Model Evaluation, Selection and Ensembles

Machine Learning

Daniele Loiacono

POLITECNICO MILANO 1863

Outline and References



- Outline
  - Bias-Variance Tradeoff (PRML 3.2, ESL 7.3)
  - Model Assessment in Practice (ISL 5.1 and 6.1.3)
  - Feature Selection (ESL 3.3)
  - Dimensionality Reduction (PRML 12.1)
  - Bagging and Boosting (PRML 14.2, 14.3)
- References
  - This slides are based on material of prof. Marcello Restelli
  - Pattern Recognition and Machine Learning, Bishop [PRML]
  - Elements of Statistical Learning, Hastie et al. [ESL]
  - Introduction to Statistical Learning, James et al. [ISL]

Bias-Variance

How to evaluate a model?

- Given a dataset $\mathcal{D} = \{\mathbf{x}_i, t_i\}$ and a model $\hat{t}_i = y(\mathbf{x}_i)$
- How do we choose a model y? / if we have more models, how do we choose?
- Is the loss function L computed on $\mathcal{D}$ a good way to evaluate a model? = should we always choose the model with the lowest loss function on D?
  - For example, for regression could we use $RSS_y = \sum_{\{\mathbf{x}_i,t_i\}\in\mathcal{D}}(t_i - y(\mathbf{x}_i))^2$

No! The loss function may be biased towards training data (we're reducing it on training data). We would like to have a loss function computed on all kind of data that we may have ⟹ $\mathbb{E}[L]$

- No! This is the performance measure we want to compute:

$$\mathbb{E}[L] = \int\int L(t, y(\mathbf{x}))p(\mathbf{x}, t)\mathrm{d}\mathbf{x}\mathrm{d}t$$

$p(\mathbf{x}, t)$ = distribution of data

  - E.g., when using a squared loss function for regression:

$$\mathbb{E}[L] = \int\int (t - y(\mathbf{x}))^2 p(\mathbf{x}, t)\mathrm{d}\mathbf{x}\mathrm{d}t$$

We usually don't know $p(\mathbf{x}, t)$!
So what should we do?

In the most ideal cases we approximate it (but still, we don't have it)

let's analyze $\mathbb{E}[L]$

Bias-Variance Decomposition

(Let's first analyze the case of **regression** problem (in the case of classification is the same))

- The Bias-Variance is a framework to analyze the performance of models
- Definitions and assumptions
  - Data: $t_i = f(\mathbf{x}_i) + \varepsilon$, with $\mathbb{E}[\varepsilon] = 0$ and $Var[\varepsilon] = \sigma^2$: target = function of the input + stochastic noise (noise with 0 mean and σ^2 as variance)
  - Model: $\hat{t}_i = y(\mathbf{x}_i)$ learned from a sampled dataset $\mathcal{D} = \{\mathbf{x}_i, t_i\}$
  - Performance: $\mathbb{E}\left[(t - y(\mathbf{x}))^2\right]$ (expected square error)

we're not saying that the noise is gaussian

expected error that we've making on a new datapoint $\mathbf{x}$. This expectation is because we have a stochastic component AND this expectation is based on the training dataset $\mathcal{D}$ ($\mathbb{E}[\mathbb{E}_{\mathcal{D}}[..]]$) It's basically a double expectation

- Thus we can decompose the **expectected square error** as:

$$\begin{aligned}
\mathbb{E}[(t - y(\mathbf{x}))^2] &= \mathbb{E}[t^2 + y(\mathbf{x})^2 - 2ty(\mathbf{x})] \\
&= \mathbb{E}[t^2] + \mathbb{E}[y(\mathbf{x})^2] - \mathbb{E}[2ty(\mathbf{x})] \\
&= \mathbb{E}[t^2] \pm \mathbb{E}[t]^2 + \mathbb{E}[y(\mathbf{x})^2] \pm \mathbb{E}[y(\mathbf{x})]^2 - 2f(\mathbf{x})\mathbb{E}[y(\mathbf{x})] \\
&= Var[t] + \mathbb{E}[t]^2 + Var[y(\mathbf{x})] + \mathbb{E}[y(\mathbf{x})]^2 - 2f(\mathbf{x})\mathbb{E}[y(\mathbf{x})] \\
&= Var[t] + Var[y(\mathbf{x})] + (f(\mathbf{x}) - \mathbb{E}[y(\mathbf{x})])^2 \\
&= \underbrace{Var[t]}_{\sigma^2} + \underbrace{Var[y(\mathbf{x})]}_{\text{Variance}} + \underbrace{\mathbb{E}[f(\mathbf{x}) - y(\mathbf{x})]^2}_{\text{Bias}^2}
\end{aligned}$$

IRREDUCIBLE

this depends on the problem (it's the variance of the noise in the data)

Bias-Variance Decomposition (2)

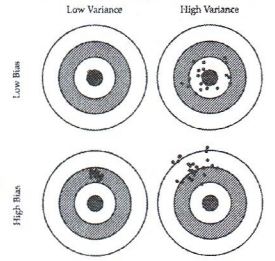
- **Model Variance**
  - If we sample several datasets $\mathcal{D}$ we will learn different model $y(\mathbf{x})$
  - variance measures the difference between each model learned from a particular dataset and what we expect to learn:

$$\text{variance} = \int \mathbb{E}\left[(y(\mathbf{x}) - \overline{y}(\mathbf{x}))^2\right] p(\mathbf{x})\mathrm{d}\mathbf{x}$$

$$\overline{y}(\mathbf{x}) = \mathbb{E}[y(\mathbf{x})]$$

  - Decreases with **simpler models**
  - Decreases with **more samples**

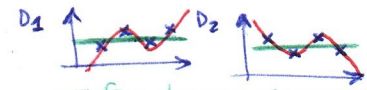
Credit: http://scott.fortmann-roe.com/docs/BiasVariance.html



This is a measure of: if we can learn the model on N different datasets, how different will they be? (we learn N models, the structure is the same, the training dataset changes) If the N models will be similar ⇒ LOW VARIANCE.

The variance is how spread the expected error will be w.r.t. all the possible datasets that we can use to learn the model.

How the variance may depend on the features space that we're deciding to use? Fewer degrees of freedom lead to lowest var:



— few degrees of freedom
— a lot of degrees of freedom

The two with low dof are the same (low variance).

Note: lower dof always lead to lower variance! if we model "quadratic data" with one dof (a line) the variance will be lower than if we model them with 2 dof, however with 1 dof the GENERAL ERROR will be higher because of **BIAS**.

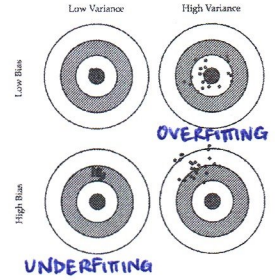
Bias-Variance Decomposition (3)

- **Model Bias**
  - bias measures the difference between **truth** (f) and what we expect to learn ($\mathbb{E}[y(\mathbf{x})]$):

$$\text{bias}^2 = \int (f(\mathbf{x}) - \overline{y}(\mathbf{x}))^2 p(\mathbf{x})\mathrm{d}\mathbf{x}$$

  - Decreases with more complex models
- **Data noise** (σ^2) is the variance of data and does not depend neither from data sampling nor from the model complexity

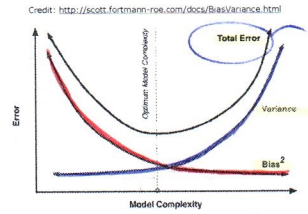
Credit: http://scott.fortmann-roe.com/docs/BiasVariance.html



We want to find the right trade-off. Notice that: the trade-off also depends (strongly) on how many data we have!

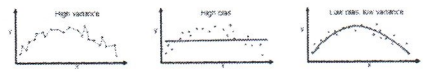
Bias-Variance Decomposition (4)

Credit: http://scott.fortmann-roe.com/docs/BiasVariance.html



final "true" error for unseen data

Credit: https://towardsdatascience.com/understanding-the-bias-variance-tradeoff-165e6942b229



overfitting | underfitting | Good balance

Note: the loss function (and so the optimization of the model on the dataset (single dataset)) consider the minimization of the BIAS, **not** both bias and variance. The variance term is taken into account when we choose the complexity of the model. Once we decide the complexity (and so, the model) we try to reduce bias as much as possible (by fitting).

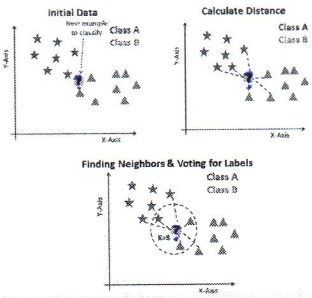
A case study: Bias-Variance for K-NN

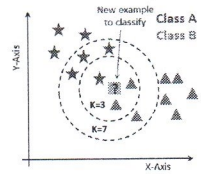
(K-nearest neighboors)

(Non-parametric approach)

- Which is the best value for K?

by changing K we change the results





Credit: https://www.datacamp.com/community/tutorials/k-nearest-neighbor-classification-scikit-learn

A case study: Bias-Variance for K-NN (2)

- Bias-Variance analysis allows to understand how K affects the performance:

$$\mathbb{E}[(t^* - y(\mathbf{x}^*))^2] = \sigma^2 + \frac{\sigma^2}{K} + \left(f(\mathbf{x}^*) - \frac{1}{K}\sum_{i=1}^{K} f(\mathbf{x}_i)\right)^2$$

there is no close form solution for the error on new data, except for the KNN method

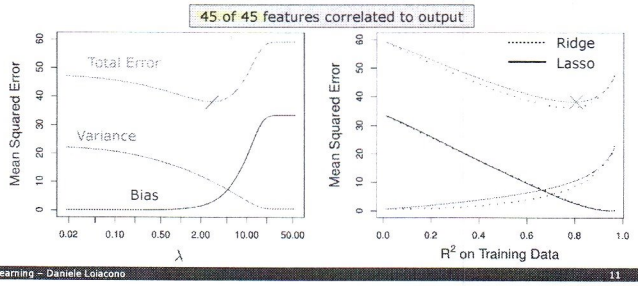
  - The data noise σ^2 is the **irreducible error**
  - The model **variance** $\frac{\sigma^2}{K}$ decreases as K increases
  - The model **bias** (the last *squared* term) depends on **smoothness** of problem space, but in general increases as K increases (because it increases the distance of the neighbours used to compute y)

Regularization is a way to deal with the bias-variance trade-off. This is because by adding the λ term, we're constraining the values of the parameters and so we're decreasing the size of the hypothesis space. The larger is the regulariz. coefficient (λ), the stronger is the constraint and the smaller is the hypothesis space and the less complex the model CAN be → smaller var. The ideal λ provides the best trade-off between var. and bias.

Regularization and Bias-Variance

- The Bias-Variance decomposition explains why regularization allows to improve the error on unseen data
- Lasso outperforms Ridge regression when few features are related to the output

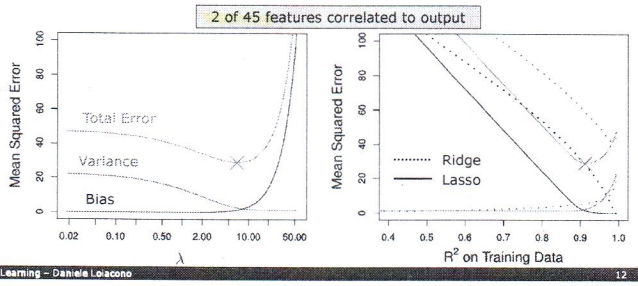
45 of 45 features correlated to output



Regularization and Bias-Variance

- The Bias-Variance decomposition explains why regularization allows to improve the error on unseen data
- Lasso outperforms Ridge regression when few features are related to the output

2 of 45 features correlated to output



Model Assessment in Practice

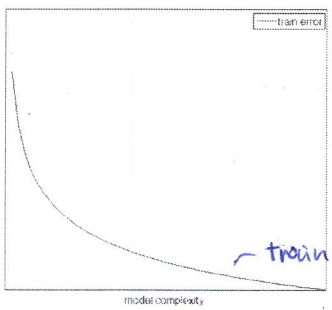
Training Error

- Given $\mathcal{D} = \{\mathbf{x}_i, t_i\}$ with $i = 1, \ldots, N$
- We can select a model based on the loss function L computed on $\mathcal{D}$:
  - Regression

$$L_{train} = \frac{1}{N}\sum_{n=1}^{N}(t_n - y(\mathbf{x}_n))^2$$

  - Classification

$$L_{train} = \frac{1}{N}\sum_{n=1}^{N}(I(t_n \neq y(\mathbf{x}_n)))$$



train error

Prediction Error

- We already saw that **training error** does not provide a good estimate of the error made on new data, the **prediction error**
  - Regression

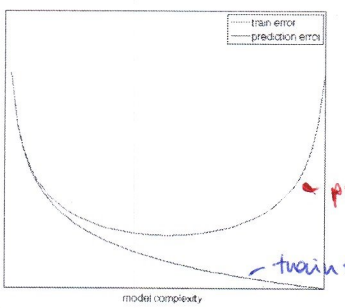
$$L_{true} = \int\int (t - y(\mathbf{x}))^2 p(\mathbf{x}, t)\mathrm{d}\mathbf{x}\mathrm{d}t$$

  - Classification

$$L_{true} = \int\int I(t \neq y(\mathbf{x}))\, p(\mathbf{x}, t)\mathrm{d}\mathbf{x}\mathrm{d}t$$

- Unfortunately, we often are not able to model $p(\mathbf{x}, t)$

→ and so we're not able to determine the prediction error



prediction error ("true" error)

train error

we know it should be like this, however we cannot evaluate it (because we don't know $p(\mathbf{x}, t)$)

So what to do in practice? Test error

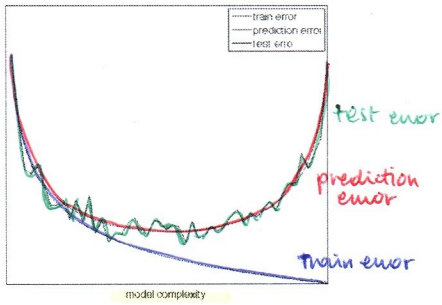
- What can we do in practice?
  - Split randomly data into a **training set** and **test set**
  - Optimize model parameters using the **training set**
  - Estimate the prediction error using the **test set**
    - Regression

$$L_{test} = \frac{1}{N_{test}} \sum_{n=1}^{N_{test}} (t_n - y(\mathbf{x}_n))^2$$

    - Classification

$$L_{test} = \frac{1}{N_{test}} \sum_{n=1}^{N_{test}} (I(t_n \neq y(\mathbf{x}_n)))$$

How to split? It's a trade-off. The more data we use for the test the better we approximate the "true" error, however we have less data to train the model



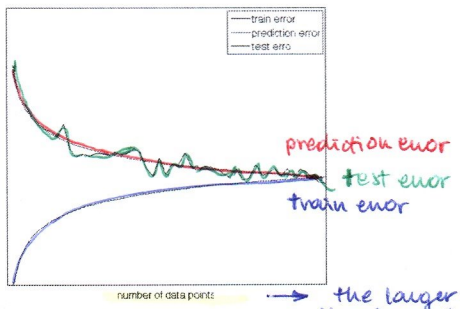
So what to do in practice? Test error

- What can we do in practice?
  - Split randomly data into a **training set** and **test set**
  - Optimize model parameters using the **training set**
  - Estimate the prediction error using the **test set**
    - Regression

$$L_{test} = \frac{1}{N_{test}} \sum_{n=1}^{N_{test}} (t_n - y(\mathbf{x}_n))^2$$

    - Classification

$$L_{test} = \frac{1}{N_{test}} \sum_{n=1}^{N_{test}} (I(t_n \neq y(\mathbf{x}_n)))$$



the larger is the number of data the closer the errors become! By increasing the number of data we decrease the variance! At some point the variance will be so small that the **true** error will not be affected anymore by the variance of the model

Analysis of train-test error

- The analysis of train-test errors allows to identify possible problems
  - **high bias**: training error is close to test error but they are both higher than expected
  - **high variance**: training error is smaller than expected and it slowly approaches the test error

this however is just a reference line (not something that we can set strictly)

looking at the relation between test error and train error we can guess in which area we are in the bias-variance tradeoff

1. UNDERFITTING — Tells us that even if we keep going the performance will not get any better. Probably the model is too simple ⟹ HIGH BIAS

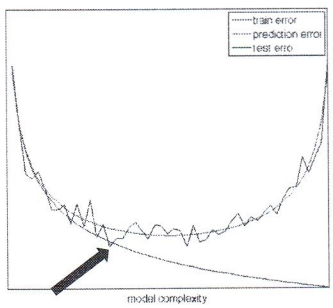
2. Tells us that there is a large variance term that we're not able to capture (because of the gap between the test set and training set) ⟹ HIGH VARIANCE — OVERFITTING

Pitfalls of using test set

the test set and the test error is something that MUST have **NOTHING** to do with the whole process of creating the model !

- Often data is limited and test error is usually small → prediction error can be either **overestimated** or **underestimated**
- Test error cannot be used for model selection → we can **overfit** test set
- Only if test set is **never used** for training and model selection, it can provide an **unbiased** estimate of prediction error
- So how to perform model selection?

using an additional set



Validation Set

- How to choose the best model and how do we set hyperparameters (e.g, α, λ)?
- We split data in three parts:
  - Training Data
  - Validation Data
  - Test Data
- How do we proceed?
  - We use **training data** to learn model parameters
  - For each model learned (i.e., different features and hyperparameters) we use **validation data** to compute the **validation error**
  - We select the model with the lowest **validation error** and finally use **test data** to estimate **prediction error**
- But...
  - to be reliable, validation data should be enough → less training data
  - we might **overfit validation data** and eventually not choose the best model

untouched until the very end

Leave-One-Out Cross Validation (LOO)

only the training data, the test data is already put away

- For each sample $\{\mathbf{x}_i, t_i\} \in \mathcal{D}$
  - We train the model on $\mathcal{D}\backslash\{\mathbf{x}_i, t_i\}$
  - We compute the error of the resulting model on $\{\mathbf{x}_i, t_i\}$
- The estimate of the prediction error of our model will be the average of all the error computed using a single sample:

$$L_{LOO} = \frac{1}{N}\sum_{i=1}^{N}(t_i - y_{\mathcal{D}_i}(\mathbf{x}_i))^2$$

  - Where $y_{\mathcal{D}_i}$ is the model trained on $\mathcal{D}\backslash\{\mathbf{x}_i, t_i\}$
- L_{LOO} provides an **almost unbiased** estimate of prediction error (**slighlty pessimistic**)
- Unfortunately, LOO is **extremely expensive** to compute
  - As an example, even if training take just 1 second, computing LOO on 100K samples, would require 100K seconds (more than one day)!

that's why we introduce:

WHY "PESSIMISTIC"?
* The error is evaluated on a model trained on k-1/k of the data. If we use the whole data (once we fix hyperparameters after this) the error may be reduced.

K-Fold Cross Validation

- We randomly split the training data $\mathcal{D}$ into k folds: $\mathcal{D}_1, \ldots, \mathcal{D}_k$
- For each fold $\mathcal{D}_i$
  - We train the model on $\mathcal{D} - \{D_i\}$
  - We compute the error on D_i

$$L_{\mathcal{D}_i} = \frac{k}{N}\sum_{(\mathbf{x}_n, t_n)\in\mathcal{D}_i}(t_n - y_{\mathcal{D}\backslash\mathcal{D}_i}(\mathbf{x}_n))^2$$

- Finally, we estimate the prediction error as the average error computed:

$$L_{k-fold} = \frac{1}{k}\sum_{i=1}^{k} L_{\mathcal{D}_i}$$

- L_{k-fold} provides a **slightly biased** estimate of prediction error (**pessimistic**) but it is much cheaper to compute (usually k=10 is used)

*

After the whole procedure we end up with k different models. What do we do? We could for example choose one of them or combine them together to obtain an ensamble. However, a good and common approach is to use this procedure only to fix the hyperparameters. Then we can use the training + validation to train the decided model.

Complexity-Adjusted Model Evaluation

- Other metrics are used to evaluate models adjusting their training error based on their complexity:
  - Mallows's C_p: $C_p = \frac{1}{N}(RSS + 2M\hat{\sigma}^2)$
  - Akaike Information Criteria: $AIC = -2lnL + 2M$
  - Bayesian Information Criteria: $BIC = -2lnL + \mathrm{M}\ln(N)$
  - Adjusted R^2: $AdjustedR^2 = 1 - \frac{RSS/(N-M-1)}{TSS/(N-1)}$

Tipically for REGRESSION

Tipically for CLASSIFICATION

we adjust the training error s.t. it takes into account also the variance (= complexity).

M: number of parameters; N: number of samples; L: loss function;
$\hat{\sigma}^2$: estimate of noise variance;
RSS: residual sum of squares; TSS: total sum of squares

- AIC and BIC are generally used when maximizing the log-likelihood
- BIC generally penalize more than AIC model complexity

How do we choose the model?

Reducing the variance

- We want to select the model with the lowest prediction error
- This can be achieved by reducing the variance of the model:
  - Feature Selection: we should design the feature space by selecting the **most effective subset** of all the possible features
  - Dimensionality Reduction: the input space can be mapped to a **lower-dimensional space**
  - Regularization: the values of the parameters are **shrunked toward zero**
- These three approaches are not necessarily mutually exclusive and they can be used toghether

work on the features space can help with variance reduction

Reducing the variance: curse of dimensionality

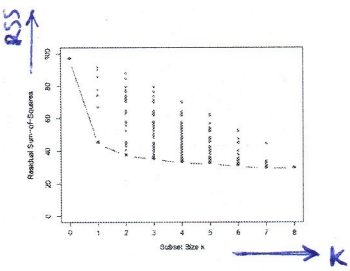
- Why dimensionality reduction and feature selection does work?
- **Curse of dimensionality**: a linear increase of the number of features would result in an exponential increase of the feature space volume
  - higher dimensional feature space means a model with more parametrs (higher complexity)
  - the number of samples necessary to train a model grows exponentially with respect to the number of features
- A common pitfall is to think that using more features is always better:
  - Is $y = w_0 + w_1 x + w_2 x^2$ always better than $y = w_0 + w_1 x$, isn't it?
  - In fact, we can always set $w_2 = 0$
  - No! Increasing the number of features, it increses the probability of overfitting the data (also because I could not have enough data for a larget feature space)!

This won't actually happen. The model during training will exploit as much as possible all the degrees of freedom (result: overfitting)

Feature Selection

Best Subset Selection

- The simplest idea seems to compare all the possible combinations of the features
- Assuming we have M features, for each $k = 1, \dots, M$:
  - Train all the $\binom{M}{k} = \frac{M!}{k!(M-k)!}$ models with exactly k features
  - Chose the model with the smallest training error
- How do we choose k?
  - AIC, BIC, AdjusterR^2, etc.
  - Cross-Validation



"exhaustive search" problem: we may have M very large and so it may be computationally not feasible

Feature Selection in Practice

- Best Subset Selection is computationally unfeasible when M is large, thus in practice Filter, Embedded or Wrapper approach is used for feature selection
- **Filter**: features are ranked **independently** based on some evaluation metrics (e.g., correlation, variance, information gain, etc.) and the top k are selected
  - Very fast but fails to capture any subsest of **mutually dependent features**
- **Embedded**: feature selection is performed as a step of the machine learning approach used (e.g., lasso, decision trees, etc.)
  - Not expensive but the features identified are **specific** to the learning approach
- **Wrapper**: a search algorithm is used to find a subset of features that are evaluated by training a model with them and assessing its performance
  - Either a simpler model or a simple machine learning approach can be used to evaluate the features
  - Greedy algorithms are generally used to search the best subset of features

based on how much features can bring (independently, alone) useful informations w.r.t. the target

basically we still try to do the best subset selection (↑) but we apply a search algorithm to reduce the combinations that we want to try (we want to avoid the exhaustive search)

of the feature and the target

if a feature is constant then it's not so useful

Wrapper Feature Selection: Search Algorithms

(greedy algorithms)

- **Backward Stepwise Selection**

1. Let M_k be the **full** model, which contains all the M features
2. For $k = M, \dots, 1$
   i. Train all the k models that contain all but one of the features in M_k
   ii. Choose as M_{k+1} the model among the ones trained in (i) with the lowest training error
3. Select the model M_k using either cross validation, AIC, BIC, or C_p

- **Forward Stepwise Selection**

1. Let M_0 be the **null** model, which contains no features
2. For $k = 0, \dots, M-1$
   i. Train all the $M-k$ models that extend M_k with an additional features
   ii. Choose as M_{k+1} the model among the ones trained in (i) with the lowest training error
3. Select the model M_k using either cross validation, AIC, BIC, or C_p

we start from the full model. At each step we try to remove one feature (the one that impact the less on the perform.)

(opposite ↑). We start with no features and we add one at each step (we choose each time the one more relevant)

Dimensionality Reduction

Dimensionality Reduction

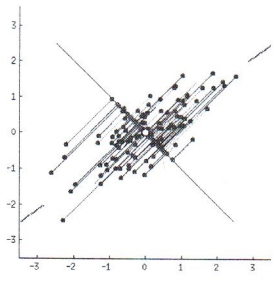
- Dimensionality reduction aims at reducing the dimensions of input space, but it differs from feature selection in two major respects:
  - it uses **all** the features and **maps** them into a lower-dimensionality space
  - it is an **unsupervised** approach
- There are many methods to perform dimensionality reduction:
  - **Principal Component Analysis (PCA)**
  - Indipendent Component Analysis (ICA)
  - Self-Organizing Maps
  - Autoencoders
  - ISOMAP
  - t-SNE
  - ...

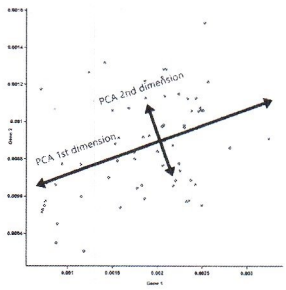
feature selection is target oriented: we look for the features that better explain the target. Instead, the dimensionality reduction aims to maintain all the input informations just in a lower space. This operation is guided by the "preserve informations", it's **not** target-driven

Benefit: we use all informations
Drawback: interpretation (usually the original features have a meaning)

PCA underlying idea

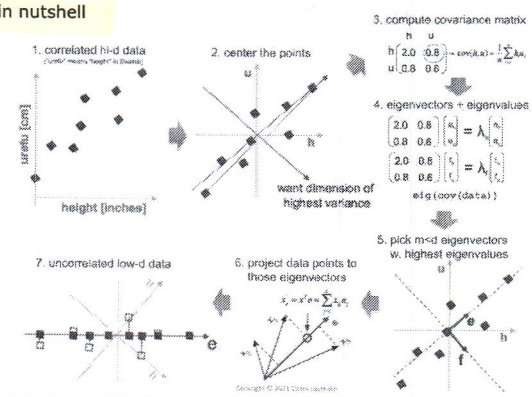
: change the basis of the input space into an orthonormal basis that captures (for each direction) the max. variance





(a reasonable step 0. be the normalization)

PCA in nutshell



1. correlated hi-d data
2. centers the points
3. compute the covariance matrix
4. eigenvectors + eigenvalues
5. pick $m < d$ eigenvectors with highest eigenvalues
6. project data points to those eigenvectors
7. uncorrelated low-d data

the highest the eigenvalue the more variability is explained by the eigenvector

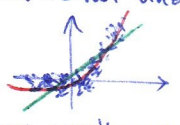
Steps and limitations of PCA

1. Center data: $\bar{\mathbf{x}}_i = \mathbf{x}_i - \bar{\mathbf{x}} = \mathbf{x}_i - \frac{1}{N}\sum_{i=1}^{N} \mathbf{x}_n$
2. Compute covariance matrix: $\mathbf{S} = \frac{1}{N-1}\sum_{i=1}^{N} \bar{\mathbf{x}}_i \bar{\mathbf{x}}_i^T$
3. Compute eigenvectors $\mathbf{e}_i$ and eigenvalues λ_i of S
4. The principal components (PCs) are the k eigenvector $\mathbf{e}_i$ $(i = 1, \dots, k)$ corresponding to the largest k eigenvalues λ_i of S
5. Project data onto the PCs: $\mathbf{X}' = \mathbf{X}\mathbf{E}_k$ where $\mathbf{E}_k = (\mathbf{e}_1, \dots, \mathbf{e}_k)$

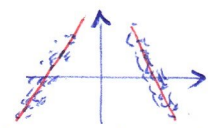
- The amount of variance in data explained by $\mathbf{X}'$ is $\sum_{i=1}^{k} \lambda_i \,/\, \sum_{i=1}^{M} \lambda_i$
- The higher the variance explained the lower will be the reconstruction error of original data from $\mathbf{X}'$
- PCs are **linear** combination of features (unlike other DR methods as ISOMAP)
- Fails when data is distributed in **multiple clusters** : this is related to the centering :
- It is not always the best data representations for regression/classification

* It may happen that there is a feature that explains a little amount of variance but it's VITAL for the explaination of the target

we won't capture anything that it's not 'linear'!

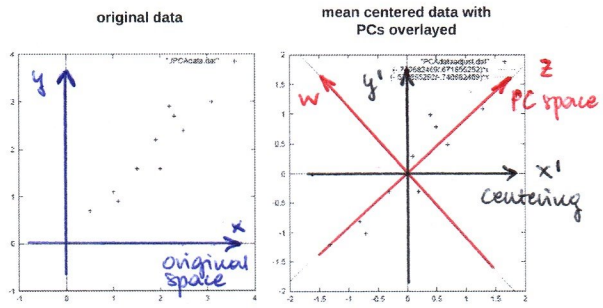


we won't capture the red line, we'll probably end up with the green one

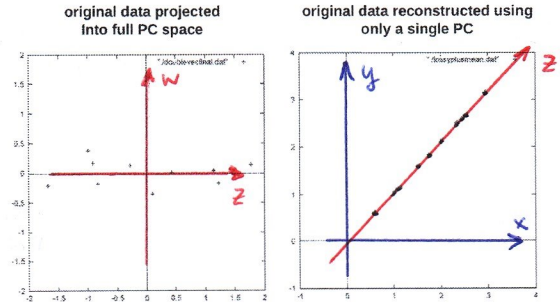


here we will **not** be able to capture the two red lines

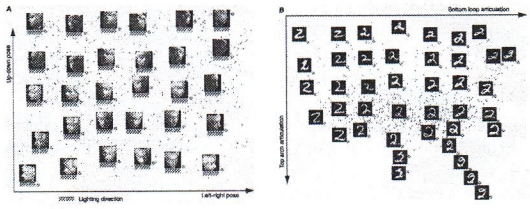
PCA: example



PCA: example



PCA: an example with complex data



[A global geometric framework for nonlinear dimensionality reduction. Tenenbaum, de Silva, and Langford. Science, 290(5500):2319–2323, 2000.]

Bagging and Boosting

Improving the Bias-Variance Tradeoff

- So far we saw how to reduce the variance, searching the best tradeoff with the increased bias...
- ... but it is possible to reduce the variance **without increasing bias**?
- Or is it possible to **reduce the bias**?
- We can achieve these results by using two **ensemble methods** that consist of learning **several models** and **combine** them:
  - Bagging
  - Boosting

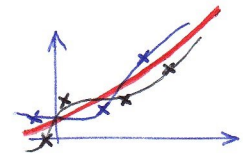
L = average of different models (in order to reduce variance (without sacrifying bias))

Which is the idea behind bagging?

- Let assume to have N datasets and to learn from them N models, $y_1, y_2, \ldots, y_N$
- Now let us compute an aggregate model as $y_{AGG} = \frac{1}{N}\sum_{i=1}^{N} y_i$
- If the datasets are **independent**, the model variance of y_{AGG} will be 1/N of the model variance of the single model y_i
  - To have an intuition of this, let assume to have random variable $\bar{x} = \frac{1}{N}\sum_N x$

$$Var(\bar{x}) = \frac{1}{N^2}\sum_N Var(x) = \frac{1}{N}Var(x)$$

- However, we generally do not have N dasets!
- So, what can we do?



The red one is the true model. Even if the blue and black model overfit, if we average them we end up with a model more similar to the red one than the single two. (if N>2 it gets better)

Bagging

- Bagging stands for Bootstrap Aggregation:
  - Generate N datasets applying **random sampling with replacement**
  - Train a model (classification or regression model) from **each** dataset generated
  - To compute the **prediction** for new samples, apply all the trained models and combine the outputs with **majority voting** (classification) or **averaging** (regression)
- Bagging is generally helpful and reduce the variance, although the sampled datasets are **not independent**
  - It helps with **unstable learners**, i.e., learners that change significantly with even small changes in the dataset (**low bias and high variance**)
  - It helps when we have a lot of overfitting (**low bias and high variance**)
  - It does not help when learners is **robust**, i.e., not sensitive to change in data (**usually higher bias but lower variance**)

hence, if we obtain **N** datasets the variance will not be reduced by **1/N** (!)

What is Boosting?

- The goal of boosting is to achieve a **small bias** by using on simple (**weak**) learners
- At the same time, using simple learners, aims at keeping a **small variance**
- This is achieved by **sequentially** training **weak learners**:
  1. Give an **equal weight** to all the samples in the training set
  2. Train a weak learner on the **weighted training set**
  3. Compute the **error** of the trained model on the weighted training set
  4. **Increase the weights** of samples missclassfied by the model
  5. Repeat from 2 until some criteria is met
- The ensemble of models learned can be applied on new samples by computing the **weighted** prediction of each model (**more accurate models weight more**)
- What **weak learners**? Several options (e.g., decision stumps or trees)
- How to compute weights? Let see AdaBoost algorithm...

AdaBoost Algorithm

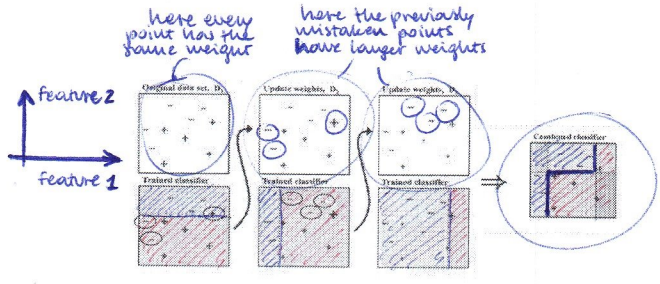
1. Initialize the observation weights $w_i = 1/N$, $i = 1, 2, \ldots, N$.
2. For $m = 1$ to M:
   (a) Fit a classifier $G_m(x)$ to the training data using weights w_i.
   (b) Compute
   $$\text{err}_m = \frac{\sum_{i=1}^{N} w_i I(y_i \neq G_m(x_i))}{\sum_{i=1}^{N} w_i}.$$
   (c) Compute $\alpha_m = \log((1 - \text{err}_m)/\text{err}_m)$.
   (d) Set $w_i \leftarrow w_i \cdot \exp[\alpha_m \cdot I(y_i \neq G_m(x_i))]$, $i = 1, 2, \ldots, N$.
3. Output $G(x) = \text{sign}\left[\sum_{m=1}^{M} \alpha_m G_m(x)\right]$.

this is used during the final aggregation

AdaBoost: Visual Explanation

here every point has the same weight

here the previously mistaken points have larger weights



combination of the models we learned

Bagging vs Boosting

Bagging

- Reduces variance
- Not good for stable learners
- Can be applied with noisy data
- Usually helps but the difference might be small
- Naturally parallel

Boosting

- Reduces bias (generally without overfitting)
- Works with stable learnes
- Might have problem with noisy data
- Not always helps but it can makes the difference
- Serial (cannot be parallelized)