Python Cheat Sheet
Dec 2019
Dennis Gluesenkamp

Data Cleansing & Pre-Processing (1/2)

Twitter: @dgluesen
GitHub: @dgluesen
Instagram: @dgluesen

1. Libraries & Data Import

Only two packages are required for data cleansing at first. The next step is to import the file.

```
>>> import numpy as np
>>> import pandas as pd

>>> df1 = pd.read_csv('file.csv')
>>> df2 = pd.read_excel('file.xlsx',
...                     sheet_name = 0,
...                     header = 1,
...                     na_values = ['#NV', 'Err'])
```

2. Inspect Data Set

Draw a random sample of rows in order to get an idea of the content of the data frame.

```
>>> a = pd.DataFrame({
...     'C0': ['1', '2', '3', '4'],
...     'C1': [.321, .654, .987, .741],
...     'C2': ['abc', 'def', 'ghi', 'jkl']},
...     index=['obj0', 'obj1', 'obj2', 'obj3'])
>>> a.sample(n=2)
        C0      C1      C2
obj3     4   0.741     jkl
obj1     2   0.654     def
>>> a. describe(include='all')
        C0      C1      C2
count    4   4.000       4
unique   4     NaN       4
top      3     NaN     jkl
freq     1     NaN       1
...    ...     ...     ...
```

3. Data Types

If columns are not correctly recognized as numeric data types, this must be corrected.

```
>>> a.dtypes # C0 should be int64 but is object
C0      object
C1     float64
C2      object
dtype: object
>>> a['C0'] = pd.to_numeric(a['C0'])
```

4. Missings

```
>>> b = pd.DataFrame(
...    {'Name': ['Anna', 'Bill', 'Charly', 'Diana'],
...     'Job': ['Data Scientist',
...             'DATA SCIENTIST',
...             'Data Engineer ',
...             ' Data Science'],
...     'Age': [26, np.nan, 30, 32],
...     'CompCar': [np.nan, np.nan, np.nan, 'BMW']},
...    index=['emp01', 'emp02', 'emp03', 'emp04'])
```

4.1 Missing Statistics

How much columns contain missing values (% of columns)?
How much missing data is in the data frame (% of cells)?

```
>>> len(b.columns[b.isna().any()])/len(b.columns)
0.5
>>> b.isnull().sum().sum()/np.product(b.shape)
0.25
```

4.2 Missing Types

It must be considered which origin missing values (probably) have. A distinction must be made between (a) non-existent and (b) non-recorded values. The processing is different here.

(a) Non-existent
Example: Employee has no company car, so no manufacturer is recorded.
Handling: Keep NaN; maybe replaced by specific value, e.g. (0, 'nothing')
(b) Non-recorded
Example: No information about age of employee
Handling: Imputation

4.3 Imputation

1) Use mean/median of column (numerical features only)

```
>>> b['Age'] = b['Age'].fillna(b['Age'].mean())
```

2) Use most frequent value in column (numerical and categorical features)

```
>>> b['Age'] = b['Age'].fillna(b['Age'].mode()[0])
```

3) Hot-deck imputation; use a random value of one of the other objects (numerical and categorical features)

```
>>> b.loc[b['Age'].isna(), ['Age']] = (
...    np.random.choice(np.unique(b['Age'].dropna()),
...                     b['Age'].isna().sum())
...    )
```

More sophisticated approaches are among others:

4) Regression and stochastic regression imputation
5) Imputation via k-NN
6) Deep Learning imputation
7) Multivariate Imputation by Chained Equations (MICE)

Some methods (e.g. stochastic regression, hot-deck, MICE) can be used with multiple imputation. Here, the uncertainty is incorporated by using not only a single but multiple imputation values.

5. Inconsistencies

The meaning of categorical data can be corrupted by incorrect spelling or useless characters. These irregularities must be removed.

5.1 Upper and lower cases

```
>>> b['Job'] = b['Job'].str.lower()
```

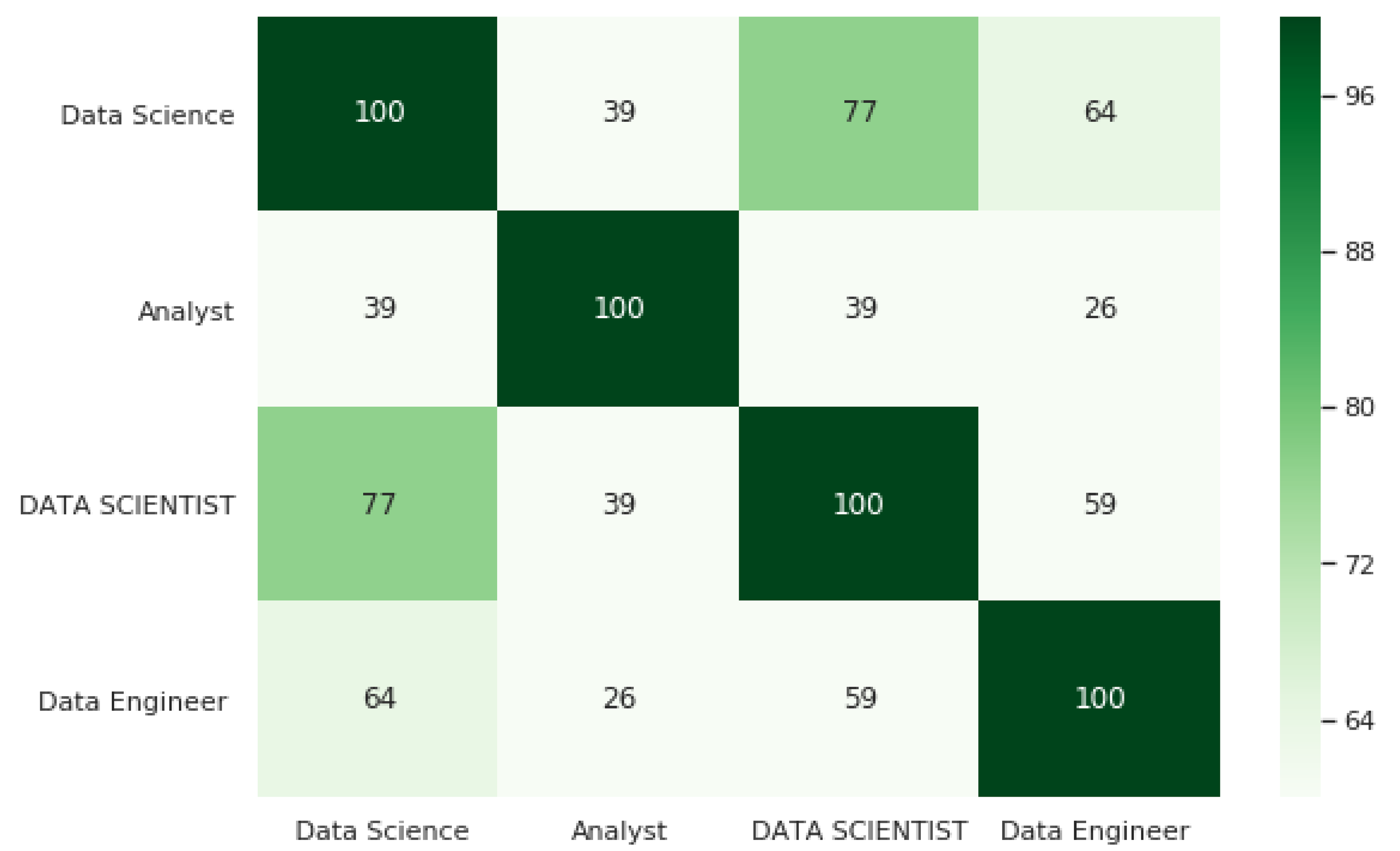
5.2 Leading and trailing white spaces

```
>>> b['Job'] = b['Job'].str.strip()
```

5.3 Notation similarities

The meaning of expressions can be the same despite different spellings. Here it is possible to make a quantitative comparison of strings and to visualize the permutations. Then it must be decided whether and if so how certain values are adapted/overwritten.

```
>>> from fuzzywuzzy import process
>>> import seaborn as sns; sns.set();
>>> def SpellCheck(df, col):
...     instances=np.unique(df[col])
...     dt=[(col, object),('score', int)]
...     result=np.zeros(shape=(len(instances),
...                            len(instances)))
...     for i, inst in enumerate(instances):
...         result[i]=np.sort(np.array(
...             process.extract(inst,instances),
...             dtype = dt),
...                    order = col)['score']
...     result=pd.DataFrame(result,
...                         index=instances,
...                         columns=instances)
...     return result
>>> # Adjust vmin and vmax to accepted similarities
>>> ax = sns.heatmap(mapping, vmin=60, vmax=100,
...                  annot=True, fmt=".0f",
...                  cmap="Reds")
```



Data Cleansing & Pre-Processing (2/2)

Twitter: @dgluesen
GitHub: @dgluesen
Instagram: @dgluesen

6. Dates

The pandas inclusive method to_datetime can convert entire columns. It detects standard date formats, but a format string can be optionally provided.

```
>>> c = pd.DataFrame(
...     {'Name': ['Anna', 'Bill', 'Charly'],
...      'DOB': ['25.08.1986',
...              '21.01.1988',
...              '10.12.1992'],
...      'Entry': ['06/01/2016',
...                '08/01/2015',
...                '04/01/2018'],
...      'Term': ['18-12-31',
...               '19-04-30',
...               '19-07-31']},
...     index=['emp01', 'emp02', 'emp03'])
>>> pd.to_datetime(c['DOB])
emp01   1986-08-25
emp02   1988-01-21
emp03   1992-10-12
Name: DateOfBirth, dtype: datetime64[ns]
>>> c['Entry]=pd.to_datetime(c['Entry'],
...                          format='%m/%d/%Y)
>>> c['Term']=pd.to_datetime(c['Term'],
...                          format='%y-%m-%d')
```

Plain calculations lead to timedelta objects in order to get time differences.

```
>>> c['Duration']=c['Term']-c['Entry']
>>> c['Duration']
emp01    943 days
emp02   1368 days
emp03    486 days
Name: Duration, dtype: timedelta64[ns]
```

7. Renaming Columns

If columns are not correctly recognized as numeric data types, this must be corrected.

```
>>> c.rename(columns={'DOB':'dateofbirth',
...                   'Entry':'entry',
...                   'Term':'termination',
...                   'Duration':'duration'},
...          inplace=True)
```

8. Column Selection

```
>>> c.loc[:,'entry':'duration']   # range of names
>>> c.loc[:,['entry','duration']] # explicit names
>>> c.iloc[:,1:3]                 # range of indices
>>> c.iloc[:,[1,3]]               # explicit indices
```

9. Joining DataFrames

```
>>> pd.merge(b,c,how='left',on='Name')
```

If tables are to be joined over several and also differently named columns, this is done by specifying the column names in an array.

```
>>> pd.merge(df_1,df_2,how='left',
...          left_on=['col1','col2'],
...          right_on=['ColumnA','ColumnB'])
```

10. Feature Encoding

Feature encoding transforms categorical data to numerical value in order to make them processable for an algorithm. Two types of categorical data are of main interest here:

(a) Nominal data
Different discrete categorical values without any rank/order or metric.
Examples: Colors of cars, genre of movies
(b) Ordinal data
Series of discrete categorical values with a defined order
Examples: Hierarchy levels in an organization, version of a technical gadget

10.1 One-hot-encoding

Unique column for each occurence of a categorical value per feature; three possible packages are e.g.:

```
>>> d = b['Name']
>>> pd.get_dummies(d) # via pandas
>>> # ---------------------------------------
>>> from sklearn import preprocessing
>>> ohenc = preprocessing.OneHotEncoder()
>>> ohenc.fit(np.array(d).reshape(-1,1))
>>> ohenc.transform(np.array(d)
...                 .reshape(-1,1)).toarray()
>>> # ---------------------------------------
>>> import category_encoders as ce
>>> ohencoder = (ce.OneHotEncoder(cols=['Name'])
...              .fit(d))
>>> ohencoder.transform(d)
```

10.2 Ordinal encoding

Transformation of categorical values to sequence of integers

```
>>> labelenc = preprocessing.LabelEncoder()
>>> labelenc.fit(d)
>>> labelenc.transform(d)
>>> # ---------------------------------------
>>> ordencoder = (ce.OrdinalEncoder(cols=['Name'])
...               .fit(d))
>>> ordencoder.transform(d)
```

For other classical encoders or further ones like contrast or bayesian encoders refer to the package documentations.