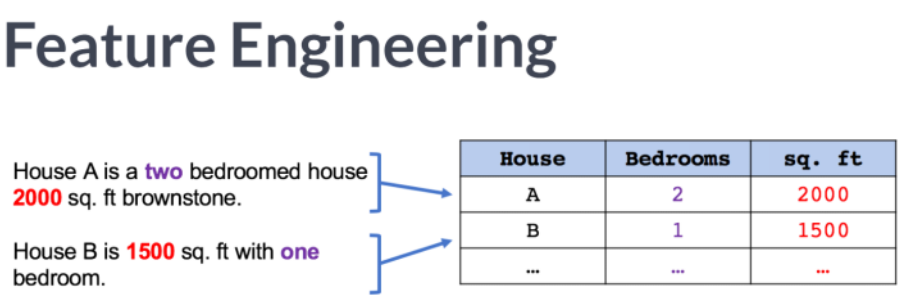
Feature Engineering for Machine Learning in Python

Wednesday, 2 September 2020 9:14 PM

Creating Features



Different types of data

- Continuous: either integers (or whole numbers) or floats (decimals)
- Categorical: one of a limited set of values, e.g. gender, country of birth
- Ordinal: ranked values, often with no detail of distance between them
- Boolean: True/False values
- Datetime: dates and times

Course structure

- Chapter 1: Feature creation and extraction
- Chapter 2: Engineering messy data
- Chapter 3: Feature normalization
- Chapter 4: Working with text features

Pandas

```
import pandas as pd
df = pd.read_csv(path_to_csv_file)
print(df.head())
```

Dataset

```
              SurveyDate  \
0    2018-02-28 20:20:00
1    2018-06-28 13:26:00
2    2018-06-06 03:37:00
3    2018-05-09 01:06:00
4    2018-04-12 22:41:00

                            FormalEducation
0    Bachelor's degree (BA. BS. B.Eng.. etc.)
1    Bachelor's degree (BA. BS. B.Eng.. etc.)
2    Bachelor's degree (BA. BS. B.Eng.. etc.)
3    Some college/university study  ...
4    Bachelor's degree (BA. BS. B.Eng.. etc.)
```

Column names

```
print(df.columns)
```

```
Index(['SurveyDate', 'FormalEducation',
       'ConvertedSalary', 'Hobby', 'Country',
       'StackOverflowJobsRecommend', 'VersionControl',
       'Age', 'Years Experience', 'Gender',
       'RawSalary'], dtype='object')
```

Column types

```
print(df.dtypes)
```

```
SurveyDate                          object
FormalEducation                     object
ConvertedSalary                    float64
...
Years Experience                     int64
Gender                              object
RawSalary                           object
dtype: object
```

Selecting specific data types

```
only_ints = df.select_dtypes(include=['int'])
print(only_ints.columns)
```

```
Index(['Age', 'Years Experience'], dtype='object')
```

Encoding categorical features

Index	Country
1	'India'
2	'USA'
3	'UK'
4	'UK'
5	'France'
...	...

Encoding categorical features

Index	Country
1	'India'
2	'USA'
3	'UK'
4	'UK'
5	'France'
…	…

Index	C_India	C_USA	C_UK	C_France
1	1	0	0	0
2	0	1	0	0
3	0	0	1	0
4	0	0	1	0
5	0	0	0	1
…	…	…	…	…

Encoding categorical features

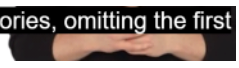
- One-hot encoding
- Dummy encoding

One-hot encoding

```
pd.get_dummies(df, columns=['Country'],
               prefix='C')
```

	C_France	C_India	C_UK	C_USA
0	0	1	0	0
1	0	0	0	1
2	0	0	1	0
3	0	0	1	0
4	1	0	0	0

On the other hand, dummy encoding creates n-1 features for n categories, omitting the first category.



Dummy encoding

```
pd.get_dummies(df, columns=['Country'],
               drop_first=True, prefix='C')
```

	C_India	C_UK	C_USA
0	1	0	0
1	0	0	1
2	0	1	0
3	0	1	0
4	0	0	0

In dummy encoding, the base value, France in this case, is encoded by the absence of all

One-hot vs. dummies

- **One-hot encoding**: Explainable features
- **Dummy encoding**: Necessary information without duplication

Both these methods have different advantages.

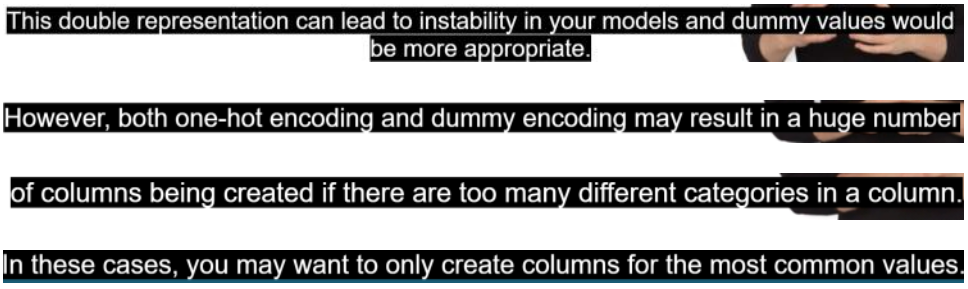
One-hot encoding generally creates much more explainable features, as each country will have its own weight that can be observed after training.

But one must be aware that one hot encoding may create features that are entirely collinear due to the same information being represented multiple times.

Index	Sex
0	Male
1	Female
2	Male

Index	Male	Female
0	1	0
1	0	1
2	1	0

Index	Male
0	1
1	0
2	1



Limiting your columns

```
counts = df['Country'].value_counts()
print(counts)
```

```
'USA'      8
'UK'       6
'India'    2
'France'   1
Name: Country, dtype: object
```

Limiting your columns

```
mask = df['Country'].isin(counts[counts < 5].index)
df['Country'][mask] = 'Other'
```

Types of numeric features

- Age
- Price
- Counts
- Geospatial data

Does size matter?

	Resturant_ID	Number_of_Violations
0	RS_1	0
1	RS_2	0
2	RS_3	2
3	RS_4	1
4	RS_5	0
5	RS_6	0
6	RS_7	4
7	RS_8	4
8	RS_9	1
9	RS_10	0

Binarizing numeric variables

```
df['Binary_Violation'] = 0
df.loc[df['Number_of_Violations'] > 0,
       'Binary_Violation'] = 1
```

Binarizing numeric variables

	Resturant_ID	Number_of_Violations	Binary_Violation
0	RS_1	0	0
1	RS_2	0	0
2	RS_3	2	1
3	RS_4	1	1
4	RS_5	0	0
5	RS_6	0	0
6	RS_7	4	1
7	RS_8	4	1
8	RS_9	1	1
9	RS_10	0	0

Consider the same dataset of restaurant health and safety ratings

containing the number of times a restaurant has had major violations.

This time we will be creating three groups, Group 1, for restaurants with no offenses, Group 2

for restaurants with one or two offenses and group 3 for all restaurants with three or more offenses.

Bins are created by using the pandas' cut() function.

You can define the intervals using the bins argument as shown here, which in this case is a list of 4 values.

You can also pass a list of labels like so.

Binning numeric variables

```
import numpy as np
df['Binned_Group'] = pd.cut(
    df['Number_of_Violations'],
    bins=[-np.inf, 0, 2, np.inf],
    labels=[1, 2, 3]
)
```

Binning numeric variables

	Resturant_ID	Number_of_Violations	Binned_Group
0	RS_1	0	1
1	RS_2	0	1
2	RS_3	2	2
3	RS_4	1	2
4	RS_5	0	1
5	RS_6	0	1
6	RS_7	4	3
7	RS_8	4	3
8	RS_9	1	2
9	RS_10	0	1

Note as we want to include 0 in the first bin, we must set the leftmost edge to lower than that, so all values between negative infinity and 0 are labeled as 1, all values equal to 1 or 2 are labeled as 2, and values greater than 2 are labeled as 3.

Dealing with Messy Data

How gaps in data occur

- Data not being collected properly
- Collection and management errors
- Data intentionally being omitted
- Could be created due to transformations of the data

Why we care?

- Some models cannot work with missing data (Nulls/NaNs)
- Missing data may be a sign of a wider data issue
- Missing data can be a useful feature

Missing value discovery

```
print(df.info())
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 999 entries, 0 to 998
Data columns (total 12 columns):
SurveyDate                    999 non-null object
...
StackOverflowJobsRecommend    487 non-null float64
VersionControl                999 non-null object
Gender                        693 non-null object
RawSalary                     665 non-null object
dtypes: float64(2), int64(2), object(8)
memory usage: 93.7+ KB
```

Finding missing values

```
print(df.isnull())
```

```
   StackOverflowJobsRecommend  VersionControl  ... \
0                        True           False  ...
1                       False           False  ...
2                       False           False  ...
3                        True           False  ...
4                       False           False  ...

   Gender  RawSalary
0   False       True
1   False      False
2    True       True
3   False      False
```

Finding missing values

```
print(df['StackOverflowJobsRecommend'].isnull().sum())
```

```
512
```

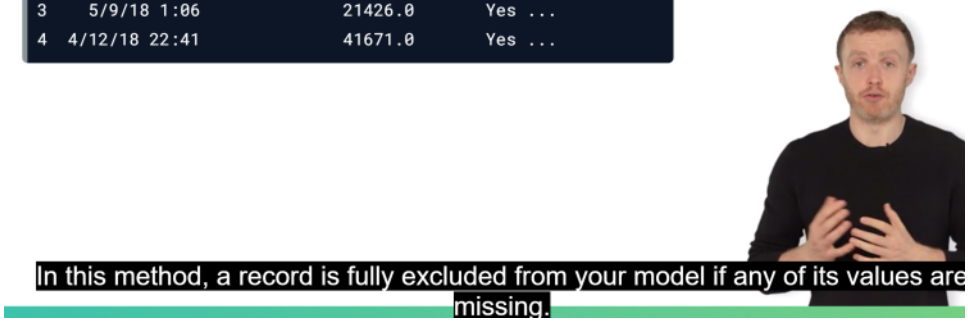
Finding non-missing values

```
print(df.notnull())
```

```
   StackOverflowJobsRecommend  VersionControl  ... \
0                       False            True  ...
1                        True            True  ...
2                        True            True  ...
3                       False            True  ...
4                        True            True  ...

   Gender  RawSalary
0    True      False
1    True       True
2   False      False
3    True       True
```

Listwise deletion

```
      SurveyDate      ConvertedSalary     Hobby ... \
0  2/28/18 20:20                 NaN       Yes ...
1  6/28/18 13:26             70841.0       Yes ...
2    6/6/18 3:37                 NaN        No ...
3    5/9/18 1:06             21426.0       Yes ...
4  4/12/18 22:41             41671.0       Yes ...
```



Listwise deletion in Python

```
# Drop all rows with at least one missing values
df.dropna(how='any')
```

Listwise deletion in Python

```
# Drop rows with missing values in a specific column
df.dropna(subset=['VersionControl'])
```

Issues with deletion

- It deletes valid data points
- Relies on randomness
- Reduces information

Replacing with strings

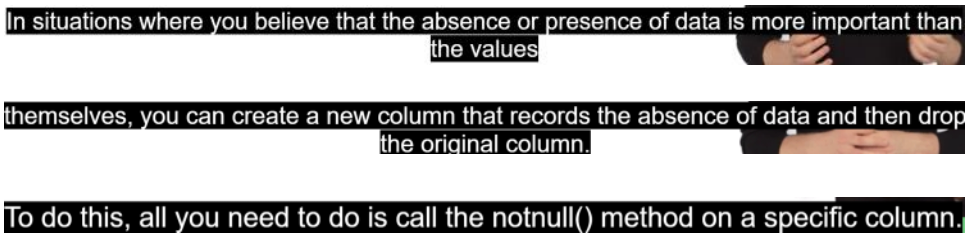
```
# Replace missing values in a specific column
# with a given string
df['VersionControl'].fillna(
    value='None Given', inplace=True
)
```

To replace the missing values in place, in other words to modify the original DataFrame, you need to set the inplace argument to True.

Recording missing values

```
# Record where the values are not missing
df['SalaryGiven'] = df['ConvertedSalary'].notnull()
```

```
# Drop a specific column
df.drop(columns=['ConvertedSalary'])
```



Deleting missing values

- Can't delete rows with missing values in the test set

What else can you do?

- **Categorical columns**: Replace missing values with the most common occurring value or with a string that flags missing values such as 'None'
- **Numeric columns**: Replace missing values with a suitable value

Measures of central tendency

- Mean
- Median

Calculating the measures of central tendency

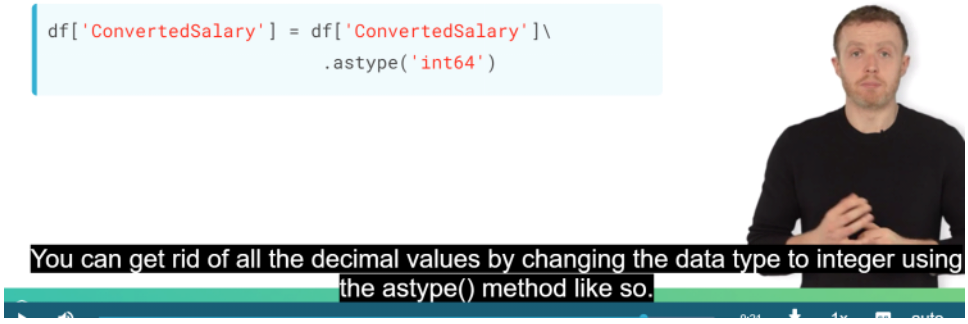
```
print(df['ConvertedSalary'].mean())
print(df['ConvertedSalary'].median())
```

```
92565.16992481203
55562.0
```

Fill the missing values

```
df['ConvertedSalary'] = df['ConvertedSalary'].fillna(
    df['ConvertedSalary'].mean()
)
```

```
df['ConvertedSalary'] = df['ConvertedSalary']\
                        .astype('int64')
```



Rounding values

```
df['ConvertedSalary'] = df['ConvertedSalary'].fillna(
    round(df['ConvertedSalary'].mean())
)
```

Bad characters

```
print(df['RawSalary'].dtype)
```

```
dtype('O')
```

Bad characters

```
print(df['RawSalary'].head())
```

```
0          NaN
1    70,841.00
2          NaN
3    21,426.00
4    41,671.00
Name: RawSalary, dtype: object
```

Numeric columns should not contain any non-numeric characters.

Dealing with bad characters

```
df['RawSalary'] = df['RawSalary'].str.replace(',', '')
```

However, the data type of this column is still object.

```
df['RawSalary'] = df['RawSalary'].astype('float')
```

But what if attempting to change the data type raises an error?

This may indicate that there are additional stray characters which you didn't account for.

Instead of manually searching for values with other stray characters you

can use the to_numeric() function from pandas along with the errors argument.

If you set the errors argument to 'coerce', Pandas will convert the column to numeric, but

all values that can't be converted to numeric will be changed to NaNs, that is missing values.

Finding other stray characters

```
print(df[coerced_vals.isna()].head())
```

```
0            NaN
2            NaN
4      $51408.00
Name: RawSalary, dtype: object
```

Chaining methods

```
df['column_name'] = df['column_name'].method1()
df['column_name'] = df['column_name'].method2()
df['column_name'] = df['column_name'].method3()
```

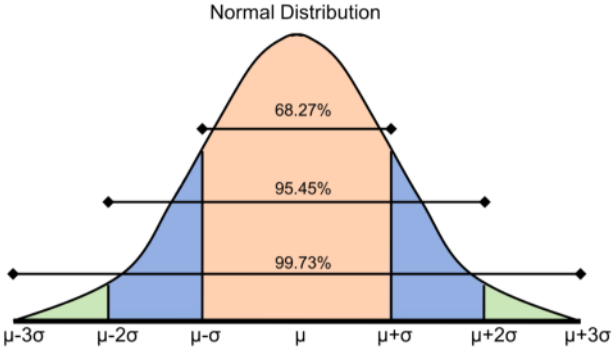
Same as:

```
df['column_name'] = df['column_name']\
                    .method1().method2().method3()
```

For example, cleaning up characters, changing the data type, normalizing the values etc.

Conforming to Statistical Assumptions

Distribution assumptions



Almost every model besides tree based models assume that your data is normally distributed.

Normal distributions follow a bell shape like shown here, the main characteristics

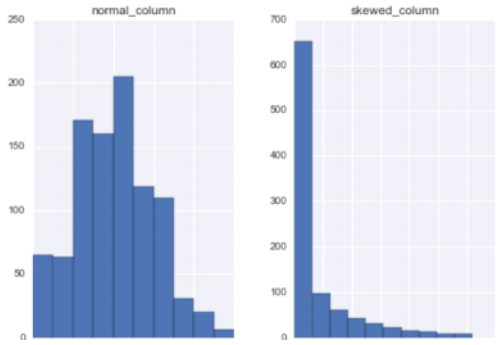
of a normal distribution is that 68 percent of the data lies within 1 standard

deviation of the mean,95% percent lies within 2 standard deviations from the mean and 99

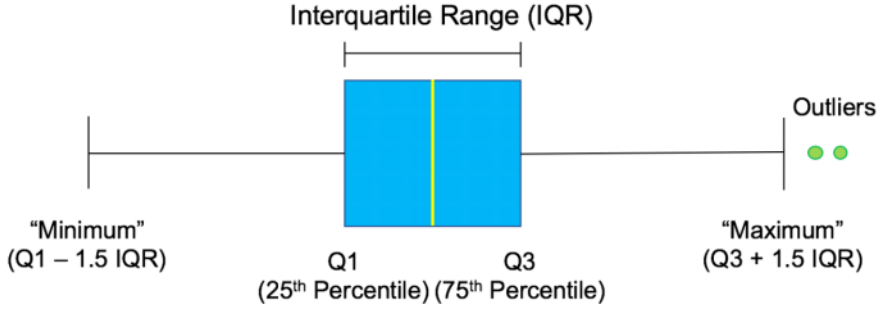
7% fall within 3 standard deviations from the mean.

Observing your data

```
import matplotlib as plt

df.hist()
plt.show()
```

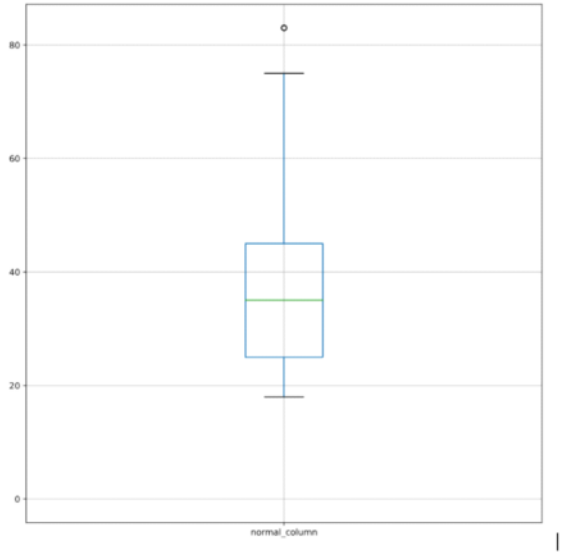


Delving deeper with box plots



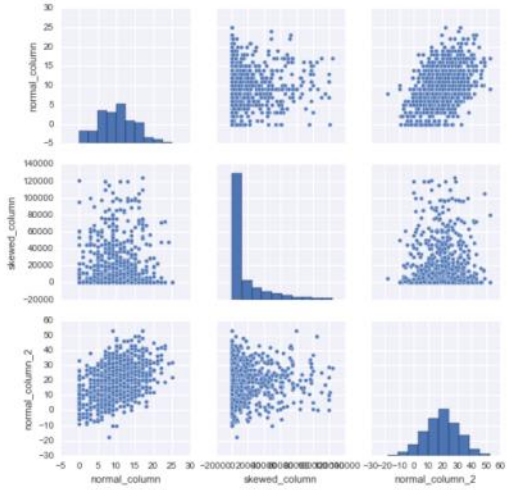
Box plots in pandas

```
df[['column_1']].boxplot()
plt.show()
```



Paring distributions

```
import seaborn as sns
sns.pairplot(df)
```

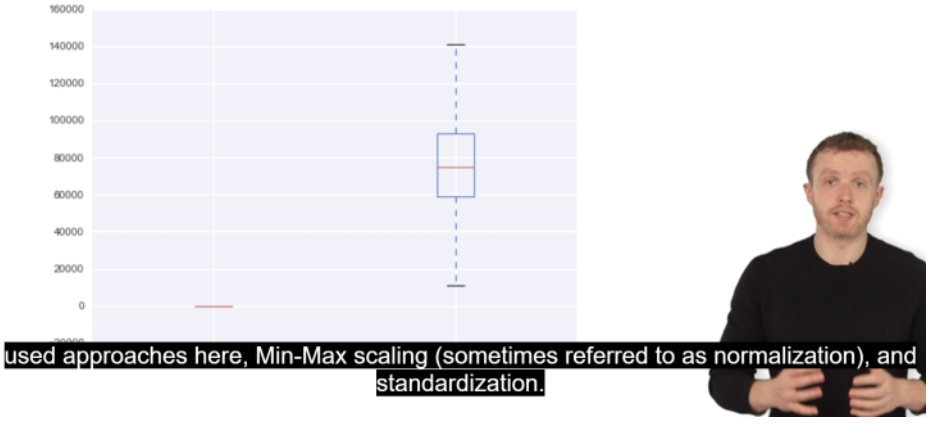


Further details on your distributions

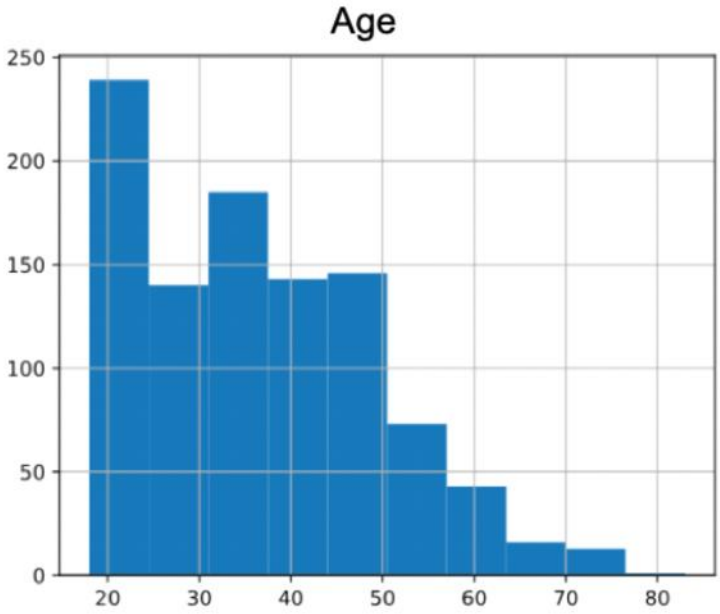
```
df.describe()
```

	Col1	Col2	Col3	Col4
count	100.000000	100.000000	100.000000	100.000000
mean	-0.163779	-0.014801	-0.087965	-0.045790
std	1.046370	0.920881	0.936678	0.916474
min	-2.781872	-2.156124	-2.647595	-1.957858
25%	-0.849232	-0.655239	-0.602699	-0.736089
50%	-0.179495	0.032115	-0.051863	0.066803
75%	0.663515	0.615688	0.417917	0.689591
max	2.466219	2.353921	2.059511	1.838561

Scaling data



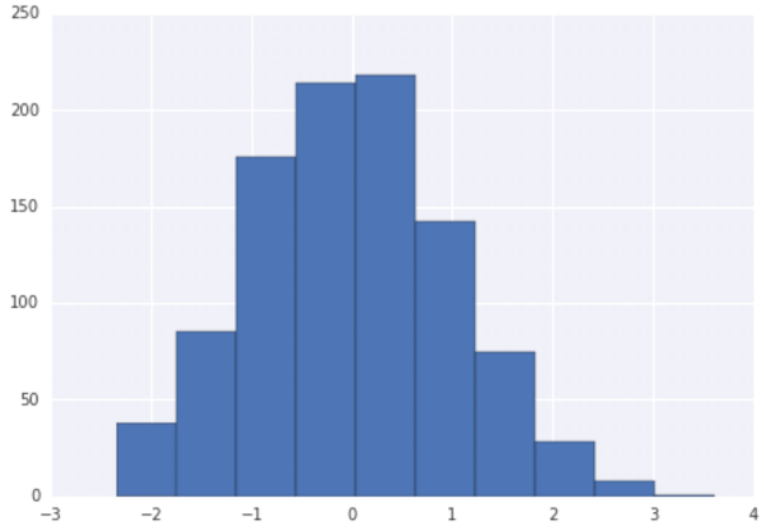
Min-Max scaling



Min-Max scaling in Python

```
from sklearn.preprocessing import MinMaxScaler

scaler = MinMaxScaler()

scaler.fit(df[['Age']])

df['normalized_age'] = scaler.transform(df[['Age']])
```

Standardization



As opposed to finding an outer boundary and squeezing everything within it,

standardization instead finds the mean of your data and centers your distribution

around it, calculating the number of standard deviations away from the mean each point is.

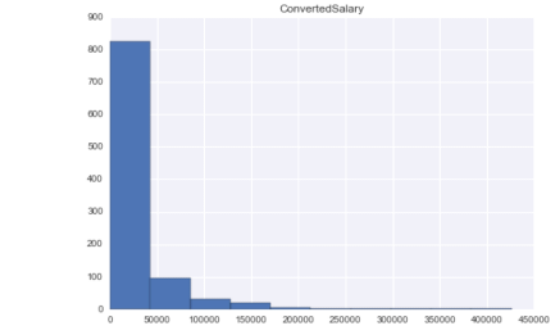
These values (the number of standard deviations) are then used as your new values.

This centers the data around 0 but technically has no limit to the maximum and minimum values as you can see here.

Standardization in Python

```
from sklearn.preprocessing import StandardScaler

scaler = StandardScaler()

scaler.fit(df[['Age']])

df['standardized_col'] = scaler\
                         .transform(df[['Age']])
```

Log Transformation

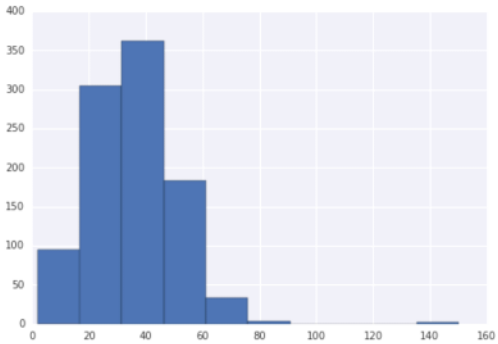


A log transformation on the other hand can be used to make highly skewed distributions less skewed.

Log transformation in Python

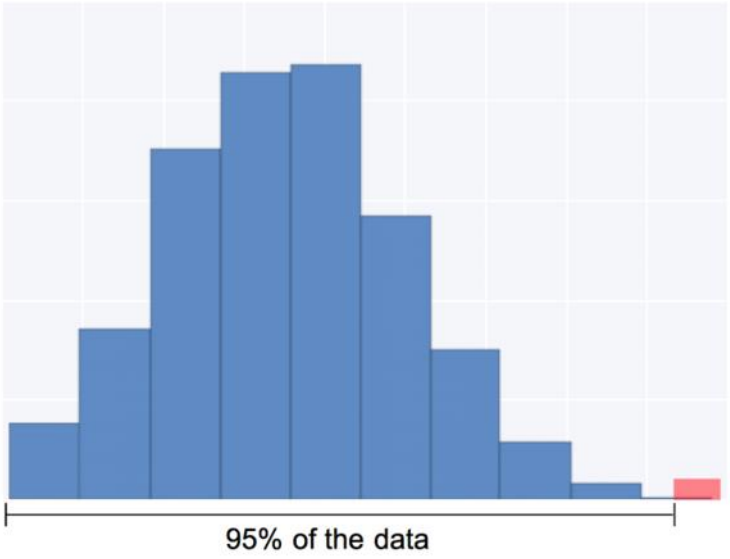
```
from sklearn.preprocessing import PowerTransformer

log = PowerTransformer()

log.fit(df[['ConvertedSalary']])

df['log_ConvertedSalary'] =
     log.transform(df[['ConvertedSalary']])
```

What are outliers?



Outliers are data points that exist far away from the majority of your data.

Quantile based detection



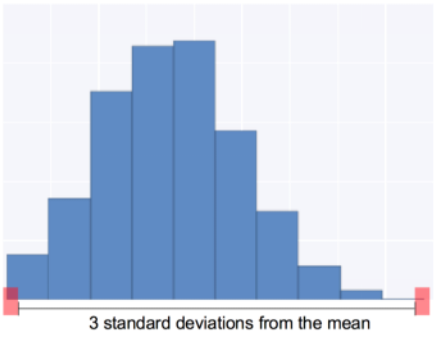
The first approach we will discuss is to remove a certain percentage of the largest and/or smallest values in your data.

For example you could remove the top 5%.

Quantiles in Python

```
q_cutoff = df['col_name'].quantile(0.95)

mask = df['col_name'] < q_cutoff

trimmed_df = df[mask]
```

Standard deviation based detection



An alternative, and perhaps more statistically sound method of removing outliers is to instead

choose what you consider to be outliers based on the mean and standard deviations of the data set.

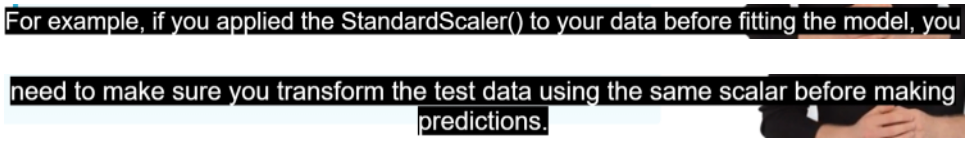
deviations from the mean as you expect those data points to be outliers.

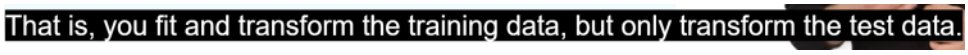
Standard deviation detection in Python

```
mean = df['col_name'].mean()
std = df['col_name'].std()

cut_off = std * 3

lower, upper = mean - cut_off, mean + cut_off

new_df = df[(df['col_name'] < upper) &
                 (df['col_name'] > lower)]
```

Reuse training scalers

```
scaler = StandardScaler()

scaler.fit(train[['col']])

train['scaled_col'] = scaler.transform(train[['col']])

# FIT SOME MODEL
# ....

test = pd.read_csv('test_csv')

test['scaled_col'] = scaler.transform(test[['col']])
```





Training transformations for reuse

```
train_mean = train[['col']].mean()
train_std = train[['col']].std()

cut_off = train_std * 3
train_lower = train_mean - cut_off
train_upper = train_mean + cut_off

# Subset train data

test = pd.read_csv('test_csv')

# Subset test data
test = test[(test[['col']] < train_upper) &
            (test[['col']] > train_lower)]
```

Why only use training data?

Data leakage: Using data that you won't have access to when assessing the performance of your model

Dealing with Text Data

Standardizing your text

Example of free text:

> Fellow-Citizens of the Senate and of the House of Representatives: AMONG the vicissitudes incident to life no event could have filled me with greater anxieties than that of which the notification was transmitted by your order, and received on the th day of the present month.

Dataset

```
print(speech_df.head())
```

```
                 Name           Inaugural Address    \
0    George Washington     First Inaugural Address
1    George Washington    Second Inaugural Address
2    John Adams                 Inaugural Address
3    Thomas Jefferson      First Inaugural Address
4    Thomas Jefferson     Second Inaugural Address

                      Date                                text
0    Thursday, April 30, 1789    Fellow-Citizens of the Sena...
1       Monday, March 4, 1793    Fellow Citizens: I AM again...
2     Saturday, March 4, 1797    WHEN it was first perceived...
3    Wednesday, March 4, 1801    Friends and Fellow-Citizens...
4       Monday, March 4, 1805    PROCEEDING, fellow-citizens...
```

Removing unwanted characters

- `[a-zA-Z]` : All letter characters
- `[^a-zA-Z]` : All non letter characters

```
speech_df['text'] = speech_df['text']\
                    .str.replace('[^a-zA-Z]', ' ')
```

Removing unwanted characters

Before:

```
"Fellow-Citizens of the Senate and of the House of
Representatives: AMONG the vicissitudes incident to
life no event could have filled me with greater" ...
```

After:

```
"Fellow Citizens of the Senate and of the House of
Representatives AMONG the vicissitudes incident to
life no event could have filled me with greater" ...
```

Standardize the case

```
speech_df['text'] = speech_df['text'].str.lower()
print(speech_df['text'][0])
```

```
"fellow citizens of the senate and of the house of
representatives among the vicissitudes incident to
life no event could have filled me with greater"...
```

Length of text

```
speech_df['char_cnt'] = speech_df['text'].str.len()
print(speech_df['char_cnt'].head())
```

```
0    1889
1     806
2    2408
3    1495
4    2465
Name: char_cnt, dtype: int64
```

Word counts

```
speech_df['word_cnt'] =
    speech_df['text'].str.split()
speech_df['word_cnt'].head(1)
```

```
['fellow', 'citizens', 'of', 'the', 'senate', 'and',...
```

Word counts

```
speech_df['word_counts'] =
    speech_df['text'].str.split().str.len()
print(speech_df['word_splits'].head())
```

```
0    1432
1     135
2    2323
3    1736
4    2169
Name: word_cnt, dtype: int64
```

Average length of word

```
speech_df['avg_word_len'] =
         speech_df['char_cnt'] / speech_df['word_cnt']
```

Text to columns

"citizens of the senate and of the house of representatives"

Index	citizens	of	the	senate	and	house	representatives
1	1	3	2	1	1	1	1

Taking just one sentence we can see that "of" occurs 3 tines, "the" 2 times and the other words once.

Initializing the vectorizer

```
from sklearn.feature_extraction.text import CountVectorizer
cv = CountVectorizer()
print(cv)
```

```
CountVectorizer(analyzer=u'word', binary=False,
        decode_error=u'strict',
        dtype=<type 'numpy.int64'>,
        encoding=u'utf-8', input=u'content',
        lowercase=True, max_df=1.0, max_features=None,
        min_df=1,ngram_range=(1, 1), preprocessor=None,
        stop_words=None, strip_accents=None,
        token_pattern=u'(?u)\\b\\w\\w+\\b',
        tokenizer=None, vocabulary=None
```

Specifying the vectorizer

```
from sklearn.feature_extraction.text import CountVectorizer

cv = CountVectorizer(min_df=0.1, max_df=0.9)
```

`min_df` : minimum fraction of documents the word must occur in
`max_df` : maximum fraction of documents the word can occur in

Transforming your text

```
cv_transformed = cv.transform(speech_df['text_clean'])
print(cv_transformed)
```

```
<58x8839 sparse matrix of type '<type 'numpy.int64'>'
```

Transforming your text

```
cv_transformed.toarray()
```

Getting the features

```
feature_names = cv.get_feature_names()
print(feature_names)
```

```
[u'abandon', u'abandoned', u'abandonment', u'abate',
u'abdicated', u'abeyance', u'abhorring', u'abide',
u'abiding', u'abilities', u'ability', u'abject'...
```

Fitting and transforming

```
cv_transformed = cv.fit_transform(speech_df['text_clean'])
print(cv_transformed)
```

```
<58x8839 sparse matrix of type '<type 'numpy.int64'>'
```

Putting it all together

```
cv_df = pd.DataFrame(cv_transformed.toarray(),
                     columns=cv.get_feature_names())\
                               .add_prefix('Counts_')
print(cv_df.head())
```

```
   Counts_aback  Counts_abandoned  Counts_a...
0             1                 0         ...
1             0                 0         ...
2             0                 1         ...
3             0                 1         ...
```

[1] ```out Counts_aback Counts_abandon Counts_abandonment 0 1 0 0 1 0 0 1 2 0 1 0 3 0 1 0 4 0 0 0```

Updating your DataFrame

```
speech_df = pd.concat([speech_df, cv_df],
                      axis=1, sort=False)
print(speech_df.shape)
```

```
(58, 8845)
```

Introducing TF-IDF

```
print(speech_df['Counts_the'].head())
```

```
0    21
1    13
2    29
3    22
4    20
```

TF-IDF

$$\text{TF-IDF} = \frac{\frac{Count\ of\ word\ occurances}{Total\ words\ in\ document}}{\log\left(\frac{Number\ of\ docs\ word\ is\ in}{Total\ number\ of\ docs}\right)}$$

Importing the vectorizer

```
from sklearn.feature_extraction.text import TfidfVectorizer
tv = TfidfVectorizer()
print(tv)
```

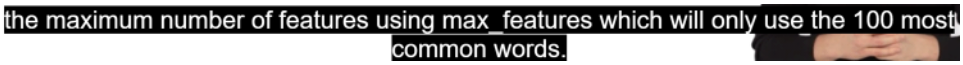
```
TfidfVectorizer(analyzer=u'word', binary=False, decode_error=
        dtype=<type 'numpy.float64'>, encoding=u'utf-8', inpu
        lowercase=True, max_df=1.0, max_features=None, min_df
        ngram_range=(1, 1), norm=u'l2', preprocessor=None, sm
        stop_words=None, strip_accents=None, sublinear_tf=Fal
        token_pattern=u'(?u)\\b\\w\\w+\\b', tokenizer=None, u
        vocabulary=None)
```

Max features and stopwords

```
tv = TfidfVectorizer(max_features=100,
                     stop_words='english')
```

`max_features` : Maximum number of columns created from TF-IDF

`stop_words` : List of common words to omit e.g. "and", "the" etc.



Fitting your text

```
tv.fit(train_speech_df['text'])
train_tv_transformed = tv.transform(train_speech_df['tex
```

Putting it all together

```
train_tv_df = pd.DataFrame(train_tv_transformed.toarray(),
                           columns=tv.get_feature_names())\
                                 .add_prefix('TFIDF_')

train_speech_df = pd.concat([train_speech_df, train_tv_df],
                            axis=1, sort=False)
```

Inspecting your transforms

```
examine_row = train_tv_df.iloc[0]
```

```
print(examine_row.sort_values(ascending=False))
```

```
TFIDF_government    0.367430
TFIDF_public        0.333237
TFIDF_present       0.315182
TFIDF_duty          0.238637
TFIDF_citizens      0.229644
Name: 0, dtype: float64
```

Applying the vectorizer to new data

```
test_tv_transformed = tv.transform(test_df['text_clean'])

test_tv_df = pd.DataFrame(test_tv_transformed.toarray(),
                          columns=tv.get_feature_names())\
                            .add_prefix('TFIDF_')

test_speech_df = pd.concat([test_speech_df, test_tv_df],
                           axis=1, sort=False)
```

Issues with bag of words

Positive meaning

Single word: *happy*

Negative meaning

Bi-gram : *not happy*

Positive meaning

Trigram : *never not happy*

Using N-grams

```
tv_bi_gram_vec = TfidfVectorizer(ngram_range = (2,2))

# Fit and apply bigram vectorizer
tv_bi_gram = tv_bi_gram_vec\
               .fit_transform(speech_df['text'])

# Print the bigram features
print(tv_bi_gram_vec.get_feature_names())
```

```
[u'american people', u'best ability ',
 u'beloved country', u'best interests' ... ]
```

Finding common words

```
# Create a DataFrame with the Counts features
tv_df = pd.DataFrame(tv_bi_gram.toarray(),
                     columns=tv_bi_gram_vec.get_feature_names())\
                        .add_prefix('Counts_')

tv_sums = tv_df.sum()
print(tv_sums.head())
```

```
Counts_administration government    12
Counts_almighty god                 15
Counts_american people              36
Counts_beloved country               8
Counts_best ability                  8
dtype: int64
```

Finding common words

```
print(tv_sums.sort_values(ascending=False)).head()
```

```
Counts_united states          152
Counts_fellow citizens         97
Counts_american people         36
Counts_federal government      35
Counts_self government         30
dtype: int64
```