EBERHARD KARLS
UNIVERSITÄT
TÜBINGEN

Philosophische Fakultät
Seminar für Sprachwissenschaft

Data Science for Linguists

Session 4: Data Cleaning and Preparation

Johannes Dellert

17 November, 2023

Data Cleaning and Preparation

- in data science, data preparation (cleaning, transforming, and rearranging of real-world data) often comprises the majority of the work; a common figure is about 80% of an analyst's time
- ad hoc processing of data using a variety of tools (plain Python, Perl, R, Java, sed, awk) is still very common in science, but this approach has many disadvantages
- using Pandas in a Jupyter notebook for these tasks has the advantages of
  - ▷ a high-level, flexible, and fast set of tools that is very accessible to other researchers
  - ▷ full documentation of the steps taken to derive the research data from the raw inputs (crucial for transparency and reproducibility!)

Table of Contents

NaN and None in Pandas

- due to their underlying implementation in NumPy, Pandas series represent missing values by means of different **sentinel values**, the choice of which depends on the datatype:
  - ▷ for data with dtype `float64`, Pandas uses the floating-point value `NaN` (Not a Number) which is defined by the ISO standard (accessible via `np.nan`)
  - ▷ for general Python objects (including variable-length strings), it uses the special value `None` from standard Python (which is very slow to compute with!)
  - ▷ because these two options are the only ones available, series of integer datatypes are silently converted into floating point numbers as soon as missing values are inserted
- one of the goals of Pandas is to make working with missing data efficient and smooth
  - ▷ there is well-defined default behaviour for any functions executed on Pandas series with missing values; such values will never cause Pandas to throw an exception, though the defalt behaviour might not be extremely useful in some cases
  - ▷ many methods (such as the ones for descriptive statistics) are implemented to silently exclude missing data per default
  - ▷ it also provides a range of methods which abstract over the somewhat inconsistent underlying implementation of null values

Pandas Nullable Dtypes

- to add support for true integer arrays with missing data, Pandas provides **nullable dtypes** like `pd.Int32` (distinguished from the default dtypes like `pd.int32` by capitalisation)
- the null value in series of these types is represented by `pd.NA`
- the other null values will be normalised without triggering implicit typecasting:

```
In [ ]: pd.Series([1, np.nan, 2, None, pd.NA], dtype='Int32')
Out[ ]: 0      1
        1   <NA>
        2      2
        3   <NA>
        4   <NA>
        dtype: Int32
```

- for more efficient handling of large amounts of string data, there is the specialised extension type `pd.StringDtype()`

Detecting Null Values

- null values of any kind can be detected using the method `data.isnull()`, which returns a Boolean mask over the data, with `True` in all positions where there is a null value
- `data.notnull()` returns a Boolean mask as well, but the opposite of the result of the previous method (`False` in all positions where there is a null value)
- `data.isna()` and `data.notna()` are aliases of `data.isnull()` and `data.notnull()`, both sentinel values are matched by both methods (unlike in the underlying implementation)

```
In [ ]: data = pd.Series([1, np.nan, 'hello', None])
In [ ]: data.isnull()
Out[ ]: 0  False
        1   True
        2  False
        3   True
        dtype: bool
```

Dropping Null Values with `dropna()`

- `data.dropna()` is a convenience method which combines filtering for and then dropping the null values, i.e. a shorthand for `data[data.notna()]`
- if `data` is a `Series`, the result will simply be a shorter series
- if `data` is a `DataFrame`, we can only drop entire rows or columns
  - ▷ by default, `data.dropna()` will drop all rows which contain any null value
  - ▷ to drop columns instead, we can provide the argument `axis=1` or `axis="columns"`
- argument `how="all"` to only drop rows/columns which consist entirely of null values
- for more fine-grained control (e.g. only weeding out the most gappy records), we can also provide an argument `thresh=`*k* to specify that any row/column with at least *k* non-null values will be kept

Filling Null Values with `fillna()`

- `data.fillna(x)` is a convenience method which combines filtering for and then overwriting the null values with a default value `x`, i.e. a shorthand for `data[data.isna()] = x`
- `x` can be a dictionary providing different fill values by column index
- `data.fillna(method="ffill")` specifies a forward fill, i.e. empty values will be replaced by the previous non-null value (which therefore gets propagated forward)
  - ▷ example: a series with data `[1 <NA> 2 <NA> <NA> 3]` will become `[1 1 2 2 2 3]`
- `data.fillna(method="bfill")` specifies a backward fill, i.e. empty values will be replaced by the subsequent non-null value (which therefore gets propagated backward)
  - ▷ example: a series with data `[1 <NA> 2 <NA> <NA> 3]` will become `[1 2 2 3 3 3]`
- if no previous or subseqent value is available, ffill and bfill leave null values at the fringes!
- for a `DataFrame`, we can again switch to propagation through columns by specifying `axis=1`

Table of Contents

Removing Duplicates

- `data.duplicated()` returns a Boolean `Series` indicating whether each row is a duplicate (all values equal to some previous row) or not
- `data.drop_duplicates()` returns a `Series` or `DataFrame` consisting of only those rows in `data` where `data.duplicated()` was `False`
- the `subset` argument allows to provide a list of column indices to specify which columns are relevant for duplicate detection (other columns are allowed to have different values)
- if the `subset` argument is used, the first variant of each duplicate will be used for the values in the irrelevant columns, `keep="last"` changes this

Replacing Values

- `data.replace(oldvals, newvals)` substitutes a set of values by replacements
  - ▷ `data.replace(-1, 0)` sets all cells with value -1 to 0
  - ▷ `data.replace([-2, -1], 0)` sets all cells with value -2 or -1 to 0
  - ▷ `data.replace([-2, -1], [-1, 0])` sets cells with value -2 or -1, and all with -1 to 0
- `data.replace(dictionary)` replaces each occurrence of a key with its value
- more complex element-wise transformation can be implemented as a function (e.g. `transform_value(x)`), and then executed on every cell by a call to `data.map(transform_value)`; this works with anonymous functions (`lambda`) as well

Renaming Axis Indexes

- the axes can be modified in place by executing the `map` method of their indices:
  - ▷ `data.columns = data.columns.map(str.title)`
  - ▷ `data.index = data.index.map(lambda x: x[:4].upper())`
- `data.rename()` allows to create a transformed version of a dataset without modifying the original (simple example: `data.rename(index=str.title, columns=str.upper)`)

Table of Contents

Discretisation and Binning

- continuous data is often discretised or otherwise separated into bins for analyis
- `pd.cut(data, k)` returns a `Categorical` object which describes the *k* equal-length bins computed on the basis of the minimum and maximum values within the data
- `pd.cut(data, cutoffs)` returns a `Categorical` object which describes the bins computed from the data based on the specified cutoff values between the bins
- `pd.qcut(data, k)` bins the data into *k* quantiles (equally sized bins)
- `pd.qcut(data, quantiles)` bins the data into the provided quantiles (between 0 and 1)
- the argument `labels=group_names` allows to override the default bin names
- a `Categorical` object `cats` has the following key applications:
  - ▷ `cats.codes` returns an array containing the bin index for each datapoint
  - ▷ `cats.categories` shows an `IntervalIndex` object representing the bins
  - ▷ `pd.value_counts(cats)` renders the counts of datapoints in each bin
- these functions will have central importance in Session 7 (aggregation and grouping)

Table of Contents

Detecting and Filtering Outliers

- outlier detection and filtering is typically performed by combinations of simple array operations
- for a normally distributed `Series` of data, we might define our outliers as all rows where the value exceeds 3 in absolute value: `data[data.abs() > 3]`
- in a `DataFrame` full of normally distibuted values, we might be interested in the rows were any of the columns has such a value: `data[(data.abs() > 3).any(axis=”columns”)]` (Boolean DataFrame generated by the comparison, on which we apply the `any()` method)
- to remove the outliers by capping values to the range [-3, 3], we can just do `data[data.abs() > 3] = np.sign(data) * 3`

Table of Contents

Permutation and Random Sampling

- permutations are sampled using `smp = np.random.permutation(k)`
- `data.take(smp)` (= `iloc`-based indexing) is then a random permutation of the first *k* lines
- `data.sample(n=k)` selects a random subset of *k* rows without replacement
- `data.sample(n=k, replace=True)` samples *k* rows with replacement

Table of Contents

Vectorised String Operations: Equivalents of Basic Methods

- cleaning up a messy dataset often requires a lot of string manipulation, but simple element-wise application using `data.map()` will fail on the null values
- the Pandas `Series` offers array-oriented and null-aware string operations which are accessible via the `str` attribut; here is a small sample:
  - ▷ `data.str.count` to count occurrences of a pattern
  - ▷ `data.str.contains(s)` returns a Boolean mask with the result of calls to `s in value` on each cell value, mixed with NaN values wherever a value was missing
  - ▷ `data.str.len` to compute the length of each string
  - ▷ `data.str.strip` to trim whitespace (including newlines) from both sides

Vectorised String Operations: Regular Expressions

- there are also vectorised and null-aware versions of the regex capabilities:
  - `matches = data.str.findall(r”somePattern”)`
  - `first_matches = matches.str.get(1)`
- `data.str.extract(pattern)` returns the captured groups of the regular expression as a new DataFrame

Vectorised String Operations: Miscellaneous Methods

- several methods emulate the capabilities of Python string operations:
  - `data.str.cat` for element-wise concatenation (with optional delimiter)
  - `data.str.get` to index each string
  - `data.str.repeat` to repeat each string
  - `data.str.slice` for extracting slices from each string
- `data.str.get` and `data.str.slice` are also available through the normal indexing syntax (`data.str[i]` and `data.str[i:j]`)

Table of Contents

Assignment 4: Tasks

1) Read up on the CoNLL-U format for dependency treebanks.
2) Create a new Jupyter notebook and import `pandas`. Load the development set of the Basque UD corpus from the file `eu_bdt-ud-dev.conllu` into a `DataFrame` object, filtering out lines which are empty or start with #. Specify the CoNLL-U field names as a column index.
3) Convert all values consisting of an underscore into an approprate missing data type, and remove all columns where more than 80% of the values are empty.
4) Reduce the dataset to all rows representing forms of the auxiliary *izan* ("to be").
5) Apply vectorised string operations which involve regular expressions to convert the morphological features into a more useful format (one new column per feature, feature values in each row; example: column `Number[abs]` with values `Sing` and `Plur`).
6) Query the database to see whether there is any syncretism in the paradigm of *izan*. (i.e. same form, different morphological features) [Hint: duplicate detection on different subsets]
7) How often does each form of *izan* occur in this development set? Create a new dataframe consisting of the form, the columns with the morphological features, and these counts.
8) Split the forms of *izan* attested in the corpus into ten bins of equal size, so that each bin groups together forms of roughly equal frequency. Detect and remove outliers if necessary.

Preliminary Course Plan

1 **27/10 IPython and Jupyter**
2 **03/11 Introduction to NumPy**
3 **10/11 Pandas and Data Frames**
4 **17/11 Data Cleaning and Preparation**
5 24/11 Linguistic Preprocessing
6 01/12 Data Wrangling: Join, Combine, Reshape
7 08/12 Data Aggregation and Grouping
8 15/12 Visualisation with Seaborn
9 22/12 Modeling and Prediction
10 12/01 Classification
11 19/01 Clustering
12 26/01 Pattern Extraction and Density Estimation
13 02/02 Statistical Inference
14 09/02 Data Science Projects

Questions

Questions?

Comments?

Suggestions?