2 Induction: Machine Learning

Learning is one of the most important things that humans do. We're not hard-coded: we have to gather information about our world in order to survive in it. Humans use the term "learning" in at least three ways: learning by trial and error, learning by listening to teachers, learning by watching others perform tasks as examples. But we're going to be more specific[26] in what we mean by learning here:

Learning is developing something which helps us understand a feature of the world, in general, after only seeing a finite number of examples.

For example, we learn after multiple examples that cookies taste good and brussels sprouts taste bad; that school is held on weekdays; what the meaning of "no" is; that the sun rises in the east and sets in the west. We gather a finite number of samples and build from them a **model** which explains what we've seen.

Another aspect of learning is **generalizing** from examples. This is an incredibly important ability. For example, imagine that you were living in India 2000 years ago. You're walking down a path with your mother when suddenly an orange tiger jumps out and eats your mom. The next day, you're wandering down a path with your father when suddenly an albino tiger appears and eats your dad. From this what have we learned? A poor learner may have only learned that orange tigers eat moms and albino tigers eat dads. A good learner will have generalized this to the maxim that *tigers eat people*. Because it is so critical to evolutionary survival, generalization is hard-wired in our brains in a big way: we cannot help but generalize from examples, and even **overgeneralize** from them. This is because there's little downside, evolutionarily speaking, to overgeneralization: but there's a *huge* downside to poor generalization (getting eaten). Our powerful tendency to overgeneralize has also provided us with a host of social ills in the form of stereotyping.

In machine learning, both overgeneralization and undergeneralization are bad: but, as in the human case, perhaps undergeneralization is worse. Machine learner researchers use the term **overfitting** to refer to undergeneralization: and **underfitting** to refer to overgeneralization.[27]

Formally, the process of machine learning is building a **model** (or **hypothesis**) after observing a finite set S of some n **samples** (or **examples**[28]) drawn from the environment. What's a model? It could be anything: a simple function, a set of rules, a tree structure, an automaton, a collection of exemplars. Different techniques build different kinds of models.

Building a Model There are three common ways in which you may build this model, and each is useful for models of specific kinds.

- **Supervised Learning** A supervised learner receives a set of samples $s \in S$ of the form $s = \langle i, o \rangle$. From this, it learns a model M which describes a function mapping the space of all possible i to their corresponding o, that is, $M(i) \rightarrow o$. For example: we might map days to temperatures: various days in the summer are *hot*, while various days in the winter are *cold*. Or we might map how much I lean a certain amount on a bicycle and turn the wheel , to the degree the bicycle will turn. We might call the space of i the *input* and the space of o the *output* of the model. There are two major kinds of supervised learning: **classification** and **regression**, which differ based on the nature of the output.

[26] Well, in fact this definition is highly general: it applies to a wide range of learning tasks humans do.

[27] These terms come straight from statistics, which has always been the evil twin of machine learning in AI.

[28] Or sometimes *exemplars.*

- **Unsupervised Learning** An unsupervised learner receives a set of samples $s \in S$ simply of the form $s = \langle i \rangle$: there's no o. From this it learns a model M which says something interesting about the relationships among the various i in the space, that is $M(i, i') \rightarrow r$. The most common use of unsupervised learning is **clustering**, where we identify groups in the space based on their similarity to one another.[29] For example, we might learn that people are more similar to other people than they are to, say, telephones (though we don't know beforehand which points are people and which are telephones).

- **Reinforcement Learning** A reinforcement learner builds a model which typically maps situations (or *states*) s to actions a that one should perform in those situations in order to optimize one's lot in life.[30] Such a model is known as **policy**, and is usually written as $\pi(s) \rightarrow a$. Unlike supervised learning, reinforcement learners build this model not from a set of example data points and their descriptions, but by trying various actions in different situations and receiving only **feedback** or **reinforcement** (reward or punishment) as a result.

Some of the methods of these types are deterministic, but quite a number of them are optimization techniques: you're trying to find a model M which is as close to the correct description of the real world as possible. This is essentially optimizing (in this case, minimizing) the **expected error** of the model: how much its approximation differs from reality. Often the error space is filled with hills and valleys rather than one large hill to climb: and so the method may get stuck in a **local optimum** far from the true **global optimum**, producing a sub-par model.

The inputs can take all sorts of forms: they could be arbitrary objects, or graph structures, or trees, or streams of data. The **representation** this data takes will determine which algorithms you can use: there are algorithms for streams of data, for Bayes Networks, and so on. But among the most common representation of your inputs is as points (vectors) in a d-dimensional space of real-valued numbers, or **features**.

In supervised and reinforcement learning the models map inputs to outputs. While the inputs can take any form, the outputs tend to be restricted to certain representations. For example, in classification (and typically in reinforcement learning) the outputs take the form of unordered **labels**. In regression, the outputs take the form of points in an m-dimensional real-valued space. Thus the representation of the outputs determine not just what kind of algorithm you'd use, but what type of problem you're dealing with.

Using a Model Once built, different models have different capabilities. Some models are **discriminative** or **predictive**: you can use them to predict some fact about a previously unseen sample from the environment. For example, imagine that you've built a model based on samples consisting of successful and unsuccessful restaurants. A discriminative model would, given a previously unseen restaurant, predict its success or failure. Other models are **generative**: you can use them to create new (typically random) samples with a desired characteristic. For example, you could use a generative model to invent new restaurants likely to be successful. There also exist models which are both discriminative and generative.

[29] Another related unsupervised learning application is **density estimation**, which guesses the underlying probability distribution which produced the samples.

[30] Formally this mapping is basically the same as in supervised learning: but how the model is developed (via reinforcement) is different.

You can always produce a generative model from a discriminative one, though it might not be cheap. A straightforward way to do this is through *rejection sampling:* for example, creating random restaurants until the model predicts that one will be successful.

Algorithm 1 *Simple Rejection Sampling*

$n \leftarrow$ desired number of generated samples
$M \leftarrow$ learned discriminative model
$c \leftarrow$ characteristic you'd like your generated samples to have

$X \leftarrow \{\}$
for n times **do**
    repeat
        $x \leftarrow$ sample generated uniformly at random
    until M predicts that x has characteristic c
    $X \leftarrow X \cup \{x\}$
return X

Why could this be expensive? Perhaps successful restaurants are very rare: you might be generating a *very* large number of samples before you find one which meets the desired criteria.

Testing a Model A primary goal of a machine learning algorithm is to produce a model from a finite set of samples which **generalizes** to the real environment as closely as possible. How do you test whether your model is any good? Let's say that your algorithm is producing discriminative models. To test generalizability, you need to test the model on unseen data.

Here's one simple approach: draw a set of random samples from the environment. Divide this set into two disjoint subsets: the **training set** and the **testing set**. Build the model by feeding your algorithm the samples in the training set. Then apply the resulting model to the testing set to see how often it correctly predicts them.

Another approach is called **k-fold validation**. Draw a set of random samples from the environment. Divide this set up into k disjoint subsets $S_1, ..., S_k$. Then for i from 1 to k do the following: use the algorithm to build a new model using the samples from all the subsets *except* S_i as the training set. Then test the generalization of the resulting model using S_i as the testing set. The generalization of the algorithm is the average performance of all k of the models.

Probability and Samples Modern machine learning theory is largely viewed in terms of probability and building distributions. For example, in supervised learning, M may be viewed an approximation of the true distribution $P(c|x)$, namely, the probability that a given point x will have a given characteristic c in the environment.

This worldview makes sense: after all, induction is essentially about *intelligent guessing*. You don't have all the facts at your disposal, just a limited number of them, so there's a certain probability that you'll be wrong. Second, in many situations, there is **noise** in your samples. Some portion of your samples are simply wrong. Probabilistic models can accommodate for this. Even if your problem by its nature has no noise, you could still cast it in a probabilistic fashion: you can say that the probability of facts being wrong is 0.0.

The issue of number of samples is a big one. Consider if your samples are represented as d-dimensional points (as is often the case). The basic problem is the dimensionality of the space of points over which you are fitting your model. This dimensionality could be very high: for example, you might be trying to cluster people according to thousands of genes they may have in common. The reason this causes a problem is that if your space is high dimensional, you need a lot of samples to fill the space well enough to understand it. Lacking these samples, your model is unlikely to match reality. [31] Unfortunately, large numbers of samples may be difficult to come by. This conundrum is so common and problematic, it has its own magic term in the machine learning field: the **curse of dimensionality.**

Algorithms We'll see various algorithms for clustering, classification, and regression. But we'll start with two well-regarded but simple ones to demonstrate the underlying notions of supervised and unsupervised learning: **K-Nearest Neighbor** classification and **K-Means** clustering.[32]

2.1 Unsupervised Learning

An unsupervised learning algorithm builds a model M which says whether two objects x and y are *similar* or not: or the degree to which they are similar. Put more formally, M might be viewed as a function $M(x, y) \rightarrow a > 0$, where a is a measure of similarity (lower is more similar). The unsupervised learner builds this model by experiencing samples of the form $x_1, ..., x_n$, each of which holds a single point drawn from the space.

The most common use of unsupervised learning is **clustering**, where we wish to identify groups in the environment based on their similarity to one another. For example, our algorithm may learn that people are more similar to other people than they are to, say, telephones.

The clustering task is a so-called **ill-posed problem**. The issue is: what really constitutes a cluster? Let's say our sample consisted of boys, girls, male penguins, and female penguins. We ask the learner to find two clusters. Should it cluster them according to humans versus birds, or according to males versus females? Both are perfectly reasonable clusters. Furthermore, what if the learner also had to figure out the "optimal" number of clusters? Is four clusters better than two in this case? Thus clustering algorithms must muddle about finding clusters which seem reasonable, even if other clusters are reasonable as well (and might be what we had really been looking for).

2.1.1 K-Means

One of the more famous — and simple — clustering algorithms is the K-Means clustering algorithm, which seeks to find k means of clusters in data. Each sample in the data takes the form of a point in a d-dimensional space. You provide the n samples[33] and the desired number of clusters.

The algorithm description below goes into unnecessary and pedantic detail. In fact, K-Means can be described quite simply. We begin with a collection of n samples X and a collection of k means M. The means are set to random values from X, such that each mean is distinct.[34] Then we repeatedly do the following two items:

[31] Often your model will underfit the data.

[32] Recall "k-fold validation". Don't ask me what's up with machine learning and using "k" for everything.

[33] Obviously n should be much larger than k.

[34] There are plenty of other ways of initializing the means, but this is the most common way.

1. Color each sample $\vec{x} \in X$ according to the mean $\vec{m} \in M$ which it is nearest to.
2. Move each mean $\vec{m} \in M$ dead center in the middle of the samples in X colored by it. If there were no samples colored by a given mean (an unfortunate situation), don't move it.

We stop when the labels of the samples don't change any more (and so likewise the means stop moving about).

The algorithm is an optimization method and can get caught in local optima of different kinds. For example, one mean may have grabbed what are obviously multiple clusters and called them for its own, forcing the other means to divvy up the remaining samples. Or a mean may be left out in the cold entirely, unable to label any samples at all. These suboptima are shown at left.

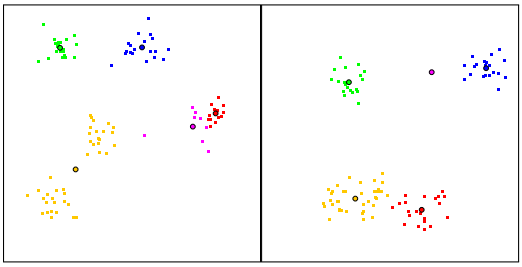

Figure 11 Two examples of convergence to local optima.

Algorithm 2 *K-Means*

To build the model
- $k \leftarrow$ desired number of means of clusters to find
- $X \leftarrow \{\vec{x}^{(1)}, ..., \vec{x}^{(n)}\}$ samples, each of them a vector of dimensionality d
- $L \leftarrow \{l_1, ..., l_n\}$ colorings, one per sample, each an integer initially set to 1
- $M \leftarrow \{\vec{m}^{(1)}, ..., \vec{m}^{(k)}\}$ means, each of them a vector of dimensionality d
- **for** j from 1 to k **do** ▷ Initialize values of M to distinct random samples from X
  - $\vec{m}^{(j)} \leftarrow$ random sample from X not yet selected for another $\vec{m}$
- **repeat**
  - **for** i from 1 to n **do** ▷ Color each sample by the mean which is presently nearest to it
    - $j \leftarrow$ the index of the mean $\vec{m}^{(j)} \in M$ nearest to the sample $\vec{x}^{(i)}$
    - $l_i \leftarrow j$
  - **for** j from 1 to k **do** ▷ Move each mean to the center of the samples which were colored by it
    - $X' \subseteq X \leftarrow$ all samples $\vec{x}^{(i)} \in X$ such that $l_i = j$
    - **if** $||X'|| = 0$ **then** ▷ It's possible that a mean colored no samples
      - HandleZeroSamples(X', X, L, M, j)
    - **else**
      - $\vec{m}^{(j)} \leftarrow \sum_{\vec{x}^{(i)} \in X'} \frac{\vec{x}^{(i)}}{||X'||}$
  - **until** L is no longer changing
  - **return** M

To query the model
- $M \leftarrow \{\vec{m}^{(1)}, ..., \vec{m}^{(k)}\}$ means, each of them a vector of dimensionality d
- $\vec{x}' \leftarrow$ as of-yet unseen sample to query, of dimensionality d
- **return** the mean $\vec{m}^{(i)} \in M$ closest to $\vec{x}'$

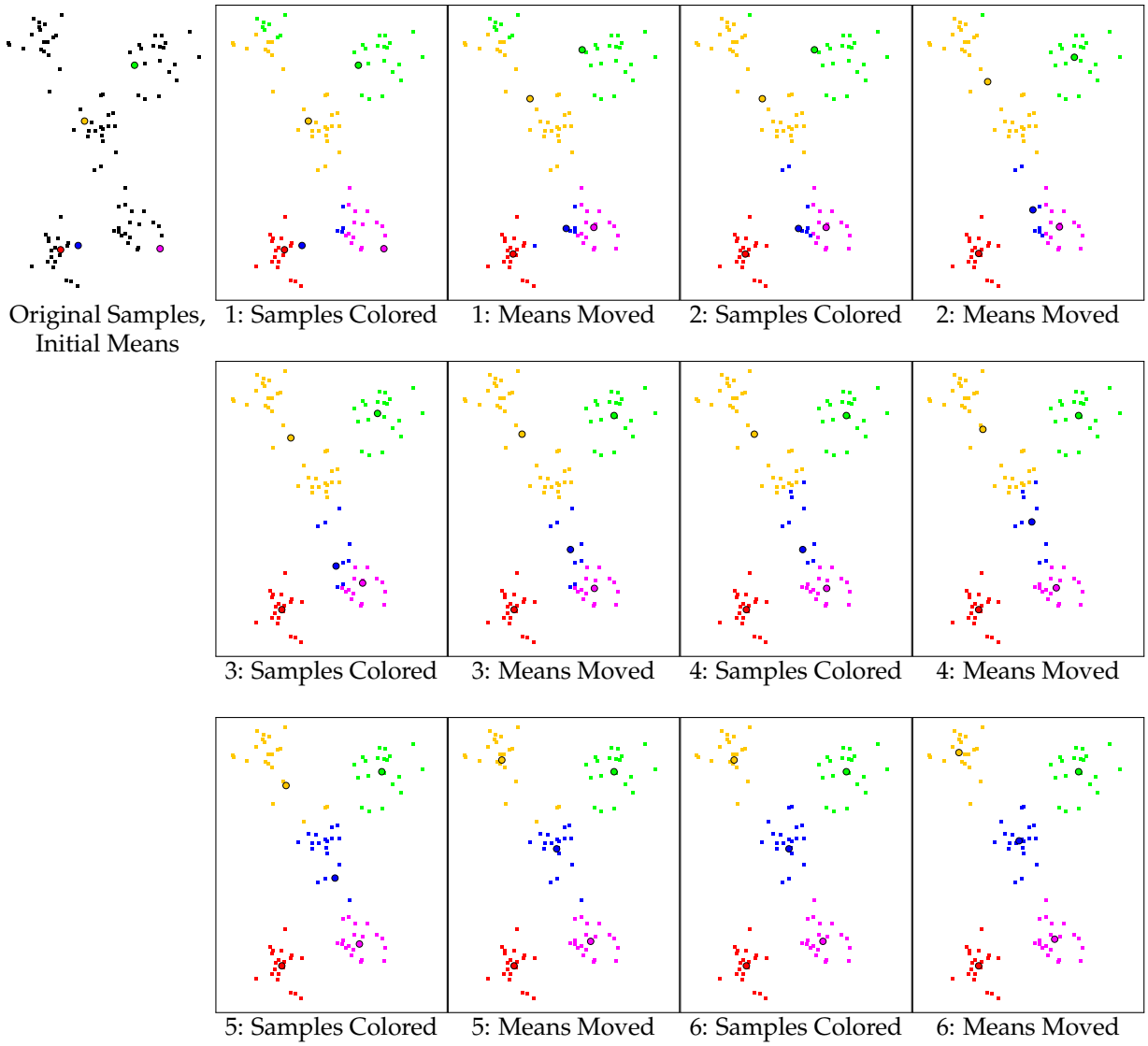

Figure 12 The K-Means algorithm, finding five means of clusters in the sample data. The process begins with five means placed at the location of five randomly-chosen samples. Each iteration the samples are first colored according to the closest mean; then the means are moved to the center of their colored samples. The process terminates after the colored samples don't change any more (in this case, six iterations).

This very simple algorithm terminates with means located at the centers of clusters of data in our sample (and its termination is guaranteed). Essentially the means jockey for supremacy as they are pulled into position by nearby clusters. Notice the undefined HandleZeroSamples(...) function. This is your chance to deal with local optima, where a mean has been pushed out of the game entirely: it's been deprived of any samples to call its own. If K-Means terminates in a situation like this, that cluster mean will be nonsensical and worthless. The default implementation of HandleZeroSamples(...) is to do nothing at all. But you could do smarter things. For example, you could move the mean into the region of another mean, perhaps the one presently with the largest-variance cluster, and split the cluster up amongst the two of them.

The two-part nature of this algorithm isn't a coincidence: K-Means is a simple example of a general unsupervised learning algorithm framework called **Expectation Maximization** or **EM**. EM can be described in two parts:

1. Label the data according to the current predictions of the model.
2. Modify the model so it matches the distributions of the labels.

Basically, the model influences the labels of the data, which then in turn influence the model. This back-and-forth continues until an optimum is reached where both are at an impasse, neither changing the other. Like K-Means, EM in general can get stuck in local optima.

2.2 Supervised Machine Learning

A supervised learner is more in-line with what you'd imagine a "learner" would do: learn to relate objects with one another. For example: when the road looks like *this*, you should turn the steering wheel like *this*. When you see the following handwritten symbol, it means an "A". When the game board looks like *this*, it means that I am losing badly. Various days in the summer are *hot*, while various days in the winter are *cold*. Or when I lean a certain amount on a bicycle, and turn the wheel a certain amount, then the bicycle will turn by *this much*.

In short, a supervised learner builds a model which maps inputs i to outputs o[35] Inputs can be anything: trees, graphs, rulesets, whatever you like: but most commonly they take the form of points $\vec{x}$ in a multidimensional space of **features**. The nature of the outputs, on the other hand, determines the kind of problem.

In a classification problem, the various o are **labels** l chosen from a finite set of possible labels. There's no ordering among these labels. For example, handwritten symbols on envelopes can only be one of a finite number of things: 26 letters, 10 digits, and a few punctuation marks: perhaps this totals to roughly 40 possible labels. Usually congressmen in the United States are either Democrats, Republicans, Democratic Socialist,[36] or Independents (3 labels). Many classification problems have only two classes: true and false, or red and purple, or Sharks and Jets. These kinds of problems are called **binary classification problems** and there are a variety of algorithms designed just for them.

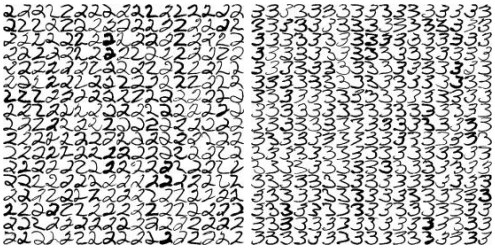

Figure 13 A bunch of handwritten 2's and 3's from the USPS handwritten digit database.

In contrast, in a regression problem, o takes the form of a point in an m-dimensional metric space. Recall that the space of inputs doesn't have to be metric, but it's *often* metric (hence why we're saying $\vec{x}$). For example: let's say you have a trebuchet: a medieval siege device used to catapult projectiles at castle walls. If you launch an anvil of a given mass from the trebuchet, and the trebuchet swing arm is a certain length, and the trebuchet is aimed in

Figure 14 A trebuchet.

[35] "Inputs" and "outputs" are my terms. Common terms in machine learning for inputs: *data* or *independent variables*. And for outputs: the **value** to be predicted; or the **dependent variable**.

[36] Bernie Sanders of Vermont.

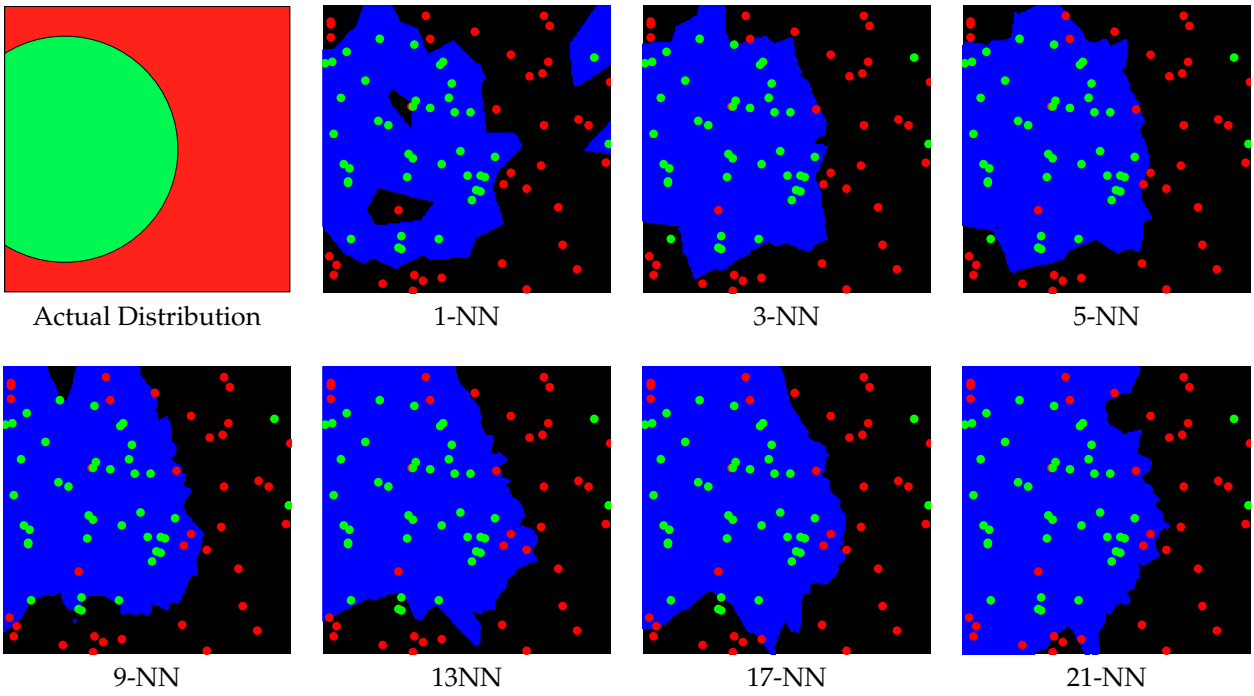

Figure 15 The K-Nearest Neighbor algorithm for various values of K. The blue region indicates the area that the model believes should be labelled green; likewise the black region indicates the area that the model believes should be labelled red. Note that in this example the 1-NN method fails to weed out noisy samples (overfitting), while increasing values of K tend more and more to encroach on valid red regions due to being outvoted by coteries of green samples. 3-NN seems about right on this problem.

a certain direction, then the anvil will likely land at a certain coordinate location and impact at a certain angle. Learning how your trebuchet work might be life-or-death stuff in the Middle Ages. In this case, the input $\vec{x}$ is a point in three-dimensional space (mass, length, aim), and the output is also a point in a three-dimensional space (x-coordinate, y-coordinate, impact angle) (we could call it $\vec{c}$). You'd like a machine learner to develop an understanding of this mapping function based on the results of a number of trebuchet launches.

In truth, for many regression problems, the dimensionality of $\vec{c}$ is 1. For example: given a current view of the road from your camera, you'd like the car steering wheel to be steered at *this angle*. Or given a situation in a checkers game, Black is ahead (likely to win) by *this amount*. Or given the current profit/loss statement of a company, its probability of failure is *this percent*.

2.2.1 K-Nearest Neighbor

This algorithm[37] (often referred to as **K-NN**) is a very simple, yet often highly effective, classification algorithm. Though it can certainly be used for multiple labels, it is most commonly used for two, that is, binary classification. K-NN is particularly simple because it doesn't *really* build up a model.[38] Instead it simply uses the original samples as the model. K-NN is mostly used for discriminative tasks.

[37] No, not *every* machine learning algorithm begins with "K".

[38] Thus in machine learning parlance, it's called a **lazy learner**.

K-NN works as follows. We first store all the samples $\langle \vec{x}, l \rangle$ (points and labels) provided us during the learning process. Then when queried about the label of a new, until-now unseen point $\vec{x}'$, we find the closest points among our samples. These points then vote based on their label values. The winning label is our prediction for the new point. If $k = 1$, then this is simply labeling the new point with the label of the nearest sample to it.[39]

The algorithm is pretty simple:

Algorithm 3 *K-Nearest Neighbor*

To build the model
  $k \leftarrow$ desired number of votes ▷ For binary classification use an odd number
  $S \leftarrow \{s_1, ..., \}$ the various labels used ▷ For binary classification, 2 ▷ An odd number
  $X \leftarrow \{\langle \vec{x}^{(1)}, l_1 \rangle, ..., \langle \vec{x}^{(n)}, l_n \rangle\}$ samples, each with a d-dimensional vector and a label
  $M \leftarrow \{k, S, X\}$ ▷ Seriously. That's all there is to building the model.
  return M

To query the model
  $\vec{x}' \leftarrow$ as of-yet unseen sample to query, of dimensionality d
  $M \leftarrow \{k, S, X : \langle \vec{x}^{(1)}, l_1 \rangle, ..., \langle \vec{x}^{(n)}, l_n \rangle\}\}$ model
  $C \leftarrow \{c_1, ..., c_m\}$ label counts, all initially all zero
  Sort samples in X by distance of $\vec{x}^{(...)}$ to $\vec{x}'$ (smaller distances come first)
  for i from 1 to k **do** ▷ Only consider the closest k samples
    $c_{l_i} \leftarrow c_{l_i} + 1$
  $j \leftarrow$ the index of the highest label count $c_j \in C$ ▷ You will have to break ties somehow
  return s_j

This can of course be used with arbitrary numbers of labels, but then you'll need to figure out what to do when there's no majority winner among the votes, and also how to break ties. If you're doing just binary classification, and if you set k to an odd number, you'll of course never have ties.

Figure 15 shows K-NN in action. Given a sample of points drawn from the environment distributed as shown, we can see the regions where K-NN would label new points as (in this example) "red" or "green". It's important to note the result with $k = 1$ (that is, "1-NN"). Here, every single point gets to dominate some region around it. This means that if you have noise in your model — which is almost always the case — the noisy samples will create incorrect regions in the model. But if you increase k, these samples get outvoted by correct samples nearby and those regions disappear. However if you increase k by *too much*, then this outvoting process starts incorrectly encroaching on areas where there aren't many samples, and nuances along the border between the red and green regions get muddied. Either way, you're producing a model with more error. You're looking for the setting of k which achieves the best balance, and thus the minimum error.

Note that when querying the model, Algorithm 3 sorts the entire set of samples just to get the first k. This is $O(n \lg n)$, and obviously you can do better than this in two ways: either by organizing

[39] This divvies up the space in the fashion of a **Voronoi Diagram**.

the samples in a **quad tree** or other spatial data structure better suited for determining the closest point; or by using a smarter algorithm to identify the smallest k elements without sorting everyone else as well.

Small values of k, such as $k = 3$, are common. But the optimal value of k varies with the data. As a rule of thumb, the best setting for k grows with the number of sample points used.

Unlike K-Means, K-Nearest Neighbor is *not* an optimization algorithm: it is deterministic. The model is directly derived from the samples without any iterative process.