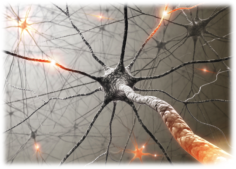
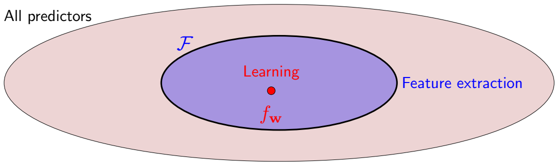
Machine learning: feature templates



- In this module, we'll talk about how to use feature templates to construct features in a flexible way.

Feature extraction + learning

$$\mathcal{F} = \{f_{\mathbf{w}}(x) = \text{sign}(\mathbf{w} \cdot \phi(x)) : \mathbf{w} \in \mathbb{R}^d\}$$



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

Want $\mathcal{F}$ to contain good predictors but not be too big

- Recall that the hypothesis class $\mathcal{F}$ is the set of predictors considered by the learning algorithm. In the case of linear predictors, $\mathcal{F}$ is given by some function of $\mathbf{w} \cdot \phi(x)$ for all $\mathbf{w}$ (sign for classification, no sign for regression). This can be visualized as a set in the figure.
- Learning is the process of choosing a particular predictor $f_{\mathbf{w}}$ from $\mathcal{F}$ given training data.
- But the question that will concern us in this module is how do we choose $\mathcal{F}$? We saw some options already: linear predictors, quadratic predictors, etc., but what makes sense for a given application?
- 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** good predictors. It's okay and expected that $\mathcal{F}$ will contain bad ones as well. Of course, we don't want $\mathcal{F}$ to be too big, or else learning becomes hard, not just computationally but statistically (as we'll explain when we talk about generalization).

Feature extraction with feature names

Example task:

classifier

$$\text{string } (x) \longrightarrow \boxed{f_{\mathbf{w}}(x) = \text{sign}(\mathbf{w} \cdot \phi(x))} \longrightarrow \text{valid email address? } (y)$$

Question: what properties of x **might be** relevant for predicting y?

Feature extractor: Given x, produce set of (feature name, feature value) pairs

abc@gmail.com

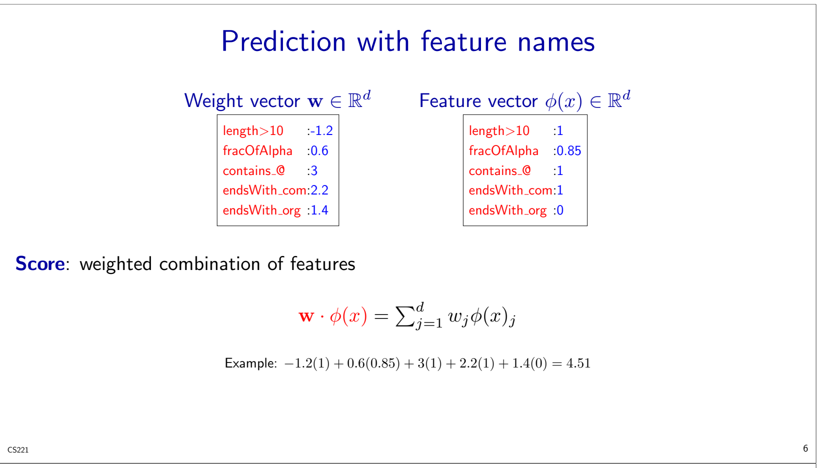
x

$$\xrightarrow[\text{arbitrary!}]{\textbf{feature extractor } \phi}$$

length>10	: 1
fracOfAlpha	: 0.85
contains_@	: 1
endsWith_.com	: 1
endsWith_.org	: 0

$\phi(x)$

- To get some intuition about feature extraction, let us consider the task of predicting whether whether a string is a valid email address or not.
- We will assume the classifier $f_{\mathbf{w}}$ is a linear classifier, which is given by some feature extractor ϕ.
- Feature extraction is a bit of an art that requires intuition about both the task and also what machine learning algorithms are capable of. The general principle is that features should represent properties of x which **might be** relevant for predicting y.
- Think about the feature extractor as producing a set of (feature name, feature value) pairs. For example, we might extract information about the length, or fraction of alphanumeric characters, whether it contains various substrings, etc.
- It is okay to add features which turn out to be irrelevant, since the learning algorithm can always in principle choose to ignore the feature, though it might take more data to do so.
- We have been associating each feature with a name so that it's easier for us (humans) to interpret and develop the feature extractor. The feature names act like the analogue of **comments** in code. Mathematically, the feature name is not needed by the learning algorithm and erasing them does not change prediction or learning.



Prediction with feature names

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

length>10	:-1.2
fracOfAlpha	:0.6
contains_@	:3
endsWith_com	:2.2
endsWith_org	:1.4

Feature vector $\phi(x) \in \mathbb{R}^d$

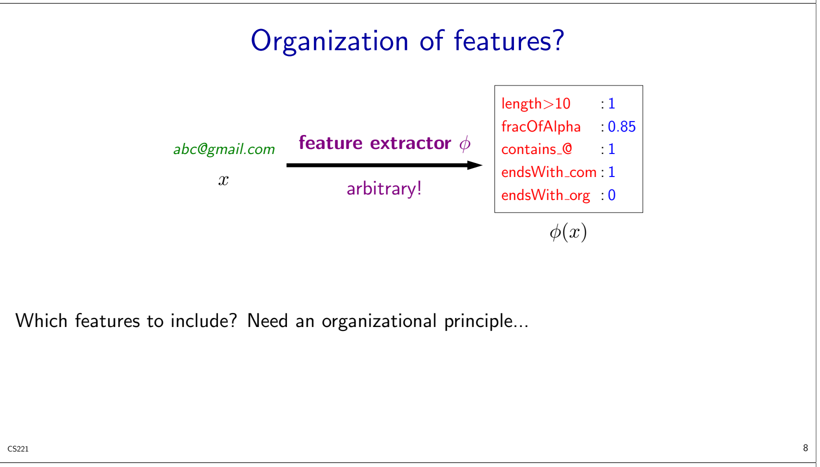
length>10	:1
fracOfAlpha	:0.85
contains_@	:1
endsWith_com	:1
endsWith_org	:0

Score: weighted combination of features

$$\mathbf{w} \cdot \phi(x) = \sum_{j=1}^{d} w_j \phi(x)_j$$

Example: $-1.2(1) + 0.6(0.85) + 3(1) + 2.2(1) + 1.4(0) = 4.51$

- A feature vector formally is just a list of numbers, but we have endowed each feature in the feature vector with a name.
- The weight vector is also just a list of numbers, but we can endow each weight with the corresponding name as well.
- Recall that the score is simply the dot product between the weight vector and the feature vector. In other words, the score aggregates the contribution of each feature, weighted appropriately.
- Each feature weight w_j determines how the corresponding feature value $\phi_j(x)$ contributes to the prediction.
- If w_j is positive, then the presence of feature j ($\phi_j(x) = 1$) favors a positive classification (e.g., ending with com). Conversely, if w_j is negative, then the presence of feature j favors a negative classification (e.g., length greater than 10). The magnitude of w_j measures the strength or importance of this contribution.
- Advanced: while tempting, it can be a bit misleading to interpret feature weights in isolation, because the learning algorithm treats $\mathbf{w}$ holistically. In particular, a feature weight w_j produced by a learning algorithm will change depending on the presence of other features. If the weight of a feature is positive, it doesn't necessarily mean that feature is positively correlated with the label.



Organization of features?

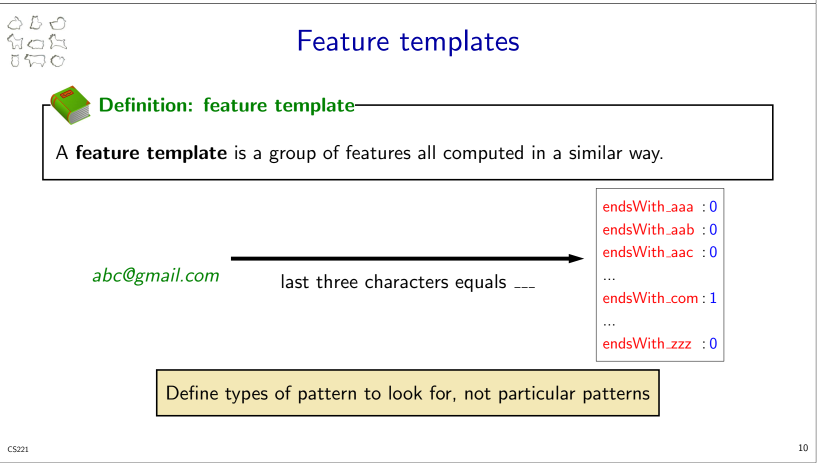
$$\underset{x}{\textit{abc@gmail.com}} \xrightarrow[\text{arbitrary!}]{\text{feature extractor } \phi}$$

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

$\phi(x)$

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

- How would we go about 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, and it's easy to miss useful features (e.g., endsWith_us), and there might be other features which are predictive but not intuitive.
- We need a more systematic way to go about this.



Feature templates

Definition: feature template

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

$$\textit{abc@gmail.com} \xrightarrow{\qquad} \text{last three characters equals \_\_\_}$$

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

Define types of pattern to look for, not particular patterns

- 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".)
- Rather than defining individual features like endsWith_com, we can define a single feature template which expands into all the features that computes whether the input x matches any three characters.
- Typically, we will write a feature template as an English description with a blank (___), which is to be filled in with an arbitrary value.
- The upshot is that we don't need to know which particular patterns (e.g., three-character suffixes) are useful, but only that **existence** of certain patterns (e.g., three-character suffixes) are useful cue to look at.
- It is then up to the learning algorithm to figure out which patterns are useful by assigning the appropriate feature weights.

Feature templates example 1

Input:

abc@gmail.com

Feature template	Example feature	
Last three characters equals ___	Last three characters equals *com*	: 1
Length greater than ___	Length greater than *10*	: 1
Fraction of alphanumeric characters	Fraction of alphanumeric characters	: 0.85

- Here are some other examples of feature templates.
- Note that an isolated feature (e.g., fraction of alphanumeric characters) can be treated as a trivial feature template with no blanks to be filled.
- In many cases, the feature value is binary (0 or 1), but they can also be real numbers.

Feature templates example 2

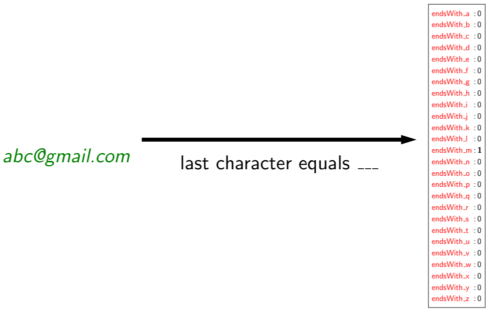
Input:

Latitude: 37.4068176
Longitude: -122.1715122

Feature template	Example feature name	
Pixel intensity of image at row ___ and column ___ (___ channel)	Pixel intensity of image at row *10* and column *93* (*red* channel)	: 0.8
Latitude is in [___, ___] and longitude is in [___, ___]	Latitude is in [*37.4*, *37.5*] and longitude is in [*-122.2*, *-122.1*]	: 1

- As another example application, suppose the input is an aerial image along with the latitude/longitude corresponding where the image was taken. This type of input arises in poverty mapping and land cover classification.
- In this case, we might define one feature template corresponding to the pixel intensities at various pixel-wise row/column positions in the image across all the 3 color channels (e.g., red, green, blue).
- Another feature template might define a family of binary features, one for each region of the world, where each region is defined by a bounding box over latitude and longitude.

Sparsity in feature vectors



abc@gmail.com

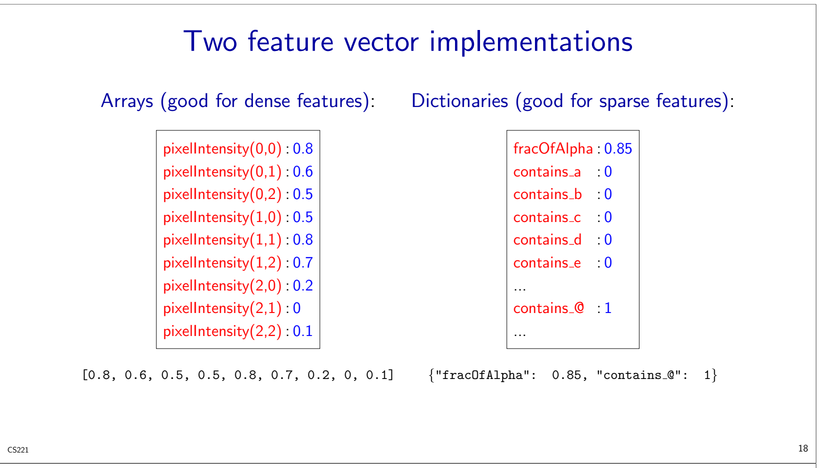
last character equals ___

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

Compact representation:

```
{"endsWith_m": 1}
```

- In general, a feature template corresponds to many features, and sometimes, **for a given input**, most of the feature values are zero; that is, the feature vector is **sparse**.
- Of course, different feature vectors have different non-zero features.
- In this case, it would be inefficient to represent all the features explicitly. Instead, we can just store the values of the non-zero features, assuming all other feature values are zero by default.



Two feature vector implementations

Arrays (good for dense features):

pixelIntensity(0,0)	: 0.8
pixelIntensity(0,1)	: 0.6
pixelIntensity(0,2)	: 0.5
pixelIntensity(1,0)	: 0.5
pixelIntensity(1,1)	: 0.8
pixelIntensity(1,2)	: 0.7
pixelIntensity(2,0)	: 0.2
pixelIntensity(2,1)	: 0
pixelIntensity(2,2)	: 0.1

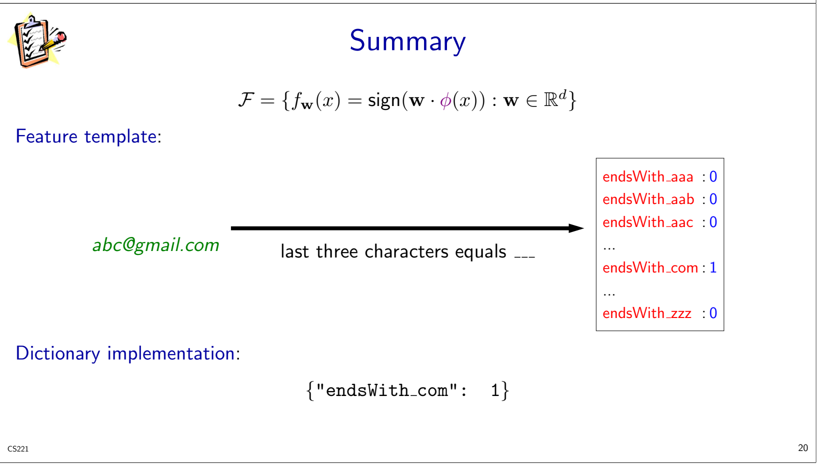
`[0.8, 0.6, 0.5, 0.5, 0.8, 0.7, 0.2, 0, 0.1]`

Dictionaries (good for sparse features):

fracOfAlpha	: 0.85
contains_a	: 0
contains_b	: 0
contains_c	: 0
contains_d	: 0
contains_e	: 0
...	
contains_@	: 1
...	

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

- In general, there are two common ways to implement feature vectors: using arrays and using dictionaries.
- **Arrays** assume a fixed ordering of the features and store the feature values as an array. This implementation 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 arrays are more common.
- However, when we have sparsity (few nonzeros), it is typically more efficient to implement the feature vector as a **dictionary** (map) from strings to doubles rather than a fixed-size array of doubles. The features not in the dictionary implicitly have a default value of zero. This sparse implementation is useful for natural language processing with linear predictors, and is what allows us to work efficiently over millions of features. In Python, one would define a feature vector $\phi(x)$ as the dictionary `{"endsWith_"+x[-3:]: 1}`. Dictionaries do incur extra overhead compared to arrays, and therefore dictionaries are much slower when the features are not sparse.
- One advantage of the sparse feature implementation is that you don't have to instantiate all the set of possible features in advance; the weight vector can be initialized to $\{\}$, and only when a feature weight becomes non-zero do we store it. This means we can dynamically update a model with incrementally arriving data, which might instantiate new features.



Summary

$$\mathcal{F} = \{f_{\mathbf{w}}(x) = \text{sign}(\mathbf{w} \cdot \phi(x)) : \mathbf{w} \in \mathbb{R}^d\}$$

Feature template:

abc@gmail.com → last three characters equals ___ →

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

Dictionary implementation:

`{"endsWith_com": 1}`

- The question we are concerned with in this module is to how to define the hypothesis class $\mathcal{F}$, which in the case of linear predictors is the question of what the feature extractor ϕ is.
- We showed how **feature templates** can be useful for organizing the definition of many features, and that we can use dictionaries to represent **sparse** feature vectors efficiently.
- Stepping back, feature engineering is one of the most critical components in the practice of machine learning. It often does not get as much attention as it deserves, mostly because it is a bit of an art and somewhat domain-specific.
- More powerful predictors such as neural networks will alleviate some of the burden of feature engineering, but even neural networks use feature vectors as the initial starting point, and therefore its effectiveness is ultimately governed by how good the features are.