Machine Learning

Jamie Saxon

Introduction to Programming for Public Policy

November 14, 2016

Introduction

Machine learning is rapidly evolving, and changing the world.

Algorithms use large amounts of data to 'learn' the answers, by tuning parameters. The algorithm contains no information about the solution.

- Ranking and classification (interventions).
- Playing (winning) human games (go).
- Search optimization.
- Speech and face recognition...

These are (ambiguous) optimization problems: inherently not-exact.

Today we'll focus on some methods; on Wednesday we'll implement and explore a simple neural network.

This is 'just for fun' – so please enjoy it and focus on the big picture.

A Few Distinctions

- Supervised learning is learning from a 'training' sample by minimizing the total error (wrong answers).
- Deep learning (opposed to shallow) is characterized by many hidden layers and (typically) unsupervised or semi-supervised feature extraction.
  - For instance: nose recognition as a part of face recognition.
- Artificial intelligence is (my view!) a slightly older pursuit of mimicking human intelligence and interactions. Machine learning is a subset of AI, focussing on general learning techniques that can be applied to a range of data problems.
- Classification problems assign (discrete) class membership; regression problems try to learn the 'right value.'

We'll work on shallow, supervised learning for classification problems.

Classification Problems in Public Policy

Use a labelled 'training sample' to learn to predict group membership.

Where and how to intervene?

- Voter outreach.
- Building or health inspections.
- Police training.

Note the potential tension between identification and measurement: maximizing discrimination v. minimizing error (real world v. academia).

Problems with Classification in Public Policy

- Big picture, this was a bad training sample (not enough black faces).
- It's important, when using machine learning, to study its biases.

BBC Sign in | News | Sport | Weather | Shop | Earth | Travel | More | Search

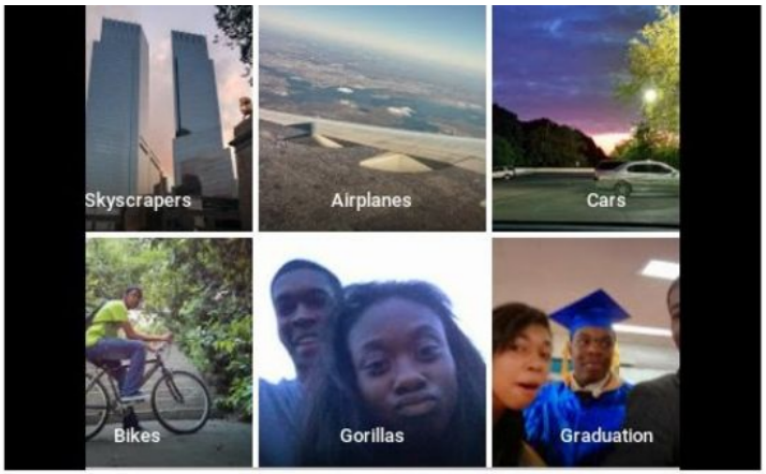
NEWS

Home | Video | World | US & Canada | UK | Business | Tech | Science | Magazine | Entertainment & Arts | Health | More

Technology

Google apologises for Photos app's racist blunder

1 July 2015 | Technology | Share



diri noir avec banan @jackyalcine · Jun 29

Top Stories

Obama may weigh in on Trump after office
President Obama says he may speak up, against tradition, if his successor threatens "core values".
1 hour ago

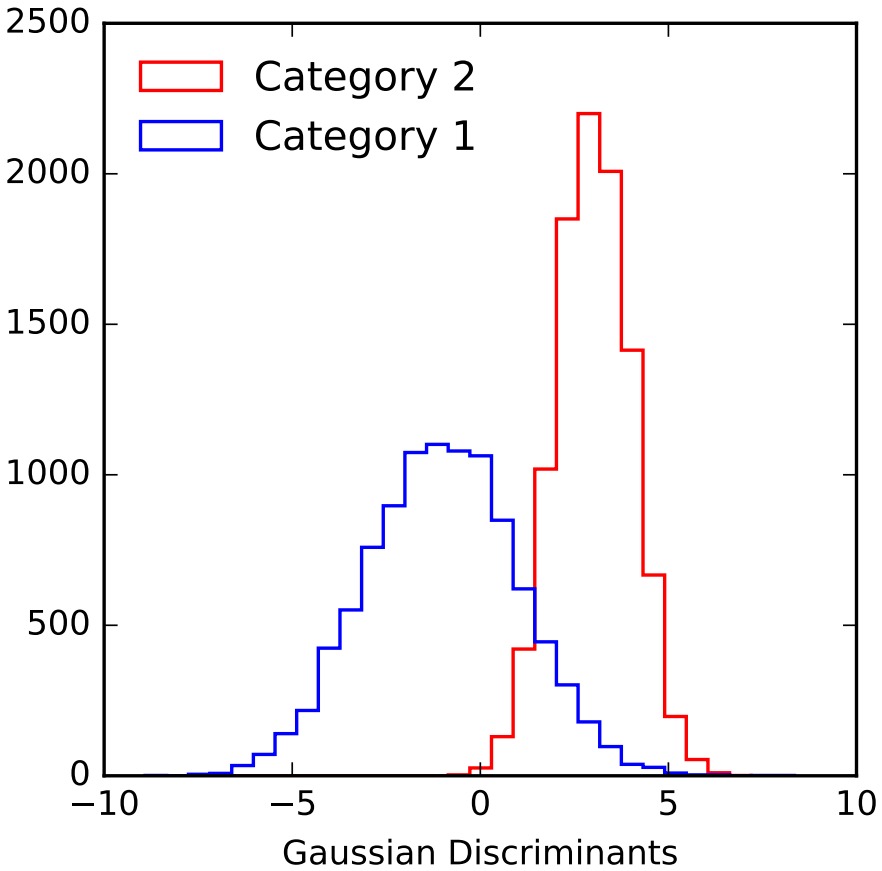
Trump eyes Romney for top diplomat job
20 November 2016

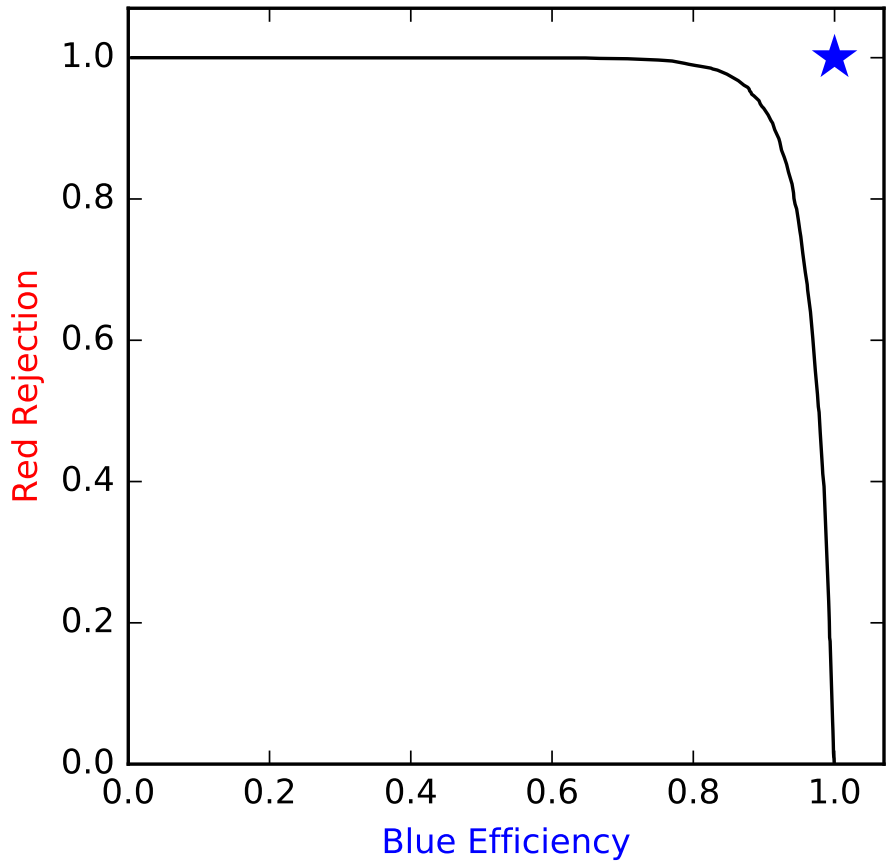
Sarkozy out of centre-right primary
9 hours ago

Features & Analysis

Classification in One Dimension

- Consider two categories of objects separated by a single variable δ.
- Select objects/people/events/etc. with $\delta \leq \delta_{\text{cut}}$ and reject others.
- Want high efficiency and low false-positive rate: $\star$ on 'ROC' curve.





More Variables, and Correlations Between Variables

Seek to combine many variables into a single, optimal discriminant.

- At the simplest level, you can just make a series of 'cuts' on many discriminants, to isolate your signal/population of interest.
  - You can already do this with pandas masks!!
- Much better: use (non-linear) correlations between variables.
  - Linear Discriminant Analysis or Principal Component Analysis.
  - Nearest Neighbors, Binary Decision Trees, **Neural Nets**.

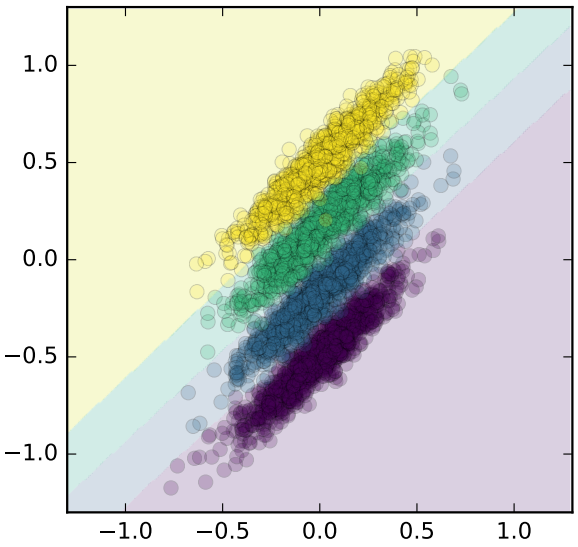
Analytic Multivariate Methods (Not 'Learning')

- You don't need to use neural nets to achieve good discrimination.

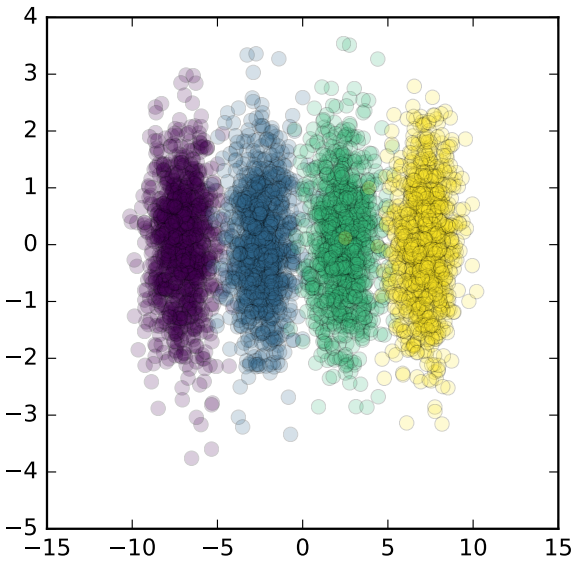
Analytic Multivariate Methods

Without formal 'learning' we can apply some analytical techniques to extract 'optimal' cuts.

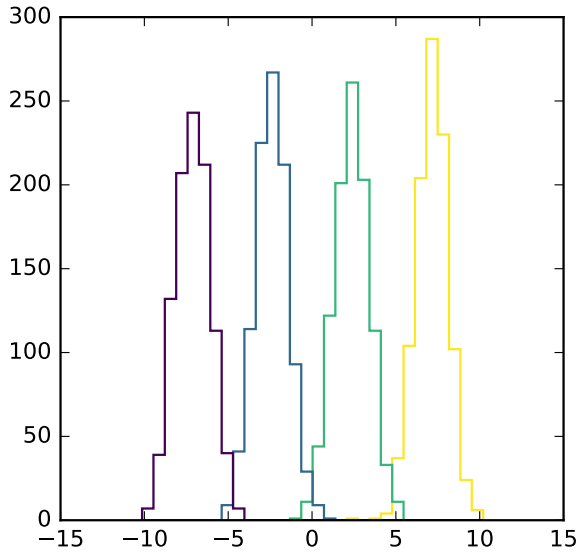
- Linear Discriminant Analysis identifies the straight lines that best separate the variables.
- Closely related to Quadratic Discriminant Analysis (two lines!), Principal Component Analysis (PCA) and factor analysis.



Dataset



Transformed



Projected

Linear Discriminant Analysis: Math (for Two Categories)

The density of a multivariate normal distribution with means $\overline{\boldsymbol{x}}$ and covariances $\boldsymbol{M}$ is:

$$f(\boldsymbol{x}) = \exp\left(-\frac{1}{2}(\boldsymbol{x}-\overline{\boldsymbol{x}})^T\boldsymbol{M}^{-1}(\boldsymbol{x}-\overline{\boldsymbol{x}})\right) \Big/ \sqrt{(2\pi)^k|\boldsymbol{M}|}$$

Hence the log-likelihood ratio for membership in two sets is

$$\frac{1}{2}(\boldsymbol{x}-\overline{\boldsymbol{x}}_0)^T\boldsymbol{M}_0^{-1}(\boldsymbol{x}-\overline{\boldsymbol{x}}_0) + \frac{1}{2}k\ln(2\pi) + \frac{1}{2}\ln|\boldsymbol{M}_0|$$
$$-\frac{1}{2}(\boldsymbol{x}-\overline{\boldsymbol{x}}_1)^T\boldsymbol{M}_1^{-1}(\boldsymbol{x}-\overline{\boldsymbol{x}}_1) - \frac{1}{2}k\ln(2\pi) - \frac{1}{2}\ln|\boldsymbol{M}_1| > \text{const.}$$

For LDA, we assume that $\boldsymbol{M} = \boldsymbol{M}_0 = \boldsymbol{M}_1$.

$$(\boldsymbol{x}-\overline{\boldsymbol{x}}_0)^T\boldsymbol{M}^{-1}(\boldsymbol{x}-\overline{\boldsymbol{x}}_0) - (\boldsymbol{x}-\overline{\boldsymbol{x}}_1)^T\boldsymbol{M}^{-1}(\boldsymbol{x}-\overline{\boldsymbol{x}}_1) > \text{const.}$$

Since $\boldsymbol{M}$ is Hermitian, $\boldsymbol{x}^T\boldsymbol{M}^{-1}\overline{\boldsymbol{x}}_i = \overline{\boldsymbol{x}}_i\boldsymbol{M}^{-1}\boldsymbol{x}$, and this simplifies to:

$$\boldsymbol{x}\boldsymbol{M}(\overline{\boldsymbol{x}}_0 - \overline{\boldsymbol{x}}_1) > \textit{const.}$$

Taking $\boldsymbol{w} \equiv \boldsymbol{M}(\overline{\boldsymbol{x}}_0 - \overline{\boldsymbol{x}}_1)$, we classify group membership by $\boldsymbol{x}\cdot\boldsymbol{w} > \text{const.}$

Linear Discriminant Analysis: More Math Notes

- Generalizing to multiple classes (as we've used here) requires a bit more work, but (can) yield eigenvalues that correspond to the directions that yield the best separation.
  - This is how we projected to one dimension.
- Alternatively, one can query each individual class against the rest and take the winner.

Using LDAs in sci-kit learn

This is implemented in sci-kit learn, with `numpy` arrays:

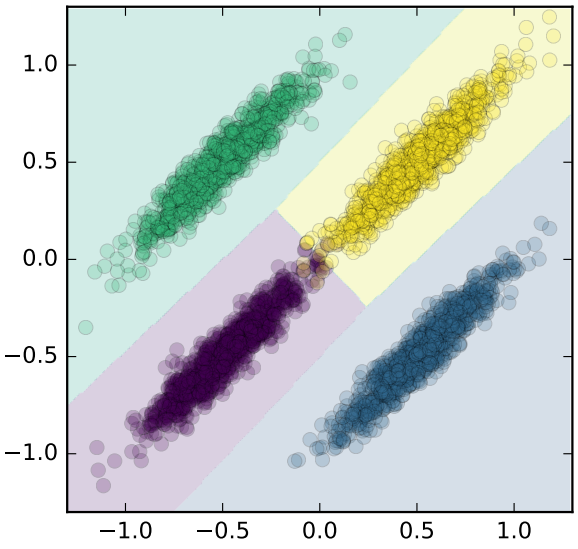
- These arrays are what `pandas` DataFrames are built on.

```
# my function for generating a dataset
# X is a 2D array with 1000 points per category.
# y is category label.
X, y = blob_data(noise_level=0.25, correlated=0.8)
K = len(set(y)) # number of classes

# Import and instantiate and LDA.
from  sklearn import  discriminant_analysis as da
lda = da.LinearDiscriminantAnalysis()
lda.fit(X, y) # fit it to data.

cat = lda.predict([[1, 1]])
```

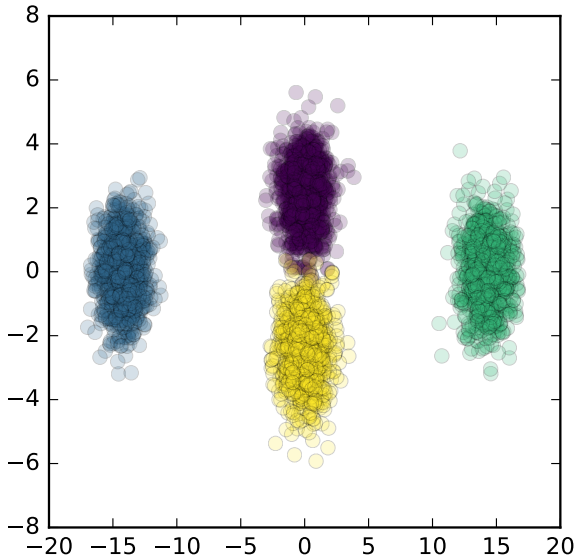
A Second LDA Example

So long as we stay linear, we're alright.

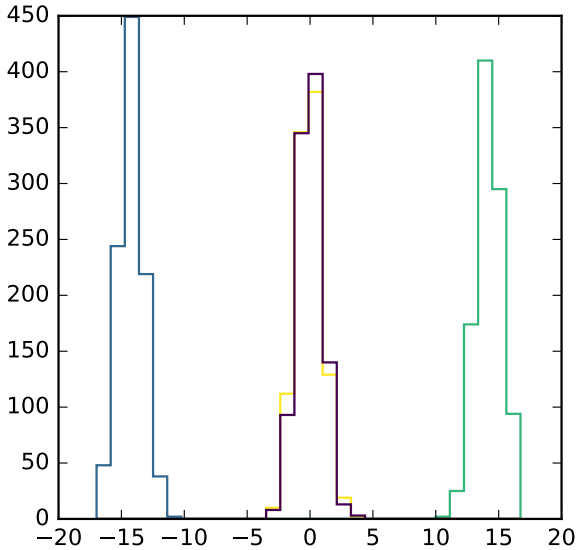
- LDAs, PCAs, etc. may be used to maximize discrimination/preprocess for learning.



Dataset



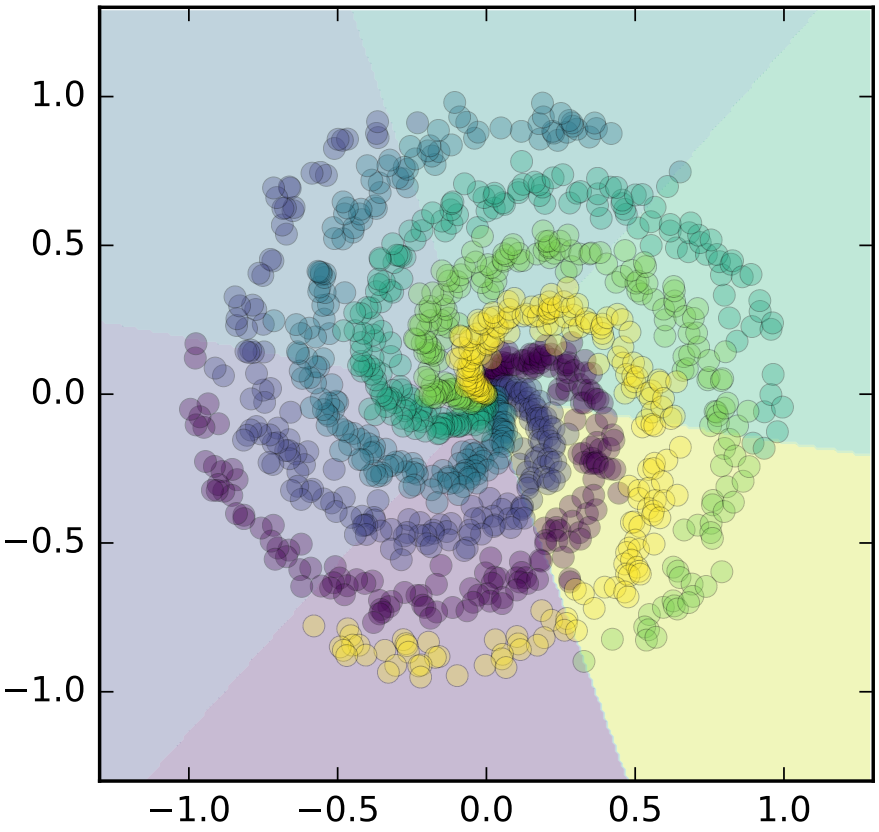
Transformed



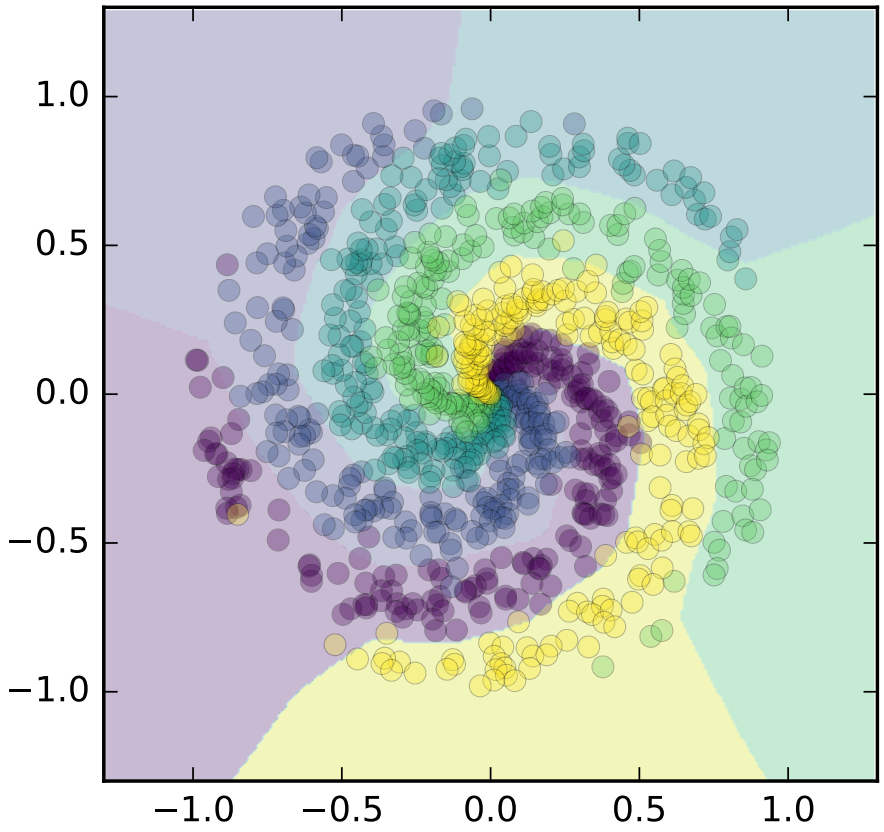
Projected

Limits of Linear Classifiers

- Move to a more complicated (though still somewhat contrived) generating function, but stick to two dimensions for exposition.
- With a 'spiral,' the linear classifiers fail, and need something stronger.



LDA Fail



NN Classifier

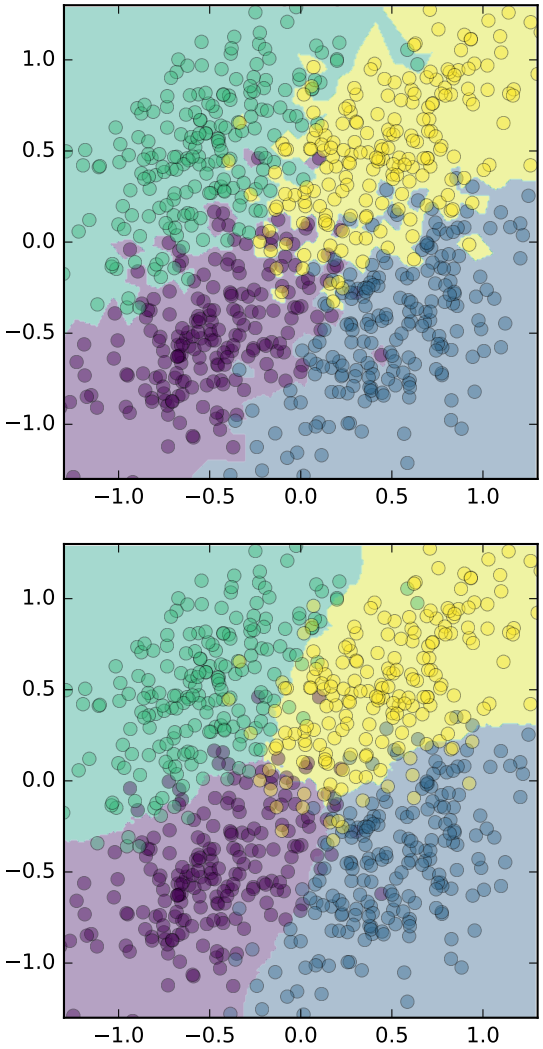
Machine Learning

Nearest Neighbors

Nearest neighbors uses a labelled 'training' dataset as a 'dictionary.'

- Simplest case: classify new objects by matching the single closest object in the reference dataset.
- Leads to canonical **over-training**: classification reflects not only the generating function, but also the statistical fluctuations of the sample.
- Therefore: average over k nearest neighbors.
  - Analogous to 'regularization' in neural nets.
  - ***k*** is our first **hyperparameter**.
- Large training dataset translates to high precision but slow evaluation.

Overtraining

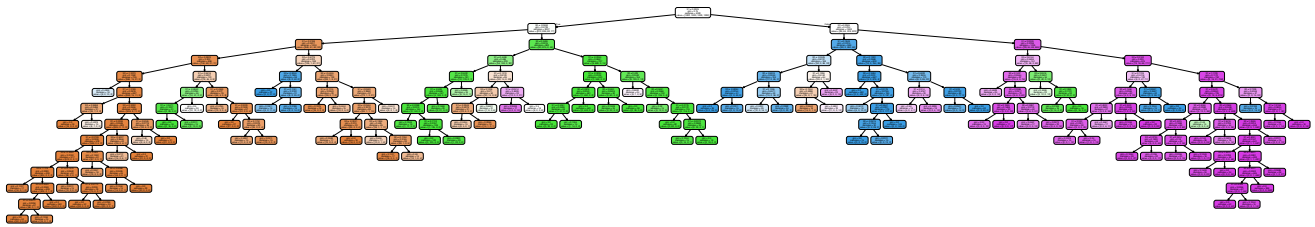


$k = 30$

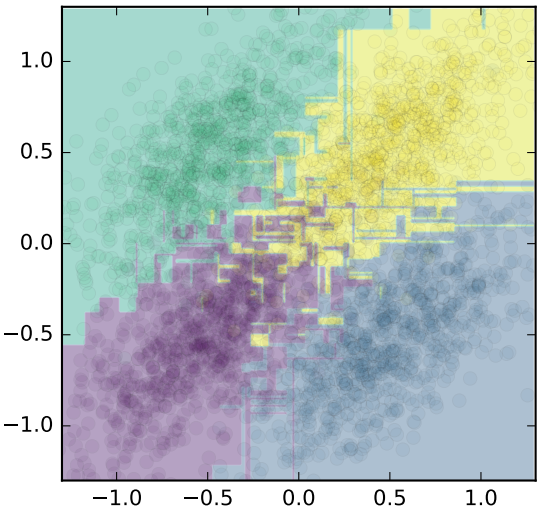
Binary Decision Trees

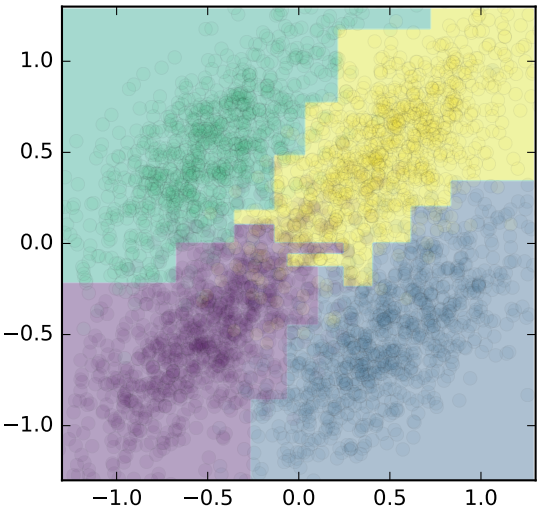
Binary decision trees use successive splits on the input variables to partition the sample.

- The average value in a 'leaf' is the output.
- Because each cut 'straight,' can be useful to rotate (PCA/LDA) before training.
- Like 'nearest neighbors,' BDTs are susceptible to overtraining. One can protect against this by requiring a minimum population in each leaf.
- Unlikely, kNN, extremely fast evaluation.



Overtrained





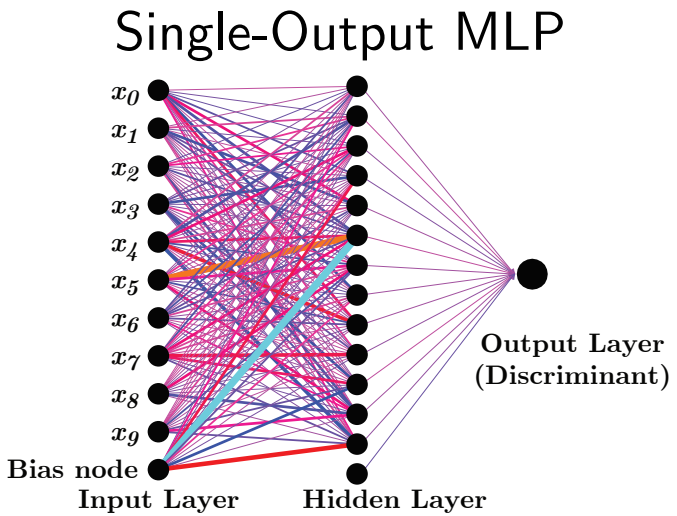
Min Leaf 15

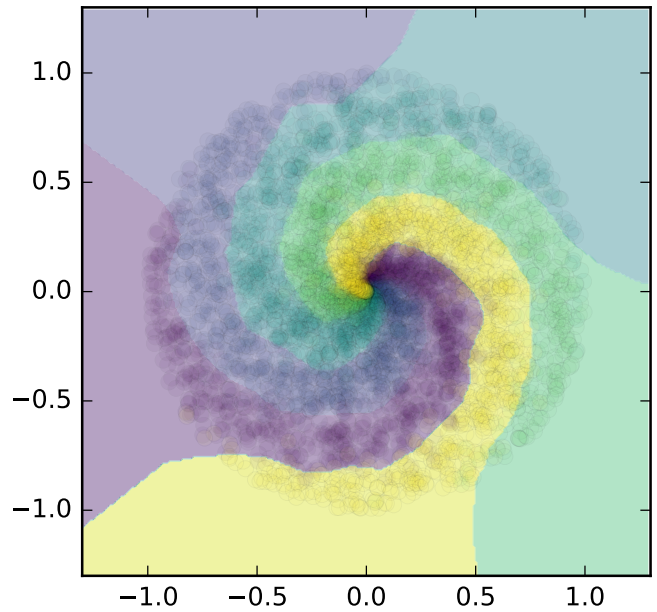
Multilayer Perceptrons (MLPs)

An MLP consists of a network of nodes with activation functions, and a collection of weights for the connections between those nodes.

$$d_k = g_k\left(\sum_{j=0}^{M} w_{kj} \times g_j\left(\sum_{i=0}^{N} w_{ji}^{h} x_i\right)\right)$$

- Activation function: sigmoid/tanh, ReLU (rectified linear unit, $\max(0, x)$), etc.
- Many hyperparameters (number and size of hidden layers, etc.).
- 'Regularization' to penalize extreme weights and avoid over-training.





Classifier

Training

We've attained some pretty magical results by 'training' BDTs and MLPs to recognize patterns.

So how does the training work!?

Define a loss as the sum of all of the mis-classifications on the entire dataset, $L = \sum L_i$. Define the element-wise loss-by

- Support vector machine: seek a gap in score of at least X above other classes.

$$L_i = \sum_{j \neq y_i} \max(0, X + f_j - f_{y_i})$$

- Softmax: the negative log 'probability' of the right class:

$$L_i = -\log(e^{f_{y_i}} / \sum_j e^{f_j})$$

- Also include a regularization term that penalizes large weights: $\frac{1}{2}\alpha w^2$.

Minimize the Loss

Iterate over the dataset, updating the biases and weights to minimize the loss: **back-propagation!!**

- Back-propagation is doing the chain-rule backwards.
  - Mainly addition with weights. Multiplication 'swaps' the gradient: the gradient on the weight is $dw = do \cdot i$, and the gradient on the input is $di = do \cdot w$.*
  - We'll use ReLU for our activation, which makes the derivatives easy: turns them off when paramater is negative.
  - Since we defined the regularization as $\alpha w^2/2$, its contribution is αw.
  - Trick is L_i to the scores: $\partial L_i / \partial f_k = p_k - \mathbb{1}(y_i = k)$.
- Save the values of all nodes in one forward pass, then work backwards.

We'll implement this on Wednesday.

*If the gradient of the full output with respect to this node is $\frac{\partial f}{\partial q}$, and $q = xy$, then $\frac{\partial f}{\partial x} = \frac{\partial f}{\partial q}\frac{\partial q}{\partial x} = \frac{\partial f}{\partial q}(y\frac{\partial x}{\partial x} + x\frac{\partial y}{\partial x}) = y\frac{\partial f}{\partial q}$

So How Do We Set the Hyperparameters

- Training only gives us the weights. How many nodes? Training iterations? Minimum leaf size? Regularization? Step size?
- Standard practice is to preserve a validation sample that we do not train against, to monitor convergence. When iterations (back-propagation) on the training sample no longer yield improvement on the validation sample, stop.
- Also preserve a totally sacrosact test sample, that you use only at the very end to test performance (it's easy to 'tune' for the validation sample, by hand).