Classification II

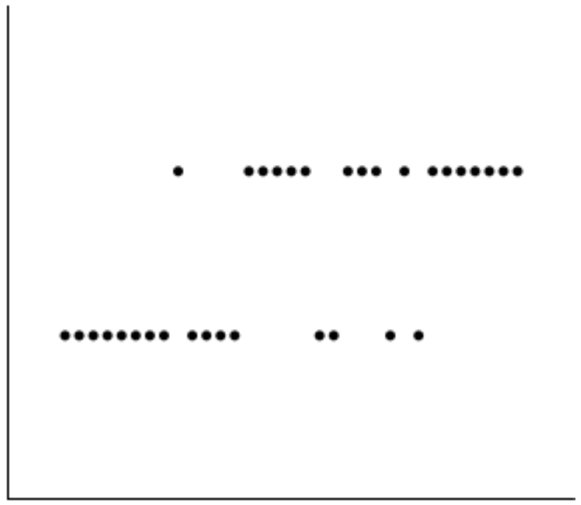
Logistic Regression, Naive Bayes, SVMs and Ensembles

Logistic Regression

Aka logit-regression, maximum entropy (MaxEnt), or log-linear classifier

Binary Output Variable

Given a binary classification target we can see that it is very difficult to establish a relationship between X and Y.

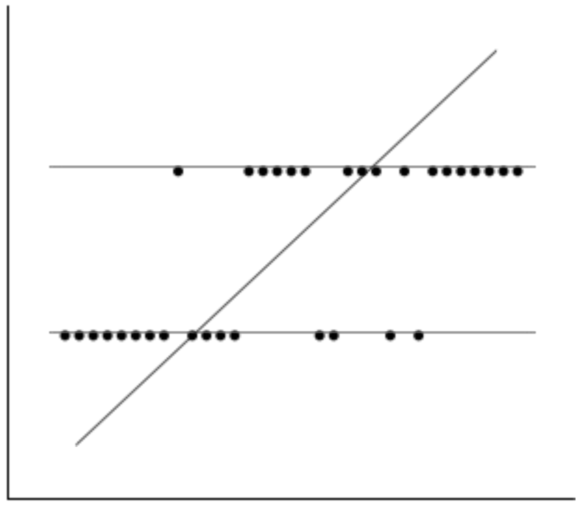


Binary Output Variable

Given a binary classification target we can see that it is very difficult to establish a relationship between X and Y.

Fitting a linear relationship on top of this model severely oversimplifies the relationship.

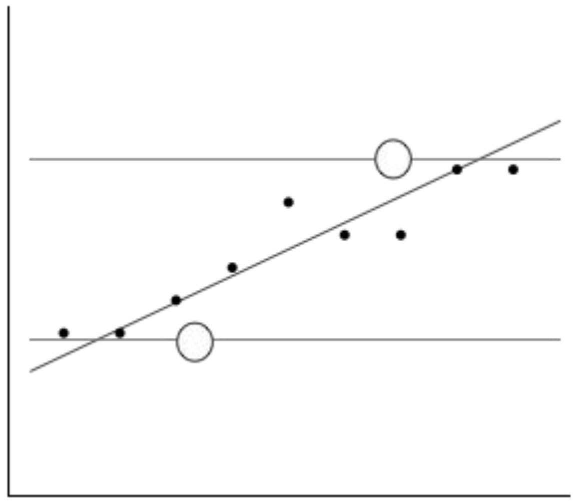
Provides unobservable predictions for small and large values of X.



Binomial Transformation

The mean of binomial variable is a proportion, so we can re-encode our data as conditional probabilities and fit our model to it: e.g. learn:

`p(y | x)`



However the linear model does not predict the **maximum likelihood estimates** (shown by the circles).

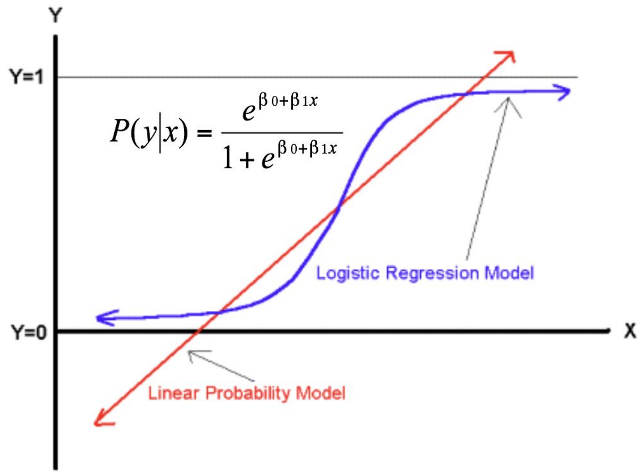
The regression is sigmoidal.

Logistic Regression

Recall we can model non-linear models via a transformation of our dataset, X.

Two possible transformations that result in sigmoidal functions:

- **Probit**: imposes cumulative normal function on the dataset X. Unfortunately, they are difficult to work with (requires integral solver).
- **Logit:** gives nearly identical values to probit, but can be modeled more easily with a linear equation.



Logit: Logarithm of Odds (Log Odds → Logit?)

Odds: range of 0 to ∞: if odds > 1 then the event is more likely to occur than not occur. If odds < 1 vent is less likely to occur than not occur.

$$odds = \frac{p}{1-p}$$

The log odds transformation creates a variable with a range from -∞ to +∞ and solves the problem with fitting the linear model to probabilities.

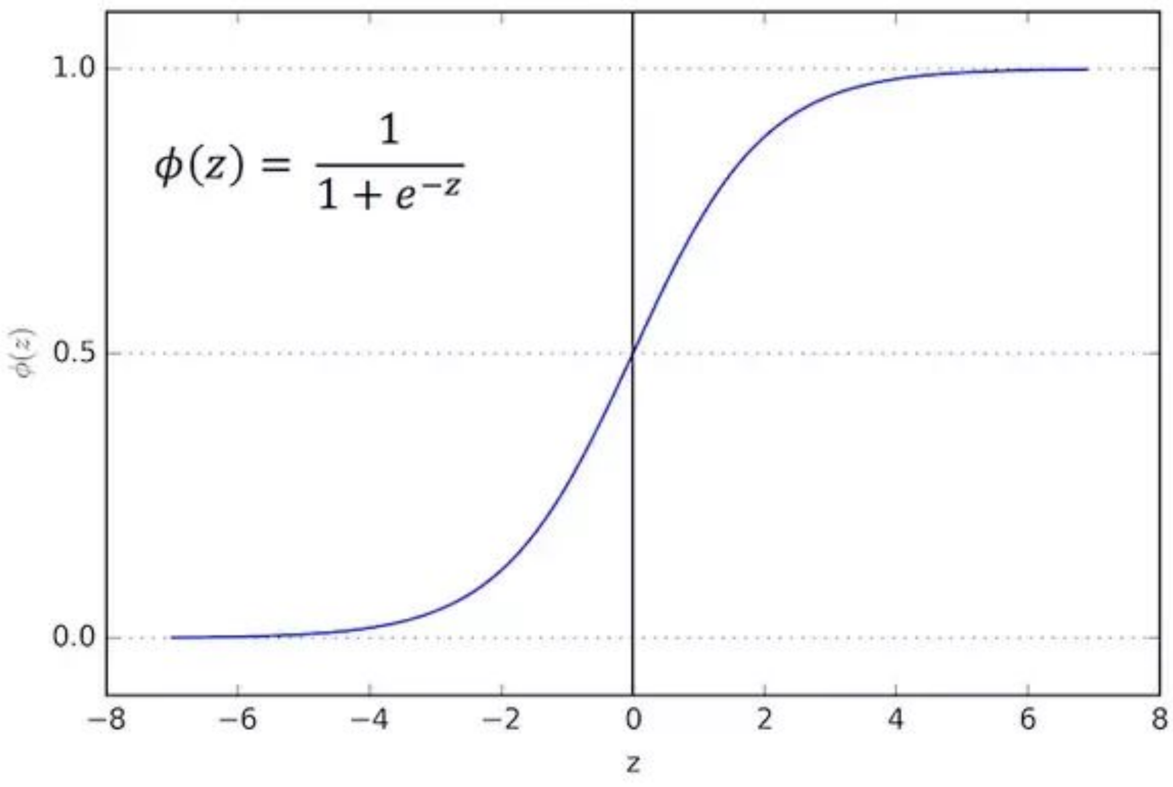
$$\ln(odds) = \ln\left(\frac{p}{1-p}\right) = \ln(p) - \ln(1-p)$$

The linear model can now capture all logits outside the range 0 to 1 and is not underfit.

In addition, if you take the exponent of the logit, you have the odds for the two groups, so the predicted logit can be transformed back to probability.

Logistic Regression

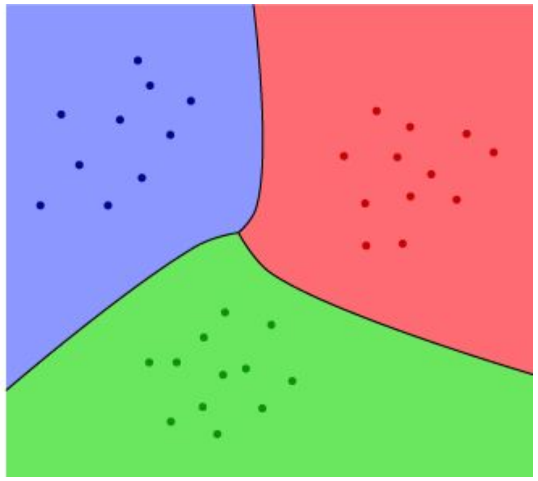
- Mathematically calculate the decision boundary between the possibilities.
- Find a line that represents a cutoff that most accurately represents the training data.
- Primarily for binary classification.

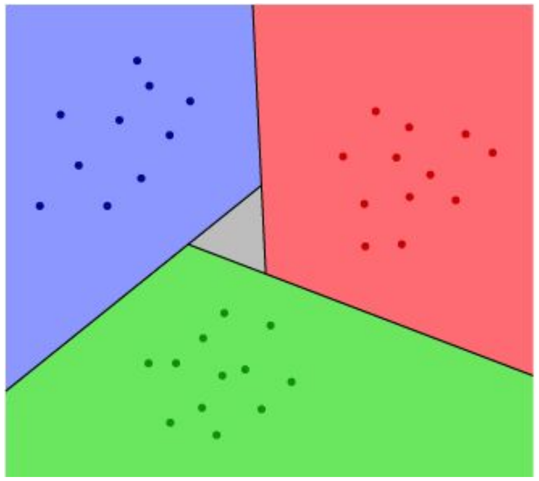


Binary vs. Multi-Class Logistic Regression

- Multiclass Logistic Regression can be computed with a multinomial logistic regression that solves the set of all binary logits. The probabilities are computed by taking the **softmax** of the odds.

- One vs. Rest strategy computes a logistic regression for A vs. not A, B vs. not B and selects the most likely estimate.





Logistic Regression in Scikit-Learn

```python
from sklearn.linear_model import LogisticRegression
from sklearn.model_selection import train_test_split as tts


X = data.features
y = data.target

X_train, y_train, X_test, y_test = tts(X, y, test_size=0.2)

penalty = l2 # norm to use for penalization
fit_int = True # whether to add bias/intercept to decision function
slvr = liblinear # algorithm to use for optimization

model = LogisticRegression(
    penalty=penalty, fit_intercept=fit_int, solver=slvr
)

model.fit(X_train, y_train)
model.predict(X_test)
```

Naive Bayes

Detecting Fraud

- Online store with at least 1000 orders per month and where 10% of orders are fraudulent
- 60 seconds to check an order, \$15 per hour to a customer service representative = \$3k per year!
- Goal: construct a model of probability that detects if an order is 50% likely to be fraudulent (reduce checking)

How likely is an order to be fraudulent?

What if we have more information, e.g. frauds are likely to use gift cards and promo codes?

Conditional Probability

$$P(A|B) = \frac{P(A \cap B)}{P(B)}$$

The probability of A given B is equal to the probability of A and B divided by the probability of B.

$$\begin{tikzcd}
P(B) = 0.20 \arrow[dr] & & \\
& P(A|B) = 0.06 / 0.20 = 0.3 \arrow[dr] & \\
& & P(A \cap B) = 0.06 \\
& P(B|A) = 0.06 / 0.30 = 0.2 \arrow[ur] & \\
P(A) = 0.30 \arrow[ur] & &
\end{tikzcd}$$

P(A|B) sits between P(A), P(B) and P(A∩B)

Detecting Fraud

We can now frame our problem in terms of the following probabilities:

$$P(Fraud|Giftcard) = \frac{P(Fraud \bigcap Giftcard)}{P(Giftcard)}$$

$$P(Fraud|Promo) = \frac{P(Fraud \bigcap Promo)}{P(Promo)}$$

But how do we compute $P(Fraud \cap Giftcard)$?

Bayes Theorem

In the 18th century, Reverend Thomas Bayes came up with the research that Pierre-Simon Laplace would beautify:

$$P(B|A)=\frac{P(A|B)P(B)}{P(A)}$$

Because:

$$P(B|A)=\frac{\dfrac{P(A\bigcap B)P(B)}{P(B)}}{P(A)}=\frac{P(A\bigcap B)}{P(A)}$$

Bayes and Fraud Detection

Now we can compute the following probabilities:

$$P(Fraud|Giftcard) = \frac{P(Giftcard|Fraud)P(Fraud)}{P(Giftcard)}$$

$$P(Fraud|Promo) = \frac{P(Promo|Fraud)P(Fraud)}{P(Promo)}$$

And these are easily computed by frequency. Let’s say that gift cards are used 10% of the time, and 60% of gift card use is fraudulent: What is the probability of P(Fraud|Giftcard)?

Bayesian Classification

You might have guessed it: use of a promo or gift card during fraud are *features* of our Bayesian model.

What we want to solve now is: `P(Fraud|Giftcard,Promo)`

Using *joint probability* we can derive the *chain rule*:

$$P(A_1,A_2,\ldots,A_n)=P(A_1)P(A_2|A_1)P(A_3|A_1,A_2)\cdots P(A_n|A_1,A_2,\ldots,A_{n-1})$$

With Bayes, we can feed lots of information into our model. Which is now as follows:

$$P(Fraud|Giftcard,Promo)=\frac{P(Giftcard,Promo|Fraud)P(Fraud)}{P(Giftcard,Promo)}$$

A Naive Assumption

Our model is still tough to solve, but let’s look at the denominator - `P(Giftcard, Promos)` - this doesn’t seem to be related to fraud.

We will make an ***assumption of independence*** and drop the denominator from our model. Using the chain rule:

$$P(Fraud,Giftcard,Promo)=P(Fraud)P(Gift|Fraud)P(Promo|Fraud,Gift)$$

Now we have to deal with P(Promo|Fraud,Gift), using the naive assumption again, along with a parameter, Z (sum of probabilities of all classes) we get the following:

$$P(Fraud|Giftcard,Promo)=\frac{1}{Z}P(Fraud)P(Gift|Fraud)P(Promo|Fraud)$$

Classification with Naive Bayes

- Building our model:
    - compute the probabilities of all features from a training set
    - compute the probabilities of all classes from training set
    - compute the parameter Z
    - Build a truth table
- Using our model:
    - compute “probabilities” of input vector
    - use truth table, trained probabilities and parameters to compute likelihood of class

	Fraud	Not Fraud
Gift Card	0.6	0.1
Promos	0.5	0.3
Class	0.1	0.9

Smoothing

- What happens if we receive new information?
- If the information has a probability = 0, then the independence assumption will cause skew.

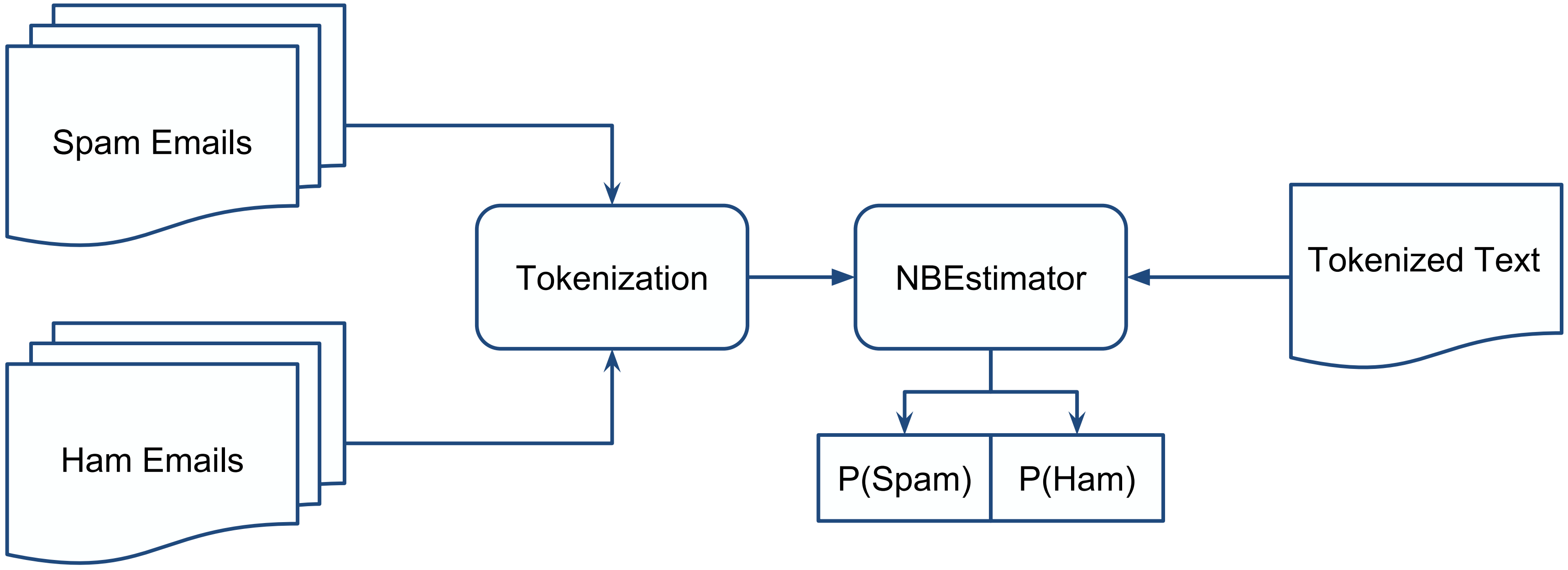
Smoothing: Donating some of the probability density to unknown information.

Easiest smoothing technique: *laplacian (additive)*
Simply give all unknown items a frequency of 1 - then recompute probabilities accordingly.

Text Classification with NB

Spam/Ham is the canonical Naive Bayesian classifier for text and is popular because if it's wide use.

Compute using "bag of words" - each word is a feature.

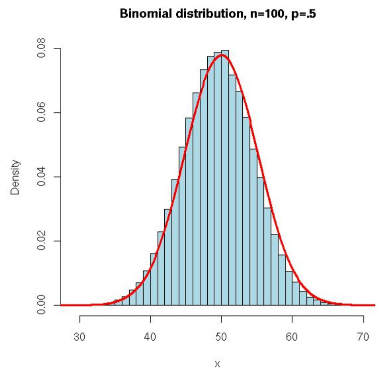


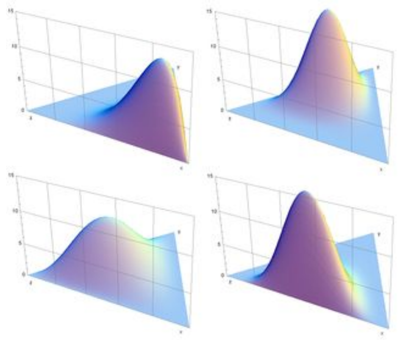
Naive Bayes in Scikit-Learn

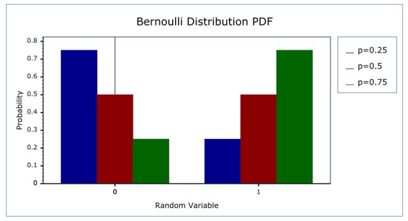
GaussianNB: the likelihood of the features is assumed to be Gaussian (standardized).

MultinomialNB: for multinomially distributed data, and is one of the two classic naive Bayes variants used in text classification (where the data are typically represented as word vector counts, although tf-idf vectors are also known to work well in practice).

BernoulliNB: for data that is distributed according to multivariate Bernoulli distributions; i.e., there may be multiple features but each one is assumed to be a binary-valued (Bernoulli, boolean) variable.



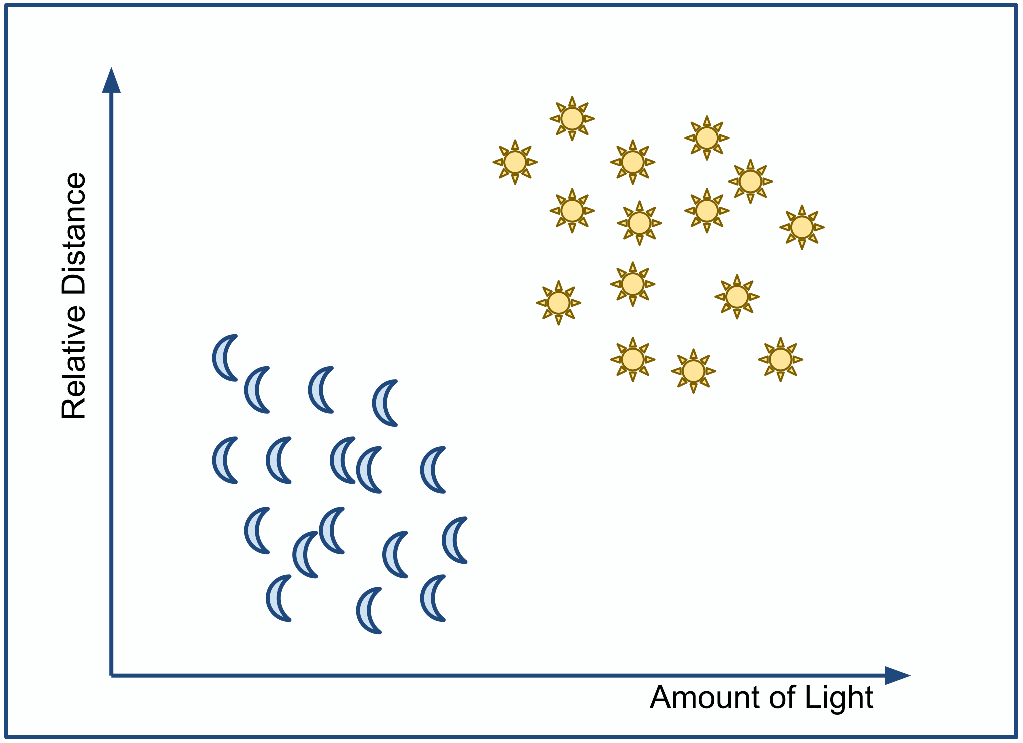




Support Vector Machines

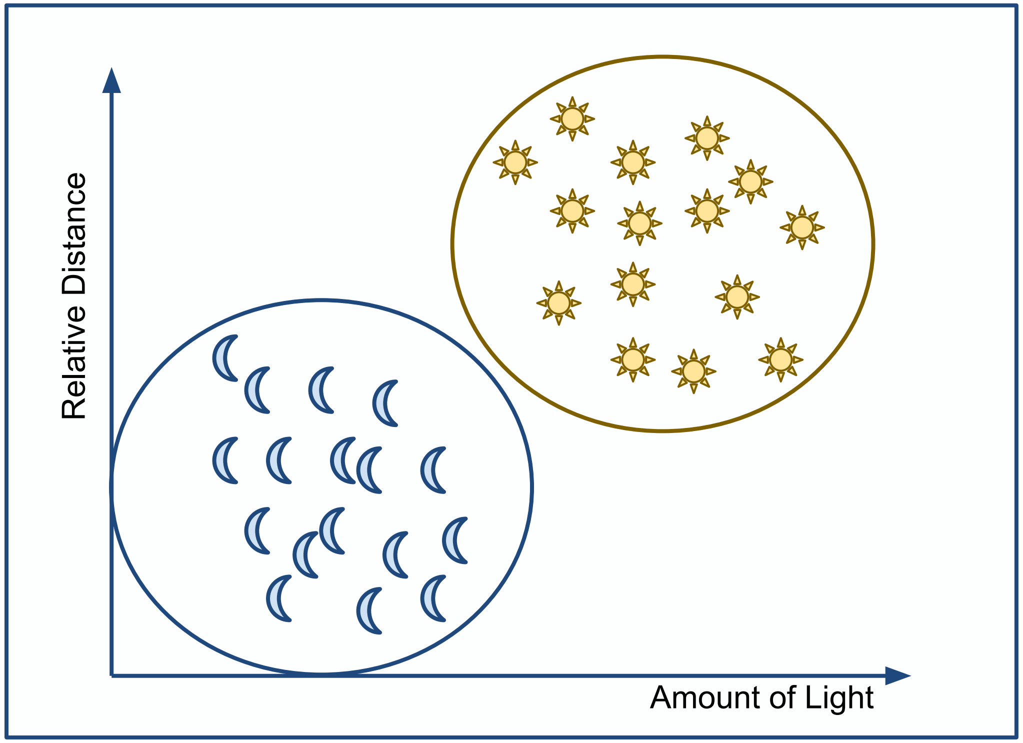
Stars or Moons?

Let’s say we know we have two discernible groups (a strong hypothesis):



Stars or Moons?

Using kNN (we’ve already looked at) - we’re highly dependent on the instances, and we get centers:



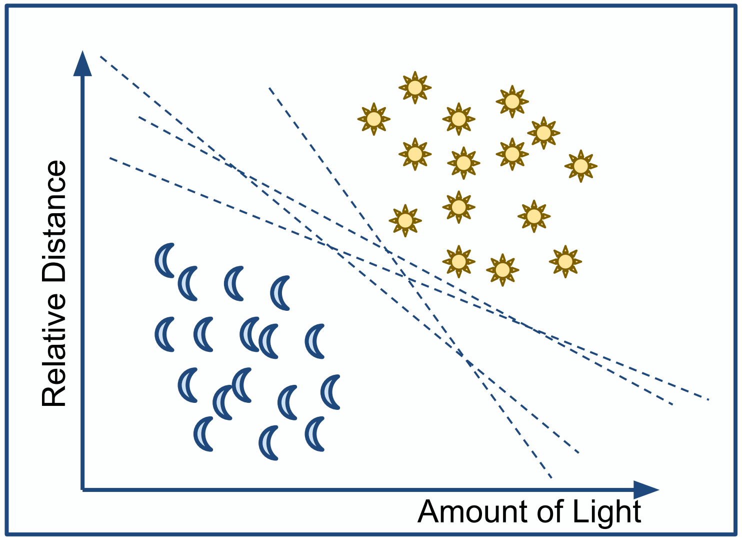
Decision Boundary

In order to *generalize* this model, we really want to find a *decision boundary* - a line between two classes of data.

How many lines fit between moons and stars?

Instead of picking an arbitrary line, we attempt to maximize the distance between classes.

Definition
In geometry a **hyperplane** is a *subspace* of one dimension less than its ambient space. If a space is 3-dimensional then its **hyperplanes** are the 2-dimensional planes, while if the space is 2-dimensional, its **hyperplanes** are the 1-dimensional lines.



Support Vector Machines

- Introduced by Vladimir Vapnik in 1980s
- Modern interpretation is less stringent (added slack)

Designed to solve a two-group classification:

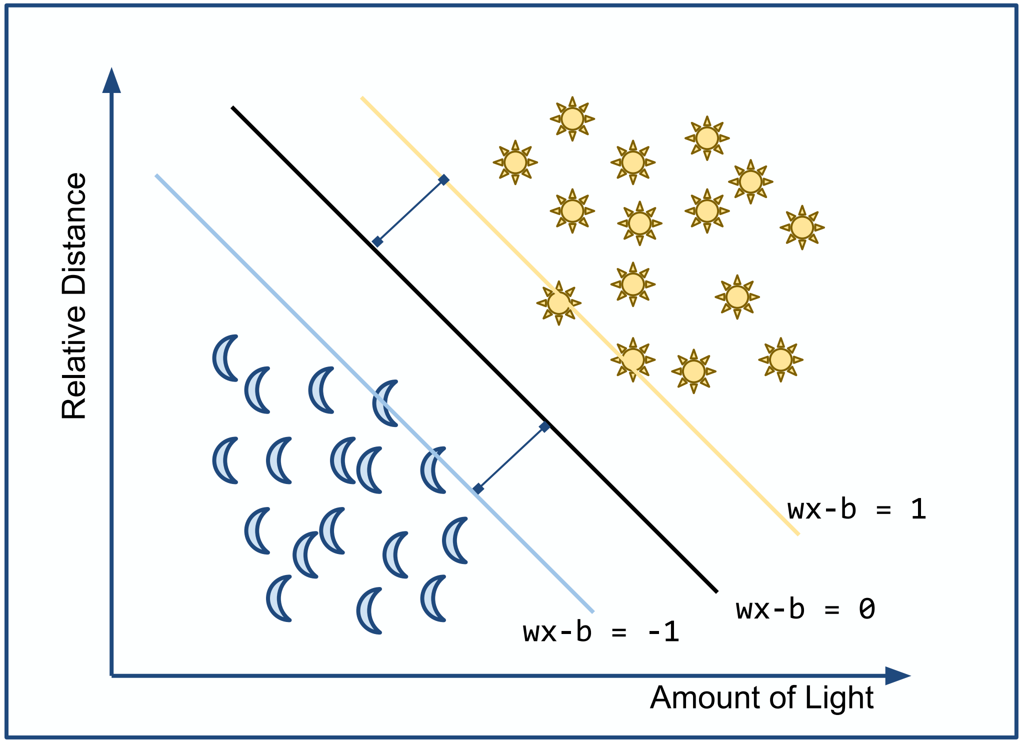
- boolean (true, false)
- ids (3,4)
- sign (1, -1)

Desirable:

- Avoids curse of dimensionality
- Works in high dimensional space
- FAST to compute

Maximizing Distance

Solve for `w` for the function `wx-b=0` to determine the hyperplane maximizing the distance between two groups.



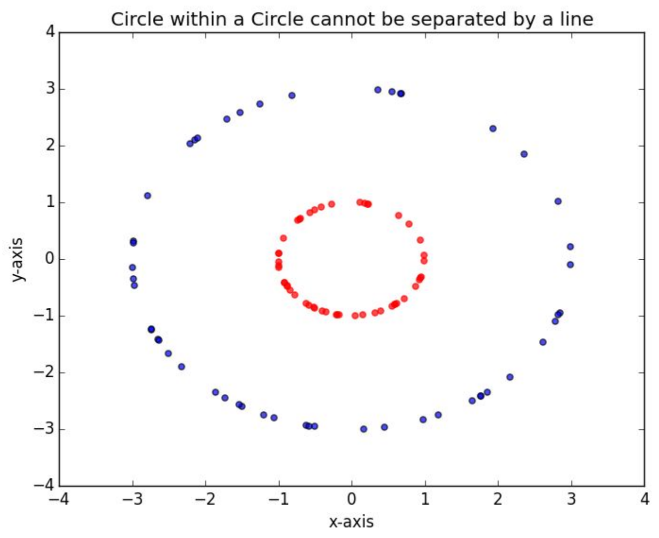
Select two “support vectors” that bound each group. Then find the parallel middle line between the support vectors.

Solvable by finding the perpendicular hyperplane to all three parallel planes and dividing by 2, which happens to be:

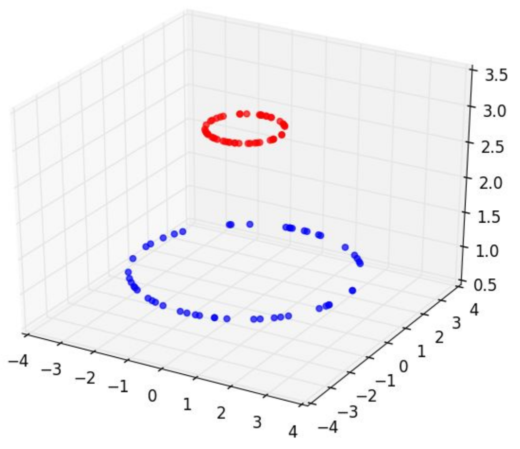
`2 / ||w||`

The Kernel Trick

- Examples are linear, real data is not (and also usually in a sparse, high-dimensional space).
- Kernel trick: transpose data into a higher dimensional space. E.g. for circles (polynomial kernel):



$\langle x, y \rangle$



$\langle x^2, \sqrt{2}xy, y^2 \rangle$

Trickier Kernel

Take a linear x and transform it into $\phi(x)$ or some function that better suits the data. Points to consider:

- $\phi(x_i)\phi(x_j)$ might take a long time to compute
- ϕ might be a complicated function
- Extensive training data will be slow

So, we try another trick: $K(x_i,x_j) = \phi(x_i) \cdot \phi(x_j)$

Mapping the dot product + cheap function (e.g. inner dot product is already calculated) = *kernel function*

Kernel Functions

Homogeneous Polynomial
d is the degree, any number > 0
works best in most cases.

$$K(x_i, x_j) = (x_i^T x_j)^d$$

Heterogeneous Polynomial
Add a non-negative, non-zero constant, c
Increases relevance of higher order features

$$K(x_i, x_j) = (x_i^T x_j + c)^d$$

Radial Basis Function (RBF)
Often used more: high-D performance
Can create infinite new dimensions
Numerator is euclidean distance
σ is a free parameter

$$K(x_i, x_j) = exp(\frac{||x_i - x_j||_2^2}{2\sigma^2})$$

Kernel Functions in Scikit-Learn

Linear Kernel:
C is slack variable

$$\gamma\langle x_i, x_j\rangle$$

Polynomial Kernel:
d is degree, r is coef0

$$(\gamma\langle x_i, x_j\rangle + r)^d$$

RBF Kernel:
gamma keyword > 0

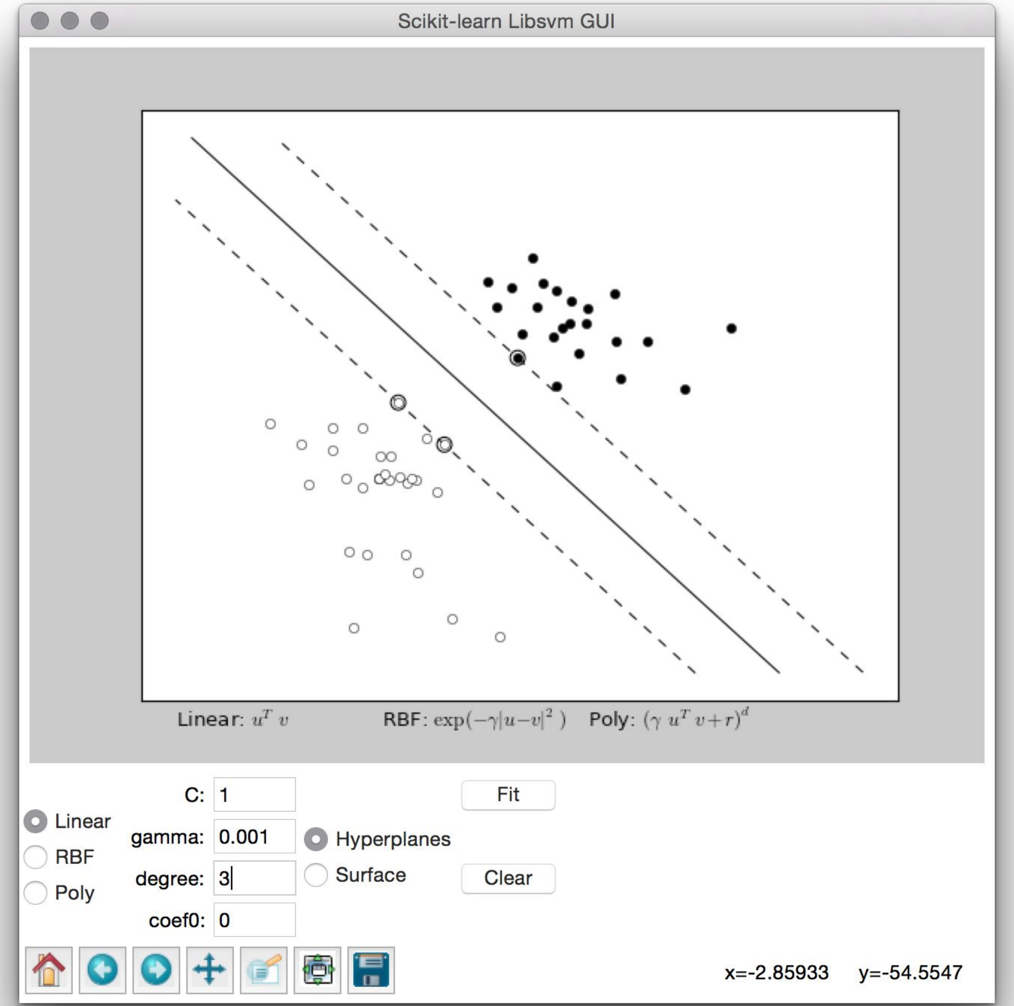
$$exp(-\gamma|x_i - x_j|^2)$$

Sigmoid Kernel:
r is specified by coef0

$$\tanh(\gamma\langle x_i, x_j\rangle + r)$$

Define your own! Specify a Python function or precompute Gram matrix.

SVM GUI



[✓ Linear] RBF Poly

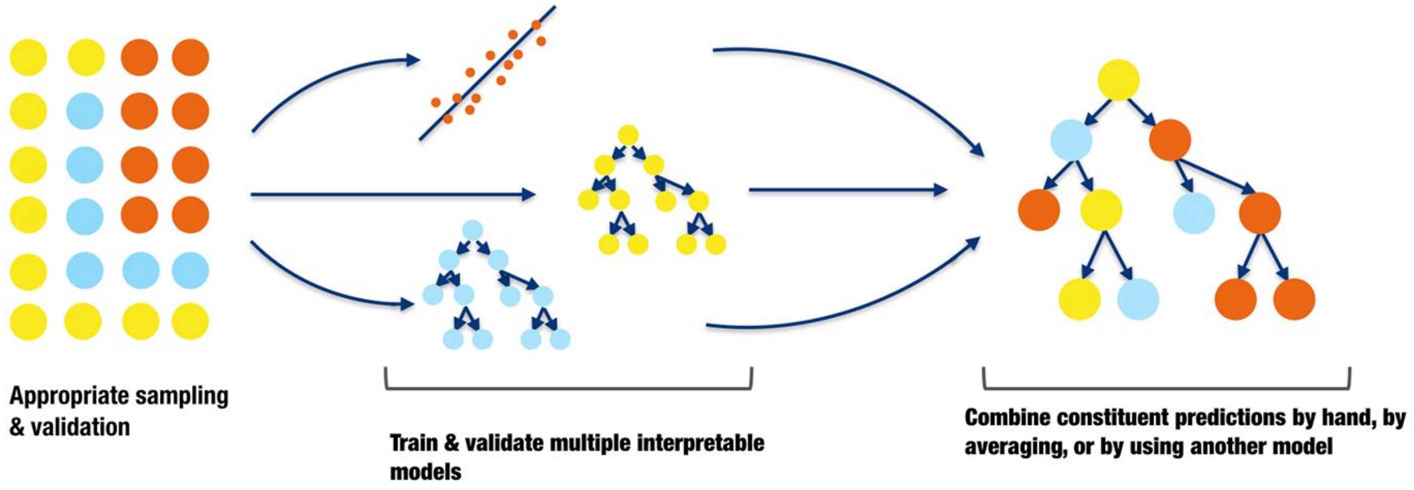
C: 1
gamma: 0.001
degree: 3
coef0: 0

[✓ Hyperplanes] Surface

Ensembles

Ensemble Methods

- Ensemble methods are models composed of multiple weaker models that are independently trained and whose predictions are combined in some way to make the overall prediction.
- Much effort is put into what types of weak learners to combine and the ways in which to combine them.
- This is a very powerful class of techniques and as such is very popular.


Appropriate sampling & validation
Train & validate multiple interpretable models
Combine constituent predictions by hand, by averaging, or by using another model

Common Types of Ensemble Methods

Boosting

- Each new model influenced by previously built models
- Weight model contribution by performance
- Reduces variance and increases accuracy
- Can be used with any loss function
- Sensitive to outliers

Voting

- Can combine conceptually different machine learning classifiers.
- Use voting or averaging to predict the class labels.
- Useful for a set of equally well-performing models.
- Can balance out individual weaknesses of voting models..

Bagging

- Individual models built separately, then combined.
- Equal weight given to all models
- Reduces variance and increases accuracy
- Robust against outliers and noise

Stacking

- Individual, diverse models trained on full dataset
- Meta-classifier fitted based on meta-features of those models
- This meta-learner learns the optimal combination of the component layers.

Boosting

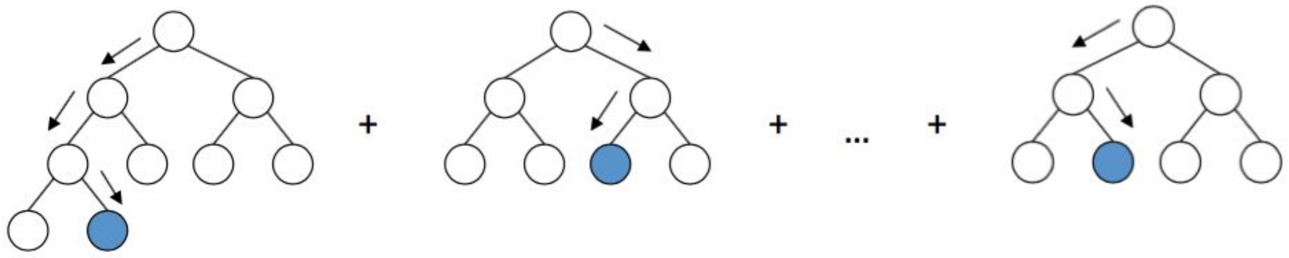
Gradient Boosting

- Traditional tree-based methods allow us to scale complexity with increasingly deep trees and more complex branching, but have a tendency to overfit to the training data.
- Boosting is an additive training technique for iteratively ensembling weak models into stronger ones.
- Gradient boosting enables us to build complex models using ensembles of shallow trees, which individually have low variance and high bias, but together can incrementally decrease bias via gradient descent.
- Gradient boosting models are also invariant to scaling inputs, meaning that they do not require careful feature normalization, as do some models (like k-nearest neighbors and support vector machines).

Gradient Boosting

Algorithm:

- Fit preliminary, naive model
- Compute error.
- Use error to train the next model.
- Continue.



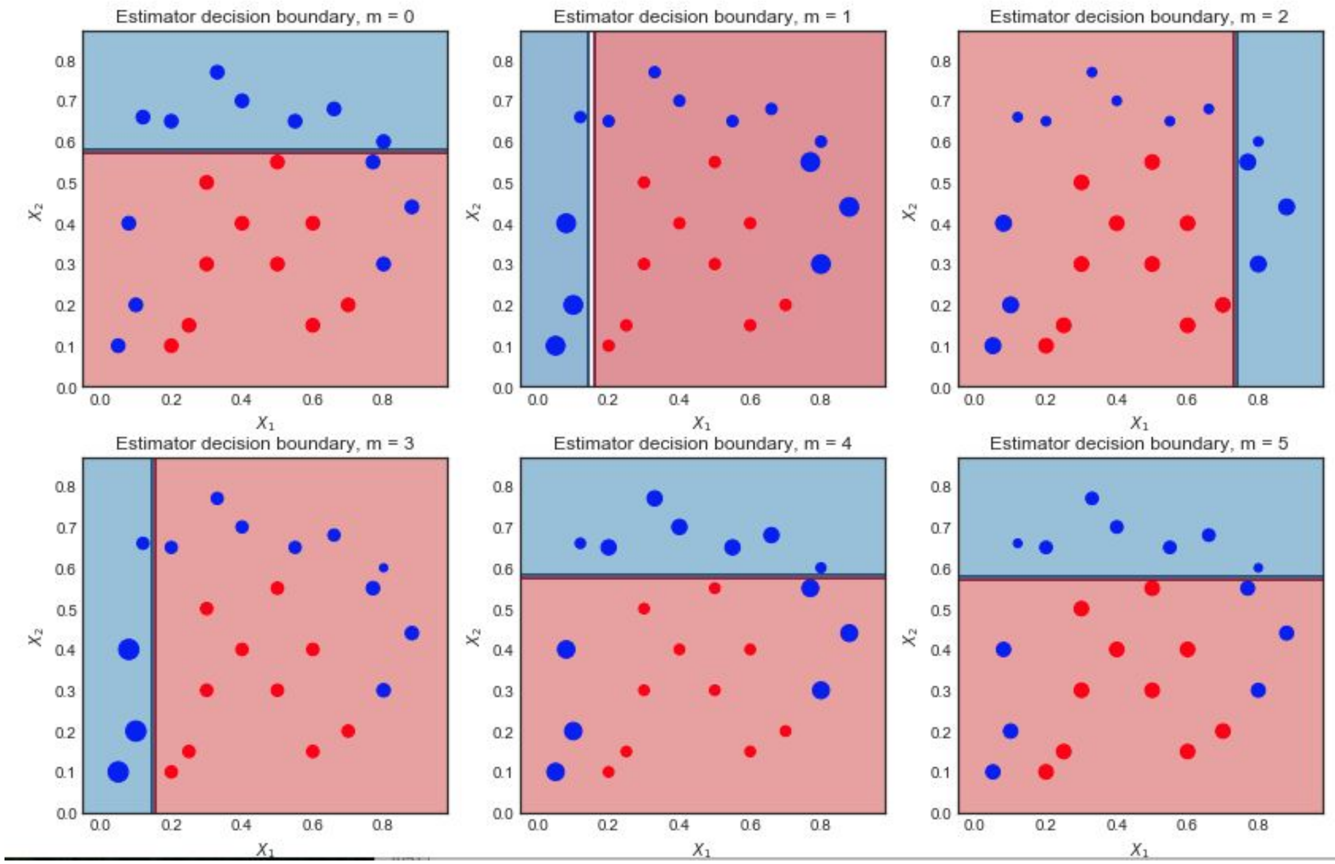
In this way, the algorithm aims to optimize the loss function over function space by iteratively fitting models that point in the negative gradient direction.

Gradient Boosting in Scikit-Learn

```
from sklearn.ensemble import GradientBoostingClassifier as gbc
from sklearn.model_selection import train_test_split as tts


X = data.features
y = data.target


X_train, y_train, X_test, y_test = tts(X, y, test_size=0.2)


loss = ‘deviance’ # specify the desired loss function


booster = gbc(loss=loss)
booster.fit(X_train, y_train)
booster.predict(X_test)
```

Adaboost

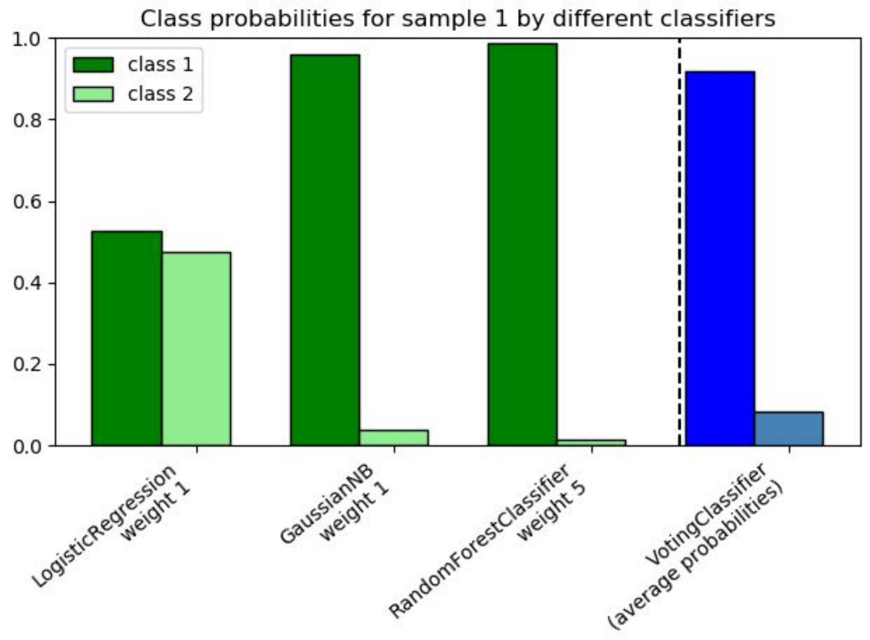
Adaboost is similar to Gradient Boosting, beginning by fitting a weak estimator on the data. It then fits additional copies of the estimator on the same dataset but adjusts the weights of incorrectly labeled instances so that subsequent iterations will focus on the instances that are harder to predict.



Voting

Voting Classifier

- Classifiers are initialized (but not fitted) and used to initialize a classifier meta-estimator.
- User supplies weights for each classifier to specify how much it should contribute overall.
- In hard voting, we predict the final class label as the class label that has been predicted most frequently.
- In soft voting, we predict the class labels by averaging the class probabilities.



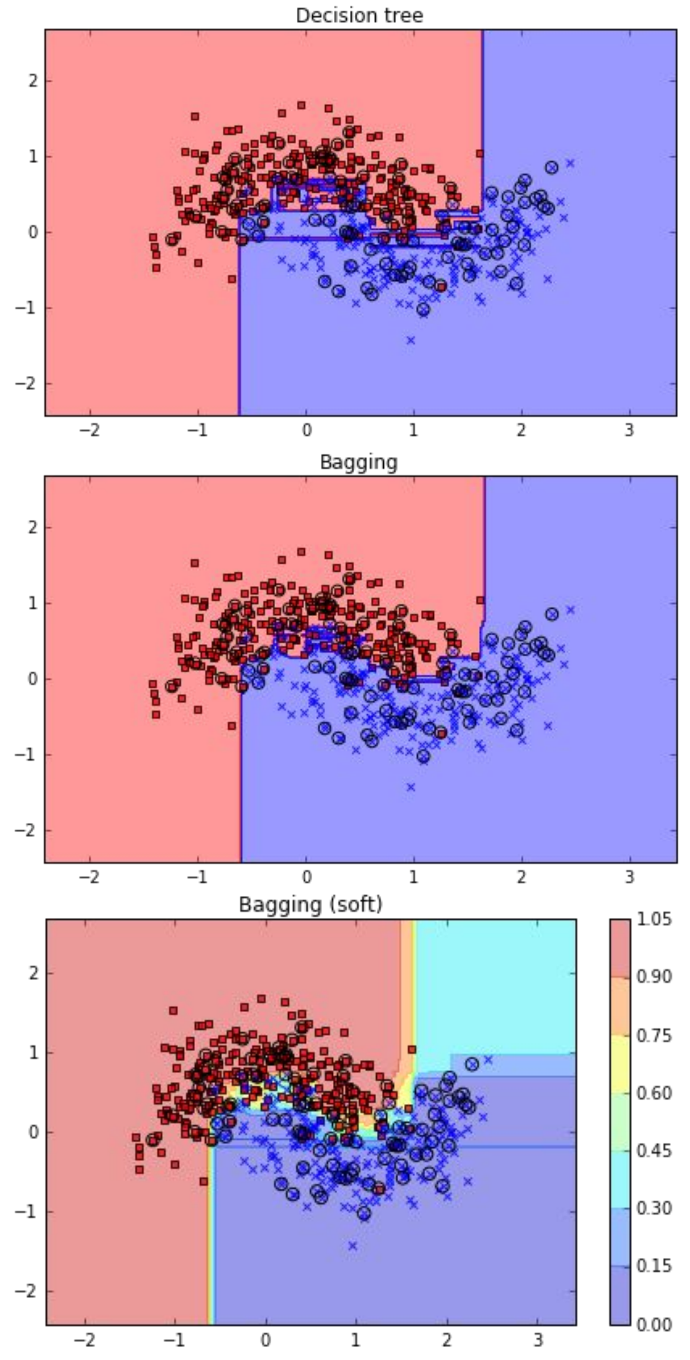
Voting Classifier in Scikit-Learn

```python
from sklearn.linear_model import LogisticRegression
from sklearn.naive_bayes import GaussianNB
from sklearn.ensemble import RandomForestClassifier, VotingClassifier



X = data.features
y = data.target


lr = LogisticRegression(
    solver='lbfgs', multi_class='multinomial', random_state=1
)
rf = RandomForestClassifier(n_estimators=50, random_state=1)
gnb = GaussianNB()

voter = VotingClassifier(
    estimators=[('lr', lr), ('rf', rf), ('gnb', gnb)], voting='hard' # or ‘soft’
).fit(X, y)
```

Bagging

Bagging

- Meta-estimator where base-learners are trained over slightly different training sets. Model aggregates individual predictions (either by voting or by averaging) to form a final prediction.
- Can be used as a way to reduce variance by introducing randomization.
- More robust to noise and outliers.
- However, model bias can’t be reduced, so we usually use classifiers with low bias (decision trees or nonlinear SVMs) as the base-learners in bagging.



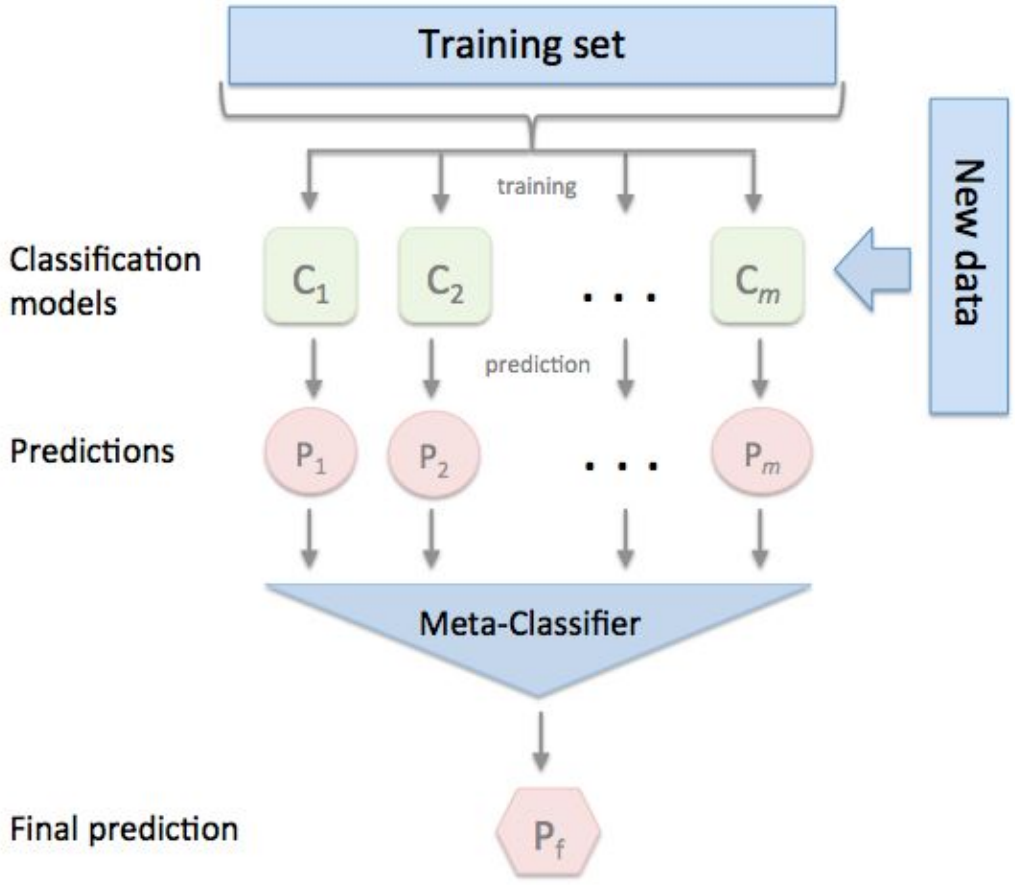
Bagging Classifier in Scikit-Learn

```
from sklearn.ensemble import BaggingClassifier
from sklearn.tree import DecisionTreeClassifier



X = data.features

y = data.target

cart = DecisionTreeClassifier()
num_trees = 100

model = BaggingClassifier(
    base_estimator=cart, n_estimators=num_trees
)
```

Stacking

Stacking Classifier

- Stacking is an ensemble learning technique to combine multiple classification models via a meta-classifier.
- Individual models are trained on the full training set; meta- classifier fitted based on outputs (meta-features) of those models.
- Can be trained on predicted class labels or probabilities.



Stacking Classifier with Scikit-Learn and Mlxtend

```python
from sklearn.linear_model import LogisticRegression
from sklearn.neighbors import KNeighborsClassifier
from sklearn.naive_bayes import GaussianNB
from sklearn.ensemble import RandomForestClassifier
from mlxtend.classifier import StackingClassifier

X = data.features

y = data.target


knn = KNeighborsClassifier(n_neighbors=1)

rf = RandomForestClassifier(random_state=1)

gnb = GaussianNB()

lr = LogisticRegression()

stack = StackingClassifier(

    classifiers=[knn, rf, gnb], meta_classifier=lr

)
```