**Feature Extraction and Engineering:**

* Objective: insert knowledge from industry by creating new variables (columns, features) from existing variables.
* However avoid **overfitting** by too many features: we don’t want to have a perfect score on the train dataset by overfitting, we want to generalise.
* **ETL Types:**
  + Remove outliers
  + Scaling to have all feature values within similar ranges
  + Rounding
  + Standardisation/normalisation:  apply to columns where the range of variation is higher than other columns; adjust them to bring down to the level of other columns
  + One Hot Encoding  :

  y = pd.get\_dummies(data=df\_classify['state'], drop\_first=True)

* + Shuffling (remove order from the data)
* **Feature selection:**
  + Intuitively remove variables with no meaning or useful information for model
  + Automatically reduce & add features:
    - **Forward Greedy Feature Selection:**adding a variable and testing the model; observe the accuracy, and stop at the moment when the increase is not significant anymore
    - **Backward Elimination:** same as above, but by eliminating features
  + Removing one of two highly correlated features:
    - import seaborn as sns

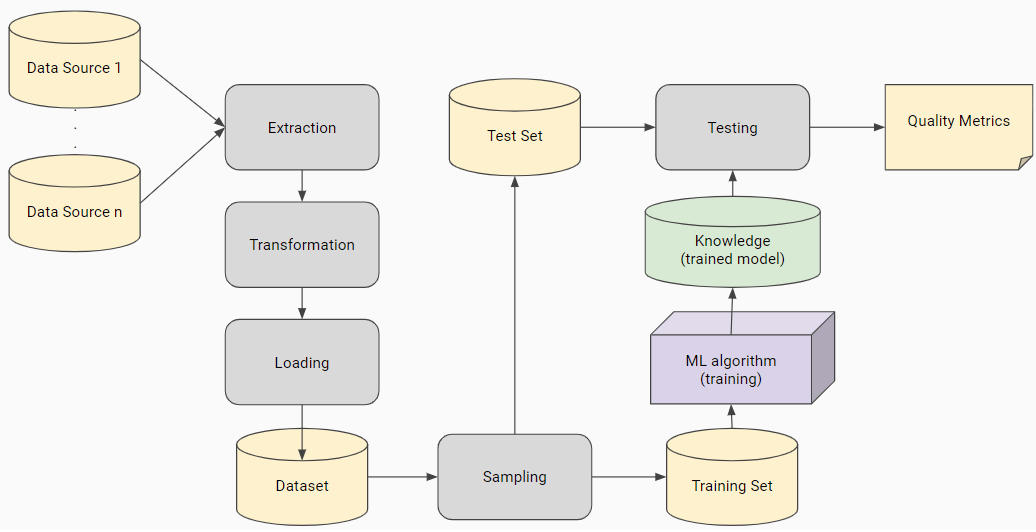
sns.heatmap(df.corr())

* + **PCA,** dimension reduction:
    - Idea: looking at the axis and identify where is the most variance, keep these axis as most representative
    - Risk of loosing information (loose of possibility to interpret data easily)
    - Each time when a choice is made to use PCA on certain variables, add features, remove features; test it on the dev data to mesure the impact on the objective

*!! In* real *life, all the data cleaning and transformation steps are performed separately, first on train dataset, and then on test dataset.*

**SUPERVISED MACHINE LEARNING:**

* **Workflow:**



* **2 types of models:**
  + **Linear Regression:**for continuous y
    - from sklearn import linear\_model

model = linear\_model.LinearRegression()

 model.fit(X\_train, y\_train)

* + **Classification:** for finished (discrete) y
    - **Logistic Regression:**
      * Goal is to separate two classes with a line drawn between the two (or many, but using one class against all other and doing it for each class)
      * from sklearn.linear\_model import LogisticRegression

model = LogisticRegression()

 model.fit(X\_train, y\_train)

 model.score(X\_train, y\_train)

* + - **SVM (Support Vector Machine):**
      * Same goal as above, but the line is drawn in horizontal
      * Looking to have the maximum distance between the furthest point from closest points of both classes
      * from sklearn.svm import SVC

 model = SVC(gamma=‘auto’)

 model.fit(X\_train, y\_train)

* + - **Naive Bayes:**
      * Model is looking at a probability distribution table given the historical data to predict future data.
      * Model assumes that the columns are independent one from another to predict the value we’re looking for
      * Model without hyper parameters
    - **Gaussian Naive Bayes (subtype of above; other subtypes: Multinominal NB, Bernouilli NB etc):**
      * Same methodology as above, but model makes the assumption that likelihood comes from a Gaussian (or normal) distribution.
      * from sklearn.naive\_bayes import GaussianNB

model = GaussianNB()

model.fit(X\_train, y\_train)

 y\_pred = model.predict(X\_test)

* + - **K-Nearest Neighbors:**
      * No learning involved
      * Takes the points within the closest distance and looks at the dominant class and chooses it as the result value
      * This algorithm is based on the idea that observations in a "neighorbood" will have the same classification.
      * Neighbours are determined by distance. We look at the labels of all the observations in the "neighborhood" and assign the most common label (the model) to the observation that we are trying to predict.
      * Number of neighbours k is defined by user. Test different models with multiple values of k.
      * from sklearn.neighbors import KNeighborsClassifier

model = KNeighborsClassifier(n\_neighbors=3)

 model.fit(X\_train, y\_train)

 y\_pred = model.predict(X\_test)

* + - **Decision Trees:**
      * Predicts the value of a target variable by learning simple decision rules inferred from the data features.
      * Model proceeds by step: it looks if a feature is contained in an interval or value -> yes/no -> then goes to next feature -> yes/no etc. until it reaches the end of features
      * The risk is that the model learns by heart on the train data and is never wrong, but has bad performance on test data (the model is struggling to generalise)
      * from sklearn.tree import DecisionTreeClassifier

 model = DecisionTreeClassifier(random state = 0)

 model.fit(X\_train, y\_train)

* + - **Random Forest:**
      * A collection of decision trees
      * Is made of lot of simple models with differently shuffled data each time
      * Select randomly a certain number of columns for each decision tree (so they all don’t work on the same columns)
      * Then select the most popular prediction
      * For each model there are hyper-parameters that can be adjusted to reach the result we are looking for (depth of decision tree etc)
      * **Gradient Boosting, XG Boost:** a variety of Random Forest model
      * from sklearn.ensemble import RandomForestClassifier

model = RandomForestClassifier(n\_estimators=100, max\_depth=2, random state = 0)

 model.fit(X\_train, y\_train) 

*!!! In real life it’s always better to test as much models (algorithms) as possible to see which one gives the best score.*

* **Splitting data:**
  + - * from sklearn.model\_selection import train\_test\_split

x\_columns = [col for col in df.columns.values if col != “Target"]

 X = df[x\_columns]

y = df[“Target"]

 X\_train, X\_test, y\_train, y\_test = train\_test\_split(X, y, train\_size = 0.8, test\_size=0.2)

* **Steps for the model:**
  + First call (initiate) the model before it starts learning:
    - linreg = LinearRegression()
  + Training the model on the train data set:
    - linreg.fit(X\_train, y\_train)
  + Predicting the data with this model:
    - y\_dev\_pred = linreg.predict(X\_dev)
  + Checking if the model is good:
    - R2score = (y\_dev, y\_dev\_pred)
  + Each time we selet the model that is the best on the dev (validation) data set:
    - y\_test\_pred.predict(X\_test)
  + Give the closest score to the score that we will have in production:
    - R2score = (y\_test, y\_test\_pred)
* **Model fitting syntax:**
  + x\_columns = [col for col in df.columns.values if col != “Target"]

x = df[x\_columns]

y = df[“Target"]

model = LogisticRegression()

 model.fit(x, y)

model.score(x, y)

* + Score returns the coefficient of determination or r squared. This number tells us what proportion of the variation in the data is explained by the model
* **Cross Validation:**
  + There is a probability that train-test split tests model on the only subset where it performs well, thus providing an unreliable quality metric. To reduce this probability, you can test on many different test sets and compute an average of the individual quality metrics obtained:

1. Randomly partition the dataset in *n* bins.
2. For every bin ( *b*):

* Train with the remaining ( *n-1*) bins.
* Test with *b* and obtain quality metric.

1. Output the average of the *n* quality metrics obtained.
   * from sklearn.model\_selection import cross\_val\_score

cross\_val\_score(model, X, y, cv=10)

*\* model = any chosen model (LR, SVM etc)*

* **Train-Validation-Test Split:**
  + Randomly partition the dataset in 3 disjoint subsets (training, validation, and test sets).
  + Initialize the hyper-parameters
  + Train with the train set.
  + Evaluate on validation set.
  + Update best performing hyper-parameters.
  + Re-compute/modify the hyper-parameters while maximum number of iterations not reached.
  + Using the best performing hyper-parameters, compute the definitive quality metric on the test set.

**Model evaluation:**

**Evaluating Regression Models:**

* **R-Squared (Coefficient of Determination):**
  + Most commonly used metric in basic linear regression problems (it is the default metric for the score function we in the LinearRegression model )
  + Looking at how large the errors are compared to the variation of the values in the function itself.
  + Useful metric because it provides a fairly robust measure of how well the model predicts the response variable.
  + from sklearn.metrics import r2\_score

score = r2\_score(actual\_values, predictions)

 score = r2\_score(y\_test, y\_predict)

* **Mean Squared Error (MSE):**
  + It takes the error between the predicted and actual value, squares it, then takes the average (mean).
  + from sklearn.metrics import mean\_squared\_error

score = mean\_squared\_error(actual\_values, predictions)

* + **Note:** Another common variation on MSE is Root Mean Squared Error (RMSE), which simply takes the square root of MSE.
  + Value is higher or equal to 0, 0 meaning that there’s no error
* **Mean Absolute Error:**
  + Average (mean) of the absolute value (positive value: -2 absolute value = 2, 2 absolute value is 2) of the errors between predicted and actual values.
  + This error metric is similar to MAE, but is less affected by individual large outlier errors. This may be more representative in cases where outliers are not hugely important.
  + from sklearn.metrics import mean\_absolute\_error

score = mean\_absolute\_error(actual\_values, predictions)

**Evaluating Classification Models:**

* **Accuracy Score:**
  + Calculates the ratio of correct to incorrect predictions.
  + Number of times this class was predicted divided by total number of predictions
  + Doesn’t work well if the classes are not represented equally and one class is less represented than others.
  + from sklearn.metrics import accuracy\_score

score = accuracy\_score(actual\_values, predictions)

* **Balanced Accuracy Score:**
  + To combat accuracy issues with unbalanced datasets, we can calculate a weighted accuracy.
  + Average of accuracies for all classes divided by all classes
  + from sklearn.metrics import balanced\_accuracy\_score

score = balanced\_accuracy\_score(actual\_values, predictions)

* **Precision vs. Recall:**
  + Precision = Ratio of prediction of the true positives compared to the number of all the predictions of that class
    - Number of true positives / (number of true positives + false positives)
  + Recall = Ratio of prediction of the true positives compared to the number of all positives
    - True positives / (True positives + False negatives)
  + F1 = average of Predict & Recall:
    - F1 =  2 x Precision x Recall / (Precision + Recall)
  + from sklearn.metrics import precision\_score, recall\_score, f1\_score

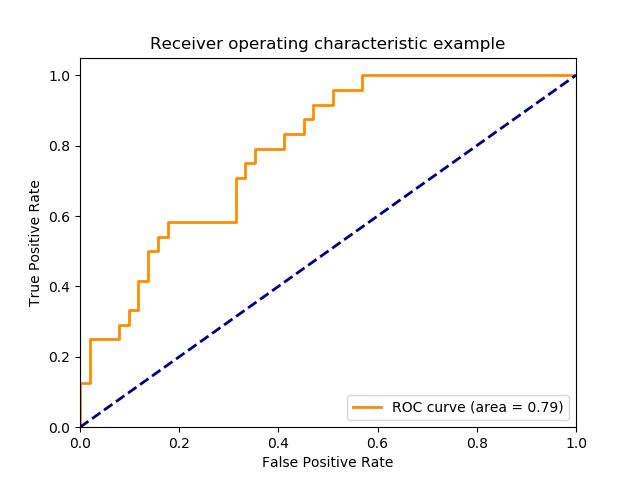
precision = precision\_score(actual\_values, predictions)

recall = recall\_score(actual\_values, predictions)

f1\_score = f1\_score(actual\_values, predictions)

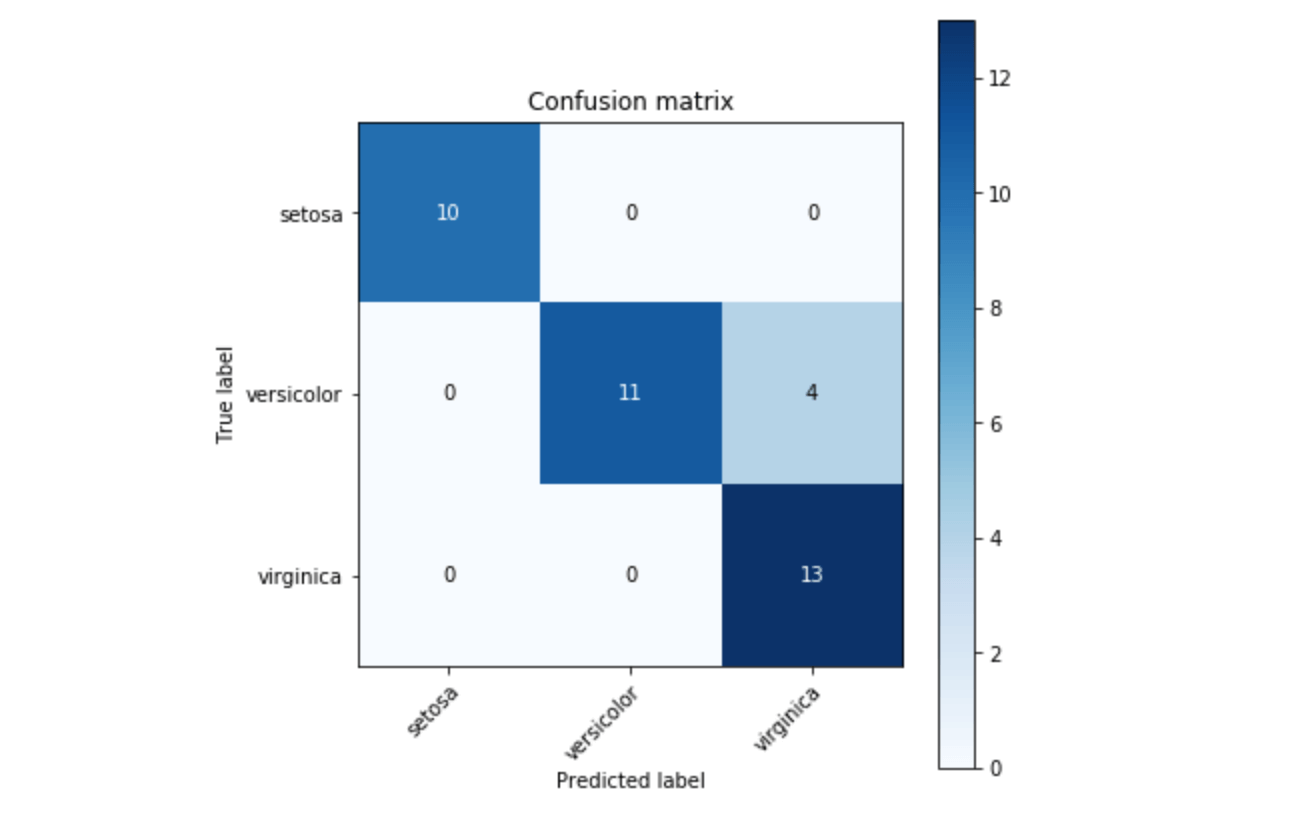
* **ROC** **(Receiving Operator Characteristic) Curve:**
  + Created by varying the classification threshold and plotting the fraction of true positives out of all positive cases (true positive rate, or TPR) vs. false positives out of all negative cases (false positive rate, or FPR) at each threshold value.
  + Can summarize the ROC curve by calculating the area under the curve (AUC). A model with a higher ROC AUC score will generally be more accurate at a given sensitivity level. The better the classification algorithm, the larger the area under the curve (AUC).
  + from sklearn.metrics import roc\_curve, roc\_auc\_score

fpr, tpr, thresholds = roc\_curve(y, scores)

roc\_auc\_score(y\_true, y\_scores) 

* + Area under curve:
    - auc

0.79

* **Confusion Matrix:**
  + The confusion matrix specifies how many observations were correctly classified and how many were incorrectly classified.
  + The diagonal of this matrix represents true predictions - that is, predicted\_class that match actual\_class. Any off-diagonal members of this matrix are incorrect predictions.
  + In the example above, the classifier is mostly correct (most of the values are on the diagonal), but it has some issues differentiating between the latter 2 classes, as shown by the 4 in the off diagonal cell.
  + from sklearn.metrics import confusion\_matrix

actual\_values = [2, 0, 2, 2, 0, 1]

predictions = [0, 0, 2, 2, 0, 2]

confusion\_matrix(actual\_values, predictions)

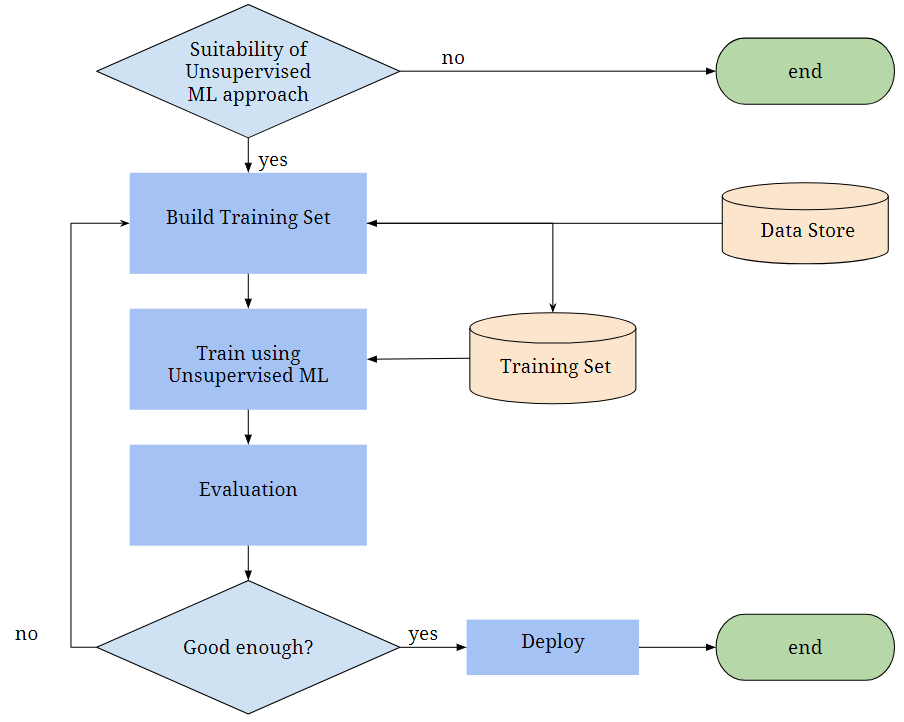
array([[2, 0, 0],

[0, 0, 1],

[1, 0, 2]])

**UNSUPERVISED MACHINE LEARNING:**

* **Workflow:**



**Types:**

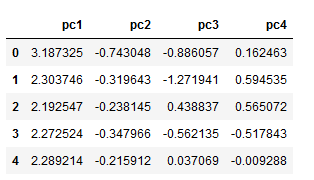
* **PCA, dimension reduction:**
  + PCA projects the data onto a lower dimensional subspace. The new data will typically reduce the dimensions of original data and will therefore, contain less variables. The first dimension will explain the most amount of variation in the data and subsequent components will explain less and less variation. This transformation will provide a smaller amount of continuous variables that can be clustered more effectively.
  + from sklearn.decomposition import PCA

pca = PCA(n\_components=4)

principalComponents = pca.fit\_transform(df)

principalDf = pd.DataFrame(data = principalComponents ,columns = ['pc1', 'pc2', 'pc3', 'pc4'])

 principalDf.head()



* + **Clustering:**a family of algorithms for uncovering relationships and insight in a dataset. The data is not labeled and so there is no ground truth answer that we are trying to predict. Instead, we use different algorithms to group observations together and uncover what they might have in common.
  + **K-Means Clustering:**
    - Most popular clustering techniques. We choose how many clusters we would like to create (k).
    - Model takes the dataset and choses the centroids of each cluster and measures all other data points to see which centroid they are closest to
    - Centroids are determined by using inertie (calculating the sum of the square distances of each point and finding the smallest value, meaning that this is then the centroid).
    - Then model selects random starting points for those cluster centroids. It computes the distance between each observation and the clusters. It reassigns a cluster to each observation and then re-computes the centroids. It keeps doing so until the labels stay constant and we no longer need to reassign.
    - The limitation of this model is that it’s not good for certain datasets (when the classes are divided in a moon shape distribution, the model will not manage to detect this correctly).
    - from sklearn.cluster import KMeans

model = KMeans(n\_clusters=4)

clusters = model.fit(df)

 df['Clusters'] = clusters.labels\_

* + **Hierarchical Clustering (Agglomerative Clustering or Ward):**
    - Create a hierarchy of clusters. The advantage: no need of predetermined number of clusters.
    - Types:
      * **Agglomerative:** bottom up approach. Starts off with a cluster for each observation and then combines similar clusters until left with only one large cluster.
      * **Divisive:**top down approach. Starts with one large cluster and keep dividing until left with no clusters.
    - Linkage: hyper-parameter that defines how a distance between clusters is calculated (by min value, by max value etc)
    - Hierarchical clustering with scikit-learn is performed using the AgglomerativeClustering function:
    - from sklearn.cluster import AgglomerativeClustering

data\_sample = df.sample(n=100)

hier\_clust = AgglomerativeClustering(linkage='ward')

data\_hier = hier\_clust.fit(data\_sample)

* + - **Dendrogram:** visualization that displays the relationship between observations in the data:
    - import matplotlib.pyplot as plt

from scipy.cluster import hierarchy

 def plot\_dendrogram(model, \*\*kwargs):

# Children of hierarchical clustering

children = model.children\_

 # Distances between each pair of children  # Since we don't have this information, we can use a uniform one for plotting

  distance = np.arange(children.shape[0])

 # The number of observations contained in each cluster level

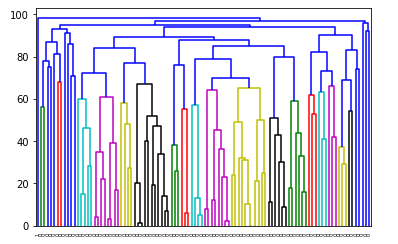
no\_of\_observations = np.arange(2, children.shape[0]+2)

 # Create linkage matrix and then plot the dendrogram  linkage\_matrix = np.column\_stack([children, distance, no\_of\_observations]).astype(float)

 # Plot the corresponding dendrogram

 hierarchy.dendrogram(linkage\_matrix, \*\*kwargs)

plot\_dendrogram(census\_hier, labels=census\_hier.labels\_)



* + **DBSCAN (Density-Based Spatial Clustering of Applications with Noise):**
    - Used in the limitation cases of K-Means; no need to chose cluster number
    - It clusters points based on how close points are to each other. It creates clusters from points that are packed together and treats points that are far apart as outliers.
    - Identifies clusters by finding zones of high density of points delimited by zones of low density
    - Hyperparameters to input:
      * ε (epsilon): size of neighbouring
      * n = minimum number of points in the neighbouring values
    - Core points: defined by points that have at least n number of other points in the neighbouring values (of epsilon size)
    - Directly reachable points: points that are not core points, but are in epsilon distance from core point
    - Reachable points: all points that are in epsilon distance from directly reachable points or other reachable points
    - Noise points (outliers)
    - Algorithm will search core points; if two points are attached, the clusters are fusionend (a cluster can contain several core points); then reachable points are being attached to their closest cluster
    - Disadvantage: it cannot handle scenarios where the density of all points is uniform.
    - 2 step code:
      * First make moons:
      * import matplotlib.pyplot as plt

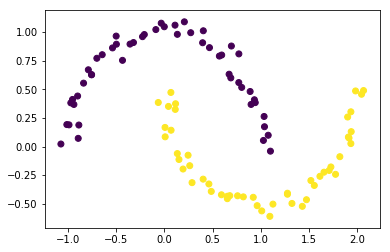
from sklearn.datasets import make\_moons

moons = make\_moons(n\_samples=100, shuffle=True, noise=0.05)

* + - * Second apply DBSCAN:
      * from sklearn.cluster import DBSCAN

dbscan = DBSCAN(eps=0.3).fit(moons[0])

plt.scatter(moons[0][:,0], moons[0][:,1], c=dbscan.labels\_)



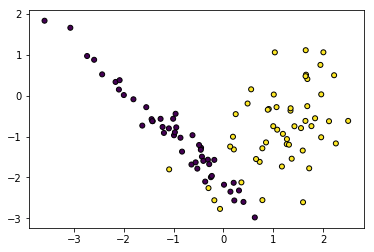
* + **Gausian Mixture Models:**
    - An algorithm allowing to do the clustering and detecting outliers
    - Is at the same time a clustering and probability density estimation algorithm
    - This algorithm is making an hypothesis that the dataset has been generated by a combination of mix of gausians (bell curves). It describes each cluster by its centroid and uses this centroid to generate a normal distribution with the centroid as the mean.
    - Can handle scenarios where clusters have overlapping points.
    - 2 steps:
      * First make classification:
      * from sklearn.datasets import make\_classification

X = make*\_classification(n\_*features=2, n*\_redundant=0, n\_*informative=2, n\_clusters\_per\_class=1)

* + - * Second apply Gausian Mixture :
      * from sklearn import mixture

gmm = mixture.GaussianMixture(n*\_components=2, covariance\_*type='full').fit(X[0])

 plt.scatter(X[0][:, 0], X[0][:, 1], marker='o', c=gmm.predict(X[0]), s=25, edgecolor='k’)



* + **Probability density estimation:**
  + Finds the random process that generated the data
  + **Model evaluation:**
  + **Mathematical evaluation metrics:**
    - **Elbow Curve (Optimal Number of Clusters):**
      * Calculate the inertia for different numbers of clusters ; the bigger the number of clusters the more the inertia will decrease; however at one point it will start decreasing less significantly: this is called the **Elbow Curve;** the point of the “elbow” is the optimal number of clusters.
      * However it’s sometimes difficult to distinguish well where the “elbow” occurs.
      * from yellowbrick.cluster import KElbowVisualizer

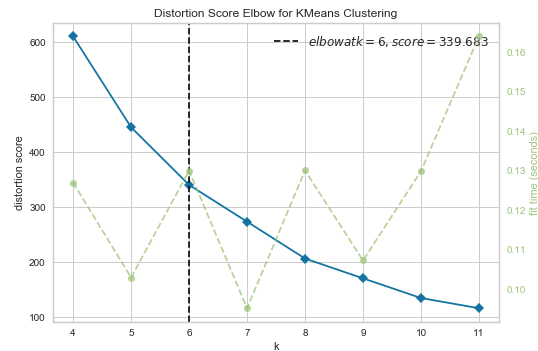
from sklearn.cluster import KMeans

model = KMeans()

visualizer = KElbowVisualizer(model, k=(4,12))

visualizer.fit(X)

visualizer.poof()



* + - **Silhouette Score:**
      * It evaluates a model based on how well defined the cluster centers are.
      * Compares the averages between the points in the cluster and the points in neighbouring cluster.
      * from sklearn.metrics import silhouette\_score

score = silhouette\_score(X, labels, metric='euclidean’)

* + - * metric argument is the distance metric to use. Euclidean distance is the simplest, but it does not work well for high dimensional data.
    - **Adjusted Rand Score:**
      * When we have a sample set with labels already, we can compare the labels that model has predicted with the existing sample and tell how much they are alike
      * It ranges from [-1, 1], with positive being better.
      * from sklearn.metrics import adjusted\_rand\_score

score = adjusted\_rand\_score(true\_labels, predicted\_labels)

* + **Visualisation:**
    - **Plotting results:**
      * Select specific features (columns) and plot them in 2D scatter plot by colouring the clusters by their labels
      * plt.scatter(X[:, 0], X[:, 1], c=y\_pred)
    - **PCA**
      * It attempts to find the orthogonal set of vectors that account for as much of the variability of the dataset in as few components as possible.
      * The first principal component will encompass as much of the dataset variation as possible in 1 dimension, the second component will encompass as much as possible of the remaining variation as possible while remaining orthogonal to the first, and so on.
      * from sklearn.decomposition import PCA

pca = PCA()

 pca.fit(X)

pca\_X = pca.transform(X)

 fig = plt.figure(figsize=(6, 6))

ax = [plt.subplot2grid((6, 1), (0, 0), rowspan=4), plt.subplot2grid((6, 1), (5, 0))]

ax[0].scatter(pca\_X[:, 0], pca\_X[:, 1])

ax[0].set\_xlabel("PCA 1”)

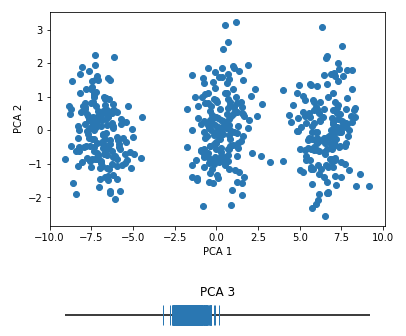
ax[0].set\_ylabel("PCA 2”)

ax[1].set\_title('PCA 3', pad=-5)

 ax[1].hlines(1, -7, 10)

y = np.ones(X.shape[0])

ax[1].plot(pca\_X[:, 2], y, '|', ms='20’)

ax[1].axis('off’)  plt.show() 

* + - **t-SNE (t*-distributed Stochastic Neighbor Embedding*):**
      * t-SNE uses a [complicated algorithm](https://en.wikipedia.org/wiki/T-distributed_stochastic_neighbor_embedding) to come up with a 2D representation of data that allows insightful visualization of high dimensional data.
      * Slow algorithm, won't use on the entire dataset & won’t use to create new features.
      * It exaggerates the distances between points (close points will be closer, far points will be further away).
      * Adjust perplexity metric as hyper-parameter to improve the algorithm.
      * from sklearn.manifold import TSNE

X, y = datasets.load\_iris(return\_X\_y=True)

fig, ax = plt.subplots(1, 4, figsize=(15, 7))

 for i, perp in enumerate([5, 30, 50, 100]):

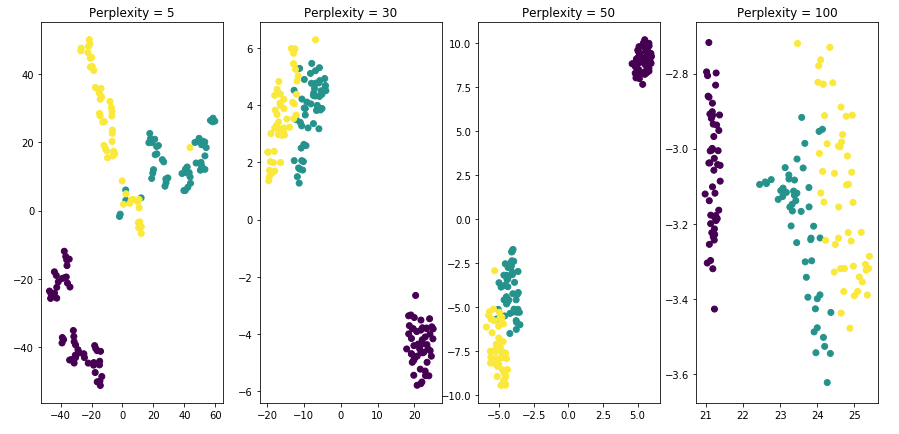
tsne = TSNE(perplexity=perp)

x\_embedded = tsne.fit\_transform(X)

ax[i].scatter(x\_embedded[:, 0], x\_embedded[:, 1], c=y)

ax[i].set\_title("Perplexity = {}".format(perp))

 plt.show()



* + - * The output can be used to visually explore data (to get a sense of the appropriate number of clusters in a high dimensional space), or as the input into another algorithm directly.
      * Clustering algorithm could be applied directly to the output of t-SNE to linearly separate non-linear clusters in the data it found and build an accurate or useful unsupervised learning pipeline.
  + **Feature Generation for Supervised Learning:**
    - Often used to fight the [curse of dimensionality](https://en.wikipedia.org/wiki/Curse_of_dimensionality), the phenomena whereby the feature space of a dataset increases so fast that the dataset quickly becomes extremely sparse, creating problems for most statistical methods. A solution: first apply an unsupervised dimensionality reduction model on the dataset, then use the resulting smaller set of features in the classifier/regressor.
    - Or to create more interpretable models; if a classifier is built using clusters as features, the relative importance of these clusters as well as the membership of the clusters can be used to gain insight about how the model is making its predictions.
    - When creating features for a supervised learning model, will use the performance of the derived supervised model to evaluate the quality of the unsupervised model.
    - For the unsupervised model to be useful, it should improve some aspect of performance on the supervised model, so user should be able to use any of the supervised learning model evaluation techniques to compare results with a baseline supervised model that doesn't use the unsupervised-derived features.

!!! To validate a clustering, also use the **knowledge of the industry**; check that the clusters do make sense just looking at the results.