SDSU | HealthLINK Center

Data Science Training
Fall 2025 Workshop: Introduction to Machine Learning

Machine Learning using Google Cloud AutoML and MATLAB

SDSU HealthLINK Center for Transdisciplinary Health Disparities Research
Funder: National Institute on Minority Health and Health Disparities (U54MD012397 & S21MD010690)
MPIs: Guadalupe X. Ayala, PhD, MPH, MA and Kristen J. Wells, PhD, MPH

Session 2: Monday, September 15, 2025
Machine Learning Using MATLAB

MATLAB ML Tools Overview

Supervised & Unsupervised Learning in MATLAB

Hands-On Exercise 2: MATLAB ML Practice

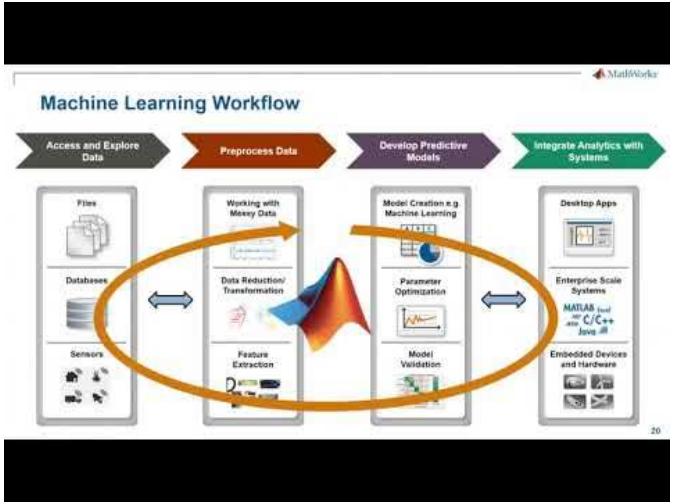
MATLAB ML Tools Overview

- Using the Classification Learner App
- Overview of ML Toolbox

Machine Learning Toolbox: MATLAB Apps

MATLAB provides a powerful environment for building machine learning applications, and its Machine Learning Toolbox is a key component. While you can certainly write code from scratch, MATLAB Apps offer a user-friendly, interactive way to perform many machine learning tasks without extensive coding.

- Apps: Ideal for initial exploration, quick prototyping, comparing different models, and when you prefer a visual, interactive approach.
- Code: Necessary for more customized workflows, complex preprocessing steps, implementing less common algorithms, or when you need to integrate machine learning into a larger application.

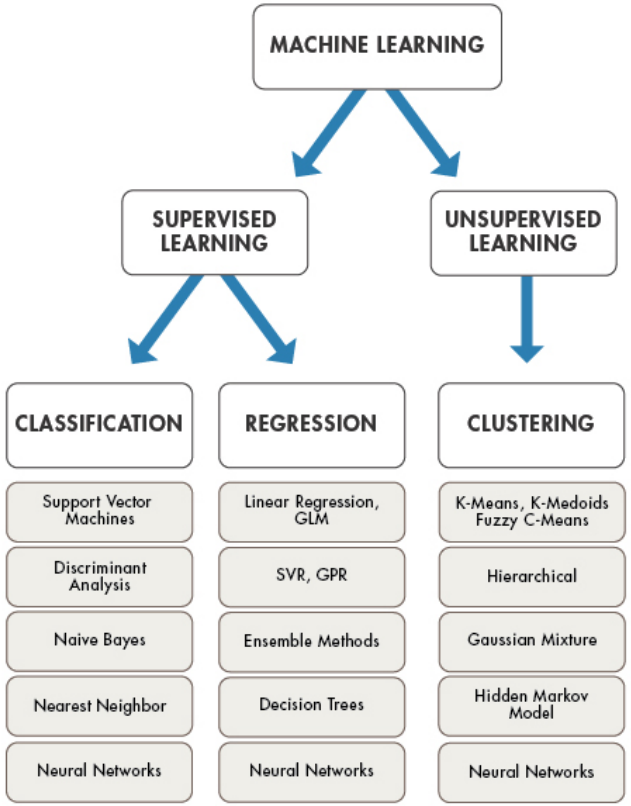


Machine Learning Toolbox: MATLAB Apps

Key Apps within the Machine Learning Toolbox (and related functionalities):

- **Classification Learner:** This app is excellent for training and comparing various classification models. You can import your data, select different algorithms, train them, evaluate their performance, and even export the trained models for use in other applications. It's a great starting point for many classification problems.

- **Clustering Toolbox (and related functions):** While not a dedicated app, MATLAB's clustering functionalities are readily integrated. You can use these in conjunction with the Classification and Regression Learners to perform clustering as a preprocessing step or to analyze the results of your models.



Machine Learning Toolbox: MATLAB Apps

Benefits of Using MATLAB Apps for Machine Learning:

Ease of Use: Reduces the need for extensive coding, especially for common tasks.

Interactive Exploration: Allows you to quickly experiment with different algorithms and settings.

Visualization: Provides tools for visualizing data and model performance.

Faster Prototyping: Accelerates the process of building and testing machine learning models.

General Workflow with MATLAB Apps for Machine Learning:

1.Data Import: Most apps allow you to import data from various sources (files, workspaces, databases, etc.).

2.Data Preprocessing: You can often perform basic preprocessing steps within the apps.

3.Model Selection: Choose the appropriate machine learning algorithm for your task (classification, regression, clustering). The Classification and Regression Learner apps provide a selection of common algorithms.

4.Training: Train the model using your data. The apps provide options for splitting the data into training and testing sets (or using cross-validation).

5.Evaluation: Assess the performance of your trained model using various metrics (accuracy, precision, recall, RMSE, etc.). The apps will often generate visualizations to help you understand the model's performance.

6.Model Export: Once you're satisfied with a model, you can export it for use in other MATLAB applications or even deploy it outside of MATLAB.

Supervised & Unsupervised Learning in MATLAB

- Building classification and regression models
- Introduction to clustering algorithms
- Neural networks and backpropagation

Step-by-Step Guide to Start MATLAB Classification Learner App

Step 1: Launch MATLAB

Start by opening MATLAB. Ensure you have access to the Statistics and Machine Learning Toolbox, which includes the Classification Learner App.

Step 2: Load or Prepare Your Dataset

Import your own dataset from a file like .csv, .xlsx, or from the MATLAB workspace.

Example:

- Import Sample Data Set ('penguins_ML4.xlsx')
- Run ('penguins_import_data.m')

The sample data set contains 1,368 observations of 3 different penguin species: Adélie, Chinstrap, and Gentoo.

Each observation has 2 features extracted from raw data:
Bill length (mm) and Bill depth (mm)

Building classification models using Matlab

Before you start (both apps)

Put your data in a table (rows = observations, columns = variables).

Make the target:

- Classification: categorical/char/string (class labels)
- Regression: numeric (continuous)
- Mark categorical predictors as categorical.

Handle missing values (remove or impute); remove obvious data leaks (ID, target-derived fields).

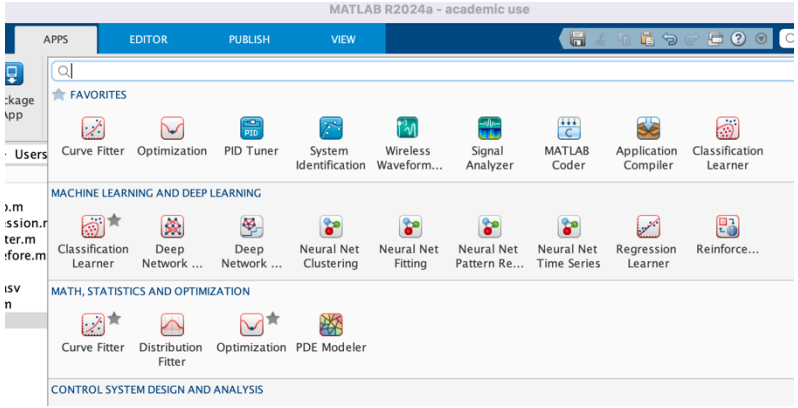
Example:

- Run (‘penguins_beforeAPP.m’)

Train classification models

Step 3: Open the Classification Learner App

From the Toolstrip:
1. Go to the "Apps" tab on the MATLAB Toolstrip.
2. Under Machine Learning and Deep Learning, click Classification Learner.

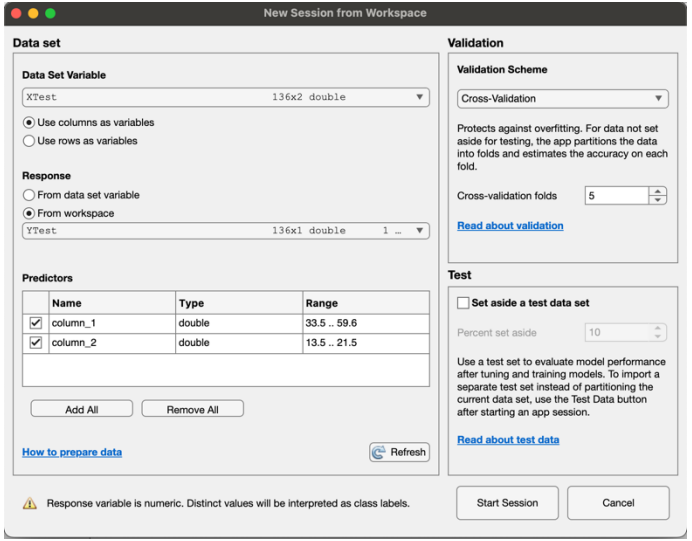


Train classification models

Step 4: Import Your Data

In the Classification Learner App:

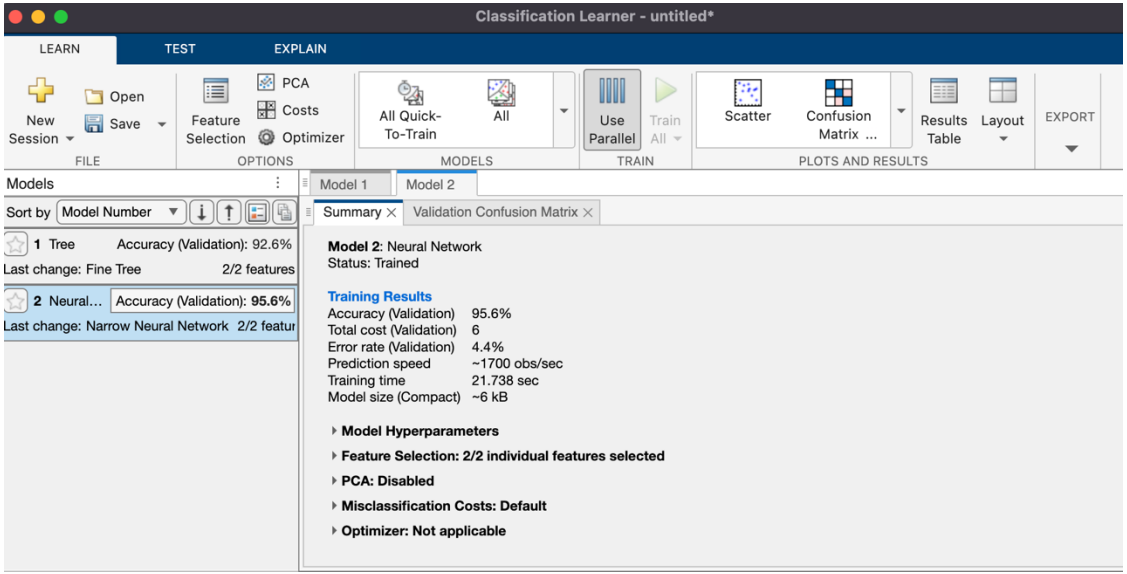
1. Click 'New Session > From Workspace' or 'From File'.
2. Select your table or matrix variable.
3. Choose the response variable (the target/class you want to predict).
4. Optionally, deselect variables you don't want to include as predictors.
5. Click 'Start Session'.



Step 5: Explore and Train Models

Once your data is loaded:

1. You'll see options like All Quick-to-Train, Decision Trees, SVM, KNN, etc.
2. Select one or more classifiers to train.
3. Click 'Train' (or 'Train All') to build your models.



Step 6: Evaluate Model Performance

After training:

- Check the accuracy, confusion matrix, ROC curves, and cross-validation results.
- Compare multiple models side by side.

Step 7: Export or Use the Trained Model

You can:

- Export the trained model to the workspace: Click 'Export Model' > 'Export Model'.
- Generate MATLAB code: Click 'Export Model' > 'Generate Function' to reuse or automate.

Workspace (after export)

- If you click **Export Model → Export Model** (the first option), MATLAB exports the model as a **trained classifier object** into the **base workspace**.
- You'll see something like trainedModel appear in your Workspace.

Structure of trainedModel

- The exported model is typically a struct, often named trainedModel, containing fields such as:
- trainedModel.ClassificationModel – the actual classification model (e.g., SVM, Decision Tree)
- trainedModel.predictFcn – a prediction function you can use directly on new data
- trainedModel.RequiredVariables – list of variables needed for prediction

MAT-file (if you save it)

If you want to **save the model permanently**, you need to explicitly save it:

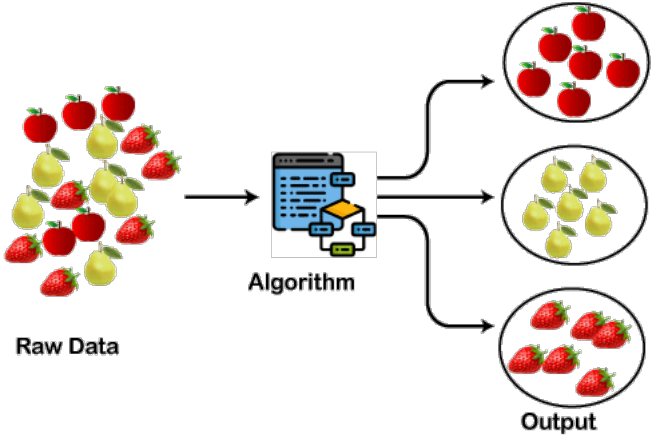
save('myModel.mat', 'trainedModel')

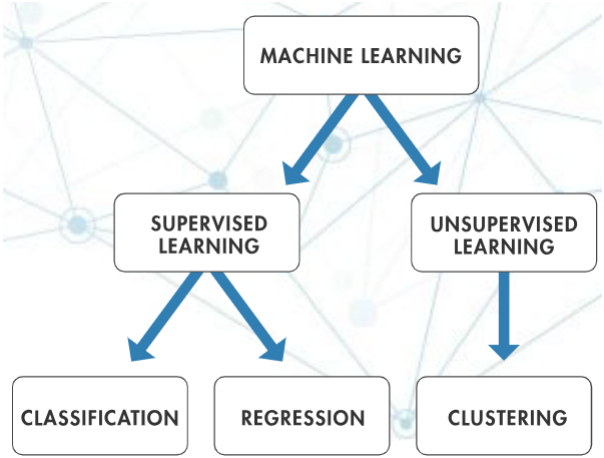
Then it can be loaded later with:

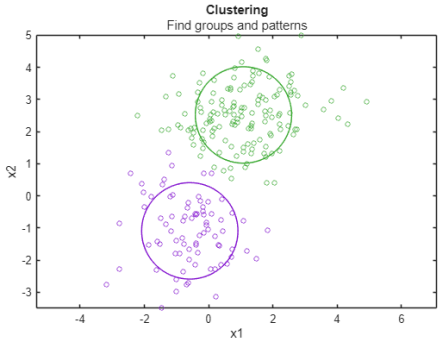
load('myModel.mat')

What is clustering?

- Clustering is a fundamental technique in unsupervised learning that allows us to discover hidden structures and patterns in unlabeled data.
- It has a wide range of applications and is an essential tool for data analysis and machine learning.
- Essentially, its goal is to partition a set of data into clusters, or collections of data points which are similar to the other data within their cluster, and dissimilar to the data in other clusters.

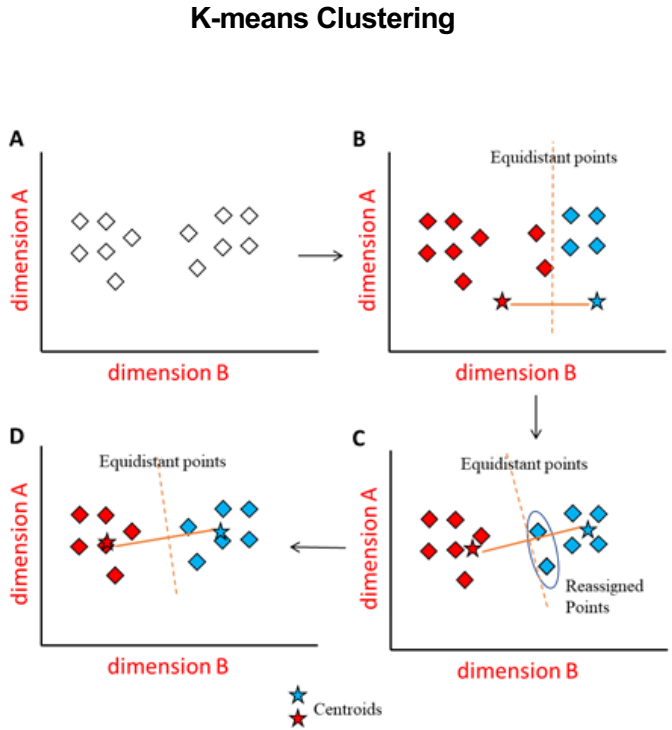






Clustering algorithms

- **K-means Clustering:** This algorithm partitions the data into a pre-defined number of clusters (k) by iteratively assigning data points to the nearest cluster center (centroid).
- **Hierarchical Clustering:** This method builds a hierarchy of clusters by either merging smaller clusters into larger ones (agglomerative) or dividing larger clusters into smaller ones (divisive)
- **Density-Based Clustering:** These algorithms identify clusters based on the density of data points in a region. They can find clusters of arbitrary shapes and are robust to outliers.
- **DBSCAN:** A popular density-based clustering algorithm that can discover clusters of different shapes and sizes, and is robust to noise.

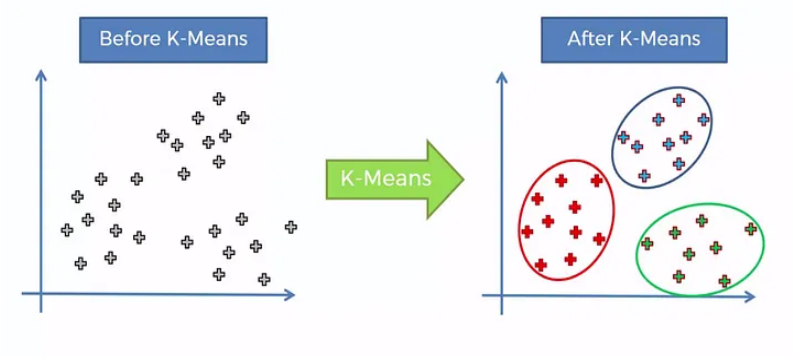


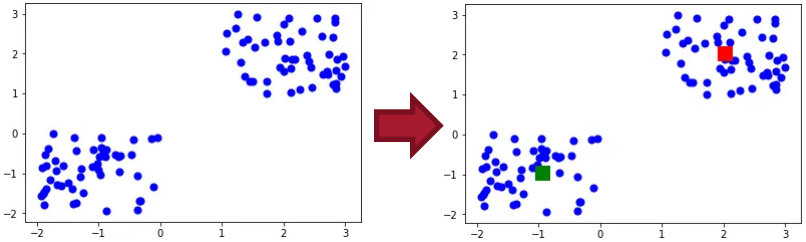
One method of clustering: k-means

The most commonly used clustering algorithm, k-means is a method in which there are k clusters. As the name of the algorithm implies, each of the k clusters is defined by its mean, or centroid.

The algorithm proceeds as follows:

1. Select k points as the initial centroids of the clusters. These can be selected randomly or by hand.
2. Assign each data point to its closest centroid. Closeness can be defined using a variety of distance measurements.
3. Re-compute the centroids by calculating the mean of all the points in each cluster. In other words, update the locations of the centroids based on their assigned data points from step 2.
4. Repeat steps 2 and 3 until the algorithm converges (until repeating steps 2 and 3 no longer changes the results of the algorithm).





Clustering Example: Data for three different species of penguins: Adélie, Chinstrap, and Gentoo.

Run (‘penguins_kmeans.m’)

X =

39.1000	18.7000
39.5000	17.4000
40.3000	18.0000
36.7000	19.3000
39.3000	20.6000
38.9000	17.8000
39.2000	19.6000
34.1000	18.1000
42.0000	20.2000
37.8000	17.1000
37.8000	17.3000
41.1000	17.6000

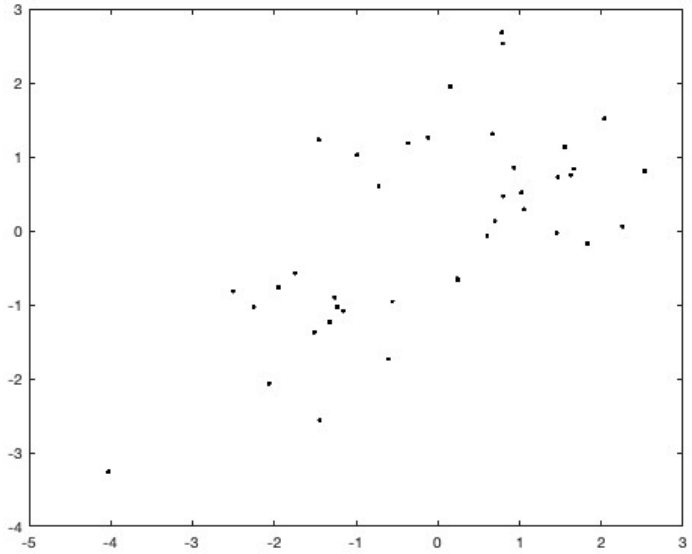

```
% Load the CSV file into a table
DataTable = readtable('penguins.csv');
head(DataTable)
LabeledData = rmmissing(DataTable);
PenguinData = LabeledData(1:end,3:4);
PenguinData2 = table2cell(PenguinData);
X = cell2mat(PenguinData2);
```

EXERCISE: Simple Clustering Example: k-means

Example: we want to assign 40 data points to 2 clusters (k = 2).

```
X = [randn(20,2)+ones(20,2); randn(20,2)-ones(20,2)];
figure(1)
plot(X(:,1),X(:,2),'k.');
```

```
X =
   -0.3617    1.1832
    1.4550   -0.0298
    0.1513    1.9492
    0.6651    1.3071
    1.5528    1.1352
                 . . .
   -0.9875    1.0237
   -4.0292   -3.2584
   -1.4570    1.2294
    0.2424   -0.6624
```



Use Matlab toolbox: kmeans.m

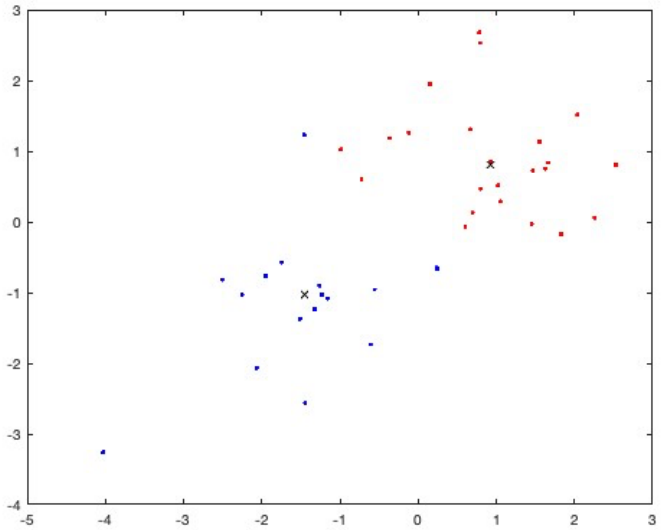
kmeans K-means clustering:

IDX = kmeans(X, K) partitions the points in the N-by-P data matrix X into K clusters.

Note: when X is a vector, kmeans treats it as an N-by-1 data matrix, regardless of its orientation. kmeans returns an N-by-1 vector IDX containing the cluster indices of each point.

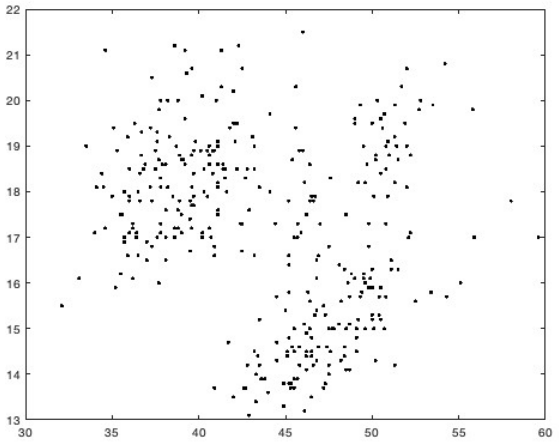
By default, kmeans uses squared Euclidean distances.

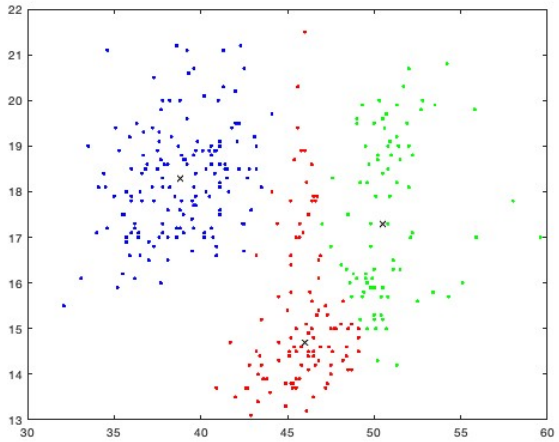
```
opts = statset('Display','final');
[cidx, ctrs] = kmeans(X, 2, 'Distance','city', ...
'Replicates',5, 'Options',opts);
figure(2)
plot(X(cidx==1,1),X(cidx==1,2),'r.', ...
X(cidx==2,1),X(cidx==2,2),'b.', ctrs(:,1),ctrs(:,2),'kx');
```



```
figure(1)
plot(X(:,1),X(:,2),'k.');

opts = statset('Display','final');
[cidx, ctrs] = kmeans(X, 3, 'Distance','city', ...
'Replicates',5, 'Options',opts);
figure(2)
plot(X(cidx==1,1),X(cidx==1,2),'r.', ...
X(cidx==2,1),X(cidx==2,2),'b.', ...
X(cidx==3,1),X(cidx==3,2),'g.', ctrs(:,1),ctrs(:,2),'kx');
```





When to Use K-Means Clustering

- K-means presents huge advantages, since it scales to large data sets, is relatively simple to implement, guarantees convergence, can warm-start the positions of centroids, it easily adapts to new examples, and generalizes to clusters of different shapes and sizes, such as elliptical clusters.

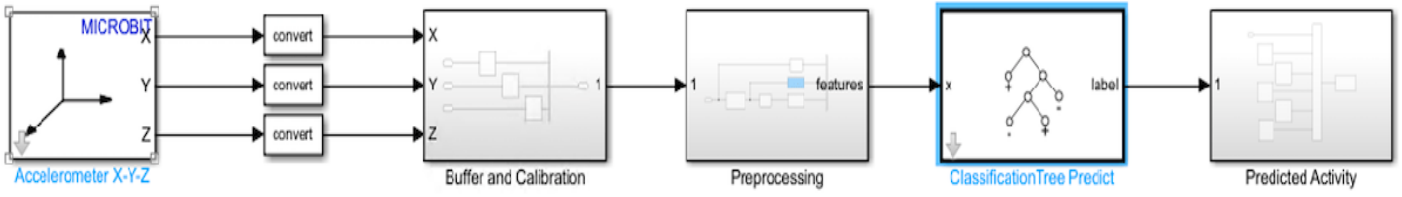
- But as any other Machine Learning method, it also presents downsides. The most obvious one is that you need to define the number of clusters manually, and, although we showed some ways to find the optimal “k”, this is a decision that will deeply affect the results.

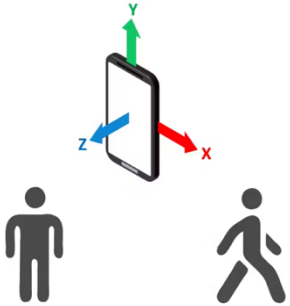
- Also, K-means is highly dependent on initial values. For low values of “k”, you can mitigate this dependence by running K-means several times with different initial values and picking the best result. Finally, K-means is very sensitive to outliers, since centroids can be dragged in the presence of noisy data.

- K-means is highly flexible and can be used to cluster data in lots of different domains. It also can be modified to adapt it to specific challenges, making it extremely powerful. Whether you’re dealing with structured data, embeddings, or any other data type, you should definitely consider using K-means.

Hands-On Exercise 2: Predicting human activity types using Matlab Machine Learning APP

This example shows how to prepare an activity prediction model that classifies human activity based on smartphone sensor signals for code generation and smartphone deployment.





EXERCISE: Predicting human activity types using Matlab Machine Learning APP

Load Sample Data Set
Load the humanactivity data set.

```
load humanactivity
```

The humanactivity data set contains 24,075 observations of five different physical human activities:
Sitting, Standing, Walking, Running, and Dancing.
Each observation has 60 features extracted from acceleration data measured by smartphone accelerometer sensors.

Data Preprocessing: You can often perform basic preprocessing steps within the apps.

The data set contains the following variables:

actid — Response vector containing the activity IDs in integers: 1, 2, 3, 4, and 5 representing Sitting, Standing, Walking, Running, and Dancing, respectively

actnames — Activity names corresponding to the integer activity IDs

feat — Feature matrix of 60 features for 24,075 observations

featlabels — Labels of the 60 features

Run (‘humanactivity_data.m’)

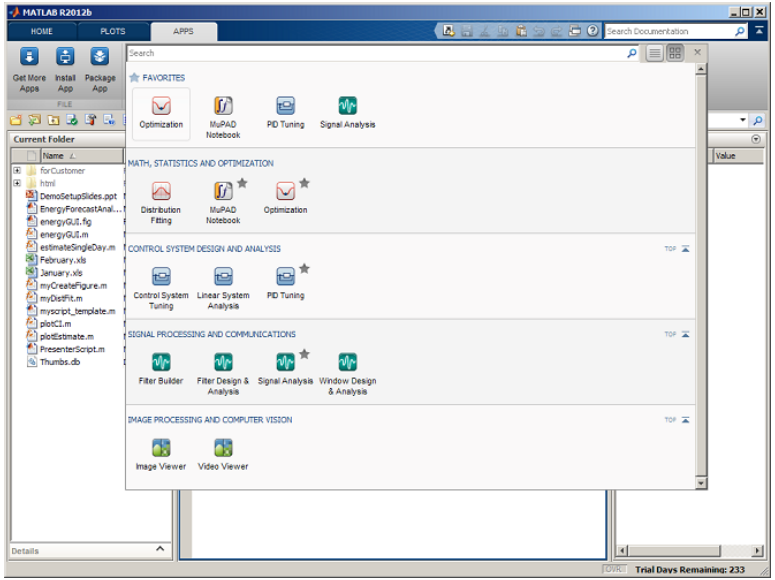
Prepare Data

```
rng('default') % For reproducibility
Partition = cvpartition(actid,'Holdout',0.10);
trainingInds = training(Partition); % Indices for the
training set
XTrain = feat(trainingInds,:);
YTrain = actid(trainingInds);
testInds = test(Partition); % Indices for the test set
XTest = feat(testInds,:);
YTest = actid(testInds);
```

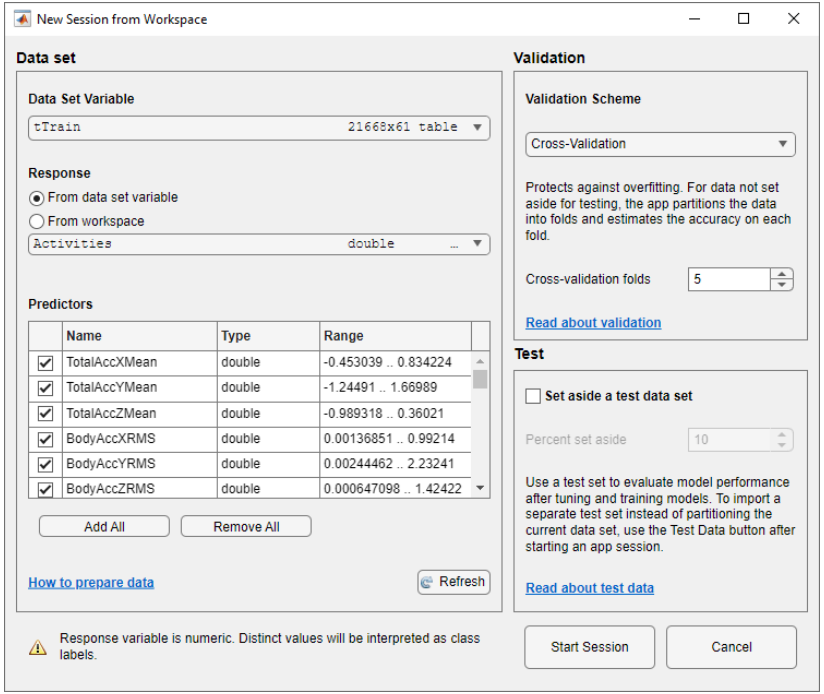
```
tTrain = array2table([XTrain YTrain]);
```

```
tTrain.Properties.VariableNames = [featlabels' 'Activities'];
```

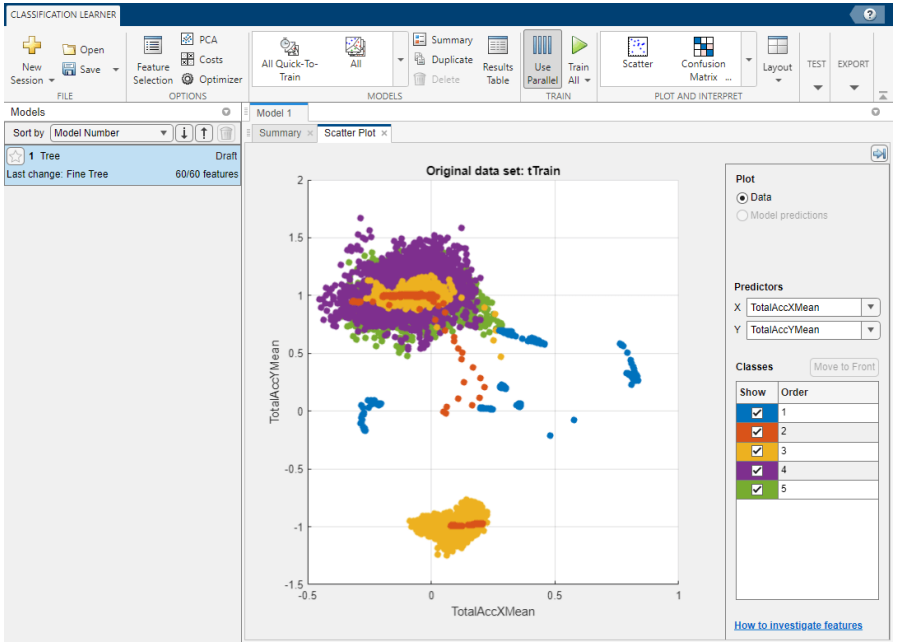
3. Model Selection: Choose the appropriate machine learning algorithm for your task (classification, regression, clustering). The Classification and Regression Learner apps provide a selection of common algorithms.

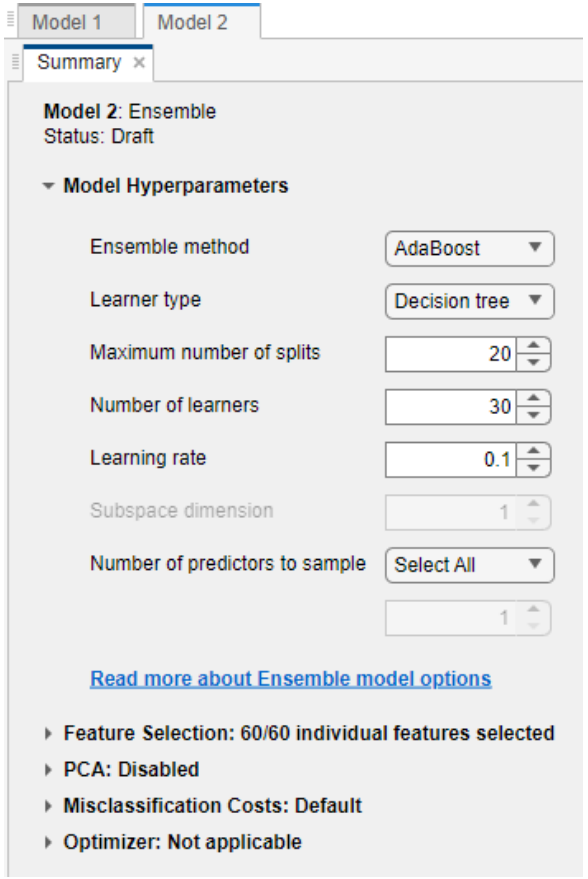


4. Training: Train the model using your data. The apps provide options for splitting the data into training and testing sets (or using cross-validation).

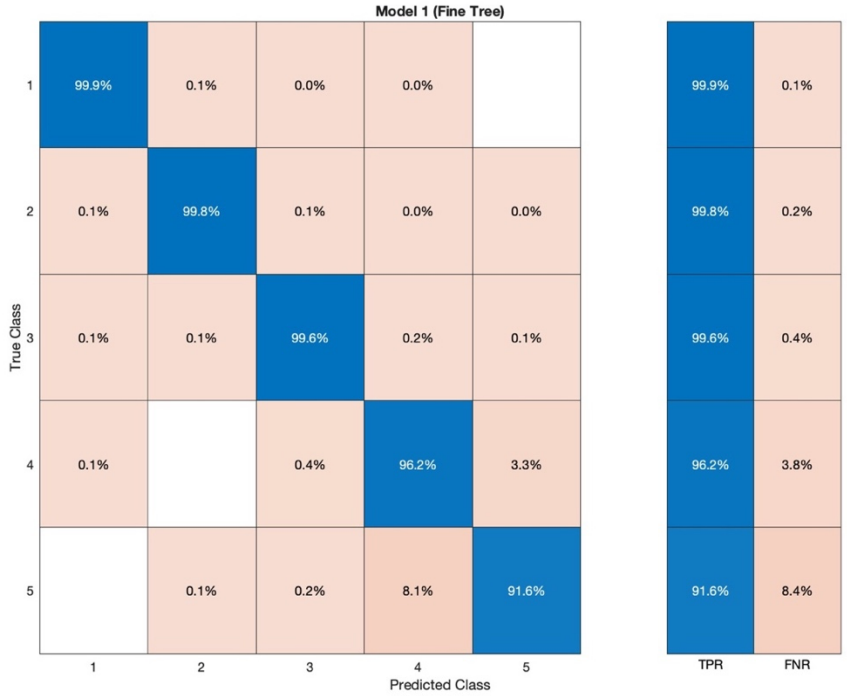


4. Training: Train the model using your data. The apps provide options for splitting the data into training and testing sets (or using cross-validation).





5. Evaluation: Assess the performance of your trained model using various metrics (accuracy, precision, recall, RMSE, etc.). The apps will often generate visualizations to help you understand the model's performance.

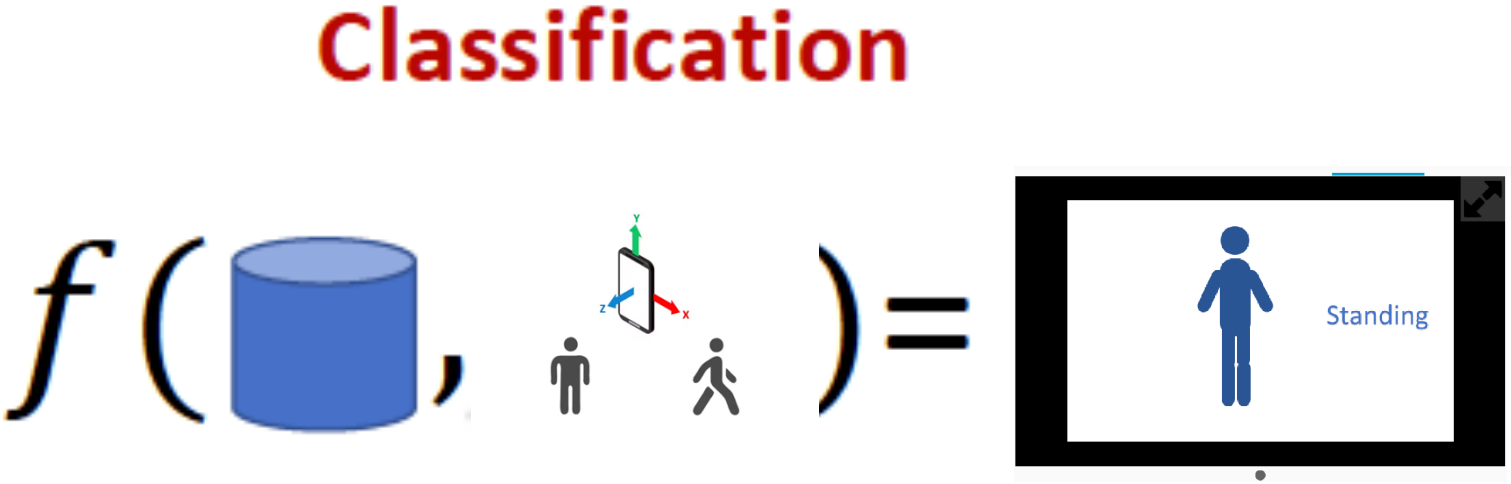


True Positive Rate (TPR): It measures the proportion of actual positive instances (true positives) that are correctly identified by the model. In other words, it tells you how well the model can identify positive cases.

A higher TPR means that the model is good at identifying positive cases, while a lower TPR suggests it might be missing many positives.

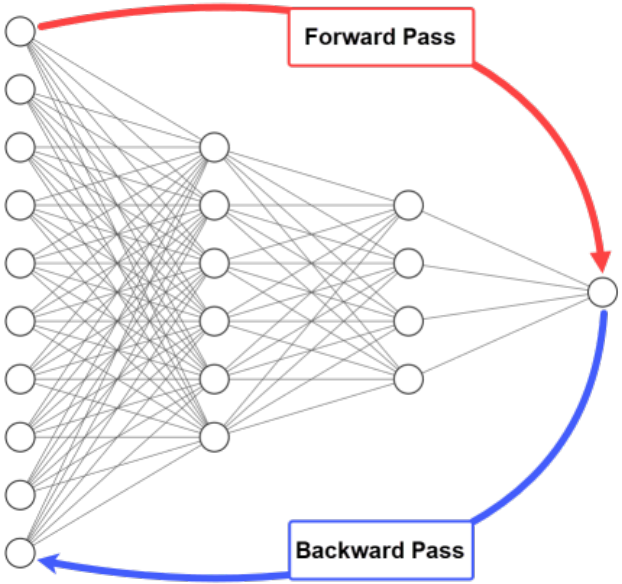
For example: In a medical test for a disease, TPR would represent the proportion of people who actually have the disease and are correctly identified as positive by the test.

6. Model Export: Once you're satisfied with a model, export it for use in other MATLAB applications.

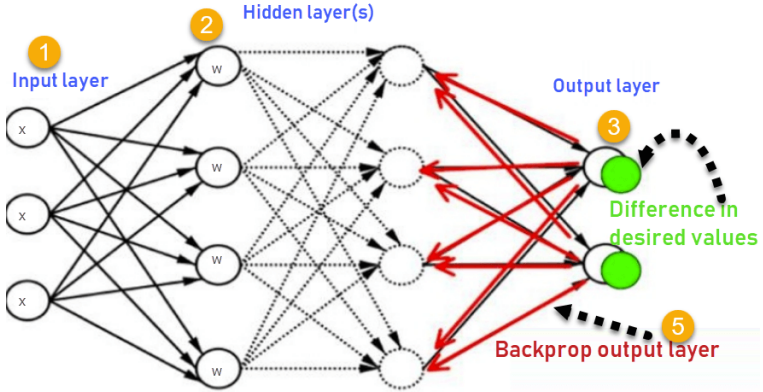


4. Training: Train the model using your data. The apps provide options for splitting the data into training and testing sets (or using cross-validation).

ANN training: Backpropagation



- Backpropagation is a machine learning technique essential to the optimization of artificial neural networks.
- It facilitates the use of gradient descent algorithms to update network weights, which is how the deep learning models driving modern artificial intelligence (AI) “learn”.



Thank you, All