**So You’ve Got a Really Big Dataset. Here’s How You Clean It.**

**A detailed, step-by-step guide to data cleaning in Python with sample code.**

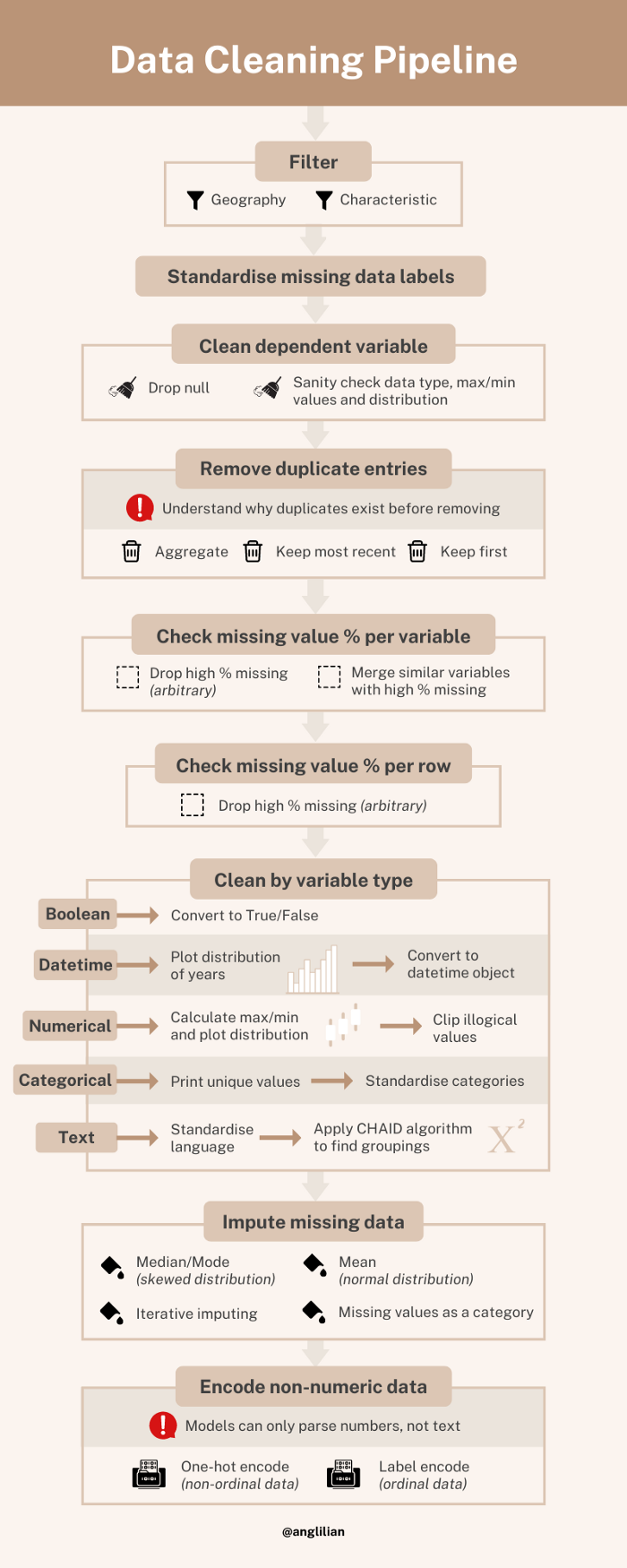


Image from [Markus Spiske](https://unsplash.com/photos/hvSr_CVecVI) (Unsplash)

You have a dataset in hand after scraping, merging, or just plain downloading it off the internet. You’re thinking about all the beautiful models you could run on it but first, you’ve got to clean it. There are a million different ways you could start and that honestly gives me choice paralysis every time I start.

After working on several *messy* datasets, here is how I’ve structured my data cleaning pipeline. If you have more efficient code or revisions to these steps to reduce bias, drop a comment and I will incorporate it into the article!

Check out this illustration I made for an overview and read on for more details on each step. The code snippets I provide are in Python and will mainly use the Pandas and matplotlib libraries.



**Filter**

If you are working with a subset of data, you want to make sure that the data represents the subset of interest. Filter for characteristics of specific columns to make sure.

With geographic data, use the shapefile of the region of interest and geopandas *sjoin* function to filter out data that is outside.

# Dataframe of point coordinates  
points\_gdf **=** geopandas**.**GeoDataFrame(  
 df[['uprn', 'LATITUDE', 'LONGITUDE']],  
 geometry**=**geopandas**.**points\_from\_xy(df**.**LONGITUDE, df**.**LATITUDE),  
 crs**=**'epsg:4326') # make sure shp\_file crs is the same*# Filter for points within boundary*  
points\_in\_boundary **=** geopandas**.**tools**.**sjoin(points\_gdf, shp\_boundary, how**=**'right')

**Standardise missing data labels**

Sometimes missing data is coded as ‘NO DATA’, ‘0’, ‘N/A’ or just an empty string. For ease of cleaning, convert all these into *np.nan.*

df **=** df**.**replace(['NO DATA','N/A', 0, ''],np**.**nan)

**Clean dependent variable**

If there isn’t a value for the variable you’re predicting, then there’s no point in having the entry in the final dataset. Check that the maximum and minimum values are within logical bounds for the variable and that all rows are of the same data type.

**Remove duplicate entries**

Having duplicates indicates possible errors in data entry. Before removing all duplicates, ask:

Why are there duplicates in the dataset?

The answer will change your strategy for removing them. For example, if records of a house are updated on the same database based on a unique property identifier, we may find the most recent entry the most relevant. Does this mean that we should simply toss out all the old entries? Does it make sense to use the old records to impute any missing fields?

In another scenario, we might have the gameplay records for players. The most recent entry might be the most up-to-date but would we get richer data by aggregating past records? Do the mechanics of the game make it such that only the most recent or first record matters?

Don’t be so quick to remove duplicates!

**Check missing value percentage per variable**

percent\_missing **=** df**.**isnull()**.**sum() **\*** 100 **/** len(df)percent\_missing**.**sort\_values(ascending**=False**)

If there are too many missing values, you will want to drop the column. But what is the cut-off point?

Generally having more than 50% of the data missing makes the column seem useless BUT before dropping any columns **understand why there is so much data missing**.

The column might only be filled under certain circumstances. For example, if a form asks for dietary restrictions and you select ‘Other’, the form might prompt you to fill another field specifying the restriction. In this case, there will be an extremely high percentage of missing values that contains useful information. This information could be cleaned and merged with another column to retain data.

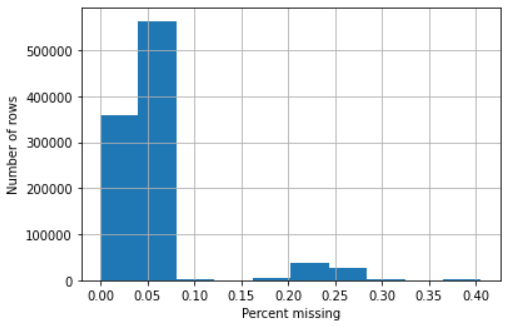
Similarly, if two columns with complementary data have high percentage of missing values, they could be used to accurately impute data. For example, if one column shows the presence or absence of solar water heating for a house and another column shows the percentage of the roof covered by solar panels, we could create a column for the presence of solar panels with low missing values.

Every column of data took work to collect and shows something possibly important. If you decide to drop the column, be clear about why and make sure you’ve thought it through!

**Check missing value percentage per row**

If the entire row has no or very little data, dropping it might be better than filling the entire entry with imputed data later on. Just make sure to question why the row is empty before dropping it!

percent\_missing\_rows = df.isnull().sum(axis=0) / len(df.columns)missing\_plt = percent\_missing\_rows.hist()missing\_plt.set\_xlabel('Percent missing')missing\_plt.set\_ylabel('Number of rows')



Histogram of percentage null per row. [Image from author]

For this dataset, I didn’t end up removing any rows because the survey data I was working with added new fields later on that increased the perceived percentage missing in earlier data.

**Clean by variable type**

We want to make sure each column has the data we expect. If you have many columns, it will be easier to go audit your variables by type (broadly): boolean, datetime, numerical (int/float), categorical and text.

df**.**select\_dtypes(include**=** 'INSERT\_TYPE')**.**columns**.**tolist()

**Boolean**

Standardise all boolean columns to have True/False as objects instead of other forms like Y/N or Yes/No.

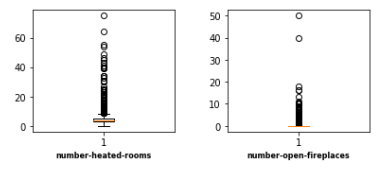
**Datetime**

Convert the column to a datetime object and plot out the dates in a histogram to make sure it is within a logical range.

**Numerical**

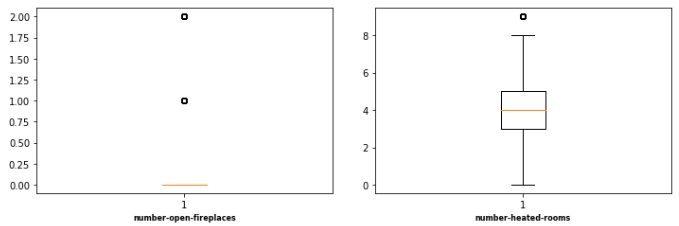
For numerical data, plot boxplots to get a snapshot of the distributions and see which variables have illogical max/min values that should be clipped. But before clipping, make sure you understand what the variables mean.

num\_var **=** df**.**select\_dtypes(include**=** 'number')**.**columns**.**tolist()plt**.**figure(figsize**=**(20,10))  
  
**for** i, var **in** enumerate(num\_var):  
 plt**.**subplot(4,7,1**+**i)  
 plt**.**boxplot(df[var]**.**dropna())  
 plt**.**xlabel(var,fontsize**=**8, weight**=**'bold')  
  
plt**.**tight\_layout()



In this dataset on houses, it doesn’t make sense that a house would have 50 fireplaces and >60 heated rooms. This indicates that these variables should be clipped. [Image from author]

audit\_num **=** df[clip\_var]**.**describe(percentiles **=** [0.01, 0.99])*# Clips the variables with the upper bound of the 99% confidence   
# interval for variables with many outliers variance (no lower clip)*plt**.**figure(figsize**=**(20,10))  
  
**for** i, var **in** enumerate(clip\_var):  
 df[var]**.**clip(lower**=-**9999.0, upper**=**audit\_num[var]['99%'], inplace**=True**)  
 plt**.**subplot(3,4,1**+**i)  
 plt**.**boxplot(df[var]**.**dropna())  
 plt**.**xlabel(var,fontsize**=**8, weight**=**'bold')  
  
plt**.**tight\_layout()



Boxplots after clipping the variables. [Image from author]

**Categorical**

Print all the unique variables for each categorical column and ensure that the values are as they should be. Otherwise, merge columns. If there are too many categories, consider grouping them to reduce complexity.

**Text**

Sometimes you’ll get data in text form that is sort of standardised but mostly a whole mess of different inputs. As much as possible use regex to standardise the entries, and then we can reduce the number of unique values with **CHAID (Chi-square Automatic Interaction Detector)**.

Generally, the algorithm creates a decision tree. It randomly groups unique responses into a node and determines which split in the node brings us closer to accurately predicting the outcome. The metric it uses is Chi-square which is a statistical test that tells us whether two categorical variables are independent. We will take the nodes of the finished tree which have the optimal groupings of unique responses and use them to reduce the number of unique values in the column.

You can read more in-depth about the method [here](https://select-statistics.co.uk/blog/chaid-chi-square-automatic-interaction-detector/). Original code below from [here](https://github.com/datasciencecampus/energy-efficiency/blob/master/01_Code/01_EPC/02_Preprocessing/05_CHAID.ipynb).

**from** CHAID **import** Treechaid\_dict **=** {}  
**for** var **in** cat\_var:  
 *#Set the inputs and outputs*  
 *#The imputs are given as a dictionary along with the type*  
 *#The output must be of string type*  
 *#I have assume all features are nominal, we can change the features dictionary to include the ordinal type*  
 features **=** {var:'nominal'}  
 label **=** 'VARIABLE\_TO\_PREDICT'  
 *#Create the Tree*  
 chaid\_dict[var] **=** {}  
 tree **=** Tree**.**from\_pandas\_df(df, i\_variables **=** features, d\_variable **=** label, alpha\_merge **=** 0.0)  
 *#Loop through all the nodes and enter into a dictionary*  
 print('\n\n\nVariable: %s' **%** var)  
 print('p-value: %f' **%** tree**.**tree\_store[0]**.**split**.**p)  
 print('Chi2: %f' **%** tree**.**tree\_store[0]**.**split**.**score)  
 **for** i **in** range(1, len(tree**.**tree\_store)):  
 count **=** tree**.**tree\_store[i]**.**members[0] **+** tree**.**tree\_store[i]**.**members[1]  
 **if** count **!=** 0:  
 rate **=** tree**.**tree\_store[i]**.**members[1] **/** count  
 print('\nNode %i:\n\tCount = %i\tRate = %f' **%** (i,count,rate))  
 print('\t%s' **%** tree**.**tree\_store[i]**.**choices)  
 chaid\_dict[var]['node' **+** str(i)] **=** tree**.**tree\_store[i]**.**choices

As a heuristic, you can take the groupings as is but it is worth using it as a starting point to see how you might group the unique entries.

**Impute missing data**

You can’t run a model if there is missing data, so here are some ways we can fill in the missing data.

A simple way is to fill the missing data based on other values in the column. If the column has skewed data, take the median (numerical) or mode (non-numeric) so that you are drawing from the majority and don’t end up shifting the distribution. If the column has unskewed data, take the mean for the same reasons!

Another method is called **iterative imputation**, sequentially using the data from each feature to fill in the missing data. We predict the missing values in the feature with the lowest percentage of missing values as if we were solving a regression problem. Then, it uses the cleaned features to predict the feature with the next lowest percentage of missing values until all features have been imputed. See [here](https://machinelearningmastery.com/iterative-imputation-for-missing-values-in-machine-learning/) for a tutorial.

With categorical or text data, you could also treat missing data as a category of data by replacing *np.nan* with ‘MISSING’. Perhaps the very fact that the data is missing might be predictive in itself.

**Encode non-numeric data**

Computations only work on numbers so we need to change our text into numbers through encoding.

If you have ordinal (ordered) data, you can use **label encoding** which converts an alphabetically sorted category into a sequence of numbers. [‘A’, ‘B’, ‘C’] = [1,2,3].

This works best when the order corresponds with the increasing value of the numbers for example: [‘short’, ‘average’, ‘tall’] = [1,2,3]. But when applied to non-ordered data like: [‘apple’, ‘pear’, ‘banana’]=[1,2,3], the model gets the sense that banana > pear > apple, which is not true!

**from** sklearn.preprocessing **import** LabelEncoderlabel\_encoder **=** LabelEncoder()  
df[var] **=** label\_encoder**.**fit\_transform(df[var])

Instead, we use **one-hot encoding** which converts numbers into vectors. So data like: [‘apple’, ‘pear’, ‘banana’] = [[0,0,1],[0,1,0],[1,0,0]]. However, where there is high cardinality (many unique values), these vectors will become very large and take up a lot of memory, so be wary.

*# One hot encode non-ordinal variable***from** sklearn.preprocessing **import** OneHotEncoder   
enc **=** OneHotEncoder(handle\_unknown**=**'ignore')  
df[var] **=** enc**.**fit\_transform(np**.**array(df[var])**.**reshape(**-**1,1))**.**toarray()

**Ready for modelling!**

This was by no means an exhaustive guide for all you could do to clean data. There are more steps you could do depending on your dataset, but this is a pipeline I found helpful at least to start. Please let me know if there are other steps I should add and other functions that might help make the process easier.

For now, at least, go run some models!