census2011

March 29, 2024

1 CS2006 Python Practical 2

Tutor: Stephen Linton
2024-3-29
Requirement Breakdown We completed all requirements in a concise, effective, and organized manner. In the completion of this project, we have gained experience using Python for data analysis. * **Student A - 230015014** - Data refinement - Unit testing for data refinement - venv - Optimization/performance analysis * **Student B - 220024634** - Data refinement - Data analysis - Basic data visualisation - Finding records w/ groupby - 3D plots - map by region plots * **Student C - 220010065** - Querying the data - ipywidgets - nbconvert

In this practical, we are given a dataset containing a sample of 1% of people in the 2011 Census database for England and Wales. We are asked to analyze this dataset using programs written by us and using components of the Python ecosystem. Our project thoroughly refines the data, describes it accurately, is repeatable, replicable, reproducible and resuable. All analytics can be executed with any data set of similar structure.

Run through our notebook from top to bottom, starting by importing and reading the data.

Please be patient if you run all cells as interactive plots will not respond until all cells have finished running

[1]:
```
import pandas as pd
import sys
import os

sys.path.append("../code")

import consistency
```

1.1 Refining the dataset

We start with exploring the content of the raw data.

[2]:
```
df = pd.read_csv("../data/census2011.csv")
df
```

[2]:
```
        Person ID     Region Residence Type  Family Composition  \
0         7394816  E12000001              H                   2
1         7394745  E12000001              H                   5
```

2	7395066	E12000001	H	3
3	7395329	E12000001	H	3
4	7394712	E12000001	H	3
…	…	…	…	…
569736	7946020	W92000004	H	1
569737	7944310	W92000004	H	3
569738	7945374	W92000004	H	3
569739	7944768	W92000004	H	1
569740	7944959	W92000004	H	2

	Population Base	Sex	Age	Marital Status	Student	Country of Birth \
0	1	2	6	2	2	1
1	1	1	4	1	2	1
2	1	2	4	1	2	1
3	1	2	2	1	2	1
4	1	1	5	4	2	1
…	…	…	…	…	…	…
569736	1	1	5	1	2	1
569737	1	1	3	1	2	1
569738	1	1	1	1	1	1
569739	1	2	8	5	2	1
569740	1	2	2	2	2	1

	Health	Ethnic Group	Religion	Economic Activity	Occupation \
0	2	1	2	5	8
1	1	1	2	1	8
2	1	1	1	1	6
3	2	1	2	1	7
4	1	1	2	1	1
…	…	…	…	…	…
569736	4	1	9	1	8
569737	2	1	1	1	7
569738	1	1	2	-9	-9
569739	3	1	9	5	9
569740	2	1	1	1	7

	Industry	Hours worked per week	Approximated Social Grade
0	2	-9	4
1	6	4	3
2	11	3	4
3	7	3	2
4	4	3	2
…	…	…	…
569736	8	3	3
569737	4	3	4
569738	-9	-9	-9
569739	2	-9	4

```
569740        4                       1                           4

[569741 rows x 18 columns]
```

We will first refine the data in order to handle inconsistencies before further analysis.

We consider inconsistencies to be:
* 0-15 age range and any form of marital status other than single * Any mismatched 'no codes' to do with student status * Anyone marked as working and in very bad health * Anyone with very bad health who is not marked as sick or disabled

[3]:
```
df = consistency.cleanDataFrame(df)
```

```
Checking for problem values…
Value checking finished.
Checking types…
Discrepancy of type in column  Residence Type expected string found object
Type checking finished.
Retyping columns ['Residence Type'] …
Retyping Residence Type from <class 'str'> to string
Retyping finished
>> Checking Age == 0-15
Requirement: [<MaritalStatusOptions: 1 -> SINGLE>] - CONTRADICTION
        Age Age DESC  Marital Status  \
26774     1     0-15               2
26821     1     0-15               2
207835    1     0-15               2
452434    1     0-15               2
467282    1     0-15               2
480533    1     0-15               2
499946    1     0-15               2
511216    1     0-15               2
546848    1     0-15               2
554079    1     0-15               4
555682    1     0-15               5

                                      Marital Status DESC
26774   Married or in a registered same-sex civil part…
26821   Married or in a registered same-sex civil part…
207835  Married or in a registered same-sex civil part…
452434  Married or in a registered same-sex civil part…
467282  Married or in a registered same-sex civil part…
480533  Married or in a registered same-sex civil part…
499946  Married or in a registered same-sex civil part…
511216  Married or in a registered same-sex civil part…
546848  Married or in a registered same-sex civil part…
554079  Divorced or formely in a same-sex civil partne…
555682  Widowed or surviving partner from a same-sexci…
Requirement: [<SocialGradeOptions: -9 -> NO_CODE>] - HOLDS
```

```
Requirement: [<HoursWorkedPerWeekOptions: -9 -> NO_CODE>] - HOLDS
Requirement: [<IndustryOptions: -9 -> NO_CODE>] - HOLDS
Requirement: [<OccupationOptions: -9 -> NO_CODE>] - HOLDS
Requirement: [<EconomicActivityOptions: -9 -> NO_CODE>] - HOLDS
>> Checking Population Base == Student living away from home during term-time
Requirement: [<FamilyCompositionOptions: -9 -> NO_CODE>] - HOLDS
Requirement: [<CountryOfBirthOptions: -9 -> NO_CODE>] - HOLDS
Requirement: [<HealthOptions: -9 -> NO_CODE>] - HOLDS
Requirement: [<EthnicityOptions: -9 -> NO_CODE>] - HOLDS
Requirement: [<ReligionOptions: -9 -> NO_CODE>] - HOLDS
Requirement: [<EconomicActivityOptions: -9 -> NO_CODE>] - HOLDS
Requirement: [<OccupationOptions: -9 -> NO_CODE>] - HOLDS
Requirement: [<IndustryOptions: -9 -> NO_CODE>] - HOLDS
Requirement: [<HoursWorkedPerWeekOptions: -9 -> NO_CODE>] - HOLDS
Requirement: [<SocialGradeOptions: -9 -> NO_CODE>] - HOLDS
>> Checking Health == Very bad health
Requirement: [<EconomicActivityOptions: 8 -> SICK_OR_DISABLED>,
<EconomicActivityOptions: 5 -> RETIRED>, <EconomicActivityOptions: 6 ->
STUDENT_INACTIVE>, <EconomicActivityOptions: 9 -> OTHER>] - CONTRADICTION
        Health      Health DESC  Economic Activity  \
1025         5  Very bad health                  2
2054         5  Very bad health                  2
2477         5  Very bad health                  7
2648         5  Very bad health                  3
2919         5  Very bad health                  1
…         …               …                …
566627       5  Very bad health                  1
567482       5  Very bad health                  4
568308       5  Very bad health                  1
568805       5  Very bad health                  1
569045       5  Very bad health                  1

                                   Economic Activity DESC
1025                   Economically active: Self-employed
2054                   Economically active: Self-employed
2477    Economically inactive: Looking after home or f…
2648                      Economically active: Unemployed
2919                        Economically active: Employee
…                                                     …
566627                      Economically active: Employee
567482             Economically active: Full-time student
568308                      Economically active: Employee
568805                      Economically active: Employee
569045                      Economically active: Employee

[971 rows x 4 columns]
>> Checking Student == No
Requirement: [<CountryOfBirthOptions: 1 -> UK>, <CountryOfBirthOptions: 2 ->
```

```
NON_UK>] - HOLDS
>> Checking Student == Yes
Requirement: [<EconomicActivityOptions: 4 -> FULL_TIME_STUDENT>,
<EconomicActivityOptions: 6 -> STUDENT_INACTIVE>, <EconomicActivityOptions: -9
-> NO_CODE>] - HOLDS
>> Checking Economic Activity == Economically inactive: Retired
Requirement: [<HoursWorkedPerWeekOptions: -9 -> NO_CODE>] - HOLDS
>> Checking Economic Activity == Economically active: Unemployed
Requirement: [<HoursWorkedPerWeekOptions: -9 -> NO_CODE>] - HOLDS
>> Checking Residence Type == Resident in a communal establishment
Requirement: [<FamilyCompositionOptions: -9 -> NO_CODE>] - HOLDS
Requirement: [<SocialGradeOptions: -9 -> NO_CODE>] - HOLDS
```

1.1.1 Data refining results

After refining the data we can see the data is of high quality. None of the values have the wrong values.

However, we do notice a few discrepencies in the data.

The first contradiction is that there are people in the census that have been married under the age of 16, and even some that are widows or divorced. Under UK law their marriage would not be legal, even in 2011: - https://www.gov.uk/government/news/legal-age-of-marriage-in-england-and-wales-rises-to-18 - https://www.gov.uk/marriage-abroad

This is likely a discrepency between what the responders understood as being married, and the actual UK laws, or a mistake in the response.

The second contradiction is that there are many people with very bad health that are still working. We interpreted "Very bad health" as being so ill that they would be unable to work - i.e the sick or disabled category. We believe this is a discrepency, but it is more arguable that the other discrepency, and this shows by the large number of rows with this contradiction.

We decided to not remove the rows with these contradictions, because although we did think there was a contradiction, it was only with a singular column and was likely a contradiction due to a disagreement between our opinion and the census responder's interpretation of the categories.

Next lets save the cleaned data to a separate file so we can reuse it later.

[4]:
```python
cleanPath = "../data/census2011-clean.csv"
df.to_csv(cleanPath, index=False) # save to csv
```

To recreate this step, navigate to the parent directory, then execute the `./run_consistency` script which takes a csv path as a parameter

1.1.2 Refinement - Unit Testing

Since we saw no invalid values in the test data, we needed to test that we would actually pick up on any invalid values. For this practical, we did not see many invalid values in the data so we needed some other way to test that our input validation was working correctly.

In order to do this, we created tests that checked various permitted and disallowed values by calling the same functions as our verification code.

We worked on tests and encoding of the variables separately, which meant that the chance of anything being missed by both was very low. It also gave us confidence that when we changed the parsing code to make it more extensible and optimise it, that we would know if anything was missed. In fact, we caught a couple of mistakes using these unit tests.

Our unit tests can be run with the `./test.sh` script

[5]:
```python
import test as tests

tests.test()
```

```
test_age_invalid (test_census_microdata_2011.TestExampleMicroData) … ok
test_age_valid (test_census_microdata_2011.TestExampleMicroData) … ok
test_birth_country_invalid_zero
(test_census_microdata_2011.TestExampleMicroData) … ok
test_birth_country_valid (test_census_microdata_2011.TestExampleMicroData) …
ok
test_economic_activity_invalid (test_census_microdata_2011.TestExampleMicroData)
… ok
test_economic_activity_valid (test_census_microdata_2011.TestExampleMicroData)
… ok
test_ethnicity_invalid (test_census_microdata_2011.TestExampleMicroData) … ok
test_ethnicity_valid (test_census_microdata_2011.TestExampleMicroData) … ok
test_family_composition_invalid
(test_census_microdata_2011.TestExampleMicroData) … ok
test_family_composition_valid (test_census_microdata_2011.TestExampleMicroData)
… ok
test_health_invalid (test_census_microdata_2011.TestExampleMicroData) … ok
test_health_valid (test_census_microdata_2011.TestExampleMicroData) … ok
test_hours_invalid (test_census_microdata_2011.TestExampleMicroData) … ok
test_hours_valid (test_census_microdata_2011.TestExampleMicroData) … ok
test_industry_invalid (test_census_microdata_2011.TestExampleMicroData) … ok
test_industry_valid (test_census_microdata_2011.TestExampleMicroData) … ok
test_invalid_regions (test_census_microdata_2011.TestExampleMicroData) … ok
test_invalid_residence_type (test_census_microdata_2011.TestExampleMicroData)
… ok
test_marital_status_invalid_zero
(test_census_microdata_2011.TestExampleMicroData) … ok
test_marital_status_valid (test_census_microdata_2011.TestExampleMicroData) …
ok
test_occupation_invalid (test_census_microdata_2011.TestExampleMicroData) … ok
test_occupation_valid (test_census_microdata_2011.TestExampleMicroData) … ok
test_population_base_invalid (test_census_microdata_2011.TestExampleMicroData)
… ok
test_population_base_valid (test_census_microdata_2011.TestExampleMicroData) …
ok
```

```
test_religion_invalid (test_census_microdata_2011.TestExampleMicroData) … ok
test_religion_valid (test_census_microdata_2011.TestExampleMicroData) … ok
test_residence_type (test_census_microdata_2011.TestExampleMicroData) … ok
test_sex_invalid_3 (test_census_microdata_2011.TestExampleMicroData) … ok
test_sex_valid (test_census_microdata_2011.TestExampleMicroData) … ok
test_social_grade_invalid (test_census_microdata_2011.TestExampleMicroData) …
ok
test_social_grade_valid (test_census_microdata_2011.TestExampleMicroData) … ok
test_student_invalid_zero (test_census_microdata_2011.TestExampleMicroData) …
ok
test_student_valid (test_census_microdata_2011.TestExampleMicroData) … ok
test_valid_regions (test_census_microdata_2011.TestExampleMicroData) … ok
test_valid_student_id (test_census_microdata_2011.TestExampleMicroData) … ok

----------------------------------------------------------------------
Ran 35 tests in 0.051s

OK
```

1.2 Design: Refining the data - Students A & B

Initially, we simply enumerated the possible values of each column and tested each column. This was a very simplistic approach, and allowed us to rapidly evaluate the quality of the data. By first doing a quick analysis of the data, we were able to make an informed decision of how to handle invalid data. Since there were no invalid values we decided that future datasets would be unlikely to have a large amount of invalid data, and so we decided to remove any invalid rows from the data set. If there were a large number of invalid rows, this could cause issues as the sample used for analysis may not be fully representative of the original data, and could lead us to draw invalid conclusions.

We wanted to make cleaning and verification data extensible to other data sets, but our current way would need to be completely rewritten for a new data set with new columns. Therefore, we developed `OptionEnum`, that extends `Enum`, and stores a mapping of key to description. We can now easily work with the data set, listing all possible values with their descriptions as well as parsing.

[6]:
```
import census_microdata_2011 as md
[f"{x.key()}: {x.desc()}" for x in md.EthnicityOptions]
```

[6]:
```
['1: White',
 '2: Mixed',
 '3: Asian or Asian British',
 '4: Black or Black British',
 '5: Chinese or Other ethnic group',
 '-9: No code required (Not resident in england or wales, students or
schoolchildren living away during term-time)']
```

We can also use this to easily search for a particular value in the dataset and easily translate the cryptic key names into the descriptive strings

```
[7]: ages_35_44 = df.loc[df["Age"] == md.AgeOptions.FROM_35_TO_44.key()].copy()
ages_35_44["Age"] = ages_35_44["Age"].replace(md.AgeOptions.mappings)
ages_35_44
```

```
[7]:        Person ID      Region Residence Type  Family Composition  \
1         7394745  E12000001              H                   5
2         7395066  E12000001              H                   3
6         7394871  E12000001              H                   5
18        7395059  E12000001              H                   1
22        7394857  E12000001              H                   2
…               …          …              …                   …
569685    7944687  W92000004              H                   2
569693    7945171  W92000004              H                   2
569706    7946284  W92000004              H                   1
569725    7945073  W92000004              H                   1
569733    7944827  W92000004              H                   5

        Population Base  Sex    Age  Marital Status  Student  \
1                     1    1  35-44               1        2
2                     1    2  35-44               1        2
6                     1    2  35-44               3        2
18                    1    1  35-44               1        2
22                    1    1  35-44               2        2
…                     …    …    …                 …        …
569685                1    1  35-44               1        2
569693                1    2  35-44               2        2
569706                1    2  35-44               1        2
569725                1    2  35-44               1        2
569733                1    2  35-44               4        2

        Country of Birth  Health  Ethnic Group  Religion  Economic Activity  \
1                      1       1             1         2                  1
2                      1       1             1         1                  1
6                      1       2             1         1                  1
18                     1       3             1         1                  1
22                     1       1             1         1                  1
…                      …       …             …         …                  …
569685                 1       2             1         2                  1
569693                 1       1             1         2                  1
569706                 1       1             1         3                  1
569725                 1       1             1         2                  1
569733                 1       1             1         1                  1

        Occupation  Industry  Hours worked per week  Approximated Social Grade
1                8         6                      4                          3
2                6        11                      3                          4
6                6        11                      2                          3
```

```
18                8         2                      3                           4
22                8         2                      3                           4
…            …        …                     …                          …
569685            8         4                      3                           4
569693            4         4                      2                           2
569706            3        11                      3                           2
569725            4        11                      3                           2
569733            6        10                      2                           4

[78641 rows x 18 columns]
```

We also wanted to be able to list possible contradictions between fields in an intuitive and easy-to change per-dataset format.

In order to do this we created a list of tuples. The example below shows one contradiction tuple, which defines that people under 16 should have NO_CODE entered for various fields.

This allows us to easily create add contradictions on a per dataset basis

[8]:
```
t = md.dataset.get_contradictions()[0]
print("IF ", t[0].__repr__())
print("THEN:", t[1])
```

```
IF  <AgeOptions: 1 -> UNDER_16>
THEN: [<MaritalStatusOptions: 1 -> SINGLE>, <SocialGradeOptions: -9 -> NO_CODE>,
<HoursWorkedPerWeekOptions: -9 -> NO_CODE>, <IndustryOptions: -9 -> NO_CODE>,
<OccupationOptions: -9 -> NO_CODE>, <EconomicActivityOptions: -9 -> NO_CODE>]
```

1.3 Descriptive analysis of cleaned data - Student B

For this basic requirement we were asked to obtain:
* the total number of records in the dataset * the type of each variable in the dataset * all different values that each variable takes and the number of occurences for each value (excluding Person ID)

To encapsulate this entire requirement, we implemented `printSummary`

[9]:
```
import stats as s
s.printSummary(df)
```

```
Number of Records: 569741
Column types-----------
Region                                 object
Residence Type                 string[python]
Family Composition                      int64
Population Base                         int64
Sex                                     int64
Age                                     int64
Marital Status                          int64
Student                                 int64
Country of Birth                        int64
Health                                  int64
```

```
Ethnic Group                          int64
Religion                              int64
Economic Activity                     int64
Occupation                            int64
Industry                              int64
Hours worked per week                 int64
Approximated Social Grade             int64
dtype: object

Residence Type       H      C
count           559087  10654

Family Composition       2      1      3      5     -9     4     6
count               300962  96690  72641  64519  18851  9848  6230

Population Base       1     2     3
count            561040  6730  1971

Sex         2       1
count  289172  280569

Age         1      4      5      3      2      6      7      8
count  106832  78641  77388  75948  72785  65666  48777  43704

Marital Status       1       2      4      5      3
count           270999  214180  40713  31898  11951

Student       2       1
count    443204  126537

Country of Birth       1      2    -9
count             485645  77292  6804

Health       1       2      3      4     5    -9
count   264971  191744  74480  24558  7184  6804

Ethnic Group       1      3      4      2    -9     5
count         483477  42712  18786  12209  6804  5753

Religion       2       1      9      6     4    -9     7     5     3     8
count     333481  141658  40613  27240  8214  6804  4215  2572  2538  2406

Economic Activity       1      -9      5      2      6      3      8      7
4      9
count              216025  112618  97480  40632  24756  18109  17991  17945
14117  10068

Occupation      -9      2      9      4      5      3      1      7      6
```

```
8
count        149984  64111  58483  53254  48546  44937  39788  38523  37297
34818

Industry      -9      4      2      8     11     10      6      3      5      9
12      7     1
count     149984  68878  53433  49960  49345  40560  35240  30708  25736  24908
20256  16776  3957

Hours worked per week      -9       3      2      4      1
count                  302321  153938  52133  35573  25776

Approximated Social Grade       2      -9       4      1      3
count                      159642  124103  123740  82320  79936
```

To see the counts of an individual column, use `getUniqueCounts`

[10]:
```python
s.getUniqueCounts(df["Country of Birth"])
```

[10]:

	Country of Birth	count
0	1	485645
1	2	77292
2	-9	6804

To recreate this step, navigate to the parent directory, then execute the `./run_summary` script which takes a csv path as a parameter

The second part of the descriptive analysis, we were told to build the following plots: * bar chart for the number of records for each region
* bar chart for the number of records for each occupation
* pie chart for the distribution of the sample by age
* pie chart for the distribution of the sample by the economic activity.

Our implementation of ipywidgets allows custom selection of columns using a dropdown, so you can view these plots and more:

[11]:
```python
%matplotlib inline

import basic_plots as b

out1 = b.Output()
out2 = b.Output()
b.interact(b.genRecordBarPlot, df=b.fixed(df), colName=b.Dropdown(options=df.
 ↪columns, value='Region'), save=b.fixed(False), _output=out1)
b.interact(b.genDistPieChart, df=b.fixed(df), colName=b.Dropdown(options=df.
 ↪columns, value='Economic Activity'), save=b.fixed(False), _output=out2)
display(out1)
display(out2)
```

```
interactive(children=(Dropdown(description='colName', index=1, options=('Person␣
↪ID', 'Region', 'Residence Type…
```

```
interactive(children=(Dropdown(description='colName', index=13, options=('Person␣
↪ID', 'Region', 'Residence Typ…
```

```
Output()
```

```
Output()
```

To recreate this step, navigate to the parent directory, then execute the `./run_plots` script which takes a csv path as a parameter and assumes the existence of "Region", "Occupation", "Age" and "Economic Activity" as columns. When running the script, the plots will be saved as png images in the images directory.

1.4 Using groupby to produce tables - Student B

We were asked to produce the following tables:
* number of records by region and industry
* number of records by occupation and social grade

To make this functionality easy to reuse, we wrote one function, `getGroupTable` which takes a dataframe and two column names and produces a table showing the number of records for the pair of columns in the given dataframe. This can be done with any pair of columns in the given dataframe.

[12]:
```
s.getGroupTable(df, "Region", "Industry")
```

[12]:
```
        Region  Industry  counts
0    E12000001        -9    6854
1    E12000001         4    3087
2    E12000001         2    2851
3    E12000001        11    2524
4    E12000001         8    1883
..         ...       ...     ...
125  W92000004         5    1641
126  W92000004         6    1500
127  W92000004        12     992
128  W92000004         7     594
129  W92000004         1     403

[130 rows x 3 columns]
```

[13]:
```
s.getGroupTable(df, "Occupation", "Approximated Social Grade").head()
```

[13]:
```
   Occupation  Approximated Social Grade  counts
0          -9                         -9  116915
1          -9                          2   17787
2          -9                          4   12169
3          -9                          3    2062
```

```
4           -9                          1    1051
```

An example of reuse of this method:

[14]:
```
s.getGroupTable(df, "Student", "Religion")
```

[14]:
```
    Student  Religion  counts
0         1         2   60401
1         1         1   35488
2         1         6   10398
3         1         9    8660
4         1        -9    6804
5         1         4    2113
6         1         7    1091
7         1         5     624
8         1         3     607
9         1         8     351
10        2         2  273080
11        2         1  106170
12        2         9   31953
13        2         6   16842
14        2         4    6101
15        2         7    3124
16        2         8    2055
17        2         5    1948
18        2         3    1931
```

1.5 Queries with pandas - Student C

We were asked to perform queries on the dataframe to find:
* Query 1 - the number of economically active people by region * Query 2 - the number of economically active people by age * any discrepancies between student status and economic activity * the number of working hours per week for students

Since we already wrote, `getGroupTable` which takes a dataframe and two column names and produces a table, we were able to extend the operation for the second easy requirment. Queries was a simple addition in the `code/queries.py` file. All of the results are presented below. Plots have been presented for both query one and two.

[15]:
```
%matplotlib widget

import queries as q
import threed_plots as p

q.find_query1(df)
p.plotScatter(s.getGroupTable(df, "Region", "Age"), "Region", "Economic␣
 ↪Activity")
```

```
Number of economically active people by region:
```

```
Region
E12000001    21371
E12000002    57513
E12000003    43073
E12000004    36861
E12000005    45258
E12000006    47674
E12000007    66212
E12000008    70306
E12000009    43807
W92000004    25048
Name: Person ID, dtype: int64


interactive(children=(Dropdown(description='Region', options=('E12000001',␣
↪'E12000002', 'E12000003', 'E1200000…
```

[16]:
```
q.find_query2(df)
p.plotScatter(s.getGroupTable(df, "Region", "Age"), "Region", "Age")
```

```
Number of economically active people by age:
Region
E12000001    21371
E12000002    57513
E12000003    43073
E12000004    36861
E12000005    45258
E12000006    47674
E12000007    66212
E12000008    70306
E12000009    43807
W92000004    25048
Name: Person ID, dtype: int64


interactive(children=(Dropdown(description='Region', options=('E12000001',␣
↪'E12000002', 'E12000003', 'E1200000…
```

[17]:
```
q.find_discrepancies(df)
```

```
Discrepancies found between student status and economic activity:
Economic Activity
-9    88582
 6    23838
 4    14117
Name: count, dtype: int64
```

[18]:
```
q.find_hours(df)
```

```
Working hours found per week for students:
199702
```

1.6 3D plots - Student B w/ ipywidgets - Student C

The following 3D plots are made using the previously described `getGroupTable` method. There are two methods to generate them, `plotScatter` which produces a 3D scatter plot and `plotSurface` which produces a 3D surface plot.

[19]:
```
p.plotScatter(s.getGroupTable(df, "Region", "Industry"), "Region", "Industry")
p.plotSurface(s.getGroupTable(df, "Region", "Industry"), "Region", "Industry")

p.plotScatter(s.getGroupTable(df, "Occupation", "Approximated Social Grade"),␣
 ↪"Occupation", "Approximated Social Grade")
p.plotSurface(s.getGroupTable(df, "Occupation", "Approximated Social Grade"),␣
 ↪"Occupation", "Approximated Social Grade")
```

```
interactive(children=(Dropdown(description='Region', options=('E12000001',␣
 ↪'E12000002', 'E12000003', 'E1200000…
```

```
interactive(children=(Dropdown(description='Region', options=('E12000001',␣
 ↪'E12000002', 'E12000003', 'E1200000…
```

```
interactive(children=(Dropdown(description='Occupation', options=(1, 2, 3, 4, 5,␣
 ↪6, 7, 8, 9, -9), value=1), Dr…
```

```
interactive(children=(Dropdown(description='Occupation', options=(1, 2, 3, 4, 5,␣
 ↪6, 7, 8, 9, -9), value=1), Dr…
```

1.7 Mapping by region - Student B

To map the data for each region on a map, we used the folium library to generate the map elements along with the Find that Postcode API to generate the region borders. This functionality is encapsulated in the method `plotMap` which takes a dataframe and a column whose data is to be visualized.

On the map itself, the data is visualized in multiple ways. Firstly, there is a colorscale representing the average value obtained from a given region. This is particularly useful when there is a correlation between the key and value (ie: Age). Secondly, when hovering over a particular region, you can view a table of the number of records for that region. To translate this information, there is a legend containing the keys and their associated values. For convenience, the legend can be dragged around the map.

[20]:
```
import map_plot as m

plot = m.plotMap(df, "Age", False)
plot
```

```
Generating region by Age map…
```

```
Done.
```

[20]:
```
<folium.folium.Map at 0x7fbf50f026d0>
```

[21]:
```
plot = m.plotMap(df, "Marital Status", False)
plot
```

```
Generating region by Marital Status map…

Done.
```

[21]:
```
<folium.folium.Map at 0x7fbf4c13f130>
```

1.8 Performance Analysis and Optimisation - Student A

One of the Hard requirements was to analyse the performance of different steps of our analysis. To do this we used the timeit library and created performance tests over a range of data set sizes.

We identified two problematic steps: **validation** and **parsing**.

Benchmarking with timeit In order to compare the optimised and unoptimised steps of the data analysis, we used the built in timeit library in python along with the census data already provided. We ran the steps on different numbers of rows from the data set, ranging from 10 rows to 400000. This allowed me not only to see whether the algorithm had sped up, but predict how it would behave on even larger data sets that our code could be used for in future.

In order to make increase the reliability of the results, we run the steps multiple times. We run the unoptimised variants 3 times, and the optimised variants 10 times, due to the large disparity in time taken to run each.

Validation The validation step is the step that checks that the entire data frame for any invalid data, and reports rows that are invalid. This was immediately observed as being slow from when the validation code was first made.

Our original implementation used the naive approach of iterating through the data frame, and checking that the encoded was one of the permitted values.

Experimenting with pandas, we found the method `Series.isin(values)`, which produces a new series with True/False values of whether each value was in the given set of values. We changed the method to use this which also allowed us to easily see which row numbers contained the problematic values.

When we changed to this we immediately saw a huge performance improvement, which we later benchmarked (see below).

Parsing The parsing step is the step that converted the encoded data into the long human readable descriptions. As we learned from the previous step that pandas is much faster than iteration, we used `Series.apply(func)` to apply a mapping function that parsed each value. However, this was still not very fast, and although it was unlikely to ever be used on the whole data frame, it was still slow on subsets.

We tried multiple different approaches, such as using a dictionary inside the parsing function, but this only improved the performance marginally. After lots of experimentation, we wondered whether the .apply() in pandas was not the best way to perform the transformation. We discovered that there was indeed a method to elimate this, `.replace()`, which takes a dictionary that maps from the key to a value. By using this we was able to see a massive performance improvement.

Running the benchmarks To run the benchmarks, the script `./run_performance` can be used, which will generate graphs in the `images/performance` directory.

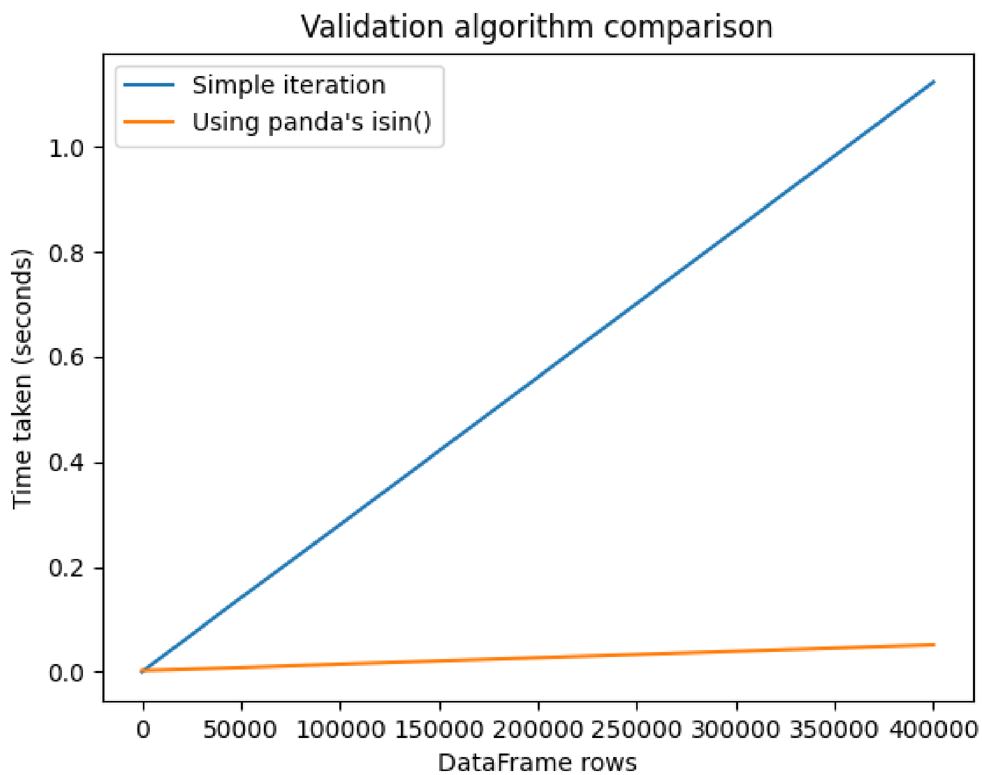
The benchmark can also be run below - normally it would be a bad idea to run benchmarks in a Jupyter notebook, but in this case the performance difference is so extreme that it should overshadow any noise.

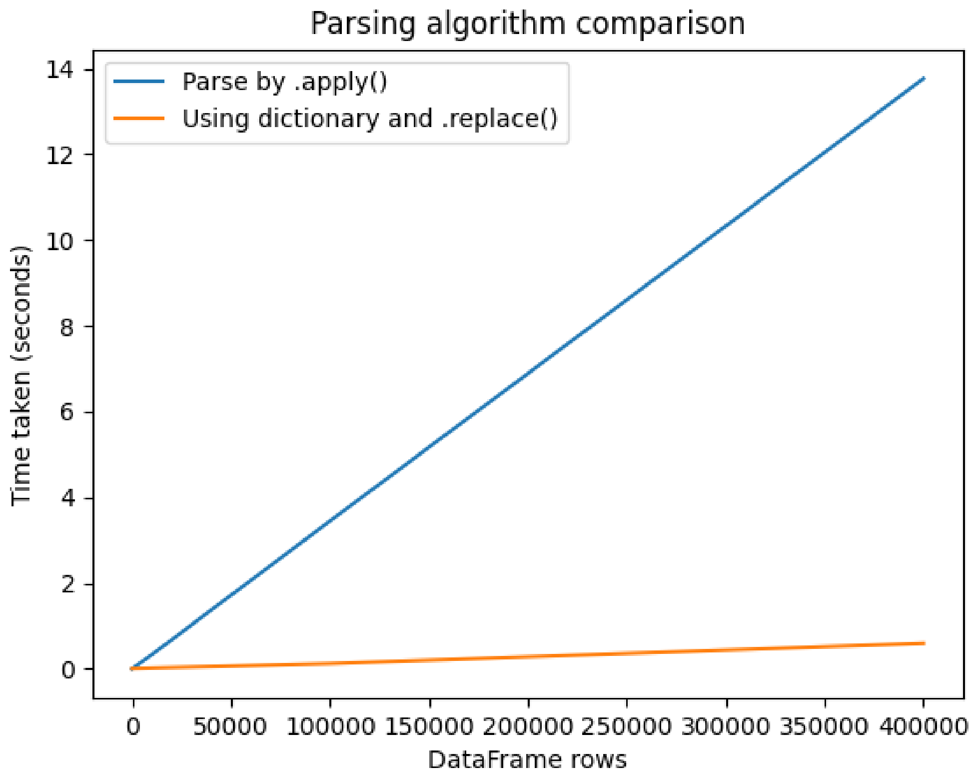
[22]:
```
#%matplotlib inline

import performance
performance.profile_and_plot(df)
```

```
10 Rows: Simple Iteration… Current implementation… Parse df list… Parse df
dict…
100 Rows: Simple Iteration… Current implementation… Parse df list… Parse
df dict…
500 Rows: Simple Iteration… Current implementation… Parse df list… Parse
df dict…
1000 Rows: Simple Iteration… Current implementation… Parse df list… Parse
df dict…
5000 Rows: Simple Iteration… Current implementation… Parse df list… Parse
df dict…
10000 Rows: Simple Iteration… Current implementation… Parse df list… Parse
df dict…
50000 Rows: Simple Iteration… Current implementation… Parse df list… Parse
df dict…
100000 Rows: Simple Iteration… Current implementation… Parse df list…
Parse df dict…
400000 Rows: Simple Iteration… Current implementation… Parse df list…
Parse df dict…
Saved validation.png
Saved parse.png
== Validation Results ==
Simple iteration: (10: 0.0002364), (100: 0.00045891), (500: 0.0015177), (1000:
0.0029661), (5000: 0.014497), (10000: 0.028496), (50000: 0.14134), (100000:
0.28008), (400000: 1.1232)
Using panda's isin(): (10: 0.0029725), (100: 0.0028284), (500: 0.0027194),
(1000: 0.0028239), (5000: 0.0032877), (10000: 0.0038061), (50000: 0.0083393),
(100000: 0.014933), (400000: 0.051708)
== Parsing Results ==
Parse by .apply(): (10: 0.0015083), (100: 0.0046068), (500: 0.017954), (1000:
0.035268), (5000: 0.16526), (10000: 0.33097), (50000: 1.715), (100000: 3.4443),
(400000: 13.761)
```

```
Using dictionary and .replace(): (10: 0.0083709), (100: 0.0085755), (500:
0.0099821), (1000: 0.010226), (5000: 0.015071), (10000: 0.021938), (50000:
0.066102), (100000: 0.12371), (400000: 0.59884)
```





1.8.1 Benchmarking Results

Validation We can clearly see that the pandas' `isin()` method massively outperforms iteration. At 400000 rows, `isin()` is **over 20x faster**: 0.044s vs 1.045s.

From the graph it appears this is only a constant improvement - both algorithms seem to have O(n) complexity, which means that with a large enough data set, the validation step could still take a long time. With our dataset however, it goes from being slightly slow to immediate, which is a much appreciated improvement.

Parsing The results from the parsing step is similar to the validation step. The `.replace()` method massively outperforms `.apply()`. At 400000 rows, `.replace()` is over 25x faster (0.49s vs 13.7s)

In order to use replace, we were able to use an answer from Ethan Furman, 2019 to create a dictionary of key to value which could be used in the replace method, massively improving performance without adding any extra code to individual datasets, and proving the extensibility of our design.

Again, the complexity of both algorithms seems to be the same - O(n).

Lessons and Recommendations The main recommendations from this experience is to - Use panda's built-in methods whenever possible - Prefer passing primitives (i.e. lists, sets) instead of functions

We believe the reason for this is that pandas is based on numpy. Numpy is designed to perform operations on multiple columns at the same time, and is partly written in C. Therefore, in order to access the best performance we need to pass arguments that can be easily translated into C, such as the primitives in python. This then allows pandas to use numpy effectively, without having to repeatedly cross the C barrier.

This article discusses the issue slightly: https://labs.quansight.org/blog/unlocking-c-level-performance-in-df-apply

1.9 Conclusion

In this project we successfully performed data-analysis on the census data set. We took care to make the design of our methods and classes easily re-useable to other similar data sets, such as a future census. Furthermore, we made our analysis highly performant, allowing it to scale to a much larger data set.

We used a wide range of graphs, including maps, pie charts, bar graphs and made them interactive within a jupyter notebook that not only demonstrated the analysis but also documented our analysis journey.

We found contradictions and interesting statistics about our data, and used unit testing where appropriate.

If we had more time, we would analyse other datasets using the framework we built during this practical and evaluate the practical reuseability of the code we designed.

We completed all the of basic and additional requirements to a high standard, and we are very proud of our submission.

1.9.1 Bibliography

"3D Scatterplot with Strings in Python." n.d. Stack Overflow. Accessed March 29, 2024. https://stackoverflow.com/questions/54113067/3d-scatterplot-with-strings-in-python.

"Add and Initilalize an Enum Class Variable in Python." n.d. Stack Overflow. Accessed March 29, 2024. https://stackoverflow.com/questions/56735081/add-and-initilalize-an-enum-class-variable-in-python.

"Check If File Is Readable with Python: Try or If/Else?" n.d. Stack Overflow. Accessed March 29, 2024. https://stackoverflow.com/questions/32073498/check-if-file-is-readable-with-python-try-or-if-else.

"How to Turn X-Axis Values into a Legend for Matplotlib Bar Graph." n.d. Stack Overflow. Accessed March 29, 2024. https://stackoverflow.com/questions/62941033/how-to-turn-x-axis-values-into-a-legend-for-matplotlib-bar-graph.

Jaramillo, Juan Felipe Alvarez. 2020. "Mapping the UK and Navigating the Post Code Maze." Medium. October 15, 2020. https://focaalvarez.medium.com/mapping-the-uk-and-navigating-the-post-code-maze-4898e758b82f.

O'Hara, Patrick. 2022. "Interactive Mapping in Python with UK Census Data." Medium. February 18, 2022. https://medium.com/@patohara60/interactive-mapping-in-python-with-uk-census-data-6e571c60ff4.

"Plotting UK Districts, Postcode Areas and Regions." n.d. Stack Overflow. Accessed March 29, 2024. https://stackoverflow.com/questions/46775667/plotting-uk-districts-postcode-areas-and-regions.

Python, Real. n.d. "Python Folium: Create Web Maps from Your Data – Real Python." Realpython.com. https://realpython.com/python-folium-web-maps-from-data/.

"Show Different Pop-Ups for Different Polygons in a GeoJSON [Folium] [Python] [Map]." n.d. Stack Overflow. Accessed March 29, 2024. https://stackoverflow.com/questions/54595931/show-different-pop-ups-for-different-polygons-in-a-geojson-folium-python-ma.

"Surface Plots in Matplotlib." n.d. Stack Overflow. Accessed March 29, 2024. https://stackoverflow.com/questions/9170838/surface-plots-in-matplotlib.

[ ]:

```
```