Machine Learning

Ilya Narsky, Caltech
Lecture 2

Linear and Quadratic Discriminant Analysis

• Each class density is a multivariate Gaussian

\[ f_k(x) = \frac{1}{(2\pi)^{d/2} |\Sigma_k|^{1/2}} \exp\left[ -\frac{1}{2} (x - \mu_k)^T \Sigma_k^{-1} (x - \mu_k) \right] \]

• Look at the log-ratio of two class probabilities:

\[ \log \frac{f_0(x)}{f_1(x)} = C + x^T (\Sigma_0^{-1} \mu_0 - \Sigma_1^{-1} \mu_1) - \frac{1}{2} x^T (\Sigma_0^{-1} - \Sigma_1^{-1}) x \]

• Assume \( \Sigma_0 = \Sigma_1 \) => LDA

• Don’t assume \( \Sigma_0 = \Sigma_1 \) => QDA

Separation of a bivariate Gaussian from uniform background by quadratic discriminant analysis.
Estimating covariance matrix for LDA

\[ \{(x_n, y_n)\}_{n=1}^{N}, \quad x_n = \left\{x_n^{(d)}\right\}_{d=1}^{D}, \quad y_n = 0 \text{ or } 1 \]

- Compute covariance matrices within each class:
  \[
  W_k = \sum_{y=k} w_n, \quad \hat{\mu}_k = \frac{\sum_{y=k} w_n x_n}{W_k} 
  \]
  \[
  \hat{\Sigma}_k = \sum_{y=k} w_n (x_n - \hat{\mu}_k) (x_n - \hat{\mu}_k)^T / W_k, \quad k = 0, 1 
  \]

- Compute common covariance matrix:
  \[
  \hat{\Sigma} = \left( W_0 \hat{\Sigma}_0 + W_1 \hat{\Sigma}_1 \right) / (W_0 + W_1) 
  \]

- Simple enough? Ok, but sometimes people do
  \[
  \hat{\Sigma} = \frac{\left( \hat{\Sigma}_0 + \hat{\Sigma}_1 \right)}{2} 
  \]

- ...and sometimes they ignore event weights altogether.

Under LDA assumptions, multivariate normal and \( \Sigma_0 = \Sigma_1 \), all matrix estimates have the same expectations. But in reality they don’t.
Some wisdom on weighting matrix estimates

- Suppose you use LDA to separate signal from background. Suppose the background sample is a few times as large as the signal sample.
- The LDA covariance matrix estimate is driven mostly by background: $W_0 > W_1$.
- But in reality you want to find a region with high signal purity. If you increase $W_1$ for covariance matrix computation, you will likely improve LDA performance in the interesting region of high signal purity.
- Before you start playing the game of optimizing weights in the covariance matrix calculation, weigh accuracy vs sophistication.
Logistic Regression

\[ L(\beta | x) = \prod_{n=1}^{N} p_{n}^{y_{n}} (1 - p_{n})^{1-y_{n}} \]  
binomial likelihood

\[ p_{n} = P(Y = 1 | X = x_{n}) \]  
prob. of event observed at \( x_{n} \) being signal

\[ \log \frac{p}{1 - p} = \log \frac{P(Y = 1 | X = x)}{P(Y = 0 | X = x)} = \beta_{0} + \beta^{T} x \]  
logistic regression

Plugging LogitR into the likelihood formula, we obtain:

\[ \log L(\beta | x) = \sum_{n=1}^{N} \left[ y_{n}(\beta_{0} + \beta^{T} x_{n}) - \log(1 + \exp(\beta_{0} + \beta^{T} x_{n})) \right] \]

Solve \( \frac{\partial \log L(\beta | x)}{\partial \beta} = 0 \) to optimize \( \log L \)
LDA or Logistic Regression?

LDA: \[ \log \frac{f_0(x)}{f_1(x)} = C + x^T \Sigma^{-1}(\mu_1 - \mu_2) \]

LogitR: \[ \log \frac{p}{1 - p} = \beta_0 + \beta^T x \]

The two formulas describe the same model, don’t they?

Yes, but the linear coefficients are found using different techniques.

LDA assumes multivariate Gaussian and computes the covariance matrix.

LogitR minimizes the likelihood without making any assumptions on the underlying distribution. Typically performs somewhat better than LDA.
Density classification

• Suppose we know that we have \( N_0 \) and \( N_1 \) events in the sample from two categories (0=background; 1=signal)
• Suppose we know the true densities \( f_0(x) \) and \( f_1(x) \).
• Bayes formula gives:

\[
P(Y = k | X = x) = \frac{N_k f_k(x)}{N_0 f_0(x) + N_1 f_1(x)} \quad k = 0,1
\]

• Even if we don’t know \( N_0 \) and \( N_1 \), the confidence of classification is given by \( f_0(x)/f_1(x) \).
• All we need to do know is come up with good estimates of \( f_0(x) \) and \( f_1(x) \).
Naïve Bayes

\[
\{(x_n, y_n)\}_{n=1}^N, \quad x_n = \{x_n^{(d)}\}_{d=1}^D, \quad y_n = \{y_n^{(k)}\}_{k=1}^K
\]

- Model each density as a product of its projections:

\[
f_k(x) = \prod_{d=1}^D f_k^{(d)}(x^{(d)})
\]

- Building models for 1D projections is typically easy. A whole bunch of methods is available.
- We have all been Naïve Bayesians, perhaps without suspecting. A typical Max Likelihood analysis in HEP finds a 1D analytic model in each projection and then takes a product of 1D projections (often without verifying that the correlations are indeed negligible).
- **Good:** A simple and robust method, very efficient in few dimensions if there are no strong correlations between the variables.
- **Bad:** Typically poor performance in high-D problems.
Local averaging

• You can think about one zillion and one approach to estimate class densities

• Here is one:

\[
\hat{f}_k(x) = \frac{N_k(x_n \in \text{neighborhood of } x \text{ of width } \sigma)}{N_k \sigma}
\]  

\((D = 1)\)

• Works almost like a histogram – except you get a sliding window instead of bins

• This “neighborhood of \(x\)” is sometimes referred to as “Parzen window” and \(\sigma\) as “Parzen width” (free parameter of the model).

• Take a smooth estimate instead of a step-like function:

\[
\hat{f}_k(x) = \frac{1}{N_k} \sum_{y_n=k} G(x, x_n) \quad \text{e.g.} \ G(x, x_n) = \frac{1}{\sigma \sqrt{2\pi}} \exp \left(-\frac{(x - x_n)^2}{2\sigma^2}\right)
\]
k-NN methods

• Classify each point by the majority vote of $k$ nearest neighbors

\[
f(x) = \frac{\sum_{y_n} y_n}{k} \quad (\text{for } y = \{0,1\} \text{ only})
\]

• Requires no training. However, choice of $k$ requires optimization on validation data (free param in the model).

• k-NN methods are fast in low dimensions. But they consume a lot of memory for large training sets.

• A whole variety of k-NN or k-NN-like methods out there. Would you like to invent one?
Example of local averaging: Range Searching

- Count signal and background events in a hypercube around chosen location $x$. Hypercube edge length is the adjustable parameter in the model.
- Deploy a fast algorithm for scanning the neighborhood of $x$ to find training events that fall inside the hypercube.

They state that this method performs well in “high-dimensional” problems… except they consider $D=5$ as a “high-dimensional” problem.

A modern researcher in machine learning would consider $D \geq 100$ as “high-dimensional”.

![Hypothetical example with 5 correlated variables](image)

**Fig. 5:** a) The dependence of the separation power, $S = \epsilon_s/\epsilon_b$, for fixed signal efficiency on the box size and the number of events stored in the binary trees for the five-dimensional example, b) computing time needed depending on the box size of the PDE-RS method and for the NN. The arrow shows how a large numbers of events allow to use smaller box-sizes which reduces the computing time needed.
Another example of local averaging: Voronoi Adjusted Boundary (by yours truly)

- First separate signal and background with an approximate linear boundary.
- Build a Voronoi cell around each point in the mixed (signal+background) sample. Each cell knows its Voronoi neighbors.
  - Voronoi cell = volume that is closer to this point than to any other point
- Disrupt the border around each border cell and see if you can increase the overall figure of merit by adding more cells to the signal or background area.

Ideally, all green cells should contain only white points and all red cells should contain only black points. This is our “optimization” problem.
Border adjustment

find 20 neighbors
expand into background area only
Accept? Yes.
draw new border
find 20 neighbors
expand into signal area only

Accept? Yes.

draw new border
• And after few more iterations…

SUCCESS!!!

A more realistic example
Performance of VAB versus cluster size: 2D Gaussian on uniform background

10K signal and 10K background points

100/sqrt(2)<FOM<100

Conclusion: choose 50<n<100 for this classification problem with these sample sizes
Another test of VAB: 4D Gaussian and uniform background

Concentric slices of the boundary

Distances between classified points and origin
VAB performance studies: BaBar tagging data

3-7 dimensions with training sets at the order of dozens or hundreds thousands of events.

Tagging Q is optimized and compared with the “official” BaBar Neural Network.
Feasibility of VAB in many dimensions

- Number of Voronoi neighbors, $M$, grows as $d^{2.8}$ with number of dimensions
- Minimal sample size, $N$, grows linearly with $M$
- Memory needed just to hold all lists of Voronoi neighbors grows as $NxM$
- Number of facets grows exponentially with $d$
- Triangulation at $d>6$ is not feasible due to memory exhaustion
“Voronoi Likelihood Ratio Approximation”

- Kyle Cranmer at BIRS workshop
- Analyze MiniBoone PID data: 50D and ~100k events
- Choose 1000 signal and 1000 background events as Voronoi centers
- Use 10k signal and 10k background events to compute signal/background density ratio in these 2000 cells
- Then apply to test data

The method works but the performance is quite poor. Performance in low dimensions should be much better.
Curse of dimensionality

- Consider uniformly distributed points in a $D$-dimensional hypercube.
- To cover fraction $r$ of the volume, we need to cover $r^{1/D}$ fraction of range in each input variable.
  - To cover 1% of volume in 1D, take 1% of input range.
  - To cover 1% of volume in 10D, take 63% of input range.
  - To cover 1% of volume in 100D, take 95% of input range...
- In many dimensions “local” methods are no longer local.
- Most points lie near the edges of the hypercube.
- To adequately sample the variable space in many dimensions, we need to increase the sample size exponentially.
  - $N_1$ points in 1D $\Rightarrow N_1^D$ points in $D$ dimensions to obtain the same density.
- “Nearest” in many dimensions is very sensitive to definition of distance.
Local Averaging: Summary

• If you think you know something about machine learning, you must invent a local averaging method… and declare it “new” (or “novel” at the very least).

• This method will probably work fine in low dimensions. With any luck, you may be able to find a problem for which it performs better than any other method considered.

• But local averaging, k-NN and similar distance-based methods are a hopeless choice for high-dimensional problems due to curse of dimensionality. Don’t even try!!!
Neural Networks

Initialize input layer: \[ f_i^{(0)} = x_i; \ i = 1, ..., D \]

1) Forward propagation from layer \( n \) to layer \( n+1 \) using linear weights and an activation function \( \varphi \).

\[ v_i^{(n+1)} = \sum_{j=1}^{D^{(n)}} w_{ij}^{(n)} f_j^{(n)}; \quad f_i^{(n+1)} = \varphi(v_i^{(n+1)}) \]

2) Compute classification error in the output layer for \( K \) classes:

\[ R = \sum_{k=1}^{K} (y_k - f_k^{(OUT)})^2 \]

3) Propagate classification error back to update weights

\[ \Delta w_{ij}^{(n)} = -\eta \frac{\partial R}{\partial w_{ij}^{(n)}} \]

Logistic, or sigmoid, activation function:

\[ \varphi(v) = \frac{1}{1 + \exp(-av)} \]

\[ \frac{d\varphi}{dv} = a \varphi(1 - \varphi) \]

Changes fastest in midrange => contributes to the stability of the network.

From now on, I will refer to this as “standard NN.”
Neural Networks: Good and Bad

- NNs have been used in plenty of HEP analyses. The only sophisticated classifier that has become truly popular in HEP.
- If properly trained, typically provides a very good separation power.
- Fragile in the presence of strongly correlated inputs, inputs of mixed type (real and integer), and irrelevant inputs.
- CPU training time scales as $O(D^2)$ or worse in high dimensions because one needs to expand the hidden layer to capture the increasing complexity of data.

Example: two bivariate Gaussians on uniform background.

NN misses the left Gaussian because the training patterns were presented to NN asymmetrically: First all patterns for the right Gaussian, then all patterns for the other. If the patterns are mixed, NN easily finds both.
Radial Basis Functions

find \( f(x) \) that minimizes \( R = \frac{1}{N} \sum_{n=1}^{N} L(y_n, f(x_n)) + \lambda |Df|^2 \)

• Introduces a penalty term to enforce smoothness. Works similar to validation, i.e., helps to avoid overtraining.
  
  • Functional form of the solution is determined by the chosen loss and penalty term.

  • Quadratic loss and the linear differential penalty term give rise to radial basis function formalism.

  \[
  f(\lambda)(x) = \sum_{n=1}^{N} w_n G(x, x_n); \quad w = (G + \lambda I)^{-1} y; \quad G_{ij} = G(x_i, x_j)
  \]

Solution is sought in terms of Green functions \( G \) for operator \( D^*D \)

Regularization Theory: the \( \lambda \) term can regularize singular matrix \( G \)

If operator \( D \) is translationally and rotationally invariant (which is most usually the case), one can show that \( G \) is radially symmetric:

\[
G(x, x') = G(|x - x'|)
\]
Two most popular RBF solutions

1) Smoothing splines

\[ G(r) = \begin{cases} 
(r / \rho)^{2m-d} \log(r / \rho) & d \text{ even} \\
(r / \rho)^{2m-d} & d \text{ odd}
\end{cases} \]

\[ f(x) = \sum_{n=1}^{N} w_n G(|x - x_n|) + P^{(m-1)}(x) \]

- d=1 m=2 => natural cubic spline \( G(r) = r^2 \log(r) \)
- d=2 m=2 => thin-plate spline

2) Gaussian kernel regression

\[ G(r) \propto \exp \left( -\frac{r^2}{2\sigma^2} \right) \]

- Nadaraya-Watson estimator \( \lambda = 0 \Rightarrow f(x) = \frac{\sum_{n=1}^{N} y_n G(|x - x_n|)}{\sum_{n=1}^{N} G(|x - x_n|)} \)
How can we make an RBF network smaller?

- Instead of looking for an exact regularized solution, search for an approximate solution in a lower-dimensional space

\[ f(x) = \sum_{i=1}^{M} w_i G(|x - t_i|); \quad i = 1, \ldots, M; \quad M < N \]

- Solution is now given by

\[ w = \left( G^T G + \lambda T \right)^{-1} G^T y; \quad G_{ij}^{(N \times M)} = G(x_i, t_j); \quad T_{ij}^{(M \times M)} = G(t_i, t_j) \]

- Now we have to select centers of expansion \( t_i \). A number of strategies can be deployed.
RBF networks

- **RBF as a network**
  - one hidden layer
  - one hidden unit per expansion center
  - RBF learning algorithm

In the NN representation there may or may not be an explicit smoothness penalty. The optimized parameters are weights, scale factors (individual for each expansion center) and locations of the centers of expansion. Smoothness is guaranteed through validation.
RBF: Good and Bad

• RBF is low-D classifier. Radial symmetry breaks in high-D due to the curse of dimensionality.
• Largely unpopular in HEP. I found only two applications to HEP analysis.

Tested RBF in up to 3D only.

Ilya Narsky
SLUO Statistics Lectures, August 2006
Lecture 2 Slide 30
Support Vector Machines

Vapnik et al., 1992

- Idea: draw an optimal separating hyperplane between two *perfectly separated* classes

\[ y_n = -1 \text{(bgrnd)} \text{ or } 1 \text{(signal)} \]

Optimization problem:

\[ \min |w| \text{ such that } y_n(w_0 + w^T x_n) \geq 1; \ n = 1, \ldots, N \]

This optimization problem can be solved, e.g., using Lagrange multipliers. It can be shown that the solution is of the form

\[ w = \sum_{n=1}^{N} \alpha_n y_n x_n \text{ subject to } \sum_{n=1}^{N} \alpha_n y_n = 0 \text{ and } \alpha_n \left[ y_n(w_0 + w^T x_n) - 1 \right] = 0 \]

\( x_n \) for which \( \alpha_n \neq 0 \) are called support vectors (there are only 3 for the picture above).
What if the classes overlap?

Solution: map input space onto a high-dimensional feature space where the two classes are linearly separable:

\[ \left\{ x^{(d)} \right\}_{d=1}^{D} \mapsto \left\{ h^{(m)}(x) \right\}_{m=1}^{M}; \quad M \gg D \]

Search for solution in the form

\[ f(x) = w_0 + w^T h(x) \]

Similar to the case with perfectly separated patterns, we obtain:

\[ f(x) = \alpha_0 + \sum_{n=1}^{N} \alpha_n y_n G(x, x_n) \quad G(x, x') = \langle h(x)h(x') \rangle \]

Surprise! What we see here is very much similar to the Radial Basis Function expression we have seen before.

We do not even care about the specific form of \( h(x) \). All we need is to find a good kernel.
Are SVM equivalent to RBF?

• SVM allow for non-radial kernels:
  \[ G(x, x') = (1 + x^T x')^d \]
  \[ G(x, x') = \tanh(\beta_0 + \beta_1 x^T x') \]

• …and this is why they perform better in high-D than RBFs.

• SVMs are quite popular these days (not in HEP though). They generally give good performance and can be quickly trained in high-D using simple polynomial kernels.
More classifiers next time…