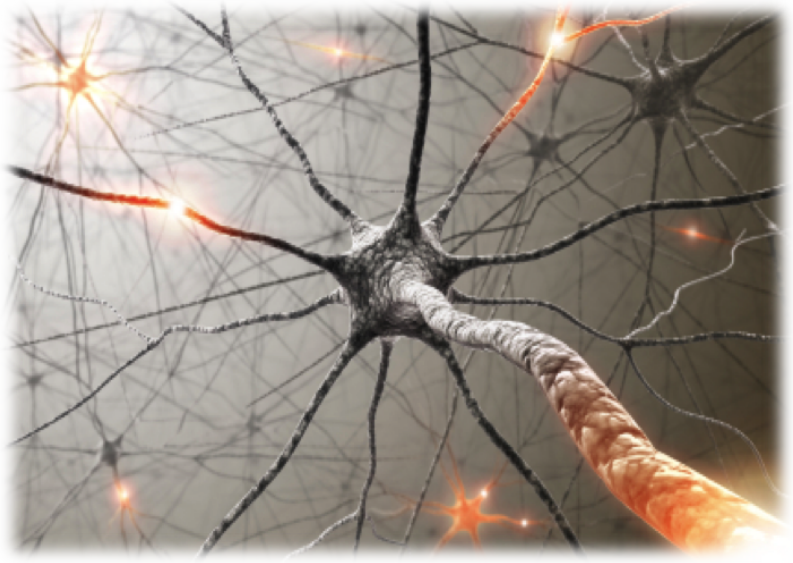
Lecture 4.1: Machine learning III



Question

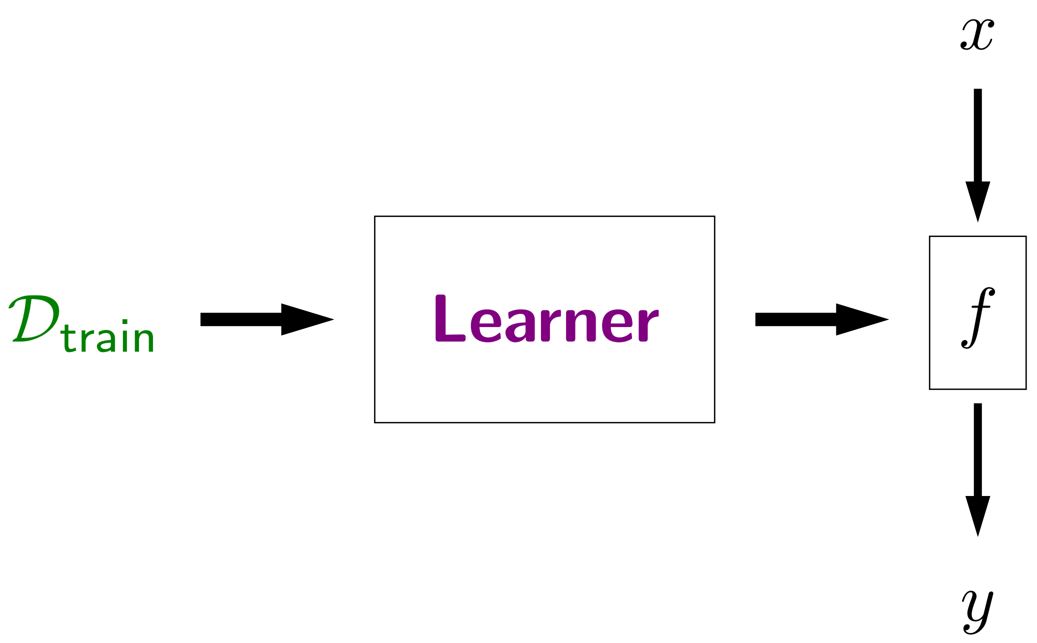
Can we obtain decision boundaries which are circles by using linear classifiers?

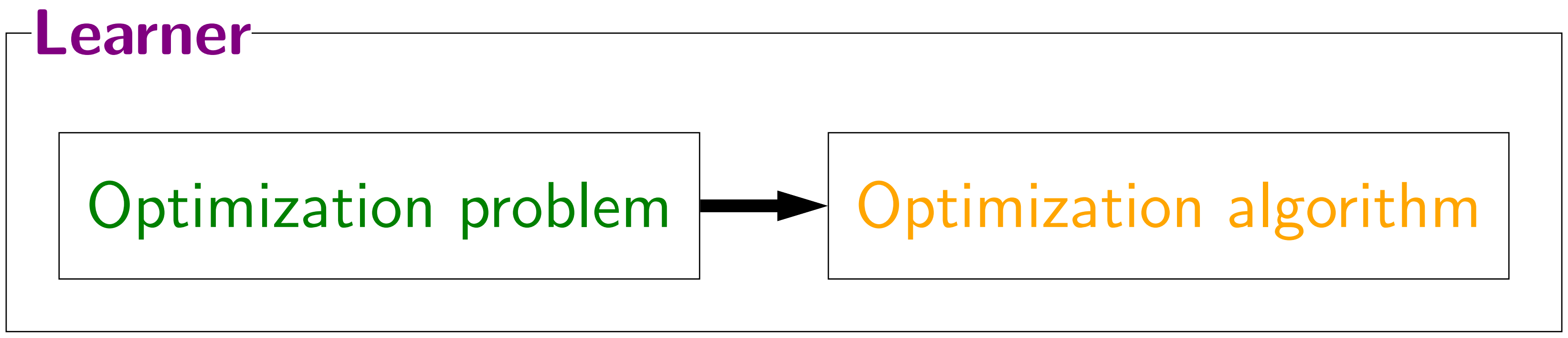
Yes

No

- The answer is yes. This might seem paradoxical since we are only working with linear classifiers. But as we will see later, **linear** refers to the relationship between the weight vector $\mathbf{w}$ and the prediction score (not the input x, which might not even be a real vector), whereas the decision boundary refers to how the prediction varies as a function of x.

Framework


x
$\mathcal{D}_{\text{train}}$
Learner
f
y


Learner
Optimization problem
Optimization algorithm

- Recall our overall learning framework. We want to learn a linear predictor f, which is parameterized by a weight vector w. Given an input x, linear predictors compute a score $\mathbf{w} \cdot \phi(x)$, which can be used for classification or regression.
- Today we will think a lot about how to construct ϕ, the function that maps input x into a feature vector $\in \mathbb{R}^d$.

Review: optimization problem

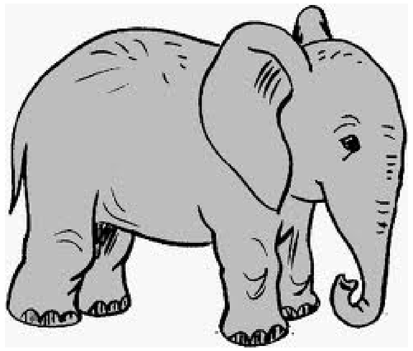
Key idea: minimize training loss

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

$$\min_{\mathbf{w}\in\mathbb{R}^d} \text{TrainLoss}(\mathbf{w})$$

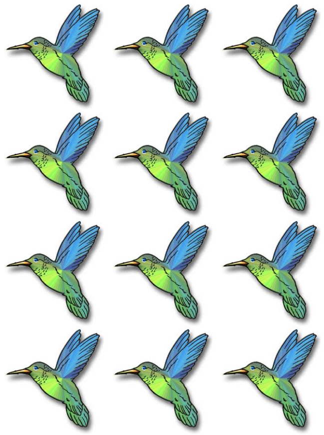
- To learn w, we minimized the training loss, which is the average loss over all the training examples. Note that our real goal is to find a w that has low loss on **future** examples; we approximate this by minimizing loss on training examples.

Review: optimization algorithms



Algorithm: gradient descent

```
Initialize $\mathbf{w} = [0, \dots, 0]$
For $t = 1, \dots, T$:
    $\mathbf{w} \leftarrow \mathbf{w} - \eta_t \nabla_{\mathbf{w}} \text{TrainLoss}(\mathbf{w})$
```



Algorithm: stochastic gradient descent

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

- We introduced two very simple algorithms to minimize the training loss, both based on iteratively computing the gradient of the objective with respect to the parameters $\mathbf{w}$ and stepping in the opposite direction of the gradient. Think about a ball at the current weight vector and rolling it down on the surface of the training loss objective.
- Gradient descent (GD) computes the gradient of the full training loss, which can be slow for large datasets.
- Stochastic gradient descent (SGD), which approximates the gradient of the training loss with the loss at a single example, generally takes less time.
- In both cases, one must be careful to set the step size η properly (not too big, not too small).

Roadmap

Features

Generalization

- The first half of this lecture is about thinking about the feature extractor ϕ. Features are a critical part of machine learning which often does not get as much attention as it deserves. Ideally, they would be given to us by a domain expert, and all we (as machine learning people) have to do is to stick them into our learning algorithm. While one can get considerable mileage out of doing this, the interface between general-purpose machine learning and domain knowledge is often nuanced, so to be successful, it pays to understand this interface.
- In the second half of this lecture, we will turn to the question of generalization: how do we build models that generalize from the training examples to yet-unseen test examples? We will learn about an important trade-off between generalization and model *expressivity*: more complicated feature extractors allow for more complicated models, but can also make it harder to find w's that generalize well.

Two components

Score (drives prediction):

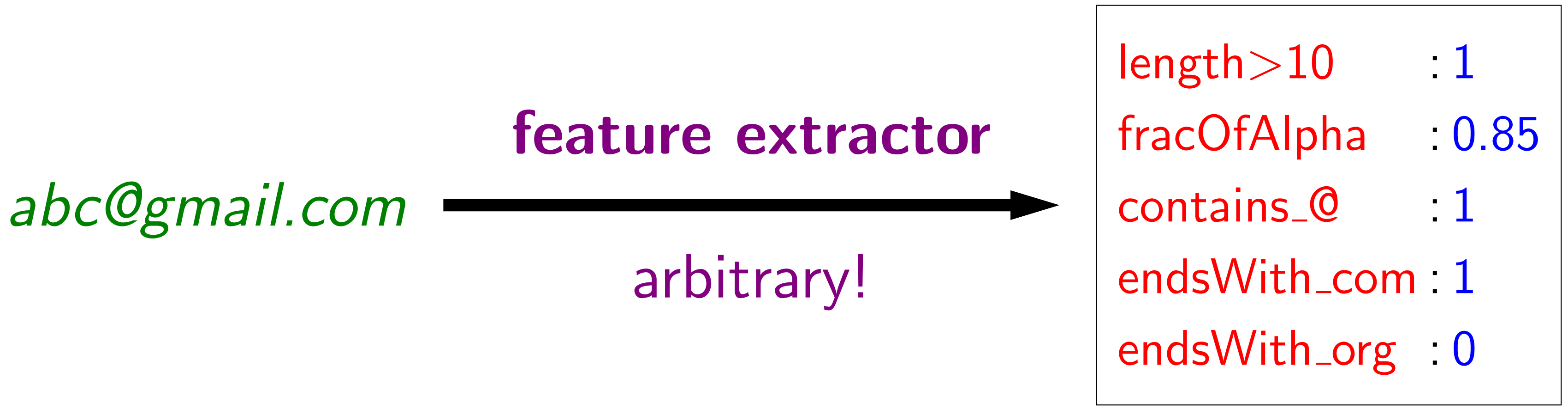
$$\mathbf{w} \cdot \phi(x)$$

- Previous: learning sets $\mathbf{w}$ via optimization
- Next: **feature extraction** specifies $\phi(x)$ based on domain knowledge

- As a reminder, the prediction is driven by the score $\mathbf{w} \cdot \phi(x)$. In regression, we predict the score directly, and in binary classification, we predict the sign of the score.
- Both $\mathbf{w}$ and $\phi(x)$ play an important role in prediction. So far, we have fixed $\phi(x)$ and used learning to set $\mathbf{w}$. Now, we will explore how $\phi(x)$ affects the prediction.

Organization of features

Task: predict whether a string is an email address



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

Which features to include? Need an organizational principle...

- How would we go about creating good features?
- Here, we used our prior knowledge to define certain features (contains_@) which we believe are helpful for detecting email addresses.
- But this is ad-hoc: which strings should we include? We need a more systematic way to go about this.

Feature templates

Definition: feature template (informal)

A **feature template** is a group of features all computed in a similar way.

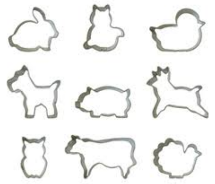
Input:

abc@gmail.com

Some feature templates:

- Length greater than ___
- Last three characters equals ___
- Contains character ___
- Pixel intensity of position ___, ___

- A useful organization principle is a **feature template**, which groups all the features which are computed in a similar way. (People often use the word "feature" when they really mean "feature template".)
- A feature template also allows us to define a set of related features (contains_@, contains_a, contains_b). This reduces the amount of burden on the feature engineer since we don't need to know which particular characters are useful, but only that existence of certain single characters is a useful cue to look at.
- We can write each feature template as a English description with a blank (___), which is to be filled in with an arbitrary string. Also note that feature templates are most natural for defining binary features, ones which take on value 1 (true) or 0 (false).
- Note that an isolated feature (fraction of alphanumeric characters) can be treated as a trivial feature template with no blanks to be filled.
- As another example, if x is a $k \times k$ image, then $\{\text{pixelIntensity}_{ij} : 1 \leq i, j \leq k\}$ is a feature template consisting of k^2 features, whose values are the pixel intensities at various positions of x.



Feature templates

Feature template: last three characters equals ___

abc@gmail.com $\longrightarrow$

endsWith_aaa	: 0
endsWith_aab	: 0
endsWith_aac	: 0
...	
endsWith_com	: 1
...	
endsWith_zzz	: 0

- This is an example of one feature template mapping onto a group of m^3 features, where m (26 in this example) is the number of possible characters.

Sparsity in feature vectors

Feature template: last character equals ___

abc@gmail.com $\longrightarrow$

endsWith_a	: 0
endsWith_b	: 0
endsWith_c	: 0
endsWith_d	: 0
endsWith_e	: 0
endsWith_f	: 0
endsWith_g	: 0
endsWith_h	: 0
endsWith_i	: 0
endsWith_j	: 0
endsWith_k	: 0
endsWith_l	: 0
endsWith_m	: **1**
endsWith_n	: 0
endsWith_o	: 0
endsWith_p	: 0
endsWith_q	: 0
endsWith_r	: 0
endsWith_s	: 0
endsWith_t	: 0
endsWith_u	: 0
endsWith_v	: 0
endsWith_w	: 0
endsWith_x	: 0
endsWith_y	: 0
endsWith_z	: 0

Inefficient to represent all the zeros...

- In general, a feature template corresponds to many features. It would be inefficient to represent all the features explicitly. Fortunately, the feature vectors are often **sparse**, meaning that most of the feature values are 0. It is common for all but one of the features to be 0. This is known as a **one-hot representation** of a discrete value such as a character.

Feature vector representations

fracOfAlpha	: 0.85
contains_a	: 0
...	
contains_@	: 1
...	

Array representation (good for dense features):

```
[0.85, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0]
```

Map representation (good for sparse features):

```
{"fracOfAlpha": 0.85, "contains_@": 1}
```

- Let's now talk a bit more about implementation. There are two common ways to define features: using arrays or using maps.
- **Arrays** assume a fixed ordering of the features and represent the feature values as an array. This representation is appropriate when the number of nonzeros is significant (the features are dense). Arrays are especially efficient in terms of space and speed (and you can take advantage of GPUs). In computer vision applications, features (e.g., the pixel intensity features) are generally dense, so array representation is more common.
- However, when we have sparsity (few nonzeros), it is typically more efficient to represent the feature vector as a **map** from strings to doubles rather than a fixed-size array of doubles. The features not in the map implicitly have a default value of zero. This sparse representation is very useful in natural language processing, and is what allows us to work effectively over trillions of features. In Python, one would define a feature vector $\phi(x)$ as {`"endsWith_"+x[-3:]: 1`}. Maps do incur extra overhead compared to arrays, and therefore maps are much slower when the features are not sparse.
- Finally, it is important to be clear when describing features. Saying "length" might mean that there is one feature whose value is the length of x or that there could be a feature template "length is equal to ___". These two encodings of the same information can have a drastic impact on prediction accuracy when using a linear predictor, as we'll see later.

Hypothesis class

Predictor:

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

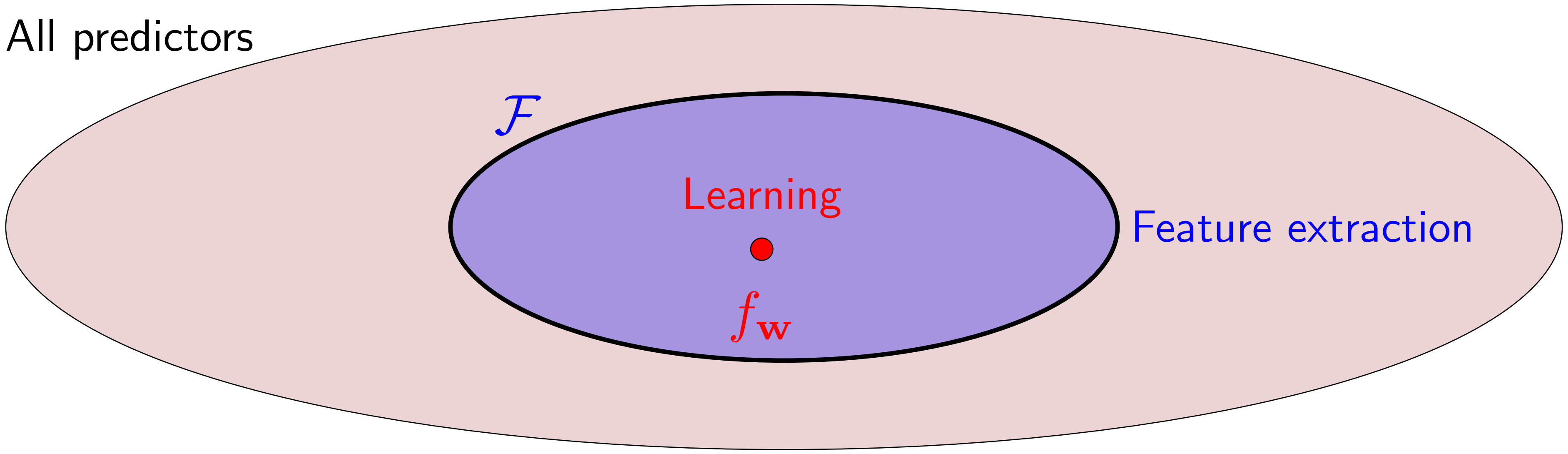
Definition: hypothesis class

A **hypothesis class** is the set of possible predictors with a fixed $\phi(x)$ and varying $\mathbf{w}$:

$$\mathcal{F} = \{f_{\mathbf{w}} : \mathbf{w} \in \mathbb{R}^d\}$$

- Having discussed how feature templates can be used to organize groups of feature and allow us to leverage sparsity, let us further study how features impact prediction.
- The key notion is that of a **hypothesis class**, which is the set of all possible predictors that you can get by varying the weight vector $\mathbf{w}$. Thus, the feature extractor ϕ specifies a hypothesis class $\mathcal{F}$. This allows us to take data and learning out of the picture.

Feature extraction + learning



- Feature extraction: set $\mathcal{F}$ based on domain knowledge
- Learning: set $f_{\mathbf{w}} \in \mathcal{F}$ based on data

- Stepping back, we can see the two stages more clearly. First, we perform feature extraction (given domain knowledge) to specify a hypothesis class $\mathcal{F}$. Second, we perform learning (given training data) to obtain a particular predictor $f_{\mathbf{w}} \in \mathcal{F}$.
- Note that if the hypothesis class doesn't contain any good predictors, then no amount of learning can help. So the question when extracting features is really whether they are powerful enough to **express** predictors which are good? It's okay and expected that $\mathcal{F}$ will contain a bunch of bad ones as well.
- Later, we'll see reasons for keeping the hypothesis class small (both for computational and statistical reasons), because we can't get the optimal $\mathbf{w}$ for any feature extractor ϕ we choose.

Example: beyond linear functions

Regression: $x \in \mathbb{R}, y \in \mathbb{R}$

Linear functions:

$$\phi(x) = [1, x]$$

$$\mathcal{F}_1 = \{x \mapsto w_1 + w_2 x + w_3 x^2 : w_1, w_2 \in \mathbb{R}, w_3 = 0\}$$

Quadratic functions:

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

$$\mathcal{F}_2 = \{x \mapsto w_1 + w_2 x + w_3 x^2 : w_1, w_2, w_3 \in \mathbb{R}\}$$

[whiteboard]

- Given a fixed feature extractor ϕ, let us consider the space of all predictors $f_{\mathbf{w}}$ obtained by sweeping $\mathbf{w}$ over all possible values.
- If we use $\phi(x) = x$, then we get linear functions that go through the origin.
- If we use $\phi(x) = [1, x]$, then we get all linear functions. This trick of adding 1 to the feature vector lets us have $w \cdot \phi(0) = w_1$, instead of always being 0. The coordinate of w that corresponds to the feature 1 is often referred to as the "bias term," and is sometimes denoted b. But it's really just another component of the weight vector.
- We may also want to have functions that "bend" (or are not monotonic). For example, if we want to predict someone's health given his or her body temperature. There is sweet spot temperature (37 C) where the health is optimal; both higher and lower values should cause the health to decline.
- If we use $\phi(x) = [1, x, x^2]$, then we get quadratic functions, which are a strict superset of the linear functions, and therefore are strictly more expressive.

Example: even more flexible functions

Regression: $x \in \mathbb{R}, y \in \mathbb{R}$

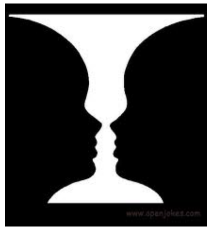
Piecewise constant functions:

$$\phi(x) = [\mathbf{1}[0 < x \le 1], \mathbf{1}[1 < x \le 2], \dots]$$

$$\mathcal{F}_3 = \{x \mapsto \sum_{j=1}^{10} w_j \mathbf{1}[j-1 < x \le j] : \mathbf{w} \in \mathbb{R}^{10}\}$$

[whiteboard]

- However, even quadratic functions can be limiting, because they have to rise and fall in a certain (parabolic) way. What if we wanted a more flexible, freestyle approach?
- We can create piecewise constant functions by defining features that "fire" (are 1) on particular regions of the input (e.g., $1 < x \leq 2$). Each feature gets associated with its own weight, which in this case corresponds to the desired function value in that region.
- Thus by varying the weight vector, we get piecewise constant functions with a particular discretization level. We can increase or decrease the discretization level as we need.
- Advanced: what happens if x were not a scalar, but a d-dimensional vector? We could perform discretization in $\mathbb{R}^d$, but the number of features grows exponentially in d, which leads to a phenomenon called the curse of dimensionality.



Linear in what?

Prediction driven by score:

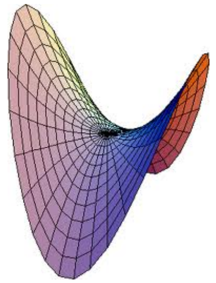
$$\mathbf{w} \cdot \phi(x)$$

Linear in $\mathbf{w}$?	Yes
Linear in $\phi(x)$?	Yes
Linear in x?	No! (x not necessarily even a vector)

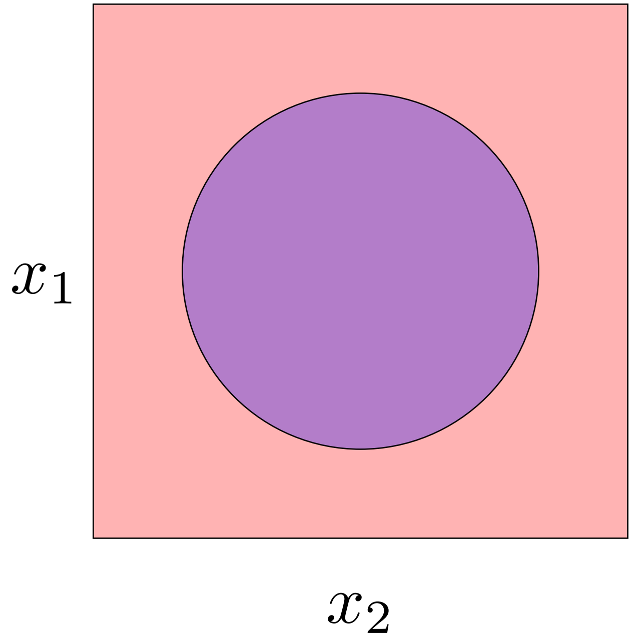
Key idea: non-linearity

- Predictors $f_{\mathbf{w}}(x)$ can be expressive **non-linear** functions and decision boundaries of x.
- Score $\mathbf{w} \cdot \phi(x)$ is **linear** function of $\mathbf{w}$, which permits efficient learning.

- Wait a minute...how were we able to get non-linear predictions using linear predictors?
- It is important to remember that for linear predictors, it is the score $\mathbf{w} \cdot \phi(x)$ that is linear in $\mathbf{w}$ and $\phi(x)$ (read it off directly from the formula). In particular, the score is not linear in x (it sometimes doesn't even make sense because x need not be a vector at all — it could be a string or a PDF file. Also, neither the predictor $f_{\mathbf{w}}$ (unless we're doing linear regression) nor the loss function TrainLoss$(\mathbf{w})$ are linear in anything.
- The significance is as follows: From the feature extraction viewpoint, we can define arbitrary features that yield very **non-linear** functions in x. From the learning viewpoint (only looking at $\phi(x)$, not x), **linearity** plays an important role in being able to optimize the weights efficiently (as it leads to convex optimization problems).



Geometric viewpoint



$$\phi(x) = [1, x_1, x_2, x_1^2, x_2^2]$$

With quadratic features, can construct decision boundaries in the shape of any conic section

[demo]

- Let's try to understand the relationship between the non-linearity in x and linearity in $\phi(x)$. We consider binary classification where our input is $x = [x_1, x_2] \in \mathbb{R}^2$ a point on the plane. With the quadratic features $\phi(x)$, we can carve out the decision boundary corresponding to an ellipse (think about the formula for an ellipse and break it down into monomials).
- We can now look at the feature vectors $\phi(x)$, which include an extra dimension. In this 3D space, a linear predictor (defined by the hyperplane) actually corresponds to the non-linear predictor in the original 2D space.

An example task

Example: detecting responses

Input x:

two consecutive messages in a chat

Output $y \in \{+1, -1\}$:

whether the second message is a response to the first

Recall: feature extractor ϕ should pick out properties of x that might be useful for prediction of y

Recall: feature extractor ϕ returns a set of (feature name, real number) pairs

- Let's apply what you've learned about feature extraction to a concrete problem. The motivation here is that messaging platforms often just show a single stream of messages, when there is generally a grouping of messages into coherent conversations. How can we build a classifier that can group messages automatically? We can formulate this as a binary classification problem where we look at two messages and determine whether these two are part of the same conversation or not.

Question

What feature templates would you use for predicting whether the second message is a response to the first?

time elapsed

time elapsed is between ___ and ___

first message contains ___

second message contains ___

two messages both contain ___

two messages have ___ common words

Summary so far

- Feature templates: organize related (sparse) features
- Hypothesis class: defined by features (what is possible)
- Linear classifiers: can produce non-linear decision boundaries

Roadmap

Features

Generalization

Training error

Loss minimization:

$$\min_{\mathbf{w}} \text{TrainLoss}(\mathbf{w})$$

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

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.

Rote learning

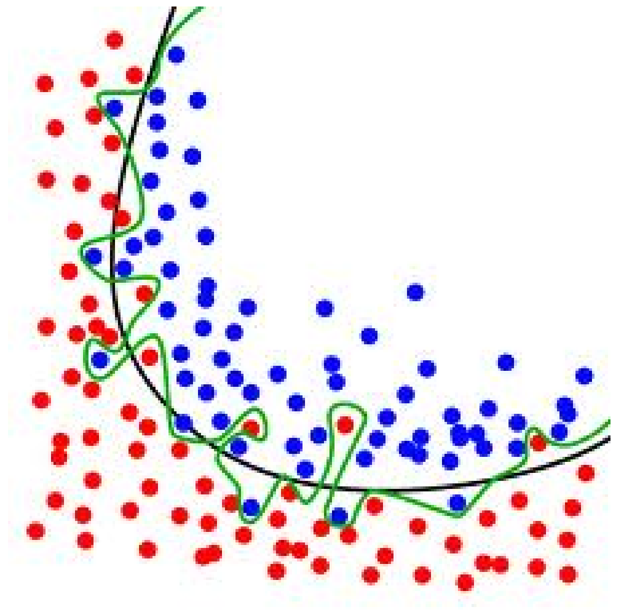
Create one feature for every example in $\mathcal{D}_{\text{train}}$

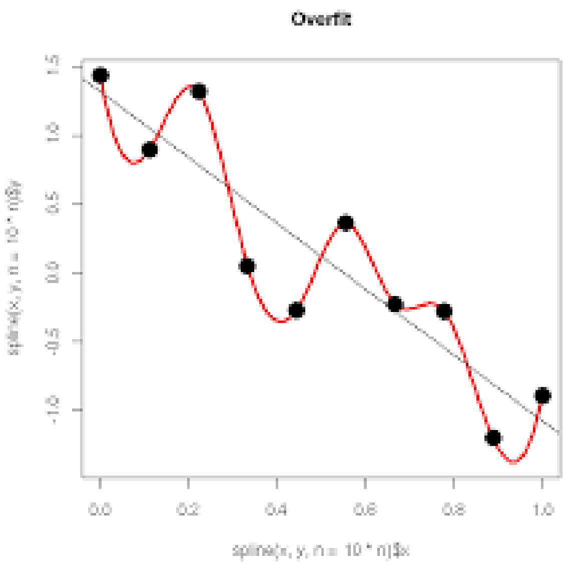
$$\textit{abc@gmail.com} \longrightarrow \boxed{\begin{array}{ll} \text{xEquals\_abc@gmail.com} & : 1 \\ \text{xEquals\_me@stanford.edu} & : 0 \\ \text{xEquals\_xyz@hotmail.com} & : 0 \\ \dots & \\ \text{xEquals\_zzz@zmail.com} & : 0 \end{array}}$$

- Easy to get zero training loss
- But clearly doesn't **generalize** to examples it hasn't seen...

- Clearly, machine learning can't be about just minimizing the training loss. If that were the case, we could easily "memorize" the training data by creating one indicator feature for every example in the data. Call our training dataset $\mathcal{D}_{\text{train}} = \{(x_1, y_1), \dots, (x_n, y_n)\}$. We can create a feature vector where the i-th feature $\phi(x)_i$ is 1 if the input x is equal to the i-th training input x_i, and 0 otherwise. Learning w is just the same as memorizing what the correct answer was for each training example: w_i would be positive if $y_i = 1$, and negative if $y_i = -1$.
- However, this is clearly not the way to build a good classifier. Using these features, the model will **overfit** to the training data instead of **generalizing** 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

$$\begin{tikzcd} \mathcal{D}_{\text{train}} \arrow[r] & \boxed{\text{Learner}} \arrow[r] & f \end{tikzcd}$$

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.

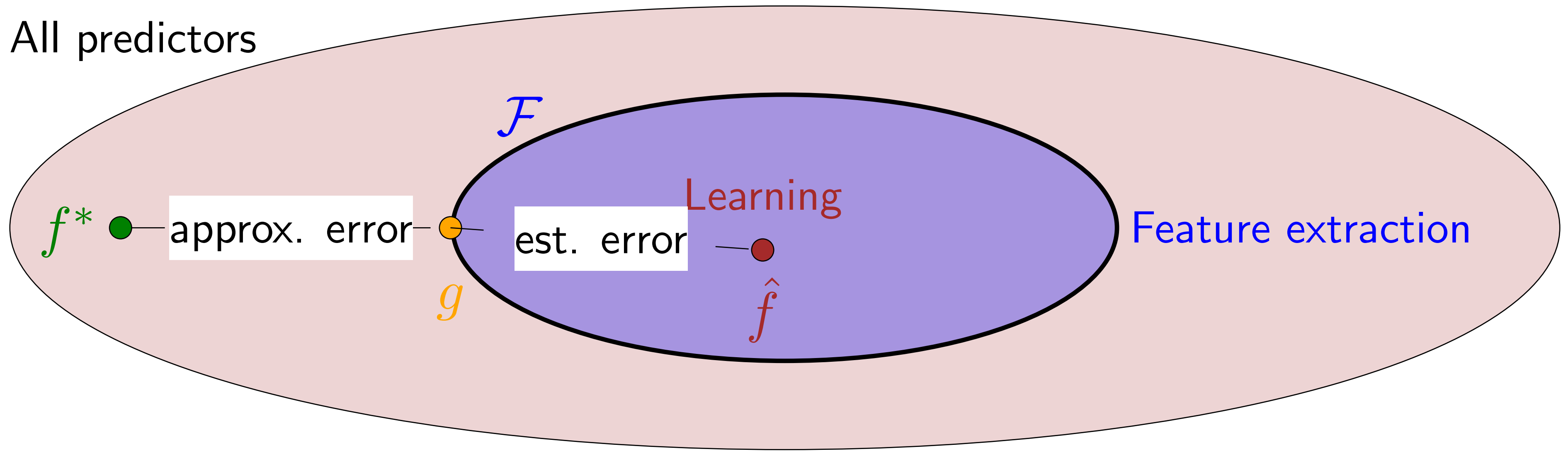
Generalization

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

$$\begin{tikzcd} \boxed{\mathcal{D}_{\text{train}}} \arrow[r] & \boxed{\mathcal{D}_{\text{test}}} \end{tikzcd}$$

- 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 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

more spurious hypotheses that are wrong but fit the data

- 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).

Learning as detective work

Goal: solve the case (find f^*)

Too few hypotheses: might not consider the real explanation

Too many hypotheses: many might fit the available clues

- What if the suspect has a secret twin brother...

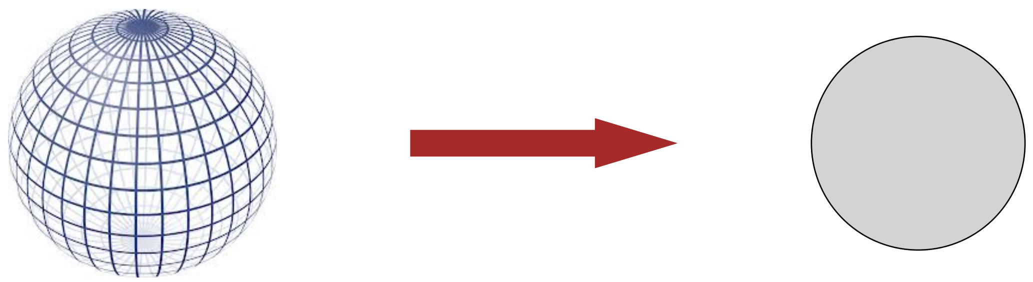
With more hypotheses, need more clues (data) to distinguish the correct hypothesis from incorrect ones

- Without formalizing it, we can understand the learning theory as being analogous to solving a mystery. The goal is to figure out who committed a crime, which is analogous to finding the optimal predictor f^*.
- If you only consider a couple suspects, it is easier to figure out which one is the most likely criminal. However, you might run the risk that none of the suspects you considered is the real criminal.
- If instead you have a much longer list of suspects, it takes a lot more work to figure out which one is most likely to be the criminal. Some of the suspects may seem to fit all of the available evidence (is the criminal Joe, or Joe's twin brother?), so you will have to collect more evidence (can we prove that Joe does not have a twin brother?).
- If you are able to collect enough clues (i.e., enough training data), it can be beneficial to have a longer list of suspects (larger hypothesis class). But when data is limited, you might get more mileage out of using a smaller hypothesis class, to avoid getting distracted by less plausible hypotheses.

Controlling size of hypothesis class

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

Keeping the dimensionality d small:



- 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.

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

- The most intuitive way to reduce overfitting is to reduce the number of features (or feature templates). 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.

Summary

- Feature engineering: many ways to improve model expressivity by changing $\phi(x)$

- Approximation error: more expressive models can better approximate the ideal function

- Estimation error: more expressive models can be harder to learn, require more data

- Let us conclude now. We discussed some general principles for designing good features for linear predictors. Just with the machinery of linear prediction, we were able to obtain rich predictors which were quite rich.
- At the same time, we learned about the trade-off between approximation error (how expressive is my hypothesis class) estimation error (how well can I find the best hypothesis in my hypothesis class). This trade-off implies that some choices of feature extractors are not good because they can express too many functions, and it will be difficult to distinguish the best predictor from other predictors that happen to fit the training data, but generalize worse.