Classification Tutorial

Stephen Coshatt[1] and Dr. WenZhan Song[1]

The University of Georgia[1]

Sensor Data Science and AI
Spring 2025

UNIVERSITY OF
GEORGIA

Outline

Table of Contents

Introduction

I. What is Classification?

II. Advantages & Disadvantages

III. Classification & Time Series Data

Table of Contents

Decision Trees

Decision trees are a flow chart like structure that are created using decisions rules inferred from the features of the data.

1. **Nodes**

   Nodes are features (attributes) of data. The top node of a tree is known as the "Root Node"

2. **Branches**

   Decision rules

3. **Leaves**

   Outcomes (labels)

Decision trees are easy to understand and easy to visualize.

Decision Trees

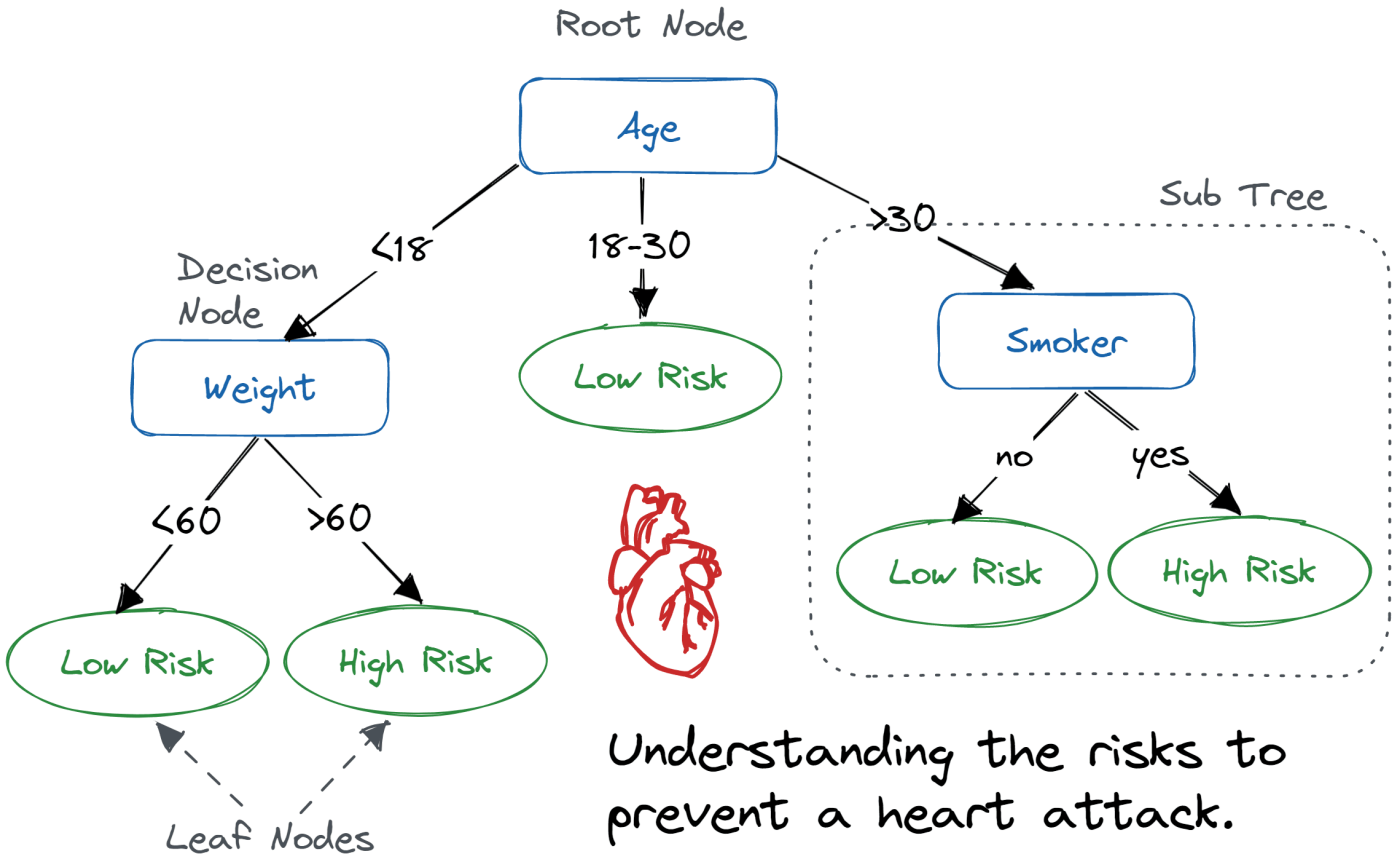

Figure: Decision Tree

Decision Algorithm

1. **Select the best attribute**

   There several Attribute Selection Measures that can be used such as Information Gain, Gain Ration, or Gini Index.

2. **Make selected attribute a decision node**

3. **Recursively build tree until one of the following conditions are met:**

   All tuples belong to the same attribute value

   No attributes remain

   No instances remain

Advantages

1. **Easy to understand & visualize (white box)**
2. **Requires little data preperation**
3. **Cost of using a tree is logarithmic**
4. **Can handle numeric and categorical data**
5. **Can handle multi-output problems**
6. **Can be validated using statistical tests**

Disdvantages

1. **Can create overly complex models that do not generalize well (overfitting)**
2. **Can be unstable due to small variations in data**
3. **Predictions are neither smooth nor continuous, but piecewise constant approximations**
4. **Cannot guarantee creation of the optimal algorithm**
5. **There are concepts that are hard to learn because decision trees do not express them easily, such as XOR**
6. **Create biased trees if some classes dominate**

Random Forest

Random Forest is a model composed of multiple decision trees. In machine learning, a model composed of multiple models is called an **ensemble**.

Each decision tree in a forest is created using a **random** subset of data and features. This technique is known as bagging.

A greedy search is used to select the values at which to split a feature.

Each tree classifies input independently, then the ensemble makes the final decision by voting.

Extra Trees

Extra Trees is an ensemble similar to Random Forest.

Unlike Random Forest, Extra Trees used the entire dataset to train the individual decision trees.

To ensure decision trees are sufficiently different, split values for each feature are randomly selected,

Extra Trees & Random Forest Parameters

1. **n_estimators**

   The number of decision trees to use in the ensemble

   More trees can improve performance, but increase computational time.

2. **max_depth**

   The maximum depth of each decision tree in the forest

   Too high of a depth can lead to overfitting while a too low depth can lead to underfitting.

3. **min_samples_leaf**

   The minimum number of samples required for a leaf node

   Higher values are less likely to overfit, while higher values create deeper and more specialized trees

Extra Trees & Random Forest Comparison

Extra Trees uses randomness to reduce variance

Extra Trees computational cost compared to Random Forest is much smaller

Extra Trees reduces bias compared to Random Forest because all the trees use the entire dataset, not just a random subset

Extra Trees excel with high dimension datasets

Extra Trees' randomness in tree building aids in preventing overfitting

Extra Trees' trains faster than Random Forests

Extra Trees can obtain high predictive accuracy, especially with large datasets

Extra Trees & Random Forest Comparison

Extra Trees and Random Forests have low interpertability

For both methods, small changes in Hyperparamters can create large changes in results

Random Forests are often more accurate than Extra Trees, except when features are reduced to the most essential, then they can perform about the same

Random Forests are less influenced by outliers than Extra Trees

Trees Tutorial Code

1. **Build Pipes**

decision_tree = skc.pipeBuild_DecisionTreeClassifier(criterion=['gini','entropy'], max_depth=[5, 10])

extra_trees = skc.pipeBuild_ExtraTreesClassifier(criterion=['gini','entropy'], n_estimators=[10], max_depth=[3, 5, 10],max_features=[1])

random_forest = skc.pipeBuild_RandomForestClassifier(criterion=['gini','entropy'], n_estimators=[10], max_depth=[3, 5, 10], max_features=[1])

Trees Tutorial Code

1. **Set Up Grid Seach**

   names=['Decision Tree', 'Extra Trees', 'Random Forest']

   pipes=[decision_tree, extra_trees, random_forest]

   skc.gridsearch_classifier(names=names, pipes=pipes, X_train=X_train, X_test=X_test, y_train=y_train, y_test=y_test, plot_number=4)

Classifiying Waveforms with Trees

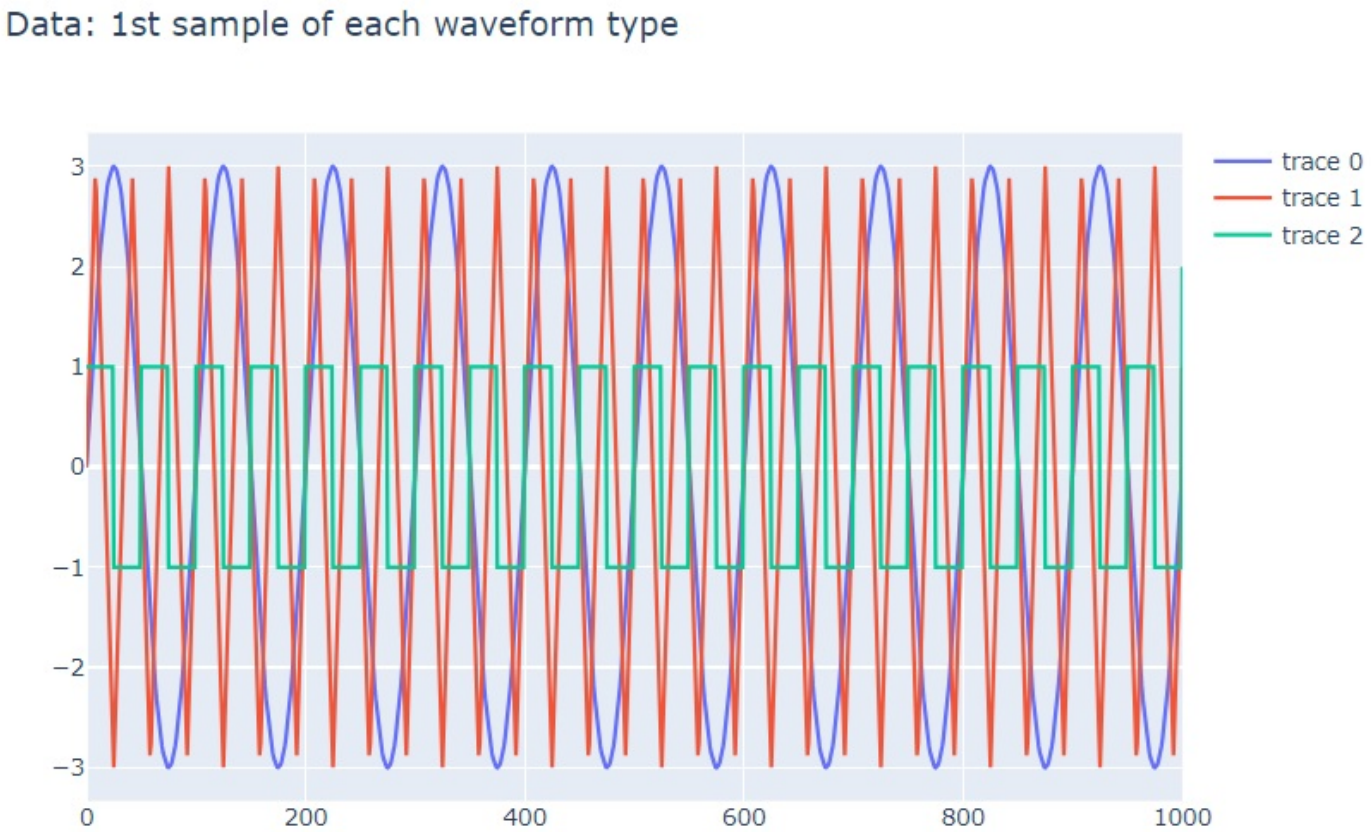

Figure: Sine, Square, & Triangle Waves

Classifiying Waveforms with Trees

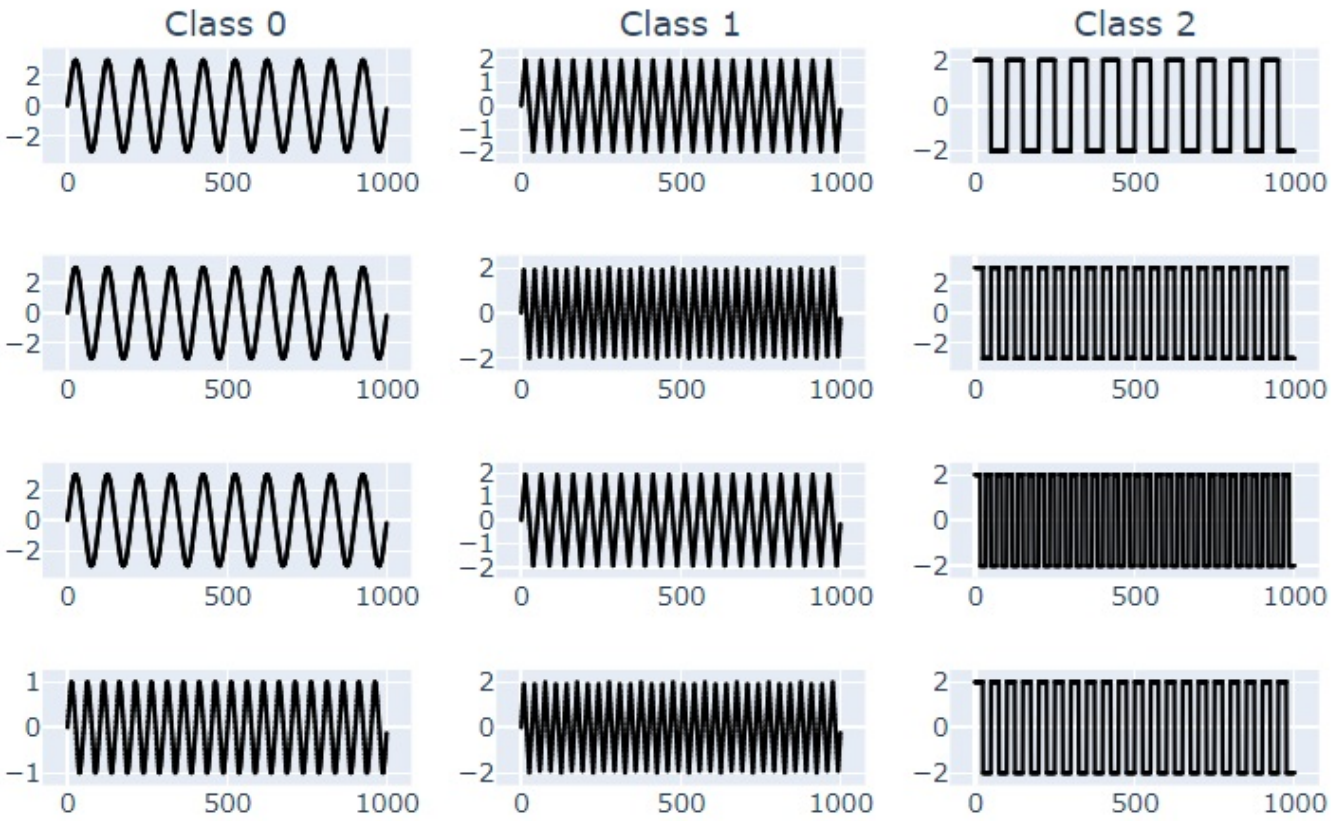

Figure: Waveforms Classified

Classifying Waveforms with Trees

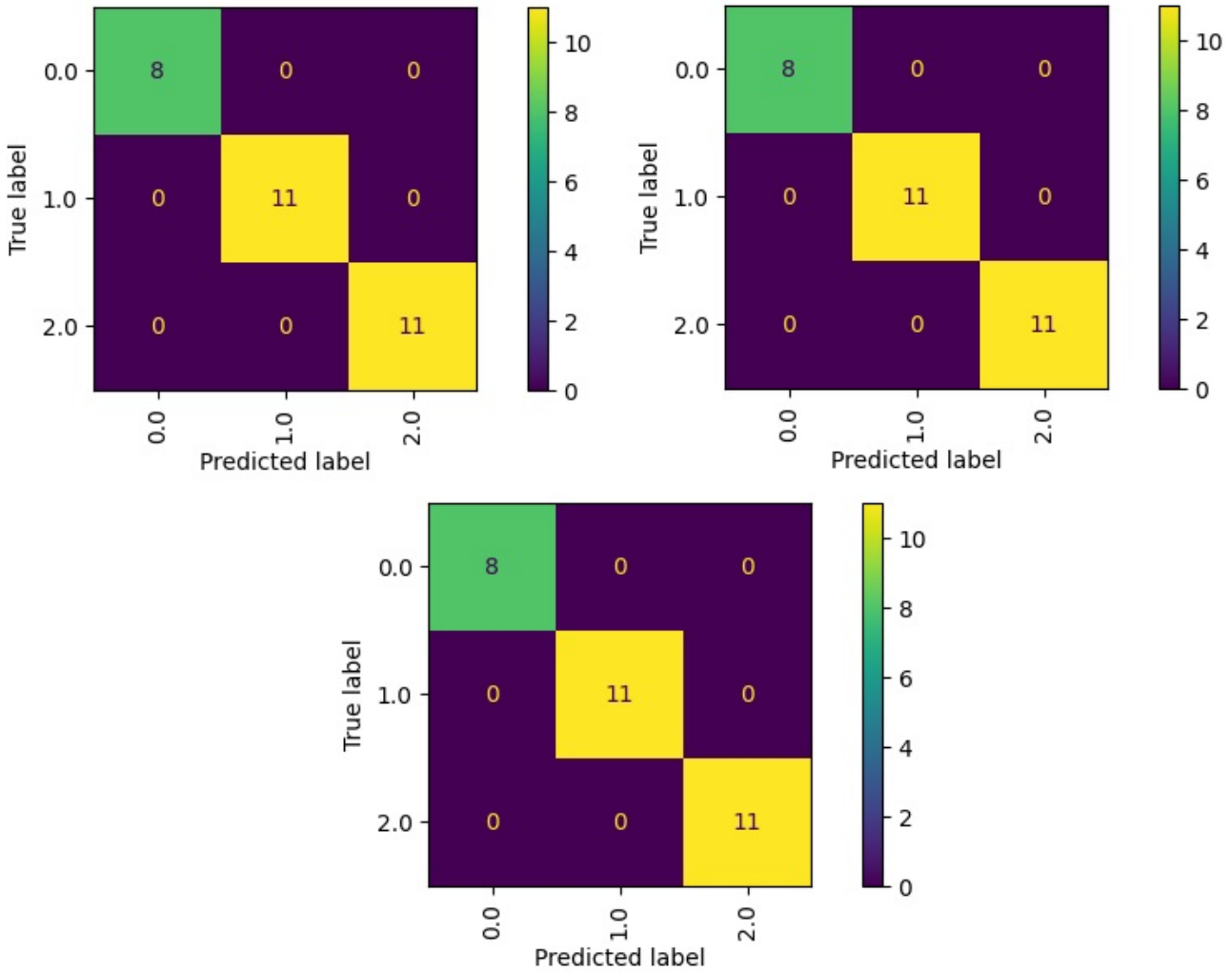

Figure: Confusion Matrices

Table of Contents

Nearest Neighbors

1. **Nearest Neighbors**

   Nearest Neighbors algorithms classify a data point by determining which class a majority of the points neighbors belong to

   There are several methods for determining neighbors

K Nearest Neighbors

1. **KNN**

   Assumption: "Similar things exist in close proximity"

   Estimates the likelihood of a data point being a member of a group or another depending on the data points nearest to it are in

   Note: There is a "Time Series" version of this algorithm avaliable in the TS Learn package. It is the same as the standard KNN, but with distance metrics more suited for times series data, such as Dynamic Time Warping.

K Nearest Neighbors

1. **KNN Algorithm**

   1. Load data

   2. Initialize **K** to your chosen # of neighbors

   3. For each datapoint: Calculate the distance between querry example & all current examples in the dataset

   4. Then add the distance & index of the example to a ordered collection

   5. Sort collection in ascending order

   6. Select the first **K** entries

   7. If regression, return the mean of the **K** labels

   7. If classification, return the mode of the **K** labels

Nearest Centroid

Nearest Centroid is one of the simplest machine learning algorithms for classification.

1. **Algorithm**

   1. Load Training Data

   2. Compute the centroid of each class in the training data

   3. For each new data point, calculate its distance from the centroid of each class

   4. Select the class with the smallest distance from the data point and assign it that class's label

Radius Nearest Neighbors

Radius NN works the same as the KNN algrothim, except it compares a data sample to all neighbors within a specified radius instead of a fixed number.

1. **Algorithm**

   1. Load data

   2. Select the length of your **radius**

   3. For each data point: Calculate the distance between query example & all current examples in the dataset

   4. Then add all points with a distance equal to or less than the specified **radius** to a collection

   5. If regression, return the mean of the collection's labels

   5. If classification, return the mode of the collection's labels

Nearest Neighbors Pros & Cons

Pros:

Fast. In particular, KNN is a **Lazy Learner**, which has no training period.

Easy to implement

Easy to understand

Useful for both Classification & Regression

Cons:

Does not work well with large datasets

Does not work well with high dimensions

Requires feature scaling

Sensitive to noise

Nearest Neighbors Tutorial Code

1. **Build Pipes**

knn = skc.pipeBuild_KNeighborsClassifier(n_neighbors=[3,5], weights=['uniform'], algorithm=['auto'])

ncent = skc.pipeBuild_NearestCentroid()

rad = skc.pipeBuild_RadiusNeighborsClassifier(radius=[50])

tsknn = skc.pipeBuild_KNeighborsTimeSeriesClassifier(n_neighbors=[3,5], weights=['uniform'], metric=['dtw','softdtw'])

Nearest Neighbors Tutorial Code

1. **Set Up Grid Search**

   names=['K Nearest Neighbors', 'Nearest Centroid', 'Radius Neighbors', 'Time Series KNN']

   pipes=[knn, ncent, rad, tsknn]

   skc.gridsearch_classifier(names=names, pipes=pipes, X_train=X_train, X_test=X_test, y_train=y_train, y_test=y_test, plot_number=3)

Nearest Neighbors Confusion Matrix

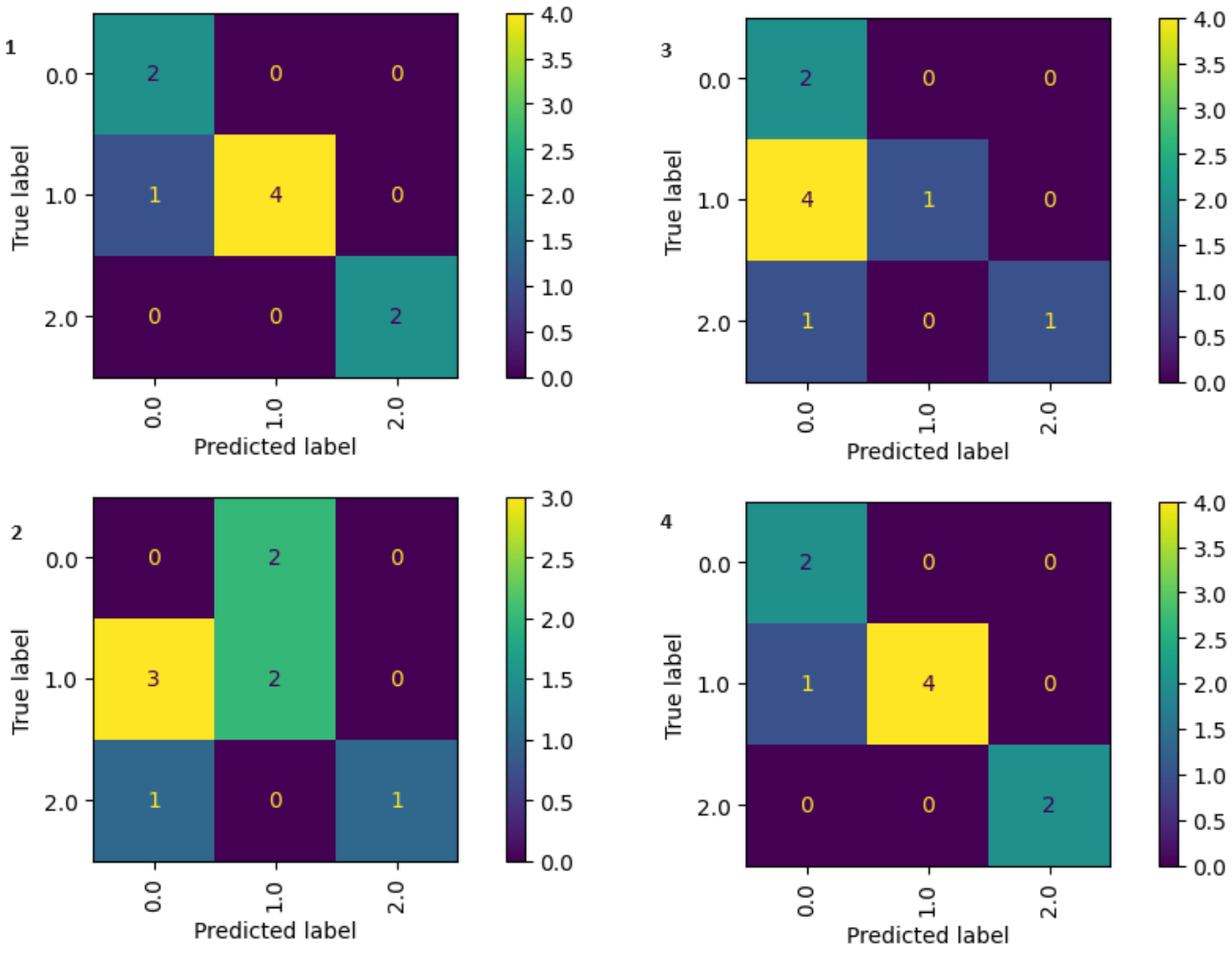

Figure: Confusion Matrices

Table of Contents

Support Vector Machines

Support Vector Machines (SVM) can be used for classification and regression.

SVMs construct a **hyperplane** in multidimensional space to separate different classes.

The **hyperplane** is iteratively constructed using error minimization to find the optimal hyperplane.

Support Vector Machine Terminology

1. **Support Vectors**

   A set of data points closest to the hyperplane.

2. **Hyperplane**

   A decision plane which separates objects by class.

3. **Margin**

   The gap between a hyperplane and support vectors. The distance is measured perpendicular from the plane to the vector.

4. **Maximum Marginal Hyperplane**

   The hyperplane that gives the best division of classes. The larger the margin, the better.

5. **Kernel**

   The kernel is the method by which an SVM transforms lower dimensional data into higher dimensional data. Hyperplanes are created in the higher dimensional space. There are multiple kernels available.

SVMs Illustrated

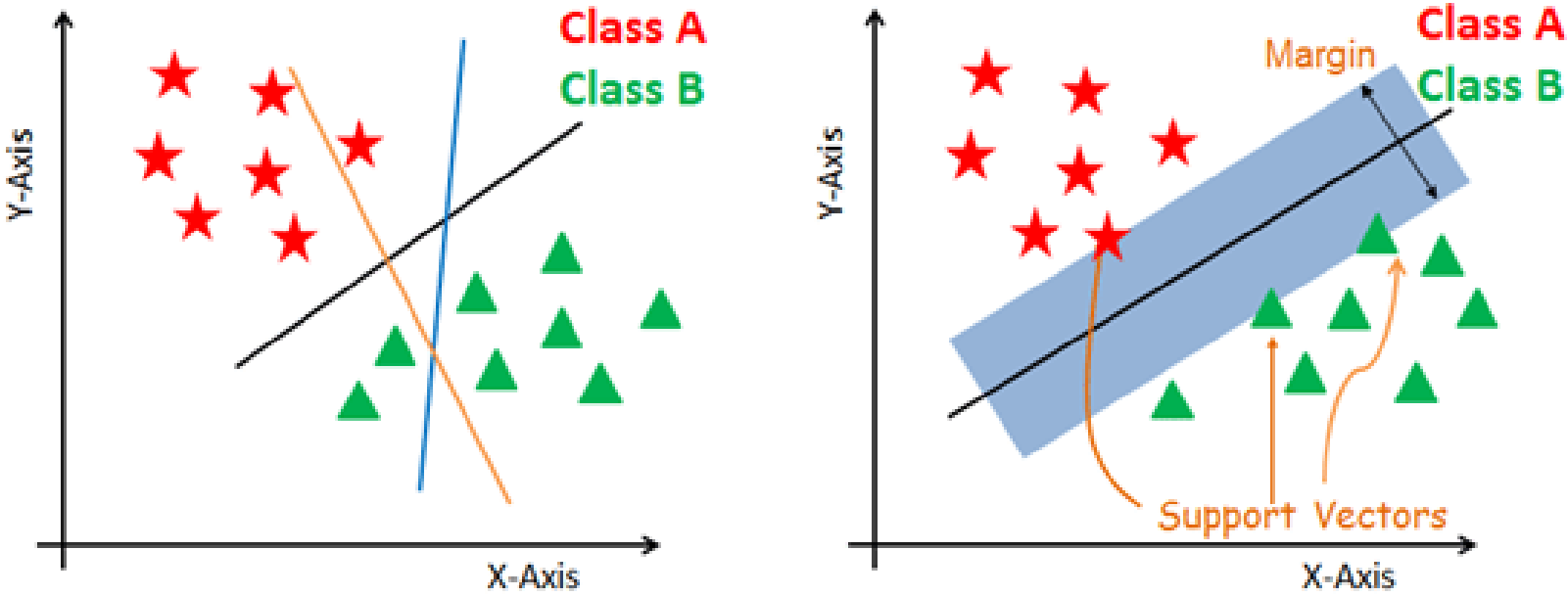

Figure: Simple SVM Illustration

SVMs Illustrated

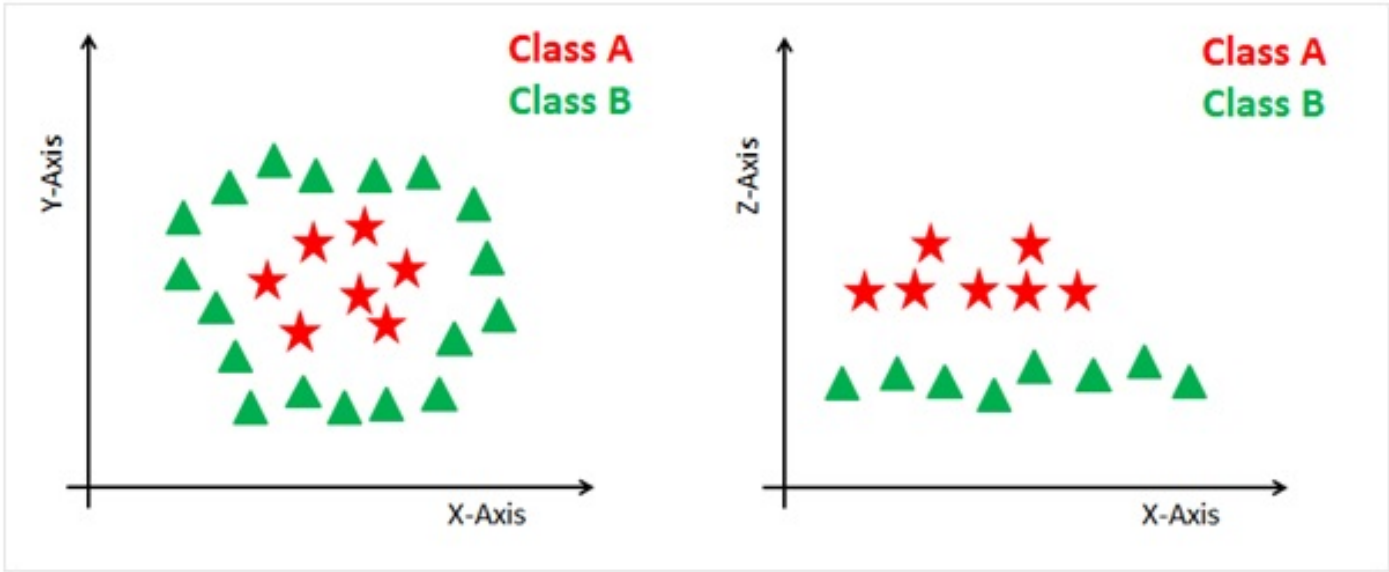

Figure: Simple Kernel Illustration

SVM Kernels

1. **Linear**

$$k(x, y) = x^T y$$

2. **Polynomial**

$$k(x, y) = (\gamma x^T y + c_0)^d$$

3. **Gaussian Radial Basis Function (RBF)**

$$k(x, y) = exp\left(-\gamma \| x - y \|^2\right)$$

SVM Kernels

1. **Sigmoid**

$$k(x, y) = tanh(\gamma x^T y + c_0)$$

2. **Global Alignment Kernal (GAK)**

$$k(x, y) = \sum_{\pi \in A(x,y)} \prod_{i=1}^{|\pi|} exp\left(-\frac{\|x_{1\pi_1(i)} - y_{\pi_{2j}}\|^2}{2\sigma^2}\right)$$

GAK Kernel

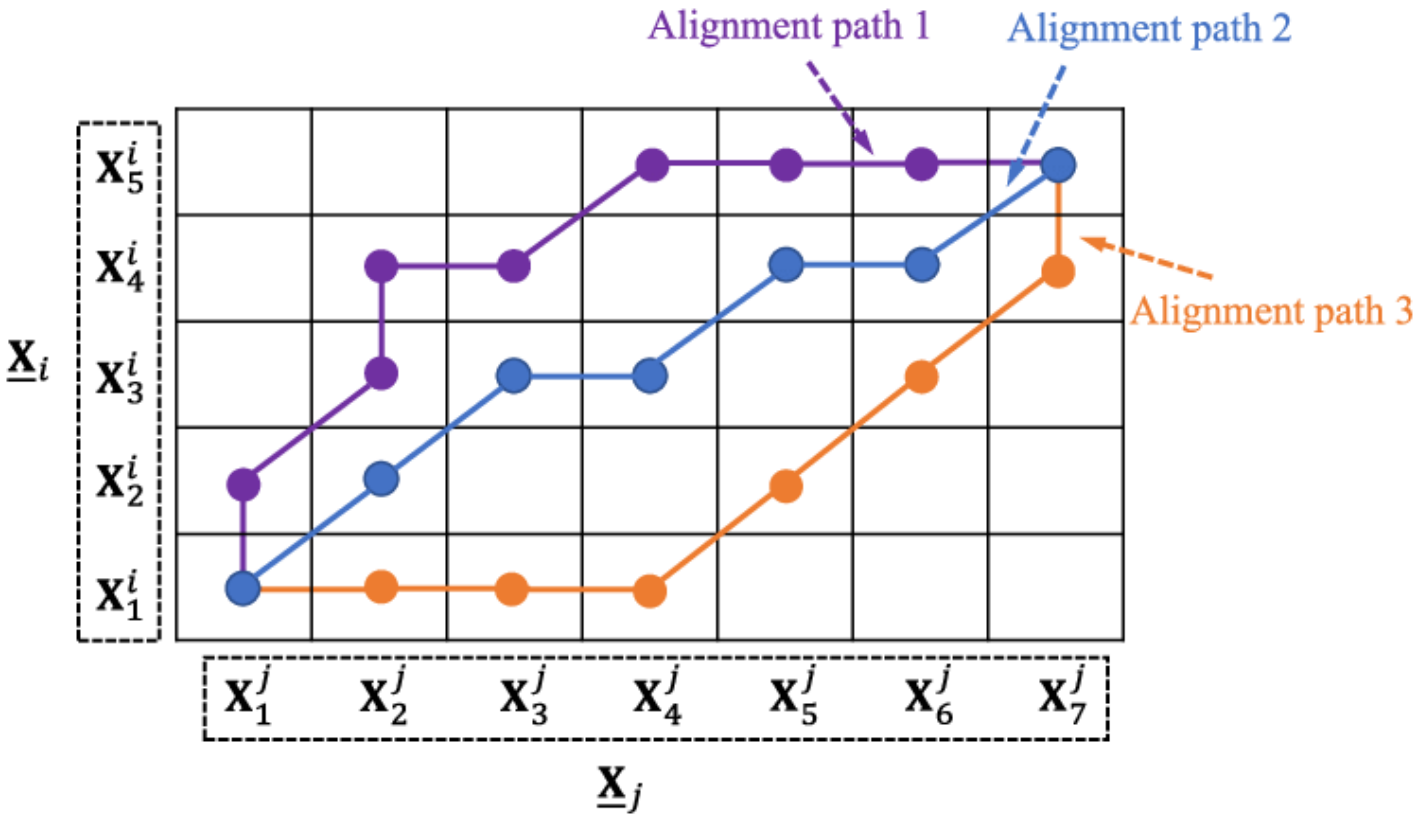

Figure: GAK Kernel Illustration

SVM Pros & Cons

1. **Pros**

   Work well when there is a clear margin of seperation

   More effective in higher dimensional space

   Effective in cases where there are more dimensions than the number of samples

   Relatively memory efficient

2. **Cons**

   Doesn't work well with large data sets

   Doesn't perform well with a lot of noise

   Under-performs when the number of features exceeds the number of samples

   Difficult to explain decision process

SVM Tutorial Code

1. **Build Pipes**

svc = skc.pipeBuild_SVC(C=[1.0], kernel=['linear'], degree=[3],gamma=['scale'], tol=[1.0e-3], random_state=None)

nusvc = skc.pipeBuild_NuSVC(nu=[0.5], kernel=['rbf'], degree=[3], gamma=['scale'], tol=[1.0e-3],random_state=None)

tssvc = skc.pipeBuild_TimeSeriesSVC(kernel=['gak'])

SVM Tutorial Code

1. **Setup Grid Search**

   names = ['SVClassifier', 'Nu-SVClassifier']

   pipes = [svc, nusvc]

   skc.gridsearch_classifier(names=names, pipes=pipes, X_train=X_train, X_test=X_test, y_train=y_train, y_test=y_test, plot_number=4)

SVM Confusion Matrices

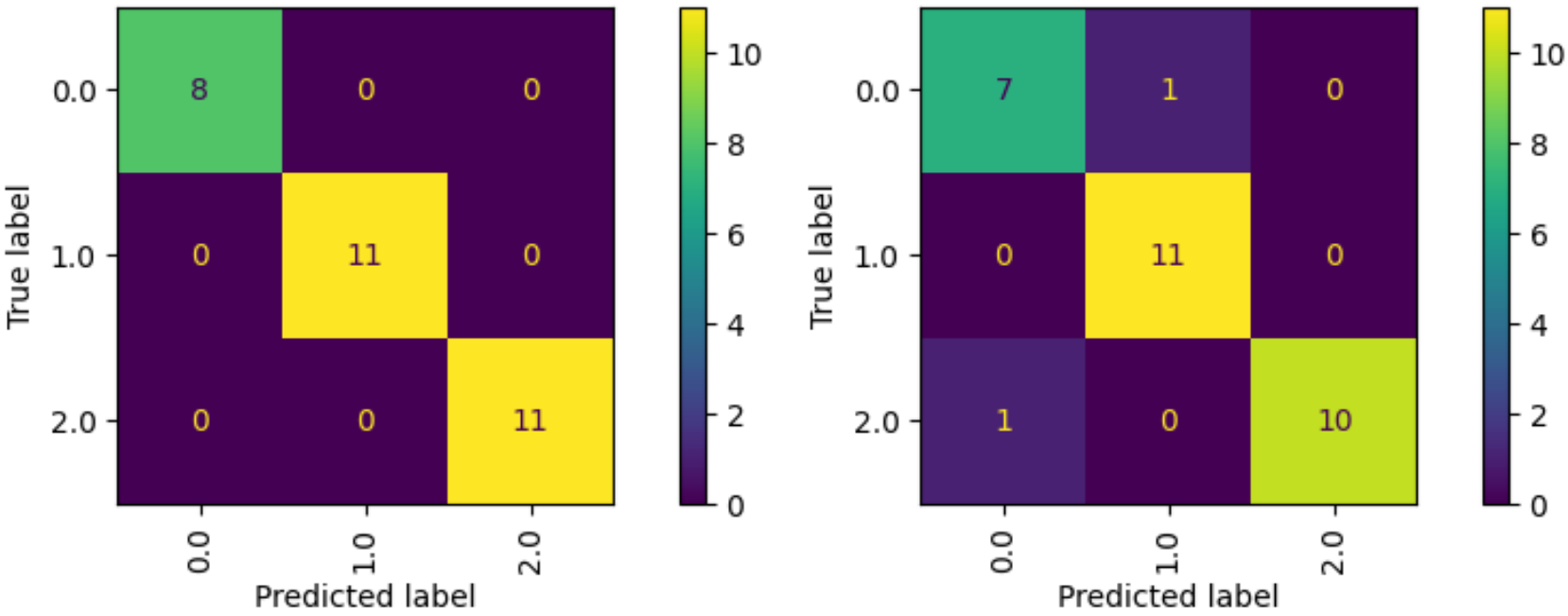

Figure: SVM & NuSVM Results

Table of Contents

Overview

1. **Weak Learners**

   An algorithm that predicts slightly better than random.

2. **Boosting**

   An ensemble of **sequential** weak learners combined for and accurate algorithm.

   Predictions are made iteratively for "T" rounds on the entire training set and each iteration improves performance using information from the previous iteration.

Overview

1. **Bagging**

   An ensemble of non-sequential weak learners combined for and accurate algorithm. This method is also known as **bootstrapping**

   Random subsets of training data are drawn for "T" rounds. Each draw is independent. Each draw is used to create a weak learner.

   The weak learners run in parallel after training, and a vote is taken to determine the class of new input. Voting may or may not be weighted.

Bagging Chart

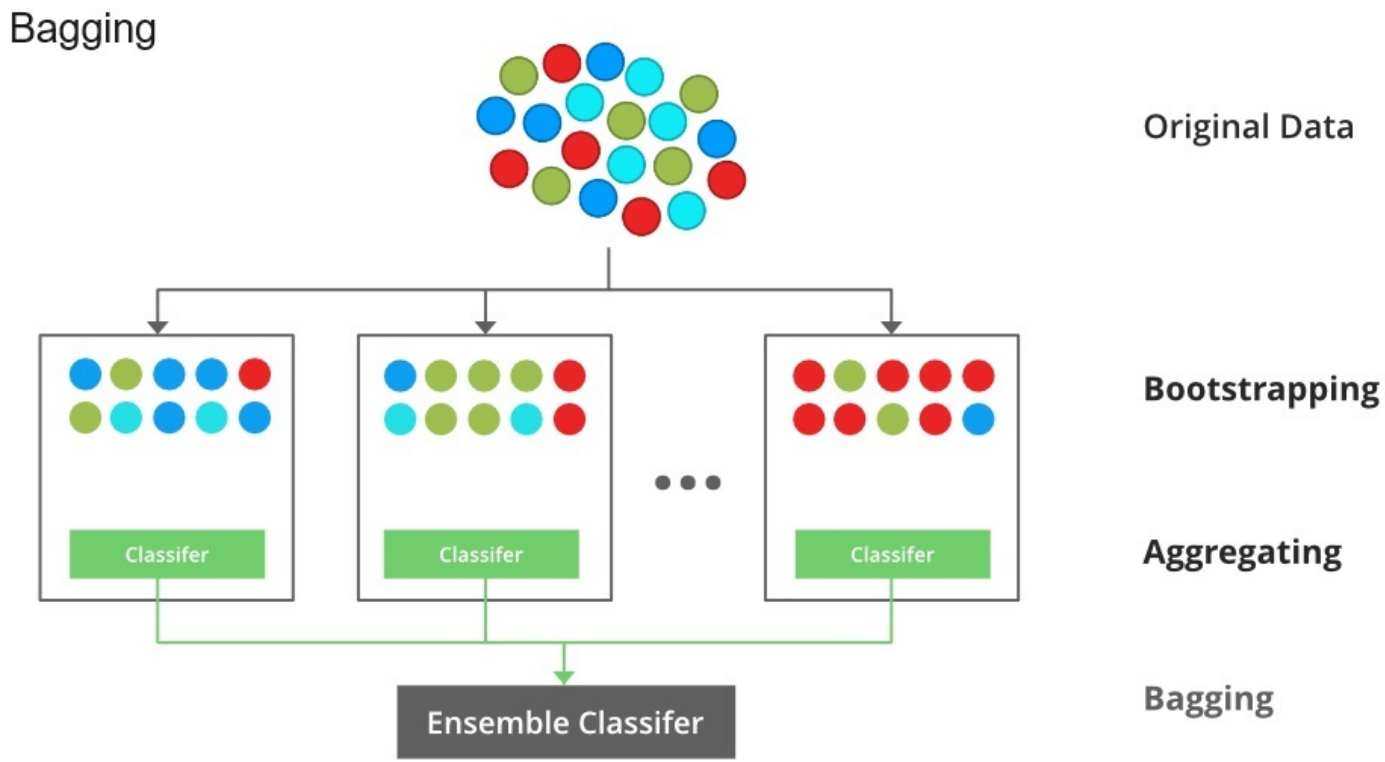

Figure: Bagging [Ref]

Boosting Chart

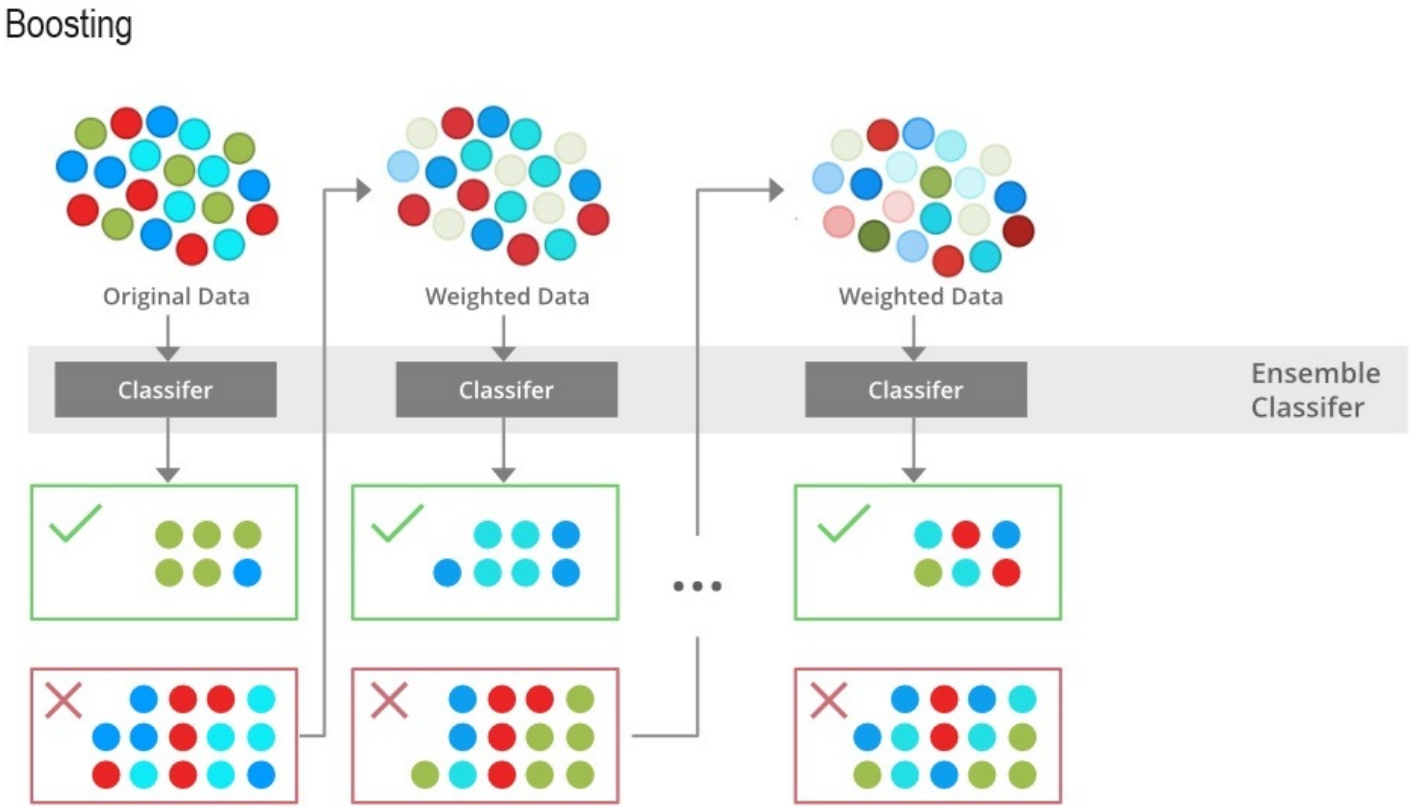

Figure: Boosting [Ref]

AdaBoost

AdaBoost is composed of sequentially grown trees.

It "punishes" incorrectly predicted samples by assigning a larger weight to them after each round during of prediction.

In **SKLearn**, AdaBoost defaults to using decision trees, but other classification algorithms may be used with it.

AdaBoost Algorithm

1. **For t in T rounds**

   1. Normalize the weight vector **w** to calculate distribution **p**. **w** initial weight is **1/N**, where **N** is the number of labeled examples

   2. Grow a decision tree using the distribution **p** then return hypothesis **h** with prediction values for each example

   3. Calculate error term **epsilon** of **h**

   4. Assign $\beta = \epsilon \frac{\epsilon}{1-\epsilon}$

   5. Update weights to **w = w * B** such that poor predictions will have higher weights and good predictions have lower weights

AdaBoost Chart

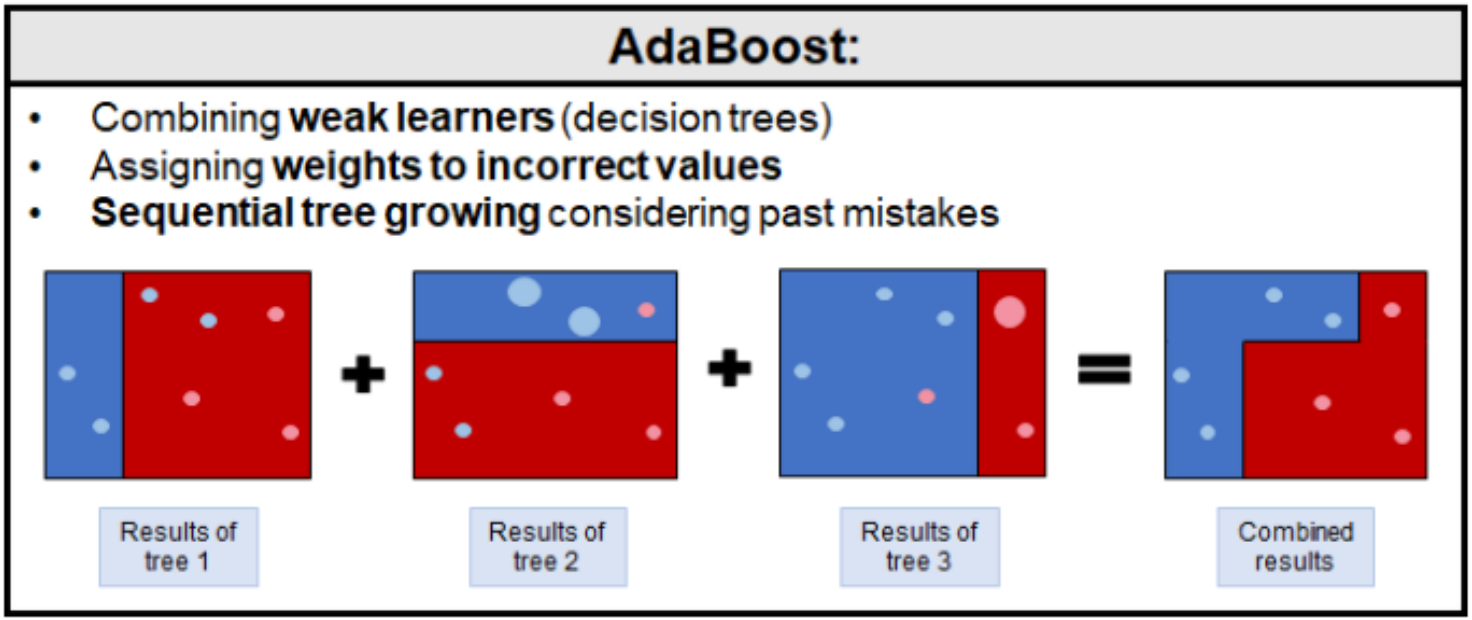

Figure: AdaBoost [Ref]

AdaBoost Pros & Cons

1. **Pros**

   Relatively robust to over-fitting on low noise data sets

   Few hyper-parameters for easy tuning

   Easy to understand & visualize

2. **Cons**

   Debate among experts as to whether or not it generalizes will with noisy data

   When irrelevant features are in the model, AdaBoost performs worse than Random Forests & Extra Trees

   Not optimized for speed

Other Bagging & Boosting Models

1. **Gradient Boosting**

   Gradient Boosting is essentially AdaBoost with Decisions Trees. It's an optimized version specifically for trees.

2. **SKLearn Bagging Classifier**

   This is a generic baggling classifier. With it, you may use the boot strapping approach with any model

Bagging Pros & Cons

1. **Pros**

   Robust against noise.

   Easy & effective way to convert weak learners into strong learners

   Reduces variance & over-fitting for more accurate models

2. **Cons**

   Requires many comparable classifiers

   Can result in under-fitting if not trained properly

   Computationally more expensive as the number of parallel models increases

Bagging & Boosting Tutorial Code

1. **Build Pipes**

decision_tree = skc.pipeBuild_DecisionTreeClassifier(criterion=['gini', 'entropy'], max_depth=[5, 10])

xtree = skc.pipeBuild_ExtraTreesClassifier(n_estimators=[50,100])

random_forest = skc.pipeBuild_RandomForestClassifier(criterion=['gini','entropy'], n_estimators=[50,100], max_depth=[3, 5, 10], max_features=[1])

ada = skc.pipeBuild_AdaBoostClassifier(n_estimators=[50,100],learning_rate=[1.0])

bag = skc.pipeBuild_BaggingClassifier(n_estimators=[50,100])

gb = skc.pipeBuild_GradientBoostingClassifier(n_estimators=[50,100])

Bagging & Boosting Tutorial Code

1. **Set Up Grid Search**

   names = ['Decision Tree', 'Extra Trees', 'Random Forest', 'Adaboost', 'Bagging', 'Gradient Boosting']

   pipes = [decision_tree, xtree, random_forest, ada, bag, gb]

   skc.gridsearch_classifier(names=names, pipes=pipes, X_train=X_train, X_test=X_test, y_train=y_train, y_test=y_test, plot_number=3)

Bagging & Boosting Confusion Matrices

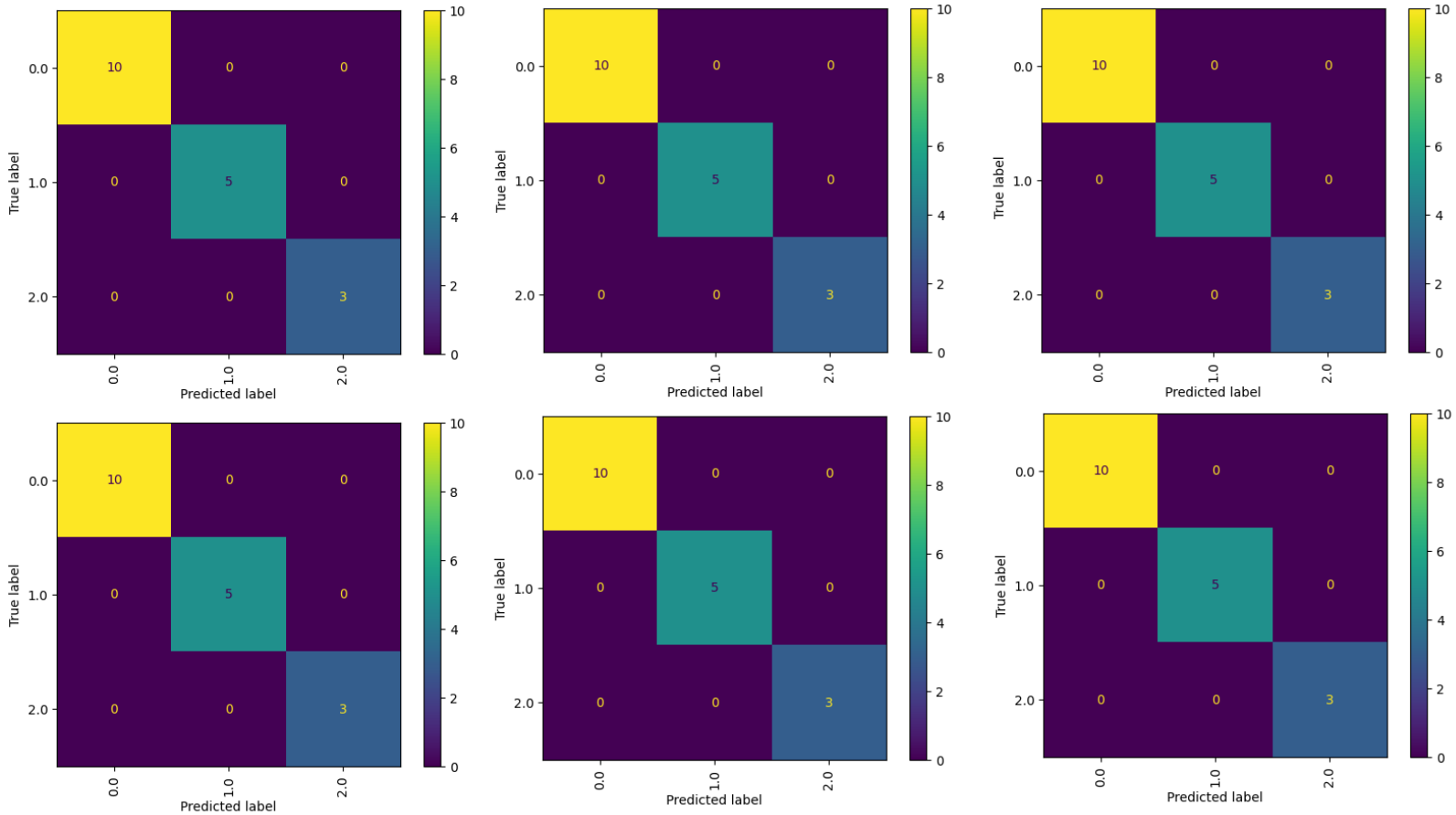

Figure: Tree Comparisons

Table of Contents

Naive Bayes

Naive Bayes algorithms are probabilistic models based on Bayes Theorem, which is base on conditional probability.

In Bayes Theorem, probability is defined as the likelihood of an event occuring. P(A/B) is the conditional probability of A given B.

We define the conditional probability as

$$P(A|B) = P(A \cap B)P(B)$$

Where $P(A \cap B)$ is the joint probability

It's called **"naive"** because it assumes that all features are independent of each other.

Types of Naive Bayes

1. **Gaussian Naive Bayes** - Used with continuous predictor values that are expected to have a Gaussian Distribution. Attributes are segmented based on output classes and the variance & mean are calculated for each class. [ref]
2. **Bernoulli Naive Bayes** - Use with boolean predictors (follow Bernoulli Distribution). The features are distributed according to multivariate Bernoulli distributions [ref].
3. **Multinomial Naive Bayes** - Used when the predictors follow a Multinomial Distribution. [ref]

NB Example

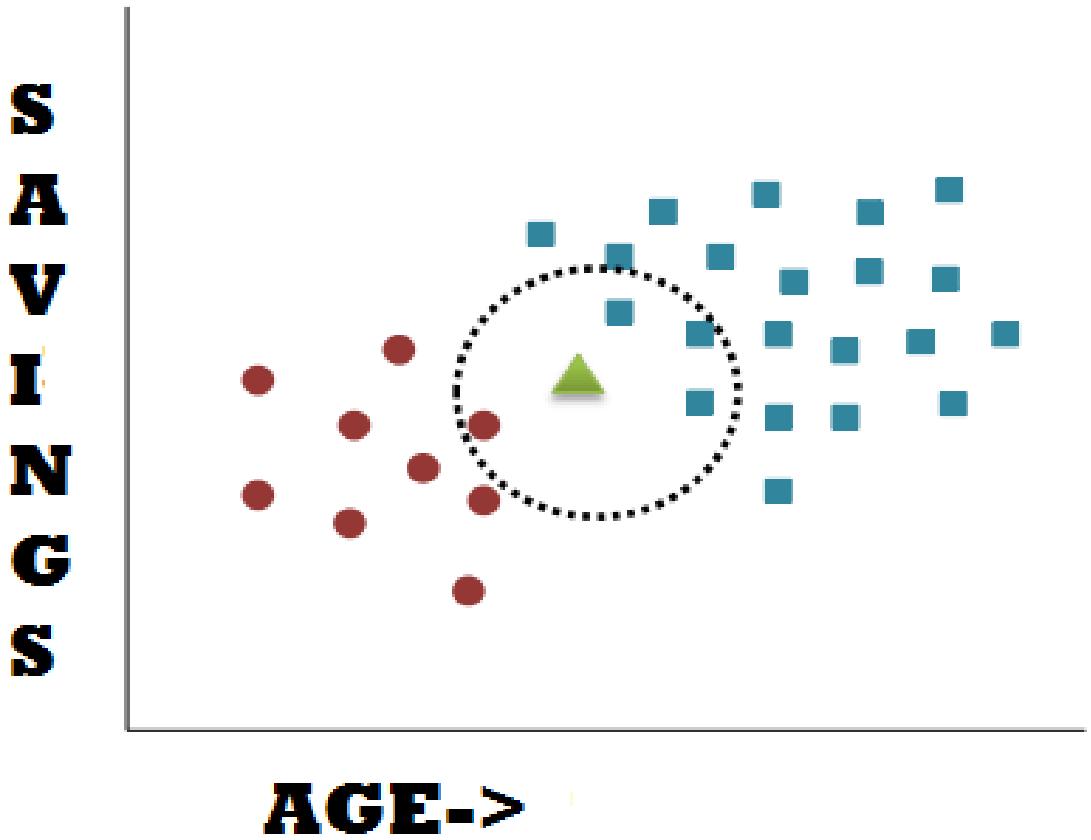

Figure: Naive Bayes Example: Bike Ownership

NB Algorithm

1. Calculate Probabilities of classes:

   P(no bike) = 9/30 & P(bike) = 21/30
2. New Data Point, find similar points within a designated radius.
3. Calculate marginal probability:

   P(point) = 4/30
4. Calculate Posterior probabilities:

   P(Point No Bike) = 1/4 = 0.25

   P(Point Bike) = 3/4 = 0.75
5. Classify new point with highest posterior probability.

NB Pros & Cons

Pros:

1. Fast
2. Useful with multi-class problems
3. Works with numerical & categorial data

Cons:

1. Can't make predictions on categorical variables not in training set
2. Can be a very bad estimator
3. Assume all features are independent

NB Code

```
bnb = skc.pipeBuild_BernoulliNB()

gnb = skc.pipeBuild_GaussianNB()

names=['Bernoulli NB','Guassian NB']

pipes=[bnb,gnb]

skc.gridsearch_classifier(names=names, pipes=pipes, X_train=X_train,
X_test=X_test, y_train=y_train, y_test=y_test, plot_number=3)
```

NB Results

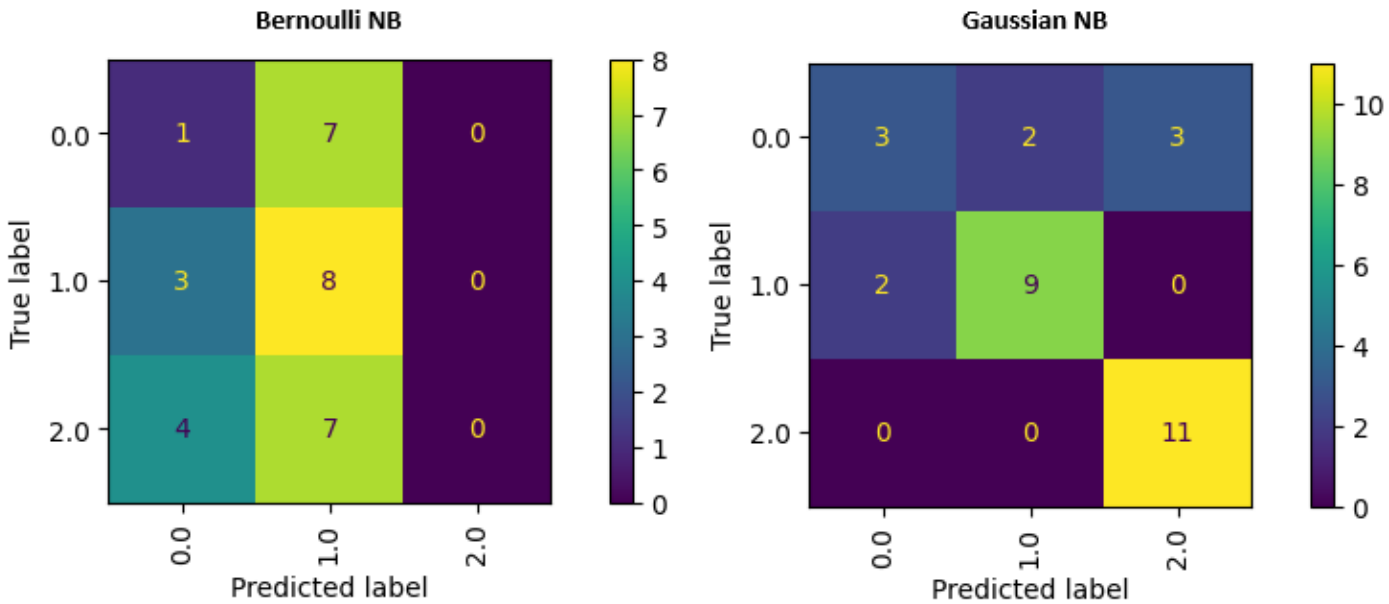

Figure: Bernoulli & Gaussian NB Confusion Matrices

Discriminant Analysis

Discriminant Analysis performs classification by projecting data onto a lower-dimensional space while attempting to maximize the distance between classes.

Types discussed in this section:

1. Quadratic Discriminant Analysis (QDA)
2. Linear Discriminant Analysis (LDA)

Note: LDA & Gaussian Naive Bayes are actually special cases of QDA.

Reference: Linear and Quadratic Discriminant Analysis

DA Continued

$P(X|y=k)$ for each class k, predictions are obtained using Bayes' rule for each training sample $x \in R^d$

$$P(y=k|x) = \frac{P(x|y=k)P(y=k)}{P(x)} = \frac{P(x|y=k)P(y=k)}{\sum_l P(x|y=l) \cdot P(y=l)}$$

select the class k which maximizes the posterior probability.

Reference: Linear and Quadratic Discriminant Analysis

LDA & QDA

Specifically for LDA & QDA, $P(x|y)$ is modeled as a multivariate Gaussian distribution with

$$P(x|y=k) = \frac{1}{(2\pi)^{d/2}|\Sigma_k|^{1/2}} exp\left(-\frac{1}{2}(x-\mu_k)^t \Sigma_k^{-1}(x-\mu_k)\right)$$

where d is the number of features

Reference: Linear and Quadratic Discriminant Analysis

QDA

Thus, the log of the posterior (QDA) is:

$$logP(y = k|x) = logP(x|y = k) + logP(y = k) + C_{st}$$

$$= -\frac{1}{2}log|\Sigma_k| - \frac{1}{2}(x - \mu_k)^t\Sigma_k^{-1}(x - \mu_k) + logP(y = k) + C_{st}$$

where constant C_{st} corresponds to denominator $P(x)$. The class prediction is the one that maximizes this equation

Note: If we assume that the covariance matrices are diagnol (i.e. the features are independent), the QDA becomes the Gaussian Naive Bayes classifier.

Reference: Linear and Quadratic Discriminant Analysis

LDA

LDA is a special case of QDA. If we assume each class shares the same covariance matrix: $\sum_k = \sum$ for all k

$$logP(y = k|x) = -\frac{1}{2}(x - \mu_k)^t \Sigma_k^{-1}(x - \mu_k) + logP(y = k) + C_{st}$$

where $-\frac{1}{2}(x - \mu_k)^t \Sigma_k^{-1}(x - \mu_k)$ is the Mahalanobis Distance between x and the mean μ_k

Reference: Linear and Quadratic Discriminant Analysis

LDA

Thus the log-posterior of LDA is:

$$logP(y = k|x) = \omega_k^t x + \omega_{k0} + C_{st}$$

where $\omega_k = \Sigma^{-1}\mu_k$ and $\omega_{k0} = -\frac{1}{2}\mu_k^t\Sigma^{-1}\mu_k + logP(y = k)$.

Reference: Linear and Quadratic Discriminant Analysis

QA Pros & Cons

Pros:

1. LDA: Simple & Fast [ref]
2. QDA: Flexible due to quadratic decision boundaries
3. QDA: Handles imbalance and/or small data sets well

Cons:

1. LDA: Requires normal distribution on features [ref]
2. LDA: Performance degrades as class categories increase [ref]
3. QDA: Assumes Gaussian distribution for each class
4. QDA: Sensitive to outliers
5. QDA: Requires sufficient training data

LDA & QDA Code

lda = skc.pipeBuild_LinearDiscriminantAnalysis()

qda = skc.pipeBuild_QuadraticDiscriminantAnalysis(priors=[None], reg_param=[0.0], store_covariance=[False], tol=[1.0e-4])

names=['Linear Discriminant Analysis','Quadratic Discriminant Analysis']

pipes=[lda,qda]

skc.gridsearch_classifier(names=names, pipes=pipes, X_train=X_train, X_test=X_test, y_train=y_train, y_test=y_test, plot_number=3)

LDA & QDA Results

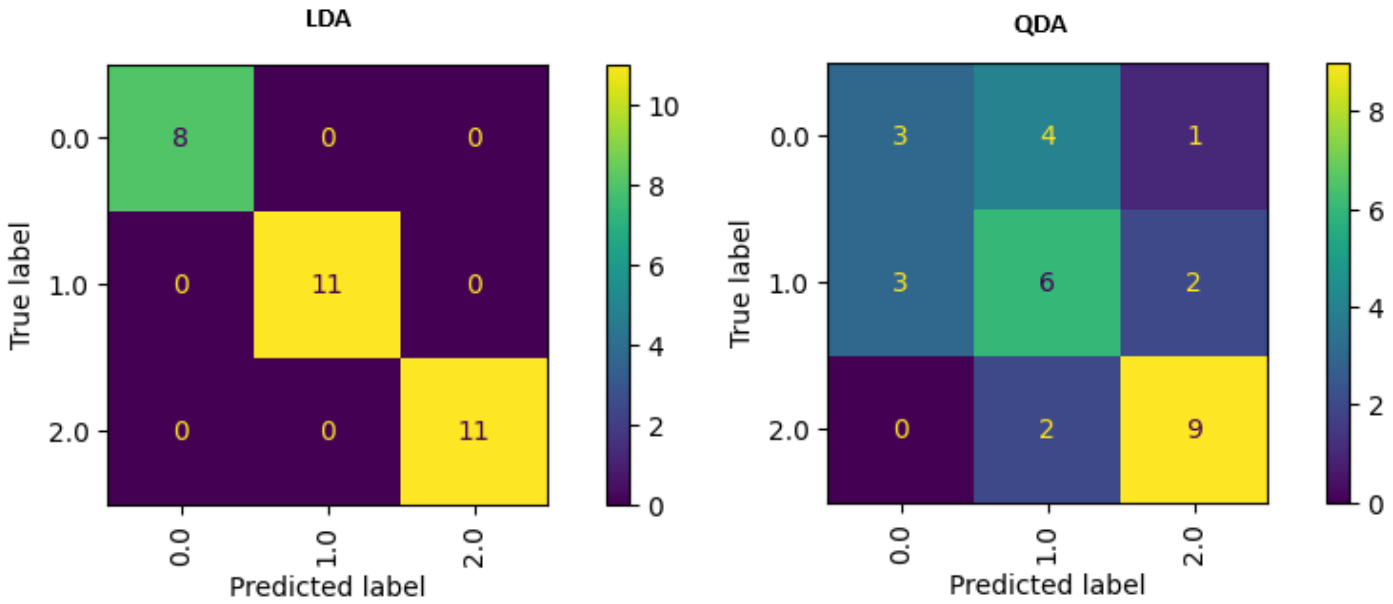

Figure: LDA & QDA Confusion Matrices

Non-Myopic Early Classification

Early Classification for time series makes the assumption that classification can be reliable made base on a subsequence of a given time series.

Early Classification on Time Series (ECTS) essentially uses a 1 nearest neighbor approach.

"The problem is expressed differently by different researchers, but it generally reduced to asking if we can classify a time series subsequence with sufficient accuracy and confidence after seeing only some prefix of a target pattern." [ref]

Non-Myopic Early Classification

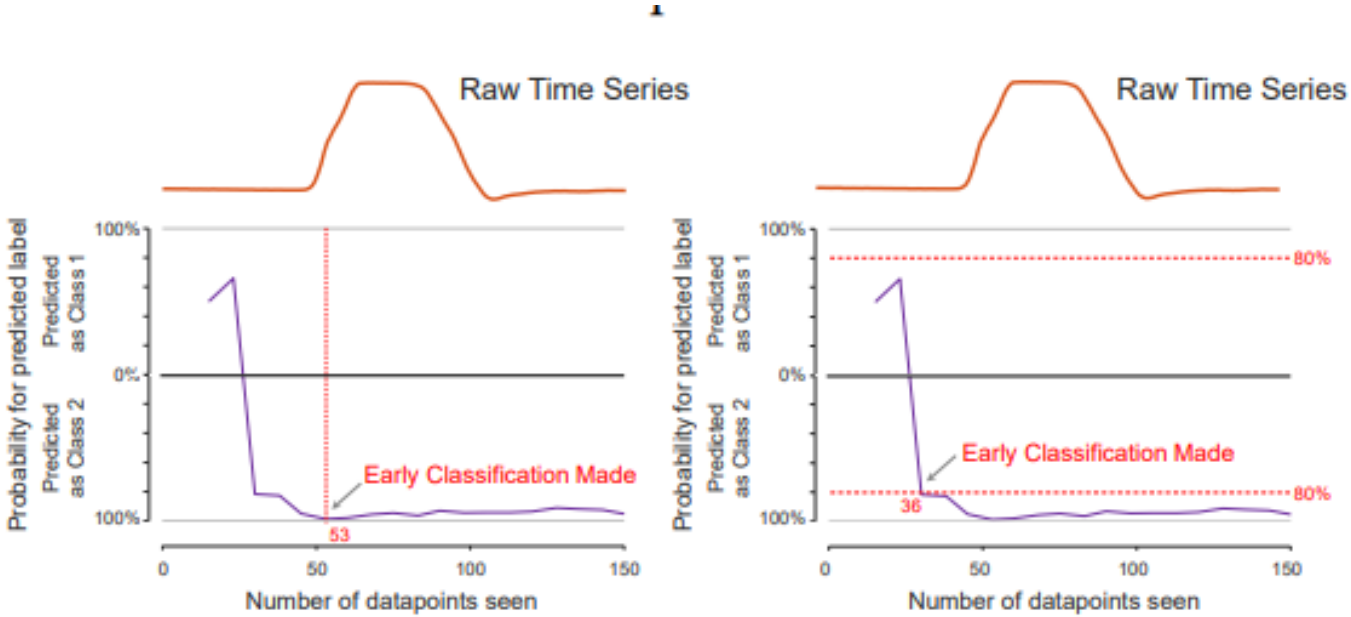

Figure: (*Left*) The model correctly predicts the class of an exemplar after seeing only 53 data points. (*right*) Other models predict only when a user-specified confidence threshold is met. [ref]

EC Code

```
early = skc.pipeBuild_NonMyopicEarlyClassifier(n_clusters=[n_classes])

names=['Non-Myopic Early']

pipes=[early]

skc.gridsearch_classifier(names=names, pipes=pipes, X_train=X_train,
X_test=X_test, y_train=y_train, y_test=y_test, plot_number=3)
```

EC Results

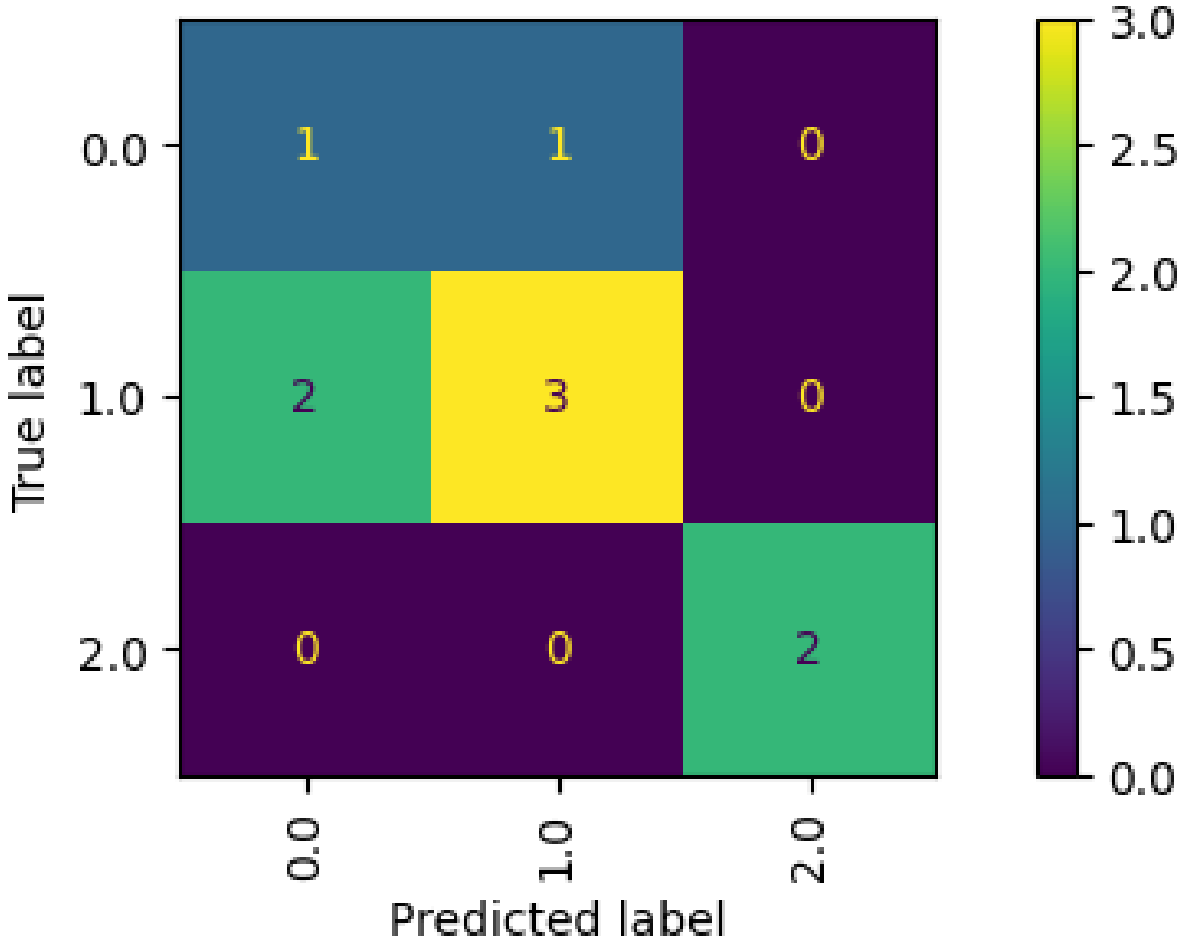

Figure: Early Classification Confusion Matrix

Table of Contents

Thank you!

E-mail: stephen.coshatt@uga.edu