universidad de león

Apuntes Curso Data Cleaning

Missing Values

The first thing to do when you get a new dataset is take a look at some of it. In this case, let's see if there are any missing values, which will be reprsented with NaN or None.

```
nfl_data.head()
```

How many missing data points do we have?

```
# get the number of missing data points per column
missing_values_count = nfl_data.isnull().sum()

# look at the # of missing points in the first ten columns
missing_values_count[0:10]
```

```
Date                0
GameID              0
Drive               0
qtr                 0
down            61154
time              224
TimeUnder           0
TimeSecs          224
PlayTimeDiff      444
SideofField       528
dtype: int64
```

That seems like a lot! It might be helpful to see what percentage of the values in our dataset were missing to give us a better sense of the scale of this problem.

```
# how many total missing values do we have?
total_cells = np.product(nfl_data.shape)
total_missing = missing_values_count.sum()

# percent of data that is missing
percent_missing = (total_missing/total_cells) * 100
print(percent_missing)
```

Almost a quarter of the cells in this dataset are empty!

Figure out why the data is missing¶

This is the point at which we get into the part of data science that I like to call "data intution", by which I mean "really looking at your data and trying to figure out why it is the way it is and how that will affect your analysis".

For dealing with missing values, you'll need to use your intution to figure out why the value is missing.

Is this value missing because it wasn't recorded or because it doesn't exist?

If a value is missing becuase it doesn't exist then it doesn't make sense to try and guess what it might be. These values you probably do want to keep as NaN.

On the other hand, if a value is missing because it wasn't recorded, then you can try to guess what it might have been based on the other values in that column and row. This is called **imputation.**

Let's work through an example. "TimesSec" has a lot of missing values in it.

By looking at the documentation, I can see that this column has information on the number of seconds left in the game when the play was made. This means that these values are probably missing because they were not recorded. So, it would make sense for us to try and guess what they should be rather than just leaving them as NA's.

On the other hand, there are other fields, like "PenalizedTeam" that also have lot of missing fields. In this case, though, the field is missing because if there was no penalty. For this column, it would make more sense to either leave it empty or to add a third value like "neither" and use that to replace the NA's.

Drop missing values

If you're in a hurry or don't have a reason to figure out why your values are missing, one option you have is to just remove any rows or columns that contain missing values.

If you're sure you want to drop rows with missing values, pandas does have a handy function, dropna() to help you do this.

```
# remove all the rows that contain a missing value
nfl_data.dropna()
```

It looks like that's removed all our data! This is because every row in our dataset had at least one missing value. We might have better luck removing all the columns that have at least one missing value instead.

```
# remove all columns with at least one missing value
columns_with_na_dropped = nfl_data.dropna(axis=1)

# just how much data did we lose?
print("Columns in original dataset: %d \n" % nfl_data.shape[1])
print("Columns with na's dropped: %d" % columns_with_na_dropped.shape[1])
```

universidad de león

Filling in missing values automatically¶

Another option is to try and fill in the missing values.

We can use the Panda's fillna() function to fill in missing values in a dataframe for us.

One option we have is to specify what we want the NaN values to be replaced with. Here, I'm saying that I would like to replace all the NaN values with 0.

```
# replace all NA's with 0
nfl_data.fillna(0)
```

I could also be a bit more savvy and replace missing values with whatever value comes directly after it in the same column. (This makes a lot of sense for datasets where the observations have some sort of logical order to them.)

```
# replace all NA's the value that comes directly after it in the same column,
# then replace all the remaining na's with 0
nfl_data.fillna(method='bfill', axis=0).fillna(0)
```

Scaling and Normalization

```
# modules we'll use
import pandas as pd
import numpy as np

# for Box-Cox Transformation
from scipy import stats

# for min_max scaling
from mlxtend.preprocessing import minmax_scaling

# plotting modules
import seaborn as sns
import matplotlib.pyplot as plt

# set seed for reproducibility
np.random.seed(0)
```

- in **scaling**, you're changing the range of your data
- in **normalization**, you're changing the shape of the distribution of your data.

universidad de león

Scaling¶

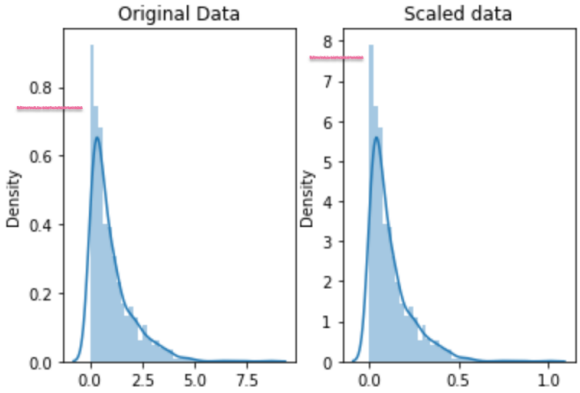
This means that you're transforming your data so that it fits within a specific scale, like 0-100 or 0-1. You want to scale data when you're using methods based on measures of how far apart data points are, like support vector machines (SVM) or k-nearest neighbors (KNN).

For example, you might be looking at the prices of some products in both Yen and US Dollars. One US Dollar is worth about 100 Yen, but if you don't scale your prices, methods like SVM or KNN will consider a difference in price of 1 Yen as important as a difference of 1 US Dollar! This clearly doesn't fit with our intuitions of the world.

```
# generate 1000 data points randomly drawn from an exponential distribution
original_data = np.random.exponential(size=1000)

# mix-max scale the data between 0 and 1
scaled_data = minmax_scaling(original_data, columns=[0])

# plot both together to compare
```



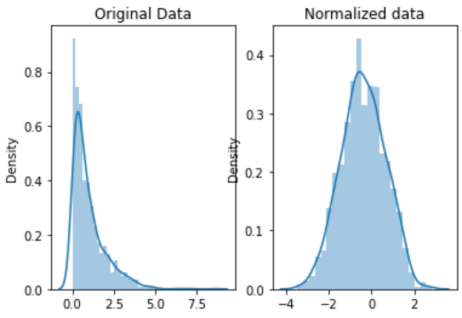
Normalization¶

Scaling just changes the range of your data. Normalization is a more radical transformation. The point of normalization is to change your observations so that they can be described as a normal distribution.

The method we're using to normalize here is called the Box-Cox Transformation.

```
# normalize the exponential data with boxcox
normalized_data = stats.boxcox(original_data)

# plot both together to compare
```



*Notice that the shape of our data has changed.

Parsing Dates

```
import datetime

print(landslides['date'].head())
```

But just because I can tell that these are dates doesn't mean that Python knows that they're dates. Notice that at the bottom of the output of head(), you can see that it says that the data type of this column is “object".

```
0     3/2/07
1    3/22/07
2     4/6/07
3    4/14/07
4    4/15/07
Name: date, dtype: object
```

If you check the pandas dtype documentation, you'll notice that there's also a specific datetime64 dtypes. Because the dtype of our column is object rather than datetime64, we can tell that Python doesn't know that this column contains dates.

Convert our date columns to datetime¶

Now that we know that our date column isn't being recognized as a date, it's time to convert it so that it is recognized as a date.

This is called "parsing dates" because we're taking in a string and identifying its component parts.

There are lots of possible parts of a date, but the most common are %d for day, %m for month, %y for a two-digit year and %Y for a four digit year.

Some examples:

- **1/17/07 has the format "%m/%d/%y"**
- **17-1-2007 has the format "%d-%m-%Y"**

Looking back up at the head of the "date" column in the landslides dataset, we can see that it's in the format "month/day/two-digit year".

```
# create a new column, date_parsed, with the parsed dates
landslides['date_parsed'] = pd.to_datetime(landslides['date'],
                                           format="%m/%d/%y")
```

Now when I check the first few rows of the new column, I can see that the dtype is datetime64. I can also see that my dates have been slightly rearranged so that they fit the default order datetime objects (year-month-day).

```
# print the first few rows
landslides['date_parsed'].head()
```

```
0   2007-03-02
1   2007-03-22
2   2007-04-06
3   2007-04-14
4   2007-04-15
Name: date_parsed, dtype: datetime64[ns]
```

⇒ **What if I run into an error with multiple date formats?** While we're specifying the date format here, sometimes you'll run into an error when there are multiple date formats in a single column. If that happens, you have pandas to try to infer what the right date format should be.

```
landslides['date_parsed'] = pd.to_datetime(landslides['Date'],
                                           infer_datetime_format=True)
```

⇒ **Why don't you always use** infer_datetime_format = True? There are two big reasons not to always have pandas guess the time format. The first is that pandas won't always been able to figure out the correct date format. The second is that it's much slower.

universidad de león

Select the day of the month¶

Now that we have a column of parsed dates, we can extract information

```
# get the day of the month from the date_parsed column
day_of_month_landslides = landslides['date_parsed'].dt.day
```

Character Encodings

```
# helpful character encoding module
import chardet
```

Character encodings are specific sets of rules for mapping from raw binary byte strings (that look like this: 0110100001101001) to characters that make up human-readable text (like “hi").

Character encoding mismatches are less common today than they used to be, but it's definitely still a problem. There are lots of different character encodings, but the main one you need to know is UTF-8.

All Python code is in UTF-8 and, ideally, all your data should be as well.

The best time to convert non UTF-8 input into UTF-8 is when you read in files.

Most files you'll encounter will probably be encoded with UTF-8. This is what Python expects by default, so most of the time you won't run into problems. However, sometimes you'll get an error.

Notice that we get the same UnicodeDecodeError we got when we tried to decode UTF-8 bytes as if they were ASCII! We don't know what encoding it actually is though.

One way to figure it out is to try and test a bunch of different character encodings. A better way, though, is to use the chardet module to try and automatically guess what the right encoding is.

```
# look at the first ten thousand bytes to guess the character encoding
with open("../input/kickstarter-projects/ks-projects-201801.csv", 'rb') as
rawdata:
result = chardet.detect(rawdata.read(10000))

# check what the character encoding might be
print(result)
```

So chardet is 73% confidence that the right encoding is “Windows-1252".

```
# read in the file with the encoding detected by chardet
kickstarter_2016 = pd.read_csv("../input/kickstarter-projects/ks-projects-201612.csv", encoding='Windows-1252')
```

Saving your files with UTF-8 encoding¶

Finally, once you've gone through all the trouble of getting your file into UTF-8, you'll probably want to keep it that way. The easiest way to do that is to save your files with UTF-8 encoding. The good news is, since UTF-8 is the standard encoding in Python.

```
# save our file (will be saved as UTF-8 by default!)
kickstarter_2016.to_csv(“ks-projects-201801-utf8.csv")
```

Inconsistent Data Entry

```
# helpful modules
import fuzzywuzzy
from fuzzywuzzy import process
import chardet

# get all the unique values in the 'Country' column
countries = professors['Country'].unique()

# sort them alphabetically and then take a closer look
countries.sort()
```

Just looking at this, I can see some problems due to inconsistent data entry: ' Germany', and 'germany', for example, or ' New Zealand' and 'New Zealand'.

The first thing I'm going to do is make everything lower case and remove any white spaces at the beginning and end of cells. Inconsistencies in capitalizations and trailing white spaces are very common in text data and you can fix a good 80% of your text data entry inconsistencies by doing this.

```
# convert to lower case
professors['Country'] = professors['Country'].str.lower()
# remove trailing white spaces
professors['Country'] = professors[‘Country'].str.strip()
```

universidad de león

Use fuzzy matching to correct inconsistent data entry

'southkorea' and 'south korea' should be the same.

We're going to use the fuzzywuzzy package to help identify which strings are closest to each other.

Fuzzy matching: The process of automatically finding text strings that are very similar to the target string. In general, a string is considered "closer" to another one the fewer characters you'd need to change if you were transforming one string into another.

Fuzzywuzzy returns a ratio given two strings. The closer the ratio is to 100, the smaller the edit distance between the two strings.

```
# get the top 10 closest matches to "south korea"
matches = fuzzywuzzy.process.extract("south korea", countries, limit=10,
    scorer=fuzzywuzzy.fuzz.token_sort_ratio)
```

We can see that two of the items in the cities are very close to "south korea": "south korea" and "southkorea". Let's replace all rows in our "Country" column that have a ratio of > 47 with "south korea".

To do this, I'm going to write a function.

```
def replace_matches_in_column(df, column, string_to_match, min_ratio = 47):
    # get a list of unique strings
    strings = df[column].unique()

    # get the top 10 closest matches to our input string
    matches = fuzzywuzzy.process.extract(string_to_match, strings,
                                         limit=10, scorer=fuzzywuzzy.fuzz.token_sort_ratio)

    # only get matches with a ratio > 90
    close_matches = [matches[0] for matches in matches if matches[1] >= min_ratio]

    # get the rows of all the close matches in our dataframe
    rows_with_matches = df[column].isin(close_matches)

    # replace all rows with close matches with the input matches
    df.loc[rows_with_matches, column] = string_to_match
```

```
# use the function we just wrote to replace close
replace_matches_in_column(df=professors, column='Country',
                          string_to_match="south korea")
```