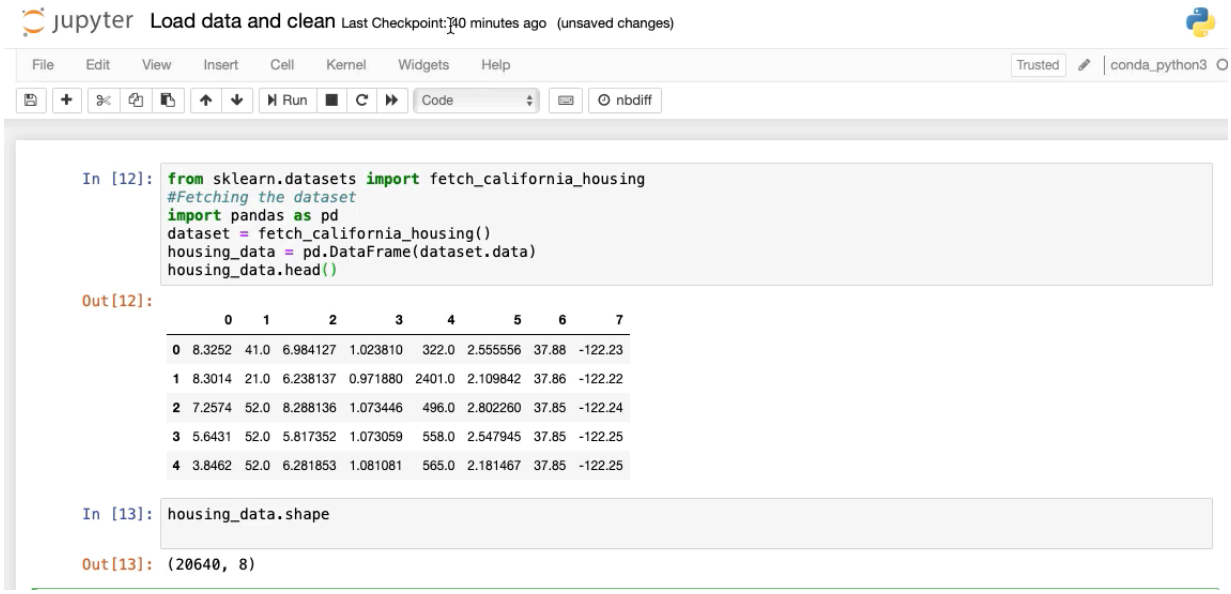
Whizlabs - ML Specialty Exam Course - Data Engineering - part 1

https://www.whizlabs.com/learn/course/aws-mls-practice-tests

1. **DATA ENGINEERING**

Loading data from scikit learn data repository in Jupiter



8 features, 20,640 features

AWS Machine Learning - Data Engineering

Handle Missing Data

- Null value replacement - Several approaches to the problem of handling missing data
  - Do nothing
  - Remove the entire record
  - Mode/median/average value replacement
  - Most frequent value
  - Model-based imputation
    - K-Nearest Neighbors
    - Regression
    - Deep Learning
  - Interpolation / Extrapolation
  - Forward filling / Backward filling
  - Hot deck imputation

Case	Attributes			Decision
	Temperature	Headache	Nausea	Flu
1	high	?	yes	yes
2	very_high	yes	no	yes
3	?	no	no	no
4	normal	yes	?	no
5	?	yes	yes	yes

XGboost can impute missing values

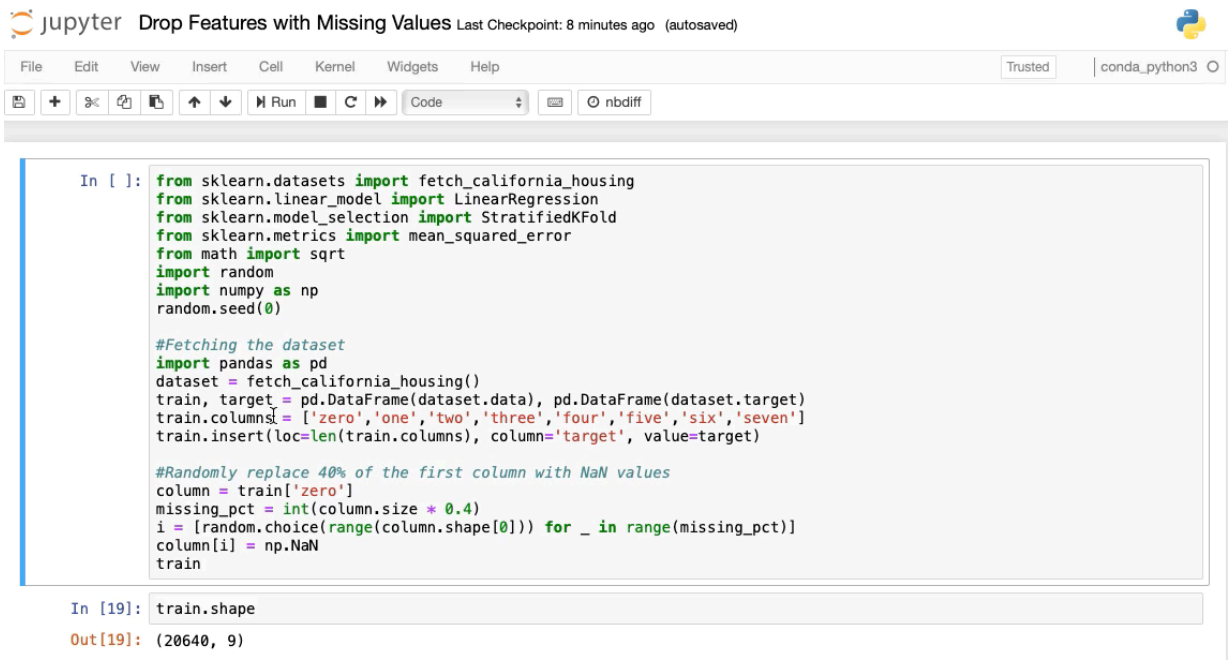
AWS Machine Learning - Handling Missing Data

Null Value Replacement - Which Method Should You Use

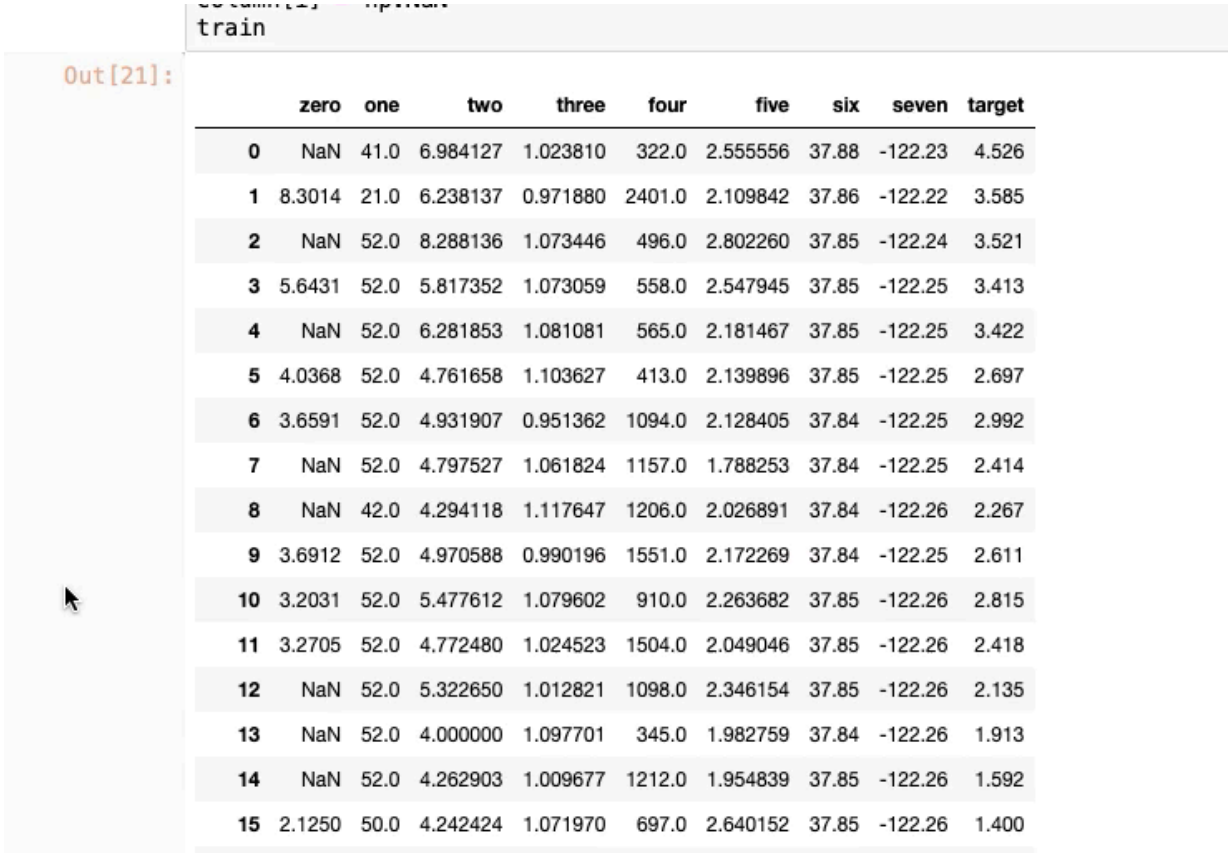
- Do nothing and let your algorithm either replace them through imputation (XGBoost) or just ignore them as LightGBM does with its *use_missing=false* parameter
  - Some algorithms will throw an error if they find missing values (LinearRegression)
- Or, replace all missing values

Case	Attributes			Decision
	Temperature	Headache	Nausea	Flu
1	high	?	yes	yes
2	very_high	yes	no	yes
3	?	no	no	no
4	normal	yes	?	no
5	?	yes	yes	yes

Using Pandas to REMOVE ROWS or Observations that don’t have Values:



40% Randomly of 1st feature now had NaN value



```
train
```

Out[21]:

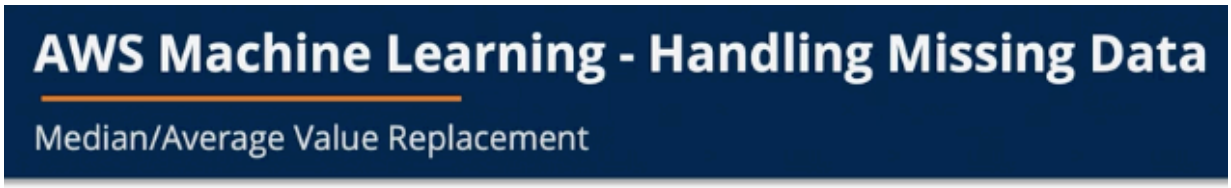
	zero	one	two	three	four	five	six	seven	target
0	NaN	41.0	6.984127	1.023810	322.0	2.555556	37.88	-122.23	4.526
1	8.3014	21.0	6.238137	0.971880	2401.0	2.109842	37.86	-122.22	3.585
2	NaN	52.0	8.288136	1.073446	496.0	2.802260	37.85	-122.24	3.521
3	5.6431	52.0	5.817352	1.073059	558.0	2.547945	37.85	-122.25	3.413
4	NaN	52.0	6.281853	1.081081	565.0	2.181467	37.85	-122.25	3.422
5	4.0368	52.0	4.761658	1.103627	413.0	2.139896	37.85	-122.25	2.697
6	3.6591	52.0	4.931907	0.951362	1094.0	2.128405	37.84	-122.25	2.992
7	NaN	52.0	4.797527	1.061824	1157.0	1.788253	37.84	-122.25	2.414
8	NaN	42.0	4.294118	1.117647	1206.0	2.026891	37.84	-122.26	2.267
9	3.6912	52.0	4.970588	0.990196	1551.0	2.172269	37.84	-122.25	2.611
10	3.2031	52.0	5.477612	1.079602	910.0	2.263682	37.85	-122.26	2.815
11	3.2705	52.0	4.772480	1.024523	1504.0	2.049046	37.85	-122.26	2.418
12	NaN	52.0	5.322650	1.012821	1098.0	2.346154	37.85	-122.26	2.135
13	NaN	52.0	4.000000	1.097701	345.0	1.982759	37.84	-122.26	1.913
14	NaN	52.0	4.262903	1.009677	1212.0	1.954839	37.85	-122.26	1.592
15	2.1250	50.0	4.242424	1.071970	697.0	2.640152	37.85	-122.26	1.400

We want to drop the rows that have a NaN
=> use dropna() method

```
In [22]: train.shape

Out[22]: (20640, 9)

In [23]: ######################################################
         # Remove observations that have missing values
         # Will drop all rows that have any missing values.
         ######################################################
         train.dropna(inplace=True)
         train.shape

Out[23]: (13783, 9)
```

Now only have 13k feature from the 20k

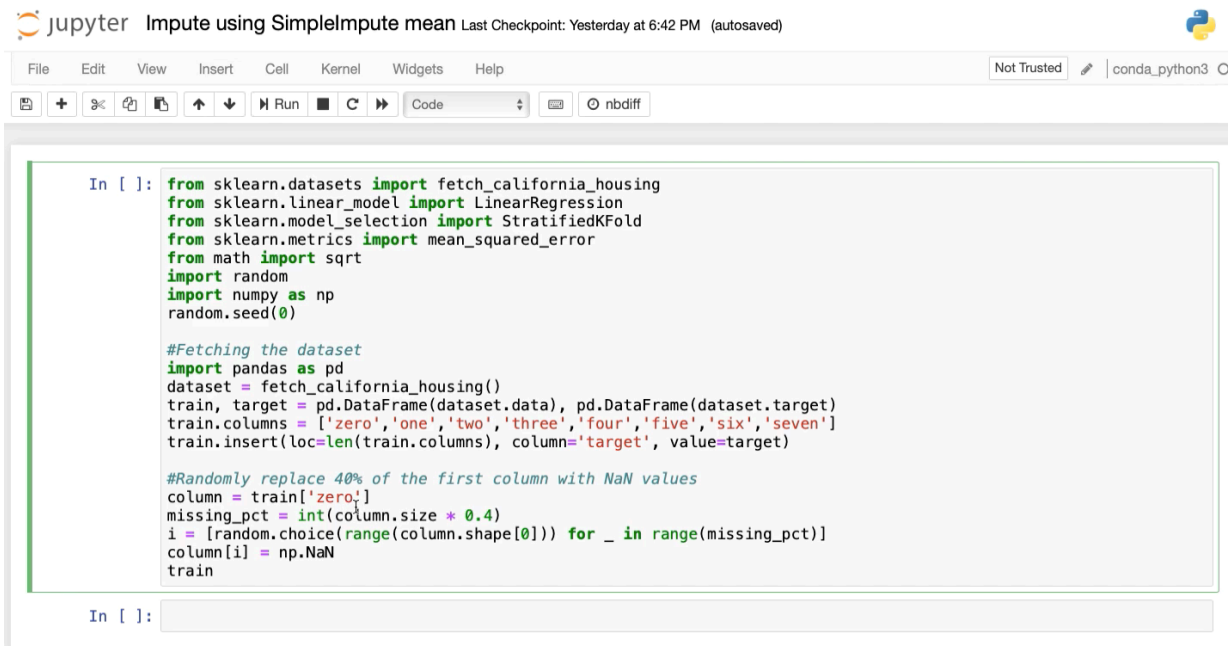
Other Imputations Techniques



- Replace the missing values with a simple median, or mean
  - Reflection of the other values in the feature
  - Doesn't factor correlation between features
  - Can't use on categorical features

Case	Attributes			Decision
	Temperature	Headache	Temperature	Flu
1	high	?	99.9	yes
2	very_high	yes	100.3	yes
3	?	no	98.6	no
4	normal	yes	?	no
5	?	yes	101.0	yes

Example via Code




```
train
```

Out[3]:

	zero	one	two	three	four	five	six	seven	target
0	NaN	41.0	6.984127	1.023810	322.0	2.555556	37.88	-122.23	4.526
1	8.3014	21.0	6.238137	0.971880	2401.0	2.109842	37.86	-122.22	3.585
2	NaN	52.0	8.288136	1.073446	496.0	2.802260	37.85	-122.24	3.521
3	5.6431	52.0	5.817352	1.073059	558.0	2.547945	37.85	-122.25	3.413
4	NaN	52.0	6.281853	1.081081	565.0	2.181467	37.85	-122.25	3.422
5	4.0368	52.0	4.761658	1.103627	413.0	2.139896	37.85	-122.25	2.697
6	3.6591	52.0	4.931907	0.951362	1094.0	2.128405	37.84	-122.25	2.992
7	NaN	52.0	4.797527	1.061824	1157.0	1.788253	37.84	-122.25	2.414
8	NaN	42.0	4.294118	1.117647	1206.0	2.026891	37.84	-122.26	2.267
9	3.6912	52.0	4.970588	0.990196	1551.0	2.172269	37.84	-122.25	2.611
10	3.2031	52.0	5.477612	1.079602	910.0	2.263682	37.85	-122.26	2.815
11	3.2705	52.0	4.772480	1.024523	1504.0	2.049046	37.85	-122.26	2.418

Now impute values with **SimpleImputer()** from Scikitlearn
Can call it with “mean”, “median”,...

ravel() method is to unravel into a vector that we can use to replace the train[zero] features

Now the missing values are replaced by the **mean**: 3.87

In [4]:
```
#Impute the values using scikit-learn SimpleImpute Class
from sklearn.impute import SimpleImputer
imputer = SimpleImputer(missing_values=np.nan, strategy='mean') #for options other than mean imputation replace
imputer = imputer.fit(train[['zero']])
train['zero'] = imputer.transform(train[['zero']]).ravel()
train
```

Out[4]:

	zero	one	two	three	four	five	six	seven	target
0	3.87794	41.0	6.984127	1.023810	322.0	2.555556	37.88	-122.23	4.526
1	8.30140	21.0	6.238137	0.971880	2401.0	2.109842	37.86	-122.22	3.585
2	3.87794	52.0	8.288136	1.073446	496.0	2.802260	37.85	-122.24	3.521
3	5.64310	52.0	5.817352	1.073059	558.0	2.547945	37.85	-122.25	3.413
4	3.87794	52.0	6.281853	1.081081	565.0	2.181467	37.85	-122.25	3.422
5	4.03680	52.0	4.761658	1.103627	413.0	2.139896	37.85	-122.25	2.697
6	3.65910	52.0	4.931907	0.951362	1094.0	2.128405	37.84	-122.25	2.992
7	3.87794	52.0	4.797527	1.061824	1157.0	1.788253	37.84	-122.25	2.414
8	3.87794	42.0	4.294118	1.117647	1206.0	2.026891	37.84	-122.26	2.267
9	3.69120	52.0	4.970588	0.990196	1551.0	2.172269	37.84	-122.25	2.611

Now using the **median** => 3.5497

In [6]:
```
#Impute the values using scikit-learn SimpleImpute Class
from sklearn.impute import SimpleImputer
imputer = SimpleImputer(missing_values=np.nan, strategy='median') #for options other th
imputer = imputer.fit(train[['zero']])
train['zero'] = imputer.transform(train[['zero']]).ravel()
train
```

Out[6]:

	zero	one	two	three	four	five	six	seven	target
0	3.5497	41.0	6.984127	1.023810	322.0	2.555556	37.88	-122.23	4.526
1	8.3014	21.0	6.238137	0.971880	2401.0	2.109842	37.86	-122.22	3.585
2	3.5497	52.0	8.288136	1.073446	496.0	2.802260	37.85	-122.24	3.521
3	5.6431	52.0	5.817352	1.073059	558.0	2.547945	37.85	-122.25	3.413
4	3.5497	52.0	6.281853	1.081081	565.0	2.181467	37.85	-122.25	3.422
5	4.0368	52.0	4.761658	1.103627	413.0	2.139896	37.85	-122.25	2.697
6	3.6591	52.0	4.931907	0.951362	1094.0	2.128405	37.84	-122.25	2.992
7	3.5497	52.0	4.797527	1.061824	1157.0	1.788253	37.84	-122.25	2.414
8	3.5497	42.0	4.294118	1.117647	1206.0	2.026891	37.84	-122.26	2.267
9	3.6912	52.0	4.970588	0.990196	1551.0	2.172269	37.84	-122.25	2.611
10	3.2031	52.0	5.477612	1.079602	910.0	2.263682	37.85	-122.26	2.815

Now using the **most_frequent** => 3.1250

In [10]:
```
#Impute the values using scikit-learn SimpleImpute Class
from sklearn.impute import SimpleImputer
imputer = SimpleImputer(missing_values=np.nan, strategy='most_frequent') #for options other than mean imputation
imputer = imputer.fit(train[['zero']])
train['zero'] = imputer.transform(train[['zero']]).ravel()
train
```

Out[10]:

	zero	one	two	three	four	five	six	seven	target
0	3.1250	41.0	6.984127	1.023810	322.0	2.555556	37.88	-122.23	4.526
1	8.3014	21.0	6.238137	0.971880	2401.0	2.109842	37.86	-122.22	3.585
2	3.1250	52.0	8.288136	1.073446	496.0	2.802260	37.85	-122.24	3.521
3	5.6431	52.0	5.817352	1.073059	558.0	2.547945	37.85	-122.25	3.413
4	3.1250	52.0	6.281853	1.081081	565.0	2.181467	37.85	-122.25	3.422
5	4.0368	52.0	4.761658	1.103627	413.0	2.139896	37.85	-122.25	2.697
6	3.6591	52.0	4.931907	0.951362	1094.0	2.128405	37.84	-122.25	2.992
7	3.1250	52.0	4.797527	1.061824	1157.0	1.788253	37.84	-122.25	2.414
8	3.1250	42.0	4.294118	1.117647	1206.0	2.026891	37.84	-122.26	2.267

AWS Machine Learning - Handling Missing Data

Most Frequent Value

- Replace missing values with the most frequently occurring value in the feature
  - Doesn't factor correlation between features
  - Works with categorical features
  - Can introduce bias into your model

Case	Attributes			Decision
	Temperature	Headache	Temperature	Flu
1	high	?	99.9	yes
2	very_high	yes	100.3	yes
3	?	no	98.6	no
4	normal	yes	?	no
5	?	yes	101.0	yes

Now using **Model Based** imputation

AWS Machine Learning - Handling Missing Data

Model-Based Imputation

- Use a machine learning algorithm to impute the missing values
  - K-Nearest Neighbors
    - Uses ‘feature similarity’ to predict missing values
  - Regression
    - Predictors of the variable with missing values identified via correlation matrix
    - Best predictors are selected and used as independent variables in a regression equation
    - Variable with missing data is used as the target variable
  - Deep Learning
    - Works very well with categorical and non-numerical features

Now let’s see model based imputation in code:
Same dataset to start with:

In [1]:
```
from sklearn.datasets import fetch_california_housing
from sklearn.linear_model import LinearRegression
from sklearn.model_selection import StratifiedKFold
from sklearn.metrics import mean_squared_error
from math import sqrt
import random
import numpy as np
random.seed(0)

#Fetching the dataset
import pandas as pd
dataset = fetch_california_housing()
train, target = pd.DataFrame(dataset.data), pd.DataFrame(dataset.target)
train.columns = ['zero','one','two','three','four','five','six','seven']
train.insert(loc=len(train.columns), column='target', value=target)

#Randomly replace 40% of the first column with NaN values
column = train['zero']
missing_pct = int(column.size * 0.4)
i = [random.choice(range(column.shape[0])) for _ in range(missing_pct)]
column[i] = np.NaN
train
```

Out[1]:

	zero	one	two	three	four	five	six	seven	target
0	NaN	41.0	6.984127	1.023810	322.0	2.555556	37.88	-122.23	4.526
1	8.3014	21.0	6.238137	0.971880	2401.0	2.109842	37.86	-122.22	3.585
2	NaN	52.0	8.288136	1.073446	496.0	2.802260	37.85	-122.24	3.521
3	5.6431	52.0	5.817352	1.073059	558.0	2.547945	37.85	-122.25	3.413
4	NaN	52.0	6.281853	1.081081	565.0	2.181467	37.85	-122.25	3.422
5	4.0368	52.0	4.761658	1.103627	413.0	2.139896	37.85	-122.25	2.697

Now we use the **KNN** algorithm to impute the missing values
In this case, using 2 neighbors

```
In [2]: #Impute the values using scikit-learn KNNImputer Class
        #Install the KNNImputer pip package in the current Jupyter kernel
        import sys
        !{sys.executable} -m pip install --upgrade pip
        !{sys.executable} -m pip install missingpy
        from missingpy import KNNImputer
        #Replace missing feature values using K-Nearest Neighbors
        imputer = KNNImputer(n_neighbors=2, weights="uniform")
        imputer.fit_transform(train[['zero']])
        train['zero'] = imputer.transform(train[['zero']]).ravel()
        train
```

5	4.03680	52.0	4.761658	1.103627	413.0	2.139896	37.85	-122.25	2.697
6	3.65910	52.0	4.931907	0.951362	1094.0	2.128405	37.84	-122.25	2.992
7	3.87794	52.0	4.797527	1.061824	1157.0	1.788253	37.84	-122.25	2.414
8	3.87794	42.0	4.294118	1.117647	1206.0	2.026891	37.84	-122.26	2.267
9	3.69120	52.0	4.970588	0.990196	1551.0	2.172269	37.84	-122.25	2.611
10	3.20310	52.0	5.477612	1.079602	910.0	2.263682	37.85	-122.26	2.815
11	3.27050	52.0	4.772480	1.024523	1504.0	2.049046	37.85	-122.26	2.418
12	3.87794	52.0	5.322650	1.012821	1098.0	2.346154	37.85	-122.26	2.135
13	3.87794	52.0	4.000000	1.097701	345.0	1.982759	37.84	-122.26	1.913
14	3.87794	52.0	4.262903	1.009677	1212.0	1.954839	37.85	-122.26	1.592

Notes: KNN only works with numerical values

Till now, we discussed missing value treatment using kNNImputer for continuous variables. Below, we create a data frame with missing values in categorical variables. For imputing missing values in categorical variables, we have to encode the categorical values into numeric values as kNNImputer works only for numeric variables. We can perform this using a mapping of categories to numeric variables.

Other methods for Imputation

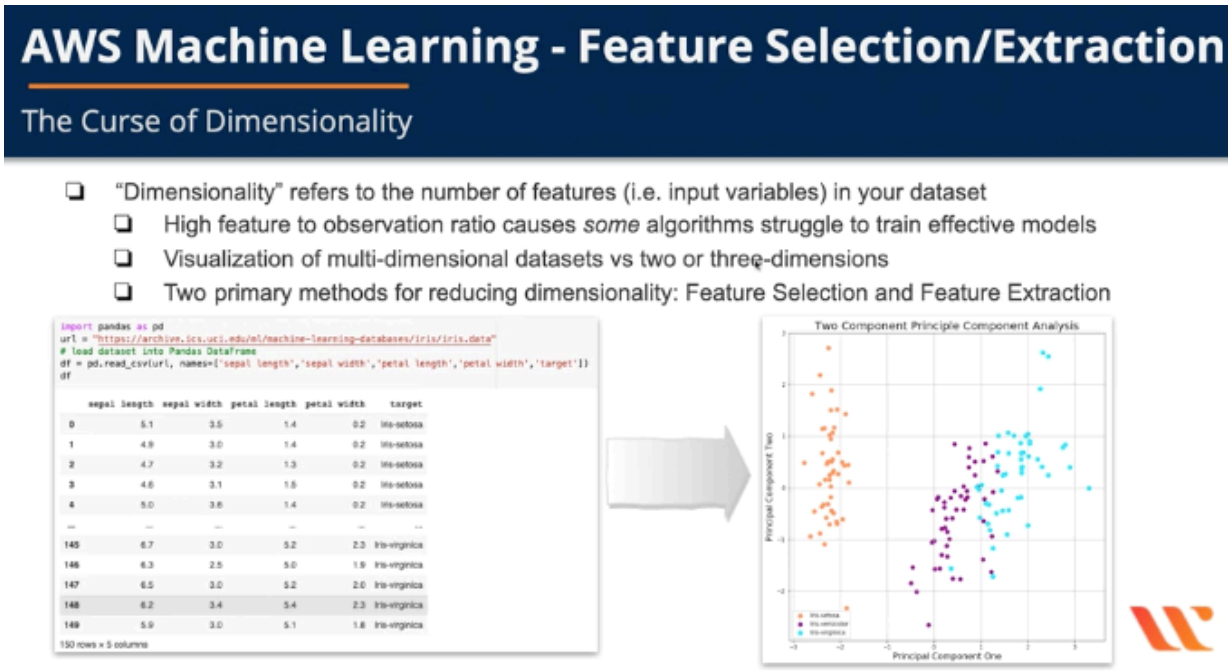
- ❏ Interpolation / Extrapolation
  - ❏ Estimate values from other observations within the range of a discrete set of known data points
- ❏ Forward filling / Backward filling
  - ❏ Fill the missing value by filling it from the preceding value or the succeeding value
- ❏ Hot deck imputation
  - ❏ Randomly choosing the missing value from a set of related and similar variables

Feature Extraction and feature selection

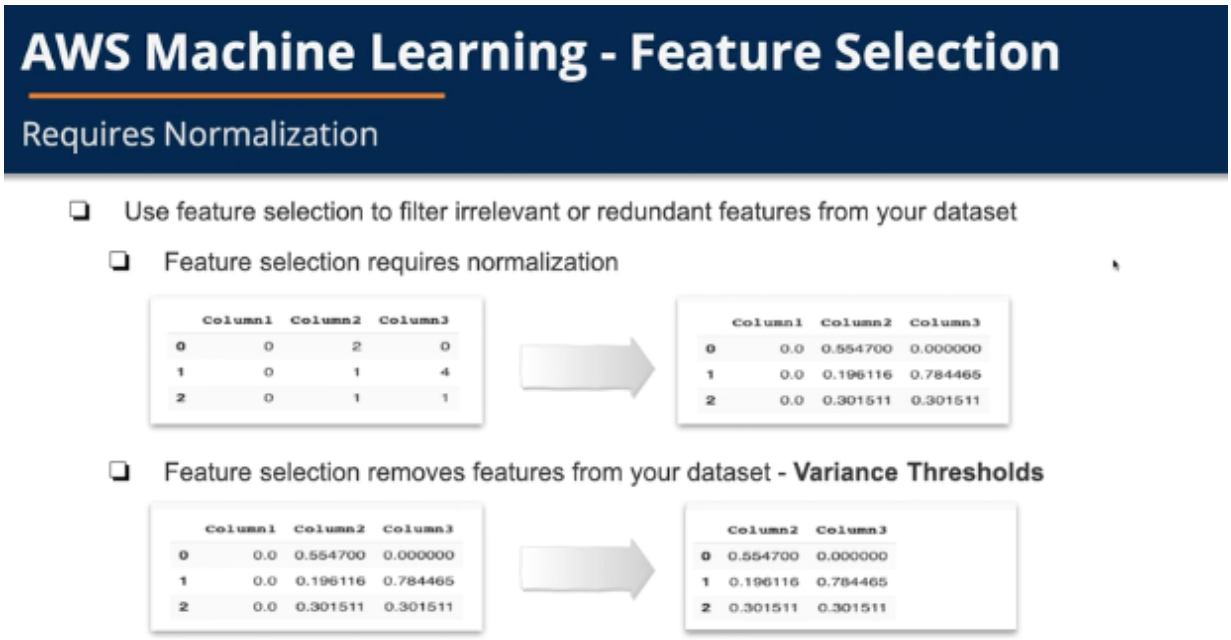
Visualizing multi-dimensions is pretty difficult - example: 4 features from Iris dataset below
There are 2 ways to reduce the features:

- feature **selection**
- feature **extraction**

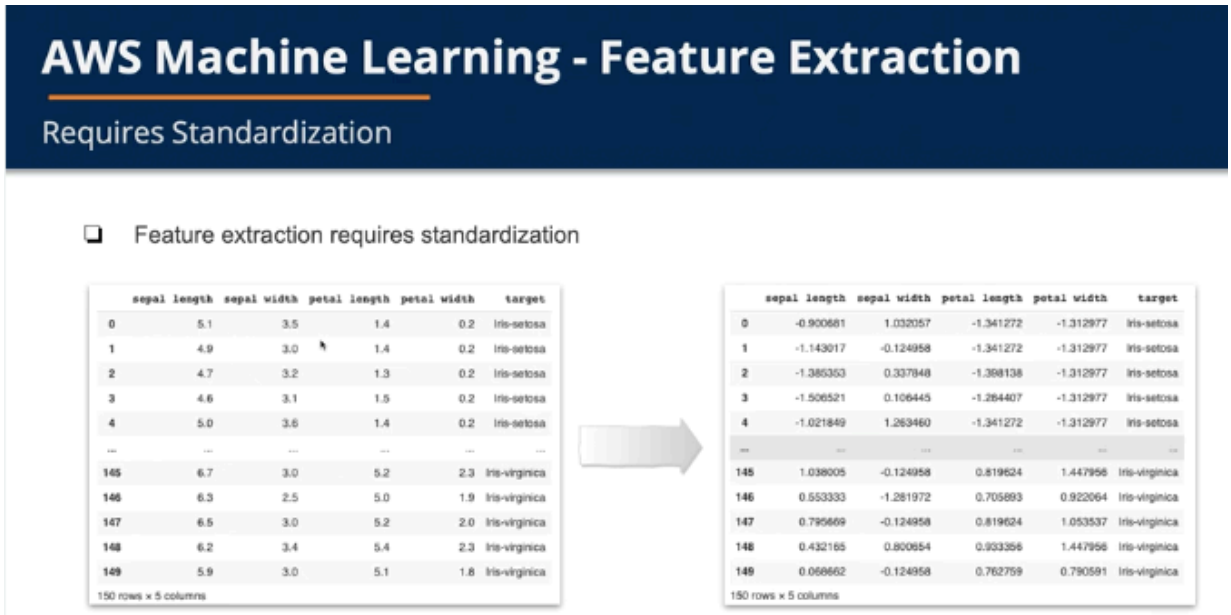
With feature extraction like PCA, we can extract features in a lower dimension space from 4 to 2 dimensions. Ex below



Feature selection requires **normalization**
allows us to remove features that add no value - it requires normalization
Ex with Variance Threshold below (and in the lab)



Feature extraction requires **standardization**
We standardize the data to a mean of 0 and std of 1 (See lab for more details)
It retains the information as best as it can



We now have 2 features, that are a combination of all 4 original values:

AWS Machine Learning - Feature Extraction

Reduces Features - Retains Information

- Creating new features from your existing features, feature extraction creates a new, smaller set of features that stills captures most of the useful information

	sepal length	sepal width	petal length	petal width	target
0	-0.900681	1.032057	-1.341272	-1.312977	Iris-setosa
1	-1.143017	-0.124958	-1.341272	-1.312977	Iris-setosa
2	-1.385353	0.337848	-1.398138	-1.312977	Iris-setosa
3	-1.506521	0.106445	-1.284407	-1.312977	Iris-setosa
4	-1.021849	1.263460	-1.341272	-1.312977	Iris-setosa
...	...	...	...	...	...
145	1.038005	-0.124958	0.819624	1.447956	Iris-virginica
146	0.553333	-1.281972	0.705893	0.922064	Iris-virginica
147	0.795669	-0.124958	0.819624	1.053537	Iris-virginica
148	0.432165	0.800654	0.933356	1.447956	Iris-virginica
149	0.068662	-0.124958	0.762759	0.790591	Iris-virginica

150 rows × 5 columns

	principal component 1	principal component 2	target
0	-2.264542	0.505704	Iris-setosa
1	-2.086426	-0.655405	Iris-setosa
2	-2.367950	-0.318477	Iris-setosa
3	-2.304197	-0.575368	Iris-setosa
4	-2.388777	0.674767	Iris-setosa
...	...	...	...
145	1.870522	0.382822	Iris-virginica
146	1.558492	-0.905314	Iris-virginica
147	1.520845	0.266795	Iris-virginica
148	1.376391	1.016362	Iris-virginica
149	0.959299	-0.022284	Iris-virginica

150 rows × 3 columns

AWS Machine Learning - Feature Extraction

Principal Component Analysis (PCA)

- **Principal component analysis** (PCA) is an unsupervised algorithm that creates new features by linearly combining original features
  - New features are uncorrelated, meaning they are orthogonal
  - New features are ranked in order of "explained variance." The first principal component (PC1) explains the most variance in your dataset, PC2 explains the second-most variance, etc.
    - Explained variance tells you how much information (variance) can be attributed to each of the principal components
    - You lose some of the variance (information) when you reduce your dimensional space

```
variance_ratio = pca.explained_variance_ratio_
variance_ratio

array([0.72770452, 0.23030523])
```

```
total_variance = variance_ratio[0] + variance_ratio[1]
total_variance

0.9580097536148197
```

Total variance: .958 (very close to 1)

PCA can be used in visualizing the data on a 2 dimensional scatter plot

AWS Machine Learning - Feature Extraction

Principal Component Analysis (PCA)

- Principal component analysis (PCA) can be used to assist in visualization of your data

PCA can also help in speeding up ML model

AWS Machine Learning - Feature Extraction

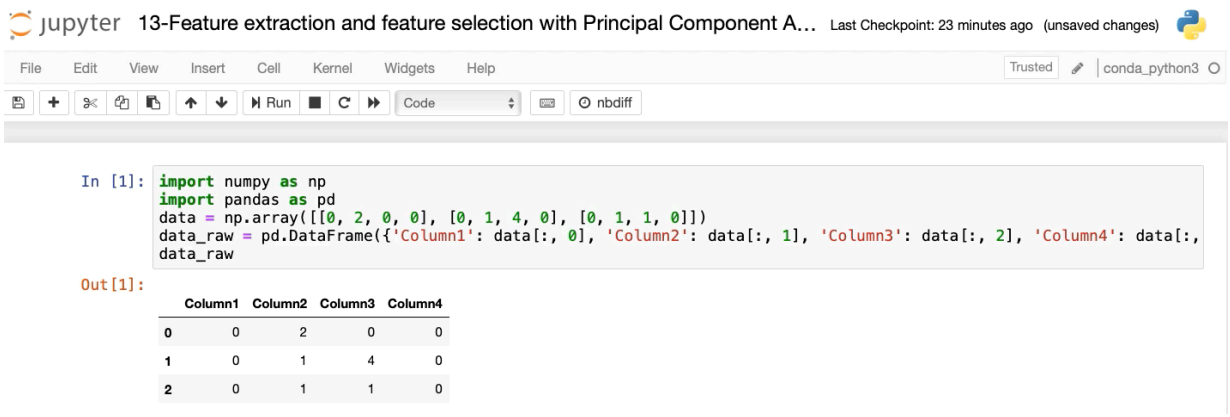
Principal Component Analysis (PCA)

- Principal component analysis (PCA) can also assist in speeding up your machine learning

Variance Retained	Number of Components	Time (seconds)	Accuracy
1.0	711	67.11	0.9155
0.99	541	54.90	0.9160
0.95	330	38.13	0.9200
0.90	236	29.19	0.9169
0.85	184	22.85	0.9154

Feature Extraction and Selection Lab

Feature Selection with VarianceThreshold():
Let’s start with a sample dataset



In [1]:
```
import numpy as np
import pandas as pd
data = np.array([[0, 2, 0, 0], [0, 1, 4, 0], [0, 1, 1, 0]])
data_raw = pd.DataFrame({'Column1': data[:, 0], 'Column2': data[:, 1], 'Column3': data[:, 2], 'Column4': data[:,
data_raw
```

Out[1]:

	Column1	Column2	Column3	Column4
0	0	2	0	0
1	0	1	4	0
2	0	1	1	0

Now let's use variance to remove features that are not relevant
First we need to **normalize** the data

In [2]:
```
from sklearn.feature_selection import VarianceThreshold
from sklearn import preprocessing
# normalize the data attributes
normalized_data = preprocessing.normalize(data)
dataset = pd.DataFrame({'Column1': normalized_data[:, 0], 'Column2': normalized_data[:, 1], 'Column3': normalize
dataset
```

Out[2]:

	Column1	Column2	Column3	Column4
0	0.0	1.000000	0.000000	0.0
1	0.0	0.242536	0.970143	0.0
2	0.0	0.707107	0.707107	0.0

We can now use the **VarianceThreshold()** to find the redundant or irrelevant features
We are now left with 2 features - 2 features were removed (col1 and col4)

In [3]:
```
selector = VarianceThreshold()
featureSelected = selector.fit_transform(normalized_data)
print(featureSelected)
dataset = pd.DataFrame({'Column2': normalized_data[:, 1], 'Column3': normalized_data[:, 2]})
dataset
```

```
[[1.         0.        ]
 [0.24253563 0.9701425 ]
 [0.70710678 0.70710678]]
```

Out[3]:

	Column2	Column3
0	1.000000	0.000000
1	0.242536	0.970143
2	0.707107	0.707107

Now using PCA for feature extraction

Loading Iris data set and mapping it to a dataframe

In [4]:
```
url = "https://archive.ics.uci.edu/ml/machine-learning-databases/iris/iris.data"
# load dataset into Pandas DataFrame
df = pd.read_csv(url, names=['sepal length','sepal width','petal length','petal width','target'])
df
```

Out[4]:

	sepal length	sepal width	petal length	petal width	target
0	5.1	3.5	1.4	0.2	Iris-setosa
1	4.9	3.0	1.4	0.2	Iris-setosa
2	4.7	3.2	1.3	0.2	Iris-setosa
3	4.6	3.1	1.5	0.2	Iris-setosa
4	5.0	3.6	1.4	0.2	Iris-setosa
5	5.4	3.9	1.7	0.4	Iris-setosa
6	4.6	3.4	1.4	0.3	Iris-setosa
7	5.0	3.4	1.5	0.2	Iris-setosa
8	4.4	2.9	1.4	0.2	Iris-setosa
9	4.9	3.1	1.5	0.1	Iris-setosa
10	5.4	3.7	1.5	0.2	Iris-setosa
145	6.7	3.0	5.2	2.3	Iris-virginica
146	6.3	2.5	5.0	1.9	Iris-virginica
147	6.5	3.0	5.2	2.0	Iris-virginica
148	6.2	3.4	5.4	2.3	Iris-virginica
149	5.9	3.0	5.1	1.8	Iris-virginica

150 rows × 5 columns

We have 150 observations and 4 features and 1 target value

To use PCA, we need **standardize the data with StandardScaler()**

In [ ]:
```
from sklearn.preprocessing import StandardScaler
features = ['sepal length', 'sepal width', 'petal length', 'petal width']
# Separating out the features
x = df.loc[:, features].values
# Separating out the target
y = df.loc[:,['target']].values
# Standardizing the features
x = StandardScaler().fit_transform(x)
xdf = pd.DataFrame({'sepal length': x[:, 0], 'sepal width': x[:, 1], 'petal length': x[:, 2], 'petal width': x[:
xdf
```

Features now have a standard deviation of 1 and a mean of 0

Out[5]:

	sepal length	sepal width	petal length	petal width	target
0	-0.900681	1.032057	-1.341272	-1.312977	Iris-setosa
1	-1.143017	-0.124958	-1.341272	-1.312977	Iris-setosa
2	-1.385353	0.337848	-1.398138	-1.312977	Iris-setosa
3	-1.506521	0.106445	-1.284407	-1.312977	Iris-setosa
4	-1.021849	1.263460	-1.341272	-1.312977	Iris-setosa
5	-0.537178	1.957669	-1.170675	-1.050031	Iris-setosa
6	-1.506521	0.800654	-1.341272	-1.181504	Iris-setosa
7	-1.021849	0.800654	-1.284407	-1.312977	Iris-setosa
8	-1.748856	-0.356361	-1.341272	-1.312977	Iris-setosa

144	1.038005	0.569251	1.103953	1.710902	Iris-virginica
145	1.038005	-0.124958	0.819624	1.447956	Iris-virginica
146	0.553333	-1.281972	0.705893	0.922064	Iris-virginica
147	0.795669	-0.124958	0.819624	1.053537	Iris-virginica
148	0.432165	0.800654	0.933356	1.447956	Iris-virginica
149	0.068662	-0.124958	0.762759	0.790591	Iris-virginica

150 rows × 5 columns

Now that features were standardized, we can use **PCA - Principal Component Analysis**
We are asking PCA to take our 4 dimensions and reduce it to 2 with **n_components=2**

In [6]:
```
from sklearn.decomposition import PCA
pca = PCA(n_components=2)
principalComponents = pca.fit_transform(x)
principalDf = pd.DataFrame(data = principalComponents, columns = ['principal component 1', 'principal component 2
principalDf
```

Out[6]:

	principal component 1	principal component 2
0	-2.264542	0.505704
1	-2.086426	-0.655405
2	-2.367950	-0.318477
3	-2.304197	-0.575368
4	-2.388777	0.674767
5	-2.070537	1.518549
6	-2.445711	0.074563

141	1.903117	0.686025
142	1.153190	-0.701326
143	2.043308	0.864685
144	2.001691	1.048550
145	1.870522	0.382822
146	1.558492	-0.905314
147	1.520845	0.266795
148	1.376391	1.016362
149	0.959299	-0.022284

150 rows × 2 columns

Now let’s concatenate the target value to our 2 features

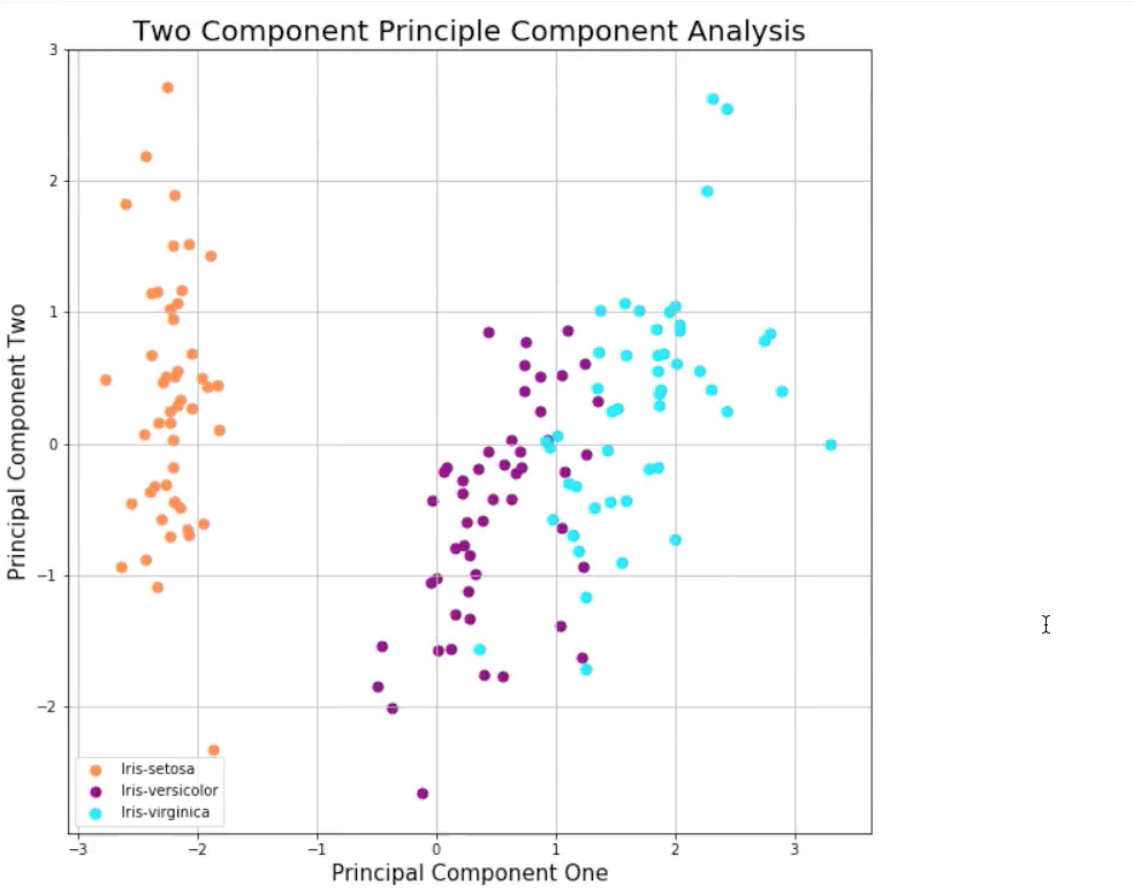
In [7]:
```
finalDf = pd.concat([principalDf, df[['target']]], axis = 1)
finalDf
```

Out[7]:

	principal component 1	principal component 2	target
0	-2.264542	0.505704	Iris-setosa
1	-2.086426	-0.655405	Iris-setosa
2	-2.367950	-0.318477	Iris-setosa
3	-2.304197	-0.575368	Iris-setosa
4	-2.388777	0.674767	Iris-setosa
5	-2.070537	1.518549	Iris-setosa
6	-2.445711	0.074563	Iris-setosa
7	-2.233842	0.247614	Iris-setosa
8	-2.341958	-1.095146	Iris-setosa

Now let's visualize this data (2 dimensions and 1 target) as a **scatter plot**
X-axis => component1
Y-axis => component 2
Color => Target

In [9]:
```
import matplotlib.pyplot as plt
fig = plt.figure(figsize = (10,10))
ax = fig.add_subplot(1,1,1)
ax.set_xlabel('Principal Component One', fontsize = 15)
ax.set_ylabel('Principal Component Two', fontsize = 15)
ax.set_title('Two Component Principle Component Analysis', fontsize = 20)
targets = ['Iris-setosa', 'Iris-versicolor', 'Iris-virginica']
colors = ['coral', 'purple', 'aqua']
for target, color in zip(targets,colors):
    indicesToKeep = finalDf['target'] == target
    ax.scatter(finalDf.loc[indicesToKeep, 'principal component 1']
               , finalDf.loc[indicesToKeep, 'principal component 2']
               , c = color
               , s = 50)
ax.legend(targets)
ax.grid()
```



We can see some separation
We can now use various training algorithms to help train on our data.

Let's see how much of the variance we have retained

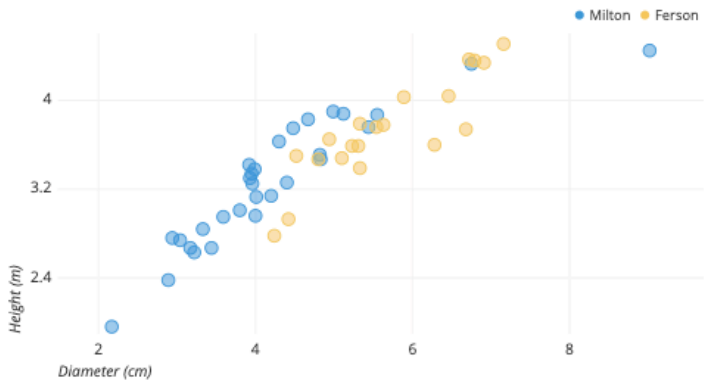
```
In [10]: variance_ratio = pca.explained_variance_ratio_
         variance_ratio
Out[10]: array([0.72770452, 0.23030523])

In [11]: total_variance = variance_ratio[0] + variance_ratio[1]
         total_variance
Out[11]: 0.9580097536148199
```

==> We have **retained 95.8% of the original variance** between the features although we reduced dimensions by 2
In other words, we retained 95.8%. Of the relevant information while significantly reducing the dimensions

Reference for Scatter plot - it's usually mapping 2 variables, but we can also use a categorical variable to color code them

Categorical third variable

A common modification of the basic scatter plot is the addition of a third variable. Values of the third variable can be encoded by modifying how the points are plotted. For a third variable that indicates categorical values (like geographical region or gender), the most common encoding is through point color. Giving each point a distinct hue makes it easy to show membership of each point to a respective group.



Now let's use **PCA to improve the performance** of the training of a model
We will use the MNIST dataset
Goal reduce over 700+ features (dimensions) down to something that is easier to train and more optimal to train without losing accuracy during training

Because we use PCA, a prerequisite it to use Standardization => use StandardScaler()
We also apply .85 for dimensions reduction

In [ ]:
```
from sklearn.datasets import fetch_mldata
mnist = fetch_mldata('MNIST original')
from sklearn.model_selection import train_test_split
from sklearn.preprocessing import StandardScaler
from sklearn.decomposition import PCA
from sklearn.linear_model import LogisticRegression
import time

# split data between train and test
train_img, test_img, train_lbl, test_lbl = train_test_split( mnist.data, mnist.target, test_size=1/7.0, random_s
scaler = StandardScaler()
# Fit training set only.
scaler.fit(train_img)

# Apply transform to both training set and test set.
train_img = scaler.transform(train_img)
test_img = scaler.transform(test_img)

# Make an instance of the Model
pca = PCA(.85)

# PCA fit
pca.fit(train_img)

# How many components did PCA choose?
print("\nNumber of components: %s\n" %pca.n_components_)
```

```
Number of components: 184
```

=> we have reduced the dimensions from 700+ to 184

Now running logistics regression

In [ ]:
```
# Transform both train and test datasets
train_img = pca.transform(train_img)
test_img = pca.transform(test_img)

logisticRegr = LogisticRegression(solver = 'lbfgs')

# Train the model
start_time = time.time()
logisticRegr.fit(train_img, train_lbl)
finish_time = time.time()

# Predict for Ten Observations (image)
logisticRegr.predict(test_img[0:10])

# Measure performance
logisticRegr.score(test_img, test_lbl)
```

We have 94.7% accuracy

Out[2]:
```
0.9154
```

In [3]:
```
run_time = finish_time - start_time
run_time
```

Out[3]:
```
18.499610424041748
```

It took 18 seconds

Now if we re-run that with more components: .95 from .85

```
test_img = scaler.transform(test_img)

# Make an instance of the Model
pca = PCA(.95)
```

```
Number of components: 330
```

We now have 330 dimensions from 700+

```
Out[5]: 0.9199

In [6]: run_time = finish_time - start_time
        run_time

Out[6]: 23.716309309005737
```

Accuracy now 91.9% and run in 23 seconds

=> Depending on how much variance we want to retain, we can reduce the # of dimensions accordingly and the model will take less time to run
The accuracy will not necessarily decrease

Principal Component Analysis (PCA)

Variance Retained	Number of Components	Time (seconds)	Accuracy
1.0	711	67.11	0.9155
0.99	541	54.90	0.9160
0.95	330	38.13	0.9200
0.90	236	29.19	0.9169
0.85	184	22.85	0.9154

=> we run several training run with various different PCA variance retained and compare
=> Automated **Model Tuning**

Encoding Categorical Values

Binarizer Encoding

- Several approaches
  - Binarizer Encoding: for features of a binary nature

We will use the Census dataset from UCI

jupyter 15-Encoding categorical values Last Checkpoint: 7 minutes ago (unsaved changes)

File Edit View Insert Cell Kernel Widgets Help Trusted conda_python3

Run Code nbdiff

In [1]:
```
# Class to colorize, bold, or underline output
class color:
   PURPLE = '\033[95m'
   CYAN = '\033[96m'
   DARKCYAN = '\033[36m'
   BLUE = '\033[94m'
   GREEN = '\033[92m'
   YELLOW = '\033[93m'
   RED = '\033[91m'
   BOLD = '\033[1m'
   UNDERLINE = '\033[4m'
   END = '\033[0m'
#

import pandas as pd
import numpy as np
# define header for our data, the UCI dataset does not have a header
headers = ["age", "workclass", "fnlwgt", "education", "education-num",
           "marital-status", "occupation", "relationship", "race",
           "sex", "capital-gain", "capital-loss", "hours-per-week", "native-country",
           "income"]
# load the data from github repo and convert the values with '?' to NaN
url = "https://raw.githubusercontent.com/vbloise3/WhizLabsML/master/CensusIncome/CensusIncomeDataset.csv"
df = pd.read_csv(url, error_bad_lines=False, header=None, names=headers, na_values="null")
# head of the df
df.head(10)
```

Out[1]:

	age	workclass	fnlwgt	education	education-num	marital-status	occupation	relationship	race	sex	capital-gain	capital-loss	hours-per-week	native-country	income
0	39	State-gov	77516	Bachelors	13	Never-married	Adm-clerical	Not-in-family	White	Male	2174	0	40	United-States	<=50K
1	50	Self-emp-not-inc	83311	Bachelors	13	Married-civ-spouse	Exec-managerial	Husband	White	Male	0	0	13	United-States	<=50K

Out[1]:

	age	workclass	fnlwgt	education	education-num	marital-status	occupation	relationship	race	sex	capital-gain	capital-loss	hours-per-week	native-country	income
0	39	State-gov	77516	Bachelors	13	Never-married	Adm-clerical	Not-in-family	White	Male	2174	0	40	United-States	<=50K
1	50	Self-emp-not-inc	83311	Bachelors	13	Married-civ-spouse	Exec-managerial	Husband	White	Male	0	0	13	United-States	<=50K
2	38	Private	215646	HS-grad	9	Divorced	Handlers-cleaners	Not-in-family	White	Male	0	0	40	United-States	<=50K
3	53	Private	234721	11th	7	Married-civ-spouse	Handlers-cleaners	Husband	Black	Male	0	0	40	United-States	<=50K
4	28	Private	338409	Bachelors	13	Married-civ-spouse	Prof-specialty	Wife	Black	Female	0	0	40	Cuba	<=50K
5	37	Private	284582	Masters	14	Married-civ-spouse	Exec-managerial	Wife	White	Female	0	0	40	United-States	<=50K
6	49	Private	160187	9th	5	Married-spouse-absent	Other-service	Not-in-family	Black	Female	0	0	16	Jamaica	<=50K
7	52	Self-emp-not-inc	209642	HS-grad	9	Married-civ-spouse	Exec-managerial	Husband	White	Male	0	0	45	United-States	>50K
8	31	Private	45781	Masters	14	Never-married	Prof-specialty	Not-in-family	White	Female	14084	0	50	United-States	>50K
9	42	Private	159449	Bachelors	13	Married-civ-spouse	Exec-managerial	Husband	White	Male	5178	0	40	United-States	>50K

Now separate the features from the target

In [2]:
```
# Separate the features
features_df = df.drop('income', axis=1)
# Separate the target
target_df = df.drop(df.columns[[0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13]], axis=1)
features_df
```

	age	workclass	fnlwgt	education	education-num	marital-status	occupation	relationship	race	sex	capital-gain	capital-loss	hours-per-week	native-country
0	39	State-gov	77516	Bachelors	13	Never-married	Adm-clerical	Not-in-family	White	Male	2174	0	40	United-States
1	50	Self-emp-not-inc	83311	Bachelors	13	Married-civ-spouse	Exec-managerial	Husband	White	Male	0	0	13	United-States
2	38	Private	215646	HS-grad	9	Divorced	Handlers-cleaners	Not-in-family	White	Male	0	0	40	United-States
3	53	Private	234721	11th	7	Married-civ-spouse	Handlers-cleaners	Husband	Black	Male	0	0	40	United-States
4	28	Private	338409	Bachelors	13	Married-civ-spouse	Prof-specialty	Wife	Black	Female	0	0	40	Cuba
5	37	Private	284582	Masters	14	Married-civ-spouse	Exec-managerial	Wife	White	Female	0	0	40	United-States
6	49	Private	160187	9th	5	Married-spouse-absent	Other-service	Not-in-family	Black	Female	0	0	16	Jamaica

In [3]:
```
# What data types are in the dataset
features_df.dtypes
```

Out[3]:
```
age                int64
workclass         object
fnlwgt             int64
education         object
education-num      int64
marital-status    object
occupation        object
relationship      object
race              object
sex               object
capital-gain       int64
capital-loss       int64
hours-per-week     int64
native-country    object
dtype: object
```

Now create a Datafame with just categorial features

```
In [4]: # Create a dataframe of only the categorical features
categorical_featuresDf = features_df.select_dtypes(include=['object']).copy()
categorical_featuresDf
```

Out[4]:

	workclass	education	marital-status	occupation	relationship	race	sex	native-country
0	State-gov	Bachelors	Never-married	Adm-clerical	Not-in-family	White	Male	United-States
1	Self-emp-not-inc	Bachelors	Married-civ-spouse	Exec-managerial	Husband	White	Male	United-States
2	Private	HS-grad	Divorced	Handlers-cleaners	Not-in-family	White	Male	United-States
3	Private	11th	Married-civ-spouse	Handlers-cleaners	Husband	Black	Male	United-States
4	Private	Bachelors	Married-civ-spouse	Prof-specialty	Wife	Black	Female	Cuba
5	Private	Masters	Married-civ-spouse	Exec-managerial	Wife	White	Female	United-States
6	Private	9th	Married-spouse-absent	Other-service	Not-in-family	Black	Female	Jamaica
7	Self-emp-not-inc	HS-grad	Married-civ-spouse	Exec-managerial	Husband	White	Male	United-States
8	Private	Masters	Never-married	Prof-specialty	Not-in-family	White	Female	United-States
9	Private	Bachelors	Married-civ-spouse	Exec-managerial	Husband	White	Male	United-States
10	Private	Some-college	Married-civ-spouse	Exec-managerial	Husband	Black	Male	United-States
11	State-gov	Bachelors	Married-civ-spouse	Prof-specialty	Husband	Asian-Pac-Islander	Male	India
12	Private	Bachelors	Never-married	Adm-clerical	Own-child	White	Female	United-States

Before we encode the categorical features, we need to remove any null or NaN values with the **isnull()** function

```
In [5]: # Find any null value entries in the categorical features
categorical_features_NaN = categorical_featuresDf[categorical_featuresDf.isnull().any(axis=1)]
categorical_features_NaN
```

Out[5]:

	workclass	education	marital-status	occupation	relationship	race	sex	native-country
14	Private	Assoc-voc	Married-civ-spouse	Craft-repair	Husband	Asian-Pac-Islander	Male	NaN
27	NaN	Some-college	Married-civ-spouse	NaN	Husband	Asian-Pac-Islander	Male	South
38	Private	Some-college	Married-civ-spouse	Sales	Husband	White	Male	NaN
51	Private	HS-grad	Never-married	Other-service	Own-child	White	Female	NaN
61	NaN	7th-8th	Married-spouse-absent	NaN	Not-in-family	White	Male	NaN
69	NaN	Some-college	Never-married	NaN	Own-child	White	Male	United-States
77	NaN	10th	Married-civ-spouse	NaN	Husband	White	Male	United-States
93	Private	HS-grad	Married-civ-spouse	Sales	Wife	Asian-Pac-Islander	Female	NaN
106	NaN	10th	Never-married	NaN	Own-child	White	Female	United-States
128	NaN	HS-grad	Married-civ-spouse	NaN	Husband	White	Male	United-States

W can see 2,399 rows with null values

2399 rows × 8 columns

3 features have a NaN value (or more)

```
In [6]: # Which features have a NaN value?
categorical_featuresDf.columns[categorical_featuresDf.isna().any()].tolist()
```

```
Out[6]: ['workclass', 'occupation', 'native-country']
```

Now let's use the **SimpleImputer() 'most_frequent'** method to replace the missing

value with the most frequent one, on all 3 features that have NaN values

```
In [ ]: #Impute the NaN values using scikit-learn SimpleImpute Class
from sklearn.impute import SimpleImputer
imputer = SimpleImputer(missing_values=np.nan, strategy='most_frequent')
# Impute workclass values
imputer = imputer.fit(features_df[['workclass']])
features_df['workclass'] = imputer.transform(features_df[['workclass']]).ravel()
# Impute occupation values
imputer = imputer.fit(features_df[['occupation']])
features_df['occupation'] = imputer.transform(features_df[['occupation']]).ravel()
# Impute native-country values
imputer = imputer.fit(features_df[['native-country']])
features_df['native-country'] = imputer.transform(features_df[['native-country']]).ravel()

# Recreate the dataframe of only the categorical features
categorical_featuresDf = features_df.select_dtypes(include=['object']).copy()

# Recheck to find any null value entries in the categorical features
categorical_features_NaN = categorical_featuresDf[categorical_featuresDf.isnull().any(axis=1)]
# Which features hava a NaN value?
categorical_featuresDf.columns[categorical_featuresDf.isna().any()].tolist()
```

We now have a Dataframe all cleanup and ready.

Now let's take care of the sex feature: "Male" or "Female" with **LabelBinariber**()

```
In [8]: # Use binary encoding for the sex feature
from sklearn.preprocessing import LabelBinarizer
# How many differernt sex feature value types
print(color.BOLD + color.PURPLE + "\nHow many different sex feature types?" + color.END)
print(categorical_featuresDf["sex"].value_counts())
```

```
How many different sex feature types?
 Male      21790
 Female    10771
Name: sex, dtype: int64
```

We will put the new code into a new feature called "sex_code"

```
In [9]: label_style = LabelBinarizer()
label_results = label_style.fit_transform(categorical_featuresDf["sex"])
print(color.BOLD + color.PURPLE + "\nLabelBinarizer of sex feature" + color.END)
categorical_featuresDf["sex_code"] = pd.DataFrame({'sex': label_results[:, 0]})
categorical_featuresDf[["sex", "sex_code"]].head(15)
```

```
LabelBinarizer of sex feature
```

Out[9]:

	sex	sex_code
0	Male	1
1	Male	1
2	Male	1
3	Male	1
4	Female	0
5	Female	0
6	Female	0
7	Male	1

=> That's how we **encode a categorical feature with LabelBinarizer**

Now let's look at **Lebel Encodin**g for encoding categorical features.

- Several approaches
  - Binarizer Encoding: for features of a binary nature
  - Label Encoding: may imply ordinality, can use Ordinal Encoder

Let's look at the **wordclass** feature: "Private",... and use **LabelEncoder()** to assign a value to each feature's value

Out[5]:

	workclass	education	marital-status	occupation	relationship	race	sex	native-country
14	Private	Assoc-voc	Married-civ-spouse	Craft-repair	Husband	Asian-Pac-Islander	Male	NaN
27	NaN	Some-college	Married-civ-spouse	NaN	Husband	Asian-Pac-Islander	Male	South
38	Private	Some-college	Married-civ-spouse	Sales	Husband	White	Male	NaN
51	Private	HS-grad	Never-married	Other-service	Own-child	White	Female	NaN
61	NaN	7th-8th	Married-spouse-absent	NaN	Not-in-family	White	Male	NaN
69	NaN	Some-college	Never-married	NaN	Own-child	White	Male	United-States
77	NaN	10th	Married-civ-spouse	NaN	Husband	White	Male	United-States
93	Private	HS-grad	Married-civ-spouse	Sales	Wife	Asian-Pac-Islander	Female	NaN
32304	Never-worked	HS-grad	Married-civ-spouse	NaN	Wife	Black	Female	United-States
32307	Self-emp-inc	Bachelors	Married-civ-spouse	Exec-managerial	Husband	White	Male	NaN
32310	NaN	Some-college	Never-married	NaN	Own-child	White	Male	United-States

In [ ]:
```
# Perform label encoding on workclass feature
from sklearn.preprocessing import LabelEncoder
label_work_class = LabelEncoder()
categorical_featuresDf["workclass_code"] = label_work_class.fit_transform(categorical_featuresDf["workclass"])
categorical_featuresDf[["workclass", "workclass_code"]].head(15)
```

Out[10]:

	workclass	workclass_code
0	State-gov	6
1	Self-emp-not-inc	5
2	Private	3
3	Private	3
4	Private	3
5	Private	3
6	Private	3
7	Self-emp-not-inc	5
8	Private	3
9	Private	3
10	Private	3
11	State-gov	6
12	Private	3
13	Private	3
14	Private	3

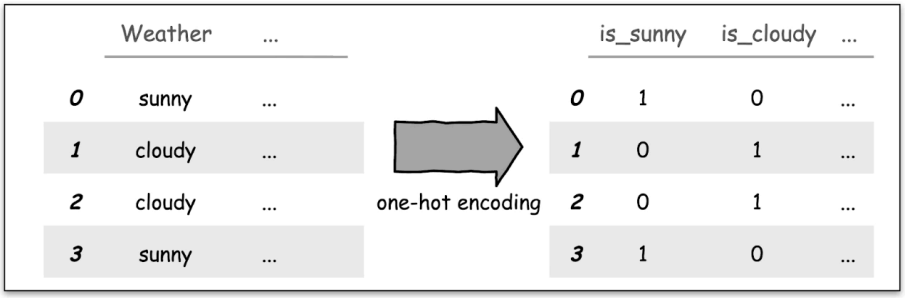
Problem with this, is that depending on the algorithm we use, the algorithm may think the order matters = these values are ordinal

To avoid that, we can use **OrdinalEncoder** or **One-hot-Encoding**

AWS Machine Learning - Data Engineering

Encoding Categorical Values

- Several approaches
  - Binarizer Encoding: for features of a binary nature
  - Label Encoding: may imply ordinality, can use Ordinal Encoder
  - One-hot-encoding: Change nominal categorical values such as 'true', 'false', or 'rainy', 'sunny' to numerical values



Let's **one-hot-encode** the wordclas feature with **get_dummies()**

In [11]:
```
# Use one-hot encoding on the workclass feature
# How many differernt workclass feature value types
print(color.BOLD + color.PURPLE + "\nHow many different workclass feature types?" + color.END)
print(categorical_featuresDf["workclass"].value_counts())
```

```
How many different workclass feature types?
 Private             24532
 Self-emp-not-inc     2541
 Local-gov            2093
 State-gov            1298
 Self-emp-inc         1116
 Federal-gov           960
 Without-pay            14
 Never-worked            7
Name: workclass, dtype: int64
```

In [12]:
```
# One-hot encode the workclass feature
pd.get_dummies(categorical_featuresDf, columns=["workclass"]).head()
```

Out[12]:

onship	race	sex	native-country	sex_code	workclass_code	workclass_Federal-gov	workclass_Local-gov	workclass_Never-worked	workclass_Private	workclass_Self-emp-inc	workclass_Self-emp-not-inc	workclass_State-gov	workclass_Without-pay
Not-in-family	White	Male	United-States	1	6	0	0	0	0	0	0	1	0
usband	White	Male	United-States	1	5	0	0	0	0	0	1	0	0
Not-in-family	White	Male	United-States	1	3	0	0	0	1	0	0	0	0
usband	Black	Male	United-States	1	3	0	0	0	1	0	0	0	0
Wife	Black	Female	Cuba	0	3	0	0	0	1	0	0	0	0

=> one hot encoding breaks out a feature so that it has several binary values, based on the outcome foe each one.
Pb: for every potential answer, we get a new binary feature that is added

Here, we have 8 feature for just wordclass_code

Let’s see one hot encoding for marital status

In [ ]:
```
# Use one-hot encoding on the marital-status feature

# How many differernt marital-status feature value types
print(color.BOLD + color.PURPLE + "\nHow many different marital-status feature types?" + color.END)
print(categorical_featuresDf["marital-status"].value_counts())

# One-hot encode the marital-status feature
pd.get_dummies(categorical_featuresDf, columns=["marital-status"]).head()
```

=> we have 7 categories ==> 7 One-hot encode features

```
How many different marital-status feature types?
 Married-civ-spouse       14976
 Never-married            10683
 Divorced                  4443
 Separated                 1025
 Widowed                   993
 Married-spouse-absent     418
 Married-AF-spouse         23
Name: marital-status, dtype: int64
```

Out[13]:

ducation	occupation	relationship	race	sex	native-country	sex_code	workclass_code	marital-status_Divorced	marital-status_Married-AF-spouse	marital-status_Married-civ-spouse	marital-status_Married-spouse-absent	marital-status_Never-married	marital-status_Separated	marital-status_Widowed
Bachelors	Adm-clerical	Not-in-family	White	Male	United-States	1	6	0	0	0	0	1	0	0
Bachelors	Exec-managerial	Husband	White	Male	United-States	1	5	0	0	1	0	0	0	0
HS-grad	Handlers-cleaners	Not-in-family	White	Male	United-States	1	3	1	0	0	0	0	0	0
11th	Handlers-cleaners	Husband	Black	Male	United-States	1	3	0	0	1	0	0	0	0
Bachelors	Prof-specialty	Wife	Black	Female	Cuba	0	3	0	0	1	0	0	0	0

Let's see how it can explode to a LOT of features with native-country

In [ ]:
```
# Use one-hot encoding on the native-country feature

# How many differernt native-country feature value types
print(color.BOLD + color.PURPLE + "\nHow many different native-country feature types?" + color.END)
print(categorical_featuresDf["native-country"].value_counts())

# One-hot encode the native-country feature
pd.get_dummies(categorical_featuresDf, columns=["native-country"]).head()
```

Now we can see a LOT of features

```
How many different native-country feature types?
 United-States                 29753
 Mexico                          643
 Philippines                     198
 Germany                         137
 Canada                          121
 Puerto-Rico                     114
 El-Salvador                     106
 India                           100
 Cuba                             95
 England                          90
 Jamaica                          81
 South                            80
 China                            75
 Italy                            73
 Dominican-Republic               70
 Vietnam                          67
 Guatemala                        64
 Japan                            62
 Poland                           60
 Columbia                         59
 Taiwan                           51
 Haiti                            44
 Iran                             43
 Portugal                         37
 Nicaragua                        34
 Peru                             31
 France                           29
 Greece                           29
 Ecuador                          28
 Ireland                          24
 Hong                             20
 Cambodia                         19
 Trinadad&Tobago                  19
```

```
Laos                            18
Thailand                        18
Yugoslavia                      16
Outlying-US(Guam-USVI-etc)      14
Honduras                        13
Hungary                         13
Scotland                        12
Holand-Netherlands               1
Name: native-country, dtype: int64
```

Out[14]:

	workclass	education	marital-status	occupation	relationship	race	sex	sex_code	workclass_code	native-country_Cambodia	...	native-country_Portugal	native-country_Puerto-Rico	native-country_Scotland	native-country_South
0	State-gov	Bachelors	Never-married	Adm-clerical	Not-in-family	White	Male	1	6	0	...	0	0	0	0
1	Self-emp-not-inc	Bachelors	Married-civ-spouse	Exec-managerial	Husband	White	Male	1	5	0	...	0	0	0	0
2	Private	HS-grad	Divorced	Handlers-cleaners	Not-in-family	White	Male	1	3	0	...	0	0	0	0
3	Private	11th	Married-civ-spouse	Handlers-cleaners	Husband	Black	Male	1	3	0	...	0	0	0	0
4	Private	Bachelors	Married-civ-spouse	Prof-specialty	Wife	Black	Female	0	3	0	...	0	0	0	0

5 rows × 50 columns

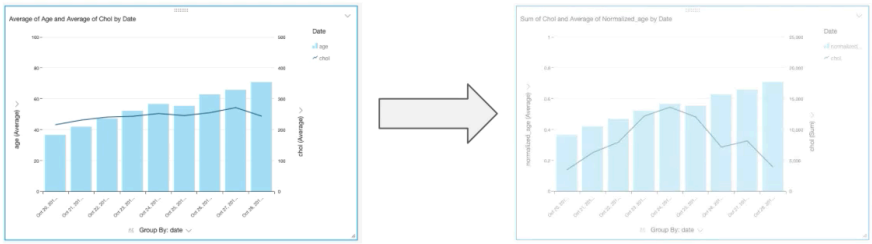
=> It's added 50 new column...
This is not what we want to do.

So we need to make a judgement call when using one-hot-encoding.
We may want to use lavbelEncoding instead even though it may infer ordinality

Numerical Feature Engineering

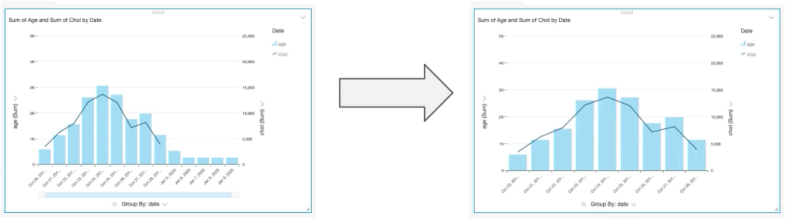
Normalization

- ❏ Transform numeric values so machine learning algorithms can better analyze them
- ❏ Change numeric values so all values are on the same scale
  - ❏ Normalization: rescales the values into a range of [0,1]



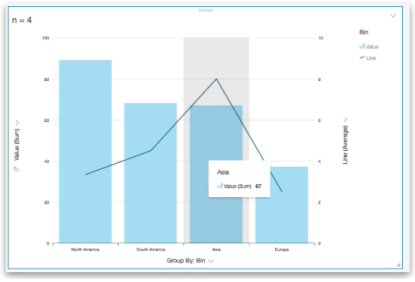
Standardization

- Standardization: rescales data to have a mean of 0 and a standard deviation of 1 (unit variance)



Binning

- Binning



Binning = discretization = quantization

- Binning - AKA discretization or quantization
  - Categorical Binning
    - Group categorical values to gain insight into data: countries by geographical region
  - Numerical Binning
    - Divides continuous feature into a specified number of categories or bins, thus making the data discrete
    - Reduces the number of discrete intervals of a continuous feature
  - Quantile Binning
    - Divide up data into equal sized bins

Example in a notebook

Normalization with **preprocessing.normalize()**:

In [ ]:
```
# Class to colorize, bold, or underline output
class color:
   PURPLE = '\033[95m'
   CYAN = '\033[96m'
   DARKCYAN = '\033[36m'
   BLUE = '\033[94m'
   GREEN = '\033[92m'
   YELLOW = '\033[93m'
   RED = '\033[91m'
   BOLD = '\033[1m'
   UNDERLINE = '\033[4m'
   END = '\033[0m'
#
import pandas as pd
import numpy as np

# Normalization: rescales the values into a range of [0,1]
data = np.array([[999999, 2, 0, 9], [35, 8, 4, 7], [27, 23, 1, 99]])
data_raw = pd.DataFrame({'Column1': data[:, 0], 'Column2': data[:, 1], 'Column3': data[:, 2], 'Column4': data[:,
print(color.BOLD + color.PURPLE + "\nData before normalization" + color.END)
print(data_raw)
from sklearn import preprocessing
# normalize the data attributes
normalized_data = preprocessing.normalize(data)
dataset = pd.DataFrame({'Column1': normalized_data[:, 0], 'Column2': normalized_data[:, 1], 'Column3': normalize
print(color.BOLD + color.PURPLE + "\nData after normalization" + color.END)
print(dataset)
```

```
Data before normalization
   Column1  Column2  Column3  Column4
0   999999        2        0        9
1       35        8        4        7
2       27       23        1       99

Data after normalization
    Column1   Column2   Column3   Column4
0  1.000000  0.000002  0.000000  0.000009
1  0.951171  0.217411  0.108705  0.190234
2  0.256736  0.218701  0.009509  0.941364
```

=> **This helps us deal with Outliers**

Standardization with **StandardScaler**():

In [ ]:
```
# Standardization: rescales data to have a mean of 0 and a standard deviation of 1 (unit variance)
from sklearn.preprocessing import StandardScaler
data = np.array([[999999, 2, 0, 999], [35, 1, 4, 7], [27, 3, 1, 99]])
data_raw = pd.DataFrame({'Column1': data[:, 0], 'Column2': data[:, 1], 'Column3': data[:, 2], 'Column4': data[:,
print(color.BOLD + color.PURPLE + "\nData before standardization" + color.END)
print(data_raw)
standardized_data = StandardScaler().fit_transform(data_raw)
dataset = pd.DataFrame({'Column1': standardized_data[:, 0], 'Column2': standardized_data[:, 1], 'Column3': standa
print(color.BOLD + color.PURPLE + "\nData after standardization" + color.END)
print(dataset)
```

```
Data before standardization
   Column1  Column2  Column3  Column4
0   999999        2        0      999
1       35        1        4        7
2       27        3        1       99

Data after standardization
    Column1   Column2   Column3   Column4
0  1.414214  0.000000 -0.980581  1.409224
1 -0.707098 -1.224745  1.372813 -0.807399
2 -0.707115  1.224745 -0.392232 -0.601825
```

We can see the data with a mean of 0 and standard deviation of 1

Categorical Binning

```
countries = pd.Series([
        'United States',
        'Canada',
        'Spain',
        'Italy',
        'Chile',
        'Brazil',
        'North Korea',
        'Vietnam',
        'Thailand',
        'Malaysia',
        'Mayanmar',
        'Iceland',
        'Germany',
        'Cuba',
        'Mexico'
    ], name = "Countries")

groups = {
        'North America': ('United States', 'Canada', 'Iceland', 'Greenland', 'Mexico'),
        'Europe': ('France', 'Germany', 'United Kingdom', 'Belgium', 'Spain', 'Italy', 'Poland'),
        'South America': ('Chile', 'Brazil', 'Peru', 'Colombia', 'Ecuador'),
        'Asia': ('North Korea', 'Vietnam', 'Thailand', 'Malaysia', 'Mayanmar')
    }
from typing import Any

def country_group_map(series: pd.Series, groups: dict,
                      othervalue: Any=-1) -> pd.Series:
    # Assign the dictionary pairs
    groups = {z: j for j, r in groups.items() for z in r}
    return series.map(groups).fillna(othervalue)

grouped_countries = country_group_map(countries, groups, othervalue='other')
df = pd.concat([countries.rename('Countries'), grouped_countries.rename('Grouped Countries')], axis=1)
print(color.BOLD + color.PURPLE + "\nCountry by region" + color.END)
df
```

Country by region

Out[3]:

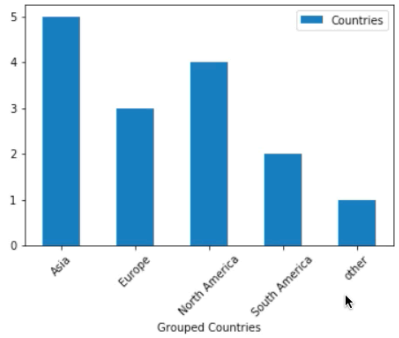
	Countries	Grouped Countries
0	United States	North America
1	Canada	North America
2	Spain	Europe
3	Italy	Europe
4	Chile	South America
5	Brazil	South America
6	North Korea	Asia
7	Vietnam	Asia
8	Thailand	Asia
9	Malaysia	Asia
10	Mayanmar	Asia
11	Iceland	North America
12	Germany	Europe
13	Cuba	other
14	Mexico	North America

We now have bins = Grouped Countries
Let’s visualize in matplotlib

In [5]:
```
# Show the counts of countries in each region
import matplotlib.pyplot as plt
data = df.groupby('Grouped Countries').count()
print(color.BOLD + color.PURPLE + "\nCountry by region" + color.END)
data.plot.bar(by='Grouped Countries', rot=45)
```

Country by region

Out[5]: <matplotlib.axes._subplots.AxesSubplot at 0x7f1b4755a748>



Noe let's look at **binning continuous data**

In [ ]:
```
import pandas as pd
import numpy as np
import seaborn as sns

# Numerical Binning Example
revenue_df = pd.read_excel('https://github.com/vbloise3/WhizLabsML/blob/master/binning/201_Revenue_Total.xlsx?ra
print(color.BOLD + color.PURPLE + "\nNumber of observations: " + str(len(revenue_df.index)) + color.END)
revenue_df
```

Number of observations: 1507

Out[6]:

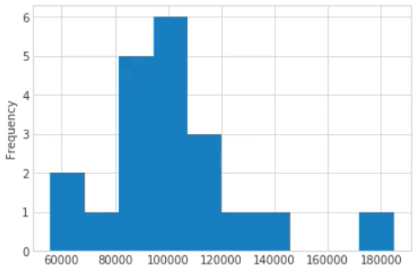
	reference number	name	price	date
0	740150	Thomas LLC	3380.91	2018-01-01 07:21:51
1	714466	Sillians-Krapfs	-63.16	2018-01-01 10:00:47
2	218895	Snodgrass Inc	2086.10	2018-01-01 13:24:58
3	307599	Craut, Oniote and Johnson	863.05	2018-01-01 15:05:22
4	412290	Judais-Pewrters	499.26	2018-01-01 23:26:55
5	714466	Sillians-Krapfs	1489.71	2018-01-02 10:07:15
6	218895	Snodgrass Inc	62.20	2018-01-02 10:57:23
7	729833	Victory Ltd	266.00	2018-01-03 06:32:11
8	714466	Sillians-Krapfs	1849.98	2018-01-03 11:29:02
9	737550	Frick, Sponge and Bobus	1146.88	2018-01-03 19:07:37
10	146832	Jones-Spank	1016.10	2018-01-03 19:39:53
11	688981	Frieling LLC	141.82	2018-01-04 00:02:36
12	786968	Francis, Malin and Smith	367.86	2018-01-04 06:51:53
13	307599	Craut, Oniote and Johnson	211.48	2018-01-04 07:53:01
14	737550	Frick, Sponge and Bobus	1645.88	2018-01-04 08:57:48

1505	424914	Green-Brown	1915.00	2018-12-22 03:31:36
1506	424914	Green-Brown	8819.00	2018-12-16 00:46:26

1507 rows × 4 columns

In [7]:
```
# Bin continuous datas by using a simple histogram
sns.set_style('whitegrid')
df = revenue_df.groupby(['reference number', 'name'])['price'].sum().reset_index()
df['price'].plot(kind='hist')
```

Out[7]:
```
<matplotlib.axes._subplots.AxesSubplot at 0x7f1b44e71748>
```



Now let's do **quantile binning** with **qcut()**

In [ ]:
```
# Quantile binning example
print(color.BOLD + color.PURPLE + "\nqcut: a quantile-based discretization function that divides up the data into
print(df['price'].describe())
df['quantile_price_quartiles'] = pd.qcut(df['price'], q=4)
df['quantile_price_deciles'] = pd.qcut(df['price'], q=10, precision=0)
bin_labels_5 = ['Rookie', 'Sophomore', 'Junior', 'Senior', 'Professional']
df['quantile_price_quintiles'] = pd.qcut(df['price'],
                                         q=[0, .2, .4, .6, .8, 1],
                                         labels=bin_labels_5)
print(color.BOLD + color.PURPLE + "\nThree examples: quartiles, deciles, quintiles" + color.END)
df.head()
```

```
qcut: a quantile-based discretization function that divides up the data into equal sized bins
count        20.000000
mean     101711.287500
std       27037.449673
min       55733.050000
25%       89137.707500
50%      100271.535000
75%      110132.552500
max      184793.700000
Name: price, dtype: float64

Three examples: quartiles, deciles, quintiles
```

Out[8]:

	reference number	name	price	quantile_price_quartiles	quantile_price_deciles	quantile_price_quintiles
0	141962	Speilmans LLC	63626.03	(55733.049000000006, 89137.708]	(55732.0, 76471.0]	Rookie
1	146832	Jones-Spank	99608.77	(89137.708, 100271.535]	(95908.0, 100272.0]	Junior
2	163416	Muns-Kalifan	77898.21	(55733.049000000006, 89137.708]	(76471.0, 87168.0]	Rookie
3	218895	Snodgrass Inc	137351.96	(110132.552, 184793.7]	(124778.0, 184794.0]	Professional
4	239344	Finley LLC	91535.92	(89137.708, 100271.535]	(90686.0, 95908.0]	Sophomore

Now let's look at bin ranges

In [ ]:
```
# Check the bin ranges
print(color.BOLD + color.PURPLE + "\nWhat ranges identify the bins?" + color.END)
results, bin_edges = pd.qcut(df['price'],
                             q=[0, .2, .4, .6, .8, 1],
                             labels=bin_labels_5,
                             retbins=True)

results_table = pd.DataFrame(zip(bin_edges, bin_labels_5),
                             columns=['Threshold', 'Tier'])
print(results_table)
print(color.BOLD + color.PURPLE + "\nBins are of unequal size ..." + color.END)
```

```
What ranges identify the bins?
    Threshold          Tier
0   55733.050        Rookie
1   87167.958     Sophomore
2   95908.156        Junior
3  103605.970        Senior
4  112290.054  Professional

Bins are of unequal size ...
```

In [ ]:
```
# Check the bin distribution
print(color.BOLD + color.PURPLE + "\nBin distribution:" + color.END)
print(df['quantile_price_quintiles'].value_counts())
print(color.BOLD + color.PURPLE + "\n... but the data is evenly distributed across the bins" + color.END)
```

```
Bin distribution:
Professional    4
Senior          4
Junior          4
Sophomore       4
Rookie          4
Name: quantile_price_quintiles, dtype: int64

... but the data is evenly distributed across the bins
```

=> Quantile binning = data distributed evenly

Now use cut() method to compare with qcut()

In [ ]:
```
# Use cut to define the bin edges

# Remove the previous bins for simplicity
df = df.drop(columns = ['quantile_price_quartiles','quantile_price_deciles', 'quantile_price_quintiles'])
# Use cut to split the data into 4 equal bin sizes
pd.cut(df['price'], bins=4)
# How was the data distributed?
print(color.BOLD + color.PURPLE + "\nAfter using cut to create 4 bins:" + color.END)
print(pd.cut(df['price'], bins=4).value_counts())
print(color.BOLD + color.PURPLE + "\nBins are of equal size, but the data is unevenly distributed across the bin
```

```
After using cut to create 4 bins:
(87998.212, 120263.375]     12
(55603.989, 87998.212]       5
(120263.375, 152528.538]     2
(152528.538, 184793.7]       1
Name: price, dtype: int64

Bins are of equal size, but the data is unevenly distributed across the bins
```

cut() gives enough size bins (by bin edges) but id does not distribute the data evenly across bins