Data Analytics Coursework 1

Oliwier Kulczycki

# Introduction

In this coursework, I aim on tackling data cleaning, conversion, visualisation and analysis on the dataset provided – credits. This dataset features many errors and mistakes, which first must be cleaned. I will be using OpenRefine for cleaning, Weka for help with visualisation, data format conversions as well as some surface level analysis. Deeper conversions, visualisation and analysis will be done with python – more specifically a Jupyter notebook. This will allow me to more easily display the code as well as have proper annotations and run it in a more controlled manner. Stock python is unable to do these levels of data analysis which is why I’ll be utilising pre-written libraries: Pandas and NumPy for data wrangling as well as matplotlib for data visualisation.

# Data Preparation

Before starting off, I had to first edit the original, dirty dataset to add headings. The entire dataset was lacking any headings which would be devastating for data cleaning and management.

## Data Cleaning

Data cleaning is a process involving basic level analysis, with the support of tools, to view and correct any mistakes with the format of the data within the dataset. This could include correcting; obvious mistakes, data corruption, typos, language/writing differences, etc. This makes sure all data adheres to a certain format and that when later analysed, isn’t subject to skewed results. A common example could be “Dataset originally takes monetary values as full pounds – data was accidentally inputted as pennies, (1000 instead of 10), for some fields.”

### Quotation marks

Because of issues, which are later encountered during the data processing section, I decided it would be best to remove all quotation marks – ‘’, from the dataset. This included essentially every string’s first and last character. I did this by going through each facet for each attribute and editing all instances.re

Facets

When performing operations on a dataset with OpenRefine, one of the most powerful tools at disposal are ‘Facets’. This tool allows for easy viewing of unique entries in fields when using the ‘Text Faucet’ tool, as well as easy viewing of numeric ranges with the ‘Numeric Faucet’ tool.

### credit\_amount

|  |  |
| --- | --- |
| The *credit\_amount* attribute contained exceedingly large values, far outside the of anything else within the range.  The 7 *out-of-range* entries all seemed to contain a lot of 0’s at the end of them. Under the assumption that these were inserted as pennies, instead of full pounds, I removed the trailing two 0’s for each of those entries. |  |

|  |  |
| --- | --- |
| Additionally, on the low side of the spectrum, present were values with decimal points. Wanting to follow the precedent set by already existing data, I opted to remove them. I tried to follow number-writing conventions and convert these into real numbers accordingly. Because *‘7.999’* is written similarly to how a visual showing of *thousand* - ‘7,999’ for ease of readability, I decided to change this to 7999.  Unlike this however, 48.43 was rounded down to 48 as it more closely resembles a representation of pennies rather than a style of writing. |  |

### class

The *class* attribute featured stand-out entries such as *‘1’* and *‘0’*. I deferred to using the **computing**approach and assuming that *‘0’* means *‘bad’*, *‘1’* means *‘good’*.

### purpose

The *purpose* attribute required the most work. It features many mistakes: **typos**, **shortenings**, **format**, etc. I fixed the **typos** for; *business, education, furniture/equipment, radio/tv, used car.* These were all obvious typos/shortenings and didn’t require much thought before editing. I followed through by cross-referencing the brief to see which values should be set as the ‘correct’ ones.

### job

The *job* attribute had some incorrect entries such as the *‘good’* and *‘poor’* entries. I designated these to ‘skilled’ and ‘unskilled resident’ respectively. I also cleaned up and made the quotation marks consistent. [*(Although this is addressed anyway).*](#_Quotation_marks_1)

### age

Some fields had the age set to exceptional values such as *1* – (Figure 3). I assumed these were misinputs and followed conventions seen in the data and assumed these were meant to be *19*, as seen, the two fields below share almost all other values except age.

## Data Transformations and Conversions

Transformations are done on the data by clicking on the field title, Edit Cells -> Common transforms -> To text/number. This is done to establish what type of data that field is. This is important for later data processing so that numeric values aren’t treated as nominal values – (strings).

*‘To number’* transformations are done on fields; *Case\_no, credit\_amount, age.* All the rest of the fields are also forcefully ‘To text’ transformed, just to be sure they are in the right format.

### Numeric Conversion

I decided to use python for the numeric conversion. Within the notebook, I added a few lines which would take care of converting the original, cleaned dataframe into an all-numeric version.

For this conversion, I took advantage of Pandas built-in Categorical Data (Pandas Documentation, 2025). This dataframe type only allows for a fixed number of possible values, which is perfect for this use case of essentially enumerating the values within the dataset.

A computer screen with text

AI-generated content may be incorrect.

(**df\_numeric** is a direct copy of the entire cleaned dataset extracted from a csv file.)

This code runs through each column in the dataframe, although avoids any previously already numeric fields such as *Case\_no, credit\_amount* and *age.* The following line casts each column within the dataframe into the dtype of category*.* As mentioned earlier, this is useful as it assigns each unique entry in that column a unique integer value and stores it inside of a temporary series object (Pandas Documentation, 2025). The .cat.codes() retrieves these mapping from the series object and the df\_numeric[col] =at the beginning of the line finally applies this to the actual column.

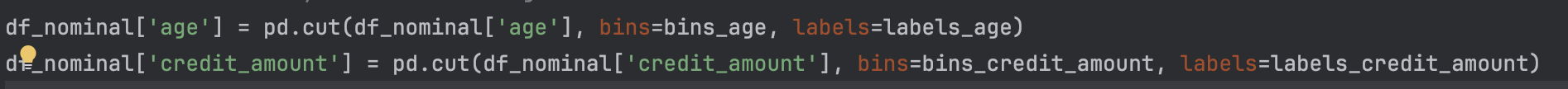
A screenshot of a credit score

AI-generated content may be incorrect.

The result of this process is a fully mapped copy of the original dataframe, with the already numeric fields intact.

### Nominal Conversion

The conversion of the dataframe to nominal was much simpler than the conversion to numeric. Firstly, I decided that the indexing attribute of ‘Case\_no’ should keep its values however that they should be cast to a string. This was done simply using the line df\_nominal['Case\_no'] = df['Case\_no'].astype(str). This simply casts the value to a string representation of it. This is important as the entire point of that attribute is to differentiate each field in case all other attributes end up being the same.



For a nominal conversion to be meaningful, a method called binning must be used for the categorisation of different value ranges within the dataset. This can be done with a built-in function in pandas; pandas.cut()(Pandas Documentation, 2025)*.*

|  |  |
| --- | --- |
| Before using this function, you must first assign the data with your chosen ranges *(bins),* these will act as cut-off points and in unison with the *labels* will be used to replace values with strings inside the dataframe.  The bins array is self-explanatory for both and decides the ranges of values for the corresponding labels. Each label maps in between two values; for example, *‘*child’ is mapped to in-between *‘0’* and *‘18’.* |  |

|  |  |
| --- | --- |
| The age brackets were chosen based on approximate real-life definitions. The brackets for *‘credit\_amount’* , were much more difficult to decide as the range of values, and their subsequent distribution is skewed. I decided to follow a similar pattern of increasing values and translate that to the different bins. A rough pattern of doubling the next value resulted in a relativity evenly split distribution. |  |

## Data Framework and Visualisation

For the visualisation of the data, I opted for the use of the matplotlib library available in python. This allows me to easily create a figure and then graph values in many ways. For this specific task I chose to go with a twin bar chart, showing two separate bars per x-axis root.

For the purposes of conciseness, I will only be showing the methods used for the chart referencing ‘personal\_status’. However, the entire process is identical for the chart referencing ‘saving\_status’, with the obvious change being ‘personal\_status’ inside the code is replaced with ‘saving\_status’.

### Gathering Data

Before being able to display any useful information, the correct data format must be gathered. I decided to go with a count of each unique value within the ‘personal\_status’ attribute, for both ‘class’ attribute value options. These are then stored as separate small dataframes which are used to display the two separate bars when plotted.

|  |  |
| --- | --- |
| The simple line of code creates a sort of **Boolean mask**, shortening the original dataframe by removing all rows where the *‘class’* attribute *isn’t* equal to *‘good’*. A simple value.counts() operation is then performed on this shortened dataframe counting occurrences of each unique entry in the ‘personal\_status’ attribute. This is all done inside a temporary dataframe, thus not altering the original. This is repeated in the opposite way – removing all rows where *‘class’* isn’t equal to *‘bad’*. | (Printing of what the gathered data looks like.) |

### Graphing

#### Creating Figure

Firstly, a figure is creating using fig\_personal\_status, g1 = plt.subplots(), this creates both the full graph - (fig\_personal\_status) and the axes - (g1). The axes will let me decorate and populate the graph, using some basic decoration functions such as setting a title, and axis labels.

#### X-Axis Labels

|  |  |
| --- | --- |
| Because the graph features two bars, per tick-label, pre-calculating the root position for these was necessary to calculate the offset per bar to display them side-by-side. This is done simply by using the built-in numpy.arange() and .set\_xticks(). |  |

.arange() simply *“returns evenly spaces values across a specified range”* (NumPy Documentation, 2025), then .set\_xticks() arranges these for the actual created graph. This creates evenly spaces ticks, which are then labelled using .set\_xticklabels, which takes the argument statuses, which is a list of each unique entry in the *‘personal\_status’* attribute.

#### Creating Bars

Final graphs can be seen at (Figure 1) – for the Distribution of Credit Class by Personal Status, and (Figure 2) – for the Distribution of Credit Class by Saving Status.

|  |  |
| --- | --- |
| A simple bar plotting function is used to plot the first half of the graph. The only major point with the call is the location. Usually represented by x, in this specific scenario, because there will be two bars displayed side-by-side, I need to offset the location by half the bar’s width. This is done to the left on the first bar, while on the second bar plotting function call, it is offset to the right. The function call comes with a specific label and colour; it also takes the [previously discussed dataframe](#_Gathering_Data) as the data points. bar\_width is declared earlier in the code as an arbitrary width of each bar. |  |

## Scientific Analysis

# Appendix

|  |  |
| --- | --- |
| A graph of a number of credit class  AI-generated content may be incorrect.  Figure 1 | A graph of a number of credit class  AI-generated content may be incorrect.  Figure 2 |

A screenshot of a computer

AI-generated content may be incorrect.

Figure 3