CS167: Machine Learning

k-Nearest Neighbor (k-NN)
Handling Missing Data
Data Normalization

Thursday, February 15th, 2024

Drake University

Announcements

- Notebook #2: kNN and Normalization is released today
  - due Wednesday 02/21 by 11:59pm
  - to submit, download the `ipynb` file from Colab
  - directly upload to CodePost

- Heads up that Quiz #1
  - will be released on Tuesday 02/20 after class
  - will be due Tuesday 02/27 by 11:59pm

Before we get started, let's load in our datasets:

- Make sure you change the path to match your Google Drive.
  - Load the `titanic.csv` file from your Google Drive

```
[2] #run this cell if you're using Colab:
    from google.colab import drive
    drive.mount('/content/drive')
```

```
#import the data:
#make sure the path on the line below corresponds to the path where you put your dataset.
import pandas as pd
path = '/content/drive/MyDrive/cs167_fall23/datasets/titanic.csv'
titanic = pd.read_csv(path)
titanic.head()
path = '/content/drive/MyDrive/cs167_fall23/datasets/irisData.csv'
iris = pd.read_csv(path)
iris.head()
```

Today's Agenda

- Topics:
  - kNN Implementation using Pandas
  - Missing Data
  - Normalization

3-Nearest Neighbor (3-NN)

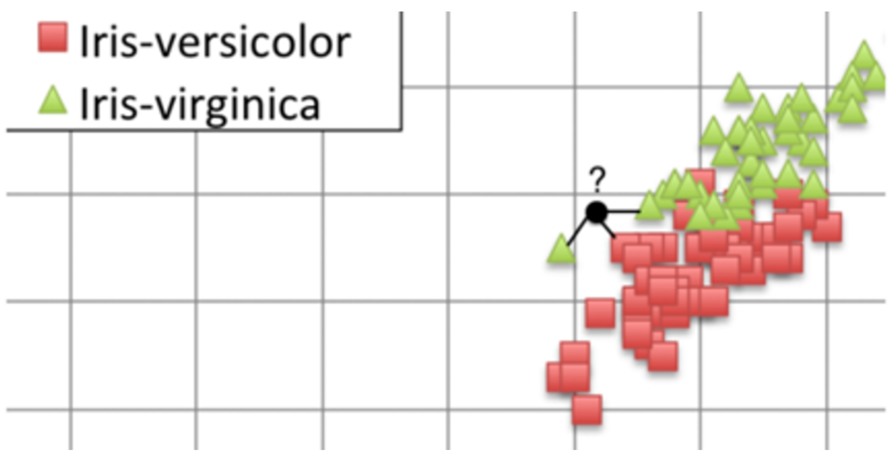
- **3-Nearest-Neighbor Algorithm:** predict the *most commonly appearing* class among the 3 closest training examples
  - In other words, k=3

- Let's assume this subset of Iris has only 2 classes (even number):

  `Iris-versicolor`

  `Iris-virginica`

- What class will a **3NN** algorithm predict?



k-Nearest Neighbor (kNN)

- **k-Nearest-Neighbor** predict the most commonly occurring class of the *k nearest neighbors*.

  1. Calculate the distance between the new point (e.g. the Iris we would like to make a prediction on), and the existing training examples.

  2. Sort the data by the newly calculated distance so that the nearest training examples are first

  3. Take the top `k` neighbors:
     - if the problem is a *classification*, **take the mode of the target variable** to find the most commonly appearing class and return that as your prediction

     - if the problem is a *regression*, **take the average of the target variables** for the k closest neighbors and return that as your prediction

k-NN Implementation in Python/Pandas

- Let's build a **5-Nearest-Neighbor** Iris classifier from scratch — using our Pandas/Python skills:

- To implement this 5NN, we need to do 3 things:

1. Calculate the distances from each of the rows to the new instance

3. Sort the data by these distances

5. Select the k closest training examples and use them to predict the most commonly occurring class of the closest neighbors.

Step 1: Calculate the Distances

- Let's start by adding a new column to our `iris` DataFrame that is the distance from each existing row to the new instance with:
  - 5.1 petal length, 7.2 sepal length, 1.5 petal width, and 2.5 sepal width
  - The syntax for adding a new column is as follows:
    - `df['new col name'] = ____________`

```
[9] iris['distance_to_new'] = np.sqrt(  (5.1 - iris['petal length'])**2 +
                                        (7.2 - iris['sepal length'])**2 +
                                        (1.5 - iris['petal width'])**2  +
                                        (2.5 - iris['sepal width'])**2 )
    iris.head()
```

Step 2: Sort the data by the Distances

- Let's now sort our data using the built in `sort_values()` function. [documentation]

- We want to find the nearest k neighbors, so sorting them in ascending order (which is the default setting for `sort_values()` will work nicely.

```
[10] k=15
     sorted_data = iris.sort_values(['distance_to_new'])
     sorted_data.head() #shortest distances first
```

	sepal length	sepal width	petal length	petal width	species	distance_to_new
76	6.8	2.8	4.8	1.4	Iris-versicolor	0.591608
52	6.9	3.1	4.9	1.5	Iris-versicolor	0.700000
77	6.7	3.0	5.0	1.7	Iris-versicolor	0.741620
50	7.0	3.2	4.7	1.4	Iris-versicolor	0.836660
129	7.2	3.0	5.8	1.6	Iris-virginica	0.866025

Step 3: Display the most common species among these 5

```
[21] print('Sorted distances closest 5')
     print(sorted_data.iloc[0:5][['distance_to_new', 'species']])

     Sorted distances closest 5
          distance_to_new          species
     76          0.591608  Iris-versicolor
     52          0.700000  Iris-versicolor
     77          0.741620  Iris-versicolor
     50          0.836660  Iris-versicolor
     129         0.866025   Iris-virginica

[22] sorted_data.iloc[0:5]['species'].mode()

     0    Iris-versicolor
     Name: species, dtype: object
```

- And Viola! We have successfully implemented our first machine learning model from scratch.

k-NN All Steps

```
[9]  iris['distance_to_new'] = np.sqrt(  (5.1 - iris['petal length'])**2 +
                                          (7.2 - iris['sepal length'])**2 +
                                          (1.5 - iris['petal width'])**2  +
                                          (2.5 - iris['sepal width'])**2 )

     iris.head()
```

```
[10] k=15
     sorted_data = iris.sort_values(['distance_to_new'])
     sorted_data.head() #shortest distances first
```

```
[21] print('Sorted distances closest 5')
     print(sorted_data.iloc[0:5][['distance_to_new', 'species']])
```

```
Sorted distances closest 5
     distance_to_new          species
76          0.591608  Iris-versicolor
52          0.700000  Iris-versicolor
77          0.741620  Iris-versicolor
50          0.836660  Iris-versicolor
129         0.866025   Iris-virginica
```

```
[22] sorted_data.iloc[0:5]['species'].mode()
```

```
0    Iris-versicolor
Name: species, dtype: object
```

Programming Exercise:

- Rewrite **k-NN** code so that it's a function.

- Pass the iris measurements (specimen), DataFrame, and k as parameters and return the predicted class.

```
def kNN(specimen, data, k):
    # write your code in here to make this function work
    # 1. calculate distances


    # 2. sort


    # 3. predict


    return prediction
```

Programming Exercise:

- Rewrite **k-NN** code so that it's a function.
- Pass the iris measurements (specimen), DataFrame, and k as parameters and return the predicted class.

```
new_iris = {}
new_iris['petal length'] = 5.1
new_iris['sepal length'] = 7.2
new_iris['petal width'] = 1.5
new_iris['sepal width'] = 2.5

# call the function you just wrote
kNN(new_iris, iris, 15)
```

Today's Agenda

- Topics:
  - kNN Implementation using Pandas
  - Missing Data
  - Normalization

Missing Data

- Most datasets you will work with will not be in perfect shape
  - you'll need to "clean" the data before you can run any machine learning algorithms on it.

- Missing data is a pretty common thing — so much so that there's a special value for missing data:
  - `NaN`, or not a number.

Missing Data

- The steps of cleaning data normally include:
  - Step 1: Detecting which columns have missing data
  - Step 2: Determining how much data is missing in each column
  - Step 3: Deciding what to do with the missing data:
    - drop it
    - fill it
    - let it be

Missing Data

- Notice, in the `deck` column, there are 3 instances of `NaN` we can see…
- But what about the other 800 or so rows? Do we have to go through and find them manually?

```
titanic.head()
```

	survived	pclass	sex	age	sibsp	parch	fare	embarked	class	who	adult_male	deck	embark_town
0	0	3	male	22.0	1	0	7.2500	S	Third	man	True	NaN	Southampton
1	1	1	female	38.0	1	0	71.2833	C	First	woman	False	C	Cherbourg
2	1	3	female	26.0	0	0	7.9250	S	Third	woman	False	NaN	Southampton
3	1	1	female	35.0	1	0	53.1000	S	First	woman	False	C	Southampton
4	0	3	male	35.0	0	0	8.0500	S	Third	man	True	NaN	Southampton

Step 1: Detecting Missing Data

- In order to identify missing data, we will use a combination of three Pandas functions:
    - isna()   https://pandas.pydata.org/docs/reference/api/pandas.isna.html
    - notna()   https://pandas.pydata.org/docs/reference/api/pandas.notna.html
    - any()   https://pandas.pydata.org/docs/reference/api/pandas.DataFrame.any.html

Step 1: Detecting Missing Data

- Using `isna()` and `notna()` to find missing data:
  - isna(): will return a boolean series where it is `True` if the element is `NaN`
  - notna(): will return a boolean series where it is `True` if the element is **`not NaN`**

Input/output

General functions

pandas.melt

pandas.pivot

pandas.pivot_table

pandas.crosstab

pandas.cut

pandas.qcut

pandas.merge

pandas.merge_ordered

API reference > General functions > **pandas.isna**

pandas.isna

pandas.isna(*obj*) [source]

Detect missing values for an array-like object.

This function takes a scalar or array-like object and indicates whether values are missing (`NaN` in numeric arrays, `None` or `NaN` in object arrays, `NaT` in datetimelike).

Parameters:

obj : *scalar or array-like*

Object to check for null or missing values.

https://pandas.pydata.org/docs/reference/api/pandas.isna.html

Step 1: Detecting Missing Data

```
titanic.loc[0:4]
```

	survived	pclass	sex	age	sibsp	parch	fare	embarked	class	who	adult_male	deck	embark_town
0	0	3	male	22.0	1	0	7.2500	S	Third	man	True	NaN	Southampton
1	1	1	female	38.0	1	0	71.2833	C	First	woman	False	C	Cherbourg
2	1	3	female	26.0	0	0	7.9250	S	Third	woman	False	NaN	Southampton
3	1	1	female	35.0	1	0	53.1000	S	First	woman	False	C	Southampton
4	0	3	male	35.0	0	0	8.0500	S	Third	man	True	NaN	Southampton

- Now, let's call `isna()`, and see what we get as an output

```
titanic.loc[:4].isna()
#look at the 'deck' column...
```

	survived	pclass	sex	age	sibsp	parch	fare	embarked	class	who	adult_male	deck	embark_town
0	False	False	False	False	False	False	False	False	False	False	False	True	False
1	False	False	False	False	False	False	False	False	False	False	False	False	False
2	False	False	False	False	False	False	False	False	False	False	False	True	False
3	False	False	False	False	False	False	False	False	False	False	False	False	False
4	False	False	False	False	False	False	False	False	False	False	False	True	False

Step 1: Detecting Missing Data

- `isna()` is pretty nifty but there should be better way to summarize this.
    - any()

pandas.DataFrame.any

```
DataFrame.any(*, axis=0, bool_only=False, skipna=True, **kwargs)
```
[source]

Return whether any element is True, potentially over an axis.

Returns False unless there is at least one element within a series or along a Dataframe axis

https://pandas.pydata.org/docs/reference/api/pandas.DataFrame.any.html

Step 1: Identifying Missing Data

- Let's use `any()` on the call to `isna()` we just did to let us know which columns have missing data:

```
titanic.isna().any()

survived       False
pclass         False
sex            False
age             True
sibsp          False
parch          False
fare           False
embarked        True
class          False
who            False
adult_male     False
deck            True
embark_town     True
alive          False
alone          False
dtype: bool
```

- Several columns are missing data: `age`, `embarked`, `deck`, and `embark_town`.
- Wouldn't it be great to know how much data is missing in each of those columns?

Missing Data

- The steps of cleaning data normally include:
  - Step 1: Identifying which columns have missing data
  - Step 2: Determining how much data is missing in each column
  - Step 3: Deciding what to do with the missing data:
    - drop it
    - fill it
    - let it be

Step 2: How much data is missing?

- To decide how to handle our missing data, it's important to know how much missing data each column has:
  - If the missing data is a small proportion of the data, we choose to drop those rows completely from the dataset
  - However, if most of the rows are missing data for a specific column, maybe it's a sign that we don't need to use that column
- There are multiple ways of doing this, but one of the quickest/easiest is using **`value_counts()`**

Step 2: How much data is missing?

- Great, so now that we know which columns are missing data, let's check to see how much data they are missing using **`value_counts()`**

pandas.Series.value_counts

`Series.value_counts(normalize=False, sort=True, ascending=False, bins=None, dropna=True)` [source]

Return a Series containing counts of unique values.

The resulting object will be in descending order so that the first element is the most frequently-occurring element. Excludes NA values by default.

https://pandas.pydata.org/docs/reference/api/pandas.Series.value_counts.html

Step 2: How much data is missing?

- Let's apply **`value_counts()`** on the various columns (eg, `deck`) of Titanic dataset

```
titanic.deck.value_counts(dropna=False)
#688 missing values
```

```
NaN    688
C       59
B       47
D       33
E       32
A       15
F       13
G        4
Name: deck, dtype: int64
```

https://pandas.pydata.org/docs/reference/api/pandas.Series.value_counts.html

Step 2: How much data is missing?

- Let's apply **`value_counts()`** on the various columns (eg, `age`) of Titanic dataset

```
titanic.age.value_counts(dropna=False)
#177 missing values
```

```
NaN      177
24.00     30
22.00     27
18.00     26
28.00     25
        ...
36.50      1
55.50      1
0.92       1
23.50      1
74.00      1
Name: age, Length: 89, dtype: int64
```

https://pandas.pydata.org/docs/reference/api/pandas.Series.value_counts.html

Step 2: How much data is missing?

- Let's apply **`value_counts()`** on the various columns (eg, `embarked`) of Titanic dataset

```
titanic.embarked.value_counts(dropna=False)
#2 missing values
```

```
S      644
C      168
Q       77
NaN      2
Name: embarked, dtype: int64
```

https://pandas.pydata.org/docs/reference/api/pandas.Series.value_counts.html

Step 2: How much data is missing?

- Let's apply **`value_counts()`** on the various columns (eg, `embark_town`) of Titanic dataset

```
titanic.embark_town.value_counts(dropna=False)
#2 missing values

Southampton    644
Cherbourg      168
Queenstown      77
NaN              2
Name: embark_town, dtype: int64
```

https://pandas.pydata.org/docs/reference/api/pandas.Series.value_counts.html

Step 2: How much data is missing?

- So, here is our results using `value_counts()`

Column	Num Rows Missing
deck	688
age	177
embarked	2
embark_town	2

- Now with this new information, it's up to us to decide what to do with these missing values

https://pandas.pydata.org/docs/reference/api/pandas.Series.value_counts.html

Missing Data

- The steps of cleaning data normally include:
  - Step 1: Identifying which columns have missing data
  - Step 2: Determining how much data is missing in each column
  - Step 3: Deciding what to do with the missing data:
    - **drop it:** drop the missing data from the dataset (either col or row)
    - **fill it:** fill the missing data with a suitable replacement
    - **let it be:** let it be and cross our fingers

Option 1: Drop it using `dropna()`

- If there isn't much missing data, and/or you have a very large dataset, dropping the row that includes the missing data is a viable option.

```
print("before: ", titanic.shape)
titanic.dropna()
print("after: " , titanic.shape)
```

```
before:  (891, 15)
after:  (891, 15)
```

- We know that there's missing data, why didn't the shape change?

Option 1: Drop it using `dropna()`

- Pandas is trying to protect you, and rather than dropping the rows "in place", it is returning a `DataFrame` with the rows dropped--as written, we're just not saving it's return. There are two ways to fix this:
  - save what `dropna()` is returning in a variable

```
print("before dropna(): ", titanic.shape)
no_missing_data = titanic.dropna()
print("after dropna(): ", no_missing_data.shape)

before dropna():  (891, 15)
after dropna():  (182, 15)
```

  - add the parameter `inplace=True` to the function call, and it will drop the rows in the original dataset (be careful with this one)

```
print("before dropna(): ", titanic.shape)
titanic.dropna(inplace=True)
print("after dropna(inplace=True): ", titanic.shape)

before dropna():  (891, 15)
after dropna(inplace=True):  (182, 15)
```

Option 1: Drop it using `dropna()`

- That's better, but wow, most of our dataset is gone now if we drop all of the rows that have missing data. If this happens to you, you'll probably want to re-load your data to have the full dataset to work with.

```
# if that happens, you'll want to re-run your data loading code:
path = '/content/drive/MyDrive/cs167_fall23/datasets/titanic.csv'
titanic = pd.read_csv(path)
```

Option 2: Fill it using `fillna()`

- If dropping all of the data will make your dataset too sparse, consider filling the missing values with something else.

- What do you think we should use to fill in the missing data in the `age` column?
  - we probably don't want to throw off our statistics...

```
titanic.head(7)
```

	survived	pclass	sex	age	sibsp	parch	fare	embarked	class	who	adult_male	deck	embark_town	alive	alone
0	0	3	male	22.0	1	0	7.2500	S	Third	man	True	NaN	Southampton	no	False
1	1	1	female	38.0	1	0	71.2833	C	First	woman	False	C	Cherbourg	yes	False
2	1	3	female	26.0	0	0	7.9250	S	Third	woman	False	NaN	Southampton	yes	True
3	1	1	female	35.0	1	0	53.1000	S	First	woman	False	C	Southampton	yes	False
4	0	3	male	35.0	0	0	8.0500	S	Third	man	True	NaN	Southampton	no	True
5	0	3	male	NaN	0	0	8.4583	Q	Third	man	True	NaN	Queenstown	no	True
6	0	1	male	54.0	0	0	51.8625	S	First	man	True	E	Southampton	no	True

Option 2: Fill it using `fillna()`

- What do you think we should use to fill in the missing data in the `age` column?
  - we probably don't want to throw off our statistics...

```
print("before: ", titanic['age'].isna().any())
age_mean = titanic['age'].mean()
titanic['age'].fillna(age_mean, inplace=True)
print("after: ", titanic['age'].isna().any())
titanic.head(7)
```

	survived	pclass	sex	age	sibsp	parch	fare	embarked	class	who	adult_male	deck	embark_town	alive	alone
0	0	3	male	22.000000	1	0	7.2500	S	Third	man	True	NaN	Southampton	no	False
1	1	1	female	38.000000	1	0	71.2833	C	First	woman	False	C	Cherbourg	yes	False
2	1	3	female	26.000000	0	0	7.9250	S	Third	woman	False	NaN	Southampton	yes	True
3	1	1	female	35.000000	1	0	53.1000	S	First	woman	False	C	Southampton	yes	False
4	0	3	male	35.000000	0	0	8.0500	S	Third	man	True	NaN	Southampton	no	True
5	0	3	male	29.699118	0	0	8.4583	Q	Third	man	True	NaN	Queenstown	no	True
6	0	1	male	54.000000	0	0	51.8625	S	First	man	True	E	Southampton	no	True

Option 3: Let it be

- What's so bad about missing data? Why do we care if some data is missing?
- What happens if we try to do math with `NaN`? Try it out for yourself:
  - Go to the bottom of the Day05_Missing_Data_Normalization.ipynb and try out

Summary: Missing Data

- The steps of cleaning data normally include:
  - Step 1: Detecting which columns have missing data
  - Step 2: Determining how much data is missing in each column
  - Step 3: Deciding what to do with the missing data:
    - drop it
    - fill it
    - let it be

Summary: Missing Data Functions

- `isna()`: returns True for any missing data
- `notna()`: returns True for any data that is **not** `NaN`
- `any()`: returns true if any of the elements in a Series is True
- `value_counts()`: returns a list of the values in a Series, use `dropna=False` to see `NaN` values
- `dropna()`: drops rows or columns (specify which axis, 1 or 0) that have missing data. Don't forget to either save the result of the call or add `inplace=True` as a parameter
- `fillna()`: replaces missing data with a given value (generally 0 or the mean)

Today’s Agenda

- Topics:
  - kNN Implementation using Pandas
  - Missing Data
  - Normalization

Normalization

- Normalizing data:
  - rescale attribute values so they're about the same
  - adjusting values measured on different scales to a common scale

A Simple Normalization:

- One simple method of normalizing data is to replace each value with a proportion relative to the max value.

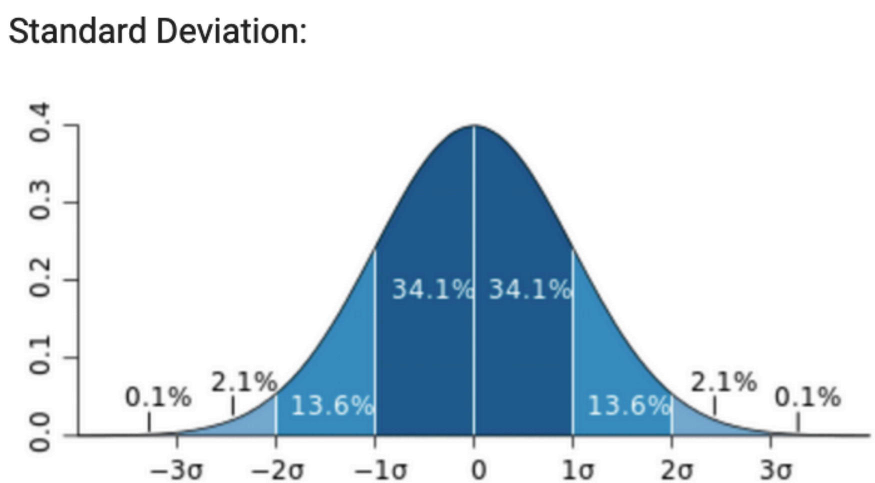
- For example, the oldest person on the Titanic was 80, so:

age	replaced by
80	80/80 = 1
50	50/80 = 0.625
48	48/80 = 0.6
25	25/80 = 0.3125
4	4/80 = 0.05

Z-Score: Another Normalization Method

- **Idea:** rather than normalize to proportion of max, normalize based on how many standard deviations they are away from the mean

- **Standard Deviation:** usually represented as σ(sigma), a kind of 'average' distance from the average value
  - a low standard deviation indicates that the values tend to be close to the mean
  - a high standard deviation indicates that the values are spread out over a wider range



Standard Deviation Calculation

- **Standard Deviation**: usually represented as σ(sigma), a kind of 'average' distance from the average value
  - Find the mean, represented as μ: mu
  - Then, for each number, subtract the mean and square the result
  - Then, find the mean of those squared differences
  - Take the square root of that and we are done

- Let μ be the mean, then standard deviation of x_1, x_2,..., x_N is:

$$\sigma = \sqrt{\frac{(x_1-\mu)^2+(x_2-\mu)^2+\ldots+(x_N-\mu)^2}{N}}$$

Corrected Sample Standard Deviation

- **Bessel's correction** says that you should divide by $N-1$ instead of N when working with a sample (as we usually do in machine learning tasks), and your estimate will be a little less biased.

$$\sigma = \sqrt{\frac{(x_1-\mu)^2+(x_2-\mu)^2+\ldots+(x_N-\mu)^2}{N-1}}$$

Computing the Z-Score

- After computing the corrected sample standard deviation, to normalize, replace each value x_i with it's **Z-Score** based on the mean (μ) and standard deviation (σ) of it's column.

$$\boxed{Z - score : \frac{x_i - \mu}{\sigma}}$$

Computing the Z-Score

- For example: On the Titanic:
  - sex mean(0:male, 1:female): 0.35
  - sex standard deviation: 0.48
  - age mean: 29.7
  - age standard deviation: 13

$$Z - score : \frac{x_i - \mu}{\sigma}$$

	sex	age
example 1	1	50
example 2	0	48

	sex	age
example 1	1	50
example 3	1	25

Z-Score for male: $(0 - 0.35)/0.48 \approx -0.73$
Z-Score for female: $(1 - 0.35)/0.48 \approx 1.35$

Z-Score for age 50: $(50 - 29.7)/13 \approx 1.56$
Z-Score for age 48: $(48 - 29.7)/13 \approx 1.41$
Z-Score for age 25: $(25 - 29.7)/13 \approx -0.36$

Distance Computation Before Normalization

	sex	age
example 1	1	50
example 2	0	48

distance: $\sqrt{(1-0)^2 + (50-48)^2} \approx 2.24$

	sex	age
example 1	1	50
example 3	1	25

distance: $\sqrt{(1-1)^2 + (50-25)^2} = 25$

Distance Computation After Normalization

	sex	age
example 1	1.35	1.56
example 2	-0.73	1.41

distance:

$$\sqrt{(1.35 - -0.73)^2 + (1.56 - 1.41)^2}$$
$$\approx 2.09$$

	sex	age
example 1	1.35	1.56
example 3	1.35	-0.36

distance:

$$\sqrt{(1.35 - 1.35)^2 + (1.56 - -0.36)^2}$$
$$= 1.92$$

Computing the Z-Score on Titanic

- Called on a dataframe, will replace values given in `to_replace` with `value`. Let's use this to make the `sex` column of the dataset numeric.

```
titanic['sex'] = titanic['sex'].replace(to_replace='female', value=1)
titanic['sex'] = titanic['sex'].replace(to_replace='male', value=0)
titanic.head()
```

	survived	pclass	sex	age	sibsp	parch	fare	embarked	class	who	adult_male	deck	embark_town	alive	alone
0	0	3	0	22.0	1	0	7.2500	S	Third	man	True	NaN	Southampton	no	False
1	1	1	1	38.0	1	0	71.2833	C	First	woman	False	C	Cherbourg	yes	False
2	1	3	1	26.0	0	0	7.9250	S	Third	woman	False	NaN	Southampton	yes	True
3	1	1	1	35.0	1	0	53.1000	S	First	woman	False	C	Southampton	yes	False
4	0	3	0	35.0	0	0	8.0500	S	Third	man	True	NaN	Southampton	no	True

Computing the Z-Score on Titanic

- Now that we have the data as 1s and 0s, let's calculate the mean and standard deviation

```
s_mean = titanic.sex.mean()
s_std = titanic.sex.std()

#replace column with each entry's z-score
titanic.sex = (titanic.sex - s_mean)/s_std
titanic.head()
```

$$Z - score : \frac{x_i - \mu}{\sigma}$$

	survived	pclass	sex	age	sibsp	parch	fare	embarked	class	who	adult_male	deck	embark_town	alive	alone
0	0	3	-0.734928	22.0	1	0	7.2500	S	Third	man	True	NaN	Southampton	no	False
1	1	1	1.359146	38.0	1	0	71.2833	C	First	woman	False	C	Cherbourg	yes	False
2	1	3	1.359146	26.0	0	0	7.9250	S	Third	woman	False	NaN	Southampton	yes	True
3	1	1	1.359146	35.0	1	0	53.1000	S	First	woman	False	C	Southampton	yes	False
4	0	3	-0.734928	35.0	0	0	8.0500	S	Third	man	True	NaN	Southampton	no	True