Classification

Although you have already been introduced to classification via logistic regression, we will develop the theory of classification in general in greater detail.

We will begin with some theoretical exposition and then move into a series of exercises with different classification techniques.

Overview

The task of *classification* is fundamental to the history and development of machine learning because the simplest machine learning problem possible is binary classification: determining whether data falls into one class or the other.

Generative vs. discriminative methods

The novice data scientist can easily fall into the trap of considering different techniques to be more-or-less disconnected and unrelated, as if machine learning were a "grab bag" of models and algorithms, each with its own odd little quirks and nuances. This is not the case! There is **structure** in this world.

Of particular importance is the distinction between **generative vs. discriminative** approaches. A generative approach models the way in which the observed data were generated in order to perform classification on new data, whereas a discriminative approach doesn't care about how the data was generated: it is *purely* concerned with the problem of classifying new data.

To make this precise, suppose that we wish to predict the label y from the training data x. In a *discriminative* approach, we are concerned with evaluating $y^\star = \arg\max_y P(y \mid x)$, that is, finding the most likely class label $y^\star$ given the data x. The only concern is how we can distinguish one class from another using x.

However, we can use Bayes' rule to rephrase our problem. Since $P(y \mid x) = P(x \mid y)P(y)/P(x)$, we have $y^\star = \arg\max_y P(x \mid y)P(y)/P(x)$. However, the value of y which maximizes $P(x \mid y)P(y)/P(x)$ is the same as the value of y which maximizes $P(x \mid y)P(y)$, because $P(x)$ does not change with y.

As such, we can reformulate the problem as the evaluation of $y^\star = \arg\max_y P(x \mid y)P(y)$. This corresponds to a *generative* model, because we are directly estimating the value of $P(y)$, that is, explicitly modeling the distribution of each class in order to perform classification.

Philosophical importance

Instead of just being two equivalent formulations of the same underlying problem, the distinction between generative and discriminative modeling is very philosophically fundamental. To quote V. N. Vapnik:

> The philosophy of science has two different points of view on the goals and the results of scientific activities.
>
> 1. There is a group of philosophers who believe that the results of scientific discovery are the real laws that exist in nature. These philosophers are called the *realists*.
> 2. There is another group of philosophers who believe the laws that are discovered by scientists are just an instrument to make a good prediction. The discovered laws can be very different from the ones that exist in Nature. These philosophers are called the *instrumentalists*.
>
> The two types of approximations defined by classical discriminant analysis (using the generative model of data) and by statistical learning theory (using the function that explains the data best) reflect the positions of realists and instrumentalists in our simple model of the philosophy of generalization, the pattern recognition model. Later we will see that the position of philosophical instrumentalism played a crucial role in the success that pattern recognition technology has achieved.

Considering classification alone, we can state this in a more precise fashion. Suppose that we have a black box, $\mathcal{B}$, which when given an input vector $\mathbf{x}_i$ returns an output $y_i = \{-1, +1\}$. Given the training data $(y_i, \mathbf{x}_i)$, $i = 1, \ldots, n$, the task of *binary classification* is to find a function which best approximates whatever rule the black box $\mathcal{B}$ is internally using to perform the operation $\mathbf{x}_i \mapsto y_i$.

There are two different ideas of what a *good approximation* means:

1. A good approximation of $\mathcal{B}$ is a function that is *similar in function space*, like how $\sin 2x$ is similar to $\sin 2.1x$, to the function which $\mathcal{B}$ uses to map $\mathbf{x}_i \mapsto y_i$.
2. A good approximation of $\mathcal{B}$ is a function that is *numerically similar* to the function which $\mathcal{B}$ uses, giving approximately the same error rate of classification as $\mathcal{B}$ itself.

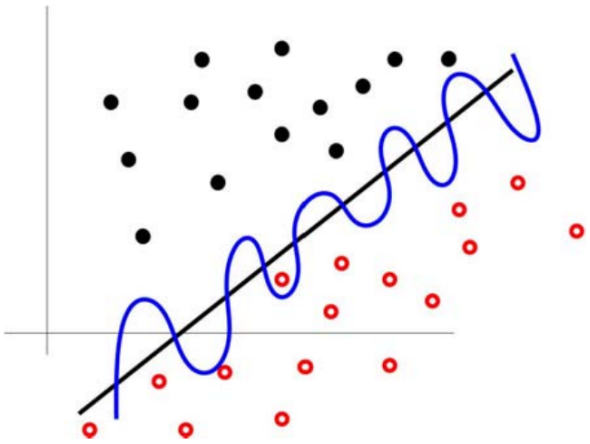

Figure 1: The linear line is a good approximation in the first sense; the curvy polynomial is a good approximation in the second sense. (It does admittedly misclassify a single point.)

The first definition is concerned with approximating the *true function*. The second definition is only concerned with *minimizing the error rate*.

Practical importance

This distinction is not just of philosophical importance; it carries practical significance:

> [I]ll-posed problems can occur when one tries to estimate *unknown reasons from observed consequences*.

That is to say, problems that are very computationally difficult to solve – where a small change in the observed data can lead to (discontinuously) large changes in the solution – are very often precisely the problems which occur when we try to use a *generative* model. If the solver possesses "very strong prior knowledge about the solution", or if the solver makes very strong prior *assumptions*, then generative models become much more tractable, but this is not always feasible.

Vapnik writes:

> Keeping in mind the structure of ill-posed problems our problem of finding the [solution for $\mathcal{B}$] can be split into two stages:
>
> 1. Among a given set of admissible functions find a subset of functions that provides an expected loss that is close to the minimal one.
> 2. Among functions that provide a small expected loss find one that is close to the $\mathcal{B}$ function.

The first stage does not lead to an ill-posed problem, but the second stage might (if the corresponding operator is unstable).

The realist view requires solving both stages of the problem, while the instrumentalist view requires solving only the first stage and choosing for prediction any function that belongs to the set of functions obtained.

The shift from the realist view to the instrumentalist view marked a significant advance in the development of machine learning.

Evaluation of classifier performance

- Read Nina Zumel's primer on the evaluation of classifier performance.

Preliminary steps

We will mainly be using simulated data for the exercises below. First, we must write functions which generate our simulated data.

For simplicity, we will restrict attention to the 2-dimensional unit square ($x \in [0,1]$, $y \in [0,1]$).

- Write a function `lin_pair(m, b, label)` that takes in integers `m`, `b`, and `label` and returns two numerics `c(x, y)` satisfying the following criteria:
  - Both `x` and `y` should be between 0 and 1.
  - If `label` is set to 1, then `y` must be greater than `m*x + b`. Conversely, if `label` is set to -1, then `y` must be less than `m*x + b`.

You can think of the function `lin_pair(m, b, label)` as picking a point in the unit square uniformly randomly from the region falling above or below the line $y = mx + b$.

- Plot the points you get from running `lin_pair()` many times with the same input parameters to visually verify that your function works correctly.
- Write a function `quad_pair(a, b, c, label)` which does the same thing except for the quadratic parabola $y = a(x-b)^2 + c$. Verify that it works by plotting the results.
- Write a function `mvnorm_pair(mu, cov)` which returns a point sampled from a multivariate normal distribution. `mu` should be a vector containing the mean for each dimension and `cov` should be a 2-by-2 covariance matrix (where `cov[1, 1]` is the variance in dimension 1, `cov[2, 2]` is the variance in dimension 2, and `cov[1, 2] = cov[2, 1]` are the covariance

of the two dimensions). You may find `mvrnorm()` from the `MASS` package useful.

  - Note that the `select()` function from `MASS` will mask the `select()` function from `dplyr` if you load `MASS` after `dplyr`. You can access the `dplyr` `select()` function with `dplyr::select()`.

In the exercises to follow, we will restrict consideration almost entirely to the problem of **binary classification** for simplicity's sake. Multiclass classification is possible with many of the following methods, but it is generally wise to work with simple examples first to gain intuition and understanding.

k-Nearest Neighbors

We will only briefly mention that k-Nearest Neighbors classification works in precisely the way one would expect it to work: for any given point, the algorithm looks at the nearest k points and finds the most common class among those k points. That class is the classification result.

Although k-NN works well when there is "enough" data, prediction is slow and there are problems with high-dimensional data. In practice, one can improve predictive performance by considering points farther away.

k-NN can be used in R for classification via the `knn()` function in the `class` library.

Discriminant methods

Ronald Fisher's method of *discriminant analysis* is one of the earliest classification methods.

The idea behind discriminant analysis is that we posit a *generative model* for our data, where each of the two classes is generated from a multivariate normal distribution, and attempt to estimate the parameters of those distributions with the training data. With the parameters, we construct a *discriminant function* which gives us a decision boundary separating the two classes (hopefully).

With p-dimensional data, for each of the p dimensions we must estimate the *mean* of both distributions. In addition, we have *covariance* terms to estimate: for each distribution, we must estimate the covariance of the first dimension with the second, the covariance of the first with the third, the covariance of the second with the third, and so on and so forth.

- In p-dimensional space, how many parameters do we have to estimate for a generative model positing that our two classes of data are generated by multivariate normal distributions?

Overview of derivation

The most important information to take away from the derivations of discriminant analysis is the *set of assumptions* which underlies each method.

For binary classification, we first assume that the data from each class is drawn from a multivariate normal distribution. From this assumption alone, we can predict class membership from looking at whether or not the logarithm of the ratio of the likelihoods exceeds a certain threshold.

After simplification, we reduce to the following criterion for membership in class $+1$ (as opposed to class -1) where $\boldsymbol{mu}$, Σ, and T represent distribution means, covariance matrices, and the threshold value:

$$\left[\left(\mathbf{x} - \boldsymbol{\mu}_{+1} \right)^{\mathsf{T}} \Sigma_{+1}^{-1} \left(\mathbf{x} - \boldsymbol{\mu}_{+1} \right)^{\mathsf{T}} - \left(\mathbf{x} - \boldsymbol{\mu}_{-1} \right) \Sigma_{-1}^{-1} \left(\mathbf{x} - \boldsymbol{\mu}_{-1} \right) \right] + \left[\ln \frac{\mid \Sigma_{+1} \mid}{\mid \Sigma_{-1} \mid} \right] > T.$$

There are quadratic terms on the left-hand side of the inequality, with entries of $\mathbf{x}$ being multiplied with other entries of $\mathbf{x}$, so this is called **quadratic discriminant analysis**.

We can further simplify this expression by making the assumption that the two distributions have the same covariance matrix. In this case, the above criterion reduces to the form

$$\mathbf{w} \cdot \mathbf{x} > c$$

for a vector of weights $\mathbf{w}$ and a constant threshold c which can be calculated from the parameters of the multivariate normal distributions.

Note that the left-hand side of the inequality is linear with respect to $\mathbf{x}$. As such, we call this **linear discriminant analysis**.

Feel free to skip the following section on the mathematical derivation of QDA and LDA if you find it too technically demanding.

Mathematical derivation

Suppose that we have p-dimensional data points $\mathbf{x}_i$, each with a binary class label $y_i \in \{-1, +1\}$. We make the assumption that the probability distribution for each class, $P(\mathbf{x} \mid y = -1)$ and $P(\mathbf{x} \mid y = +1)$, are both normally distributed with means and covariances $(\boldsymbol{\mu}_{-1}, \Sigma_{-1})$ and $(\boldsymbol{\mu}_{-1}, \Sigma_{+1})$ respectively.

With this assumption alone, we are ready to look at the the *log-likelihood* of class membership. Specifically, we can predict class membership to be $+1$ if the log of the ratio of the likelihoods is above some threshold T, that is,

$$\log \frac{P(\mathbf{x} \mid y = +1)}{P(\mathbf{x} \mid y = -1)} > T.$$

Since we assumed that both probability densities are multivariate normal distributions, they have the functional form

$$P(\mathbf{x} \mid y = +1) = \frac{1}{(2\pi)^{p/2} \mid \Sigma_{+1} \mid^{1/2}} \exp\left(-\frac{1}{2} (\mathbf{x} - \boldsymbol{\mu}_{+1})^{\mathsf{T}} \Sigma_{+1}^{-1} (\mathbf{x} - \boldsymbol{\mu}_{+1})\right).$$

As such, after taking the logs and simplifying, we have the following criterion for membership in class $+1$:

$$\left[(\mathbf{x} - \boldsymbol{\mu}_{+1})^{\mathsf{T}} \Sigma_{+1}^{-1} (\mathbf{x} - \boldsymbol{\mu}_{+1})^{\mathsf{T}} - (\mathbf{x} - \boldsymbol{\mu}_{-1}) \Sigma_{-1}^{-1} (\mathbf{x} - \boldsymbol{\mu}_{-1})\right] + \left[\ln \frac{\mid \Sigma_{+1} \mid}{\mid \Sigma_{-1} \mid}\right] > T.$$

As it stands, the classification of points with the above criterion is called **quadratic discriminant analysis** (QDA), because of the cross terms in the multiplication which lead to quadric decision boundaries.

We can *further* assume that the covariances of the two distributions are equal,[1] so $\Sigma = \Sigma_{-1} = \Sigma_{+1}$. After further simplification, we obtain the criterion

$$\mathbf{w} \cdot \mathbf{x} > c,$$

where

$$\mathbf{w} = \Sigma^{-1} (\boldsymbol{\mu}_{-1} - \boldsymbol{\mu}_{+1})$$

and

$$c = \frac{1}{2} \left(T - \boldsymbol{\mu}_{+1}^{\mathsf{T}} \Sigma^{-1} \boldsymbol{\mu}_{+1}^{\mathsf{T}} + \boldsymbol{\mu}_{-1}^{\mathsf{T}} \Sigma^{-1} \boldsymbol{\mu}_{-1}^{\mathsf{T}}\right).$$

This time, our solution is *linear* in $\mathbf{x}$, and accordingly is named **linear discriminant analysis** (LDA).

Note that the criterion $\mathbf{w} \cdot \mathbf{x} > c$ is equivalent to specifying that $\mathbf{x}$ must be on a particular side of a *hyperplane* parameterized by $\mathbf{w}$ and c. We can see this by realizing that the dot product $\mathbf{w} \cdot \mathbf{x}$ is proportional to the projection of vector $\mathbf{x}$ onto vector \$$\mathbf{w}$. As such, if the dot product is positive, then $\mathbf{x}$ is on one side of a

[1] We also assume that the covariance matrices have full rank.

hyperplane orthogonal to $\mathbf{w}$, and if it negative, than $\mathbf{x}$ is on the other side of the hyperplane.

Effectively, LDA finds a *separating hyperplane* which divides the two classes of training data, with slope and position set by $\mathbf{w}$ and c.

Estimating parameters

The QDA and LDA criterion depend on knowing the covariances Σ_{-1} and Σ_{+1} (or just Σ) as well as the means $\boldsymbol{\mu}_{-1}$ and $\boldsymbol{\mu}_{+1}$ of the two multivariate distributions which we assume generate our data.

In practice, we do not actually know the true values of these parameters, but we can estimate them using the sample mean and covariance of our training data.

There is no general rule for choosing the value of the threshold T. If the data are projected onto $\mathbf{w}$, the one-dimensional distribution of projections can be analyzed to choose the threshold which seems to best separate the data.

Intuition

We will consider just LDA for simplicity, but the intuition here can be extended to QDA as well.

We know that LDA finds a hyperplane which is supposed to separate the two classes of data. However, what can we say about the properties of this hyperplane?

Since LDA works with the assumption that the two distributions (one corresponding to each class) are multivariate normal distributions, it effectively finds a separating hyperplane which does the following:

1. Maximizes the between-class variation, and
2. Minimizes the within-class variation.

"Variation" in the above refers to the notion of Mahalanobis distance, which measures the distance between a point and distribution.

Because of the generative nature of the model, LDA works well when the underlying data really are drawn from multivariate normal distributions. Moreover, since the *between-class variation* is implicitly minimized by LDA, the two classes are well separated. However, if the means of the distributions are close together, LDA will struggle to find a good separating hyperplane.

Moreover, QDA needs to estimate many more parameters than LDA, so it often requires more data. If the covariances of the underlying distributions are very dissimilar, then QDA will eventually perform better than LDA; however, in

cases where the covariance matrices are relatively similar, it is possible that LDA will perform better than QDA up until a very high sample size is achieved.

Discriminant analysis in R

LDA and QDA are implemented in R as `lda()` and `qda()` in the `MASS` package. The resulting fit objects can be directly printed in the console for an overview of the calculated parameters, and class predictions can be made with `predict()` as usual. (For a prediction object, the actual class level assignments will be stored in `$class`.)

We'll first look at simulated data, where we know what the underlying distributions of the two classes look like.

- Generate 100 data points in the unit square. 50 of them should be drawn from a multivariate normal distribution centered at $(0.25, 0.25)$, and 50 of them should be drawn from a multivariate normal distribution centered at $(0.75, 0.75)$. Their covariance matrices should be identical, with the variance in each direction equal to 0.2 and the covariance between the dimensions equal to 0.1. Plot the data.
- Generate a vector with class labels for the data (-1 or +1 depending on which distribution a point is drawn from). Run both LDA and QDA on the data. Look at the estimates of the sample mean and evaluate the accuracy of the predictions.
- Generate equal amounts of data from two multivariate normal distributions in the unit square with extremely different covariance matrices. Run both LDA and QDA on the data. How does the performance of each algorithm in classifying the training data vary as you change the size of the dataset?
- Generate 200 data points, 100 of which fall above the line $y = 1.5x + 0.2$ and 100 of which fall below the line $y = 1.5x + 0.05$. Plot the data.
- Run both LDA and QDA on the generated data and evaluate their performance. Do the same for smaller and larger datasets generated identically.

The `partimat()` function from the `klaR` library can be used to graph the decision boundary for LDA or QDA in a 2-dimensional setting.

- Use `partimat()` to view the results of LDA or QDA on each of your simulated datasets. Interpret the differences.

Next, we'll run our discriminant analysis methods on some real datasets, starting with the aggregated speed dating dataset.

- Load the aggregated speed dating dataset and restrict to self-rated activity participation and the gender of the person rated.

- For each gender, compute the means and covariance matrices of the associated subset of the dataset. Compare them and predict the relative performance of LDA and QDA on this dataset.
- Create random subsets of the data that are 1%, 2%, 5%, 10%, 25%, 50%, 75%, and 100% of the size of the original dataset.
- Run LDA and QDA on each of the variably sized subsets of data, classifying gender with respect to self-rated activity participation. Plot their classification accuracy on the training data as a function of the proportion of the dataset used. Interpret the results.

The theoretical framework of LDA and QDA can easily be extended to *multiclass classification*.

- Install the `rattle` package, run `data(wine, package='rattle')`, and set `df_wine = wine`. This dataset conists of chemical measurements of wine from three different cultivars, with the cultivar reported in the Type column. Run LDA to predict wine type from the other variables, look at the group means and coefficients of the linear discriminants, and interpret the results.
- To see how each of the two linear discriminant separates the groups, call `predict()` on the LDA fit itself to generate values of the discriminant functions for the training data. Read the `predict.lda()` documentation to figure out how to access the values of the discriminant functions and pass either of the two sets of discriminant values in to `ldahist()` along with the column of group assignments.
- Plot the values of the two discriminant functions against each other. Interpret the results.

Read the following post about visualizing the results of LDA:

- Compute and graph the LDA decision boundary

Extensions of discriminant analysis

A variant of LDA is occasionally used for dimensionality reduction.

Logistic regression

You already have substantial experience applying logistic regression in a computational sense; as such, there will be no R exercises in this section. This section has two goals: first, to demonstrate further how logistic regression fits into the framework of classification and machine learning as a whole; second, to provide more refinement to your intuitions about how logistic regression works.

Like discriminant analysis, logistic regression is a *generative* method, which makes certain assumptions about the distributions from which the data are drawn. Instead of beginning with linear regression as motivation and developing logistic regression, we will instead begin from the standpoint of trying to answer the question: "How can we improve linear discriminant analysis?" Beautifully, we will see that the answers coincide.

For a review of the standard exposition of logistic regression, see Chapter 11 in Advanced Data Analysis from an Elementary Point of View.

Mathematical derivation

The difficulty of the following mathematical exposition will vary from paragraph to paragraph. If you get caught up on understanding a specific statement, it's better to move on and return to it later.

I had the pleasure of learning this derivation from *Statistical Methods for Pharmaceutical Research Planning* by Bergman and Gittins (1985).

The motivation behind the following derivation of logistic regression is the consideration of instances where the two classes in binary classification are *not* sampled from multivariate normal distributions with equal covariances. That is, we would like to relax the assumptions of LDA to allow for probability densities that are not normal.

In particular, instead of assuming that $P(\mathbf{x} \mid y = i) \propto \mathcal{N}(\boldsymbol{\mu}_i, \Sigma)$ for $i \in \{-1, +1\}$, where $\mathcal{N}(\boldsymbol{\mu}_i, \Sigma)$ denotes a multivariate normal distribution with mean $\boldsymbol{\mu}_i$ and covariance matrix Σ_i, we posit that

$$P(\mathbf{x} \mid y = i) \propto \mathcal{N}(\boldsymbol{\mu}_i, \Sigma)\phi(\mathbf{x}).$$

We relax our assumptions by saying that each of the two classes is drawn from a distribution equal to the *product* of a multivariate normal distribution with some unknown function $\phi(\mathbf{x})$.

The *discriminant functions* – that is, the essential inequalities which define QDA and LDA – stay the same! This is because in their derivations we take the *ratio* of the probability densities for the two classes, so the function $\phi(\mathbf{x})$ cancels and disappears. However, this is *not* a free lunch. A more subtle problem arises: the sample means and covariances, which we originally used as approximations to the parameters of the underlying multivariate normal distributions, are no longer the maximum likelihood estimators for those parameters in this more complex situation where we multiply the multivariate normal distribution by $\phi(\mathbf{x})$ (unless $\phi(\mathbf{x}) = 1$).

We will pause for a brief interlude in order to consider the likelihood function $\mathcal{L}$ directly. The function $\mathcal{L}(\mathbf{x}, \boldsymbol{\theta})$ with respect to a collection of parameters $\boldsymbol{\theta}$ is

equal to *the probability of observing the data* $\boldsymbol{x}$ *given the parameter values* $\boldsymbol{\theta}$ *for the underlying distributions from which the data are generated*. To do *maximum likelihood estimation* of the parameters, we find the combination of parameters which maximizes $\mathcal{L}$.

Now, let us take write the training data as $T = T_{-1} \cup T_{+1}$, that is, the union of the classes -1 and $+1$. The likelihood function for the generative model we are considering is given by

$$\mathcal{L} = \prod_{\mathbf{x} \in T_{-1}} \left[P(\mathbf{x} \mid y = -1) P(y = -1) \right] \prod_{\mathbf{x} \in T_{+1}} \left[P(\mathbf{x} \mid y = +1) P(y = +1) \right].$$

Again, we can intuitively think of this as being the *probability of observing the data which we observed given the parameters of the underlying distributions*, where the parameters are implicitly encoded in the probabilities $P(y = -1)$ and $P(y = +1)$.

Expanding out the probabilities, we obtain

$$\mathcal{L} = \prod_{\mathbf{x} \in T_{-1}} \left[\gamma_{-1} P(y = -1) \mathcal{N}(\boldsymbol{\mu}_{-1}, \Sigma) \phi(\mathbf{x}) \right] \prod_{\mathbf{x} \in T_{+1}} \left[P(y = +1) \mathcal{N}(\boldsymbol{\mu}_{+1}, \Sigma) \phi(\mathbf{x}) \right]$$

by substituting in the explicit expressions for the distributions of each class, where the constants γ_{-1} and γ_{+1} are chosen so that the distributions properly integrate to 1. To extend the method of discriminant analysis even further, in the case where the function $\phi(\mathbf{x})$ is known we can maximize the likelihood function above and obtain the parameters of the multivariate normal distributions. However, the difficulty of this will vary depending on the form of $\phi(\mathbf{x})$.

We have more or less run into the limits of how far discriminant analysis can take us without using substantially more sophisticated methods. However, we can develop the method of *logistic classification* by first rewriting the likelihood function as follows:

$$\mathcal{L} = \prod_{\mathbf{x} \in T} P(\mathbf{x}) \left[\prod_{\mathbf{x} \in T_{-1}} P(y = -1 \mid \mathbf{x}) \prod_{\mathbf{x} \in T_{+1}} P(y = +1 \mid \mathbf{x}) \right].$$

What we have done here is *reverse the direction of the conditional dependences*. Instead of conditioning on class membership, we now look at the probability of class membership conditional on the location of each data point.

Now, due to laws of probability, we have the relation

$$P(\mathbf{x}) = P(\mathbf{x} \mid y = -1) P(y = -1) + P(\mathbf{x} \mid y = +1) P(y = +1).$$

Moreover, from Bayes' theorem, we have

$$P(y = +1 \mid \mathbf{x}) = \frac{P(\mathbf{x} \mid y = +1)P(y = +1)}{P(\mathbf{x} \mid y = -1)P(y = -1) + P(\mathbf{x} \mid y = +1)P(y = +1)}.$$

Substituting in the explicit expression for the probability densities from which each class is drawn and simplifying, we arrive at

$$P(y = +1 \mid \mathbf{x}) = \frac{\exp\left(\mathbf{x}^\intercal \mathbf{c} + d\right)}{1 + \exp\left(\mathbf{x}^\intercal \mathbf{c} + d\right)}$$

where $\mathbf{c} = \Sigma^{-1}\left(\boldsymbol{\mu}_{+1} - \boldsymbol{\mu}_{-1}\right)$ and

$$d = -\frac{1}{2}\left(\boldsymbol{\mu}_{+1} - \boldsymbol{\mu}_{-1}\right)^\intercal \Sigma^{-1}\left(\boldsymbol{\mu}_{+1} + \boldsymbol{\mu}_{-1}\right) + \log\frac{P(y = +1)\gamma_{+1}}{P(y = -1)\gamma_{-1}}.$$

Substituting the expressions for $P(\mathbf{x})$ and $P(y = +1 \mid \mathbf{x})$ into the likelihood function, we obtain

$$\mathcal{L} = l_{+1}l_{-1}\prod_{\mathbf{x}\in T}\left\{\phi(\mathbf{x})\left[\gamma_{+1}P(y = +1)\exp f(+1) + \gamma_{-1}P(y = -1)\exp f(-1)\right]\right\}$$

where

$$f(i) = -\frac{1}{2}\left(\mathbf{x} - \boldsymbol{\mu}_i\right)^\intercal \Sigma^{-1}\left(\mathbf{x} - \boldsymbol{\mu}_i\right)$$

and

$$l_{+1} = \prod_{\mathbf{x}\in T_{+1}} \frac{\exp\left(\mathbf{x}^\intercal \mathbf{c} + d\right)}{1 + \exp\left(\mathbf{x}^\intercal \mathbf{c} + d\right)}$$

and

$$l_{-1} = \prod_{\mathbf{x}\in T_{-1}} \frac{1}{1 + \exp\left(\mathbf{x}^\intercal \mathbf{c} + d\right)}.$$

We can maximize this expression with respect to $\phi(\mathbf{x})$ as well as the means and shared covariance of the multivariate normal distributions. It is argued in Day and Kerridge (1967), *A General Maximum Likelihood Discriminant*, that the likelihood $\mathcal{L}$ is primarily dependent on the values of $\mathbf{c}$ and d, being relatively

insensitive to the product over T in practice. As such, we obtain the following simplification for the *reduced likelihood*:

$$\mathcal{L}_{\text{red}} = l_{+1} l_{=1} = \prod_{\mathbf{x} \in T_{+1}} \frac{\exp\left(\mathbf{x}^\intercal \mathbf{c} + d\right)}{1 + \exp\left(\mathbf{x}^\intercal \mathbf{c} + d\right)} \prod_{\mathbf{x} \in T_{-1}} \frac{1}{1 + \exp\left(\mathbf{x}^\intercal \mathbf{c} + d\right)}.$$

Our task is much simpler now: to simply maximize this straightforward-seeming function with respect to the vector $\mathbf{c}$ and the constant d. Equivalently, we can recast the problem into an even simpler form by taking the *logarithm*, which turns products into sums:

$$\log \mathcal{L}_{\text{red}} = \sum_{\mathbf{x} \in T} -\log\left(1 + \exp\left(\mathbf{x}^\intercal \mathbf{x} + d\right)\right) + \sum_{\mathbf{x} \in T_{+1}} \left(\mathbf{x}^\intercal \mathbf{c} + d\right).$$

Looking at Shalizi's Advanced Data Analysis from an Elementary Point of View, equation 11.10 for the log-likelihood in his derivation of logistic regression is:

$$\mathcal{L}\left(\beta_0, \boldsymbol{\beta}\right) = \sum_{i=1}^{n} -\log\left(1 + e^{\beta_0 + \mathbf{x}_i \cdot \boldsymbol{\beta}}\right) + \sum_{i=1}^{n} y_i \left(\beta_0 + \mathbf{x}_i \cdot \boldsymbol{\beta}\right).$$

This is the same as our own log-likelihood $\log \mathcal{L}_{\text{red}}$*!* The only difference in the second sum can be explained by the fact that in Shalizi's derivation, we have $y_i \in 0, 1$ for the two classes, so in effect the sum is only over the values of $\beta_0 + x_i \cdot \beta$ for $x_i \in \{x_i \mid y_i = 1\}$.

Indeed, Bergman and Gittins themselves make the following remark:

> The logistic classification procedure may also be derived by a different line of argument. One could assume that the underlying population has a logistic density with unknown parameters $\mathbf{c}$ and d. This again yields the classification function $\mathbf{x}^\intercal \mathbf{c} + d$. This is of some interest, for it implies that classification on the basis of the model above is essentially the same as classification on the basis of the log-linear model with no interaction terms (see, *e.g.*, Bishop *et al.*, 1975).

For data in p-dimensional space, we only have $p + 1$ parameters to estimate! This can easily be done using a standard optimization technique such as the Newton-Raphson method. After estimating the parameters, we can then explicitly calculate the probability of class membership for any data point with

$$P(y = +1 \mid \mathbf{x}) = \frac{\exp\left(\mathbf{x}^\intercal \mathbf{c} + d\right)}{1 + \exp\left(\mathbf{x}^\intercal \mathbf{c} + d\right)}$$

and, of course, $P(y = -1 \mid \mathbf{x}) = 1 - P(y = +1 \mid \mathbf{x})$. Finally, we can classify a point $\mathbf{x}$ as belonging to group $+1$ if and only if $P(y = +1 \mid \mathbf{x}) > T$ for some threshold T depending on the costs of misclassification for each class.

Guaranteed linear separation

Interestingly, the method of logistic regression is guaranteed to return a hyperplane which perfectly separates the two classes in the training data if indeed it is possible to do so.

Indeed, suppose that $g(\mathbf{x}) = \mathbf{x}^\intercal \mathbf{c} + d$ be an arbitrary linear classification function, and assume without loss of generality that $g(\mathbf{x})$ classifies $\mathbf{x}$ as belonging to class $+1$ if and only if $g(\mathbf{x}) > 0$. Now, the corresponding reduced likelihood function is given by

$$\mathcal{L}_{\text{red}}(g(\mathbf{x})) = \prod_{\mathbf{x} \in T_{+1}} \frac{\exp g(\mathbf{x})}{1 + \exp g(\mathbf{x})} \prod_{\mathbf{x} \in T_{-1}} \frac{\exp -g(\mathbf{x})}{1 + \exp -g(\mathbf{x})},$$

since

$$\frac{\exp -g(\mathbf{x})}{1 + \exp -g(\mathbf{x})} = \frac{1}{1 + \exp g(\mathbf{x})}.$$

From the form of the product, we may see that each term takes on a value in $[1/2, 1]$ if the corresopnding point $\mathbf{x}$ is correctly classified and a value in $[0, 1/2]$ otherwise.

Now, assume that $g^\star(\mathbf{x})$ is a linear classification function which correctly classifies all $\mathbf{x} \in T$, that is to say, $g^\star(\mathbf{x}) > 0$ if $\mathbf{x} \in T_{+1}$ and $g^\star(\mathbf{x}) < 0$ if $\mathbf{x} \in T_{-1}$. If $g^\star(\mathbf{x})$ is a perfect classification function, then so is $kg^\star(\mathbf{x})$ for any constant $k > 0$, so

$$\lim_{k \to \infty} \mathcal{L}_{\text{red}}(kg^\star(\mathbf{x})) = 1.$$

Since the function $\mathcal{L}_{\text{red}}(kg^\star(\mathbf{x}))$ is continuous with respect to k, it must be the case that there exists some value $k = k^\star$ such that

$$\mathcal{L}_{\text{red}}(k^\star g^\star(\mathbf{x})) > \frac{1}{2}.$$

In other words, there exists a function $g'(\mathbf{x}) = k^\star g^\star(\mathbf{x})$ such that $\mathcal{L}_{\text{red}}(g'(\mathbf{x})) > 1/2$. Therefore, as we maximize $\mathcal{L}_{\text{red}}$, we will eventually arrive at values of the parameters $\mathbf{c}$ and d which correspond to the function $g'(\mathbf{x})$, which by construction fully separates the two classes. (Indeed, if $g'(\mathbf{x})$ were *not* fully separating,

then one of the terms in the product $\mathcal{L}_{\text{red}}(g'(\mathbf{x}))$ would be below $1/2$, and so the entire product would also be below $1/2$.)

Closing notes

Given the popularity of logistic regression today, it is amusing that Bergman and Gittins say:

> Apparently, this model has yet to receive attention in the pharmaceutical literature. In view of its promise, it certainly warrants some consideration.

20+ years later, it is part of the standard repertoire of any data scientist. Sadly, the connection between linear discriminant analysis and logistic regression is known to very few of them. It is only after learning this derivation that I myself began to feel much more comfortable with classification as a whole, because it allowed me to view the varied techniques as part of the same cohesive whole, like one single flowing stream.

Perceptrons

We will proceed to considering the **perceptron algorithm**, one of the first and simplest purely predictive classification techniques developed. Unlike the methods of discriminant analysis and logistic regression, which are *generative* methods, the perceptron simply aims to find the best hyperplane possible for class separation.

- Read about the history of the perceptron on Wikipedia.

For labeled training data in n-dimensional space, the perceptron algorithm will attempt to construct an $(n-1)$-dimensional hyperplane which separates the two classes.

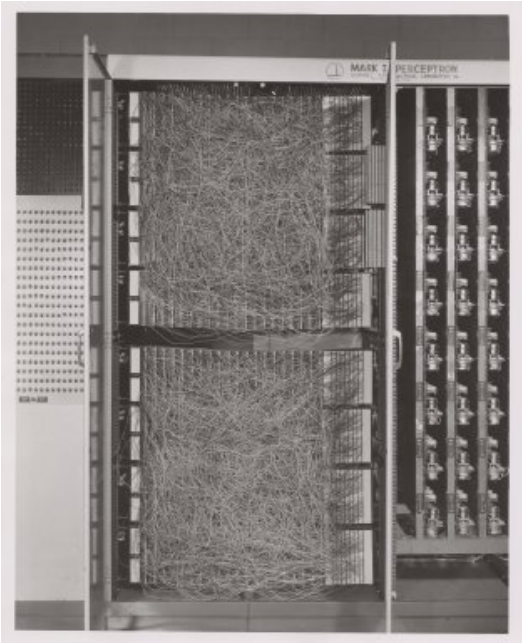

Figure 2: The first (hardware) implementation of the perceptron algorithm, called the Mark I Perceptron.

Getting started

First, we will generate linearly separable data to use with a perceptron.

- Generate 1000 data points falling above the line $1.5x + 0.2$ and 1000 data points falling below the line $1.5x + 0.05$, binding them into the same matrix.
- Create a `labels` vector where the ith entry is 1 or -1 according to whether the ith data point in the matrix was generated with `label=1` or `label=-1`. Graph the results with `qplot()`, using the `color=labels` parameter to color the points.
- Add a column to your matrix of data consisting purely of 1s. The perceptron algorithm needs this to work properly; it serves as an "intercept term" that allows the separating hyperplane it generates to be shifted up or down instead of being necessarily centered at 0.

More precisely: for each point $\mathbf{x}_i$, the perceptron algorithm forms an augmented point $\mathbf{x}_i' = (\mathbf{x}, 1)$. It works by updating a vector of *weights* $\mathbf{w}$ step-by-step as it iterates over the augmented data points, adjusting the weights whenever it encounters a training point which it misclassifies. To perform classification, it

looks at the *sign* of $\mathbf{w} \cdot \mathbf{x}_i'$, classfiying positive results as 1 and negative results as -1.

- Write a function `dot(x, y)` which computes the dot product between the vectors `x` and `y`, denoted as $x \cdot y$. It is equal to the sum of $x_1y_1 + x_2y_2 + \cdot + x_ny_n$.
- Write a function `perceptron(xs, y, w, rate, niter)` following these specifications:
  - Take as input a matrix of data `xs`, a vector of labels (1 or -1) `y`, a vector of weights `w`, a learning rate `rate`, and a number of iterations `niter`. `niter` should default to the number of data points.
  - Generate `niter` random indices from the row indices of the data, sampled *without* replacement.
  - For each sampled row, do the following: For convenience, call the sampled row `xi`. Compute the *dot product* between `xi` and `w`. The classification of `xi`, according to the perceptron, is the `sign()` of the dot product. If the classification is a false negative, update `w` by adding `rate*xi`, and if the classification is a false positive, update `w` by subtracting `rate*xi`.
  - Return the weights after the `niter` iterations have finished.

Visualizing your results

The weights of the perceptron parametrize a *line* given by `w[1]*x + w[2]*y + w[3] = 0`. This is called the *decision boundary*. Intuitively, the vector `w`, if we ignore its last entry, is a vector which is *perpendicular* to the separating hyperplane.

- What are the slope and y-intercept of this line in terms of the components of `w`?
- Set the seed for consistency and try `perceptron()` on your generated data with `rate=1` and the default value for `niter`. You should initialize the weights vector to a vector of zeroes and then pass in the output of the first `perceptron()` call to the initial weights of the next. After each call to `perceptron()`, plot the *decision boundary* corresponding to your results overlaid on top of the scatterplot of data points by adding a `geom_abline()` call with the `intercept` and `slope` parameters.

A perceptron is *guaranteed to converge* to a line which fully separates two class *if* the two classes are linearly separable. If a perceptron converges to a solution, that means that it will correctly classify all of the training data.

- Write a function `perceptron_conv()` which takes in all the arguments of `perceptron()` aside from `niter`. It should run `perceptron()` until the

solution converges. It should return a list with the final weights as well as the number of total iterations (it's fine if the number of iterations is an overestimate).

- Set the seed for consistency and plot the decision boundary which `perceptron_conv()` gives you for your output. Try a variety of different seeds. How much variation do you observe?
- How does the learning rate of the perceptron affect the speed of convergence for your current data? What if you have 20 data points (generated in the same way) instead of 2000?

Congratulations: you have successfully implemented one of the classic algorithms of machine learning. It even counts as a neural network (with a single neuron)!

Regarding the perceptron's advantages, Vapnik writes:

> By the time the Perceptron was introduced, classical discriminant analysis based on Gaussian distribution functions had been studied in great detail. [...]
>
> [T]o construct this model using classical methods requires the estimation of about $0.5n^2$ parameters where n is the dimensionality of the space. Roughly speaking, to estimate one parameter of the model requires C examples. Therefore to solve the ten-digit recognition problem using the classical technique one needs $\approx 10(400)^2C$ examples. The Perceptron used only 512.
>
> This shocked theorists. It looked as if the classical statistical approach failed to overcome the curse of dimensionality in a situation where a heuristic method that minimized the empirical loss easily overcame this curse.

Unfortunately, despite its advantages, the standard perceptron cannot classify data which is not linearly separable, because the algorithm will simply not converge.

- Verify the above statement by generating a dataset that is *not* linearly separable with `quad_pair()`.

Thankfully, the weights will eventually *cycle* in the case of nonseparability, but detecting a cycle can take a long time if the sample size is large.

Also, as you have seen, when it *does* converge, the perceptron's solution is highly dependent upon choice of random seed. When the training data are linearly separable *and* there exists a "gap" or "margin" between them, there are infinitely many possible choices of a valid decision boundary. We will see later how to resolve these problems.

- Read more about the fascinating history of perceptrons in the Wikipedia article for Minsky and Papert's book about perceptrons.

Support vector machines

Support vector machines (SVMs) were developed by Vladimir Vapnik in the late 20th century. During a lull in neural net research due to lack of computation power, SVMs emerged as a very powerful method for both classification (initially) and regression (later). Even now, they remain very competitive with other popular machine learning methods.

Overview

Support vector machines are based on the idea of **margin**. Recall that with a perceptron, we have infinitely many choices of separating hyperplane. It is with the idea of *margin* that we can define what it means for a particular separating hyperplane to be the best choice.

Intuitively, we don't really care about the points which are far away from the decision boundary – they're easy to classify. However, with the points quite close to the decision boundary, we would like to have the most "leeway for error" possible with them. If our decision boundary is very close to one side of the "corridor" which separates the two classes, then a little bit of extra error in newly generated data could end up causing the data to be misclassified!

As such, we can find *two* parallel hyperplanes which separate the two classes of data perfectly such that the distance between them, the *margin*, is as large as possible. The *maximum-margin hyperplane*, which we can use for classification, is the hyperplane exactly between the two parallel hyperplanes.

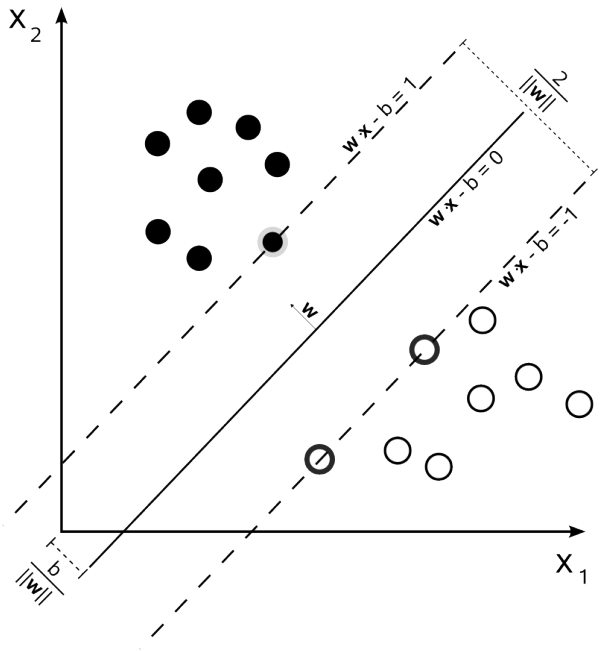

Figure 3: Two parallel hyperplanes and the maximum-margin hyperplane in between.

One may notice that the boundaries of the two classes are completely determined by the data points lying at the very edge at the margin, which are called **support vectors**. The maximum-margin hyperplane can be expressed as a linear function of the support vectors S:

$$\sum_{i=1}^{|S|} y_i \alpha_i \langle \mathbf{x}, \mathbf{x}_i \rangle + b = 0,$$

where $y_i \in \{-1, +1\}$ as usual and the α_is are positive constants. (If you are unfamiliar with the $\langle$ and $\rangle$ brackets, which represent the inner product, you can just imagine in its place $\mathbf{x} \cdot \mathbf{x}_i$ or $\mathbf{x}^\intercal \mathbf{x}_i$.) The non-support vectors contribute *nothing* to the final description of the hyperplane.

The version of SVMs described above, which is linear and finds a hyperplane between linearly separable classes, is called **hard-margin SVM**, because we refuse to allow any points to be misclassified.

In contrast, **soft-margin SVM** maximizes the margin with one modification: the algorithm is allowed to misclassify points at a particular cost C. Hard-margin SVMs have some problems with overfitting to the data, so C is like a regularization parameter. (In the limit of $C \to \infty$, the soft-margin SVM becomes

a hard-margin SVM.) For soft-margin SVM, the margin is in fact allowed to be *negative*; the more negative it becomes, the larger the number of support vectors.

In practice, soft-margin SVMs are superior to hard-margin SVMs even when the data are linearly separable. Hard-margin SVMs are sensitive to noise and outliers, as shown in the diagram below.

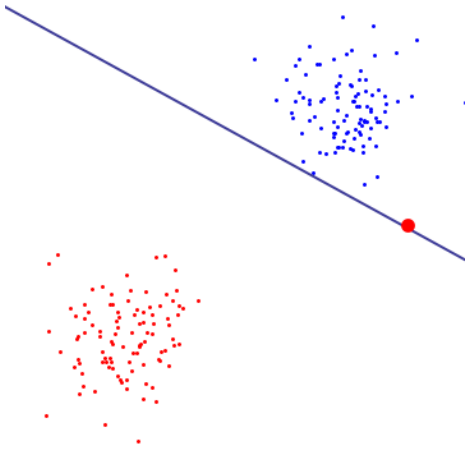

Figure 4: Here, the single red outlier is greatly affecting the angle of the boundary of the blue points, so the boundary has overfit to the training data.

The cost parameter C can be intuitively understood by appealing to the optimization problem of maximizing the margin. In this optimization problem, the resulting *Lagrange multipliers* are the α_i constants mentioned previously, and C imposes an upper bound on the size of each α_i. As such, the smaller the cost C, the less each individual point can influence the position of the maximum-margin hyperplane.

Consider, for instance, the diagram below, where the support vectors for each separating hyperplane are circled:

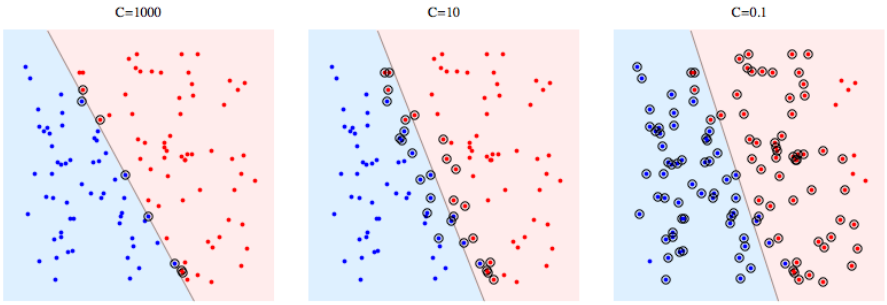

Figure 5: Increasing C trades off stability in return for linear separability.

With $C = 1000$, the soft-margin solution is very similar to what one would obtain with a hard-margin SVM. In contrast, for $C = 0.1$, the hyperplane does a poorer job of separating the training data but is more robust to changes in the position of any one of its (many) support vectors.

SVMs in R

Support vector machines are implemented in R as `svm()` in the `e1071` package.

When you call `svm()` in the following exercises, you should always specify `kernel="linear"`, because `svm()` defaults to using a *radial* kernel. We will discuss kernel methods later.

- Return to the linearly separable data with 2000 points which you initially used for the perceptron. Convert the matrix into a dataframe, add on a column with the class labels (which should be either $+1$ or -1), and convert the class label column into a factor. Name the columns of your dataframe appropriately.
- Run `svm()` on your data to predict class label from x and y coordinates. Visualize the resulting model with `plot(fit, df)`. The points marked with an "X" are the support vectors of the soft-margin hyperplane.
- Vary the `cost` parameter of `svm()` from 0.1 all the way to much higher values. Each time, plot the resulting SVM fit. Interpret the results.
- Generate 40 data points, where 20 of them fall above $y = 3(x - 0.5)^2 + 0.55$ and the other 20 fall below $y = 3(x - 0.5)^2 + 0.4$. Plot the data. (If you want a better visualization of what the class boundaries are like, plot the result with 2000 points.) Turn your matrix of data into a dataframe and add a column with class labels; turn the class label column into a factor.
- Try classifying the data with a linear SVM. How well does it work? How does the hyperplane change as you vary C?

- How well can a linear SVM separate data when each class is drawn from a different multivariate normal distribution? Try it for both distributions which have means close to each other and distributions with have means far away from each other.
- Return to the aggregated speed dating dataset and use a linear SVM to separate males from females in terms of their self-rated activity participation. Use cross-validation to select the best value of C. Read the documentation for `plot.svm()` to figure out how to plot the SVM results for two specific dimensions. Visually explore and interpret the results.

Closing notes

Having used SVMs to classify some data, it may not yet be entirely clear why support vector machines receive so much attention. Is the support vector machine not simply a slightly better version of the perceptron? Is it not almost entirely reliant upon the data being approximately linearly separable?

In the next section, you will see how we can cleverly overcome these challenges.

Note that SVms can be used to perform regression, not just classification. However, the formulation of SVM regression is not as elegant, and it is an uncommonly used technique. Nevertheless, be aware that it exists.

Kernel methods

It is through the ingenious theory of *kernel methods* with which we are able to turn linear methods (such as the linear SVM) into very complex nonlinear methods. Note that **kernel methods are not intrinsically tied to SVMs**; one can apply kernel methods to other linear techniques as well, hence the existence of kernel logistic regression, kernel Fisher discriminant analysis, and so on and so forth. People often confuse kernel methods with SVMs themselves, which is a terrible, terrible error; the two were developed hand-in-hand and kernel methods work particularly well with SVMs, but they are *separate ideas*.

We will only briefly touch upon kernel methods in the context of SVMs, giving particular attention to the *radial (Gaussian) kernel* Nevertheless, you should be aware that the field of kernel methods is immensely rich and deep, with many far-reaching applications.

Overview

The general idea behind a kernel method is that if your data are not linearly separable, you can use a *kernel function* to map them into a higher-dimensional

space where they *are* separable.

Consider, for example, the following 2-dimensional data, for which the two classes are not linearly separable.

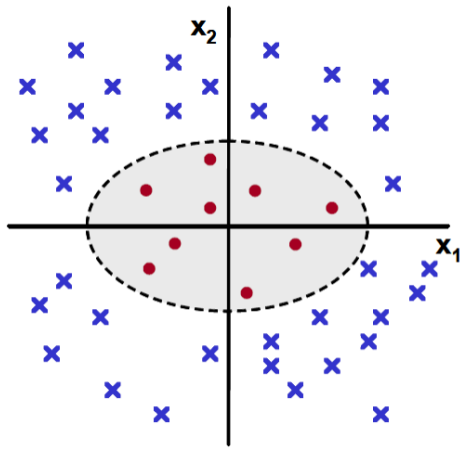

Figure 6: Data in original coordinates.

No line you draw through that graph will separate all the red dots from the blue crosses.

However, we can map the data into a 3-dimensional space as follows: for each point (x_1, x_2) we map it into the point $(x_1^2, \sqrt{2}x_1x_2, x_2^2)$. By making this mapping, we have successfully made our data linearly separable with a hyperplane in 3-dimensional space.

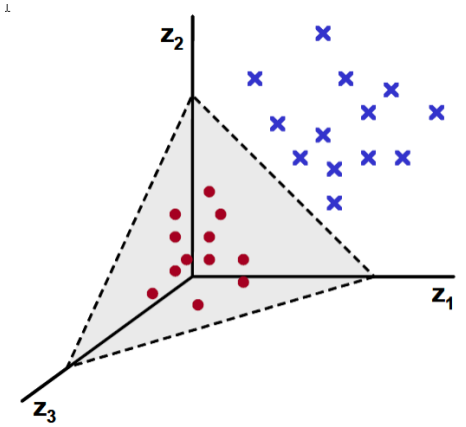

Figure 7: Data mapped into a higher-dimensional space.

Although this is a simplistic depiction of mapping into higher-dimensional spaces, the intuition remains valid for kernel methods.

Kernels, aside from being intrinsically useful, serve as a superior alternative to manually mapping features to more complex nonlinear terms, like in the above example. Doing so can massively increase the number of features and take an enormous amount of computation time. Kernels allow us to avoid this problem by not actually explicitly computing the mapping into a high-dimensional space with the *kernel trick*. The specific form of this mapping is adapted so as to compel the data to take on a linearly separable form in the higher-dimensional space, even if the two classes are hopelessly intertwined together in 2-dimensional space.

There are two essential mathematical facts which underlie the usage of kernel methods.

Fact 1: The VC dimension of hyperplanes

In connection with SVMs, Vapnik developed the theory of the Vapnik-Chervonenkis dimension, which measures the flexibility of a classification algorithm, allowing him to obtain hard bounds on the generalizable error of SVMs and perform very powerful analysis of machine learning algorithms in general. The higher the VC dimension of a particular classifier, the more flexible the algorithm is.

Precisely, consider a set of classification algorithms H. Now, consider the largest possible dataset such that for any arbitrary labeling of that dataset into one of two classes, there exists some $h \in H$ capable of perfectly classifying the data. The size of that dataset is precisely the VC dimension of H, and we say that all such datasets are *shattered* by H.

For example, methods which use $(p-1)$-dimensional hyperplanes in $\mathbb{R}^p$ to classify points have VC dimension $p+1$, because one can construct an example of $p+1$ points in $\mathbb{R}^p$ that are shattered by the set of separating hyperplanes in $\mathbb{R}^p$. These methods include linear discriminant analysis, logistic regression, the perceptron algorithm, and linear support vector machines.

In general, we would like the VC dimension of the algorithm which we use to be high enough to classify the training data very well. However, if the VC dimension is *too* high, then the classification algorithm will be unnecessarily flexible and may overfit to our training data. Due to this, Vapnik articulated the principle of structural risk minimization. Broadly speaking, we should choose an algorithm with as low of a VC dimension as possible which still manages to fully separate the two classes in our data.

The importance of VC dimension is given by the following theorem. Let the vectors $\mathbf{x} \in \mathbb{R}^p$ belong to the sphere of radius 1. Then the VC dimension h of the set of hyperplanes with margin $\rho = \langle \mathbf{w}, \mathbf{w} \rangle^{-1}$ is bounded by

$$h \leq \min\{\langle \mathbf{w}, \mathbf{w} \rangle, p\} + 1.$$

Now, suppose that we work in an infinite dimensional Hilbert space, where $p = \infty$. Then the VC dimensions of the set of separating hyperplanes with some given margin ρ is entirely dependent on ρ^{-1}. The larger we make the margin, the smaller the VC dimension of the corresopnding hyperplanes.

We can then follow the principle of structural risk minimization by simply using this strategy:

> Map input vectors $\mathbf{x} \in X$ into (a rich) Hilbert space $\mathbf{z} \in Z$, and construct the maximal margin hyperplane in this space.

Moreover, VC dimension theory can be applied to obtain a bound on the generalization error of a classification algorithm as a function of the VC dimension! As such, simply by controlling the margin of the separating hyperplanes that we look for, we can control the ability of our classification model to generalize to new data generated in the same way as our training data (as magical as it may sound, it's true).

However, we have a problem: we don't really know what it means to map $X \to Z$. If we had a way to map our vectors into a very high-dimensional space, we could find separating hyperplanes in that space; however, we don't immediately have any ways to do so. Indeed, for an infinite-dimensional Hilbert space, we would need to compute vectors of infinite length, which would likely be difficult to work with in practice.

Wait! Do we really need to map $X \to Z$? Recall that the separating hyperplane for a linear, hard-margin SVM is given by

$$\sum_{i=1}^{|S|} y_i \alpha_i \langle \mathbf{x}, \mathbf{x}_i \rangle + b = 0.$$

The only place where vectors from the space X show up are within the inner product. As such, our problem is not really to find a mapping from $X \to Z$; rather, it is to *find a way to compute the inner product of X vectors in Z*. This is a subtly different and slightly less challenging problem, which Mercer's theorem allows us to attack.

Fact 2: Mercer's theorem

Mercer's theorem states the following:

> Let vectors $\mathbf{x} \in X$ be mapped into vectors $\mathbf{z} \in Z$ of some Hilbert space.

1. Then there exists in X space a symmetric positive definite function $K(\mathbf{x}_i, \mathbf{x}_j)$ that defines the corresponding inner product in Z space:
$$\langle \mathbf{z}_i, \mathbf{z}_j \rangle = K(\mathbf{x}_i, \mathbf{x}_j).$$
2. Also, for any symmetric positive definite function $K(\mathbf{x}_i, \mathbf{x}_j)$ in X space there exists a mapping from X to Z such that this function defines an inner product in Z space.

This means that our separating hyperplane calculated in Z space has the form

$$\sum_{i=1}^{|S|} y_i \alpha_i K(\mathbf{x}, \mathbf{x}_i) + b = 0.$$

Mercer's theorem says that instead of having to directly find some function $\phi : X \to Z$ and to then calculate $\langle \phi(\mathbf{x}), \phi(\mathbf{x}_i) \rangle$, *all we need* is an appropriate *kernel function* $K(\cdot, \cdot)$! The computation of the inner product in Hilbert space directly instead of having to explicitly evaluate the infinitely long vectors $\phi(\cdot)$ is known as the **kernel trick**.

Summary

Here are the main takeaways about what the *kernel trick* allows us to do:

1. With an appropriately chosen kernel, we can get perfect separation of *any* training data. (We might not want to, because of overfitting, but we certainly *can*.)
2. This separation is done by finding *hyperplanes* in a high-dimensional space to which our data is mapped. It does not look like a linear process in the original space, but is linear in that high-dimensional space.
3. The mapping from the data space to the high-dimensional space, $\phi : X \to Z$, is *not* explicitly computed. Rather, it is *implicitly induced* by our choice of kernel function $K : X \times X \to \mathbb{R}$.
4. The space to which we map our features is infinite-dimensional. It is because of the kernel trick that we can do any sort of work with these infinite-dimensional vectors: instead of having to directly compute them, which would be impossible, the kernel function calculates the inner product which we need.

The radial kernel

Aside from the linear kernel, the most commonly used kernel is the radial kernel, also known as the Gaussian kernel or the RBF (radial basis function) kernel. It is given by

$$K(\mathbf{x}, \mathbf{x}') = \exp\left(-\frac{\|\mathbf{x} - \mathbf{x}'\|^2}{2\sigma^2}\right).$$

When this kernel is used in a machine learning algorithm, it has one associated hyperparameter, namely σ. In practice, when used for classification with a support vector machine, the radial kernel works *extremely* well. Intuitively, it is because Gaussian functions are very smooth, and the usage of a radial kernel can be thought of as a sort of "regularized k-Nearest Neighbors".

More precisely, it is because the particular Hilbert space corresponding to the Gaussian kernel represents functions which are very smooth. Because of this smoothness, the functions learned by your SVM are themselves very smooth, and the assumption that decision boundaries vary in a sensible and smooth manner is usually very reasonable.

Kernelized SVMs in R

The `svm()` function supports `"radial"`, `"polynomial"`, and `"sigmoid"` as arguments for its `kernel` parameter. The documentation describes how to set the hyperparameters corresponding to each type of kernel.

- Try using a radial kernel SVM with different costs and values for σ on both the linearly separable and the quadratically separable data, plotting the results of each fit. How does the best value of σ vary with the sample size?
- Are there sample sizes for which a linear kernel can do better than a radial kernel?
- Write better versions of `lin_pair()` and `quad_pair()` that return points in p-dimensional space for a specified value of p. (You will need to pass in parameters which define a $(p-1)$ dimensional hyperplane or quadric surface and then check which side of the surface a generated point is on.) When p is much greater than the number of data points, how does the performance of a linear kernel compare to the performance of a radial kernel?
- Just for fun, try to use the polynomial and sigmoid kernels to classify the data and see how well they do.
- Use a radial kernel SVM to classify speed dating participants' gender with their self-rated activity performance. If you use cross-validation to select the best values of C and σ, how much better can you do over a linear SVM?

Advanced kernel methods

Any machine learning algorithm in which data points only appear as part of inner products can be kernelized. For example, we have the kernel perceptron, kernel logistic regression, kernel ridge regression, kernel nearest-neighbors, and many others. In fact, even unsupervised learning techniques (clustering and dimensionality reduction) can be kernelized to add nonlinearity! We have kernel principal components analysis, kernel factor analysis, kernel K-means, and much, much more...

Moreover, one can choose some very interesting kernel functions as well. For example:

- The string kernel measures string similarity and has applications in natural language processing, allowing algorithms to work much better on text.
- Graph kernels let you compute inner products on graphs, with application in chemiinformatics, bioinformatics, and social network analysis.

That being said, outside of kernels which have domain-specific use cases, people tend to stick to the linear and radial kernels in practice for standard classification tasks.

Decision trees for classification

It is also possible to use random forests and gradient boosted trees for classification.

- Using the `caret` package for hyperparameter tuning, compare the performance of both random forests and gradient boosted trees for classification on a variety of different datasets: simulated and linearly separable, simulated and quadratically separable, simulated as multivariate normal distributions, perhaps a mixture of multiple datasets, and so on and so forth.

The usage of random forests and gradient boosted trees is more or less the same as with regression, so there are not many new insights to add here. They are part of the predictive class of models, not the generative class, but being decision tree-based algorithms are inherently nonlinear and are not greatly related to the other algorithms discussed in this lesson.

Which classifier should you use?

These three papers perform various empirical comparisons of classifier methods:

1. Caruana and Niculescu-Mizil (2008): An Empirical Comparison of Supervised Learning Algorithms
2. Caruana *et al.* (2008): An Empirical Evaluation of Supervised Learning in High Dimensions
3. Fernández-Delgado (2014): Do we Need Hundreds of Classifiers to Solve Real World Classification Problems?

The third paper is particularly informative, being a comparison of 179 different classifiers from 17 families. There is also a follow-up from the authors about the performance of gradient boosted trees.

- Read through their abstracts. If any details interest you, look within the papers to answer your questions.