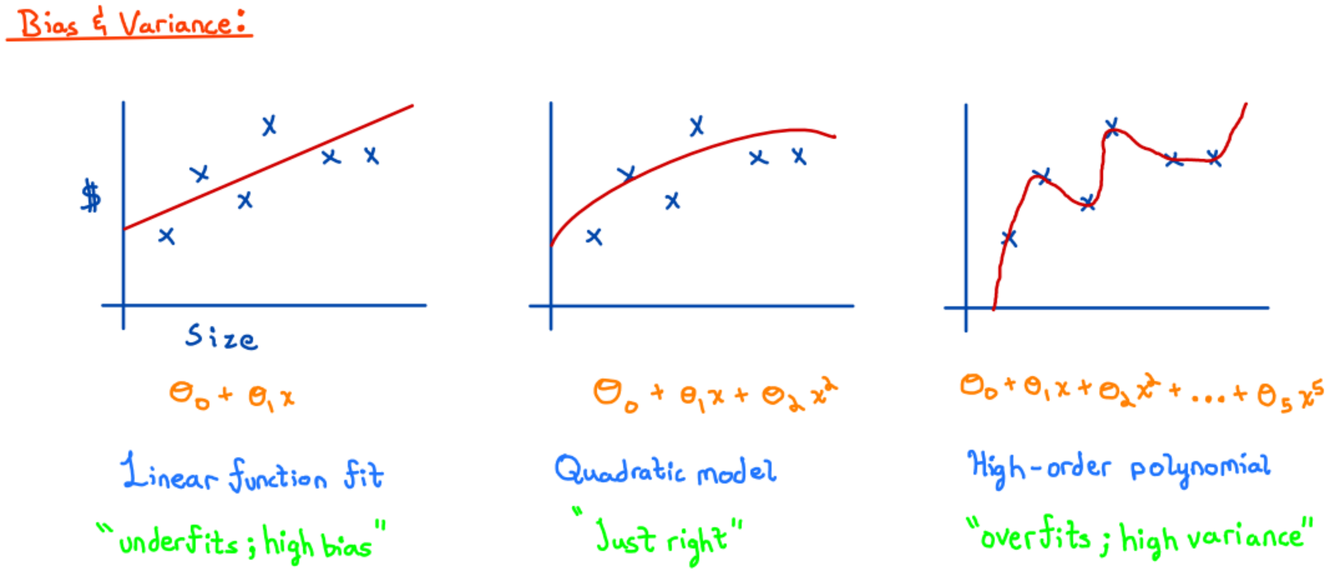
Lecture 8 [Data Splits, Models, & Cross-Validation]

Bias / Variance:

Bias and **Variance** is one of those concepts that's sort of easy to understand but hard to master. People that understand this concept are much more efficient in terms of how you develop learning algorithms and make your algorithms work

#*#Mathematical aspects of Learning Theories:(Error Decomposition, Uniform Convergence, VC Dimension)#*

Let's say you have this data set and you have a Housing price prediction problem:



This high-order polynomial is a 5th order polynomial

To name these phenomenon, assuming the one in the middle is what we like, fitting a ***Quadratic function*** is maybe pretty good, so we'll call it “Just right”

Whereas the example on the left, it ***underfits*** the data, as in it's not capturing the trend that is maybe semi-evident in the data, and we say that this algorithm has ***high bias***

The term ***bias*** has multiple meanings in the English language. We, as a society, want to avoid racial bias, gender bias, discrimination against people's orientation, and things like that

The term ***bias*** in Machine Learning has a completely separate meaning and it just means that this learning algorithm had very strong preconceptions that the data could be fit by ***Linear functions***. This algorithm had a very strong bias or a very strong preconception that the relationship between pricing and house size is linear, and this bias turns out not to be true

In contrast with the example on the right, we say that this is ***overfitting*** the data and this algorithm has high variance. The term ***high variance*** comes from this intuition that you happen to get these six examples, but if a friend of yours was to collect a slightly different set of six examples and rerun the algorithm on the new data set, then this algorithm will fit some totally other varying function on it, and so your predictions will have very high variance

So if a friend of yours does the same experiment and they just get a slightly different data set, just due to random noise, then this algorithm fitting a fifth-order polynomial, results in a totally different result. So that's why we say that this algorithm has a very ***high variance***. There's a lot of variability in the predictions this algorithm will make

When you train the learning algorithm, it almost never works the first time

So when you're developing learning algorithms, your standard workflow could be to often train an algorithm (often to train up something quick and dirty), and then try to understand if the algorithm has a problem of ***high bias*** or ***high variance*** (if it's ***underfitting*** or ***overfitting*** the data), and then use that insight to decide how to improve the learning algorithm

The problems of ***bias*** and ***variance*** also hold true for ***classification problems***

In the era of GPU computing ability to train models with a lot of features. Take a ***SVM***, if you add enough features to it (if you have a high enough dimensional feature space) or if you take a ***Linear Regression*** or ***Logistic Regression*** model and you just add enough features to it, you can often ***overfit*** the data. And it turns out that one of the most effective ways to prevent ***overfitting*** is **Regularization**

Regularization will be one of those techniques that won't take that long to explain, but don't underestimate how widely used it is. It's not used in every single machine learning model but it's used very, very often

Regularization:

The idea is to take the ***Optimization objective for Linear Regression*** and just add one extra term (sometimes you write **λ/2** to make some of the derivations come out easier

What this does is it takes your cost function for logistic regression, which you try to minimize the squared error fit to the data, and you're creating an incentive term for the algorithm to make the parameter's **θ** smaller. This is called the **regularization term**

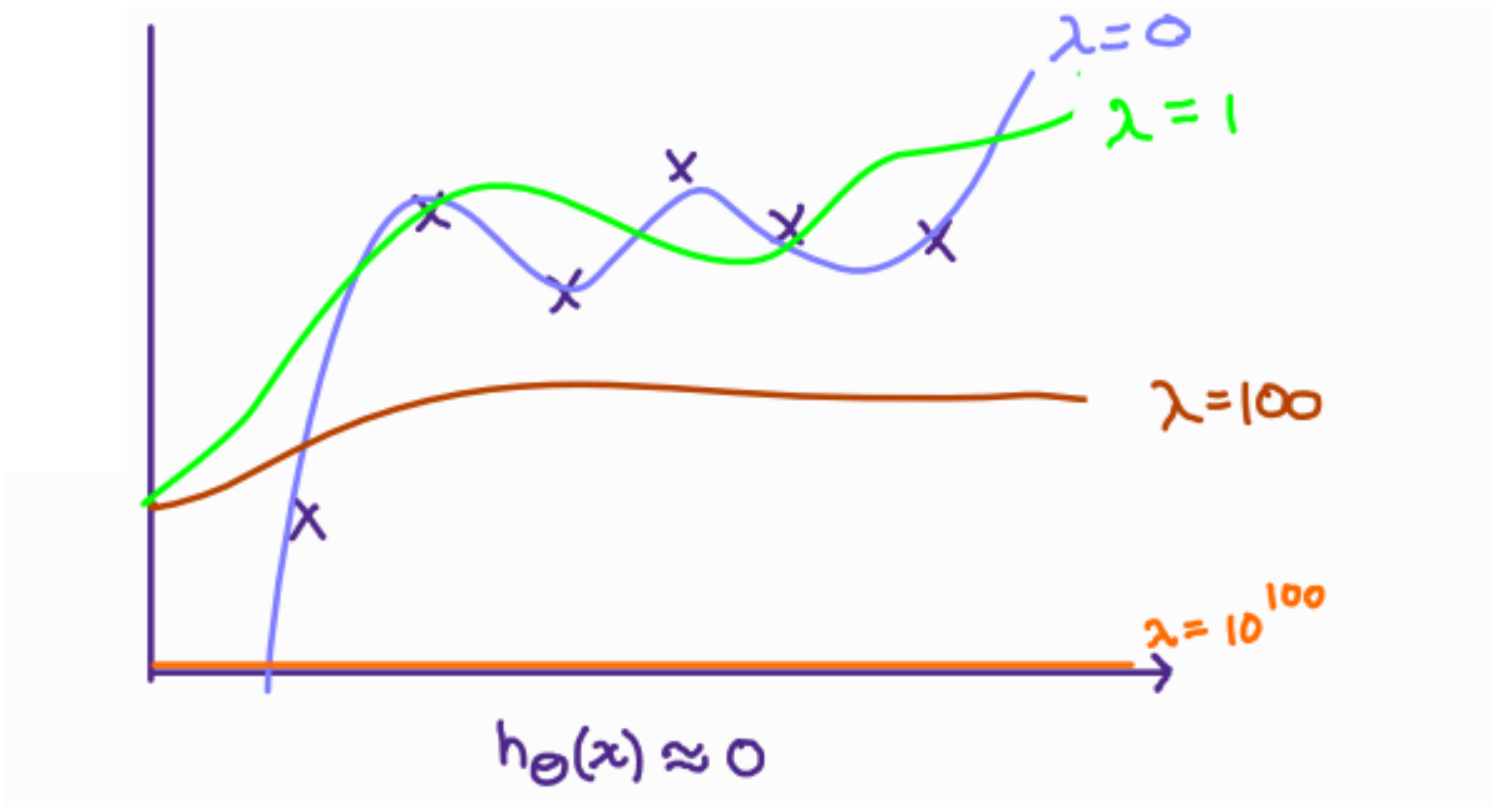
Regularization:

$$\min_{\theta} \frac{1}{2} \sum_{i=1}^{m} \| y^{(i)} - \theta^T x^{(i)} \|^2 + \overbrace{\lambda \|\theta\|^2}^{\text{Regularization}}$$

Regularization:

$$\min_{\theta} \frac{1}{2} \sum_{i=1}^{m} \| y^{(i)} - \theta^T x^{(i)} \|^2 + \overbrace{\underset{\overline{2}}{\lambda} \|\theta\|^2}^{\text{Regularization}}$$

Using the ***linear regression*** overfitting example:



If you set **ƛ=0**, then it's just ***linear regression*** over the fifth order polynomial features

As you increase **ƛ**, when you solve for the minimization problem, the ***regularization term*** penalizes the parameters being too big and you'll end up with a fit that maybe look like this **(green)**

By preventing the parameter's **ϴ** from being too big, you're making it harder for the learning algorithm to overfit the data. It turns out that fitting a very high-order polynomial like that **(periwinkle)** may result in values of **ϴ** that are very large, and if you set **ƛ** to be too large, you end up in an underfitting regime

There'll usually be some optimal value of **ƛ** where, if **ƛ=0** you're not using any ***regularization*** (so you're maybe overfitting), if **ƛ** is way too big then you're forcing all the parameters to be too close to **0**

More generally, if you have a logistic regression problem, for example, where this is your cost function:

$$\arg\max_{\theta} \sum_{i=1}^{n} \log p(y^{(i)} \mid x^{(i)}; \theta)$$

Then, to add regularization:

$$\arg\max_{\theta} \sum_{i=1}^{n} \log p(y^{(i)} \mid x^{(i)}; \theta) - \lambda\|\theta\|^2$$

One of the reasons the ***SVM*** doesn't overfit too badly even though it could be working in an infinite-dimensional feature space by using ***kernels***, is that it turns out that the optimization objective of the ***SVM*** was to minimize **||w||**$^{\mathbf{2}}$. This corresponds to maximizing the margin (the ***geometric margin SVM***)

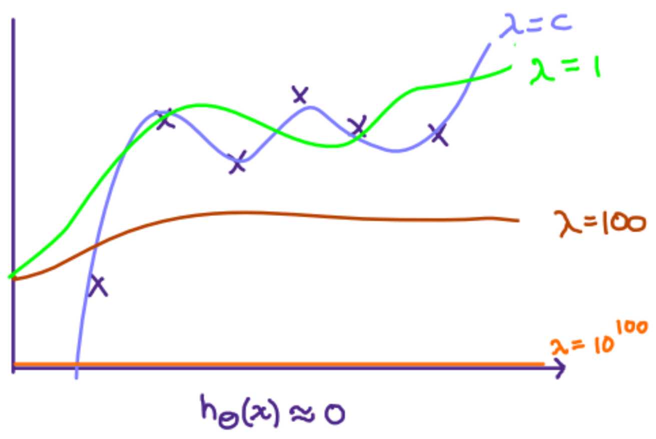
It's possible to prove that **||w||**$^{\mathbf{2}}$ has a similar effect as **ƛ||ϴ||**$^{\mathbf{2}}$, which is why the ***SVM***, despite working in infinite-dimensional feature space sometimes, by forcing the parameters to be small, it's difficult for the ***SVM*** to overfit the data too much

$$\arg\max_{\theta} \sum_{i=1}^{n} \log p(y^{(i)} \mid x^{(i)}; \theta) - \lambda\|\theta\|^2 \swarrow \min \|w\|^2$$

Regularization:

$$\min_{\theta} \frac{1}{2} \sum_{i=1}^{m} \| y^{(i)} - \theta^T x^{(i)} \|^2 + \overbrace{\underline{\lambda} \|\theta\|^2}^{\text{Regularization}}$$

$$\arg\max_{\theta} \sum_{i=1}^{n} \log p(y^{(i)} | x^{(i)}; \theta) - \lambda \|\theta\|^2 \swarrow \min \|w\|^2$$



Remembering ***Naive Bayes*** as a ***text classification algorithm***, let's say you have **100** examples but you have **10,000** dimensional features and you construct your feature like this:

Text Classification:

$m = 100$

$n = 10{,}000$

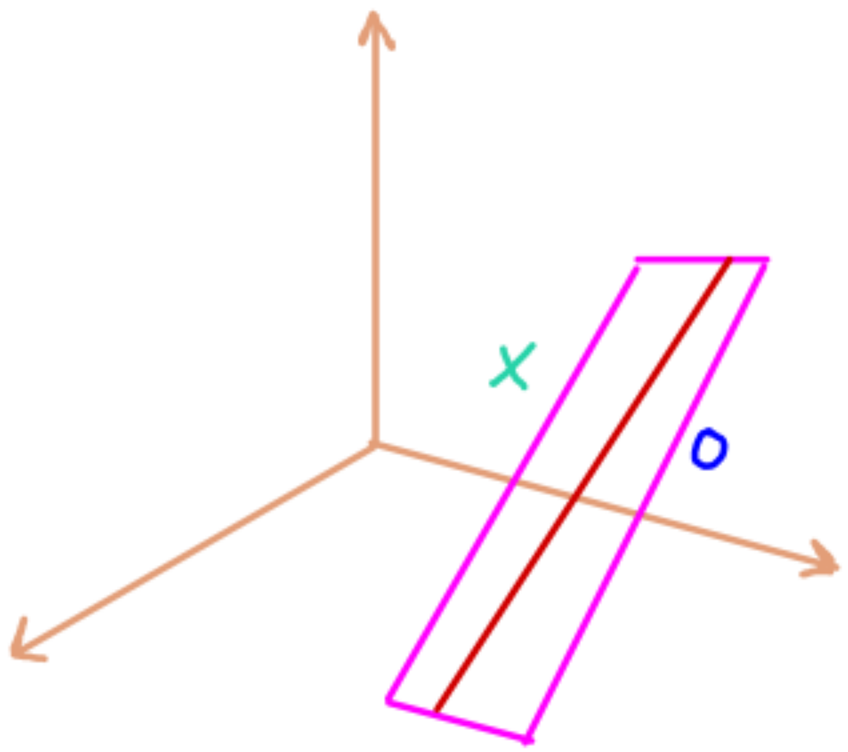
$$x = \left[\begin{array}{c}1\\0\\1\\\vdots\\1\\0\\\vdots\end{array}\right]\begin{array}{l}\text{a}\\\text{aardvark}\\\vdots\\ \\ \\ \\ \end{array}$$

It turns out that if you fit ***logistic regression*** to this type of data where you have **10,000** parameters and **100** examples, this will probably overfit the data. But if you use ***logistic regression*** with ***regularization***, this is actually a pretty good algorithm for text classification

Because this is ***logistic regression***, you need to implement ***gradient descent*** or something to solve local value parameters

In terms of performance accuracy, this will usually (***logistic regression with regularization for text classification***) outperform ***Naive Bayes*** on a classification accuracy standpoint. Without ***regularization***, ***logistic regression*** will badly overfit this data

Imagine you have a 3-dimensional subspace where you have two examples, then all you can do is fit a straight line for the hyperplane to separate these two examples



One rule of thumb for ***logistic regression*** is that if you do not use ***regularization***, it's nice if the number of examples is at least on the order of the number of parameters you want to fit

# Why don't we ***regularize*** per parameter?

Instead of:

$\lambda \|\theta\|^2$

It would be:

$$\sum_j \lambda_j \theta_j^2$$

The reason we don't do this is because you then end up with **10,000** parameters here:

$$\lambda \|\theta\|^2$$

$$\sum_j \underset{\substack{\uparrow \\ 10{,}000}}{\lambda_j} \underset{\substack{\uparrow \\ 10{,}000}}{\theta_j^2}$$

and choosing all there **10,000 λ**'s is as difficult as just choosing all these parameters in the first place. So we don't have a good way to do this

In order to make sure that the different **λ**'s are on the similar scale, a common pre-processing step we use in learning algorithms is to take your different features where size of the house is a range $\mathbf{x_1}$ and the number of bedrooms is $\mathbf{x_2}$, then these features are on very different scales and normalizing them to all be on a similar scale (subtract out the mean and divide it by the standard deviation)

So scale all of these things to be between:

x_1 - 500 - 10,100

x_2 - 1 - 5

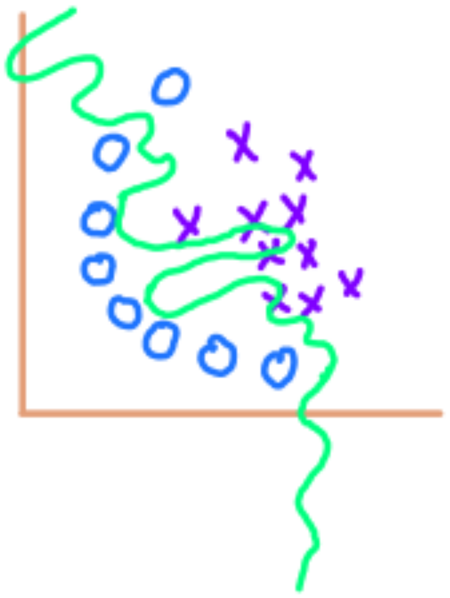
$$\begin{cases} 0 - 1 \\ -1 \ldots 1 \end{cases}$$

This would be a good pre-processing step before applying these methods

It turns out that this will make ***gradient descent*** run faster as well. It's a common pre-processing step to scale each individual feature to be on a similar range of values

In general, models that have high bias tend to underfit and models that have high variance tend to overfit

One way to think of ***high bias*** and ***high variance*** is if your dataset looks like this and if somehow your classifier has very high complexity, there's a very very complicated function, but for some reason it's still not fitting your data well:



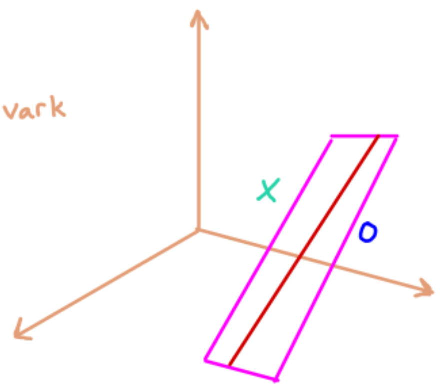
So that would be one way to have ***high bias*** and ***high variance***

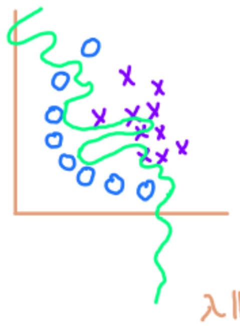
Text Classification:

$m = 100$

$n = 10{,}000$

$$x = \begin{array}{c} \left[\begin{array}{c} 1 \\ 0 \\ 1 \\ \vdots \\ 1 \\ 0 \\ \vdots \end{array}\right] \end{array} \begin{array}{l} a \\ \text{aardvark} \\ \vdots \\ \\ \\ \\ \\ \end{array}$$





$\lambda \|\theta\|^2$

$$\sum_j \underset{\overset{\uparrow}{10{,}000}}{\lambda_j} \; \underset{\overset{\uparrow}{10{,}000}}{\theta_j^2}$$

x_1 - 500 - 10,100

x_2 - 1 - 5

$$\left\{ \begin{array}{l} 0 - 1 \\ -1 \ldots 1 \end{array} \right.$$

Mechanically, the way you implement ***regularization*** is by adding that penalty on the norm of the parameters. It turns out that there's another way to think about ***regularization***

Remember: When we talked about ***linear regression***, we talked about minimizing squared error and then later on we saw that ***linear regression*** was ***maximum likelihood estimation*** on a certain ***generalized linear model*** using a ***Gaussian distribution*** as the choice for the ***exponential family*** (as a member of the ***exponential family***)

There's a similar point of view you can take on the regularization algorithm that we just saw

Let's say **S** is the ***training set***:

$$S = \{(x^{(i)}, y^{(i)})\}_{i=1}^{m}$$

Given a ***training set***, you want to find the most likely value of **Θ**:

$$S = \{(x^{(i)}, y^{(i)})\}_{i=1}^{m}$$

$$P(\theta|S)$$

By Bayes' rule:

$$S = \{(x^{(i)}, y^{(i)})\}_{i=1}^{m}$$

$$P(\theta|S) = \frac{P(S|\theta)\,P(\theta)}{P(S)}$$

So, if you want to pick the value of **ϴ** that's the most likely value of **ϴ** given the data you saw, then because the denominator is just a constant:

$$S = \{(x^{(i)}, y^{(i)})\}_{i=1}^{m}$$

$$P(\theta|S) = \frac{P(S|\theta)\,P(\theta)}{P(S)}$$

$$\arg\max_{\theta} P(\theta|S) = \arg\max_{\theta} P(S|\theta)P(\theta)$$

If you're using ***logistic regression***, then:

$$S = \{(x^{(i)}, y^{(i)})\}_{i=1}^{m}$$

$$P(\theta|S) = \frac{P(S|\theta)\,P(\theta)}{P(S)}$$

$$= \arg\max_{\theta} \Big(\underbrace{\prod_{i=1}^{m} P(y^{(i)}|x^{(i)};\theta)}_{\text{Logistic Regression}}\Big) P(\theta)$$

$$\arg\max_{\theta} P(\theta|S) = \arg\max_{\theta} P(S|\theta)P(\theta)$$

It turns out that if you assume **P(Θ)** is Gaussian, then:

$$P(\theta):\ \theta \sim \mathcal{N}(0, \tau^2 I)$$

$$P(\theta) = \frac{1}{\sqrt{2\pi}\,|\tau^2 I|^{1/2}} \exp\left(-\frac{1}{2}\theta^T(\tau^2 I)^{-1}\theta\right)$$

$$= \arg\max_{\theta} \Big(\underbrace{\prod_{i=1}^{m} P(y^{(i)}|x^{(i)};\theta)}_{\text{Logistic Regression}}\Big) P(\theta)$$

$$\underset{\text{"prior probability on }\theta\text{"}}{P(\theta)}:\ \theta \overset{\text{"is Gaussian"}}{\sim} \mathcal{N}(\underbrace{0}_{\text{"mean"}}, \overset{\text{"variance"}}{\overbrace{\overset{\text{"Tau"}}{\tau}^2 I}})$$

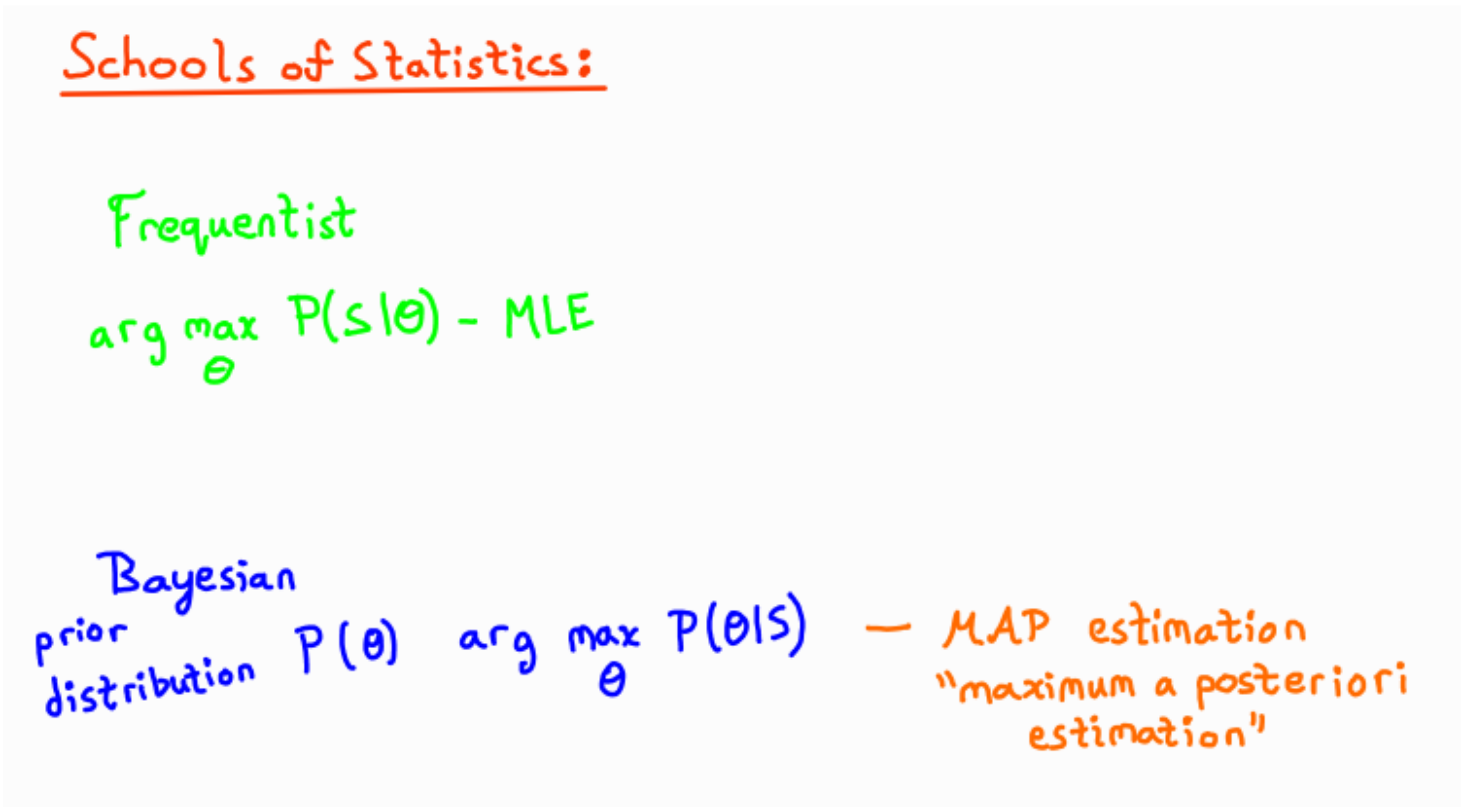
$$P(\theta) = \frac{1}{\sqrt{2\pi}\,|\tau^2 I|^{1/2}} \exp\left(-\frac{1}{2}\theta^T(\tau^2 I)^{-1}\theta\right)$$

If this is your ***prior distribution*** for **Θ** and you plug this in here (**orange**) and you take logs, compute the max, and so on, then you end up with exactly the ***regularization*** technique that we found just now

In everything we've been doing so far, we've been taking a ***frequentist interpretation***. The two main schools of statistics are the ***Frequentist school of statistics*** and the ***Bayesian school of statistics***

In the ***frequentist school of statistics***, we say that there is some data and we want to find the value of **Θ** that makes the data as likely as possible, and that's where we got ***maximum likelihood estimation***. And in the ***frequentist school of statistics***, we view there as being some true value of **Θ** out in the world that is unknown. So there is some true value of **Θ** that generated all these housing prices, and our goal is to estimate this true parameter

In the ***bayesian school of statistics***, we say that **Θ** is unknown, but before you see even any data, you already have some prior beliefs about how housing prices are generated out in the world and your prior beliefs are captured in a ***prior distribution***, denoted by **P(Θ)** (called the ***Gaussian prior***). And in the ***bayesian*** view of the world, our goal is to find the value of **Θ** that is most likely after we have seen the data. This is called ***MAP estimation***



Schools of Statistics:

Frequentist

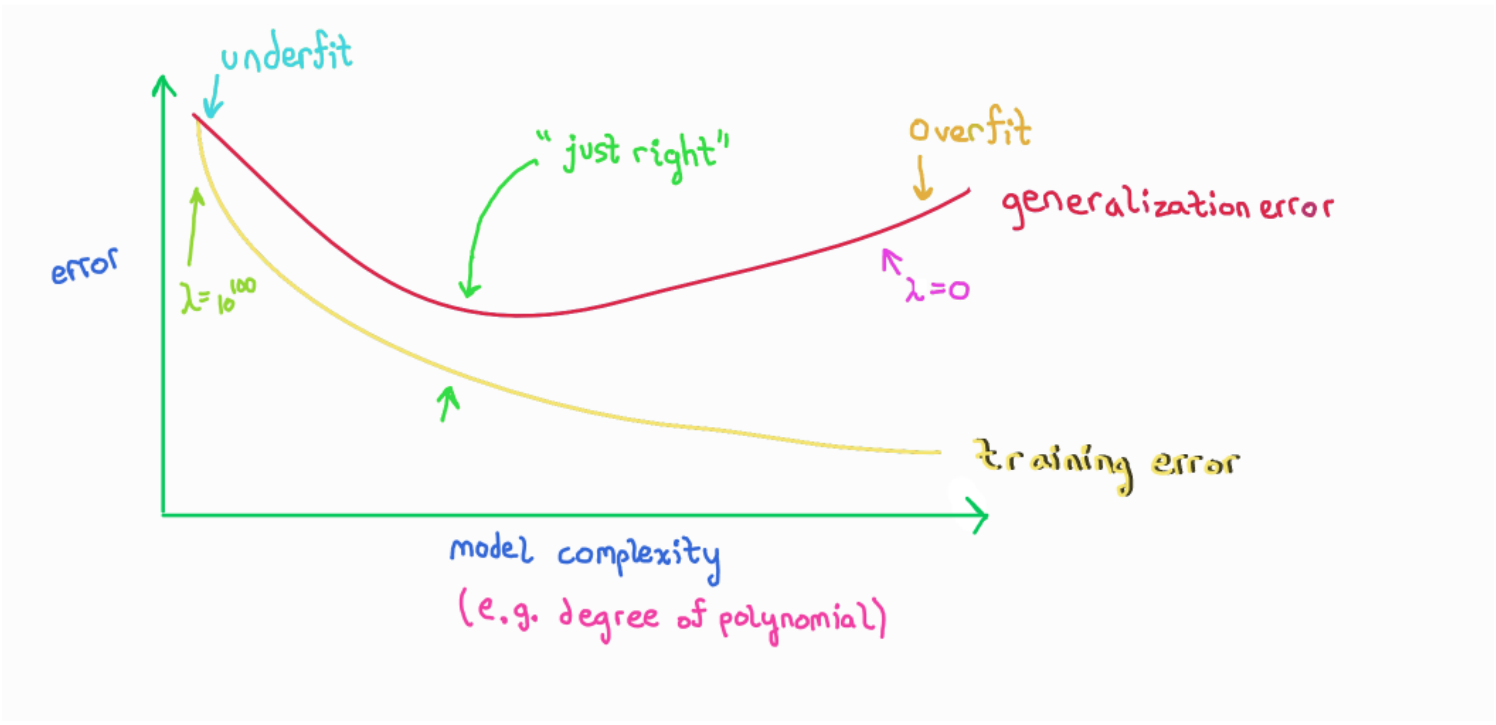
$\arg\max_{\theta} P(S|\theta)$ - MLE

Bayesian

prior distribution $P(\theta)$ $\arg\max_{\theta} P(\theta|S)$ — MAP estimation "maximum a posteriori estimation"

As you increase model complexity, If you do not regularize and you fit a linear function, quadratic function, cubic function, etc, you find that the higher the degree of your polynomial, the better your training error because a fifth-order polynomial always fits the data better than a fourth-order polynomial if you do not regularize

What we saw with the original picture was that the ability of the algorithm to generalize kind of goes down and then starts to go back up. So if you were to have a separate test set and evaluate your classifier on a set of data that the algorithm hasn't seen so far, so measure how well the algorithm generalizes to a different novel set of data, then if you fit a linear function(**red**):

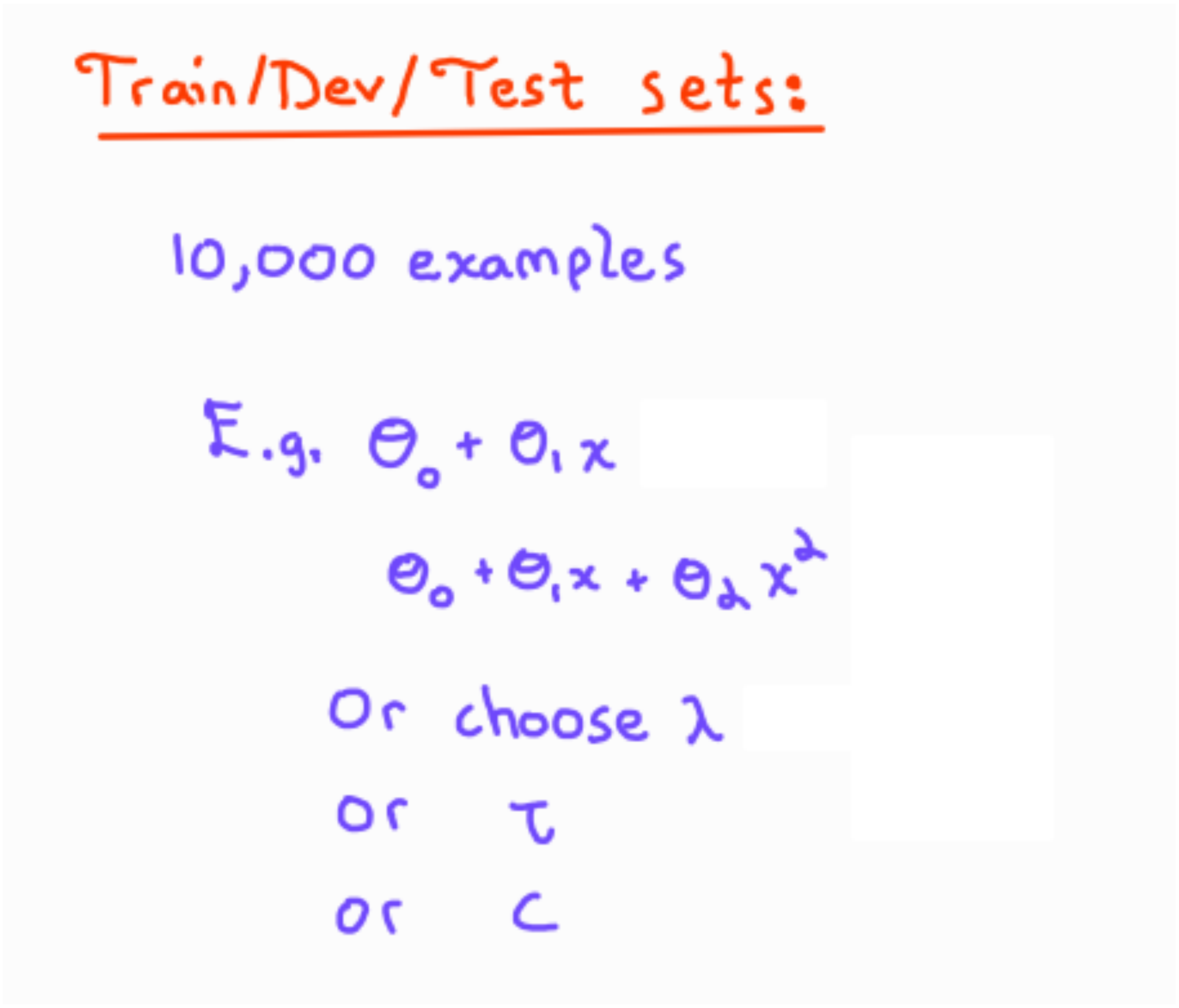


This curve is true for ***regularization*** as well. So, say you apply ***linear regression*** with **10,000** features to a very small training example, if **λ** was much too big, then it will ***underfit*** and if **λ** was **0** (so you're not ***regularizing*** at all), then it will ***overfit*** and there will be some intermediate value of **λ** that's not too big and not too small that balances ***overfitting*** and ***underfitting***

Train / Dev / Test Splits:

Given a dataset, what we'll often do is take your dataset and split it into different subsets, and a good hygiene is to take the data into a train, dev, and test sets

If you have **10,000** examples and you're trying to carry out this model selection problem, so for example, let's say you're trying to decide what order polynomial you want to fit, or you're trying to choose the value of **λ**, or you're trying to choose the value of **τ** (the bandwith parameter in locally weighted regression), or you're trying to choose a value **C** in a ***support vector machine***



Train/Dev/Test sets:

10,000 examples

E.g. $\theta_0 + \theta_1 x$

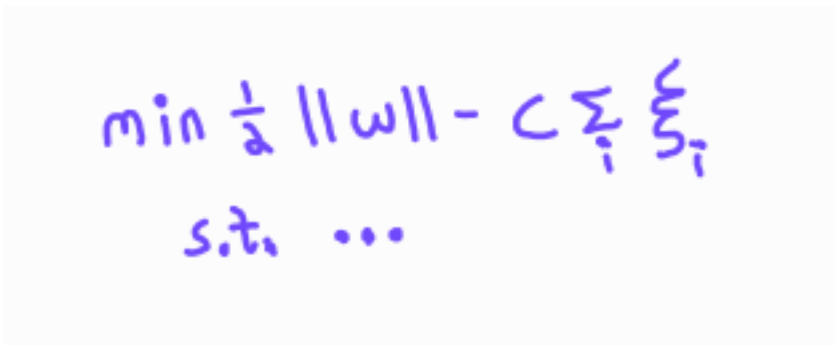
$\theta_0 + \theta_1 x + \theta_2 x^2$

Or choose λ

or τ

or C

The ***SVM objective*** was this:



$$\min \frac{1}{2} \|w\| - C \sum_i \xi_i$$

s.t. ...

For the ***L_1 norm soft margin***, you're trying to minimize the ***norm of w*** and there there was this additional parameter **C** that trades off how much you insist on classifying every training example perfectly

So whichever of these decisions you're trying to make, how do you choose a polynomial size or choose **λ** or choose **τ** or choose parameter **C** which also has this ***bias-variance tradeoff***. There'll be some values of **C** that are too large and some values of **C** that are too small

Train/Dev/Test sets:

10,000 examples

E.g. $\theta_0 + \theta_1 x$

$\theta_0 + \theta_1 x + \theta_2 x^2$

Or choose λ

or τ

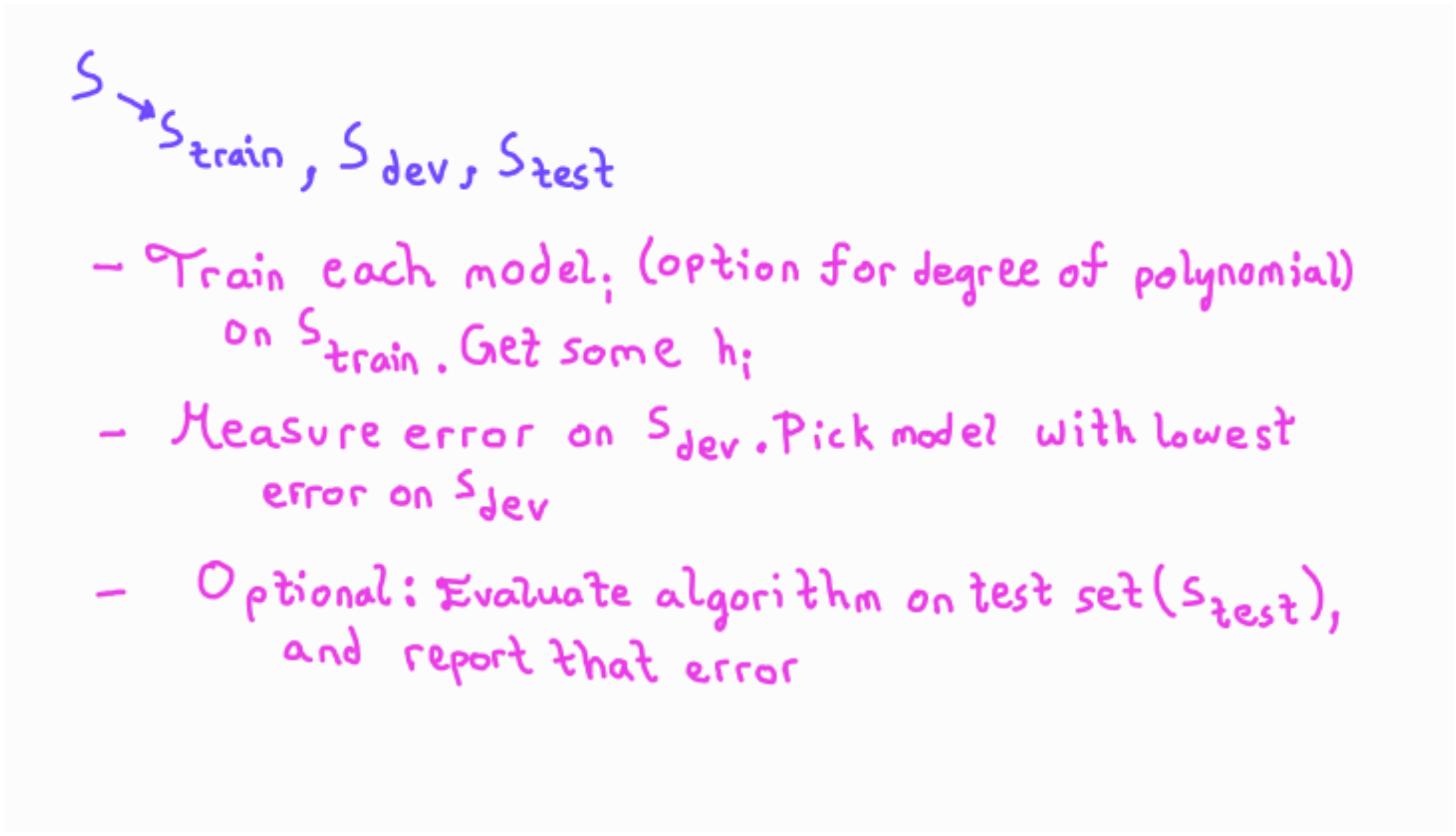
or C

$$\min \frac{1}{2} \|w\| - C \sum_i \xi_i$$

s.t. ...

Here's one thing you can do which is, split your training data **S** into a subset, called the ***real training set***, **S**$_{\textbf{train}}$, and then some subset called **S**$_{\textbf{dev}}$ and a separate test set **S**$_{\textbf{test}}$

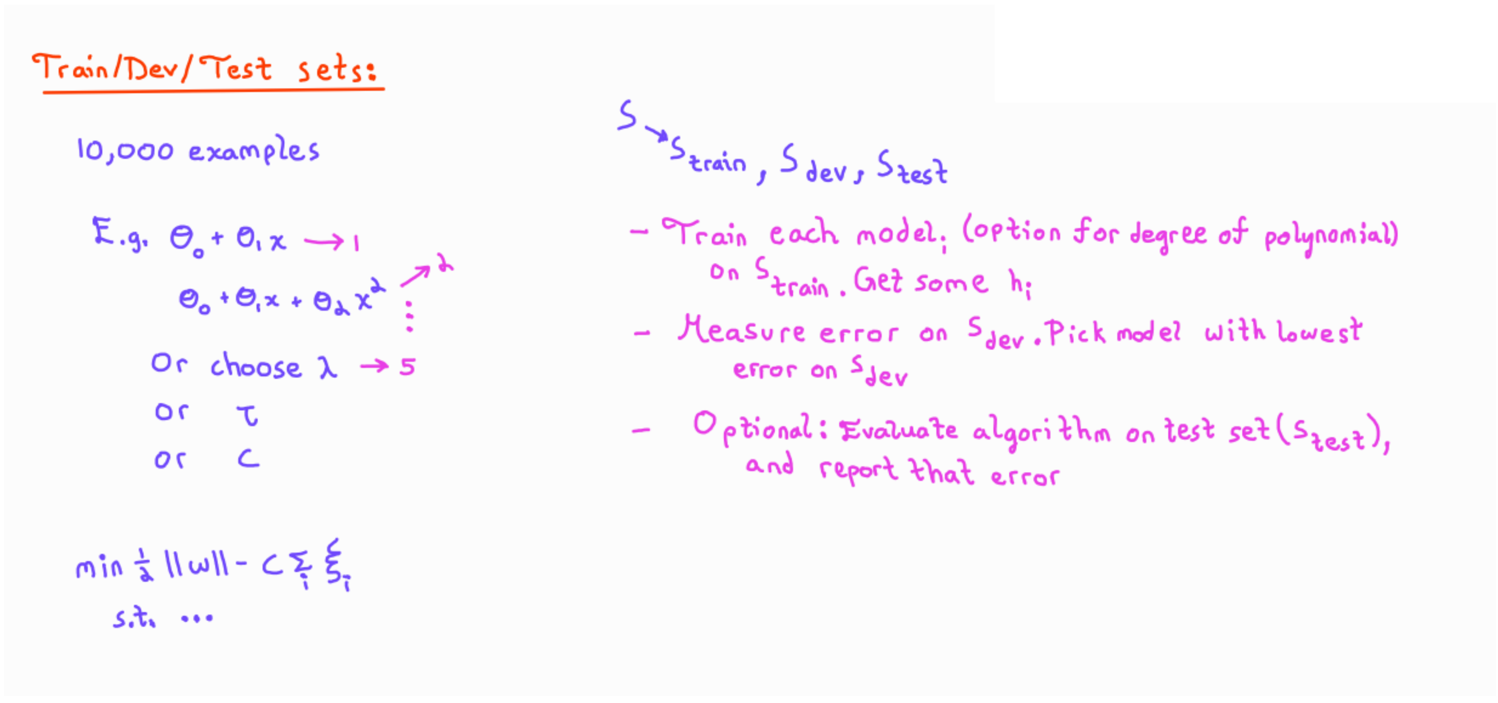
What you can do is:



$S \to S_{train}, S_{dev}, S_{test}$

- Train each model$_i$ (option for degree of polynomial) on S_{train}. Get some h_i
- Measure error on S_{dev}. Pick model with lowest error on S_{dev}
- Optional: Evaluate algorithm on test set (S_{test}), and report that error

You're evaluating a menu of models, so let's say these are models 1, 2, and so on:



Train/Dev/Test sets:

10,000 examples

E.g. $\theta_0 + \theta_1 x \to 1$

$\theta_0 + \theta_1 x + \theta_2 x^2 \nearrow 2$

$\vdots$

Or choose $\lambda \to 5$

or τ

or C

$\min \frac{1}{2}\|w\| - C\sum_i \xi_i$

s.t. ...

$S \to S_{train}, S_{dev}, S_{test}$

- Train each model$_i$ (option for degree of polynomial) on S_{train}. Get some h_i
- Measure error on S_{dev}. Pick model with lowest error on S_{dev}
- Optional: Evaluate algorithm on test set (S_{test}), and report that error

So the two subsets of the data, the ***training set*** and the ***development set***, after training first-order polynomial, second-order polynomial, third-order polynomial on the ***training set***, evaluate all of these different models on a separate ***held-out development sets*** and then pick the one with the lowest error on the ***development set***

But the one thing to not do would be to evaluate all these algorithms instead on the ***training set*** and then pick the one with the lowest error on the ***training set***. If do this, you will overfit because you'll always end up picking the fifth order polynomial because the more complex algorithm will always do better on the ***training set***. So if you do this, this will always cause you to say "let's use the fifth-order polynomial or the highest possible order polynomial", so this won't help you realize, in the housing price prediction example, the second-order polynomial is a better fit to the data

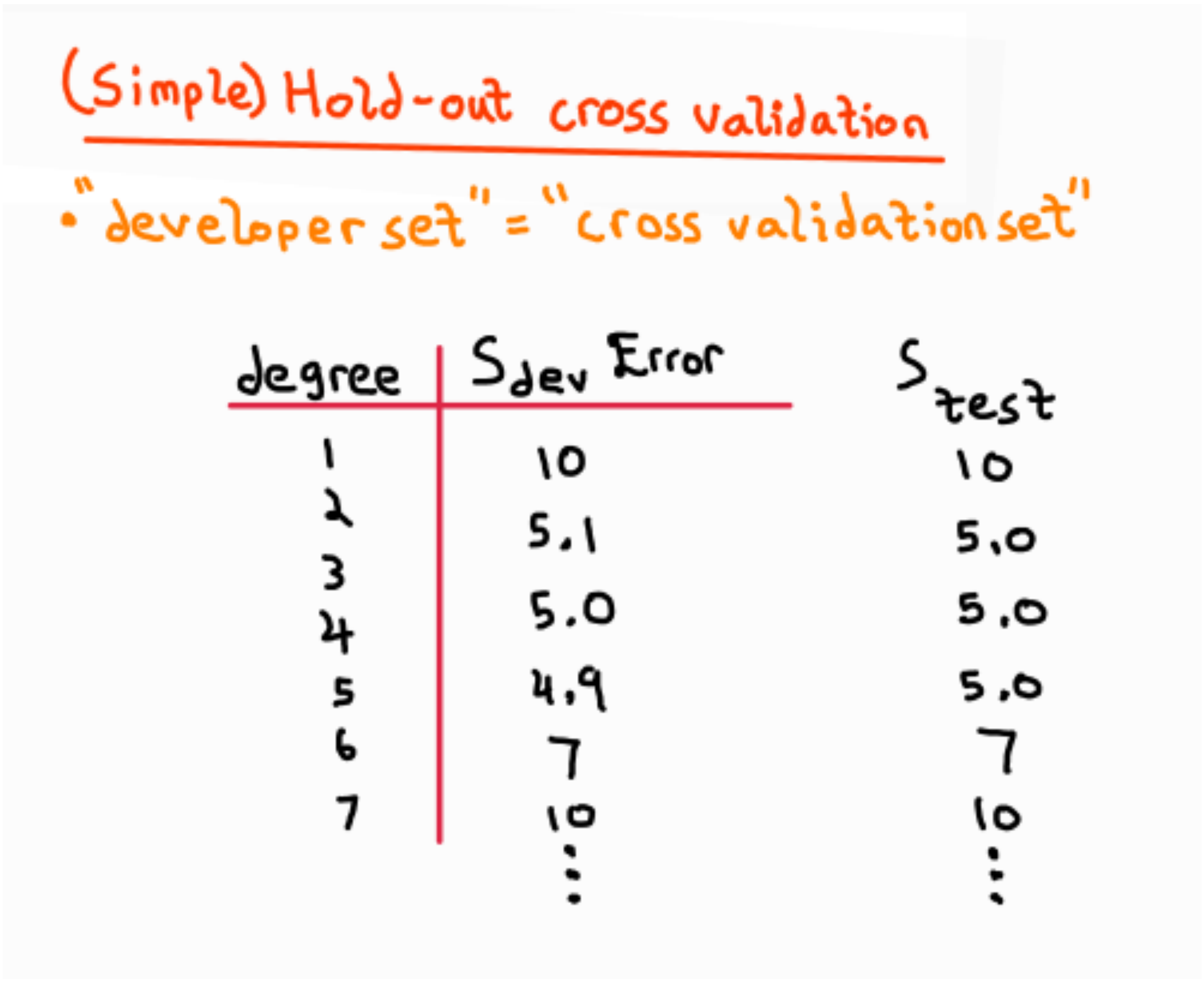
That's why for this procedure, if you evaluate your model's error on a separate ***development set*** that the algorithm did not see during training, this allows you to hopefully pick a model that neither ***overfits*** nor ***underfits***

If you are publishing an academic paper on machine learning that say's your algorithm achieves 90% accuracy on this dataset, it's not valid to report the results on the ***dev set*** because the algorithm has already been optimized to the ***dev set***. In particular, information about what's the best degree of polynomial to choose was derived from the ***development set***. And so, if you're publishing a paper or you want to report an unbiased result, evaluate the algorithm on a separate ***test set***, S_{test}, and report that error

So if you're publishing a paper, it would be considered good hygiene to report the error on a completely separate ***test set*** that you did not in any way, shape, or form, look at during the development of your model (during the training procedure)

Model Selection & Cross Validation:

Let's say you're trying to fit a degree of polynomial and you want to choose the ***dev error***. So we can fill the first, second, third, fourth, fifth degree polynomial. After fitting all of these, let's say that the ***squared error*** (just to use round numbers for illustrative purposes) is:



(Simple) Hold-out cross validation

• "developer set" = "cross validation set"

degree	S_{dev} Error	S_{test}
1	10	10
2	5.1	5.0
3	5.0	5.0
4	4.9	5.0
5	7	7
6	10	10
7	⋮	⋮

If you're using the ***dev error*** to pick the best hypothesis, you would say that using the fifth-order polynomial gets you **4.9 *squared error***, but did you really earn that **4.9 *squared error*** or did you just get lucky? Because there is some noise and so maybe all of these actually have error that's close to **5.0** but some are just higher, some are just lower, and you just got a little bit lucky that on the ***dev set*** it did better. Which is why, if you look at your ***dev set error***, your ***dev set error*** is a biased estimate

And so, where as your ***test set*** (or very large ***test set***), maybe the true numbers are your actual expected ***squared errors***. It's just that because of a little bit of noise, you got lucky and reported **4.9**. And so this would be bad thing to do in an academic paper because what you earned was an error of **5.0**, you didn't earn an error of **4.9**. It's just that because you're over-fitting a little bit in the ***dev set***, you chose the thing that looked best for the dev set but your algorithm didn't actually achieve that error, it's just because of noise

Reporting on the ***dev error*** isn't really a valid unbiased procedure, so it's considered good practice to report the ***test error***

One of the problems with some of the machine learning benchmarks that people worked on for a long time, is this unavoidable mental overfitting. The people had gotten to use the dataset and everyone's working on the same, trying to publish the best numbers from the same ***test set***. So the academic committee on machine learning does have some amount of overfitting to the standard benchmarks that people have worked on for a long time and this is an unfortunate result

When the ***test set*** is very very large, the amounts of overfitting is probably smaller, but when the ***test set*** is not big enough, then the overfitting result can cause, sometimes even research papers, to publish results that are probably overfitted to the dataset

One standard academic benchmark was the dataset called CIFAR (it's quite small). There's actually a very interesting research paper analyzing results on CIFAR, arguing that some fraction of their progress that was made was actually perhaps researchers unintentionally overfitting to this dataset

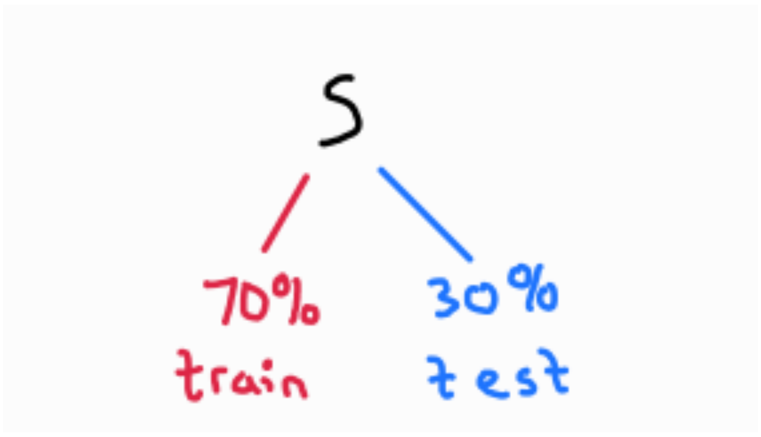
One thing to do when building production machine learning systems (for example, you just want to build a speech recognition system and just make it work and you're not trying to publish a paper, not trying to make some claim), sometimes you don't have to bother to have a ***test set***. This means you won't know the true error of the system sometimes, but you should be very conscious of that

If you don't have a lot of data, sometimes you may decide to just not have a ***test set*** and it means you don't try to report the ***test set*** number. You can report the ***dev set*** number, which you know is biased, and you just don't report the ***test set*** number

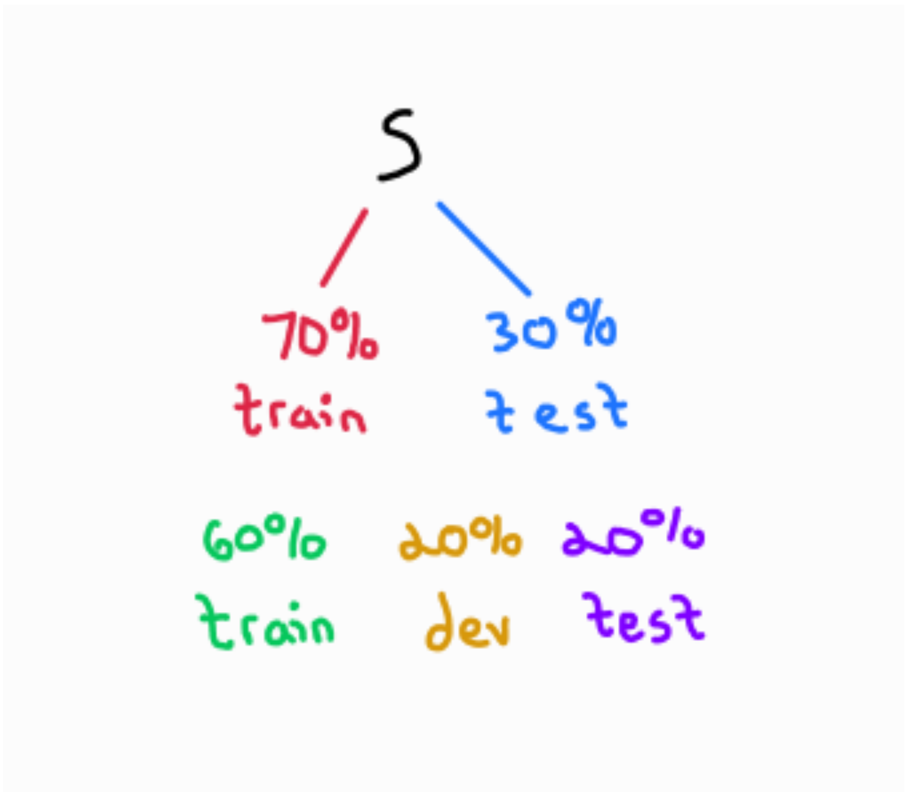
Don't do this if you're publishing an academic paper. This is not good if you're publishing a paper or making claims on the outside, but all we're doing is building a product and not writing a paper out

The next topic about seeting up the ***train***, ***dev***, ***test split*** is, ‘How do you decide how much data should go into each of these three subsets?’

Historical perspective: Historically, the rule of thumb was you take a ***training set*** **S**:



Modern perspective: If you're not doing ***model selection*** where you've already picked a model and not revising:



These are rules of thumb that people used to give. These are decent rules of thumb when you don't have a massive dataset. So you may have **100** examples, maybe you have **1,000** examples, maybe several thousand examples, these rules of thumb are perfectly fine

As you move to machine learning problems with really really giant datasets, the percentage of data you send to ***dev*** and ***test*** are shrinking

In the normal workflow of developing a learning algorithm, when you're given a dataset, you could split it into a ***training set*** and a ***dev set***, and if you need a ***test set*** then also a ***test set***, and then keep on fitting the parameters to the ***training set*** and evaluating the performance of your algorithm on the ***dev set***, and using that to come up with new features, choose the model size, choose the regularization parameter like **λ**, really try out lots of different things and spend several days or weeks to optimize the performance on the ***dev set***, and then when you want to know "how well is your algorithm performing?" to then evaluate the model on the **test set**. And the thing to be careful not to do is to make any decisions about your model using your ***test set***, because then your assigned to fit the data to the ***test set*** and it's no longer an unbiased estimate

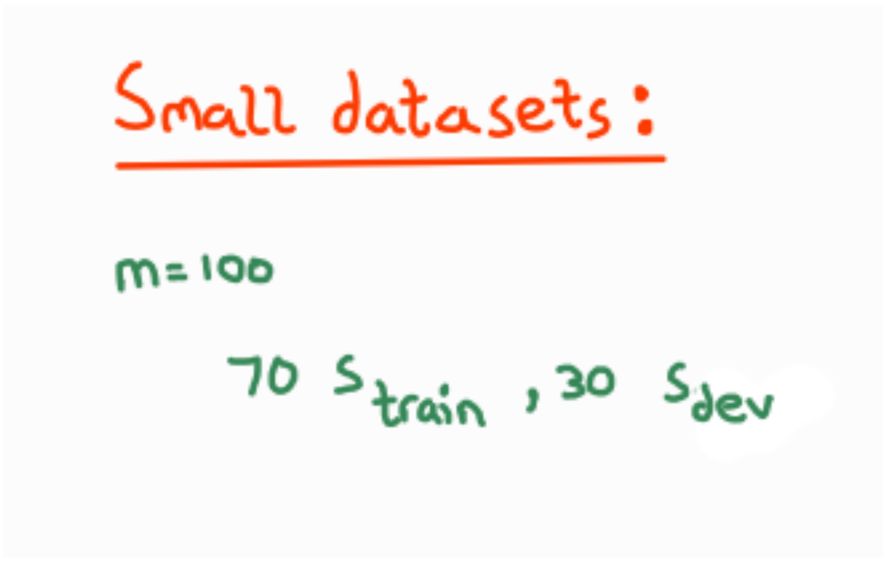
One thing that is actually okay to do is, if you have a team that's working on a problem, if every week they measure the performance on the ***test set*** and report out on a chart the performance of the ***test set***, that's actually okay. You can evaluate the model multiple times on the ***test set***. You can actually give out a weekly report saying 'this week, for our online advertising system, we have this result on the ***test set***'

It's actually okay to evaluate your algorithm repeatedly on the ***test set***. What's not okay is to use those evaluations to make any decisions about your learning algorithms

For example, if one day you notice that your model is doing worse this week than last week on the ***test set***, if you use that to revert back to an older model, then you've just made a decision that's based on the ***test set*** and your ***test set*** is no longer unbiased. But if all you do is report out the results but not make any decisions based on the ***test set*** performance, such as whether to revert to an earlier model, then you can it's actually okay to keep on using the same ***test set*** to track your team's performance over time

When you have very large datasets, this is the procedure for defining the ***train***, ***dev***, and ***test sets*** and this procedure can be used to choose the model of polynomial. It can also be used to choose the regularization parameter **λ** or the parameter **C** or the parameter **τ** from now ***locally weighted regression***

What if you have a very small dataset?

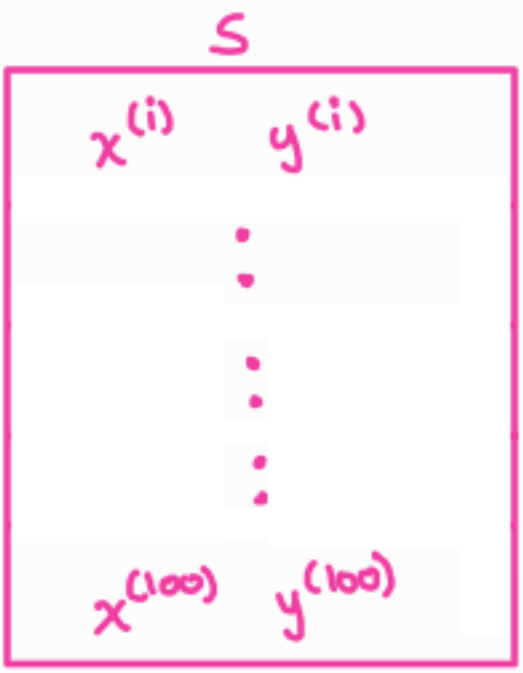

Small datasets:
m=100
70 S_train , 30 S_dev

Is there a way to, say, do ***model selection*** such as, choose the degree of polynomial, without 'slightly wasting so much of the data'?

There is a procedure that you should use only if you have a small dataset called **K-fold Cross Validation**. This is in contrast to **Simple Cross Validation**

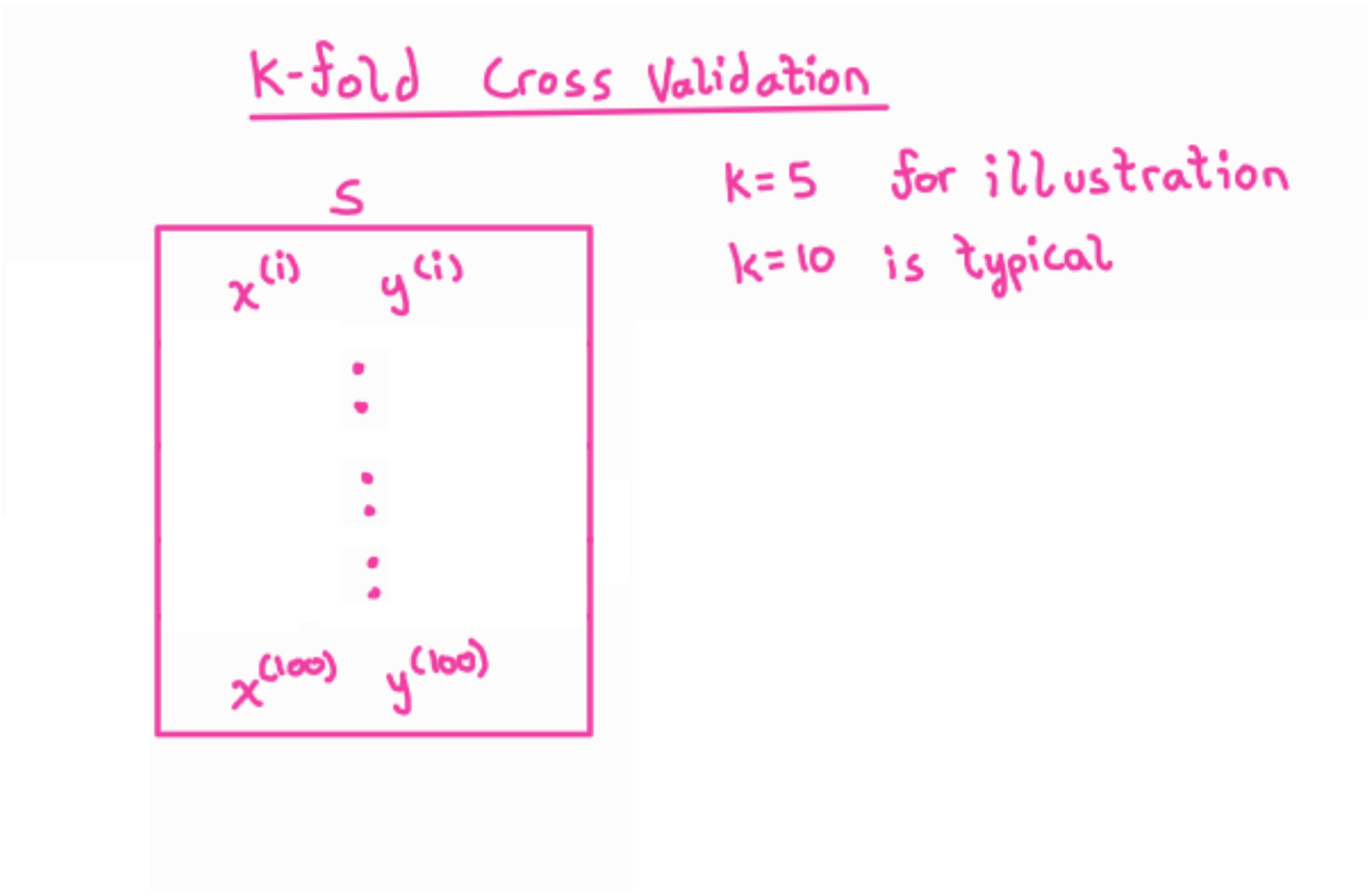
Let's say this is your ***training set* S**:

K-fold Cross Validation

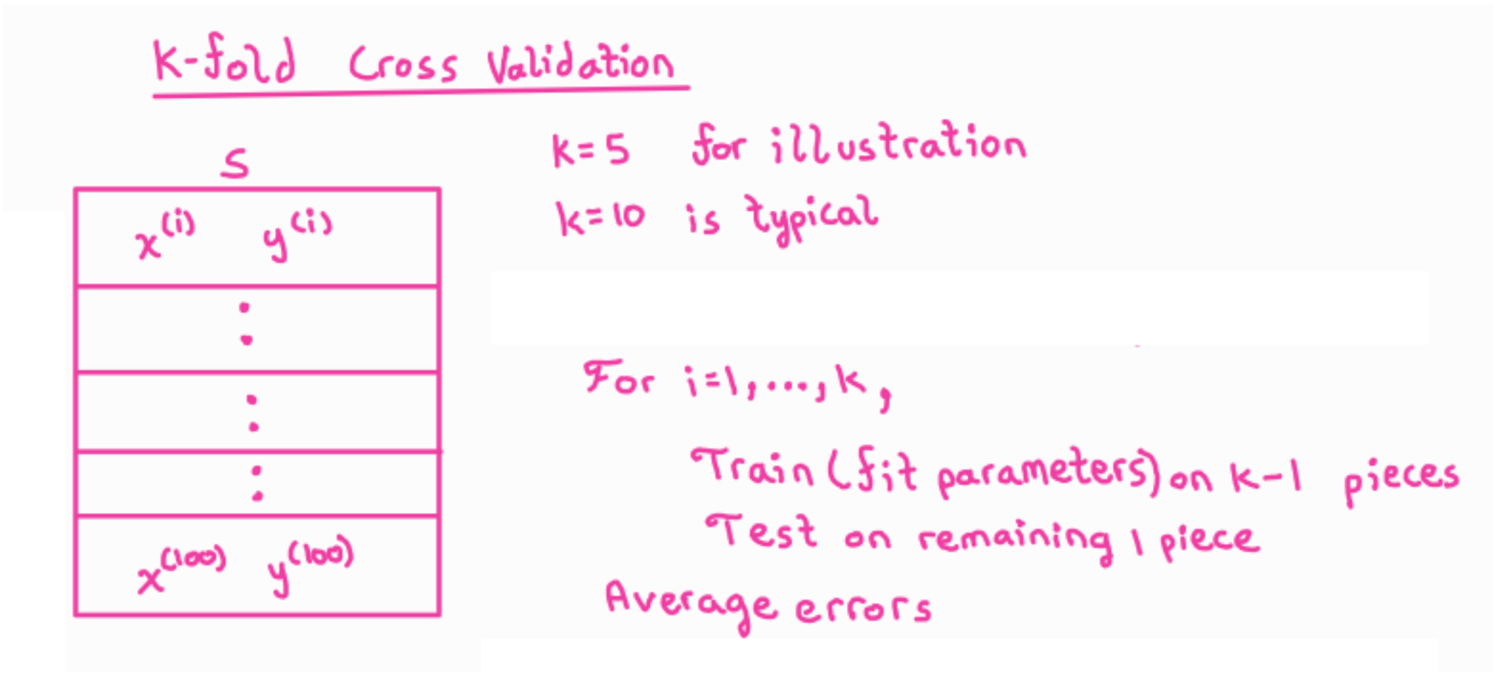


What we're going to do is take the ***training set*** and divide it into **k** pieces

For the purpose of illustration, we're going to use **k=5**, but **k=10** is typical

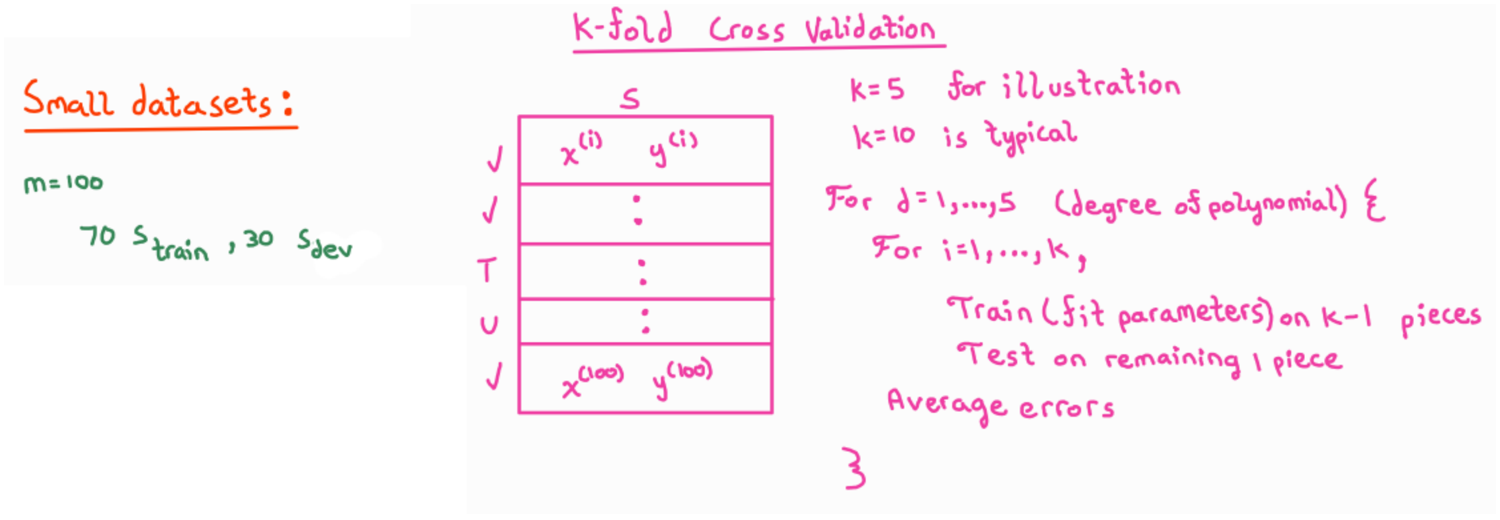


What you do is take your dataset and divide it into five different subsets and what you do is:



In other words, when **k=5**, we're going to loop through five times. In the first iteration, we're going to train on 4 out of five of the pieces and evaluate your model on the fifth piece

If you're trying to choose a degree of polynomial, what you would do is:



Small datasets:

$m=100$

70 S_{train}, 30 S_{dev}

K-fold Cross Validation

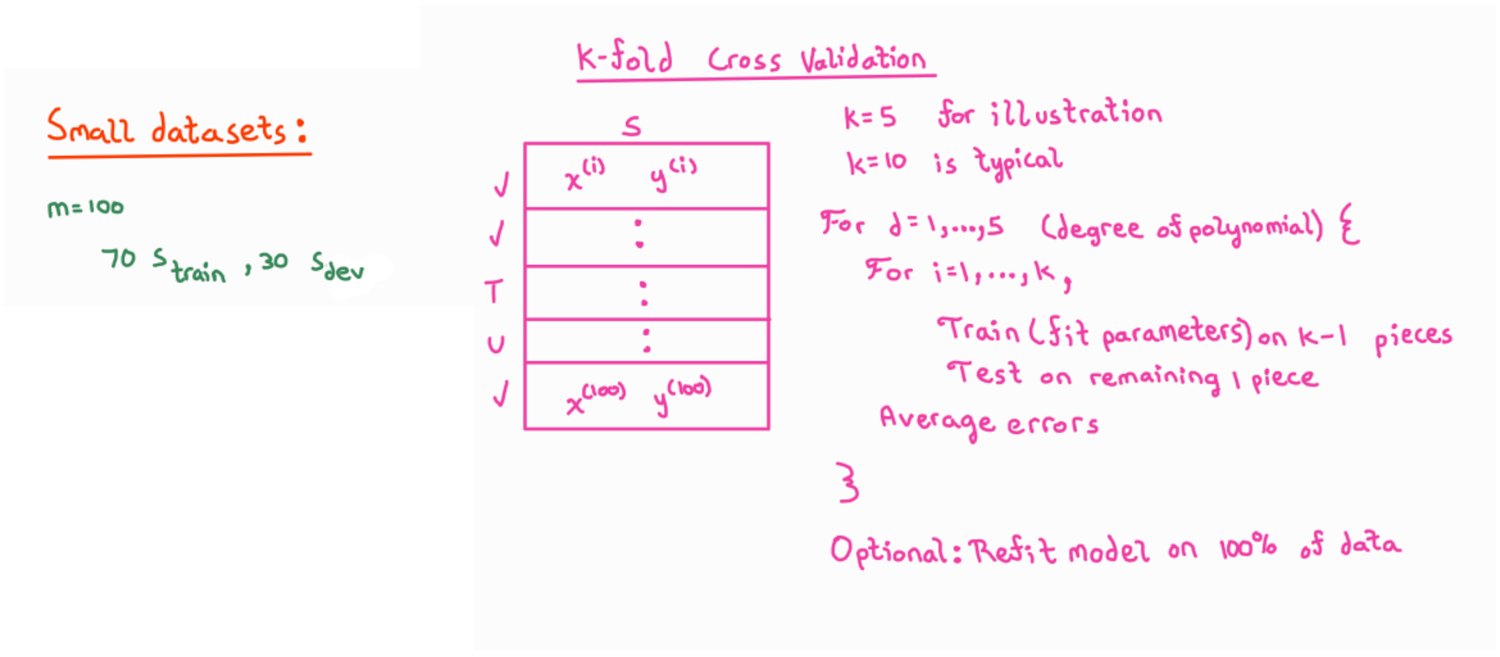
$k=5$ for illustration

$k=10$ is typical

```
For d = 1,...,5 (degree of polynomial) {
    For i=1,...,k,
        Train (fit parameters) on k-1 pieces
        Test on remaining 1 piece
    Average errors
}
```

You do this procedure for a first-order polynomial, you fit a linear regression model five times, each time on four-fifths of the model and test on the remaining one fifth, and then for each of these models you would then average the five estimates you have for ***S error*** (test errors), and you repeat this whole procedure for the quadratic function, repeat this whole procedure for the cubic function, and so on. After doing this for every order polynomial from 1, 2, 3, 4, 5, you would then pick the degree of polynomial that did best according to this metric

Now you actually end up with five classifiers. Because you have five classifiers, each one fits on four-fifths of the data and then there's a final optional step:



Small datasets:

$m=100$

70 S_{train}, 30 S_{dev}

K-fold Cross Validation

$k=5$ for illustration

$k=10$ is typical

```
For d = 1,...,5 (degree of polynomial) {
    For i=1,...,k,
        Train (fit parameters) on k-1 pieces
        Test on remaining 1 piece
    Average errors
}
```

Optional: Refit model on 100% of data

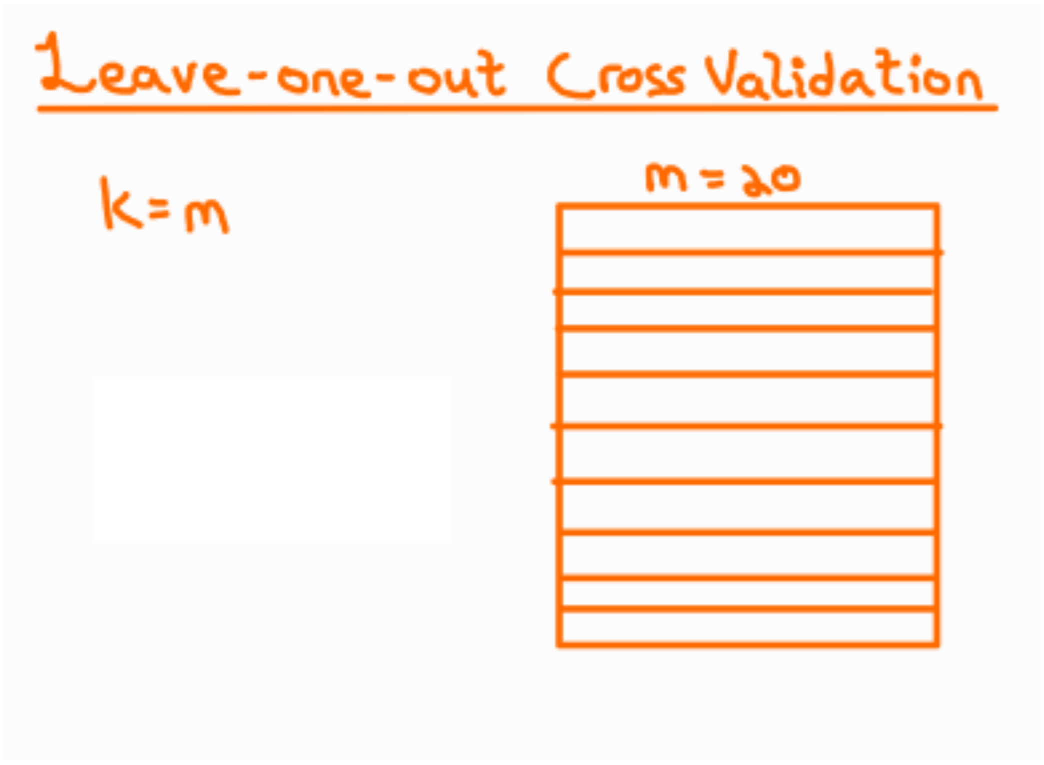
The advantage of ***k-fold cross validation*** is that instead of leaving out 30% of your data for your ***dev set*** on each iteration, you're only leaving out **1/k** of your data This procedure compared to ***simple cross validation***, it makes more efficient use of the data because you're holding out only 10% of the data on each iteration

The disadvantage of this is it's computationally very expensive, that you're now fitting each model 10 times instead of just once

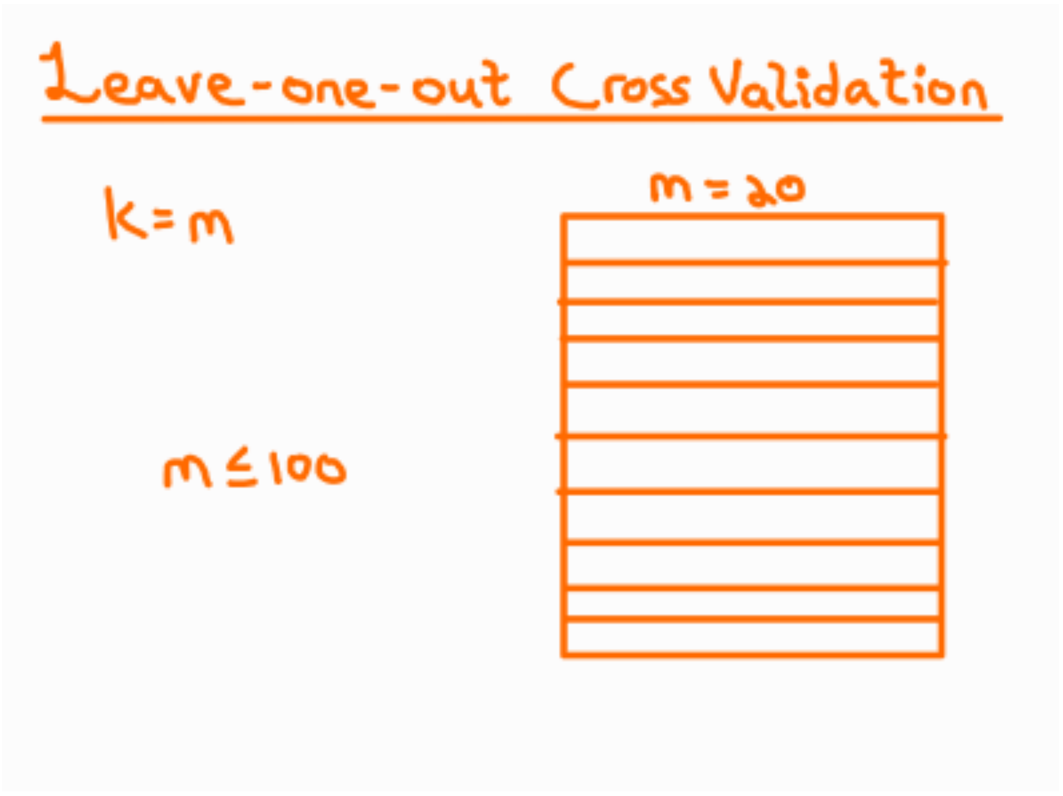
But when you have a small dataset, this is actually a better procedure than ***simple cross validation*** if you don't mind the computational expense of fitting each model 10 times. This actually lets you get away with holding out less data

There's one even more extreme version of this, which you should use if you have very very small datasets. So sometimes you might have an even smaller dataset. If you're doing a class project with 20 examples, there's an extreme version of ***k-fold cross validation*** called ***leave-one-out cross validation***, which is if you set **k=m**

In other words, here's your ***training set*** of maybe 20 examples, you're going to divide this into as many pieces as you have training examples and what you do is leave out one example, train on the other 19, and test on the one example you held out, and then leave out the second example, train on the other 19, and test then test on the one example you held out, and do that 20 times, and then you average this over the 20 outcomes to evaluate how good different orders of polynomials are



The huge downside of this is this is computationally very very expensive because now you need to train your algorithm **m** times. So you kind of never do this unless **m** is really small. So if **m** is less than 100, you could consider this procedure, but if **m** is **1,000**, fitting a linear model **1,000** times just seems like a lot of work so you'd usually use ***k-fold cross validation*** instead



#Since you have **k** estimates, say **10** estimates, you're using 10-fold cross validation, can you measure the variance on those 10 estimates?

It turns out that those 10 estimates are correlated because each of the 10 classifiers, 8/9 of the sets of data they trained on overlap

https://mlyearning.org

Feature Selection:

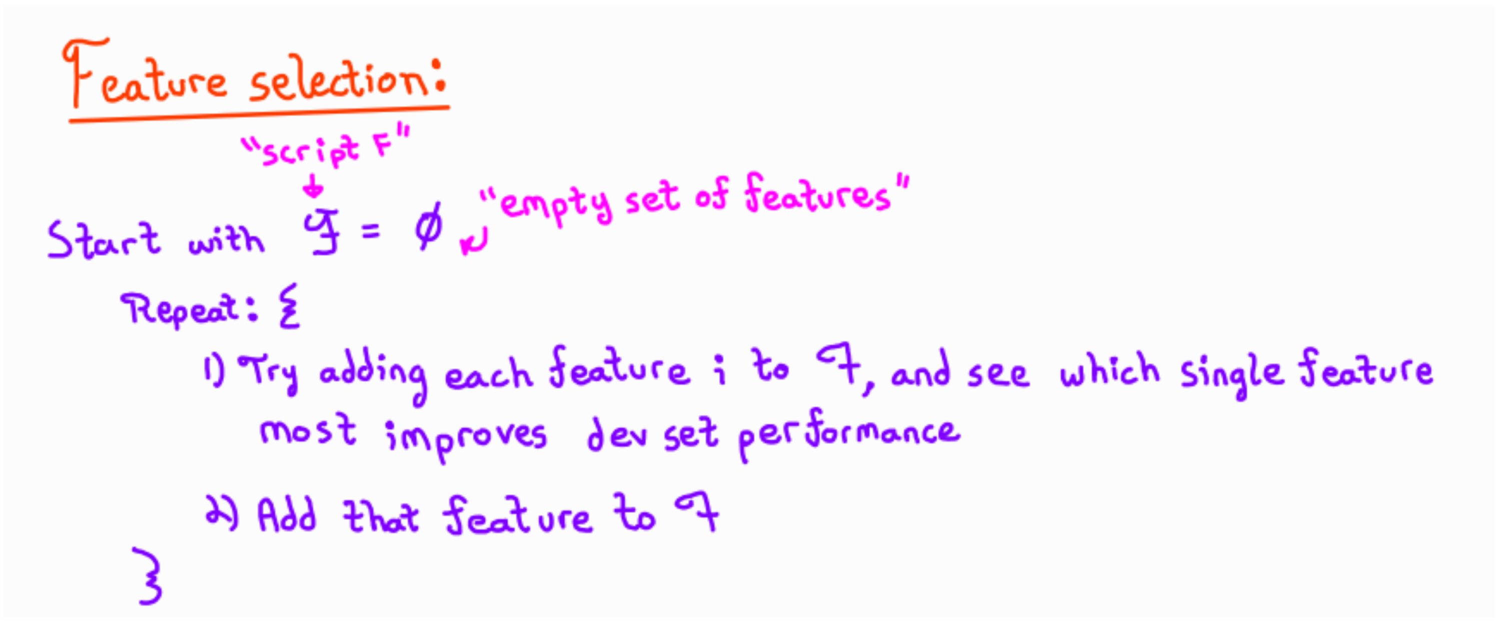
Sometimes you have a lot of features but you might suspect that a lot of the features are not important. So if you have a lot of features, sometimes one way to reduce overfitting is to try to find a small subset of the features that are most useful for your task. This takes judgement

There are some problems, like ***computer vision*** where you have a lot of features corresponding to there being a lot of pixels in every image, but ,probably, every pixel is somewhat relevant. So you don't want to select a subset of pixels for most computer vision tasks

But there are some other problems where you might have a lot of features and you suspect the way to prevent overfitting is to find a small subset of the most relevant features for your task

Feature selection is a special case of ***model selection*** that applies to when you suspect that even though you have **10,000** features, maybe only **50** of them are highly relevant

One example: If you are measuring a lot of things going on in a truck in order to figure out if the truck is about to break down, for preventive maintenance, you might measure hundreds of variables or many hundreds of variables, but you might secretly suspect that there are only a few things that predict when this truck is about to go down. So if you suspect that's the case, then feature selection would be a reasonable approach to try



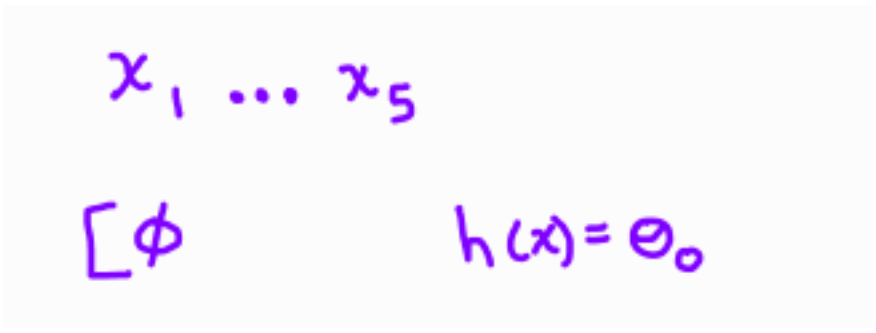
Feature selection:

"script F"

Start with $\mathcal{F} = \emptyset$ "empty set of features"

```
Repeat: {
    1) Try adding each feature $i$ to $\mathcal{F}$, and see which single feature
       most improves dev set performance
    2) Add that feature to $\mathcal{F}$
}
```

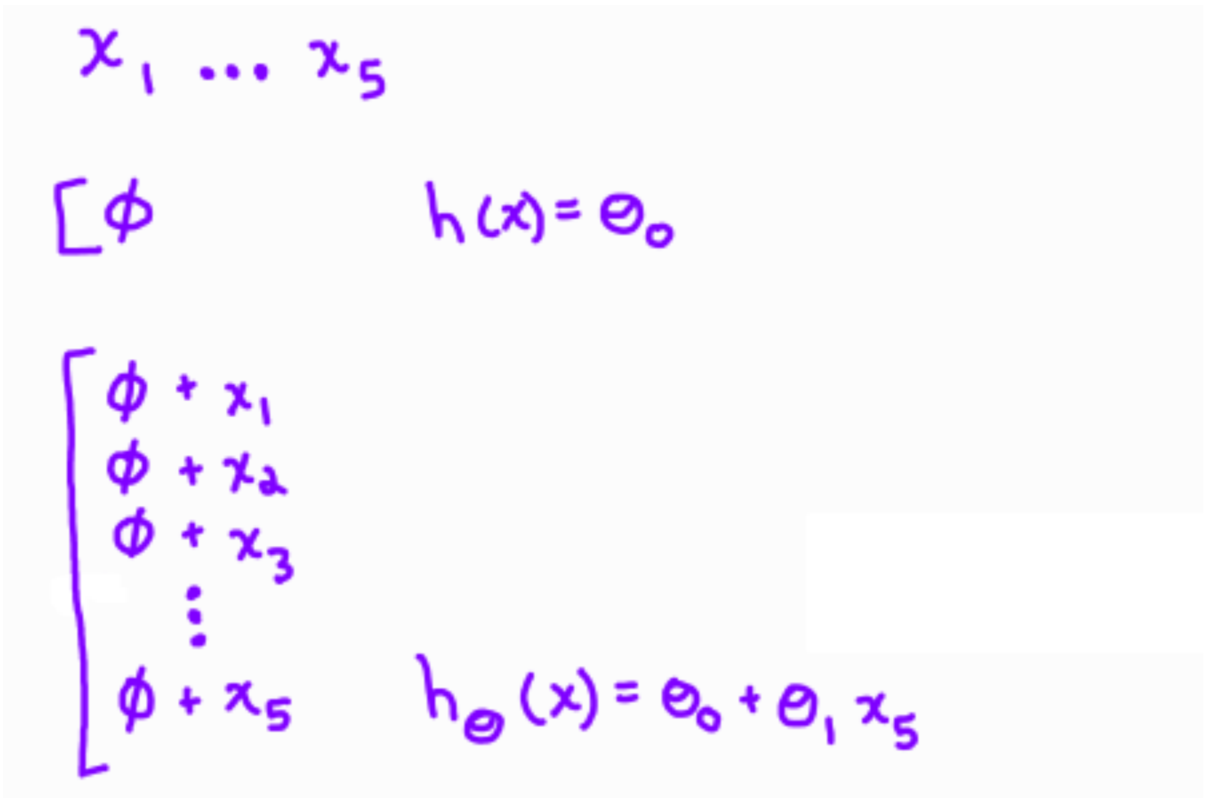
Let's say you have 5 features. So start off with an empty set of features and train a linear classifier with no features:



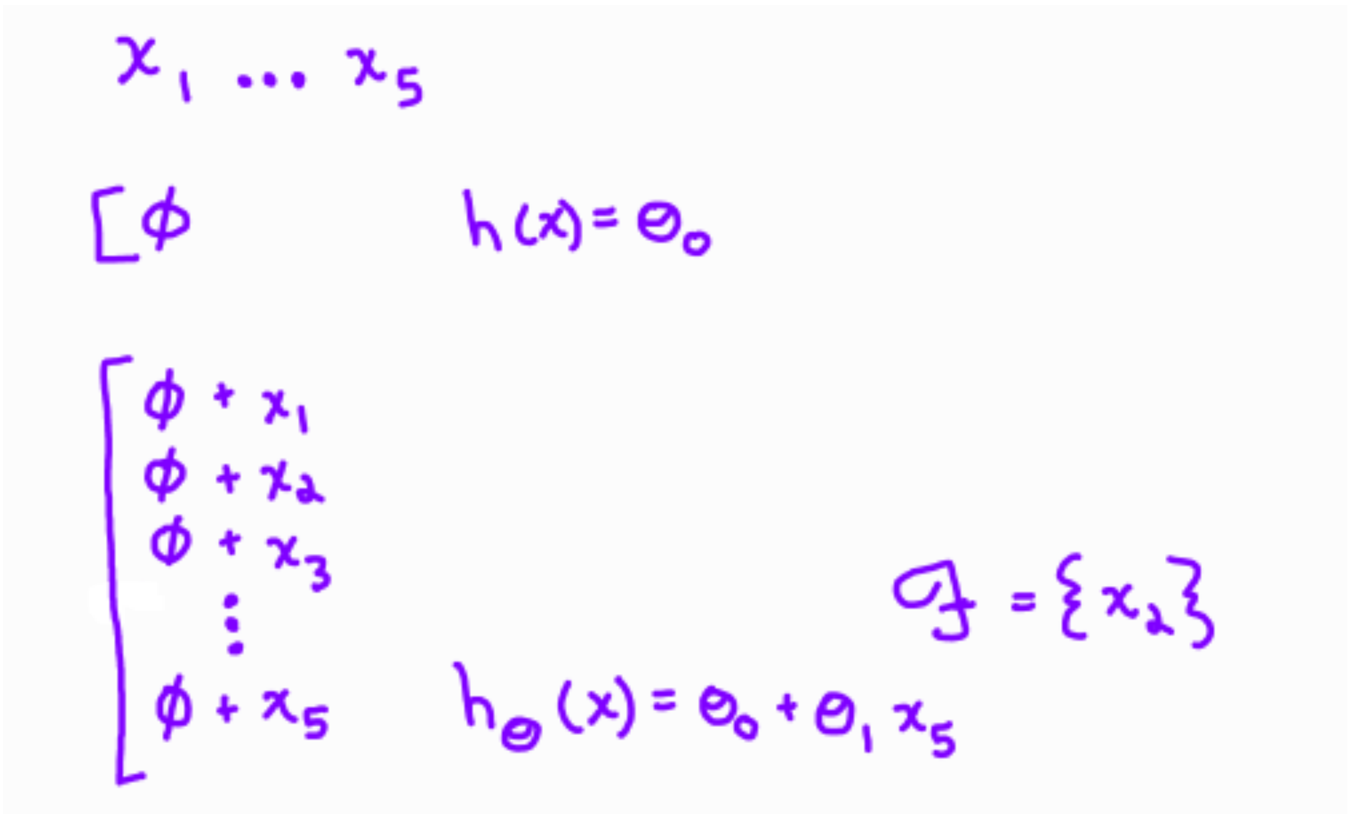
$x_1 \ldots x_5$

$[\emptyset \qquad h(x) = \theta_0$

This won't be a very good model but see how well this does on your ***dev set***. So this way you average the **y**'s, so it's not for your model (*first step*). In the second iteration, you would then take each of these features and add it to the empty set, and for each of these you would fit a corresponding model

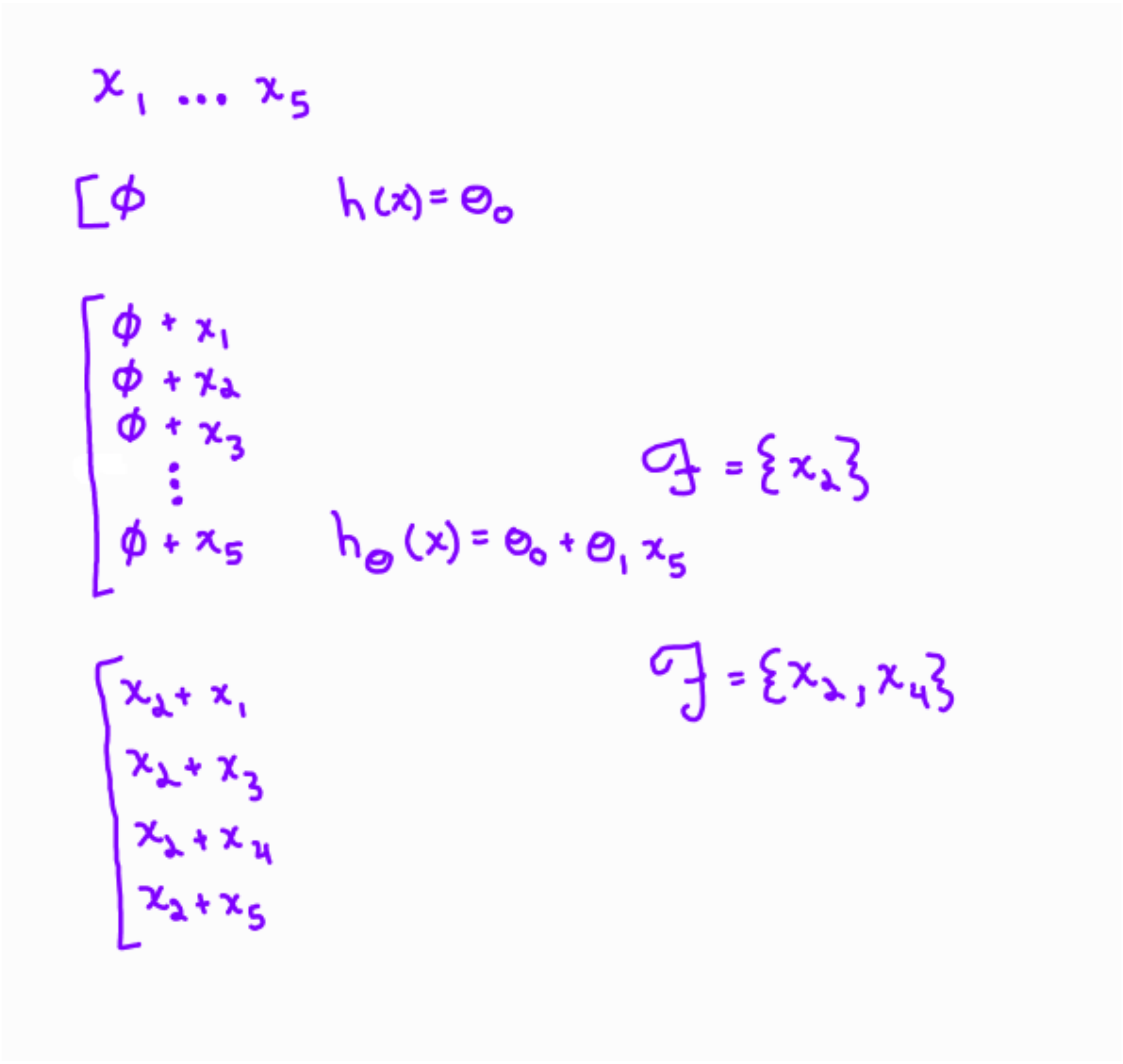


So try adding one feature to your model and see which model best improves your performance on the ***dev set***. And let's say you find that adding feature two is the best choice. So now, what we'll do is set the set of feature to be $\mathbf{x_2}$



For the next step, you would then consider starting with $\mathbf{x_2}$ and adding $\mathbf{x_1}$, $\mathbf{x_3}$, $\mathbf{x_4}$, or $\mathbf{x_5}$. So if your model is already using the feature $\mathbf{x_2}$, what the additional feature most helps your algorithm?

Let's say that it's $\mathbf{x_4}$. So you fit four models, see which one does best, and now you commit to using the features $\mathbf{x_2}$ and $\mathbf{x_4}$



You kind of keep on doing this, keep on adding features greedily, keep on adding features one at a time to see which single feature addition helps improve your algorithm the most, and you can keep iterating until adding more features now hurts performance, and then pick whichever feature subset allows you to have the best possible performance on the ***dev set***

This is a special case of ***model selection*** called ***forward search***. It's called ***forward search*** because we started with an empty set of features and adding features one at a time. There's also a procedure called ***backwards search*** where we start with all the features and remove features one at a time

Forward search would be a reasonable feature selection algorithm. The disadvantage of this is it is quite computationally expensive, but this can help you select a decent set of features