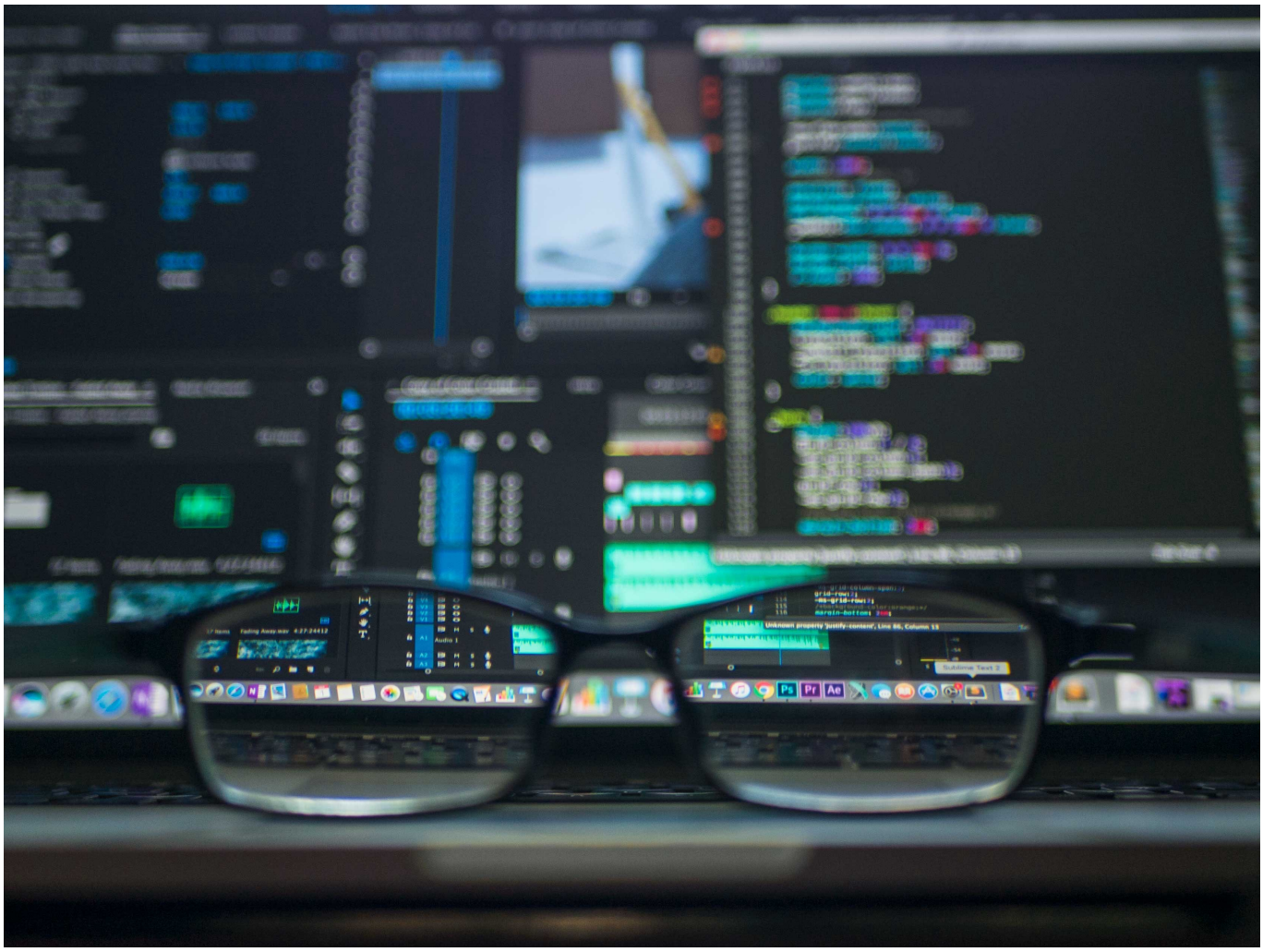

All you want to know about preprocessing: Data preparation

This is an introduction part, where we are going to discuss how to check and prepare your data for further preprocessing.

Maksym Balatsko Follow

May 29, 2019 · 6 min read ★

Nowadays, almost all ML/data mining projects workflow run on a standard `CRISP-DM` (Cross-industry standard process for data mining) or its IBM enhance `ASUM-DM` (Analytics

Solutions Unified Method for Data Mining/Predictive Analytics). The longest and the most important step in this workflow is **Data preparation/preprocessing**, which approximately takes 70% of the time. This step is important because in most situations data provided by the customer has a bad quality or just cannot be directly fed to some kind of ML model. My favorite byword, that I'll mention in all my posts, concerning data preprocessing, says: `Garbage in, garbage out (GIGO)` . In other words, if you feed your model with miserable data, don't expect it to perform well. In this post we are going to discuss:

1. `Data types`
2. `Data validation`
3. `Handling dates`
4. `Handling nominal and ordinal categorical values`

In the next posts we are going to talk about more advanced preprocessing techniques:

1. Data cleaning and standardization: `Normalization and standartization` , `Handling missing data` , `Handling outliers`
2. Feature selection and dataset balancing: `Dataset balancing` , `Feature extraction` , `Feature selection` .

This post series represents the usual preprocessing flow order, but, in fact, all these parts are divided in such a way to be independent and not to require the knowledge of the previous parts.

This is an introduction part, where we are going to discuss how to check and prepare your data for further preprocessing.

Data types

To start, let's define what data types exist and what measurement scales they have:

`Numeric`

1. `Discrete` - integer values. Example: `number of products bought in the shop`

2. `Continuos` - any value in some admissable range (float, double). Example: `average length of words in text`

`Categorical`

The variable value selected from a predefined number of categories

1. `Ordinal` - categories could be meaningfully ordered. Example: `grade (A, B, C, D, E, F)`
2. `Nominal` - categories don't have any order. Example: `religion (Christian, Muslim, Hindu, etc.)`
3. `Dichotomous/Binary` - the special case of nominal, with only 2 possible categories. Example: `gender (male, female)`

`Date`

String, python datetime, timestamp. Example: `12.12.2012`

`Text`

Multidimensional data, more about text preprocessing see in my previous post

`Images`

Multidimensional data, more about image preprocessing see in my next posts

`Time series`

Data points indexed in the time order, more about time series preprocessing see in my next posts.

Data validation

The first step is the simplest and the most obvious: you have to investigate and validate your data. To be able to validate the data you have to have a deep understanding of your data. Easy rule: `Don't dismiss the description of the dataset` . Validation step consists of:

`Data type and data representation consistency check`

Same things have to be represented in the same way and in the same format. Examples:

1. Dates have the same format. Several times in my practice I've got data where a part of dates was in American format, the other part in European.
2. Integers are really integers, not strings or floats
3. Categorical data doesn't have duplicates because of whitespaces, lower/upper cases
4. Other data representations don't contain an error

`Data domain check`

Data is in range of permissible values. Example: numerical variables are in admissable (min, max) range.

`Data integrity check`

Check permitted relationships and fulfillment of the constraints. Examples:

1. Check name titles with sex, age of birth with age
2. Historical data have the right chronology. Delivery after purchase, Bank account opening before the first payment, etc.
3. The actions are made by allowed entities. The mortgage could be approved only for people older than 18 years old, etc.

Ok, we have found some errors, what could we do?

1. `Correct` them, if you are sure, what the problem is, or consult with the specialist or data provider if possible.
2. `Discard` samples with errors, in many cases it is a good choice because you aren't able to fulfill 1.
3. `Do nothing`, this, of course, could cause undesired effects in future steps.

Handling dates

Different systems stores dates in different formats: `11.12.2019`, `2016-02-12`, `Sep 24, 2003` etc. But for building models on dates data, we need to somehow convert it to a numeric format.

To start, I'll show you an example of how to convert a date string into python `datetime type` , which is much more convenient for further steps. The example is demonstrated on pandas dataframe. Let's assume that `date_string` column contains dates in strings:

```
# Converts date string column to python datetime type
# `infer_datetime_format=True` says method to guess date format from
string

df['datetime'] = pd.to_datetime(df['date_string'],
infer_datetime_format=True)

# Converts date string column to python datetime type
# `format` argument specifies the format of date to parse, fails on
errors

df['datetime'] = pd.to_datetime(df['date_string'], format='%Y.%m.%d')
```

Frequently, just the year (YYYY) is sufficient. But if we want to store months, days or even more detailed data, our numeric format has to fulfill 1 sufficient constraint, it has to save intervals, it means that for example, Monday — Friday in one week has to have the same difference as 1. — 5. of any month. So `YYYYMMDD` format will be not an option, because the last day of the month and the first day of the next month have a bigger distance than the first and the second day of the month. Actually, there are 4 most common methods to transform date to numeric format:

`Unix timestamp`

Number of seconds since 1970

Pros:

1. perfectly preserve intervals
2. good if hours, minutes and seconds matters

Cons:

1. values are non-obvious
2. don’t help intuition and knowledge discovery

3. harder to verify, easier to make an error

Converting `datetime` column to timestamp in pandas:

```
# Coverts column in python datetime type to timestamp
df['timestamp'] = df['datetime'].values.astype(np.int64) // 10 ** 9
```

`KSP date format`

$$KSP\_date = year + \frac{days\_from\_1\_Jan - 0.5}{365 + 1\_if\_leap\_year}$$

Pros:

1. the year and quarter are obvious
2. easy intuition and knowledge discovery
3. can be extended to include time

Cons:

1. preserves intervals (almost)

Converting `python datetime` column to KSP format in pandas:

```
import datetime as dt
import calendar

def to_ksp_format(datetime):
    year = datetime.year
    day_from_jan_1 = (datetime - dt.datetime(year, 1, 1)).days
    is_leap_year = int(calendar.isleap(year))
    return year + (day_from_jan_1 - 0.5) / (365 + is_leap_year)

df['ksp_date'] = df['datetime'].apply(to_ksp_format)
```

`Divide into several features`

Year, month, days, etc.

Cons:

1. perfectly preserve intervals
2. easy intuition and knowledge discovery

Pros:

1. more dimensions you add, more complex your model could get, but it is not always bad.

`Construct new feature`

Construct a new feature based on date features. For example:

`date of birth` -> `age`

`date order created` and `date order delivered` -> `time to delivery`.

Cons:

1. easy intuition and knowledge discovery

Pros:

1. manual feature construction might lead to important information loss

Handling categorical values

Flashback: `Categorical` - the variable value selected from a predefined number of categories. Categorical values, like any other non-numeric types, has also to be converted into numeric values. How to do it right?

`Ordinal`

Categories could be meaningfully ordered. Can be converted into numeric values saving its natural order. Grades: `A+ - 4.0`, `A- - 3.7`, `B+ - 3.3`, `B - 3.0`, etc.

Demonstration in pandas:

```
grades = {
    'A+': 4.0,
    'A-': 3.7,
    'B+': 3.3,
    'B' : 3.0
}
df['grade_numeric'] = df['grade'].apply(lambda x: grades[x])
```

`Dichotomous/Binary`

Only one of two possible categories. In this case, you can convert values into indicator values `1/0`. For example: `Male - 1` or `Female - 0`, or you can do it oppositely.

Demonstration in pandas:

```
df['gender_indicator'] = df['gender'].apply(lambda x: int(x.lower()
== 'Male'))
```

`Nominal`

One or more of all possible categories. In this case, `One hot encoding` have to be used. This method assumes creating an indicator value for every category(`1` - the sample is in the category, `0` - if not). This method is applicable also for `Dichotomous/Binary` categorical values. **NEVER USE ORDINAL REPRESENTATION FOR NOMINAL VALUES**, it would cause terrible side effects and your model will not be able to handle the categorical feature in the right way.

Demonstration in pandas:

```
# Pandas `.get_dummies()` method
df = pd.concat([df, pd.get_dummies(df['category'],
prefix='category')],axis=1)

# now drop the original 'category' column (you don't need it anymore)
df.drop(['category'],axis=1, inplace=True)
```

Demonstration in sklearn and pandas:

```
from sklearn.preprocessing import LabelEncoder, OneHotEncoder

prefix = 'category'

ohe = OneHotEncoder(sparse=False)
ohe = ohe.fit(df[['category']])
onehot_encoded = ohe.transform(df[['category']])

features_names_prefixed = [ f"{prefix}_{category}" for category in
onehot_encoder.categories_[0]]

df = pd.concat([df, pd.DataFrame(onehot_encoded,
columns=features_names_prefixed)], axis=1)

# now drop the original 'category' column (you don't need it anymore)
df.drop(['category'],axis=1, inplace=True)
```

I hope you'll like my post. Feel free to ask questions in comments.

P.S. These are very and very basic and simple things, but they are very important in practice. Much more interesting stuff is coming in the next posts!

Data Science Data Preprocessing Data Preparation Categorical Data Data Cleaning

About Help Legal