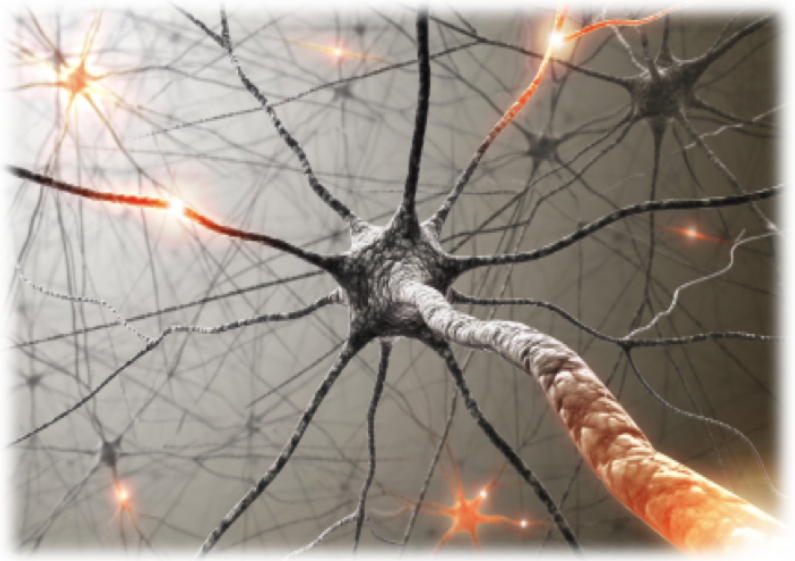
Lecture 4: Machine learning III

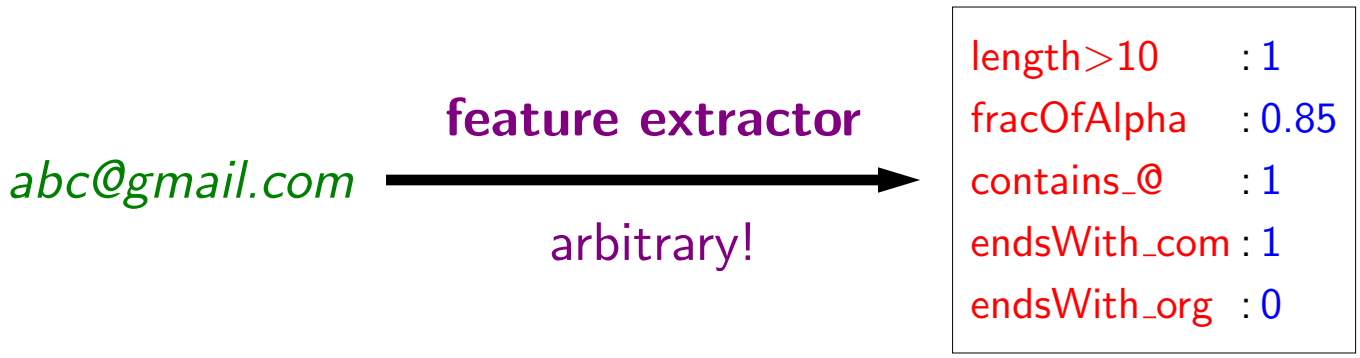


Announcements

- Homework 1 (foundations): Thursday 11pm is 2 late day **hard deadline**

- Section Thursday 3:30pm: backpropagation example, nearest neighbors, scikit-learn

Review: feature extractor



- Last lecture, we spoke at length about the importance of features, how to organize them using feature templates, and how we can get interesting non-linearities by choosing the feature extractor ϕ judiciously. This is you using all your domain knowledge about the problem.

Review: prediction score

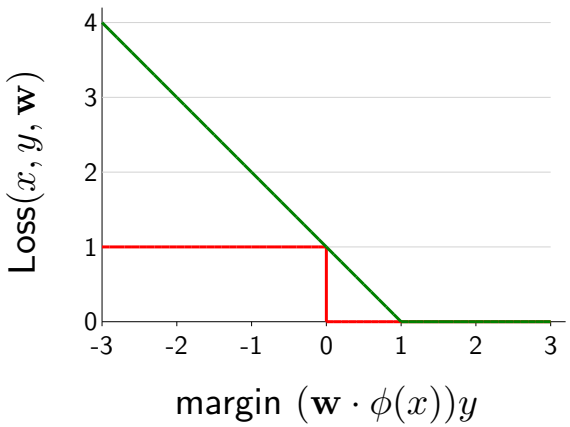
- Linear predictor: score $= \mathbf{w} \cdot \phi(x)$

- Neural network: score $= \sum_{j=1}^{k} w_j \sigma(\mathbf{v}_j \cdot \phi(x))$

- Given the feature extractor ϕ, we can use that to define a prediction score, either using a linear predictor or a neural network. If you use neural networks, you typically have to work less hard at designing features, but you end up with a harder learning problem. There is a human-machine tradeoff here.

Review: loss function

Loss$(x, y, \mathbf{w})$:



(for binary classification)

$$\text{TrainLoss}(\mathbf{w}) = \frac{1}{|\mathcal{D}_{\text{train}}|} \sum_{(x,y) \in \mathcal{D}_{\text{train}}} \text{Loss}(x, y, \mathbf{w})$$

Stochastic gradient descent:

$$\mathbf{w} \leftarrow \mathbf{w} - \eta \nabla_{\mathbf{w}} \text{Loss}(x, y, \mathbf{w})$$

- The prediction score is the basis of many types of predictions, including regression and binary classification. The loss function connects the prediction score with the correct output y, and measures how unhappy we are with a particular weight vector $\mathbf{w}$.
- This leads to an optimization problem, that of finding the $\mathbf{w}$ that yields the lowest training loss. We saw that a simple algorithm, stochastic gradient descent, works quite well.

Question

What's the true objective of machine learning?

- minimize error on the training set
- minimize training error with regularization
- minimize error on the test set
- minimize error on unseen future examples
- learn about machines

- We have written the average training loss as the objective function, but it turns out that that's not really the true goal. That's only what we tell our optimization friends so that there's something concrete and actionable. The true goal is to minimize error on unseen future examples; in other words, we need to **generalize**. As we'll see, this is perhaps the most important aspect of machine learning and statistics — albeit a more elusive one.

Roadmap

Generalization

Unsupervised learning

Summary

Training error

$$\text{TrainLoss}(\mathbf{w}) = \frac{1}{|\mathcal{D}_{\text{train}}|} \sum_{(x,y) \in \mathcal{D}_{\text{train}}} \text{Loss}(x, y, \mathbf{w})$$

Is this a good objective?

- Now let's be a little more critical about what we've set out to optimize. So far, we've declared that we want to minimize the training loss.

A strawman algorithm

Algorithm: rote learning

Training: just store $\mathcal{D}_{\text{train}}$.

Predictor $f(x)$:

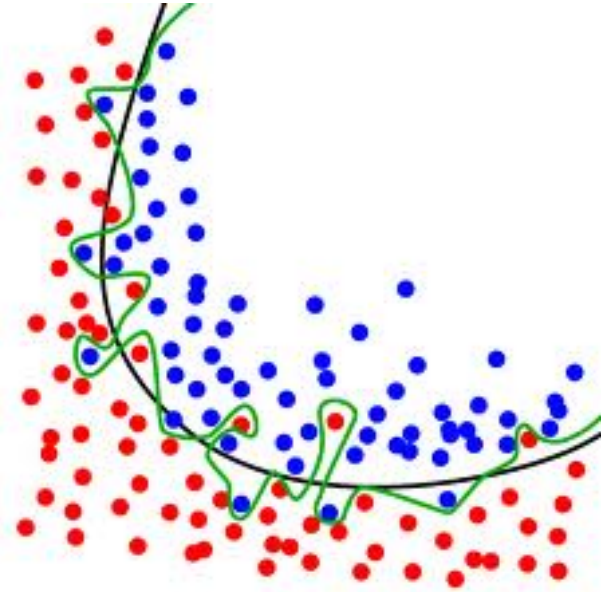
If $(x, y) \in \mathcal{D}_{\text{train}}$: return y.

Else: **segfault**.

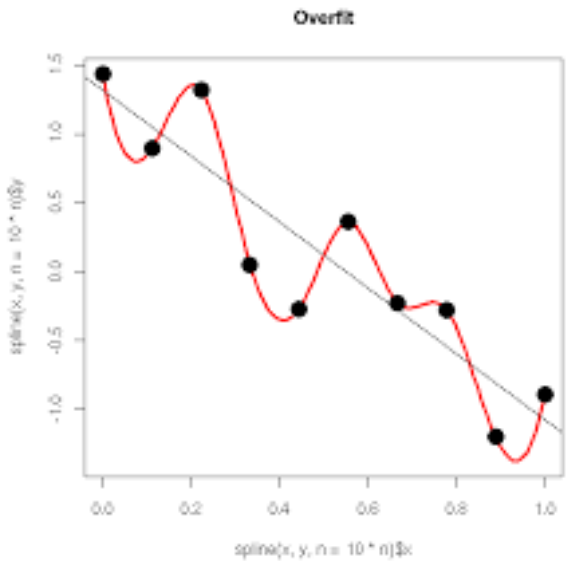
Minimizes the objective perfectly (zero), but clearly bad...

- Clearly, machine learning can't be about just minimizing the training loss. The rote learning algorithm does a perfect job of that, and yet is clearly a bad idea. It **overfits** to the training data and doesn't **generalize** to unseen examples.

Overfitting pictures



Classification



Regression

- Here are two pictures that illustrate what can go wrong if you only try to minimize the training loss for binary classification and regression.
- On the left, we see that the green decision boundary gets zero training loss by separating all the blue points from the red ones. However, the smoother and simpler black curve is intuitively more likely to be the better classifier.
- On the right, we see that the predictor that goes through all the points will get zero training loss, but intuitively, the black line is perhaps a better option.
- In both cases, what is happening is that by over-optimizing on the training set, we risk fitting **noise** in the data.

Evaluation

How good is the predictor f?

Key idea: the real learning objective

Our goal is to minimize **error on unseen future examples**.

Don't have unseen examples; next best thing:

Definition: test set

Test set $\mathcal{D}_{\text{test}}$ contains examples not used for training.

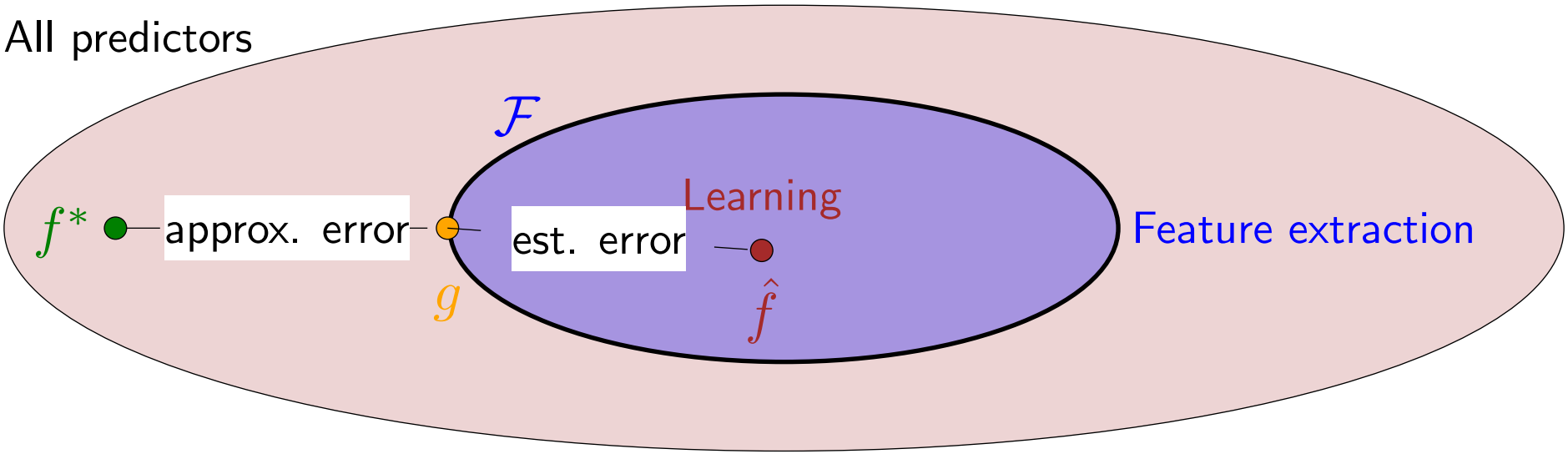
- So what is the true objective then? Taking a step back, what we're doing is building a system which happens to use machine learning, and then we're going to deploy it. What we really care about is how accurate that system is on those **unseen future** inputs.
- Of course, we can't access unseen future examples, so the next best thing is to create a **test set**. As much as possible, we should treat the test set as a pristine thing that's unseen and from the future. We definitely should not tune our predictor based on the test error, because we wouldn't be able to do that on future examples.
- Of course at some point we have to run our algorithm on the test set, but just be aware that each time this is done, the test set becomes less good of an indicator of how well you're actually doing.

Generalization

When will a learning algorithm **generalize** well?

- So far, we have an intuitive feel for what overfitting is. How do we make this precise? In particular, when does a learning algorithm generalize from the training set to the test set?

Approximation and estimation error

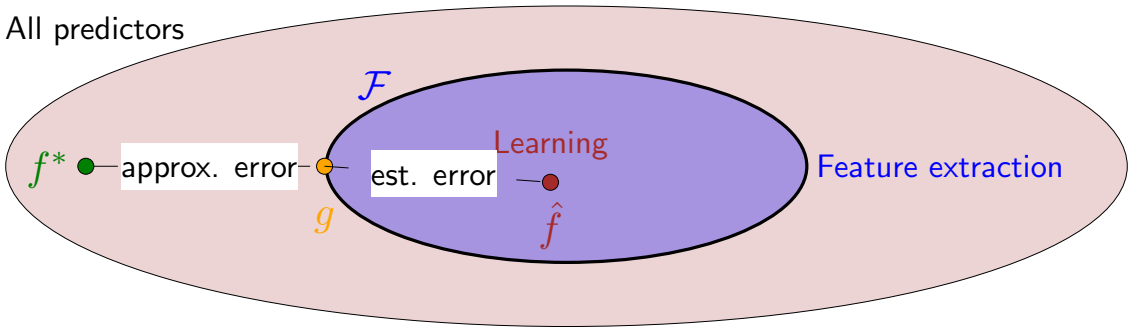


- Approximation error: how good is the hypothesis class?
- Estimation error: how good is the learned predictor **relative to** the potential of the hypothesis class?

$$\underbrace{\textbf{Err}(\hat{f}) - \textbf{Err}(g)}_{\textbf{estimation}} + \underbrace{\textbf{Err}(g) - \textbf{Err}(f^*)}_{\textbf{approximation}}$$

- Here's a cartoon that can help you understand the balance between fitting and generalization. Out there somewhere, there is a magical predictor f^* that classifies everything perfectly. This predictor is unattainable; all we can hope to do is to use a combination of our domain knowledge and data to approximate that. The question is: how far are we away from f^*?
- Recall that our learning framework consists of (i) choosing a hypothesis class $\mathcal{F}$ (by defining the feature extractor) and then (ii) choosing a particular predictor $\hat{f}$ from $\mathcal{F}$.
- **Approximation error** is how far the entire hypothesis class is from the target predictor f^*. Larger hypothesis classes have lower approximation error. Let $g \in \mathcal{F}$ be the best predictor in the hypothesis class in the sense of minimizing test error $g = \arg\min_{f \in \mathcal{F}} \text{Err}(f)$. Here, distance is just the differences in test error: $\text{Err}(g) - \text{Err}(f^*)$.
- **Estimation error** is how good the predictor $\hat{f}$ returned by the learning algorithm is with respect to the best in the hypothesis class: $\text{Err}(\hat{f}) - \text{Err}(g)$. Larger hypothesis classes have higher estimation error because it's harder to find a good predictor based on limited data.
- We'd like both approximation and estimation errors to be small, but there's a tradeoff here.

Effect of hypothesis class size



As the hypothesis class size increases...

Approximation error decreases because:

taking min over larger set

Estimation error increases because:

harder to estimate something more complex

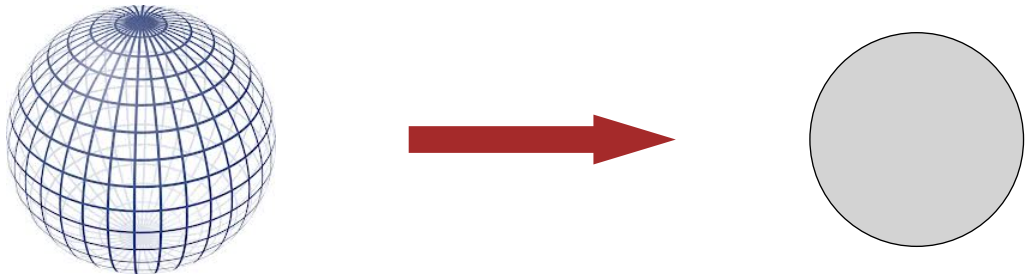
How do we control the hypothesis class size?

- The approximation error decreases monotonically as the hypothesis class size increases for a simple reason: you're taking a minimum over a larger set.
- The estimation error increases monotonically as the hypothesis class size increases for a deeper reason involving statistical learning theory (explained in CS229T).
- For each weight vector $\mathbf{w}$, we have a predictor $f_{\mathbf{w}}$ (for classification, $f_{\mathbf{w}}(x) = \mathbf{w} \cdot \phi(x)$). So the hypothesis class $\mathcal{F} = \{f_{\mathbf{w}}\}$ is all the predictors as $\mathbf{w}$ ranges. By controlling the number of possible values of $\mathbf{w}$ that the learning algorithm is allowed to choose from, we control the size of the hypothesis class and thus guard against overfitting.

Strategy 1: dimensionality

$$\mathbf{w} \in \mathbb{R}^d$$

Reduce the dimensionality d:



- One straightforward strategy is to change the dimensionality, which is the number of features. For example, linear functions are lower-dimensional than quadratic functions.

Controlling the dimensionality

Manual feature (template) selection:

- Add features if they help
- Remove features if they don't help

Automatic feature selection (beyond the scope of this class):

- Forward selection
- Boosting
- L_1 regularization

- Mathematically, you can think about removing a feature $\phi(x)_{37}$ as simply only allowing its corresponding weight to be zero ($w_{37} = 0$).
- Operationally, if you have a few feature templates, then it's probably easier to just manually include or exclude them — this will give you more intuition.
- If you have a lot of individual features, you can apply more automatic methods for selecting features, but these are beyond the scope of this class.

Strategy 2: norm

$$\mathbf{w} \in \mathbb{R}^d$$

Reduce the norm (length) $\|\mathbf{w}\|$:

[whiteboard: $x \mapsto w_1 x$]

Controlling the norm

Regularized objective:

$$\min_{\mathbf{w}} \text{TrainLoss}(\mathbf{w}) + \frac{\lambda}{2}\|\mathbf{w}\|^2$$

Algorithm: gradient descent

Initialize $\mathbf{w} = [0, \ldots, 0]$

For $t = 1, \ldots, T$:

$$\mathbf{w} \leftarrow \mathbf{w} - \eta(\nabla_{\mathbf{w}} [\text{TrainLoss}(\mathbf{w})] + \lambda \mathbf{w})$$

Same as gradient descent, except shrink the weights towards zero by λ.

- A related way to keep the weights small is called **regularization**, which involves adding an additional term to the objective function which penalizes the norm (length) of $\mathbf{w}$. This is probably the most common way to control the norm.
- This form of regularization is also known as L_2 regularization, or weight decay in deep learning literature.
- We can use gradient descent on this regularized objective, and this simply leads to an algorithm which subtracts a scaled down version of $\mathbf{w}$ in each iteration. This has the effect of keeping $\mathbf{w}$ closer to the origin than it otherwise would be.
- Note: Support Vector Machines are exactly hinge loss + regularization.

Controlling the norm: early stopping

Algorithm: gradient descent

Initialize $\mathbf{w} = [0, \ldots, 0]$

For $t = 1, \ldots, T$:

$\mathbf{w} \leftarrow \mathbf{w} - \eta \nabla_{\mathbf{w}} \text{TrainLoss}(\mathbf{w})$

Idea: simply make T smaller

Intuition: if have fewer updates, then $\|\mathbf{w}\|$ can't get too big.

Lesson: try to minimize the training error, but don't try too hard.

- A really cheap way to keep the weights small is to do **early stopping**. As we run more iterations of gradient descent, the objective function improves. If we cared about the objective function, this would always be a good thing. However, our true objective is not the training loss.
- Each time we update the weights, $\mathbf{w}$ has the potential of getting larger, so by running gradient descent a fewer number of iterations, we are implicitly ensuring that $\mathbf{w}$ stays small.
- Though early stopping seems hacky, there is actually some theory behind it. And one paradoxical note is that we can sometimes get better solutions by performing less computation.

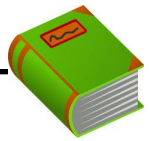
Summary so far

Key idea: keep it simple

Try to minimize training error, but keep the hypothesis class small.

- We've seen several ways to control the size of the hypothesis class (and thus reducing variance) based on either reducing the dimensionality or reducing the norm.
- It is important to note that what matters is the **size** of the hypothesis class, not how "complex" the predictors in the hypothesis class look. To put it another way, using complex features backed by 1000 lines of code doesn't hurt you if there are only 5 of them.
- Now the question is: how do we actually decide how big to make the hypothesis class, and in what ways (which features)?

Hyperparameters



Definition: hyperparameters

Properties of the learning algorithm (features, regularization parameter λ, number of iterations T, step size η, etc.).

How do we choose hyperparameters?

Choose hyperparameters to minimize $\mathcal{D}_{\text{train}}$ error? **No** - solution would be to include all features, set $\lambda = 0$, $T \to \infty$.

Choose hyperparameters to minimize $\mathcal{D}_{\text{test}}$ error? **No** - choosing based on $\mathcal{D}_{\text{test}}$ makes it an unreliable estimate of error!

Validation

Problem: can't use test set!

Solution: randomly take out 10-50% of training data and use it instead of the test set to estimate test error.

Definition: validation set

A **validation set** is taken out of the training data which acts as a surrogate for the **test set**.

- However, if we make the hypothesis class too small, then the approximation error gets too big. In practice, how do we decide the appropriate size? Generally, our learning algorithm has multiple **hyperparameters** to set. These hyperparameters cannot be set by the learning algorithm on the training data because we would just choose a degenerate solution and overfit. On the other hand, we can't use the test set either because then we would spoil the test set.
- The solution is to invent something that looks like a test set. There's no other data lying around, so we'll have to steal it from the training set. The resulting set is called the **validation set**.
- With this validation set, now we can simply try out a bunch of different hyperparameters and choose the setting that yields the lowest error on the validation set. Which hyperparameter values should we try? Generally, you should start by getting the right order of magnitude (e.g., $\lambda = 0.0001, 0.001, 0.01, 0.1, 1, 10$) and then refining if necessary.
- In K-fold **cross-validation**, you divide the training set into K parts. Repeat K times: train on $K-1$ of the parts and use the other part as a validation set. You then get K validation errors, from which you can report both the mean and the variance, which gives you more reliable information.

Development cycle

Problem: simplified named-entity recognition

Input: a string x (e.g., *Governor [Gavin Newsom] in*)

Output: y, whether x contains a person or not (e.g., $+1$)

Algorithm: recipe for success

- Split data into train, val, test
- Look at data to get intuition
- Repeat:
  - Implement feature / tune hyperparameters
  - Run learning algorithm
  - Sanity check train and val error rates, weights
  - Look at errors to brainstorm improvements
- Run on test set to get final error rates

[live solution]

- This slide represents the most important yet most overlooked part of machine learning: how to actually apply it in practice.
- We have so far talked about the mathematical foundation of machine learning (loss functions and optimization), and discussed some of the conceptual issues surrounding overfitting, generalization, and the size of hypothesis classes. But what actually takes most of your time is not writing new algorithms, but going through a **development cycle**, where you iteratively improve your system.
- Suppose you're given a binary classification task (backed by a dataset). What is the process by which you get to a working system? There are many ways to do this; here is one that I've found to be effective.
- The key is to stay connected with the data and the model, and have intuition about what's going on. Make sure to empirically examine the data before proceeding to the actual machine learning. It is imperative to understand the nature of your data in order to understand the nature of your problem. (You might even find that your problem admits a simple, clean solution sans machine learning.) Understanding trained models can be hard sometimes, as machine learning algorithms (even linear classifiers) are often not the easiest things to understand when you have thousands of parameters.

- First, maintain data hygiene. Hold out a test set from your data that you don't look at until you're done. Start by looking at the data to get intuition. You can start to brainstorm what features / predictors you will need. You can compute some basic statistics.
- Then you enter a loop: implement a new feature. There are three things to look at: error rates, weights, and predictions. First, sanity check the error rates and weights to make sure you don't have an obvious bug. Then do an **error analysis** to see which examples your predictor is actually getting wrong. The art of practical machine learning is turning these observations into new features.
- Finally, run your system once on the test set and report the number you get. If your test error is much higher than your validation error, then you probably did too much tweaking and were **overfitting** (at a meta-level) the validation set.

Roadmap

Generalization

Unsupervised learning

Summary

Supervision?

Supervised learning:

- Prediction: $\mathcal{D}_{\text{train}}$ contains input-output pairs (x, y)
- Fully-labeled data is very **expensive** to obtain (we can maybe get thousands of labeled examples)

Unsupervised learning:

- Clustering: $\mathcal{D}_{\text{train}}$ only contains inputs x
- Unlabeled data is much **cheaper** to obtain (we can maybe get billions of unlabeled examples)

- We have so far covered the basics of **supervised learning**. If you get a labeled training set of (x, y) pairs, then you can train a predictor. However, where do these examples (x, y) come from? If you're doing image classification, someone has to sit down and label each image, and generally this tends to be expensive enough that we can't get that many examples.
- On the other hand, there are tons of **unlabeled examples** sitting around (e.g., Flickr for photos, Wikipedia, news articles for text documents). The main question is whether we can harness all that unlabeled data to help us make better predictions? This is the goal of **unsupervised learning**.

[Brown et al, 1992]

Word clustering

Input: raw text (100 million words of news articles)...

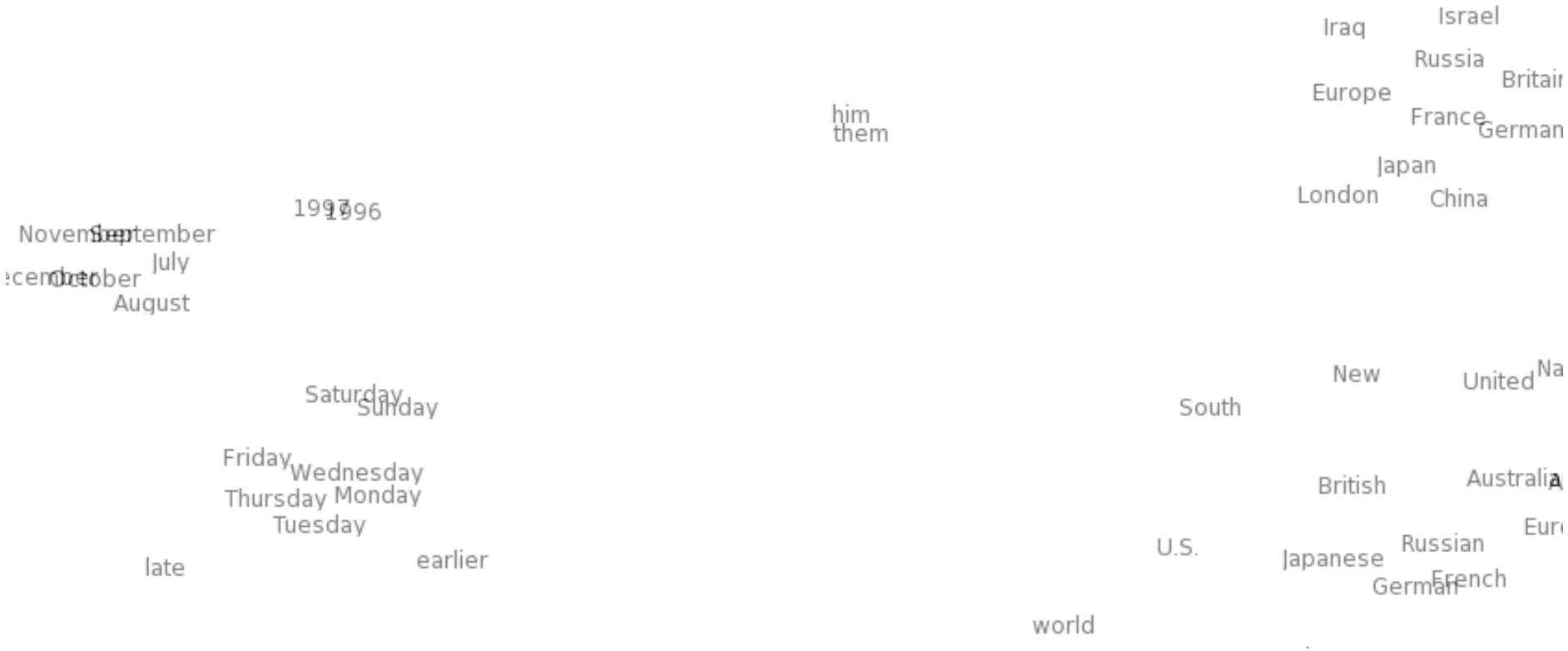
Output:

Cluster 1: Friday Monday Thursday Wednesday Tuesday Saturday Sunday weekends Sundays Saturdays
Cluster 2: June March July April January December October November September August
Cluster 3: water gas coal liquid acid sand carbon steam shale iron
Cluster 4: great big vast sudden mere sheer gigantic lifelong scant colossal
Cluster 5: man woman boy girl lawyer doctor guy farmer teacher citizen
Cluster 6: American Indian European Japanese German African Catholic Israeli Italian Arab
Cluster 7: pressure temperature permeability density porosity stress velocity viscosity gravity tension
Cluster 8: mother wife father son husband brother daughter sister boss uncle
Cluster 9: machine device controller processor CPU printer spindle subsystem compiler plotter
Cluster 10: John George James Bob Robert Paul William Jim David Mike
Cluster 11: anyone someone anybody somebody
Cluster 12: feet miles pounds degrees inches barrels tons acres meters bytes
Cluster 13: director chief professor commissioner commander treasurer founder superintendent dean custodian
Cluster 14: had hadn't hath would've could've should've must've might've
Cluster 15: head body hands eyes voice arm seat eye hair mouth

- Empirically, unsupervised learning has produced some pretty impressive results. HMMs (more specifically, Brown clustering) can be used to take a ton of raw text and cluster related words together.
- It is important to note that no one told the algorithm what days of the week were or months or family relations. The clustering algorithm discovered this structure automatically by simply examining the statistics of raw text.

[Mikolov et al., 2013]

Word vectors

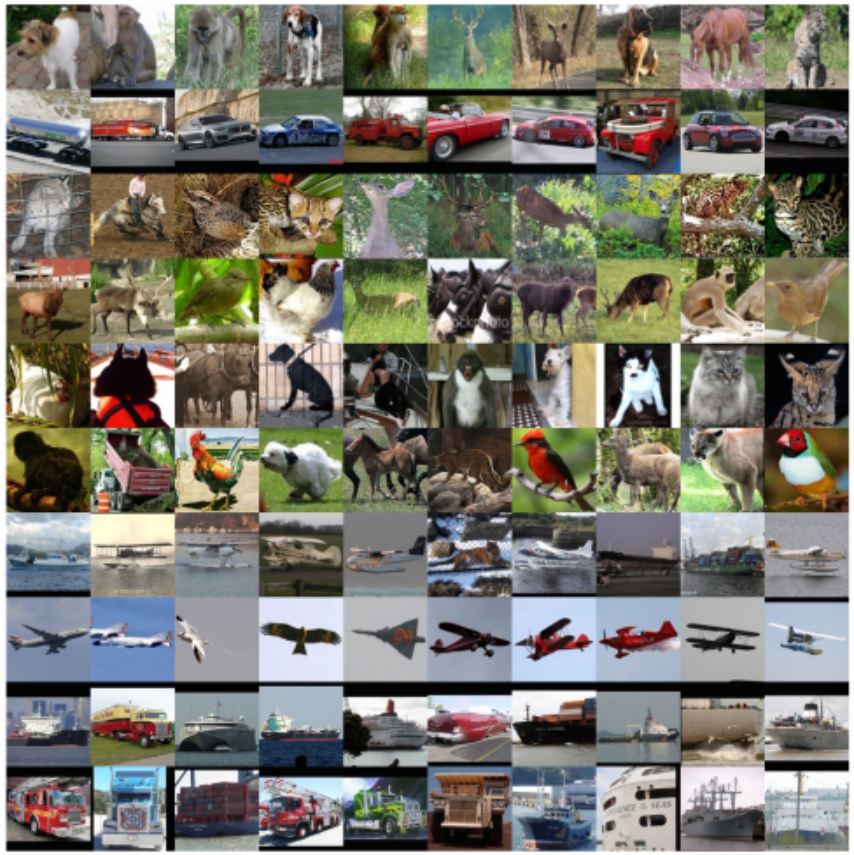


- A related idea are word vectors, which became popular after Tomas Mikolov created word2vec in 2013 (though the idea of vector space representations had been around for a while).
- Instead of representing a word by discrete clusters, a word is represented by a vector, which gives us a notion of similarity between words.
- More recently, **contextualized word representations** such as ELMo, BERT, XLNet, ALBERT, etc. have been very impactful. These methods also are unsupervised in that they only require raw text as input, but they produce representations of words in context. These representations essentially serve as good features for any NLP task, and empirically these methods have resulted in significant gains.

[Xie et al., 2015]

Clustering with deep embeddings

(a) MNIST



(b) STL-10

- In an example from vision, one can learn a feature representation (embedding) for images along with a clustering of them.

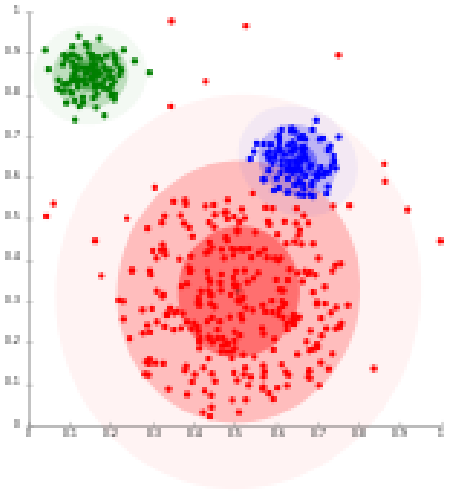
Key idea: unsupervised learning

Data has lots of rich **latent** structures; want methods to discover this **structure** automatically.

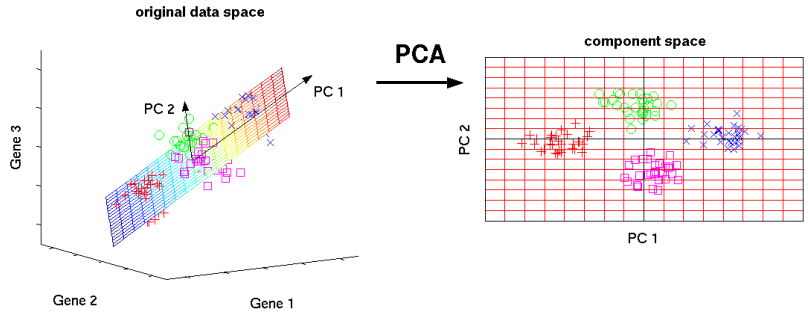
- Unsupervised learning in some sense is the holy grail: you don't have to tell the machine anything — it just "figures it out." However, one must not be overly optimistic here: there is no free lunch. You ultimately still have to tell the algorithm something, at least in the way you define the features or set up the optimization problem.

Types of unsupervised learning

Clustering (e.g., K-means):

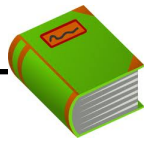


Dimensionality reduction (e.g., PCA):



- There are many forms of unsupervised learning, corresponding to different types of latent structures you want to pull out of your data. In this class, we will focus on one of them: clustering.

Clustering



Definition: clustering

Input: training set of input points

$$\mathcal{D}_{\text{train}} = \{x_1, \ldots, x_n\}$$

Output: assignment of each point to a cluster

$$[z_1, \ldots, z_n] \text{ where } z_i \in \{1, \ldots, K\}$$

Intuition: Want similar points to be in same cluster, dissimilar points to be in different clusters

[whiteboard]

- The task of clustering is to take a set of points as input and return a partitioning of the points into K clusters. We will represent the partitioning using an **assignment vector** $z = [z_1, \ldots, z_n]$. For each i, $z_i \in \{1, \ldots, K\}$ specifies which of the K clusters point i is assigned to.

K-means objective

Setup:

- Each cluster $k = 1, \ldots, K$ is represented by a **centroid** $\mu_k \in \mathbb{R}^d$
- Intuition: want each point $\phi(x_i)$ close to its assigned centroid μ_{z_i}

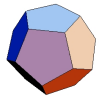
Objective function:

$$\text{Loss}_{\text{kmeans}}(z, \mu) = \sum_{i=1}^{n} \|\phi(x_i) - \mu_{z_i}\|^2$$

Need to choose centroids μ and assignments z **jointly**

- **K-means** is a particular method for performing clustering which is based on associating each cluster with a **centroid** μ_k for $k = 1, \ldots, K$. The intuition is to assign the points to clusters **and** place the centroid for each cluster so that each point $\phi(x_i)$ is close to its assigned centroid μ_{z_i}.

K-means: simple example



Example: one-dimensional

Input: $\mathcal{D}_{\text{train}} = \{0, 2, 10, 12\}$

Output: $K = 2$ centroids $\mu_1, \mu_2 \in \mathbb{R}$

If know centroids $\mu_1 = 1$, $\mu_2 = 11$:

$$z_1 = \arg\min\{(0-1)^2, (0-11)^2\} = 1$$
$$z_2 = \arg\min\{(2-1)^2, (2-11)^2\} = 1$$
$$z_3 = \arg\min\{(10-1)^2, (10-11)^2\} = 1$$
$$z_4 = \arg\min\{(12-1)^2, (12-11)^2\} = 1$$

If know assignments $z_1 = z_2 = 1$, $z_3 = z_4 = 2$:

$$\mu_1 = \arg\min_\mu (0-\mu)^2 + (2-\mu)^2 = 1$$
$$\mu_2 = \arg\min_\mu (10-\mu)^2 + (12-\mu)^2 = 11$$

- How do we solve this optimization problem? We can't just use gradient descent because there are discrete variables (assignment variables z_i). We can't really use dynamic programming because there are continuous variables (the centroids μ_k).
- To motivate the solution, consider a simple example with four points. As always, let's try to break up the problem into subproblems.
- What if we knew the optimal centroids? Then computing the assignment vectors is trivial (for each point, choose the closest center).
- What if we knew the optimal assignments? Then computing the centroids is also trivial (one can check that this is just averaging the points assigned to that center).
- The only problem is that we don't know the optimal centroids or assignments, and unlike in dynamic programming, the two depend on one another cyclically.

K-means algorithm

$$\min_{z} \min_{\mu} \text{Loss}_{\text{kmeans}}(z, \mu)$$

Key idea: alternating minimization

Tackle hard problem by solving two easy problems.

- And now the leap of faith is this: start with an arbitrary setting of the centroids (not optimal). Then alternate between choosing the best assignments given the centroids, and choosing the best centroids given the assignments. This is the K-means algorithm.

K-means algorithm (Step 1)

Goal: given centroids $\mu_1, \ldots, \mu_K$, assign each point to the best centroid.

Algorithm: Step 1 of K-means

For each point $i = 1, \ldots, n$:

Assign i to cluster with closest centroid:

$$z_i \leftarrow \arg\min_{k=1,\ldots,K} \|\phi(x_i) - \mu_k\|^2.$$

- **Step 1** of K-means fixes the centroids. Then we can optimize the K-means objective with respect to z alone quite easily. It is easy to show that the best label for z_i is the cluster k that minimizes the distance to the centroid μ_k (which is fixed).

K-means algorithm (Step 2)

Goal: given cluster assignments $z_1, \ldots, z_n$, find the best centroids $\mu_1, \ldots, \mu_K$.

Algorithm: Step 2 of K-means

For each cluster $k = 1, \ldots, K$:

Set μ_k to average of points assigned to cluster k:

$$\mu_k \leftarrow \frac{1}{|\{i : z_i = k\}|} \sum_{i:z_i=k} \phi(x_i)$$

- Now, turning things around, let's suppose we knew what the assignments z were. We can again look at the K-means objective function and try to optimize it with respect to the centroids μ. The best μ_k is to place the centroid at the average of all the points assigned to cluster k; this is **step two**.

K-means algorithm

Objective:

$$\min_{z} \min_{\mu} \text{Loss}_{\text{kmeans}}(z, \mu)$$

Algorithm: K-means

Initialize $\mu_1, \ldots, \mu_K$ randomly.

For $t = 1, \ldots, T$:

Step 1: set assignments z given μ

Step 2: set centroids μ given z

[demo]

- Now we have the two ingredients to state the full K-means algorithm. We start by initializing all the centroids randomly. Then, we iteratively alternate back and forth between steps 1 and 2, optimizing z given μ and vice-versa.

K-means: simple example

Example: one-dimensional

Input: $\mathcal{D}_{\text{train}} = \{0, 2, 10, 12\}$

Output: $K = 2$ centroids $\mu_1, \mu_2 \in \mathbb{R}$

Initialization (random): $\mu_1 = 0, \mu_2 = 2$

Iteration 1:

- Step 1: $z_1 = 1, z_2 = 2, z_3 = 2, z_4 = 2$
- Step 2: $\mu_1 = 0, \mu_2 = 8$

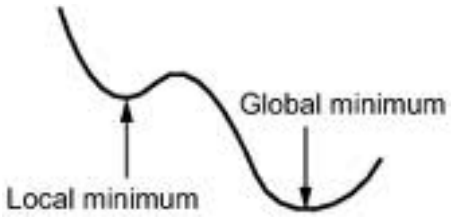
Iteration 2:

- Step 1: $z_1 = 1, z_2 = 1, z_3 = 2, z_4 = 2$
- Step 2: $\mu_1 = 1, \mu_2 = 11$

- Here is an example of an execution of K-means where we converged to the correct answer.

Local minima

K-means is guaranteed to converge to a local minimum, but is not guaranteed to find the global minimum.



[demo: getting stuck in local optima, seed = 100]

Solutions:

- Run multiple times from different random initializations
- Initialize with a heuristic (K-means++)

- K-means is guaranteed to decrease the loss function each iteration and will converge to a local minimum, but it is not guaranteed to find the global minimum, so one must exercise caution when applying K-means.
- One solution is to simply run K-means several times from multiple random initializations and then choose the solution that has the lowest loss.
- Or we could try to be smarter in how we initialize K-means. K-means++ is an initialization scheme which places centroids on training points so that these centroids tend to be distant from one another.

Unsupervised learning summary

- Leverage tons of unlabeled data

- Difficult optimization:

latent variables z  parameters μ

Roadmap

Generalization

Unsupervised learning

Summary

Summary

- Feature extraction (think hypothesis classes) [modeling]
- Prediction (linear, neural network, k-means) [modeling]
- Loss functions (compute gradients) [modeling]
- Optimization (stochastic gradient, alternating minimization) [learning]
- Generalization (think development cycle) [modeling]

- This concludes our tour of the foundations of machine learning, although machine learning will come up again later in the course. You should have gotten more than just a few isolated equations and algorithms. It is really important to think about the overarching principles in a modular way.
- First, feature extraction is where you put your domain knowledge into. In designing features, it's useful to think in terms of the induced **hypothesis classes** — what kind of functions can your learning algorithm potentially learn?
- These features then drive **prediction**: either linearly or through a neural network. We can even think of k-means as trying to predict the data points using the centroids.
- **Loss functions** connect predictions with the actual training examples.
- Note that all of the design decisions up to this point are about modeling. Algorithms are very important, but only come in once we have the right **optimization problem** to solve.
- Finally, machine learning requires a leap of faith. How does optimizing anything at training time help you **generalize** to new unseen examples at test time? Learning can only work when there's a common core that cuts past all the idiosyncrasies of the examples. This is exactly what features are meant to capture.

A brief history

1795: Gauss proposed least squares (astronomy)

1940s: logistic regression (statistics)

1952: Arthur Samuel built program that learned to play checkers (AI)

1957: Rosenblatt invented Perceptron algorithm (like SGD)

1969: Minsky and Papert "killed" machine learning

1980s: neural networks (backpropagation, from 1960s)

1990: interface with optimization/statistics, SVMs

2000s-: structured prediction, revival of neural networks, etc.

- Many of the ideas surrounding fitting functions was known in other fields long before computers, let alone AI.
- When computers arrived on the scene, learning was definitely on people's radar, although this was detached from the theoretical, statistical and optimization foundations.
- In 1969, Minsky and Papert wrote a famous book *Perceptrons*, which showed the limitations of linear classifiers with the famous XOR example (similar to our car collision example), which killed off this type of research. AI largely turned to symbolic methods.
- Since the 1980s, machine learning has increased its role in AI, been placed on a more solid mathematical foundation with its connection with optimization and statistics.
- While there is a lot of optimism today about the potential of machine learning, there are still a lot of unsolved problems.

Challenges

Capabilities:

- More complex prediction problems (translation, generation)
- Unsupervised learning: automatically discover structure

Responsibilities:

- Feedback loops: predictions affect user behavior, which generates data
- Fairness: build classifiers that don't discriminate?
- Privacy: can we pool data together
- Interpretability: can we understand what algorithms are doing?

- Going ahead, one major thrust is to improve the capabilities of machine learning. Broadly construed, machine learning is about learning predictors from some input to some output. The simplest case is when the output is just a label, but increasingly, researchers have been using the same machine learning tools for doing translation (output is a sentence), speech synthesis (output is a waveform), and image generation (output is an image).
- Another important direction is being able to leverage the large amounts of unlabeled data to learn good representations. Can we automatically discover the underlying structure (e.g., a 3D model of the world from videos)? Can we learn a causal model of the world? How can we make sure that the representations we are learning are useful for some other task?
- A second major thrust has to do with the context in which machine learning is now routinely being applied, for example in high-stakes scenarios such as self-driving cars. But machine learning does not exist in a vacuum. When machine learning systems are deployed to real users, it changes user behavior, and since the same systems are being trained on this user-generated data, this results in feedback loops.
- We also want to build ML systems which are fair. The real world is not fair; thus the data generated from it will reflect these discriminatory biases. Can we overcome these biases?
- The strength of machine learning lies in being able to aggregate information across many individuals. However, this appears to require a central organization that collects all this data, which seems like poor practice from the point of view of protecting privacy. Can we perform machine learning while protecting individual privacy? For example, local differential privacy mechanisms inject noise into an individual's measurement before sending it to the central server.
- Finally, there is the issue of trust of machine learning systems in high-stakes situations. As these systems become more complex, it becomes harder for humans to "understand" how and why a system is making a particular decision.

Machine learning

Key idea: learning

Programs should improve with experience.

So far: reflex-based models

Next time: state-based models

- If we generalize for a moment, machine learning is really about programs that can improve with experience.
- So far, we have only focused on reflex-based models where the program only outputs a yes/no or a number, and the experience is examples of input-output pairs.
- Next time, we will start looking at models which can perform higher-level reasoning, but machine learning will remain our companion for the remainder of the class.