Classification: Model Selection and Evaluation

Assume that a classification model is built from the training data.

Example: Suppose you used data from previous sales to build a classifier to predict customer purchasing behavior.

You would like an estimate of how accurately the classifier can predict the purchasing behavior of future customers, that is, future customer data on which the classifier has not been trained.

You may even have tried different methods to build more than one classifier and now wish to compare their accuracy.

- ***But what is accuracy?***
- ***How can we estimate it?***
- ***Are some measures of a classifier's accuracy more appropriate than others?***
- ***How can we obtain a reliable accuracy estimate?***

These questions are addressed in this section.

Evaluation metrics for the predictive accuracy of a classifier:

- ***Holdout***
- ***random subsampling***
- ***cross-validation and***
- ***bootstrap methods***

All these methods are based on randomly sampled partitions of the given data.

What if we have more than one classifier and want to choose the "best" one?

This is referred to as **Model Selection** (i.e., choosing one classifier over another).

How to use tests of statistical significance to assess whether the difference in accuracy between two classifiers is due to chance.

How to compare classifiers based on cost–benefit and receiver operating characteristic (ROC) curves.

Metrics for Evaluating Classifier Performance

Measures for assessing how good or how "accurate" your classifier is at predicting the class label of tuples.

Case-01: The class tuples are more or less evenly distributed,
Case-02: The classes are unbalanced (e.g., where an important class of interest is rare such as in medical tests).

The classifier evaluation measures include:

- ***Accuracy (also known as Recognition Rate)***
- ***Sensitivity (or Recall)***
- ***Specificity***
- ***Precision***
- ***F1, and F_β.***

Accuracy: It is also used as a general term to refer to a classifier's predictive abilities.

Using training data to derive a classifier and then estimate the accuracy of the resulting learned model can result in misleading overoptimistic estimates due to overspecialization of the learning algorithm to the data.

It is better to measure the classifier's accuracy on a test set consisting of class-labeled tuples that were not used to train the model.

Positive Tuples: Tuples of the main class of interest

Negative Tuples: All other tuples.

Given two classes, for example, the positive tuples may be *buys_computer = yes* while the negative tuples are *buys_computer = no*.

Suppose we use our classifier on a test set of labeled tuples.

P is the number of positive tuples and *N* is the number of negative tuples.

For each tuple, we compare the classifier's class label prediction with the tuple's known class label.

There are four additional terms we need to know that are the "building blocks" used in computing many evaluation measures.

Understanding them will make it easy to grasp the meaning of the various measures.

True Positives (*TP)*: These refer to the positive tuples that were correctly labeled by the classifier.

True Negatives (*TN)*: These are the negative tuples that were correctly labeled by the classifier.

False Positives (*FP*): These are the negative tuples that were incorrectly labeled as positive (e.g., tuples of class *buys_computer=no* for which the classifier predicted *buys_computer=yes*).

False Negatives (*FN*): These are the positive tuples that were mislabeled as negative (e.g., tuples of class *buys computer* D *yes* for which the classifier predicted *buys_computer=no*).

These terms are summarized in the **confusion matrix.**

		Predicted class		
		yes	*no*	Total
Actual class	*yes*	*TP*	*FN*	*P*
	no	*FP*	*TN*	*N*
	Total	P'	N'	$P + N$

Figure: Confusion matrix, shown with totals for positive and negative tuples.

Classes	*buys_computer = yes*	*buys_computer = no*	*Total*	*Recognition (%)*
buys_computer = yes	**6954**	**46**	7000	99.34
buys_computer = no	**412**	**2588**	3000	86.27
Total	7366	2634	10,000	95.42

Figure: Confusion matrix for the classes *buys computer* D *yes* and *buys computer* D *no,* where an entry in row *i* and column *j* shows the number of tuples of class *i* that were labeled by the classifier as class *j*. Ideally, the nondiagonal entries should be zero or close to zero.

The **Confusion Matrix** is a useful tool for analyzing how well your classifier can recognize tuples of different classes.

TP* and *TN tell us when the classifier is getting things right.

FP* and *FN tell us when the classifier is getting things wrong (i.e., mislabeling).

Given m classes (where $m \geq 2$), a **confusion matrix** is a table of at least size m by m.

$CM_{i,j}$: in the first m rows and m columns indicates the number of tuples of class i that were labeled by the classifier as class j.

For a classifier to have good accuracy, ideally most of the tuples would be represented along the diagonal of the confusion matrix, from entry $CM_{1,1}$ **to entry** $CM_{m,m}$, with the rest of the entries being zero or close to zero.

i.e. Ideally, *FP* and *FN* are around zero.

The table may have additional rows or columns to provide totals.

For example, in the confusion matrix of Figure above, *P* and *N* are shown.

In addition, *P*0 is the number of tuples that were labeled as positive **(*TP* + *FP*)** and ***N*'** is the number of tuples that were labeled as negative ***(TN + FN)***.

Total number of tuples: ***(TP + TN + FP +* F*N)***, or ***(P + N)***, or ***(P' + N')***.

Note that although the confusion matrix shown is for a binary classification problem, confusion matrices can be easily drawn for multiple classes in a similar manner.

Accuracy: On a given test set is the percentage of test set tuples that are correctly classified by the classifier.

$$accuracy = \frac{TP + TN}{P + N}$$

In the pattern recognition literature, this is also referred to as the overall **recognition rate** of the classifier, that is, it reflects how well the classifier recognizes tuples of the various classes.

An example of a confusion matrix for the two classes *buys_computer = yes* (positive) and *buys_computer = no* (negative) is given in Figure above.

Totals are shown, as well as the recognition rates per class and overall.

By glancing at a confusion matrix, it is easy to see if the corresponding classifier is confusing two classes.

For example, we see that it mislabeled 412 *"no"* tuples as *"yes."*

Accuracy is most effective when the class distribution is relatively balanced.

Error Rate or **Misclassification Rate (**of a classifier, *M)* = 1-*accuracy*(*M*).

This also can be computed as:

$$error\ rate = \frac{FP + FN}{P + N}$$

If we were to use the training set (instead of a test set) to estimate the error rate of a model, this quantity is known as the **Resubstitution Error**.

This error estimate is optimistic of the true error rate (and similarly, the corresponding accuracy estimate is optimistic) because the model is not tested on any samples that it has not already seen.

Measure	*Formula*
accuracy, recognition rate	$\frac{TP+TN}{P+N}$
error rate, misclassification rate	$\frac{FP+FN}{P+N}$
sensitivity, true positive rate, recall	$\frac{TP}{P}$
specificity, true negative rate	$\frac{TN}{N}$
precision	$\frac{TP}{TP+FP}$
F, F_1, F-score, harmonic mean of precision and recall	$\frac{2 \times precision \times recall}{precision + recall}$
F_β, where β is a non-negative real number	$\frac{(1+\beta^2) \times precision \times recall}{\beta^2 \times precision + recall}$

Figure: Evaluation measures. Note that some measures are known by more than one name. *TP*,*TN*,*FP*,*P*, *N* refer to the number of true positive, true negative, false positive, positive, and negative samples, respectively.

The E-Measure

- A measure that combines recall and precision
- The idea is to allow the user to specify whether he is more interested in recall or in precision
- The E measure is defined as follows

$$E(j) = 1 - \frac{1 + b^2}{\frac{b^2}{r(j)} + \frac{1}{P(j)}}$$

- where
  - $r(j)$ is the recall at the j-th position in the ranking
  - $P(j)$ is the precision at the j-th position in the ranking
  - $b \geq 0$ is a user specified parameter
  - $E(j)$ is the E metric at the j-th position in the ranking

- The parameter b is specified by the user and reflects the relative importance of recall and precision
- If $b = 0$
  - $E(j) = 1 - P(j)$
  - low values of b make $E(j)$ a function of precision
- If $b \rightarrow \infty$
  - $lim_{b \rightarrow \infty} \ E(j) = 1 - r(j)$
  - high values of b make $E(j)$ a function of recal
- For $b = 1$, the E-measure becomes the F-measure

- The function F assumes values in the interval $[0, 1]$
- It is 0 when no relevant documents have been retrieved and is 1 when all ranked documents are relevant
- Further, the harmonic mean F assumes a high value only when both recall and precision are high
- To maximize F requires finding the best possible compromise between recall and precision
- Notice that setting $b = 1$ in the formula of the E-measure yields

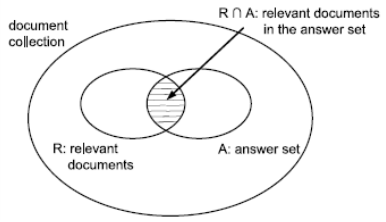
$$F(j) = 1 - E(j)$$

Introduction

- *Retrieval performance evaluation* consists of associating a quantitative metric to the results produced by an IR system
  - This metric should be directly associated with the relevance of the results to the user
  - Usually, its computation requires comparing the results produced by the system with results suggested by humans for a same set of queries

Precision and Recall

- Consider,
  - I: an information request
  - R: the set of relevant documents for I
  - A: the answer set for I, generated by an IR system
  - $R \cap A$: the intersection of the sets R and A

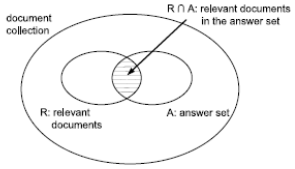


- The recall and precision measures are defined as follows
  - **Recall** is the fraction of the relevant documents (the set R) which has been retrieved i.e.,

$$Recall = \frac{|R \cap A|}{|R|}$$

  - **Precision** is the fraction of the retrieved documents (the set A) which is relevant i.e.,

$$Precision = \frac{|R \cap A|}{|A|}$$



- The definition of precision and recall assumes that all docs in the set A have been examined
- However, the user is not usually presented with all docs in the answer set A at once
  - User sees a ranked set of documents and examines them starting from the top
- Thus, precision and recall vary as the user proceeds with their examination of the set A
- Most appropriate then is to plot a **curve of precision versus recall**

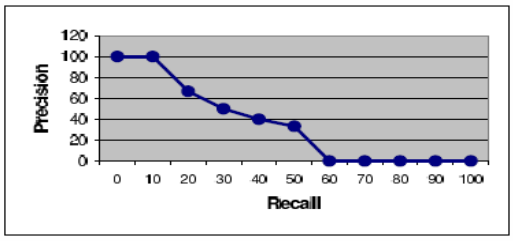
- Consider a reference collection and a set of test queries
- Let R_{q_1} be the set of relevant docs for a query q_1:
  - $R_{q_1} = \{d_3, d_5, d_9, d_{25}, d_{39}, d_{44}, d_{56}, d_{71}, d_{89}, d_{123}\}$
- Consider a new IR algorithm that yields the following answer to q_1 (relevant docs are marked with a bullet):

01. d_{123} •	06. d_9 •	11. d_{38}
02. d_{84}	07. d_{511}	12. d_{48}
03. d_{56} •	08. d_{129}	13. d_{250}
04. d_6	09. d_{187}	14. d_{113}
05. d_8	10. d_{25} •	15. d_3 •

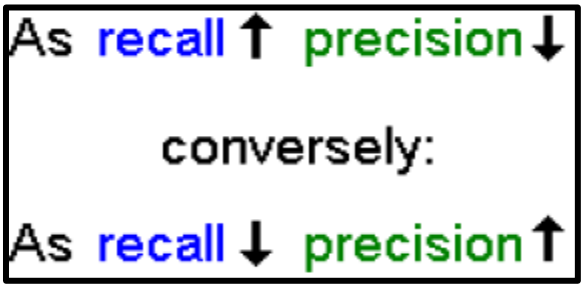
- If we examine this ranking, we observe that
  - The document d_{123}, ranked as number 1, is relevant
    - This document corresponds to 10% of all relevant documents
    - Thus, we say that we have a precision of 100% at 10% recall
  - The document d_{56}, ranked as number 3, is the next relevant
    - At this point, two documents out of three are relevant, and two of the ten relevant documents have been seen
    - Thus, we say that we have a precision of 66.6% at 20% recall

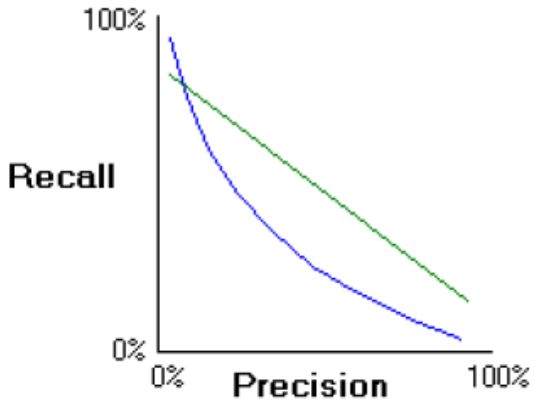
01. d_{123} •	06. d_9 •	11. d_{38}
02. d_{84}	07. d_{511}	12. d_{48}
03. d_{56} •	08. d_{129}	13. d_{250}
04. d_6	09. d_{187}	14. d_{113}
05. d_8	10. d_{25} •	15. d_3 •

- If we proceed with our examination of the ranking generated, we can plot a curve of precision versus recall as follows:



Recall	Precision
0	100
10	100
20	66.6
30	50
40	40
50	33.3
60	0
70	0
80	0
90	0
100	0





In the graph above, the two lines may represent the performance of different search systems.

While the exact slope of the curve may vary between systems, the general inverse relationship between recall and precision remains.

Algorithm 5.4 Perceptron learning algorithm.

1: Let $D = \{(\mathbf{x}_i, y_i) \mid i = 1, 2, \ldots, N\}$ be the set of training examples.
2: Initialize the weight vector with random values, $\mathbf{w}^{(0)}$
3: **repeat**
4: **for** each training example $(\mathbf{x}_i, y_i) \in D$ **do**
5: Compute the predicted output $\hat{y}_i^{(k)}$
6: **for** each weight w_j **do**
7: Update the weight, $w_j^{(k+1)} = w_j^{(k)} + \lambda(y_i - \hat{y}_i^{(k)})x_{ij}$.
8: **end for**
9: **end for**
10: **until** stopping condition is met

SUPPORT VECTOR MACHINES

A method for the classification of both linear and nonlinear data.

It uses a nonlinear mapping to transform the original training data into a higher dimension.

Within this new dimension, it searches for the linear optimal separating hyperplane (i.e., a "decision boundary" separating the tuples of one class from another).

With an appropriate nonlinear mapping to a sufficiently high dimension, data from two classes can always be separated by a hyperplane.

The SVM finds this hyperplane using *support vectors* ("essential" training tuples) and *margins* (defined by the support vectors).

Although the training time of even the fastest SVMs can be extremely slow, they are highly accurate, owing to their ability to model complex nonlinear decision boundaries.

They are much less prone to overfitting than other methods.

The support vectors found also provide a compact description of the learned model.

SVMs can be used for numeric prediction as well as classification.

Applications:

- Handwritten digit recognition,
- Object recognition,
- speaker identification,
- Benchmark time-series prediction tests.

The Case When the Data Are Linearly Separable

To explain the mystery of SVMs, let's first look at the simplest case—a two-class problem where the classes are linearly separable.

Let the data set D be given as $(\boldsymbol{X}_1, y_1)$, $(\boldsymbol{X}_2, y_2)$, … , $(\boldsymbol{X}_D, y_D)$, where $\boldsymbol{X}_i$ is the set of training tuples with associated class labels, y_i.

Each y_i can take one of two values, either +1 or -1 (i.e. $y_i \ \varepsilon \ \{+1, -1\}$), corresponding to the classes *buys_computer = yes* and *buys_computer = no*, respectively.

To aid in visualization, let's consider an example based on two input attributes, A_1 and A_2, as shown in Figure below:

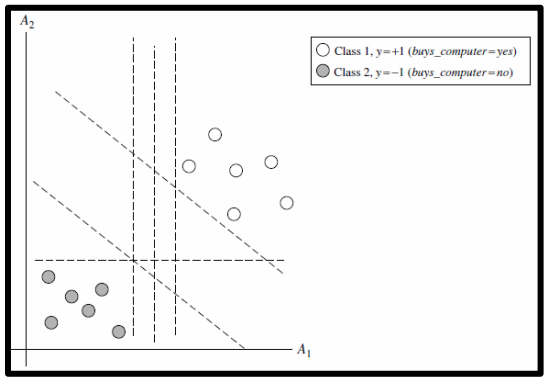


Figure: The 2-D training data are linearly separable. There are an infinite number of possible separating hyperplanes or "decision boundaries," some of which are shown here as dashed lines. Which one is best?

From the graph, we see that the 2-D data are **linearly separable** (or "linear," for short), because a straight line can be drawn to separate all the tuples of class +1 from all the tuples of class -1.

There are an infinite number of separating lines that could be drawn.

We want to find the "best" one, that is, one that (we hope) will have the minimum classification error on previously unseen tuples.

How can we find this best line? Note that if our data were 3-D (i.e., with three attributes), we would want to find the best separating *plane*.

Generalizing to *n* dimensions, we want to find the best *hyperplane*.

We will use "hyperplane" to refer to the decision boundary that we are seeking, regardless of the number of input attributes.

So, in other words, how can we find the best hyperplane?

An SVM approaches this problem by searching for the **Maximum Marginal Hyperplane**.

Consider Figure below, which shows two possible separating hyperplanes and their associated margins.

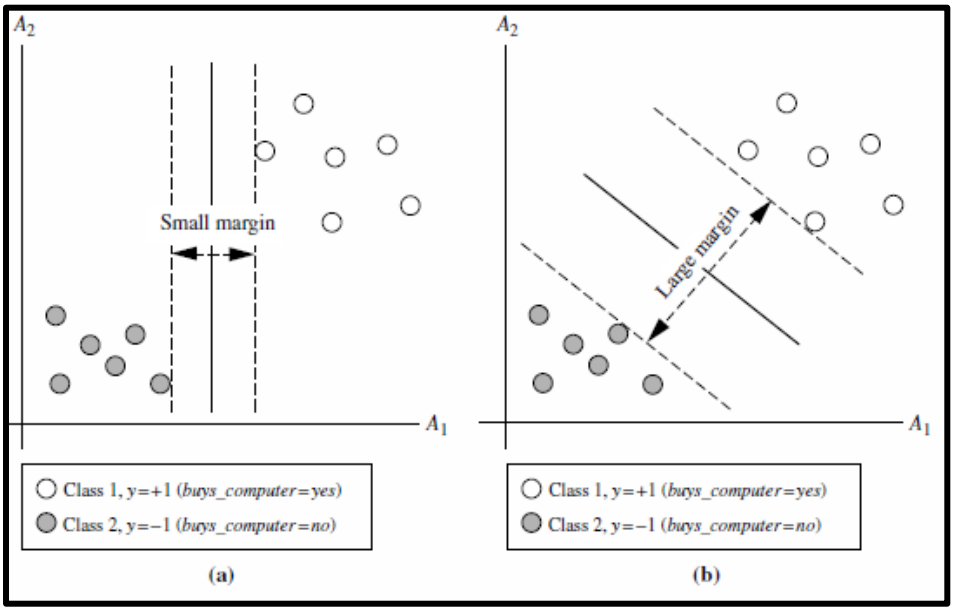


Figure: Here we see just two possible separating hyperplanes and their associated margins. Which one is better? The one with the larger margin (b) should have greater generalization accuracy.

Define the hyperplanes H such that:

$w \cdot x_i + b \geq +1$ when $y_i = +1$

$w \cdot x_i + b \leq -1$ when $y_i = -1$

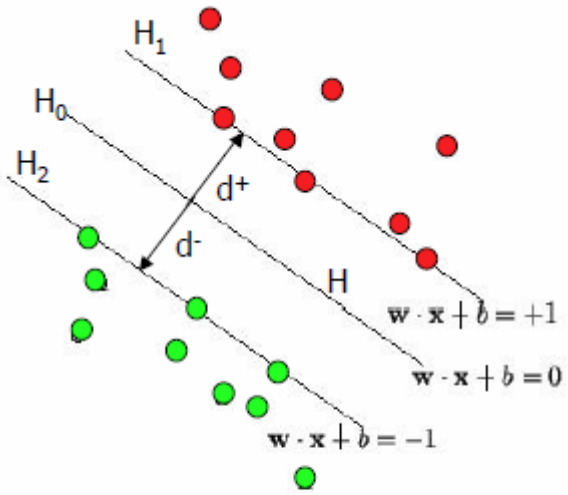
H_1 and H_2 are the planes:

H_1: $w \cdot x_i + b = +1$

H_2: $w \cdot x_i + b = -1$

The points on the planes H_1 and H_2 are the tips of the Support Vectors

The plane H_0 is the median in between, where $w \cdot x_i + b = 0$



d+ = the shortest distance to the closest positive point

d- = the shortest distance to the closest negative point

The margin (gutter) of a separating hyperplane is d+ + d–.

Before we get into the definition of margins, let's take an intuitive look at this figure.

Both hyperplanes can correctly classify all the given data tuples.

Intuitively, however, we expect the hyperplane with the larger margin to be more accurate at classifying future data tuples than the hyperplane with the smaller margin.

This is why (during the learning or training phase) the SVM searches for the hyperplane with the largest margin, that is, the *maximum marginal hyperplane* (MMH).

The associated margin gives the largest separation between classes.

Getting to an informal definition of **margin**, we can say that the shortest distance from a hyperplane to one side of its margin is equal to the shortest distance from the hyperplane to the other side of its margin, where the "sides" of the margin are parallel to the hyperplane.

When dealing with the MMH, this distance is, in fact, the shortest distance from the MMH to the closest training tuple of either class.

A **separating hyperplane** can be written as:

$$\mathbf{W}.\mathbf{X} + b = 0$$

Where **W** is a weight vector, namely, **W** = {w1, w2, …, wn}; n is the number of attributes; and b is a scalar, often referred to as a bias.

To aid in visualization, let's consider two input attributes, A_1 and A_2, as in Figure above.

Training tuples are 2-D (e.g., **X** = (x_1, x_2)), where x1 and x2 are the values of attributes A_1 and A_2, respectively, for **X**.

If we think of b as an additional weight, w_0, we can rewrite the above equation as:

$$w_0 + w_1 x_1 + w_2 x_2 = 0$$

Thus, any point that lies above the separating hyperplane satisfies:

$$w_0 + w_1 x_1 + w_2 x_2 > 0$$

Similarly, any point that lies below the separating hyperplane satisfies:

$$w_0 + w_1 x_1 + w_2 x_2 < 0$$

The weights can be adjusted so that the hyperplanes defining the "sides" of the margin can be written as:

$$H_1 : w_0 + w_1 x_1 + w_2 x_2 \geq 1 \quad \text{for } y_i = +1,$$
$$H_2 : w_0 + w_1 x_1 + w_2 x_2 \leq -1 \quad \text{for } y_i = -1.$$

That is, any tuple that falls on or above *H*1 belongs to class +1, and any tuple that falls on or below *H*2 belongs to class -1.

Combining the above two inequalities:

$$y_i(w_0 + w_1 x_1 + w_2 x_2) \geq 1, \ \forall i.$$

Any training tuples that fall on hyperplanes *H*1 or *H*2 (i.e., the "sides" defining the margin) satisfy Eq. (9.18) and are called **support vectors**.

That is, they are equally close to the (separating) MMH.

In Figure below, the support vectors are shown encircled with a thicker border.

Essentially, the support vectors are the most difficult tuples to classify and give the most information regarding classification.

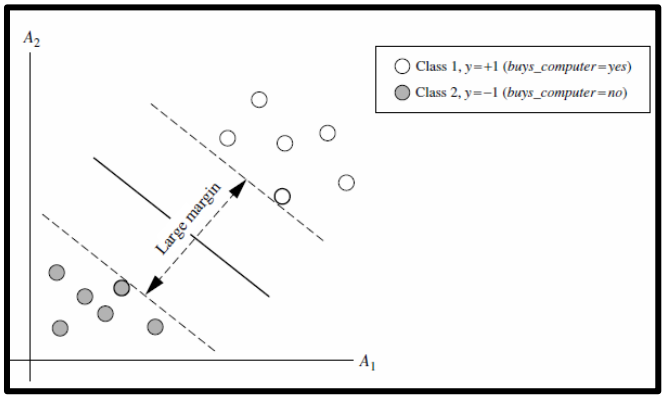


Figure: Support vectors. The SVM finds the maximum separating hyperplane, that is, the one with maximum distance between the nearest training tuples. The support vectors are shown with a thicker border.

Once we've found the support vectors and MMH (note that the support vectors define the MMH!), we have a trained support vector machine.

The MMH is a linear class boundary, and so the corresponding SVM can be used to classify linearly separable data. We refer to such a trained SVM as a **Linear SVM**.

The distance between the two hyperplanes for the positive and negative instances is also referred to as the margin.

The goal is to maximize this margin.

What is the distance (or margin) between these two parallel hyperplanes?

One can use linear algebra to show that the distance between two parallel hyperplanes is the normalized difference between their constant terms, where the normalization factor is the $L2$-norm:

$$||\overline{W}|| = \sqrt{\sum_{i=1}^{d} w_i^2}$$

of the coefficients.

Because the difference between the constant terms of the two aforementioned hyperplanes is 2, it follows that the distance between them is $2/||W||$.

This is the margin that needs to be maximized with respect to the aforementioned constraints.

Note that each training data point leads to a constraint, which tends to make the optimization problem rather large, and explains the high computational complexity of SVMs.

However, perfect linear separability is a rather contrived scenario, and real data sets usually will not satisfy this property.

"*Once I've got a trained support vector machine, how do I use it to classify test (i.e., new) tuples?*"

Based on the Lagrangian formulation mentioned before, the MMH can be rewritten as the decision boundary.

$$d(\boldsymbol{X}^T) = \sum_{i=1}^{l} y_i \alpha_i \boldsymbol{X}_i \boldsymbol{X}^T + b_0$$

where y_i is the class label of support vector $\boldsymbol{X}_i$; $\boldsymbol{X}^T$ is a test tuple; α_i and b_0 are numeric parameters that were determined automatically by the optimization or SVM algorithm noted before; and l is the number of support vectors.

α_i are Lagrangian multipliers.

For linearly separable data, the support vectors are a subset of the actual training tuples.

Given a test tuple, $\boldsymbol{X}^T$, we plug it into the above equation, and then check to see the sign of the result.

This tells us on which side of the hyperplane the test tuple falls.

If the sign is positive, then $\boldsymbol{X}^T$ falls on or above the MMH, and so the SVM predicts that $\boldsymbol{X}^T$ belongs to class +1 (representing *buys computer* D *yes*, in our case).

If the sign is negative, then $\boldsymbol{X}^T$ falls on or below the MMH and the class prediction is -1 (representing

buys computer D *no*).

Notice that the Lagrangian formulation of our problem (Eq. 9.19) contains a dot product between support vector $\boldsymbol{X}_i$ and test tuple $\boldsymbol{X}^T$.

This will prove very useful for finding the MMH and support vectors for the case when the given data are nonlinearly separable,

Before we move on to the nonlinear case, there are two more important things to note.

The complexity of the learned classifier is characterized by the number of support vectors rather than the dimensionality of the data.

Hence, SVMs tend to be less prone to overfitting than some other methods.

The support vectors are the essential or critical training tuples—they lie closest to the decision boundary (MMH).

If all other training tuples were removed and training were repeated, the same separating hyperplane would be found.

Furthermore, the number of support vectors found can be used to compute an (upper) bound on the expected error rate of the SVM classifier, which is independent of the data dimensionality.

An SVM with a small number of support vectors can have good generalization, even when the dimensionality of the data is high.

The Case When the Data Are Linearly Inseparable

What if the data are not linearly separable?

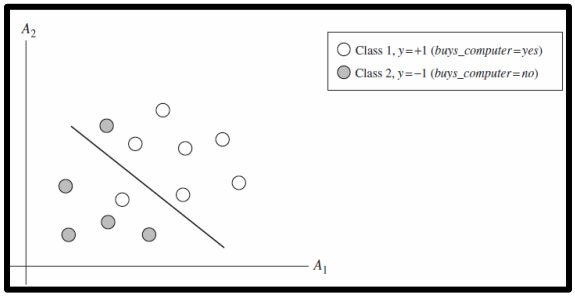


Figure: The decision boundary is nonlinear

Example:

Nonlinear transformation of original input data into a higher dimensional space. Consider the following example. A 3-D input vector $X = (x_1, x_2, x_3)$ is mapped into a 6-D space, Z, using the mappings $\phi_1(X) = x_1$, $\phi_2(X) = x_2$, $\phi_3(X) = x_3$, $\phi_4(X) = (x_1)^2$, $\phi_5(X) = x_1x_2$, and $\phi_6(X) = x_1x_3$. A decision hyperplane in the new space is $d(Z) = WZ + b$, where W and Z are vectors. This is linear. We solve for W and b and then substitute back so that the linear decision hyperplane in the new (Z) space corresponds to a nonlinear second-order polynomial in the original 3-D input space:

$$\begin{aligned} d(Z) &= w_1x_1 + w_2x_2 + w_3x_3 + w_4(x_1)^2 + w_5x_1x_2 + w_6x_1x_3 + b \\ &= w_1z_1 + w_2z_2 + w_3z_3 + w_4z_4 + w_5z_5 + w_6z_6 + b. \end{aligned}$$

But there are some problems.

First, how do we choose the nonlinear mapping to a higher dimensional space?

Second, the computation involved will be costly.

Refer to the above equation for the classification of a test tuple, $\boldsymbol{X}^{\mathbf{T}}$.

Given the test tuple, we have to compute its dot product with every one of the support vectors.

In training, we have to compute a similar dot product several times in order to find the MMH.

This is especially expensive.

Hence, the dot product computation required is very heavy and costly.

We need another trick!

Luckily, we can use another math trick. It so happens that in solving the quadratic optimization problem of the linear SVM (i.e., when searching for a linear SVM in the new higher dimensional space), the training tuples appear only in the form of dot products, $\phi(X_i) \cdot \phi(X_j)$, where $\phi(X)$ is simply the nonlinear mapping function applied to transform the training tuples. Instead of computing the dot product on the transformed data tuples, it turns out that it is mathematically equivalent to instead apply a *kernel function*, $K(X_i, X_j)$, to the original input data. That is,

$$K(X_i, X_j) = \phi(X_i) \cdot \phi(X_j).$$

In other words, everywhere that $\phi(\mathbf{X}_i)$. $\phi(\mathbf{X}_j)$ appears in the training algorithm, we can replace it with $K(\mathbf{X}_i, \mathbf{X}_j)$.

In this way, all calculations are made in the original input space, which is of potentially much lower dimensionality.

After applying this trick, we can then proceed to find a maximal separating hyperplane.

"*What are some of the kernel functions that could be used?*"

Properties of the kinds of kernel functions that could be used to replace the dot product scenario just described have been studied.

Three admissible kernel functions are:

Polynomial kernel of degree *h*:	$K(X_i, X_j) = (X_i \cdot X_j + 1)^h$
Gaussian radial basis function kernel:	$K(X_i, X_j) = e^{-\|X_i - X_j\|^2/2\sigma^2}$
Sigmoid kernel:	$K(X_i, X_j) = \tanh(\kappa X_i \cdot X_j - \delta)$

There are no golden rules for determining which admissible kernel will result in the most accurate SVM.

In practice, the kernel chosen does not generally make a large difference in resulting accuracy.

SVM training always finds a global solution, unlike neural networks, such as backpropagation, where many local minima usually exist.

So far, we have described linear and nonlinear SVMs for binary (i.e., two-class) classification.

SVM classifiers can be combined for the multiclass case.

Methods for Constructing an Ensemble Classifier

The basic idea is to construct multiple classifiers from the original data and then aggregate their predictions when classifying unknown examples.

(1) Manipulating the Training Set:

Multiple training sets are created by resampling the original data according to sampling distribution.

The sampling distribution determines how likely is that an example is selected for training, and it may vary from one trial to another.

A classifier is then built from each training set using a particular learning algorithm.

Bagging and **Boosting** are 2 examples of Ensemble methods that manipulate their training sets.

(2) Manipulating the Input Features:

A subset of the input features is chosen to form the training set.

The subset can be chosen either randomly or based on the recommendations of the domain experts.

Some studies have shown that this approach works very well with data sets that contain highly redundant features.

Random Forest is an ensemble method that manipulates its features and uses decision trees as its base classifiers.

(3) Manipulating Class Labels:

This method can be used when the number of classes is sufficiently large.

The training data is transformed into a binary class problem by randomly partitioning the class labels into two disjoint subsets A_0 and A_1.

Training examples whose class labels the subset A0 are assigned to class 0, while those that belong to subset A1 are assigned to class 1.

The relabeled examples are then used to train a base classifier.

By repeating the class relabeling and model-building multiple times, an ensemble of base classifiers is obtained.

When a test example is presented, each base classifer Ci is used to predict its class label.

If the test example is predicted as class 0, then all the classes that belong to A0 will receive a vote.

Conversely, if ti is predicted to be class 1, then all the classes that belong to A1 will receive a vote.

The votes are tallied and the class that receives the highest vote is assigned to the test example.
Example: Error-Correcting-Output-Coding method.

(4) Manipulating the Learning Algorithm:

Many learning algorithms can be manipulated in such a way that applying the algorithm several times on the same training data may result in different models.

For example, an ANN can produce different models by changing its network topology or the initial weights of the links between neurons.

Similarly, an ensemble of decision trees can be constructed by injecting randomness into the tree-growing procedure.

For example, instead of choosing the best split attribute at each node, we can randomly choose one of the top k attributes for splitting.

The first three approaches are generic methods that are applicable to any classifiers.

The fourth approach depends on the type of the classifier used.

The base classifiers for most of these approaches can be generated sequentially or in parallel (all at once).

The algorithm 5.5 below shows the steps needed to build an ensemble classifier in a sequential manner.

Step-01: Create a training set from the original data D.

Depending on the type of ensemble method used, the training sets are either identical to or slight modifications of the data set D.

The size of the training set is often kept the same as the original data, but the distribution of examples may not be identical; i.e. some examples may appear multiple times in the training set, while others may not appear even once.

A base classifier C_i is then constructed from each training set D_i.

Ensemble methods work better with unstable classifiers, i.e. base classifers that are sensitive to minor perturbations in the training set.
Examples of unstable classifiers: Decision trees, Rule-based classifiers, and ANN's.

Note: The variability among training examples is one of the primary sources of errors in a classifier.
By aggregating the base classifiers built from different training sets, this may help to reduce the types of errors.

Finally a test example **'X'** is classified by combining the predictions made by the classifers Ci(**X):**

$$C^*(\mathbf{x}) = Vote(C_1(\mathbf{x}), C_2(\mathbf{x}), \ldots, C_k(\mathbf{x})).$$

The class can be obtained by taking a majority vote on the individual predictions or by weighting each prediction with the accuracy of the base classifier.

Algorithm 5.5: General procedure for Ensemble Method (Page No. 280 in Pang-nin Tang textbook)

Algorithm 5.5 General procedure for ensemble method.

1: Let D denote the original training data, k denote the number of base classifiers, and T be the test data.
2: **for** $i = 1$ to k **do**
3: Create training set, D_i from D.
4: Build a base classifier C_i from D_i.
5: **end for**
6: **for** each test record $x \in T$ **do**
7: $C^*(x) = Vote(C_1(\mathbf{x}), C_2(\mathbf{x}), \ldots, C_k(\mathbf{x}))$
8: **end for**

<u>Bias-Variance Decomposition</u> **(Page No. 281 in Pang-nin Tang textbook)**

BAGGING AND BOOSTING

First, let me explain what Bagging and Boosting is and then delineate the differences.

Both Boosting and Bagging are ensemble methods and meta learners

Boosting Steps:

1. Draw a random subset of training samples d1 without replacement from the training set D to train a weak learner C1
2. Draw second random training subset d2 without replacement from the training set and add 50 percent of the samples that were previously falsely classified/misclassified to train a weak learner C2
3. Find the training samples d3 in the training set D on which C1 and C2 disagree to train a third weak learner C3
4. Combine all the weak learners via majority voting.

Bagging:

Before understand Bagging lets understand the concept of Bootstrap which is nothing but choosing a Random sample with replacement.

As everyone pointed Bagging is nothing but **B**ootstrap **AGG**regat**ING**

1. Generate *n* different bootstrap training sample
2. Train Algorithm on each bootstrapped sample separately
3. Average the predictions at the end

One of the Key differences is the way how use sample each training set. Bagging allows replacement in bootstrapped sample but Boosting doesn't.

In theory Bagging is good for reducing variance(Over-fitting) where as Boosting helps to reduce both Bias and Variance as per this Boosting Vs Bagging, but in practice Boosting (**Ada**ptive **Boos**ting) know to have high variance because of over-fitting.

Bootstrap

In the bootstrap method, the labeled data is sampled uniformly *with replacement*, to create a training data set, which might possibly contain duplicates.

The labeled data of size n is sampled n times with replacement.

This results in a training data with the same size as the original labeled data.

However, the training typically contains duplicates and also misses some points in the original labeled data.

The probability that a particular data point is not included in a sample is given by $(1-1/n)$.

Therefore, the probability that the data point is not included in n samples is given by $(1-1/n)^n$.

For large values of n, this expression evaluates to approximately $1/e$, where e is the base of the natural logarithm.

The fraction of the labeled data points included at least once in the training data is therefore $1-1/e \approx 0.632$.

The training model M is constructed on the bootstrapped sample containing duplicates.

The overall accuracy is computed using the original set of full labeled data as the test examples.

The estimate is highly optimistic of the true classifier accuracy because of the large overlap between training and test examples.

In fact, a 1-nearest neighbor classifier will always yield 100% accuracy for the portion of test points included in the bootstrap sample and the estimates are therefore not realistic in many scenarios.

By repeating the process over b different bootstrap samples, the mean and the variance of the error estimates may be determined.

A better alternative is to use *leave-one-out bootstrap*. In this approach, the accuracy $A(\overline{X})$ of each labeled instance $\overline{X}$ is computed using the classifier performance on only the subset of the b bootstrapped samples in which $\overline{X}$ is not a part of the bootstrapped sample of training data. The overall accuracy A_l of the leave-one-out bootstrap is the mean value of $A(\overline{X})$ over all labeled instances $\overline{X}$. This approach provides a pessimistic accuracy estimate. The 0.632-bootstrap further improves this accuracy with a "compromise" approach. The average *training-data* accuracy A_t over the b bootstrapped samples is computed. This is a highly optimistic estimate. For example, A_t will always be 100 % for a 1-nearest neighbor classifier. The overall accuracy A is a weighted average of the leave-one-out accuracy and the training-data accuracy.

$$A = (0.632) \cdot A_l + (0.368) \cdot A_t$$