CS4650 Topic 14a
Supervised Learning

Machine Learning

- *Machine Learning* programs automatically adjust their performance as information is discovered in the data. The programs *adapt* to the data.
- The code has a parameterized model. It is these parameter values that are changed to perform the adaption.
- With these broad definitions, a large number of programs can be considered machine learning systems.

Machine Learning and Artificial Intelligence

- Machine Learning is considered a subfield of Artificial Intelligence (AI).
- Machine Learning is divided into three major classes:
  - *Supervised Learning*: Algorithms learn from a training set of labeled examples (*exemplars*), then attempt to generalize when new samples are presented. Examples of supervised learning include logistic regression, support vector machines, decision trees, and random forest.
  - *Unsupervised Learning:* Algorithms learn from an unlabeled training set, exploring data according to some statistical or geometrical criterion. Examples of unsupervised learning are k-means clustering and kernel density estimation.
  - *Reinforcement Learning*: Algorithms learn via reinforcement from criticism that indicates the quality of the solution, but not how to improve it. Improved solutions are achieved by iteratively exploring the solution space.

Supervised Learning

- There are two main branches of Supervised Learning, based on the type of answer we are seeking.
- If the answer is Yes/No, this is a classification problem. Given a new sample, does this sample belong in group 1 or group 2? Examples:
  - Given the results of that test, does the patient have this disease?
  - Giving this activity, is this a fraudulent transaction?
- If the answer is a numerical result, this is a regression problem. Given a new sample, what is the expected value for a resultant quantity? Examples:
  - Given a description of a car, what is the expected market value?
  - Given the recent weather data, what is the expected yield of crops?

Classification Problems

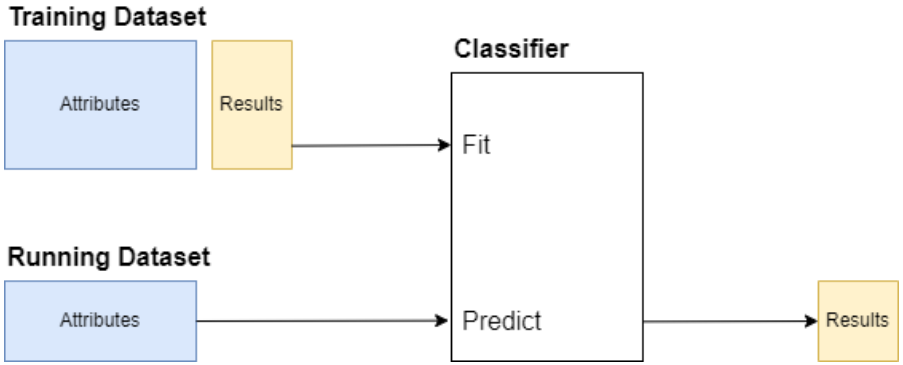
- We will look at a number of algorithms used to perform classification.
- In a future lecture, we will discuss algorithms used for regression analysis.
- Some of the algorithms we discuss can be used for both type of analysis.
- There are *many* classification algorithms; we will discuss some of the more common or popular.

- This lecture will focus on how the classifiers work.
- The next lecture will focus more on using the classifiers.

Samples

- With *Supervised Learning*, the classifier is *trained* using labeled samples.
- We say that a sample is labeled if it contains all of the input parameter values, but it also contains the correct output value.
- The labeled samples provide examples of correct outputs, from which the classifier learns (or at least, that is the goal!)
- Once the classifier is trained, then unlabeled samples are passed to the classifier, which then computes an expected output value.

Classifier



- This diagram shows the general form of a classifier in *sklearn* (scikit-learn).
- The classifier is passed a training set (through it's *fit* method). The algorithm inside the classifier is then 'tuned' to 'fit' the training data.
- After it has been 'fit', the classifier can be passed one or more unlabeled samples (through it's *predict* method). The classifier's algorithm computes the *estimated result* for each sample.
- Ideally, the result matches what would be found in the population.

Classifier Performance

- When we talk about the *performance* of a classifier, we are not talking about the speed at which the program completes the task, but rather the quality of the results.
- The basic measure of the performance is its *accuracy*: the number of correctly predicted examples divided by the total number of examples.

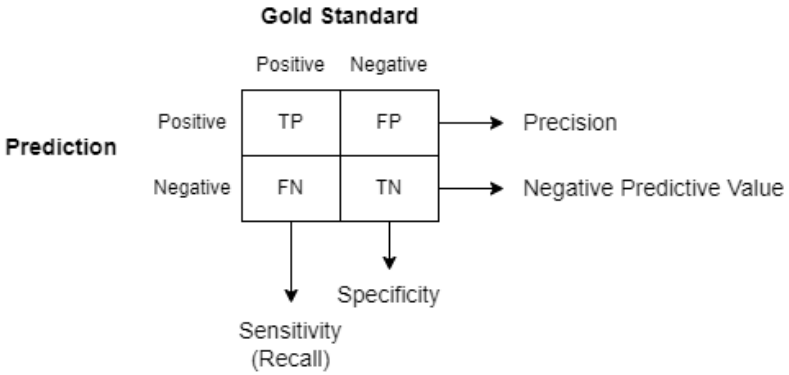
$$acc = \frac{Number\ of\ correct\ predictions}{n}$$

- Accuracy is related to the error rate by *acc = 1 - err*.

Classifier Performance

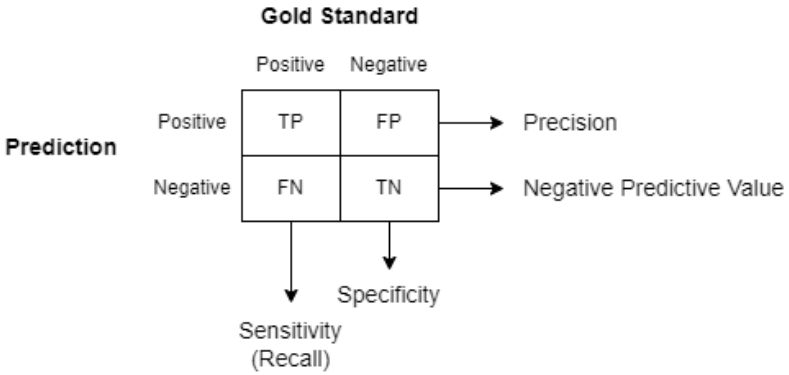
- Accuracy is the most normal metric for evaluating a classifier.
- Sometimes it is more important to correctly predict elements in one class rather than another. For example, incorrectly predicting the answer as 'Yes' might be more costly than incorrectly predicting the answer as 'No'.
- For these cases, a *confusion matrix* is a useful tool.
- First, two other definitions:
  - *Gold Standard*: This is the correct classification of the exemplar.
  - *Prediction*: This is the classification determined by the algorithm.
- Ideally we want the Prediction to match the Gold Standard.

Confusion Matrix



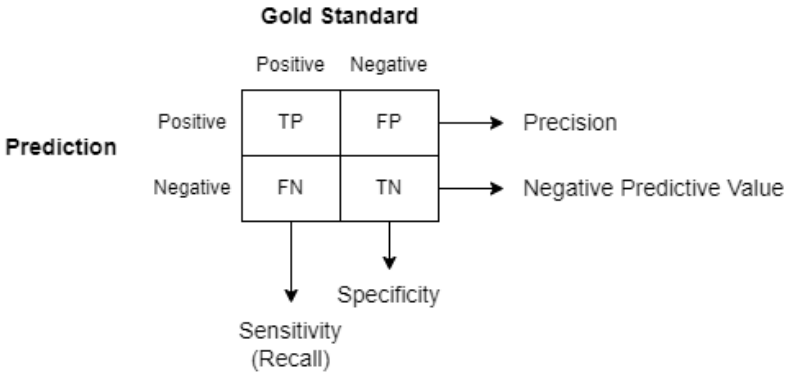
- For a binary problem, there are four possible cases:
  - *True Positives (TP)*: The Gold Standard and the Prediction both say 'Yes'
  - *False Positives (FP)*: The Prediction is 'Yes' but the Gold Standard is 'No'
  - *True Negatives (TN)*: The Gold Standard and the Prediction both say 'No'
  - *False Negatives (FN)*: The Prediction is 'No' but the Gold Standard is 'Yes'

Confusion Matrix



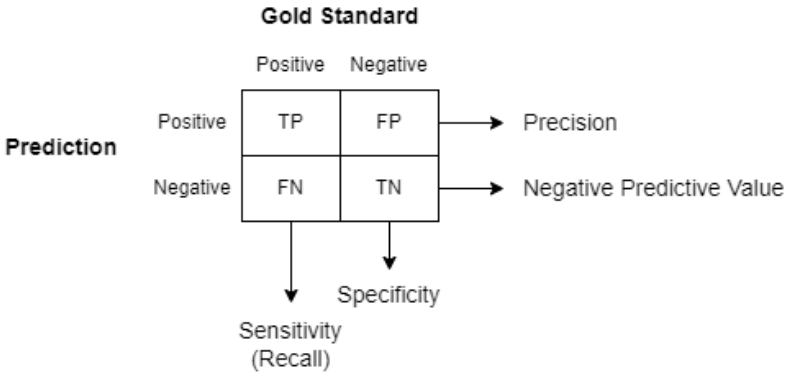
- The *accuracy* is (TP + TN) / (TP + TN + FP + FN), the number of correct predictions divided by the total number of samples.

Confusion Matrix



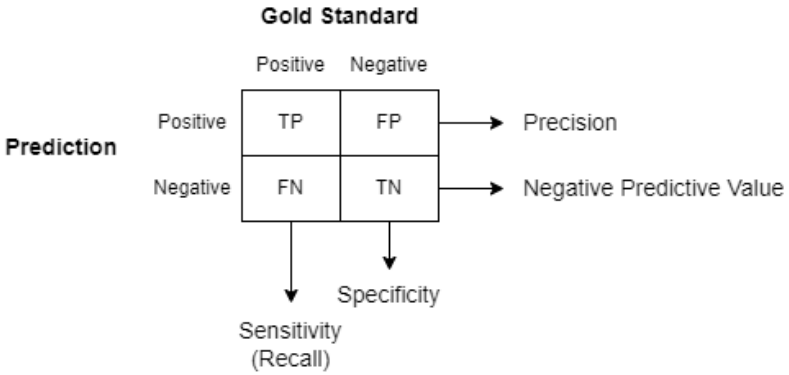
- The *sensitivity* or *recall* is TP / (TP + FN), the fraction of correctly positive predictions divided by the total number of actually positive exemplars in the Gold Standard.

Confusion Matrix



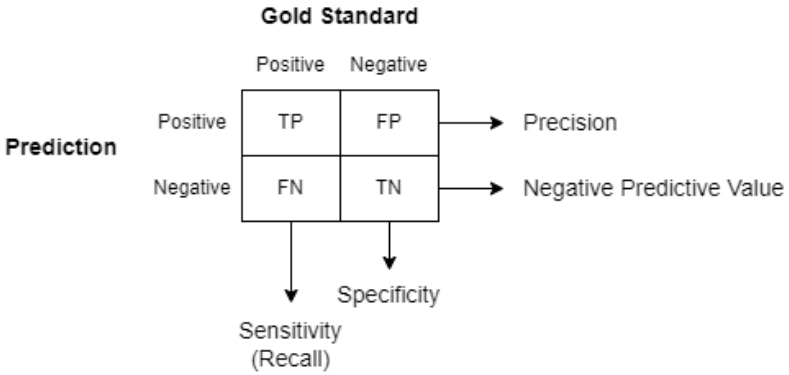
- The *specificity* is TN / (TN + FP), the fraction of correctly negative predictions divided by the total number of actually negative exemplars in the Gold Standard.

Confusion Matrix



- The *precision* or *positive predictive value* is TP / (TP + FP), the fraction of correctly positive predictions divided by the total number of positive predictions.

Confusion Matrix

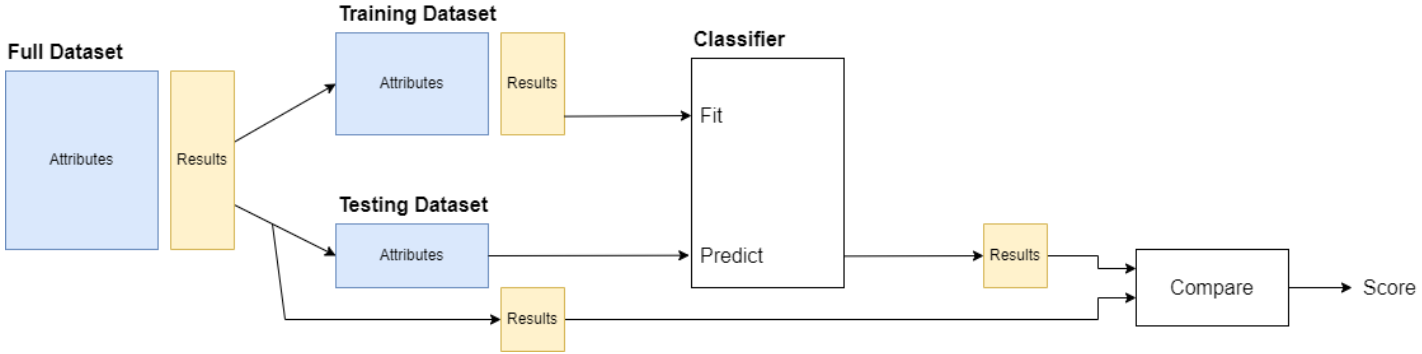


- The *negative predictive value (NPV)* is TN / (TN + FN), the fraction of correctly negative predictions divided by the total number of negative predictions.

Evaluating the Classifier

- How do we know how good of a job the classifier does in computing results?
- Presumably the classifier would do a very good job of computing the result for a sample from the training set, it already knows the correct value.
- If the sample is from the running set, which the classifier has not yet encountered, we don't 'know' the correct value.
- Consequently, we don't know the quality of the classifier's results…
- …but maybe there is a way we can estimate this quality.

Evaluating Classifier



- The usual approach is to split the Dataset into two parts, the training set and the testing set. The training set is usually larger than the testing set.
- The output values (results) of the testing set are removed, held to the side.
- The classifier is then trained with the training set, and then evaluated with the testing set.
- The output of the classifier is then compared with the values removed from the testing set.

Minor Detail

- In the previous slides, we indicated that the results were incorporated in the samples with the attributes, and that we 'extracted' these results to build the test dataset.
- In typical practice, the attributes and results are in separate arrays. If we have *n* samples and *d* attributes, the attributes are in an $n \times d$ array and the results are in a $n \times 1$ array (vector).
- The *fit* function is therefore passed an array and a vector, while the *predict* function is passed just the array.

In-sample Errors, Out-of-sample Errors

- The assumption is that, since the classifier learns from the training set, where it knows the correct answers, then if we do a prediction of one of these values, we should get the correct result. However, as we will see in the next lecture, this is not always true, sometimes an incorrect prediction is given. This is called an *in-sample error,* E_{in}, a mistake evaluating the training set.
- When new samples are encountered, the classifier may or may not predict the correct result. If an incorrect prediction is made on one of these samples, this is called an *out-of-sample error,* E_{out}.
- We will see empirically that $E_{out} \geq E_{in}$.

Cross Validation

- As mentioned earlier, there are a number of classifiers available.
- While one classifier might be the more accurate algorithm to use for one problem, another problem might be better solved with a different classifier.
- We can run multiple classifiers, then examine the E_{out} values that were produced from the test data. We then choose the classifier that has the lowest expected error (E_{out}).
- This selection process is called *cross validation*.

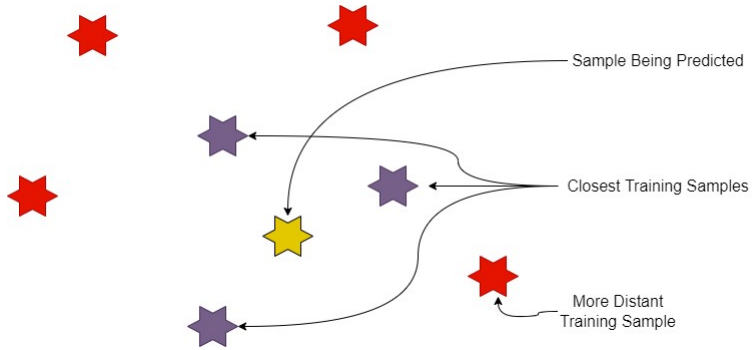
What is Learning?

- We have already seen some examples of learning: as the classifier is being *fit* to the training data.
- Most if not all of the classifiers have some parameters that can be adjusted to tune the classifier.
- The selection of these parameter values can be seen as a second level of learning.
- The goal is the minimization of E_{out}, to maximize the accuracy of the classification.

SKLearn Classifiers

- These are the *sklearn* classifiers that we will be using:
  - knn (k-nearest neighbors)
  - Decision Trees
  - Random Forest
  - SVM
- These classifiers have different approaches, and produce different results.
- Some problems achieve better results with one classifier while others would use a different classifier.
- We will explore how each of these classifiers work. This knowledge is not required to be able to use these classifiers, but understanding this may help as you tune the classifiers.

KNN



- The KNearestNeighbors classifier is conceptually very simple:
  - The classifier builds an internal table holding the training data, both the attributes and the result.
  - When a prediction is sought, KNN compares the attributes of the sample in question with all of the stored results, keeping the *k* trained values that are the closest to the new sample.
  - Based on these nearest neighbors, KNN determines a form of 'average' to provide the answer.

KNN

- We have not yet talked about what 'closest' means!
- Each KNN has its own method of 'averaging' the values of nearest neighbors to get the value of the prediction.
  - One example might be to pick the majority.
  - A more complex algorithm might use the inverse of the distance as a weight into the majority determination.
- The simplest case is *k = 1*, where the classifier simply finds the closest match. The prediction is the answer from that training sample.

KNN

- Consider the *k* = *3* approach, when searching for a Yes/No answer:
  - Find the 3 closest matches
  - Find the majority answer (for example, if two had the answer Yes and one had No, then the answer would be Yes).
  - A more complex version might notice that the very close answer was No, but the two Yes answers were much further apart. In this case, the classifier might pick No.

KNN Considerations

- KNN is quite sensitive to outliers. Consequently, the data should be cleaned before use in training.
- Another problem has to do with scaling. Consider a dataset with one column giving weight in pounds and another giving time in minutes. The weight data might range from 1 to 20 pounds, but the time varies from 1 to 1440 minutes (the number of minutes in a day). Two samples that are an hour apart would seem a lot more distant than two weights that are 18 pounds apart. The data may need to be scaled, so the different columns have appropriate relative weight.
- Some columns might be irrelevant, and should be deleted.

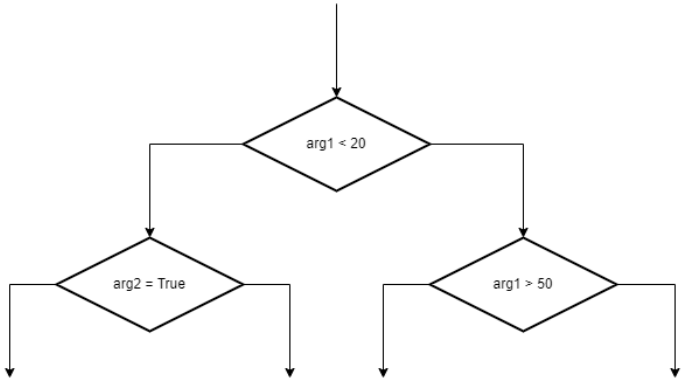
KNN Considerations

- Another thing to consider is columns that are essentially enumerated values. For example, the weather being sunny, cloudy, windy, rainy, or snowing. What would be the distance between these values? Perhaps for the algorithm there is an ordering of these values, and a rough computation for distance. In this case, this column of data could be mapped to appropriately scaled numerical values.
- By default, KNN uses Minkowski distance, but can be set to Manhattan or Euclidean distance. *Actually, both Manhattan and Euclidean distances are special cases of Minkowski distance!*

KNN Considerations

- By far, the most common 'tuning parameter' of KNN is the value for *K*, the number of neighbors.
- One rule of thumb is to set *K* to be the square root of the number of samples in the training set.
- However, once you have the initial value for *K*, try varying the value to find improvements in the results.

Decision Trees



- A Decision Tree consists of a collection of nodes.
- Each node references one of the data attributes and a test, such as equal, greater than, etc.
- The nodes each have two 'output' paths, one to take if the test was True, the other for a False test.
- At the leaves of the tree, nodes contain the values for the output of the classifier.

Decision Tree Example

- A few years ago, I was working on a program for solving MasterMind.
- In the MasterMind game, one person picks four pegs of various colors, setting them in some order, but hides this sequence.
- The other player then makes a guess by placing a sequence of four colored pegs.
- The first player then 'scores' the guess, giving a black peg for each guess that was the right color in the right position and a white peg for each guess that was the right color, but in the wrong position.
- The second player then uses this information to come up with a second guess.
- This process repeats until the player guesses the correct combination.

Decision Tree Example

- There were 4 pegs, and 6 colors, so there were 6^4 possible combinations.
- The program built a decision tree, with a couple of differences:
  - In the node was the sequence of colors for a guess.
  - There were several possible output paths, one for each combination of score pegs (so one path might be 4w and another might be 2b1w).
- The first node in the tree simply had an arbitrary sequence of four colors.
- One of the branches from that first node was 1b1w. To get the next node on that path, the program looked at all possible answers that would give the score of 1b1w for the given guess.
- To pick the guess for that node, the program tried each possible guess, determining how well that guess 'split' the list of possible values.

Decision Tree Example

- There were two algorithms I used for deciding the best split of the values:
  - Which guess had the most possible branches out of the node.
  - Which guess divided the set of possibilities to have the smallest largest set size.
    - One value might result in subsets of size 2, 3, 4, and 7, while another might give subsets 3, 4, 4, and 5. This second choice would be better.

Decision Trees

- The decision trees used in analytics was similar to the trees I built for MasterMind. The major differences were:
  - I had multiple branches out of a node, where as Decision Trees only have two: *True* and *False*.
  - The 'test' inside the Decision Tree node indicates one of the parameters, then gives a boolean expression using that parameter, such as 'the parameter's value is greater than *n*', or 'the parameter's value is equal to *n*'.
  - The depth of the Decision Tree can be limited. This is one of the main tuning parameters for the Decision Tree.

Decision Trees

- Decision Trees are built automatically (and quickly!) by examining the training data.
- At each level of the tree, it considers all of the training values that 'belong' to this branch.
- Looking at these values, it selects a parameter and a value for that parameter that optimally divides the training values into two groups.
- As part of that optimal division, if all of the samples in one of the two sub-groups have the same result (output value), then no further tests are required, the output is known.

Decision Tree Considerations

- Decision Trees can handle non-numeric columns easily.
- They work out-of-the-box, with no complex tuning required.
- They are considered a 'white-box' algorithm: the Tree can be plotted, and a person can easily visualize what the tree is doing. Conversely, a neural network cannot be visualized, a person in general cannot understand what it is doing.
- One problem is that the results have a high degree of variance. Since the trees are randomly built (not completely randomly), two different trees will produce different results.
- A lot of work has gone into algorithms and heuristics to improve Decision Trees.

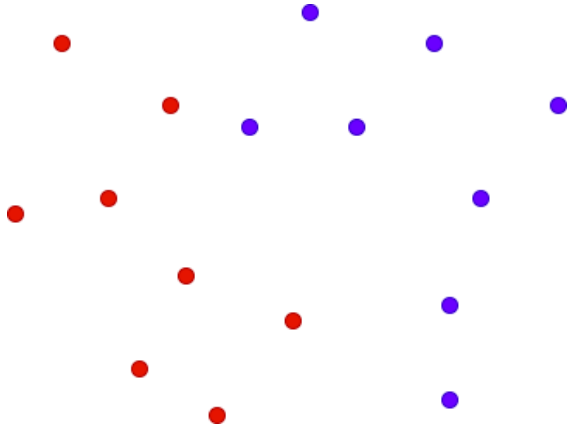
Random Forests

- One of the shortcomings of Decision Trees is that the results can have a large variance.
- One solution is to make multiple trees, then average their results.
- The resulting ensemble of Decision Trees is called a Random Forest.
- Actually, the forest could also contain other types of classifiers, it is not limited to only Decision Trees.
- Random Forests work best when the Decision Trees are uncorrelated.
- Uncorrelated trees can be built by:
  - Choosing different subsets of the training data to build each tree.
  - Select different parameters to form branches near the root of the tree.

Random Forests

- Recall that the depth of the tree can be limited, so trees of varying height can be used.
- One interesting idea for optimizing the Forest is this:
  - Build one Decision Tree from the subsample.
  - Find all of the sample values where this tree computes the wrong result, then build a new tree using those samples.
- Most Random Forest implementations build many Decision Trees.

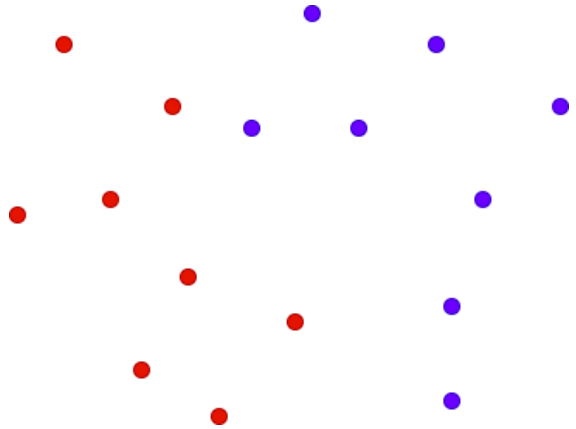
Support Vector Machines



- Support Vector Machines (SVM) are conceptually pretty simple, but are hard to visualize because of the dimensionality of the data.
- Each attribute is essentially its own dimension, so the machines have many dimensions.
- The example uses 2 dimensions.
- The training data has a number of points, divided into the two sets (Yes and No).
- The SVM finds a line that divides the points into the two groups.

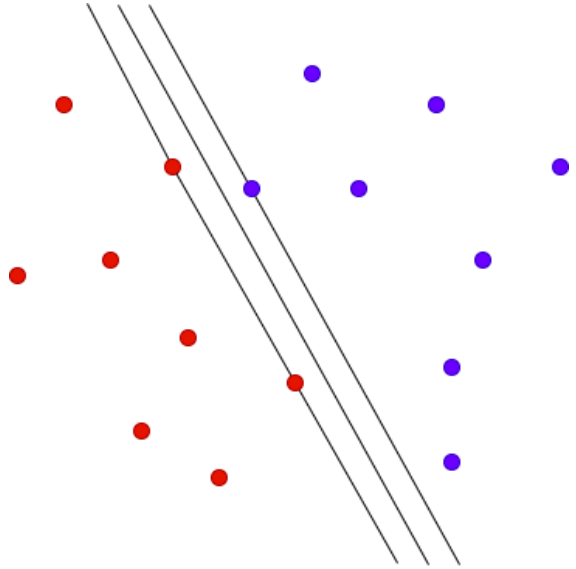
Support Vector Machines

- Already we are into complexities here!
- We talk about one line dividing the data, but sometimes a single straight line could not be used. In which case a *patchwork solution* is used. In our two dimensions, we use a line segment or collection of line segments.

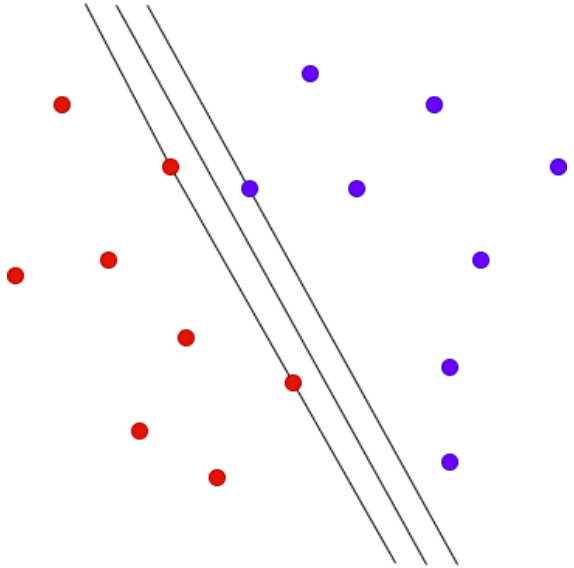


Support Vector Machines

- To find this ideal line, SVM picks the closest points to the line from the two sets, then draws parallel lines through the points.
- Those closest points are the *Support Vectors*.
- The dividing line is a third parallel line, of equal distance from the two given lines.
- A distance can be computed from all points to this line. All of the points in one set have a positive distance, all points in the other set have a negative distance.



Support Vector Machines



- This SVM can be given a new datapoint.
- It computes the distance from that point to the dividing line, then categorizes the point accordingly.
- The book was very sparse on this description. It could be that:
  - The final classifier is an ensemble of SVMs.
  - It could be a decision tree is built with the SVM at each node.
  - It could be there is a single SVM, and there will be misclassifications, but we pick the best line (hyperplane) to minimize the error.