Linear classifiers

- and not so linear ones

Last time

- History of DL
- Classification
- Detection
- The *k*-NN classifier

Linear least squares

- Predict the scalar response *y* from an input vector **x**
- For example in 2D:

$$f(\mathbf{x}) = w_0 + w_1 x_1 + w_2 x_2 = \mathbf{w}^\top \mathbf{x} \qquad \mathbf{w} = [w_0, w_1, w_2]^\top \qquad \mathbf{x} = [1, x_1, x_2]^\top$$

- In least squares, we “learn” this predictor by solving for **w**
- Define the *loss L*:

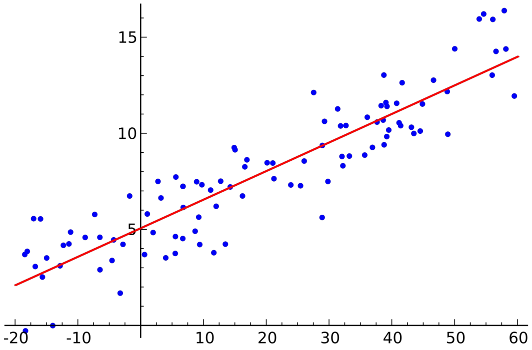
$$L(f(\mathbf{x}), y) = \frac{1}{2} \sum_{i=1}^{m} (y_i - \mathbf{w}^\top \mathbf{x}_i)^2 = \frac{1}{2} \|\mathbf{y} - X\mathbf{w}\|^2$$

- Differentiate *L* w.r.t. **w** and solve by setting to zero:

$$\mathbf{w} = (X^\top X)^{-1} X^\top \mathbf{y}$$

$$\mathbf{w} = X^\dagger \mathbf{y}$$

LLS in 1D



https://en.wikipedia.org/wiki/Linear_least_squares

One-hot (1 of K) encoding

- Regression problems:
  - Predict one or more continuous *y*'s
- Classification problems
  - Predict *K* categories (2 or more)
- Multiple options
  - Integer labels: 0, 1, 2 etc. or -1, 1 (two-class problems)
  - One-hot encoding: create a *K*-dimensional vector per desired output *y* and put 1s in it
  - Example (three-class problems):
    [1,0,0]
    [0,1,0]
    [0,0,1]

Least squares classifier

- Now we change *y* from a continuous value to a discrete label
- Instead of scalar *y*, we use one-hot encoding
- Now each class C_k density is approximated by its own regression model:
  $p(C_k|\mathbf{x}) \approx f_k(\mathbf{x}) = \mathbf{w}_k^\top \mathbf{x}$
- We now need to solve for a matrix of coefficients (one model per column):
  $L(f(\mathbf{x}), \mathbf{y}) = \frac{1}{2}\sum_{i=1}^{m}(\mathbf{x}_i^\top W - \mathbf{y}_i)^2 = \frac{1}{2}\|XW - Y\|^2$
- Again the solution becomes:
  $W = X^\dagger Y$

Perceptron

- Now we apply a *non-linear* activation on top of the linear transform:

$$f(\mathbf{x}) = g(\mathbf{w}^\top \mathbf{x})$$

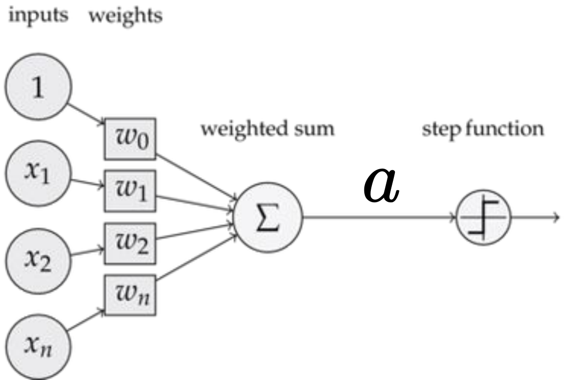
- Remember:

$$\mathbf{w} = [w_0, w_1, w_2, \ldots]^\top \qquad \mathbf{x} = [1, x_1, x_2, \ldots]^\top$$

- The perceptron defines the step function:

$$g(a) = \begin{cases} 1 & \text{if } \ a \geq 0 \\ -1 & \text{otherwise} \end{cases}$$

https://blog.knoldus.com/introduction-to-perceptron-neural-network



Perceptron

- The perceptron uses special values for the two classes:

$$C_1: \quad y = \ \ 1$$
$$C_2: \quad y = -1$$

- The loss is called the *perceptron criterion*:

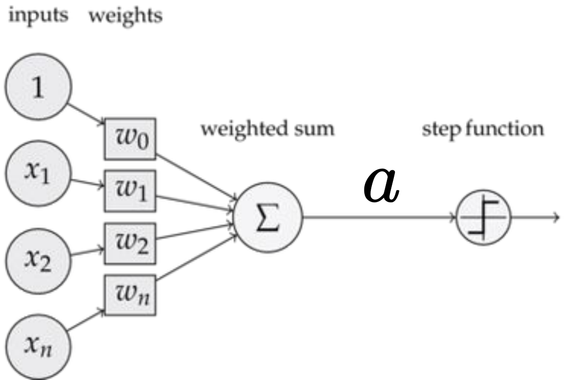
$$L(f(\mathbf{x}), y) = -\sum_{i=1}^{m} \mathbf{w}^\top \mathbf{x}_i y_i$$

- Differentiate w.r.t. the weights and we get (per example):

$$\nabla L = -\mathbf{x}_i y_i$$

https://blog.knoldus.com/introduction-to-perceptron-neural-network

$$g(a) = \begin{cases} 1 & \text{if } \ a \geq 0 \\ -1 & \text{otherwise} \end{cases}$$



Perceptron

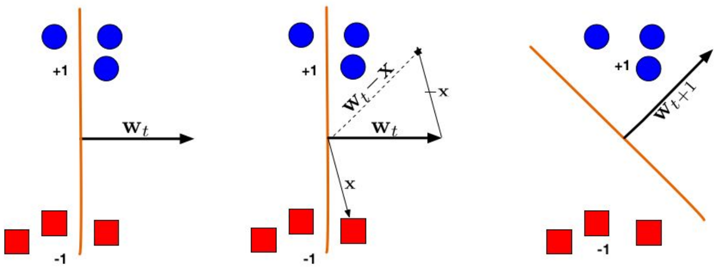
- Learning is done by *stochastic gradient descent*
  - Stochastic: select the training examples one by one in random order
  - Gradient descent: use the negative of the gradient to update the weights
- Then we get (again per example):

$$\mathbf{w} \leftarrow \mathbf{w} - \nabla L$$

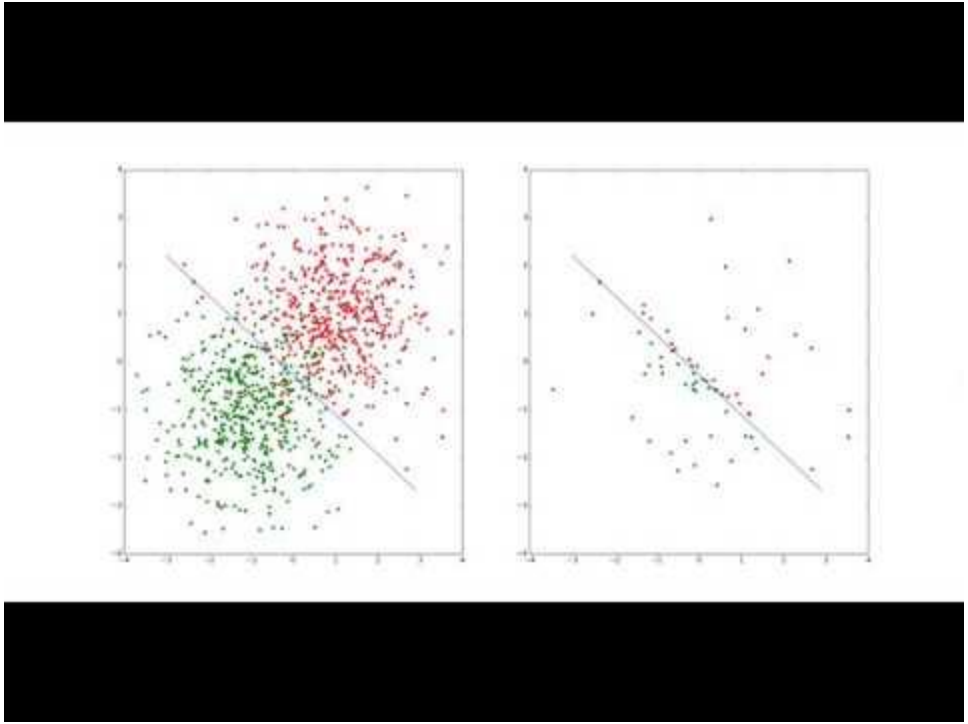
$$\mathbf{w} \leftarrow \mathbf{w} + \mathbf{x}_i y_i$$

- Here's how a single update looks (from time *t* to *t+1*):

http://www.cs.cornell.edu/courses/cs4780/2018fa/lectures/lecturenote03.html



Perceptron



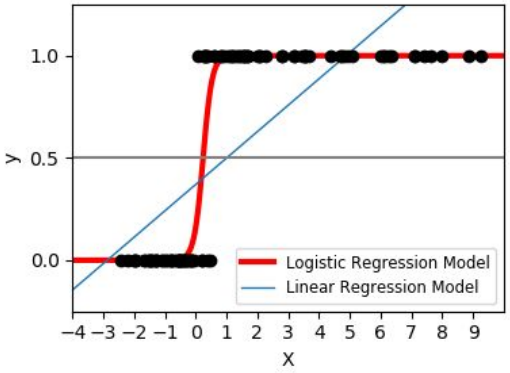
Logistic regression

- Again we consider a two-class problem
- Now we switch to the more frequently-used labels 0 and 1
- The prediction model now uses the *sigmoid*:

$$p(C_1|\mathbf{x}) = \sigma(\mathbf{w}^\top \mathbf{x}) \qquad \sigma(a) = \frac{1}{1+e^{-a}}$$

- The second class is trivial:

$$p(C_2|\mathbf{x}) = 1 - p(C_1|\mathbf{x})$$



https://scikit-learn.org/stable/auto_examples/linear_model/plot_logistic.html

Logistic regression

- Since LR is probabilistic by nature, we can use *maximum likelihood* to find **w**
- Each output is either 0 or 1, which defines a Bernoulli trial: $p(C_1|\mathbf{x})^y(1 - p(C_1|\mathbf{x}))^{1-y}$
- The *likelihood function* for all outputs then becomes:

$$\mathcal{L}(\mathbf{w}) = \prod_{i=1}^{m} \sigma(\mathbf{w}^\top \mathbf{x}_i)^{y_i} \left(1 - \sigma(\mathbf{w}^\top \mathbf{x}_i)\right)^{1-y_i}$$

- Now let's shorten the linear part a bit:

$$a = \mathbf{w}^\top \mathbf{x} \qquad\qquad \mathcal{L}(\mathbf{w}) = \prod_{i=1}^{m} \sigma(a_i)^{y_i} (1 - \sigma(a_i))^{1-y_i}$$

- As usual in ML estimation, it's easier to take the logarithm:

$$\log \mathcal{L}(\mathbf{w}) = \sum_{i=1}^{m} \left[y_i \log \sigma(a_i) + (1 - y_i) \log(1 - \sigma(a_i))\right]$$

Log-likelihood

Logistic regression

- Now we want the gradient of $\log \mathcal{L}(\mathbf{w}) = \sum_{i=1}^{m} \left[y_i \log \sigma(a_i) + (1 - y_i) \log(1 - \sigma(a_i)) \right]$
- We will use a nice property of the sigmoid: $\boxed{\sigma'(a) = \sigma(a)(1 - \sigma(a))}$
- We omit the index *i* for readability:

$$\nabla \log \mathcal{L}(\mathbf{w}) = y \frac{\partial \log}{\partial \sigma(a)} \sigma'(a) \frac{\partial a}{\partial \mathbf{w}} + (1 - y) \frac{\partial \log}{\partial (1 - \sigma(a))} (-\sigma'(a)) \frac{\partial a}{\partial \mathbf{w}}$$

- Using the sigmoid property and $\frac{\partial \log}{\partial x} = \frac{1}{x}$

$$\begin{aligned} \nabla \log \mathcal{L}(\mathbf{w}) &= y \frac{1}{\sigma(a)} \sigma(a)(1 - \sigma(a))\mathbf{x} + (1 - y) \frac{1}{1 - \sigma(a)} (-\sigma(a)(1 - \sigma(a)))\mathbf{x} \\ &= y(1 - \sigma(a))\mathbf{x} - (1 - y)\sigma(a)\mathbf{x} \\ &= (y - y\sigma(a))\mathbf{x} - (\sigma(a) - y\sigma(a))\mathbf{x} \\ &= (y - \sigma(a))\mathbf{x} \end{aligned}$$

- So all in all we have:

$$\nabla \log \mathcal{L}(\mathbf{w}) = \sum_{i=1}^{m} (y_i - \sigma(\mathbf{w}^\top \mathbf{x}_i))\mathbf{x}_i$$

https://medium.com/howilearntpiano/phew-c521b843b00d

But wait...

- You've probably seen linear regression like this:

$$y = f(\mathbf{x}) + \epsilon = \mathbf{w}^\top \mathbf{x} + \epsilon \qquad \epsilon \sim \mathcal{N}(0, \sigma^2)$$

- This means:

$$p(y|\mathbf{x}) \sim \mathcal{N}(f(\mathbf{x}), \sigma^2)$$

- Here we can also compute a likelihood:

$$\mathcal{L}(\mathbf{w}) = \prod_{i=1}^m \mathcal{N}(f(\mathbf{x}_i), \sigma^2) = \prod_{i=1}^m \frac{1}{\sqrt{2\pi\sigma^2}} \exp\left(-\frac{(y_i - f(\mathbf{x}_i))^2}{2\sigma^2}\right)$$

- Take the log again:

$$\log \mathcal{L}(\mathbf{w}) = \sum_{i=1}^m \left[-\frac{1}{2}\log 2\pi\sigma^2 - \frac{1}{2}\frac{\boxed{(y_i - f(\mathbf{x}_i))^2}}{\sigma^2}\right]$$

https://www.youtube.com/watch?v=u8Q9DknFFuM

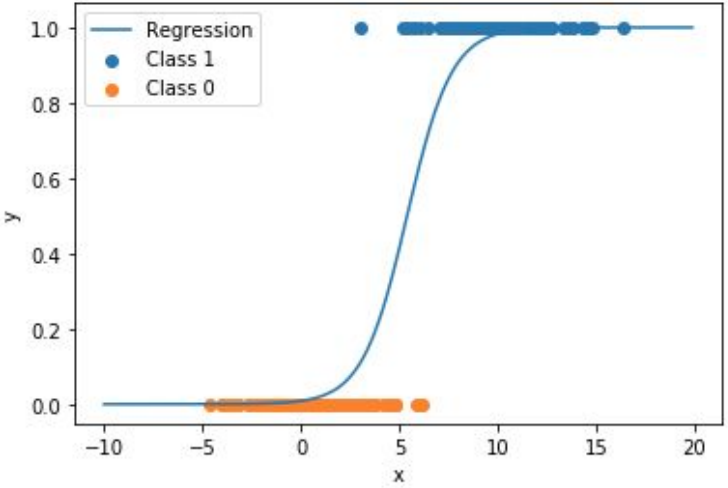
$$\boxed{L(f(\mathbf{x}), y) = \frac{1}{2}\sum_{i=1}^m (y_i - \mathbf{w}^\top \mathbf{x}_i)^2}$$

In other words, the MSE loss also comes from ML!

Back to logistic regression

- We have a log-likelihood $\log \mathcal{L}(\mathbf{w}) = \sum_{i=1}^{m} \left[y_i \log \sigma(a_i) + (1 - y_i) \log(1 - \sigma(a_i)) \right]$
- - and its gradient w.r.t. the parameters:
  $\nabla \log \mathcal{L}(\mathbf{w}) = \sum_{i=1}^{m} (y_i - \sigma(\mathbf{w}^\top \mathbf{x}_i))\mathbf{x}_i$
- That’s all we need!

https://news.yahoo.com/obama-says-choices-now-govern-future-economy-100125580.html



Multiclass logistic regression

- Now we assume *K* classes instead of two
- This also means we have *K* weight vectors
- For each class, we now model the density by the *softmax* function:

$$p(C_k|\mathbf{x}) = \frac{\exp a_k}{\sum_{i=1}^{K} \exp a_i} \qquad a_k = \mathbf{w}_k^\top \mathbf{x}$$

- This is just a generalization of the logistic function!

Solving the softmax classifier

$$p(C_k|\mathbf{x}) = \frac{\exp a_k}{\sum_{i=1}^{K} \exp a_i}$$

- First the likelihood:
  $$\mathcal{L}(\mathbf{w}_1, \ldots, \mathbf{w}_K) = \prod_{i=1}^{m} \prod_{k=1}^{K} p(C_k|\mathbf{x}_i)^{y_{i,k}} = \prod_{i=1}^{m} \prod_{k=1}^{K} \left(\frac{\exp a_k}{\sum_{j=1}^{K} \exp a_j} \right)^{y_{i,k}}$$
- Let's call our estimated output $\hat{y}$
  $$\mathcal{L}(\mathbf{w}_1, \ldots, \mathbf{w}_K) = \prod_{i=1}^{m} \prod_{k=1}^{K} \hat{y}_{i,k}^{y_{i,k}}$$
- We take the logarithm:
  $$\log \mathcal{L}(\mathbf{w}_1, \ldots, \mathbf{w}_K) = \sum_{i=1}^{m} \sum_{k=1}^{K} y_{i,k} \log \hat{y}_{i,k}$$
  Log-likelihood
- \- where we again use one-hot encoding for the targets *y*
- The gradient w.r.t. **w** becomes - again - wonderfully simple:
  $$\nabla_{\mathbf{w}_k} \mathcal{L}(\mathbf{w}_1, \ldots, \mathbf{w}_K) = \sum_{i=1}^{m} (y_{i,k} - \hat{y}_{i,k}) \mathbf{x}_i$$

Softmax classifier in practice

- Again we have a log-likelihood: $\log \mathcal{L}(\mathbf{w}_1, \ldots, \mathbf{w}_K) = \sum_{i=1}^m \sum_{k=1}^K y_{i,k} \log \hat{y}_{i,k}$
- And we have a gradient: $\nabla_{\mathbf{w}_k} \mathcal{L}(\mathbf{w}_1, \ldots, \mathbf{w}_K) = \sum_{i=1}^m (y_{i,k} - \hat{y}_{i,k})\mathbf{x}_i$
- We now put the **w**'s as column in a matrix: $W = [\mathbf{w}_1, \ldots, \mathbf{w}_K]$
- Don't forget that $\hat{Y}$ follows a discrete distribution and Y is one-hot encoded
- Here's an artificial example:

$$Y = \begin{bmatrix} 0,0,1,0,\ldots \\ 1,1,0,0,\ldots \\ 0,0,0,1,\ldots \end{bmatrix} \qquad \hat{Y} = \begin{bmatrix} 0.26, 0.08, 0.71, 0.21, \ldots \\ 0.42, 0.88, 0.08, 0.32, \ldots \\ 0.32, 0.04, 0.21, 0.47, \ldots \end{bmatrix}$$

- Then you can do it all as matrix operations:

$$\hat{Y} = \mathrm{softmax}\,(W^\top X) \qquad \nabla_W \mathcal{L} = X(Y - \hat{Y})^\top$$

Challenge

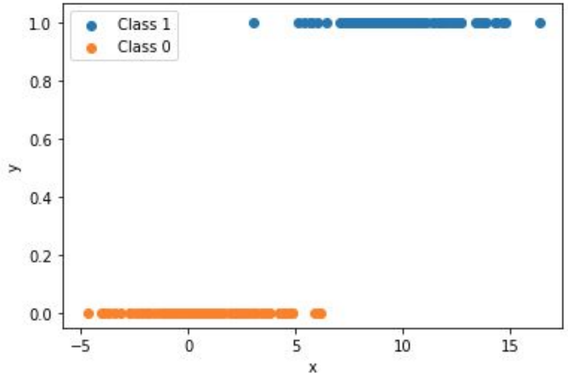
- A two-way classification problem
- Two 1D Gaussians
- Generate inputs like this
- For learning, you need to include a 1 for the bias parameter:

```
x = np.vstack((np.ones((1,200)),
               np.hstack((c0,c1))))
y = np.hstack((np.zeros_like(c0), np.ones_like(c1)))
```

- Randomly initialize **w**:

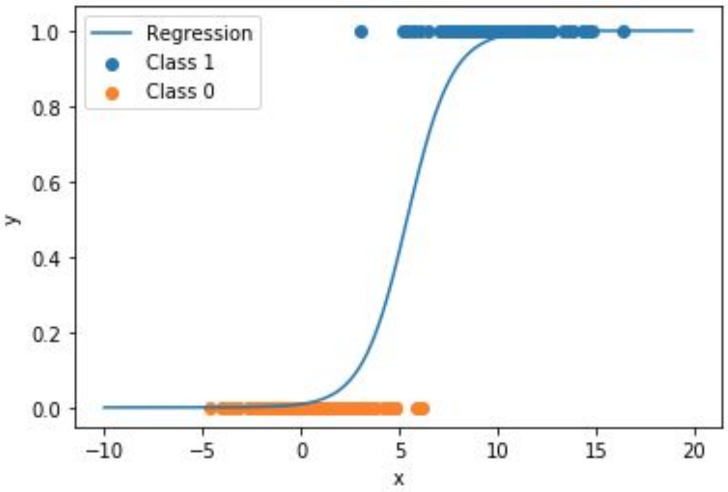
```
w = np.random.rand(2,1)
```

```python
import numpy as np
import matplotlib.pyplot as plt

c0 = np.random.normal(loc=1, scale=2.5, size=(1,100))
c1 = np.random.normal(loc=10, scale=2.5, size=(1,100))

plt.scatter(c1, np.ones_like(c1), label='Class 1')
plt.scatter(c0, np.zeros_like(c0), label='Class 0')
plt.xlabel('x')
plt.ylabel('y')
plt.legend()
plt.show()
```



Challenge

- Implement the optimization loop:
  - Compute the **avg.** log-likelihood and report it
  - Compute the **avg.** gradient **g** (2x1)
  - Update **w** by adding **g**
  - (Repeat many times)
- Tips
  - NumPy operations like `@` and `np.sum()` instead of loops
- Plot your resulting fit over [-10, 20]
  - Use **`np.arange(-10,20,0.1)`** for the x-values
  - Then evaluate your fitted sigmoid to get y-values
  - What's the probability that $x = 5$ belongs to C_1?



Challenge (harder)

- Three-way classification, 2D inputs
- Like before, create training data:

```
m = 300
X = np.vstack((np.ones((1,m)),
               np.hstack((c0,c1,c2))))
```

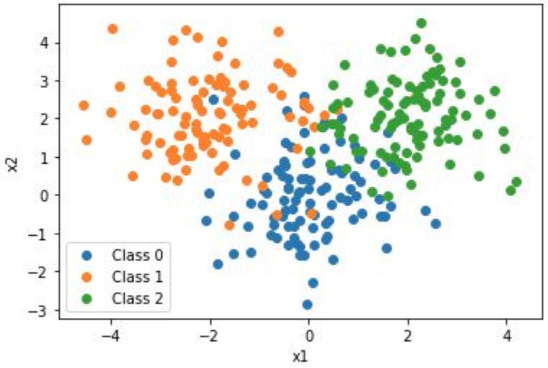
- One-hot encode the *y*'s:

```
Ylabel = np.hstack((np.zeros((1,100)),
                    np.ones((1,100)),
                    2*np.ones((1,100))
                  ))
from sklearn.preprocessing import label_binarize
Y = label_binarize(Ylabel.T, classes=[0,1,2]).T
```

- Again, randomly init the **w**'s:

```
W = np.random.rand(3,3)
```

```
c0 = np.random.multivariate_normal([0,0],np.eye(2), 100).T # 2x100
c1 = np.random.multivariate_normal([-2,2],np.eye(2), 100).T
c2 = np.random.multivariate_normal([2,2],np.eye(2), 100).T

plt.scatter(c0[0,:], c0[1,:], label='Class 0')
plt.scatter(c1[0,:], c1[1,:], label='Class 1')
plt.scatter(c2[0,:], c2[1,:], label='Class 2')
plt.xlabel('x')
plt.ylabel('y')
plt.legend()
plt.show()
```



Challenge (harder)

- Just like before:
  - Compute current softmax output using **W** and **X** (a 3x*m* data matrix)
  - Compute the **avg.** log-likelihood
  - Compute the **avg.** gradient (a 3x3 matrix)
  - Update **W**
- Tips
  - `np.sum()` with `axis` argument
  - Broadcasting in NumPy when e.g. dividing
  - NumPy operations instead of loops
  - Write the dimensions of all your variables
- What classification rate do you get?