Announcements

- No lecture on monday - catch up on HW2 + supplemental material

- Section this Friday at 3:15

Roadmap

Topics in the lecture:

Nearest neighbors

Generalization

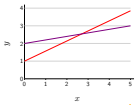
Best practices

Unsupervised learning

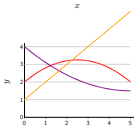
- In this lecture, I want to go through some more advanced and conceptual topics
- First we will discuss nearest neighbors - which are methods that can model arbitrarily complex functions
- Then we are going to ask - are more complex hypotheses classes always better
- Armed with some good conceptual knowledge from this discussion we will discuss some best practices
- Finally, we are going to change topics a bit and discuss what we should do if we dont have labeled data

Non-linear predictors (recap)

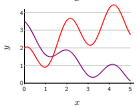
Linear predictors:
$f_{\mathbf{w}}(x) = \mathbf{w} \cdot \phi(x),\ \phi(x) = [1, x]$



Non-linear (quadratic) predictors:
$f_{\mathbf{w}}(x) = \mathbf{w} \cdot \phi(x),\ \phi(x) = [1, x, x^2]$



Non-linear neural networks:
$f_{\mathbf{w}}(x) = \mathbf{w} \cdot \sigma(\mathbf{V}\phi(x)),\ \phi(x) = [1, x]$



- Recall this slide from last lecture
- We build neural nets, which could flexibly model nonlinear functions
- You might believe that neural nets work precisely because they are so expressive
- We are going to see that is maybe not true

Nearest neighbors

Algorithm: nearest neighbors

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

Predictor $f(x')$:

- Find $(x, y) \in \mathcal{D}_{\text{train}}$ where $\|\phi(x) - \phi(x')\|$ is smallest
- Return y

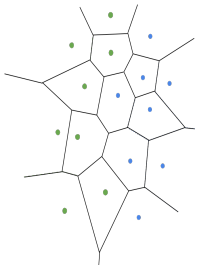
Key idea: similarity

Similar examples tend to have similar outputs.

- **Nearest neighbors** is perhaps conceptually one of the simplest learning algorithms. In a way, there is no learning. At training time, we just store the entire training examples. At prediction time, we get an input x' and we just find the input in our training set that is **most similar**, and return its output.
- The intuition being expressed here is that similar (nearby) points tend to have similar outputs. This is a reasonable assumption in most cases; all else equal, having a body temperature of 37 and 37.1 is probably not going to affect the health prediction by much.

Expressivity of nearest neighbors

Decision boundary: based on Voronoi diagram



Can model *arbitrary* smooth functions (is that better?):

- **Non-parametric**: the hypothesis class adapts to number of examples

- Let's look at the decision boundary of nearest neighbors. The input space is partitioned into regions, such that each region has the same closest point (this is called a Voronoi diagram), and each region could get a different output.
- In particular, one interesting property is that the complexity of the decision boundary adapts to the number of training examples. As we increase the number of training examples, the number of regions will also increase. Such methods are called **non-parametric**.
- This is an example of a model that is more expressive than a neural net - but does that mean its better?

Roadmap

Topics in the lecture:

Nearest neighbors

Generalization

Best practices

Unsupervised learning

- We saw how nearest neighbors could model arbitrarily complex functions
- But is more complex always better? why would we use a neural net over a linear function or a nearest-neighbor one?
- Generalization theory tells us the answers

Minimizing training loss

Hypothesis class:

$$f_{\mathbf{w}}(x) = \mathbf{w} \cdot \phi(x)$$

Training objective (loss function):

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

Optimization algorithm:

stochastic gradient descent

Is the training loss a good objective to optimize?

- Recall that our machine learning framework consists of specifying the hypothesis class, loss function, and the optimization algorithm.
- The hypothesis class could be linear predictors or neural networks. The loss function could be the hinge loss or the squared loss, which is averaged to produce the training loss.
- The default optimization algorithm is (stochastic) gradient descent.
- But let's be a bit more critical about the training loss. Is the training loss the right thing to optimize?

A strawman algorithm

Algorithm: rote learning

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

Predictor $f(x)$:

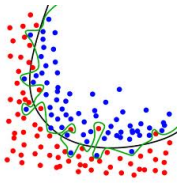
If $(x, y) \in \mathcal{D}_{\text{train}}$: return y.

Else: **segfault**.

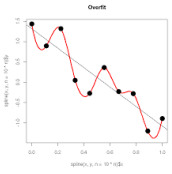
Minimizes the objective perfectly (zero), but clearly bad...

- Here is a strategy to consider: the rote learning algorithm, which just memorizes the training data and crashes otherwise.
- The rote learning algorithm does a perfect job of minimizing the training loss.
- But it's clearly a bad idea: It **overfits** to the training data and doesn't **generalize** to unseen examples.
- So clearly machine learning can't be about just minimizing the training loss.

Overfitting



Classification



Regression

- This is an extreme example of **overfitting**, which is when a learning algorithm outputs a predictor that does well on the training data but not well on new examples.
- Here are two pictures that illustrate what overfitting looks like 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

$\mathcal{D}_{\text{train}}$ → learning algorithm → f

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, machine learning is just a means to an end. What we're really doing is building a predictor to be deployed in the real world, and we just happen to be using machine learning. What we really care about is how accurate that predictor 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. We definitely should not tune our predictor based on the test set, 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 your predictor is actually doing.

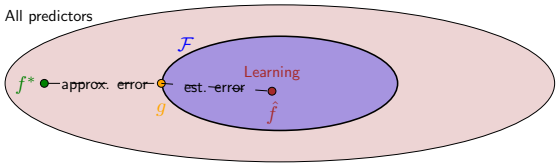
Generalization

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

$\mathcal{D}_{\text{train}}$ → $\mathcal{D}_{\text{test}}$

- 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

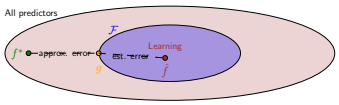


- Approximation error: how good is the hypothesis class?
- Estimation error: how good is the learned predictor **relative to** the potential of the hypothesis class?

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

- 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 $\mathcal{F}$ (e.g., 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

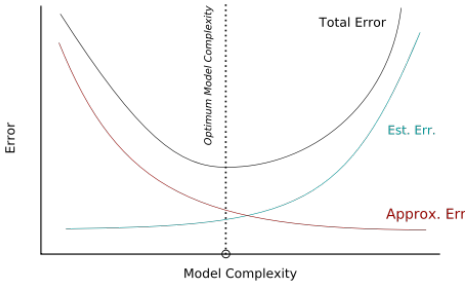
Estimation error increases because:

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).

The tradeoff



Classical generalization theory says there is an optimal model size

- we can visually see this tradeoff here
- this shows the tradeoff between approx error and estimation error
- notice how approx error goes down, and est error goes up with model size
- this results in a cup shape, where optimal model size is in the middle
- This is the visual version of what we talked about, and a major part of classical generalization theory
- Neural nets have somewhat upturned this view since they sometimes do better th bigger they are

Strategy 1: dimensionality

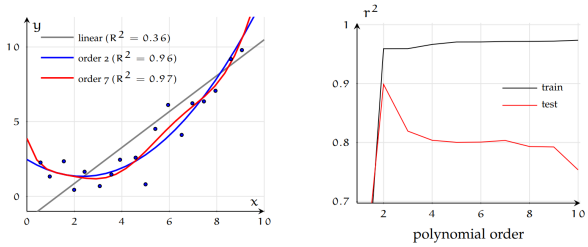
$$\mathbf{w} \in \mathbb{R}^d$$

Reduce the dimensionality d (number of features):

- Let's focus our attention to linear predictors. 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.
- One straightforward strategy is to change the dimensionality, which is the number of features. For example, linear functions are lower-dimensional than quadratic functions.

Example: polynomial features with linear regression

$$\phi(x) := [1, x, x^2 ...]$$



Accuracy improves .. to a point:

Approximation error and estimation error tradeoff

- lets walk through a simple example
- here is a linear regression problem, with polynomial features
- is more degrees of polynomial better?
- the example here is nonlinear, so cleary order 3 is better than 1
- on the right side, we see that train performance keeps going up
- but for the test set, there is an optimal degree of 2

Controlling the dimensionality

Manual feature (template) selection:

- Add feature templates if they help
- Remove feature templates if they don't help

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

- Forward selection
- Boosting
- L_1 regularization

It's the number of features that matters

- 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.
- An important point is that it's the number of features that matters, not the number of feature templates. (Can you define one feature template that results in severe overfitting?) Nor is it the amount of code that you have to write to generate a particular feature.

Strategy 2: norm

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

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

- 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.

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.

- 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 + L_2 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

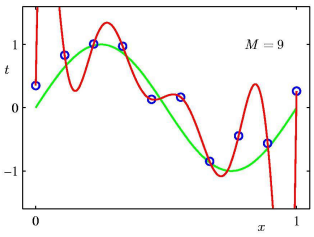
Intuition: if have fewer updates, then $\|\mathbf{w}\|$ can't get too big.

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.

Overfitting: a story in 3 slides

Lets try to fit a big polynomial:

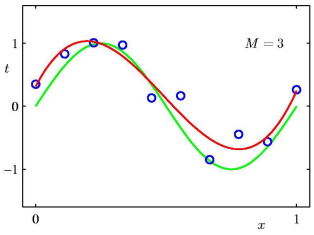


Overfitting happens with $\phi(x) := [1, x, x^2 ..., x^9]$

- lets put this all together into a simple story
- im fitting a regression model, using polynomial features
- i start out with degree 9, but this turns out to overfit.
- what should i do?

Overfitting: a story in 3 slides

Solution 1: reduce number of parameters

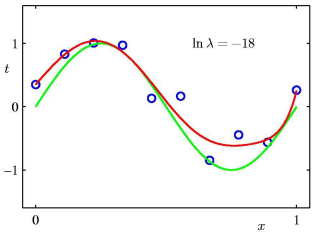


Good fit with $\phi(x) := [1, x, x^2, x^3]$

- we can first try to reduce the number of parameters
- a degree 3 polynomial fits pretty well

Overfitting: a story in 3 slides

Solution 2: use regularization

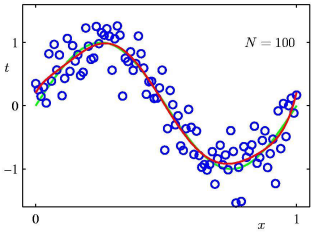


Make the weight norm small $\|w\|_2^2$

- i can also do something different, and use all 9 orders, but regularize instead
- since my weights are closer to zero, i can no longer perfectly fit my points
- i get a nice, well-fit curve even with degree 9

Bonus solution

Bonus solution: What if we just get more data?



Large amounts of data can reduce estimation error

- finally, its important to keep in mind that we can sometimes brute force this
- more data will inevitably push the estimation error down
- we see getting a hundred more points lets us fits the original model without issue

Summary

Not the real objective: training loss

Real objective: loss on unseen future examples

Semi-real objective: test loss

Key idea: keep it simple

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

- In summary, we started by noting that the training loss is not the objective. Instead it is minimizing unseen future examples, which is approximated by the test set provided you are careful.
- We've seen several ways to control the size of the hypothesis class (and thus reducing the estimation error) 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.
- So far, we've talked about the various knobs that we can turn to control the size of the hypothesis class, but how much do we turn each knob?

Roadmap

Topics in the lecture:

Nearest neighbors

Generalization

Best practices

Unsupervised learning

- Now how does generalization affect what we do in building machine systems?
- We are now going to discuss best practices in developing ML systems.

Choose your own adventure

Hypothesis class:

$f_{\mathbf{w}}(x) = \text{sign}(\mathbf{w} \cdot \phi(x))$

Feature extractor ϕ: linear, quadratic

Architecture: number of layers, number of hidden units

Training objective:

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

Loss function: hinge, logistic

Regularization: none, L2

Optimization algorithm:

Algorithm: stochastic gradient descent

Initialize $\mathbf{w} = [0, \ldots, 0]$
For $t = 1, \ldots, T$:
  For $(x, y) \in \mathcal{D}_{\text{train}}$:
    $\mathbf{w} \leftarrow \mathbf{w} - \eta \nabla_{\mathbf{w}} \text{Loss}(x, y, \mathbf{V}, \mathbf{w})$

Number of epochs

Step size: constant, decreasing, adaptive

Initialization: amount of noise, pre-training

Batch size

Dropout

- Recall that there are three design decisions for setting up a machine learning algorithm: the hypothesis class, the training objective, and the optimization algorithm.
- For the hypothesis class, there are two knobs you can turn. The first is the feature extractor ϕ (linear features, quadratic features, indicator features on regions, etc. The second is the architecture of the predictor: linear (one layer) or neural network with layers, and in the case of neural networks, how many hidden units (k) do we have.
- The second design decision is to specify the training objective, which we do by specifying the loss function depending how we want the predictor to fit our data, and also whether we want to regularize the weights to guard against overfitting.
- The final design decision is how to optimize the predictor. Even the basic stochastic gradient descent algorithm has at least two knobs: how long to train (number of epochs) and how aggressively to update (the step size). On top of that are many enhancements and tweaks common to training deep neural networks: changing the step size over time, perhaps adaptively, how we initialize the weights, whether we update on batches (say of 16 examples) instead of 1, and whether we apply dropout to guard against overfitting.
- So it is really a choose your own machine learning adventure. Sometimes decisions can be made via prior knowledge and are thoughtful (e.g., features that capture periodic trends). But in many (even most) cases, we don't really know what the proper values should be. Instead, we want a way to have these just set automatically.

Hyperparameters

Definition: hyperparameters

Design decisions (hypothesis class, training objective, optimization algorithm) that need to be made before running the learning algorithm.

How do we choose hyperparameters?

Choose hyperparameters to minimize $\mathcal{D}_{\text{train}}$ error?

No - optimum would be to include all features, no regularization, train forever

Choose hyperparameters to minimize $\mathcal{D}_{\text{test}}$ error?

No - choosing based on $\mathcal{D}_{\text{test}}$ makes it an unreliable estimate of error!

- Each of these many design decisions is a **hyperparameter**.
- We could choose the hyperparameters to minimize the training loss. However, this would lead to a degenerate solution. For example, by adding additional features, we can always decrease the training loss, so we would just end up adding all the features in the world, leading to a model that wouldn't generalize. We would turn off all regularization, because that just gets in the way of minimizing the training loss.
- What if we instead chose hyperparameters to minimize the test loss. This might lead to good hyperparameters, but is problematic because you then lose the ability to measure how well you're doing. Recall that the test set is supposed to be a surrogate for unseen examples, and the more you optimize over them, the less unseen they become.

Validation set

Definition: validation set

A **validation set** is taken out of the training set and used to optimize hyperparameters.

For each setting of hyperparameters, train on $\mathcal{D}_{\text{train}} \setminus \mathcal{D}_{\text{val}}$, evaluate on $\mathcal{D}_{\text{val}}$

- 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**.
- The size of the validation set should be large enough to give you a reliable estimate, but you don't want to take away too many examples from the training 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 compute and report both the mean and the variance, which gives you more reliable information.

Model development strategy

Algorithm: Model development strategy

- Split data into train, validation, test
- Look at data to get intuition
- Repeat:
  - Implement model/feature, adjust hyperparameters
  - Run learning algorithm
  - Sanity check train and validation error rates
  - Look at weights and prediction errors
- Evaluate on test set to get final error rates

- 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.
- 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.
- 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 (training or validation) 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 model architecture or feature template. 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.

Tips

Start simple:

- Run on small subsets of your data or synthetic data
- Start with a simple baseline model
- Sanity check: can you overfit 5 examples

Log everything:

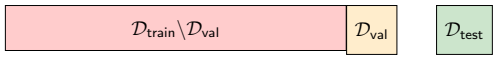
- Track training loss and validation loss over time
- Record hyperparameters, statistics of data, model, and predictions
- Organize experiments (each run goes in a separate folder)

Report your results:

- Run each experiment multiple times with different random seeds
- Compute multiple metrics (e.g., error rates for minority groups)

- There is more to be said about the practice of machine learning. Here are some pieces of advice. Note that many are simply related to good software engineering practices.
- First, don't start out by coding up a large complex model and try running it on a million examples. Start simple, both with the data (small number of examples) and the model (e.g., linear classifier). Sanity check that things are working first before increasing the complexity. This will help you debug in a regime where things are more interpretable and also things run faster. One sanity check is to train a sufficiently expressive model on a very few examples and see if the model can overfit the examples (get zero training error). This does not produce a useful model, but is a diagnostic to see if the optimization is working. If you can't overfit on 5 examples, then you have a problem: maybe the hypothesis class is too small, the data is too noisy, or the optimization isn't working.
- Second, log everything so you can diagnose problems. Monitor the losses over epochs. It is also important to track the training loss so that if you get bad results, you can find out if it is due to bad optimization or overfitting. Record all the hyperparameters, so that you have a full record of how to reproduce the results.
- Third, when you report your results, you should be able to run an experiment multiple times with different randomness to see how stable the results are. Report error bars. And finally, if it makes sense for your application to report more than just a single test accuracy. Report the errors for minority groups and add if your model is treating every group fairly.

Summary



Don't look at the test set!

Understand the data!

Start simple!

Practice!

- To summarize, we've talked about the practice of machine learning.
- First, make sure you follow good data hygiene, separating out the test set and don't look at it.
- But you should look at the training or validation set to get intuition about your data before you start.
- Then, start simple and make sure you understand how things are working.
- Beyond that, there are a lot of design decisions to be made (hyperparameters). So the most important thing is to practice, so that you can start developing more intuition, and developing a set of best practices that works for you.

Roadmap

Topics in the lecture:

Nearest neighbors

Generalization

Best practices

Unsupervised learning

- Finally, we are going to change topics a bit and disuss unsupervised learning
- This is a whole different kind of machine learning than the regression and classification settings we considered

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:

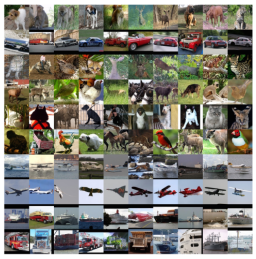
- Clustering: $\mathcal{D}_{\text{train}}$ only contains inputs x
- Unlabeled data is much **cheaper** to obtain (we can maybe get billions of unlabeled examples)

- 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**.

[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.

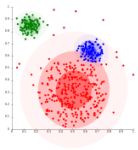
Key idea: unsupervised learning

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

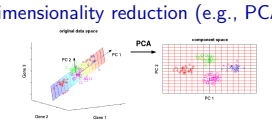
- Unsupervised learning in some sense is the holy grail: you don't have to tell the machine anything — it just "figures it out." However, one must not be overly optimistic here: there is no free lunch. You ultimately still have to tell the algorithm something, at least in the way you define the features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

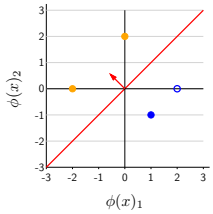
Classification (supervised learning)

training data $\mathcal{D}_{\text{train}}$

$\phi(x)_1$	$\phi(x)_2$	y
0	2	1
-2	0	1
1	-1	-1

learning algorithm → f classifier

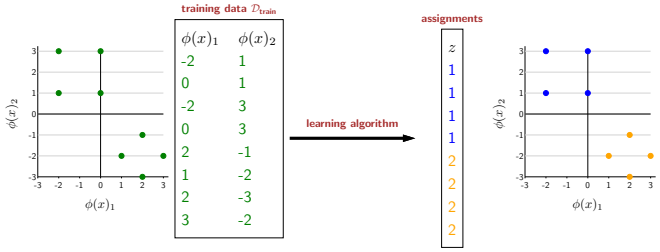
[2, 0] → -1



Labeled data is expensive to obtain

- I want to contrast unsupervised learning with supervised learning.
- Recall that in classification you're given a set of **labeled** training examples.
- A learning algorithm produces a classifier that can classify new points.
- Note that we're now plotting the (two-dimensional) feature vector rather than the raw input, since the learning algorithms only depend on the feature vectors.
- However, the main challenge with supervised learning is that it can be expensive to collect the labels for data.

Clustering (unsupervised learning)



Intuition: Want to assign nearby points to same cluster

Unlabeled data is very cheap to obtain

- In contrast, in clustering, you are only given **unlabeled** training examples.
- Our goal is to assign each point to a cluster. In this case, there are two clusters, 1 (blue) and 2 (orange).
- Intuitively, nearby points should be assigned to the same cluster.
- The advantage of unsupervised learning is that unlabeled data is often very cheap and almost free to obtain, especially text or images on the web.

Clustering task

Definition: clustering

Input: training points

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

Output: assignment of each point to a cluster

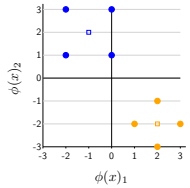
$z = [z_1, \ldots, z_n]$ where $z_i \in \{1, \ldots, K\}$

- Formally, 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.

Centroids

Each cluster $k = 1, \ldots, K$ is represented by a **centroid** $\mu_k \in \mathbb{R}^d$

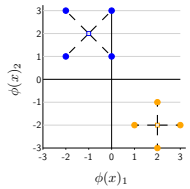
$$\mu = [\mu_1, \ldots, \mu_K]$$



Intuition: want each point $\phi(x_i)$ to be close to its assigned centroid μ_{z_i}

- What makes a cluster? The key assumption is that each cluster k is represented by a **centroid** μ_k.
- Now the intuition is that we want each point $\phi(x_i)$ to be close to its assigned centroid μ_{z_i}.

K-means objective

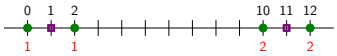


$$\text{Loss}_{\text{kmeans}}(z, \mu) = \sum_{i=1}^{n} \|\phi(x_i) - \mu_{z_i}\|^2$$

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

- To formalize this, we define the K-means objective (distinct from the K-means algorithm).
- The variables are the assignments z and centroids μ.
- We examine the squared distance (dashed lines) from a point $\phi(x_i)$ to the centroid of its assigned cluster μ_{z_i}. Summing over all these squared distances gives the K-means objective.
- This loss can be interpreted as a reconstruction loss: imagine replacing each data point by its assigned centroid. Then the objective captures how lossy this compression was.
- Now our goal is to minimize the K-means loss.

Alternating minimization from optimum



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

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

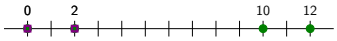
If know assignments $z_1 = z_2 = 1$, $z_3 = z_4 = 2$:

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

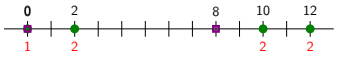
- Before we present the K-means algorithm, let us form some intuitions.
- Consider the following one-dimensional clustering problem with 4 points. Intuitively there are two clusters.
- Suppose we know the centroids. Then for each point the assignment that minimizes the K-means loss is the closer of the two centroids.
- Suppose we know the assignments. Then for each cluster, we average the points that are assigned to that cluster.

Alternating minimization from random initialization

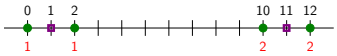
Initialize μ:



Iteration 1:



Iteration 2:



Converged.

- But of course we don't know either the centroids or assignments.
- So we simply start with an arbitrary setting of the centroids.
- 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

Algorithm: K-means

Initialize $\mu = [\mu_1, \ldots, \mu_K]$ randomly.
For $t = 1, \ldots, T$:
  Step 1: set assignments z given μ
    For each point $i = 1, \ldots, n$:
      $z_i \leftarrow \arg\min_{k=1,\ldots,K} \|\phi(x_i) - \mu_k\|^2$
  Step 2: set centroids μ given z
    For each cluster $k = 1, \ldots, K$:
      $\mu_k \leftarrow \frac{1}{|\{i : z_i = k\}|} \sum_{i:z_i=k} \phi(x_i)$

- Now we can state the K-means algorithm formally. 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.
- **Step 1** of K-means fixes the centroids μ. 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).
- **Step 2** turns things around and fixes the assignments z. We can again look at the K-means objective function and 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.

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:

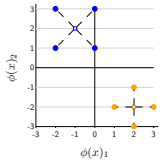
- Run multiple times from different random initializations
- Initialize with a heuristic (K-means++)

- 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.
- Advanced: One solution is to simply run K-means several times from multiple random initializations and then choose the solution that has the lowest loss.
- Advanced: 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.

Summary

Clustering: discover structure in unlabeled data

K-means objective:



assignments z

K-means algorithm:

centroids μ

Unsupervised learning use cases:

- Data exploration and discovery
- Providing representations to downstream supervised learning

- In summary, K-means is a simple and widely-used method for discovering cluster structure in data.
- Note that K-means can mean two things: the objective and the algorithm.
- Given points we define the K-means objective as the sum of the squared differences between a point and its assigned centroid.
- We also defined the K-means algorithm, which performs alternating optimization on the K-means objective.
- Finally, clustering is just one instance of unsupervised learning, which seeks to learn models from the wealth of unlabeled data alone. Unsupervised learning can be used in two ways: exploring a dataset which has not been labeled (let the data speak), and learning representations (discrete clusters or continuous embeddings) useful for downstream supervised applications.