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.
  - For this function, **don't** restrict the return values to be within the unit square, although when using `mvnorm_pair()` you should pass in means that are inside the unit square.
  - 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 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 10%, 15%, 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, passing in the true group assignments as labels for the points (*e.g.* with `color=df_wine$Type` in `qplot()`. Interpret the results. Does the accuracy of the LDA classification make sense?

Read the following post about visualizing the results of LDA:

- Compute and graph the LDA decision boundary

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.

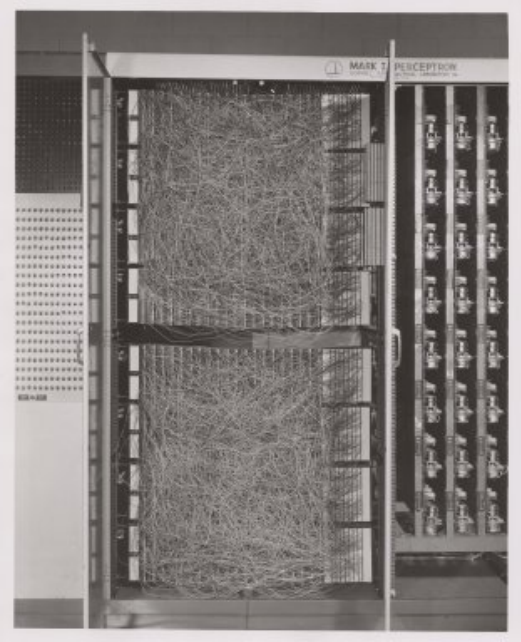

Figure 1: The first (hardware) implementation of the perceptron algorithm, called the Mark I Perceptron.

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 3 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.

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 corresponding maximum-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 and evaluate the accuracy of the linear SVM.

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.

- Our current formulation of support vector machines only supports binary classification. In general, if you have a binary classifier (of any type), how can you use it for multiclass classification? Come up with at least one potentially viable method.

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?

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.