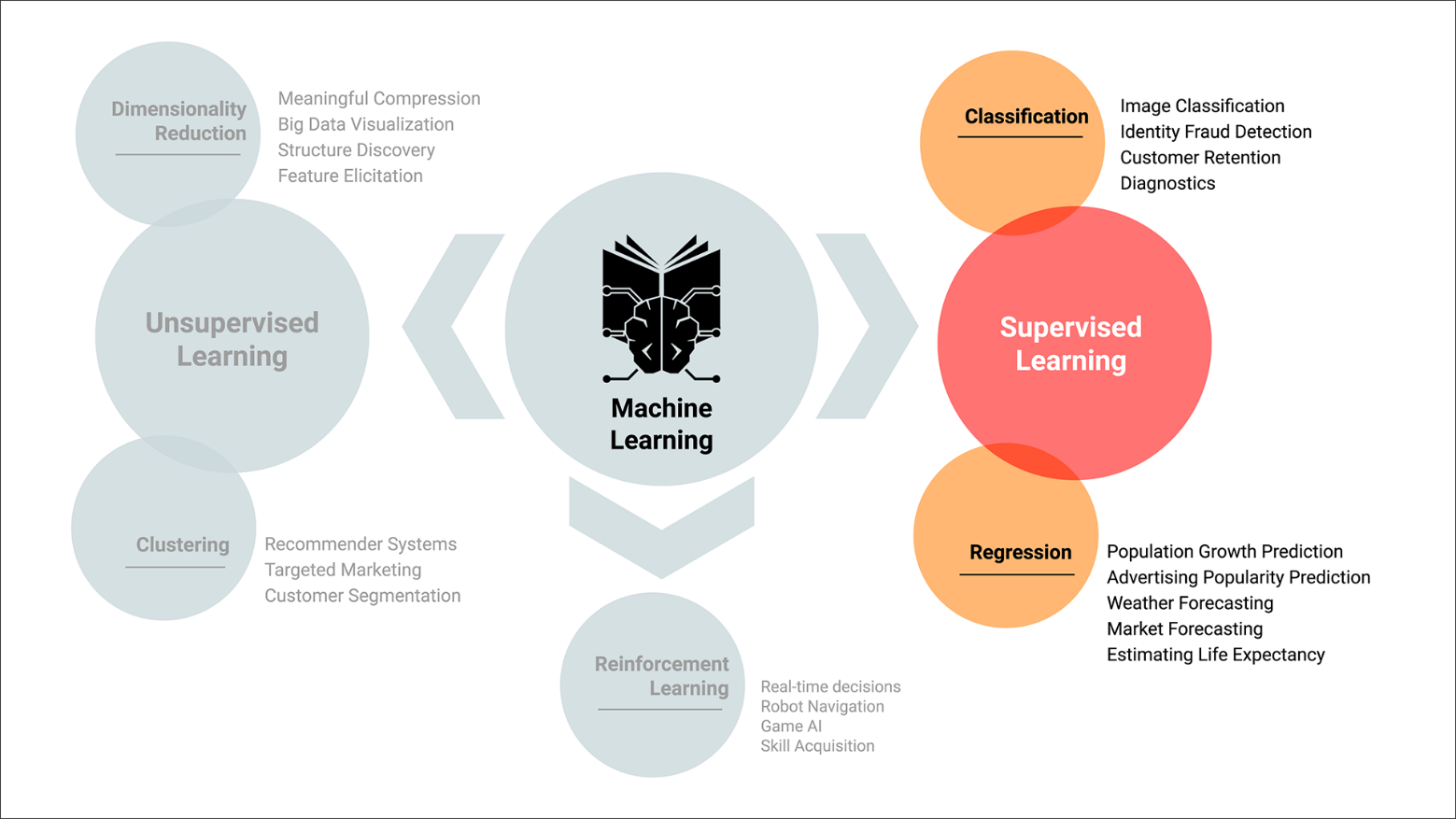
**Supervised Machine Learning**

Theory and Concepts:

Diagram of types of machine learning and their applications:



**Types of Supervised Machine Learning:**

*Regression*: Used to predict continuous variables. A model generated by an algorithm would be used to predict a conclusion and the algorithm would attempt to learn patterns that exist among the data from factors considered relevant to the desired conclusion.

Examples of continuous variables: (a person’s weight and factors of height, dietary data, exercise patterns…)

*Consider the question posed for finding the model’s answer as “over a period of time, what would…”*

*Classification*: Used to predict discrete outcomes and the model’s algorithm would attempt to learn patterns from the data presented to gain the ability to predict accurate conclusions.

Examples of discrete variables: (a person’s voting selection based on factors of age, sex, income, geographic location…)

*Consider the question posed as “based on the data, what is the answer to…”*

Both cases have the dataset divided into ***Features*** and ***Target***, where:

*Features*: the variables used to make a prediction

*Target*: the predicted outcome

**Key Commands and Actions**:

*Reference Format:* DataFrame with columns indicated as - df.column\_1, df.column\_2 etc…,

*Values property:* df.column\_name.values 🡪 used to select the values in the column

*Reshape Method:* df.column\_name.values.reshape(-1, 1)

Note: used to fit the requirements of the Scikit-learn library

Syntax:

* First argument “-1” indicates the number of rows specified and the “-1” means the number is unspecified.
* Second argument “1” indicates the number of columns, and “1” means there is only 1 column of independent variables for X.

*Shape Property*: X.shape, which returns a description of the data as (#\_rows, #\_columns)

*Coefficient Property*: model.coef\_ , which returns the slope of the linear regression model’s line

*Intercept Property*: model.incercept\_ , which provides the y-intercept for the line of the linear regression model

Generation of Different Models

**Overall Linear Regression Construction:**

1. X = df.ColumnName.values.reshape(-1, 1)

where the column selected has the independent variables for the dataset

1. Y = df.ColumnName

where the column selected has the dependent variables for the dataset

1. model = LinearRegression( )

define the model template as a linear regression

1. model.fit(X, y)

fit the data into the model template

1. y\_pred = model.predict(X)

the y values are predicted based on X and the model is used to determine those values

1. print(y\_pred.shape)

prints the description of the predicted y values, where the pair is (#\_1 #\_2) giving the #\_1 as the number of returned predicts for a straight line that can be drawn between the data points in X.

***Create Train and Test Sets From Dataset:***

*Import the module from Scikit-learn:*

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

*Create the training and test variables:*

X\_train, X\_test, y\_train, y\_test = train\_test\_split(X, y, random\_state=1, stratify=y)

***NOTE***: the **random\_state** argument is used here only for pedagogical purposes, so that you obtain the same results when you run the notebook. You will not use it when creating your own analyses.

***NOTE:*** the **stratification** argument divides the dataset into training and test sets proportionally, with the ratio of data points in each class reserved. Without this distinction the test and training sets could present different conclusions about the data concentration for each class in the actual dataset.

***NOTE:*** It’s important to consider stratifying the data especially when the classes are severely unbalanced or the dataset is small

**Logistic Regression Model Creation**:

1. from sklearn.linear\_model import LogisticRegression

Import LogisticRegression module from Scikit-learn

1. classifier = LogisticRegression(solver=’lbfgs’, random\_state=1

create the classifier variable with the solver “lbfgs” being the default setting and used for an algorithm focused on learning and optimization.

1. LogisticRegression(C=1.0, class\_weight=None, dual=False, fit\_intercept=True, intercept\_scaling=1, max\_iter=100, multi\_class=’warn’, penalty=’12’, random\_state=1, solver=’lbfgs’, tol=0.0001, warm\_start=False)

Included to display the different arguments included in the LogisticRegression module

1. classifier.fit(X\_train, y\_train)

fit the training data to the model

1. predictions = classifier.predict(X\_test)

pd.DataFrame({“Prediction”: predictions, “Actual”: y\_test})

create a dataframe to compare the testing predictions to the actual values from the testing values

1. from sklearn.metrics import accuracy\_score

import the accuracy score module

1. accuracy\_score(y\_test, predictions)

test the accuracy of the different values and return a value between “0” to “1” with the closer value to “1” indicating the greater amount of accuracy to perfect alignment between predictions and testing.

Simplified Summary:

1. Create a model with LogisticRegression( )
2. Train the model with model.fit( )
3. Make predictions with model.predict( )
4. Validate the model with accuracy\_score( )

***Testing For Accuracy:***

**Accuracy Tests: Precision vs. Sensitivity:**

F1 Score/Harmonic Mean: 2(Precision \* Sensitivity)/(Precision + Sensitivity)

Precision = TP/(TP + FP)

Sensitivity = TP/(TP + FN)

**Confusion Matrix Generation**:

Import the Confusion\_Matrix and classification report modules from Scikit-learn

from sklearn.metrics import confusion\_matrix, classification\_report

Generate the Matrix and Print

matrix = confusion\_matrix(y\_test, y\_pred)

print(matrix)

Create Classification Report for Precision, Recall (sensitivity) and F1 score

report = classification\_report(y\_test, y\_pred)

print(report)

**Support Vector Machines:**

**Description**: a binary classifier that categorizes by yes (1) / no (0) determinations.

* Instead of a logistic regression fitting a line that best ***includes*** the data, it determines a line that best **separates** the two classes of data.
* After dividing the classes by the ***farthest margins*** between each other, a **hyperplane** (a line between two margins of class clusters with equidistant spacing between the margins) is determined.
* ***Support vectors*** are defined by the **closest data points** to the hyperplane
* Cases of the support vector data ***crossing*** the hyperplane can be accommodated by using **soft margins**

***Steps in Using SVM***

1. ***Split dataset***

- Import the train test split module

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

- Create training and testing sets for X and y

X\_train, X\_test, y\_train, y\_test = train\_test\_split(X, y, stratify=y)

- Examine the dataset’s shape for a description

X\_train.shape

1. ***Create and train the model***

- Import the SVC (support vector classification) Module

from sklearn.svm import SVC

- Create the model variable

model = SVC(kernel = ’linear’)

- Train the model

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

1. ***Create predictions***

- Generate the y predictions (the output values)

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

results = pd.DataFrame({“Prediction”: y\_pred, “Actual”: y\_test}).reset\_index(drop=True)

results.head( )

1. ***Validate the model***

- Assess the Accuracy Score

from sklearn.metrics import accuracy\_score

accuracy\_score(y\_test, y\_pred)

- Generate the Confusion Matrix

from sklearn.metrics import confusion\_matrix

confusion\_matrix(y\_test, y\_pred)

from sklearn.metrics import classification report

print(classification\_report(y\_test, y\_pred) )

Data Preparation:

Machine-learning algorithms typically use numerical data exclusively, meaning text data will consistently need to be prepared into an acceptable format for processing.

**Data Encoding:**

***Get Dummies Method*** - creating a binary format of a two-value dataset:

new\_df = pd.get\_dummies(df, columns=[“column\_name”, “column\_name\_2”, etc…])

new\_df.head( )

NOTE: Usually this is used for a boolean case (yes/no) or gender criteria (male/female), where “1” is true and “0” is false. This is done by **splitting** the column into two different ones that take the criteria and apply the appropriate indication of being a true or false case.

***Label Encoding:***

- Import LabelEncoder

from sklearn.preprocessing import LabelEncoder

le = LabelEncoder( )

- Train the Label Encoder, Then Convert Text Data into Numerical

df[‘Column\_Name’] = le.fit\_transform(df[‘Column\_Name’] )

NOTE: The resulting column includes the data with an assigned number from 0 to the max unique categories in the column. The column will not include any indication of what number representations have been substituted for.

***Custom Encoding***:

- Create a Dictionary To Specifically Reference Each Number to a Text Value

dict\_name = {“text\_1”: 1, “text\_2”: 2, … }

- Create a Function to Perform the Conversion, Unnamed or Defined

df\_name[“dict\_name”] = df\_name[“Column\_name”].apply(lambda x: dict\_name[x] )

NOTE: Uses the apply method to replace the x variable in currently in the column as a key for the corresponding value in the input dictionary. Can use a named function if the encoder will be needed multiple times.

**Data Scaling and Normalizing:**

***Using StandardScaler Module***:

- Import the StandardScaler

from sklearn.preprocessing import StandardScaler

data\_scaler = StandardScaler( )

- Train the Scaler and Transform the Data

df\_scaled = data\_scaler.fit\_transform(encoded\_df)

NOTE:

* The input dataframe must *already* be encoded for the next step in preprocessing to be successfully executed.
* After standardization, the mean should be 0 and the standard deviation should be 1, or an equivalent form for adjusted data. Check with:

import numpy as np

print(np.mean(df\_scaled[:, 0] ) )

print(np.std(df\_scaled[:, 0] ) )

Decision Trees:

**Components of a decision tree:** Root node / parent node 🡪 the top the decision tree and initial division

Child nodes 🡪 the two subnodes split from the root node

Decision node 🡪 subnodes that split from the child nodes

Terminal nodes / leaf nodes 🡪 the final nodes for the decision tree

**Steps For Using Decision Trees:**

1. Split the Data Into Training and Testing Sets
2. Scale the Training and Testing Data
3. Fit the Decision Tree Model

***Implementing Decision Trees:***

Create the decision tree classifier instance

model = tree.DecisionTreeClassifier( )

Fit the Model

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

Make predictions using the testing data

predictions = model.predict(X\_test\_scaled)

1. Evaluate the Model

Calculate the confusion matrix

cm = confusion\_matrix(y\_test, predictions)

Create a dataframe from the confusion matrix

cm\_df = pd.DataFrame(cm, index=[“Actual 0”, “Actual 1”], columns=[“Predicted 0”, Predicted 1”])

cm\_df

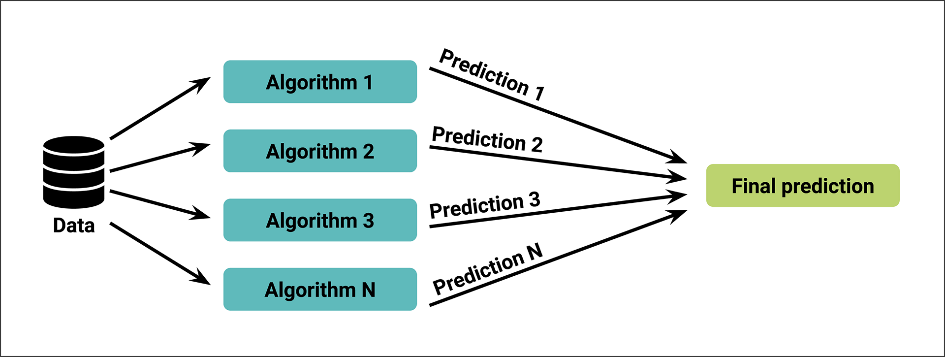
**Ensemble Learning:**

***Basic Concept***: The idea of combining multiple models and reaching a final prediction based on the accumulated predictions from each algorithm.

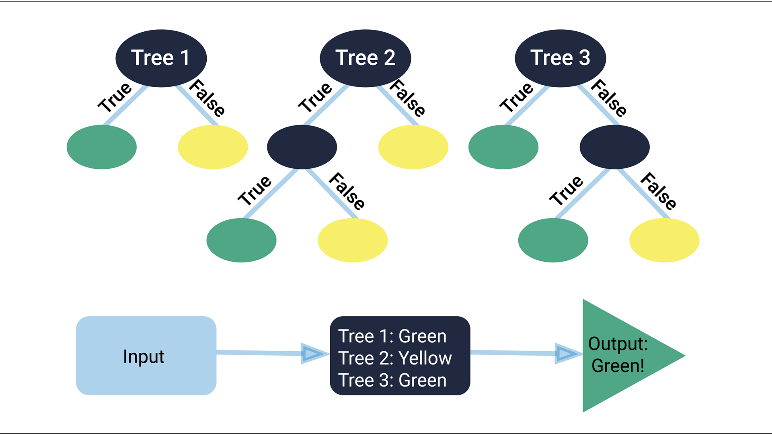
***Weak Learners***: Algorithms that tend to have very few branches on a decision tree

- Can be combined in a specific algorithm (Random Forests, GradientBoostedTree, XGBoost)

- NOTE: Weak learners can be valuable when combined to create a more accurate prediction engine. They can perform just as well as any strong learner



**Random Forests:**

****

***Steps For Implementing Random Forests:***

1. Preprocess the Data

Define the features set

X = X.drop(“Column\_selection”, axis=1)

Define the target set

y = df\_name[“Target\_Column\_name”].ravel( )

NOTE: The ravel( ) method does the same thing as the values attribute

Split into training and testing sets

X\_train, X\_test, y\_train, y\_test = train\_test\_split(X, y, … )

Scale the data

scaler = StandardScaler( )

X\_scaler = scaler.fit(X\_train)

X\_train\_scaled = X\_scaler.transform(X\_train)

X\_test\_scaled = X\_scaler.transform(X\_test)

1. Fit the Random Forest Model

Create the random forest classifier

rf\_model = RandomForestClassifier(n\_estimators=128)

NOTE: The “n\_estimators” argument sets the number of trees that will be created for the algorithm. PROS: stronger predictions and more stable, CONS: slows down the result and requires more computational resources.

(Best practice is to use between 64 and 128, but higher numbers can be used)

Fit the model

rf\_model = rf\_model.fit(X\_train\_scaled, y\_train)

1. Make Predictions Using the Testing Data

Make predictions

predictions = rf\_model.predict(X\_test\_scaled)

1. Evaluate the Model

Calculate confusion matrix

cm = confusion\_matrix(y\_test, predictions)

Create the DataFrame from confusion matrix

cm\_df = pd.DataFrame(cm, index=[“Actual 0”, “Actual 1”], columns=[“Predicted 0”, “Predicted 1”] )

cm\_df

Calcuate the accuracy score

acc\_score = accuracy\_score(y\_test, predictions)

1. Rank the Importance of Features

Calculate the feature importance in the random forest model

importances = rf\_model.feature\_importances\_

importances

Sort the features by importance

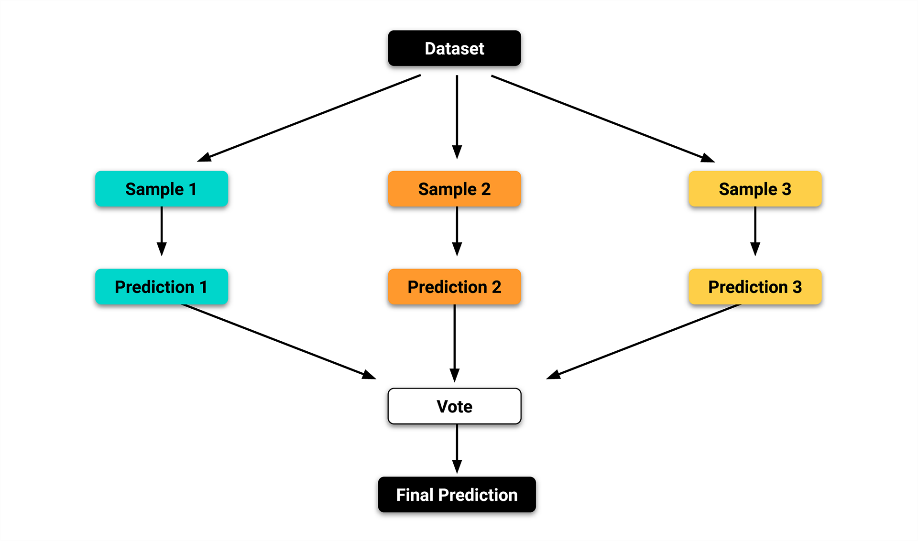
sorted(zip(rf\_model.feature\_importances\_, X.columns), reverse=True)

NOTE: The “zip” module will zip the returned list for greater memory control as compressed data

Bootstrap Aggregation:

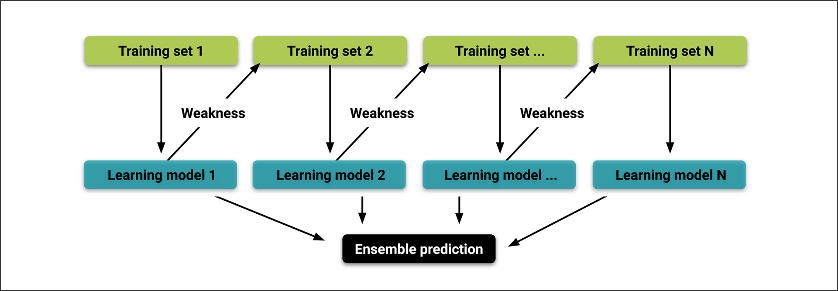
**Concepts and Theory**: Also Called “Bagging” Because of the Sample Selection Method

* Decision trees are prone to overfitting – being tailored for accuracy when applied to a specific dataset, but will less effective when applied to a new dataset – and bootstrap aggregation
* It is a sampling technique which randomly draws samples from the general pool and returns them to be part of the pool to be sampled from on the following draw.
  + In summary: bootstrapping is a sampling technique in which a number of samples are made and in which an observation can occur multiple times.
* Multiple weak learners are combined at the same time to reach a combined result



**Boosting:**

* Instead of combining multiple weak learners at the same time, they are used sequentially and build on the results of the previous iteration of the model.
  + Basically, one model learns from the mistakes of the previous iterations to create a much more powerful resulting model.



**Adaptive Boosting:**

* + Each model is trained and evaluated, the errors are noted, and each subsequent model has greater weight given to the errors of the previous model. The weighting for previous errors minimizes the impact of similar errors in the subsequent iterations.

**Gradient Boosting:**

* + Instead of the AdaBoost and basic Boosting methods’ formats, the process follows

1. A small tree (a “stump”) is added to the model and the errors are evaluated
2. A second stump is added to the first to minimize the first one’s errors

(These errors are called pseudo-residuals)

1. A third stump is added to the previous two to minimize the pseudo-residuals
2. The process is repeated with new stumps until the errors are sufficiently minimized or the specified number of repetitions has been completed.

***Steps For Applying Gradient Boosting:***

1. Declare X and y variables from dataset
2. Create training and testing sets from dataset
3. Scale the training and testing sets
4. Implement the Gradient Boosting Classifier:

- Import the dependencies

from sklearn.ensemble import GradientBoostingClassifier

- Establish learning rates

learning\_rates = [0.05, 0.1, 0.25, 0.5, 0.75, 1]

- Create a for loop to run the iterations of the model

for learning\_rate in learning\_rates:

classifier = GradientBoostingClassifier(n\_estimators=20, learning\_rate=learning\_rate,

max\_features=5,

max\_depth=3,

random\_state=0)

classifier.fit(X\_train\_scaled, y\_train.ravel)

- Print results from the classifier

print(“Learning rate: “, learning\_rate)

print(“Accuracy score (training): {0: .3f}”.format(

classifier.score(X\_train\_scaled, y\_train) ) )

print(“Accuracy score (validation): {0: .3f}”.format(

classifier.score(X\_test\_scaled, y\_test) ) )

NOTE: Previously, accuracy\_score was used to validate the model. Here, classifier.score( ) is used and the result is the same between the two different modules applied to the input data.

1. Create predictions from the model

classifier = GradientBoostingClassifier(n\_estimators=20, learning\_rate=0.5, max\_features=5, max\_depth=3, random\_state=0)

classifier.fit(X\_train\_scaled, y\_train)

predictions = classifier.predict(X\_test\_scaled)

1. Generate a confusion matrix

- Create the matrix variable

cm = confusion\_matrix(y\_test, predictions)

cm\_df = pd.DataFrame(cm, index=[“Actual 0”, “Actual 1”], columns=[“Predicted 0”, “Predicted 1”] )

display(cm\_df)

1. Print classification report

print(“Classification Report”)

print(classification\_report(y\_test, predictions) )

Dealing With Imbalances:

* **Class imbalance**: A common problem when one class is much larger than the other class.

Techniques for dealing with clss imbalance:

* + *Oversampling:* If one class has too few instances in the training set, choose more instances from that class for training until it’s larger.
    - Random Oversampling:

🡪 Does not generate synthetic observations.

Using the Counter( ) module tracks the class and the number of member data points in it and returns the values in a dictionary.

Counter(y\_train) 🡪 Counter( {0: #\_class1, 1: #\_class2} )

- Import the RandomOverSampler

from imblearn.over\_sampling import RandomOverSampler

- Declare the random oversampler

ros = RandomOverSampler(random\_state=1)

- Fit the X\_train and y\_train values to the random oversampler

X\_resampled, y\_resampled = ros.fit\_resample(X\_train, y\_train)

Running the Counter( ) on y\_resampled will provide a more balanced class distribution

Following this, the steps are the same with any model creation, training and prediction formation: fit model with the resampled values, generate predictions, create CM, validate model, produce report – with exception:

print(classification\_report\_imbalanced(y\_test, y\_pred) )

* + - Synthetic Minority Oversampling Technique (SMOTE): Instead of random selection to increase the minority class’ size, the new values are interpolated – meaning, a data point in the minority is selected and a number of its closest neighbors are selected. Based on these values, new values are created to be added to the minority class.

🡪 Generates synthetic observations based on relative neighboring observations near current minority class observations.

- Import the SMOTE dependency

from imblearn.over\_sampling import SMOTE

- Define resampled X and y values

X\_resampled, y\_resampled = SMOTE(sampling\_strategy=’auto’).fit\_resample(X\_train, y\_train)

NOTE: While capable of being more accurate than random sampling, SMOTE is particularly vulnerable to proliferating outliers in cases where the selected neighboring values are outliers themselves.

* + ***Undersampling:***
    - Random Undersampling: Randomly selected instances from the majority class are removed until the size of the majority class is reduced to an acceptable ratio. Typically to the level of the minority class.

- Select X and y data columns, create train and test sets

- Import the RandomUnderSampler dependency

from imblearn.under\_sampling import RandomUnderSampler

- Define the ROS variable

ros = RandomUnderSampler(random\_state=1)

- Fit the training data into the resampler

X\_resampled, y\_resampled = ros.fit\_resample(X\_train, y\_train)

Counter(y\_resampled) 🡪 To confirm the rebalancing was successful

- Fit data into the model

- Make predictions and generate the Confusion Matrix

- Import the Balanced Accuracy Score

from sklearn.metrics import balanced\_accuracy\_score

- Calculate the Balanced Accuracy Score

balanced\_accuracy\_score(y\_test, y\_pred)

- Import the classification imbalanced report

from imblearn.metrics import classification\_report\_imbalanced

- Print the report

print(classification\_report\_imbalanced(y\_test, y\_pred) )

* + - Cluster Centroid Undersampling: Akin to SMOTE, clusters in the majority class are identified and synthetic data points are generated – called centroids – that are representative of the clusters. The majority class is then undersampled down to the size of the minority class.

- Import the ClusterCentroids dependency

from imblearn.under\_sampling import ClusterCentroids

- Declare the cluster centroids variable

cc = ClusterCentroids(random\_state=1)

- Fit the module with the training set data

X\_resampled, y\_resampled = cc.fit\_resample(X\_train, y\_train)

* + ***Sampling Combination***:
    - SMOTEENN -

(Synthetic Minority Oversampling Technique and Edited Nearest Neighbors):

Two step process:

1. Oversample the minority class with SMOTE
2. Clean the resulting data with an undersampling strategy.

* If the two nearest neighbors of a data point belong to two different classes, that data point is dropped.

- Import the dependency for SMOTEENN

from imblearn.combine import SMOTEENN

- Define the method as a variable

smote\_enn = SMOTEENN(random\_state=0)

- Fit the training data to the method algorithm

X\_resampled, y\_resampled = smote\_enn.fit\_resample(X, y)

- Again apply the data to a model and use the

🡪 balanced accuracy score

🡪 classification\_report\_imbalanced

**Unsupervised Machine Learning**

Key Differences To Supervised Machine Learning:

1. No paired inputs and outputs
2. Model uses a whole dataset as the input

- Kind of like Macro vs. Micro Dataset Analysis

Key Uses For Unsupervised Machine Learning:

1. Transformations - To transform data to create an intuitive representation for analysis or use in another machine learning model

- Specifically, raw data is input and transform to make it easier to understand meanings behind it. It can reduce extra data dimensions that only obscure the useful patterns, the data can then be processed for use in other algorithms or narrowed down to a simplified form to be viewed on its own.

1. Clustering - To cluster or determine patterns in a data grouping rather than to predict a classification

- Used to group similar objects into clusters and divide them based on a recognized pattern. Rather than reducing the data down to an understandable format – such as with transformations – this method groups the current data together in a more simplified form of presentation but preserves the actual data values.

Can be thought of as – Non-Directed Machine Learning or Without Specified Instructions. The program is only instructed to find patterns and group them accordingly, as opposed to certain details for what groups, how many, or what criteria defines them.

Data Preparation:

Important Questions To Consider During Preparation:

* What general ideas could potentially be gained from running an unsupervised model on this dataset
* What data is available and what data is missing, what can be removed. Also, what type of data is available
* Is the data in a usable format for passing into an unsupervised learning model
* Can this data be quickly offered for others to use in its current form

Steps in Data Preparation:

1. **Data Selection** – Select the relevant data fields that you want, and drop the extras that will add noise to the other fields.

--- ***Looking for what you, as the initial analyst, want in the data.***

1. **Data Processing** – Organize, clean and format the data into a fully usable dataset.

--- ***Setting up what the unsupervised model needs out of the data***

* 1. Assess data types – df.dtypes
  2. Assess null values – df.isnull( )
  3. Remove NaN values – df.dropna( )
  4. Identify and potentially remove duplicate data values – df.duplicated( ).sum( )
  5. Drop unnecessary columns – df.drop(columns=[“Column\_Name”], inplace=True]
  6. Ensure only numerical data is input – If needed, create a function, such as:

def change\_string(string):

if string == “Yes”:

return 1

else:

return 0

Then use: df.apply(change\_string)

* 1. Resize data if needed – df[“Column\_Name”] = df[“Column\_Name”] (operator \* or /) #

1. **Data Transformation** – Configure the dataset into an ideal source for input into an algorithm. Particularly ideal – write the new data to a CSV file.

--- ***Setting up what the user of the dataset will need for running an algorithm to generate the model***

Export the dataset as a CSV:

file\_path = “../data\_cleaned.csv”

df.to\_csv(file\_path, index=False)

Clustering Data:

Clustering data based on similarities between multiple dimensions of data fields finds some of the clearest patterns/trends by visual presentation.

**K-Means Algorithm:** Creates clusters based on ‘K’, which represents how many clusters will be created.

Key Aspects:

- ‘K’ clusters are determined by the ***MEAN*** values of all points in that cluster

- Inclusion in a cluster is determined based on the similarity of data points, or distance measure, to a centroid

- A ‘Centroid’ is a data point that is the arithmetic mean position of all the points on a cluster. The closest to absolute mean of the cluster and used as a reference point

🡪 The Centroid is found by taking the mean of all x values in a cluster and mean of all y values in a cluster. Effectively:

Centroid (x, y) = (mean(x), mean(y)) , per cluster

**Implementing the K-Means Algorithm:**

1. **Import dependencies**:

import pandas as pd

import plotly.express as px

import hvplot.pandas

from sklearn.cluster import KMeans

1. **Import the dataset**:

file\_path = “../datafile.csv”

df = pd.read\_csv(file\_path)

df.head( )

1. **Initialize K Starting Centroids**:

Start with K = #, for # = the number of classes in a dataset. Often an unknown value, meaning the assigned # will need to be determined through trial-and-error until a satisfactory model is reached.

model = KMeans(n\_clusters=#, random\_state=#)

model

***Trial and Error Methods:***

* 1. Plot the DataFrame with:

df.hvplot.scatter(x=”Column\_Name1”, y=”Column\_Name2”)

* 1. Create a function to cluster and plot the dataset more easily:

def test\_cluster\_amount(df, clusters):

model = KMeans(n\_clusters=clusters, random\_state=#)

model

# Fit the model

model.fit(df)

# Predict clusters

predictions = model.predict(df)

# Add a new class column to the dataframe

df[“class”] = model.labels\_

return df

* 1. Test the number of clusters by inputting the Dataframe name and number to try as:

test\_cluster\_amount(df, #\_of\_clusters)

df.hvplot.scatter(x=”Column\_Name1”, y=”Column\_Name2”, by=”class”)

*Alternatively, the trial clusters can be tested in a 3D plot with:*

fig = px.scatter\_3d(df, x=”Col1”, y=”Col2”, z=”Col3”, color=”class”, symbol=”class”, width=800/#)

fig.update\_layout(legend=dict(x=0, y=1) )

fig.show( )

* 1. Determine how many clusters make the data clearer while still making something usable compared to individually plotted points without clusters

1. **Assign Data Points to Nearest Centroid** - Fit the DataFrame to the model:

model.fit(df)

1. **Group The Data Points** - Corresponding clusters for every datapoint in the dataset are found:

predictions = model.predict(df)

print(predictions)

1. **Add The Class Column** – The classes are the labels for the clusters:

The Model Includes the K-Means Algorithm That Will Assign The Class Labels With Property ‘labels\_’ Applied

df[“Class”] = models.labels\_

1. **Visualize The Results** – With features numbering more than 3, the data cannot all be plotted and key features must be selected to generate a plot for the clusters:

2D – df.hvplot.scatter(x=”feature\_1”, y=”feature\_2”, by=”class”)

3D – fig = px.scatter\_3d(df, x=”feature\_1”, y=”feature\_2”, z=”feature\_3”, color=”class”, symbol=”class”, size=”feature\_4”, width=(fitted as necessary: 800/1000/#) )

fig.update\_layout(legend=dict(x=0, y=1) )

fig.show( )

**# 2. Continued -- The Elbow Curve Method For Estimating The Number of Needed Clusters:**

By plotting the clusters on the x-axis and the values of a selected **objective function** on the y-axis, the curve reaches a key value that curves it into a similar shape to an elbow.

🡪 A Common Objective Function: The **Interia** function

1. Store the “K” Values to Plot:

- inertia = [ ]

k = list(range(1, 11) )

2. Look for the best K:

- for I in k:

km = KMeans(n\_clusters=I, random\_state=0)

km.fit(df)

intertia.append(km.intertia\_)

3. Create a DataFrame and Plot the Elbow Curve:

- elbow\_data = {“k”: k, “interia”: intertia}

df\_elbow = pd.DataFrame(elbow\_data)

df\_elbow.hvplot.line(x=”k”, y=”interia”, title=”Elbow Curve”, xticks=k)

The resulting graph will show more level slopes with each cluster increase, the “k” value that is highest before the slope begins to dramatically level off for the rest of the cases is the best option for reducing data variation but not reducing clarity in patterns.

Overall, the final determination of how many clusters are needed is up to the analyst:

- Choose the trend needed to break down the data into the parts needed for a useful conclusion

***Managing Data Features:***

**- Dimension Reduction**: How to reduce features to the key components

NOTE: Too many features not only obscure trends and make models ineffective at finding accurate patterns – Too many features also can lead to Overfitting and more reduced accuracy

1. **Feature Elimination**:

- Used to remove features, but also means removed features no longer provide potentially useful data that could provide insight

1. **Feature Extraction**:

- Combines all features into a new set that is ordered by how well they predict the original variable. Meaning – Transforms a large set of variables into a smaller one and is meant to order by the most important parts that contribute to determine the original dataset variable.

- **Principal Component Analysis (PCA):** A process that is “complicated to understand, but easy to code” – indicating a lot of math aspects and understanding is key for applying it correctly.

Steps For Implementation:

1. Import Dependencies

import pandas as pd

from sklearn.preprocessing import StandardScaler

from sklearn.decomposition import PCA

from sklearn.cluster import KMeans

import hvplot.pandas

1. Standardize the Scaling of the Features

df\_scaled = StandardScaler( ).fit\_transform(df)

print(df\_scaled[0: 5])

1. Initialize PCA Model: With “n\_components” meaning target number of features to be reduced down to.

pca = PCA(n\_components=2)

1. Fit the PCA with the Scaled Data for Dimensionality Reduction

df\_pca = pca.fit\_transform(df\_scaled)

1. Create a DataFrame With Columns For Each N-Component

new\_df\_pca = pd.DataFrame(data=df\_pca, columns=[“principal component 1”, “principal component 2”] )

new\_df\_pca.head( )

1. Determine the Variance Ratio For How Much Information The Components Contribute To The Original Dataset

pca.explained\_variance\_ratio\_

The resulting array includes the percentage values calculated for how much information each component contributed to the original dataset overall

**Understanding the PCA Concepts:**

Important Mathematical Terms To Review (Specifically Geometric Terms):

* **Variance:** The sum of the squared distances each point is located from the center and divided by the total number of points considered.

v = [ (x\_1) – (x\_center) ]^2 + [ (x\_2) – (x\_center)^2 ] / (3)

- Divided by 3 because of points x\_1, x\_2, and x\_center

- Same method applied to y-axis values and added together

* Covariance: The sum of the product from each pair of coordinates and the difference from the mean divided by the total number of data points involved.

Cov (x, y) = ∑(xi -xmean)(yi – ymean) / N

🡪 Meaning: - Find the mean of all x values and mean of all y values

- Subtract the mean values from each x and each y coordinate value

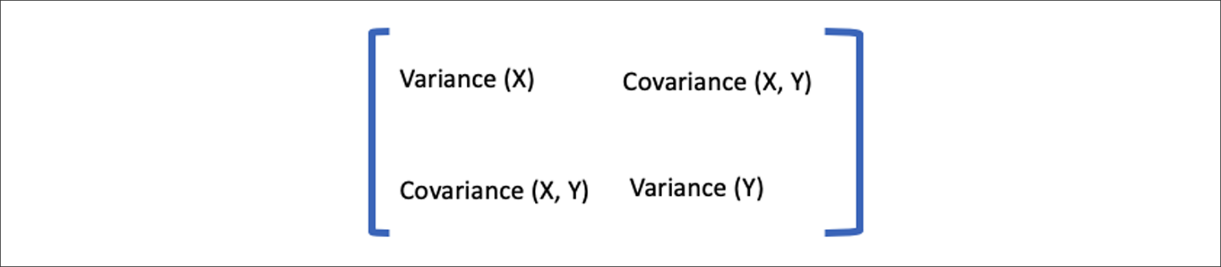
- Multiply each point’s resulting x and y combinations together

- Add all resulting point values together

- Divide the resulting number by the number of data points involved

NOTE: Covariance is used to describe the relationship between points, such as positive and negative relationships.

* **Linear Transformation:**



With a set of points on a graph, to center them the method is:

- Take the average of the coordinates (x) and (y)

- Find the balance point and move it to zero

- Create the matrix of values, as shown above

🡪 the top row is a datapoint and the bottom is another

* + Eigenvectors: The directions the points stretch out from the center point
  + Eigenvalues: The magnitude values for each eigenvector

- Determining these values is most easily done through WolframAlpha’s calculator function

***NOTE:*** The general meaning behind these linear algebra terms is to show the spread of a dataset’s points and measure how much the spread is.

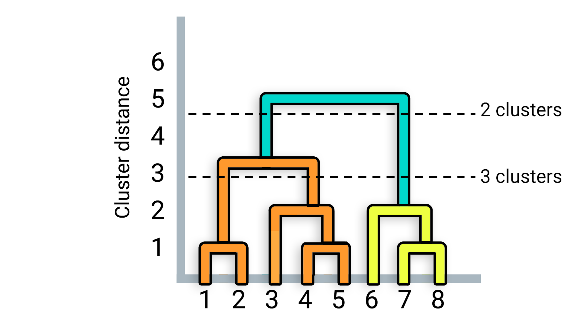
- Using these values, the higher eigenvalues indicate greater impact on the data and helps reduce the lower impact dimensions of data. The result is a fitted estimate for how the most important data fields play into creating the dataset’s values and by how much they impact it.

**Hierarchical Clustering:** Starts by declaring each datapoint its own cluster and merges the two most similar clusters on a consecutive system up until a stopping point declared in the code.

Methods for Determining How Points Are Linked:

* Ward (default): Selects clusters that will result in the least amount of variance when merged. *Often* this means combining clusters of relatively equal size.
* Average: Connects clusters that have the smallest average distance between all points in the cluster, then connects clusters based on the smallest average distance between all of their points compared to all other clusters.
* Complete: Merges clusters that have the smallest maximum distance

- **Dendogram**: Similar to an elbow curve, it shows how many clusters will be needed depending on how refined the data clustering needs to be. Builds a hierarchy of larger clusters of data and indicates the difference between each smaller cluster’s initial points



Steps for Implementing Hierarchical Clustering:

1. Import dependencies:

import pandas as pd

from sklearn.preprocessing import StandardScaler

from sklearn.decomposition import PCA

from sklearn.cluster import AgglomerativeClustering

import hvplot.pandas

import plotly.figure\_factory as ff

1. Create the dendogram figure after PCA use:

fig = ff.create\_dendrogram(df\_pca, color\_threshold=0)

fig.update\_layout(width=800, heigh=500)

fig.show( )

1. Run hierarchical algorithm:

agg = AgglomerativeClustering(n\_clusters=#)

model = agg.fit(df)

1. Add the class column to label the different clusters:

df\_pca[“class”] = model.labels\_

df\_pca.head( )

1. Plot the data results:

df\_pca.hvplot.scatter(x=”principal component 1”, y=”principal component 2”, hover\_cols=[“class”], by=”class”)

**Pros and Cons of K-Means vs Hierarchical Clustering:**

- K-Means is dependent on random initialization, meaning the outcome is dependent on a random seed

- K-Means works best with spherical-looking data with points of similar density closely grouped together

- Hierarchical and dendrograms make it easier to pick how many clusters to use without making assumptions since a “K” doesn’t need to be known ahead of time

- The dendrogram doesn’t always create a clear choice and leaves the final decision much more up to the analyst

- Hierarchical might not work as well on larger datasets because they have a slower run-time and take longer overall

- Hierarchical also leaves a lot of decisions to be made about when to merge groups of clusters

