**PCA:**

import numpy as np

from sklearn.decomposition import PCA

import pandas as pd

import matplotlib.pyplot as plt

from sklearn.preprocessing import scale

#Load data set

data = pd.read\_csv('https://archive.ics.uci.edu/ml/machine-learning-databases/blood-transfusion/transfusion.data',

sep= ',', header = 0 )

print ("Dataset Lenght: ", len(data))

print ("Dataset Shape: ", data.shape)

print ("Dataset: ",data.head())

#convert it to numpy arrays

X=data.values

#Scaling the values

X = scale(data)

pca = PCA(n\_components=5)

pca.fit(X)

#The amount of variance that each PC explains

var= pca.explained\_variance\_ratio\_

print("Variance of each component")

print(var)

#Cumulative Variance explains

var1=np.cumsum(np.round(pca.explained\_variance\_ratio\_, decimals=4)\*100)

print("Cumulative variance")

print(var1)

plt.plot(var1)

plt.show()

#first 3 pca cumulatively give 100 percent vriance.so 3 variables taken

pca = PCA(n\_components=3)

pca.fit(X)

X1=pca.fit\_transform(X)

print("Fitting and dimensionality reduction")

print(X1)

KMEANS:

*class*sklearn.cluster.**KMeans**(*n\_clusters=8*, *init=’k-means++’*, *n\_init=10*, *max\_iter=300*, *tol=0.0001*, *precompute\_distances=’auto’*, *verbose=0*, *random\_state=None*, *copy\_x=True*, *n\_jobs=1*, *algorithm=’auto’*)[[source]](https://github.com/scikit-learn/scikit-learn/blob/a24c8b46/sklearn/cluster/k_means_.py#L707)

K-Means clustering

Read more in the [User Guide](http://scikit-learn.org/stable/modules/clustering.html#k-means).

|  |  |
| --- | --- |
| **Parameters:** | **n\_clusters** : int, optional, default: 8  The number of clusters to form as well as the number of centroids to generate.  **init** : {‘k-means++’, ‘random’ or an ndarray}  Method for initialization, defaults to ‘k-means++’:  ‘k-means++’ : selects initial cluster centers for k-mean clustering in a smart way to speed up convergence. See section Notes in k\_init for more details.  ‘random’: choose k observations (rows) at random from data for the initial centroids.  If an ndarray is passed, it should be of shape (n\_clusters, n\_features) and gives the initial centers.  **n\_init** : int, default: 10  Number of time the k-means algorithm will be run with different centroid seeds. The final results will be the best output of n\_init consecutive runs in terms of inertia.  **max\_iter** : int, default: 300  Maximum number of iterations of the k-means algorithm for a single run.  **tol** : float, default: 1e-4  Relative tolerance with regards to inertia to declare convergence  **precompute\_distances** : {‘auto’, True, False}  Precompute distances (faster but takes more memory).  ‘auto’ : do not precompute distances if n\_samples \* n\_clusters > 12 million. This corresponds to about 100MB overhead per job using double precision.  True : always precompute distances  False : never precompute distances  **verbose** : int, default 0  Verbosity mode.  **random\_state** : int, RandomState instance or None, optional, default: None  If int, random\_state is the seed used by the random number generator; If RandomState instance, random\_state is the random number generator; If None, the random number generator is the RandomState instance used by np.random.  **copy\_x** : boolean, default True  When pre-computing distances it is more numerically accurate to center the data first. If copy\_x is True, then the original data is not modified. If False, the original data is modified, and put back before the function returns, but small numerical differences may be introduced by subtracting and then adding the data mean.  **n\_jobs** : int  The number of jobs to use for the computation. This works by computing each of the n\_init runs in parallel.  If -1 all CPUs are used. If 1 is given, no parallel computing code is used at all, which is useful for debugging. For n\_jobs below -1, (n\_cpus + 1 + n\_jobs) are used. Thus for n\_jobs = -2, all CPUs but one are used.  **algorithm** : “auto”, “full” or “elkan”, default=”auto” |

import numpy as np

import pandas as pd

from sklearn.preprocessing import StandardScaler

from sklearn.cluster import KMeans

from sklearn.metrics import confusion\_matrix

from sklearn.cross\_validation import train\_test\_split

from sklearn.metrics import accuracy\_score

from sklearn.metrics import classification\_report

balance\_data = pd.read\_csv('https://archive.ics.uci.edu/ml/machine-learning-databases/blood-transfusion/transfusion.data',

sep= ',', header = 0 )

print ("Dataset Lenght: ", len(balance\_data))

print ("Dataset Shape: ", balance\_data.shape)

print ("Dataset: ",balance\_data.head())

X = balance\_data.values[:, 0:5]

X\_train, X\_test = train\_test\_split(

X, test\_size = 0.3, random\_state = 100)

kmeans = KMeans(n\_clusters=5, random\_state=0).fit(X\_train)

print("Cluster centers")

print(kmeans.cluster\_centers\_)

print("Predicting")

print(X\_test.shape)

print(kmeans.predict(X\_test))

**AGGLOMERATIVE CLUSTERING:**

*class*sklearn.cluster.**AgglomerativeClustering**(*n\_clusters=2*, *affinity=’euclidean’*, *memory=None*, *connectivity=None*, *compute\_full\_tree=’auto’*, *linkage=’ward’*, *pooling\_func=<function mean>*)[[source]](https://github.com/scikit-learn/scikit-learn/blob/a24c8b46/sklearn/cluster/hierarchical.py#L593)

Agglomerative Clustering

Recursively merges the pair of clusters that minimally increases a given linkage distance.

Read more in the [User Guide](http://scikit-learn.org/stable/modules/clustering.html#hierarchical-clustering).

|  |  |
| --- | --- |
| **Parameters:** | **n\_clusters** : int, default=2  The number of clusters to find.  **affinity** : string or callable, default: “euclidean”  Metric used to compute the linkage. Can be “euclidean”, “l1”, “l2”, “manhattan”, “cosine”, or ‘precomputed’. If linkage is “ward”, only “euclidean” is accepted.  **memory** : None, str or object with the joblib.Memory interface, optional  Used to cache the output of the computation of the tree. By default, no caching is done. If a string is given, it is the path to the caching directory.  **connectivity** : array-like or callable, optional  Connectivity matrix. Defines for each sample the neighboring samples following a given structure of the data. This can be a connectivity matrix itself or a callable that transforms the data into a connectivity matrix, such as derived from kneighbors\_graph. Default is None, i.e, the hierarchical clustering algorithm is unstructured.  **compute\_full\_tree** : bool or ‘auto’ (optional)  Stop early the construction of the tree at n\_clusters. This is useful to decrease computation time if the number of clusters is not small compared to the number of samples. This option is useful only when specifying a connectivity matrix. Note also that when varying the number of clusters and using caching, it may be advantageous to compute the full tree.  **linkage** : {“ward”, “complete”, “average”}, optional, default: “ward”  Which linkage criterion to use. The linkage criterion determines which distance to use between sets of observation. The algorithm will merge the pairs of cluster that minimize this criterion.   * ward minimizes the variance of the clusters being merged. * average uses the average of the distances of each observation of the two sets. * complete or maximum linkage uses the maximum distances between all observations of the two sets.   **pooling\_func** : callable, default=np.mean  This combines the values of agglomerated features into a single value, and should accept an array of shape [M, N] and the keyword argument axis=1, and reduce it to an array of size [M]. |
| **Attributes:** | **labels\_** : array [n\_samples]  cluster labels for each point  **n\_leaves\_** : int  Number of leaves in the hierarchical tree.  **n\_components\_** : int  The estimated number of connected components in the graph.  **children\_** : array-like, shape (n\_nodes-1, 2)  The children of each non-leaf node. Values less than n\_samples correspond to leaves of the tree which are the original samples. A node i greater than or equal to n\_samples is a non-leaf node and has children children\_[i - n\_samples]. Alternatively at the i-th iteration, children[i][0] and children[i][1] are merged to form node n\_samples + i |

**Methods**

|  |  |
| --- | --- |
| [**fit**](http://scikit-learn.org/stable/modules/generated/sklearn.cluster.AgglomerativeClustering.html#sklearn.cluster.AgglomerativeClustering.fit)(X[, y]) | Fit the hierarchical clustering on the data |
| [**fit\_predict**](http://scikit-learn.org/stable/modules/generated/sklearn.cluster.AgglomerativeClustering.html#sklearn.cluster.AgglomerativeClustering.fit_predict)(X[, y]) | Performs clustering on X and returns cluster labels. |
| [**get\_params**](http://scikit-learn.org/stable/modules/generated/sklearn.cluster.AgglomerativeClustering.html#sklearn.cluster.AgglomerativeClustering.get_params)([deep]) | Get parameters for this estimator. |
| [**set\_params**](http://scikit-learn.org/stable/modules/generated/sklearn.cluster.AgglomerativeClustering.html#sklearn.cluster.AgglomerativeClustering.set_params)(\*\*params) | Set the parameters of this estimator. |

**\_\_init\_\_**(*n\_clusters=2*, *affinity=’euclidean’*, *memory=None*, *connectivity=None*, *compute\_full\_tree=’auto’*, *linkage=’ward’*, *pooling\_func=<function mean>*)[[source]](https://github.com/scikit-learn/scikit-learn/blob/a24c8b46/sklearn/cluster/hierarchical.py#L670)

**fit**(*X*, *y=None*)[[source]](https://github.com/scikit-learn/scikit-learn/blob/a24c8b46/sklearn/cluster/hierarchical.py#L682)

Fit the hierarchical clustering on the data

|  |  |
| --- | --- |
| **Parameters:** | **X** : array-like, shape = [n\_samples, n\_features]  The samples a.k.a. observations.  **y** : Ignored |
| **Returns:** | **self** : |

**fit\_predict**(*X*, *y=None*)[[source]](https://github.com/scikit-learn/scikit-learn/blob/a24c8b46/sklearn/base.py" \l "L395)

Performs clustering on X and returns cluster labels.

|  |  |
| --- | --- |
| **Parameters:** | **X** : ndarray, shape (n\_samples, n\_features)  Input data. |
| **Returns:** | **y** : ndarray, shape (n\_samples,)  cluster labels |

**get\_params**(*deep=True*)[[source]](https://github.com/scikit-learn/scikit-learn/blob/a24c8b46/sklearn/base.py#L213)

Get parameters for this estimator.

|  |  |
| --- | --- |
| **Parameters:** | **deep** : boolean, optional  If True, will return the parameters for this estimator and contained subobjects that are estimators. |
| **Returns:** | **params** : mapping of string to any  Parameter names mapped to their values. |

**set\_params**(*\*\*params*)[[source]](https://github.com/scikit-learn/scikit-learn/blob/a24c8b46/sklearn/base.py#L250)

Set the parameters of this estimator.

The method works on simple estimators as well as on nested objects (such as pipelines). The latter have parameters of the form <component>\_\_<parameter> so that it’s possible to update each component of a nested object.

|  |  |
| --- | --- |
| **Returns:** | **self** : |

import numpy as np

import pandas as pd

from sklearn.preprocessing import StandardScaler

from sklearn.cluster import AgglomerativeClustering

from sklearn.metrics import confusion\_matrix

from sklearn.cross\_validation import train\_test\_split

from sklearn.metrics import accuracy\_score

from sklearn.metrics import classification\_report

balance\_data = pd.read\_csv('https://archive.ics.uci.edu/ml/machine-learning-databases/blood-transfusion/transfusion.data',

sep= ',', header = 0 )

print ("Dataset Lenght: ", len(balance\_data))

print ("Dataset Shape: ", balance\_data.shape)

print ("Dataset: ",balance\_data.head())

X = balance\_data.values[:, 0:5]

X\_train, X\_test = train\_test\_split(

X, test\_size = 0.3, random\_state = 100)

agglo = AgglomerativeClustering(n\_clusters=5,affinity='euclidean').fit\_predict(X\_train)

print(agglo)

#print("Cluster centers")

#print(kmeans.cluster\_centers\_)

#print("Predicting")

#print(X\_test.shape)

#print(agglo.predict(X\_test))

**KNN:**

*class*sklearn.neighbors.**KNeighborsClassifier**(*n\_neighbors=5*, *weights=’uniform’*, *algorithm=’auto’*, *leaf\_size=30*, *p=2*, *metric=’minkowski’*, *metric\_params=None*, *n\_jobs=1*, *\*\*kwargs*)[[source]](https://github.com/scikit-learn/scikit-learn/blob/a24c8b46/sklearn/neighbors/classification.py#L23)

Classifier implementing the k-nearest neighbors vote.

Read more in the [User Guide](http://scikit-learn.org/stable/modules/neighbors.html#classification).

|  |  |
| --- | --- |
| **Parameters:** | **n\_neighbors** : int, optional (default = 5)  Number of neighbors to use by default for **[kneighbors](http://scikit-learn.org/stable/modules/generated/sklearn.neighbors.KNeighborsClassifier.html" \l "sklearn.neighbors.KNeighborsClassifier.kneighbors" \o "sklearn.neighbors.KNeighborsClassifier.kneighbors)** queries.  **weights** : str or callable, optional (default = ‘uniform’)  weight function used in prediction. Possible values:   * ‘uniform’ : uniform weights. All points in each neighborhood are weighted equally. * ‘distance’ : weight points by the inverse of their distance. in this case, closer neighbors of a query point will have a greater influence than neighbors which are further away. * [callable] : a user-defined function which accepts an array of distances, and returns an array of the same shape containing the weights.   **algorithm** : {‘auto’, ‘ball\_tree’, ‘kd\_tree’, ‘brute’}, optional  Algorithm used to compute the nearest neighbors:   * ‘ball\_tree’ will use **[BallTree](http://scikit-learn.org/stable/modules/generated/sklearn.neighbors.BallTree.html" \l "sklearn.neighbors.BallTree" \o "sklearn.neighbors.BallTree)** * ‘kd\_tree’ will use **[KDTree](http://scikit-learn.org/stable/modules/generated/sklearn.neighbors.KDTree.html" \l "sklearn.neighbors.KDTree" \o "sklearn.neighbors.KDTree)** * ‘brute’ will use a brute-force search. * ‘auto’ will attempt to decide the most appropriate algorithm based on the values passed to [**fit**](http://scikit-learn.org/stable/modules/generated/sklearn.neighbors.KNeighborsClassifier.html#sklearn.neighbors.KNeighborsClassifier.fit) method.   Note: fitting on sparse input will override the setting of this parameter, using brute force.  **leaf\_size** : int, optional (default = 30)  Leaf size passed to BallTree or KDTree. This can affect the speed of the construction and query, as well as the memory required to store the tree. The optimal value depends on the nature of the problem.  **p** : integer, optional (default = 2)  Power parameter for the Minkowski metric. When p = 1, this is equivalent to using manhattan\_distance (l1), and euclidean\_distance (l2) for p = 2. For arbitrary p, minkowski\_distance (l\_p) is used.  **metric** : string or callable, default ‘minkowski’  the distance metric to use for the tree. The default metric is minkowski, and with p=2 is equivalent to the standard Euclidean metric. See the documentation of the DistanceMetric class for a list of available metrics.  **metric\_params** : dict, optional (default = None)  Additional keyword arguments for the metric function.  **n\_jobs** : int, optional (default = 1)  The number of parallel jobs to run for neighbors search. If -1, then the number of jobs is set to the number of CPU cores. Doesn’t affect [**fit**](http://scikit-learn.org/stable/modules/generated/sklearn.neighbors.KNeighborsClassifier.html#sklearn.neighbors.KNeighborsClassifier.fit) method. |

import numpy as np

import pandas as pd

from sklearn.neighbors import NearestNeighbors

from sklearn.neighbors import KNeighborsClassifier

from sklearn.preprocessing import StandardScaler

from sklearn.cluster import AgglomerativeClustering

from sklearn.metrics import confusion\_matrix

from sklearn.cross\_validation import train\_test\_split

from sklearn.metrics import accuracy\_score

from sklearn.metrics import classification\_report

balance\_data = pd.read\_csv('https://archive.ics.uci.edu/ml/machine-learning-databases/blood-transfusion/transfusion.data',

sep= ',', header = 0 )

print ("Dataset Lenght: ", len(balance\_data))

print ("Dataset Shape: ", balance\_data.shape)

print ("Dataset: ",balance\_data.head())

X = balance\_data.values[:, 0:4]

Y = balance\_data.values[:, 4]

X\_train, X\_test,Y\_train,Y\_test = train\_test\_split(

X, Y, test\_size = 0.3, random\_state = 100)

#['auto', 'ball\_tree', 'kd\_tree', 'brute']

nbrs = NearestNeighbors(n\_neighbors=2, algorithm='ball\_tree').fit(X\_train)

distances, indices = nbrs.kneighbors(X)

print("distane and index")

print(distances)

print(indices)

print(nbrs.kneighbors\_graph(X).toarray())

neigh = KNeighborsClassifier(n\_neighbors=2)

neighbour=neigh.fit(X\_train,Y\_train)

y\_pred=(neighbour.predict(X\_test))

print(y\_pred)

print(neighbour.predict\_proba(X\_test))

print("Confusion Matrix: ",confusion\_matrix(Y\_test, y\_pred))

print("Accuracy : ",accuracy\_score(Y\_test,y\_pred)\*100)

print("Report : ",classification\_report(Y\_test, y\_pred))

print(neighbour.predict([[1,2,3,99]]))

print(neighbour.predict\_proba([[1,2,3,4]] ))

**KMODES:**

import numpy as np

from kmodes.kmodes import KModes

import numpy as np

import pandas as pd

from sklearn.preprocessing import StandardScaler

from sklearn.cluster import KMeans

from sklearn.metrics import confusion\_matrix

from sklearn.cross\_validation import train\_test\_split

from sklearn.metrics import accuracy\_score

from sklearn.metrics import classification\_report

balance\_data = pd.read\_csv('https://archive.ics.uci.edu/ml/machine-learning-databases/blood-transfusion/transfusion.data',

sep= ',', header = 0 )

print ("Dataset Lenght: ", len(balance\_data))

print ("Dataset Shape: ", balance\_data.shape)

print ("Dataset: ",balance\_data.head())

X = balance\_data.values[:, 0:5]

X\_train, X\_test = train\_test\_split(

X, test\_size = 0.3, random\_state = 100)

kmeans = kmodes.KModes(n\_clusters=5, random\_state=0).fit(X\_train)

print("Cluster centers")

print(kmeans.cluster\_centers\_)

print("Predicting")

print(X\_test.shape)

print(kmeans.predict(X\_test))

# random categorical data

#data = np.random.choice(20, (100, 10))

#m = KModes(n\_clusters=4, init='Huang', n\_init=5, verbose=1)

#clusters = km.fit(data)

#Print the cluster centroids

#print(km.cluster\_centroids\_)

**EM ALGORITHH:**

*class*sklearn.mixture.**GMM**(*n\_components=1*, *covariance\_type='diag'*, *random\_state=None*, *thresh=0.01*, *min\_covar=0.001*)

Gaussian Mixture Model

Representation of a Gaussian mixture model probability distribution. This class allows for easy evaluation of, sampling from, and maximum-likelihood estimation of the parameters of a GMM distribution.

Initializes parameters such that every mixture component has zero mean and identity covariance.

|  |  |
| --- | --- |
| **Parameters :** | **n\_components** : int, optional  Number of mixture components. Defaults to 1.  **covariance\_type** : string (read-only), optional  String describing the type of covariance parameters to use. Must be one of ‘spherical’, ‘tied’, ‘diag’, ‘full’. Defaults to ‘diag’.  **rng** : numpy.random object, optional  Must support the full numpy random number generator API.  **min\_covar** : float, optional  Floor on the diagonal of the covariance matrix to prevent overfitting. Defaults to 1e-3.  **thresh** : float, optional  Convergence threshold. |

|  |  |  |
| --- | --- | --- |
| covariance\_type | string | String describing the type of covariance parameters used by the GMM. Must be one of ‘spherical’, ‘tied’, ‘diag’, ‘full’. |
| *weights\_* | array, shape (*n\_components*,) | Mixing weights for each mixture component. |
| *means\_* | array, shape (*n\_components*, *n\_features*) | Mean parameters for each mixture component. |
| *covars\_* | array | Covariance parameters for each mixture component. The shape depends on *covariance\_type*:  (n\_components,) if 'spherical',  (n\_features, n\_features) if 'tied',  (n\_components, n\_features) if 'diag',  (n\_components, n\_features, n\_features) if 'full' |
| *converged\_* | bool | True when convergence was reached in fit(), False otherwise. |

**Methods**

from sklearn.mixture import GMM

import matplotlib.pyplot as plt

import numpy as np

import numpy as np

import pandas as pd

from sklearn.preprocessing import StandardScaler

from sklearn.cluster import KMeans

from sklearn.metrics import confusion\_matrix

from sklearn.cross\_validation import train\_test\_split

from sklearn.metrics import accuracy\_score

from sklearn.metrics import classification\_report

balance\_data = pd.read\_csv('https://archive.ics.uci.edu/ml/machine-learning-databases/blood-transfusion/transfusion.data',

sep= ',', header = 0 )

print ("Dataset Lenght: ", len(balance\_data))

print ("Dataset Shape: ", balance\_data.shape)

print ("Dataset: ",balance\_data.head())

X = balance\_data.values[:, 0:5]

X\_train, X\_test = train\_test\_split(

X, test\_size = 0.3, random\_state = 100)

gmm = GMM(n\_components=4).fit(X\_train)

labels = gmm.predict(X\_test)

#plt.scatter(X\_train[:, 0], X\_train[:, 1], c=labels, s=40, cmap='viridis');

print(labels)

print(labels.size)

probs = gmm.predict\_proba(X\_test)

print(probs)

print(probs.size)

print(np.round(gmm.weights\_, 2))

print(np.round(gmm.means\_, 2))

print(np.round(gmm.covars\_, 2))

**SCALING:**

import numpy as np

import pandas as pd

from sklearn.preprocessing import StandardScaler

from sklearn.cross\_validation import train\_test\_split

from sklearn.preprocessing import MinMaxScaler

# Python code for binarization

from sklearn.preprocessing import Binarizer

from sklearn.preprocessing import scale

balance\_data = pd.read\_csv('https://archive.ics.uci.edu/ml/machine-learning-databases/blood-transfusion/transfusion.data',

sep= ',', header = 0 )

print ("Dataset Lenght: ", len(balance\_data))

print ("Dataset Shape: ", balance\_data.shape)

print ("Dataset: ",balance\_data.head())

X = balance\_data.values[:, 0:4]

Y = balance\_data.values[:, 4]

scaler = MinMaxScaler(feature\_range=(0, 1))

rescaledX = scaler.fit\_transform(X)

X\_train, X\_test,Y\_train,Y\_test = train\_test\_split(

X, Y, test\_size = 0.3, random\_state = 100)

print(X)

print(rescaledX)

binarizer = Binarizer(threshold=0.0).fit(X)

binaryX = binarizer.transform(X)

print(binaryX)

scaler = StandardScaler().fit(X)

rescaledX = scaler.transform(X)

print(rescaledX)

scaled=scale(X\_train)

print(scaled)

**Missing Data In pandas Dataframes**

20 Dec 2017

**import modules**

**import** **pandas** **as** **pd**

**import** **numpy** **as** **np**

**Create dataframe with missing values**

raw\_data = {'first\_name': ['Jason', np.nan, 'Tina', 'Jake', 'Amy'],

'last\_name': ['Miller', np.nan, 'Ali', 'Milner', 'Cooze'],

'age': [**42**, np.nan, **36**, **24**, **73**],

'sex': ['m', np.nan, 'f', 'm', 'f'],

'preTestScore': [**4**, np.nan, np.nan, **2**, **3**],

'postTestScore': [**25**, np.nan, np.nan, **62**, **70**]}

df = pd.DataFrame(raw\_data, columns = ['first\_name', 'last\_name', 'age', 'sex', 'preTestScore', 'postTestScore'])

df

|  | **first\_name** | **last\_name** | **age** | **sex** | **preTestScore** | **postTestScore** |
| --- | --- | --- | --- | --- | --- | --- |
| **0** | Jason | Miller | 42.0 | m | 4.0 | 25.0 |
| **1** | NaN | NaN | NaN | NaN | NaN | NaN |
| **2** | Tina | Ali | 36.0 | f | NaN | NaN |
| **3** | Jake | Milner | 24.0 | m | 2.0 | 62.0 |
| **4** | Amy | Cooze | 73.0 | f | 3.0 | 70.0 |

**Drop missing observations**

df\_no\_missing = df.dropna()

df\_no\_missing

|  | **first\_name** | **last\_name** | **age** | **sex** | **preTestScore** | **postTestScore** |
| --- | --- | --- | --- | --- | --- | --- |
| **0** | Jason | Miller | 42.0 | m | 4.0 | 25.0 |
| **3** | Jake | Milner | 24.0 | m | 2.0 | 62.0 |
| **4** | Amy | Cooze | 73.0 | f | 3.0 | 70.0 |

**Drop rows where all cells in that row is NA**

df\_cleaned = df.dropna(how='all')

df\_cleaned

|  | **first\_name** | **last\_name** | **age** | **sex** | **preTestScore** | **postTestScore** |
| --- | --- | --- | --- | --- | --- | --- |
| **0** | Jason | Miller | 42.0 | m | 4.0 | 25.0 |
| **2** | Tina | Ali | 36.0 | f | NaN | NaN |
| **3** | Jake | Milner | 24.0 | m | 2.0 | 62.0 |
| **4** | Amy | Cooze | 73.0 | f | 3.0 | 70.0 |

**Create a new column full of missing values**

df['location'] = np.nan

df

|  | **first\_name** | **last\_name** | **age** | **sex** | **preTestScore** | **postTestScore** | **location** |
| --- | --- | --- | --- | --- | --- | --- | --- |
| **0** | Jason | Miller | 42.0 | m | 4.0 | 25.0 | NaN |
| **1** | NaN | NaN | NaN | NaN | NaN | NaN | NaN |
| **2** | Tina | Ali | 36.0 | f | NaN | NaN | NaN |
| **3** | Jake | Milner | 24.0 | m | 2.0 | 62.0 | NaN |
| **4** | Amy | Cooze | 73.0 | f | 3.0 | 70.0 | NaN |

**Drop column if they only contain missing values**

df.dropna(axis=**1**, how='all')

|  | **first\_name** | **last\_name** | **age** | **sex** | **preTestScore** | **postTestScore** |
| --- | --- | --- | --- | --- | --- | --- |
| **0** | Jason | Miller | 42.0 | m | 4.0 | 25.0 |
| **1** | NaN | NaN | NaN | NaN | NaN | NaN |
| **2** | Tina | Ali | 36.0 | f | NaN | NaN |
| **3** | Jake | Milner | 24.0 | m | 2.0 | 62.0 |
| **4** | Amy | Cooze | 73.0 | f | 3.0 | 70.0 |

**Drop rows that contain less than five observations**

This is really mostly useful for time series

df.dropna(thresh=**5**)

|  | **first\_name** | **last\_name** | **age** | **sex** | **preTestScore** | **postTestScore** | **location** |
| --- | --- | --- | --- | --- | --- | --- | --- |
| **0** | Jason | Miller | 42.0 | m | 4.0 | 25.0 | NaN |
| **3** | Jake | Milner | 24.0 | m | 2.0 | 62.0 | NaN |
| **4** | Amy | Cooze | 73.0 | f | 3.0 | 70.0 | NaN |

**Fill in missing data with zeros**

df.fillna(**0**)

|  | **first\_name** | **last\_name** | **age** | **sex** | **preTestScore** | **postTestScore** | **location** |
| --- | --- | --- | --- | --- | --- | --- | --- |
| **0** | Jason | Miller | 42.0 | m | 4.0 | 25.0 | 0.0 |
| **1** | 0 | 0 | 0.0 | 0 | 0.0 | 0.0 | 0.0 |
| **2** | Tina | Ali | 36.0 | f | 0.0 | 0.0 | 0.0 |
| **3** | Jake | Milner | 24.0 | m | 2.0 | 62.0 | 0.0 |
| **4** | Amy | Cooze | 73.0 | f | 3.0 | 70.0 | 0.0 |

**Fill in missing in preTestScore with the mean value of preTestScore**

inplace=True means that the changes are saved to the df right away

df["preTestScore"].fillna(df["preTestScore"].mean(), inplace=True)

df

|  | **first\_name** | **last\_name** | **age** | **sex** | **preTestScore** | **postTestScore** | **location** |
| --- | --- | --- | --- | --- | --- | --- | --- |
| **0** | Jason | Miller | 42.0 | m | 4.0 | 25.0 | NaN |
| **1** | NaN | NaN | NaN | NaN | 3.0 | NaN | NaN |
| **2** | Tina | Ali | 36.0 | f | 3.0 | NaN | NaN |
| **3** | Jake | Milner | 24.0 | m | 2.0 | 62.0 | NaN |
| **4** | Amy | Cooze | 73.0 | f | 3.0 | 70.0 | NaN |

**Fill in missing in postTestScore with each sex’s mean value of postTestScore**

df["postTestScore"].fillna(df.groupby("sex")["postTestScore"].transform("mean"), inplace=True)

df

|  | **first\_name** | **last\_name** | **age** | **sex** | **preTestScore** | **postTestScore** | **location** |
| --- | --- | --- | --- | --- | --- | --- | --- |
| **0** | Jason | Miller | 42.0 | m | 4.0 | 25.0 | NaN |
| **1** | NaN | NaN | NaN | NaN | 3.0 | NaN | NaN |
| **2** | Tina | Ali | 36.0 | f | 3.0 | 70.0 | NaN |
| **3** | Jake | Milner | 24.0 | m | 2.0 | 62.0 | NaN |
| **4** | Amy | Cooze | 73.0 | f | 3.0 | 70.0 | NaN |

**Select some raws but ignore the missing data points**

*# Select the rows of df where age is not NaN and sex is not NaN*

df[df['age'].notnull() & df['sex'].notnull()]

|  | **first\_name** | **last\_name** | **age** | **sex** | **preTestScore** | **postTestScore** | **location** |
| --- | --- | --- | --- | --- | --- | --- | --- |
| **0** | Jason | Miller | 42.0 | m | 4.0 | 25.0 | NaN |
| **2** | Tina | Ali | 36.0 | f | 3.0 | 70.0 | NaN |
| **3** | Jake | Milner | 24.0 | m | 2.0 | 62.0 | NaN |
| **4** | Amy | Cooze | 73.0 | f | 3.0 | 70.0 | NaN |

## INDING THE MISSING DATA

It’s essential to find missing data in your dataset to avoid getting incorrect results from your analysis. The following code shows how you could obtain a listing of missing values without too much effort.

import pandas as pd

import numpy as np

s = pd.Series([1, 2, 3, np.NaN, 5, 6, None])

print s.isnull()

print

print s[s.isnull()]

A dataset could represent missing data in several ways. In this example, you see missing data represented as np.NaN (NumPy Not a Number) and the Python None value.

Use the isnull() method to detect the missing values. The output shows Truewhen the value is missing. By adding an index into the dataset, you obtain just the entries that are missing. The example shows the following output:

0 False

1 False

2 False

3 True

4 False

5 False

6 True

dtype: bool

3 NaN

6 NaN

dtype: float64

## ENCODING MISSINGNESS

After you figure out that your dataset is missing information, you need to consider what to do about it. The three possibilities are to ignore the issue, fill in the missing items, or remove (drop) the missing entries from the dataset. Ignoring the problem could lead to all sorts of problems for your analysis, so it’s the option you use least often. The following example shows one technique for filling in missing data or dropping the errant entries from the dataset:

import pandas as pd

import numpy as np

s = pd.Series([1, 2, 3, np.NaN, 5, 6, None])

print s.fillna(int(s.mean()))

print

print s.dropna()

The two methods of interest are fillna(), which fills in the missing entries, and dropna(), which drops the missing entries. When using fillna(), you must provide a value to use for the missing data. This example uses the mean of all the values, but you could choose a number of other approaches. Here’s the output from this example:

0 1

1 2

2 3

3 3

4 5

5 6

6 3

dtype: float64

0 1

1 2

2 3

4 5

5 6

dtype: float64

Working with a series is straightforward because the dataset is so simple. When working with a DataFrame, however, the problem becomes significantly more complicated. You still have the option of dropping the entire row. When a column is sparsely populated, you might drop the column instead. Filling in the data also becomes more complex because you must consider the dataset as a whole, in addition to the needs of the individual feature.

## IMPUTING MISSING DATA

The previous information hints at the process of imputing missing data (ascribing characteristics based on how the data is used). The technique you use depends on the sort of data you’re working with.

For example, when working with a tree ensemble, you may simply replace missing values with a –1 and rely on the imputer (a transformer algorithm used to complete missing values) to define the best possible value for the missing data. The following example shows a technique you can use to impute missing data values:

import pandas as pd

import numpy as np

from sklearn.preprocessing import Imputer

s = pd.Series([1, 2, 3, np.NaN, 5, 6, None])

imp = Imputer(missing\_values=‘NaN’,

strategy=‘mean’, axis=0)

imp.fit([1, 2, 3, 4, 5, 6, 7])

x = pd.Series(imp.transform(s).tolist()[0])

print x

In this example, s is missing some values. The code creates an Imputer to replace these missing values. The missing\_values parameter defines what to look for, which is NaN. You set the axis parameter to 0 to impute along columns and 1 to impute along rows. The strategy parameter defines how to replace the missing values:

* mean: Replaces the values by using the mean along the axis
* median: Replaces the values by using the medium along the axis
* most\_frequent: Replaces the values by using the most frequent value along the axis

Before you can impute anything, you must provide statistics for the Imputer to use by calling fit(). The code then calls transform() on s to fill in the missing values. However, the output is no longer a series. To create a series, you must convert the Imputer output to a list and use the resulting list as input to Series(). Here’s the result of the process with the missing values filled in:

0 1

1 2

2 3

3 4

4 5

5 6

6 7

dtype: float64

print(data.isnull().sum())

Most libraries (including scikit-learn) will give you an error if you try to build a model using data with missing values. So you'll need to choose one of the strategies below.

## Solutions

## 1) A Simple Option: Drop Columns with Missing Values

If your data is in a DataFrame called original\_data, you can drop columns with missing values. One way to do that is

data\_without\_missing\_values = original\_data.dropna(axis=1)

In many cases, you'll have both a training dataset and a test dataset. You will want to drop the same columns in both DataFrames. In that case, you would write

cols\_with\_missing = [col for col in original\_data.columns

if original\_data[col].isnull().any()]

redued\_original\_data = original\_data.drop(cols\_with\_missing, axis=1)

reduced\_test\_data = test\_data.drop(cols\_with\_missing, axis=1)

If those columns had useful information (in the places that were not missing), your model loses access to this information when the column is dropped. Also, if your test data has missing values in places where your training data did not, this will result in an error.

So, it's somewhat usually not the best solution. However, it can be useful when most values in a column are missing.

## 2) A Better Option: Imputation

Imputation fills in the missing value with some number. The imputed value won't be exactly right in most cases, but it usually gives more accurate models than dropping the column entirely.

This is done with

from sklearn.preprocessing import Imputer

my\_imputer = Imputer()

data\_with\_imputed\_values = my\_imputer.fit\_transform(original\_data)

The default behavior fills in the mean value for imputation. Statisticians have researched more complex strategies, but those complex strategies typically give no benefit once you plug the results into sophisticated machine learning models.

One (of many) nice things about Imputation is that it can be included in a scikit-learn Pipeline. Pipelines simplify model building, model validation and model deployment.

## 3) An Extension To Imputation

Imputation is the standard approach, and it usually works well. However, imputed values may by systematically above or below their actual values (which weren't collected in the dataset). Or rows with missing values may be unique in some other way. In that case, your model would make better predictions by considering which values were originally missing. Here's how it might look:

# make copy to avoid changing original data (when Imputing)

new\_data = original\_data.copy()

# make new columns indicating what will be imputed

cols\_with\_missing = (col for col in new\_data.columns

if new\_data[c].isnull().any())

for col in cols\_with\_missing:

new\_data[col + '\_was\_missing'] = new\_data[col].isnull()

# Imputation

my\_imputer = Imputer()

new\_data = my\_imputer.fit\_transform(new\_data)

In some cases this approach will meaningfully improve results. In other cases, it doesn't help at all.

# Example (Comparing All Solutions)

We will see am example predicting housing prices from the Melbourne Housing data. To master missing value handling, fork this notebook and repeat the same steps with the Iowa Housing data. Find information about both in the **Data** section of the header menu.

### Basic Problem Set-up

In [1]:

import pandas as pd

*# Load data*

melb\_data = pd.read\_csv('../input/melbourne-housing-snapshot/melb\_data.csv')

from sklearn.ensemble import RandomForestRegressor

from sklearn.metrics import mean\_absolute\_error

from sklearn.model\_selection import train\_test\_split

melb\_target = melb\_data.Price

melb\_predictors = melb\_data.drop(['Price'], axis=1)

*# For the sake of keeping the example simple, we'll use only numeric predictors.*

melb\_numeric\_predictors = melb\_predictors.select\_dtypes(exclude=['object'])

### Create Function to Measure Quality of An Approach

We divide our data into **training** and **test**. If the reason for this is unfamiliar, review [Welcome to Data Science](https://www.kaggle.com/dansbecker/welcome-to-data-science-1).

We've loaded a function score\_dataset(X\_train, X\_test, y\_train, y\_test) to compare the quality of diffrent approaches to missing values. This function reports the out-of-sample MAE score from a RandomForest.

### Get Model Score from Dropping Columns with Missing Values

In [3]:

cols\_with\_missing = [col for col **in** X\_train.columns

if X\_train[col].isnull().any()]

reduced\_X\_train = X\_train.drop(cols\_with\_missing, axis=1)

reduced\_X\_test = X\_test.drop(cols\_with\_missing, axis=1)

print("Mean Absolute Error from dropping columns with Missing Values:")

print(score\_dataset(reduced\_X\_train, reduced\_X\_test, y\_train, y\_test))

>>> balamce\_data.median()

Traceback (most recent call last):

File "<pyshell#23>", line 1, in <module>

balamce\_data.median()

NameError: name 'balamce\_data' is not defined

>>> balance\_data['Frequency (times)'].mean()

5.514705882352941

>>>

data\_name.isnull().sum()

This will tell us the total number of NaN in or data.

If the missing value isn’t identified as NaN , then we have to first convert or replace such non NaN entry with a NaN.

data\_name[‘column\_name’].replace(0, np.nan, inplace= True)

This will replace values of zero with NaN in the column named column\_name of our data\_name .

#### ****1) DROPPING NULL OR MISSING VALUES****

This is the fastest and easiest step to handle missing values. However, it is not generally advised. This method reduces the quality of our model as it reduces sample size because it works by deleting all other observations where any of the variable is missing. The process can be done by:

data\_name.dropna()

#### ****2) FILLING MISSING VALUES****

This is the most common method of handling missing values. This is a process whereby missing values are replaced with a test statistic like mean, median or mode of the particular feature the missing value belongs to. One can also specify a forward-fill or back-fill to propagate the next values backward or previous value forward.

Filling missing values with a test statistic

#Age is a column name for our train data

mean\_value=train['Age'].mean()  
train['Age']=train['Age'].fillna(mean\_value)

#this will replace all NaN values with the mean of the non null values

#For Median

meadian\_value=train['Age'].median()  
train['Age']=train['Age'].fillna(median\_value)

Alternative way of filling missing value with test statistic is by using our Imputer method found in sklearn.preprocessing.

In [1]: from sklearn.preprocessing import Imputer  
In [2]: imp = Imputer(missing\_values='NaN', strategy='mean', axis=0)  
In [3]: imp.fit(train)  
In [4]: train= imp.transform(train)  
#This will look for all columns where we have NaN value and replace the NaN value with specified test statistic.

#for mode we specify strategy='most\_frequent'

*For Back-fill or forward-fill to propagate next or previous values respectively:*

#for back fill

train.fillna(method='bfill')  
#for forward-fill

train.fillna(method=''ffill)

#one can also specify an axis to propagate (1 is for rows and 0 is for columns)

train.fillna(method='bfill', axis=1)

Please note that if a previous or next value isn’t available or rather if it is also a NaN value, then, the NaN remains even after back-filling or forward-filling.

Also, the disadvantage of using mean is that the mean is greatly affected by outliers in our data. As a result, if outliers are present in our data, then median will be the best out of the box tool to use.