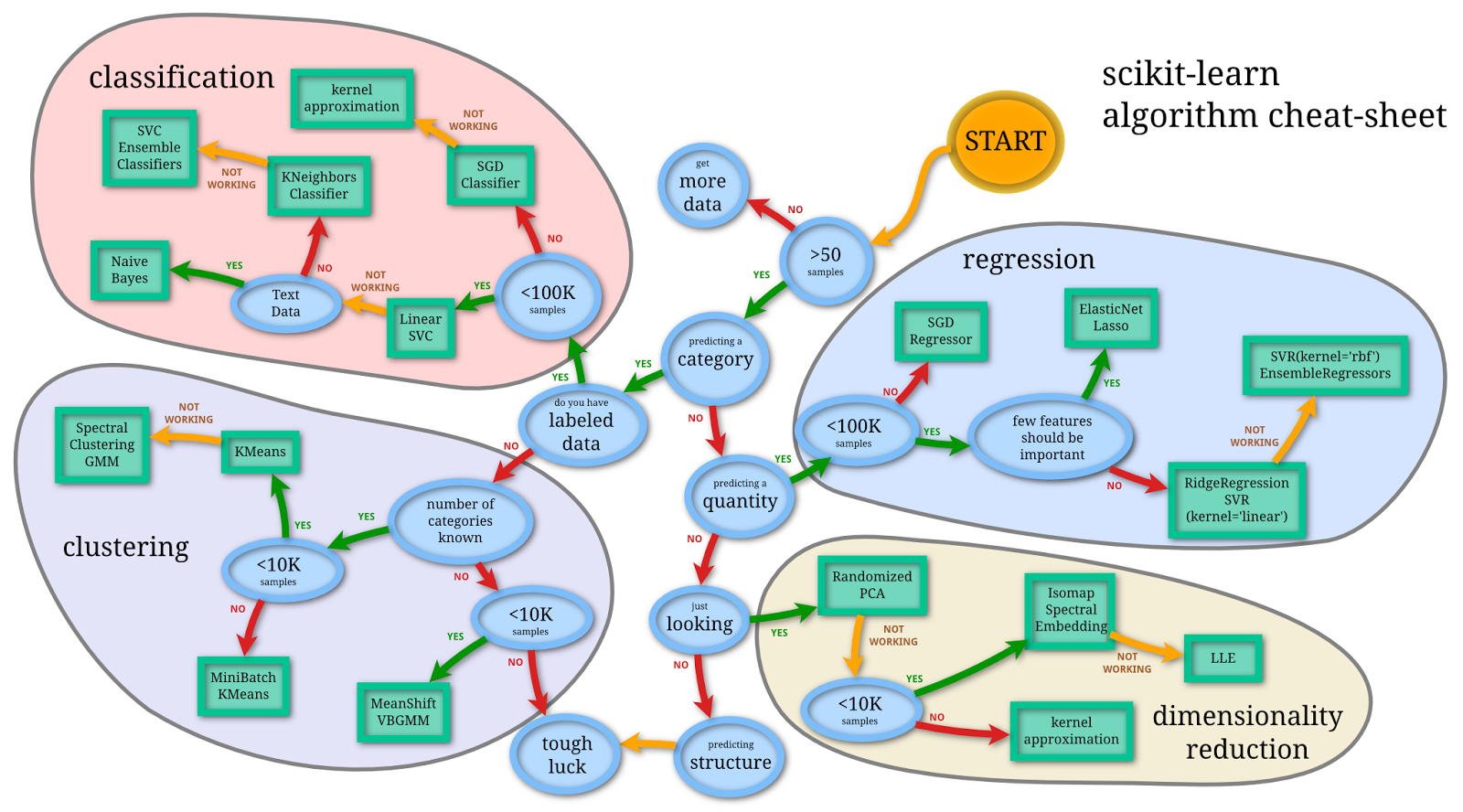
Build and evaluate a kNN classifier

After exploration and preparation of the data we will pick and apply a Machine Learning algorithm. There are a great range to choose from ([*http://scikit-learn.org/*](http://scikit-learn.org/)) and they all have their strengths and weaknesses. It is however important to remember that one of the most crucial aspects of Machine Learning is that we have a good understanding of our data (previous module). If we feed inappropriate data to any algorithm, nothing good will come from it, i.e. "garbage in, garbage out".

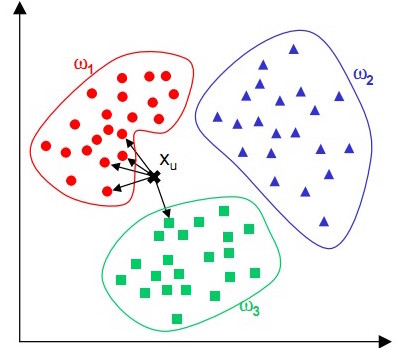
Throughout the bootcamp we will be learning different Machine Learning algorithms. Now we are focusing on **classification** problems. In scikit-learn, there are built-in functions for several Machine Learning algorithms for classification, including:

* Logistic regression: This results in a model that is the classification equivalent of a linear regression model. Instead of a continuous value, the output is a class.
* Support vector classification (SVC): This results in a model, the support vector machine (SVM), that consists of a new set of features (usually a smaller set) that ‘best’ explains differences between classes. This form of classification is also related to principal components analysis.
* Naive Bayes: This results in a probabilistic model that assumes the features are conditionally independent from each other but conditionally related to the output.
* k-Neighbours: This results in a model that directly reflects the data points in the data used to train the model.

**Figure 3-1.**The famous sklearn cheat sheet by Andreas Mueller.

Introduction to k-nearest neighbours (kNN) algorithm

The principle behind kNN is very simple. Given a dataset where you know the class labels (yy values), when a new point is introduced, you want to find a particular number of points in the data closest in distance to the new point. These are called the “nearest neighbours”. You then use the labels associated with these nearest neighbour points (which may or may not be different from each other) to predict the label of the new point.

**Figure 3-2.**kNN algorithm with three classes.

Given nn observations of the input features (you can think of them as the rows of the X matrix), the kNN algorithm identifies the nearest neighbours of a point, XuXu, regardless of the labels. In the simple example above, we have three classes ω1,ω2,ω3ω1,ω2,ω3 and want to assign the right class to XuXu. If k=5k=5 we take the 5 nearest neighbours of XuXu. As you can see, of the 5 nearest neighbours, 4 belong to the class ω1ω1 and 1 to the class ω3ω3, so in this case, we assign the class ω1ω1 to element XuXu.

There are two types of kNN classifiers, derived from two types of learning algorithm:

* **k-nearest neighbour learning**: The number of nearest neighbour points k is specified directly by the user. So when a new point is introduced, it looks for the k points closest to it. In scikit-learn, KNeighborsClassifier allows you to learn models based on this approach.
* **Radius-based neighbour learning**: The number of nearest neighbour points k is dependent on how many points fall into a defined radius (r), which is specified by the user. So when a new point is introduced, it looks for all the points lying within the distance r. In scikit-learn, RadiusNeighborsClassifier allows you to learn models based on this approach.

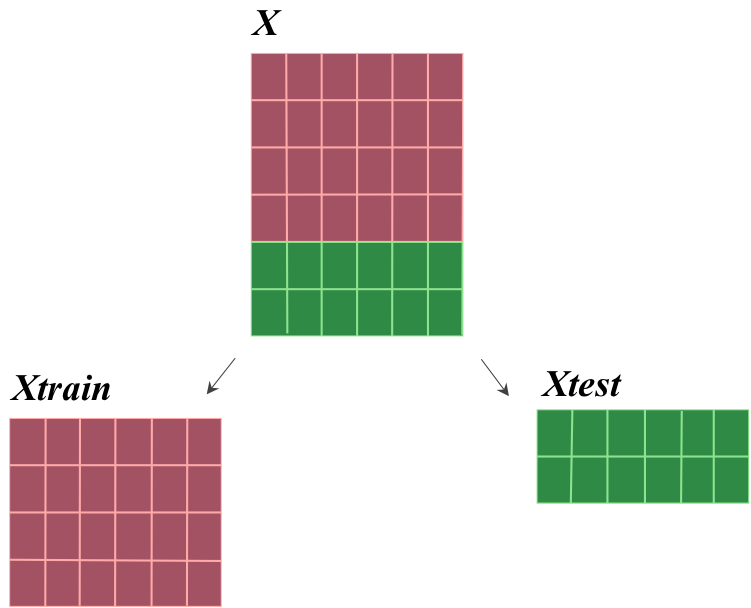
Some remarks about kNN

* For a 2-class (binary) problem, we need to select an odd value of K so that we always have a majority class and don’t result in a tie. More generally, in order to avoid ties, K must not be a multiple of the number of classes.
* The main drawback of kNN is the high computational cost. This is mainly due to the complexity of searching for the neighbours nearest to each sample. In a large dataset, complexity is a problem because with a large number of elements, it will be extremely expensive to check the distance between each element and the element that we want to classify.
* Neighbours-based classification is a type of *non-generalizing* learning: it does not construct a general model, but simply stores instances of the training data and computes the classification from a simple majority vote. This makes it prone to “over-fitting”, i.e. being easily affected by noisy training data.

**Split the data into training and test sets**

Training and testing a classification model on the same dataset is **a methodological mistake**: a model that would just repeat the labels of the samples that it has just seen would overestimate the score and would fail to predict anything useful on yet-unseen data, leading to poor generalisation performance.

To use different datasets for training and testing, we need to split the wine quality dataset into two mutually exclusive datasets, the train and test set; this validation approach is referred to as the Holdout method and is depicted as follows:

**Figure 3-3.**Holdout approach (random split into two disjoint datasets, the train and test set).

In the next module (module 4) we will look into a more general form of the Holdout approach, namely cross-validation (more specifically K-fold cross-validation). However let us first see how to implement the Holdout approach. In Python, you need to use the train\_test\_split function available through [sklearn.cross\_validation](http://scikit-learn.org/stable/modules/generated/sklearn.cross_validation.train_test_split.html). The function takes two mandatory arguments, which are the matrix *X* holding the input data and the vector yy holding the targets (class labels).

Finally, the random\_state argument specifies a value for the seed of the random generator. By setting this seed to a particular value, each time the code is run, the split between train and test datasets will be exactly the same. If this value is not specified, a different split will be output each time since the random generator driving the split will be seeded by a pseudo-random number.

*# Split into training and test sets*

XTrain, XTest, yTrain, yTest = train\_test\_split(X, y, random\_state=1)

The function train\_test\_split splits arrays into subsets of training and test data by randomly assigning each of the observations into one of the arrays. This gives you four arrays. XTrain and yTrain are the two arrays you use to train your model. XTest and yTest are the two arrays that you use to evaluate your model. By default, scikit-learn splits the data so that 25% of it is used for testing, but you can also specify the proportion of data you want to use for training and testing.

As previously, you can check the sizes of the different training and test sets by using the shape attribute:

*# Print the dimensionality of the individual splits*

**print**("XTrain dimensions: ", XTrain.shape)

**print**("yTrain dimensions: ", yTrain.shape)

**print**("XTest dimensions: ", XTest.shape)

**print**("yTest dimensions: ", yTest.shape)

This results in the following output:

> XTrain dimensions: (1115, 10)

> yTrain dimensions: (1115, )

> XTest dimensions: (372, 10)

> yTest dimensions: (372, )

You can also investigate how the class labels are distributed within the *yTest* vector by using the itemfreq function from module 2:

*# Calculate the frequency of classes in yTest*

yFreq = scipy.stats.itemfreq(yTest)

**print**(yFreq)

> [[ 0 129]

[ 1 243]] [http://online.cambridgecoding.com/assets/content/callouts/1.png](http://online.cambridgecoding.com/content/ml_intro/module3#callout_build_and_evaluate_a_knn_classifier_CO1-1)

**[http://online.cambridgecoding.com/assets/content/callouts/1.png](http://online.cambridgecoding.com/content/ml_intro/module3#co_build_and_evaluate_a_knn_classifier_CO1-1)**

In this case, we can see that yTest includes 129 random samples of class 0 (high quality) and 243 random samples of class 1 (low quality).

**Build a kNN classification model with scikit-learn**

To build kNN models using scikit-learn, you will be using KNeighborsClassifier, which allows you to set the value of k using the n\_neighbors parameter. The optimal choice of the value of k is highly data-dependent: in general a larger k suppresses the effects of noise, but makes the classification boundaries less distinct (we will investigate this effect when we visualise the classification boundaries later on in this module). Start by trying two extreme values, k=3 and k=99.

*# Build a kNN classifier with 3 nearest neighbors*

knn3 = KNeighborsClassifier(n\_neighbors=3) [http://online.cambridgecoding.com/assets/content/callouts/1.png](http://online.cambridgecoding.com/content/ml_intro/module3#callout_build_and_evaluate_a_knn_classifier_CO2-1)

knn3.fit(XTrain, yTrain) [http://online.cambridgecoding.com/assets/content/callouts/2.png](http://online.cambridgecoding.com/content/ml_intro/module3#callout_build_and_evaluate_a_knn_classifier_CO2-2)

yPredK3 = knn3.predict(XTest) [http://online.cambridgecoding.com/assets/content/callouts/3.png](http://online.cambridgecoding.com/content/ml_intro/module3#callout_build_and_evaluate_a_knn_classifier_CO2-3)

**print**("Overall Accuracy:", round(metrics.accuracy\_score(yTest, yPredK3), 2)) [http://online.cambridgecoding.com/assets/content/callouts/4.png](http://online.cambridgecoding.com/content/ml_intro/module3#callout_build_and_evaluate_a_knn_classifier_CO2-4)

> Overall Accuracy: 0.87

**[http://online.cambridgecoding.com/assets/content/callouts/1.png](http://online.cambridgecoding.com/content/ml_intro/module3#co_build_and_evaluate_a_knn_classifier_CO2-1)**

Build the classification model

**[http://online.cambridgecoding.com/assets/content/callouts/2.png](http://online.cambridgecoding.com/content/ml_intro/module3#co_build_and_evaluate_a_knn_classifier_CO2-2)**

Train (fit) the model

**[http://online.cambridgecoding.com/assets/content/callouts/3.png](http://online.cambridgecoding.com/content/ml_intro/module3#co_build_and_evaluate_a_knn_classifier_CO2-3)**

Test (predict) with the independent test set

**[http://online.cambridgecoding.com/assets/content/callouts/4.png](http://online.cambridgecoding.com/content/ml_intro/module3#co_build_and_evaluate_a_knn_classifier_CO2-4)**

Report the overall accuracy by comparing the actual with the predicted values on yTest

In a similar way:

*# Build a kNN classifier with 99 nearest neighbors*

knn99 = KNeighborsClassifier(n\_neighbors=99)

knn99.fit(XTrain, yTrain)

yPredK99 = knn99.predict(XTest)

**print**("Overall Accuracy:", round(metrics.accuracy\_score(yTest, yPredK99), 2))

> Overall Accuracy: 0.85

In this case, we can see that by increasing the number of k, we do not necessarily gain a better overall test accuracy.

**Calculate validation metrics for your classifier**

In a classification task, once you have created your predictive model, you will always need to evaluate it. Evaluation functions help you to do this by reporting the performance of the model through four main performance metrics: precision, sensitivity (or recall, true positive rate) and specificity (or true negative rate) for the different classes, as well as overall accuracy. As opposed to overall classification accuracy, these metrics are class-specific: they may differ for the two classes, whereas the overall accuracy may remain the same. To understand these metrics, it is useful to create a *confusion matrix*, which records all the true positive, true negative, false positive and false negative values.

You can compute the confusion matrix for the first classification model you built in the previous Learning Activity (where k=3) using the confusion\_matrix function from the metricsmodule.

*# Get the confusion matrix for your classifier*

mat = metrics.confusion\_matrix(yTest, yPredK3) [http://online.cambridgecoding.com/assets/content/callouts/1.png](http://online.cambridgecoding.com/content/ml_intro/module3#callout_build_and_evaluate_a_knn_classifier_CO3-1)

**print**(mat)

> [[99 30]

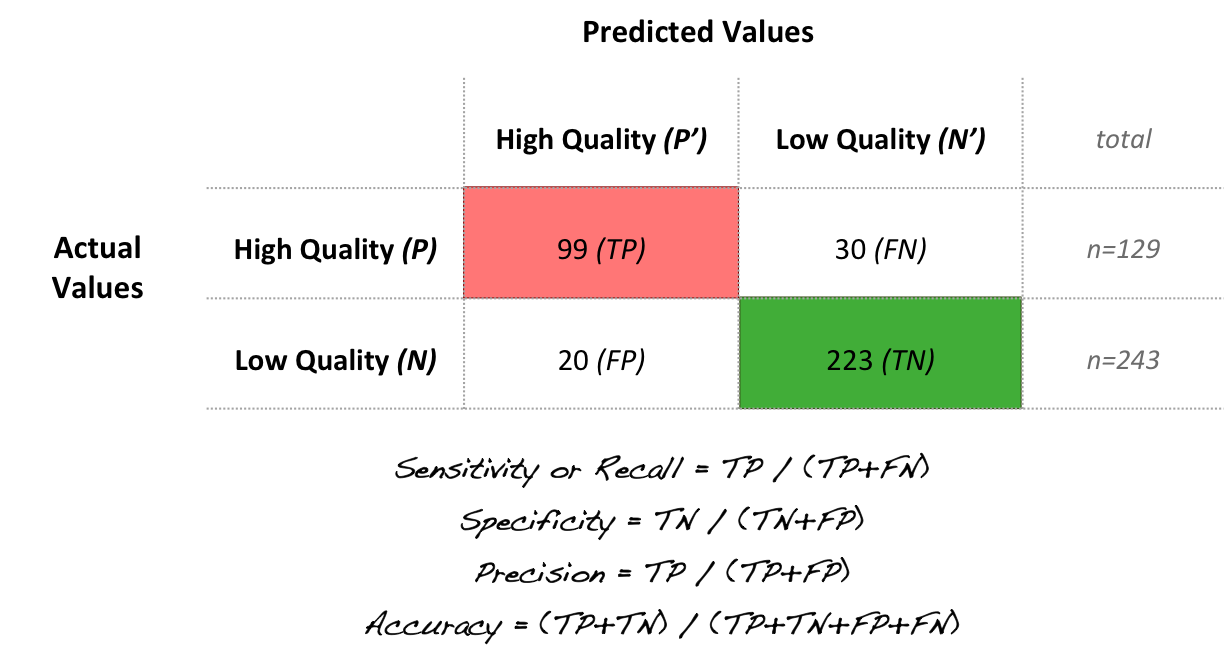
[20 223]] [http://online.cambridgecoding.com/assets/content/callouts/2.png](http://online.cambridgecoding.com/content/ml_intro/module3#callout_build_and_evaluate_a_knn_classifier_CO3-2)

**[http://online.cambridgecoding.com/assets/content/callouts/1.png](http://online.cambridgecoding.com/content/ml_intro/module3#co_build_and_evaluate_a_knn_classifier_CO3-1)**

Compute the confusion matrix for our predictions. Remember that the test data contain observations that are **not** in the training data.

**[http://online.cambridgecoding.com/assets/content/callouts/2.png](http://online.cambridgecoding.com/content/ml_intro/module3#co_build_and_evaluate_a_knn_classifier_CO3-2)**

The first value in the first row (99) is the number of True Positives (TP); the second value in the first row (30) is the number of False Negatives (FN); the first value in the second row (20) is the number of False Positives (FP), and the second value in the second row (223) is the number of True Negatives (TN). We could represent this schematically as follows:

**Figure 3-4.**Example of a confusion matrix for a given classification model using the wine quality dataset.

Validation Metrics

* **Accuracy:** Accuracy is the overall "correctness" of the model and is calculated as the number of correctly classified observations divided by the total number of observations. Accuracy is defined by

Accuracy=(tp+tn)/totalAccuracy=(tp+tn)/total

where tptp and tntn are the numbers of true positive and true negative predictions and total is the total number of classifications.

* **Precision (for a class):** Precision is a measure of the accuracy for a specific class, it reports the proportion of correct classifications for a specific class. It is defined by:

Precision=tp/(tp+fp)Precision=tp/(tp+fp)

where tptp and fpfp are the numbers of true positive and false positive predictions for the considered class, e.g. ability to correctly classify wine as being of high quality or poor quality. tp+fptp+fp is the total number of elements labelled as belonging to the considered class by the classifier.

* **Recall, aka. Sensitivity, True positive rate (for a class):** Recall reports the ability of a model to select instances of a certain class from a dataset, e.g. a classifier that has high sensitivity with regards to the poor quality class will do well at correctly classifying wine as being of poor quality (although this may make it more likely to incorrectly include more high quality wine in this class). It is defined by:

Recall=Sensitivity=tp/(tp+fn)Recall=Sensitivity=tp/(tp+fn)

where tptp and fnfn are the numbers of true positive and false negative predictions for the considered class. tp+fntp+fn is the total number of elements that actually belong to the considered class.

* **Specificity, True negative rate (for a class):** Specificity reports the ability of the model to correctly exclude class non-members in a dataset from the class, e.g. a classifier that has high specificity wrt the poor quality wine class will do well at correctly excluding high quality wine from the class (although this may make it more likely to miss poor quality wine). It is defined by:

Specificity=tn/(fp+tn)Specificity=tn/(fp+tn)

where tntn and fpfp are the numbers of true negative and false positive predictions for the considered class. fp+tnfp+tn is the total number of elements that should not be included in the class.

* **F1-score (for a class):** This measures the accuracy of the model with respect to a particular class, and is the harmonic mean of precision and recall. It is defined by:

F1=2∗(precision∗recall)/(precision+recall)F1=2∗(precision∗recall)/(precision+recall)

Because the calculation of performance metrics is such an important step of model evaluation, scikit-learn offers a wrapper around these functions, metrics.classification\_report, to facilitate their computation. This includes all the class-level metrics outlined above except for specificity, and supplements the overall accuracy computed by the function metrics.accuracy\_score you already used in the previous Learning Activity.

*# Report the metrics using metrics.classification\_report*

**print**(metrics.classification\_report(yTest, yPredK3))

**print**("accuracy: ", round(metrics.accuracy\_score(yTest, yPredK3), 2))

> precision recall f1-score support

0 0.83 0.77 0.80 129

1 0.88 0.92 0.90 243

avg / total 0.86 0.87 0.86 372

accuracy: 0.87

**Plot the decision boundaries for different models**

The border between two neighboring regions of different classes is known as the *decision boundary*. You can visualise the effects of the kNN parameters such as n\_neighbors by looking at the decision boundaries.

We have provided you with a pre-defined function in the visplots library called knnDecisionPlot that allows you to do this. This takes five arguments: XTrain: X from the training set; yTrain: y from the training set; header: the feature names; n\_neighbors: number of neighbours; and weights: how weights should be assigned, e.g. "uniform" (default case) or "distance".

*# Check the arguments of the function*

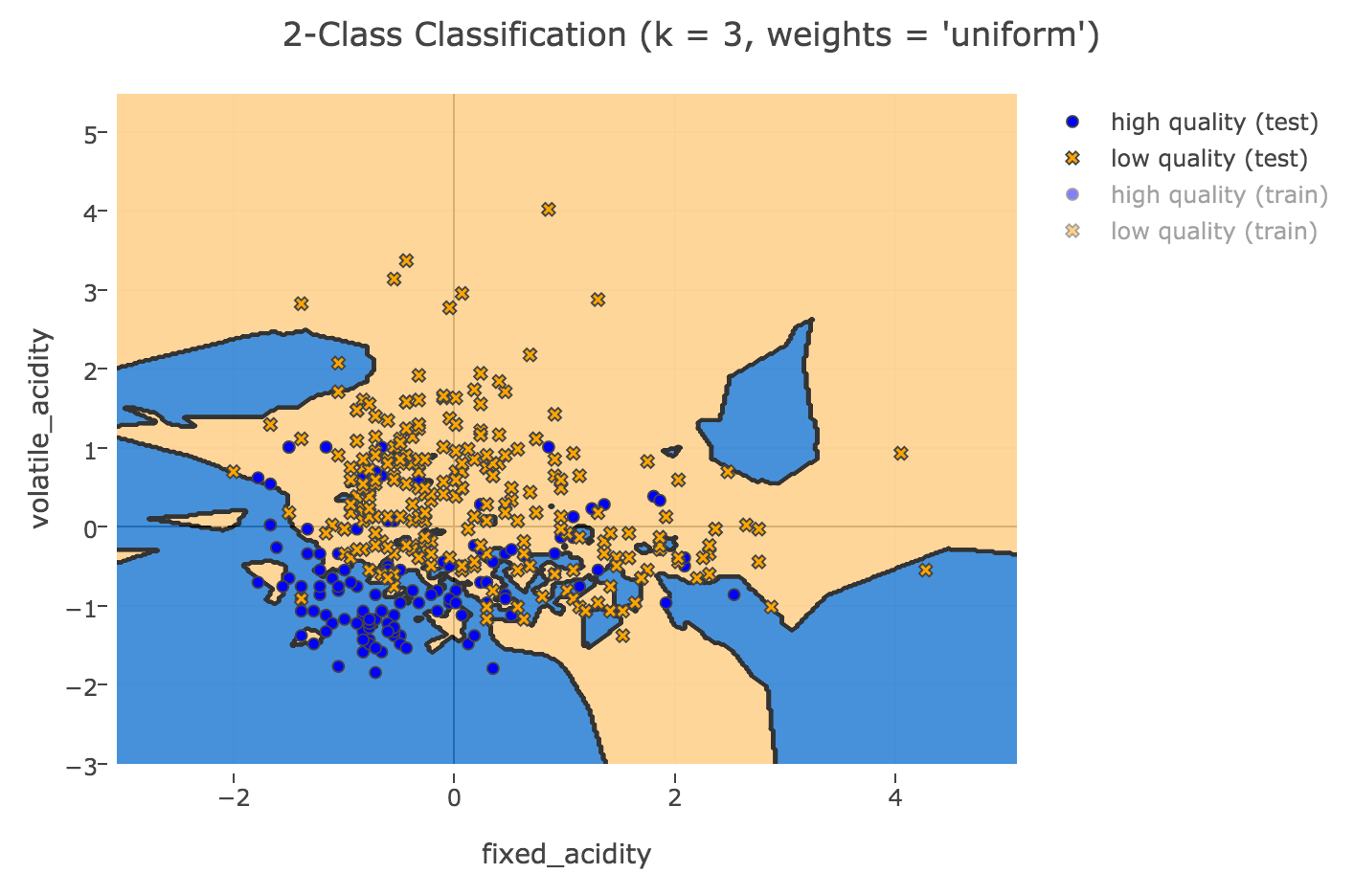
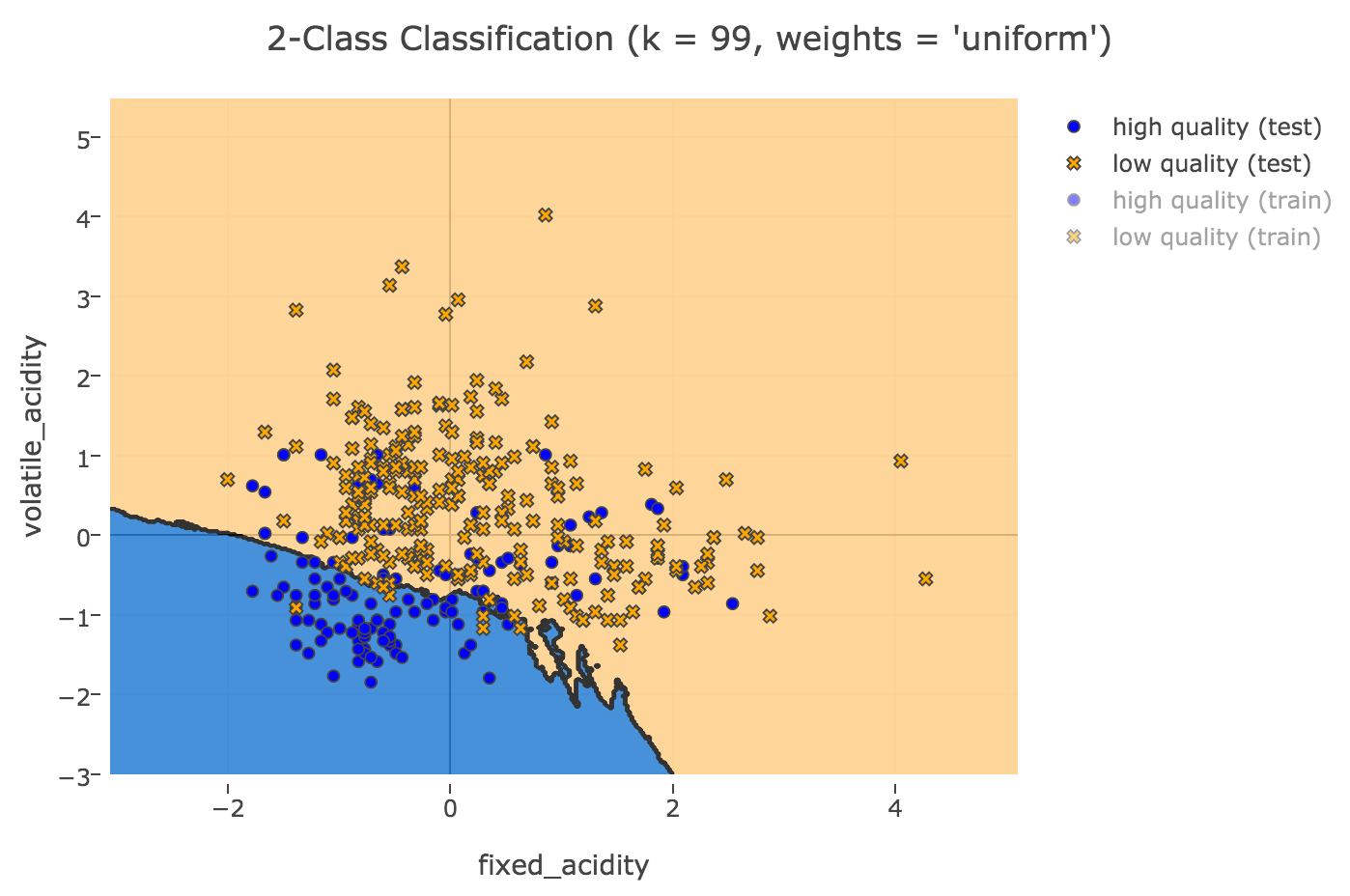
help(visplots.knnDecisionPlot)

*# Visualise the boundaries*

visplots.knnDecisionPlot(XTrain, yTrain, XTest, yTest, header, n\_neighbors = 3)

visplots.knnDecisionPlot(XTrain, yTrain, XTest, yTest, header, n\_neighbors = 99)

The plots below show the first two features of X coloured by the two class labels (red, blue). The background colours (also red and blue) correspond to the predicted class of all points in the mesh plot. To make the plot more visually simple, only the *test* samples are depicted in the plot; remember though that the decision boundary has been built using the *training* data!

**Figure 3-5.**kNN classifier with k = 3.**Figure 3-6.**kNN classifier with k = 99.

Quiz: What do you observe from these plots?

For smaller values of K the decision boundaries present many "creases". In this case the models may suffer from instances of **overfitting**. For larger values of K, we can see that the decision boundaries are less distinct, smoother and tend towards linearity. In these cases the boundaries may be too simple and unable to classify effectively, leading to cases of **underfitting**. Remember, the optimal choice of the value k is highly data-dependent, and in the next module we will learn how to optimise this parameter.

**Try different weight configurations**

So far, the nearest neighbors classification algorithm that you implemented uses *uniform* weights: that is, the value assigned to a query point is computed from a simple majority vote of the nearest neighbors.

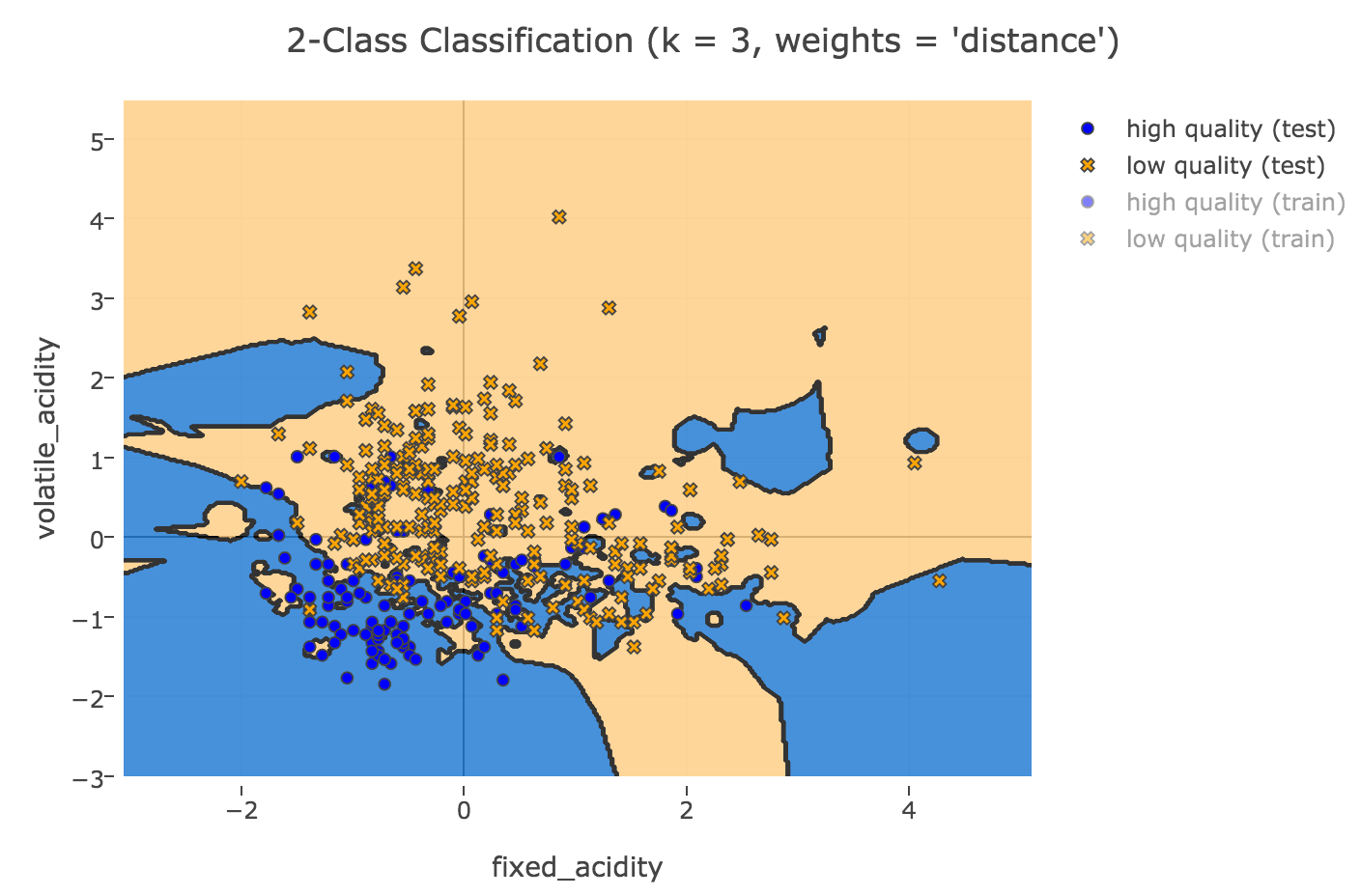
Under some circumstances, it is better to give more importance ("weight" in computing terms) to nearer neighbors. This can be accomplished through the weights parameter. When weights = "distance", weights are assigned to the training data points in a way that is proportional to the inverse of the distance from the query point. In other words, neighbors that are closer in distance contribute more to the fit.

What if we use weights based on distance? Does it improve the overall performance?

*# Build the classifier with two pre-defined parameters (n\_neighbors and weights)*

*# Visualise the boundaries of a kNN model with weights equal to "distance"*

For example, try to generate the following plot:

**Figure 3-7.**kNN classifier with k = 3 and weights = distance.

Wrap up of Module 3

* In k-nearest neighbour learning, when a new point is introduced, the algorithm looks for the k points closest to it. It then assigns the same label to this new point as the label of the greatest number of those k points.
* To ensure there is never a tie, k should not be a multiple of the number of classes (labels).
* Neighbours-based classification is a type of non-generalizing learning and can be more prone to over-fitting.
* For a binary classification problem, there are four possible outcomes: true positive (tp), true negative (tn), false positive (fp), false negative (fn).
* Accuracy=(tp+tn)/totalAccuracy=(tp+tn)/total
* For a given class:
  + Precision=tp/(tp+fp)Precision=tp/(tp+fp)
  + Recall=Sensitivity=tp/(tp+fn)Recall=Sensitivity=tp/(tp+fn)
  + Specificity=tn/(fp+tn)Specificity=tn/(fp+tn)
  + F1=2∗(precision∗recall)/(precision+recall)F1=2∗(precision∗recall)/(precision+recall)
* If k is too large, there is a danger of under-fitting, while if k is too small, there is a danger of over-fitting.

*### Learning Activity 1: Split the data into training and test sets*

*# Split into training and test sets*

XTrain, XTest, yTrain, yTest = train\_test\_split(X, y, random\_state=1)

*# Print the dimensionality of the individual splits*

**print** ("XTrain dimensions: ", XTrain.shape)

**print** ("yTrain dimensions: ", yTrain.shape)

**print** ("XTest dimensions: ", XTest.shape)

**print** ("yTest dimensions: ", yTest.shape)

*# Calculate the frequency of classes in yTest*

yFreq = scipy.stats.itemfreq(yTest)

**print**(yFreq)

*### Learning Activity 2: Apply KNN classification algorithm with scikit-learn*

*# Build a KNN classifier with 3 nearest neighbors*

knn3 = KNeighborsClassifier(n\_neighbors=3)

knn3.fit(XTrain, yTrain)

yPredK3 = knn3.predict(XTest)

**print** ("Overall Accuracy:", round(metrics.accuracy\_score(yTest, yPredK3), 2))

*# Build a KNN classifier with 99 nearest neighbors*

knn99 = KNeighborsClassifier(n\_neighbors=99)

knn99.fit(XTrain, yTrain)

yPredK99 = knn99.predict(XTest)

**print** ("Overall Accuracy:", round(metrics.accuracy\_score(yTest, yPredK99), 2))

*### Learning Activity 3: Calculate validation metrics for your classifier*

*# Get the confusion matrix for your classifier using metrics.confusion\_matrix*

mat = metrics.confusion\_matrix(yTest, yPredK3)

**print** (mat)

*# Report the metrics using metrics.classification\_report*

**print** (metrics.classification\_report(yTest, yPredK3))

**print** ("accuracy: ", round(metrics.accuracy\_score(yTest, yPredK3), 2))

*### Learning Activity 4: Plot the decision boundaries for different models*

*# Check the arguments of the function*

help(visplots.knnDecisionPlot)

*# Visualise the boundaries*

visplots.knnDecisionPlot(XTrain, yTrain, XTest, yTest, header, n\_neighbors= 3)

visplots.knnDecisionPlot(XTrain, yTrain, XTest, yTest, header, n\_neighbors= 99)

*### Test Activity 5: Try different weight configurations*

*# Build the classifier with two parameters*

knnW3 = KNeighborsClassifier(n\_neighbors=3, weights='distance')

knnW3.fit(XTrain, yTrain)

predictedW3 = knnW3.predict(XTest)

**print** (metrics.classification\_report(yTest, predictedW3))

**print** ("Overall Accuracy:", round(metrics.accuracy\_score(yTest, predictedW3), 2))

visplots.knnDecisionPlot(XTrain, yTrain, XTest, yTest, header, n\_neighbors= 3,

weights="distance")