgt:
      y = wgt
    else:
      if var == "y":
        y = "y1"
      else:
        y = "y"
  def check_c(c):
    return estimate_Burr_fit_for_c(data, c, var, wgt, ecdf, x, y)["fit"]
  sol = min_s(check_c, bracket=[-20000,-5000], options={"xtol":1.4e-5})
  return estimate_Burr_fit_for_c(data, sol.x, var, wgt, ecdf, x, y)
```

And add the functions to the correct dictionaries.

```
distribution["Burr"] = cdf_Burr
density["Burr"]      = density_Burr
likelihood["Burr"]   = L_Burr
estimator["Burr"]    = estimate_Burr
```

And Burr is finished!

Fisk Fisk distribution. Parameters:

- $\alpha > 0$: shape.

- $\beta > 0$: scale.
- c: shift (for shifted Fisk)

The cumulative distribution is

$$y = \frac{1}{1 + \left(\frac{x-c}{\beta}\right)^{-\alpha}}$$

```
def cdf_Fisk(x, params):
  a, b, c = params
  x = x.astype(np.float64)
  mask = x <= c
  x[mask] = 0
  frac = exp(-a * (log(x[~mask] - c) - log(b)))
  x[~mask] = 1 / (1 + frac)
  return x
```

The density is

$$y = \frac{\alpha(x-c)^{\alpha-1}}{\beta^{\alpha}\left(1 + \left(\frac{x-c}{\beta}\right)^{\alpha}\right)^{2}}$$

Code:

```
def density_Fisk(x, params):
  a, b, c = params
  if x <= c:
    return 0
  else:
    num   = log(a) + (a-1) * log(x-c)
    denom = a * log(b) + 2 * log(1 + ((x-c)/b)**a)
    return exp(num - denom)
```

Likelihood is

$$L = n \log \alpha + (\alpha - 1) \sum_{i=1}^{n} w_i \log(x_i - c) - n\alpha \log \beta - 2 \sum_{i=1}^{n} w_i \log\left(1 + \left(\frac{x_i - c}{\beta}\right)^{\alpha}\right)$$

Code:

```
def L_Fisk(data, params, var, wgt=None):
  validate_var_wgt(data, var, wgt)
  a, b, c = params
  if wgt:
    n = data[wgt].sum()
    term1 = n * log(a)
    term2 = (a-1) * (data[wgt] * log(data[var] - c)).sum()
    term3 = -n * a * log(b)
    term4 = -2 * (data[wgt] * log(1 + ((data[var]-c)/b)**a)).sum()
```

```
  else:
    n = len(data)
    term1 = n * log(a)
    term2 = (a-1) * log(data[var] - c).sum()
    term3 = -n * a * log(b)
    term4 = -2 * log(1 + ((data[var]-c)/b)**a).sum()
  return term1 + term2 + term3 + term4
def L_Fisk_unshift(data, params, var, wgt=None):
  a, b = params
  return L_Fisk(data, [a, b, 0], var, wgt)
```

Use Fisk moment estimators to make initial guess for Fisk distribution. :)

```
estimate_Fisk_initial = fisk_moments
```

Functions are the same as previously.

```
def estimate_Fisk_unshift(data, var, wgt):
  guess = estimate_Fisk_initial(data, var, wgt)
  def neg_L(params):
    return -L_Fisk_unshift(data, params, var, wgt)
  sol = min(neg_L, x0=guess, method="Nelder-Mead")
  return sol.x
```

The `fit_for_c` function.

```
def estimate_Fisk_fit_for_c(data, c, var, wgt, ecdf, x, y):
  shift_data = data.copy()
  shift_data[var] = shift_data[var] - c
  shift_data = shift_data[shift_data[var] > 0]
  a, b = estimate_Fisk_unshift(shift_data, var, wgt)
  def F(x):
    return cdf_Fisk(x, [a, b, c])
  temp = {
    "fit": kolmogorov_smirnov(F, ecdf=ecdf, x=x, y=y),
    "parameters": [a, b, c]}
  return temp
```

And main estimation function.

```
def estimate_Fisk(data, var, wgt=None, *, ecdf=None, x=None, y=None):
  validate_var_wgt(data, var, wgt)
  if isinstance(ecdf, type(None)):
    ecdf = make_ecdf(data, var, wgt)
    x = var
    if wgt:
      y = wgt
    else:
      if var == "y":
        y = "y1"
```

```
      else:
        y = "y"
  def check_c(c):
    return estimate_Fisk_fit_for_c(data, c, var, wgt, ecdf, x, y)["fit"]
  sol = min_s(check_c, bracket=[data[var].min() - 2000, data[var].min()],
    options={"xtol":1.4e-5})
  return estimate_Fisk_fit_for_c(data, sol.x, var, wgt, ecdf, x, y)
```

Add functions to the dictionaries.

```
distribution["Fisk"] = cdf_Fisk
density["Fisk"]      = density_Fisk
likelihood["Fisk"]   = L_Fisk
estimator["Fisk"]    = estimate_Fisk
```

Done with the GB2 family of models!

InvG Inverse-Gamma distribution. Parameters:

- $\alpha > 0$: shape.
- $\beta > 0$: scale.
- c: shift. (For shifted inverse gamma)

The cumulative distribution is

$$y = Q\left(\alpha, \frac{\beta}{x - c}\right),$$

where Q is the upper incomplete gamma function.

```
def cdf_InvG(x, params):
  a, b, c = params
  x = x.astype(np.float64)
  mask = x <= c
  x[mask] = 0
  frac = exp(log(b) - log(x[~mask] - c))
  x[~mask] = Q(a, frac)
  return x
```

The density is

$$y = \frac{\beta^{\alpha}}{\Gamma(\alpha)} \frac{e^{-\frac{\beta}{x-c}}}{(x - c)^{1+\alpha}}$$

This model is much simpler than previous distributions.

```
def density_InvG(x, params):
  a, b, c = params
```

Similar to `GB2`, we manually check if we need to approximate $\log \Gamma(\alpha)$.

```
  if a < 10:
    log_G_a = log_G(a)
```

```
  else:
    log_G_a = log_G_approx(a)
  if x <= c:
    return 0
  else:
    num = a * log(b) - b / (x-c)
    denom = log_G_a + (1+a) * log(x-c)
    return exp(num - denom)
```

Likelihood:

$$L = n\alpha \log \beta - n \log \Gamma(\alpha) - \beta \sum_{i=1}^{n} \frac{w_i}{x_i - c} - (1 + \alpha) \sum_{i=1}^{n} w_i \log(x_i - c)$$

```
def L_InvG(data, params, var, wgt=None):
  validate_var_wgt(data, var, wgt)
  a, b, c = params
```

Again manually check if we need to approximate $\log \Gamma(\alpha)$.

```
  if a < 10:
    log_G_a = log_G(a)
  else:
    log_G_a = log_G_approx(a)
  if wgt:
    n = data[wgt].sum()
    term1 = n * a * log(b)
    term2 = -n * log_G_a
    term3 = -b * (data[wgt] / (data[var] - c)).sum()
    term4 = -(1+a) * (data[wgt] * log(data[var] - c)).sum()
  else:
    n = len(data)
    term1 = n * a * log(b)
    term2 = -n * log_G_a
    term3 = -b * (1 / (data.income - c)).sum()
    term4 = -(1+a) * log(data.income - c).sum()
  return term1 + term2 + term3 + term4
def L_InvG_unshift(data, params, var, wgt=None):
  validate_var_wgt(data, var, wgt)
  a, b = params
  return L_InvG(data, [a, b, 0], var, wgt)
```

We can find the MLE in more-or-less closed form. Sort of.

$$\frac{dL}{d\alpha} = n \log \beta - n\psi(\alpha) - \sum_{i=1}^{n} w_i \log(x_i - c) = 0$$

$$\frac{dL}{d\beta} = n\frac{\alpha}{\beta} - \sum_{i=1}^{n} \frac{w_i}{x_i - c} = 0$$

$$\beta = \frac{\alpha}{\frac{1}{n}\sum_{i=1}^{n}\frac{w_i}{x_i - c}}$$

$$\log\alpha - \psi(\alpha) - \log\left(\frac{1}{n}\sum_{i=1}^{n}\frac{w_i}{x_i - c}\right) - \frac{1}{n}\sum_{i=1}^{n} w_i \log(x_i - c) = 0$$

If we know c, the last equation gives us α, and the second-to-last equation gives us β in terms of α. So we have to numerically find the positive real root of $\log x - \psi(x) = C$. We define a dummy function called `_numeric` inside the `unshift` estimator, and we solve it numerically. Then we find β as above.

```
def estimate_InvG_unshift(data, var, wgt):
  if wgt:
    n = data[wgt].sum()
    recip_data = (1/n) * (data[wgt] / data[var]).sum()
    log_data = (1/n) * (data[wgt] * log(data[var])).sum()
  else:
    n = len(data)
    recip_data = (1/n) * (1 / data[var]).sum()
    log_data = (1/n) * log(data[var]).sum()
  def estimate_InvG_numeric(x):
    return log(x) - psi(x) - log(recip_data) - log_data
```

Getting the bounds on α right for the numerical solver can be a little tricky. In most cases and certainly for all the final estimates, α will be at most 5 or 10, but occasionally, we want to find α and β where α ends up being very large. So we dynamically find bounds for a good `bracket` for α. We store the current value of the numberical expression in `temp` because then we can handle the case when (if) we (miraculously) stumble on the root when we're searching for good bounds.

```
  temp = estimate_InvG_numeric(10)
  if temp < 0:
    temp1 = estimate_InvG_numeric(0.01)
    a = root(estimate_InvG_numeric, bracket=[0.01, 10],
             fprime = lambda x: (1/x) - psi1(x)).root
  elif temp == 0:
    a = 10
  else:
    prev_bound = 10
    curr_bound = 30
    temp = estimate_InvG_numeric(curr_bound)
    while temp > 0:
      prev_bound = curr_bound
      curr_bound = curr_bound + 20
      temp = estimate_InvG_numeric(curr_bound)
    if temp == 0:
      a = curr_bound
```

```
    else:
      a = root(estimate_InvG_numeric, bracket=[prev_bound,curr_bound],
                fprime = lambda x: (1/x) - psi1(x)).root
  b = a / recip_data
  return [a, b]
```

Now the `fit_for_c` function.

```
def estimate_InvG_fit_for_c(data, c, var, wgt, ecdf, x, y):
  shift_data = data.copy()
  shift_data[var] = shift_data[var] - c
  shift_data = shift_data[shift_data[var] > 0]
  a, b = estimate_InvG_unshift(shift_data, var, wgt)
  def F(x):
    return cdf_InvG(x, [a, b, c])
  temp = {
    "fit": kolmogorov_smirnov(F, ecdf=ecdf, x=x, y=y),
    "parameters": [a, b, c]}
  return temp
```

And finally the unshifted estimator

```
def estimate_InvG(data, var, wgt=None, *, ecdf=None, x=None, y=None):
  validate_var_wgt(data, var, wgt)
  if isinstance(ecdf, type(None)):
    ecdf = make_ecdf(data, var, wgt)
    x = var
    if wgt:
      y = wgt
    else:
      if var == "y":
        y = "y1"
      else:
        y = "y"
  def check_c(c):
    return estimate_InvG_fit_for_c(data, c, var, wgt, ecdf, x, y)["fit"]
  sol = min_s(check_c, method="bounded", bounds=[-23000,-1000])
  return estimate_InvG_fit_for_c(data, sol.x, var, wgt, ecdf, x, y)
```

And add to the dictionaries.

```
distribution["InvG"] = cdf_InvG
density["InvG"]      = density_InvG
likelihood["InvG"]   = L_InvG
estimator["InvG"]    = estimate_InvG
```

Davis Davis distribution. Parameters:

- $\alpha > 0$: shape
- $\beta > 0$: scale

- c: shift

This probability distribution hasn't gotten as much attention recently as others. It has similar appearance to the inverse-gamma distribution except that it puts more mass in the tail. The probability density is

$$f(x) = \frac{\beta^\alpha}{\Gamma(\alpha)\zeta(\alpha)}\left[\frac{1}{\left(e^{\frac{\beta}{x-c}}-1\right)(x-c)^{1+\alpha}}\right]$$

We can't write the cumulative distribution neatly, but we can express it as an infinite sum in terms of the upper incomplete gamma function.

$$\begin{aligned}
F(x) &= \frac{\beta^\alpha}{\Gamma(\alpha)\,\zeta(\alpha)}\int_c^x \frac{dt}{\left(e^{\frac{\beta}{t-c}}-1\right)(t-c)^{1+\alpha}} \\
&= \frac{\beta^\alpha}{\Gamma(\alpha)\,\zeta(\alpha)}\int_\infty^{\frac{\beta}{x-c}} \frac{u^{1+\alpha}}{\beta^{1+\alpha}(e^u-1)}\left(-\frac{\beta}{u^2}\right)du && u = \frac{\beta}{s-c} \\
&= \frac{1}{\Gamma(\alpha)\,\zeta(\alpha)}\int_{\frac{\beta}{x-c}}^\infty \frac{u^{\alpha-1}}{e^u-1}\,du \\
&= \frac{1}{\Gamma(\alpha)\,\zeta(\alpha)}\int_{\frac{\beta}{x-c}}^\infty \frac{u^{\alpha-1}e^{-u}}{1-e^{-u}}\,du.
\end{aligned}$$

At this point, we can rewrite the denominator as a geometric series with ratio e^{-u} (since $0 < e^{-u} < 1$). If we write the lower limit as p and consider just the integral, we have

$$\begin{aligned}
\int_p^\infty \frac{u^{\alpha-1}e^{-u}}{1-e^{-u}}\,du &= \int_p^\infty u^{\alpha-1}e^{-u}\sum_{k=0}^\infty e^{-ku}\,du \\
&= \int_p^\infty u^{\alpha-1}\sum_{k=1}^\infty e^{-ku}\,du \\
&= \sum_{k=1}^\infty \int_p^\infty u^{\alpha-1}e^{-ku}\,du \\
&= \sum_{k=1}^\infty \int_{kp}^\infty \left(\frac{v}{k}\right)^{\alpha-1} e^{-v}\,\frac{dv}{k} && v = ku \\
&= \sum_{k=1}^\infty \frac{1}{k^\alpha}\int_{kp}^\infty v^{\alpha-1}e^{-v}\,dv.
\end{aligned}$$

It follows that

$$F(x) = \frac{1}{\Gamma(\alpha)\,\zeta(\alpha)}\sum_{k=1}^\infty \frac{1}{k^\alpha}\Gamma\left(\alpha, \frac{k\beta}{x-c}\right) = \frac{1}{\zeta(\alpha)}\sum_{k=1}^\infty \frac{1}{k^\alpha}Q\left(\alpha, \frac{k\beta}{x-c}\right),$$

where Q is the regularized upper incomplete gamma function. What a fun distribution.

To code it, we start by writing a function to calculate the sum. We expect that the input `x` will have the form $\beta/(x_i - c)$, so we only need to provide α as an argument. (Note: we use `np.max` because `x` could be an array.)

```
def cdf_Davis_sum(x, alpha):
  k = 1
  temp = Q(alpha, x)
  var = temp
  while np.max(np.abs(temp)) > 0.000001:
    k = k + 1
    temp = exp(-alpha * log(k) + Q(alpha, exp(log(k) + log(x))))
    var = var + temp
  return var
```

And calculate the actual cumulative distribution value.

```
def cdf_Davis(x, params):
  a, b, c = params
  x = x.astype(float)
  mask = (x <= c)
  x[mask] = 0
  x[~mask] = cdf_Davis_sum(b / (np.array(x[~mask]) - c), a)
  return x / zeta(a)
```

Code for density.

```
def density_Davis(x, params):
  a, b, c = params
  if x <= c:
    return 0
  else:
    num = a * log(b)
    denom = log(G(a)) + log(zeta(a)) + \
            log(exp(exp(log(b) - log(x - c))) - 1) + \
            (1 + a) * log(x - c)
    return exp(num - denom)
```

The likelihood is

$$L = n\alpha \log \beta - n \log \Gamma(\alpha) - n \log \zeta(\alpha) - \sum_{i=1}^{n} w_i \log \left(e^{\frac{\beta}{x_i - c}} - 1 \right) - (1+\alpha) \sum_{i=1}^{n} w_i \log(x_i - c)$$

Code.

```
def L_Davis(data, params, var, wgt=None):
  validate_var_wgt(data, var, wgt)
  a, b, c = params
  if isinstance(wgt, type(None)):  # if no weight provided
    n = len(data)
    term4 = -log(exp(exp(log(b) - log(data[var] - c))) - 1).sum()
```

```
      term5 = -(1 + a) * log(data[var] - c).sum()
    else:                             # if weight provided
      n = data[wgt].sum()
      term4 = -(data[wgt] *
        log(exp(exp(log(b) - log(data[var] - c))) - 1)).sum()
      term5 = -(1 + a) * (data[wgt] * log(data[var] - c)).sum()
```

The first three terms don't depend on the weights, only on the (weight or length) total `n`, so we can code them after the conditional block, which makes the code cleaner. (Even if the terms are out of order now.)

```
    term1 =  n * a * log(b)
    term2 = -n * log(G(a))
    term3 = -n * log(zeta(a))
    return term1 + term2 + term3 + term4 + term5
```

Unshifted version is

```
  def L_Davis_unshift(data, params, var, wgt=None):
    return L_Davis(data, [*params, 0], var, wgt)
```

Differentiating the likelihood function and setting equal to 0 gives us

$$\frac{dL}{d\alpha} = n \log \beta - n\psi(\alpha) - n\frac{\zeta'(\alpha)}{\zeta(\alpha)} - \sum_{i=1}^{n} w_i \log(x_i - c) = 0$$

$$\frac{dL}{d\beta} = \frac{n\alpha}{\beta} - \sum_{i=1}^{n} \left(\frac{w_i}{x_i - c}\right) \frac{e^{\frac{\beta}{x_i - c}}}{e^{\frac{\beta}{x_i - c}} - 1} = 0$$

We would like to solve these equations numerically. First we need to code ζ'. We have

$$\zeta'(z) = -\sum_{k=2}^{\infty} \frac{\log k}{k^z}$$

Code:

```
  def zeta_prime(z):
    k = 2
    temp = 1
    val = -log(k) / k ** z
    while np.abs(temp) > 0.000001:
      k = k + 1
      temp = -exp(log(log(k)) - z * log(k))
      val = val + temp
    return val
```

Simultaneously finding roots α and β for $dL/d\alpha$ and $dL/d\beta$ doesn't work because the numerics don't converge. However, we can still maximize the likelihood function. Notice that it is straightforward to find α that maximizes likelihood conditional on β and

c, and we write the `_a_from_b` function to do this.

```
def estimate_Davis_a_from_b(data, b, var, wgt=None):
  if isinstance(wgt, type(None)):
    n = len(data)
    wgt = pd.Series(1, index=data.index)
  else:
    n = data[wgt].sum()
    wgt = data[wgt]
```

The function `dL_da` is what we want to solve. Because it doesn't depend on α, we store the $\log \beta$ and $\log(x_i - c)$ terms in `temp` so we don't have to calculate their value every time the numerical solver evaluates `dL_da`. We've also divided everything by `n` to make things simpler.

```
  temp = log(b) - (1/n) * (wgt * log(data[var])).sum()
  def dL_da(a):
    return temp - psi(a) - zeta_prime(a) / zeta(a)
```

The expression

$$-\psi(\alpha) - \frac{\zeta'(\alpha)}{\zeta(\alpha)}$$

is monotonic decreasing in α, and that makes it very easy to find a bracket for `dL_da`. This is the same way we found a bracket for α when we estimated the `InvG` model. We start with `lower_bound` $= 2$, and we repeatedly scale `lower_bound` up or down until we find `lower_bound` and `upper_bound` where `dL_da` has different signs.

```
  lower_bound = 2
  if dL_da(lower_bound) == 0:         # if lower_bound is a root
    return bound1
  elif dL_da(lower_bound) > 0:        # if lower_bound is too small
    upper_bound = 1.2 * lower_bound
    while dL_da(upper_bound) > 0:
      lower_bound = upper_bound
      upper_bound = 1.2 * upper_bound
```

If `lower_bound` is too big, we decrease it. We take roots instead of multiplying or dividing because we don't want to cross the asymptote of ζ at $\alpha = 1$.

```
  elif dL_da(lower_bound) < 0:        # if lower_bound is too big
    while dL_da(lower_bound) < 0:
      upper_bound = lower_bound
      lower_bound = lower_bound ** 0.8
  else:
    raise RuntimeError("Something weird happened")
  return opt.root_scalar(dL_da, x0=2,
    bracket=[lower_bound,upper_bound]).root
```

Now we do the estimation. Same thing as usual. We have the `unshift` function to calculate α and β given c, the `fit_for_c` function to find the Kolmogorov-Smirnov statistic, and the final estimation function to do the overall estimation. As initial guesses, we use α and β from the inverse-gamma model.

```
def estimate_Davis_unshift(data, var, wgt):
  guess = estimate_InvG_unshift(data, var, wgt)
```

Because we know how to find α in terms of β, we can numerically maximize the likelihood as a function of a single argument β. This function is nicely behaved.

```
  def neg_L_with_a_in_terms_of_b(b):
    a = estimate_Davis_a_from_b(data, b, var, wgt)
    return -L_Davis_unshift(data, [a,b], var, wgt)
  sol = opt.minimize_scalar(neg_L_with_a_in_terms_of_b,
    bracket=[guess[1] - 20000, guess[1]])
  return [estimate_Davis_a_from_b(data, sol.x, var, wgt), sol.x]
```

The `fit_for_c` function.

```
def estimate_Davis_fit_for_c(data, c, var, wgt, ecdf, x, y):
  shift_data = data.copy()
  shift_data[var] = data[var] - c
  shift_data = shift_data[shift_data[var] > 0]
  a, b = estimate_Davis_unshift(shift_data, var, wgt)
  def F(x):
    return cdf_Davis(x, [a, b, c])
  temp = {
    "fit": kolmogorov_smirnov(F, ecdf=ecdf, x=x, y=y),
    "parameters": [a, b, c]}
  return temp
```

And the main estimation function.

```
def estimate_Davis(data, var, wgt=None, *, ecdf=None, x=None, y=None):
  validate_var_wgt(data, var, wgt)
  if isinstance(ecdf, type(None)):
    ecdf = make_ecdf(data, var, wgt)
    x = var
    if wgt:
      y = wgt
    else:
      if var == "y":
        y = "y1"
      else:
        y = "y"
  def check_c(c):
    return estimate_Davis_fit_for_c(data, c, var, wgt, ecdf, x, y)["fit"]
  sol = opt.minimize_scalar(check_c,
    bracket=[-20000,-7000], options={"xtol":1.4e-5})
  return estimate_Davis_fit_for_c(data, sol.x, var, wgt, ecdf, x, y)
```

And add to the dictionaries.

```
    distribution["Davis"] = cdf_Davis
    density["Davis"]      = density_Davis
    likelihood["Davis"]   = L_Davis
    estimator["Davis"]    = estimate_Davis
```

CS_InvG We're not using this model, but I've kept it for historical reasons. Constant-shift inverse-gamma distribution. Parameters:

- $\alpha > 0$: shape parameter.
- β: scale parameter.

The distribution also has a constant ϕ. This is the same as shifted inverse-gamma distribution except that we set $c = \phi\beta$, so the shift is proportional to the scale. For our purposes, $\phi \approx -0.13$. Cumulative distribution is

$$y = Q\left(\alpha, \frac{\beta}{x - \phi\beta}\right).$$

```
def cdf_CS_InvG(x, phi, params):
  a, b = params
  shift = phi * b
  x = x.astype(np.float64)
  mask = x <= shift
  x[mask] = 0
  frac = exp(log(b) - log(x[~mask] - shift))
  x[~mask] = Q(a, frac)
  return x
```

Density:

$$y = \frac{\beta^{\alpha}}{\Gamma(\alpha)} \frac{e^{-\frac{\beta}{x - \phi\beta}}}{(x - \phi\beta)^{1+\alpha}}$$

```
def density_CS_InvG(x, phi, params):
  a, b = params
  shift = phi * b
  if x <= shift:
    return 0
  else:
    num = a * log(b) - b / (x - shift)
    denom = log_G(a) + (1 + alpha) * log(x - shift)
    return exp(num - denom)
```

Likelihood:

$$L = n\alpha \log\beta - n \log\Gamma(\alpha) - \beta \sum_{i=1}^{n} \frac{w_i}{x_i - \phi\beta} - (1 + \alpha) \sum_{i=1}^{n} w_i \log(x_i - \phi\beta)$$

```
def L_CS_InvG(data, phi, params, var, wgt=None):
```

```
validate_var_wgt(data, var, wgt)
a, b = params
shift = phi * b
if wgt:
  n = data[wgt].sum()
  term1 = n * a * log(b)
  term2 = -n * log_G(a)
  term3 = -b * (data[wgt] / (data[var] - shift)).sum()
  term4 = -(1+a) * (data[wgt] * log(data[var] - shift)).sum()
else:
  n = len(data)
  term1 = n * a * log(b)
  term2 = -n * log_G(a)
  term3 = -b * (1 / (data[var] - shift)).sum()
  term4 = -(1+a) * log(data[var] - shift).sum()
return term1 + term2 + term3 + term4
```

We can't solve the MLE for the constant-shift inverse-gamma distribution in closed form, but we can get close. Unfortunately, it's a bit more complicated than with the separate c parameter.

$$\begin{aligned}\frac{dL}{d\alpha} &= n\log\beta - n\psi(\alpha) - \sum_{i=1}^{n} w_i \log(x_i - \phi\beta) \\ &= -n\psi(\alpha) + \sum_{i=1}^{n} w_i \log\left(\frac{\beta}{x_i - \phi\beta}\right) = 0\end{aligned}$$

$$\begin{aligned}\frac{dL}{d\beta} &= n\frac{\alpha}{\beta} - \sum_{i=1}^{n} w_i\left(\frac{1}{x_i - \phi\beta} + \frac{\phi\beta}{(x_i - \phi\beta)^2}\right) - (1+\alpha)\sum_{i=1}^{n}\frac{-\phi w_i}{x_i - \phi\beta} \\ &= n\frac{\alpha}{\beta} - \sum_{i=1}^{n}\frac{w_i x_i}{(x_i - \phi\beta)^2} + \phi(1+\alpha)\sum_{i=1}^{n}\frac{w_i}{x_i - \phi\beta} = 0\end{aligned}$$

Looking at the $d\beta$ equation, we see that

$$\begin{aligned}n\frac{\alpha}{\beta} + \phi(1+\alpha)\sum_{i=1}^{n}\frac{w_i}{x_i - \phi\beta} &= \sum_{i=1}^{n}\frac{w_i x_i}{(x_i - \phi\beta)^2} \\ n\frac{\alpha}{\beta} + \phi\alpha\sum_{i=1}^{n}\frac{w_i}{x_i - \phi\beta} + \phi\sum_{i=1}^{n}\frac{w_i}{x_i - \phi\beta} &= \sum_{i=1}^{n}\frac{w_i x_i}{(x_i - \phi\beta)^2} \\ \frac{\alpha}{\beta} + \frac{\phi\alpha}{n}\sum_{i=1}^{n}\frac{w_i}{x_i - \phi\beta} &= \frac{1}{n}\sum_{i=1}^{n}\frac{w_i x_i}{(x_i - \phi\beta)^2} - \frac{\phi}{n}\sum_{i=1}^{n}\frac{w_i}{x_i - \phi\beta} \\ \alpha\left[\frac{1}{\beta} + \frac{\phi}{n}\sum_{i=1}^{n}\frac{w_i}{x_i - \phi\beta}\right] &= \frac{1-\phi}{n}\sum_{i=1}^{n}\frac{w_i x_i}{(x_i - \phi\beta)^2} + \frac{\phi^2\beta}{n}\sum_{i=1}^{n}\frac{w_i}{(x_i - \phi\beta)^2}\end{aligned}$$

Altogether we get

$$\psi(\alpha) = \frac{1}{n}\sum_{i=1}^{n} w_i \log\left(\frac{\beta}{x_i - \phi\beta}\right)$$

$$\alpha = \frac{\dfrac{1-\phi}{n}\displaystyle\sum_{i=1}^{n}\frac{w_i x_i}{(x_i-\phi\beta)^2} + \dfrac{\phi^2\beta}{n}\displaystyle\sum_{i=1}^{n}\frac{w_i}{(x_i-\phi\beta)^2}}{\dfrac{1}{\beta} + \dfrac{\phi}{n}\displaystyle\sum_{i=1}^{n}\frac{w_i}{x_i-\phi\beta}}$$

To do the estimation, we can solve this system of equations numerically for β. We begin with the two expressions of interest, which we call `alpha1` and `alpha2`. We assume that the data has not been modified at all, so we should translate it and remove all negative (after shifting) incomes. We store the new dataset in `shift_data`.

```
def estimate_CS_InvG_alpha1(data, phi, beta, var, wgt):
  shift = phi * beta
  shift_data = data.copy()
  shift_data = shift_data[shift_data[var] > shift]
  shift_data[var] = shift_data[var] - shift
  if wgt:
    n = data[wgt].sum()
    sum = (shift_data[wgt] * log(beta / shift_data[var])).sum()
  else:
    n = len(data)
    sum = log(beta / shift_data[var]).sum()
  return (1/n) * sum
```

Same thing for the second α expression.

```
def estimate_CS_InvG_alpha2(data, phi, beta, var, wgt):
  shift = phi * beta
  shift_data = data.copy()
  shift_data = shift_data[shift_data[var] > shift]
  shift_data[var] = shift_data[var] - shift
  if wgt:
    n = data[wgt].sum()
    num1 = (1-phi) / n * (shift_data[wgt] * (shift_data[var] + shift) /
      shift_data[var] ** 2).sum()
    num2 = (phi**2) * (beta/n) * (shift_data[wgt] /
      shift_data[var] ** 2).sum()
    denom = (1/beta) + (phi/n) * (shift_data[wgt] / shift_data[var]).sum()
  else:
    n = len(data)
    num1 = (1-phi) / n * ((shift_data[var] + shift) /
      shift_data[var] ** 2).sum()
    num2 = (phi**2) * (beta/n) * (1 / shift_data[var] ** 2).sum()
    denom = (1/beta) + (phi/n) * (1 / shift_data[var]).sum()
  return (num1 + num2) / denom
```

Because we don't have a separate shift parameter, we can perform the estimation in a single step rather than two (or more). The `temp_numeric` function will be the expression we want to numerically find the root of with respect to β. Consider the asymptotics of both expressions for α. (Assuming that total weight has been normalized to n.) As β gets large, the sum approaches

$$\begin{aligned}\frac{1}{n}\sum_{i=1}^{n} w_i \log\left(\frac{\beta}{x_i - \phi\beta}\right) &= \frac{1}{n}\sum_{i=1}^{n} w_i \log\left(\frac{1}{\frac{x_i}{\beta} - \phi}\right) \\ &\to \frac{1}{n}\sum_{i=1}^{n} w_i \log\left(\frac{1}{\phi}\right) = \log\left(\frac{1}{\beta}\right).\end{aligned}$$

For the big fraction, we have

$$\begin{aligned}&\frac{\dfrac{1-\phi}{n}\displaystyle\sum_{i=1}^{n}\frac{w_i x_i}{(x_i - \phi\beta)^2} + \dfrac{\phi^2\beta}{n}\displaystyle\sum_{i=1}^{n}\frac{w_i}{(x_i - \phi\beta)^2}}{\dfrac{1}{\beta} + \dfrac{\phi}{n}\displaystyle\sum_{i=1}^{n}\frac{w_i}{x_i - \phi\beta}} \\ &\qquad = \frac{\dfrac{1-\phi}{n}\displaystyle\sum_{i=1}^{n}\frac{w_i x_i}{\left(\frac{x_i}{\sqrt{\beta}} - \phi\sqrt{\beta}\right)^2} + \dfrac{\phi^2}{n}\displaystyle\sum_{i=1}^{n}\frac{w_i}{\left(\frac{x_i}{\beta} - \phi\right)^2}}{1 + \dfrac{\phi}{n}\displaystyle\sum_{i=1}^{n}\frac{w_i}{\frac{x_i}{\beta} - \phi}}\end{aligned}$$

The first term in the numerator approaches 0, and the second approaches 1. The denominator approaches 0, so the quotient blows up. It follows that the `temp_numeric` function in the estimator gets big negative for large β.

```
def estimate_CS_InvG(data, phi, var, wgt=None, *, ecdf=None, x=None, y=None):
  validate_var_wgt(data, var, wgt)
  if isinstance(ecdf, type(None)):
    ecdf = make_ecdf(data, var, wgt)
    x = var
    if wgt:
      y = wgt
    else:
      if var == "y":
        y = "y1"
      else:
        y = "y"
  def temp_numeric(b):
    return estimate_CS_InvG_alpha1(data, phi, b, var, wgt) - \
      psi(estimate_CS_InvG_alpha2(data, phi, b, var, wgt))
```

Around the root, the second term in `temp_numeric` grows faster than the first, but this doesn't hold for small β. There the function jumps around a lot and may give bad results, i.e. roots that we do not care about. Because the β value jumps around a lot, we calculate bounds dynamically. This is very similar to what we did to find α in the estimation for `InvG`. In the limit as β gets large, `temp_numeric` gets big negative, so our approach will be to start with a large value of β and successively decrease it until we get bounds that work. Then we find α accordingly.

```
right_bound = 200000
left_bound = 100000
temp = temp_numeric(left_bound)
if temp > 0:
  rtemp = temp_numeric(right_bound)
  while rtemp > 0:
    left_bound = right_bound
    right_bound = right_bound * 1.05
    rtemp = temp_numeric(right_bound)
  if rtemp == 0:
    beta = right_bound
  else:
    beta = root(temp_numeric, bracket=[left_bound,right_bound]).root
elif temp == 0:
  beta = left_bound
else:
  while temp < 0:
    right_bound = left_bound
    left_bound = left_bound * 0.98
    temp = temp_numeric(left_bound)
  if temp == 0:
    beta = left_bound
  else:
    beta = root(temp_numeric, bracket=[left_bound,right_bound]).root
alpha = estimate_CS_InvG_alpha2(data, phi, beta, var, wgt)
```

Create and return the dictionary. Unlike dictionaries from the other estimators, this one contains a key called `"phi"`.

```
def F(x):
  return cdf_CS_InvG(x, phi, [alpha, beta])
temp = {
  "fit": kolmogorov_smirnov(F, ecdf=ecdf, x=x, y=y),
  "phi": phi,
  "parameters": [alpha, beta]}
return temp
```

And add to dictionaries.

```
distribution["CS_InvG"] = cdf_CS_InvG
density["CS_InvG"]      = density_CS_InvG
likelihood["CS_InvG"]   = L_CS_InvG
estimator["CS_InvG"]    = estimate_CS_InvG
```

CSS_InvG Constant-shift-scale inverse-gamma distribution. Parameters:

- $\alpha > 0$: shape parameter

This distribution is the same as in inverse-gamma distribution except that we take values for β and c based on linear functions of α. Specifically, $c = \psi_0 + \psi_1 t + \psi_2\alpha$, and $c = \phi\beta$. From `check_constants.py`, we approximate values of the constants as follows:

- ϕ: -0.134
- ψ_0: 721,038
- ψ_1: -363
- ψ_2: -2252

See Table 7 for more information. For this set of functions, we will have several more arguments for the functions than usual. The cumulative distribution function is

$$y = Q\left(\alpha, \frac{1}{\phi}\frac{\psi_0 + \psi_1 t + \psi_2\alpha}{x - (\psi_0 + \psi_1 t + \psi_2\alpha)}\right)$$

Code:

```
def cdf_CSS_InvG(x, t, phi, psi, a):
  psi0, psi1, psi2 = psi
  b = (psi0 + psi1 * t + psi2 * a) / phi
  c = psi0 + psi1 * t + psi2 * a
  x = x.astype(np.float64)
  mask = x <= c
  x[mask] = 0
  frac = exp(log(b) - log(x[~mask] - c))
  x[~mask] = Q(a, frac)
  return x
```

The density is

$$y = \frac{(\psi_0 + \psi_1 t + \psi_2\alpha)^\alpha}{\Gamma(\alpha)} \frac{e^{-\frac{\psi_0 + \psi_1 t + \psi_2\alpha}{x - \phi(\psi_0 + \psi_1 t + \psi_2\alpha)}}}{(x - \phi(\psi_0 + \psi_1 t + \psi_2\alpha))^{1+\alpha}}$$

```
def density_CSS_InvG(x, t, phi, psi, a):
  psi0, psi1, psi2 = psi
  beta = (psi0 + psi1 * t + psi2 * a) / phi
  c = psi0 + psi1 * t + psi2 * a
  if x <= c:
    return 0
  else:
    return density_InvG(x, [a, beta, c])
```

Likelihood:

$$L = n\alpha\log(\psi_0 + \psi_1 t + \psi_2\alpha) - n\Gamma(\alpha) - (\psi_0 + \psi_1 t + \psi_2\alpha)\sum_{i=1}^{n} \frac{w_i}{x_i - \phi(\psi_0 + \psi_1 t + \psi_2\alpha)}$$

$$- (1+\alpha) \sum_{i=1}^{n} w_i \log[x_i - \phi(\psi_0 + \psi_1 t + \psi_2 \alpha)]$$

```
def L_CSS_InvG(data, t, phi, psi, a, var, wgt=None):
  validate_var_wgt(data, var, wgt)
  psi0, psi1, psi2 = psi
  beta = (psi0 + psi1 * t + psi2 * a) / phi
  c = psi0 + psi1 * t + psi2 * a
  return L_InvG(data, [a, beta, c], var, wgt)
```

Unfortunately, maximum likelihood does not work properly with the constant-shift-scale inverse-gamma distribution. So we minimize the Kolmogorov-Smirnov statistic instead. The `fit_for_a` function will calculate the Kolmogorov-Smirnov statistic.

```
def estimate_CSS_InvG_fit_for_a(t, phi, psi, a, ecdf, x, y):
  psi0, psi1, psi2 = psi
  beta = (psi0 + psi1 * t + psi2 * a) / phi
  c = psi0 + psi1 * t + psi2 * a
  def F(x):
    return cdf_InvG(x, [a, beta, c])
  temp = {
    "fit": kolmogorov_smirnov(F, ecdf=ecdf, x=x, y=y),
    "phi": phi,
    "psi": psi,
    "parameters": [a]}
  return temp
```

And the main estimation function calculates the minimum of the `fit_for_a` function. The `a0` argument is the starting value for α in the minimization. We will take `a0` to be the value of α from year `t` under the 3-parameter model.

```
def estimate_CSS_InvG(data, t, phi, psi, a0, var, wgt=None, *, \
                      ecdf=None, x=None, y=None):
  validate_var_wgt(data, var, wgt)
  if isinstance(ecdf, type(None)):
    ecdf = make_ecdf(data, var, wgt)
    x = var
    if wgt:
      y = wgt
    else:
      if var == "y":
        y = "y1"
      else:
        y = "y"
  def check_a(a):
    return estimate_CSS_InvG_fit_for_a(t, phi, psi, a, ecdf, x, y)["fit"]
  sol = min(check_a, x0=a0).x[0]
  return estimate_CSS_InvG_fit_for_a(t, phi, psi, sol, ecdf, x, y)
```

And add entries in the dictionaries.

```
    distribution["CSS_InvG"] = cdf_CSS_InvG
    density["CSS_InvG"]      = density_CSS_InvG
    likelihood["CSS_InvG"]   = L_CSS_InvG
    estimator["CSS_InvG"]    = estimate_CSS_InvG
```

CSS_InvG_prop Constant-shift-scale inverse-gamma distribution where c is proportional to α and time instead of a linear function of α and time. Parameters:

- $\alpha > 0$: shape parameter

This distribution is the same as in inverse-gamma distribution except that we fix β and c to be proportional to α. Specifically, $c_t = \alpha_t(\psi_0 + \psi_1 t) = \phi\beta_t$ (and equivalently $\beta = c/\phi$). From `check_constants.py`, we approximate values of the constants as follows:

- ϕ: -0.1337
- ψ_0: \$206,824
- ψ_1: $-$\$105.33/year

See also Table 7. For this set of estimation functions, we will have several more arguments than for the other models. The cumulative distribution function is

$$y = Q\left(\alpha, \frac{1}{\phi}\frac{\alpha(\psi_0 + \psi_1 t)}{x - \alpha(\psi_0 + \psi_1 t)}\right)$$

Code:

```
def cdf_CSS_InvG_prop(x, t, phi, psi, a):
  psi0, psi1 = psi
  b = a * (psi0 + psi1 * t) / phi
  c = a * (psi0 + psi1 * t)
  x = x.astype(np.float64)
  mask = (x <= c)
  x[mask] = 0
  frac = exp(log(b) - log(x[~mask] - c))
  x[~mask] = Q(a, frac)
  return x
```

The density is

$$y = \frac{\alpha^\alpha(\psi_0 + \psi_1 t)^\alpha}{\phi^\alpha\Gamma(\alpha)}\frac{e^{-\frac{1}{\phi}\frac{\alpha(\psi_0 + \psi_1 t)}{x - \alpha(\psi_0 + \psi_1 t)}}}{(x - \alpha(\psi_0 + \psi_1 t))^{1+\alpha}}$$

```
def density_CSS_InvG_prop(x, t, phi, psi, a):
  psi0, psi1 = psi
  beta = a * (psi0 + psi1 * t) / phi
  c = a * (psi0 + psi1 * t)
  if x <= c:
    return 0
  else:
    return density_InvG(x, [a, beta, c])
```

Likelihood:

$$L = n\alpha \log\left(\frac{\alpha(\psi_0 + \psi_1 t)}{\phi}\right) - n \log \Gamma(\alpha) - \left(\frac{\alpha(\psi_0 + \psi_1 t)}{\phi}\right) \sum_{i=1}^{n} \frac{w_i}{x - \alpha(\psi_0 + \psi_1 t)}$$
$$- (1+\alpha) \sum_{i=1}^{n} w_i \log(x_i - \alpha(\psi_0 + \psi_1 t))$$

```
def L_CSS_InvG_prop(data, t, phi, psi, a, var, wgt=None):
  validate_var_wgt(data, var, wgt)
  psi0, psi1 = psi
  beta = a * (psi0 + psi1 * t) / phi
  c = a * (psi0 + psi1 * t)
  return L_InvG(data, [a, beta, c], var, wgt)
```

Unfortunately, maximum likelihood does not work properly with the constant-shift-scale inverse-gamma distribution. So we minimize the Kolmogorov-Smirnov statistic instead. The `fit_for_a` function will calculate the Kolmogorov-Smirnov statistic.

```
def estimate_CSS_InvG_prop_fit_for_a(t, phi, psi, a, ecdf, x, y):
  if a <= 0:
    return {"fit": 1, "parameters": a}
  else:
    psi0, psi1 = psi
    beta = a * (psi0 + psi1 * t) / phi
    c = a * (psi0 + psi1 * t)
    def F(x):
      return cdf_InvG(x, [a, beta, c])
    temp = {
      "fit": kolmogorov_smirnov(F, ecdf=ecdf, x=x, y=y),
      "parameters": [a]}
    return temp
```

And the main estimation function calculates the minimum of the `fit_for_a` function. The `a0` argument is the starting value for α in the minimization. We will take `a0` to be the value of α from year `t` under the 3-parameter model.

```
def estimate_CSS_InvG_prop(data, t, phi, psi, a0, var, wgt=None, *, \
                      ecdf=None, x=None, y=None):
  validate_var_wgt(data, var, wgt)
  if isinstance(ecdf, type(None)):
    ecdf = make_ecdf(data, var, wgt)
    x = var
    if wgt:
      y = wgt
    else:
      if var == "y":
        y = "y1"
      else:
```

```
      y = "y"
  def check_a(a):
    return estimate_CSS_InvG_prop_fit_for_a(t, phi, psi, a, ecdf, x, y)["fit"]
  sol = min(check_a, x0=a0).x[0]
  return estimate_CSS_InvG_prop_fit_for_a(t, phi, psi, sol, ecdf, x, y)
```

And add entries in the dictionaries.

```
distribution["CSS_InvG_prop"] = cdf_CSS_InvG_prop
density["CSS_InvG_prop"]      = density_CSS_InvG_prop
likelihood["CSS_InvG_prop"]   = L_CSS_InvG_prop
estimator["CSS_InvG_prop"]    = estimate_CSS_InvG_prop
```

All inverse-gamma distributions done!

LogN_P_cut Pareto and log-normal distributions with cutoff. Parameters:

- μ: log mean
- $\sigma^2 > 0$: log variance
- $k > x_m + c$: cutoff (we need the cutoff to exceed the lower bound on the support of shifted Pareto)
- $x_m > 0$: Pareto scale parameter
- $\alpha > 0$ Pareto shape parameter
- c: shift parameter (for shifted distribution)

Cumulative distribution function:

$$y = \begin{cases} \Phi\left(\dfrac{\log(x-c)-\mu}{\sigma}\right) & \text{if } x < k \\ 1 - \left(\dfrac{x_m}{x-c}\right)^{\alpha} & \text{if } x \geq k \end{cases}$$

```
def cdf_LogN_P_cut(x, params):
  mu, sigma_sq, k, x_m, a, c = params
  x = x.astype(np.float64)
  mask_c = x <= c
  mask_k = x <= k
  mask_xm = x < x_m
  if k < c:
    raise RuntimeError("k was less than c; problem?")
```

The `LogN_P_cut` distribution is a bit different from the other models in that its domain gets partitioned into two separate pieces. So we will end up transforming two separate subarrays, one for data less than k and one for data greater than k.

```
  x[mask_c] = 0
  x[(~mask_c) & mask_k] = Phi(
    (log(x[(~mask_c) & mask_k] - c) - mu) / sqrt(sigma_sq))
  x[(~mask_k) & mask_xm] = 0
  x[(~mask_k) & (~mask_xm)] = \
    1 - (x_m / (x[(~mask_k) & (~mask_xm)] - c)) ** a
  return x
```

Density:

$$y = \begin{cases} \dfrac{1}{(x-c)\sigma\sqrt{2\pi}} e^{-(\log(x-c)-\mu)^2/2\sigma^2} & \text{if } x < k \\ \dfrac{\alpha x_m^\alpha}{(x-c)^{1+\alpha}} & \text{if } x \geq k \end{cases}$$

Code:

```
def density_LogN_P_cut(x, params):
  mu, sigma_sq, k, x_m, a, c = params
  if k < c:
    print("k was less than c; setting k=c")
    k = c
  if x <= c:
    return 0
  elif x > c and x < k:
    frac = 1/((x-c) * sqrt(2 * pi * sigma_sq))
    exponent = -(log(x-c) - mu)**2 / (2 * sigma_sq)
    return frac * exp(exponent)
  elif x >= k:
    if x <= x_m + c:
      return 0
    else:
      return exp(log(a) + a * log(x_m) - (1+a) * log(x-c))
  else:
    raise RuntimeError("Something weird happened")
```

Likelihood:

$$L = \sum_{i=1}^{n} w_i \psi_i,$$

where

$$\psi_i = \begin{cases} -\log(x_i - c) - \log(\sigma) - \dfrac{1}{2}\log(2\pi) - \dfrac{(\log(x_i - c) - \mu)^2}{2\sigma^2} & \text{if } x_i < k \\ \log\alpha + \alpha\log(x_m) - (1+\alpha)\log(x_i - c) & \text{if } x_i \geq k \end{cases}$$

Code:

```
def L_LogN_P_cut(data, params, var, wgt=None):
  validate_var_wgt(data, var, wgt)
  mu, sigma_sq, k, x_m, a, c = params
  data_down = data[data[var] < k]
  data_up = data[data[var] >= k]
  if wgt:
    n_down = data_below_k[wgt].sum()
    n_up = data_above_k[wgt].sum()
    down_terms = -(data_down[wgt] * log(data_down[var] - c)).sum() - \
```

```
      (n_down/2) * log(2 * sigma_sq * pi) - \
      (data_down[wgt] * (log(data_down[var] - c) - mu)**2).sum() * \
      (1 / (2*sigma_sq))
    up_terms = n_up * (log(a) + a * log(x_m)) + \
      (1+a) * (data_up[wgt] * log(data_up[var] - c)).sum()
  else:
    n_down = len(data_below_k)
    n_up = len(data_above_k)
    down_terms = -log(data_down[var] - c).sum() - \
      (n_down/2) * log(2 * sigma_sq * pi) - \
      ((log(data_down.income() - c) - mu)**2).sum() / (2*sigma_sq)
    up_terms = n_up * (log(a) + a * log(x_m)) + \
      (1+a) * log(data_up.income - c).sum()
  return down_terms + up_terms
```

Unshifted version:

```
def L_LogN_P_cut_unshift(data, params, var, wgt=None):
  mu, sigma_sq, k, x_m, a = params
  return L_LogN_P_cut(data, [mu, sigma_sq, k, x_m, a, 0], var, wgt)
```

We can solve for the MLE symbolically. (Or at least solve most of the way.) The algebra is a bit ugly though. The one restriction is that the density should integrate to 1. In other words,

$$\Phi\left(\frac{\log k - \mu}{\sigma}\right) + \left(\frac{x_m}{k}\right)^{\alpha} = 1,$$

where Φ is the cumulative distribution of a standard normal random variable. We can maximize the likelihood subject to this restriction. The restricted likelihood function/Lagrangian is

$$L = \sum_{i=1}^{n} w_i \psi_i + \lambda\left(1 - \Phi\left(\frac{\log k - \mu}{\sigma}\right) - \left(\frac{x_m}{k}\right)^{\alpha}\right)$$

Assume that k and c are constant, and let j be the index that divides the incomes between [less than k] and [greater than k]. If we let $n_{\downarrow}$ and $n^{\uparrow}$ be the sums of weights for incomes less than k and greater than k respectively, then differentiating gives us

$$\begin{aligned}
\frac{dL}{d\mu} &= \sum_{i=1}^{j} w_i \left(-\frac{1}{2\sigma^2}\right)(-1)(\log(x_i - c) - \mu) - \lambda\left(-\frac{1}{\sigma}\right)\Phi'\left(\frac{\log k - \mu}{\sigma}\right) \\
&= \frac{1}{\sigma^2}\sum_{i=1}^{j} w_i(\log(x_i - c) - \mu) + \lambda\left(\frac{1}{\sigma}\right)\Phi'\left(\frac{\log k - \mu}{\sigma}\right) = 0 \\
\frac{dL}{d\sigma} &= -n_{\downarrow}\left(\frac{1}{\sigma}\right) - \left(-\frac{1}{\sigma^3}\right)\sum_{i=1}^{j} w_i(\log(x_i - c) - \mu)^2 \\
&\qquad - \lambda\left(-\frac{\log k - \mu}{\sigma^2}\right)\Phi'\left(\frac{\log k - \mu}{\sigma}\right)
\end{aligned}$$

$$= -\frac{n_\downarrow}{\sigma} + \frac{1}{\sigma^3}\sum_{i=1}^{j} w_i(\log(x_i - c) - \mu)^2$$

$$+ \lambda\left(\frac{\log k - \mu}{\sigma^2}\right)\Phi'\left(\frac{\log k - \mu}{\sigma}\right) = 0$$

$$\frac{dL}{d\alpha} = n^\uparrow\left[\frac{1}{\alpha} + \log(x_m)\right] - \sum_{i=j+1}^{n} w_i \log(x_i - c) - \lambda \log\left(\frac{x_m}{k}\right)\left(\frac{x_m}{k}\right)^{\alpha} = 0$$

$$\frac{dL}{dx_m} = n^\uparrow \frac{\alpha}{x_m} - \lambda\left(\frac{\alpha}{k}\right)\left(\frac{x_m}{k}\right)^{\alpha-1} = 0$$

From the x_m equation, we have

$$n^\uparrow \frac{\alpha}{x_m} - \lambda\left(\frac{\alpha}{k}\right)\left(\frac{x_m}{k}\right)^{\alpha-1} = 0$$

$$n^\uparrow - \lambda\left(\frac{x_m}{\alpha}\right)\left(\frac{\alpha}{k}\right)\left(\frac{x_m}{k}\right)^{\alpha-1} = 0$$

$$n^\uparrow - \lambda\left(\frac{x_m}{k}\right)^{\alpha} = 0$$

Combining with the derivative for α gives us

$$n^\uparrow\left[\frac{1}{\alpha} + \log(x_m)\right] - \sum_{i=j+1}^{n} w_i \log(x_i - c) - n^\uparrow \log\left(\frac{x_m}{k}\right) = 0$$

$$\frac{1}{\alpha} + \log(x_m) - \frac{1}{n^\uparrow}\sum_{i=j+1}^{n} w_i \log(x_i - c) - \log(x_m) + \log k = 0$$

$$\frac{1}{\alpha} - \frac{1}{n^\uparrow}\sum_{i=j+1}^{n} w_i \log\left(\frac{x_i - c}{k}\right) = 0$$

$$\alpha = \frac{1}{\frac{1}{n^\uparrow}\sum_{i=j+1}^{n} w_i \log\left(\frac{x_i - c}{k}\right)}$$

And that gives us α in terms of data and constants! (Again assuming that c and k are constant.) Consider the derivatives with respect to μ and σ. Rearranging and dividing both equations gives us

$$\frac{-\frac{n_\downarrow}{\sigma} + \frac{1}{\sigma^3}\sum_{i=1}^{j} w_i(\log(x_i - c) - \mu)^2}{\frac{1}{\sigma^2}\sum_{i=1}^{j} w_i(\log(x_i - c) - \mu)} = \frac{-\lambda\left(\frac{\log k - \mu}{\sigma^2}\right)\Phi'\left(\frac{\log k - \mu}{\sigma}\right)}{-\lambda\left(\frac{1}{\sigma}\right)\Phi'\left(\frac{\log k - \mu}{\sigma}\right)}$$

$$-\frac{n_\downarrow}{\sigma} + \frac{1}{\sigma^3}\sum_{i=1}^{j} w_i(\log(x_i - c) - \mu)^2 = \left(\frac{\log k - \mu}{\sigma}\right)\frac{1}{\sigma^2}\sum_{i=1}^{j} w_i(\log(x_i - c) - \mu)$$

Altogether this gives us

$$\sigma^2 = \frac{1}{n_\downarrow}\sum_{i=1}^{j} w_i(\log(x_i - c) - \mu)^2 - \frac{\log k - \mu}{n_\downarrow}\sum_{i=1}^{j} w_i(\log(x_i - c) - \mu)$$

for σ^2 in terms of μ. Substituting back into (either) one of the derivatives gives us

$$\begin{aligned}
-\frac{n_\downarrow}{\sigma} + \frac{1}{\sigma^3}\sum_{i=1}^{j} w_i(\log(x_i - c) - \mu)^2 + \lambda\left(\frac{\log k - \mu}{\sigma^2}\right)\Phi'\left(\frac{\log k - \mu}{\sigma}\right) &= 0\\
-n_\downarrow\sigma^2 + \sum_{i=1}^{j} w_i(\log(x_i - c) - \mu)^2 + \lambda\sigma(\log k - \mu)\,\Phi'\left(\frac{\log k - \mu}{\sigma}\right) &= 0\\
-n_\downarrow\left[\frac{1}{n_\downarrow}\sum_{i=1}^{j} w_i(\log(x_i - c) - \mu)^2 - \frac{\log k - \mu}{n_\downarrow}\sum_{i=1}^{j} w_i(\log(x_i - c) - \mu)\right]&\\
+\sum_{i=1}^{j} w_i(\log(x_i - c) - \mu)^2 + \lambda\sigma(\log k - \mu)\,\Phi'\left(\frac{\log k - \mu}{\sigma}\right) &= 0\\
(\log k - \mu)\sum_{i=1}^{j} w_i(\log(x_i - c) - \mu) + \lambda\sigma(\log k - \mu)\,\Phi'\left(\frac{\log k - \mu}{\sigma}\right) &= 0\\
\sum_{i=1}^{j} w_i(\log(x_i - c) - \mu) + \lambda\sigma\Phi'\left(\frac{\log k - \mu}{\sigma}\right) &= 0\\
\sum_{i=1}^{j} w_i\log(x_i - c) - n_\downarrow\mu + \lambda\sigma\Phi'\left(\frac{\log k - \mu}{\sigma}\right) &= 0
\end{aligned}$$

If we examine the formula for σ^2, we see that we can simplify it.

$$\begin{aligned}
\sigma^2 &= \frac{1}{n_\downarrow}\sum_{i=1}^{j} w_i(\log(x_i - c) - \mu)^2 - \frac{\log k - \mu}{n_\downarrow}\sum_{i=1}^{j} w_i(\log(x_i - c) - \mu)\\
&= \frac{1}{n_\downarrow}\sum_{i=1}^{j} w_i\log(x_i - c)^2 - 2\mu\frac{1}{n_\downarrow}\sum_{i=1}^{j} w_i\log(x_i - c) + \mu^2\\
&\qquad - \frac{\log k}{n_\downarrow}\sum_{i=1}^{j} w_i\log(x_i - c) + \frac{\mu}{n_\downarrow}\sum_{i=1}^{j} w_i\log(x_i - c) + (\log k - \mu)\,\mu\\
&= \frac{1}{n_\downarrow}\sum_{i=1}^{j} w_i\log(x_i - c)^2 - \frac{\log k}{n_\downarrow}\sum_{i=1}^{j} w_i\log(x_i - c)\\
&\qquad + \mu\left[\log k - \frac{1}{n_\downarrow}\sum_{i=1}^{j} w_i\log(x_i - c)\right]\\
&= p_0 + p_1\mu
\end{aligned}$$

The result is that σ^2 is a linear function of μ. It follows that $\sigma = \sqrt{p_0 + p_1\mu}$. Looking back to our mass condition and the condition for x_m, we see that

$$
\begin{aligned}
n^{\uparrow} &= \lambda\left(\frac{x_m}{k}\right)^{\alpha} \\
\lambda &= \frac{n^{\uparrow}}{\left(\frac{x_m}{k}\right)^{\alpha}} \\
&= \frac{n^{\uparrow}}{1 - \Phi\left(\frac{\log k - \mu}{\sigma}\right)}
\end{aligned}
$$

It follows that μ must solve

$$
\sum_{i=1}^{j} w_i \log(x_i - c) - n_{\downarrow}\mu + n^{\uparrow}\sqrt{p_0 + p_1\mu}\,\frac{\Phi'\left(\frac{\log k - \mu}{\sqrt{p_0 + p_1\mu}}\right)}{1 - \Phi\left(\frac{\log k - \mu}{\sqrt{p_0 + p_1\mu}}\right)} = 0
$$

where

$$
\begin{aligned}
p_0 &= \frac{1}{n_{\downarrow}}\sum_{i=1}^{j} w_i \log(x_i - c)^2 - \frac{\log k}{n_{\downarrow}}\sum_{i=1}^{j} w_i \log(x_i - c) \\
p_1 &= \log k - \frac{1}{n_{\downarrow}}\sum_{i=1}^{j} w_i \log(x_i - c)
\end{aligned}
$$

This is a single equation with a single unknown and nothing that is obviously a problem for a numerical solver. So we will solve the equation numerically for μ. Then we find σ^2 easily because $\sigma^2 = p_0 + p_1\mu$. We already have α, and we can find x_m by solving the mass condition. This gives

$$
x_m = k\left[1 - \Phi\left(\frac{\log k - \mu}{\sigma}\right)\right]^{1/\alpha}
$$

for x_m.

We code this solution in several steps:

1. Write a function that returns p_0 and p_1 in terms of k and (unshifted) data. For convenience, we will say

$$
p_2 = \frac{1}{n_{\downarrow}}\sum_{i=1}^{j} w_i \log(x_i - c) = \log k - p_1
$$

   and return that value too. The `constants` will be the return values of this function, specifically p_0, p_1, p_2, $n_{\downarrow}$, and $n^{\uparrow}$.

2. Write a function that finds μ given the `constants`.
3. Write a function that finds α in terms of k and (unshifted) data.
4. Once we know μ and α it is easy to find the remaining parameters.
5. Then check whether this c-k pair results in a good-fitting model to the data.

We are expecting to feed pre-shifted data to the function, so the function does not have a shift parameter in its argument. (And the k value will also have been shifted by c. Shifting things with a cutoff parameter can get complicated quickly.) We use `_down` and `_up` to distinguish between DataFrames with incomes less than or greater than k respectively, similarly to how we notated $n_{\downarrow}$ and $n^{\uparrow}$.

```
def estimate_LogN_P_cut_cons(data, k, var, wgt):
  data_down = data[data[var] < k]
  data_up = data[data[var] >= k]
  if wgt:
    n_down = data_down[wgt].sum()
    n_up = data_up[wgt].sum()
    p2 = (1/n_down) * (data_down[wgt] * log(data_down[var])).sum()
    p0 = (1/n_down) * (data_down[wgt] * log(data_down[var])**2).sum() - \
      log(k) * p2
    p1 = log(k) - p2
  else:
    n_down = len(data_down)
    n_up = len(data_up)
    p2 = (1/n_down) * log(data_down[var]).sum()
    p0 = (1/n_down) * (log(data_down[var])**2).sum() - log(k) * p2
    p1 = log(k) - p2
  return [p0, p1, p2, n_down, n_up, k]
```

Now we define a function that accepts data as its argument and returns the value of μ. The function will also accept k as its argument and a list of constants (which come from the `_const` estimation function). The `eq_to_solve` returns the value of the expression that μ should solve.

```
def estimate_LogN_P_cut_get_mu(cons):
  p0, p1, p2, n_down, n_up, k = cons
  def eq_to_solve(mu):
    ratio = (log(k) - mu) / sqrt(p0 + p1 * mu)
    if Phi(ratio) == 1:
      hazard = np.inf
    else:
      hazard = Phi_prime(ratio) / (1 - Phi(ratio))
    return n_down * p2 - n_down * mu + n_up * sqrt(p0 + p1 * mu) * hazard
```

For the bounds on μ for the numerical solver, we set `min_val` to be the value of μ where $\sigma = 0$, i.e. $-p_0/p_1$. An application of L'Hospital's rule shows that

$$\lim_{x \to \infty} \frac{1}{\sqrt{x}} \frac{\Phi'(x)}{1 - \Phi(x)} = \infty,$$

so close to `min_val`, the expression will be positive. Then we set `max_val` equal to `min_val` and steadily increment it until we reach a candidate for μ where the expression is negative. This is guaranteed to happen eventually because as μ increases, the ratio $\log k - \mu/\sigma$ decreases and eventually becomes negative. It follows that the quotient $\Phi'/1 - \Phi$ is eventually bounded above by 1, and because $p_1 > 0$, the $-n_{\downarrow}\mu$ term dominates. Then we feed everything into the numerical solver to recover μ.

```
  min_val = -p0 / p1 + 1.0e-6
  max_val = min_val
  while eq_to_solve(max_val) > 0:
    max_val = max_val + 2
  sol = root(eq_to_solve, bracket=[min_val, max_val])
  return sol.root
```

And we need a function to find α.

```
def estimate_LogN_P_cut_get_alpha(data, cons, var, wgt):
  p0, p1, p2, n_down, n_up, k = cons
  data_up = data[data[var] >= k]
  if wgt:
    temp = (data_up[wgt] * log(data_up[var] / k)).sum()
  else:
    temp = log(data_up[var] / k).sum()
  return n_up / temp
```

If we know μ, we can recover the other parameters.

```
def estimate_LogN_P_cut_get_params(mu, alpha, cons):
  p0, p1, p2, n_down, n_up, k = cons
  sigma_sq = p0 + p1 * mu
  x_m = k * (1 - Phi((log(k) - mu) / sqrt(sigma_sq))) ** (1/alpha)
  return [sigma_sq, x_m]
```

And we are finally ready to proceed with the estimation. Same as before, we define the `_unshift` estimation function that does the heavy lifting, and then the `fit_for_c_k` and main estimator functions will find c and k to minimize the chi-squared statistic. In this function, we have already shifted k. (Because everything is unshifted here.)

```
def estimate_LogN_P_cut_unshift(data, k, var, wgt):
  constants     = estimate_LogN_P_cut_cons(data, k, var, wgt)
  mu            = estimate_LogN_P_cut_get_mu(constants)
  alpha         = estimate_LogN_P_cut_get_alpha(data, constants, var, wgt)
  sigma_sq, x_m = estimate_LogN_P_cut_get_params(mu, alpha, constants)
  return [mu, sigma_sq, x_m, alpha]
```

And now the `_fit_for_c_k` function.

```
def estimate_LogN_P_cut_fit_for_c(data, c, k, var, wgt, ecdf, x, y):
  shift_data = data.copy()
  shift_data[var] = shift_data[var] - c
```

```
  shift_data = shift_data[shift_data[var] > 0]
  shift_k = k - c
  mu, sigma_sq, x_m, alpha = estimate_LogN_P_cut_unshift(shift_data,
    shift_k, var, wgt)
  def F(x):
    return cdf_LogN_P_cut(x, [mu, sigma_sq, k, x_m, alpha, c])
  temp = {
    "fit": kolmogorov_smirnov(F, ecdf=ecdf, x=x, y=y),
    "parameters": [mu, sigma_sq, k, x_m, alpha, c]}
  return temp
```

The main estimator function will minimize the Kolmogorov-Smirnov statistic like in the previous other cases. Now we have an initial guess rather than bounds because the minimization is two-dimensional. We will use Nelder-Mead algorithm. Examination of the data suggests that the minimum chi-squared value occurs somewhere around $c = -\$9000$ and $k = \$110{,}000$, at least for 2019.

```
def estimate_LogN_P_cut(data, var, wgt=None, *, ecdf=None, x=None, y=None):
  validate_var_wgt(data, var, wgt)
  if isinstance(ecdf, type(None)):
    ecdf = make_ecdf(data, var, wgt)
    x = var
    if wgt:
      y = wgt
    else:
      if var == "y":
        y = "y1"
      else:
        y = "y"
  def check_c_k(p):
    c, k = p
    f = estimate_LogN_P_cut_fit_for_c(data, c, k, var, wgt, ecdf, x, y)["fit"]
    return f
  sol = min(check_c_k, method="Nelder-Mead", x0=[-9000,110000],
            options={"xatol": 0.1})
  c, k = sol.x
  return estimate_LogN_P_cut_fit_for_c(data, c, k, var, wgt, ecdf, x, y)
```

And add the functions to the appropriate dictionaries

```
distribution["LogN_P_cut"] = cdf_LogN_P_cut
density["LogN_P_cut"]      = density_LogN_P_cut
likelihood["LogN_P_cut"]   = L_LogN_P_cut
estimator["LogN_P_cut"]    = estimate_LogN_P_cut
```

LogN_P_mix Log-normal and Pareto mixture. Parameters:

- μ: log mean
- $\sigma^2 > 0$: log variance

- $x_m > 0$: Pareto scale parameter
- $\alpha > 0$ Pareto shape parameter
- γ: mixture parameter
- c: shift parameter (for shifted distribution)

Cumulative distribution function:

$$y = \gamma \Phi\left(\frac{\log(x-c)-\mu}{\sigma}\right) + (1-\gamma)\chi_{x \geq x_m + c}\left[1 - \left(\frac{x_m}{x-c}\right)^{\alpha}\right]$$

```
def cdf_LogN_P_mix(x, params):
  mu, sigma_sq, gamma, x_m, alpha, c = params
  if gamma < 0 or gamma > 1:
    raise ValueError("gamma is outside unit interval")
  mask_c = x <= c
  mask_c_xm = x < x_m + c
  term1 = x.astype(np.float64)
  term1[mask_c] = 0
  term1[~mask_c] = Phi((log(term1[~mask_c] - c) - mu) / sqrt(sigma_sq))
  term2 = x.astype(np.float64)
  term2[mask_c_xm] = 0
  term2[~mask_c_xm] = \
    1 - exp(alpha * (log(x_m) - log(term2[~mask_c_xm] - c)))
  return gamma * term1 + (1 - gamma) * term2
```

where $\chi_{x \geq x_m + c}$ is an indicator function. Density is

$$y = \frac{\gamma}{(x-c)\sigma\sqrt{2\pi}} e^{-(\log(x-c)-\mu)^2/2\sigma^2} + (1-\gamma)\chi_{x \geq x_m + c}\frac{\alpha x_m^{\alpha}}{(x-c)^{1+\alpha}},$$

where $\chi_{x \geq x_m + c}$ is an indicator function. Code:

```
def density_LogN_P_mix(x, params):
  mu, sigma_sq, gamma, x_m, alpha, c = params
  if gamma < 0 or gamma > 1:
    raise ValueError("gamma is outside unit interval")
  if x <= c:
    return 0
  else:
    log_n_term = exp(log(gamma) - log(x - c) -
                     0.5 * log(2 * pi * sigma_sq) -
                     (log(x - c) - mu) ** 2 / (2 * sigma_sq))
    if x <= x_m + c:
      pareto_term = 0
    else:
      pareto_term = exp(log(1 - gamma) + log(alpha) + alpha * log(x_m) -
                        (1 + alpha) * log(x - c))
    return log_n_term + pareto_term
```

Likelihood:

$$L = \sum_{i=1}^{n} w_i \log \left[\frac{\gamma}{(x_i - c)\sigma\sqrt{2\pi}} e^{-(\log(x_i - c) - \mu)^2/2\sigma^2} + (1-\gamma)\chi_{x_i \geq x_m + c} \frac{\alpha x_m^{\alpha}}{(x_i - c)^{1+\alpha}} \right]$$

Again, we use the data_up/data_down naming convention for the portion of the data that is above versus below the Pareto cutoff.

```
def L_LogN_P_mix(data, params, var, wgt=None):
  validate_var_wgt(data, var, wgt)
  mu, sigma_sq, gamma, x_m, alpha, c = params
  if gamma < 0 or gamma > 1:
    raise ValueError("gamma is outside unit interval")
  data_down = data[data[var] <= x_m + c]
  data_up = data[data[var] > x_m + c]
  if wgt:
    down_terms = (data_down[wgt] *
      (log(gamma) - log(data_down[var]- c) - 0.5 * log(2 * pi * sigma_sq) -
      (log(data_down[var] - c) - mu) ** 2 / (2 * sigma_sq))).sum()
    up_terms = (data_up[wgt] * log(
      exp(log(gamma) - log(data_up[var] - c) -
          0.5 * log(2 * pi * sigma_sq) -
          (log(data_up[var] - c) - mu)**2 / (2 * sigma_sq)) +
      exp(log(1 - gamma) + log(alpha) + alpha * log(x_m) -
          (1 + alpha) * log(data_up[var] - c)))).sum()
  else:
    down_terms = (log(gamma) -
      log(data_down[var] - c) - 0.5 * log(2 * pi * sigma_sq) -
      (log(data_down[var] - c) - mu)**2 / (2 * sigma_sq)).sum()
    up_terms = log(
      exp(log(gamma) - log(data_up[var] - c) -
          0.5 * log(2 * pi * sigma_sq) -
          (log(data_up[var] - c) - mu)**2 / (2 * sigma_sq)) +
      exp(log(1 - gamma) + log(alpha) + alpha * log(x_m) -
          (1 + alpha) * log(data_up[var] - c))).sum()
  return down_terms + up_terms
```

Unshifted version:

```
def L_LogN_P_mix_unshift(data, params, wgt, var=None):
  mu, sigma_sq, gamma, x_m, alpha = params
  return L_LogN_P_mix(data, [mu, sigma_sq, gamma, x_m, alpha, 0], var, wgt)
```

Estimating the mixture distribution is tricky. Direct numerical maximization of the likelihood function tends to converge but not to the same results every time. Accordingly, we will estimate in three steps:

1. The first (innermost) step is to maximize the likelihood with respect to μ, σ^2, and α conditional on the remaining parameters. We find μ and σ^2 using a truncated maximum-likelihood estimator on the portion of the data below $x_m + c$. Then we find α using a one-dimensional maximization.

2. The second step is choosing γ according to a first-order condition from the maximum-likelihood estimation. Potential values of γ must be large enough that the log-normal component of the mixture places enough mass on the portion of its support below $x_m + c$.
3. The third (outermost) step is a Kolmogorov-Smirnov statistic minimization with regard to c and x_m. Exploratory analysis suggests this minimum occurs for values of c and x_m relatively similar to their estimates under the cutoff model.

The three-step estimation appears to fit the model correctly.

For the log-normal parameters, we use a truncated maximum-likelihood estimator. However, simply maximizing the weighted log-sum of

$$\tilde{f}(x) = \frac{\dfrac{\gamma}{(x-c)\sigma\sqrt{2\pi}} e^{-(\log(x-c)-\mu)^2/2\sigma^2} + (1-\gamma)\chi_{x \geq x_m + c} \dfrac{\alpha x_m^\alpha}{(x-c)^{1+\alpha}}}{\gamma\Phi\left(\dfrac{\log(x-c)-\mu}{\sigma}\right) + (1-\gamma)\chi_{x \geq x_m + c}\left[1 - \left(\dfrac{x_m}{x-c}\right)^\alpha\right]\Bigg|_{x = x_m + c}}$$

$$= \frac{\gamma}{(x-c)\sigma\sqrt{2\pi}} e^{-(\log(x-c)-\mu)^2/2\sigma^2} \Bigg/ \gamma\Phi\left(\frac{\log(x_m)-\mu}{\sigma}\right)$$

$$= \frac{1}{(x-c)\sigma\sqrt{2\pi}} e^{-(\log(x-c)-\mu)^2/2\sigma^2} \Bigg/ \Phi\left(\frac{\log(x_m)-\mu}{\sigma}\right)$$

over data points less than $x_m + c$ doesn't work because it reduces to the standard maximum-likelihood estimator for the truncated log-normal distribution! In this case, the resulting probability mass allocated by the model to $[c, x_m + c]$ will be too small by a factor of λ. Instead, we maximize the likelihood subject to a mass constraint on $[c, x_m + c]$ given by the empirical mass fraction in this region. Let η be the fraction of data points less than $x_m + c$. Then the mass condition becomes

$$\Phi\left(\frac{\log(x_m)-\mu}{\sigma}\right) = \frac{\eta}{\gamma}$$

For algebraic convenience, we take logs of both sides. Then the Lagrangian is

$$L = -n\log\Phi\left(\frac{\log(x_m)-\mu}{\sigma}\right) - n\log\sigma - \frac{n}{2}\log(2\pi) - \sum_{i=1}^{n} w_i \log x_i$$
$$- \sum_{i=1}^{n} \frac{(\log x_i - \mu)^2}{2\sigma^2} + \lambda\left[\log\left(\frac{\eta}{\gamma}\right) - \log\Phi\left(\frac{\log(x_m)-\mu}{\sigma}\right)\right]$$

where λ is the Lagrange multiplier. (Throughout we assume for notational purposes that total weight has been normalized to n. But the exact total weight doesn't matter.) Notice also that we dropped the cs. To keep the notation cleaner, we perform the calculation for data that begins at 0. (So to apply to a DataFrame of income data, we should first subtract c from all incomes.) Taking derivatives gives us

$$\frac{dL}{d\mu} = -n\left(-\frac{1}{\sigma}\right)\frac{\Phi'\left(\dfrac{\log(x_m)-\mu}{\sigma}\right)}{\Phi\left(\dfrac{\log(x_m)-\mu}{\sigma}\right)} - \frac{1}{\sigma^2}\sum_{i=1}^{n}(\mu - \log x_i) - \lambda\left(-\frac{1}{\sigma}\right)\frac{\Phi'\left(\dfrac{\log(x_m)-\mu}{\sigma}\right)}{\Phi\left(\dfrac{\log(x_m)-\mu}{\sigma}\right)}$$

$$= -\frac{1}{\sigma^2}\left[n\mu - \sum_{i=1}^{n} w_i \log x_i\right] + (n+\lambda)\left(\frac{1}{\sigma}\right)\frac{\Phi'\left(\dfrac{\log(x_m)-\mu}{\sigma}\right)}{\Phi\left(\dfrac{\log(x_m)-\mu}{\sigma}\right)} = 0$$

and

$$\frac{dL}{d\sigma} = -n\left(-\frac{\log(x_m)-\mu}{\sigma^2}\right)\frac{\Phi'\left(\dfrac{\log(x_m)-\mu}{\sigma}\right)}{\Phi\left(\dfrac{\log(x_m)-\mu}{\sigma}\right)} - \frac{n}{\sigma} - \left(-\frac{1}{\sigma^3}\right)\sum_{i=1}^{n} w_i(\log x_i - \mu)^2$$

$$- \lambda\left(-\frac{\log(x_m)-\mu}{\sigma^2}\right)\frac{\Phi'\left(\dfrac{\log(x_m)-\mu}{\sigma}\right)}{\Phi\left(\dfrac{\log(x_m)-\mu}{\sigma}\right)}$$

$$= -\frac{n}{\sigma} + \frac{1}{\sigma^3}\sum_{i=1}^{n}(\log x_i - \mu^2) + (\lambda+n)\left(\frac{\log(x_m)-\mu}{\sigma^2}\right)\frac{\Phi'\left(\dfrac{\log(x_m)-\mu}{\sigma}\right)}{\Phi\left(\dfrac{\log(x_m)-\mu}{\sigma}\right)} = 0$$

We bring the (rightmost) terms containing λ to the other side and divide both equations. This gives us

$$\frac{\log(x_m)-\mu}{\sigma} = \frac{\dfrac{n}{\sigma} - \dfrac{1}{\sigma^3}\displaystyle\sum_{i=1}^{n}(\log x_i - \mu)^2}{\dfrac{n\mu}{\sigma^2} - \dfrac{1}{\sigma^2}\displaystyle\sum_{i=1}^{n}\log x_i}$$

$$\Phi^{-1}\left(\frac{\eta}{\gamma}\right)\left[\frac{n\mu}{\sigma^2} - \frac{1}{\sigma^2}\sum_{i=1}^{n}\log x_i\right] = \frac{n}{\sigma} - \frac{1}{\sigma^3}\sum_{i=1}^{n}(\log x_i - \mu)^2$$

$$\Phi^{-1}\left(\frac{\eta}{\gamma}\right)\left[\mu\sigma - \frac{\sigma}{n}\sum_{i=1}^{n}\log x_i\right] = \sigma^2 - \frac{1}{n}\sum_{i=1}^{n}(\log x_i - \mu)^2$$

$$\Phi^{-1}\left(\frac{\eta}{\gamma}\right)(\mu\sigma - \sigma\,\overline{\log x_i}) = \sigma^2 - (\overline{\log x_i^2} - 2\mu\,\overline{\log x_i} + \mu^2)$$

$$\Phi^{-1}\left(\frac{\eta}{\gamma}\right)\sigma(\mu - \overline{\log x_i}) = \sigma^2 - \overline{\log x_i^2} + 2\mu\,\overline{\log x_i} - \mu^2$$

From the mass condition, we know that

$$\begin{aligned}\Phi^{-1}\left(\frac{\eta}{\gamma}\right) &= \frac{\log(x_m) - \mu}{\sigma} \\ \sigma &= \frac{1}{\Phi^{-1}\left(\frac{\eta}{\gamma}\right)}(\log(x_m) - \mu)\end{aligned}$$

Substituting back gives us

$$(\log(x_m) - \mu)(\mu - \overline{\log x_i}) = \left[\frac{1}{\Phi^{-1}\left(\frac{\eta}{\gamma}\right)}(\log(x_m) - \mu)\right]^2 - \overline{\log x_i^2} + 2\mu\,\overline{\log x_i} - \mu^2$$

$$\Phi^{-1}\left(\frac{\eta}{\gamma}\right)^2\Big[-\mu^2 + \mu(\log(x_m) + \overline{\log x_i}) - \log(x_m)\,\overline{\log x_i}\Big]$$
$$= (\log(x_m) - \mu)^2 + \Phi^{-1}\left(\frac{\eta}{\gamma}\right)^2\left[-\overline{\log x_i^2} + 2\mu\,\overline{\log x_i} - \mu^2\right]$$

$$\Phi^{-1}\left(\frac{\eta}{\gamma}\right)^2\Big[\mu(\log(x_m) + \overline{\log x_i}) - \log(x_m)\,\overline{\log x_i}\Big]$$
$$= \mu^2 - 2\mu\log(x_m) + \log(x_m)^2 + \Phi^{-1}\left(\frac{\eta}{\gamma}\right)^2\left[-\overline{\log x_i^2} + 2\mu\,\overline{\log x_i}\right]$$

$$\mu^2 + \mu\left[\Phi^{-1}\left(\frac{\eta}{\gamma}\right)^2(\overline{\log x_i} - \log(x_m)) - 2\log(x_m)\right]$$
$$+ \left[\Phi^{-1}\left(\frac{\eta}{\gamma}\right)^2(\log(x_m)\,\overline{\log x_i} - \overline{\log x_i^2}) + \log(x_m)^2\right] = 0$$

So μ solves a quadratic equation $x^2 + bx + c = 0$ with

$$\begin{aligned}b &= \Phi^{-1}\left(\frac{\eta}{\gamma}\right)^2\Big[\overline{\log x_i} - \log(x_m)\Big] - 2\log(x_m) \\ c &= \Phi^{-1}\left(\frac{\eta}{\gamma}\right)^2\Big[\log(x_m)\,\overline{\log x_i} - \overline{\log x_i^2}\Big] + \log(x_m)^2\end{aligned}$$

```
def estimate_LogN_P_mix_logn_trunc(data, gamma, k, var, wgt):
  data_down = data[data[var] <= k]
  if wgt:
    n_down = data_down[wgt].sum()
    eta = n_down / data[wgt].sum()
    sum_log = (data_down[wgt] * log(data_down[var])).sum() / n_down
```

```
    sum_log_sq = (data_down[wgt] * log(data_down[var]) ** 2).sum() / n_down
  else:
    n_down = len(data_down)
    eta = n_down / len(data)
    sum_log = log(data_down[var]).sum() / n_down
    sum_log_sq = (log(data_down[var]) ** 2).sum() / n_down
  phi_coef = Phinv(eta / gamma)  # eta is empirical mass
  phi_coef_sq = phi_coef ** 2
  b = phi_coef_sq * (sum_log - log(k)) - 2 * log(k)
  c = phi_coef_sq * (log(k) * sum_log - sum_log_sq) + log(k) ** 2
```

Given the estimate for μ, we recover σ from the mass condition. The two solutions μ_1 and μ_2 should yield σ_1 and σ_2 with opposite signs, so we take the greater σ_i as our estimate for σ (because we require $\sigma > 0$). Because σ is monotonic decreasing in μ, this also means we want the smaller μ_i. We have

$$\mu = \frac{-b \pm \sqrt{b^2 - 4c}}{2},$$

so we use the value of μ with the negative square root.

```
  mu = (-b - sqrt(b ** 2 - 4 * c)) / 2
  sigma = (log(k) - mu) / phi_coef
  return [mu, sigma ** 2]
```

The `_inner_params` function finds the values of μ, σ^2, and α conditional on remaining parameters. Here the `data` should be unmodified (start from $-c$), and we shift it ourselves. We find α by directly maximizing the likelihood function.

```
def estimate_LogN_P_mix_inner_params(data, gamma, x_m, c, var, wgt):
  data_above_c = data[data[var] > c]
  shift_data = data_above_c.copy()
  shift_data[var] = shift_data[var] - c
  mu, sigma_sq = estimate_LogN_P_mix_logn_trunc(shift_data,
    gamma, x_m, var, wgt)
  def neg_L(a):
    return -L_LogN_P_mix(data_above_c,
      [mu, sigma_sq, gamma, x_m, a, c], var, wgt)
  sol = min(neg_L, x0=1, bounds=[(0.5, 5)], method="Nelder-Mead")
  return [mu, sigma_sq, sol.x[0]]
```

Now we need to find γ. Let j be the index dividing points less than $x_m + c$ from points greater. Taking the derivative of the likelihood function with respect to γ gives us

$$\begin{aligned}
\frac{dL}{d\gamma} &= \sum_{i=1}^{n} w_i \frac{\frac{1}{(x_i - c)\sigma\sqrt{2\pi}} e^{-(\log(x_i - c) - \mu)^2/2\sigma^2} - \chi_{x_i \geq x_m + c} \frac{\alpha x_m^\alpha}{(x_i - c)^{1+\alpha}}}{f(x_i)} \\
&= \sum_{i=1}^{j} \frac{w_i}{\gamma} + \sum_{i=j+1}^{n} w_i \frac{\frac{1}{(x_i - c)\sigma\sqrt{2\pi}} e^{-(\log(x_i - c) - \mu)^2/2\sigma^2}}{f(x_i)}
\end{aligned}$$

$$
\begin{aligned}
&- \sum_{i=j+1}^{n} \frac{\chi_{x_i \geq x_m + c} \frac{\alpha x_m^{\alpha}}{(x_i - c)^{1+\alpha}}}{f(x_i)} \\
&= \frac{n_{\downarrow}}{\gamma} + \sum_{i=j+1}^{n} \frac{w_i}{\gamma + (1-\gamma) l_i} - \sum_{i=j+1}^{n} \frac{w_i l_i}{\gamma + (1-\gamma) l_i} \\
&= \frac{n_{\downarrow}}{\gamma} + \sum_{i=j+1}^{n} \frac{w_i(1 - l_i)}{\gamma + l_i},
\end{aligned}
$$

where f is the mixture density and l_i is the likelihood ratio

$$
l_i = \frac{\alpha x_m^{\alpha}}{(x_i - c)^{1+\alpha}} \sigma \sqrt{2\pi} e^{-(\log(x_i - c) - \mu)^2 / 2\sigma^2},
$$

i.e. the ratio of Pareto to log-normal densities. We want to choose γ such that the derivative is 0. Explatory analysis suggests that the derivative is always negative for γ sufficiently close to 1, and we also require that $\eta < \gamma$ for the amount of probability mass below $x_m + c$ to be well-defined. Accordingly, we first search for a maximum on $(\eta, 1)$, and then we look for a root between the maximum and 1. In the `find_gamma` function, we will assume unmodified data (i.e. incomes start at c).

```
def estimate_LogN_P_mix_find_gamma(data, c, x_m, var, wgt):
  data_up = data[data[var] > x_m + c]
  if wgt:
    n_down = data[data[var] <= x_m + c][wgt].sum()
    n_up = data_up[wgt].sum()
    eta = n_down / data[wgt].sum()
  else:
    n_down = len(data[data[var] <= x_m + c])
    n_up = len(data_up)
    eta = n_down / len(data)
```

The function `dL` returns the derivative of the likelihood function. On $[\eta, 1]$, the maximum occurs at `max_g` and has value `max_dL`.

```
  def dL(g):
    if g <= eta or g >= 1:
      print("bad g for dL")
      return float("nan")
    mu, sigma_sq, alpha = estimate_LogN_P_mix_inner_params(data,
      g, x_m, c, var, wgt)
    l = exp(log(alpha) + alpha * log(x_m) -
      alpha * log(data_up[var] - c) + 0.5 * log(2 * pi * sigma_sq) +
      (log(data_up[var] - c) - mu) ** 2 / (2 * sigma_sq))
    term1 = n_down / g
    term2 = (data_up.weight * (1 - l) / (g + (1 - g) * l)).sum()
    return {"val": term1 + term2, "parameters": [mu, sigma_sq, g, alpha]}
  max_g = min_s(lambda x: -dL(x)["val"], method="bounded",
    bounds=[0.99 * eta + 0.01, 0.01 * eta + 0.99]).x
  max_dL = dL(max_g)["val"]
```

Now we want to find the root of dL, which will be our estimate of γ conditional on x_m and c. We know that dL is maximized at max_g, and the function should be negative near 1. If max_dL is negative, then the function is negative everywhere, and we have no estimate for γ, which means the original c and x_m values were bad. If max_dL is positive, then dL has a root somewhere between max_g and 1, which we find with a bounded root finder.

```
  if max_dL == 0:
    return dL(max_g)["parameters"]
  elif max_dL > 0:
    delta = 0.01
    while dL((delta) * max_g + (1 - delta))["val"] > 0:
      delta = delta / 10
    bound = (delta) * max_g + (1 - delta)
    g = root(lambda x: dL(x)["val"], bracket=[max_g, bound], xtol=1e-5).root
    return dL(g)["parameters"]
  elif max_dL < 0:
    return False
```

Now we have the fit_for_c function. This function calls the find_gamma function to calculate optimal paramter values conditional on x_m and c. Then it returns the value of the objective function and parameters. If the return value of find_gamma is a dictionary, we carry out the calculation normally. Otherwise, we have bad values, so we set the fit value to 1.

```
def estimate_LogN_P_mix_fit_for_c(data, c, x_m, var, wgt, ecdf, x, y):
  shift_data = data.copy()
  shift_data[var] = shift_data[var] - c
  shift_data = shift_data[shift_data[var] > 0]
  temp_params = estimate_LogN_P_mix_find_gamma(data, c, x_m, var, wgt)
  if temp_params:
    mu, sigma_sq, gamma, alpha = temp_params
    def F(x):
      return cdf_LogN_P_mix(x, [mu, sigma_sq, gamma, x_m, alpha, c])
    temp = {
      "fit": kolmogorov_smirnov(F, ecdf=ecdf, x=x, y=y),
      "parameters": [mu, sigma_sq, gamma, x_m, alpha, c]}
    return temp
  else:
    print("bad gamma, xm = {0}, c = {1}".format(x_m, c))
    return {"fit": 1, "parameters": []}
```

Unfortunately, numerically minimizing the objective function appears to work poorly for the mixture model. We appear to be running into some practical identifiability issues here. So we do a brute-force minimization similar to how we handled GB2.

```
def estimate_LogN_P_mix_get_cxm(dict):
  return [dict["parameters"][-1], dict["parameters"][3]]
```

Now we code the main estimation function. The setup is the same as for GB2 with a brute-force search using three iterations with the same mesh sizes. In the first iteration, we use a partition mesh size of $500, and in the second iteration, we use a partition mesh size of $100 around the ten best-fitting pairs from the first iteration. In the third iteration, we use a partition mesh size of $10 around the ten best-fitting pairs from the second iteration. Unlike with GB2, we don't have to check the value of the α parameter.

```
def estimate_LogN_P_mix(data, var, wgt=None, *, ecdf=None, x=None, y=None):
  validate_var_wgt(data, var, wgt)
  if isinstance(ecdf, type(None)):
    ecdf = make_ecdf(data, var, wgt)
    x = var
    if wgt:
      y = wgt
    else:
      if var == "y":
        y = "y1"
      else:
        y = "y"
```

First iteration. The $\langle x_m \; or \; c\rangle$`_vals` lists contain the test values of the variables we are using, and the `best_fits` list stores corresponding values of the objective function.

```
  x_m_vals = [35000 + 500*i for i in range(41)]  # xm in [35000, 55000]
  c_vals = [-12000 + 500*i for i in range(21)]   # c  in [-12000,-2000]
  best_fits = []
  the_time()
  print("First iteration of brute-force search")
  for x_m in x_m_vals:
    for c in c_vals:
      temp = estimate_LogN_P_mix_fit_for_c(data, c, x_m, var, wgt, ecdf, x, y)
```

Again, same as with GB2, we modify `best_fits` dynamically. If the list has fewer than ten entries, we add the current results. Otherwise, we check whether the current result fits better than the last entry in `best_fits`. If no, we ignore. If yes, we insert it into `best_fits` at the first position where the current result fits worse than previous items in `best_fits`.

```
      if len(best_fits) < 10:
        best_fits.append(temp)
      else:
        if temp["fit"] < best_fits[-1]["fit"]:
          i = 0
          while temp["fit"] > best_fits[i]["fit"]:
            i = i + 1
          best_fits.insert(i, temp)
          best_fits.pop()
```

For the second iteration, we store the (c, x_m) values in `pairs`. First extract the values from best_fits, and then we populate the grid of parameter values to test. We turn pairs into a set to avoid duplicate parameter pairs.

```
pairs = [estimate_LogN_P_mix_get_cxm(i) for i in best_fits]
pairs = [(p[0] - 500 + 100*i, p[1]) for p in pairs for i in range(11)]
pairs = [(p[0], p[1] - 500 + 100*i) for p in pairs for i in range(11)]
pairs = set(pairs)
best_fits = []
the_time()
print("Second iteration of brute-force search")
for p in pairs:
  temp = estimate_LogN_P_mix_fit_for_c(data, *p, var, wgt, ecdf, x, y)
  if len(best_fits) < 10:
    best_fits.append(temp)
  else:
    if temp["fit"] < best_fits[-1]["fit"]:
      i = 0
      while temp["fit"] > best_fits[i]["fit"]:
        i = i + 1
      best_fits.insert(i, temp)
      best_fits.pop()
```

Now do this again for the third iteration. This time, we save just the best-fitting entry, not the whole list.

```
pairs = [estimate_LogN_P_mix_get_cxm(i) for i in best_fits]
pairs = [(p[0] - 100 + 10*i, p[1]) for p in pairs for i in range(21)]
pairs = [(p[0], p[1] - 100 + 10*i) for p in pairs for i in range(21)]
pairs = set(pairs)
solution = {"fit":2}
the_time()
print("Third iteration of brute-force search")
for p in pairs:
  temp = estimate_LogN_P_mix_fit_for_c(data, *p, var, wgt, ecdf, x, y)
  if temp["fit"] < solution["fit"]:
    solution = temp
return solution
```

And add functions to dictionaries.

```
distribution["LogN_P_mix"] = cdf_LogN_P_mix
density["LogN_P_mix"]      = density_LogN_P_mix
likelihood["LogN_P_mix"]   = L_LogN_P_mix
estimator["LogN_P_mix"]    = estimate_LogN_P_mix
```

File V

check_constants.py

This file contains code to investigate possible values for the constants in the inverse-gamma variants. We know that β and c are proportional, and we observe a time-dependent relationship between these parameters and α. Accordingly, we will model our coefficients as

$$c_t = \phi\beta_t$$
$$c_t = \psi_0 + \psi_1 t + \psi_2 \alpha_t \qquad \text{or} \qquad c_t = \alpha_t(\psi_0 + \psi_1 t),$$

and we want to estimate ϕ, ψ_0, ψ_1, and possibly ψ_2 for this relationship. It doesn't make sense to estimate ϕ in a way that depends on α since ϕ has nothing to do with α. So we'll estimate these constants in two steps: (1) find ϕ and then (2) minimize an objective function for ψ. If β and c are proportional, then we only need to determine the ratio of their magnitudes. It makes most sense to use the L^1 norm for magnitude in this context, so

$$\phi = \frac{-||c||_1}{||\beta||_1} = \sum_t c_t \Big/ \sum_t \beta_t,$$

Then we find ψ. We want to balance making $\alpha_t(\psi_0 + \psi_1 t)$ close to both $\phi\beta_t$ and c_t, so our objective function will be

$$L = \sum_t (\phi\beta_t - f(\alpha_t, t))^2 + \sum_t (c_t - f(\alpha_t, t))^2,$$

where ϕ is the conversion factor from above and f is the relationship between α and the other two parameters. It will be helpful to write our objective function in terms of vectors and matrices because the analysis is simpler with linear algebra. Let

$$A = \begin{bmatrix} 1 & \alpha_{t_0} & t_0 \\ 1 & \alpha_{t_1} & t_1 \\ \vdots & \vdots & \\ 1 & \alpha_{t_{n-1}} & t_{n-1} \end{bmatrix} \qquad \text{or} \qquad \begin{bmatrix} \alpha_{t_0} & \alpha_{t_0} t_0 \\ \alpha_{t_1} & \alpha_{t_1} t_1 \\ \vdots & \vdots \\ \alpha_{t_{n-1}} & \alpha_{t_{n-1}} t_{n-1} \end{bmatrix}$$

$$B = \begin{bmatrix} \beta_{t_0} \\ \beta_{t_1} \\ \vdots \\ \beta_{t_{n-1}} \end{bmatrix} \qquad\qquad C = \begin{bmatrix} c_{t_0} \\ c_{t_1} \\ \vdots \\ c_{t_{n-1}} \end{bmatrix}$$

be matrices containing the parameter data and (in the case of A) years. If $\psi = [\psi_0 \;\; \psi_1 \;\; \psi_2]^{\mathrm{T}}$ or $[\psi_0 \;\; \psi_1]^{\mathrm{T}}$, then our modeling equations become

$$C = A\psi \qquad\qquad \phi B = A\psi,$$

so we can write our objective function as

$$L = ||C - A\psi||^2 + ||\phi B - A\psi||^2$$

$$
\begin{aligned}
&= (C - A\psi)^{\mathrm{T}}(C - A\psi) + (\phi B - A\psi)^{\mathrm{T}}(\phi B - A\psi) \\
&= C^{\mathrm{T}}C - \psi^{\mathrm{T}}A^{\mathrm{T}}C - C^{\mathrm{T}}A\psi + \phi^2 B^{\mathrm{T}}B - \phi\psi^{\mathrm{T}}A^{\mathrm{T}}B \\
&\qquad - \phi B^{\mathrm{T}}A\psi + 2\psi^{\mathrm{T}}A^{\mathrm{T}}A\psi.
\end{aligned}
$$

Each term in this sum is a scalar, so we can freely take the transpose of $C^{\mathrm{T}}A\psi$ and $B^{\mathrm{T}}A\psi$. This gives us

$$L = \phi^2 B^{\mathrm{T}}B + C^{\mathrm{T}}C - 2\psi^{\mathrm{T}}A^{\mathrm{T}}(\phi B + C) + 2\psi^{\mathrm{T}}A^{\mathrm{T}}A\psi.$$

Taking the derivative with respect to ψ and setting equal to 0 gives us

$$
\begin{aligned}
\frac{dL}{d\psi} = -2(\phi B^{\mathrm{T}} + C^{\mathrm{T}})A\psi + 4\psi^{\mathrm{T}}A^{\mathrm{T}}A &= 0 \\
4\psi^{\mathrm{T}}A^{\mathrm{T}}A &= 2(\phi B^{\mathrm{T}} + C^{\mathrm{T}})A \\
\psi^{\mathrm{T}} &= \frac{1}{2}(\phi B^{\mathrm{T}} + C^{\mathrm{T}})A(A^{\mathrm{T}}A)^{-1} \\
\psi &= \frac{1}{2}(A^{\mathrm{T}}A)^{-1}A^{\mathrm{T}}(\phi B + C)
\end{aligned}
$$

Interestingly, this is the same estimate for ψ that we get from minimizing

$$\left|\left|\frac{\phi B + C}{2} - A\psi\right|\right|.$$

Now we implement. We first import modules and data.

```
import pandas as pd
import numpy as np
import numpy.linalg as la
```

We will import this file within `main.py`. To avoid running code on import, we run the analysis inside a function. First we define a couple helper functions. The first helper function raises a TypeError if `x` is not a DataFrame.

```
def check_df(x):
  if not isinstance(x, pd.DataFrame):
    msg = """
The second arguments of main() should be a Pandas
DataFrame. Right now one it is {0}
instead.\n""".format(type(x))
    raise TypeError(msg)
```

Second helper function. It checks if `col` is a column in `x`.

```
def check_col(x, col):
  if col not in x.columns:
    msg = """
The third, fourth, and fifth arguments of main() should
be columns in data (second argument). However, it looks
like {0} is not a column in data.\n""".format(col)
    raise KeyError(msg)
```

Table 7: (Proportional) Constants for Inverse Gamma

Variable	Value
ϕ	-0.1337
ψ_0	\$206,824
ψ_1	$-$\$105.33/year

Main function. We begin by checking arguments and disabling the chained assignment warning from Pandas.

```
def main(years, data, a_col, b_col, c_col):
  check_df(data)
  check_col(data, a_col)
  check_col(data, b_col)
  check_col(data, c_col)
  pd.options.mode.chained_assignment = None
```

It will be easier to code everything if we save everything as arrays now.

```
  alpha = data[a_col].to_numpy()
  temp = data[a_col].to_numpy() * years.to_numpy()
  A_lin  = np.concatenate([np.ones([len(years), 1]),
    np.transpose([years]), np.transpose([alpha])], axis=1)
  A_prop = np.concatenate([np.transpose([alpha]),
    np.transpose([temp])], axis=1)
  B = data[b_col].to_numpy()
  C = data[c_col].to_numpy()
```

Now estimate ϕ.

```
  phi = np.sum(C) / np.sum(B)
```

And ψ.

```
  psi_lin = 0.5 * la.inv(A_lin.transpose() @ A_lin) @ \
    A_lin.transpose() @ (phi * B + C)
  psi_prop = 0.5 * la.inv(A_prop.transpose() @ A_prop) @ \
    A_prop.transpose() @ (phi * B + C)
```

Finally, we return the values.

```
  return {
    "linear":
      {"phi": phi, "psi0": psi_lin[0],
       "psi1": psi_lin[1], "psi2": psi_lin[2]},
    "proportional":
      {"phi": phi, "psi0": psi_prop[0], "psi1": psi_prop[1]}}
```

Table 7 contains the values of constants from this estimation on the public CPS data.

File VI

`bootstrap.py`

This file implements the bootstrapping to find standard errors on the constant-shift-scale inverse-gamma distributions. The original approach was to implement a synthetic clustering and stratification design based on the synthetic survey design described in Jolliffe (2003). However, that approach appears to be problematic because it produces biased estimates of the parameter under the synthetic datasets. So this file will contain three functions: one for a naive bootstrap, one for a bootstrap with a synthetic survey design based on Jolliffe (2003), and one with a synthetic survey design based on Jolliffe (2003) except with a different set of clusters that results in unbiased parameter estimates. Eventually we will bound the standard error in the parameter between standard deviations from the first and third boostrap functions above.

In this file, we will define the following functions:

- `bootstrap_naive(`⟨*estim*⟩`, `⟨*data*⟩`, `⟨*var*⟩`, `⟨*wgt*⟩`, `⟨*n*⟩`=100, **kwargs)`—function that implements a naive bootstrap using the variable `var` from `data` with `wgt` as the survey weights. The number of replicates `n` defaults to 200. The `kwargs` get fed to the estimation function.
- `bootstrap_Jol(`⟨*estim*⟩`, `⟨*data*⟩`, `⟨*var*⟩`, `⟨*wgt*⟩`, `⟨*strat*⟩`, `⟨*clust*⟩`, `⟨*n*⟩`=100, **kwargs)` —function that implements a bootstrap according to the synthetic survey design outlined in Jolliffe (2003). The `strat` and `clust` arguments should be names of columns in the data that we use for stratification and clustering. For income data, `strat` will be region or state, and `clust` will be household id.
- `bootstrap_Jol_sim(`⟨*estim*⟩`, `⟨*data*⟩`, `⟨*var*⟩`, `⟨*wgt*⟩`, `⟨*strat*⟩`, `⟨*cluster_size*⟩`=10, `⟨*n*⟩`=100, **kwargs)`—function that implements a simplified version of the Jolliffe (2003) synthetic survey design. In this case, the clusters will be collections of successive observations when we sort the dataset by `var`.

A minor technical note is that changes in the weights in the head and the tail can bias individual parameter estimates, i.e. create extreme outliers. To avoid issues from this problem, we use a normalized interquartile range to estimate the standard deviation of the parameter estimates.

We begin by importing modules and defining a few bookkeeping functions.

```
import numpy as np
import pandas as pd
def check_column(data, col):
  if not isinstance(data, pd.DataFrame):
    raise TypeError("Data should be a DataFrame.")
  if col not in data.columns:
    raise KeyError("{0} is not a column in the data.".format(col))
generator = np.random.default_rng(12345)
```

The bootstrap function is relatively straightforward.

```
def bootstrap_naive(F, df, var, wgt=None, n=100, **kwargs):
  data = df.copy()
```

```
check_column(data, var)
#data.sort_values(var, inplace=True)
```

The with_wgt boolean will store whether we called the bootstrap function with a wgt column specified in the function argument.

```
if isinstance(wgt, type(None)):
  with_wgt = False
else:
  check_column(data, wgt)
  with_wgt = True
```

Now add the synthetic weights to the DataFrame, and estimate parameters. If the user called bootstrap with wgt specified, then we set lam to the wgt column of data. Otherwise, we use all 1's for the weights. We create new_weight by simulating n Poisson distribution values for each row of the data, where the parameter of the Poisson distribution is lam (either 1 for wgt unspecified or the corresponding entry from wgt column otherwise).

```
if with_wgt:
  data[wgt] = data[wgt] / data[wgt].sum() * len(data)
  lam = data[wgt]
else:
  lam = 1
```

The generator.poisson creates a matrix of Poisson realizations using a vector lam of parameters. Each element of lam determines the Poisson rate parameter used for the realizations in each column, and the elements of a single column are identically distributed. So to use this array in the definition of our DataFrame, we first transpose it.

```
new_weights = pd.DataFrame(
  generator.poisson(lam=lam, size=[n,len(data)]).transpose(),
  index=data.index,
  columns=["_brw" + str(i) for i in range(n)])
data = pd.concat([data, new_weights], axis=1)
temp = [F(data, var=var, wgt="_brw"+str(i), **kwargs) for i in range(n)]
return temp
```

The function returns a list of dictionaries.

The bootstrap_Jol() function will calculate the bootstrap using a synthetic stratification and clustering scheme from Jolliffe (2003). The process contains several steps:

1. Find the strata by using .unique() method on the stratification column.
2. Within each stratum, create a list of total income for each household (distinct entries in the clust column) using .groupby(), and sort the household id's by income.
3. Use np.arrange() to add a column of integers to the DataFrame and integer-divide by 4 to get the synthetic cluster id's. These will go in the column id.
4. Loop through household identifiers, and for each household, add the corresponding cluster id to the id column in data.
5. Then we make the bootstrap replicate weights. Each replicate weight is a Poisson random variable. Choosing the Poisson parameter is a bit tricky. Let n_i denote the number of clusters in stratum i, and let w_i denote the total weight attached to stratum i. Let c_j be the total weight on cluster j in stratum i. Then c_j/w_i is the fraction of

stratum weight that corresponds to cluster j. From Wolter (2007), we know that we should think in terms of picking clusters for the bootstrap rather than individual observations, so we want to multiply this fraction by n_i to get the correct Poisson parameter for cluster j. Because

$$c_j = \sum_{k \text{ in cluster } j} p_k,$$

where p_k is the weight attached to observation k, we should use $n_i p_k / w_i$ for the Poisson parameters for each individual.

6. Replace each individual weight with the total weight of individuals in the same cluster.
7. Normalize the weight in each stratum to be the same as the original weight. The result will be the bootstrap replicate weights.

Once we have the new weights, we call the estimation function on the data with each set of synthetic weights. The function returns the results from each estimate in a list. We begin the function definition with some error checking and bookkeeping variables.

```
def bootstrap_Jol(F, df, var, wgt, strat, clust, n=100, **kwargs):
  data = df.copy()
  check_column(data, var)
  check_column(data, strat)
  check_column(data, clust)
  check_column(data, wgt)
  #data.sort_values(var, inplace=True)
  household_incomes = {}
  clusters_per_strat = {}
  strata = data[strat].unique()
  weight_per_strat = data.groupby(strat).sum()[wgt]
```

We add an `id` column containing household id, a column `_weight_per_strat` containing the total weight of the stratum corresponding to the current row, and a column `_clusters_per_strat` containing the total number of clusters in the stratum corresponding to the current row. We don't do checking of column names since this function is mostly for the current project, where we already know all the column names.

```
  data["_id"] = 0
  data["_weight_per_strat"] = 0
  data["_clusters_per_strat"] = 0
```

Loop through the strata. Put the Series of household income levels in `household_incomes` dictionary. We add an `id` column that contains the cluster identifier.

```
  for s in strata:
    household_incomes[s] = \
      data[data[strat] == s].groupby(clust).sum()[[var]].sort_values(var)
    household_incomes[s]["_id"] = np.arange(len(household_incomes[s])) // 4
    clusters_per_strat[s] = household_incomes[s]["_id"].max() + 1
    data.loc[data[strat] == s, "_weight_per_strat"] = weight_per_strat[s]
    data.loc[data[strat] == s, "_clusters_per_strat"] = clusters_per_strat[s]
```

Now loop through the list of households in the stratum, and add each household's cluster identifier to `data`. It would be simpler and faster to use `np.repeat()` here (as well as in a number of other places in this function and the `Jol_sim` function). Unfortunately, I didn't know about `np.repeat()` when I wrote this file, and it's not worth the effort to redo it.

```
    for h in household_incomes[s].index:
      data.loc[data[clust] == h, "_id"] = household_incomes[s].loc[h, "_id"]
```

We are ready to create the new synthetic survey weights. Again `new_weights` is a DataFrame of Poisson distribution realizations, where the weights are the total weight on each synthetic cluster.

```
  new_weights = pd.DataFrame(
    generator.poisson(lam=(data[wgt] * data["_clusters_per_strat"] /
      data["_weight_per_strat"]),
      size=[n,len(data)]).transpose().astype(float),
    index=data.index,
    columns=["_brw" + str(i) for i in range(n)])
```

Now sum the Poisson weights across people in each synthetic cluster.

```
  for s in strata:
    for c in range(clusters_per_strat[s]):
      new_weights.loc[(data[strat] == s) & (data["_id"] == c)] = \
        np.array(new_weights.loc[(data[strat] == s) &
                                 (data["_id"] == c)].sum(axis=0))
```

Normalize the total weight in each stratum to be the original weight in that stratum.

```
  for s in strata:
    new_weights[data[strat] == s] = (
      new_weights[data[strat] == s]
      * weight_per_strat[s]
      / np.array(new_weights[data[strat] == s].sum()))
```

Finally, we concatenate the weights with the original data and call the estimation function on each set of bootstrap replicate weights.

```
  data = pd.concat([data, new_weights], axis=1)
  temp = [F(data, var=var, wgt="_brw"+str(i), **kwargs) for i in range(n)]
  return temp
```

We return a list of estimates.

Using synthetic clusters adapted directly as it's written in the paper produces biased parameter estimates, so we use a simplified approach to get (what we hope to be?) an upper bound on the standard error in the parameter estimate. The `bootstrap_Jol_sim` function groups observations by stratum, then forms clusters by ranking observations by income and grouping 10 consecutive observations, since clusters in the CPS have about four households on average, and the average household size in the U.S. is about 2.5 people. The `clust_size` argument stores the cluster size.

```
def bootstrap_Jol_sim(F, df, var, wgt, strat, clust_size=10, n=100, **kwargs):
  data = df.copy()
  check_column(data, var)
  check_column(data, strat)
  check_column(data, wgt)
  #data.sort_values(var, inplace=True)
```

```
clusters_per_strat = {}
strata = data[strat].unique()
weight_per_strat = data.groupby(strat).sum()[wgt]
```

Add a `_weight_per_strat` and `_clusters_per_strat` columns containing weight and number of clusters per stratum. We don't need household id's this time since we aren't using houshold information, but we will still add an `_id` column to keep track of the synthetic clusters.

```
data["_id"] = 0
data["_weight_per_strat"] = 0
data["_clusters_per_strat"] = 0
```

Loop through the strata. Put the Series of household income levels in `household_incomes` dictionary. We set the `id` column to contain the cluster identifier.

```
for s in strata:
  temp = data[data[strat] == s].sort_values(var)
  temp["_id"] = np.arange(len(temp)) // clust_size
  clusters_per_strat[s] = temp["_id"].max() + 1
  data.loc[data[strat] == s, "_id"] = temp["_id"]
  data.loc[data[strat] == s, "_weight_per_strat"] = weight_per_strat[s]
  data.loc[data[strat] == s, "_clusters_per_strat"] = clusters_per_strat[s]
```

We are ready to create the new synthetic survey weights. For each individual, we replace that individual's weight by the sum of all weights of individuals in the same cluster. Again `new_weights` is a DataFrame of Poisson distribution realizations, where the weights are the total weight on each synthetic cluster.

```
new_weights = pd.DataFrame(
  generator.poisson(lam=(data[wgt] * data["_clusters_per_strat"] /
    data["_weight_per_strat"]),
    size=[n,len(data)]).transpose().astype(float),
  index=data.index,
  columns=["_brw" + str(i) for i in range(n)])
```

Now sum the Poisson weights across people in each synthetic cluster.

```
for s in strata:
  for c in range(clusters_per_strat[s]):
    new_weights.loc[(data[strat] == s) & (data["_id"] == c)] = \
      np.array(new_weights.loc[(data[strat] == s) &
                               (data["_id"] == c)].sum(axis=0))
```

Normalize the total weight in each stratum to be the original weight in that stratum.

```
for s in strata:
  new_weights[data[strat] == s] = (
    new_weights[data[strat] == s]
    * weight_per_strat[s]
    / np.array(new_weights[data[strat] == s].sum()))
```

As in the other two functions, we concatenate the weights and data and call the estimation function using each set of bootstrap replicate weights.

```
data = pd.concat([data, new_weights], axis=1)
```

```
temp = [F(data, var=var, wgt="_brw"+str(i), **kwargs) for i in range(n)]
return temp
```

Again the result will be a list of estimates (dictionaries in this context).

File VII
make_figures.py

This file contains the code to make figures. All handling of Matplotlib and Pyplot commands happens here, and `main.py` calls the functions and provides the data. We define several graphics-producing functions:

- `single_graph`—makes a single graph
- `triple_graph`—makes three panels where the first two are next to each other, and the third is below them and centered
- `lin_loglog_graphs`—makes three rows of two graphs each, where the first column is linear and the second column is loglog scaling
- `lin_graphs`—makes six linearly scaled graphs
- `loglog_graphs`—makes six loglog scaled graphs
- `four_graphs`—makes four graphs in 2x2 layout; hook for extra code
- `single_graph_ext`—makes a single graph but with more control over the contents; similar to `four_graphs` in its implementation/interface

All of these functions use the `plt.savefig()` function to save the figure directly as a pdf at the correct size. The first thing to do is import modules.

```
import bin
import matplotlib as mpl
import matplotlib.pyplot as plt
import numpy as np
import pandas as pd
```

Some `rcParams` to make the figures look nice. You will need a working TeX distribution on your machine to produce figures, or you may have to rewrite some figure titles and labels.

```
mpl.rcParams["legend.fontsize"] = "small"
mpl.rcParams["font.family"] = "serif"
mpl.rcParams["text.usetex"] = True      # tells Pyplot to use TeX
mpl.rcParams["figure.constrained_layout.use"] = True
mpl.rcParams["savefig.dpi"] = 300
mpl.rcParams["savefig.format"] = "pdf"
```

Bookkeeping functions.

```
def check_var(data, var):
  if var not in data:
    raise KeyError("{0} is not a column in the data".format(var))
def check_list(x):
  is_list_like = hasattr(x, "__len__") and \
                 hasattr(x, "__getitem__") and \
                 hasattr(x, "__iter__")
  if not is_list_like or isinstance(x, str):
    raise TypeError("Please use list instead of {0} for {1}".format(type(x),x))
```

We begin by creating a function to show a single line graph. We will use this to create the figure of constant-shift-scale inverse-gamma parameters.

```
def single_graph(data, parameter, *, filename, title=None):
  plt.close(plt.gcf())
  if isinstance(title, type(None)):
    called_with_title = False
  else:
    called_with_title = True
  check_var(data, parameter)
  plt.plot(data.index, data[parameter], c="black", lw=0.5)
  if called_with_title:
    plt.title(title)
  #plt.show()
  plt.gcf().set_size_inches(3.25, 2.5)
  plt.savefig(filename)
  plt.close(plt.gcf())
```

The next function creates three panels (for showing inverse-gamma parameter estimates). We accept one DataFrame (of inverse-gamma parameters), parameter names, and titles for the subplots.

```
def triple_graph(data, shape, scale, shift, *, filename, titles=[]):
  plt.close(plt.gcf())
  check_list(titles)
  if len(titles) > 0 and len(titles) < 3:
    raise ValueError("Please specify zero or all titles")
  elif len(titles) == 0:
    called_with_titles = False
  else:
    called_with_titles = True
  check_var(data, shape)
  check_var(data, scale)
  check_var(data, shift)
```

Now fill the first three subplots. We use a GridSpec object to get the positioning right.

```
  plt.figure()
  grid = mpl.gridspec.GridSpec(4, 4, figure=plt.gcf())
  grid_boxes = [grid[0:2, 0:2], grid[0:2, 2:4], grid[2:4, 1:3]]
  parameter_names = [shape, scale, shift]
  for i in range(3):
    plt.subplot(grid_boxes[i])
    plt.plot(data.index, data[parameter_names[i]], lw=0.5, c="black")
    if called_with_titles:
      plt.title(titles[i])
```

Plot the figure.

```
  #plt.tight_layout()
  #plt.show()
  plt.gcf().set_size_inches(6.5, 5)
  plt.savefig(filename)
  plt.close(plt.gcf())
```

The `lin_loglog_graphs` function is more complicated. It makes a 3x2 plot of three rows of sample and model densities in linear and log-log scales. Arguments of the function are

- Three DataFrames
- `var` (and `wgt=None`) columns
- Several lists of things, where we plot everything from each list on one row of the subplot array
  - `list_F1`, `list_F2`, and `list_F3`—density functions, which take a single argument and return a (nonnegative) real number
  - `list_c1`, `list_c2`, and `list_c3`—minimum values of the support
  - `list_opt1`, `list_opt2`, and `list_opt3`—list of dictionaries containing keyword-options for plotting each density function
- Bounds on the graphs
- A filename for saving the figure
- An optional list of titles and boolean to determine if using a legend

For the lists of functions, constants, and options, we will plot each list on one row of the subplot array. The left column is a linear scaling, and the right column is a loglog scaling. We need to use lists of functions and constants so that we can make a figure where each row plots multiple functions. We begin with error checking and setting the `called_with_titles` boolean.

```
def lin_loglog_graphs(data1, data2, data3, var,
    list_F1, list_c1, list_opt1,
    list_F2, list_c2, list_opt2,
    list_F3, list_c3, list_opt3,
    x_bounds_lin, y_bounds_lin, x_bounds_log, y_bounds_log, *,
    filename, wgt=None, titles=[], with_legend=False):
  plt.close(plt.gcf())
  check_list(titles)
  if len(titles) > 0 and len(titles) < 6:
    raise ValueError("Please specify zero or all titles")
  elif len(titles) == 0:
    called_with_titles = False
  else:
    called_with_titles = True
```

We loop through the rows of the subplot. For each row, we store the lists of density functions, constants, and option dictionaries in new variables. Then we check that all of them are in fact a list. We also make a new variable for the data, check that `var` is a column, and feed the data to `bin.bin_data`.

```
  for i in range(1, 4):
    list_F = eval("list_F" + str(i))
    list_c = eval("list_c" + str(i))
    list_opt = eval("list_opt" + str(i))
    check_list(list_F)
    check_list(list_c)
    check_list(list_opt)
```

```
    d = eval("data" + str(i))
    check_var(d, var)
    binned_data = bin.bin_data(d, var, wgt)
```

Start with the linear plot. We add a title, a curve for the density, and sample density points. The `x_vals` list starts with `c_i` so that it is clear on the figure where the support of the model ends. We loop through the lists of functions, constants, and options concurrently and plot each one separately on the current plot.

```
    plt.subplot(3, 2, 2 * i - 1)
    if called_with_titles:
      plt.title(titles[2 * i - 2])
    for F, c, opt in zip(list_F, list_c, list_opt):
      x_vals = np.linspace(c, x_bounds_lin[1], 200)
      y_vals = list(map(F, x_vals))
      plt.plot(x_vals, y_vals, **opt)
    plt.scatter(binned_data["mid"], binned_data["dens"], s=2, c="blue")
    plt.xlim(x_bounds_lin)
    plt.ylim(y_bounds_lin)
    if with_legend:
      plt.legend(loc="upper right")
```

Now do the loglog plot. The code is similar except that this time, the `x_vals` list runs for the entire length of the horizontal axis.

```
    plt.subplot(3, 2, 2 * i)
    if called_with_titles:
      plt.title(titles[2 * i - 1])
    x_vals = np.geomspace(*x_bounds_log, 200)
    for F, c, opt in zip(list_F, list_c, list_opt):
      y_vals = list(map(F, x_vals))
      plt.plot(x_vals, y_vals, **opt)
    plt.scatter(binned_data["mid"], binned_data["dens"], s=2, c="blue")
    plt.xlim(x_bounds_log)
    plt.ylim(y_bounds_log)
    plt.loglog()
    if with_legend:
      plt.legend(loc="lower left")
```

After the for-loop, plot the figure.

```
  #plt.show()
  plt.gcf().set_size_inches(6.5, 7.5)
  plt.savefig(filename)
  plt.close(plt.gcf())
```

The `lin_graphs` and `loglog_graphs` functions will be the same as `lin_loglog_graphs` except that they take more data and scale their plots all the same way.

```
def lin_graphs(data1, data2, data3,
    data4, data5, data6, var,
    list_F1, list_c1, list_opt1,
    list_F2, list_c2, list_opt2,
```

```
    list_F3, list_c3, list_opt3,
    list_F4, list_c4, list_opt4,
    list_F5, list_c5, list_opt5,
    list_F6, list_c6, list_opt6,
    x_bounds, y_bounds, *,
    filename, wgt=None, titles=[], with_legend=True):
  plt.close(plt.gcf())
  check_list(titles)
  if len(titles) > 0 and len(titles) < 6:
    raise ValueError("Please specify zero or all titles")
  elif len(titles) == 0:
    called_with_titles = False
  else:
    called_with_titles = True
```

Now loop through the panels, create pointers, and check list properties.

```
  for i in range(1, 7):
    list_F = eval("list_F" + str(i))
    list_c = eval("list_c" + str(i))
    list_opt = eval("list_opt" + str(i))
    check_list(list_F)
    check_list(list_c)
    check_list(list_opt)
    d = eval("data" + str(i))
    check_var(d, var)
    binned_data = bin.bin_data(d, var, wgt)
```

Plot the figures.

```
    plt.subplot(3, 2, i)
    if called_with_titles:
      plt.title(titles[i - 1])
    for F, c, opt in zip(list_F, list_c, list_opt):
      x_vals = np.linspace(c, x_bounds[1], 200)
      y_vals = list(map(F, x_vals))
      plt.plot(x_vals, y_vals, **opt)
    plt.scatter(binned_data["mid"], binned_data["dens"], s=2, c="blue")
    plt.xlim(x_bounds)
    plt.ylim(y_bounds)
    if with_legend:
      plt.legend(loc="upper right")
```

After the for-loop, plot the figure.

```
  #plt.show()
  plt.gcf().set_size_inches(6.5, 7.5)
  plt.savefig(filename)
  plt.close(plt.gcf())
```

Same but with loglog plots.

```
def loglog_graphs(data1, data2, data3,
    data4, data5, data6, var,
```

```
    list_F1, list_opt1,
    list_F2, list_opt2,
    list_F3, list_opt3,
    list_F4, list_opt4,
    list_F5, list_opt5,
    list_F6, list_opt6,
    x_bounds, y_bounds, *,
    filename, wgt=None, titles=[], with_legend=True):
  plt.close(plt.gcf())
  check_list(titles)
  if len(titles) > 0 and len(titles) < 6:
    raise ValueError("Please specify zero or all titles")
  elif len(titles) == 0:
    called_with_titles = False
  else:
    called_with_titles = True
```

Now loop through the panels, create pointers, and check list properties.

```
  for i in range(1, 7):
    list_F = eval("list_F" + str(i))
    list_opt = eval("list_opt" + str(i))
    check_list(list_F)
    check_list(list_opt)
    d = eval("data" + str(i))
    check_var(d, var)
    binned_data = bin.bin_data(d, var, wgt)
```

Create the figures.

```
    plt.subplot(3, 2, i)
    if called_with_titles:
      plt.title(titles[i - 1])
    for F, opt in zip(list_F, list_opt):
      x_vals = np.geomspace(*x_bounds, 200)
      y_vals = list(map(F, x_vals))
      plt.plot(x_vals, y_vals, **opt)
    plt.scatter(binned_data["mid"], binned_data["dens"], s=2, c="blue")
    plt.xlim(x_bounds)
    plt.ylim(y_bounds)
    plt.loglog()
    if with_legend:
      plt.legend(loc="lower left")
```

After the for-loop, plot the figure.

```
  #plt.show()
  plt.gcf().set_size_inches(6.5, 7.5)
  plt.savefig(filename)
  plt.close(plt.gcf())
```

The `four_graphs` is similar to the `lin_loglog_graphs`, etc. in that it accepts lists of information for creating multiple figures on the subplot. It is slightly lower-level than previous plotting functions in that we feed it actual data rather than a DataFrame and column. The arguments are

- `x_data` for the horizontal axes
- Several lists of things, one for each panel of the figure
    1. `list_y1`, `list_y2`, `list_y3`, `list_y4`—lists of data for the vertiacal axis
    2. `list_plot1`, `list_plot2`, `list_plot3`, `list_plot4`—lists of plotting functions. Most of these will be `plt.plot()`
    3. `list_opt1`, `list_opt2`, `list_opt3`, `list_opt4`—lists of options for plotting functions
- A filename for the figure
- An optional list of titles for the subgraphs

We begin with error checking and setting the `called_with_titles` boolean.

```
def four_graphs(
    list_x1, list_y1, list_plot1, list_opt1,
    list_x2, list_y2, list_plot2, list_opt2,
    list_x3, list_y3, list_plot3, list_opt3,
    list_x4, list_y4, list_plot4, list_opt4, *,
    filename, titles=[], extra_code=""):
  plt.close(plt.gcf())
  check_list(titles)
  if len(titles) > 0 and len(titles) < 4:
    raise ValueError("Please specify zero or all titles")
  elif len(titles) == 0:
    called_with_titles = False
  else:
    called_with_titles = True
```

We loop through the four panels. On each iteration, we first create new pointers to the lists for that iteration and check that they are actually lists. Then we loop through the three lists of y-axis data, plotting functions, and options. We call the plotting function with the x-axis data, corresponding y-axis data, and corresponding plotting options.

```
  for i in range(1,5):
    list_x = eval("list_x" + str(i))
    list_y = eval("list_y" + str(i))
    list_plot = eval("list_plot" + str(i))
    list_opt = eval("list_opt" + str(i))
    check_list(list_x)
    check_list(list_y)
    check_list(list_plot)
    check_list(list_opt)
```

Now plot the data for this subplot.

```
    plt.subplot(2, 2, i)
    for x, y, plot, opt in zip(list_x, list_y, list_plot, list_opt):
      plot(x, y, **opt)
    if called_with_titles:
```

```
      plt.title(titles[i-1])
```

Execute any extra code. Used to provide a hook into the function.

```
  exec(extra_code)
```

Then save the figure.

```
  #plt.show()
  plt.gcf().set_size_inches(6.5, 5)
  plt.savefig(filename)
  plt.close(plt.gcf())
```

A single-panel with more control. This function uses the same intervace as the `four_graphs` function.

```
def single_graph_ext(
    list_x, list_y, list_plot, list_opt, *,
    filename, title=None, extra_code=""):
  plt.close(plt.gcf())
  if isinstance(title, type(None)):
    called_with_title = False
  else:
    called_with_title = True
```

Now actually make the graph and execute the extra code.

```
  for x, y, plot, opt in zip(list_x, list_y, list_plot, list_opt):
    plot(x, y, **opt)
  if called_with_title:
    plt.title(title)
  exec(extra_code)
```

And save the figure.

```
  #plt.show()
  plt.gcf().set_size_inches(3.25, 2.5)
  plt.savefig(filename)
  plt.close(plt.gcf())
```

File VIII

`main.py`

This file contains the main code to carry out the estimations and write parameters to files. We have several tasks:

1. Load the data files that we made through `gen_files.py`. Make sure to drop the incomes of $0!
2. Estimate the parameters of all models for 2023, store the parameter estimates, and write them to files.
3. Estimate parameters, Fisk, inverse-gamma, and constant-shift-scale inverse-gamma distributions in all years.
4. Bootstrap the standard errors for the constant-shift-scale inverse-gamma parameter estimates.
5. Make figures.

- List `dist` of distribution strings. See table 6 for the distribution strings.
- List `years` of years. Both `dist` and `years` are for reference.
- Dictionary `data_min_max` of minimum and maximum cutoff values for data in each year. The keys are years, and the values are 2-tuples of boundary values.
- Dictionary `data` with years as keys and DataFrames as values.
- Dictionary `ecdfs` with years as keys and ecdfs as values.
- DataFrames of `parameters` containing the parameter estimates for the (shifted, constant-shift, and constant-shift-scale) inverse-gamma distributions. The columns are parameters, and the rows are years.
- Several `CSS_bootstrap` DataFrames containing synthetic parameter estimates and estimates of standard errors.
- `trim_data(`⟨*data*⟩`,`⟨*var*⟩`,`⟨*year*⟩`)`—trims the data according to the information in `data_min_max` and returns the trimmed DataFrame.

Several switches control which portion of this file runs. Make sure to set the appropriate switches to `True`.

```
##### Switches! #####
do_load_data = False
do_2023_short = False
do_2023_long = False
do_Fisk = False
do_InvG = False
do_CS_InvG = False    # <--- present for historical reasons
do_CSS_InvG = False
do_bootstrap = False
do_figures = False
do_test = False
```

```
#####################
if not (do_load_data
        or do_2023_short
        or do_2023_long
        or do_InvG
        or do_Fisk
        or do_CS_InvG
        or do_CSS_InvG
        or do_bootstrap
        or do_figures
        or do_test):
  print()
  print("Warning. This run of estimate_parameters will not do anything.")
  print("To enable data analysis, set one or more do_ booleans to True.")
  print()
  quit()
```

We begin by importing modules, creating the list of distribution strings, and creating the list of years. We need Pandas and the file `estimate_params.py`. See also Table 6. We have data for all years between 1967 and 2023 with the exception of 1970.

```
import bin
import bootstrap
import check_constants as cc
import estimate_parameters as est
import matplotlib as mpl
import matplotlib.pyplot as plt
import make_figures
import numpy as np
import pandas as pd
import scipy.special as spec
import time
dist = ["GB2", "Dagum", "Burr", "Fisk", "InvG", "CS_InvG", "CSS_InvG",
  "Davis", "LogN_P_cut", "LogN_P_mix"]
dist_names = {
  "GB2": "Generalized Beta, type II",
  "Dagum": "Dagum",
  "Burr": "Burr (Singh-Maddala)",
  "Fisk": "Fisk",
  "InvG": "Inverse Gamma",
  "CS_InvG": "Constant-Shift Inverse Gamma",
  "CSS_InvG": "Constant-Shift-Scale Inverse Gamma",
  "Davis": "Davis",
  "LogN_P_cut": "Log-Normal, Pareto Cutoff",
  "LogN_P_mix": "Log-Normal, Pareto Mixture"}
years = [i for i in range(1967, 2024) if (i != 1970)]
data = {}
ecdfs = {}
```

A function to print the time.

```
def the_time():
  print("The time is", time.asctime())
```

The maximum and minimum values are by inspection.

```
dat_max_min = {
  1967:(-4000,60000),
  1968:(-4000,90000),
  1969:(-4000,90000),
  1971:(-9000,90000),
  1972:(-9000,90000),
  1973:(-9000,90000),
  1974:(-9000,90000),
  1975:(-9000,90000),
  1976:(-9000,90000),
  1977:(-9000,100000),
  1978:(-9000,100000),
  1979:(-9000,100000),
  1980:(-9000,100000),
  1981:(-9000,100000),
  1982:(-9000,120000),
  1983:(-9000,120000),
  1984:(-9000,120000),
  1985:(-9000,200000),
  1986:(-9000,150000),
  1987:(-9000,200000),
  1988:(-9000,200000),
  1989:(-9000,200000),
  1990:(-9000,200000),
  1991:(-9000,200000),
  1992:(-9000,200000),
  1993:(-9000,200000),
  1994:(-9000,250000),
  1995:(-9000,250000),
  1996:(-8000,450000),
  1997:(-4000,450000),
  1998:(-9000,450000),
  1999:(-9000,450000),
  2000:(-9000,350000),
  2001:(-9000,450000),
  2002:(-11000,450000),
  2003:(-11000,550000),
  2004:(-11000,550000),
  2005:(-11000,650000),
  2006:(-11000,550000),
  2007:(-9000,650000),
  2008:(-9000,650000),
  2009:(-9000,550000),
```

```
  2010:(-9000,550000),
  2011:(-9000,1000000),
  2012:(-9000,1000000),
  2013:(-9000,1000000),
  2014:(-9000,1000000),
  2015:(-9000,1000000),
  2016:(-9000,1000000),
  2017:(-9000,1000000),
  2018:(-9000,1000000),
  2019:(-9000,1000000),
  2020:(-9000,1000000),
  2021:(-9000,1000000),
  2022:(-9000,1000000),
  2023:(-9000,1000000)}
```

The `parameters` dictionaries will hold a DataFrame to store the (shifted, constant-shift, and constant-shift-scale) inverse-gamma parameter estimates. We initialize them to be empty DataFrames. By specifying the column information at initialization, we can more easily concatenate parameter estimates later. See Tables 1 and 6 for discussion of the notation for different parameters. We store constants in the `CS_InvG` and `CSS_InvG` parameter tables for reference. The `CSS_bootstrap` DataFrames will store the synthetic parameter estimates from each bootstrapping function, and `CSS_bootstrap_se` stores the standard error estimates for each year and type of bootstrap.

```
Fisk_parameters = pd.DataFrame(index=years, columns=["alpha", "beta", "c"])
InvG_parameters = pd.DataFrame(index=years, columns=["alpha", "beta", "c"])
CS_InvG_parameters = pd.DataFrame({"alpha": [], "beta": [], "phi": []})
CSS_InvG_parameters = pd.DataFrame(index=years, columns=["alpha"])
CSS_bootstrap_naive = pd.DataFrame()
CSS_bootstrap_Jol = pd.DataFrame()
CSS_bootstrap_Jol_sim = pd.DataFrame()
CSS_bootstrap_se = pd.DataFrame(index=years)
```

The function `trim_data` accepts a year as its argument. It selects the corresponding data from `raw_data`, slices it, and returns the trimmed version.

```
def trim_data(data, var, year):
  if not isinstance(data, pd.DataFrame):
    raise TypeError("The data should be a DataFrame")
  if var not in data.columns:
    raise KeyError("Variable of interest is not a column in the data")
  bounds = dat_max_min[year]
  return data[(data[var] >= bounds[0]) &
              (data[var] <= bounds[1])].reset_index(drop=True).dropna()
```

Results

First we load all the data files, and then we calculate the paramter estimates for 2023. See Table 8 for approximate lengths of time each estimator takes. The time depends on the computer's processing speed and may vary between machines.

Table 8: Approximate Time to Perform Each Estimation

Model	Approximate Time to Estimate
`GB2`	Several hours
`Dagum`	Two minutes
`Burr`	Two minutes
`Fisk`	One minute
`InvG`	Less than one minute
`Davis`	One minute
`CS_InvG`	Less than one minute
`CSS_InvG`	Two minutes
`LogN_P_cut`	Two minutes
`LogN_P_mix`	One day

Load all the data files. We trim the ends according to `dat_max_min`. (Don't forget to remove the 0's!)

```
###############
## Load Data ##
###############
print()
the_time()
if do_load_data:
  print("Loading data:")
  for i in years:
    print("Year {0}".format(i))
    temp_data = pd.read_csv("data_{0}.txt".format(i), header=0)  # read file
    temp_data = trim_data(temp_data, "income", i)                # trim ends
    temp_data = temp_data[temp_data["income"] != 0]              # remove 0s
    temp_data = temp_data[temp_data["weight"] >= 0]              # weight > 0
    data[i] = temp_data
    ecdfs[i] = est.make_ecdf(data[i], "income", "weight")
else:
  print("Skipping loading data")
print()
```

Now find parameters for 2023 data. We handle the constant-scale and constant-shift-scale inverse-gamma distributions separately. If we loaded the 2023 data previously, we reference it from `data`. Otherwise, we load it now.

```
###############
## 2023 Data ##
###############
the_time()
if do_2023_short or do_2023_long:
  if 2023 in data:  # if loaded data_2023.txt
    data_2023 = data[2023]
```

```
  else:             # otherwise, load it now
    data_2023 = pd.read_csv("data_2023.txt", header=0)
    data_2023 = trim_data(data_2023, "income", 2023)
    data_2023 = data_2023[data_2023["income"] != 0]
    data_2023 = data_2023[data_2023["weight"] >= 0]
  print("Estimating parameters for 2023 data:")
```

First the case of the "short" distributions. (Where the time to run the estimation algorithm is short.) We open the file `2023_parameters_short.txt` for writing, loop through the models, and write in the file the parameter estimate for 2023 data under each model.

```
  if do_2023_short:
    f = open("2023_parameters_short.txt", "w")
    for model in dist:
      if model != "CS_InvG" and model != "CSS_InvG" and \
         model != "LogN_P_mix" and model != "GB2":
        print(model)
        f.write("---{0}---\n".format(model))
        temp = est.estimator[model](data_2023, "income", "weight")
        for i in temp:
          f.write("{0}: {1}\n".format(i, temp[i]))
    f.close()
  else:
    print("Skipping Dagum, Burr, Fisk, InvG, Davis, and LogN_P_cut")
```

The "long" distributions are those whose estimation algorithms take a long time, specifically `GB2` and `LogN_P_mix`.

```
  if do_2023_long:
    f = open("2023_parameters_long.txt", "w")
    for model in ["GB2", "LogN_P_mix"]:
      print(model)
      f.write("---{0}---\n".format(model))
      temp = est.estimator[model](data_2023, "income", "weight")
      for i in temp:
        f.write("{0}: {1}\n".format(i, temp[i]))
    f.close()
  else:
    print("Skipping GB2 and LogN_P_mix")
else:
  print("Skipping estimating 2023 data")
print()
```

Time for the Fisk distribution. We estimate the parameters for all years of data. First we check that `do_load_data` is true. Then we loop through `years` and estimate parameters for that year. We store them in `Fisk_parameters` DataFrame, and after the loop, we write the results to `Fisk_parameters.txt`.

```
##########
## Fisk ##
##########
the_time()
```

```
if do_Fisk:
  if not do_load_data:
    raise RuntimeError("Please load data before estimating Fisk")
  print("Estimating Fisk parameters:")
  for i in years:
    print("Year {0}".format(i))
    p = est.estimator["Fisk"](data[i], "income", "weight",
      ecdf=ecdfs[i], x="income", y="weight")["parameters"]
    Fisk_parameters.loc[i] = p
  Fisk_parameters.to_csv("Fisk_parameters.txt")
else:
  print("Skipping Fisk estimation")
print()
```

Now calculate the parameter estimates for all years of data using inverse gamma. Same approach as with Fisk distribution. First we check that `do_load_data` is true. Then we loop through `years`. For each year, we estimate inverse-gamma parameters for that year and store them in `InvG_parameters` DataFrame. After the loop, we write the results to `InvG_parameters.txt`.

```
###################
## Inverse Gamma ##
###################
the_time()
if do_InvG:
  if not do_load_data:
    raise RuntimeError("Please load data before estimating InvG")
  print("Estimating inverse-gamma parameters:")
  for i in years:
    print("Year {0}".format(i))
    p = est.estimator["InvG"](data[i], "income", "weight",
      ecdf=ecdfs[i], x="income", y="weight")["parameters"]
    InvG_parameters.loc[i] = p
  InvG_parameters.to_csv("InvG_parameters.txt")
else:
  print("Skipping inverse-gamma estimation")
print()
```

For historical reasons, we include code to estimate parameters for the constant-shift inverse-gamma model, where we have $c = \phi\beta$. We don't use this code for anything. It involves the formula

$$\phi = \sum_{i=1}^{n} \beta_i c_i \Big/ \sum_{i=1}^{n} \beta_i^2,$$

which minimizes the total squared distance between β and ϕc. This comes out to be approximately $\phi = -0.13$. The procedure is the same as for inverse-gamma distribution: loop through years, estimate the parameters for that year's data, and store the results in `CS_InvG_parameters` DataFrame.

```
##################################
```

```
## Constant-shift Inverse Gamma ##
##################################
the_time()
if do_CS_InvG:
  if not do_load_data:
    raise RuntimeError("Please load data before estimating CS_InvG")
  print("Estimating constant-scale inverse-gamma parameters:")
  phi = (InvG_parameters["beta"] * InvG_parameters["c"]).sum() / \
    (InvG_parameters["beta"] ** 2).sum()
  for i in years:
    print("Year {0}".format(i))
    p = est.estimator["CS_InvG"](data[i], phi, "income", "weight",
      ecdf=ecdfs[i], x="income", y="weight")["parameters"]
    CS_InvG_parameters = pd.concat([CS_InvG_parameters,
      pd.DataFrame({"alpha": p[0], "beta": p[1], "phi": phi}, index=[i])])
  CS_InvG_parameters.to_csv("CS_InvG_parameters.txt")
else:
  print("Skipping constant-shift inverse-gamma estimation")
print()
```

For the constant-shift-scale case, we need to calculate constants before we can estimate it. So we import `check_constants` and call the `main()` function from that file. We store the inverse-gamma parameter estimates for each year in `InvG_parameters` DataFrame.

```
########################################
## Constant-shift-scale Inverse Gamma ##
########################################
the_time()
if do_CSS_InvG:
  if not do_load_data:
    raise RuntimeError("Please load data before estimating CSS_InvG")
```

We call `main()` from `check_constants.py` and store the constants in `temp`.

```
  print("Checking constants")
  InvG_parameters = pd.read_csv("InvG_parameters.txt", header=0, index_col=0)
  temp = cc.main(InvG_parameters.index, InvG_parameters,
    "alpha", "beta", "c")["linear"]
  phi = temp["phi"]
  psi = [temp["psi0"], temp["psi1"], temp["psi2"]]
```

Then we write the values in a file.

```
  f = open("CSS_InvG_constants.txt", "w")
  for i in temp:
    f.write("{0},{1}\n".format(i, temp[i]))
  f.close()
```

Now we estimate parameters for the constant-shift-scale inverse-gamma distribution. Similar to `Fisk` and `InvG`, we store the end results in `CSS_InvG_parameters.txt`.

```
  print("Estimating constant-shift-scale inverse-gamma parameters:")
  for i in years:
    print("Year {0}".format(i))
```

```
    p = est.estimator["CSS_InvG"](data[i], i, phi, psi,
      InvG_parameters.loc[i, "alpha"], "income", "weight", ecdf=ecdfs[i],
      x="income", y="weight")["parameters"]
    CSS_InvG_parameters.loc[i] = p
  CSS_InvG_parameters.to_csv("CSS_InvG_prop_parameters.txt")
else:
  print("Skipping constant-shift-scale inverse-gamma distribution")
print()
```

Finally, it is bootstrapping time. First, we read in the inverse-gamma parameter estimates and the variables from `check_constants.py`.

```
################
## bootstraps ##
################
the_time()
if do_bootstrap:
  if not do_load_data:
    raise RuntimeError("Please load data before bootstrapping")
  print("Bootstrapping standard errors:")
  InvG_parameters = pd.read_csv("InvG_parameters.txt", header=0, index_col=0)
  f = open("CSS_InvG_constants.txt")
  for line in f:
    temp = line[:-1].split(",")
    exec("{0} = {1}".format(temp[0], temp[1]))
  f.close()
```

For each year, we take the data for that year and feed it into one of the three bootstrapping functions. We will store the `parameter` column values in a DataFrame. The `CSS_pos_args` list and `CSS_named_args` dictionary pack common arguments of the bootstrap functions for convenience. We manually set the index of each DataFrame to be a list of integers if we want to be able to easily add the synthetic parameter estimates as columns. If we change the number of estimates in the bootstrap, we will need to change the index to a different range or list.

```
  CSS_bootstrap_naive.index = range(100)
  CSS_bootstrap_Jol.index = range(100)
  CSS_bootstrap_Jol_sim.index = range(100)
  for i in years:
    CSS_pos_args = [est.estimator["CSS_InvG"], data[i], "income", "weight"]
    CSS_named_args = {"t": i, "phi": phi, "psi": [psi0, psi1],
      "a0": InvG_parameters.loc[i,"alpha"]}
    print("Year {0}".format(i))
```

Naive bootstrap is first.

```
    temp = pd.DataFrame(bootstrap.bootstrap_naive(
        *CSS_pos_args,
        **CSS_named_args)
      )["parameters"]
    temp = temp.explode().astype(np.float64)
    CSS_bootstrap_naive[i] = temp
```

Jolliffe (2003) stratification and clustering is next.

```
    temp = pd.DataFrame(bootstrap.bootstrap_Jol(
        *CSS_pos_args,
        "region", "household",
        **CSS_named_args)
      )["parameters"]
    temp = temp.explode().astype(np.float64)
    CSS_bootstrap_Jol[i] = temp
```

Stratification and simplified clustering based on Jolliffe (2003) is third.

```
    temp = pd.DataFrame(bootstrap.bootstrap_Jol_sim(
        *CSS_pos_args,
        "region",
        **CSS_named_args)
      )["parameters"]
    temp = temp.explode().astype(np.float64)
    CSS_bootstrap_Jol_sim[i] = temp
```

After creating the DataFrames, we write them to files, find the normalized interquartile range, and store that in `CSS_bootstrap_se`. After the loop, we write `CSS_bootstrap_se` to a file.

```
  for i in ["naive", "Jol", "Jol_sim"]:
    temp = eval("CSS_bootstrap_" + i)
    temp.to_csv("CSS_bootstrap_" + i + ".txt", index=False)
    iqrs = 0.7413 * (temp.quantile(0.75, numeric_only=True) -
                     temp.quantile(0.25, numeric_only=True))
    CSS_bootstrap_se[i] = iqrs
  CSS_bootstrap_se.to_csv("CSS_bootstrap_se.txt")
  print("se's are:")
  print(CSS_bootstrap_se)
else:
  print("Skipping bootstrapping")
print()
```

Now figures. We will make several:

1. `CSS_lin_densities.pdf`—`CSS_InvG` densities in 1967, 1995, and 2023, plotted on linear and log-log plots with linear relationship between parameters
2. `CSS_prop_densities.pdf`—`CSS_InvG` densities in 1967, 1995, and 2023, plotted on linear and log-log plots with proportional relationship between parameters
3. `InvG_densities.pdf`—`InvG` densities in 1967, 1995, and 2023, plotted on linear and log-log plots
4. `Fisk_densities.pdf`—`Fisk` densities in 1967, 1995, and 2023, plotted on linear and log-log plots
5. `InvG_parameter_graphs.pdf`—`InvG` parameter estimates in three panels and a fourth panel for normalized parameter estimates
6. `InvG_parameter_regression.pdf`—One panel with differences of normalized parameters, one panel with quotients of normalized parameters, and two panels with β and c predictions.

7. `Fisk_parameters_normalized.pdf`—normalized Fisk parameter estimates
8. `CSS_InvG_parameters_graph.pdf`—single panel; `CSS_InvG` α estimates
9. `comparison_linear_graphs.pdf`—linear graphs for GBII, Burr, Dagum, Davis, LogN-P mix/cut using 2023 data
10. `comparison_loglog_graphs.pdf`—loglog graphs for GBII, Burr, Dagum, Davis, LogN-P mix/cut using 2023 data
11. `gini_compare_graphs.pdf`

First, we load the `InvG`, `Fisk`, and `CSS_InvG` parameters.

```
#############
## figures ##
#############
the_time()
if do_figures:
  if not do_load_data:
    raise RuntimeError("Please load data before making figures")
  print("Making figures")
  InvG_parameters = pd.read_csv("InvG_parameters.txt", header=0, index_col=0)
  CSS_InvG_lin_parameters = \
    pd.read_csv("CSS_InvG_lin_parameters.txt", header=0, index_col=0)
  CSS_InvG_prop_parameters = \
    pd.read_csv("CSS_InvG_prop_parameters.txt", header=0, index_col=0)
  Fisk_parameters = pd.read_csv("Fisk_parameters.txt", header=0, index_col=0)
```

Again, we load the `CSS_InvG` constants

```
  f = open("CSS_InvG_lin_constants.txt")
  for line in f:
    temp = line[:-1].split(",")
    exec("{0}_lin = {1}".format(temp[0], temp[1]))
  f.close()
  f = open("CSS_InvG_prop_constants.txt")
  for line in f:
    temp = line[:-1].split(",")
    exec("{0}_prop = {1}".format(temp[0], temp[1]))
  f.close()
```

Figure 1: `CSS_lin_densities.pdf`. We begin with the plots of empirical and model density for past years of data. To keep things simple, we use data from 1967, 1995, and 2023. We define density functions for each year. To make it simple to change years later, we save the years in variables `year1`, `year2`, and `year3`.

```
  print("Making CSS_lin_densities.pdf")
  year1, year2, year3 = [1967, 1995, 2023]
  for i in [year1, year2, year3]:
    exec("""def F_{0}(x):
      return est.density["CSS_InvG"](x, {0}, phi_lin,
        [psi0_lin, psi1_lin, psi2_lin],
        CSS_InvG_lin_parameters.loc[{0}, "alpha"])""".format(i))
    exec("""c_{0} = psi0_lin + psi1_lin * {0} + \
```

```
    psi2_lin * CSS_InvG_lin_parameters.loc[{0}, "alpha"]""".format(i))
make_figures.lin_loglog_graphs(
  data[year1], data[year2], data[year3], "income",
  [eval("F_{0}".format(year1))], [eval("c_{0}".format(year1))],
                                 [{"c": "black", "lw": 0.7}],
  [eval("F_{0}".format(year2))], [eval("c_{0}".format(year2))],
                                 [{"c": "black", "lw": 0.7}],
  [eval("F_{0}".format(year3))], [eval("c_{0}".format(year3))],
                                 [{"c": "black", "lw": 0.7}],
  [-20000,100000], [-0.05e-4,2.05e-4], [1000,1100000], [1e-10,2e-4],
  filename="CSS_lin_densities", wgt="weight",
  titles=["{0} Data (Linear Scale)".format(year1),
          "{0} Data (Log Scale)".format(year1),
          "{0} Data (Linear Scale)".format(year2),
          "{0} Data (Log Scale)".format(year2),
          "{0} Data (Linear Scale)".format(year3),
          "{0} Data (Log Scale)".format(year3)])
```

Figure 2: `CSS_prop_densities.pdf`. Same thing except with proportional relationship imposed on parameters.

```
print("Making CSS_prop_densities.pdf")
year1, year2, year3 = [1967, 1995, 2023]
for i in [year1, year2, year3]:
  exec("""def F_{0}(x):
    return est.density["CSS_InvG_prop"](x, {0}, phi_prop,
      [psi0_prop, psi1_prop],
      CSS_InvG_prop_parameters.loc[{0}, "alpha"])""".format(i))
  exec("""c_{0} = CSS_InvG_prop_parameters.loc[{0}, "alpha"] * \
    (psi0_prop + psi1_prop * {0})""".format(i))
make_figures.lin_loglog_graphs(
  data[year1], data[year2], data[year3], "income",
  [eval("F_{0}".format(year1))], [eval("c_{0}".format(year1))],
                                 [{"c": "black", "lw": 0.7}],
  [eval("F_{0}".format(year2))], [eval("c_{0}".format(year2))],
                                 [{"c": "black", "lw": 0.7}],
  [eval("F_{0}".format(year3))], [eval("c_{0}".format(year3))],
                                 [{"c": "black", "lw": 0.7}],
  [-20000,100000], [-0.05e-4,2.05e-4], [1000,1100000], [1e-10,2e-4],
  filename="CSS_prop_densities", wgt="weight",
  titles=["{0} Data (Linear Scale)".format(year1),
          "{0} Data (Log Scale)".format(year1),
          "{0} Data (Linear Scale)".format(year2),
          "{0} Data (Log Scale)".format(year2),
          "{0} Data (Linear Scale)".format(year3),
          "{0} Data (Log Scale)".format(year3)])
```

Figure 3: `InvG_densities.pdf`. Same thing except with inverse-gamma.

```
print("Making InvG_densities.pdf")
year1, year2, year3 = [1967, 1995, 2023]
```

```
for i in [year1, year2, year3]:
  exec("""def F_{0}(x):
    return est.density["InvG"](x, InvG_parameters.loc[{0}])""".format(i))
  exec("""c_{0} = InvG_parameters.loc[{0}, "c"]""".format(i))
make_figures.lin_loglog_graphs(
  data[year1], data[year2], data[year3], "income",
  [eval("F_{0}".format(year1))], [eval("c_{0}".format(year1))],
                                 [{"c": "black", "lw": 0.7}],
  [eval("F_{0}".format(year2))], [eval("c_{0}".format(year2))],
                                 [{"c": "black", "lw": 0.7}],
  [eval("F_{0}".format(year3))], [eval("c_{0}".format(year3))],
                                 [{"c": "black", "lw": 0.7}],
  [-20000,100000], [-0.05e-4,2.05e-4], [1000,1100000], [1e-10,2e-4],
  filename="InvG_densities", wgt="weight",
  titles=["{0} Data (Linear Scale)".format(year1),
          "{0} Data (Log Scale)".format(year1),
          "{0} Data (Linear Scale)".format(year2),
          "{0} Data (Log Scale)".format(year2),
          "{0} Data (Linear Scale)".format(year3),
          "{0} Data (Log Scale)".format(year3)])
```

Figure 4: `Fisk_densities.pdf`. Same thing except with Fisk densities.

```
print("Making Fisk_densities.pdf")
year1, year2, year3 = [1967, 1995, 2023]
for i in [year1, year2, year3]:
  exec("""def F_{0}(x):
    return est.density["Fisk"](x, Fisk_parameters.loc[{0}])""".format(i))
  exec("""c_{0} = Fisk_parameters.loc[{0}, "c"]""".format(i))
make_figures.lin_loglog_graphs(
  data[year1], data[year2], data[year3], "income",
  [eval("F_{0}".format(year1))], [eval("c_{0}".format(year1))],
                                 [{"c": "black", "lw": 0.7}],
  [eval("F_{0}".format(year2))], [eval("c_{0}".format(year2))],
                                 [{"c": "black", "lw": 0.7}],
  [eval("F_{0}".format(year3))], [eval("c_{0}".format(year3))],
                                 [{"c": "black", "lw": 0.7}],
  [-20000,100000], [-0.05e-4,2.05e-4], [1000,1100000], [1e-10,2e-4],
  filename="Fisk_densities", wgt="weight",
  titles=["{0} Data (Linear Scale)".format(year1),
          "{0} Data (Log Scale)".format(year1),
          "{0} Data (Linear Scale)".format(year2),
          "{0} Data (Log Scale)".format(year2),
          "{0} Data (Linear Scale)".format(year3),
          "{0} Data (Log Scale)".format(year3)])
```

Figure 5: `InvG_parameter_graphs.pdf`—The next set of graphs shows the graphs of inverse-gamma parameters. We have four panels: one for each parameter and one with the normalized parameters. Before we create the figure, we need a few more Series. The `norm` Series

are inverse-gamma parameters normalized by the sum of that parameter across all years.

```
  print("Making InvG_parameter_graphs.pdf")
  norm_alpha = InvG_parameters["alpha"] / InvG_parameters["alpha"].sum()
  norm_beta = InvG_parameters["beta"] / InvG_parameters["beta"].sum()
  norm_c = InvG_parameters["c"] / InvG_parameters["c"].sum()
  make_figures.four_graphs(
    [years], [InvG_parameters["alpha"]], [plt.plot],
      [{"c": "black", "lw": 0.5}],
    [years], [InvG_parameters["beta"]], [plt.plot],
      [{"c": "black", "lw": 0.5}],
    [years], [InvG_parameters["c"]], [plt.plot],
      [{"c": "black", "lw": 0.5}],
    [*[years]*3],
      [norm_alpha, norm_beta, norm_c], [plt.plot, plt.plot, plt.plot],
      [{"c": "blue", "lw": 0.7, "ls": "--", "label": "Shape"},
       {"c": "black", "lw": 0.5, "label": "Scale"},
       {"c": "red", "lw": 1.0, "ls": ":", "label": "Shift"}],
    filename="InvG_parameter_graphs", titles=["Shape Parameter",
      "Scale Parameter", "Shift Parameter", "Normalized Parameters"],
      extra_code = \
"""plt.subplot(2,2,4)
plt.legend()""")
```

Figure 6: `InvG_parameter_regression.pdf`—The next figure is a plot illustrating the relationship between the parameters. The upper row will show the difference and quotient between normalized shape and the other two parameters. The lower row will comparing actual and predicted values for β and c. The `beta_hat` and `c_hat` Series are the predicted values from the linear regression of β and c on α and the year.

```
  print("Making InvG_parameter_regression.pdf")
  beta_hat = (psi0_lin + psi1_lin * InvG_parameters.index + \
    psi2_lin * InvG_parameters["alpha"]) / phi_lin
  c_hat = psi0_lin + psi1_lin * InvG_parameters.index + \
    psi2_lin * InvG_parameters["alpha"]
  norm_beta_hat = beta_hat / beta_hat.sum()
  norm_c_hat = c_hat / c_hat.sum()
```

We make numpy arrays out of the differences and quotients of the normalized parameters.

```
  diff_beta_alpha = (norm_beta - norm_alpha).to_numpy()
  diff_c_alpha = (norm_c - norm_alpha).to_numpy()
  quot_beta_alpha = (norm_beta / norm_alpha).to_numpy()
  quot_c_alpha = (norm_c / norm_alpha).to_numpy()
```

Now two linear regressions for the combinations of normalized parameters. We are approximating the quotient and differences as linear functions of time. First the differences.

```
  year_cons = pd.DataFrame({"cons": 1, "year": years}).to_numpy()
  reg_diff = np.linalg.inv(np.transpose(year_cons) @ year_cons) @ \
    np.transpose(year_cons) @ (0.5 * (diff_beta_alpha + diff_c_alpha))
  reg_diff_vals = reg_diff[0] + reg_diff[1] * InvG_parameters.index
```

And quotient.

```
reg_quot = np.linalg.inv(np.transpose(year_cons) @ year_cons) @ \
  np.transpose(year_cons) @ (0.5 * (quot_beta_alpha + quot_c_alpha))
reg_quot_vals = reg_quot[0] + reg_quot[1] * InvG_parameters.index
```

Now make the figure. Every list of x-values to use will be the same, so we put copies of `years` in a list for each of those function arguments.

```
from matplotlib.markers import MarkerStyle
make_figures.four_graphs(
  [*[years]*3],
  [diff_beta_alpha, diff_c_alpha, reg_diff_vals],
    [plt.scatter, plt.scatter, plt.plot],
    [{"c": "blue", "s": 2.0, "label": "$\\bar\\beta_t-\\bar\\alpha_t$"},
     {"c": "red", "s": 10.0, "label": "$\\bar c_t-\\bar\\alpha_t$",
       "marker": "2"},
     {"c": "black", "lw": 0.7, "ls": "--", "label": "Trendline"}],
  [*[years]*3],
  [quot_beta_alpha, quot_c_alpha, reg_quot_vals],
    [plt.scatter, plt.scatter, plt.plot],
    [{"c": "blue", "s": 2.0, "label": "$\\bar\\beta_t/\\bar\\alpha_t$"},
     {"c": "red", "s": 10.0, "label": "$\\bar c_t/\\bar\\alpha_t$",
       "marker": "2"},
     {"c": "black", "lw": 0.7, "ls": "--", "label": "Trendline"}],
  [*[years]*2],
  [InvG_parameters["beta"], beta_hat], [plt.plot, plt.plot],
    [{"c": "black", "lw": 0.5, "label": "Observed"},
     {"c": "blue", "lw": 0.7, "ls": "--", "label": "Predicted"}],
  [*[years]*2],
  [InvG_parameters["c"], c_hat], [plt.plot, plt.plot],
    [{"c": "black", "lw": 0.5, "label": "Observed"},
     {"c": "blue", "lw": 0.7, "ls": "--", "label": "Predicted"}],
  filename="InvG_parameter_regression",
  titles=["(Normalized) Differences",
    "(Normalized) Quotients", "Predicted Scale",
    "Predicted Shift"],
```

The extra code for this function call will be a loop through subplots, and on each iteration, we specify adding a legend for that subplot.

```
    extra_code= \
"""for i in range(1, 5):
  plt.subplot(2, 2, i)
  plt.legend()""")
```

Figure 7: `Fisk_parameters_normalized.pdf`—a graph of normalized Fisk parameters. We start by creating series of normalized parameters, and then we put them in a figure.

```
print("Making Fisk_parameters_normalized.pdf")
Fisk_alpha_norm = Fisk_parameters["alpha"] / Fisk_parameters["alpha"].sum()
Fisk_beta_norm = Fisk_parameters["beta"] / Fisk_parameters["beta"].sum()
Fisk_c_norm = Fisk_parameters["c"] / Fisk_parameters["c"].sum()
```

```
make_figures.single_graph_ext(
 [*[years]*3],
  [Fisk_alpha_norm, Fisk_beta_norm, Fisk_c_norm],
  [plt.plot, plt.plot, plt.plot],
  [{"c": "blue", "lw": 0.7, "ls": "--", "label": "Shape"},
   {"c": "black", "lw": 0.5, "label": "Scale"},
   {"c": "red", "lw": 1.0, "ls": ":", "label": "Shift"}],
 filename="Fisk_parameters_normalized", title="Normalized Fisk Parameters",
  extra_code = "plt.legend()")
```

Figure 8: `CSS_InvG_parameters_graph.pdf`—A graph of the constant-shift-scale inverse-gamma parameter estimates.

```
print("Making CSS_InvG_parameters_graph.pdf")
make_figures.single_graph(CSS_InvG_lin_parameters, "alpha",
 filename="CSS_InvG_parameters_graph", title="Parameter Estimates")
```

Figure 9: `comparison_linear_graphs.pdf`—Now we make graphs for the different densities in 2023. First we load the files with parameter estimates saved as dictionaries. The dictionary `parameters` will save the parameters for each distribution.

```
print("comparison_linear_graphs.pdf")
parameters = {}
def add_from_parameter_file(filename, parameter_dict):
 f = open(filename)
 curr_model = ""
 for line in f:
  if line[0] == "-":
   curr_model = line[3:-4]
  elif line[0] == "f":
   pass
  elif line[0] == "p":
   temp = line[13:-2].split(", ")
   parameter_dict[curr_model] = list(map(float, temp))
 f.close()
 return parameter_dict
parameters = add_from_parameter_file("2023_parameters_short.txt", parameters)
parameters = add_from_parameter_file("2023_parameters_long.txt", parameters)
```

Now we create the density functions and constants.

```
for i in ["GB2", "Dagum", "Burr", "Davis", "LogN_P_cut",
     "LogN_P_mix"]:
 exec("""def F_{0}(x):
  return est.density['{0}'](x, parameters['{0}'])""".format(i))
 exec("c_{0} = parameters['{0}'][-1]".format(i))
```

Constant-shift-scale inverse-gamma density.

```
def F_CSS_InvG(x):
 return est.density["CSS_InvG"](x, 2023, phi_lin,
  [psi0_lin, psi1_lin, psi2_lin],
  CSS_InvG_lin_parameters.loc[2023, "alpha"])
c_CSS_InvG = psi0_lin + psi1_lin * 2023 + \
```

```
    psi2_lin * CSS_InvG_lin_parameters.loc[2023, "alpha"]
```

Make the graphs. We create two sets of comparison graphs where we put the inverse-gamma density on each plot and one alternative model on each plot. The first set of graphs will be linear scale, and the second one will be loglog scale.

```
make_figures.lin_graphs(*[data[2023]]*6, "income",
  [F_CSS_InvG, F_GB2], [c_CSS_InvG, c_GB2],
    [{"c": "black", "lw": 0.5, "label": "Inverse-Gamma"},
     {"c": "blue", "lw": 0.7, "ls": "--", "label": "Gen Beta II"}],
  [F_CSS_InvG, F_Dagum], [c_CSS_InvG, c_Dagum],
    [{"c": "black", "lw": 0.5, "label": "Inverse-Gamma"},
     {"c": "blue", "lw": 0.7, "ls": "--", "label": "Dagum"}],
  [F_CSS_InvG, F_Burr], [c_CSS_InvG, c_Burr],
    [{"c": "black", "lw": 0.5, "label": "Inverse-Gamma"},
     {"c": "blue", "lw": 0.7, "ls": "--", "label": "Burr"}],
  [F_CSS_InvG, F_Davis], [c_CSS_InvG, c_Davis],
    [{"c": "black", "lw": 0.5, "label": "Inverse-Gamma"},
     {"c": "blue", "lw": 0.7, "ls": "--", "label": "Davis"}],
  [F_CSS_InvG, F_LogN_P_cut], [c_CSS_InvG, c_LogN_P_cut],
    [{"c": "black", "lw": 0.5, "label": "Inverse-Gamma"},
     {"c": "blue", "lw": 0.7, "ls": "--", "label": "Cutoff"}],
  [F_CSS_InvG, F_LogN_P_mix], [c_CSS_InvG, c_LogN_P_mix],
    [{"c": "black", "lw": 0.5, "label": "Inverse-Gamma"},
     {"c": "blue", "lw": 0.7, "ls": "--", "label": "Mixture"}],
  [-20000,100000], [-0.05e-5,2.05e-5],
  filename="comparison_linear_graphs", wgt="weight",
  titles=["Gen Beta II", "Dagum", "Burr", "Davis",
    "Log-Normal/Pareto Cutoff", "Log-Normal/Pareto Mix"])
```

Figure 10: `comparison_loglog_graphs.pdf`—Same thing with loglog scaling.

```
print("Making comparison_loglog_graphs.pdf")
make_figures.loglog_graphs(*[data[2023]]*6, "income",
  [F_CSS_InvG, F_GB2],
    [{"c": "black", "lw": 0.5, "label": "Inverse-Gamma"},
     {"c": "blue", "lw": 0.7, "ls": "--", "label": "Gen Beta II"}],
  [F_CSS_InvG, F_Dagum],
    [{"c": "black", "lw": 0.5, "label": "Inverse-Gamma"},
     {"c": "blue", "lw": 0.7, "ls": "--", "label": "Dagum"}],
  [F_CSS_InvG, F_Burr],
    [{"c": "black", "lw": 0.5, "label": "Inverse-Gamma"},
     {"c": "blue", "lw": 0.7, "ls": "--", "label": "Burr"}],
  [F_CSS_InvG, F_Davis],
    [{"c": "black", "lw": 0.5, "label": "Inverse-Gamma"},
     {"c": "blue", "lw": 0.7, "ls": "--", "label": "Davis"}],
  [F_CSS_InvG, F_LogN_P_cut],
    [{"c": "black", "lw": 0.5, "label": "Inverse-Gamma"},
     {"c": "blue", "lw": 0.7, "ls": "--", "label": "Cutoff"}],
  [F_CSS_InvG, F_LogN_P_mix],
    [{"c": "black", "lw": 0.5, "label": "Inverse-Gamma"},
```

```
      {"c": "blue", "lw": 0.7, "ls": "--", "label": "Mixture"}],
    [1000,1100000], [1e-10,2e-4],
    filename="comparison_loglog_graphs", wgt="weight",
    titles=["Gen Beta II", "Dagum", "Burr", "Davis",
      "Log-Normal/Pareto Cutoff", "Log-Normal/Pareto Mix"])
```

Figure 11: `gini_compare_graphs.pdf`—For fun we make a final graphic about the Gini coefficient, which is

$$\frac{\beta}{\beta + (\alpha - 1)c} \frac{\Gamma(\alpha - \frac{1}{2})}{\sqrt{\pi}\Gamma(\alpha)} = \frac{1}{1 + (\alpha - 1)\phi} \frac{\Gamma(\alpha - \frac{1}{2})}{\sqrt{\pi}\Gamma(\alpha)}$$

We will use `four_graphs()` again. The `gini` Series contains values of the Gini coefficient in each year. The `gini_unshift` is the value of the shape parameter in the unshifted Gini coefficient formula. The third and fourth graphs show the Gini coefficient for different values of α.

```
print("Making gini_compare_graphs.pdf")
gini = 1 / (1 + (CSS_InvG_lin_parameters["alpha"] - 1) * phi_lin) * \
  est.G(CSS_InvG_lin_parameters["alpha"] - 0.5) / (np.sqrt(np.pi) *
  est.G(CSS_InvG_lin_parameters["alpha"]))
gini_unshift = \
  est.G(CSS_InvG_lin_parameters["alpha"] - 0.5) / (np.sqrt(np.pi) *
  est.G(CSS_InvG_lin_parameters["alpha"]))
make_figures.single_graph(pd.DataFrame({"gini": gini}), "gini",
  filename="gini_graph")
make_figures.single_graph(pd.DataFrame({"gini": gini_unshift}), "gini",
  filename="gini_unshift_graph")
```

Now make the Gini coefficient for different values of α. The graph with Gini coefficient of unshifted inverse-gamma distribution is straightforward. For shifted inverse-gamma distribution, we have a minimum value around 3. Setting the derivative equal to 0 gives us

$$\begin{aligned}
y &= \frac{1}{1 + (x - 1)\phi} \frac{\Gamma\left(x - \frac{1}{2}\right)}{\sqrt{\pi}\Gamma(x)} \\
\log y &= -\log(1 + (x - 1)\phi) + \log\Gamma\left(x - \frac{1}{2}\right) - \log\Gamma(x) - \log\sqrt{\pi} \\
\frac{y'}{y} &= -\frac{\phi}{1 + (x - 1)\phi} + \psi\left(x - \frac{1}{2}\right) - \psi(x) = 0
\end{aligned}$$

We can solve this equation numerically for α. The `right_singularity` variable is the x-value where the Gini coefficient has a singularity because the denominator blows up, in other words $1 + (\alpha - 1)\phi \to 0$. The `alpha_vals`, `gini_vals_shift`, and `gini_vals_unshift` lists are lists of x and y-values for plotting the Gini coefficient.

```
right_singularity = 1 - 1 / phi_lin
alpha_vals = np.linspace(0.6, right_singularity - 0.2, 200)
gini_vals_shift = 1 / (1 + (alpha_vals - 1) * phi_lin) * \
  est.G(alpha_vals - 0.5) / (np.sqrt(np.pi) *
  est.G(alpha_vals))
```

```
gini_vals_unshift = \
  est.G(alpha_vals - 0.5) / (np.sqrt(np.pi) *
  est.G(alpha_vals))
```

The `D_gini` function is the log-derivative y'/y of the Gini coefficient for shifted inverse-gamma distribution. As noted above, we can find the alpha corresponding to minimum Gini coefficient when `D_gini` $= 0$. We set `min_alpha` and `max_alpha` to be the minimum and maximum values of α in the data.

```
def D_gini(x):
  return est.psi(x - 0.5) - est.psi(x) - phi_lin / (1 + (x - 1) * phi_lin)
min_gini = est.root(D_gini, bracket=[1, right_singularity - 0.2]).root
min_alpha = CSS_InvG_lin_parameters["alpha"].min()
max_alpha = CSS_InvG_lin_parameters["alpha"].max()
```

Now make the figure. We have several graphs on each subplot:

1. Shape parameter and horizontal lines for minimum and maximum values
2. Gini coefficient calculated from parameter estimates
3. Gini coefficient as a function of α under a shifted inverse-gamma distribution as well as vertical lines at the minimum and maximum values of α observed in the data
4. Gini coefficient as a function of α under an unshifted inverse-gamma distribution as well as vertical lines at the minimum and maximum values of α observed in the data

The file will be `gini_compare_graphs.pdf`.

```
make_figures.four_graphs(
  [years, (years[0], years[-1]), (years[0], years[-1])],
    [CSS_InvG_lin_parameters["alpha"],
      (min_alpha, min_alpha), (max_alpha, max_alpha)],
    [plt.plot, plt.plot, plt.plot],
    [{"c": "black", "lw": 0.5},
     {"c": "red", "lw": 0.7, "ls": "--"},
     {"c": "blue", "lw": 0.7, "ls": "--"}],
  [years], [gini], [plt.plot],
    [{"c": "black", "lw": 0.5}],
  [alpha_vals, (min_alpha, min_alpha), (max_alpha, max_alpha)],
    [gini_vals_shift, (0,2), (0,2)], [plt.plot, plt.plot, plt.plot],
    [{"c": "black", "lw": 0.5},
     {"c": "red", "lw": 0.7, "ls": "--"},
     {"c": "blue", "lw": 0.7, "ls": "--"}],
  [alpha_vals, (min_alpha, min_alpha), (max_alpha, max_alpha)],
    [gini_vals_unshift, (0,2), (0,2)], [plt.plot, plt.plot, plt.plot],
    [{"c": "black", "lw": 0.5},
     {"c": "red", "lw": 0.7, "ls": "--"},
     {"c": "blue", "lw": 0.7, "ls": "--"}],
  filename="gini_compare_graphs",
  titles=["Shape Parameter",
    "Gini Coefficient from Parameters",
    "Gini Coefficient Function (With Shift)",
    "Gini Coefficient Function (No Shift)"],
```

The `extra_code` for this function sets the vertical limits for the third and fourth subplots and adds a light blue rectangular patch to the third graph to denote the portion of the domain where the Gini coefficient is increasing.

```
    extra_code= \
"""plt.subplot(2, 2, 3)
plt.ylim([0,2])
plt.gca().add_patch(mpl.patches.Rectangle([{min_gini},0],
  {max_alpha} - {min_gini}, 2, color="aliceblue"))
plt.subplot(2, 2, 4)
plt.ylim([0,2])""".format(min_gini=min_gini, max_alpha=alpha_vals[-1]))
```

Done with figures.

```
else:
  print("Skipping figures")
print()
```

Random code to test things.

```
## Code for testing
if do_test:
  the_time()
  print("Testing...")

  # This section is intentionally blank

  print()
```

End of code!

```
the_time()
print("End of main.py")
print()
```