Elements Of Data Science - F2021

Week 8: Data Cleaning and Feature Engineering

11/8/2021

TODOs

- Readings:
  - PDSH 5.9 PCA
  - [Recommended] PML v2:Ch7 (v3:Ch5) : Compressing Data via Dimensionality Reduction
  - [Recommended] Pandas: Merge, join, concatenate and compare
- **Quiz 8**, due **Sunday Nov 14th, 11:59pm ET**
- **HW3**, out Tues due **Tues Nov 23rd, 11:59pm ET**

Today

- **Data Cleaning**
    - Duplicates
    - Missing Data
    - Dummy Variables
    - Rescaling
    - Dealing With Skew
    - Removing Outliers
- **Feature Engineering**
    - Binning
    - One-Hot encoding
    - Derived
        - string functions
        - datetime functions

Questions?

Environment Setup

Environment Setup

In [1]:
```
import numpy
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
import seaborn as sns

from mlxtend.plotting import plot_decision_regions

sns.set_style('darkgrid')

%matplotlib inline
```

Data Cleaning

Why do we need clean data?

- Want one row per observation (need to remove duplicates)
- Most models cannot handle missing data (need to remove/fill missing)
- Most models require fixed length feature vectors (need to engineer features)

- Different models require different types of data (transformation)
    - Linear models: real valued features with similar scale
    - Distance based: real valued features with similar scale
    - Tree based: can handle unscaled real and categorical

Example Data

Example Data

In [2]:
```
# read in example data
df_shop_raw = pd.read_csv('../data/flowershop_data_with_dups.csv',
                          header=0,
                          parse_dates=['purchase_date'],
                          delimiter=',')

# make a copy for editing
df_shop = df_shop_raw.copy()

df_shop.info()
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 1001 entries, 0 to 1000
Data columns (total 6 columns):
 #   Column           Non-Null Count  Dtype
---  ------           --------------  -----
 0   purchase_id      1001 non-null   int64
 1   lastname         1001 non-null   object
 2   purchase_date    1001 non-null   datetime64[ns]
 3   stars            1001 non-null   int64
 4   price            979 non-null    float64
 5   favorite_flower  823 non-null    object
dtypes: datetime64[ns](1), float64(1), int64(2), object(2)
memory usage: 47.0+ KB
```

Duplicated Data

- Only drop duplicates if you know data should be unique
    - Example: if there is a unique id per row

Duplicated Data

- Only drop duplicates if you know data should be unique
  - Example: if there is a unique id per row

In [3]:
```
df_shop.duplicated().iloc[:3] # are first 3 rows duplicates?
```

Out[3]:
```
0    False
1    False
2    False
dtype: bool
```

Duplicated Data

- Only drop duplicates if you know data should be unique
    - Example: if there is a unique id per row

In [3]:
```
df_shop.duplicated().iloc[:3] # are first 3 rows duplicates?
```

Out[3]:
```
0    False
1    False
2    False
dtype: bool
```

In [4]:
```
df_shop[df_shop.duplicated(keep='first')] # default: keep first duplicated row
```

Out[4]:

	purchase_id	lastname	purchase_date	stars	price	favorite_flower
1000	1010	FERGUSON	2017-05-04	2	21.0183	daffodil

Duplicated Data

- Only drop duplicates if you know data should be unique
    - Example: if there is a unique id per row

In [3]:
```
df_shop.duplicated().iloc[:3] # are first 3 rows duplicates?
```

Out[3]:
```
0    False
1    False
2    False
dtype: bool
```

In [4]:
```
df_shop[df_shop.duplicated(keep='first')] # default: keep first duplicated row
```

Out[4]:

	purchase_id	lastname	purchase_date	stars	price	favorite_flower
1000	1010	FERGUSON	2017-05-04	2	21.0183	daffodil

In [5]:
```
df_shop[df_shop.duplicated(keep=False)] # keep=False to show all duplicated rows
```

Out[5]:

	purchase_id	lastname	purchase_date	stars	price	favorite_flower
10	1010	FERGUSON	2017-05-04	2	21.0183	daffodil
1000	1010	FERGUSON	2017-05-04	2	21.0183	daffodil

Duplicated Data for Subset of Columns

Duplicated Data for Subset of Columns

In [6]:
```
df_shop[df_shop.duplicated(subset=['purchase_id'],keep=False)].sort_values(by='purchase_id')
```

Out[6]:

	purchase_id	lastname	purchase_date	stars	price	favorite_flower
10	1010	FERGUSON	2017-05-04	2	21.018300	daffodil
1000	1010	FERGUSON	2017-05-04	2	21.018300	daffodil
100	1101	WEBB	2017-07-13	2	8.004356	iris
101	1101	BURKE	2017-08-16	4	18.560260	daffodil

Dropping Duplicates

Dropping Duplicates

In [7]:
```
df_new = df_shop.drop_duplicates(subset=None     # consider subset of columns
                                 ,keep='first'   # or 'last' or False)
                                 ,inplace=False)
```

Dropping Duplicates

In [7]:
```
df_new = df_shop.drop_duplicates(subset=None     # consider subset of columns
                                 ,keep='first'   # or 'last' or False)
                                 ,inplace=False)
```

In [8]:
```
# or can use inplace to change the original dataframe
df_shop.drop_duplicates(subset=None,keep='first',inplace=True)
```

Dropping Duplicates

In [7]:
```
df_new = df_shop.drop_duplicates(subset=None     # consider subset of columns
                                 ,keep='first'   # or 'last' or False)
                                 ,inplace=False)
```

In [8]:
```
# or can use inplace to change the original dataframe
df_shop.drop_duplicates(subset=None,keep='first',inplace=True)
```

In [9]:
```
# drop rows with duplicates considering only a subset of columns
df_shop = df_shop.drop_duplicates(subset=['purchase_id'])
```

Missing Data

- Reasons for missing data
  - Sensor error (random?)
  - Data entry error (random?)
  - Survey-subject decisions (non-random?)
  - etc.

Missing Data

- Reasons for missing data
    - Sensor error (random?)
    - Data entry error (random?)
    - Survey-subject decisions (non-random?)
    - etc.

- Dealing with missing data
    - Drop rows
    - Impute from data in the same column
    - Infer from other features
    - Fill with adjacent data

Missing Data in Pandas: `np.nan`

- Missing values represented by `np.nan` : Not A Number

Missing Data in Pandas: `np.nan`

- Missing values represented by `np.nan` : Not A Number

In [10]:
```
# Earlier, we saw missing values in the dataframe summary
df_shop.info()
```

```
<class 'pandas.core.frame.DataFrame'>
Int64Index: 999 entries, 0 to 999
Data columns (total 6 columns):
 #   Column           Non-Null Count  Dtype
---  ------           --------------  -----
 0   purchase_id      999 non-null    int64
 1   lastname         999 non-null    object
 2   purchase_date    999 non-null    datetime64[ns]
 3   stars            999 non-null    int64
 4   price            977 non-null    float64
 5   favorite_flower  821 non-null    object
dtypes: datetime64[ns](1), float64(1), int64(2), object(2)
memory usage: 54.6+ KB
```

Missing Data in Pandas: `np.nan`

- Missing values represented by `np.nan` : Not A Number

In [10]:
```
# Earlier, we saw missing values in the dataframe summary
df_shop.info()
```

```
<class 'pandas.core.frame.DataFrame'>
Int64Index: 999 entries, 0 to 999
Data columns (total 6 columns):
 #   Column           Non-Null Count  Dtype
---  ------           --------------  -----
 0   purchase_id      999 non-null    int64
 1   lastname         999 non-null    object
 2   purchase_date    999 non-null    datetime64[ns]
 3   stars            999 non-null    int64
 4   price            977 non-null    float64
 5   favorite_flower  821 non-null    object
dtypes: datetime64[ns](1), float64(1), int64(2), object(2)
memory usage: 54.6+ KB
```

In [11]:
```
# can we check for NaN using "x == np.nan"? No!
np.nan == np.nan
```

Out[11]:
```
False
```

Missing Data in Pandas: `np.nan`

- Missing values represented by `np.nan` : Not A Number

In [10]:
```
# Earlier, we saw missing values in the dataframe summary
df_shop.info()
```

```
<class 'pandas.core.frame.DataFrame'>
Int64Index: 999 entries, 0 to 999
Data columns (total 6 columns):
 #   Column           Non-Null Count  Dtype
---  ------           --------------  -----
 0   purchase_id      999 non-null    int64
 1   lastname         999 non-null    object
 2   purchase_date    999 non-null    datetime64[ns]
 3   stars            999 non-null    int64
 4   price            977 non-null    float64
 5   favorite_flower  821 non-null    object
dtypes: datetime64[ns](1), float64(1), int64(2), object(2)
memory usage: 54.6+ KB
```

In [11]:
```
# can we check for NaN using "x == np.nan"? No!
np.nan == np.nan
```

Out[11]: `False`

In [12]:
```
# however
np.nan is np.nan
```

Out[12]: `True`

How to check for NaN: `.isna()` and `.notna()`

How to check for NaN: `.isna()` and `.notna()`

In [13]:
```
# some missing data
df_shop.loc[20:21,'price']
```

Out[13]:
```
20          NaN
21    10.525912
Name: price, dtype: float64
```

How to check for NaN: `.isna()` and `.notna()`

In [13]:
```
# some missing data
df_shop.loc[20:21,'price']
```

Out[13]:
```
20          NaN
21    10.525912
Name: price, dtype: float64
```

In [14]:
```
# .isna() returns True where data is missing, False otherwise
df_shop.loc[20:21,'price'].isna()
```

Out[14]:
```
20     True
21    False
Name: price, dtype: bool
```

How to check for NaN: `.isna()` and `.notna()`

In [13]:
```
# some missing data
df_shop.loc[20:21,'price']
```

Out[13]:
```
20          NaN
21    10.525912
Name: price, dtype: float64
```

In [14]:
```
# .isna() returns True where data is missing, False otherwise
df_shop.loc[20:21,'price'].isna()
```

Out[14]:
```
20     True
21    False
Name: price, dtype: bool
```

In [15]:
```
# .notna() returns True where data is NOT missing, False otherwise
df_shop.loc[20:21,'price'].notna()
```

Out[15]:
```
20    False
21     True
Name: price, dtype: bool
```

How to check for NaN: `.isna()` and `.notna()`

In [13]:
```
# some missing data
df_shop.loc[20:21,'price']
```

Out[13]:
```
20          NaN
21    10.525912
Name: price, dtype: float64
```

In [14]:
```
# .isna() returns True where data is missing, False otherwise
df_shop.loc[20:21,'price'].isna()
```

Out[14]:
```
20     True
21    False
Name: price, dtype: bool
```

In [15]:
```
# .notna() returns True where data is NOT missing, False otherwise
df_shop.loc[20:21,'price'].notna()
```

Out[15]:
```
20    False
21     True
Name: price, dtype: bool
```

In [16]:
```
# find rows where price is missing
df_shop[df_shop.price.isna()].head()
```

Out[16]:

	purchase_id	lastname	purchase_date	stars	price	favorite_flower
20	1020	CLARK	2017-01-05	3	NaN	NaN
41	1041	PETERS	2017-02-01	4	NaN	orchid
54	1054	GREEN	2017-02-13	5	NaN	daffodil
63	1063	BARNETT	2017-08-27	4	NaN	gardenia
145	1145	CARROLL	2017-07-29	3	NaN	tulip

Counting NaNs

Counting NaNs

In [17]:
```
# How many nan's in a single column?
df_shop.price.isna().sum()
```

Out[17]:
```
22
```

Counting NaNs

In [17]:
```
# How many nan's in a single column?
df_shop.price.isna().sum()
```

Out[17]:
```
22
```

In [18]:
```
# How many nan's per column?
df_shop.isna().sum()
```

Out[18]:
```
purchase_id          0
lastname             0
purchase_date        0
stars                0
price               22
favorite_flower    178
dtype: int64
```

Counting NaNs

In [17]:
```
# How many nan's in a single column?
df_shop.price.isna().sum()
```

Out[17]:
```
22
```

In [18]:
```
# How many nan's per column?
df_shop.isna().sum()
```

Out[18]:
```
purchase_id          0
lastname             0
purchase_date        0
stars                0
price               22
favorite_flower    178
dtype: int64
```

In [19]:
```
# How many total nan's?
df_shop.isna().sum().sum()
```

Out[19]:
```
200
```

Missing Data: Drop Rows

Missing Data: Drop Rows

In [20]:
```
df_shop.shape
```

Out[20]:
```
(999, 6)
```

Missing Data: Drop Rows

In [20]:
```
df_shop.shape
```
Out[20]:
```
(999, 6)
```

In [21]:
```
# drop rows with nan in any column
df_shop.dropna().shape
```
Out[21]:
```
(801, 6)
```

Missing Data: Drop Rows

In [20]:
```
df_shop.shape
```

Out[20]: (999, 6)

In [21]:
```
# drop rows with nan in any column
df_shop.dropna().shape
```

Out[21]: (801, 6)

In [22]:
```
# drop only rows with nan in price using subset
df_shop.dropna(subset=['price']).shape
```

Out[22]: (977, 6)

Missing Data: Drop Rows

In [20]:
```
df_shop.shape
```

Out[20]: (999, 6)

In [21]:
```
# drop rows with nan in any column
df_shop.dropna().shape
```

Out[21]: (801, 6)

In [22]:
```
# drop only rows with nan in price using subset
df_shop.dropna(subset=['price']).shape
```

Out[22]: (977, 6)

In [23]:
```
# drop only rows with nans in all columns
df_shop.dropna(how='all').shape
```

Out[23]: (999, 6)

Missing Data: Drop Rows Cont.

Missing Data: Drop Rows Cont.

In [24]:
```python
# save a new dataframe with dropped rows
df_shop_nodups = df_shop.dropna()
df_shop_nodups.shape
```

Out[24]:
```
(801, 6)
```

Missing Data: Drop Rows Cont.

In [24]:
```
# save a new dataframe with dropped rows
df_shop_nodups = df_shop.dropna()
df_shop_nodups.shape
```

Out[24]: (801, 6)

In [25]:
```
# drop rows in current dataframe
df_shop_nodups = df_shop.copy()

df_shop_nodups.dropna(inplace=True)
df_shop_nodups.shape
```

Out[25]: (801, 6)

Missing Data: Drop Rows

- Pros:
  - easy to do
  - simple to understand
- Cons:
  - potentially large data loss

Missing Data: Fill with Constant

- Use .fillna()
- Common filler: 0, -1

Missing Data: Fill with Constant

- Use .fillna()
- Common filler: 0, -1

In [26]:
```
df_shop.price[20:22]
```

Out[26]:
```
20          NaN
21    10.525912
Name: price, dtype: float64
```

Missing Data: Fill with Constant

- Use .fillna()
- Common filler: 0, -1

In [26]:
```
df_shop.price[20:22]
```

Out[26]:
```
20          NaN
21    10.525912
Name: price, dtype: float64
```

In [27]:
```
df_shop.price[20:22].fillna(0)
```

Out[27]:
```
20     0.000000
21    10.525912
Name: price, dtype: float64
```

Missing Data: Fill with Constant

Pros:

- easy to do
- simple to understand

Cons:

- values may not be realistic

Missing Data: Impute

- Impute: fill with value infered from existing values in that column
- Use .fillna() or sklearn methods
- Common filler values:
  - mean
  - median
  - "most frequent" aka mode

Missing Data: Impute

Missing Data: Impute

In [28]:
```
df_shop.price.mean()
```

Out[28]:
```
23.408197893394266
```

Missing Data: Impute

In [28]:
```
df_shop.price.mean()
```

Out[28]: 23.408197893394266

In [29]:
```
# make a copy to keep our original df
df_shop_impute = df_shop.copy()
```

Missing Data: Impute

In [28]:
```
df_shop.price.mean()
```

Out[28]:
```
23.408197893394266
```

In [29]:
```
# make a copy to keep our original df
df_shop_impute = df_shop.copy()
```

In [30]:
```
# fill missing price with mean of price
df_shop_impute.price = df_shop.price.fillna(df_shop.price.mean())
```

Missing Data: Impute

In [28]:
```
df_shop.price.mean()
```

Out[28]:
```
23.408197893394266
```

In [29]:
```
# make a copy to keep our original df
df_shop_impute = df_shop.copy()
```

In [30]:
```
# fill missing price with mean of price
df_shop_impute.price = df_shop.price.fillna(df_shop.price.mean())
```

In [31]:
```
# check to make sure all nulls filled
assert df_shop_impute.price.isna().sum() == 0
```

Missing Data: Impute

In [28]:
```
df_shop.price.mean()
```

Out[28]:
```
23.408197893394266
```

In [29]:
```
# make a copy to keep our original df
df_shop_impute = df_shop.copy()
```

In [30]:
```
# fill missing price with mean of price
df_shop_impute.price = df_shop.price.fillna(df_shop.price.mean())
```

In [31]:
```
# check to make sure all nulls filled
assert df_shop_impute.price.isna().sum() == 0
```

In [32]:
```
# inplace works here as well
df_shop_impute.price.fillna(df_shop_impute.price.mean(),inplace=True)
```

Missing Data: Impute Cont.

Missing Data: Impute Cont.

In [33]:
```
df_shop.favorite_flower.mode()
```

Out[33]:
```
0    lilac
dtype: object
```

Missing Data: Impute Cont.

In [33]:
```
df_shop.favorite_flower.mode()
```

Out[33]:
```
0    lilac
dtype: object
```

In [34]:
```
# can also handle categorical data
df_shop_impute.favorite_flower.fillna(df_shop_impute.favorite_flower.mode().iloc[0],inplace=True)

df_shop_impute.info()
```

```
<class 'pandas.core.frame.DataFrame'>
Int64Index: 999 entries, 0 to 999
Data columns (total 6 columns):
 #   Column           Non-Null Count  Dtype
---  ------           --------------  -----
 0   purchase_id      999 non-null    int64
 1   lastname         999 non-null    object
 2   purchase_date    999 non-null    datetime64[ns]
 3   stars            999 non-null    int64
 4   price            999 non-null    float64
 5   favorite_flower  999 non-null    object
dtypes: datetime64[ns](1), float64(1), int64(2), object(2)
memory usage: 86.9+ KB
```

Missing Data: Impute Cont. Using SimpleImputer

Missing Data: Impute Cont. Using SimpleImputer

In [35]:
```
df_shop.price.loc[20:22]
```

Out[35]:
```
20          NaN
21    10.525912
22    19.771789
Name: price, dtype: float64
```

Missing Data: Impute Cont. Using SimpleImputer

In [35]:
```
df_shop.price.loc[20:22]
```

Out[35]:
```
20          NaN
21    10.525912
22    19.771789
Name: price, dtype: float64
```

In [36]:
```
from sklearn.impute import SimpleImputer

imp = SimpleImputer(strategy='mean').fit(df_shop[['price']])
imp.transform(df_shop.loc[20:22,['price']])
```

Out[36]:
```
array([[23.40819789],
       [10.52591185],
       [19.77178904]])
```

Missing Data: Impute Cont. Using SimpleImputer

In [35]:
```
df_shop.price.loc[20:22]
```

Out[35]:
```
20          NaN
21    10.525912
22    19.771789
Name: price, dtype: float64
```

In [36]:
```
from sklearn.impute import SimpleImputer

imp = SimpleImputer(strategy='mean').fit(df_shop[['price']])
imp.transform(df_shop.loc[20:22,['price']])
```

Out[36]:
```
array([[23.40819789],
       [10.52591185],
       [19.77178904]])
```

In [37]:
```
df_shop.favorite_flower[:3]
```

Out[37]:
```
0         iris
1          NaN
2    carnation
Name: favorite_flower, dtype: object
```

Missing Data: Impute Cont. Using SimpleImputer

In [35]:
```
df_shop.price.loc[20:22]
```

Out[35]:
```
20          NaN
21    10.525912
22    19.771789
Name: price, dtype: float64
```

In [36]:
```
from sklearn.impute import SimpleImputer

imp = SimpleImputer(strategy='mean').fit(df_shop[['price']])
imp.transform(df_shop.loc[20:22,['price']])
```

Out[36]:
```
array([[23.40819789],
       [10.52591185],
       [19.77178904]])
```

In [37]:
```
df_shop.favorite_flower[:3]
```

Out[37]:
```
0         iris
1          NaN
2     carnation
Name: favorite_flower, dtype: object
```

In [38]:
```
imp = SimpleImputer(strategy='most_frequent').fit(df_shop[['favorite_flower']])
imp.transform(df_shop.loc[:2,['favorite_flower']])
```

Out[38]:
```
array([['iris'],
       ['lilac'],
       ['carnation']], dtype=object)
```

Missing Data: Impute

- Pros:
  - easy to do
  - simple to understand
- Cons:
  - may missing feature interactions

Missing Data: Infer

- Predict values of missing features using a model
- Ex: Can we predict price based on any of the other features?
- Additional feature engineering may be needed prior to this

Missing Data: Infer

- Predict values of missing features using a model
- Ex: Can we predict price based on any of the other features?
- Additional feature engineering may be needed prior to this

In [39]:
```
from sklearn.linear_model import LinearRegression

df_shop_infer = df_shop.copy()

not_missing = df_shop_infer.price.notna()
missing = df_shop_infer.price.isna()

lr = LinearRegression().fit(df_shop_infer.loc[not_missing,['stars']],
                            df_shop_infer[not_missing].price)

df_shop_infer.loc[missing,'price'] = lr.predict(df_shop_infer.loc[missing,['stars']])
```

Missing Data: Adjacent Data

- Use `.fillna()` with method:
  - ffill: propagate last valid observation forward to next valid
  - bfill: use next valid observation to fill gap backwards
- Use when there is reason to believe data not i.i.d. (eg: timeseries)

Missing Data: Adjacent Data

- Use `.fillna()` with method:
  - ffill: propagate last valid observation forward to next valid
  - bfill: use next valid observation to fill gap backwards
- Use when there is reason to believe data not i.i.d. (eg: timeseries)

In [40]:
```
df_shop.price.loc[19:21]
```

Out[40]:
```
19    20.451789
20          NaN
21    10.525912
Name: price, dtype: float64
```

Missing Data: Adjacent Data

- Use `.fillna()` with method:
  - ffill: propagate last valid observation forward to next valid
  - bfill: use next valid observation to fill gap backwards
- Use when there is reason to believe data not i.i.d. (eg: timeseries)

In [40]:
```
df_shop.price.loc[19:21]
```

Out[40]:
```
19    20.451789
20          NaN
21    10.525912
Name: price, dtype: float64
```

In [41]:
```
df_shop.price.fillna(method='ffill').loc[19:21]
```

Out[41]:
```
19    20.451789
20    20.451789
21    10.525912
Name: price, dtype: float64
```

Missing Data: Dummy Columns

- Data may be missing for a reason!
- Capture "missing" before filling

Missing Data: Dummy Columns

- Data may be missing for a reason!
- Capture "missing" before filling

In [42]:
```
df_shop_dummy = df_shop.copy()

# storing a column of 1:missing, 0:not-missing
df_shop_dummy['price_missing'] = df_shop.price.isna().astype(int)

# can now fill missing values
df_shop_dummy['price'] = df_shop.price.fillna(df_shop.price.mean())
```

Missing Data: Dummy Columns

- Data may be missing for a reason!
- Capture "missing" before filling

In [42]:
```
df_shop_dummy = df_shop.copy()

# storing a column of 1:missing, 0:not-missing
df_shop_dummy['price_missing'] = df_shop.price.isna().astype(int)

# can now fill missing values
df_shop_dummy['price'] = df_shop.price.fillna(df_shop.price.mean())
```

In [43]:
```
# finding where data was missing
np.where(df_shop_dummy.price_missing == 1)
```

Out[43]:
```
(array([ 20,  41,  54,  63, 144, 185, 193, 202, 211, 359, 366, 381, 428,
        468, 521, 569, 594, 725, 791, 820, 973, 977]),)
```

Missing Data: Dummy Columns

- Data may be missing for a reason!
- Capture "missing" before filling

In [42]:
```
df_shop_dummy = df_shop.copy()

# storing a column of 1:missing, 0:not-missing
df_shop_dummy['price_missing'] = df_shop.price.isna().astype(int)

# can now fill missing values
df_shop_dummy['price'] = df_shop.price.fillna(df_shop.price.mean())
```

In [43]:
```
# finding where data was missing
np.where(df_shop_dummy.price_missing == 1)
```

Out[43]:
```
(array([ 20,  41,  54,  63, 144, 185, 193, 202, 211, 359, 366, 381, 428,
        468, 521, 569, 594, 725, 791, 820, 973, 977]),)
```

In [44]:
```
df_shop_dummy[['price','price_missing']].iloc[20:23]
```

Out[44]:

	price	price_missing
20	23.408198	1
21	10.525912	0
22	19.771789	0

Rescaling

- Often need features to be in the same scale
- Methods of rescaling
    - Standardization (z-score)
    - Min-Max rescaling
    - others...

Rescaling

- Often need features to be in the same scale
- Methods of rescaling
    - Standardization (z-score)
    - Min-Max rescaling
    - others...

In [45]:
```
# load taxi data
df_taxi = pd.read_csv('../data/yellowcab_tripdata_2017-01_subset10000rows.csv',
                      parse_dates=['tpep_pickup_datetime','tpep_dropoff_datetime'])

# create trip_duration
df_taxi['trip_duration'] = (df_taxi.tpep_dropoff_datetime - df_taxi.tpep_pickup_datetime).dt.seconds

# select subset
df_taxi = df_taxi[(df_taxi.trip_duration < 3600) & (df_taxi.tip_amount > 0) & (df_taxi.tip_amount < 10)]
```

Rescaling

- Often need features to be in the same scale
- Methods of rescaling
    - Standardization (z-score)
    - Min-Max rescaling
    - others...

In [45]:
```
# load taxi data
df_taxi = pd.read_csv('../data/yellowcab_tripdata_2017-01_subset10000rows.csv',
                      parse_dates=['tpep_pickup_datetime','tpep_dropoff_datetime'])

# create trip_duration
df_taxi['trip_duration'] = (df_taxi.tpep_dropoff_datetime - df_taxi.tpep_pickup_datetime).dt.seconds

# select subset
df_taxi = df_taxi[(df_taxi.trip_duration < 3600) & (df_taxi.tip_amount > 0) & (df_taxi.tip_amount < 10)]
```

In [46]:
```
df_taxi[['trip_duration','tip_amount']].agg(['mean','std','min','max'],axis=0)
```

Out[46]:

	trip_duration	tip_amount
mean	765.030683	2.405944
std	496.831608	1.552848
min	2.000000	0.010000
max	3556.000000	9.990000

Rescaling: Standardization

- rescale to 0 mean, standard deviation of 1
  - X_scaled = (X - X.mean()) / X.std()

Rescaling: Standardization

- rescale to 0 mean, standard deviation of 1
  - X_scaled = (X - X.mean()) / X.std()

In [47]:
```
from sklearn.preprocessing import StandardScaler

# instantiate
ss = StandardScaler() # default is center and scale

# fit to the data
ss.fit(df_taxi[['trip_duration','tip_amount']])

# transform the data
X_ss = ss.transform(df_taxi[['trip_duration','tip_amount']])
X_ss[:2]
```

Out[47]:
```
array([[-0.50127786, -0.48040987],
       [-0.16512088, -0.90546941]])
```

Rescaling: Standardization

- rescale to 0 mean, standard deviation of 1
    - X_scaled = (X - X.mean()) / X.std()

In [47]:
```
from sklearn.preprocessing import StandardScaler

# instantiate
ss = StandardScaler() # default is center and scale

# fit to the data
ss.fit(df_taxi[['trip_duration','tip_amount']])

# transform the data
X_ss = ss.transform(df_taxi[['trip_duration','tip_amount']])
X_ss[:2]
```

Out[47]:
```
array([[-0.50127786, -0.48040987],
       [-0.16512088, -0.90546941]])
```

In [48]:
```
df_taxi_ss = pd.DataFrame(X_ss,columns=['trip_duration_scaled','tip_amount_scaled'])
df_taxi_ss.agg(['mean','std','min','max'],axis=0)
```

Out[48]:

	trip_duration_scaled	tip_amount_scaled
mean	4.622808e-17	-1.358307e-16
std	1.000080e+00	1.000080e+00
min	-1.535917e+00	-1.543059e+00
max	5.617987e+00	4.884357e+00

Rescaling: Min-Max

- rescale values between 0 and 1
- X_scaled = (X - X.min()) / (X.max() - X.min())
- removes negative values

Rescaling: Min-Max

- rescale values between 0 and 1
- X_scaled = (X - X.min()) / (X.max() - X.min())
- removes negative values

In [49]:
```
from sklearn.preprocessing import MinMaxScaler

# default is to rescale between 0 and 1
X_mms = MinMaxScaler(feature_range=(0,1)).fit_transform(df_taxi[['trip_duration','tip_amount']])

df_taxi_mms = pd.DataFrame(X_mms,columns=['trip_duration_scaled','tip_amount_scaled'])
df_taxi_mms.agg(['mean','std','min','max'])
```

Out[49]:

	trip_duration_scaled	tip_amount_scaled
mean	0.214696	0.240075
std	0.139795	0.155596
min	0.000000	0.000000
max	1.000000	1.000000

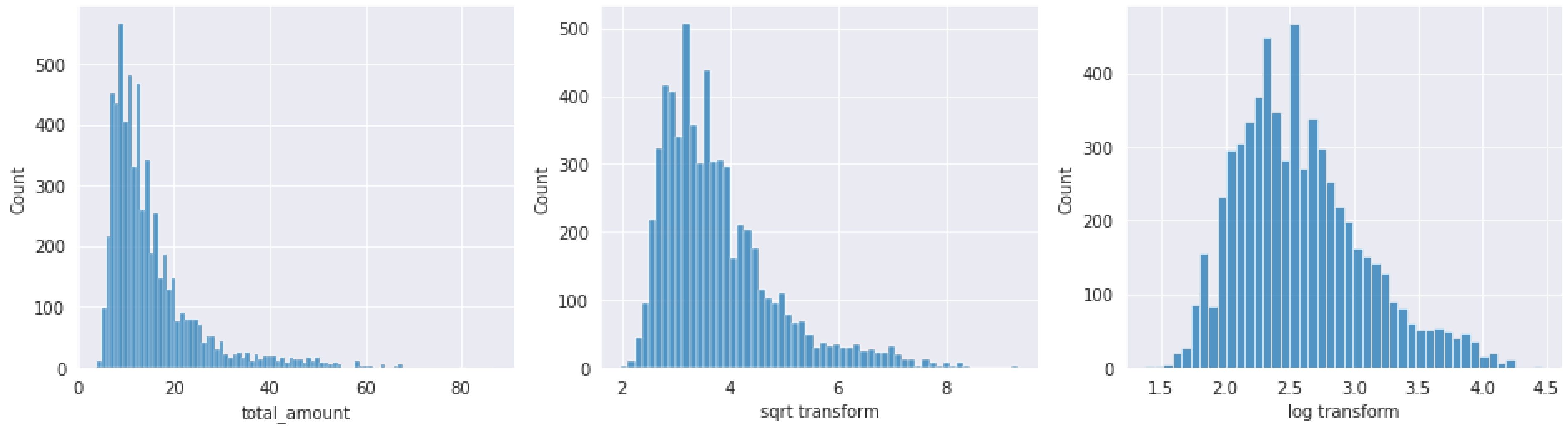
Dealing with Skew

- Many models expect "normal", symmetric data (ex: linear models)
- Highly skewed: tail has larger effect on model (outliers?)
- Transform with `log` or `sqrt`

Dealing with Skew

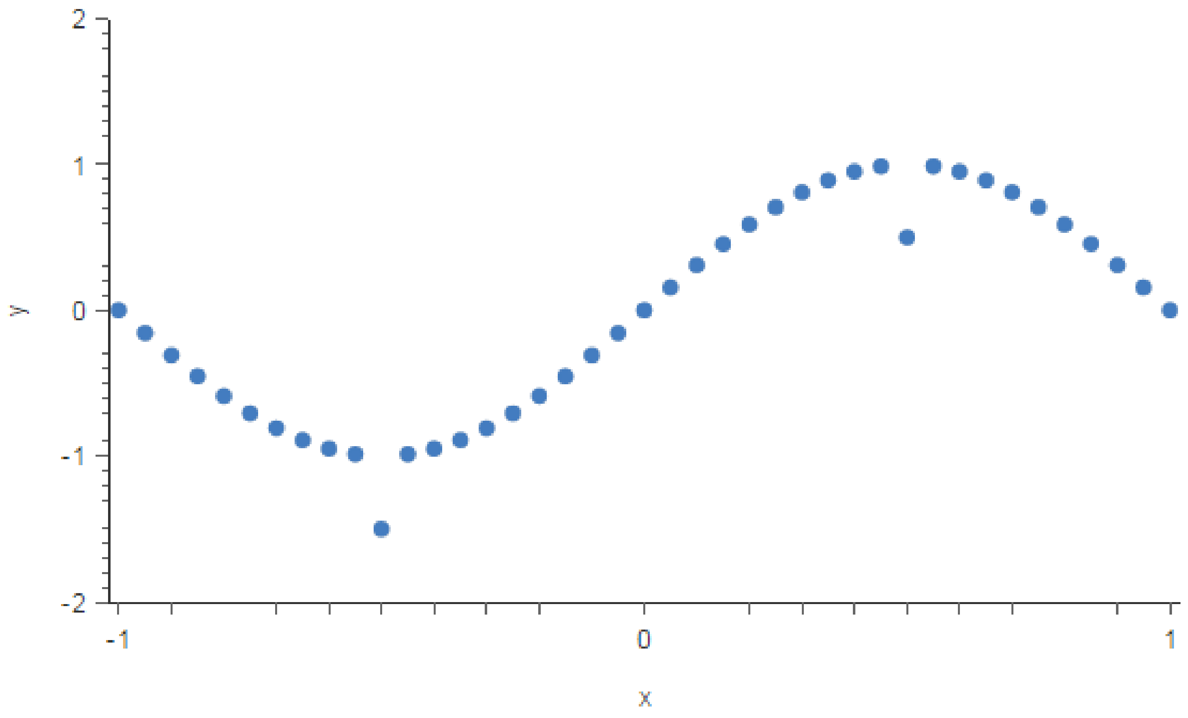
- Many models expect "normal", symmetric data (ex: linear models)
- Highly skewed: tail has larger effect on model (outliers?)
- Transform with `log` or `sqrt`

In [50]:
```
fig,ax = plt.subplots(1,3,figsize=(16,4))
sns.histplot(x=df_taxi.total_amount, ax=ax[0]);
sns.histplot(x=df_taxi.total_amount.apply(np.sqrt), ax=ax[1]); ax[1].set_xlabel('sqrt transform');
sns.histplot(x=df_taxi.total_amount.apply(np.log),  ax=ax[2]); ax[2].set_xlabel('log transform');
```



Outliers

- Similar to missing data:
    - human data entry error
    - instrument measurement errors
    - data processing errors
    - natural deviations



Outliers

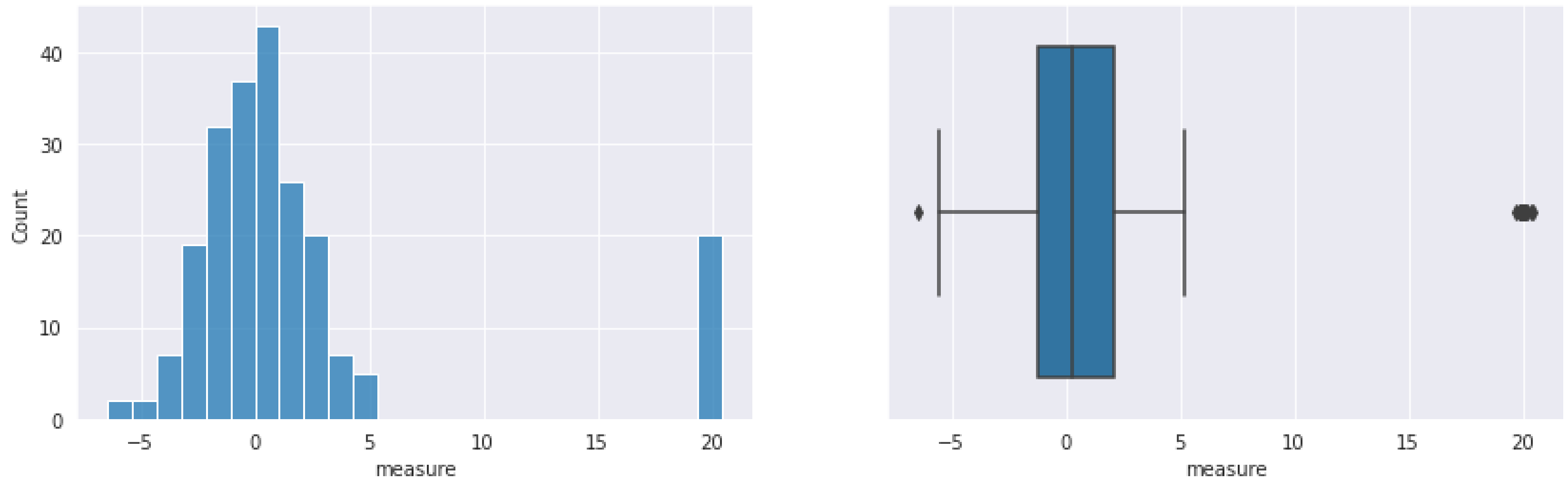
- Why worry about them?
    - can give misleading results
    - can indicate issues in data/measurement

- Detecting Outliers
    - understand your data!
    - visualizations
    - 1.5*IQR
    - z-scores
    - etc..

Detecting Outliers

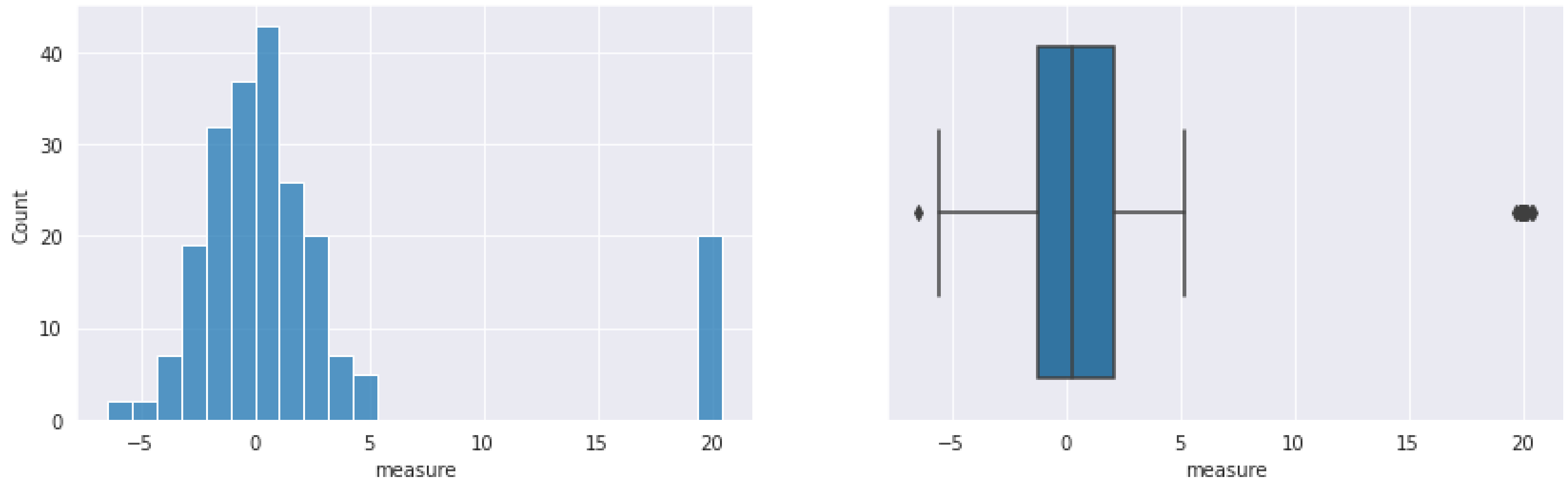
Detecting Outliers

In [51]:
```
np.random.seed(123)
df_rand = pd.DataFrame(np.random.normal(0,2,200), columns=['measure'])
df_rand = df_rand.append(pd.DataFrame(np.random.normal(20,.2,20), columns=['measure'])).reset_index(drop=True)

fig,ax = plt.subplots(1,2, figsize=(14,4))
sns.histplot(x=df_rand.measure,ax=ax[0]);sns.boxplot(x=df_rand.measure,ax=ax[1]);
```



Detecting Outliers

In [51]:
```
np.random.seed(123)
df_rand = pd.DataFrame(np.random.normal(0,2,200), columns=['measure'])
df_rand = df_rand.append(pd.DataFrame(np.random.normal(20,.2,20), columns=['measure'])).reset_index(drop=True)

fig,ax = plt.subplots(1,2, figsize=(14,4))
sns.histplot(x=df_rand.measure,ax=ax[0]);sns.boxplot(x=df_rand.measure,ax=ax[1]);
```

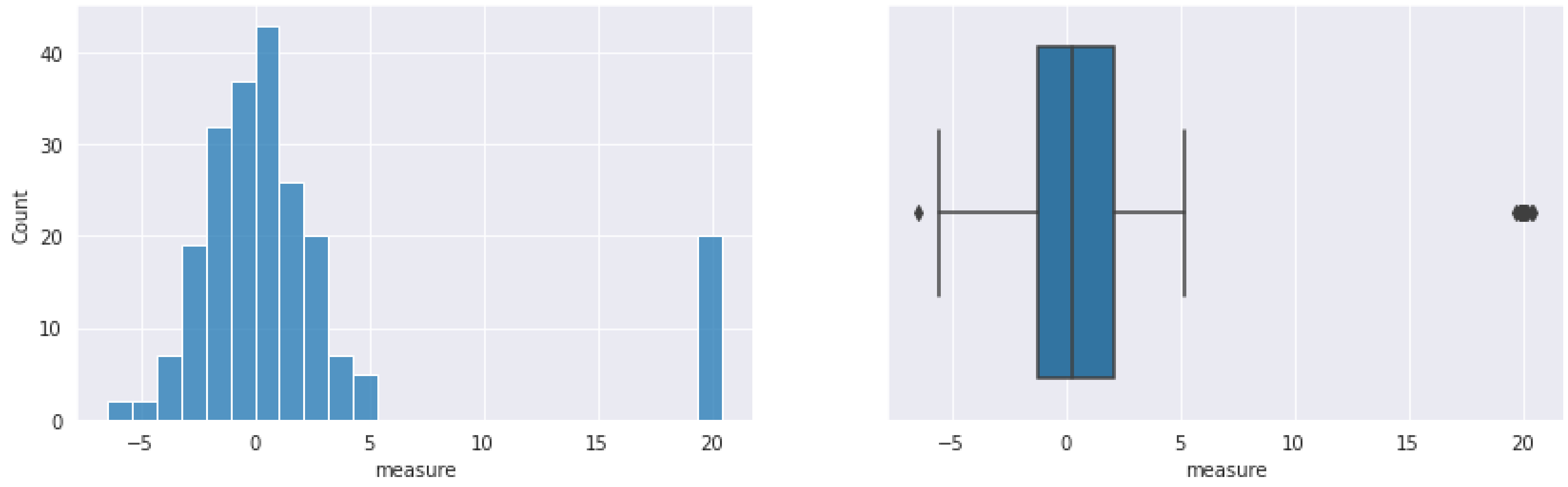


In [52]:
```
# Calculating IQR
p25,p75 = df_rand.measure.quantile(.25),df_rand.measure.quantile(.75)
iqr = p75 - p25
iqr.round(2)
```

Out[52]: 3.3

Detecting Outliers

In [51]:
```
np.random.seed(123)
df_rand = pd.DataFrame(np.random.normal(0,2,200), columns=['measure'])
df_rand = df_rand.append(pd.DataFrame(np.random.normal(20,.2,20), columns=['measure'])).reset_index(drop=True)

fig,ax = plt.subplots(1,2, figsize=(14,4))
sns.histplot(x=df_rand.measure,ax=ax[0]);sns.boxplot(x=df_rand.measure,ax=ax[1]);
```



In [52]:
```
# Calculating IQR
p25,p75 = df_rand.measure.quantile(.25),df_rand.measure.quantile(.75)
iqr = p75 - p25
iqr.round(2)
```

Out[52]: 3.3

In [53]:
```
# Finding outliers with IQR
df_rand.measure[(df_rand.measure > p75+(1.5*iqr)) | (df_rand.measure < p25-(1.5*iqr))].sort_values().head(2).round(2)
```

Out[53]:
```
195    -6.46
213    19.72
Name: measure, dtype: float64
```

Detecting Outliers with z-score

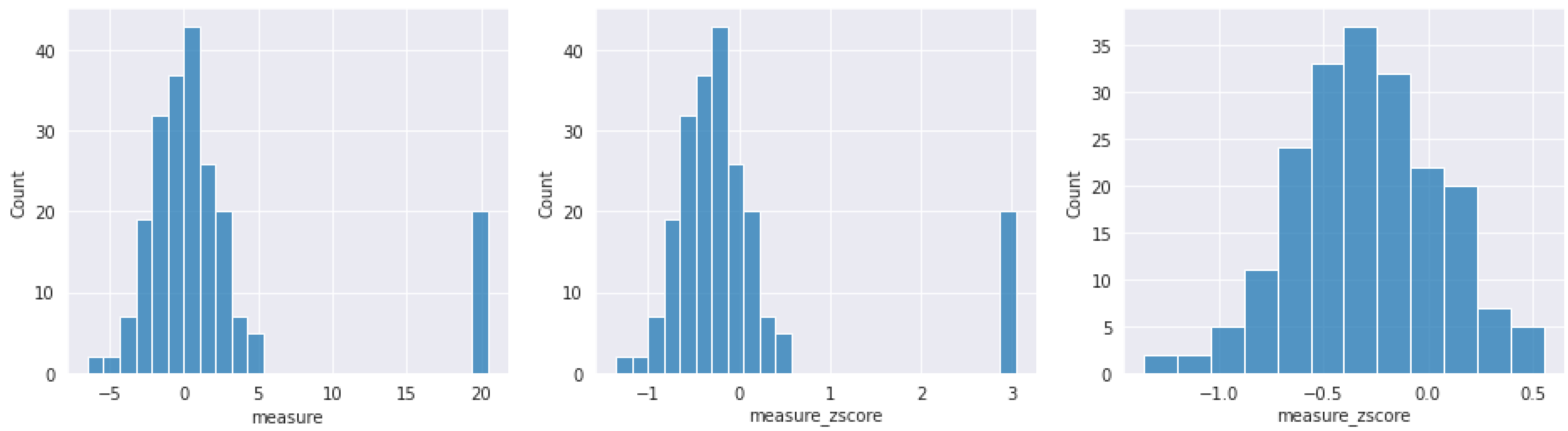
Detecting Outliers with z-score

In [54]:

```
# zscore
df_rand['measure_zscore'] = (df_rand.measure - df_rand.measure.mean()) / df_rand.measure.std()

fig, ax = plt.subplots(1,3,figsize=(16,4))
sns.histplot(x=df_rand.measure,ax=ax[0]);
sns.histplot(x=df_rand.measure_zscore, ax=ax[1]);

keep_idx = np.abs(df_rand.measure_zscore) < 2
sns.histplot(x=df_rand[keep_idx].measure_zscore, ax=ax[2]);

# sample of points getting dropped
df_rand[np.abs(df_rand.measure_zscore) >= 2].sort_values(by='measure').head(3).round(2)
```

Out[54]:

	measure	measure_zscore
213	19.72	2.93
207	19.82	2.94
218	19.85	2.95



Other Outlier Detection Methods

- Many more parametric and non-parametric methods
    - Standardized Residuals
    - DBScan
    - ElipticEnvelope
    - IsolationForest
    - other Anomoly Detection techniques
    - See sklearn docs on Outlier Detection for more details

Dealing with Outliers

- How to deal with outliers?
  - drop data
  - treat as missing
  - encode with dummy variable first?

Data Cleaning Review

- duplicate data
- missing data
- rescaling
- dealing with skew
- outlier detection

Feature Engineering

- Binning
- One-Hot encoding
- Derived

Binning

- Transform continuous features to categorical
- Use:
  - pd.cut
  - sklearn.preprocessing.KBinsDiscretizer (combined binning and one-hot-encoding)

Binning

- Transform continuous features to categorical
- Use:
    - pd.cut
    - sklearn.preprocessing.KBinsDiscretizer (combined binning and one-hot-encoding)

In [55]:
```
trip_duration_bins = [df_taxi.trip_duration.min(),
                      df_taxi.trip_duration.median(),
                      df_taxi.trip_duration.quantile(0.75),
                      df_taxi.trip_duration.max(),]
```

Binning

- Transform continuous features to categorical
- Use:
  - pd.cut
  - sklearn.preprocessing.KBinsDiscretizer (combined binning and one-hot-encoding)

In [55]:
```
trip_duration_bins = [df_taxi.trip_duration.min(),
                      df_taxi.trip_duration.median(),
                      df_taxi.trip_duration.quantile(0.75),
                      df_taxi.trip_duration.max(),]
```

In [56]:
```
df_taxi_bin = df_taxi.copy()
df_taxi_bin['trip_duration_binned'] = pd.cut(df_taxi.trip_duration,
                                             bins=trip_duration_bins,          # can pass bin edges or number of bins
                                             labels=['short','medium','long'],
                                             right=True,                       # all bins right-inclusive
                                             include_lowest=True               # first interval left-inclusive
                                            )
df_taxi_bin[['trip_duration','trip_duration_binned']].iloc[:3]
```

Out[56]:

	trip_duration	trip_duration_binned
1	516	short
2	683	medium
7	834	medium

One-Hot Encoding

- Encode categorical features for models that can't handle categorical (eg. Linear)
- One column per category, `'1'` in only one column per row
- Use `pd.get_dummies()` or `sklearn.preprocessing.OneHotEncoder`

One-Hot Encoding

- Encode categorical features for models that can't handle categorical (eg. Linear)
- One column per category, '1' in only one column per row
- Use `pd.get_dummies()` or `sklearn.preprocessing.OneHotEncoder`

In [57]:
```
pd.get_dummies(df_taxi_bin.trip_duration_binned, prefix='trip_duration').iloc[:2]
```

Out[57]:

	trip_duration_short	trip_duration_medium	trip_duration_long
1	1	0	0
2	0	1	0

One-Hot Encoding

- Encode categorical features for models that can't handle categorical (eg. Linear)
- One column per category, '1' in only one column per row
- Use `pd.get_dummies()` or `sklearn.preprocessing.OneHotEncoder`

In [57]:
```
pd.get_dummies(df_taxi_bin.trip_duration_binned, prefix='trip_duration').iloc[:2]
```

Out[57]:

	trip_duration_short	trip_duration_medium	trip_duration_long
1	1	0	0
2	0	1	0

In [58]:
```
# to add back to dataframe, use join (will discuss .join() next time)
df_taxi_bin.join(pd.get_dummies(df_taxi_bin.trip_duration_binned, prefix='trip_duration')).iloc[:2,-6:] # not being saved
```

Out[58]:

	total_amount	trip_duration	trip_duration_binned	trip_duration_short	trip_duration_medium	trip_duration_long
1	9.96	516	short	1	0	0
2	10.30	683	medium	0	1	0

One-Hot Encoding

- Encode categorical features for models that can't handle categorical (eg. Linear)
- One column per category, '1' in only one column per row
- Use `pd.get_dummies()` or `sklearn.preprocessing.OneHotEncoder`

In [57]:
```
pd.get_dummies(df_taxi_bin.trip_duration_binned, prefix='trip_duration').iloc[:2]
```

Out[57]:

	trip_duration_short	trip_duration_medium	trip_duration_long
1	1	0	0
2	0	1	0

In [58]:
```
# to add back to dataframe, use join (will discuss .join() next time)
df_taxi_bin.join(pd.get_dummies(df_taxi_bin.trip_duration_binned, prefix='trip_duration')).iloc[:2,-6:] # not being saved
```

Out[58]:

	total_amount	trip_duration	trip_duration_binned	trip_duration_short	trip_duration_medium	trip_duration_long
1	9.96	516	short	1	0	0
2	10.30	683	medium	0	1	0

In [59]:
```
# or let pandas determine which columns to one-hot
pd.get_dummies(df_taxi_bin).iloc[:2,-6:] # not being saved
```

Out[59]:

	trip_duration	store_and_fwd_flag_N	store_and_fwd_flag_Y	trip_duration_binned_short	trip_duration_binned_medium	trip_duration_binned_long
1	516	1	0	1	0	0
2	683	1	0	0	1	0

One-Hot Encoding with sklearn

One-Hot Encoding with sklearn

In [60]:
```
from sklearn.preprocessing import OneHotEncoder

ohe = OneHotEncoder(categories=[['short','medium','long']],  # or leave as 'auto'
                    sparse=True,
                    handle_unknown='ignore')                 # will raise error otherwise

ohe.fit(df_taxi_bin[['trip_duration_binned']])
ohe.categories_
```

Out[60]:
```
[array(['short', 'medium', 'long'], dtype=object)]
```

One-Hot Encoding with sklearn

In [60]:
```
from sklearn.preprocessing import OneHotEncoder

ohe = OneHotEncoder(categories=[['short','medium','long']],  # or leave as 'auto'
                    sparse=True,
                    handle_unknown='ignore')                 # will raise error otherwise

ohe.fit(df_taxi_bin[['trip_duration_binned']])
ohe.categories_
```

Out[60]:
```
[array(['short', 'medium', 'long'], dtype=object)]
```

In [61]:
```
ohe.transform(df_taxi_bin[['trip_duration_binned']])[:3]
```

Out[61]:
```
<3x3 sparse matrix of type '<class 'numpy.float64'>'
	with 3 stored elements in Compressed Sparse Row format>
```

One-Hot Encoding with sklearn

In [60]:
```
from sklearn.preprocessing import OneHotEncoder

ohe = OneHotEncoder(categories=[['short','medium','long']],  # or leave as 'auto'
                    sparse=True,
                    handle_unknown='ignore')                 # will raise error otherwise

ohe.fit(df_taxi_bin[['trip_duration_binned']])
ohe.categories_
```

Out[60]:
```
[array(['short', 'medium', 'long'], dtype=object)]
```

In [61]:
```
ohe.transform(df_taxi_bin[['trip_duration_binned']])[:3]
```

Out[61]:
```
<3x3 sparse matrix of type '<class 'numpy.float64'>'
	with 3 stored elements in Compressed Sparse Row format>
```

In [62]:
```
ohe.transform(df_taxi_bin[['trip_duration_binned']])[:3].todense()
```

Out[62]:
```
matrix([[1., 0., 0.],
        [0., 1., 0.],
        [0., 1., 0.]])
```

Bin and One-Hot Encode with sklearn

Bin and One-Hot Encode with sklearn

In [63]:
```
from sklearn.preprocessing import KBinsDiscretizer

# NOTE: We're not setting the bin edges explicitly
#       For control over bin edges, use Binarizer
kbd = KBinsDiscretizer(n_bins=3,
                       encode="onehot",     # or onehot (sparse), ordinal
                       strategy="quantile", # or uniform or kmeans (clustering)
                      ).fit(df_taxi[['trip_duration']])

kbd.bin_edges_
```

Out[63]:
```
array([array([2.000e+00, 4.780e+02, 8.700e+02, 3.556e+03])], dtype=object)
```

Bin and One-Hot Encode with sklearn

In [63]:
```python
from sklearn.preprocessing import KBinsDiscretizer

# NOTE: We're not setting the bin edges explicitly
#       For control over bin edges, use Binarizer
kbd = KBinsDiscretizer(n_bins=3,
                       encode="onehot",     # or onehot (sparse), ordinal
                       strategy="quantile", # or uniform or kmeans (clustering)
                      ).fit(df_taxi[['trip_duration']])

kbd.bin_edges_
```

Out[63]:
```
array([array([2.000e+00, 4.780e+02, 8.700e+02, 3.556e+03])], dtype=object)
```

In [64]:
```python
df_taxi[['trip_duration']].head(3)
```

Out[64]:

	trip_duration
1	516
2	683
7	834

Bin and One-Hot Encode with sklearn

In [63]:
```
from sklearn.preprocessing import KBinsDiscretizer

# NOTE: We're not setting the bin edges explicitly
#       For control over bin edges, use Binarizer
kbd = KBinsDiscretizer(n_bins=3,
                       encode="onehot",      # or onehot (sparse), ordinal
                       strategy="quantile", # or uniform or kmeans (clustering)
                      ).fit(df_taxi[['trip_duration']])

kbd.bin_edges_
```

Out[63]:
```
array([array([2.000e+00, 4.780e+02, 8.700e+02, 3.556e+03])], dtype=object)
```

In [64]:
```
df_taxi[['trip_duration']].head(3)
```

Out[64]:

	trip_duration
1	516
2	683
7	834

In [65]:
```
kbd.transform(df_taxi[['trip_duration']])[:3]
```

Out[65]:
```
<3x3 sparse matrix of type '<class 'numpy.float64'>'
	with 3 stored elements in Compressed Sparse Row format>
```

Bin and One-Hot Encode with sklearn

In [63]:
```
from sklearn.preprocessing import KBinsDiscretizer

# NOTE: We're not setting the bin edges explicitly
#       For control over bin edges, use Binarizer
kbd = KBinsDiscretizer(n_bins=3,
                       encode="onehot",     # or onehot (sparse), ordinal
                       strategy="quantile", # or uniform or kmeans (clustering)
                      ).fit(df_taxi[['trip_duration']])

kbd.bin_edges_
```

Out[63]:
```
array([array([2.000e+00, 4.780e+02, 8.700e+02, 3.556e+03])], dtype=object)
```

In [64]:
```
df_taxi[['trip_duration']].head(3)
```

Out[64]:

	trip_duration
1	516
2	683
7	834

In [65]:
```
kbd.transform(df_taxi[['trip_duration']])[:3]
```

Out[65]:
```
<3x3 sparse matrix of type '<class 'numpy.float64'>'
        with 3 stored elements in Compressed Sparse Row format>
```

In [66]:
```
kbd.transform(df_taxi[['trip_duration']])[:3].todense()
```

Out[66]:
```
matrix([[0., 1., 0.],
        [0., 1., 0.],
        [0., 1., 0.]])
```

Dealing with Ordinal Variables

Dealing with Ordinal Variables

In [67]:
```
df_pml = pd.DataFrame([['green','M',10.1,'class2'],
                       ['red','L',13.5,'class1'],
                       ['blue','XL',15.3,'class2']],
                      columns=['color','size','price','classlabel'])
df_pml
```

Out[67]:

	color	size	price	classlabel
0	green	M	10.1	class2
1	red	L	13.5	class1
2	blue	XL	15.3	class2

Dealing with Ordinal Variables

In [67]:

```
df_pml = pd.DataFrame([['green','M',10.1,'class2'],
                       ['red','L',13.5,'class1'],
                       ['blue','XL',15.3,'class2']],
                      columns=['color','size','price','classlabel'])
df_pml
```

Out[67]:

	color	size	price	classlabel
0	green	M	10.1	class2
1	red	L	13.5	class1
2	blue	XL	15.3	class2

In [68]:

```
# if we know the numerical difference between ordinal values
# eg XL = L+1 = M+2

size_mapping = {'XL':3,
                'L':2,
                'M':1}

df_pml_features = pd.DataFrame()

df_pml_features['size'] = df_pml['size'].map(size_mapping)
df_pml_features
```

Out[68]:

	size
0	1
1	2
2	3

Dealing with Ordinal Variables Cont.

Dealing with Ordinal Variables Cont.

In [69]:
```
df_pml
```

Out[69]:

	color	size	price	classlabel
0	green	M	10.1	class2
1	red	L	13.5	class1
2	blue	XL	15.3	class2

Dealing with Ordinal Variables Cont.

In [69]:
```
df_pml
```

Out[69]:

	color	size	price	classlabel
0	green	M	10.1	class2
1	red	L	13.5	class1
2	blue	XL	15.3	class2

In [70]:
```
# if we don't know the numerical difference between ordinal values
# generate threshold features
df_pml_features = pd.DataFrame()
df_pml_features['x > M'] = df_pml['size'].apply(lambda x: 1 if x in {'L','XL'} else 0)
df_pml_features['x > L'] = df_pml['size'].apply(lambda x: 1 if x == 'XL' else 0)
df_pml_features
```

Out[70]:

	x > M	x > L
0	0	0
1	1	0
2	1	1

Derived Features

- Anything that is a transformation of our data
- This is where the money is!
- Examples:
  - "is a high demand pickup location"
  - "is a problem house sale"
  - "high-performing job candidate"

Polynomial Features

Polynomial Features

In [71]:
```
from sklearn.preprocessing import PolynomialFeatures

pf = PolynomialFeatures(degree=2,
                        include_bias=False)
X_new = pf.fit_transform(df_taxi[['passenger_count','trip_duration']])

new_columns = ['passenger_count','trip_duration','passenger_count^2','passenger_count*trip_duration','trip_duration^2']
pd.DataFrame(X_new[3:5],columns=new_columns)
```

Out[71]:

	passenger_count	trip_duration	passenger_count^2	passenger_count*trip_duration	trip_duration^2
0	3.0	298.0	9.0	894.0	88804.0
1	1.0	396.0	1.0	396.0	156816.0

Python String Functions

Python String Functions

In [72]:
```python
doc = "D.S. is fun!"
doc
```

Out[72]:
```
'D.S. is fun!'
```

Python String Functions

In [72]:
```
doc = "D.S. is fun!"
doc
```

Out[72]:
```
'D.S. is fun!'
```

In [73]:
```
doc.lower(),doc.upper()       # change capitalization
```

Out[73]:
```
('d.s. is fun!', 'D.S. IS FUN!')
```

Python String Functions

In [72]:
```
doc = "D.S. is fun!"
doc
```

Out[72]: 'D.S. is fun!'

In [73]:
```
doc.lower(),doc.upper()       # change capitalization
```

Out[73]: ('d.s. is fun!', 'D.S. IS FUN!')

In [74]:
```
doc.split() , doc.split('.')  # split a string into parts (default is whitespace)
```

Out[74]: (['D.S.', 'is', 'fun!'], ['D', 'S', ' is fun!'])

Python String Functions

In [72]:
```
doc = "D.S. is fun!"
doc
```

Out[72]:
```
'D.S. is fun!'
```

In [73]:
```
doc.lower(),doc.upper()        # change capitalization
```

Out[73]:
```
('d.s. is fun!', 'D.S. IS FUN!')
```

In [74]:
```
doc.split() , doc.split('.')  # split a string into parts (default is whitespace)
```

Out[74]:
```
(['D.S.', 'is', 'fun!'], ['D', 'S', ' is fun!'])
```

In [75]:
```
'|'.join(['ab','c','d'])      # join items in a list together
```

Out[75]:
```
'ab|c|d'
```

Python String Functions

In [72]:
```
doc = "D.S. is fun!"
doc
```

Out[72]: 'D.S. is fun!'

In [73]:
```
doc.lower(),doc.upper()       # change capitalization
```

Out[73]: ('d.s. is fun!', 'D.S. IS FUN!')

In [74]:
```
doc.split() , doc.split('.')  # split a string into parts (default is whitespace)
```

Out[74]: (['D.S.', 'is', 'fun!'], ['D', 'S', ' is fun!'])

In [75]:
```
'|'.join(['ab','c','d'])      # join items in a list together
```

Out[75]: 'ab|c|d'

In [76]:
```
'|'.join(doc[:5])             # a string itself is treated like a list of characters
```

Out[76]: 'D|.|S|.| '

Python String Functions

```
In [72]: doc = "D.S. is fun!"
         doc

Out[72]: 'D.S. is fun!'

In [73]: doc.lower(),doc.upper()       # change capitalization

Out[73]: ('d.s. is fun!', 'D.S. IS FUN!')

In [74]: doc.split() , doc.split('.')  # split a string into parts (default is whitespace)

Out[74]: (['D.S.', 'is', 'fun!'], ['D', 'S', ' is fun!'])

In [75]: '|'.join(['ab','c','d'])      # join items in a list together

Out[75]: 'ab|c|d'

In [76]: '|'.join(doc[:5])             # a string itself is treated like a list of characters

Out[76]: 'D|.|S|.| '

In [77]: '  test   '.strip()           # remove whitespace from the beginning and end of a string

Out[77]: 'test'
```

Python String Functions

```
In [72]: doc = "D.S. is fun!"
         doc

Out[72]: 'D.S. is fun!'

In [73]: doc.lower(),doc.upper()       # change capitalization

Out[73]: ('d.s. is fun!', 'D.S. IS FUN!')

In [74]: doc.split() , doc.split('.')  # split a string into parts (default is whitespace)

Out[74]: (['D.S.', 'is', 'fun!'], ['D', 'S', ' is fun!'])

In [75]: '|'.join(['ab','c','d'])      # join items in a list together

Out[75]: 'ab|c|d'

In [76]: '|'.join(doc[:5])             # a string itself is treated like a list of characters

Out[76]: 'D|.|S|.| '

In [77]: '  test   '.strip()           # remove whitespace from the beginning and end of a string

Out[77]: 'test'
```

and more, see https://docs.python.org/3.8/library/string.html

String Functions in Pandas

String Functions in Pandas

In [78]:
```
df_shop.iloc[:2].loc[:,'lastname']
```

Out[78]:
```
0     PERKINS
1    ROBINSON
Name: lastname, dtype: object
```

String Functions in Pandas

In [78]:
```
df_shop.iloc[:2].loc[:,'lastname']
```

Out[78]:
```
0     PERKINS
1    ROBINSON
Name: lastname, dtype: object
```

In [79]:
```
df_shop.loc[:,'lastname'].iloc[:2].str.lower()
```

Out[79]:
```
0     perkins
1    robinson
Name: lastname, dtype: object
```

String Functions in Pandas

In [78]:
```
df_shop.iloc[:2].loc[:,'lastname']
```
Out[78]:
```
0     PERKINS
1    ROBINSON
Name: lastname, dtype: object
```

In [79]:
```
df_shop.loc[:,'lastname'].iloc[:2].str.lower()
```
Out[79]:
```
0     perkins
1    robinson
Name: lastname, dtype: object
```

In [80]:
```
df_shop.lastname[:2].str.capitalize()
```
Out[80]:
```
0     Perkins
1    Robinson
Name: lastname, dtype: object
```

String Functions in Pandas

In [78]:
```
df_shop.iloc[:2].loc[:,'lastname']
```

Out[78]:
```
0     PERKINS
1    ROBINSON
Name: lastname, dtype: object
```

In [79]:
```
df_shop.loc[:,'lastname'].iloc[:2].str.lower()
```

Out[79]:
```
0     perkins
1    robinson
Name: lastname, dtype: object
```

In [80]:
```
df_shop.lastname[:2].str.capitalize()
```

Out[80]:
```
0     Perkins
1    Robinson
Name: lastname, dtype: object
```

In [81]:
```
df_shop.lastname[:2].str.startswith('ROB') # .endswith() , .contains()
```

Out[81]:
```
0    False
1     True
Name: lastname, dtype: bool
```

String Functions in Pandas

In [78]:
```
df_shop.iloc[:2].loc[:,'lastname']
```

Out[78]:
```
0     PERKINS
1    ROBINSON
Name: lastname, dtype: object
```

In [79]:
```
df_shop.loc[:,'lastname'].iloc[:2].str.lower()
```

Out[79]:
```
0     perkins
1    robinson
Name: lastname, dtype: object
```

In [80]:
```
df_shop.lastname[:2].str.capitalize()
```

Out[80]:
```
0     Perkins
1    Robinson
Name: lastname, dtype: object
```

In [81]:
```
df_shop.lastname[:2].str.startswith('ROB') # .endswith() , .contains()
```

Out[81]:
```
0    False
1     True
Name: lastname, dtype: bool
```

In [82]:
```
df_shop.lastname[:2].str.replace('R','^')
```

Out[82]:
```
0     PE^KINS
1    ^OBINSON
Name: lastname, dtype: object
```

String Functions in Pandas

In [78]:
```
df_shop.iloc[:2].loc[:,'lastname']
```

Out[78]:
```
0     PERKINS
1    ROBINSON
Name: lastname, dtype: object
```

In [79]:
```
df_shop.loc[:,'lastname'].iloc[:2].str.lower()
```

Out[79]:
```
0     perkins
1    robinson
Name: lastname, dtype: object
```

In [80]:
```
df_shop.lastname[:2].str.capitalize()
```

Out[80]:
```
0     Perkins
1    Robinson
Name: lastname, dtype: object
```

In [81]:
```
df_shop.lastname[:2].str.startswith('ROB') # .endswith() , .contains()
```

Out[81]:
```
0    False
1     True
Name: lastname, dtype: bool
```

In [82]:
```
df_shop.lastname[:2].str.replace('R','^')
```

Out[82]:
```
0     PE^KINS
1    ^OBINSON
Name: lastname, dtype: object
```

Pandas datetime functions

Pandas datetime functions

```
In [83]: df_taxi.iloc[:2].tpep_pickup_datetime
```

```
Out[83]: 1   2017-01-05 15:14:52
         2   2017-01-11 14:47:52
         Name: tpep_pickup_datetime, dtype: datetime64[ns]
```

Pandas datetime functions

In [83]:
```
df_taxi.iloc[:2].tpep_pickup_datetime
```

Out[83]:
```
1   2017-01-05 15:14:52
2   2017-01-11 14:47:52
Name: tpep_pickup_datetime, dtype: datetime64[ns]
```

In [84]:
```
df_taxi.iloc[:2].tpep_pickup_datetime.dt.day
```

Out[84]:
```
1     5
2    11
Name: tpep_pickup_datetime, dtype: int64
```

Pandas datetime functions

In [83]:
```
df_taxi.iloc[:2].tpep_pickup_datetime
```

Out[83]:
```
1   2017-01-05 15:14:52
2   2017-01-11 14:47:52
Name: tpep_pickup_datetime, dtype: datetime64[ns]
```

In [84]:
```
df_taxi.iloc[:2].tpep_pickup_datetime.dt.day
```

Out[84]:
```
1     5
2    11
Name: tpep_pickup_datetime, dtype: int64
```

In [85]:
```
df_taxi.iloc[:2].tpep_pickup_datetime.dt.day_of_week
```

Out[85]:
```
1    3
2    2
Name: tpep_pickup_datetime, dtype: int64
```

Pandas datetime functions

In [83]:
```
df_taxi.iloc[:2].tpep_pickup_datetime
```

Out[83]:
```
1   2017-01-05 15:14:52
2   2017-01-11 14:47:52
Name: tpep_pickup_datetime, dtype: datetime64[ns]
```

In [84]:
```
df_taxi.iloc[:2].tpep_pickup_datetime.dt.day
```

Out[84]:
```
1     5
2    11
Name: tpep_pickup_datetime, dtype: int64
```

In [85]:
```
df_taxi.iloc[:2].tpep_pickup_datetime.dt.day_of_week
```

Out[85]:
```
1    3
2    2
Name: tpep_pickup_datetime, dtype: int64
```

In [86]:
```
df_taxi.iloc[:2].tpep_pickup_datetime.dt.isocalendar().week
```

Out[86]:
```
1    1
2    2
Name: week, dtype: UInt32
```

Pandas datetime functions

In [83]:
```
df_taxi.iloc[:2].tpep_pickup_datetime
```

Out[83]:
```
1   2017-01-05 15:14:52
2   2017-01-11 14:47:52
Name: tpep_pickup_datetime, dtype: datetime64[ns]
```

In [84]:
```
df_taxi.iloc[:2].tpep_pickup_datetime.dt.day
```

Out[84]:
```
1     5
2    11
Name: tpep_pickup_datetime, dtype: int64
```

In [85]:
```
df_taxi.iloc[:2].tpep_pickup_datetime.dt.day_of_week
```

Out[85]:
```
1    3
2    2
Name: tpep_pickup_datetime, dtype: int64
```

In [86]:
```
df_taxi.iloc[:2].tpep_pickup_datetime.dt.isocalendar().week
```

Out[86]:
```
1    1
2    2
Name: week, dtype: UInt32
```

In [87]:
```
(df_taxi.tpep_dropoff_datetime - df_taxi.tpep_pickup_datetime).dt.seconds.iloc[:2]
```

Out[87]:
```
1    516
2    683
dtype: int64
```

Pandas datetime functions

In [83]:
```
df_taxi.iloc[:2].tpep_pickup_datetime
```

Out[83]:
```
1   2017-01-05 15:14:52
2   2017-01-11 14:47:52
Name: tpep_pickup_datetime, dtype: datetime64[ns]
```

In [84]:
```
df_taxi.iloc[:2].tpep_pickup_datetime.dt.day
```

Out[84]:
```
1     5
2    11
Name: tpep_pickup_datetime, dtype: int64
```

In [85]:
```
df_taxi.iloc[:2].tpep_pickup_datetime.dt.day_of_week
```

Out[85]:
```
1    3
2    2
Name: tpep_pickup_datetime, dtype: int64
```

In [86]:
```
df_taxi.iloc[:2].tpep_pickup_datetime.dt.isocalendar().week
```

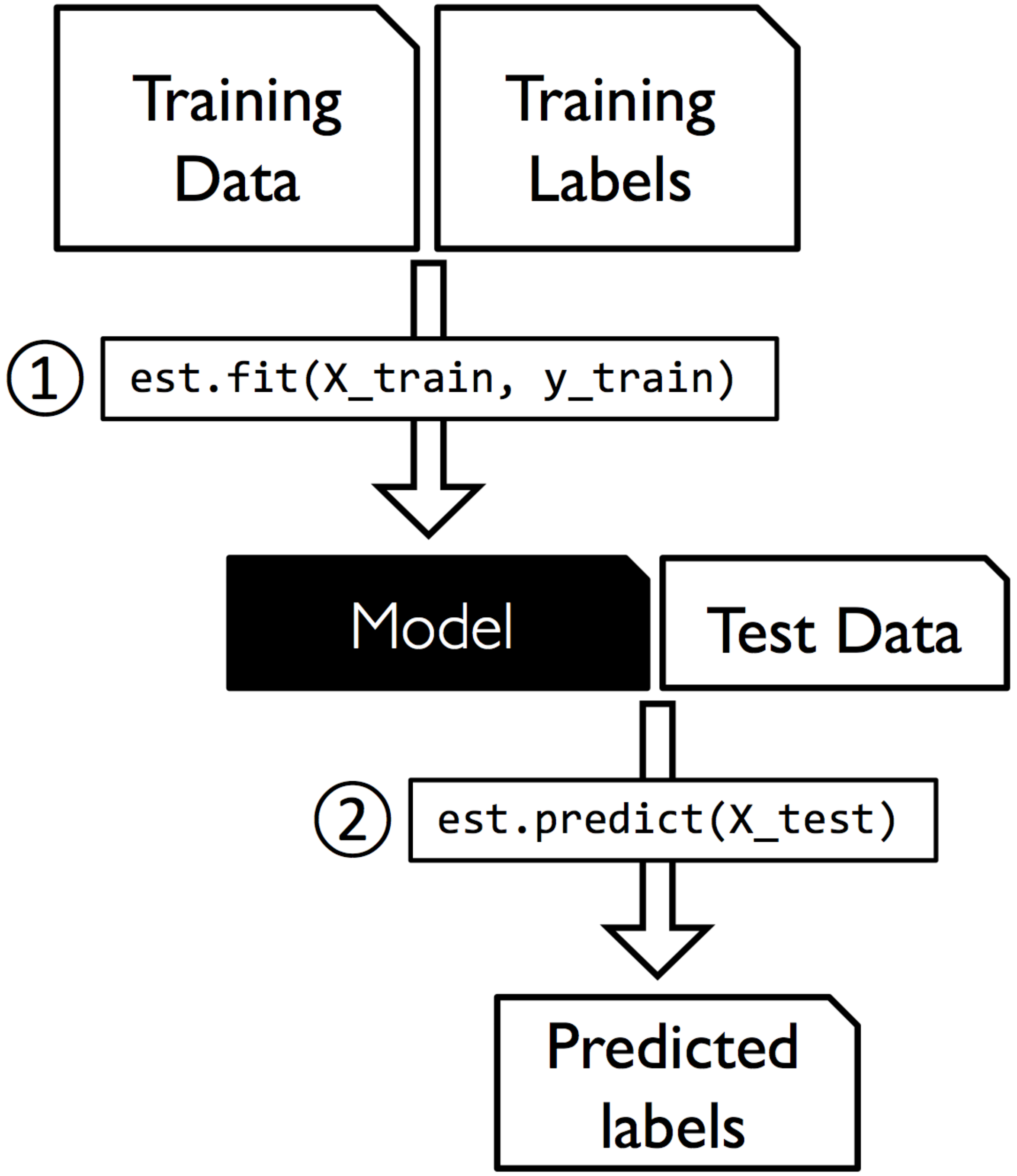
Out[86]:
```
1    1
2    2
Name: week, dtype: UInt32
```

In [87]:
```
(df_taxi.tpep_dropoff_datetime - df_taxi.tpep_pickup_datetime).dt.seconds.iloc[:2]
```

Out[87]:
```
1    516
2    683
dtype: int64
```

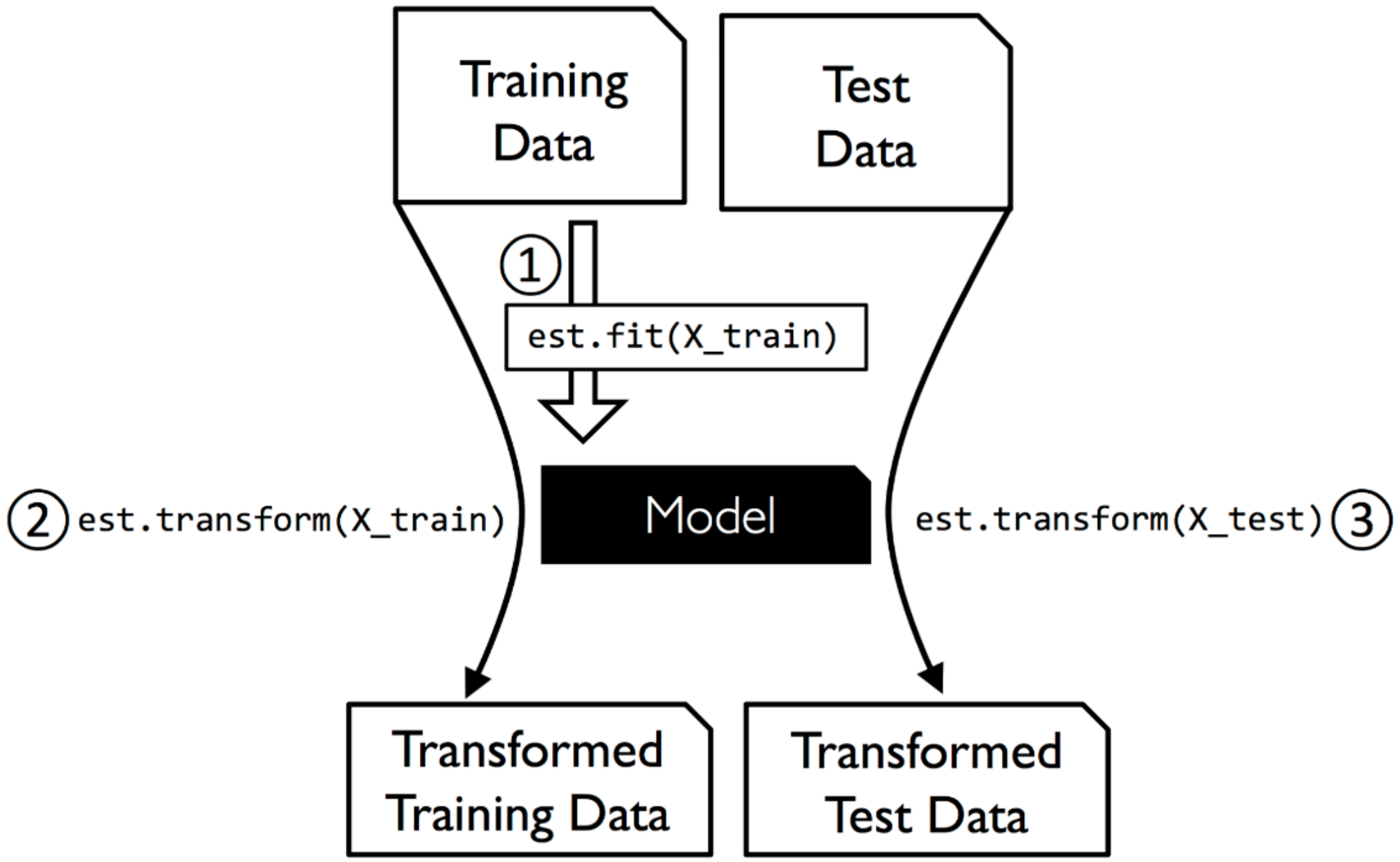
Predicting with Train/Test Split

- When training a model for prediction



Transforming with Train/Test Split

- When performing data transformation



Next

- Dimensionality Reduction
    - Feature Selection
    - Feature Extraction

Questions?