Machine Learning for Intelligent Systems

Personal notes from Cornell CS 4780/5780 (Spring 2017)

Himanshu Dongre

Based on lectures by Kilian Q. Weinberger

These are personal study notes written while following Kilian Q. Weinberger's Cornell CS 4780 lecture series. They are not official Cornell course materials and are not endorsed by Cornell University or the instructor. Quotations are attributed to the lecturer; the organization, annotations, and any errors are my own.

The notes follow the Spring 2017 CS 4780/5780 offering, whose official course page calls the course *Machine Learning*. The accompanying YouTube playlist is titled *CORNELL CS4780 "Machine Learning for Intelligent Systems"*. The public playlist metadata available during this revision did not establish its uploader/owner, so no claim of official playlist ownership is made. Sources: official Spring 2017 course page and YouTube playlist.

The mathematical organization and many pedagogical cues follow Weinberger's public lectures. Short quotations in quotation marks were transcribed from those lectures and remain the words of Kilian Q. Weinberger. They are used for attribution and commentary; they are not covered by the license below. The course page states copyright, and no Creative Commons license was established for the course materials or playlist videos. Accordingly, no Cornell or Weinberger material is relicensed here.

The source repository also contains imported image files whose provenance and reuse permission could not be established. Those imported files are deliberately not embedded in this public-release PDF and are not covered by its license. The figures embedded in this PDF were generated originally for these notes.

Contents

Prologue

Machine learning (ML) deals with *algorithms* that *improve* on some *task* with *experience*. We give a computer data and the expected output so that it outputs a program that takes new data as input to give unseen output. More representative data can improve the program generated, although more data alone does not guarantee a better model, which is why it's called *learning*. This not only seems like the computer programming itself but the program is such that we typically wouldn't have been able to write it by hand.

Machine learning as a field emerged from the study of artificial intelligence. The perceptron was an early and highly influential learning algorithm which created enormous excitement in ML and AI. Minsky and Papert later emphasized limitations of single-layer perceptrons, including their inability to represent XOR; this was one factor in a broader decline in enthusiasm and funding often associated with the AI winter, not its sole direct cause. As a result, some new approaches to AI were rebranded as machine learning. The difference between AI and ML is one of approach. AI takes a top down, human-first approach where the goal is to replicate human capabilities. ML is more practical with smaller goals and resembles a bottom up, computer-first approach. Unlike traditional computer science which is entrenched in logic, machine learning is based on statistics and optimization. "Now let's get crack'n."

1 Supervised Learning Setup

Supervised learning aims to make predictions from data. Setup:

- We collect data $\mathcal{D} = \{(\vec{x_1}, y_1), \ldots, (\vec{x_n}, y_n)\} \subseteq \mathcal{X} \times \mathcal{Y}$. The data is sampled from some distribution $(\vec{x_i}, y_i) \sim \mathcal{P}$, which we of course don't know and can't write down.
- We assume that the data is **iid (independent and identically distributed)**.
- $\vec{x_i} \in \mathcal{X} \subseteq \mathbb{R}^d$ (could be a subspace). Examples of $\mathcal{X}$:
  - Patient data: $\vec{x} = \begin{bmatrix} \vdots \end{bmatrix}$
  - Text documents: BoW representation (word order is overrated, only look at which words are in the text). The vector is of dimension equal to the number of words in the English language and each element of $\vec{x_i}$ is the count of the corresponding word in the document.
  - Image: $\vec{x_i}$ has the RGB values of the first pixel as the first three values, those of the second pixel as the next three values, and so on.
- $y_i \in \mathcal{Y}$. Examples of $\mathcal{Y}$:
  - Binary classification: $\mathcal{Y} = \{0, 1\}$, $\mathcal{Y} = \{1, -1\}$
  - Multiclass classification: $\mathcal{Y} = \{1, \ldots, k\}$
  - Regression: $\mathcal{Y} = \mathbb{R}$

$\star$ A **dense** representation means almost all dimensions will be filled. A **sparse** representation means that most dimensions will be 0.

We define a set of possible functions that we could learn from $\mathcal{D}$ and call it the **hypothesis class**, $\mathcal{H}$. The job of the ML algorithm is to find the best function $h \in \mathcal{H}$ that fits the data. Examples:

- Decision trees.
- Linear classifiers.
- Artificial neural networks.
- Support vector machines.

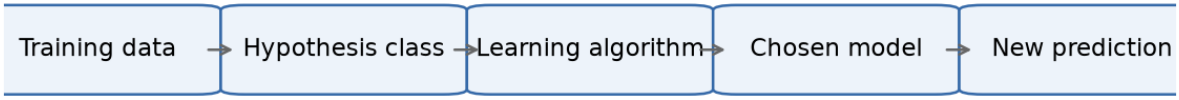

Figure 1: Machine-learning pipeline.

⋆ There is no best algorithm. It depends on the problem and the data which one is the right algorithm.

Terrible Algorithm #1. Pick $h \in \mathcal{H}$ randomly. This could work if $\mathcal{H}$ is restricted enough that all options are terrible.

Terrible Algorithm #2. Try out every single $h \in \mathcal{H}$ and pick the best.

1.1 Loss Functions

A loss function evaluates how well any $h \in \mathcal{H}$ does. A lower loss is better. A loss of 0 implies no more mistakes.

- **0/1-Loss:** $\ell_{0/1}(h; \mathcal{D}) = \frac{1}{n} \sum_{(\vec{x_i}, y_i) \in \mathcal{D}} \delta[h(\vec{x_i}) \neq y_i]$.
- **Squared loss:** $\ell_{\text{sq}}(h; \mathcal{D}) = \frac{1}{n} \sum_{(\vec{x_i}, y_i) \in \mathcal{D}} (h(\vec{x_i}) - y_i)^2$.
- **Absolute loss:** $\ell_{\text{abs}}(h; \mathcal{D}) = \frac{1}{n} \sum_{(\vec{x_i}, y_i) \in \mathcal{D}} |h(\vec{x_i}) - y_i|$.

1.2 Learning and Generalization

⋆ **Learning** is the process $\mathcal{H} \xrightarrow{\text{choose}} h \in \mathcal{H}$.

Terrible Algorithm #3. $h(\vec{x}) = \begin{cases} y_i & \text{if } \vec{x} = \vec{x_i} \text{ for } (\vec{x_i}, y_i) \in \mathcal{D} \\ y_1 & \text{otherwise} \end{cases}$.

We want the classifier $h \in \mathcal{H}$ st $\forall\, (\vec{x}, y) \sim \mathcal{P},\ h(\vec{x}) = y$; i.e. we want to learn an $h \in \mathcal{H}$ that *generalizes*. The **generalization loss** is the expected loss of a random sample from $\mathcal{P}$:

$$\mathbb{E}_{(\vec{x},y)\sim\mathcal{P}}[\ell(h; (\vec{x}, y))]$$

We can estimate the generalization loss by splitting $\mathcal{D}$ into a **training set** and a **test set**. We learn h from the training set and estimate its generalization loss as the loss from running h on

the test set. This is a fair estimate since the test set here is sampled from $\mathcal{P}$ just like the training set. The generalization loss is truly estimated from the test set only when it is used **exactly once**. Running multiple models on the test set to determine which one is the best is wrong because now we're overfitting to the test set. Hence, we make a third split of $\mathcal{D}$ called the **validation set** that is used to make decisions like choosing the best model out of multiple or determining the depth of a decision tree, but the final error reported is on the test set after running on it **once and only once**. The validation and test sets should be "not too large, not too small" to balance the trade-off of estimating the error correctly and underfitting. If the data has a temporal component, a rule of thumb is to split it by time. Otherwise, it should be randomly shuffled and split for a proper split.

> "There is no one algorithm that does well at everything. It cannot exist."

The point is that learning requires assumptions connecting the training distribution to future data; different algorithms make different assumptions, so none is best for every possible problem.

2 Bayes Optimal Classifier

In reality, we never have access to the *elusive* distribution, $\mathcal{P}$, from which our data originates. But if we did, we could define a classifier:

$$h^*(\vec{x}) = \arg\max_{y \in \mathcal{Y}} \ \mathcal{P}(y \mid \vec{x}) \, .$$

This is called the **Bayes optimal classifier**. Under the true distribution, for 0/1 loss and measurable classifiers, no other classifier has lower expected classification risk (ties may yield several optimal classifiers).

Proof. Admitted. □

3 k-Nearest Neighbors

KNN is one of the oldest machine learning algorithms. It makes the assumption that *data points that are similar have similar labels.* Formalism:

- We have a test point $\vec{x}$ and a dataset $\mathcal{D}$.
- We define a dataset $S_x \subset \mathcal{D}$ st
  - $|S_x| = k$.
  - $\forall \, (\vec{x'}, y') \in \mathcal{D} \setminus S_x, \ d(\vec{x}, \vec{x'}) \geq \max_{(\vec{x''}, y'') \in S_x} d(\vec{x}, \vec{x''}) \, .$

The key behind k-nearest neighbors is to use a good distance metric; i.e. the distance metric should actually reflect the KNN assumption that similar points have similar labels. If we have a bad metric such that it measures similarity that isn't compatible/correlated with the label, KNN will break. Intuitively, if two points are close together in the feature space then it is likely that they will have similar labels. We use the Minkowski distance, or the ℓ_p-norm:

$$d(\vec{x}, \vec{z}) = \left(\sum_{r=1}^{d} |x_r - z_r|^p \right)^{\frac{1}{p}} \, .$$

- $p = 1 \equiv$ Manhattan distance.
- $p = 2 \equiv$ Euclidean distance.
- $p \to \infty \equiv \max_r$.

Theorem. For binary classification with iid training data in a suitable metric space (so nearest neighbors converge to the query point), the asymptotic expected 1NN error satisfies $R_{1\text{NN}} \leq 2R^*(1 - R^*) \leq 2R^*$, where R^* is the Bayes 0/1 risk.

Proof. Admitted. □

Curse of dimensionality. If we look at n data points uniformly drawn from a **unit** hypercube in $\mathbb{R}^d$ and pick one data point $\vec{x}$. How big should a little hypercube (of side ℓ) around $\vec{x}$ be so that it contains the k-nearest neighbors of $\vec{x}$?

- The volume of the little hypercube is $\ell^d \approx \frac{k}{n}$ because of the uniform distribution $\Rightarrow \ell = \left(\frac{k}{n}\right)^{\frac{1}{d}}$.
- Setting $k = 10$ and $n = 1000$, $\ell = \left(\frac{1}{100}\right)^{\frac{1}{d}}$. Here comes the shocker:

d	ℓ
2	0.1
10	0.63
100	0.955
1000	0.9954

Basically, in 1000D space, the box that contains the 10 nearest neighbors around $\vec{x}$ is almost as large as the box from which the data was sampled. The other $n - k = 1000 - 10 = 990$ points lie in the super narrow margin squeezed between the two boxes. This is problematic because the k-nearest neighbors are on the edge of the interior box, and the other points are in the narrow margin—right next to them. Hence, there is no notion of closeness and *all* the points are roughly at the edges. They are all so far away from each other. Hence the KNN assumption that nearby points have the same label **is nonsense** because *there is nothing nearby.* The labels change faster than the sample density. This is the curse of dimensionality. Everything is really far away from everything and there is no notion of similarity anymore. To put it another way, we would need an *exponential* amount of data to obtain a local similarity. In high dimensional spaces, with no assumptions about the data, KNN **will not work**. But if we know that the ambient (universal) space is high-dimensional, but the data lies only in a **low-dimensional subspace** or a **low-dimensional submanifold**, then KNN can work. Hence, another assumption of KNN is that we have a **low intrinsic dimensionality**. This is why KNN typically works well with handwritten digit classification, image classification, etc. It's a good preprocessing step to reduce the dimensionality (like PCA, etc.). It should be one of the first tools in the toolset.

$\star$ The brute-force inference time complexity of KNN is $O(nd)$ per query—linear in n. So although KNN can do very well as n becomes very large, brute-force inference becomes very slow.

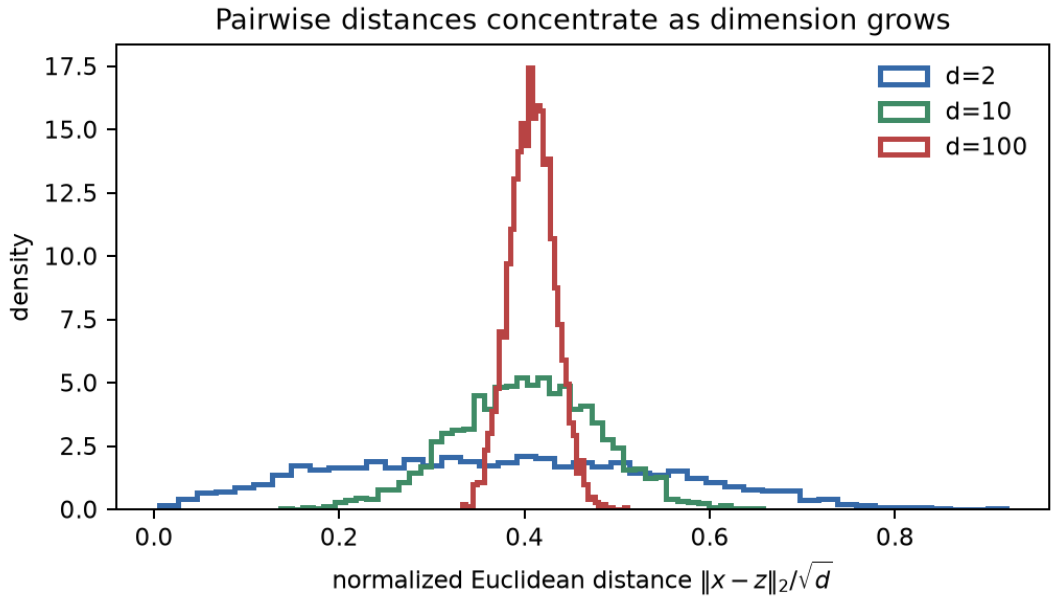

Figure 2: Visualization of distance concentration as dimension grows.

3.1 kD Trees

KNN is slow because for every test point, we have to look through all training points to find the k nearest neighbors. We can speed up this search by repeatedly partitioning the training data in half to build a data structure which we can use during inference to compute the k nearest neighbors of a test point. The data structure turns out to be a tree. The precise steps to build it are:

1. Pick one feature (doesn't matter which but prefer the one with the largest variance as a heuristic), compute the median value for that feature across the training data, and partition the dataset along it.

2. Let the picked feature be x_{r_1}. Initialize a root node that asks the question whether $x_{r_1} > t_1$, where t_1 is the median value of x_{r_1}, and two edges emanate out of the node corresponding to the answers "yes" and "no."

3. At each of the two children nodes, recursively run the algorithm again from step 1, but this time with the chosen feature being some $x_{r_2} \neq x_{r_1}$. Note that the chosen feature is the same across all children nodes $\Rightarrow$ the next threshold, t_2, for splitting is the same at all children nodes.

We continue this until we've exhausted all features x_1 through x_d. The resulting data structure is a perfect binary tree with height $\log n$ such that all training data resides in the leaves; i.e. each leaf is a *bin* that contains one or more training data points and every data point can only be in exactly one leaf. All internal nodes are decision (yes or no) nodes. During inference, we parse the tree from the root to a leaf using the feature values in the test point and run the nearest neighbors search on the data points in that leaf. Then we back up to the parent to continue our search. Our next destination to search is the sibling of the first leaf (note that every node has exactly one sibling since every parent has exactly two children) in a **depth-first** fashion. A full depth-first search would be equivalent to searching through all the data points, which is the worst-case of the algorithm. However, if we have looked at at least k points so far, we can *prune* our search. We are at a parent

that asks the question whether some feature value is greater than its median ($x_i > t_i$?) and we have searched one branch of it. Let the farthest neighbor we've found so far be $\vec{x_f}$. **If the distance between the test point $\vec{x}$ and the hyperplane $x_i = t_i$ is greater than the distance between $\vec{x}$ and $\vec{x_f}$, then we can simply skip searching the sibling, back up another step and carry out the same pruning step**. With this, we can skip entire subtrees if needed hence speeding up the nearest neighbors search. The other case is that the first leaf had less than k points in which case we have no option but to search its sibling. Once we hit the k points threshold then we can start looking for opportunities to prune the search on every back up.

This data structure we came up with is called a **kD tree**. Turns out that the curse of dimensionality ruins things with kD trees as well. When d is high, it is very unlikely for a test point to find itself far from *all* the threshold hyperplanes $\Rightarrow$ the search gets pruned rarely $\Rightarrow$ we typically end up with the worst case performance. kD trees often lose their advantage as dimension grows, although no universal cutoff such as $d \geq 10$ applies.

3.2 Ball Trees

If d is large but the data is intrinsically low dimensional then **ball trees** are a helpful data structure. They are similar to kD trees but they relax the constraint that we split the dataset only along axes (kD trees split only along axes). This constraint backfires in high dimensional spaces. Ball trees, on the other hand, build big balls (`XD`) to partition the space. The idea is that we find *a* general direction in the space along which the data is spread (this direction may not be axis-aligned). We do this *fast* by picking a random point, then finding the point that's the farthest away from it, then finding the point that's the farthest away from the second point. We finally pick the second and third points. This heuristic is great for picking two points *fast* that are *really* far away. The points being really far away means that the dataset has a wide spread along the direction of the line that connects the two points. We compute that direction (*really fast*), project all data points onto that line (worth one dot product each $\Rightarrow$ *really fast*) and split along the median. Hence, we have a partition of the space: points whose projection is less than the median vs. points whose projection is greater. We somehow need to capture the separate partitions and the trick we use is to draw spheres around those points. For each partition, we compute the mean point and draw a hypersphere with that as the center and its distance from the farthest point in that partition as the radius. We recursively repeat this algorithm within each partition i.e. within each sphere. Hence, the data structure we end up with looks like balls within balls within balls (`XD`) ... a ball tree (`XD`). During inference, we run a DFS on the ball tree (similar to kD trees) except that the pruning step involves computing the distance of the test point from a hypersphere instead of from a hyperplane. Ball-tree pruning depends more on metric geometry than on axis alignment, but its performance is not completely invariant to ambient dimension. This is because we're not drawing boundaries along axes anymore, but balls based on the data. The hyperspheres can be centered anywhere. The leaves of the ball tree end up being many small hyperspheres that capture the structure of the data pretty well. Ball trees fulfill the purpose of giving a massive saving in terms of the number of distance computations during inference. However, it may so happen that although ball trees give a massive time saving on paper, the wall clock time for the original brute force computation without fancy data structures may be lower when d grows past a certain threshold. This is because of systems performance (CS 463) reasons where although there are way more computations in the brute force approach, they are such that they are highly optimized on the hardware, and multiple consecutive computations are vectorized (vs. one-by-one computations in ball trees), and so on.

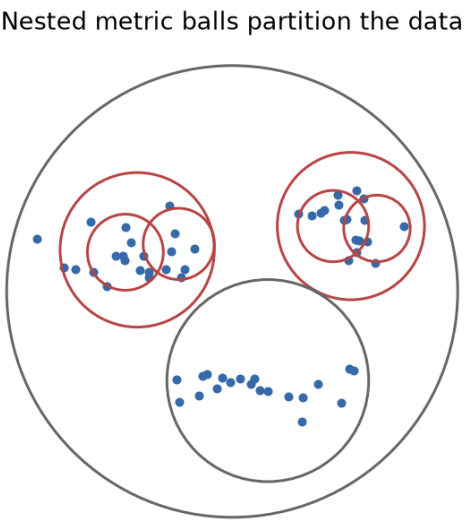

Figure 3: Schematic of a ball-tree partition.

4 Perceptron

The perceptron algorithm was formulated *before* k-nearest neighbors by Frank Rosenblatt at Cornell University in 1957.

$\star$ **Assumption.** There must be a hyperplane that separates one class from the other (assuming a binary classification problem here).

A hyperplane is defined by a vector $\vec{w}$ and a bias b such that $\vec{w} \cdot \vec{x} + b = 0$. This defines a hyperplane with a dimension of one less than that of the ambient space. During test time, I look at $\text{sign}(\vec{w} \cdot \vec{x} + b)$. Formalism:

- $\mathcal{Y} = \{-1, 1\}$. We have to learn the hyperplane $\psi : \vec{w} \cdot \vec{x} + b = 0$ that separates the data.
- We get rid of the bias b by redefining $\vec{x_i} \leftarrow \begin{bmatrix} \vec{x_i} \\ 1 \end{bmatrix}$ and $\vec{w} \leftarrow \begin{bmatrix} \vec{w} \\ b \end{bmatrix}$. Our hyperplane now becomes $\psi : \vec{w} \cdot \vec{x} = 0$. Geometrically, our hyperplane now always goes through the origin.

With this setup, our goal is to learn a $\vec{w}$ such that $\forall\, (\vec{x}, y) \in \mathcal{D},\ \vec{w} \cdot y\vec{x} > 0$. This is the same as saying that the hyperplane $\psi : \vec{w} \cdot \vec{x} = 0$ separates the data points.

Algorithm. We initialize $\vec{w} = 0$. We go through the data points $(\vec{x}, y) \in \mathcal{D}$ one by one (turns out order is not important) and check whether $\vec{w} \cdot y\vec{x} > 0$. If this holds, the point is classified correctly, and we move on. If this doesn't hold, we have to tweak our parameter $\vec{w}$:

- If $y = 1$ and we get $\vec{w} \cdot \vec{x} \leq 0$, we update $\vec{w} \leftarrow \vec{w} + \vec{x}$.
- If $y = -1$ and we get $\vec{w} \cdot \vec{x} \geq 0$, we update $\vec{w} \leftarrow \vec{w} - \vec{x}$.

In summary, on each mistake $(\vec{x}, y)$, we update $\vec{w} \leftarrow \vec{w} + y\vec{x}$. The perceptron update step does not fix the misclassification in one shot, but rather gets the misclassified $\vec{x}$ value closer to the correct side by changing $\vec{w} \cdot y\vec{x}$ by a factor of $\vec{x} \cdot \vec{x}$ towards the correct side. Note that the distance of any

Algorithm 1: Perceptron Learning Algorithm

Input: Training set $\mathcal{D} = \{(\vec{x}_i, y_i)\}_{i=1}^n$
Output: Parameter $\vec{w}$

```
Initialize w⃗ ← 0;
repeat
    m = 0;
    for each (x⃗, y) ∈ D do
        if w⃗ · yx⃗ ≤ 0 then
            w⃗ ← w⃗ + yx⃗;
            m ← m + 1;
        end
    end
until m = 0;
return w⃗;
```

point $\vec{x}$ from the hyperplane $\psi : \vec{w} \cdot \vec{x} = 0$ is $\hat{w} \cdot \vec{x}$. Define the **margin** as:

$$\gamma = \min_{(\vec{x},y)\in\mathcal{D}} |\hat{w^*} \cdot \vec{x}| .$$

The margin is always nonnegative. A large margin is generally preferable in machine learning.

We can formalize the perceptron assumption as $\exists \vec{w}^*,\ \forall (\vec{x}, y) \in \mathcal{D},\ \vec{w}^* \cdot y\vec{x} > 0$. This also tells us that *scaling* $\vec{w}^*$ by any positive factor α will give us a separating hyperplane as $\alpha\vec{w}^* \cdot y\vec{x} > 0$. This allows us to normalize $\vec{w}^*$ and set $\|\vec{w}^*\| = 1$. This also allows us to write our margin as $\gamma = \min_{(\vec{x},y)\in\mathcal{D}} |\vec{w}^* \cdot \vec{x}|$. We can use the same trick for all our data points and say that scaling $\vec{x_i}$ by a positive factor does not change the fact that $\vec{w}^*$ separates the data. This allows us to set $\|\vec{x_i}\| \leq 1$ for all i. Generally, we can scale down each data point by the factor of $\max_i \|\vec{x_i}\|$.

With the two previous steps, our data is now all inside a unit hypersphere and our separating hyperplane passes through the origin and its normal vector is a radius of the hypersphere. We can rescale the data back to the original later.

Convergence. The claim is that, for linearly separable data, the perceptron reaches some separating $\vec{w}$ in a finite number of update steps; it need not recover the particular $\vec{w}^*$ used as a reference in the proof. To check this we watch the quantities $\vec{w} \cdot \vec{w}^*$ and $\vec{w} \cdot \vec{w}$.

Proof. The update step is

$$\vec{w} \leftarrow \vec{w} + y\vec{x},$$

which only happens when

$$\vec{w} \cdot y\vec{x} \leq 0.$$

The change in $\vec{w} \cdot \vec{w}^*$ is

$$\begin{aligned} & (\vec{w} + y\vec{x}) \cdot \vec{w}^* - \vec{w} \cdot \vec{w}^* \\ = \ & \vec{w} \cdot \vec{w}^* + \vec{w}^* \cdot y\vec{x} - \vec{w} \cdot \vec{w}^* \\ = \ & \vec{w}^* \cdot y\vec{x} \end{aligned}$$

But $\vec{w}^* \cdot y\vec{x}$ is positive because $\vec{w}^*$ is the separating hyperplane

$$\Rightarrow (\vec{w} + y\vec{x}) \cdot \vec{w}^* - \vec{w} \cdot \vec{w}^* > 0$$

$\Rightarrow$ The quantity $\vec{w} \cdot \vec{w}^*$ increases on an update. Further,

$$\vec{w}^* \cdot y\vec{x} \geq \gamma \geq 0$$

$\Rightarrow$ The quantity $\vec{w} \cdot \vec{w}^*$ increases by at least γ.

The change in the quantity $\vec{w} \cdot \vec{w}$ is

$$\begin{aligned} & (\vec{w} + y\vec{x}) \cdot (\vec{w} + y\vec{x}) - \vec{w} \cdot \vec{w} \\ &= \vec{w} \cdot \vec{w} + 2\vec{w} \cdot y\vec{x} + \vec{x} \cdot \vec{x} - \vec{w} \cdot \vec{w} \\ &= 2\vec{w} \cdot y\vec{x} + \|\vec{x}\|^2 \end{aligned}$$

But $\vec{w} \cdot y\vec{x} \leq 0$ because $(\vec{x}, y)$ was a misclassification, and $\|\vec{x}\|^2 \leq 1$ because we have scaled it that way

$$\Rightarrow 2\vec{w} \cdot y\vec{x} + \|\vec{x}\|^2 \leq 1$$

$\Rightarrow$ The quantity $\vec{w} \cdot \vec{w}$ increases by at most 1.

Since 1 update changes $\vec{w} \cdot \vec{w}^*$ by at least γ, M updates change $\vec{w} \cdot \vec{w}^*$ by at least $M\gamma$. Hence, assuming we start from $\vec{w} = 0$, after M updates,

$$\begin{aligned} M\gamma &\leq \vec{w} \cdot \vec{w}^* \\ &= |\vec{w} \cdot \vec{w}^*| && \text{(Since we start from 0 and keep increasing by at least } \gamma\text{)} \\ &\leq \|\vec{w}\|\|\vec{w}^*\| && \text{(Cauchy–Schwarz inequality)} \end{aligned}$$

But we have set $\|\vec{w}^*\| = 1$

$$\begin{aligned} \Rightarrow M\gamma &\leq \|\vec{w}\| \\ &= \sqrt{\vec{w} \cdot \vec{w}} \\ &\leq \sqrt{M}. && \text{(Since } \|\vec{w}\|^2 \text{ increases by at most 1 per update)} \end{aligned}$$

We have shown that $M\gamma \leq \sqrt{M}$

$$\Rightarrow M^2\gamma^2 \leq M$$

$$\Rightarrow \boxed{M \leq \frac{1}{\gamma^2}}$$

$\therefore$ The perceptron algorithm converges in a finite number of steps, which only depends on the margin. $\square$

The convergence proof bounds the number of updates in terms of the margin of a reference separator. It does **not** show that the ordinary perceptron returns the maximum-margin separator.

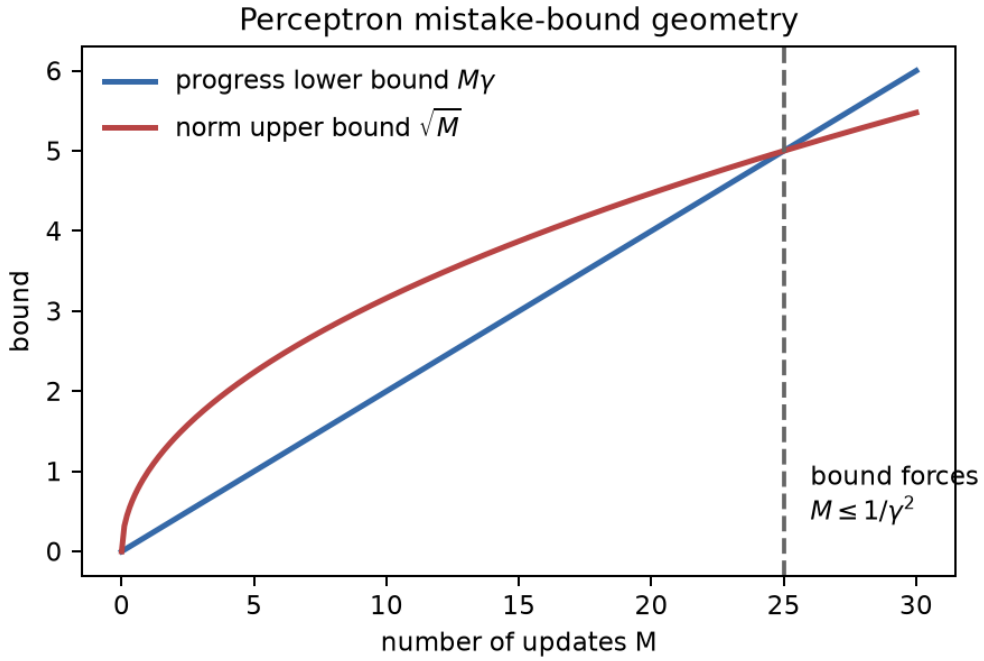

Figure 4: Visualization of the perceptron mistake-bound argument.

5 Maximum Likelihood Estimation

As mentioned earlier about the Bayes optimal classifier, the moment we have access to the *elusive* joint distribution $\mathcal{P}(\vec{x}, y)$, we have an optimal classifier. But in reality we don't have it, and a logical next step would be to try and empirically estimate it from data to then use something like the Bayes optimal classifier.

> "In some sense, one could say that's what all of machine learning is about. All machine learning algorithms, in one way or another, try to estimate the probability $\mathcal{P}(\vec{x}, y)$."

$\star$ $\mathcal{P}(\vec{x}, y)$ can be estimated as $\mathcal{P}(y \mid \vec{x})\mathcal{P}(\vec{x})$ or as $\mathcal{P}(\vec{x} \mid y)\mathcal{P}(y)$. The former is called **discriminative learning** and the latter is called **generative learning**, which are both approaches of machine learning.

Consider a 1D setup of tossing a coin 10 times, where I get 4 heads, and 6 tails. I want to estimate the probability of a toss landing a head. **The principle behind maximum likelihood estimation is that I say given that I observed this data, which parameter will make it most likely that I observed what I observed.** The data $\mathcal{D}$, here, is the composed of the results of the 10 tosses, and the parameter, θ, here, is the probability of a single toss landing a head. Then, I estimate:

$$\theta = \arg\max_{\theta} \; P(\mathcal{D}; \theta) \, .$$

The probability $P(\mathcal{D}; \theta)$ is essentially a binomial distribution since $\mathcal{D}$ is the sequence of results of n tosses.

$$P(\mathcal{D}; \theta) = \binom{n_H + n_T}{n_H} \theta^{n_H} (1 - \theta)^{n_T} \, .$$

This is the **likelihood function**, $L(\mathcal{D}; \theta) = P(\mathcal{D}; \theta)$, and we want to maximize it to find $\arg\max_{\theta} P(\mathcal{D}; \theta)$. We instead work on maximizing the **log-likelihood** to make the computation

easy.

$$\log L = \log \binom{n_H + n_T}{n_H} + n_H \log \theta + n_T \log(1-\theta)\,.$$

We maximize it by taking its derivative with respect to θ and equating it to 0.

$$\begin{aligned}
&\frac{\partial \log L}{\partial \theta} = \frac{n_H}{\theta} - \frac{n_T}{1-\theta} = 0 \\
\Rightarrow &\frac{n_H}{\theta} = \frac{n_T}{1-\theta} \\
\Rightarrow &\boxed{\theta = \frac{n_H}{n_H + n_T}}\,.
\end{aligned}$$

This breaks down when the data is too small or too extreme (all tosses are heads?!). In such a case, we reduce our faith in the data and weigh our prior belief into the final estimate. In the coin toss example, the prior belief is weighed by *hallucinating* some coin tosses according to the prior belief of 50-50 before $\mathcal{D}$ is observed. So the dataset now becomes some sequence of m heads and m tails followed by $\mathcal{D}$, and the estimate becomes:

$$\theta = \frac{n_H + m}{n_H + n_T + 2m}\,.$$

This is called **m-smoothing**.

Maximum likelihood estimation (MLE) is **frequentist statistics**, which is different from **Bayesian statistics**. The latter says that not only is $\mathcal{D}$ a random variable but so is $\theta \Rightarrow \theta$ has its own probability distribution instead of simply being a parameter. In Bayesian statistics, we combine the likelihood $P(\mathcal{D} \mid \theta)$ with a prior $P(\theta)$ to obtain a posterior; MAP maximizes that posterior (wars have been fought), and we draw $P(\theta)$ from our prior belief. $P(\theta)$ is the prior and $P(\theta \mid \mathcal{D})$ is the posterior which we maximize in Bayesian statistics. Likelihood estimation by maximizing the posterior is called **maximum *a posteriori* estimation (MAP)**.

$$P(\theta \mid \mathcal{D}) = \frac{P(\mathcal{D} \mid \theta) P(\theta)}{P(\mathcal{D})}\,.$$

In our coin tossing example, assume a Beta prior:

$$P(\theta; \alpha, \beta) = \frac{\theta^{\alpha-1}(1-\theta)^{\beta-1}}{B(\alpha, \beta)}\,.$$

Maximum *a posteriori* estimation:

$$\begin{aligned}
P(\theta \mid \mathcal{D}) &\propto P(\mathcal{D} \mid \theta) P(\theta) \\
&\propto \binom{n_H + n_T}{n_H} \theta^{n_H} (1-\theta)^{n_T} \theta^{\alpha-1} (1-\theta)^{\beta-1} \\
&\propto \theta^{n_H + \alpha - 1} (1-\theta)^{n_T + \beta - 1}\,.
\end{aligned}$$

Maximizing and solving for θ gives us:

$$\boxed{\theta = \frac{n_H + \alpha - 1}{n_H + n_T + \alpha + \beta - 2}}\,.$$

This is exactly the same thing as smoothing (wars are futile).

Following a truly Bayesian approach, we could justify the entire idea of MLE and MAP by showing how it helps with prediction. MAP leads to the "holy grail" of machine learning where we are able to take an expectation across all models to make a prediction:

$$\mathcal{P}(y \mid \vec{x}) =_{\text{estimate}} \int_\theta P(y \mid \theta) P(\theta \mid \mathcal{D})\, d\theta\,.$$

"Truly beautiful."

6 Naïve Bayes

If we apply the MLE idea to a 2D setup, where we have a dataset $\mathcal{D} = \{(\vec{x_i}, y_i)\}_{i=1}^n \sim P(\vec{x}, y)$ and we want to estimate quantities like $P(\vec{x})$ and $P(y \mid \vec{x})$. We can estimate $P(\vec{x})$ as:

$$P(\vec{x}) = \frac{1}{n} \sum_{i=1}^{n} \mathbb{I}(\vec{x_i} = \vec{x})$$

Similarly,

$$P(y \mid \vec{x}) = \frac{\sum_{i=1}^n \mathbb{I}(\vec{x_i} = \vec{x} \wedge y_i = y)}{\sum_{i=1}^n \mathbb{I}(\vec{x_i} = \vec{x})}\,.$$

But $\forall \vec{x_i}\ \vec{x_i} \in \mathbb{R}^d$, and when we write the condition $\vec{x_i} = \vec{x}$, we are writing $\forall\, 1 \le r \le d,\ x_{ir} = x_r$. We are using this condition to estimate $P(y \mid \vec{x})$. But this condition is quite unreasonable when d is large because then it becomes very unlikely to find a point in $\mathcal{D}$ that has *exactly* the same combination of features. For example, if we have a labeled dataset of faces and $\mathcal{Y}$ is the set of two labels: student and professor, we are estimating the probability of a face being a professor by using the number of times *that exact face* (all pixels exactly the same, all features exactly the same) appears in the dataset. The solution to this problem is the **naïve Bayes estimate**, which estimates:

$$P(y \mid \vec{x}) = \frac{P(\vec{x} \mid y) P(y)}{P(\vec{x})}\,,$$

where the denominator is just a normalizing factor, and $P(y)$ can be easily estimated from $\mathcal{D}$ as $\frac{1}{n}\sum_{i=1}^n \mathbb{I}(y_i = y)$. The caveat is in the computation of $P(\vec{x} \mid y)$, where naïve Bayes makes a crucial but, well, naïve assumption that **all features are independent given the label**. This can be expressed as:

$$P(\vec{x} \mid y) = \prod_{r=1}^{d} P(x_r \mid y)\,.$$

We are cutting corners, a *huge* corner, but it makes computation easier and often doesn't break things too badly. We can write the naïve bayes estimator as:

$$\begin{aligned} h(\vec{x}) &= \arg\max_y\ P(y \mid \vec{x}) \\ &\propto \arg\max_y\ P(y) \prod_{r=1}^{d} P(x_r \mid y) \\ &= \arg\max_y\ \log P(y) + \sum_{r=1}^{d} \log P(x_r \mid y)\,, \end{aligned}$$

where both $P(y)$ and $P(x_r \mid y)$ can now be estimated from $\mathcal{D}$ as discussed before.

Naïve Bayes is a **linear classifier** for multinomial features and for Gaussian class-conditionals when each feature variance is shared across classes; class-dependent Gaussian variances generally give a quadratic boundary.

Proof (multinomial features and binary classification). Let x_r be the count of feature r, let θ_{yr} be the class-conditional multinomial parameter, and let $\pi_y = P(y)$. Up to the multinomial coefficient, which is independent of the class,

$$P(\vec{x} \mid y)P(y) \propto \pi_y \prod_{r=1}^{d} \theta_{yr}^{x_r}.$$

Taking the log posterior odds between classes $+1$ and -1 gives

$$\begin{aligned} \log \frac{P(y=+1 \mid \vec{x})}{P(y=-1 \mid \vec{x})} &= \log \frac{\pi_{+1}}{\pi_{-1}} + \sum_{r=1}^{d} x_r \log \frac{\theta_{+1,r}}{\theta_{-1,r}} \\ &= b + \vec{w}^{\mathsf{T}} \vec{x}, \end{aligned}$$

where $b = \log(\pi_{+1}/\pi_{-1})$ and $w_r = \log(\theta_{+1,r}/\theta_{-1,r})$. Therefore the decision boundary is $\vec{w}^{\mathsf{T}}\vec{x} + b = 0$, which is linear. □

Naïve Bayes is a generative algorithm, unlike the perceptron which is a discriminative algorithm. The perceptron finds a hyperplane separating the data, whereas naïve Bayes first models the data into a Gaussian or a multinomial distribution as per the assumption, and then finds a hyperplane separating the two distributions $P(y = 1 \mid \vec{x})$ and $P(y = -1 \mid \vec{x})$. Hence, naïve Bayes will find a hyperplane even if the data is not linearly separable (it will just model the distributions according to the data and find a separating hyperplane between them), whereas the perceptron will keep looping over the data indefinitely and never converge.

The Gaussian naïve Bayes approach elicits a closed-form expression in terms of parameters $\vec{w}$ and b:

$$P(y \mid \vec{x}) = \frac{1}{1 + e^{-y(\vec{w}\cdot\vec{x}+b)}} \qquad \text{(Sigmoid function)},$$

where the hyperplane $\psi : \vec{w} \cdot \vec{x} + b = 0$ separates the two Gaussians. Hence, for any point, $\vec{x}$, the expression $\vec{w} \cdot \vec{x} + b$ is proportional to the distance of $\vec{x}$ from the hyperplane ψ. The expression obtained for $P(y \mid \vec{x})$ says that the larger the distance of a point $\vec{x}$ from ψ for a y, the higher the probability that the label of $\vec{x}$ is y.

7 Logistic Regression

The logistic regression algorithm is the discriminative counterpart of naïve Bayes. Naïve Bayes finds $P_\theta(\vec{x} \mid y)P_{\theta'}(y)$ (which makes it generative) and in the case where a Gaussian prior for the features is assumed, this elicits a closed-form expression in terms of parameters $\vec{w}_\theta$ and b_θ. Logistic regression says that since there is a nice closed-form expression at the other end of the tunnel, why don't we try to estimate $\vec{w}$ and b directly. Hence, it estimates $P_{\vec{w},b}(y \mid \vec{x})$, which makes it discriminative.

⋆ **Assumption.** The probability of the label given the features, $P(y \mid \vec{x})$, has the form $\frac{1}{1+e^{-y(\vec{w}\cdot\vec{x}+b)}}$ for some parameters $\vec{w}$ and b.

The big difference is that now we're making an assumption about $P(y \mid \vec{x})$, whereas in naïve Bayes we make an assumption about $P(\vec{x} \mid y)$. With this assumption, we find the parameters $\vec{w}$ and b that best fit the data. First, we absorb b into $\vec{w}$ as we had done for the perceptron. Then, we use MLE:

$$\begin{aligned}
\vec{w} &= \arg\max_{\vec{w'}} \prod_{(\vec{x},y)\in\mathcal{D}} P_{\vec{w'}}(y \mid \vec{x}; \vec{w'}) \\
&= \arg\max_{\vec{w'}} \sum_{(\vec{x},y)\in\mathcal{D}} \log\left(P_{\vec{w'}}(y \mid \vec{x}; \vec{w'})\right) \\
&= \arg\max_{\vec{w'}} \sum_{(\vec{x},y)\in\mathcal{D}} \log\left(\frac{1}{1+e^{-\vec{w'}\cdot y\vec{x}}}\right) \\
&= \arg\min_{\vec{w'}} \sum_{(\vec{x},y)\in\mathcal{D}} \log\left(1+e^{-\vec{w'}\cdot y\vec{x}}\right) .
\end{aligned}$$

Now we're stuck because there isn't actually a closed-form solution for $\vec{w}$ here (we have to use gradient descent (stay tuned!)).

Let's try with MAP:

$$\begin{aligned}
\vec{w} &= \arg\max_{\vec{w'}} \; P(\vec{w'} \mid \mathcal{D}) \\
&= \arg\max_{\vec{w'}} \; P(\mathcal{D} \mid \vec{w'})P(\vec{w'}) .
\end{aligned}$$

Here, $P(\mathcal{D} \mid \vec{w'})$ is exactly the same thing as MLE, and $P(\vec{w'})$ comes from our prior belief of how the hyperplane should be. We say that it's good to assume $\vec{w'}$ to be centered around $\vec{0}$ so that it is simple. This translates to the fact that $\vec{w'}$ is normally distributed, or $\vec{w'} \sim \mathcal{N}\left(\vec{0},\, \tau^2 I_d\right)$. This gives:

$$\vec{w} = \arg\min_{\vec{w'}} \sum_{(\vec{x},y)\in\mathcal{D}} \log\left(1+e^{-\vec{w'}\cdot y\vec{x}}\right) + \frac{1}{2\tau^2}\vec{w'}\cdot\vec{w'} ,$$

where an additional term depending on $\vec{w'}\cdot\vec{w'}$ gets tacked onto the MLE result that says that the learnt $\vec{w}$ should be simple.

8 Convex Optimization

> Notation note: For a function $\ell(\vec{w})$, $\vec{\nabla}\ell(\vec{w})$ is the same as $\frac{d\ell}{d\vec{w}}$, and $\vec{\nabla}_{w_i}\ell(\vec{w})$ is the same as $\frac{\partial\ell}{\partial w_i}$. For a function $f(\vec{x},\vec{y})$, $\vec{\nabla}_{\vec{x}} f$ is the same as $\frac{\partial f}{\partial \vec{x}}$.

Now we'll try to tackle the abstract problem of finding the minimum of some function $\ell(\vec{w})$, where we got stuck in logistic regression. We know that the function ℓ is continuous, differentiable, and convex. By Taylor's expansion, we can approximate:

$$\ell(\vec{w}+\vec{s}) \approx \ell(\vec{w}) + \vec{\nabla}\ell(\vec{w})\cdot\vec{s}.$$

We are approximating the function as linear *locally* and arguing that this is not a bad approximation. We can make a better approximation by approximating it locally as quadratic:

$$\ell(\vec{w}+\vec{s}) \approx \ell(\vec{w}) + \vec{\nabla}\ell(\vec{w})\cdot\vec{s} + \frac{1}{2}\nabla^2\ell(\vec{w})\vec{s}\cdot\vec{s},$$

where $\nabla^2 \ell(\vec{w})$ is the **Hessian** of ℓ at $\vec{w}$—a symmetric matrix such that:

$$\left(\nabla^2 \ell(\vec{w})\right)_{ij} = \vec{\nabla}_{w_i} \cdot \vec{\nabla}_{w_j} \ell(\vec{w}) \,.$$

This assumption only holds for small steps $\vec{s}$.

8.1 Gradient Descent

Gradient descent (GD) is an algorithm for convex optimization that believes the locally linear approximation and sets the vector $\vec{s}$ to be a multiple of the negative gradient of the loss at $\vec{w}$ so that when we add $\vec{s}$ to $\vec{w}$, we go downhill:

$$\vec{s} = -\alpha \vec{\nabla} \ell(\vec{w}) \,,$$

where α is called the **learning rate** and is such that $\alpha > 0$ and small enough for the Taylor approximation to hold.

Sanity check:

$$\begin{aligned}
\ell(\vec{w} + \vec{s}) &= \ell(\vec{w} - \alpha \vec{\nabla} \ell(\vec{w})) \\
&\approx \ell(\vec{w}) - \alpha \vec{\nabla} \ell(\vec{w}) \cdot \vec{\nabla} \ell(\vec{w}) && \text{(by Taylor approximation)} \\
&= \ell(\vec{w}) - \alpha \|\vec{\nabla} \ell(\vec{w})\|^2 \\
&< \ell(\vec{w}) \,. && (\because \|\vec{\nabla} \ell(\vec{w})\|^2 > 0 \text{ when we're not at the minimum})
\end{aligned}$$

8.2 AdaGrad

Gradient descent where the step size is adaptively learnt.

Algorithm 2: AdaGrad algorithm

Input: Loss function $\ell : \mathbb{R}^d \to \mathbb{R}$, initial step size $\alpha > 0$, tolerance $\delta > 0$, smoothing $\varepsilon > 0$
Output: Optimized parameter vector $\vec{w}^*$

Initialize $\vec{w}_0 \leftarrow \vec{0}$; `// or random`
Initialize $\vec{s}_0 \leftarrow \vec{0}$;
$t \leftarrow 0$;
repeat
  $\vec{g}_t \leftarrow \vec{\nabla} \ell(\vec{w}_t)$;
  $\vec{s}_{t+1} \leftarrow \vec{s}_t \; + \; \vec{g}_t \odot \vec{g}_t$;
  $\vec{w}_{t+1} \leftarrow \vec{w}_t \; - \; \dfrac{\alpha}{\sqrt{\vec{s}_{t+1} + \varepsilon}} \; \odot \; \vec{g}_t$;
  $t \leftarrow t + 1$;
until $\|\vec{w}_{t+1} - \vec{w}_t\| < \delta$;
return $\vec{w}_t$;

8.3 Newton's Method

Newton's method is similar to gradient descent except that now we're approximating the function as locally quadratic. The assumption is:

$$\ell(\vec{w}+\vec{s}) \approx \ell(\vec{w}) + \vec{\nabla}\ell(\vec{w}) \cdot \vec{s} + \frac{1}{2}\nabla^2\ell(\vec{w})\vec{s} \cdot \vec{s}.$$

So, at each iteration, we take the step that takes us to the minimum of the parabola that approximates ℓ locally:

$$\vec{s} = \arg\min_{\vec{s}} \ \ell(\vec{w}) + \vec{\nabla}\ell(\vec{w}) \cdot \vec{s} + \frac{1}{2}\nabla^2\ell(\vec{w})\vec{s} \cdot \vec{s}.$$

Computing the gradient with respect to $\vec{s}$ and equating it to $\vec{0}$:

$$\vec{\nabla}\ell(\vec{w}) + \nabla^2\ell(\vec{w})\vec{s} = \vec{0}$$

$$\Rightarrow \boxed{\vec{s} = -\left(\nabla^2\ell(\vec{w})\right)^{-1}\vec{\nabla}\ell(\vec{w})}.$$

Near a nondegenerate optimum and under regularity conditions, Newton's method can converge quadratically, but it is not unconditionally faster than gradient descent. Step size is an important parameter for gradient descent, whereas the initial $\vec{w} = \vec{w_0}$ is an important parameter for Newton's method. Newton's method may diverge if the gradient is too shallow making the step size too big since Taylor's approximation doesn't hold for big step sizes. The recommendation is to do gradient descent (or AdaGrad with adaptive step size updates) until we are close to the minimum, and then jump to the minimum using Newton's method.

9 Linear Regression

Note: Notation change $\vec{w} \cdot \vec{x} \longrightarrow \vec{w}^\mathsf{T}\vec{x}$ (because we finished 18.06)

In logistic regression, the objective we got was

$$\ell(\vec{w}) = \sum_{(\vec{x},y)\in\mathcal{D}} \log\left(1 + e^{-\vec{w}^\mathsf{T}\vec{x}y}\right),$$

which can be interpreted as a loss function: the **logistic loss**. In logistic regression, there is a notion of how correctly we predict the label for a point, that correlates with the distance of that point from the separating hyperplane on the correct side. The more correctly we predict a point, the greater its distance from the hyperplane $\Rightarrow$ greater $\vec{w}^\mathsf{T}\vec{x}$ in the correct direction i.e. the number $\vec{w}^\mathsf{T}\vec{x}y$ is more positive. Hence, $e^{-\vec{w}^\mathsf{T}\vec{x}y} \to 0 \Rightarrow$ the loss on that data point goes to 0. The more incorrectly we predict a point, the greater its distance from the hyperplane $\Rightarrow$ greater $\vec{w}^\mathsf{T}\vec{x}y$ in the wrong direction i.e. the number $\vec{w}^\mathsf{T}\vec{x}y$ is more negative. Hence $e^{-\vec{w}^\mathsf{T}\vec{x}y} \to \infty \Rightarrow$ the loss on that data point goes to $-\vec{w}^\mathsf{T}\vec{x}y$. Logistic regression, despite what the name suggests, is a classification algorithm. Linear regression, on the other hand, is a regression algorithm, where for each x, we have a value $y \in \mathbb{R}$. We no longer have classes. This is also called **ordinary least squares (OLS)**.

$\star$ The **assumptions** of linear regression are:

- We assume that the correspondence between x and y is linear i.e. there is some linear function (line) that models that.

- The data may not lie on a line because the measurements/values may be noisy. So, we assume a distribution or a **noise model** that says that the probability distribution of a data point lying on the line is Gaussian.

Formally, the assumption is that $\forall\,(\vec{x_i}, y_i) \in \mathcal{D},\ y_i = \vec{w}^\mathsf{T}\vec{x_i} + \varepsilon_i$, where $\varepsilon_i \sim \mathcal{N}\,(0,\,\sigma^2)$; or equivalently, $y_i \sim \mathcal{N}\left(\vec{w}^\mathsf{T}\vec{x_i},\,\sigma^2\right)$. We want to estimate $\vec{w}$ by maximizing the likelihood $P(y_i \mid \vec{x_i}; \vec{w}) = \frac{1}{\sqrt{2\pi\sigma^2}} e^{-\frac{\left(\vec{w}^\mathsf{T}\vec{x_i} - y_i\right)^2}{2\sigma^2}}$.

- Solving this with MLE gives the least squares objective $\vec{w} = \arg\min_{\vec{w'}} \sum_{(\vec{x_i}, y_i)\in\mathcal{D}} \left(\vec{w'}^\mathsf{T}\vec{x_i} - y_i\right)^2$. This can be interpreted as the **squared loss**. We typically multiply by $\frac{1}{n}$ for uniformity across differently sized datasets, and minimize the mean squared loss:

$$\boxed{\ell(\vec{w}) = \frac{1}{n}\sum_{i=1}^{n}\left(\vec{w}^\mathsf{T}\vec{x_i} - y_i\right)^2}.$$

- Solving this with MAP, with the prior assumption that $\vec{w} \sim \mathcal{N}\left(\vec{0},\,\tau^2 I_d\right)$, gives:

$$\vec{w} = \arg\min_{\vec{w'}}\ \frac{1}{n}\sum_{i=1}^{n}\left(\vec{w'}^\mathsf{T}\vec{x_i} - y_i\right)^2 + \frac{\sigma^2}{\tau^2}\|\vec{w'}\|_2^2\,.$$

$\star$ The objective is a parabola and hence has a closed-form minimum, which can be computed with one step of Newton's method.

Define a matrix $X = \begin{bmatrix} \vec{x_1}^\mathsf{T} \\ \vec{x_2}^\mathsf{T} \\ \vdots \\ \vec{x_n}^\mathsf{T} \end{bmatrix}$ and a vector $\vec{y} = \begin{bmatrix} y_1 \\ y_2 \\ \vdots \\ y_n \end{bmatrix}$. Then,

$$\begin{aligned}
\ell(\vec{w}) &= \|X\vec{w} - \vec{y}\|_2^2 \\
&= (X\vec{w} - \vec{y})^\mathsf{T}(X\vec{w} - \vec{y}) \\
&= \vec{w}^\mathsf{T}X^\mathsf{T}X\vec{w} - 2\vec{y}^\mathsf{T}X\vec{w} + \vec{y}^\mathsf{T}\vec{y}\,.
\end{aligned}$$

Taking the derivative w.r.t $\vec{w}$ and setting it to 0,

$$\begin{aligned}
&\frac{d\ell}{d\vec{w}} = 2X^\mathsf{T}X\vec{w} - 2X^\mathsf{T}\vec{y} = 0 \\
\Rightarrow &X^\mathsf{T}X\vec{w} = X^\mathsf{T}\vec{y} \\
\therefore &\boxed{\vec{w} = \left(X^\mathsf{T}X\right)^{-1}X^\mathsf{T}\vec{y}}.
\end{aligned}$$

Forming the normal equations costs $O(nd^2)$ time and $O(d^2)$ space, and a dense solve costs $O(d^3)$ time, which makes gradient descent practically more feasible.

10 Support Vector Machines

If data for a classification problem is linearly separable then the perceptron finds *a* separating hyperplane. However, it is weak in the sense that it gives no guarantees as to which one it will find. There does exist a separating hyperplane that we indeed want, and it can be characterized based on the concept of the **margin** (also introduced in the perceptron).

$\star$ The **margin** for a separating hyperplane is the (perpendicular) distance to the data point closest to it. Formally, suppose the hyperplane is $\psi : \vec{w}^\mathsf{T}\vec{x} + b = 0$, then the distance of any point $\vec{v}$ from ψ is $\frac{\left|\vec{w}^\mathsf{T}\vec{v}+b\right|}{\|\vec{w}\|_2}$. Hence, the margin of ψ, denoted by γ, is:

$$\gamma\left(\vec{w}, b\right) = \min_{(\vec{x},y)\in\mathcal{D}} \frac{\left|\vec{w}^\mathsf{T}\vec{x} + b\right|}{\|\vec{w}\|_2}.$$

The separating hyperplane that we want is the one with the maximum margin. A **support vector machine (SVM)** finds exactly that. The SVM objective becomes:

$$\max_{\vec{w},b} \ \gamma\left(\vec{w}, b\right).$$

This objective only says that the margin should be maximized without baking in that the hyperplane should be separating. This would optimize to a hyperplane at infinity. But we also want the hyperplane to be *separating*. Hence, we must subject the optimization to the constraint: $\forall i \ y_i\left(\vec{w}^\mathsf{T}\vec{x_i} + b\right) \geq 0$.

Expanding $\gamma\left(\vec{w}, b\right)$ in the objective,

$$\begin{aligned}\vec{w^*}, b^* &= \arg\max_{\vec{w},b} \left(\min_{(\vec{x},y)\in\mathcal{D}} \frac{\left|\vec{w}^\mathsf{T}\vec{x} + b\right|}{\|\vec{w}\|_2}\right) \\ &= \arg\max_{\vec{w},b} \frac{1}{\|\vec{w}\|_2}\left(\min_{(x,y)\in\mathcal{D}} \left|\vec{w}^\mathsf{T}\vec{x} + b\right|\right).\end{aligned}$$

Cool trick: The hyperplane $\vec{w}^\mathsf{T}\vec{x} + b = 0$ does not change on scaling $\vec{w}$ or b. This is a degree of freedom we can utilize by fixing a scale that's really convenient to us. We fix the scale such that:

$$\min_{(\vec{x},y)\in\mathcal{D}} \left|\vec{w}^\mathsf{T}\vec{x} + b\right| = 1.$$

We rescale it in exactly this way (making this a constraint). The SVM objective then becomes:

$$\begin{aligned}\vec{w^*}, b^* &= \arg\max_{\vec{w},b} \frac{1}{\|\vec{w}\|_2} \\ &= \arg\max_{\vec{w},b} \frac{1}{\sqrt{\vec{w}^\mathsf{T}\vec{w}}} \\ &= \arg\max_{\vec{w},b} \frac{1}{\vec{w}^\mathsf{T}\vec{w}}.\end{aligned}$$

Maximization is for losers. We minimize:

$$\vec{w^*}, b^* = \arg\min_{\vec{w},b} \ \vec{w}^\mathsf{T}\vec{w}.$$

At this point, our optimization problem is subject to two constraints:

1. $\forall i \; y_i \left(\vec{w}^\mathsf{T} \vec{x_i} + b \right) \geq 0$,
2. $\min_{(\vec{x},y) \in \mathcal{D}} \; \left| \vec{w}^\mathsf{T} \vec{x} + b \right| = 1$.

In the optimum, i.e. for the $\vec{w}$ and b which minimize $\vec{w}^\mathsf{T} \vec{w}$, these constraints are exactly equivalent to ($\Leftrightarrow$) the single constraint $\forall i \; y_i \left(\vec{w}^\mathsf{T} \vec{x_i} + b \right) \geq 1$. Finally, we have the SVM objective as $\min_{\vec{w},b} \; \vec{w}^\mathsf{T} \vec{w}$ subject to the constraint $\forall i \; y_i \left(\vec{w}^\mathsf{T} \vec{x_i} + b \right) \geq 1$. This is an optimization problem with a quadratic objective subject to a linear constraint, which makes it a **quadratic program** that can be solved using a quadratic programming (QP) solver. It is entirely possible that the QP solver outputs infeasible i.e. there is no solution satisfying the constraints. Do we then give up, or still output something reasonable that separates most of the points? To do the latter, we relax the constraint by adding some **slack**:

$$\forall i \; y_i \left(\vec{w}^\mathsf{T} \vec{x_i} + b \right) \geq 1 - \xi_i,$$

where $\forall i \; \xi_i \geq 0$. This relaxes the constraint because whatever the value on the LHS is, we can always subtract enough from 1 on the RHS to satisfy the constraint for all i. However, we still want the original constraint to be satisfied as much as possible, i.e. we want the ξ_i to be as small as possible. Hence, we modify our objective:

$$\min_{\vec{w},b} \; \vec{w}^\mathsf{T} \vec{w} + C \sum_{i=1}^{n} \xi_i$$

subject to the constraints $\forall i \; y_i \left(\vec{w}^\mathsf{T} \vec{x_i} + b \right) \geq 1 - \xi_i$ and $\forall i \; \xi_i \geq 0$. This is the soft-margin SVM formulation. C is some cost that penalizes bigger ξ_i's i.e. more slack. It is a hyperparameter that is finetuned by the user as opposed to learnt by the machine learning algorithm.

The soft-margin SVM objective can be solved using a QP solver. Alternatively, we know the value of ξ_i:

$$\xi_i = \begin{cases} 1 - y_i \left(\vec{w}^\mathsf{T} \vec{x_i} + b \right) & \text{if } y_i \left(\vec{w}^\mathsf{T} \vec{x_i} + b \right) < 1 \\ 0 & \text{if } y_i \left(\vec{w}^\mathsf{T} \vec{x_i} + b \right) \geq 1 \end{cases}.$$

We can stick it directly into the objective:

$$\min_{\vec{w},b} \; \vec{w}^\mathsf{T} \vec{w} + C \sum_{i=1}^{n} \max \left(1 - y_i \left(\vec{w}^\mathsf{T} \vec{x_i} + b \right), 0 \right).$$

Notice now that this takes the form of minimizing a loss function plus a regularizer, where

$$\ell \left(\vec{w}, b \right) = C \sum_{i=1}^{n} \max \left(1 - y_i \left(\vec{w}^\mathsf{T} \vec{x_i} + b \right), 0 \right) \; + \; \vec{w}^\mathsf{T} \vec{w}.$$

This loss function is called the **hinge loss**. With this, we can use a subgradient or another convex optimizer and do not need a QP solver.

11 Empirical Risk Minimization

For the SVM, we started out with a geometric intuition and it turned out that we could write it as a loss function, similar to logistic regression. As it turns out, a lot of machine learning can be written in exactly that format. Be it a probabilistic intuition (e.g.: LR), or a geometric intuition (e.g.: SVM), we end up with a function we are trying to minimize. Typically, this function has two terms: one that we call the **loss** and the other that we call the **regularizer**.

1. The loss function involves the data and the parameters we are fitting (e.g.: $\vec{x}$, y, and $\vec{w}$) and it measures how many mistakes we are making. The loss function is large if we make many misclassifications (mistakes) on the training data.

2. The regularizer doesn't involve the data. It just involves the parameters we're fitting and it penalizes overly complicated answers/solutions (e.g.: in the SVM, simplicity is measured by the squared norm of $\vec{w}$).

$\star$ In general, we would like to find an answer that explains our training data and is as simple as possible.

The general objective is:

$$\min_{\vec{w}} \ \frac{1}{n}\sum_{i=1}^{n} \ell\left(h_{\vec{w}}\left(\vec{x_i}\right), y_i\right) \ + \ \lambda r(\vec{w}),$$

where $h_{\vec{w}}$ is the classifier corresponding to the parameters $\vec{w}$, ℓ is the loss function which measures how close I am to the true label (large if far), r is the regularizer which measures the complexity of the solution (large if complex), and λ trades-off between the two (e.g.: in the SVM, $\lambda = \frac{1}{C}$). The MLE and MAP procedures also yield similar formats. A lot of theory of machine learning is built on top of this framework which we call **empirical risk minimization (ERM)**.

11.1 Loss

Some popular classification losses:

1. **Hinge loss:** $\left(\max\left(1 - h_{\vec{w}}\left(\vec{x}\right) y, 0\right)\right)^p$.
   - $p = 1 \rightarrow$ SVM, $p = 2 \rightarrow$ squared loss SVM. People use $p = 2$ because it makes the function differentiable and some solvers only work for differentiable functions, etc.
2. **Log-loss:** $\log\left(1 + e^{-h_{\vec{w}}(\vec{x})y}\right)$.
   - We get well-calibrated probabilities from the log-loss which provide as confidence estimates of a point being class 1 or -1.
3. **Exponential loss:** $e^{-h_{\vec{w}}(\vec{x})y}$.
   - Tiniest mistakes are heavily penalized. It's the loss that loses its nerves and freaks out at you.
   - It doesn't allow any misclassifications $\Rightarrow$ great if data is not noisy, but blows up in one's face otherwise.
4. **0/1 loss:** $\delta_K\left(\text{sign}\left(h_{\vec{w}}\left(\vec{x}\right)\right) \neq y\right)$.
   - Cannot be minimized in practice, but is typically used to measure the training error.

Some popular regression losses:

1. **Squared loss:** $\left(h_{\vec{w}}\left(\vec{x}\right) - y\right)^2$.
   - It tries to fit the average i.e. estimate the mean, since minimizing the squared loss with a constant function $h_{\vec{w}}$ gives the $h_{\vec{w}}$ as the mean of the training outputs.
   - Minimizing the squared loss for classification with a linear classifier is simply OLS.
   - Throws off model if outliers exist in training data since it tries to fit to outliers.

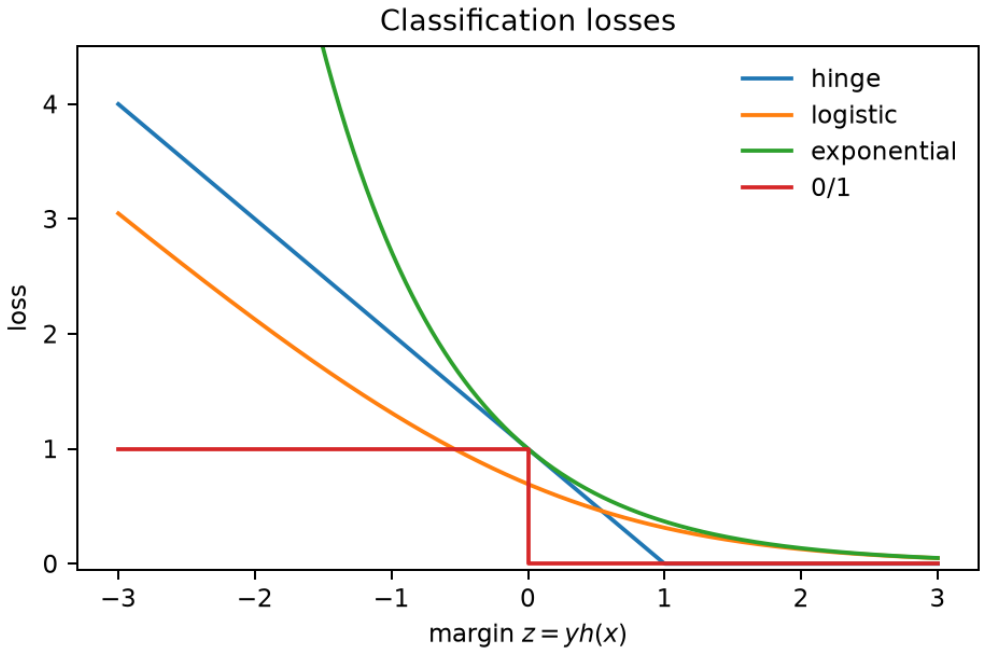

Figure 5: Comparison of classification losses.

2. **Absolute loss:** $|h_{\vec{w}}(\vec{x}) - y|$.
   - It tries to estimate the median (just like the squared loss tries to estimate the mean).
   - Agnostic to outliers but not differentiable everywhere.
3. **Huber loss:** $\begin{cases} \frac{1}{2}\left(h_{\vec{w}}(\vec{x}) - y\right)^2 & \text{if } |h_{\vec{w}}(\vec{x}) - y| < \delta \\ \delta\left(|h_{\vec{w}}(\vec{x}) - y|\right) - \frac{\delta}{2} & \text{otherwise} \end{cases}$.
   - Best of both worlds, preferred loss.
   - For large losses we get the penalty of the absolute loss and not the much larger squared loss. Hence, it is outlier-friendly.
4. **Log-cosh loss:** $\log\left(\cosh\left(h_{\vec{w}}(\vec{x}) - y\right)\right)$.
   - Computationally expensive since computing exponentials is slow.

11.2 Regularization

⋆ A regularizer can be viewed as saying give me a simple solution.

In optimization theory, the general objective:

$$\min_{\vec{w}} \frac{1}{n}\sum_{i=1}^{n} \ell\left(h_{\vec{w}}(\vec{x_i}), y_i\right) + \lambda r(\vec{w})$$

is the **Lagrangian formulation** of the objective:

$$\min_{\vec{w}} \frac{1}{n}\sum_{i=1}^{n} \ell\left(h_{\vec{w}}(\vec{x_i}), y_i\right)$$

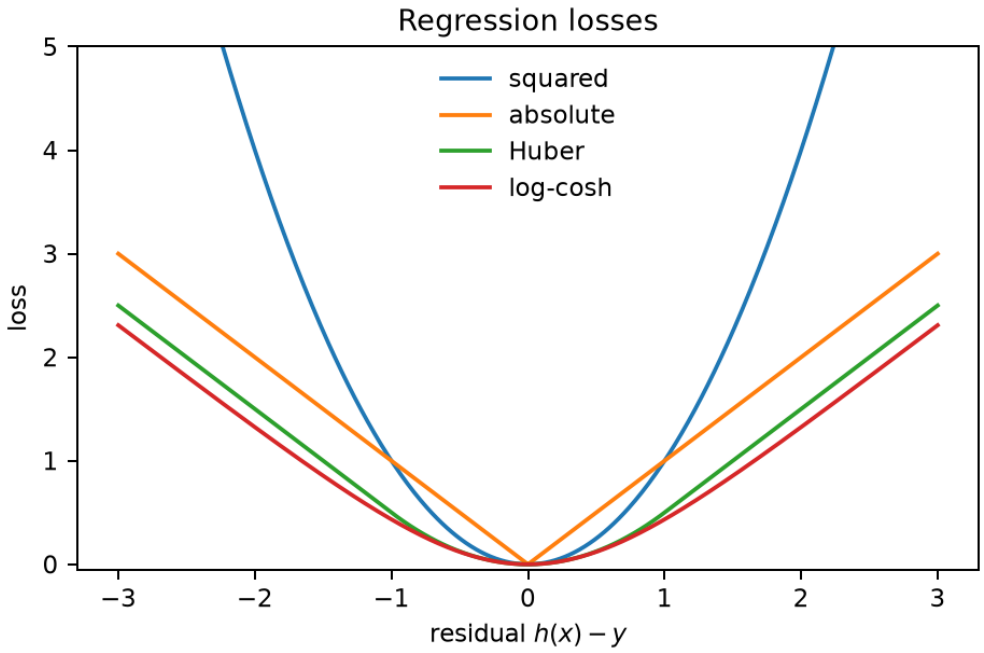

Figure 6: Comparison of regression losses.

subject to the constraint $r(\vec{w}) \leq B$, where B is a constant budget. Under convexity and appropriate constraint qualifications, solutions of the penalized and constrained formulations correspond for suitable λ and B; the mapping need not be one-to-one. If λ is large then I'm mainly trying to minimize $r(\vec{w})$ in the general objective and if λ is small then I'm mainly trying to minimize the loss. Smaller $\lambda \Rightarrow$ larger $B \Rightarrow$ more relaxed the constraint $\Rightarrow$ less regularization. Hence, the regularizer is nothing but a constraint. For instance, if the regularizer is $r(\vec{w}) = \vec{w}^{\mathsf{T}}\vec{w}$ like in SVM or LR, then it translates to the constraint that $\vec{w}^{\mathsf{T}}\vec{w} \leq B \Rightarrow \|\vec{w}\|_2 \leq \sqrt{B}$ while optimizing the loss. Hence, the constraint is that the $\vec{w}$ must lie within a ball of radius $\sqrt{B}$. If gradient descent on the loss finds a $\vec{w^*}$ that is outside the ball, then the solution becomes the point on the hypersphere closest to $\vec{w^*}$. If the regularizer is the squared Euclidean norm, $r(\vec{w}) = \vec{w}^{\mathsf{T}}\vec{w} = \|\vec{w}\|_2^2$, then it is called ℓ_2-regularization. If the regularizer is the ℓ_1-norm, $r(\vec{w}) = \|\vec{w}\|_1$, then it is called ℓ_1-regularization. The constraint that ℓ_2-regularization imposes is $\vec{w}^{\mathsf{T}}\vec{w} \leq B$ i.e. $\vec{w}$ must lie within a ball. The constraint that ℓ_1-regularization imposes is $\|\vec{w}\|_1 \leq B$ i.e. $\vec{w}$ must lie within an ℓ_1-ball (diamond in 2D, octahedron in 3D, etc.). An ℓ_1-ball has corners. In n dimensions, it has $2n$ corners or vertices, with each coordinate axis in $\mathbb{R}^n$ penetrating two of the vertices.

With ℓ_1-regularization, if gradient descent finds a solution, $\vec{w^*}$, outside the ℓ_1-ball, it is very likely that the point on the ℓ_1-ball closest to $\vec{w^*}$ is a vertex, which becomes the final solution. But, for a vertex in n dimensions, $n-1$ dimensions are 0's $\Rightarrow$ the final solution is very sparse. Hence, the ℓ_1-regularizer, unlike the ℓ_2-regularizer, simply kills off the features that contribute little by making their weights 0. It increases sparsity. ℓ_1-regularization can be used to identify which features are most correlated with the output. The tighter the budget B, the smaller the permissible ℓ_1-ball, the higher the chance that all the weight is concentrated only on one feature. We can start out with a somewhat tight budget to identify the best feature and then gradually increase the budget to keep identifying the second best, third best, etc. features. One downside of the ℓ_1-regularizer is that it's not strictly convex. This means that if we have two same (or highly correlated) features, ℓ_1 wouldn't be able to tell the difference between the solutions having the same sums of weights of the

two features in terms of optimality. This leads to multiple solutions (e.g.: all weight on the first and no weight on the second, no weight on the first and all weight on the second, $\frac{1}{2}$-$\frac{1}{2}$, $\frac{1}{4}$-$\frac{3}{4}$, etc.). On the other hand, the squared ℓ_2 penalty is strictly convex. When combined with a convex loss and applied to all coefficients, it makes the full objective strictly convex and yields a unique solution which is $\frac{1}{2}$-$\frac{1}{2}$ because cost of weighing one feature more increases quadratically. To circumvent the downside, people do **elastic net** regularization, which is a little bit of both: $\lambda \|\vec{w}\|_1 + \mu \|\vec{w}\|_2^2$, where μ is very small. With $\mu > 0$, the elastic-net penalty is strictly convex and combines sparsity with a squared ℓ_2 term.

Both ℓ_1 and ℓ_2 regularizations lead to different notions of complexity.

1. ℓ_1-regularization says that the solution that puts little weights on all the features is more complicated than the one that puts all its weight on a few features.
2. ℓ_2-regularization, on the other hand, says that the solution that puts all its weight on a few features is more complex than the one that puts little weights on all the features.
3. Mathematically, ℓ_1 tries to minimize $\|\vec{w}\|_1 = \sum_\alpha |w_\alpha|$, whereas ℓ_2 tries to minimize $\|\vec{w}\|_2 = \sum_\alpha w_\alpha^{\mathbf{2}}$. A high weight leads to a much higher square which is why ℓ_2 cannot afford high weights and it tries to put little weights on all features (a big square is larger than the sum of many small squares). On the other hand, little weights on all features raise the sum of the weights more than one high weight does.
4. As a result, ℓ_1-regularization is brittle, whereas ℓ_2 is more robust since the weight is more spread out.

In practice, we want to identify the most useful features (need ℓ_1 with a tight budget) but also come up with a good predictor (conflicts with a tight budget). ℓ_1-regularization under a tight budget does the former well but then goes on to cramp/confine the best/selected features to a small space within the tight ℓ_1-ball when actually their (w_α^*) lie well outside it. Hence, people use ℓ_1-regularization with a tight budget for good best-feature-identification, but then train with ℓ_2 or no regularization to get a good predictor.

Squared loss with ℓ_2-regularization is called **ridge**. Squared loss with ℓ_1-regularization is called **lasso**. Squared loss with both ℓ_1 and ℓ_2 is called **elastic net**.

12 Bias-Variance Trade-off

> "The most important topic in machine learning."
>
> "...distinguishes children from adults."
>
> "...difference between people playing around with algorithms and people who really know what they're doing."

Recall the generalization error. Until now we've tried to minimize the training error and hoped that it would minimize the generalization error. Consider a regression setting: given dataset $\mathcal{D} = \{(\vec{x}_i, y_i)\}_{i=1}^n$, where $y_i \in \mathbb{R}$, and the n data points are drawn iid from a distribution $\mathcal{P}(\vec{x}, y) = \mathcal{P}(y \mid \vec{x})\,\mathcal{P}(\vec{x})$. Analogous to the Bayes optimal classifier for classification, in this regression setting, we want to predict the expected label given $\vec{x}$:

$$\overline{y}(\vec{x}) = \mathbb{E}_{y|\vec{x}}[y] = \int_{y \in \mathcal{Y}} y\,\mathcal{P}(y \mid \vec{x})\; dy.$$

In practice, we typically don't know the elusive probability distribution $\mathcal{P}(\vec{x}, y)$. So we use a machine learning algorithm $\mathcal{A}$ that takes the dataset $\mathcal{D}$ and outputs a regressor $h_{\mathcal{D}}$:

$$h_{\mathcal{D}} = \mathcal{A}(\mathcal{D}).$$

Given $h_{\mathcal{D}}$, the expected test error using the squared loss is:

$$\mathbb{E}_{\vec{x},y\sim\mathcal{P}}\left[\left(h_{\mathcal{D}}(\vec{x}) - y\right)^2\right] = \int_{\vec{x}\in\mathcal{X}}\int_{y\in\mathcal{Y}}\left[\left(h_{\mathcal{D}}(\vec{x}) - y\right)^2\right]\mathcal{P}(\vec{x}, y)\; dy\, d\vec{x}.$$

Note that $\vec{x}$ and y are random variables, which makes the dataset $\mathcal{D}$ also a random variable since it's the set of n random variables. The regressor $h_{\mathcal{D}} : \mathcal{X} \to \mathcal{Y}$ is a function of the random variable $\mathcal{D}$ and so is itself a random variable. Hence, we take the expectation and calculate the expected regressor:

$$\overline{h} = \mathbb{E}_{\mathcal{D}\sim\mathcal{P}^n}\left[\mathcal{A}(\mathcal{D})\right] = \int_{\mathcal{D}} h_{\mathcal{D}}\, \mathcal{P}(\mathcal{D})\, d\mathcal{D}.$$

$\forall\, \vec{x} \in \mathcal{X}$, the output of the expected regressor $\overline{h}$ is the average of the infinitely many regressors $h_{\mathcal{D}}$ corresponding to infinitely many training datasets $\mathcal{D}$.

Earlier we said that the generalization error is the expected error of one particular classifier/regressor h. But since h is a random variable, we can actually compute the expected error of the machine learning algorithm $\mathcal{A}$, which is the procedure to come up with the regressors h:

$$\ell_{\mathcal{A}} = \mathbb{E}_{\mathcal{D}\sim\mathcal{P}^n,\ (\vec{x},y)\sim\mathcal{P}}\left[\left(h_{\mathcal{D}}(\vec{x}) - y\right)^2\right] = \int_{\mathcal{D}}\int_{\vec{x}\in\mathcal{X}}\int_{y\in\mathcal{Y}}\left(h_{\mathcal{D}}(\vec{x}) - y\right)^2\, \mathcal{P}(\vec{x}, y)\, \mathcal{P}(\mathcal{D})\, dy\, d\vec{x}\, d\mathcal{D}.$$

We are going to decompose the expression for $\ell_{\mathcal{A}}$ to see where does the error actually come from. The first trick is to add and subtract $\overline{h}(\vec{x})$ within the squared loss expression:

$$\begin{aligned}
\ell_{\mathcal{A}} &= \mathbb{E}_{\mathcal{D},(\vec{x},y)}\left[\left(h_{\mathcal{D}}(\vec{x}) - y\right)^2\right] \\
&= \mathbb{E}_{\mathcal{D},(\vec{x},y)}\left[\Big(h_{\mathcal{D}}(\vec{x}) - \overline{h}(\vec{x}) + \overline{h}(\vec{x}) - y\Big)^2\right] \\
&= \mathbb{E}_{\mathcal{D},(\vec{x},y)}\left[\Big(h_{\mathcal{D}}(\vec{x}) - \overline{h}(\vec{x})\Big)^2 + \Big(\overline{h}(\vec{x}) - y\Big)^2 + 2\Big(h_{\mathcal{D}}(\vec{x}) - \overline{h}(\vec{x})\Big)\Big(\overline{h}(\vec{x}) - y\Big)\right].
\end{aligned}$$

Expanding by linearity of expectation,

$$\ell_{\mathcal{A}} = \mathbb{E}_{\mathcal{D},(\vec{x},y)}\left[\Big(h_{\mathcal{D}}(\vec{x}) - \overline{h}(\vec{x})\Big)^2\right] + \mathbb{E}_{\mathcal{D},(\vec{x},y)}\left[\Big(\overline{h}(\vec{x}) - y\Big)^2\right] + 2\,\mathbb{E}_{\mathcal{D},(\vec{x},y)}\Big[\Big(h_{\mathcal{D}}(\vec{x}) - \overline{h}(\vec{x})\Big)\Big(\overline{h}(\vec{x}) - y\Big)\Big].$$

Simplify the third term using independence of $\mathcal{D}$ and $(\vec{x}, y)$:

$$\begin{aligned}
&\mathbb{E}_{\mathcal{D},(\vec{x},y)}\Big[\Big(h_{\mathcal{D}}(\vec{x}) - \overline{h}(\vec{x})\Big)\Big(\overline{h}(\vec{x}) - y\Big)\Big] \\
&= \mathbb{E}_{(\vec{x},y)}\Big[\Big(\mathbb{E}_{\mathcal{D}}\Big[h_{\mathcal{D}}(\vec{x}) - \overline{h}(\vec{x})\Big]\Big)\Big(\overline{h}(\vec{x}) - y\Big)\Big] \\
&= \mathbb{E}_{(\vec{x},y)}\Big[\Big(\mathbb{E}_{\mathcal{D}}\Big[h_{\mathcal{D}}(\vec{x})\Big] - \overline{h}(\vec{x})\Big)\Big(\overline{h}(\vec{x}) - y\Big)\Big] \\
&= \mathbb{E}_{(\vec{x},y)}\Big[\Big(\overline{h}(\vec{x}) - \overline{h}(\vec{x})\Big)\Big(\overline{h}(\vec{x}) - y\Big)\Big] \\
&= \mathbb{E}_{(\vec{x},y)}\Big[0\,\Big(\overline{h}(\vec{x}) - y\Big)\Big] \\
&= 0.
\end{aligned}$$

Hence, $\ell_{\mathcal{A}}$ becomes:

$$\ell_{\mathcal{A}} = \mathbb{E}_{\mathcal{D},(\vec{x},y)}\Big[\Big(h_{\mathcal{D}}(\vec{x}) - \overline{h}(\vec{x})\Big)^2\Big] + \mathbb{E}_{\mathcal{D},(\vec{x},y)}\Big[\Big(\overline{h}(\vec{x}) - y\Big)^2\Big].$$

Decompose the second term using the same tricks as before:

$$\begin{aligned}
&\mathbb{E}_{\mathcal{D},(\vec{x},y)}\Big[\Big(\overline{h}(\vec{x}) - y\Big)^2\Big] = \mathbb{E}_{(\vec{x},y)}\Big[\Big(\overline{h}(\vec{x}) - y\Big)^2\Big] \\
&= \mathbb{E}_{(\vec{x},y)}\Big[\Big(\overline{h}(\vec{x}) - \overline{y}(\vec{x}) + \overline{y}(\vec{x}) - y\Big)^2\Big] \\
&= \mathbb{E}_{(\vec{x},y)}\Big[\Big(\overline{h}(\vec{x}) - \overline{y}(\vec{x})\Big)^2\Big] + \mathbb{E}_{(\vec{x},y)}\Big[\Big(\overline{y}(\vec{x}) - y\Big)^2\Big] + 2\,\mathbb{E}_{(\vec{x},y)}\Big[\Big(\overline{h}(\vec{x}) - \overline{y}(\vec{x})\Big)\Big(\overline{y}(\vec{x}) - y\Big)\Big] \\
&= \mathbb{E}_{(\vec{x},y)}\Big[\Big(\overline{h}(\vec{x}) - \overline{y}(\vec{x})\Big)^2\Big] + \mathbb{E}_{(\vec{x},y)}\Big[\Big(\overline{y}(\vec{x}) - y\Big)^2\Big] + 2\,\mathbb{E}_{\vec{x}}\Big[\Big(\overline{h}(\vec{x}) - \overline{y}(\vec{x})\Big)\Big(\mathbb{E}_{y|\vec{x}}\Big[\overline{y}(\vec{x}) - y\Big]\Big)\Big] \\
&= \mathbb{E}_{(\vec{x},y)}\Big[\Big(\overline{h}(\vec{x}) - \overline{y}(\vec{x})\Big)^2\Big] + \mathbb{E}_{(\vec{x},y)}\Big[\Big(\overline{y}(\vec{x}) - y\Big)^2\Big] + 2\,\mathbb{E}_{\vec{x}}\Big[\Big(\overline{h}(\vec{x}) - \overline{y}(\vec{x})\Big)\Big(\overline{y}(\vec{x}) - \mathbb{E}_{y|\vec{x}}\Big[y\Big]\Big)\Big] \\
&= \mathbb{E}_{(\vec{x},y)}\Big[\Big(\overline{h}(\vec{x}) - \overline{y}(\vec{x})\Big)^2\Big] + \mathbb{E}_{(\vec{x},y)}\Big[\Big(\overline{y}(\vec{x}) - y\Big)^2\Big] + 2\,\mathbb{E}_{\vec{x}}\Big[\Big(\overline{h}(\vec{x}) - \overline{y}(\vec{x})\Big)\Big(\overline{y}(\vec{x}) - \overline{y}(\vec{x})\Big)\Big] \\
&= \mathbb{E}_{(\vec{x},y)}\Big[\Big(\overline{h}(\vec{x}) - \overline{y}(\vec{x})\Big)^2\Big] + \mathbb{E}_{(\vec{x},y)}\Big[\Big(\overline{y}(\vec{x}) - y\Big)^2\Big] + 2\,\mathbb{E}_{\vec{x}}\Big[\Big(\overline{h}(\vec{x}) - \overline{y}(\vec{x})\Big)\,0\Big] \\
&= \mathbb{E}_{(\vec{x},y)}\Big[\Big(\overline{h}(\vec{x}) - \overline{y}(\vec{x})\Big)^2\Big] + \mathbb{E}_{(\vec{x},y)}\Big[\Big(\overline{y}(\vec{x}) - y\Big)^2\Big] + 0 \\
&= \mathbb{E}_{(\vec{x},y)}\Big[\Big(\overline{h}(\vec{x}) - \overline{y}(\vec{x})\Big)^2\Big] + \mathbb{E}_{(\vec{x},y)}\Big[\Big(\overline{y}(\vec{x}) - y\Big)^2\Big].
\end{aligned}$$

Therefore, we get $\ell_{\mathcal{A}}$ as:

$$\boxed{\ell_{\mathcal{A}} = \mathbb{E}_{\mathcal{D},(\vec{x},y)}\Big[\Big(h_{\mathcal{D}}(\vec{x}) - \overline{h}(\vec{x})\Big)^2\Big] + \mathbb{E}_{(\vec{x},y)}\Big[\Big(\overline{h}(\vec{x}) - \overline{y}(\vec{x})\Big)^2\Big] + \mathbb{E}_{(\vec{x},y)}\Big[\Big(\overline{y}(\vec{x}) - y\Big)^2\Big].}$$

Therefore, the expected error of a machine learning algorithm $\mathcal{A}$ decomposes into three terms:

- The 1st term is the expectation of the squared difference between the predictions of a particular regressor $h_{\mathcal{D}}$ and those of the mean regressor $\overline{h}$. Hence, by definition, the first term is the **variance in the regressors**. It captures how much the regressor changes if it is trained on a different training set. It also captures the size of the data since if the dataset is very large, the regressors most likely won't vary too much.
- Similarly, the 3rd term is the variance of the data labels, although it's more insightful to think of it in another way. For a data point $(\vec{x}, y)$, the actual label it has is y, but the *expected* label it *should* have according to some distribution $\mathcal{P}(\vec{x}, y)$ we expect it to originate from is $\overline{y}(\vec{x})$. Predicting $\overline{y}(\vec{x})$ is the absolute best we can do, but the actual label is different because in reality the data just doesn't follow a nice, precise distribution like $\mathcal{P}(\vec{x}, y)$. If $y = \overline{y}(\vec{x})$ for all data points $(\vec{x}, y)$ then that third term is 0 and the data does indeed precisely originate from a nice distribution. Otherwise the data is noisy to some degree, and the third term captures the **noise in the data**.
- The 2nd term is the expectation of the squared difference between the prediction of the expected (mean) regressor that $\mathcal{A}$ outputs, and the expected label for that feature vector. In this term,

we forget the noise since we consider the expected label; we take the best possible regressor since we consider the mean of the regressors that $\mathcal{A}$ can output, and we capture the error that remains even under these ideal conditions of the best rergressor and no noise. Hence, this error is an attribute of $\mathcal{A}$ and it arises because $\mathcal{A}$ **biases** its regressors towards a solution other than the expected label. The second term captures the **squared bias of the model**. The bias is in the model/an attribute of $\mathcal{A}$ and no amount of data will convince it otherwise. For instance, if the data is nonlinear, but $\mathcal{A}$ is a linear regression algorithm, then the model is biased towards linear solutions.

Therefore, the generalization error for a machine learning algorithm $\mathcal{A}$ can be written as:

$$\boxed{\ell_{\mathcal{A}} = \text{BIAS}^2 + \text{VARIANCE} + \text{NOISE}}.$$

This is all there is to the error. Hence, to reduce the error of an algorithm, we must trade off between the bias, the variance, and the noise. There's always a trade-off: if one goes down something else goes up, but it needn't go up by the same amount. For instance, if our algorithm has a high bias problem then we must identify it as such correctly, bring the bias down a lot whilst bringing the others up a little. That is the skill as a data scientist.

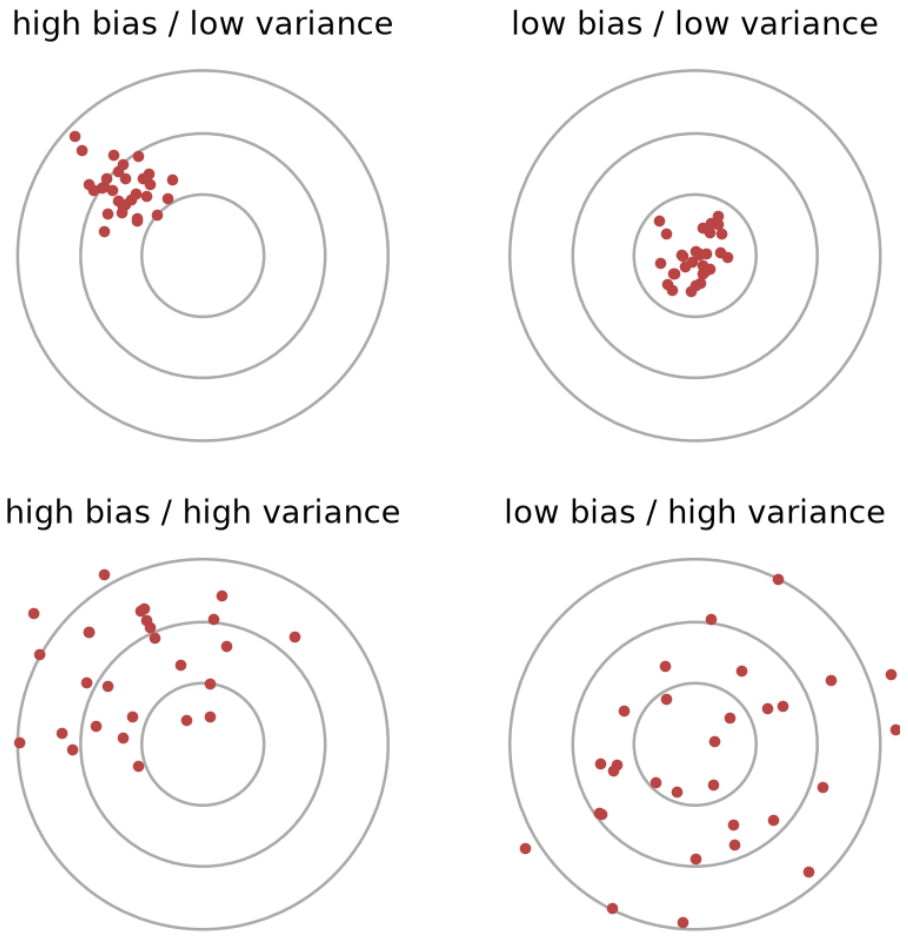

Figure 7: Dartboard analogy for bias and variance.

It's a pessimistic view that noise in the data cannot be reduced. A data scientist can influence all three terms. One noise reduction method is to add more features. However, it is true that the noise

is the stuff that we can never get right. It's what even the Bayes optimal classifier gets wrong with access to $\mathcal{P}(\vec{x}, y)$.

Consider the ERM objective (with ℓ_2-regularization):

$$\min_{\vec{w}} \sum_{(\vec{x},y)\in\mathcal{D}} \ell(\vec{x}, y; \vec{w}) \; + \; \lambda\|\vec{w}\|_2^2.$$

We want to look at the test error and the training error as λ changes. If λ is very small then the optimizer has very little weight on the regularizer and the only thing it's trying to optimize is the loss on the training dataset. Hence for a small λ, the training error gets very low since that's what we're trying to do. However, if λ is really small, then we're focusing too much on the training data set and the test error is high. As λ gets really large, the optimization is dominated by trying to find a simple solution. In the extreme case we get the zero vector, which is as good as random guessing, and the training error is really high. If we return the zero vector then the testing error will also be really large since we return the bogus classifier. Therefore, the training error increases as λ increases, but the test error is roughly convex with a sweet (min) spot somewhere in the middle. The regime towards the left of a low training error but a high test error is called **overfitting**, and the regime towards the right of a high training error and a high test error is called **underfitting**.

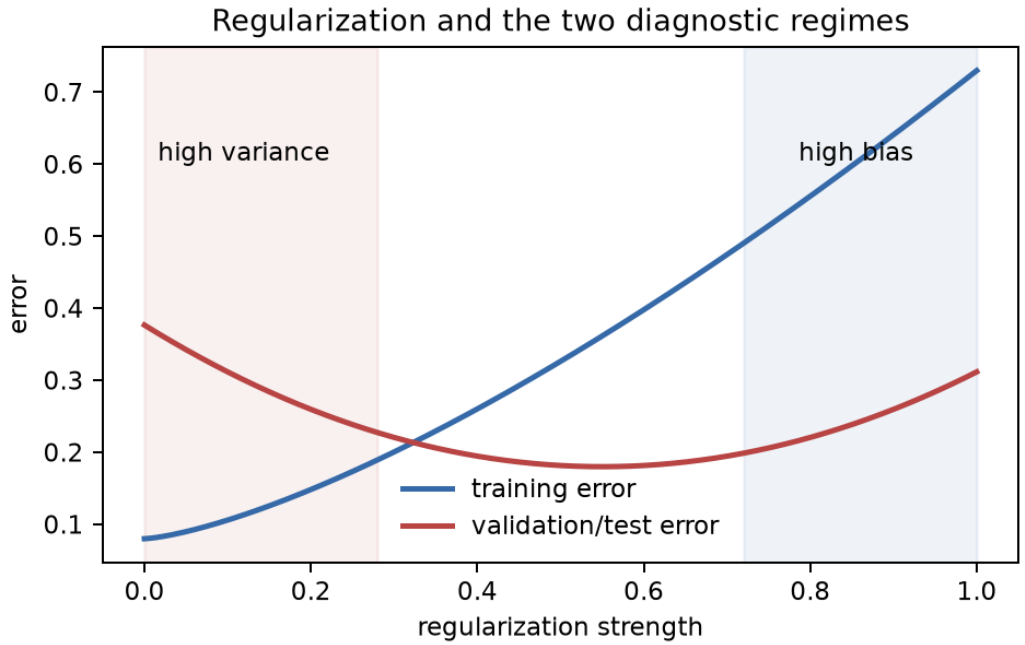

Figure 8: Visualization of training and validation error across regularization regimes.

Overfitting is a **high variance** regime where we're memorizing aspects of the training dataset that are too sepcific (does not generalize), i.e. the classifier does really well on the training data but that doesn't translate to the general setting. Underfitting is a **high bias** regime where the classifier is bad even on the training data. In this example, an over-regularized model is underfitted because it loses it's expressive power to get a even a point it has seen right. λ is the hyperparameter of our model that balances the trade-off between overfitting and underfitting.

How do we find the sweet-spot λ, i.e. how do we optimally *tune* the hyperparameter λ? The simplest things that people use is called **k-fold cross validation**: I take my training dataset, split it up into

training and validation sets, try out a whole bunch of different values of λ (train on the training set and test on the validation), record the error that I get for each one, and pick the λ corresponding to the lowest error. The goal is to try as many different values of λ as possible, but too many values may cause an overhead. To overcome this, one can do a **telescopic search**, which is to zoom in on the optimal value of λ by searching at different orders of magnitude: starting at a high order, finding the optimal at that order, and searching again after lowering the order by one. Another issue we may encounter after trying with many values of λ is that we may overfit on the validation set. Hence, I don't want to pick just one validation set but every point in my dataset should be a validation point. This is where the *k-fold* comes in. I divide my dataset into k different buckets such that $k-1$ of them are used for training and 1 is used for validation. This can be done exactly k times with a different validation set each time. Hence, for every value of λ that I try, I get k validation errors instead of one, and I simply take their mean.

- The larger k the better, but the slower the process. There is a trade-off.
- $k = n \Rightarrow$ leave-one-out cross validation.

One way to regulate overfitting is regularization. Another common one is called **early stopping**. When we try to optimize the parameter vector $\vec{w}$, we don't optimize all the way to the end and instead, well, stop early. For instance, we could stop early doing gradient descent and return a suboptimal solution. There is a deep connection between regularization and early stopping: regularization is the characterization of early stopping in terms of a space constraint, whereas early stopping does it as a time constraint. Also, since we characterized overfitting as a high variance regime, it's important to understand why early stopping (or regularization) reduces the variance. The key is in noting that the loss function that we optimize over is actually a function of the dataset $\ell_{\mathcal{D}}$. Hence, if we optimize all the way to convergence, we are getting to the minimum (the depths) of *that particular* loss for *that particular* dataset, which increases the variance. Analogous to how λ is the parameter for regularization, the number of iterations, M, is the parameter for early stopping. We can do k-fold cross validation with M to find the sweet-spot M. This is much faster than k-fold cross validation with λ since I need to *retrain* the model everytime I test with a different value of λ, whereas I can test with all values of M in one sweep while storing the relevant results as I go.

12.1 ML Debugging

To improve our model i.e. to lower the test error, we must address the right problem: bias, variance, or noise. A way to identify the right problem is to start with a split of the training set such that we're using a single point for training and the remaining $n-1$ points for validation. Hopefully, the training error here is 0, and we note the validation error. We then gradually increase the size of the training set while decreasing the size of the validation set, and note the training and validation errors for each split. Qualitatively, the plot looks like this as the training set size increases:

Suppose our goal is to bring the test error below a threshold, ε. If we have trained on most or all of our dataset and are still stuck towards the left in the plot, we have a high variance problem. If we manage to go all the way to the right in the plot by training on most or all of our dataset and yet get a test error greater than ε, we have a high bias problem. High bias means that even the training error is higher than the desired threshold which means that we have modeled the data less optimally. Hence, adding more training data will **not** solve a high bias problem since the model is inherently wrong. It may reduce test error, but training error is not a deterministic lower bound on test error; the learning-curve diagnosis here is a heuristic rather than a universal rule.

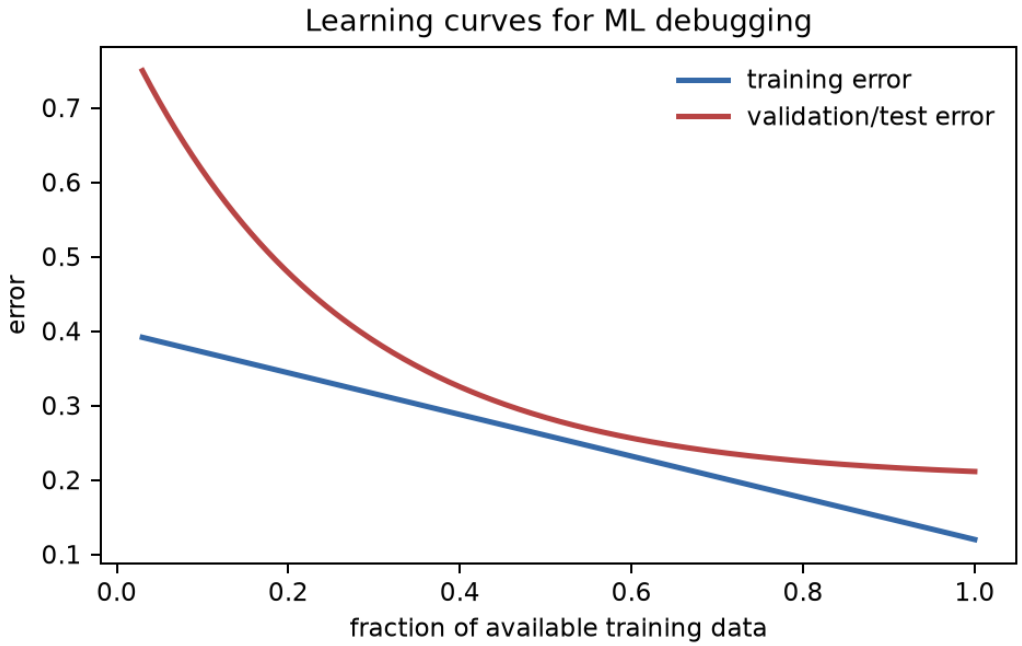

Figure 9: Learning curves for diagnosing bias and variance.

If the problem is high bias, we have to make the model class more complex, i.e. increase the power of the algorithm. A few ways of doing this are:

- Kernelization (stay tuned!).
- Adding features (make the algorithm more expressive).
- Boosting (stay tuned!).
- Decreasing regularization (increase model complexity).

These methods could reduce the training error a lot but the test error only a little (such that it is still above ε) hence creating a big gap between the training and test errors, which is a high variance regime. If this happens, we have traded off bias for variance.

If the problem is high variance, we are in good shape with the training error but we need to bring the test error down; i.e. shrink the gap between the training and test errors. This is exactly what adding more training data will do. It will bring the training error up and the test error down and hence shrink the gap. If in doing so, the training error goes above ε, we are now in a high bias regime and must deal with it in that way. If this happens, we have traded off variance for bias. A few more ways of reducing variance are:

- Bagging (stay tuned!).
- Reducing model complexity.
  - Increasing regularization.
  - Early stopping.

The bias-variance trade-off provides a systematic model to debug and improve machine learning algorithms.

13 Kernels

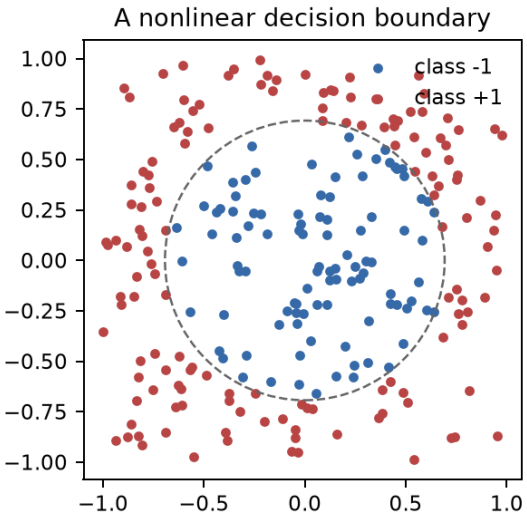

Figure 10: Nonlinear classification dataset motivating feature maps.

Consider a 2D dataset such that the decision boundary is a circle:

A linear classifier like SVM will surely not be able to classifiy this dataset very well since it *biases* the classifiers to find linear decision boundaries. SVMs, here, suffer from high bias. One of the ways to mitigate high bias is to add more features.

Question. Which single feature when added to $\begin{bmatrix} x_1 \\ x_2 \end{bmatrix}$ will make the data linearly separable?

Answer. If we add the squared distance from the origin: $x_1^2 + x_2^2$ as a feature, then the dataset is linearly separable in the 3D feature space.

Alternatively, a feature transformation from Cartesian coordinates to polar coordinates $\begin{bmatrix} r \\ \theta \end{bmatrix}$ will make the data linearly separable in the r-θ space. In this toy example, we just look at the data and we know immediately that the golden feature to add is $x_1^2 + x_2^2$. In fact, we don't even need x_1 and x_2 as separate features then. However, typically, we don't know which features to add to make the classifier more expressive. A good thing to try is to add any possible features we can think of. We can add all possible nonlinear interactions between the different features of our feature vector as separate features to capture nonlinearity in the data through a linear classifier. We can also add

the constant 1 as a bias term because why not. The transformation is: $\begin{bmatrix} x_1 \\ \vdots \\ x_d \end{bmatrix} \mapsto \begin{bmatrix} 1 \\ x_1 \\ \vdots \\ x_d \\ x_1x_2 \\ x_1x_3 \\ \vdots \\ x_{d-1}x_d \\ x_1x_2x_3 \\ \vdots \\ x_1x_2\cdots x_d \end{bmatrix} \in \mathbb{R}^{2^d}$ (2^d = number of possible subsets of d features). Denote this transformation by $\vec{x} \mapsto \phi_0(\vec{x})$. Hence, the transformation $\phi_0 : \mathbb{R}^d \to \mathbb{R}^{2^d}$ maps feature vectors to a higher dimensional space and makes it possible to model nonlinear interactions between the original features using a linear model in the higher dimensional space. This is not surprising because of **Cover's theorem** which says that given a set of feature vectors in $\mathbb{R}^d$, there is a higher chance of the set being linearly separable (i.e. a higher chance of there existing a separating hyperplane) for a larger d. We see that doing a feature transformation to a higher dimension by combining features with each other not only reduces bias since we're adding more features, but also allows us to stick with a simple model like a linear classifier because of Cover's theorem. So this seems like a good approach. However, a transformation like ϕ_0 should scare the living daylights out of you because a feature space could easily be 1000D and 2^{1000} is more than the number of electrons in the universe. To see how to deal with this, we build up some tricks.

Consider the squared loss optimization:

$$\ell(\vec{w}) = \sum_{i=1}^{n} \Big(\vec{w}^\mathsf{T}\vec{x_i} - y_i\Big)^2$$

$$\Rightarrow \vec{\nabla}\ell(\vec{w}) = \frac{\partial \ell}{\partial \vec{w}} = \sum_{i=1}^{n} 2\Big(\vec{w}^\mathsf{T}\vec{x_i} - y_i\Big)\vec{x_i}\,.$$

Claim. $\vec{w}$ can be written as a linear combination of the input feature vectors:

$$\vec{w} = \sum_{i=1}^{n} \eta_i \vec{x_i}$$

for some assignment of η's.

Proof by induction. In our algorithm, we find $\vec{w}$ by doing gradient descent on the squared loss objective, which is convex. Since it is convex, we can start with any initial value like $\vec{w_0}$ and we will reach the minimum. Let's assign $\vec{w_0} = \vec{0}$. The proof works by showing by induction that at every step t, the claim holds true for $\vec{w_t}$ i.e.

$$\vec{w_t} = \sum_{i=1}^{n} \eta_{ti} \vec{x_i}$$

for some assignment of η_t's. If this is shown, then the claim holds true for the final $\vec{w}$ as well and that completes the proof.

- *Base case.* $\vec{w_0} = \vec{0} = \sum_{i=1}^{n} 0\vec{x_i} \Rightarrow$ a good assignment is $\forall i\ \eta_{0i} = 0$.

- *Inductive hypothesis.* Suppose $\vec{w}_t = \sum_{i=1}^{n} \eta_{ti}\vec{x_i}$ at some iteration $t \geq 1$.
- *Inductive step.* By the GD update,

$$\begin{aligned}\vec{w}_{t+1} &\leftarrow \vec{w}_t - \alpha\vec{\nabla}\ell(\vec{w}_t) \\ &= \vec{w}_t - \alpha\sum_{i=1}^{n} 2\Big(\vec{w}_t{}^{\mathsf{T}}\vec{x_i} - y_i\Big)\vec{x_i}\,.\end{aligned}$$

  But $2\Big(\vec{w}_t{}^{\mathsf{T}}\vec{x_i} - y_i\Big)$ is a scalar. Let $\sigma_{ti} = 2\Big(\vec{w}_t{}^{\mathsf{T}}\vec{x_i} - y_i\Big)$. Then,

$$\begin{aligned}\vec{w}_{t+1} &\leftarrow \vec{w}_t - \alpha\sum_{i=1}^{n}\sigma_{ti}\vec{x_i} \\ &= \sum_{i=1}^{n}\eta_{ti}\vec{x_i} - \alpha\sum_{i=1}^{n}\sigma_{ti}\vec{x_i} \\ &= \sum_{i=1}^{n}(\eta_{ti} - \alpha\sigma_{ti})\vec{x_i}\,.\end{aligned}$$

  Hence, we have shown that there exists an assignment $\eta_{(t+1)i} = \eta_{ti} - \alpha\sigma_{ti}$ such that $\vec{w}_{t+1} = \sum_{i=1}^{n}\eta_{(t+1)i}\vec{x_i} \Rightarrow$ the claim holds for $\vec{w}_{t+1}$.

$\therefore$ The final $\vec{w} = \vec{w}^*$ can be expressed as a linear combination of the input feature vectors. $\square$

Corollary. We can modify the gradient descent algorithm by iterating on the coefficients instead of the vector $\vec{w}$:

- Initially, $\forall i\ \eta_i = 0$.
- Compute σ_i per iteration.
- The update step becomes $\eta_i \leftarrow \eta_i - \alpha\sigma_i$.

The above claim and corollary give us the first trick to deal with the 2^dD space. Using a linear classifier in 2^dD space $\Rightarrow$ computing a $2^d D$ $\vec{w}$ through GD using new feature vectors $\vec{x_i} \leftarrow \phi_0(\vec{x_i})$. This would take up all the disks on the planet including those in North Korea. But the claim and corollary save us from having to do this. Instead, we can store n numbers, which is in fact independent of the dimensionality of the data. The claim allows us to do GD, store $\vec{w}$, etc. in a way that is independent of the dimensionality of the data $\Rightarrow$ we can scale up to as high as we want. With the first trick, we can easily do the training without ever having to explicitly compute $\vec{w}$. Once we have $\vec{w}$, we can do the testing by computing:

$$h(\vec{x}) = \vec{w}^{\mathsf{T}}\phi_0(\vec{x}) = \sum_{i=1}^{n}\eta_i\phi_0(\vec{x_i})^{\mathsf{T}}\phi_0(\vec{x})\,.$$

Hence, we never need $\vec{w}$ during testing as well.

However, we still have to deal with ridiculously high dimensional vectors apart from $\vec{w}$ viz. the input feature vectors $\phi_0(\vec{x_i})$'s during training, and an additional feature vector $\phi_0(\vec{x})$ during testing.

- During training, the issue is in the GD algorithm. In the description of the modified GD algorithm (GD in terms of the coefficients), the issue is hidden within "compute σ_i." As part of computing σ_i, we need to compute

$$\vec{w}^{\mathsf{T}}\phi_0(\vec{x_i}) = \left(\sum_{j=1}^{n}\eta_j\phi_0(\vec{x_j})\right)^{\mathsf{T}}\phi_0(\vec{x_i}) = \left(\sum_{j=1}^{n}\eta_j\phi_0(\vec{x_j})^{\mathsf{T}}\right)\phi_0(\vec{x_i}) = \sum_{j=1}^{n}\eta_j\phi_0(\vec{x_j})^{\mathsf{T}}\phi_0(\vec{x_i})\,.$$

This involves computing inner products of ridiculously high dimensional vectors.

- During testing, we need to compute:

$$h(\vec{x}) = \vec{w}^\mathsf{T}\phi_0(\vec{x}) = \sum_{i=1}^{n} \eta_i \phi_0(\vec{x_i})^\mathsf{T}\phi_0(\vec{x})\,,$$

  which again involves inner products in a ridiculously high dimension.

The silver lining here is that all the input vectors and the test vector are only ever accessed in an inner product and never individually or explicitly. This is good because the inner products are just numbers, and we can store them very easily. The second trick is to store the pre-computed inner products of all the pairs of the input feature vectors in a matrix K such that $K_{ij} = \phi_0(\vec{x_i})^\mathsf{T}\phi_0(\vec{x_j})$. This will enable us to simply lookup the numbers during training instead of computing them synchronously. Hence, during training, we do:

$$\vec{w}^\mathsf{T}\vec{x_i} = \sum_{j=1}^{n} \eta_j K_{ij}\,.$$

However, there is still the problem of actually computing the inner products albeit once and we haven't yet addressed the issue during testing, which also boils down to actually computing inner products. Hence, the one issue we have remaining is how do we actually compute these ridiculously high dimensional inner products?

For this we have a third and final trick. Let $\vec{v}$ and $\vec{z}$ by any two vectors in $\mathbb{R}^d$. Then, the problem is to compute $\phi_0(\vec{v})^\mathsf{T}\phi_0(\vec{z})$:

$$\begin{aligned}
\phi_0(\vec{v})^\mathsf{T}\phi_0(\vec{z}) &= \begin{bmatrix} 1 \\ v_1 \\ \vdots \\ v_d \\ v_1v_2 \\ v_1v_3 \\ \vdots \\ v_{d-1}v_d \\ v_1v_2v_3 \\ \vdots \\ v_1v_2\cdots v_d \end{bmatrix}^\mathsf{T} \begin{bmatrix} 1 \\ z_1 \\ \vdots \\ z_d \\ z_1z_2 \\ z_1z_3 \\ \vdots \\ z_{d-1}z_d \\ z_1z_2z_3 \\ \vdots \\ z_1z_2\cdots z_d \end{bmatrix} \\
&= \underbrace{1 + v_1z_1 + v_2z_2 + \ldots + v_dz_d + v_1v_2z_1z_2 + \ldots + v_1\cdots v_dz_1\cdots z_d}_{2^d \text{ terms}} \qquad \text{(takes 10 billion years)} \\
&= \prod_{i=1}^{d}(1 + v_iz_i) \qquad \text{(takes 10 milliseconds)}
\end{aligned}$$

Therefore, we have shown that we can take our data, map it into an exponentially high dimensional space and train and test a linear classifier without once having to compute a single vector in this scare-you-out-of-your-pants space. We only ever need to compute inner products which we can do very cheaply. We've shown this for the squared loss but it holds for any other loss as well. As a result, we're taking simple linear classifiers and making them extremely powerful since the mapping

ϕ_0 captures nonlinear interactions within the features $\Rightarrow$ a linear separation in the high dimensional space may translate to a nonlinear decision boundary in the original space. In fact, ϕ_0 is just one possible mapping. We can use any general mapping ϕ into an arbitrarily high dimensional space (even ∞D) to crank up the power. The tricks remain the same:

1. We never compute $\vec{w}$ but instead express it as a linear combination of the n input feature vectors and work with the n coefficients throughout the training and testing.
2. We, in fact, never compute a *single* vector in the humongous dimensional space. All we need is an inner product function $k : \mathbb{R}^d \times \mathbb{R}^d \to \mathbb{R},\ k(\vec{v}, \vec{z}) = \phi(\vec{v})^\mathsf{T}\phi(\vec{z})$ such that it computes inner products cheaply.
3. We compute inner products for all pairs of the input feature vectors once, store them in a matrix K such that $K_{ij} = k(\vec{x_i}, \vec{x_j})$ beforehand, and simply look them up during training. During testing, we use k to compute inner products with unseen vectors.

It is clear from this that we don't even need to know what the mapping ϕ looks like! The cheaply computable inner product function k is sufficient information to carry out everything. For instance, I could've told you just the inner product function $k_0(\vec{v}, \vec{z}) = \prod_{i=1}^{d}(1 + v_i z_i)$ and you could've done the whole deal without needing to know anything about the mapping ϕ_0 corresponding to k_0!

The function k is called a **kernel function** because by using it, we're mapping our data in a reproducing kernel Hilbert space (RKHS). The matrix K is called a **kernel matrix**. This entire ordeal of using a linear classifier in a higher dimensional space to obtain nonlinear decision boundaries is called **kernelization** or **the kernel trick**.

The natural question is whether there are kernel functions other than k_0 that do a good job. Turns out there are tons and tons of them and we can even make our own ones. A few examples are:

- **Linear kernel:** $k(\vec{v}, \vec{z}) = \vec{v}^\mathsf{T}\vec{z}$.
  - If we use a linear kernel, we're not actually changing the dimensionality at all. The corresponding mapping is $\phi_I : \mathbb{R}^d \to \mathbb{R}^d,\ \phi_I(\vec{x}) = \vec{x}$. Hence, the algorithm is just the good old linear classifier.
  - Although, it would actually be beneficial to use the linear kernel in the case where $d > n$.
- **Polynomial kernel:** $k(\vec{v}, \vec{z}) = (1 + \vec{v}^\mathsf{T}\vec{z})^p$.
  - Using this kernel, we can obtain polynomial decision boundaries up to a degree p. For instance, if $p = 2$, we can obtain quadratic and linear decision boundaries.
  - If $p = 1$, this reduces to the linear kernel since constants don't matter.
- **Radial Basis Function (RBF) kernel:** $k(\vec{v}, \vec{z}) = e^{-\frac{\|\vec{x}-\vec{z}\|^2}{\sigma^2}}$.
  - This is the most popular kernel function: the Brad Pitt of kernels. It's the cousin of the Gaussian distribution.
  - It can be proved that the RBF kernel is a universal approximator i.e. we can fit any function arbitrarily closely given a few assumptions. On compact domains, its RKHS is dense in suitable spaces of continuous functions. This is an approximation statement, not a claim that every finite-sample problem becomes learnable or generalizes without assumptions.
  - The RBF kernel corresponds to a space that is infinite dimensional so we can't write down $\phi(\vec{x})$.

$\star$ A real-valued k is a valid kernel function if it is symmetric and every Gram matrix obtained by applying k to all pairs in any finite set is positive semidefinite.
Proof sketch. If K is a kernel matrix for the set $S = \{\vec{x_i}\}_{i=1}^n$, then $K_{ij} = k(\vec{x_i}, \vec{x_j}) = \phi(\vec{x_i})^\mathsf{T}\phi(\vec{x_j})$ for some mapping ϕ (by definition of the kernel matrix and kernel function). Define matrix $\Phi = \begin{bmatrix}\phi(\vec{x_1}) & \phi(\vec{x_2}) & \cdots & \phi(\vec{x_n})\end{bmatrix}$. Then, $K = \Phi^\mathsf{T}\Phi$. But K is symmetric $\Rightarrow K = Q\Lambda Q^\mathsf{T}$ by the spectral theorem $\Rightarrow \Phi^\mathsf{T}\Phi = Q\Lambda Q^\mathsf{T}$. This is possible if and only if we can take the $\frac{1}{2}$-power of Λ such that $Q\Lambda Q^\mathsf{T} = \left(\Lambda^{\frac{1}{2}}Q^\mathsf{T}\right)^\mathsf{T}\left(\Lambda^{\frac{1}{2}}Q^\mathsf{T}\right) = \Phi^\mathsf{T}\Phi$. But to be able to take the $\frac{1}{2}$-power of Λ, we must be able to take the square roots of all eigenvalues $\Rightarrow$ all $\lambda \geq 0 \Rightarrow K$ is positive semidefinite.

Here is an example of using the RBF kernel with the ridge regression algorithm of obtaining a linear classifier. Here, σ (variance in the RBF kernel) and λ (amount of ℓ_2-regularization) are the hyperparameters of the model:

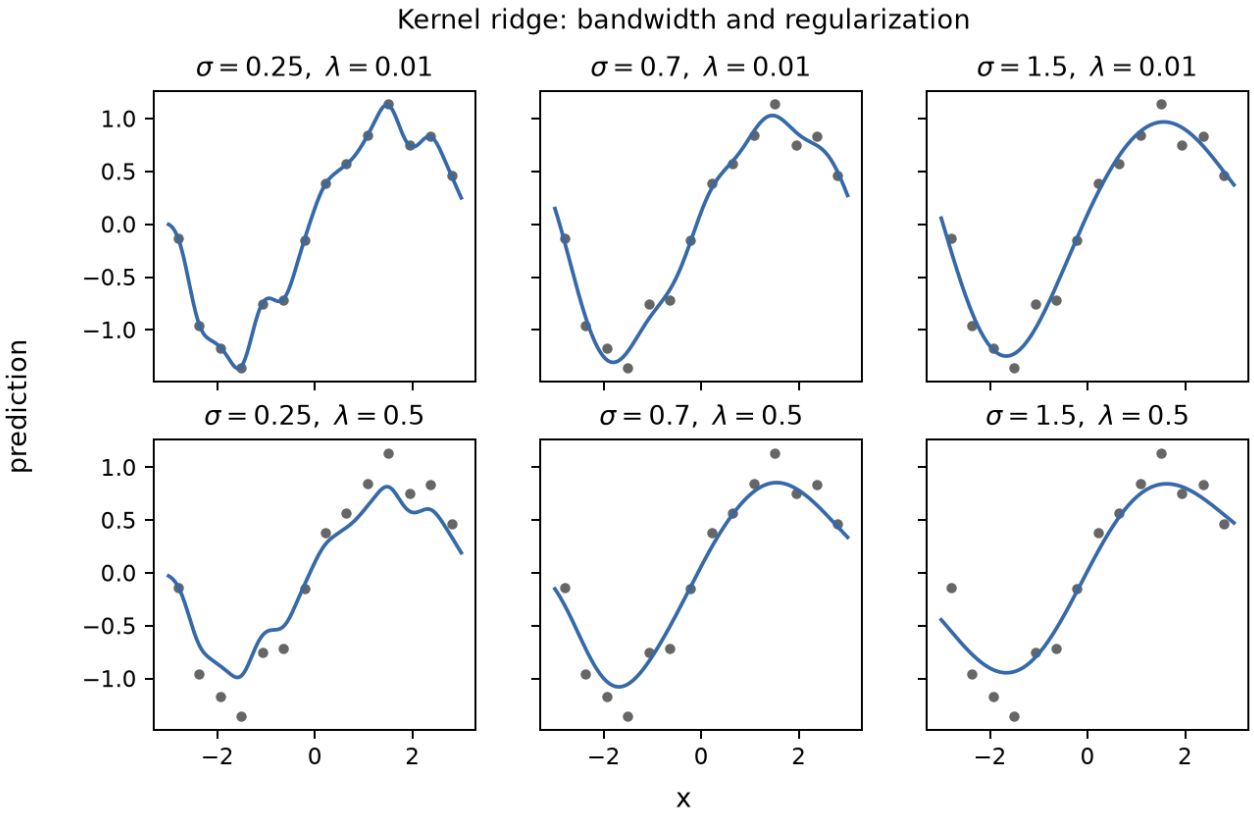

Figure 11: Kernel-ridge fits across bandwidth and regularization choices.

Rules to construct kernels:

1. The linear kernel, $k(\vec{v}, \vec{z}) = \vec{v}^\mathsf{T}\vec{z}$, is a kernel.
2. Multiplying a kernel function by a positive constant gives a kernel.
3. The sum of two kernel functions is a kernel.
4. Any polynomial with positive coefficients composed of a kernel function is a kernel. If $p(x)$ is a polynomial with positive coefficients and $k(\vec{v}, \vec{z})$ is a kernel, then $(p \circ k)(\vec{v}, \vec{z})$ is a kernel.
5. The product of two kernel functions is a kernel.

6. If $f : \mathbb{R}^d \to \mathbb{R}^m$ is any function and $k(\vec{v}, \vec{z})$ is a kernel function, then $\widetilde{k}(\vec{v}, \vec{z}) = f(\vec{v})^\mathsf{T} f(\vec{z}) k(\vec{v}, \vec{z})$ is a kernel.
7. Exponentiating a kernel function gives a kernel.
8. If A is any positive semidefinite matrix, then $k(\vec{v}, \vec{z}) = \vec{v}^\mathsf{T} A \vec{z}$ is a kernel.

For example, the RBF function is a kernel by rules (1), (2), (7), and (6). Another kernel we could have is a *set kernel*, which is useful when the dataset is in terms of feature sets instead of vectors. For finite sets S and T, $k(S, T) = e^{|S \cap T|}$ is a kernel, since if S and T are encoded as one-hot vectors $\vec{v}$ and $\vec{z}$ respectively, then $k(\vec{v}, \vec{z}) = e^{\vec{v}^\mathsf{T} \vec{z}}$.

13.1 Kernel KNN

The way to kernelize any machine learning algorithm is to identify the places where the data is accessed as inner products, or force access only as inner products, and simply swap in the kernel function (or the kernel matrix lookup) in there. For the KNN algorithm, we need to find the nearest points according to the Euclidean distance: $d(\vec{x_1}, \vec{x_2}) = \|\vec{x_1} - \vec{x_2}\|_2$. The nearest neighbors will have the least Euclidean distance i.e. also the least squared Euclidean distance: $d^2(\vec{x_1}, \vec{x_2}) = \|\vec{x_1} - \vec{x_2}\|_2^2 = \vec{x_1}^\mathsf{T} \vec{x_1} - 2\vec{x_1}^\mathsf{T} \vec{x_2} + \vec{x_2}^\mathsf{T} \vec{x_2} = k(x_1, x_1) - 2k(x_1, x_2) + k(x_2, x_2)$. And hallelujah, KNN is kernelized because that's all we need.

However, kernel KNN doesn't make sense because KNN is already a nonlinear algorithm and high bias is not one of its problems. KNN has a high variance problem but kernelization reduces bias.

13.2 Kernel Regression

Let $\Phi = \begin{bmatrix} \phi(\vec{x_1}) & \cdots & \phi(\vec{x_n}) \end{bmatrix}$, so Φ has one transformed training point per column, and let $K = \Phi^\mathsf{T} \Phi \in \mathbb{R}^{n \times n}$. Kernel ridge regression minimizes

$$\|\Phi^\mathsf{T} \vec{w} - \vec{y}\|_2^2 + \rho \|\vec{w}\|_2^2,$$

with $\rho > 0$. By the representer theorem, $\vec{w} = \Phi \vec{\eta}$ for some $\vec{\eta} \in \mathbb{R}^n$. Substitution gives the correctly dimensioned solution

$$\boxed{\vec{\eta} = (K + \rho I_n)^{-1} \vec{y}}.$$

For a test point $\vec{x}$, define $\vec{k}(\vec{x}) \in \mathbb{R}^n$ by $k_i(\vec{x}) = k(\vec{x_i}, \vec{x})$. Then

$$\boxed{h(\vec{x}) = \vec{k}(\vec{x})^\mathsf{T} (K + \rho I_n)^{-1} \vec{y} = \vec{k}(\vec{x})^\mathsf{T} \vec{\eta}}.$$

The unregularized expression with K^{-1} is valid only when K is invertible.

Question. Is kernel regression parametric or nonparametric?

A **parametric** algorithm is one that has a fixed number/size of parameters that we learn during training and apply during testing. The number of parameters doesn't change even if we change the amount of training data, i.e. we don't have to learn more parameters if we give it more data. On the other hand, a **nonparametric** algorithm is where the model size grows as we give it more training data. E.g.: KNN since the data is also the model.

Answer. Kernel regression in its usual dual form is nonparametric since the size of $\vec{\eta}$ grows with the training-set size n. If a kernel has an explicit fixed finite-dimensional feature map and one trains

that finite-dimensional primal model, it may instead be treated as parametric. For the RBF kernel, the feature space is infinite-dimensional and the usual estimator is nonparametric. Non-kernelized finite-dimensional linear regression is parametric.

13.3 Kernel SVM

Recall the SVM optimization problem:

$$\begin{aligned}
\text{Objective: } & \min_{\vec{w},b} \ \vec{w}^\mathsf{T}\vec{w} + C\sum_{i=1}^{n}\xi_i\,, \\
\text{Constraint: } & \forall i \ y_i(\vec{w}^\mathsf{T}\vec{x_i} + b) \geq 1 - \xi_i\,, \\
\text{Constraint: } & \forall i \ \xi_i \geq 0\,.
\end{aligned}$$

The objective is a quadratic, convex function of $\vec{w}$. From optimization theory, optimization problems come in pairs of **primal** and **dual problems**. The dual problem is basically the reverse of the primal and has the same solution. The dual problem for the primal optimization problem of the SVM is a maximization problem whose solution conincides with its primal counterpart, $\vec{w}^*$. Optimization theory gives the methods to derive the dual problems from the primal problems. For the SVM, the dual problem reveals the purely inner product accesses required for kernelization, and we can simply optimize that for kernel SVM. The dual problem looks like:

$$\begin{aligned}
\text{Objective: } & \min_{\alpha_1,\ldots,\alpha_n} \ \frac{1}{2}\sum_{i=1}^{n}\sum_{j=1}^{n}\alpha_i\alpha_j y_i y_j \vec{x_i}^\mathsf{T}\vec{x_j} \ - \ \sum_{i=1}^{n}\alpha_i\,, \\
\text{Constraint: } & \forall i \ 0 \leq \alpha_i \leq C\,, \\
\text{Constraint: } & \sum_{i=1}^{n}\alpha_i y_i = 0\,,
\end{aligned}$$

where $\vec{x_i}^\mathsf{T}\vec{x_j}$ in the objective can be replaced with K_{ij}. Here, the α_i's are some n weights such that $\vec{w} = \sum_{i=1}^{n}\alpha_i y_i \vec{x_i} = X(\vec{\alpha}\odot\vec{y})$. Hence, $\vec{w} = X\vec{\eta}$, where $\vec{\eta} = \vec{\alpha}\odot\vec{y}$. Of course, we will never compute $\vec{w}$ in kernel SVM. Solving the dual problem is just as complex as solving the primal problem. It turns out that when we solve the dual problem, almost all α_i's are 0's, and only the support vectors have nonzero weights; i.e. $\alpha_i \geq 0 \Leftrightarrow y_i(\vec{w}^\mathsf{T}\vec{x_i} + b) = 1$. From this, we can calculate b since $y_i\vec{w}^\mathsf{T}\vec{x_i} = y_i\vec{\eta}^\mathsf{T}X^\mathsf{T}\vec{x_i}$, and $X^\mathsf{T}\vec{x_i}$ is just the i^{th} column of the kernel matrix $K \Rightarrow y_i\vec{\eta}^\mathsf{T}K_i + y_i b = 1 \Rightarrow b = \frac{1}{y_i} - \vec{\eta}^\mathsf{T}K_i = y_i - \vec{\eta}^\mathsf{T}K_i$ ($\because \mathcal{Y} = \{+1, -1\}$).

Although we never compute $\vec{w}$ in kernel SVM, we need b for inference. During inference, we compute $h(\vec{x}) = \vec{w}^\mathsf{T}\vec{x} + b = \vec{\eta}^\mathsf{T}X^\mathsf{T}\vec{x} + b$, where $X^\mathsf{T}\vec{x}$ can be easily computed using the kernel function, and we've already computed b. I've abused notation here: $h(\vec{x})$ and b should be written in terms of $\phi(\cdot)$ and Φ when they are computed in the higher dimension. Hence in kernel SVM, $b = y_i - \vec{\eta}^\mathsf{T}\vec{K_i}$ where $\vec{K_i} = \Phi^\mathsf{T}\phi(\vec{x_i})$ such that $K_{ir} = k(\vec{x_r}, \vec{x_i})$, and $h(\vec{x}) = \vec{\eta}^\mathsf{T}\Phi^\mathsf{T}\phi(\vec{x}) + b = \sum_{i=1}^{n}\alpha_i y_i \phi(\vec{x_i})^\mathsf{T}\phi(\vec{x}) \ + \ b = \sum_{i=1}^{n}\alpha_i y_i k(\vec{x_i}, \vec{x}) \ + \ b$.

13.4 KNN vs. Kernel SVM

In the KNN classification algorithm, we predict the label of a test point as:

$$h_{\text{KNN}}(\vec{x}) = \text{sign}\left(\sum_{i=1}^{n} y_i \, \mathbb{I}(\vec{x_i} \in S_{\text{KNN}}(\vec{x}))\right),$$

assuming $\mathcal{Y} = \{+1, -1\}$. In kernel SVM, we do inference as:

$$h_{\text{SVM}}(\vec{x}) = \text{sign}\left(\sum_{i=1}^{n} y_i \alpha_i k(\vec{x_i}, \vec{x}) \ + \ b\right).$$

Suppose we use the RBF kernel. Then,

$$h_{\text{SVM}}(\vec{x}) = \text{sign}\left(\sum_{i=1}^{n} y_i \alpha_i e^{-\frac{\|\vec{x_i} - \vec{x}\|^2}{\sigma^2}} \ + \ b\right).$$

Are you watching closely? If we replace $\mathbb{I}(\vec{x_i} \in S_{\text{KNN}}(\vec{x}))$ in h_{KNN} with $\alpha_i e^{-\frac{\|\vec{x_i} - \vec{x}\|^2}{\sigma^2}}$ then we get h_{SVM} (ignoring the bias). Hence, KNN and kernel SVM are very similar. In KNN, we do inference using a fixed number, k, of nearest neighbors (the k nearest neighbors are assigned 1 and the rest 0 by the indicator variable). In kernel SVM with the RBF kernel, the term that replaces the indicator variable, $\alpha_i e^{-\frac{\|\vec{x_i} - \vec{x}\|^2}{\sigma^2}}$, draws a Gaussian around the test point and assigns weights accordingly. The neighbors $\vec{x_i}$ that are far from $\vec{x}$ have a large $\|\vec{x_i} - \vec{x}\|^2 \Rightarrow$ small $e^{-\frac{\|\vec{x_i} - \vec{x}\|^2}{\sigma^2}} \Rightarrow$ small $\alpha_i e^{-\frac{\|\vec{x_i} - \vec{x}\|^2}{\sigma^2}}$ (since α_i is positive and bounded by C) $\Rightarrow$ small weight. Similarly, the nearest neighbors are assigned the highest weights. Thus, the weights for the neighbors are now on a Gaussian distribution instead of being just 0 or 1 like in KNN. With this difference, the kernel SVM does exactly the same thing as KNN except smarter. SVM thins out the training dataset for inference since we know that $\alpha_i = 0$ if $\vec{x_i}$ is not a support vector. SVM says that we only need the points on the margin for inference and throws out the rest.

The ordinary (non-kernelized) SVM algorithm can be similarly compared to KNN since non-kernelized SVM is identical to kernel SVM with the linear kernel. The linear kernel is straight up the squared Euclidean distance and so it in fact has more in common with KNN. The comparison boils down to 1 and 0 being weights in KNN versus the α_i's in SVM. Therefore, **SVM is a soft nearest neighbors algorithm**.

14 Gaussian Processes

> "... not complicated by perceived as complicated ..."

Gaussian processes are an application of kernels. They are the regression counterpart of SVMs; i.e. just like SVMs are great for classification (simple, fast, and reliable) but awkward for regression, Gaussian processes are great for regression but awkward for classification. Gaussian processes heavily exploit the Gaussian distribution; more precisely, the following properties of Gaussian distributions:

- By the central limit theorem, the average of many random variables is Gaussian. This is one of the reasons why so many things in the real world are Gaussian distributed.
- The other reason why Gaussians are pervasive is that they are the blackhole of distributions:
  - The sum of two (or more) Gaussians is a Gaussian.
  - The product of two (or more) Gaussian densities is proportional to a Gaussian density; it must be normalized to become a probability density.
  - Marginalization of a multidimensional Gaussian gives a Gaussian.

– Gaussians are closed under conditioning. For instance, if I have a 2D Gaussian, then the distribution of one variable *given* a value of the other is still a Gaussian.

The last regression algorithm we saw was ridge regression (kernelized and not) where the assumption was that the distribution for y was Gaussian such that $y_i \sim \mathcal{N}(\vec{w}^\mathsf{T}\vec{x_i}, \sigma^2)$. The two approaches we took to learn $\vec{w}$ were maximum likelihood estimation and maximum *a posteriori* estimation. After we learnt $\vec{w}$ we did inference on a test point by computing $h(\vec{x}) = \vec{w}^\mathsf{T}\vec{x}$. But this seems pretty indirect. Instead of making assumptions based on a model parameter $\vec{w}$, why don't we marginalize $\vec{w}$ out and try to directly model $P(y \mid \vec{x}, \mathcal{D})$ for any test point:

$$P(y \mid \vec{x}, \mathcal{D}) = \int_{\vec{w}} P(y \mid \vec{x}, \vec{w})\, P(\vec{w} \mid \mathcal{D})\, d\vec{w}\,.$$

But $P(\vec{w} \mid \mathcal{D}) = P(\mathcal{D} \mid \vec{w})P(\vec{w})$ times some normalizer. But $P(\vec{w})$ is Gaussian from the prior assumption in MAP (we're in the Bayesian world now) and $P(\mathcal{D} \mid \vec{w})$ is Gaussian from our linear regression assumption $\Rightarrow$ their product is Gaussian $\Rightarrow$ $P(\vec{w} \mid \mathcal{D})$ is Gaussian. Also, $P(y \mid \vec{x}, \vec{w})$ is Gaussian from our assumption that given the model $\vec{w}$, $P(y \mid \vec{x})$ is Gaussian $\Rightarrow$ the integrand is Gaussian $\Rightarrow$ the integral is the marginalization of a Gaussian which is a Gaussian $\Rightarrow$ the LHS is a Gaussian! This is more beautiful than linear regression because now we're not sticking to one model $\vec{w}$, but taking the expectation over all possible models and yet getting a distribution we understand: the Gaussian. Taking the expectation is more powerful by the weak law of large numbers which says that the mean of many value tends to the true value. In other words, we have simply omitted out from our algorithm the need to depend on one particular model. Instead of committing to one line, we are now taking the expectation over all possible lines while making a prediction. The prediction will be heavily influenced by the lines that fit the data versus those that make no sense but we're nevertheless taking the mean over all possible models. This also circumvents a common problem in regression where a regression algorithm spits out a particular label value and we have no idea how much we can trust it or how sure the model is of its answer. However, here, we're computing a full-fledged distribution over all possible labels which allows us to compute confidence intervals as well.

Now that we know that taking the expectation over all possible models gives a Gaussian we can try to directly learn that Gaussian since we know what a Gaussian looks like:

$$P(y \mid \vec{x}, \mathcal{D}) \sim \mathcal{N}(\mu, \sigma^2)$$

where μ and σ are the parameters that we fit. This algorithm is called a **Gaussian process**.

The key assumption in Gaussian process regression (GPR) is not that the inputs are Gaussian. Rather, every finite collection of latent function values is jointly Gaussian; in standard GPR the observations additionally have independent Gaussian noise. Thus we assume:

$$P\left(\underbrace{\begin{bmatrix} y_1 \\ \vdots \\ y_n \\ y \end{bmatrix}}_{\text{training labels + test label}} \middle| \underbrace{\vec{x}}_{\text{test point}}, \underbrace{\begin{bmatrix} \vec{x_1} & \vec{x_2} & \cdots & \vec{x_n} \end{bmatrix}}_{X}\right) \sim \mathcal{N}(\vec{\mu}, \Sigma)\,.$$

This is the assumption we make upfront and get to the same result much quicker.

Inference in Gaussian process regression computes a posterior predictive Gaussian distribution (from which sampling is optional), $\mathcal{N}(\vec{\mu}, \Sigma)$, where μ and Σ are the parameters we come up with during training. To simplify the training, we actually fit $\mathcal{N}(\vec{0}, \Sigma)$ and add the mean $\vec{\mu}$ to it later without loss of generality. Hence, the only parameter we need to learn is Σ which is the **covariance matrix** with dimensions $(n+1) \times (n+1)$. The diagonal elements Σ_{ii} for $1 \leq i \leq n$ are the variances for the training labels, and the element $\Sigma_{(n+1)(n+1)}$ is the variance for the test label. The off-diagonal elements are the covariances i.e. Σ_{ij} with $i \neq j$ is a measure of how correlated y_i is to y_j. It may seem complicated to learn this matrix, but we have a trick up our sleeves. A property of the covariance matrix is that it is positive semidefinite. Does that ring a bell? It is a kernel!! We can simply use a kernel function to fill in Σ. One caveat here is that Σ captures the correlation between the different labels but the kernel function operates on the feature vectors. The idea is that the labels are highly correlated if and only if the corresponding feature vectors are highly correlated, and the same is true for independence. Let $K_{n \times n}$ be the kernel matrix such that $K_{ij} = k(\vec{x_i}, \vec{x_j})$. Then we know that the $n \times n$ block formed by the first n rows and the first n columns in Σ is K. Let $\vec{k_*}$ be the vector such that $k_{*i} = k(\vec{x_i}, \vec{x})$ where $\vec{x}$ is our test point, and let $k_{**} = k(\vec{x}, \vec{x})$. Then, we get Σ as:

$$\Sigma = \begin{bmatrix} K & \vec{k_*} \\ \vec{k_*}^\mathsf{T} & k_{**} \end{bmatrix}.$$

This is the covariance matrix for the distribution $P\left(\begin{bmatrix}\vec{y} \\ y\end{bmatrix} \,\middle|\, \vec{x},\, X\right) = \mathcal{N}(\vec{0}, \Sigma)$. With independent observation noise of variance σ_n^2, let $K_y = K + \sigma_n^2 I_n$. The posterior distribution for the latent value f_* is: $\boxed{P(f_* \mid \mathcal{D}, \vec{x}) = \mathcal{N}\left(\vec{k_*}^\mathsf{T} K_y^{-1} \vec{y},\; k_{**} - \vec{k_*}^\mathsf{T} K_y^{-1} \vec{k_*}\right)}$. For a noisy future observation, add σ_n^2 to the predictive variance.Are you watching closely? The posterior mean has the same algebraic form as kernel ridge regression when its ridge parameter equals the GP noise variance under the matching scaling! So we did all this work to ultimately get back to kernel ridge regression, but we also got the *variance*. Gaussian processes not only pinpointedly predict the label of a test vector $\vec{x}$ as y, but also define a Gaussian around it from which many other things can be calculated (confidence intervals, etc.). Gaussian processes hence also model the uncertainty around the expectation which is extremely useful and important in real world problems. Moreover, the Gaussian process regression algorithm is extremely elegant where we simply compute the mean and the variance with a kernel function and model the estimated Gaussian (1 line of Julia). Under matched kernel, mean function, noise/regularization, and scaling, Gaussian process regression gives the same posterior-mean prediction as kernel ridge regression, the full distribution that it gives alongside is extremely valuable in almost all scenarios.

One great application of Gaussian processes is in function minimization of unknown expensive functions. One example area is hyperparameter search in machine learning. Suppose we're using kernel regression with the RBF kernel and we want to tune the hyperparameters σ and λ. We can come up with a procedure that tells us how good some given values of σ and λ are which entails training the model with the given values and testing it on a validation set. The corresponding validation error is the measure we seek. This is essentially a function f that takes as input σ and λ and gives as output the validation error. This function is unknown, expensive (since each run involves training the model followed by inference), and we want to minimize it to find the best combination of hyperparameters. One widely used way is to conduct a **grid search**: compute $f(\sigma, \lambda)$ for some combinations of σ and λ and pick the combination that gives the lowest value. Another way is called **Bayesian optimization (BO)**, which is where Gaussian process regression comes

in handy. With GPR, we can look at the combinations, of which we are very uncertain, and try out those new values, or we can try out the combinations that have worked well in the past. This is called the **exploration-exploitation trade-off** that we balance while optimizing f with GPR. Since GPR quantifies the uncertainty associated with each prediction, we only explore new values if the uncertainty in their predictions is high enough, otherwise we exploit the values that have worked best so far and dig around them. To sketch out this algorithm: we actually compute f (train the model and so on) for some random values to begin with. This becomes the "training data" for GPR. Using this, GPR fits a Gaussian in the σ-λ space, and makes predictions on test combinations while also providing the associated uncertainties. We locate new points to try out in the space according to the uncertainties, obtain true values by running f on those points, refit the Gaussian, and keep navigating the space till we somewhat confidently attain the best combination (minimum value). This can be more sample-efficient than grid search when evaluations are expensive and the surrogate assumptions are useful, but it is not always better.

15 Decision Trees

> "Decision trees' nightmares are made of spirals."

Recall kD trees all the way from the KNN section. During inference, we do a DFS where the back tracking serves the purpose of finding the true nearest neighbors just in case there are some hiding in the bins other than the first. However, in a classification task, we could trade off accuracy for speed and simply return the label of the nearest neighbor we find in the first leaf (approximate 1NN). Although this approximate nearest neighbor search is less accurate, it is *blazingly fast*. We can make this even faster: we don't even have to compute the nearest neighbor in the first bin if we know that all the points in that bin have the same label. This is just as accurate as the approximate 1NN, but *even* faster, and in fact takes less storage since we needed $O(nd)$ space to store the training dataset in the data structure, but now we only need $O(n)$ space for the data structure since we only store the labels in the leaves. The assumption here is that every bin ends up with points of the same label. This is unlikely so we can tweak it and say that we store the counts of each label in each bin and return the result as probabilities using the discrete uniform law. However, we *can* make each bin end up with points of the same label by splitting the tree enough times for there to be exactly one data point per leaf in the extreme case and stopping early if we see that all leaves contain points having the same label. **This is always possible if we rule out the case that two identical data points (exactly the same features) have different labels.** Adding noise merely makes feature vectors distinct and does not guarantee that a restricted tree representation can isolate every point; it may also change the problem. The danger of splitting so many times that we eventually classify all training points right is that we may overfit; i.e. trees of high depth suffer from high variance and those of low depth have high bias. Hence, the goal of the decision tree algorithm is to find the smallest tree that classifies all the training points correctly. Turns out that this is an NP-hard problem. Decision trees are horrible machine learning algorithms but they have such clear bias and variance problems that we can easily address the issues with bagging (stay tuned!) for variance and boosting (stay tuned!) for bias so that they become amazing classifiers. Search engines are basically boosted (stay tuned!) decision trees.

Instead of being lazy and building up from kD trees, we must address the subtle difference that our goal with decision trees has changed. We now want *pure* leaves where all points in the same leaf have the same label. We need some metric to measure the *impurity* associated with a set of data points w.r.t. the labels, for which we come up with **impurity functions**:

1. **Gini index:** Let $S = \{(\vec{x_i}, y_i)\}_{i=1}^{k}$ be a set of data points. Let Y be the random variable associated with the experiment of picking a random point from S and reporting its label. Let S_y be the subset of S that contains all points in S having the label y. Hence,

$$P_Y(y) = \frac{|S_y|}{|S|}\,.$$

Also, all the S_y for $y \in \mathcal{Y}$ exhaustively cover S and are mutually disjoint i.e. they form a partition of the set S (the relation defined over S such that $p \sim q$ iff p and q have the same labels is an equivalence relation). Based on this, the Gini index of set S is defined as:

$$G(S) = \sum_{y \in \mathcal{Y}} P_Y(y)(1 - P_Y(y))\,.$$

The Gini index for a set has the highest value when all the labels are equally common (most impure) and it has a low value when the set is dominated by one class.

2. **Entropy:** Let S, Y and S_y be the same as defined above. The worst possible probability distribution of Y in a leaf of a decision tree is a uniform distribution $P_Y = \mathcal{U}_Y$. We want to find a different distribution that's as far away as possible from the uniform distribution. One way to measure how different two distributions are is the **Kullback-Leibler (KL) divergence**, which is defined as:

$$D_{\mathrm{KL}}(P \mid\mid Q) = \sum_{y \in \mathcal{Y}} P(y) \log \frac{P(y)}{Q(y)}\,.$$

We use this measure of KL divergence to get as far as possible from the evil $\mathcal{U}_Y$. If $Q = \mathcal{U}_Y$ then we want the P_Y such that their KL divergence is maximum. Let $|\mathcal{Y}| = m$. Then, plugging in $Q = \mathcal{U}_Y = \frac{1}{m}$,

$$\begin{aligned}
D_{\mathrm{KL}}(P_Y \mid\mid \frac{1}{m}) &= \sum_{y \in \mathcal{Y}} P_Y(y) \log \left(m P_Y(y)\right) \\
&= \sum_{y \in \mathcal{Y}} P_Y(y) \log P_Y(y) + P_Y(y) \log m \\
&= \sum_{y \in \mathcal{Y}} P_Y(y) \log P_Y(y) + \sum_{y \in \mathcal{Y}} P_Y(y) \log m \\
&= \sum_{y \in \mathcal{Y}} P_Y(y) \log P_Y(y) + \log m \underbrace{\sum_{y \in \mathcal{Y}} P_Y(y)}_{1} \\
&= \sum_{y \in \mathcal{Y}} P_Y(y) \log P_Y(y) \;+\; \log m\,.
\end{aligned}$$

Maximizing this KL divergence w.r.t. P_Y,

$$\begin{aligned}
P_Y^*(y) &= \arg\max_{P_Y} \sum_{y \in \mathcal{Y}} P_Y(y) \log P_Y(y) \;+\; \log m \\
&= \arg\max_{P_Y} \sum_{y \in \mathcal{Y}} P_Y(y) \log P_Y(y)\,.
\end{aligned}$$

Maximization is for losers. We minimize:

$$\boxed{P_Y^*(y) = \arg\min_{P_Y} \; -\sum_{y \in \mathcal{Y}} P_Y(y) \log P_Y(y)}\,.$$

From this, we define the **entropy** of a set S to be the minimand: $H(S) = -\sum_{y \in \mathcal{Y}} P_Y(y) \log P_Y(y)$. The higher the entropy of a set, the higher the impurity. We want to build a decision tree such that the entropies in the leaves are minimized.

Turns out that entropy is an important concept in information theory, especially compression. We could view all of machine learning as compression. One of the earliest ways to do spam filtering was to compress a bunch of spam emails in a zip file and a bunch of non-spam emails in another zip file. For testing, we add the new email to both the zip files and see how the sizes change. The zip file that grows less is more likely the correct class of the new email since the new email is more similar to the data in that zip file (that's how compression works). The way compression algorithms work is that we have a data stream and the algorithm essentially tries to predict the next data from the past data based on a statistical model. Read the details here. Machine learning is very similar to compression.

We use the entropy as the loss function to construct decision trees. We have a well-formulated goal in hand that we want to minimize the entropy at the leaves. We start with $\mathcal{D}$ and an empty tree. At any stage, the **entropy of the tree** is defined as the weighted average of the entropies of the sets at the leaves weighted by the cardinalities of the sets. Hence, the entropy of the empty tree is $H(\mathcal{D})$. If at the root node, we partition $\mathcal{D}$ into D_{left} and D_{right} then the new entropy of the tree will be:

$$\frac{|D_{\text{left}}|H(D_{\text{left}}) + |D_{\text{right}}|H(D_{\text{right}})}{n}.$$

If we further partition D_{left} and D_{right} into two sets each then the new entropy of the tree would be a weighted average containing four terms. Our goal is to reduce the entropy of the tree with every split such that we ultimately find the tree with the least entropy. Finding a globally optimal decision tree is NP-hard, but the best axis-aligned threshold split at one node can be found by exhaustive search. Instead, we approximate it really well using a greedy algorithm that is trivially simple. The simplification we make is that we only ever split the data along an axis similar to the kD tree. We choose the feature and the threshold such that we achieve the maximum reduction in entropy, and we do this through an exhaustive search. Since we have n data points, we have $n-1$ gaps to split along one dimension $\Rightarrow$ we try out $(n-1)d$ splits in total, and for each split, we compute the resulting entropy of the tree. The computation of the final entropy takes $O(mn)$ time since we take $O(n)$ time to see which point is in which child, $O(m)$ time for the computation of the entropy at each child, and $O(1)$ time to put the entropies togther to form the entropy of the tree. Hence, the algorithm takes $O(mn^2d)$ time to find the best split at a node with n data points. The n^2 factor is concerning since if we double the amount of data, say, then the algorithm is four times slower. But we can be smart and shave off a factor of n while evaluating a split by reusing computation from the previous split. The $O(n)$ time required to check which point is in which child is unnecessary since when we go from one split to another we only update counters (just one counter actually) with how the numbers of positive and negative points in the children have changed as compared to the previous split. This is an $O(1)$ operation $\Rightarrow$ the final time is reduced to $O(mnd)$. We have shoved a $\lg n$ factor that sorting would introduce under the rug. But practically, $\lg n$ is at most 30.

$\star$ Empirically, it does not pay off to consider non-axis-aligned splits since its time cost generally outweighs its benefit, which is anyways limited to pathological cases.

15.1 Regression Trees

Decision trees are commonly called CART (classification and regression trees). For classification, we used entropy as the loss function, or rather as a measure of impurity. For regression, we use the squared loss as the measure for impurity. For a set $S = \{(\vec{x_i}, y_i)\}_{i=1}^k$ of data points, we compute $\mu = \frac{1}{k}\sum_{i=1}^k y_i$ and instead of $H(S)$, we compute the impurity as $L(S) = \frac{1}{k}\sum_{i=1}^k (y_i - \mu)^2$, which is nothing but the variance of the labels in the set S. The regression tree algorithm is identical to the classification tree algorithm except we replace $H(S)$ with $L(S)$ everywhere. Ultimately, we get one value of y per leaf of the tree and we predict that value for every test point that falls into that leaf. Hence, the regression tree algorithm approximates a function using many horizontal lines.

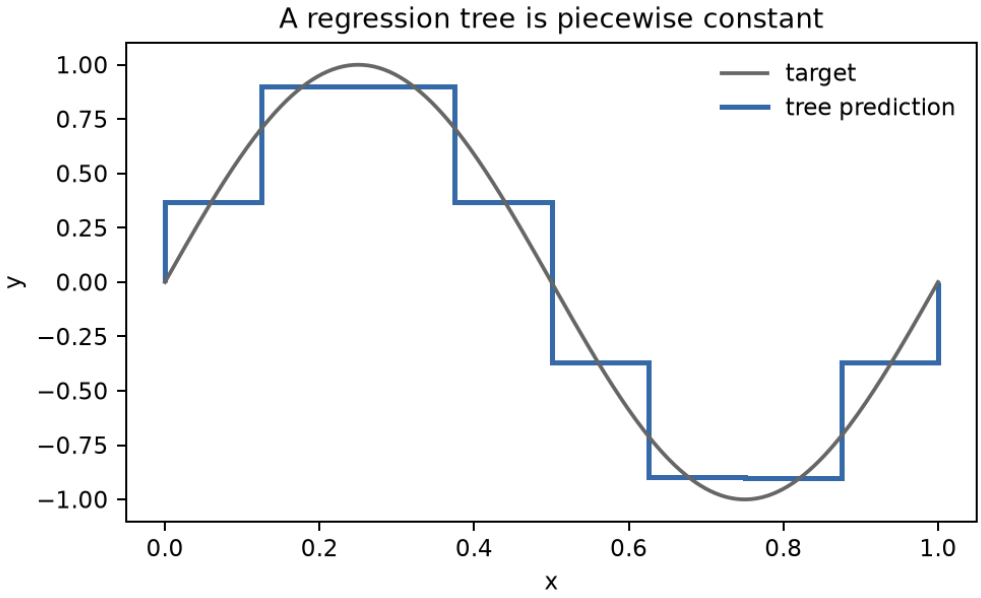

Figure 12: Piecewise-constant regression-tree approximation.

⋆ Unlike classification trees, we don't know how deep to go with regression trees since the labels are continuous values but if we don't stop, we may end up overfitting. So we make a judgment call.

⋆ We can use any loss function instead of the squared loss. Since we're doing an exhaustive search, the loss function may be non-differentiable which tends to be powerful.

15.2 Drawback

Decision trees are a different way of doing machine learning than ERM with linear classifiers. There's a loss function associated with decision trees which they minimize by repeatedly splitting the data instead of gradient descent or convex optimization techniques. Decision trees also have a regularizer which keeps the solution (tree) simple. This regularizer is the depth of the tree, actually the inverse of the depth since deeper trees tend to overfit which is the little regularization (high variance) regime and shallow trees fall in the high regularization, high bias regime. The depth is what we have to balance the bias-variance trade-off in decision trees. In practice, the number of nodes works as a better hyperparameter for regularization but we'll stick with the depth for now. A sufficiently expressive axis-aligned classification tree can drive training error to zero when distinct feature vectors do not carry conflicting labels; depth limits, minimum-leaf constraints, or duplicate conflicting points

can prevent this. But at that extreme the test error is high because we're overfitting. **The problem with decision trees is that it is very hard to find the sweet spot**. In ERM, the trade-off hyperparameter, λ, was a real number but in decision trees that hyperparameter must be an integer which reduces the flexibility of optimizing to the sweet spot. In practice, integer steps turn out to be too coarse-grained while optimizing to the sweet spot which is why decision trees don't work well in practice.

16 Bagging

Bagging is a technique invented at UC Berkeley in the 90's that reduces the variance of a classifier. Recall from the bias-variance trade-off that a classifier having a high variance means that this term in the generalization error is high:

$$\mathbb{E}_{\mathcal{D},(\vec{x},y)}\left[\left(h_{\mathcal{D}}(\vec{x}) - \overline{h}(\vec{x})\right)^2\right].$$

Intuitively, high variance says that our machine learning algorithm $\mathcal{A}$ is such that training with a slightly different dataset may give a very different classifier with very different predictions. This checks out with decision trees because the splits are very data-specific, and training with different data would give different splits along different features that would result in a completely different classifier. Hence, a good thing to do to reduce variance would be to train not on a single dataset $\mathcal{D}$ but multiple datasets $D_1, \ldots, D_m$ and take the the average of the multiple classifiers:

$$h(\vec{x}) = \frac{1}{m}\sum_{i=1}^{m} h_{D_i}(\vec{x}).$$

This is an antidote to high variance because as $m \to \infty$, $h \to \overline{h}$ by the weak law of large numbers. In theory, if we could make m arbitrarily large, we could drive the variance term arbitrarily close to 0. In practice, however, we *don't have* m times more data, but a single dataset $\mathcal{D}$. The solution is to magically invent m different datasets from a single dataset $\mathcal{D}$: we **sample with replacement** data points from $\mathcal{D}$ to fill up m datasets $D_1, \ldots, D_m$ with n data points each. Then we go on to train m different classifiers and our final classifier is their mean. This bootstrapping technique is called **bagging**. Bagging not only reduces variance of the classifier, but its vote fraction can be used as an uncertainty heuristic, although it is not automatically a well-calibrated probability since we can count how many of the m classifiers gave that result which tells us how (un)certain the model is of the prediction.

16.1 Random Forests

The random forest algorithm was invented by the same guy who invented bagging. The algorithm was forgotten in machine learning but it came back big time. Random forests are bagged decision trees with a small modification:

1. We start with the dataset $\mathcal{D}$ and invent m new datasets: $D_1, \ldots, D_m$.
2. On each dataset, we go all in with a decision tree i.e. we keep splitting until the training error is driven to 0. Hence, each h_{D_i} is a decision tree classifier with massive variance, but that's ok because we're bagging.

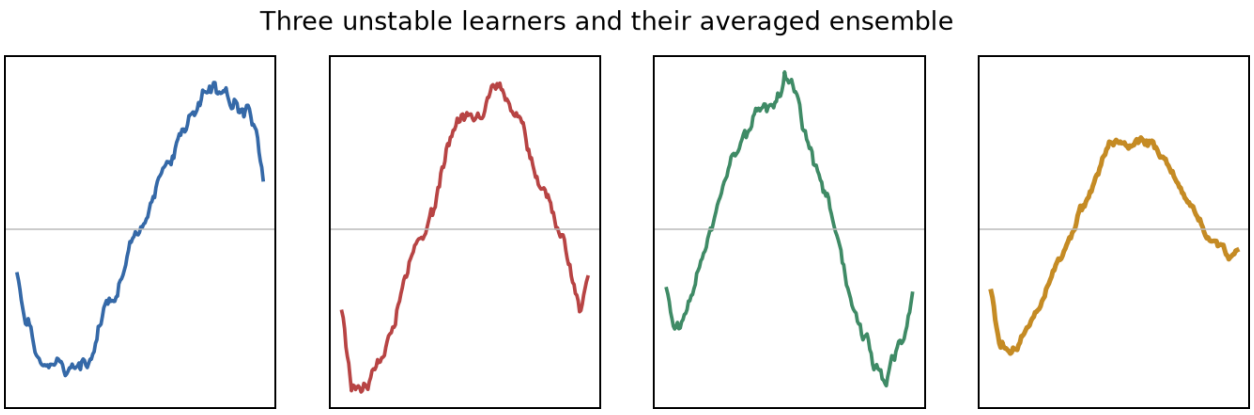

Figure 13: Schematic of bagging unstable learners into an averaged ensemble.

3. ⋆ **Key modification:** While building the decision trees, at each node, instead of searching through all d features for the best split, we randomly sample k out of d features and carry out the exhaustive search for the best split amongst them; i.e. we only split on $k < d$ features where we sample k out of d features every time we split. Instead of introducing k as a separate hyperparameter, we can set $k = \lceil \sqrt{d} \rceil$ for all practical purposes.
4. The final random forest prediction is the average of all classifiers: $h(\vec{x}) = \frac{1}{m} \sum_{i=1}^{m} h_{D_i}(\vec{x})$.

The key modification of randomly sampling $k < d$ features while searching for the best splits amplifies the difference between the m classifiers that we obtain. This is beneficial because very different classifiers make very different mistakes during test time which enable the mistakes to be averaged out when we average the classifiers. This makes random forests extremely effective. Another reason why random forests are awesome is because no preprocessing (scaling, modification, etc.) of raw data is needed before running random forests on it. The only hyperparameter of random forests is m, which we can make arbitrarily large in theory but the pay off plateaus after a point.

⋆ Random forests provide feature-importance scores that can be used as one input to feature selection; importance is not itself feature selection and can be biased or unstable.

⋆ Random forests can handle many irrelevant features better than some methods, but they are not immune to the curse of dimensionality.

16.2 Out of Bag Error

During model selection, we split our training data into a training set and a validation set so that we could train on the training set and estimate the test error by testing on the validation set. The downside of this is that we reduce the amount of data to train with. However, with bagged classifiers, we not only don't reduce the amount of training data but also make it possible to estimate the test error by testing on the *entire* training dataset $\mathcal{D}$. Hence, we get a better estimation at no cost. The idea that makes this possible is that we have multiple classifiers h_{D_j} that are not all trained with a particular test point $(\vec{x_i}, y_i)$ in their training data. In fact we can exhaustively search and pick out the ones that are not trained on that data point. Let $J_i = \{j \mid (\vec{x_i}, y_i) \notin D_j\}$ then define the loss for that data point as:

$$l(\vec{x_i}, y_i) = \frac{1}{|J_i|} \sum_{j \in J_i} \ell(h_{D_j}(\vec{x_i}), y_i),$$

where ℓ is some loss function. Hence, the **out of bag error** for the classifier is:

$$\boxed{\varepsilon_{\text{oob}} = \frac{1}{n}\sum_{i=1}^{n} l(\vec{x_i}, y_i) = \frac{1}{n}\sum_{i=1}^{n} \frac{1}{|J_i|}\sum_{j\in J_i} \ell(h_{D_j}(\vec{x_i}), y_i)}\,.$$

If m is large, then computing J_i for all $1 \leq i \leq n$ may be computationally expensive. In that case, we use the fact that in expectation, about $e^{-1} \approx 36.8\%$ of the bootstrap datasets D_j don't have the point $(\vec{x_i}, y_i)$ in their training dataset. Hence, we use the actual out-of-bag subset J_i for that point, and then sum the losses for all points to obtain the out of bag error. This works really well.

The out of bag error is a useful, often approximately unbiased estimate of test error under common iid sampling conditions and we get it for free i.e. without giving up any training data in the process. We don't have to worry about the nuances associated with a training-validation split. "It doesn't get easier than that."

$\star$ Vanilla training error is not very interesting for bagged algorithms like random forests.

$\star\star$ Bagging primarily reduces variance; its effect on bias depends on the base learner and aggregation and need not be exactly zero.

During training of random forests, we can use the out of bag error to prune the decision trees. If a split does not reduce the out of bag error then we simply prune it out.

17 Boosting

"Random forests is pretty awesome, but there's only one thing that's more awesome."

Boosting, like bagging, assembles a classifier from an ensemble of classifiers. Boosting reduces bias. In the high bias regime, the gap between the training and the test error is small but the training error itself is high. Hence, a high bias machine learning algorithm is a *weak learner* since it gives classifiers that do poorly even on the training data. It does slightly better than a random guesser which is a pretty low bar. Under a suitable weak-learning condition, boosting can assemble weak learners into a strong learner and drive training error toward 0. Like bagging, boosting involves taking the weighted sum of the ensemble of classifiers:

$$H(\vec{x}) = \sum_{i=1}^{m} \alpha_i h_i(\vec{x})\,.$$

Here, H is the strong Frankenstein classifier we're assembling from the ensemble of weak learners comprised of h_i's. In fact, boosting was invented to answer the question whether many weak learners can be combined to form a strong learner. Rob Schapire famously showed that this is possible by inventing the boosting algorithm. A weak learner is a classifier that does slightly better than a random guesser *on the training data*. We develop the intuition behind boosting in the following setup: we have a training dataset $\mathcal{D}$ with n data points, we can define a vector $\vec{y} \in \mathbb{R}^n$ whose components are the n labels. This is the true label vector for the training set. With a classifier h, we can obtain predictions for all the training points and define vector $\vec{y_h} \in \mathbb{R}^n$ whose components are the n predictions. This is the predicted label vector for the training set. For instance, $\frac{1}{n}\|\vec{y_h} - \vec{y}\|_2^2$ is nothing but the squared loss training error $\ell_{\text{sq}}(h)$.

In this setup, the prediction vector of a weak learner h would be extremely far off from the true label vector, but we can be assured that *the dot product between the two will be positive*; i.e. $\vec{y_h}$

and $\vec{y}$ will always lie in the same halfplane of the hyperplane orthogonal to $\vec{y}$. This assurance is given by the weak learner property that although the predictions are lousy, they are not completely orthogonal, hence weak. Also, if the weak learner comes up with a $\vec{y_h}$ whose dot product with $\vec{y}$ is negative, we can detect this and flip the predictions in $\vec{y_h}$ so that we end up in the same halfplane as $\vec{y}$. We want to be in the same halfplane as $\vec{y}$ because then we roughly know where to go to get to $\vec{y}$, which is what a strong learner would predict. Since a weak learner only provides a rough idea of the right direction, we don't trust it completely and take a small step in the direction of $\vec{y_h}$ and as a result get very slightly closer to $\vec{y}$. From our new point, we again point ourselves roughly in the direction of $\vec{y}$ by computing the predictions of another weak learner and take a small step in that direction to get ourselves closer to $\vec{y}$. We iteratively repeat this process with many weak learners until we get to $\vec{y}$. This is exactly like gradient descent where we take a small step in roughly the right direction to get ourselves closer to a goal. **Boosting is identical to gradient descent in function space**. The iterative process of taking small steps in the directions of weak learners is represented by the summation:

$$H(\vec{x}) = \sum_{i=1}^{m} \alpha_i h_i(\vec{x}) \,.$$

The α_i's are some weights (in bagging $\alpha_i = \frac{1}{m}$). In the setup, we need a loss function to evaluate the classifiers. We define the loss of a classifier as:

$$\ell(h) = \frac{1}{n} \sum_{i=1}^{n} \ell(h(\vec{x_i}), y_i) \,,$$

i.e. the average loss of the classifier on the training data.

17.1 AnyBoost

Although, like bagging, we have an ensemble of classifiers to start with, unlike bagging it is unnecessary (unwise even) to include all of them in the weighted sum. Bagging solves the variance problem which by definition says that we are too far from the mean, so we take the mean of as many classifiers as possible. In the boosting setup, however, we only add a weak learner with a nonzero weight if it reduces the loss of the net assembly. If we think of the assembly as an iterative process, then at each step we must be greedy and add the weak learner that reduces the loss the most:

$$h_t = \arg\min_h \ \ell(H_t + \alpha_t h) \,,$$

where h_t is the weak learner to add in the t^{th} iteration and H_t is the classifier assembled until the t^{th} iteration. We build up from something like $H_1 = h_0 =$ the all 0 classifier. For now, let's assume that all α_t's are the same and equal to some constant α. Then,

$$\begin{aligned}
h_t &= \arg\min_h \ \ell(H_t + \alpha h) \\
&\approx \arg\min_h \ \underbrace{\ell(H_t)}_{\text{constant}} + \underbrace{\frac{\partial \ell}{\partial H_t} \cdot \alpha h}_{\text{inner product b/w functions}} \\
&= \arg\min_h \ \frac{\partial \ell}{\partial H_t} \cdot \alpha h \\
&= \arg\min_h \ \alpha \sum_{i=1}^{n} \frac{\partial \ell}{\partial H_t(\vec{x_i})} h(\vec{x_i}) \,.
\end{aligned}$$

For instance, if ℓ is the squared loss, then $\ell(H_t) = \frac{1}{n}\sum_{i=1}^{n}(H_t(\vec{x_i}) - y_i)^2 \Rightarrow \frac{\partial \ell}{\partial H_t(\vec{x_i})} = \frac{2}{n}(H_t(\vec{x_i}) - y_i)$. Therefore we get:

$$h_t = \arg\min_h \ \alpha \vec{\nabla}\ell(\vec{y}_{H_t})^{\mathsf{T}} \vec{y_h}\,.$$

But an inner product is minimum when the vectors are anti-parallel. Hence, the required h_t is the one whose y_{h_t} is in the direction opposite to the gradient of the loss. Are you watching closely? This is *exactly* what we do in gradient descent by taking a step in the direction opposite to the gradient of the loss. For the squared loss,

$$\vec{\nabla}\ell_{\text{sq}}(\vec{y}_{H_t}) = \frac{2}{n}(\vec{y}_{H_t} - \vec{y})\,,$$

which is a vector from $\vec{y}$ to $\vec{y}_{H_t}$. Therefore, at every step, we need the h_t such that $\vec{y}_{h_t}$ points exactly towards $\vec{y}$ from $\vec{y}_{H_t}$, like a compass. Such a perfect h_t would get us to $\vec{y}$ in one step. If such a weak learner doesn't exist in the ensemble, then we use the one closest to it, which is what the optimization says. This algorithm is called **AnyBoost**, which is identical to gradient descent in function space. This is a generic algorithm, of which two special cases are particularly important.

17.2 Gradient-Boosted Trees

⋆ Search engines use gradient-boosted trees deliver search results.

Consider a regression problem where our ensemble of weak learners consists of regression trees of limited depth which makes them high bias classifiers. At each iteration we want to find the regression tree that solves the AnyBoost optimization:

$$h_t = \arg\min_h \ \alpha \vec{\nabla}\ell(\vec{y}_{H_t})^{\mathsf{T}} \vec{y_h}\,.$$

The problem is that our ensemble is huge and exhaustive search simply won't work to find a solution. Our only solution is to construct the weak regression tree on the fly at every iteration. For this, we note four facts:

1. We can consider α to be positive without loss of generality. This is because we preemptively flip $\vec{y_h}$'s that have a positive inner product with the gradient of the loss. Consider this baked into the definition of $\vec{y_h}$. This consumes any negative sign that would've gone into α. Hence, α is positive $\Rightarrow$ removing it from the minimand won't change the solution to the optimization.
2. The solution to the above optimization problem does not change when we minimize twice the minimand instead.
3. $\vec{\nabla}\ell(\vec{y}_{H_t})$ is a constant in the above optimization since it's independent of $h \Rightarrow \|\vec{\nabla}\ell(\vec{y}_{H_t})\|^2$ is a constant $\Rightarrow$ adding it to the minimand won't change the solution to the optimization.
4. Although $\vec{y_h}$ is not a constant, we can set $\|\vec{y_h}\|$ to be a constant, i.e. set its scale, since we're only interested in its direction. This is because we're looking for the $\vec{y_h}$ that is closest to the one that points opposite to the gradient and hence we're only looking at the directions of the $\vec{y_h}$'s. The step size is anyways offloaded to α. Therefore, we set $\|\vec{y_h}\|$ to be a constant $\Rightarrow \|\vec{y_h}\|^2$ is a constant $\Rightarrow$ adding it to the minimand won't change the solution to the optimization.

From (1), (2), (3), (4),

$$h_t = \arg\min_h \ \|\vec{\nabla}\ell(\vec{y}_{H_t})\|^2 + 2\vec{\nabla}\ell(\vec{y}_{H_t})^\mathsf{T}\vec{y_h} + \|\vec{y_h}\|^2$$
$$= \arg\min_h \ \|\vec{y_h} - (-\vec{\nabla}\ell(\vec{y}_{H_t}))\|^2 .$$

But the final expression is nothing but the squared loss which is precisely what regression trees minimize! Hence, we construct a regression tree of limited depth with the above minimand as the loss function.

Algorithm 3: Gradient-Boosted Trees

Input: Training set $\mathcal{D} = \{(\vec{x_i}, y_i)\}_{i=1}^n$, number of iterations T, step size α
Output: Strong learner H

Initialize $H \leftarrow \vec{0}$; `// all 0 classifier`
for $t = 1 : T$ **do**
  $\vec{g} \leftarrow \vec{y} - \vec{y_H}$; `// negative gradient of the squared loss` $\ell_{\texttt{sq}}$
  $h \leftarrow \arg\min_h \|\vec{y_h} - g\|^2$; `// build a regression tree of limited depth`
  $H \leftarrow H + \alpha h$;
end
return H;

17.3 AdaBoost

"When we're all dead, AdaBoost will still be boosting."

AdaBoost was the first boosting algorithm. It stands for adaptive boosting. It was later realized that this is one specific instance of a general framework called gradient boosting which we saw so far. Gradient boosting is a whole family of algorithms which use a loss function and a weak learner to obtain a strong learner by doing gradient descent in function space. AdaBoost chooses the loss function to be the exponential loss:

$$\ell(h) = \sum_{i=1}^n e^{-y_i h(\vec{x_i})} .$$

The second choice that AdaBoost makes is that it adaptively sets the step size by optimizing α to take the ideal step using a line search. Earlier, we set α to be a constant. This means that during gradient descent, we take steps of the same size along every $\vec{y}_{h_t}$ for every weak learner h_t. On the other hand, for every weak learner, AdaBoost takes the step until the point where the loss is minimized. Hence, it $\underbrace{\text{searches along the line}}_{\text{line search}}$ in the direction of $\vec{y}_{h_t}$ to find the ideal step. This is why AdaBoost converges very fast. Another choice that AdaBoost makes is that it only works for classification, $\mathcal{Y} = \{+1, -1\}$, and the same applies for all the weak learner it uses. We know that at iteration t of boosting, we want to add the weak learner h_t to H_t such that:

$$h_t = \arg\min_h \ \alpha\vec{\nabla}\ell(\vec{y}_{H_t})^\mathsf{T}\vec{y_h} .$$

For AdaBoost,

$$\ell(y_{H_t}) = \vec{1}^{\mathsf{T}} \exp(-\vec{y} \odot \vec{y}_{H_t})$$
$$\Rightarrow \vec{\nabla}\ell(\vec{y}_{H_t}) = -\vec{y} \odot \exp(-\vec{y} \odot \vec{y}_{H_t})\,.$$

Note that an inner product with a vector of 1's gives the sum of components of the vector. In the gradient, the vector $\exp(-\vec{y} \odot \vec{y}_{H_t})$ is a vector of weights where each component is the contribution of the corresponding data point to the total loss (which is a summation of all the contributions). Let:

- $\hat{\vec{w}}_t = \exp(-\vec{y} \odot \vec{y}_{H_t})$ [hat does not signify unit vector, it is only notation],
- $z_t = \ell(\vec{y}_{H_t}) = \vec{1}^{\mathsf{T}} \exp(-\vec{y} \odot \vec{y}_{H_t}) = \vec{1}^{\mathsf{T}} \hat{\vec{w}}_t\,,$
- $\vec{w}_t = \frac{\hat{\vec{w}}_t}{z_t}\,.$

Clearly $\vec{1}^{\mathsf{T}} \vec{w}_t = 1$ since $\vec{1}^{\mathsf{T}} \vec{w}_t = \vec{1}^{\mathsf{T}} \frac{\hat{\vec{w}}_t}{z_t} = \frac{\vec{1}^{\mathsf{T}} \hat{\vec{w}}_t}{z_t} = \frac{z_t}{z_t} = 1$. Putting the gradient of the loss for AdaBoost in the objective of gradient boosting, we get:

$$\begin{aligned}
h_t &= \arg\min_h \ \alpha(-\vec{y} \odot \hat{\vec{w}}_t)^{\mathsf{T}} \vec{y_h} \\
&= \arg\min_h \ -\alpha(\vec{y} \odot \hat{\vec{w}}_t)^{\mathsf{T}} \vec{y_h} \\
&= \arg\min_h \ -\frac{1}{z_t}\alpha(\vec{y} \odot \hat{\vec{w}}_t)^{\mathsf{T}} \vec{y_h} \\
&= \arg\min_h \ -\alpha(\vec{y} \odot \vec{w}_t)^{\mathsf{T}} \vec{y_h} \\
&= \arg\min_h \ -\alpha \sum_{i=1}^{n} w_{ti} \underbrace{y_i h(\vec{x_i})}_{\in\{+1,-1\}}\,.
\end{aligned}$$

Define the set $S_h = \{i \mid y_i h(\vec{x_i}) = +1,\ 1 \le i \le n\}$. This means that $S_h^{\mathsf{C}} = \{1, 2, \ldots, n\} \setminus S_h = \{i \mid y_i h(\vec{x_i}) = -1,\ 1 \le i \le n\}$. With this, we can split up the summation in the minimand into two summations as:

$$\begin{aligned}
h_t &= \arg\min_h \ -\alpha \left(\sum_{i \in S_h} w_{ti} \ + \ \sum_{i \in S_h^{\mathsf{C}}} -w_{ti} \right) \\
&= \arg\min_h \ -\alpha \left(\sum_{i \in S_h} w_{ti} \ - \ \sum_{i \in S_h^{\mathsf{C}}} w_{ti} \right)
\end{aligned}$$

The second summation is the sum of the weights of all the points h got wrong. Hence, it is the *weighted error* (we scale the classification loss $\ell_{0/1}$ by the weights). Let the weighted error be ε_t. Then, the first summation is the *weighted accuracy* (the sum of all the weights of the points h got

correct). This is equal to $1 - \varepsilon_t$ since $\sum_{i=1}^{n} w_{ti} = 1$. Writing the minimand in terms of ε_t, we get:

$$\begin{aligned} h_t &= \arg\min_h \ -\alpha(1 - 2\varepsilon_t) \\ &= \arg\min_h \ \underbrace{\alpha}_{\text{constant}} (2\varepsilon_t - 1) \\ &= \arg\min_h \ \varepsilon_t \\ &= \arg\min_h \ \sum_{h(\vec{x_i}) \neq y_i} w_{ti} \,. \end{aligned}$$

Therefore, $\boxed{h_t = \arg\min_h \sum_{h(\vec{x_i}) \neq y_i} w_{ti}}$. This is a really profound result. This says that if our weak learner is such that it takes in a dataset and weights, and returns the classifier that minimizes the weighted error, then we can boost it using AdaBoost. We start with all $w_{0i} = 1$. With this setup at $t = 0$ our weak learner will return some classifier h_0. Notice that at iteration t, the weights w_{ti} depend on the (mis)classifications of the previous iterations since $\vec{w}_t$ is a function of $\vec{y}_{H_t}$. Hence, at $t = 1$, the points that were misclassified by $H_1 = h_0$ will have heavy weights as compared to those that were classified correctly. With the dataset and these new weights, our weak learner will spew out another classifier h_1 which will help determine the weights w_{2i} for $t = 2$ since H_2 "contains" h_1. This continues until we get every single data point right.

⋆ AdaBoost often overfits slowly in practice while driving down training error, but this is not a universal logarithmic law and noisy data can break the pattern.

⋆ AdaBoost is not great when the data is too noisy.

We've ignored α until now. It's time to determine the ideal step size α_t at iteration t. For gradient-boosted trees, we took α to be a small constant. For AdaBoost, we use a line search that says that we want to step to the point on the line in the direction of $\vec{y}_{h_t}$ where the loss is minimum. Hence, at iteration t, we want:

$$\begin{aligned} \alpha_t &= \arg\min_\alpha \ \ell(H_t + \alpha h_t) \\ &= \arg\min_\alpha \ \sum_{i=1}^{n} e^{-y_i[H_t(\vec{x_i}) + \alpha h_t(\vec{x_i})]} \\ &= \arg\min_\alpha \ \sum_{i=1}^{n} e^{-y_i H_t(\vec{x_i}) - \alpha y_i h_t(\vec{x_i})} \,. \end{aligned}$$

We've assembled H_t until the previous iteration, and we've found h_t by minimizing ε_t. We differentiate

w.r.t. α and equate with 0:

$$\begin{aligned}
\frac{d\ell}{d\alpha} &= -\sum_{i=1}^{n} \underbrace{y_i h_t(\vec{x_i})}_{\in\{+1,-1\}} e^{-y_i H_t(\vec{x_i})-\alpha y_i h_t(\vec{x_i})} \\
&= -\sum_{i\in S_{h_t}} e^{-y_i H_t(\vec{x_i})-\alpha} \; + \sum_{i\in S_{h_t}^{\complement}} e^{-y_i H_t(\vec{x_i})+\alpha} \\
&= -\sum_{i\in S_{h_t}} w_{ti} e^{-\alpha} \; + \sum_{i\in S_{h_t}^{\complement}} w_{ti} e^{\alpha} \\
&= -e^{-\alpha} \sum_{i\in S_{h_t}} w_{ti} \; + \; e^{\alpha} \sum_{i\in S_{h_t}^{\complement}} w_{ti} \\
&= -(1-\varepsilon_t) e^{-\alpha} + \varepsilon_t e^{\alpha} \\
&= \frac{-1+\varepsilon_t+\varepsilon_t e^{2\alpha}}{e^{\alpha}} = 0 \\
\Rightarrow e^{2\alpha} &= \frac{1-\varepsilon_t}{\varepsilon_t} \\
\Rightarrow \quad \alpha &= \frac{1}{2}\ln\Big(\frac{1-\varepsilon_t}{\varepsilon_t}\Big) \; .
\end{aligned}$$

Therefore, $\boxed{\alpha_t = \frac{1}{2}\ln\Big(\frac{1-\varepsilon_t}{\varepsilon_t}\Big)}$. The great thing about AdaBoost is that solving the line search optimization gives the optimal step size in a closed-form, which does not happen in general. Generally, we need to try out many step sizes and pick the best one since a magical closed-form formula for the optimal step size as in AdaBoost is not available.

Algorithm 4: AdaBoost

Input: Training set $\mathcal{D} = \{(\vec{x_i}, y_i)\}_{i=1}^{n}$, number of iterations T, weak learner $\mathcal{A}$
Output: Strong learner H

Initialize $H \leftarrow \vec{0}$; `// all 0 classifier`
Initialize $\vec{w} \leftarrow \frac{\vec{1}}{n}$; `// initialize all weights to` $\frac{1}{n}$
for $t = 1 : T$ **do**
  $h \leftarrow \mathcal{A}(\mathcal{D}, \vec{w})$;
  $\varepsilon \leftarrow \sum_{i:h(\vec{x_i})\neq y_i} w_i$;
  if $0 < \varepsilon < \frac{1}{2}$ **then**
    $\alpha \leftarrow \frac{1}{2}\ln\Big(\frac{1-\varepsilon}{\varepsilon}\Big)$;
    $H \leftarrow H + \alpha h$;
    $\vec{w} \leftarrow \frac{1}{2\sqrt{\varepsilon(1-\varepsilon)}} \vec{w} \odot \exp(-\alpha \vec{y} \odot \vec{y_h})$;
    continue;
  end
  return H;
end
return H;

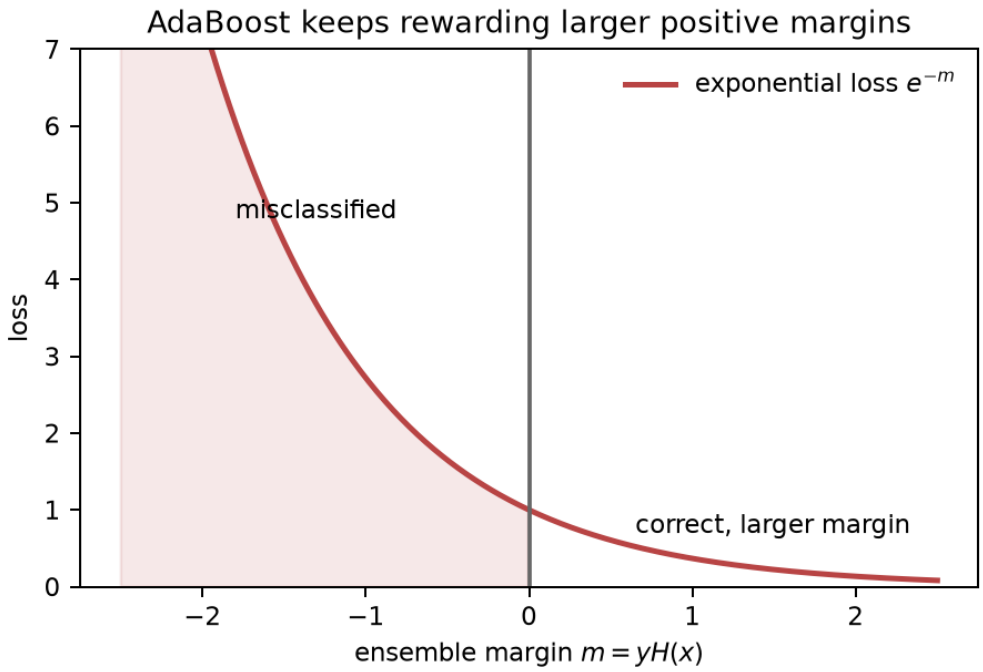

Figure 14: Visualization of AdaBoost's exponential loss as a function of ensemble margin.

⋆ Under a uniform weak-learning edge $\gamma > 0$ at every round, AdaBoost's training error is at most $e^{-2\gamma^2 T}$, so reaching error below $1/n$ takes $O((\log n)/\gamma^2)$ rounds. If we keep boosting after achieving 0 training error, AdaBoost keeps increasing the \underline{margin, until it starts overfitting a long time later. Hence, AdaBoost is a large margin classifier.

18 Neural Networks

"Genetic programming is the Bud Light of machine learning algorithms."

Machine learning using neural networks was earlier called machine learning using **multilayer perceptron (MLP)**, and recently it's called **deep learning**. It's very related to kernels. Kernels put simple linear classifiers on steroids by mapping $\vec{x} \mapsto \phi(\vec{x})$ and computing $h(\vec{x}) = \vec{w}^\mathsf{T}\phi(\vec{x}) + \vec{b}$. The function ϕ is well-crafted such that inner products in the higher dimension can be computed in closed-form and the mapping is actually *implicit*. This is restrictive because we can only pick hand-crafted functions ϕ for which we know how to compute the inner products using a nice kernel function. Neural networks *learn* the transformation ϕ instead. They use a **nonlinear activation function** σ to represent ϕ:

$$\phi(\vec{x}) = \sigma(W\vec{x} + \vec{c})\,.$$

For a long time σ was taken to be the sigmoid function $\sigma(z) = \frac{1}{1+e^{-z}}$. Now people use the **rectified linear unit (ReLU)**: $\sigma(z) = \max(0, z)$. If the transformation didn't use nonlinearity then $\vec{w}^\mathsf{T}\phi(\vec{x}) + \vec{b}$ would still be a linear classifier. The activation function is mostly a matter of convenience: what's best for optimization. Sigmoid activation is flat at the ends (no gradients) so ReLU is more convenient. When we train a neural network, we learn $\underbrace{\vec{w},\ \vec{b}}_{\text{hyperplane}}$, $\underbrace{W,\ \text{and}\ \vec{c}}_{\text{representation}}$ at the same time. The benefit of deep learning over kernel machines is that deep learning scales linearly with data, whereas kernel machines scale quadratically since they use matrices. Qualitatively, a plot of accuracy vs. the number of training data points for kernel machines is one of diminishing returns since it ultimately asymptotes. In order to make kernel machines work on massive data, approximations in computing the kernel matrix become unavoidable which cost us accuracy. After a point, the improvement we get from more data is roughly the same order of magnitude as we have to pay for scaling the algorithm up. Hence, kernel machines ultimately stop improving. This is not the case for neural networks. They keep getting better with more data.

At the end of the day, deep learning optimizes a loss function using gradient descent, or rather, a variant of it. It can be shown that neural networks are **universal approximators** i.e. they can estimate any function *arbitrarily* closely. For instance, if we fit a regression curve in 2D with a neural network using the squared loss and ReLU activation, the neural network will estimate the function using a piecewise linea function (using many lines). If we add an arbitrarily large number of linear pieces, we can fit the function arbitrarily closely i.e. more powerful the neural network. Hence, the bigger the matrix W (in terms of number of rows), the more pieces we have to work with, and the more powerful the neural network. It is impractical to have an arbitrarily large W. This is where the idea of **layers** comes in to make the neural network more powerful. The idea is that we use *multiple nested transformations* instead of a single one. Until now, our hypothesis has a single transformation:

$$\begin{aligned} \vec{x} \xrightarrow{\text{input}}\ & \phi(\vec{x}) = \sigma(W\vec{x} + \vec{c})\,, \\ & h(\vec{x}) = \vec{w}^\mathsf{T}\phi(\vec{x}) + \vec{b}\,. \xrightarrow{\text{output}} h(\vec{x}) \end{aligned}$$

With *multiple nested transformations*, the hypothesis becomes:

$$\begin{aligned}
\vec{x} \xrightarrow{\text{input}} \quad \phi_1(\vec{x}) &= \sigma(W_1\vec{x} + \vec{c_1})\,, \\
\phi_2(\vec{x}) &= \sigma(W_2\phi_1(\vec{x}) + \vec{c_2})\,, \\
&\vdots \\
\phi_m(\vec{x}) &= \sigma(W_m\phi_{m-1}(\vec{x}) + \vec{c_m})\,, \\
h(\vec{x}) &= \vec{w}^{\mathsf{T}}\phi_m(\vec{x}) + \vec{b}\,. \xrightarrow{\text{output}} h(\vec{x})
\end{aligned}$$

Each transformation corresponds to one *layer*. Neural networks having many layers are "deep." Earlier, people only had single layers although they knew that multilayered architectures were possible (e.g. MLP, etc.). This is because multilayered architectures required much more compute (no GPUs back then!). Also, it can be shown that there is no function in the world that can be learned by a deep network that cannot also be learned by a shallow network. "All you ever need is one layer." *Depth does not improve expressivity.* However, what people didn't look into is the size of the W matrix required for the same expressivity as a deep network. *A single layer must be exponentially wide to achieve the same expressivity as a deep network.* In a deep network with ReLU activation, the first layer approximates a function using a piecewise linear function, the second layer takes that piecewise linear function, say f, and uses is as the building block to approximate the target function; i.e. the second layer approximates the target function as a *piecewise f-shaped function*, say g. The third layer approximates the target function as a piecewise g-shaped function, and so on. Hence, a deep network offers an *exponential explosion of expressivity* as we go down its layers. The bias terms $\vec{b}$ and $\vec{c_i}$'s can be incorporated into the weight matrices by considering them as hardwired constant inputs to the corresponding layers.

As mentioned before, neural networks learn the good old way by optimizing a loss function using gradient descent. The only difference is that they learn multiple parameters at the same time. To recall, suppose $\ell(\vec{w})$ is a convex loss function in terms of parameter $\vec{w}$, then gradient descent starts with some $\vec{w} = \vec{w_0}$ and iteratively takes steps by updating $\vec{w}$ as $\vec{w} \leftarrow \vec{w} - \alpha\vec{\nabla}\ell(\vec{w})$ until $\vec{w}$ converges to the minimum. This is exactly what a neural network does but for all parameters it wants to learn. Suppose we have a 4-layered neural network:

$$\begin{aligned}
\vec{x} \xrightarrow{\text{input}} \quad \phi_1(\vec{x}) &= \sigma(W_1\vec{x})\,, \\
\phi_2(\vec{x}) &= \sigma(W_2\phi_1(\vec{x}))\,, \\
\phi_3(\vec{x}) &= \sigma(W_3\phi_2(\vec{x}))\,, \\
h(\vec{x}) &= \vec{w}^{\mathsf{T}}\phi_3(\vec{x})\,. \xrightarrow{\text{output}} h(\vec{x})
\end{aligned}$$

Here, the NN learns the parameters W_1, W_2, W_3, and $\vec{w}$ by optimizing a generally nonconvex objective:

$$\mathscr{L}(W_1, W_2, W_3, \vec{w}) = \sum_{i=1}^{n} \ell(h(\vec{x_i}), y_i)\,.$$

To optimize $\mathscr{L}$ using gradient descent, it needs to compute the gradients:

$$\begin{aligned}
\vec{\nabla}_{W_1}\mathscr{L} &= \frac{\partial \mathscr{L}}{\partial W_1}\,, \\
\vec{\nabla}_{W_2}\mathscr{L} &= \frac{\partial \mathscr{L}}{\partial W_2}\,, \\
\vec{\nabla}_{W_3}\mathscr{L} &= \frac{\partial \mathscr{L}}{\partial W_3}\,, \\
\vec{\nabla}_{\vec{w}}\mathscr{L} &= \frac{\partial \mathscr{L}}{\partial \vec{w}}\,,
\end{aligned}$$

in each iteration and do the GD update for all parameters. The "last" gradient is simple:

$$\frac{\partial \mathscr{L}}{\partial \vec{w}} = \sum_{i=1}^{n} \frac{\partial \ell(h(\vec{x_i}), y_i)}{\partial h(\vec{x_i})} \cdot \underbrace{\frac{\partial h(\vec{x_i})}{\partial \vec{w}}}_{\phi_3(\vec{x_i})}\,.$$

For the rest of the gradients, we use the chain rule of differentiation. Let:

$$\begin{aligned}
a_1(\vec{x}) &= W_1\vec{x} \quad [\text{i.e. } \phi_1(\vec{x}) = \sigma(a_1)]\,, \\
a_2(\vec{x}) &= W_2\phi_1(\vec{x}) \quad [\text{i.e. } \phi_2(\vec{x}) = \sigma(a_2)]\,, \\
a_3(\vec{x}) &= W_3\phi_2(\vec{x}) \quad [\text{i.e. } \phi_3(\vec{x}) = \sigma(a_3)]\,.
\end{aligned}$$

Then we compute:

$$\begin{aligned}
\frac{\partial \mathscr{L}}{\partial W_1} &= \sum_{i=1}^{n} \frac{\partial \ell(h(\vec{x_i}), y_i)}{\partial h(\vec{x_i})} \cdot \frac{\partial h(\vec{x_i})}{\partial \phi_3} \cdot \frac{\partial \phi_3}{\partial a_3} \cdot \frac{\partial a_3}{\partial a_2} \cdot \frac{\partial a_2}{\partial a_1} \cdot \underbrace{\frac{\partial a_1}{\partial W_1}}_{id_V}\,, \\
\frac{\partial \mathscr{L}}{\partial W_2} &= \sum_{i=1}^{n} \frac{\partial \ell(h(\vec{x_i}), y_i)}{\partial h(\vec{x_i})} \cdot \frac{\partial h(\vec{x_i})}{\partial \phi_3} \cdot \frac{\partial \phi_3}{\partial a_3} \cdot \frac{\partial a_3}{\partial a_2} \cdot \underbrace{\frac{\partial a_2}{\partial W_2}}_{\phi_1}\,, \\
\frac{\partial \mathscr{L}}{\partial W_3} &= \sum_{i=1}^{n} \frac{\partial \ell(h(\vec{x_i}), y_i)}{\partial h(\vec{x_i})} \cdot \frac{\partial h(\vec{x_i})}{\partial \phi_3} \cdot \frac{\partial \phi_3}{\partial a_3} \cdot \underbrace{\frac{\partial a_3}{\partial W_3}}_{\phi_2}\,.
\end{aligned}$$

Out of the many gradients on the right-hand sides, we have already computed $\frac{\partial a_3}{\partial W_3} = \phi_2$, $\frac{\partial a_2}{\partial W_2} = \phi_1$, $\frac{\partial a_1}{\partial W_1} = id_V$, where id_V is the identity function for vectors. Also, $\frac{\partial \ell(h(\vec{x_i}), y_i)}{\partial h(\vec{x_i})} \cdot \frac{\partial h(\vec{x_i})}{\partial \phi_3} \cdot \frac{\partial \phi_3}{\partial a_3}$ is used in all computations, so we compute it once and reuse it. Similarly, we compute $\frac{\partial a_3}{\partial a_2}$ once for $\frac{\partial \mathscr{L}}{\partial W_1}$ and reuse it for $\frac{\partial \mathscr{L}}{\partial W_2}$. Effectively, we only need to compute three gradients: $\frac{\partial \mathscr{L}}{\partial a_3}$, $\frac{\partial a_3}{\partial a_2}$, $\frac{\partial a_2}{\partial a_1}$. Hence, we see that once we "go" all the way from the *output layer* to the *input layer* using the chain rule to compute $\frac{\partial \mathscr{L}}{\partial W_1}$, we already have all the pieces we need to compute the rest of the gradients. It may seem that computing the many gradients in a neural network requires tremendous compute, but the reuse of gradients makes it much more economical and easier. All we need is one *backward pass* from the output layer to the input layer to compute all the gradients we need, and then we can simply reuse them wherever else. This is called **back propagation** in a neural network. Another way of looking at it is all we need to calculate the gradient w.r.t. W_k at the k^{th} layer (may be deep

within the neural network) is the gradient w.r.t. W_{k-1} (until the previous layer) and one additional gradient. Hence, we effectively need one new gradient per layer $\Rightarrow$ the back-propagation algorithm is linear in the depth of the NN, which is very efficient. Now that we have all the gradients, all we need to do is gradient descent:

$$W_1 \leftarrow W_1 - \alpha \frac{\partial \mathscr{L}}{\partial W_1},$$
$$W_2 \leftarrow W_2 - \alpha \frac{\partial \mathscr{L}}{\partial W_2},$$
$$W_3 \leftarrow W_3 - \alpha \frac{\partial \mathscr{L}}{\partial W_3},$$
$$\vec{w} \leftarrow \vec{w} - \alpha \frac{\partial \mathscr{L}}{\partial \vec{w}}.$$

But we make two changes with neural networks:

1. The loss function $\mathscr{L}$ is *not* convex! Although ℓ is convex, $\mathscr{L}$ is not. This is because of the nonlinear activation function. $\mathscr{L}$ is convex w.r.t $\vec{w}$ but it's not convex w.r.t. the W's. Hence, when we minimize the loss using gradient descent (convex optimization), we are almost sure to trapped in a local minimum (exponentially low chance of finding the global minimum). All we can do is try and find a good local minimum (which local minima are good? answer in point 2). This means that we need a good starting point (one that takes us to a good local minimum) for initializing our convex optimization. When the loss was actually convex, the initial value didn't matter (we generally set it to $\vec{0}$ or chose it randomly) but now it does tremendously. Turns out that initializing neural network optimization with $\vec{0}$ is a horrible idea because it greatly restricts the expressivity of the NN since all dimensions end up getting the same value. The way to go is to initialize the optimization randomly: typically, $\vec{w}$ and the weight matrices are initialized with low-order Gaussian noise. As such, different neural networks trained on the same data are bound to give different results since their optimizations are sure to be initialized differently which may cause them to optimize to different local minima. This is not the case with SVMs, say, since they find the globally optimal hyperplane. It is not unusual that two neural networks trained on the same data are *very* different because of the inherent randomness in neural networks. This makes neural networks extremely suitable for ensembling since very different neural networks make very different errors which get averaged out when ensembled. For bagging neural networks, we don't even need to sample multiple datasets. The inherent randomness in neural networks takes care of giving us different classifiers.

2. If we use aggressive optimization algorithms (e.g. approximate Hessian steps) we will race to the nearest local minimum (from our initialization point) blazingly fast. This is generally suboptimal because in a highly non-convex loss landscape such as ours, it is highly likely that the local minimum *nearest* to our (randomly) chosen starting point is a terrible one. We would much rather prefer being less aggressive with the optimization and end up in a local minimum farther away (since it's most likely better). This is where **stochastic gradient descent (SGD)** comes in. Notice that the gradients of $\mathscr{L}$ w.r.t. the different parameters are all summations over the n data points ($\sum_{i=1}^{n} \ldots$), and hence in the GD algorithm iterate over all data points to calculate the gradients. The idea behind SGD, however, is that instead of computing the gradients over all the data points (which is the correct thing to do), we

approximate it *with a single point*. At each iteration of the optimization, we pick one data point at random, compute the gradient at that point (instead of summing over all data points) and use that gradient for that step. Moreover, we take smaller steps than we would in SGD so that although the direction at each step is completely wrong, we go in the correct direction *in expectation* (like boosting). SGD is a terrible convex optimization algorithm because although it takes us close to the minimum, it's never really able to pin-point the damn thing (which we don't need in machine learning anyway but nevertheless). However, this works in our favor in neural network optimization because SGD is so *noisy* that it simply escapes out of the "small" local minima in a non-convex loss landscape. It only gets stuck if the local minimum is "big" enough to trap it. Note that "big" does not mean "deep." Rather "big" local minima are the "wide" (but maybe shallow) ones. They are preferable because they ensure that we *don't overfit*. What we really want is to get as low as possible on our loss function *in a really wide minimum* and that's the only thing that SGD is capable of finding.

- Therefore, good = "wide."

In practice, instead of approximating the gradient using just a single data point, we approximate it with a *minibatch* of k data points. This is called **batched gradient descent (BGD)**. Typically k is chosen to be something like 64 or 128 for better efficiency (performance, CS 463, L1 cache size, etc.). Also, in practice, the learning rate is set to be large initially so that the steps become larger and noisier and we more easily walk out of small local minima towards a desirable wide local minimum. It's like getting an express ticket to the *area* of the wide local minimum by bypassing all the small ones. Once we are stuck in the local minimum we desire, we drop the learning rate so that we are able to drive down the loss as much as possible. We would not be able to pinpoint the minimum with a large learning rate (large steps) because it is noisy.

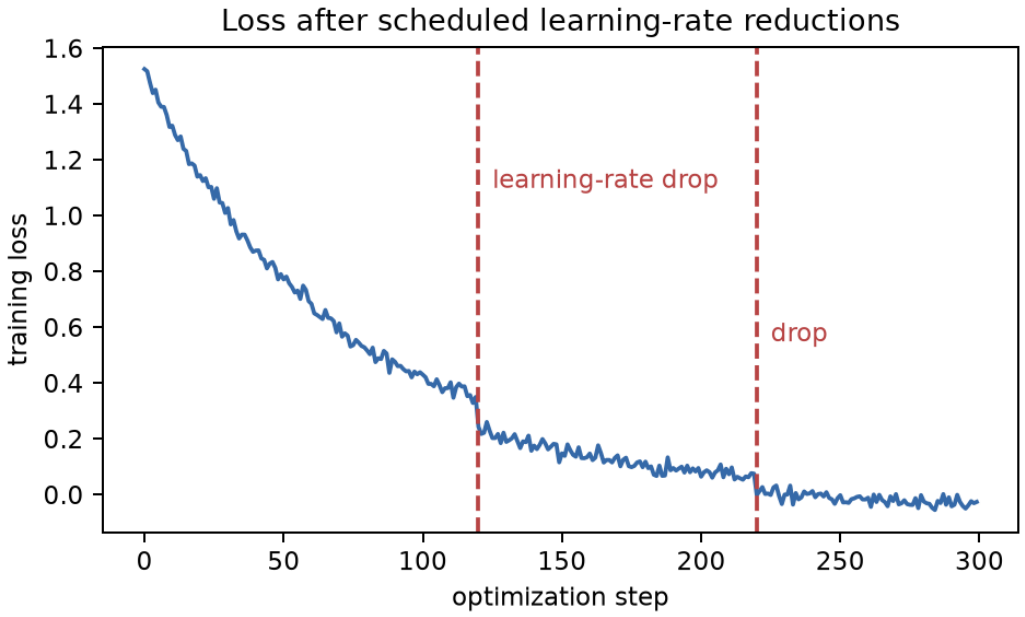

Figure 15: Optimization trace showing scheduled learning-rate reductions.

18.1 The Neural View

People claim the *artificial* neural networks to be inspired from the neural networks in the brain and how the neurons fire when the synapse is above a threshold or something like that. The analogy extends to drawing neural networks as connected graphs where each node is compared to a neuron. But the brain doesn't use gradient descent. People think the artificial neurons do more than they're actually doing. One neuron is really just the operation on one feature in the input vector.

> "It's like a lasagne: more layers is usually better."

18.2 Conv Nets

If we represent a 2D image as a single vector, we're losing almost all the spatial information. Convolution is **translation equivariant**: translating the input translates its feature map. Invariance of a final prediction requires additional operations or design choices. If we represent a 2D image as a single vector, this property is completely lost. Hence, if our trained neural network is able to identify an image it won't necessarily be able to identify its translated version (unless it has seen it enough in the training set). **Convolutional neural networks (CNNs)** change the function ϕ to exploit this equivariance and can be designed to produce approximately translation-invariant predictions. The golden principle is that if we change our data input a little bit by translating things around, it won't change the output. Originally, with a single vector representation, we would have needed to include many possible translational transformations of images to come up with a decent model. Using the golden principle in convolutional neural networks reduces our data exponentially. Initially, a CNN splits an image into channels—typically R, G, B—while maintaining the image form; i.e. the $\vec{x}$'s now consist of three different images. The function ϕ_1 takes such an $\vec{x}$ as input and generates new images which could be however many in number. The function ϕ_2 takes those images as input and generates newer images, and so on. At the very end, the CNN outputs an $h(\vec{x})$. The trick here is the *convolution operator*. Each function ϕ is associated with a particular convolution or **filter** or **kernel**: a small matrix of weights that is slid across an input image to obtain an output image. Each pixel of an output image is computed using an inner product between the filter and a matrix of pixels of an input image, making the output pixel a *weighted average* of the input pixels. Each convolution operation in a CNN is a way of detecting features which progressively keep getting more detailed. Each convolution is a more complex pattern detector than the previous.

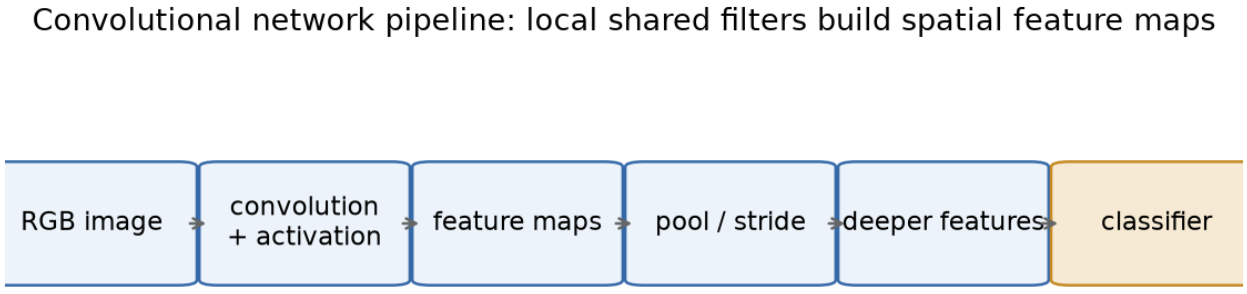

Figure 16: Convolutional-network pipeline.

A CNN can be thought of as a special case of a fully connected neural network where we assign

weights to only the first small matrix of pixels and zero out everything else. We can take a fully connected network and make it *identical* to a CNN. A sufficiently large fully connected network can represent a convolutional layer, but neither architecture is unconditionally more powerful at a fixed parameter or data budget. In CNNs we restrict the network to only learn functions that are translational invariant. The kernels in CNNs *are* the weights so that is what the CNNs learn. In addition, a CNN may have to learn one bias term per layer and more weights for dense layers towards the end. The operation of a convolution layer can be represented as:

$$\phi(\vec{x}) = \sigma(\vec{w}^{\mathsf{T}} X + \vec{b}) \,,$$

where $\vec{w}$ is the flattened kernel and each column of X is a flattened local patch.

19 More Machine Learning

Machine learning for intelligent systems tries to make predictions from data for instance in self-driving cars, or stock market predictions, or autonomous decision-making problems. Machine learning for data science tries to understand the data for instance finding cluster structure, low dimensional subspaces, etc.