Fundamentals of Learning Course Notes
Cynthia Rudin

Welcome! Machine learning is a broad field within machine learning and predictive statistics. It involves the design of algorithms that learn by example. Machine learning is (in a broad sense) pattern recognition. It is not real intelligence.

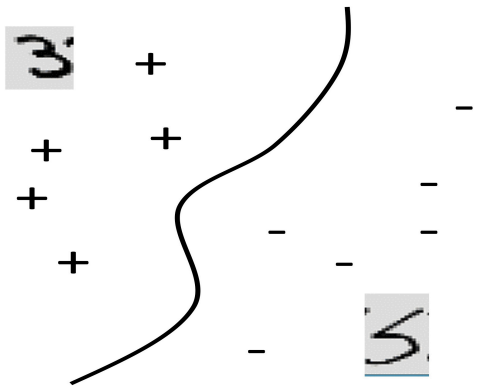
In machine learning, we often represent objects as vectors (that is, an ordered set of numbers). For instance, an image can be represented by its red-green-blue pixel values. A medical patient can be represented by numbers that represent the patient's age, number of prior strokes, whether they have had past congenital heart failure, whether they take specific dosages of specific drugs, etc.

I'll start by listing some of the important machine learning problems, starting with the most famous one: classification.

List of Some (Not All) Important Problems in Machine Learning

1. Classification

   - Input: $\{(x_i, y_i)\}_{i=1}^n$ "examples," "instances with labels," "observations"
   - $x_i \in \mathcal{X} \subset \mathbf{R}^p$. When $y_i \in \{-1, 1\}$ it is "binary classification."



   - Output: $f : \mathcal{X} \to \mathbf{R}$ and use $\text{sign}(f)$ to classify. If $\text{sign}(f) \neq y$, the point is misclassified. This can also be written $y \cdot \text{sign}(f) \leq 0$.
   - Applications: automatic handwriting recognition, face recognition, speech recognition, biometrics, document classification, predicting medical outcomes (e.g., disease vs no disease), predicting equipment failure, predicting credit default.

- Multiclass problems: If there are more than two classes, the problem is a multiclass problem. Binary classification techniques generalize nicely to multiclass. For instance, we could solve binary "one-vs-all" classification problems for each class.
- Algorithms: There are many famous classification algorithms, including logistic regression, AdaBoost, random forest, support vector machines, k-nearest neighbors, decision tree optimization algorithms, and various types of neural networks. People often use the term "classification" interchangeably with "conditional probability estimation" since you can generally get a probability from f.

2. Conditional probability estimation

- Input: $\{(x_i, y_i)\}_{i=1}^n$, $x_i \in \mathcal{X}, y_i \in \{-1, 1\}$
- Output: $f : \mathcal{X} \to [0, 1]$ as "close" to $p(y = 1|x)$ as possible.
- Applications: estimate probability of failure, probability to default on loan, probability to commit another crime
- Algorithms: Most classification algorithms, such as logistic regression, AdaBoost, decision trees, and neural networks can be used for conditional probability estimation too.

3. Regression

- Input: $\{(x_i, y_i)\}_{i=1}^n$, $x_i \in \mathcal{X}, y_i \in \mathbf{R}$
- Output: $f : \mathcal{X} \to \mathbf{R}$
- Applications: predicting an individual's income, predict house prices, predict demand for energy, predict test scores, predict amount that customer will spend
- Algorithms: Least squares, ridge regression, L1-penalized regression (lasso), kernel least squares, kernel ridge regression, kernel regression (which is different from kernel ridge regression), gaussian process regression

4. Ranking - between classification and regression.

- Input: $\{x_i\}_{i=1}^n$, $x_i \in \mathcal{X}$, and also we have a set of pairs (i, k) that are labeled that i should be ranked above k, that is, we should have $f(x_i) > f(x_k)$ for these special pairs.

- Output: $f : \mathcal{X} \to \mathbf{R}$ such that $f(x_i) > f(x_k)$ for the specified (i, k) pairs as often as possible.
- Applications: Some search engines use ranking methods. Useful also for predictive maintenance applications since you want to rank the equipment most likely to fail on top. Conditional probability estimation algorithms can be used for ranking if you have binary labels y, where you rank by f. In that case, one could view text generation algorithms as ranking algorithms, since, to determine what should be the next word in the sentence, they rank all words and choose the top one to include.

5. Finding patterns (correlations) in large datasets, rule mining, frequent itemset mining
   - Input: $\{x_i\}_{i=1}^n$, $x_i \in \mathcal{X}$, (labels can be included optionally)
   - Output for Frequent Itemset Mining: Find frequent combinations of items (e.g., Milk & Cookies).
   - Output for Rule Mining: Find rules such as (Diapers $\to$ Beer) that are "interesting" according to some interestingness criteria.
   - There are thousands of algorithms that all produce the same result - the list of frequent patterns. The Apriori and FP-Growth algorithms are the most famous. Rules themselves can serve as very small machine learning models (e.g., "People who buy diapers also tend to buy beer.")
6. Clustering - grouping data into clusters that "belong" together - objects within a cluster are more similar to each other than to those in other clusters. This is unsupervised.
   - Input: $\{x_i\}_{i=1}^n$, $x_i \in \mathcal{X}$
   - Output: $f : \mathcal{X} \to \{1, \ldots, K\}$ (K clusters)
   - Applications: clustering consumers for market research, clustering genes into families, image segmentation (medical imaging)
   - Algorithms: Kmeans, Kmedians, Hierarchical agglomerative clustering, Gaussian mixture models
7. Density estimation. This is unsupervised.
   - $\{x_i\}_{i=1}^n$, $x_i \in \mathcal{X}$

- Output: $f : \mathcal{X} \to [0, 1]$ as "close" to $p(x)$ as possible.
- Applications: anomaly detection (anomalous mechanical behavior of a piece of equipment), useful for variance estimates (lower density areas might have larger confidence intervals)
- Algorithms: Adaptive kernel density estimation methods are probably the most popular.

Clustering, density estimation, and (generally) rule mining are **unsupervised problems** (no ground truth), whereas classification, ranking, and conditional probability estimation are **supervised problems** (there is ground truth). In all of these problems, we do not necessarily assume we know the distribution (or even the form of the distribution) that the data are drawn from.

In "self-supervised" problems, the y comes from the x. E.g., language models, where you remove a word and try to predict the missing word using the surrounding words.

Training and Testing (in-sample and out-of-sample) for *supervised learning.*

Training: training data are input, and model f is the output.

$$\{(x_i, y_i)\}_{i=1}^n \Longrightarrow \boxed{\text{Algorithm}} \Longrightarrow f.$$

Testing: Evaluate the model. You want to predict y for a new x, where (x, y) comes from the *same* distribution as $\{(x_i, y_i)\}_{i=1}^n$.

That is, $(x, y) \sim D(\mathcal{X}, \mathcal{Y})$ and each $(x_i, y_i) \sim D(\mathcal{X}, \mathcal{Y})$.

Problems arise in practice when the training data and test data are not drawn from the same distribution.

To judge the quality of predictions, we need to know how to answer: How well does $f(x)$ match the label y? We will measure the goodness of f using a loss function $\ell : \mathcal{Y} \times \mathcal{Y} \to \mathbf{R}$. It is a function that takes $f(x)$ and y and produces a (generally nonnegative) number.

For instance,

$$\begin{aligned} \ell(f(x), y) &= (f(x) - y)^2 \quad \text{least squares loss, or} \\ \ell(f(x), y) &= \mathbf{1}_{[\text{sign}(f(x)) \neq y]} \quad \text{(mis)classification error} \end{aligned}$$

The expected loss over the distribution that the data comes from is:

$$\begin{aligned} R^{\text{test}}(f) &= \mathbf{E}_{(x,y)\sim D} \ell(f(x), y) \\ &= \int_{(x,y)\sim D} \ell(f(x), y) dD(x, y). \end{aligned}$$

R^{test} is also called the **true risk** or the **test error**. This is the main quantity considered in supervised learning.

We can't calculate it. But we want R^{test} to be small. If so, that would mean $f(x)$ is a good predictor of y.

How can we ensure $R^{\text{test}}(f)$ is small? The answer involves a few magic ingredients: a low training error, a large number of training points, and a simple model class. Let's discuss that.

Let's look at how well f performs (on average) on the training set $\{(x_i, y_i)\}_{i=1}^n$:

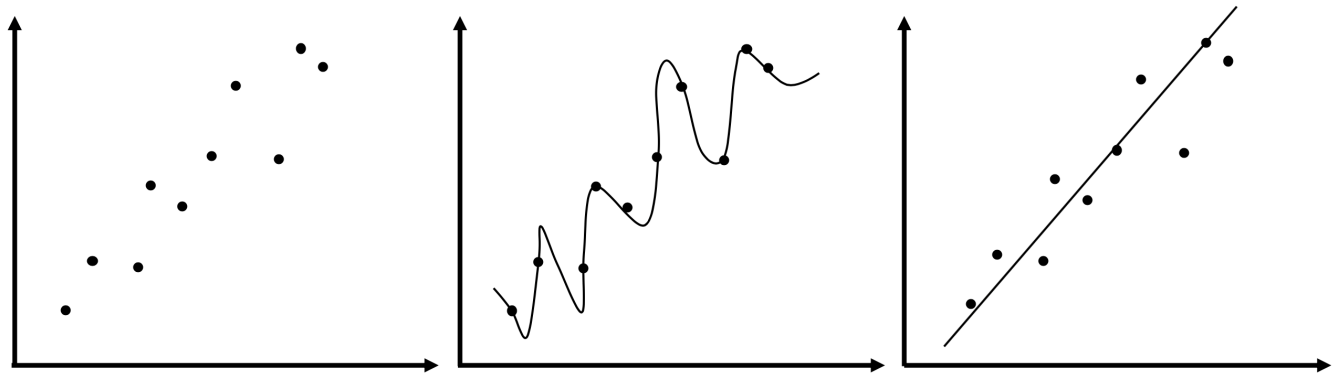
$$R^{\text{train}}(f) = \frac{1}{n} \sum_{i=1}^{n} \ell(f(x_i), y_i).$$

R^{train} is also called the **empirical risk** or **training error**. For example,

$$R^{\text{train}}(f) = \frac{1}{n} \sum_{i=1}^{n} \mathbf{1}_{[\text{sign}(f(x_i)) \neq y_i]}.$$

(For instance, this might represent how many handwritten digits f classified incorrectly.)

Say our algorithm constructs f so that $R^{\text{train}}(f)$ is small. If $R^{\text{train}}(f)$ is small, hopefully $R^{\text{test}}(f)$ is too. But that doesn't always happen. Sometimes we overfit. Overfitting is the enemy of machine learning.



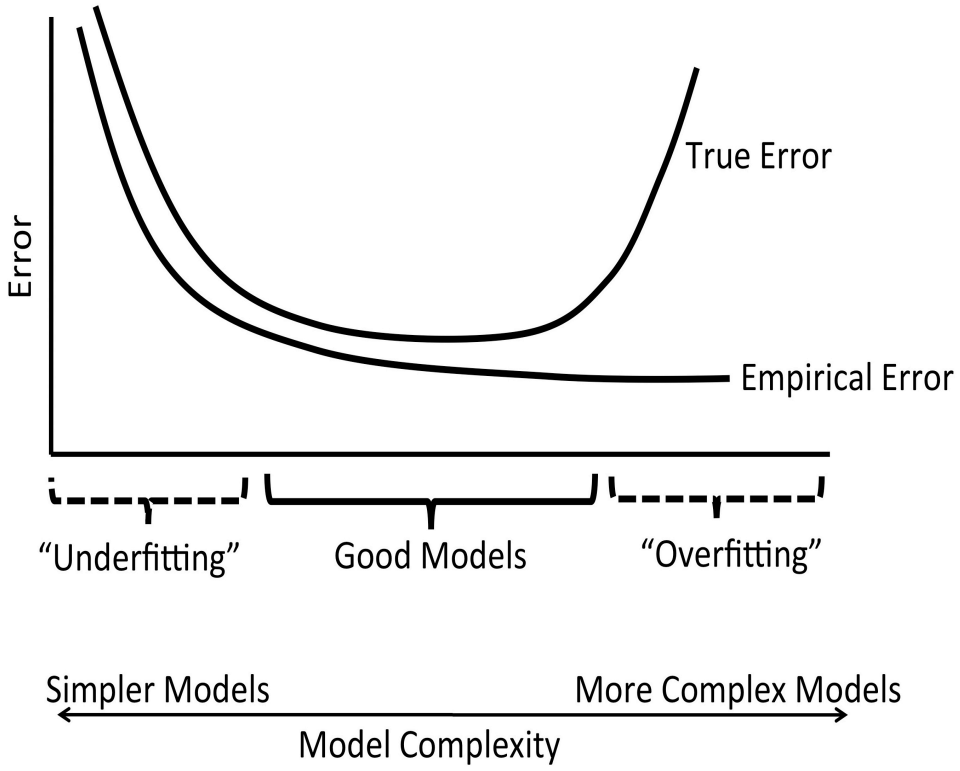
The left figure contains the data, and the right figure is a good fit. In the middle figure, f was overfitted to the data, modeled the noise, "memorized" the examples, and didn't give us much other useful information. In other words, it doesn't "generalize," i.e., predict. We didn't "learn" anything!

Thus, it's not enough to have a small training error, because we could overfit. We also need a guarantee on how close R^{train} is to R^{test}. Hence we need the other two ingredients for when R^{train} would be close to R^{test}:

- If n is large.
- If f is "simple."

A beautiful mathematical theory (discussed next) formalizes what these mean, and how we can get a guarantee on the test risk being close to the training risk. This guarantee is important, because it tells us that if we have all three ingredients (low training error, large training set, simple models), then our test risk is also likely to be small.

Computational Learning Theory, a.k.a. Statistical Learning Theory, a.k.a., Learning Theory, and in particular, Vapnik's **Structural Risk Minimization** (SRM) addresses generalization (Vapnik and Chervonenkis, 1971). Here's SRM's classic picture:



Structural Risk Minimization says that we need models to be somewhat simple in order to learn/generalize/avoid overfitting.

Statistical learning theory addresses how to construct probabilistic guarantees on the true risk. In order to do this, it quantifies classes of "simple models," discussed formally later.

What can we use as a definition of a "simple" model in practice?

There are many definitions. For instance:

- "simple" linear models have small coefficients.

$$f(x) = \sum_j \lambda_j x^{(j)} \text{ where } \|\boldsymbol{\lambda}\|_2^2 = \sum_j \lambda_j^2 < C.$$

- "simple" models could be models that are very smooth, and can't change too fast as we move around the space $\mathcal{X}$.
- "simple" models have small values of a "simplicity function" which we generally call a regularization term.

- "simple" Bayesian models might be models that have large values of a Bayesian prior. (As we will learn, this is equivalent to having a small regularization term.)

Later we will learn some formal definitions of simplicity.

This expression below is kind of omnipresent. This form captures many algorithms: SVM, boosting, ridge regression, LASSO, and logistic regression.

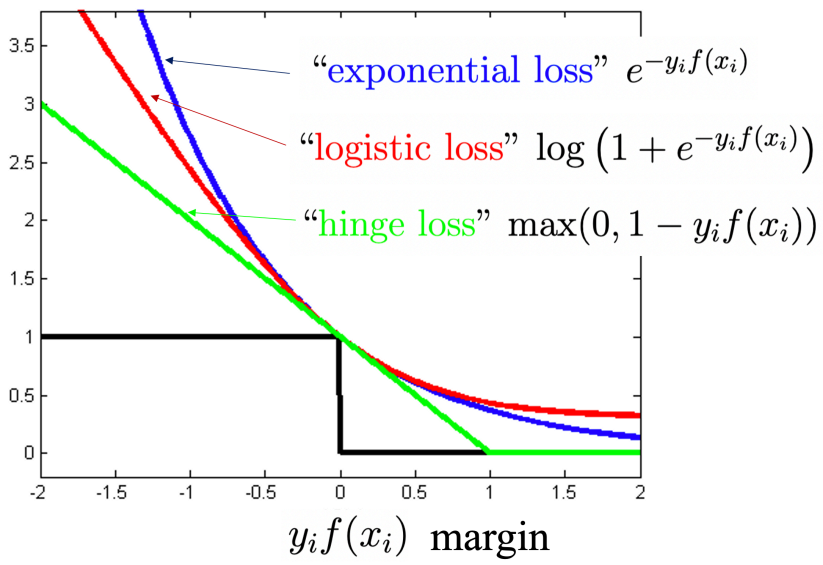
Regularized Learning Expression:

$$\sum_i \ell(f(x_i), y_i) + CR^{\text{reg}}(f).$$

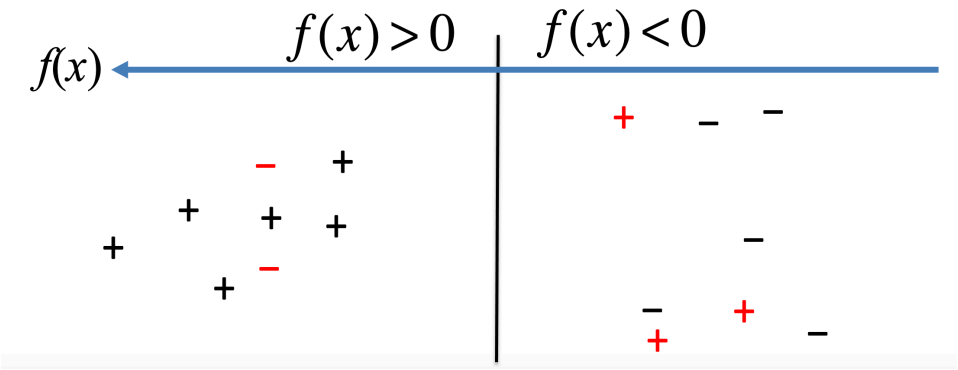
In the regularized learning expression, there are two terms: the loss term (which controls training performance) and the regularization term (which controls model simplicity). The loss $\ell(f(x_i), y_i)$ is often misclassification error $\mathbf{1}_{[y_i \neq \text{sign}(f(x_i))]} = \mathbf{1}_{[y_i f(x_i) \leq 0]}$ for classification problems. The regularization parameter C is a number that trades off between training loss and simplicity. I'll tell you later how to choose it, using cross-validation.

Note that minimizing $\sum_i \mathbf{1}_{[y_i f(x_i) \leq 0]}$ with respect to f is computationally hard. This is why we often minimize upper bounds for it that are convex and easier to minimize. Here are some of them. I have noted which common machine learning algorithms they are used in.

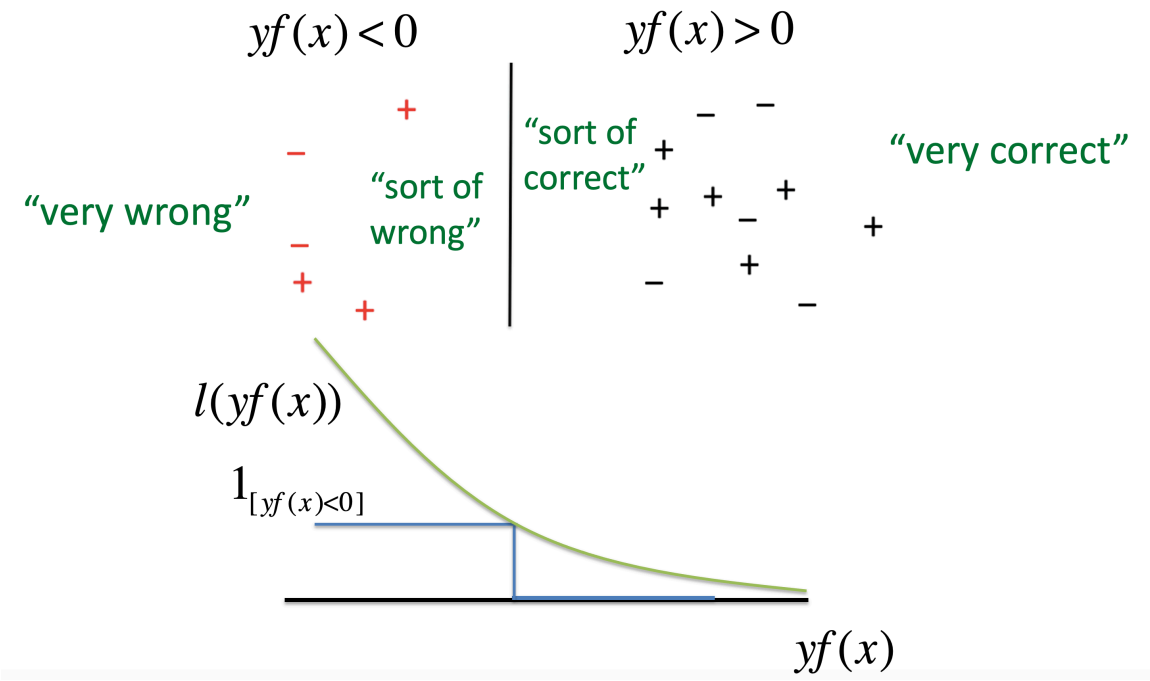
- "logistic loss" $\log\left(1 + e^{-y_i f(x_i)}\right) \Longleftarrow$ logistic regression, neural networks
- "hinge loss" $\max(0, 1 - y_i f(x_i)) \Longleftarrow$ support vector machines (SVM)
- "exponential loss" $e^{-y_i f(x_i)} \Longleftarrow$ AdaBoost



The margin is an important quantity in classification. A point with a margin of 0 means it is on the decision boundary. A large positive margin means that the point is correctly classified and far from the decision boundary. A negative margin means the point is misclassified. Let's show an example on this simple 2D dataset, where the function f increases to the left.



Now let's plot the margins. I'll flip the points over the decision boundary so that all the correctly classified points on the right and all the incorrectly classified points on the left. That way, the margin is now along the horizontal axis.



Intuitively, the margin tells us how well the point was classified. As we have shown, the major loss functions of machine learning are defined in terms of margins.

Let us define various options for the regularization term $R^{\text{reg}}(f)$. Usually f is linear, $f(x) = \sum_j \lambda_j x^{(j)}$ (where notation $x^{(j)}$ means the jth component of $\mathbf{x}$). There are a variety of choices for $R^{\text{reg}}(f)$ such as:

- $\|\boldsymbol{\lambda}\|_2^2 = \sum_j \lambda_j^2 \Longleftarrow$ used by ridge regression and SVM
- $\|\boldsymbol{\lambda}\|_1 = \sum_j |\lambda_j| \Longleftarrow$ used by LASSO, approximately used by AdaBoost
- $\|\boldsymbol{\lambda}\|_0 = \sum_j \mathbf{1}_{[\lambda_j \neq 0]} \Longleftarrow$ counts the number of nonzero terms.

The method called ridge regression is simply the squared loss with ℓ_2 regularization:

$$\frac{1}{n}\sum_i (y_i - f(x_i))^2 + C\|\boldsymbol{\lambda}\|_2^2.$$

Support vector machines use the hinge loss and the ℓ_2 norm:

$$\frac{1}{n}\sum_i \max(0, 1 - y_i f(x_i)) + C\|\boldsymbol{\lambda}\|_2^2.$$

AdaBoost uses the exponential loss without regularization, (although it acts as if it uses ℓ_1 regularization in some cases):

$$\frac{1}{n}\sum_i e^{-y_i f(x_i)}.$$

Regularized logistic regression uses the logistic loss with either the ℓ_1 or ℓ_2 norm.

It is also possible to use the ℓ_0 norm (well, it's actually a semi-norm), which is the count of non-zero terms in the model. It's more difficult to regularize this because it's not continuous in its arguments, but we will use it when we discuss modern optimization methods. As we'll see, using ℓ_0 has some major advantages if we can do it computationally. The major one is that C gains more intuitive meaning – it becomes the amount of loss we are willing to trade off for one fewer term in the model.

You might have noticed that the differences between these algorithms, as I have described them so far, is somewhat small. In fact, these algorithms all tend to perform similarly in practice on many datasets *if they use the same features*. The catch is that they use very different kinds of features, as you will see.

References:

V. N. Vapnik and A. Y. Chervonenkis. "On the uniform convergence of relative frequencies of events to their probabilities." Theory of Probability and its Applications, 16(2):264-280, 1971.