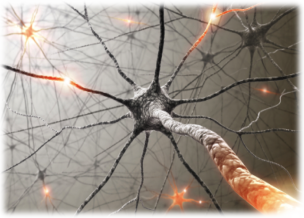
Lecture 5.1: Machine learning IV



Review: Features

Feature extractor ϕ:

abc@gmail.com → **feature extractor** (arbitrary!) →

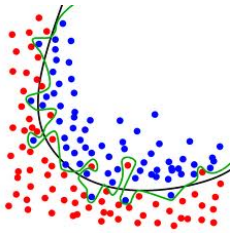
length>10	: 1
fracOfAlpha	: 0.85
contains_@	: 1
endsWith_com	: 1
endsWith_org	: 0

Prediction score:

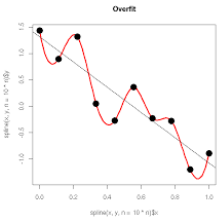
- Linear predictor: score $= \mathbf{w} \cdot \phi(x)$

- First a review: 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: Overfitting



Classification



Regression

- More complex features let us build more expressive models, but they also increase the risk of overfitting to spurious patterns in the training data.

Review: Evaluation

$\mathcal{D}_{\text{train}}$ → Learner → f

Key idea: the real learning objective

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

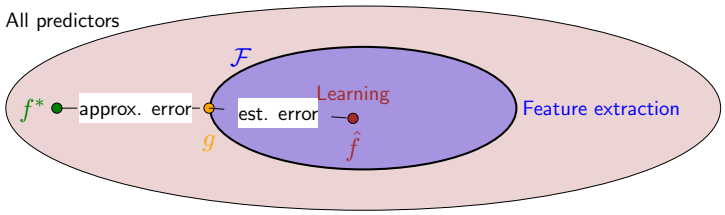
Don't have unseen examples; next best thing:

Definition: test set

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

- Overfitting happens when the learner minimizes training error in a way that doesn't minimize the true objective: error on future examples. Since we can't directly query the future, the best proxy we have is to create a test set $\mathcal{D}_{\text{test}}$ which is disjoint from the training set $\mathcal{D}_{\text{train}}$. We train on the training set and then evaluate error on the test set.

Review: Approximation and Estimation



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

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

- Finally, we saw the idea of the trade-off between approximation and estimation error. Approximation error measures how well anything in the hypothesis class can approximate the best possible predictor. Estimation error measures how well the learner can actually find a good predictor within the hypothesis class.
- By using a larger hypothesis class (e.g., more complicated features), we can decrease approximation error. However, this may make it more difficult for the learner to find the best predictor, so estimation error may increase.

Roadmap

Regulariztion

Best practices

Unsupervised learning

Controlling size of hypothesis class

Linear predictors are specified by weight vector $\mathbf{w} \in \mathbb{R}^d$

Keeping the dimensionality d small:

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

[whiteboard: $x \mapsto w_1 x$]

- 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.
- Last time, we saw that limiting the number of features (i.e., keeping d small) is one way to control the hypothesis class size.
- Another option is to keep the norm $\|\mathbf{w}\|$ (length of $\mathbf{w}$) small. Intuitively, lower norm values of $\mathbf{w}$ correspond to less complicated predictors.

Controlling the norm: regularization

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.

Note: SVM = hinge loss + regularization

- 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.
- 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.

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)?

Roadmap

Regulariztion

Best practices

Unsupervised learning

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 (development) 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.

Development cycle

Problem: simplified named-entity recognition

Input: a string x (e.g., *President [Barack Obama] in*)
Output: y, whether x contains a person or not (e.g., $+1$)

Algorithm: recipe for success

- Split data into train, dev, test
- Look at data to get intuition
- Repeat:
  - Implement feature / tune hyperparameters
  - Run learning algorithm
  - Sanity check train and dev 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, 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 development error, then you probably did too much tweaking and were **overfitting** (at a meta-level) the development set.

Summary

- Test set: only for final evaluation
- Use validation set to tune hyperparameters
- Look at the data!

Announcements

- Section tomorrow: machine learning review + Scikit-learn
- Assignment 3 (sentiment): due next Tuesday

Roadmap

Regulariztion

Best practices

Unsupervised learning

Supervision?

Supervised learning:

- Prediction: $\mathcal{D}_{\text{train}}$ contains input-output pairs (x, y)
- Fully-labeled data is very **expensive** to obtain (we can get 10,000 labeled examples)

Unsupervised learning:

- Clustering: $\mathcal{D}_{\text{train}}$ only contains inputs x
- Unlabeled data is much **cheaper** to obtain (we can get 100 million 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 using HMMs

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

Impact: used in many state-of-the-art NLP systems

- 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. Word vectors (e.g., word2vec) do something similar.

[Le et al, 2012]

Feature learning using neural networks

Input: 10 million images (sampled frames from YouTube)

Output:

Impact: state-of-the-art results on object recognition (22,000 categories)

- An unsupervised variant of neural networks called autoencoders can be used to take a ton of raw images and output clusters of images. No one told the learning algorithms explicitly what the clusters should look like — they just figured it out.

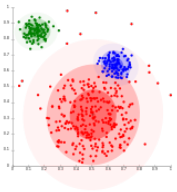
Key idea: unsupervised learning

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

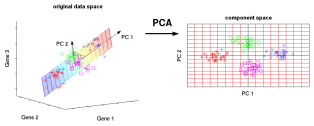
- Unsupervised learning in some sense is the holy grail: you don't have to tell the machine anything — it just "figures it out." However, one must not be overly optimistic here: there is no free lunch. You ultimately still have to tell the algorithm something, at least in the way you define the features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

$[z_1, \ldots, z_n]$ 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 $\mu_1 = 1$, $\mu_2 = 11$:

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

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

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

- 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: $\mu_1 = 0, \mu_2 = 8$

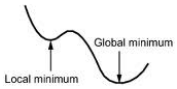
Iteration 2:

- 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++)

Unsupervised learning summary

- Leverage tons of unlabeled data

- Difficult optimization:

latent variables z  parameters μ

- 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.