Tuning Models for Supervised Statistical Learning

Siva Balakrishnan
Data Mining: 36-462/36-662

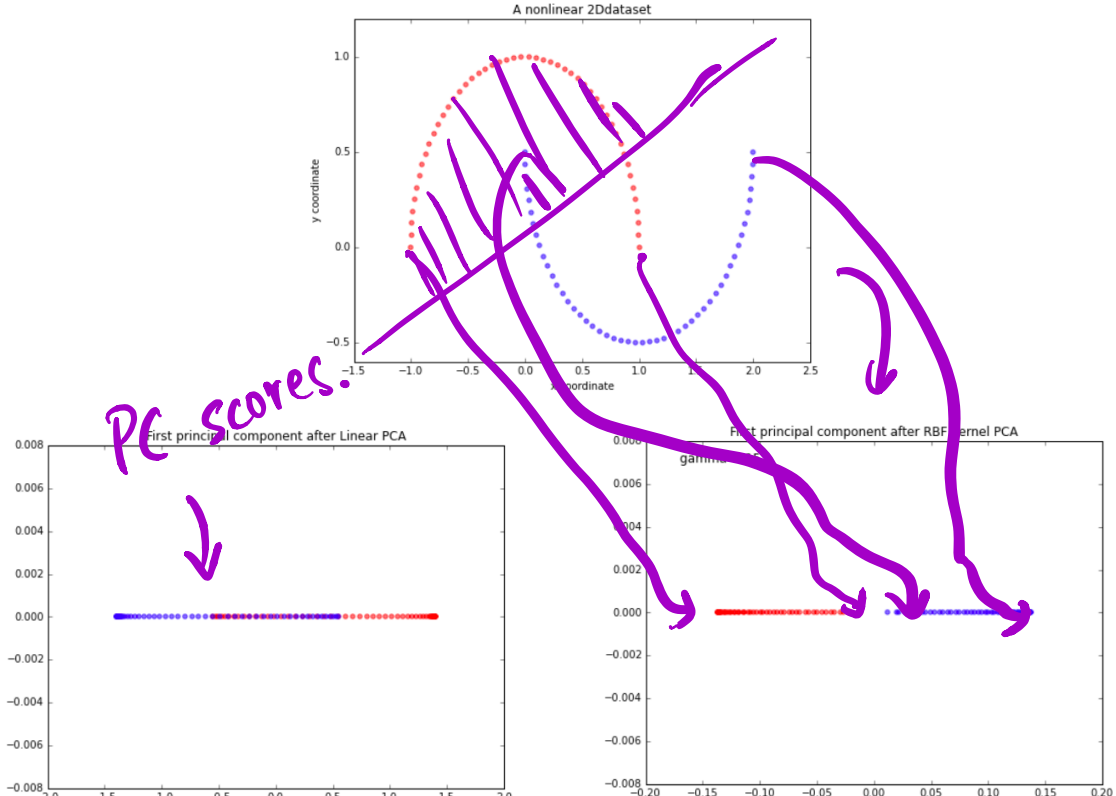
April 16th, 2019

Outline for Today

- Recap: Non-linear Dimension Reduction Methods
- Some General Model Training Guidelines

Non-linear Dimension Reduction

- Just like non-linear classifiers can be flexible and useful, we often want to construct non-linear mappings of our data to a lower dimensional space for visualization.
- PCA is a linear dimension reduction method.

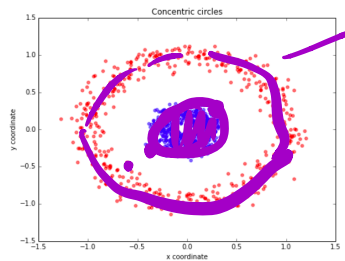


Kernel PCA

- Observe that to compute the PCA projection we do not need X, just the inner-product matrix XX^T suffices.
- Can use the kernel-trick: replace $x_i^T x_j$ by $K(x_i, x_j)$ for some kernel. This gives us the kernel gram matrix K, and we can compute its top few eigenvectors to get the PC scores and visualize our data.

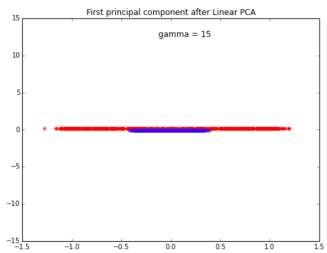
inner product

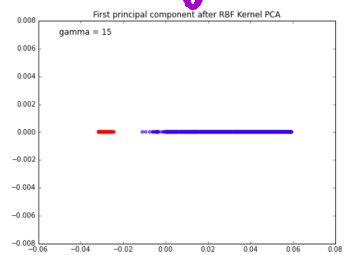
$\exp(-\|x_i - x_j\|_2^2)$



graph dist better.

PCA after non-linear map.



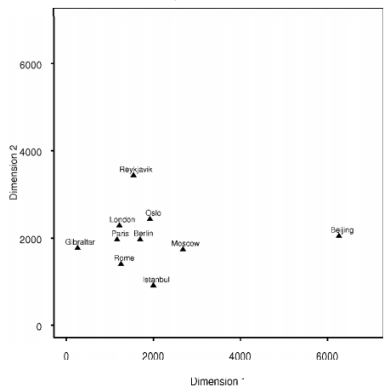


From pairwise distances to visualization

- Given just the matrix of pairwise distances $\Delta_{ij} = \|X_i - X_j\|_2$ we can visualize our data. This is called classical *multi-dimensional scaling*.

"map".

	London	Berlin	Oslo	Moscow	Paris	Rome	Beijing	Istanbul	Gibraltar	Reykjavik
London	–									
Berlin	570	–								
Oslo	710	520	–							
Moscow	1550	1000	1020	–						
Paris	210	540	830	1540	–					
Rome	890	730	1240	1470	680	–				
Beijing	5050	4570	4360	3600	5100	5050	–			
Istanbul	1550	1080	1520	1090	1040	850	4380	–		
Gibraltar	1090	1450	1790	2410	960	1030	6010	1870	–	
Reykjavik	1170	1480	1080	2060	1380	2040	4900	2560	2050	–

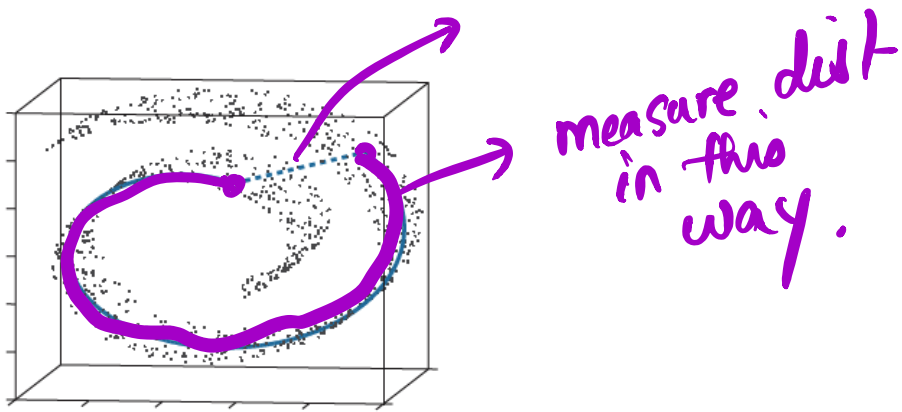


- Key idea: use some transformations to go from $\Delta \to XX^T$ and then compute the PC scores.

Recap: From non-Euclidean pairwise distances to visualization

There is a class of methods which construct a fancier metric Δ_{ij} between high-dimensional points $x_1, \ldots x_n \in \mathbb{R}^p$, and then they feed these Δ_{ij} through multidimensional scaling to get a low-dimensional representation $z_1, \ldots z_n \in \mathbb{R}^k$.

In this case, we don't just get principal component scores, and our low-dimensional representation can end up being a nonlinear function of the data



Recap: Isomap Algorithm

- Build graph.
- Compute shortest path distances.
- Perform MDS on shortest path distances to obtain visualization.

Recap: Laplacian Eigenmaps

XX^T , Δ.

We can derive another non-linear visualization method by trying to optimize the locations of the points so as to not distort small distances. This leads to Laplacian Eigenmaps.

- Build graph.
- Find Laplacian, compute bottom k eigenvectors $V \in \mathbb{R}^{n \times k}$.
- Visualize using V.

The same reason why V helps us in clustering applies here, it helps us in visualization by non-linearly preserving clusters.

→ plot rows of V.

You have data, now what?

- Most of this is good to know, and some of it will be useful for your project.
- Do you have a prediction (classification/regression) task in mind?
- Perhaps someone gives you data. In more realistic circumstances, you've scraped together your own data-set. Now what?
- Note: much of what follows contains opinion and over-simplification/over-generalization, rather than pure fact. There are many ways to approach such problems.

Getting started

- Figuring out what you have: Variable types, distributions, relationships
- Looking for problems: Outliers, missing data, inconsistent coding

Get a feel for what's in your data set

Just explore your data set, so you know what to expect

- Dimension of the data, variables and observations.
- Variable types: factors, dates, numeric, geographic locations.
- Variable distributions: Number of factor levels, imbalance of factor levels, skewness of variables.
- Simple relationships between variables. Compute/visualize covariance matrix.

Plot your data!

You will have seen plenty of this in graphics/other classes, so we won't get into it much, but will give some ideas.
Ideas:

- Plot univariate and bivariate distributions of variables. Are variables continuous? Bump like? Mixture like?
- Look at conditional plots (for different classes – we did this for Naïve Bayes and Logistic Regression) and investigate differences

This will:

- Inform your modeling
- Guide your feature extraction

Outliers

Outliers are not very well-defined, so there's no one method for finding them.

You are worried about points that are abnormal enough that they unduly influence your model. They may represent:

- Mistakes
- Observations that are not coming from the process you want to model

Think of them as points that are not good to think of as part of $\mathcal{P}$.

They can be identified in plots, or by a variety of rules (large Mahalanobis distance, high leverage, and many more).

Sometimes you can fix them (typos, special events). Other times this entry is essentially missing data.

Missing data

→ Missing CAR
→ Not at random.

You will often find missing values in data sets. You need to decide how to handle them when you make predictions.

- You could remove these whole observations (complete-case analysis). If the missingness is not *informative*, this only costs you data. If it is informative, you are also losing information and biasing your population.

→ complete-case analysis can be biased.

- You could try to fill in the missing values: imputation
  1. Use a strongly correlated variable to predict this one.
  2. Nearest neighbors: find the K closest other points and average their value for this entry.

→ try to fill in missing data.

- You could code the NA information as another value (sometimes)

What is your outcome of interest?

Is it a continuous regression problem?

- Do you really care about the value, or just whether it's big?
- How is your variable behaved? Heavily skewed? Do you see a lot of the big values that you care about?

Is it a classification problem?

- How many classes do you have?
- What are the default class proportions? What is your base rate? Do you have severe imbalance?
- What is your goal and what are your costs?
  - Do you want better misclassification, or just enriched subsets of the data?
  - Are your costs asymmetric enough that you should be thinking about different cutoffs, sensitivity/specificity, etc?

Is it neither??

Maybe you're really trying to:

- Find groups of things
- Identify patterns of behavior

} more like clustering.

- Rank items → ranking.
- Recommend advertisements/friends/movies to users

⋮

Constraints on my classifier

Are there practical constraints on my classifier?

- Is my training time severely limited? (Pretty rare)
- Is my evaluation/prediction time severely limited? (Very common)
  - Maybe I can only afford to evaluate linear classifications, and random forests are too expensive
- Does my model need to be interpretable?

Note: even if my final classifier must be simple (e.g., linear), I can still learn a lot from more flexible models.

Start learning

Now you're set up for your learning task. What are the first things to think about?

- Validation! You want to plan this workflow from the start, before you lie to yourself
- Features! Extract reasonable first set of features to use for prediction (an art, lots of work for speech, images, text, graphs, time series, ...)
- What to use? Usually start with something simple and interpretable, and something flexible. This should be quick and dirty. Iterating is often much more illuminating.

Validation

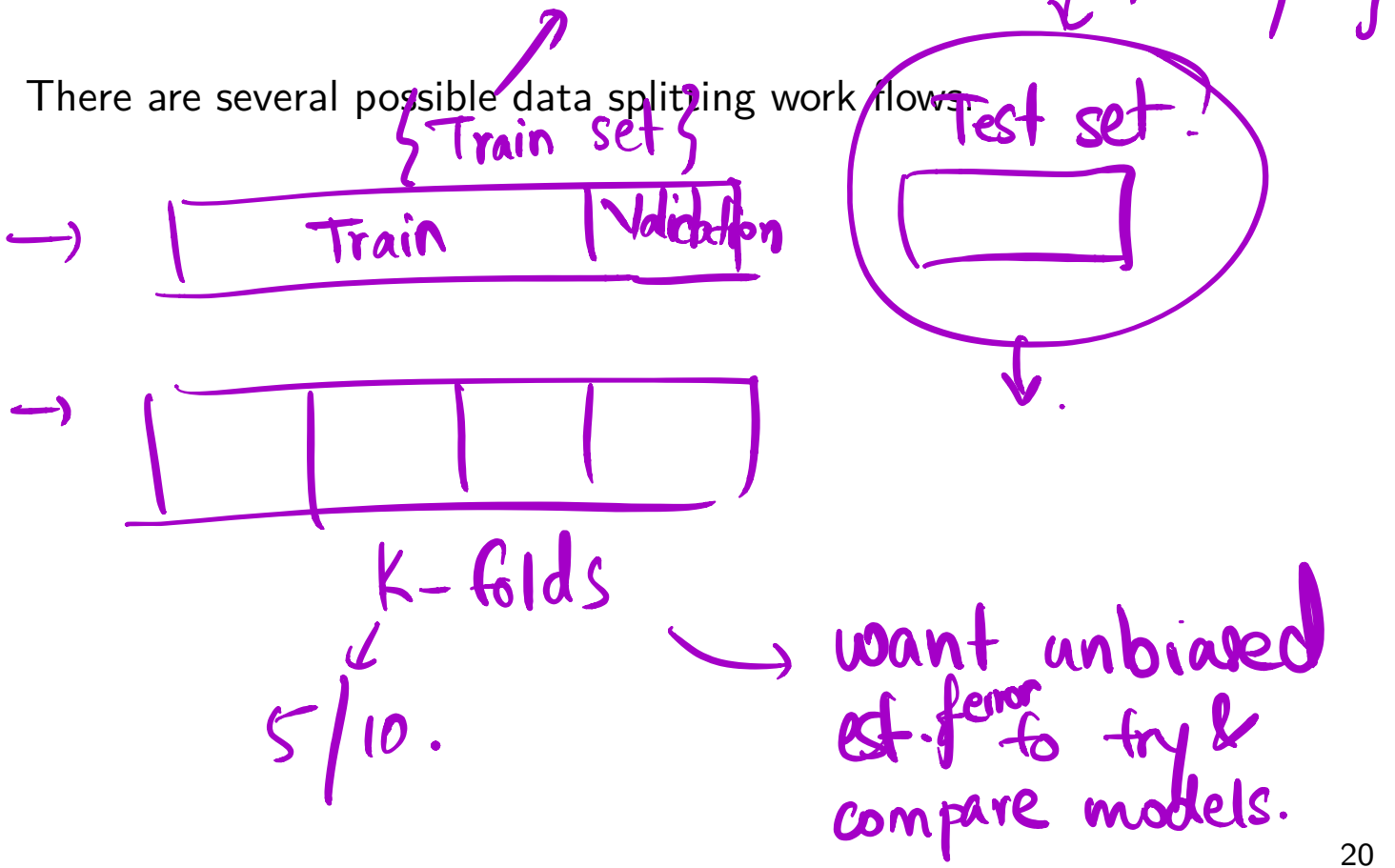
If we magically knew exactly the right model and how well it worked, we could just fit our model on all n points.

In reality, we need "extra" data to:

- Determine the appropriate model to fit
- Estimate the chosen model's performance on new data

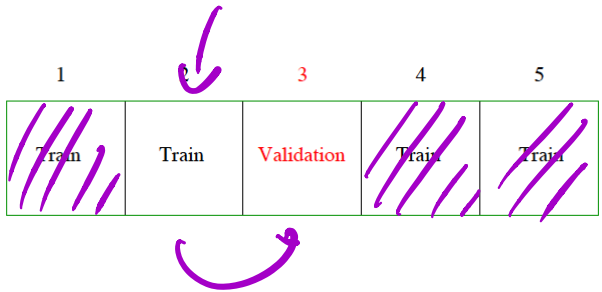
Work flows

There are several possible data splitting work flows:



Cross-validation

One approach to avoid splitting too much is cross-validation. This is an efficient way to select our model without wasting too much data.



However, there is still some concern of overfitting on the cross-validation set. This is especially an issue if we experiment with many models.

→ Validation error unbiased for a single model.

→ Not unbiased for "best model".

Validation plan

A realistic approach:

1. Split the data into a training set and a testing set.
2. You can further split the training set into a train set and a validation set (this is easy to understand conceptually but rarely used in practice). More common: use cross-validation or out-of-bag errors only on this set to pick your model.
3. Fit your chosen model on the whole training set.
4. At the very end, use your testing set to get an accurate picture of how well you actually do with that model.

You never touch the testing set during your model experimentation and selection.

Why do we need the test set?

Some warnings about cross-validation

Cross-validation seems like a magical solution to an impossible problem. There are a few things to beware of:

1. Can have a high computational cost, especially as the number of parameters/models grows

2. When the number of models is very large, it can give misleading results. It is not immune to the usual winner's curse problem.

→ validation error is downward biased for "best model".

3. The folds need to include the entire estimation pipeline. A frequent mistake is to carry out part of the model selection before breaking the data into folds, which invalidates the result.

Warning: high computational cost

Suppose it costs c to fit an estimator once.

To fit it across a grid of 100 λ values, it would cost $100c$.

If we do 10-fold cross-validation across this grid, it would cost $1000c$.

If we do LOOCV, it would cost $100n \cdot c$, which could potentially be tremendously large

If we decide to tune two parameters with 10-fold cross-validation (each with a 100-value grid), it would cost $100000c \ldots$

We want to be somewhat judicious in what we choose to tune with CV (especially given the next slide). There is also a literature in statistics and computer science which works on tricks to avoid these high costs.

Warning: high number of models

If we search a very large number of models, one of them may look better completely by chance.

Imagine we are in a binary classification problem ($y_i \in \{0, 1\}$). If each model were terrible and just flipped coins to make the test set prediction, eventually we would try one that looked quite good. In particular, we would eventually find one that performed better in cross-validation error than the true best model. This is the winner's curse.

This is just something to be aware of when the set of possible models is expanded dramatically. It is less of a risk in more constrained settings

Warning: excluding part of the pipeline

This mistake happens a lot, and looks slightly different each time.

Remember: For cross validation (or test sets) to work, every data-driven decision about the model has to exclude the held-out fold (or the test set).

Remove validation before feature selection.

Bad CV example: Suppose that we observe y and $x_1, \ldots, x_p \in \mathbb{R}^n$, with $p \gg n$. We would like to test our magical prediction algorithm $\hat{f}_\lambda$. Because p is so large, we first filter for x_i which have some reasonable correlation with y, building $\tilde{X}$ with columns $\{x_i : |\mathrm{cor}(x_i, y)| > \delta\}$.

Now we estimate $\hat{f}_\lambda$ on $\tilde{X}, y$ and use cross validation to tune λ and estimate the error.

What goes wrong??

Feature generation

Again several over-generalizations:

- The general lesson that has been learned through many applications and contests is that the data and features is often more important than the exact algorithm that you apply.
- Tweaking the algorithm gives small gains. Developing new features to feed into your algorithms gives big gains.
- This is where the power of forests, boosting, and deep learning come in. They have an ability to use features more flexibly.

Feature generation

It is difficult to give much specific advice about feature generation, since it is so problem-dependent.

A few comments:

- With modern algorithms (e.g., Lasso, random forests), having extra features does not hurt much. You can afford to generate many useless features in pursuit of a few really good ones.

  This means you can consider multiple transformations or versions of the same information.

- Make sure only to use information you would have in new samples! For example: you can't use signs that a company has gone out of business when predicting its viability. (This happens . .)

You're ready to *learn*!

You've seen many models, and hopefully obtained an intuition for how they behave. You can think about which models are good for your situation.

- Linear/logistic regression
- Ridge regression/logistic regression
- Lasso or logistic lasso
- Trees
- Random forests
- Boosted trees
- SVM
- LDA

You should consider – which of these are flexible, non-linear, interpretable, good at dealing with irrelevant features.

First models

The quick and dirty approach:

- Something simple and interpretable. Often Lasso (prediction) or Logistic Lasso (classification).
- Something flexible and more powerful. Often random forests.

Differences between the two models can hint at what you might be ignoring with the simple model. You can explore the difference and try to see why.

Variable importance and partial dependence plots from the random forest can suggest important variables, new transformations, and important interactions.

Now what?

You have your data. You have some features. You've fit a couple models and have estimates of their error.

Now what? How do you improve your error?

(Also, what error should you actually care about?)

One approach: Think about Bias-Variance Tradeoff! Remember, all the error you can correct is either in your bias or your variance.

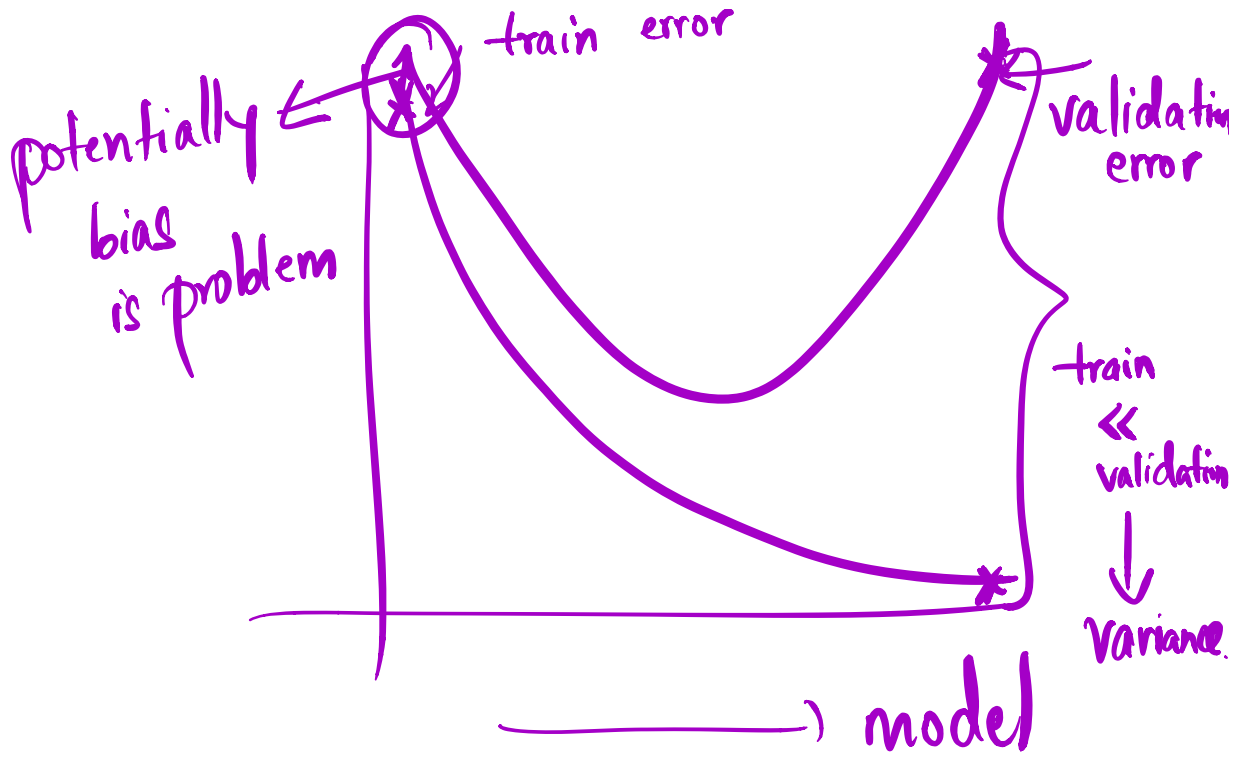
Is your current model too biased or too variable?

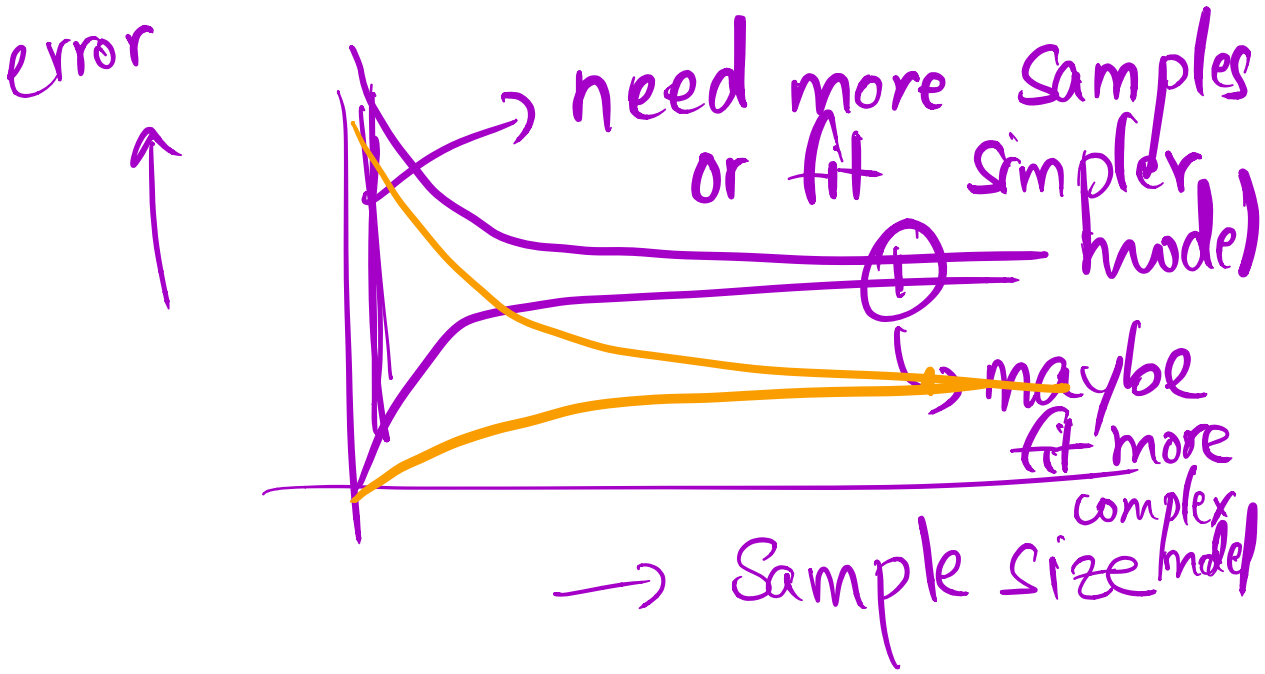
One way to get a hint: look at the difference between training and test error.

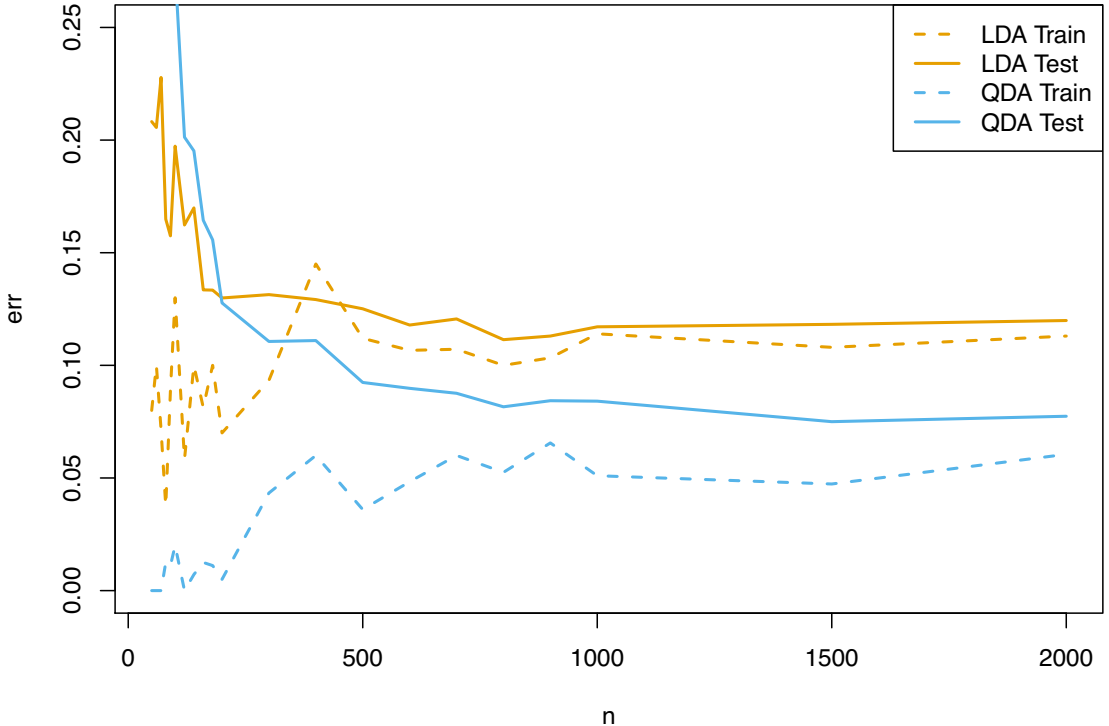
What does it mean if:

- The training error is much smaller than the test (CV) error?
- The training error is about the same as the test (CV) error?

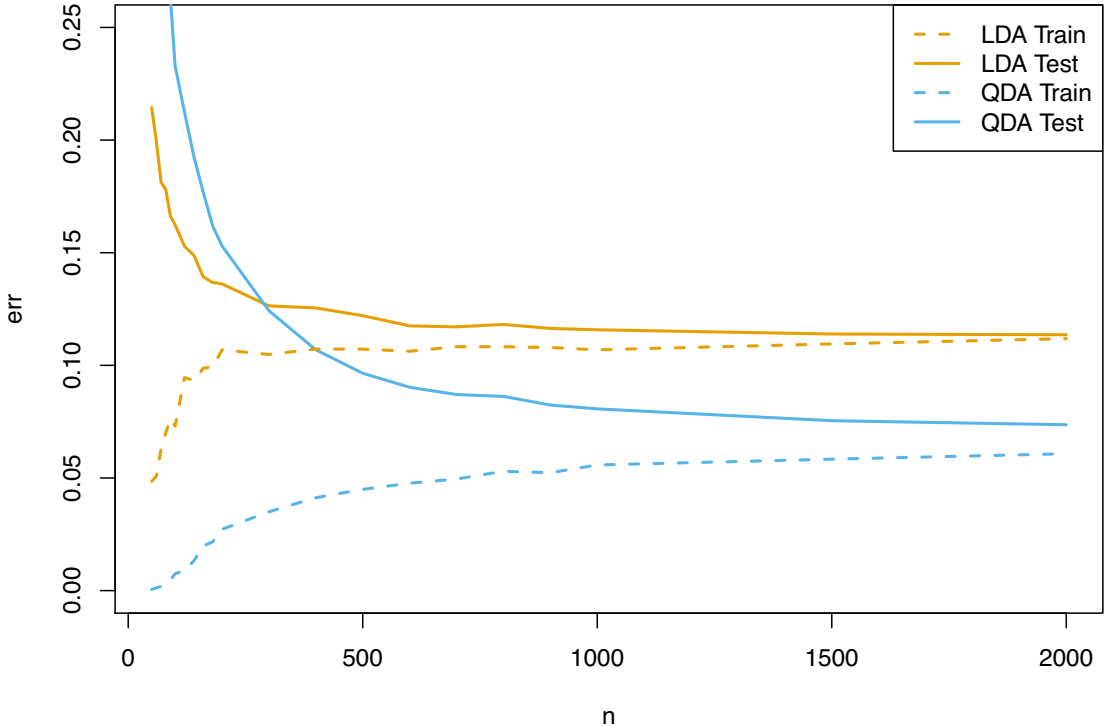
↳ bias problem


train error
potentially
bias
is problem
validation
error
train
≪
validation
↓
variance
model


error
need more samples
or fit simpler
model
maybe
fit more
complex
model
Sample size



LDA with fixed p and varying n. True model fits QDA assumptions, but it can be too high variance.



LDA with fixed p and varying n. True model fits QDA assumptions, but it can be too high variance.

What can you do to change your model? What do these do to the bias and variance?

- Get more data:
- Make more/better features:
- Use a more flexible model:
- Use a more regularized model:

Tweaking features

You can look at the variables that are currently important to give some hints of better variables.

If a variable is important, how can you make it more useful? What other variables might it suggest are useful?

If it is not important but you thought it should be, what is going wrong?

Variable importance plots can hint at useful variables when included with transformations or interactions. Partial dependence plots can reveal better transformations and even interactions.

What errors do you care about?

In regression problems: Is least squares error reasonable for your problem?

In classification: Is misclassification error appropriate? Do you care about false positives and false negatives equally? How should you trade them off?

Correcting for badly imbalanced samples

When one class is much smaller than the other, most of our classifiers don't do very well. There are a few approaches to fixing this:
Suppose that you have 90% class 1, and 10% class 2.

- Downsampling: Sample one (or a few) elements of class 1 for each element of class 2 to make it more balanced
- Upsampling: Duplicate elements of class 2 to make it more balanced.
- Artificially change your prior weights, class weights, or case weights. This depends a bit on which method you are using.

Always keep an eye on your base rate: the fraction of each class in your data. That keeps you aware of this problem, and lets you know what good performance really is: 10% misclassification in the problem above is not very impressive...

Siva's Summary

1. Train-validation-test splits
2. Featurization
3. Quick and dirty (fit a couple of models). Understand base rates (fit naive predictors), understand what error metric you should be using, know that you can trade-off things like precision and recall in many predictors.
4. Diagnose bias/variance problems (use sample-size curves, model-complexity curves, regularization curves, compare different models)
5. Fix bias/variance problems (different set of fixes in each case). Iterate 2,4,5. Think carefully about how to not get bogged down by the tyranny of tuning parameters (use smaller data sets, be parsimonious in choices to try out).
6. Error analysis (diagnose points on which you are predicting poorly, are they outliers? Can you design useful features for them?).
7. Maybe you need more training data but this should be last resort.