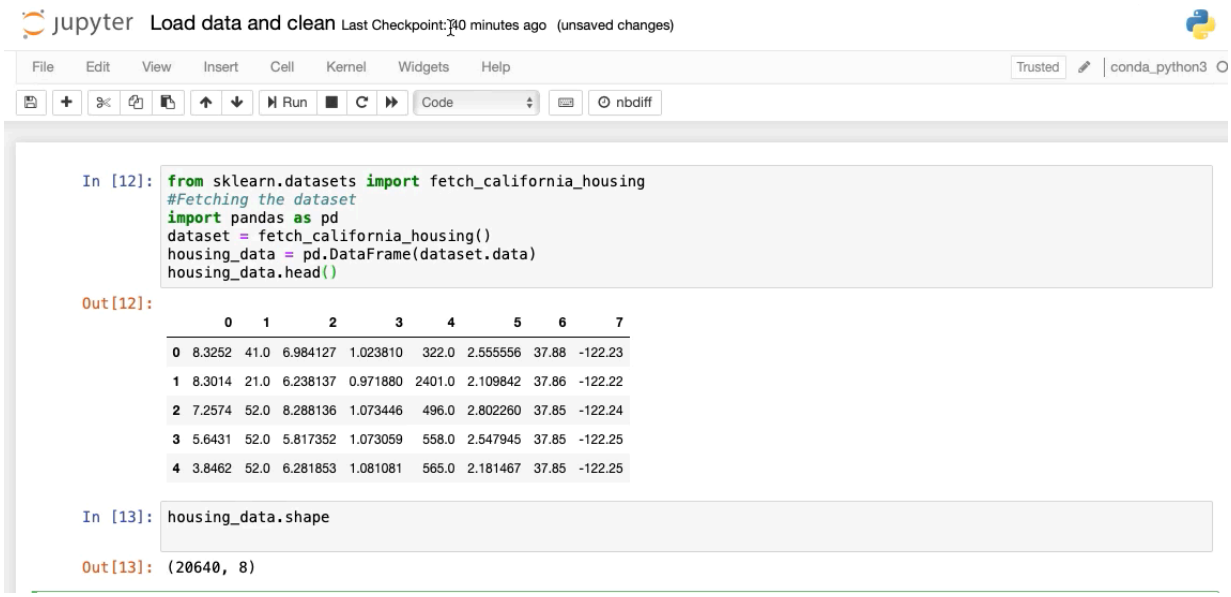
Whizlabs - ML Specialty Exam Course - Data Engineering

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

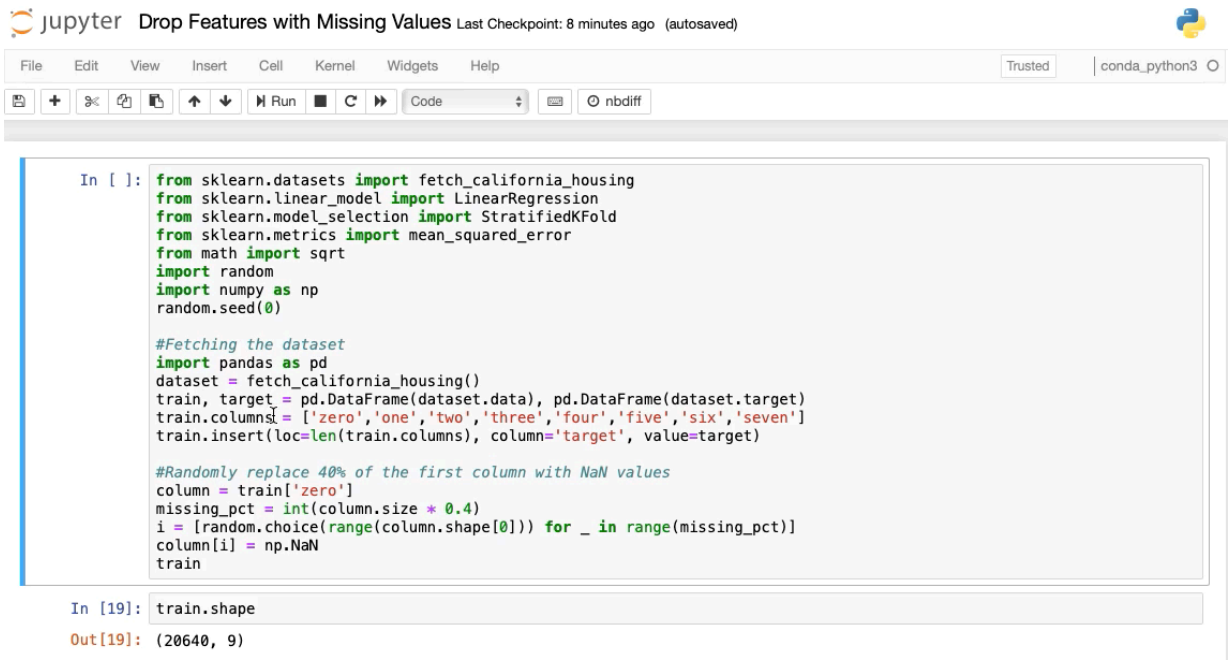
AWS Machine Learning - Handling Missing Data

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]:

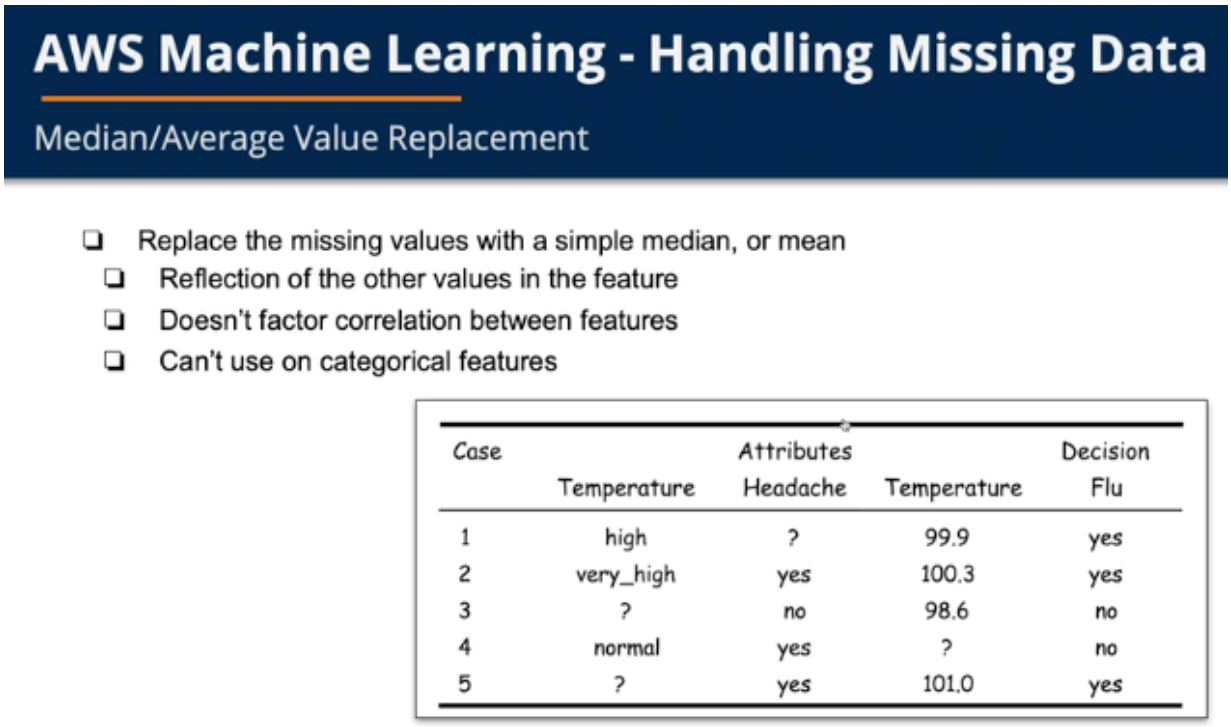
	zero	one	two	three	four	five	six	seven	target
0	NaN	41.0	6.984127	1.023810	322.0	2.555556	37.88	-122.23	4.526
1	8.3014	21.0	6.238137	0.971880	2401.0	2.109842	37.86	-122.22	3.585
2	NaN	52.0	8.288136	1.073446	496.0	2.802260	37.85	-122.24	3.521
3	5.6431	52.0	5.817352	1.073059	558.0	2.547945	37.85	-122.25	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

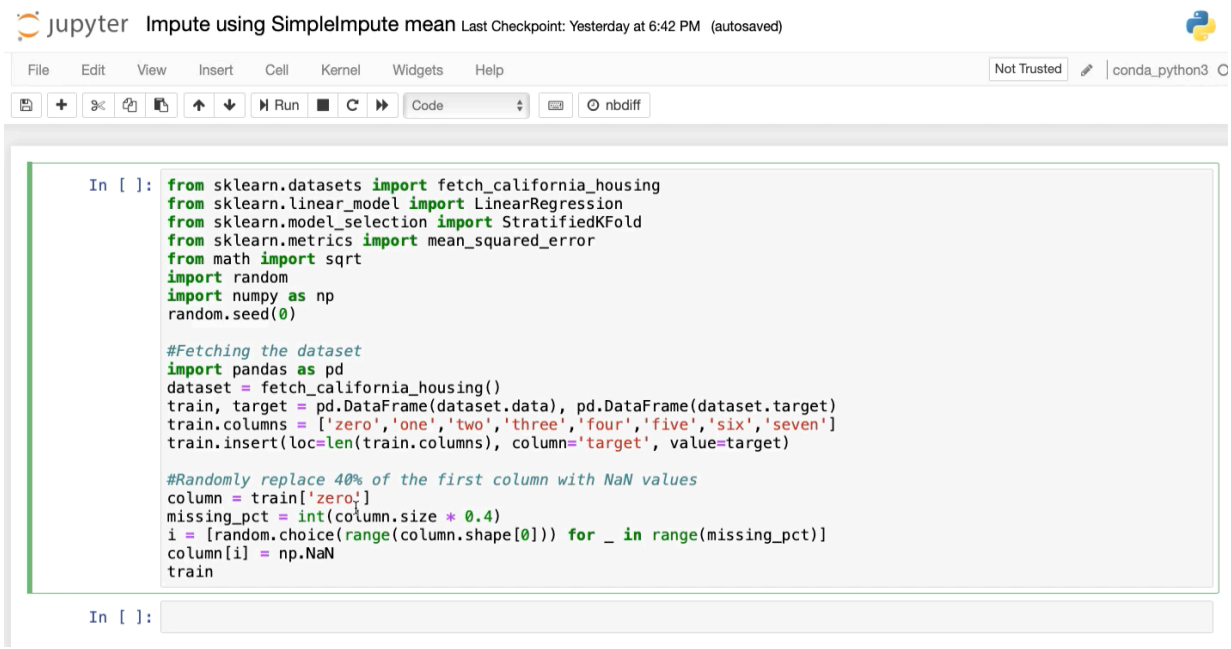
```
In [22]: train.shape

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



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

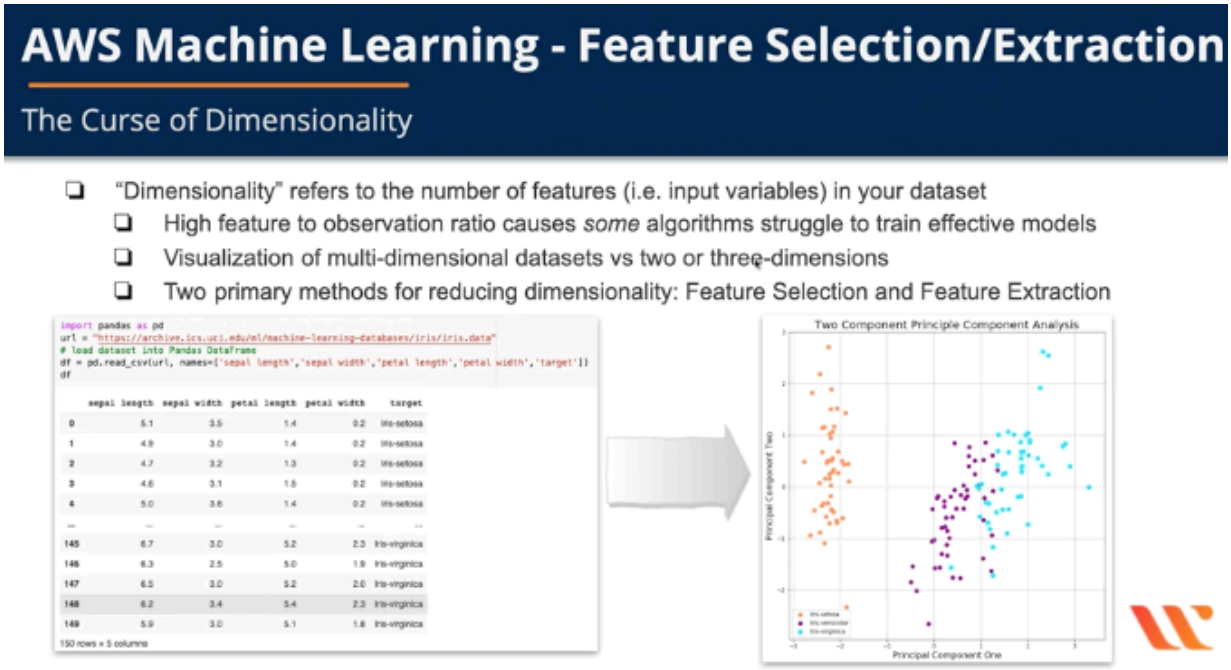
- ❏ Interpolation / Extrapolation
  - ❏ Estimate values from other observations within the range of a discrete set of known data points
- ❏ Forward filling / Backward filling
  - ❏ Fill the missing value by filling it from the preceding value or the succeeding value
- ❏ Hot deck imputation
  - ❏ Randomly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

Feature Extraction and feature selection

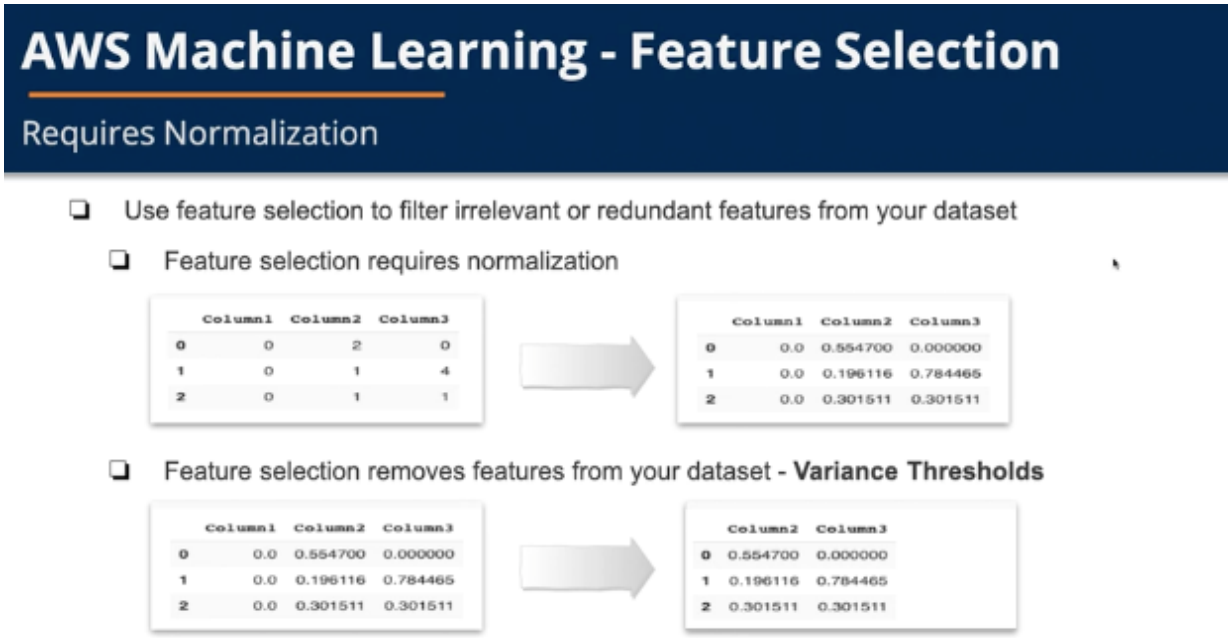
Visualizing multi-dimensions is pretty difficult - example: 4 features from Iris dataset below
There are 2 ways to reduce the features:

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

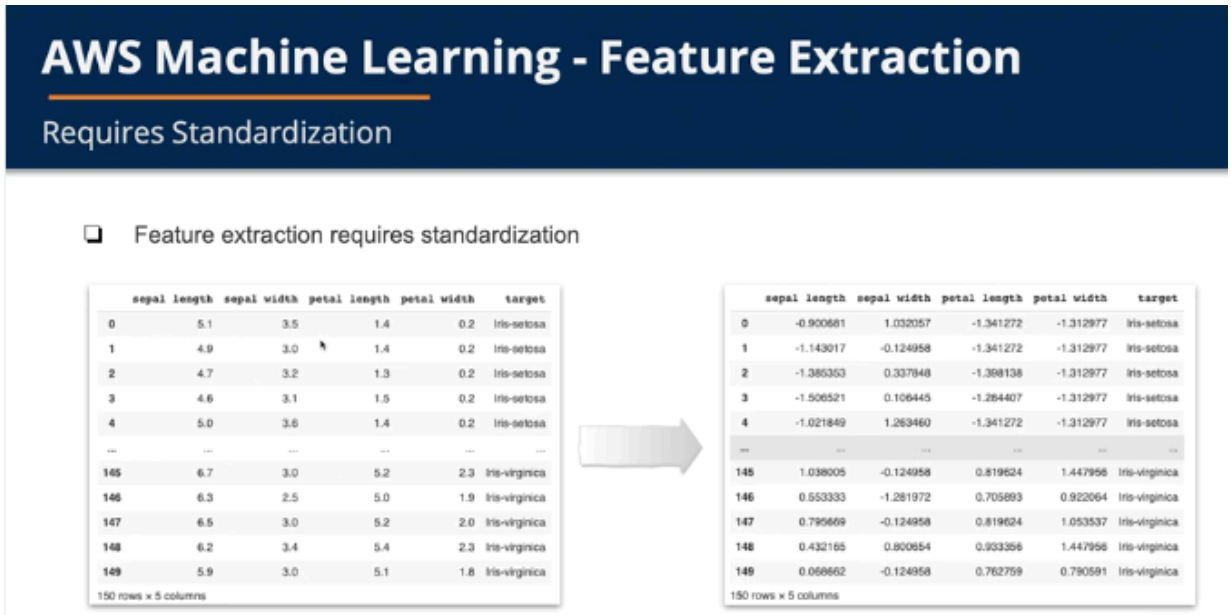
With feature extraction like PCA, we can extract features in a lower dimension space from 4 to 2 dimensions. Ex below



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

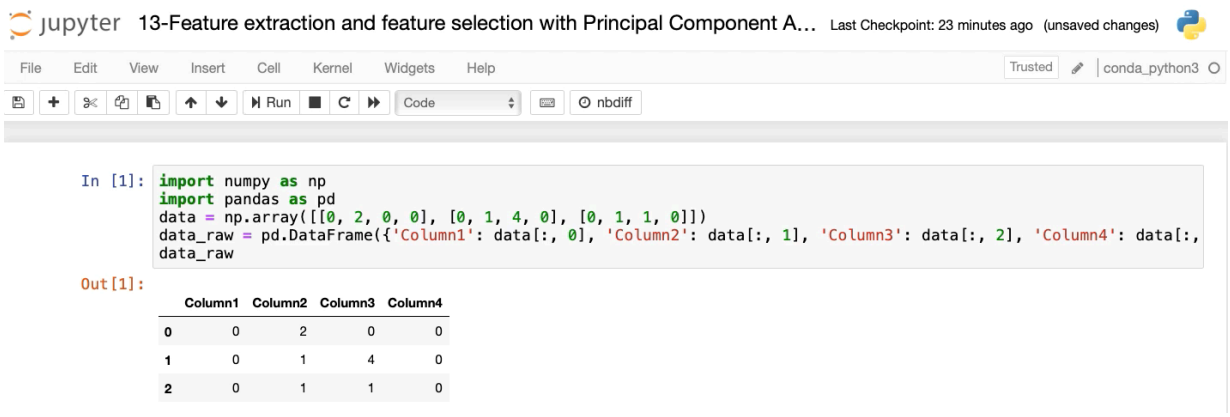
Principal Component Analysis (PCA)

- 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



	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

	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)

	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

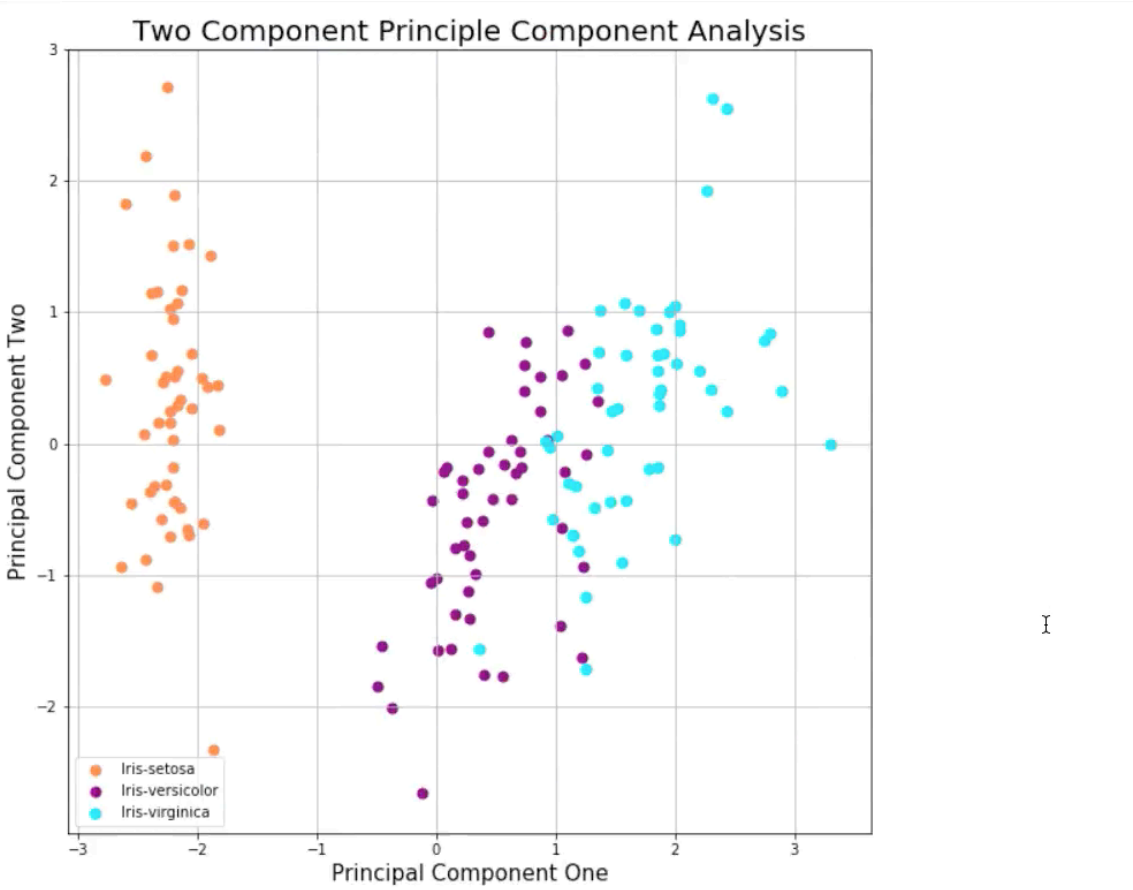
In [7]:
```
finalDf = pd.concat([principalDf, df[['target']]], axis = 1)
finalDf
```

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.