class: middle

Data Science for Developers (Advanced)

@DrPhilWinder

DrPhilWinder

http://WinderResearch.com

http://TrainingDataScience.com

Schedule

- 09:00 Start
  - Neurons, Multi-layer Perceptrons
- 10:15 Morning break
  - Autoencoders, Convolutional Neural Networks
- 12:00-13:00 Lunch
  - Working with Text
- 14:15 Afternoon break
  - Ensemble methods, Grand challenge.
- 16:30 End

Information

Name: Phil Winder

Title: CEO Winder Research

Occupation: Engineer (cloud software and data science)

This training: First of three parts

Other training: visit https://WinderResearch.com/training

Online Training: visit https://TrainingDataScience.com

Tweet! @DrPhilWinder

Email: phil@WinderResearch.com

.center[Winder Research]

Content and Questions

- I will email slides and workshops. To do this I need your email addresses.
- Please interrupt to ask questions. Data Science is all about asking the right questions.
- We're working on a cloud environment. You don't need to install anything (unless you want to).

Setting Expectations

- This *is* difficult
- There *is* a lot of content here
- We *will* move fast (this is intentional)
- Data Science is *chaotic*; learning is not as easy as A.B.C.

The Goal:

1. Overview. Understand what you don't know.
2. Go away and continue learning.

The goal is not to become an expert in 1/2/3 days

Models we will Learn Today

- Neural Networks / Deep Learning
- Ensemble methods

Concepts we will Learn Today

- Basic understanding of Neural Networks. Not covering all architectures.
- How to work with text data
- How to use ensemble methods to increase performance (and win competitions!)

Artificial Neurons

In the 1940's, experiments showed that the brain fired *electrical* signals.

- 1943: the *McCullock-Pitts (MCP) neuron*

Neurons are simplified models of our brain.

Later it was found that the brain also communicates via hormones.

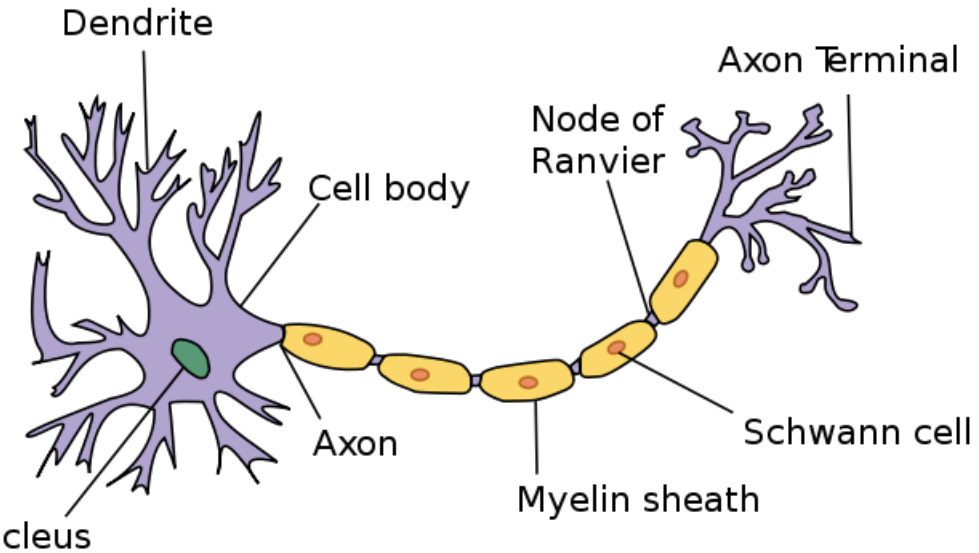
???

Before we start talking about the algorithms in detail, let's take a look at how this all started.

In 1943 the *McCullock-Pitts (MCP) neuron* was introduced.

Neurons are massively interconnected nerve cells in the brain.

They are involved in processing vast amounts of data through chemical (hormone) and electrical signals.



.center[ ]

???

They described a nerve cell as a simple logic gate with binary outputs.

Multiple inputs to the dendrites are integrated in the nucleus.

If the accumulated sum exceeds a threshold an output signal is generated and passed onto the axons.

Let 1 represent the positive class and -1 represent the negative class.

Given linear combination of input values, $\mathbf{x}$ and weights, $\mathbf{w}$ we can then calculate the *net input*, z.

$$\mathbf{w} = \begin{bmatrix} w_1 \\ \vdots \\ w_m \end{bmatrix}, \mathbf{x} = \begin{bmatrix} x_1 \\ \vdots \\ x_m \end{bmatrix}$$

$z = w_1 x_1 + \cdots + w_m x_m$

???

We can describe this model formally.

Imagine a binary classification problem.

Imagine we could alter the weights to allow us to predict a class. How can we generate the output?

- This is known as the activation function

$$\phi(z) = \begin{cases} 1 & \text{if } z \geq \theta, \\ -1 & \text{otherwise} \end{cases}$$

This activation function is known as a *unit step function* (a.k.a. *heaviside step function*).

Using this activation function we can make a decision as to which class this input belongs.

Perceptron

Now we have two equations that represent the Perceptron:

$z = w_0 x_0 + w_1 x_1 + \cdots + w_m x_m$

Where $w_0 = -\theta$ and $x_0 = 1$ (the bias term).

Then the activation function becomes:

$$\phi(z) = \begin{cases} 1 & \text{if } z \geq 0, \\ -1 & \text{otherwise} \end{cases}$$

Also, this means we can write the net input in matrix form as the *dot product* of $\mathbf{x}$ and $\mathbf{w}$:

$z = w_0 x_0 + w_1 x_1 + \cdots + w_m x_m = \sum_m^{j=0} \mathbf{w}_j \mathbf{x}_j = \mathbf{w}^T \mathbf{x}$

Rosenblatt's Perceptron

Now, the challenge is how do we train the neuron to fire when we want it to?

We use the *perceptron* model. By now you will think that this is quite simple.

1. Initialise the weights
2. For each training sample:
   a. Compute the output value $\hat{y}$
   b. Compare $\hat{y}$ to the expected output, y
   c. Update the weights to improve $\hat{y}$

In the perceptron model, $\hat{y}$ is generated from the output of the activation function.

???

This model of the MCP is the idea behind *Rosenblatt's thresholded perceptron* model.

It attempts to mimic how a single neuron in the brain behaves; it either fires, or it doesn't.

Updating Perceptron Weights

To update the weights we can define a simple update procedure:

$w_j = w_j + \Delta w_j$

Where Δw_j is the Perceptron learning rule:

$\Delta w_j = \eta \left(y - \hat{y} \right) x_j$

And η is a learning rate (a constant between 0 and 1).

Sanity check

- Imagine a single input and weight and we make the correct prediction

$\Delta w_j = \eta\,(y - \hat{y})\,x_j = \eta(1-1)x = \eta 0x$

When we apply the correction to the weight update the result is:

$w_j = w_j + \Delta w_j = w_j + 0$

???

To make sure this makes sense, let's do a quick sanity check.

Imagine we have a single input and weight. If we made the correct class prediction then the learning rule is:

When we apply the correction to the weight update the result is:

$w_j = w_j + \Delta w_j = w_j + 0$

I.e. the weight remains unchanged because it made the correct prediction.

- Incorrect prediction:

$\Delta w_j = \eta\,(y - \hat{y})\,x_j = \eta(1-(-1))x = \eta 2x$

And the update would be:

$w_j = w_j + \Delta w_j = w_j + \eta 2x$

Push the weight towards the target class

???

Or in other words, push the weight towards the target class (which was 1 in this case). The opposite, if the target class was -1, then the result of the learning rule would be $-\eta 2x$.

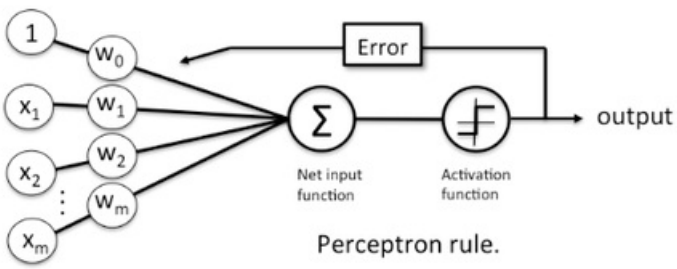
Perceptron Notes

Convergence is only guaranteed if:

- the two classes are linearly separable
- the learning rate is sufficiently small

Perceptron Model

To summarise what we have just talked about, this illustrative model is commonly used.

.center[

]

.bottom[(https://sebastianraschka.com)]

Python Implementation

Let's implement the Rosenblatt Perceptron in Python.

You will do this yourself in the workshop, but for now let's take a quick look at the code...

```
class Perceptron():
    def __init__(self, itr=10, eta=0.01) -> None:
        self.n_iterations = itr
        self.eta = eta
```

$$\phi(z) = \begin{cases} 1 & \text{if } z \geq 0, \\ -1 & \text{otherwise} \end{cases}$$

```
class Perceptron():
    # def __init__(self, itr=10, eta=0.01) -> None:

    def activation(self, z):
        if z >= 0:
            return 1
        else:
            return -1
```

$$z = w_0 x_0 + w_1 x_1 + \cdots + w_m x_m = \sum_m^{j=0} \mathrm{w}_j \mathrm{x}_j = \mathrm{w}^T \mathrm{x}$$

```
class Perceptron():
    # def __init__(self, itr=10, eta=0.01) -> None:
    # def activation(self, z):

    def net_input(self, X):
        return np.dot(self.w_[1:].T, X) + self.w_[0]    # Sum of weighted inputs
```

```
class Perceptron():
    # def __init__(self, itr=10, eta=0.01) -> None:
    # def activation(self, z):
    # def net_input(self, X):

    def predict(self, X):
        return self.activation(self.net_input(X))       # Given X => net input => activate
```

1. Initialise the weights
2. For each training sample:
   a. Compute the output value $\hat{y}$
   b. Compare $\hat{y}$ to the expected output, y
   c. Update the weights to improve $\hat{y}$

```
class Perceptron():
    # def __init__(self, itr=10, eta=0.01) -> None:
```

```
    # def activation(self, z):
    # def net_input(self, X):
    # def predict(self, X):

    def fit(self, X, y) -> None:
        self.w_ = np.zeros(X.shape[1] + 1)              # Add one for bias input (threshold)
        self.errors_ = np.zeros(self.n_iterations)      # Array for holding errors

        for i in range(self.n_iterations):              # Foreach iteration
            for (x_i, y_i) in zip(X, y):                # Foreach training instance
                y_hat = self.predict(x_i)               # Predict output
                self.errors_[i] += np.abs(y_i - y_hat)  # Add error to the current error sum

                update = self.eta * (y_i - y_hat)       # Weights update equation
                self.w_[1:] += update * x_i             # Update the weights given input
                self.w_[0] += update                    # Update the bias weight
```

```
class Perceptron():
    def __init__(self, itr=10, eta=0.01) -> None:
        self.n_iterations = itr
        self.eta = eta

    def activation(self, z):
        if z >= 0:
            return 1
        else:
            return -1

    def net_input(self, X):
        return np.dot(self.w_[1:].T, X) + self.w_[0]    # Sum of weighted inputs

    def predict(self, X):
        return self.activation(self.net_input(X))       # Given X => net input => activate

    def fit(self, X, y) -> None:
        self.w_ = np.zeros(X.shape[1] + 1)              # Add one for bias input (threshold)
        self.errors_ = np.zeros(self.n_iterations)      # Array for holding errors

        for i in range(self.n_iterations):              # Foreach iteration
            for (x_i, y_i) in zip(X, y):                # Foreach training instance
                y_hat = self.predict(x_i)               # Predict output
                self.errors_[i] += np.abs(y_i - y_hat)  # Add error to the current error sum

                update = self.eta * (y_i - y_hat)       # Weights update equation
                self.w_[1:] += update * x_i             # Update the weights given input
                self.w_[0] += update                    # Update the bias weight
```

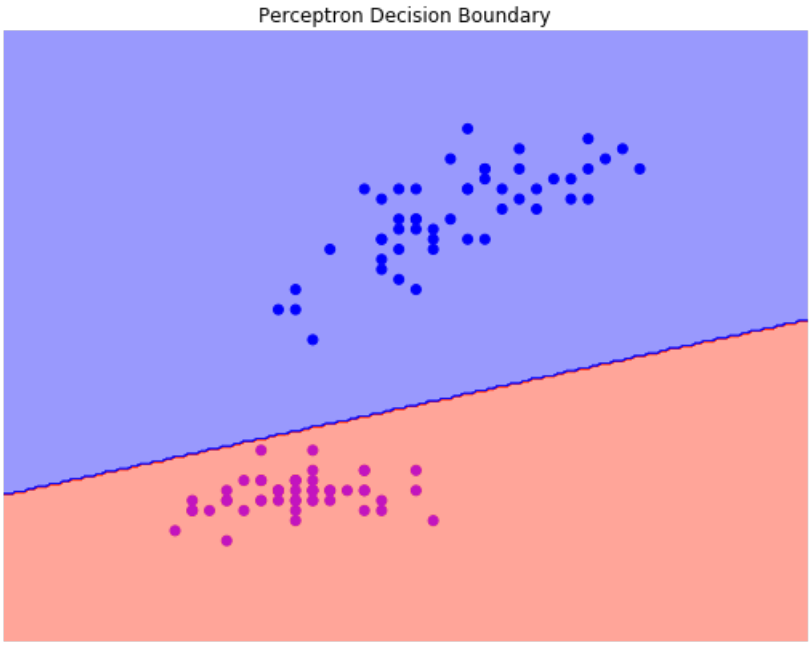
Let's use our perceptron on the iris dataset. This is the result.

.left-column[

Pretty incredible huh?!

Now you try!

]



.right-column[ ]

Workshop: 10-perceptron

Adaptive Linear Neurons

- ADAptive LInear NEuron (ADALINE) – Widrow and Hoff

Very similar to MCP

- Interesting because it introduces the key concept of minimising cost functions

Like we saw in previous sections

Instead of a unit step function, it uses a linear activation function. I.e.

$\phi(z) = \mathrm{w}^T\mathrm{x}$

This means that we need a new block on the output to actually generate a class. This is known as a quantiser.

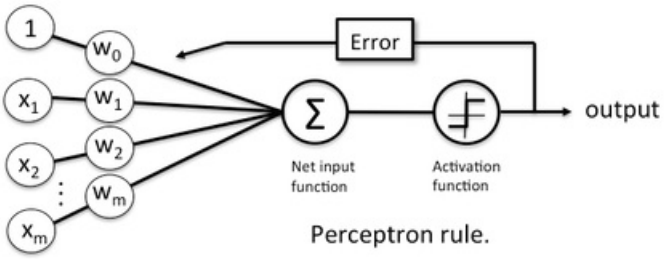
???

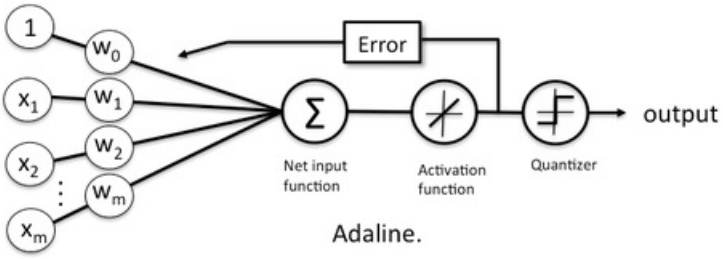
A few years after the Rosenblatt perceptron was released, Widrow and Hoff released a very similar single layer neural network called an ADAptive LInear NEuron (ADALINE).

This implementation of a neuron is particularly interesting because it introduces the key concept of minimising cost functions.

And, just in case you thought neural networks were complicated, this only has one minor change over and above the perceptron.

Comparing to the perceptron, the difference is that we need to learn how to use the continuous valued output from the linear activation function, rather than the class labels.





Easy eh?

.bottom[(https://sebastianraschka.com)]

???

But adding the linear activation function has one crucial implication.

When we were calculating the error made by the perceptron, we were simply differencing the class targets with the predicted classes.

This meant that our classifier would stop moving as soon as it made a clear pass through the classes. That's why some of you might have had different results for the class boundary.

This is probably not optimal.

Furthermore, if our linear activation function outputs a value of 1000 instead of 1, how can we quantify how close that is to the desired output?

Cost Function

We can define a measure of how bad a fit is by a cost function, J.

Like many other algorithms, the most popular choice of cost function is the sum of squared errors (SSE), defined as:

$J(\mathrm{w}) = \frac{1}{2}\sum_i (y - \phi(z))^2$

Because of this choice of cost function, we can now take advantage of something we learnt in the first day.

Because the cost function is differentiable, we can optimise the cost function using gradient descent!

So instead of updating the weights like:

$w_j = w_j + \Delta w_j = w_j + \eta\,(y - \hat{y})\,x_j$

We can change Δw_j to:

$\Delta w_j = -\eta \nabla J(\mathrm{w})$

We've mentioned it before, but the partial derivative of $\nabla J(\mathrm{w})$ is:

$\nabla J(\mathrm{w_j}) = \frac{\delta J}{\delta w_j} = -\sum (y - \phi(z)) x_j$

Which makes our new weight update function:

$w_j = w_j + \Delta w_j = w_j - \eta \sum (y - \phi(z)) x_j$

Wait a minute!!!!

Confusion, Where There is None

Taken from https://github.com/Droogans/unmaintainable-code:

> *In the interests of creating employment opportunities in the Java programming field, I am passing on these tips from the masters on how to write code that is so difficult to maintain, that the people who come after you will take years to make even the simplest changes.*
>
> *Further, if you follow all these rules religiously, you will even guarantee yourself a lifetime of employment, since no one but you has a hope in hell of maintaining the code.*

???

I bring this up because it feels like a little confusion is being purposfully added to neural networks/deep learning to keep those pesky data scientists in a job.

...

This is a linear equation:

$$f(\mathbf{x}) = w_0 + w_1 x_1 + w_2 x_2 + \dots \tag{1}$$
$$= \mathbf{w}^T \cdot \mathbf{x} \tag{2}$$

This is the partial derivative of the mean squared error used in gradient descent:

$\frac{\partial}{\partial \mathbf{w}_j} MSE(\mathbf{w}) = \frac{2}{m} \sum_{i=1}^{m} \left(\mathbf{w}^T \cdot \mathbf{x}^{(i)} - \mathbf{y}^{(i)} \right) \mathbf{x}_j^{(i)}$

This is our new ADALINE activation function:

$\phi(z) = \mathbf{w}^T \mathbf{x}$

And the update function is:

$w_j = w_j + \Delta w_j = w_j - \eta \sum (y - \phi(z)) x_j$

See what I mean?

They're both the same! ADALINE is a linear classifier! :-O <- (insert appropriate jaw-drop emoji here)

???

But seriously, this makes sense. We've created a classifier. The classifier is trying to classify separable classes using a linear decision boundary.

Whats the best classifier for that problem? A linear classifier.

There is no difference, we've just attacked the same problem from different angles.

But the crucial part is this.

At their heart, all neural networks, all of deep learning, every amazing meme-generating-gizmo-thing you see on the internet today is built from lots and lots of simple variations of linear classifiers.

Easy eh?

Now. Let's code up an ADALINE.

```
class ADALINE:
    def __init__(self, itr=10, eta=0.01) -> None:
        self.n_iterations = itr
        self.eta = eta
```

```
class ADALINE:
    #def __init__(self, itr=10, eta=0.01) -> None:

    def net_input(self, X):
        return np.dot(self.w_[1:].T, X) + self.w_[0]    # Sum of weighted inputs
```

```
class ADALINE:
    #def __init__(self, itr=10, eta=0.01) -> None:
    #def net_input(self, X):

    def activation(self, X):                            # Activation is just linear
        return self.net_input(X)                        #  combination, same as net_input
```

```
class ADALINE:
    #def __init__(self, itr=10, eta=0.01) -> None:
    #def net_input(self, X):
    #def activation(self, X):                              # Activation is just linear

    def predict(self, X):
        if self.activation(X) >= 0:                      # Quantiser
            return 1.0
        else:
            return -1.0
```

$$\nabla J(\mathrm{w_j}) = \frac{\delta J}{\delta w_j} = -\sum(y - \phi(z))x_j$$

$$w_j = w_j + \Delta w_j = w_j - \eta\sum(y - \phi(z))x_j$$

```
class ADALINE:
    #def __init__(self, itr=10, eta=0.01) -> None:
    #def net_input(self, X):
    #def activation(self, X):                              # Activation is just linear
    #def predict(self, X):

    def fit(self, X, y) -> None:
        self.w_ = np.zeros(X.shape[1] + 1)               # Add one for bias input (threshold)
        self.errors_ = np.zeros(self.n_iterations)       # Array for holding errors

        for i in range(self.n_iterations):               # Foreach iteration
            for (x_i, y_i) in zip(X, y):                 # Foreach training instance
                y_hat = self.net_input(x_i)              # Not output now, just estimate
                self.errors_[i] += ((y_i - y_hat) ** 2).sum() / 2.0  # Cost function

                # Update the weights. This is just linear gradient descent.
                self.w_[1:] += self.eta * x_i.T.dot(y_i - y_hat)
                self.w_[0] += self.eta * (y_i - y_hat).sum()    # Update the bias weight
```

```
class ADALINE:
    def __init__(self, itr=10, eta=0.01) -> None:
        self.n_iterations = itr
        self.eta = eta

    def fit(self, X, y) -> None:
        self.w_ = np.zeros(X.shape[1] + 1)               # Add one for bias input (threshold)
        self.errors_ = np.zeros(self.n_iterations)       # Array for holding errors

        for i in range(self.n_iterations):               # Foreach iteration
            for (x_i, y_i) in zip(X, y):                 # Foreach training instance
                y_hat = self.net_input(x_i)              # Not output now, just estimate
                self.errors_[i] += ((y_i - y_hat) ** 2).sum() / 2.0  # Cost function

                # Update the weights. This is just linear gradient descent.
                self.w_[1:] += self.eta * x_i.T.dot(y_i - y_hat)
                self.w_[0] += self.eta * (y_i - y_hat).sum()    # Update the bias weight

    def activation(self, X):                             # Activation is just linear
        return self.net_input(X)                         #  combination, same as net_input

    def net_input(self, X):
        return np.dot(self.w_[1:].T, X) + self.w_[0]     # Sum of weighted inputs

    def predict(self, X):
```

```
        if self.activation(X) >= 0:                    # Quantiser
            return 1.0
        else:
            return -1.0
```

Workshop: 11-adaline

Stochastic Gradient Descent

- We covered SGD in detail in earlier sections

SGD is important for several reasons

- Helps avoid local minima and plateaus
- Reduces memory footprint
- Allows for the tuning of batch size and update rate
- And can therefore converge more quickly (using less data)

???

In earlier chapters we discussed how stochastic gradient descent improved linear regression on large datasets.

The reason for improved performance is due to the frequent updates. Usually after every new instance the weights are update.

This makes is vital that the instances are presented in a random order. Otherwise it could learn a function of a subset of the data.

An adaptive learning rate is also commonly used. The value of eta, the learning rate parameter, is updated over time.

The ADALINE algorithm implemented previously almost implements SGD, we just need to make some alterations to shuffle the data. (That is unless you vectorised everything!)

```
def _shuffle(self, X, y):
    r = np.random.permutation(len(y))
    return X[r], y[r]

def _update_weights(self, x_i, y_i):
    net_output = self.net_input(x_i)  # Output of net_input
    error = (y_i - net_output)
    cost = (error ** 2).sum() / 2.0  # Cost function

    # Update the weights. This is just linear gradient descent.
    self.w_[1:] += self.eta * x_i.T.dot(error)
    self.w_[0] += self.eta * error  # Update the bias weight
    return cost
```

```
def fit(self, X, y) -> None:
    self.w_ = np.zeros(X.shape[1] + 1)              # Add one for bias input (threshold)
    self.errors_ = np.zeros(self.n_iterations)      # Array for holding errors

    for i in range(self.n_iterations):              # Foreach iteration
```

```
        X, y = self._shuffle(X, y)                  # Randomise the data on each iteration
        for (x_i, y_i) in zip(X, y):                # Foreach training instance
            cost = self._update_weights(x_i, y_i)   # Update weights
            self.errors_[i] += cost                 # Add cost
        self.errors_[i] /= len(y)                   # Average cost per iteration
```

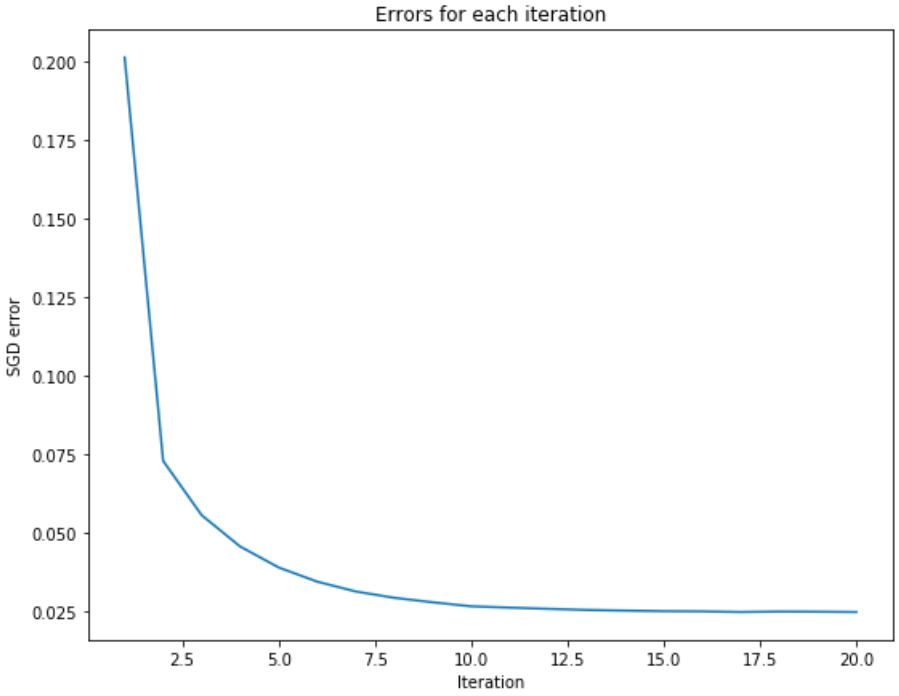
Another benefit of sgd is that we can update the weights online. I.e. if we were working with live data then we could use some of those samples to update the weights in the model.

We can add a `partial_fit` method to update the weights given new data.

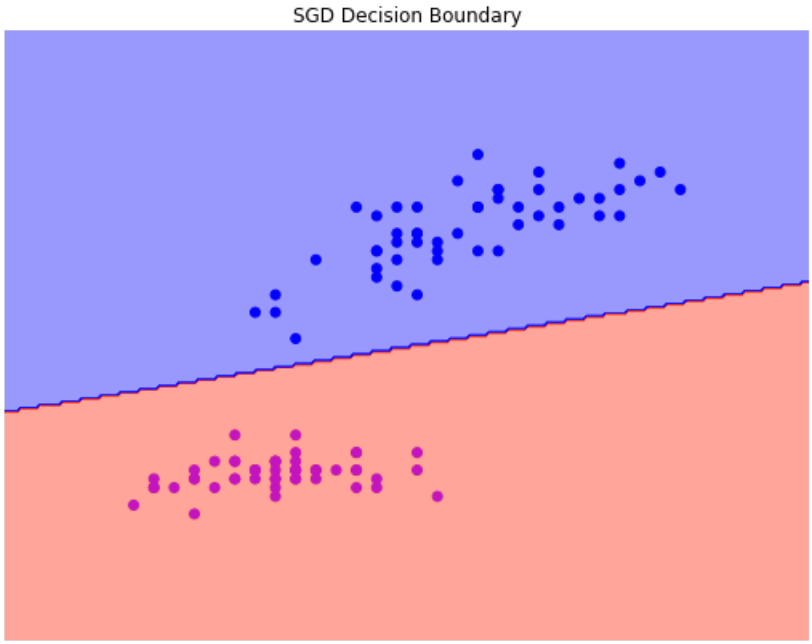
```
def partial_fit(self, X, y):
    X, y = self._shuffle(X, y)
    for (x_i, y_i) in zip(X, y):  # Foreach training instance
        self._update_weights(x_i, y_i)
```

We can see that this is basically the same as ADALINE. This is because we accidentally implemented ADALINE in a `foreach sample` loop, which is basically most of SGD!

But generally, SGD converges faster because we're updating weights more frequently. You just have to make sure to randomise the data.



.left-column[ ]



.right-column[ ]

Workshop: 12-sgd

Sigmoid Activation

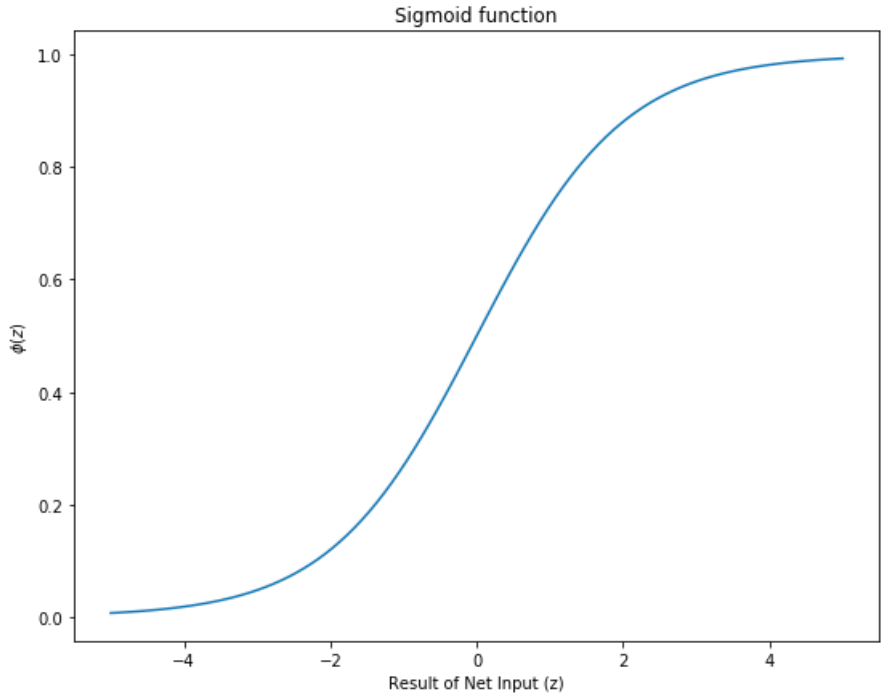
.left-column[

A sigmoid is exactly the same as a logistic function.

$\phi(z) = \frac{1}{1+e^{-z}}$

So again we take the net input from the previous layer and feed it into the natural exponential function.

]



.right-column[ ]

???

In this section we will implement another type of activation function (there are lots) called a sigmoid.

One element of the Perceptron model that is commonly altered is the activation function.

But because we're changing the activation function, we're changing the input to the error calculation, which means that the cost function changes.

- If we were to calculate the derivative of the new cost function, you would end up with the new gradient descent update rule:

$w_j = w_j + \Delta w_j = w_j - \eta \sum((y - \phi(z)) \cdot \phi(z) \cdot (1 - \phi(z)))x_j$

In other words, the update should now be:

`update = (y_i - phi_z) * phi_z * (1 - phi_z )`

This looks a little hairy, but remember, all we're doing is changing the activation function to a different shape.

Let's take a look at the new code and the results.

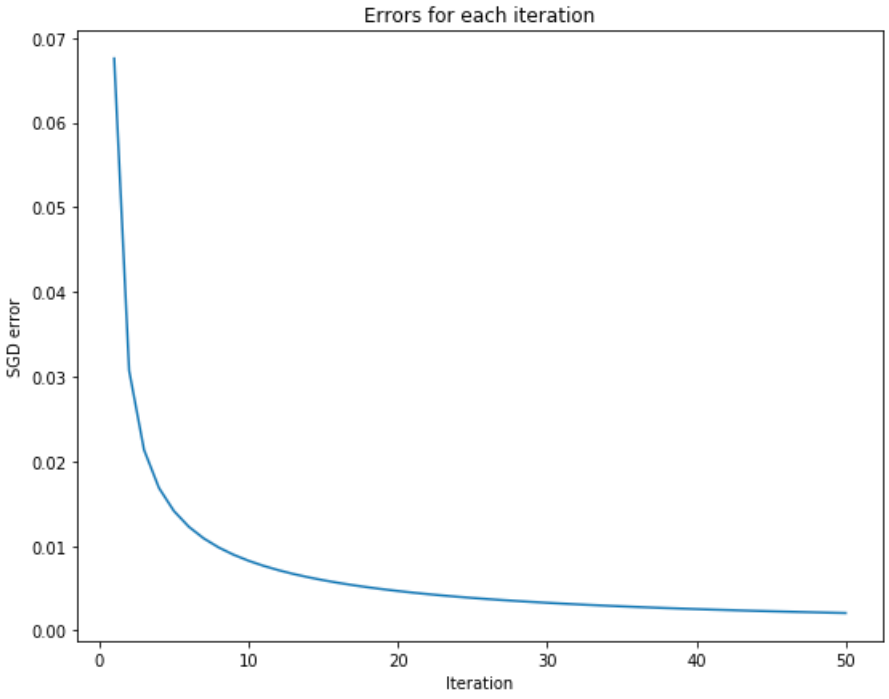
```
def _sigmoid(self, x):
    return 1 / (1 + np.exp(-x))

def activation(self, X):
    net_exp = self.net_input(X)
    return self._sigmoid(net_exp)
```

```
def predict(self, X):
    if self.activation(X) >= 0.5:                  # Quantiser changed for sigmoid
        return 1.0
    else:
        return 0

def _update_weights(self, x_i, y_i):
```

```
        phi_z = self.activation(x_i)                    # Output of activation function
        error = (y_i - phi_z)
        cost = error ** 2 / 2                           # Cost is still MSE
        update = error * phi_z * (1 - phi_z )           # New update function for sigmoid

        # Update the weights. This is just linear gradient descent.
        self.w_[1:] += self.eta * x_i.T.dot(update)
        self.w_[0] += self.eta * update
        return cost
```

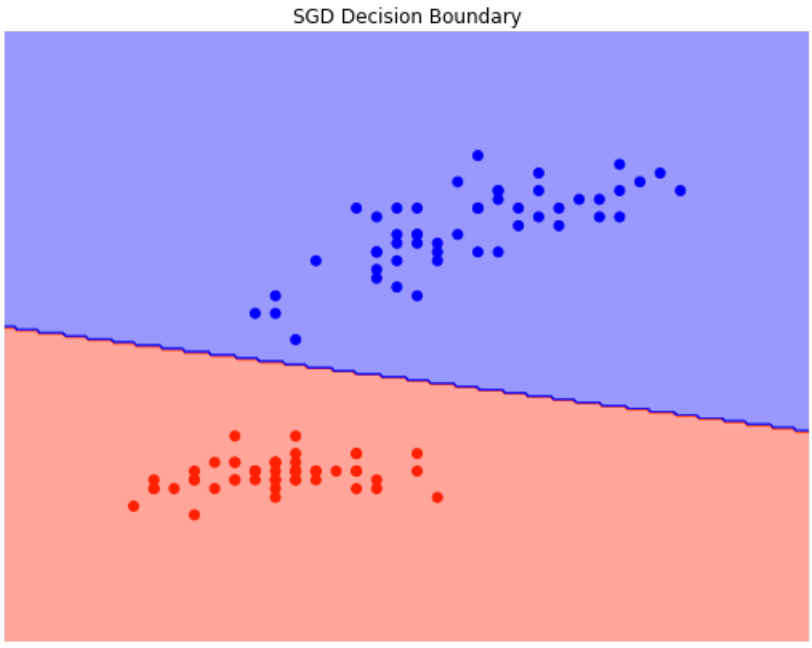
The results are quite different this time. You'll notice that it takes a long time to converge (more about that next) but the decision boundary is looking really good.

.left-column[



]

.right-column[



]

Have you seen the Sigmoid used in a similar way before?

Key Point: Yes! It's the function used in logistic regression.

- The ADALINE Perceptron with a sigmoid activation function is exactly the same as logistic regression! :-O

This also explains why Sigmoids are preferable, especially at the output of a system.

Sigmoids provide the optimal solution to finding the decision boundary to a two-class, gaussian problem.

Sigmoids also have the capability of providing true class probabilities.

Workshop: 13-sigmoid

Hyperbolic Tangent Activation

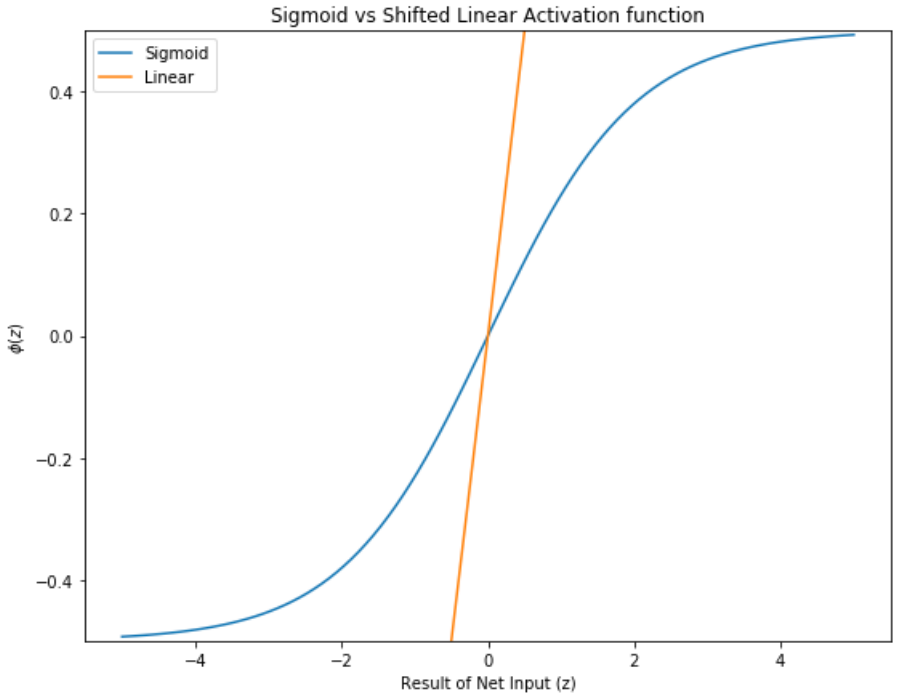
Remember that the version with the Sigmoid activation was slow to converge?

The curve showing the errors converged much slower than the linear equivalent.

Key point: The gradient of the activation function affects training performance

???

Let's compare the linear activation to a sigmoid.



.center[ ]

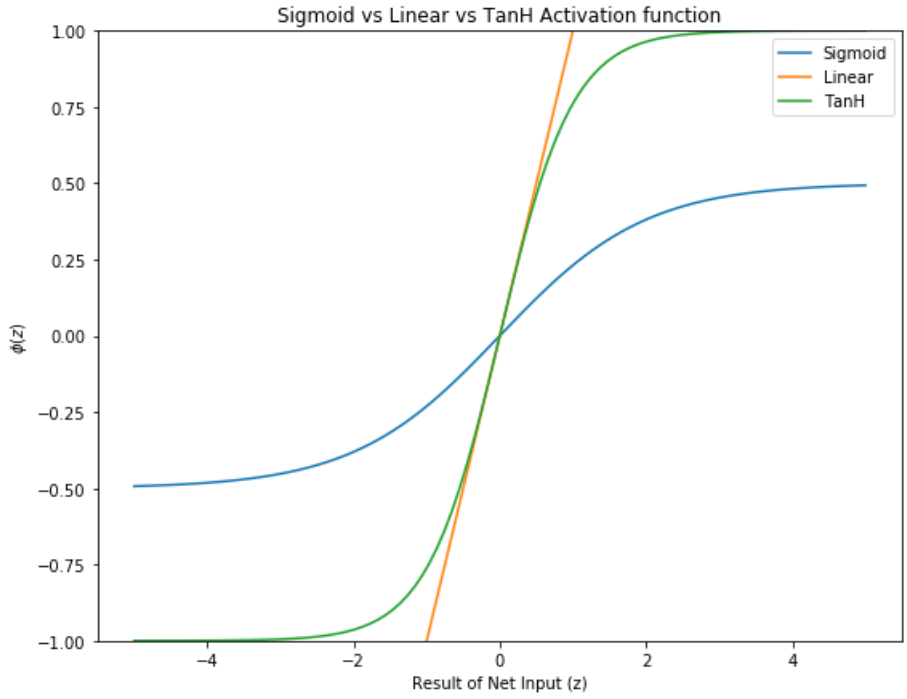
This is the same plot of a Sigmoid activation function as before, with a linear activation plotted on top. (I've shifted the sigmoid to be symmetrical around zero)

Note the gradient of the two curves.

- The gradient descent operates by traversing the gradient to minimise the error.
- The gradient of the linear activation function is higher than that of the sigmoid. Hence the linear activation function converges faster, given the same parameters, than the sigmoid.
- There is another type of sigmoid that is also used to increase the speed of convergence.

The Hyperbolic Tangent has the same gradient as a linear activation function at zero.

But also has the benefit of continuously penalising the net input, as opposed to the linear function which is simply truncated by the quantiser at -1 and 1.



.center[ ]

Let's quickly alter the code to use tanh instead of sigmoid. Again, because we're altering the activation, we're altering the cost function. So we need a new activation of:

$$\phi(z) = \tanh(z) = \frac{2}{1+e^{-2z}} - 1$$

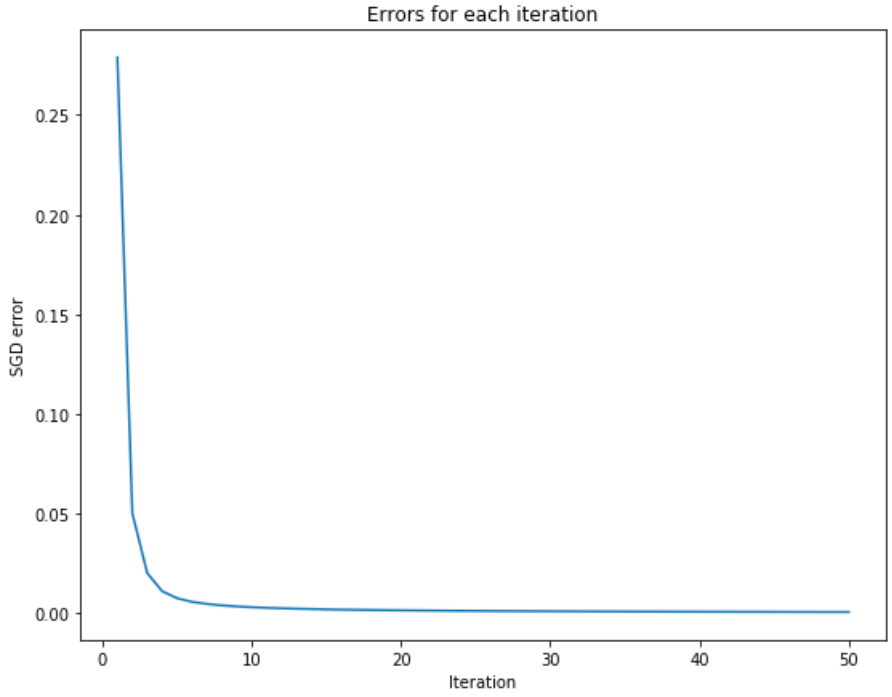
And a new update of:

$$\Delta \mathrm{w} = 1 - \tanh(z)^2$$

```
def activation(self, X):
    net_exp = self.net_input(X)
    return self._tanh(net_exp)

def _tanh(self, x):
    return np.tanh(x)

def _update_weights(self, x_i, y_i):
    phi_z = self.activation(x_i)
    error = (y_i - phi_z)
    cost = error ** 2 / 2
    z = self.net_input(x_i)                 # Need this for update
    update = error * (1 - np.tanh(z) ** 2)  # New tanh update

    # Update the weights. This is just linear gradient descent.
    self.w_[1:] += self.eta * x_i.T.dot(update) #  eta ( y - y_hat ) phi ( 1 - phi) x_i
    self.w_[0] += self.eta * update  # Update the bias weight
    return cost

def predict(self, X):
    if self.activation(X) >= 0:                    # Quantiser
        return 1.0
    else:
        return -1.0
```

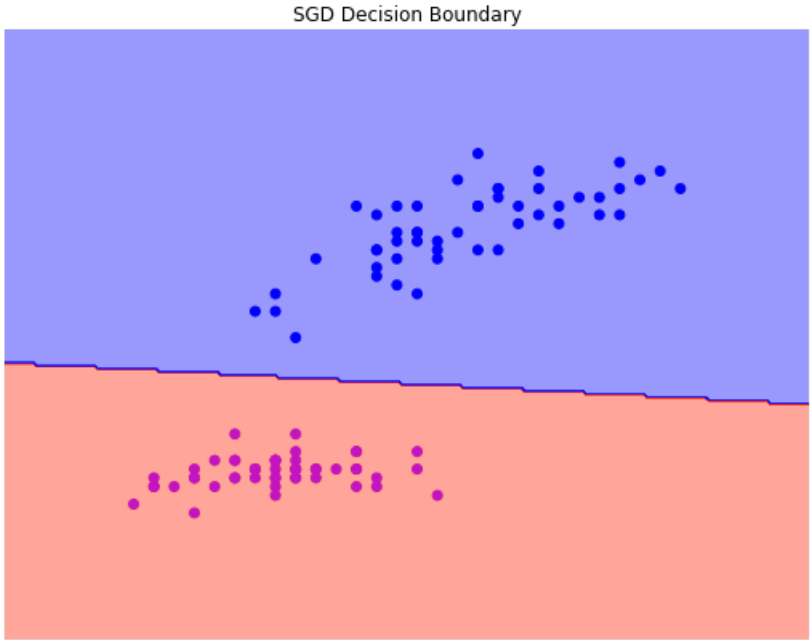
Notice how fast the convergence is now (I've kept the same number of iterations as the sigmoid for comparison. The rate of convergence is similar to the linear activation.

.left-column[



]

.right-column[



]

Workshop: 14-tanh

More Activations?

Yes, loads. And we'll see some later.

But generally the sigmoid/tanh is the standard.

All others make some approximation to a sigmoid, usually to make it faster to compute.

Why did we do this?

- The goal of this section was to prove to you that deep learning is, at it's heart, very simple.
- Deep Learning is just a combination of these Perceptrons. That's it.
- And those Perceptrons are just linear/logistic classifiers!
- Now I hope you understand why we need to understand the basics first.

Most of this you have seen before

- If you wanted, you could build your own multi-layer perceptron by feeding the output of one perceptron into another.

Key point: But instead of using our own, we're going to make use of well-optimised libraries.

You might have also noticed (e.g. when your weights are zero) you had some divide by zeros. Thankfully, someone has already take care of all that for us. :-)

Deep Learning

Deep Learning. It's all the rage. But what is it?

The *learning* part boils down to:

- Step 1: Provide a cost function
- Step 2: Calculate the gradient of the cost function
- Step 3: Iterate to profit.

???

We've covered lots of learning techniques over the last few days.

All the way from linear regression, with a closed form solution, through to SGD for applications with no single solution.

So that's the learning part done. Let's take a look at deep...

- Most machine learning problems are an attempt to find a hyperplane that separates or describes data

Most problems aren't nice. E.g. classes don't separate cleanly.

E.g. Images

So generally we've got two problems.

1. The data are so complex that we need non-linear hyperplanes to be able to separate them
2. How on earth do we reduce the number of features to make the problem computationally feasible?

???

In the examples we've seen today, and many of the examples over the last two days, the task was to pick a hyperplane that would form a decision boundary, or hyperplane of best fit for regression.

The issue is that most problems aren't that nice. They're not linear. They're not simple. Classes don't separate nicely. Etc.

The best examples of difficult data are images. They are highly dimensional (red green and blue colours for every pixel, usually with many millions of pixels).

Can you imagine trying to fit a linear classifier to a 1 million feature dataset?

So generally we've got two problems.

1. The data are so complex that we need non-linear hyperplanes to be able to separate them
2. How on earth do we reduce the number of features to make the problem computationally possible?

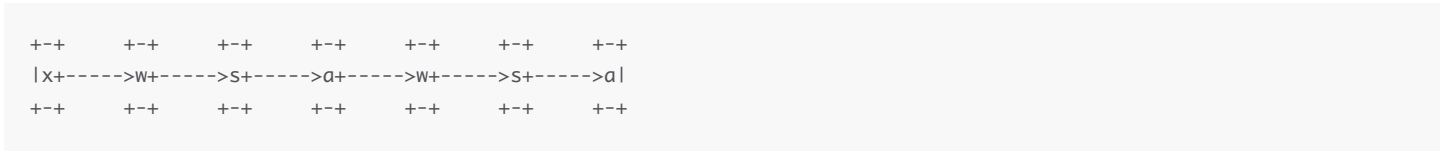
Let's tackle the former first.

Multi-Layer Perceptron

- A single Perceptron provides a linear decision boundary

Imagine we had more complex data, with (for simplicity) a single input

- We can add a second Perceptron:



- The sum's aren't needed for now, because we only have one input.

???

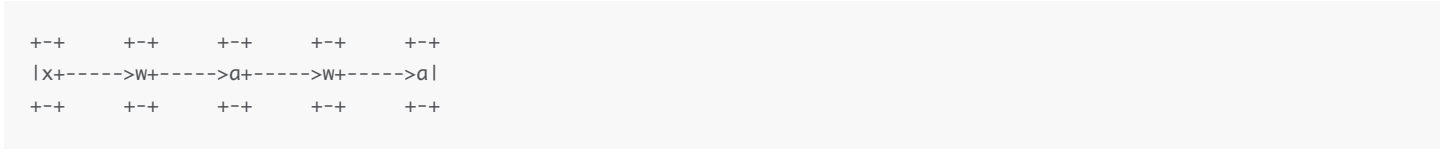
Recall that a single Perceptron is able to provide the weight to form a linear decision boundary for a single input.

Imagine that we had data that did not have a linear boundary, like the moons or circle datasets we saw in previous days.

A single Perceptron wouldn't be able to plot a decision boundary through that data.

However, if we were to add a second Perceptron to the output of the first activation function then the resultant computational graph would look like:

Where x is the input, w is the weight, s is the net input sum and a is the activation. There is only input in this graph so we don't need the sum, resulting in:



Working from the end, we can convert that into an equation:

$output = \phi(w \cdot \phi(w \cdot x))$

In other words, we're

- taking a linear combination of an input and a weight,
- transforming it by a non-linear function
- then performing some more scaling with another weight.

This is just like the kernel trick

???

The input is multiplied by a weight, transformed by an activation function, multiplied by another weight and transformed by another activation function.

In other words, we're taking a linear combination of an input and a weight, transforming it by a non-linear function then performing some more scaling with another weight.

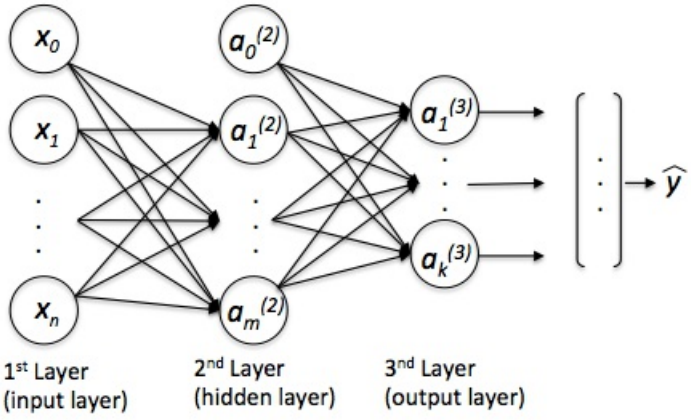
This allows us to fit shapes that are more complex than a straight hyperplane.

Let's generalise this structure to multiple inputs.

In the single Perceptron we fed the inputs into a single net-input activation combination.

In a multi-layer Perceptron we feed the inputs into all Perceptrons in a middle layer. They in turn feed them to an output layer.

A diagram of this is shown below.



.center[ ]

The layers in the middle of the MLP are known as "hidden" layers.

.bottom[(https://sebastianraschka.com)]

- The number of neurons in the hidden layer is a hyperparameter
- The number of outputs is usually chosen to match the number of target variables
- The number of inputs usually match the number of unravelled inputs
- Each new layer requires a bias layer to avoid re-normalisation

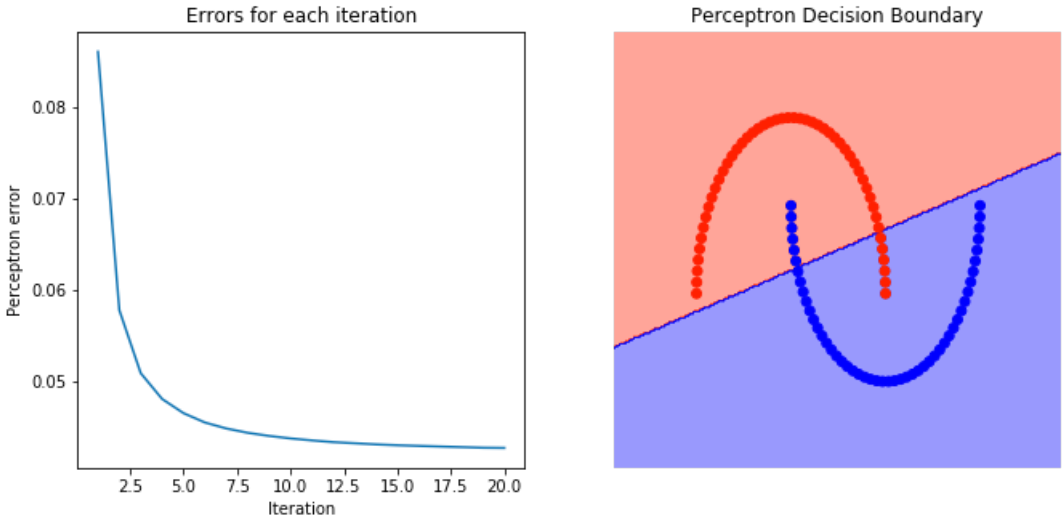
???

- The number of neurons in the hidden layer is arbitrary. Generally, the more neurons you have in the hidden layer, the more complex hyperplane you can model.
- The number of outputs is usually chosen to match the number of target variables.
- The number of input neurons typically matches the number of unravelled inputs. E.g. In our earlier examples we would need two inputs (with one more for the bias). Images would need a number matching the total number of pixels.

- Each new layer (except the output) requires a bias layer to allow the hyperplane to move in the first dimension.

Below we plot the error curve and decision boundary for the Tanh class we wrote in the previous section.

Because the Perceptron can only generate linear decision boundaries, it cannot create a decision plan complex enough to separate this data.

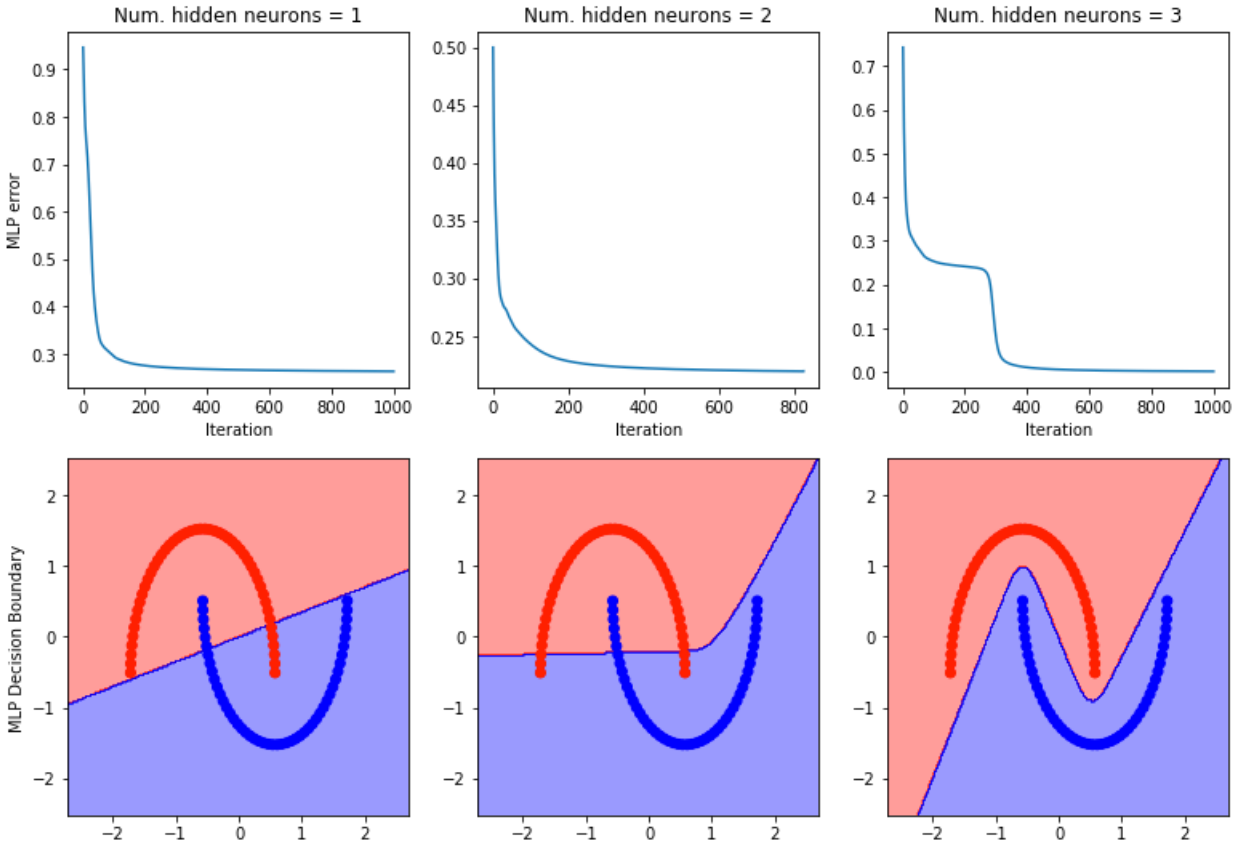


.center[ ]

Let's see how the MLP does.

This is a plot of an MLP for different numbers of hidden neurons.

We can see that for every new hidden neuron we can add a new non-linearity.



.center[ ]

Let's step back a little. What have we just done?

1) Created a multi-layer Perceptron, where all inputs were fed into a number of hidden neurons which were fed into two output neurons.

Rather than having a single neuron with 0 to represent the negative class and 1 to represent the positive, the MLP instead trains the weights to optimise the output that corresponds to a particular output neuron.

This allows the MLP to scale to any number of output classes.

---

2) Trained the MLP with the training data

This process is similar to before, except that the cost function is now a combination of two layers.

Recall that in the single layer Perceptron the cost function was: $J(\mathrm{w}) = \frac{1}{2}\sum_i (y - \phi(z))^2$?

Well in the three layer case it is now: $J(\mathrm{w}) = \frac{1}{2}\sum_i (y - \phi(w \cdot \phi(z)))^2$.

The change in cost function alters the gradient, which therefore alters the weight updates.

Key point: So how do we calculate the gradient for this complex cost function? This is the key algorithm that enables multi-layer neural networks.

Backpropagation

- In short, backpropagation is an efficient algorithm for computing the gradient of an arbitrarily complex cost function.

???

We need it because as we add more weights we are increasing the number of dimensions we have to search to find the optimum.

In contrast to the error surface seen previously, this high-dimensional cost function is so complex that we can't visualise it. And the surface is riddled with local minima and plateaus.

Backpropagation Derivation

The first trick is to remember the chain rule. For a nested function $f(g(x)) = y$ we can compute the derivative of x with respect to y by:

$\frac{\delta y}{\delta x} = \frac{\delta f}{\delta g}\frac{\delta g}{\delta x}$

???

This simplifies the problem significantly, because it now means we just need to calculate the gradient for each layer.

Next, a significant amount of academic time has been spend developing a set of techniques called *automatic differentiation*. It comes in two forms, forwards and backwards.

- Goal: Minimise the error

We could achieve this by performing forward differentiation.

It turns out that this is difficult due to the size and number of matrices produced

- Instead, we differentiate *backwards*.

Start at the output and work back. It turns out this is much faster.

???

Recall that the goal was to minimise the error. That means we need to calculate the error first. One the error has been calculated we need to work backwards to update the weights.

Once we have the error, we could perform forward differentiation to calculate the gradient.

But it turns out that we would be performing lots of matrix multiplies. (The differentiation of $\mathbf{w}\phi(z)$ results in a *Jacobian* matrix which is then multiplied by another Jacobian in the next layer).

Instead, we can get the same result by doing backward differentiation. I.e. we start at the output and work back. This results in matrix-vector multiplications, which result in another vector for the next layer. This is much faster.

This is why we use the backwards form of automatic differentiation and why backpropagation gets its name.

We're propagating the errors back through the layers via differentiation of the cost function.

I'm going to leave it there, to avoid getting into too much math.

If you'd like to know more then I'd recommend you read:

Automatic Differentiation of Algorithms for Machine Learning. https://arxiv.org/pdf/1404.7456.pdf

Key point: The main point to get across is that backpropagation is *the* most algorithm in deep learning. Without it we wouldn't be able to optimise the weights efficiently enough. This is still an active area of research.

Now, let's have a play with `sklearn` s implementation of MLP.

Workshop: 21-mlp

MLP Regression

- Backpropagation requires an error at the output to start the backwards differentiation.

Much of the results you see today are supervised

- We can also optimise for regression tasks, since they are supervised

???

The thing to note about using backpropagation is that it requires an error at the output to start the backwards differentiation. This forms the first vector.

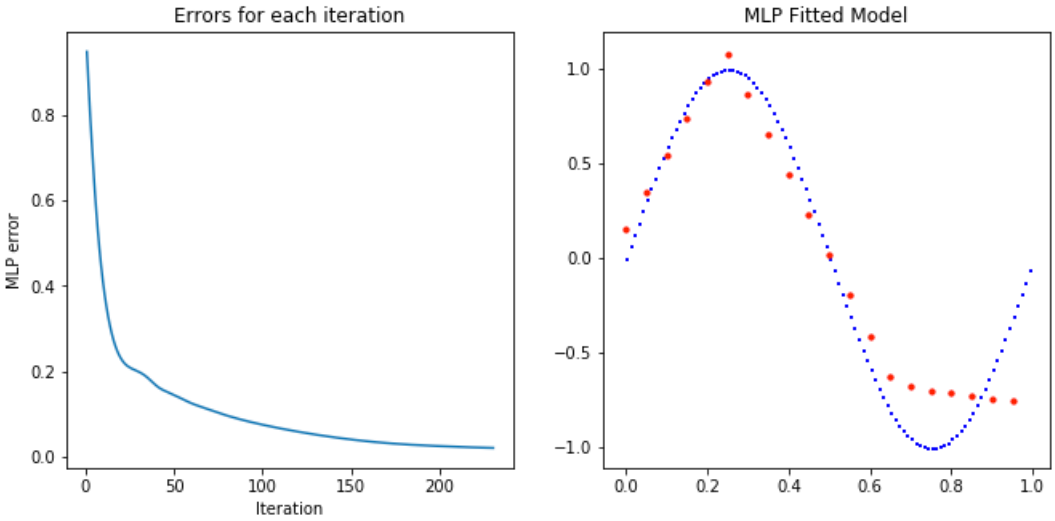
To generate an error that means we need a target. Hence, backpropagation and most deep learning algorithms are supervised only.

(There is one caveat with that statement where we set the output to be the same as the input, auto-encoders. We'll touch on this later.)

However, that means that we can also use MLP's for regression tasks, since that is supervised. We're trying to optimise the weights to fit a function.

An MLP regression task is largely the same as a classification task. We define the target, select the model and its parameters, then use SGD to optimise the weights of the model to fit the target.

The result of fitting a model to a sine wave is shown below.



Let's implement this now, to get you used to tuning MLP hyperparameters.

Workshop: 22-mlp-regression

Interlude: Keras

Just before we start with autoencoders, let's introduce a new library called Keras (https://keras.io).

Keras is a high-level deep learning library. It is unique because it can take advantage of a number of backends, the most popular being Tensforlow and Theano.

It's great for testing out models because it is very high level; it's very easy to try new structures with minimal changes.

Keras has several core classes:

Class Name	Description
`Sequential`	The base for all models. A linear stack of layers.
`Dense`	A standard layer of neurons (i.e. wx)
`Activation`	An activation function

To use the model we need to create a `Sequential` instance, add dense layers and activations, then compile and fit

the model.

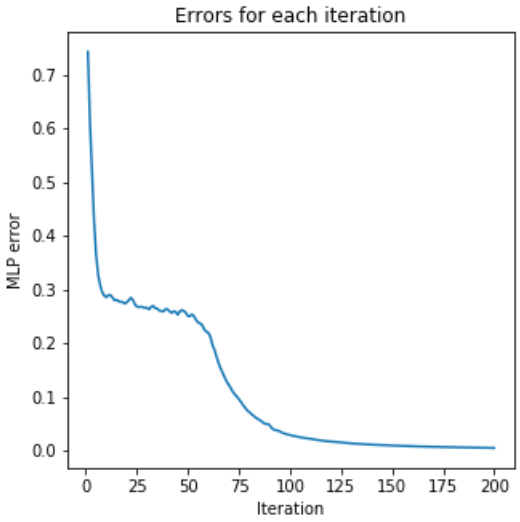
```
# Create the model
model = Sequential()

# Add hidden layer
model.add(Dense(3, input_dim=2))

# Add activation
model.add(Activation('tanh'))

# Add output layer
model.add(Dense(1))

# Add activation
model.add(Activation('sigmoid'))

# Compile the model
model.compile()

# Fit the model
history = model.fit(x_train, y_train, epochs=200, verbose=0, shuffle=True)
```

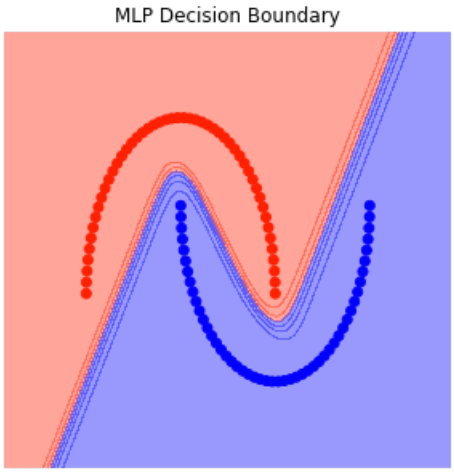
However, let's make create an MLP that is equivalent to the `sklearn` implementation...

```
# Create the model
model = Sequential()

# Create special intialisation (to be the same as sklearn)
initialisation=glorot_uniform()

# Add hidden layer
model.add(Dense(3, input_dim=2, bias_initializer=initialisation))

# Add activation
model.add(Activation('tanh'))

# Add output layer
model.add(Dense(1, bias_initializer=initialisation))

# Add activation
model.add(Activation('sigmoid'))

# Create optimiser
adam = Adam(lr=0.05)

# Compile the model
model.compile(loss='binary_crossentropy', optimizer=adam)

# Fit the model
history = model.fit(x_train, y_train, epochs=200, verbose=0, shuffle=True)
```

(I had to go crawling through the sklearn code to find that they used a special initialisation!)

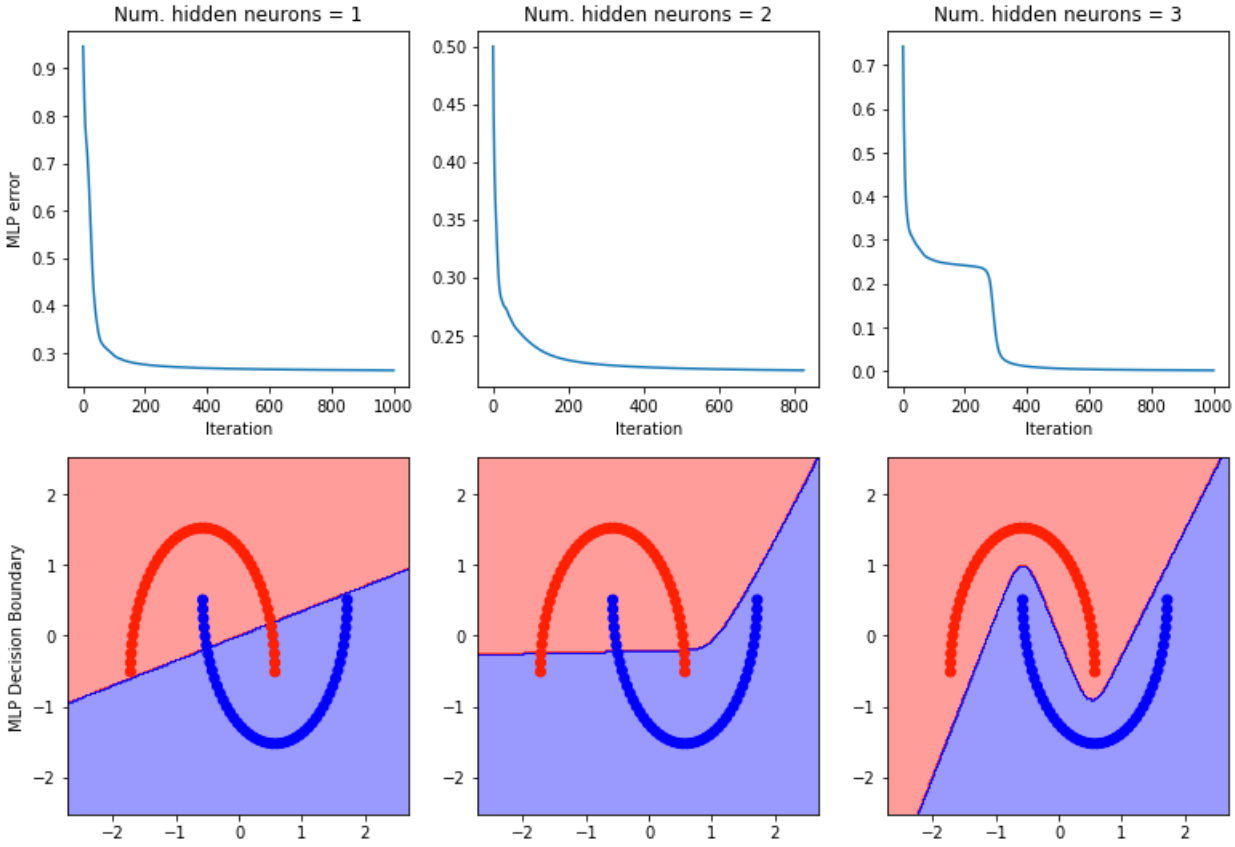
And this is the result:





.center[ ]

Comparing that to the `sklearn` results...



.center[ ]

It seems to have converged faster. So not quite equivalent. But almost. Probably some difference in the optimiser.

Let's get some experience with keras before we deal with autoencoders...

Workshop: 23-keras

Autoencoders

.left-column[

Remember SOMs?

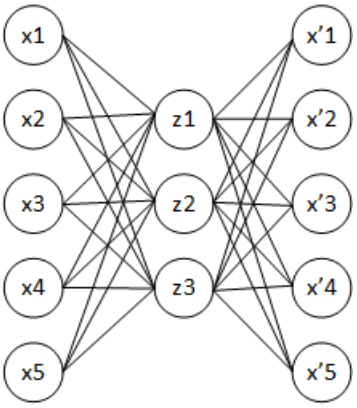
- Autoencoders constrain the numbers of neurons to learn a lower dimensional equivalent

Autotencoders can be thought of as a generalisation of SOMs.

- The final layer has the same number of outputs as inputs
- The result is compared to the input

I.e. the target is the input data

]

.right-column.center[

]

???

Remember Self Organising Maps (SOM)? SOMs take a high dimensional dataset and map inputs to weights with a lower number of dimensions.

Autoecoders are an extention of that idea. They take a high dimensional input feed the outputs into a lower-dimensional input.

This constrains the the amount of information the neural network can hold. It is forced to make generalisations of the data.

The final layer has the same number of outputs as inputs to the network.

The task is to optimise the weights so that even when the dimensionality is compressed it can still produce a satisfactory version of the input.

I.e. The target variable is the input data.

Let's use the digits dataset, create an autoencoder and visualise the results.

(Using a visual dataset for testing architectures is more difficult due to the inherent dimensionality in images. But it provides a great intuitive feel if you can "see" what is happening.)

Each image has 28×28 pixels.

```
n_inputs = 784
```

We create a sequential model (you could also use the `Model` api) into which we feed the data into 32 neurons.

```
# Create the model
model = Sequential()
model.add(Dense(32, input_dim=n_inputs,
```

```
                    bias_initializer='glorot_uniform', activation='relu'))
```

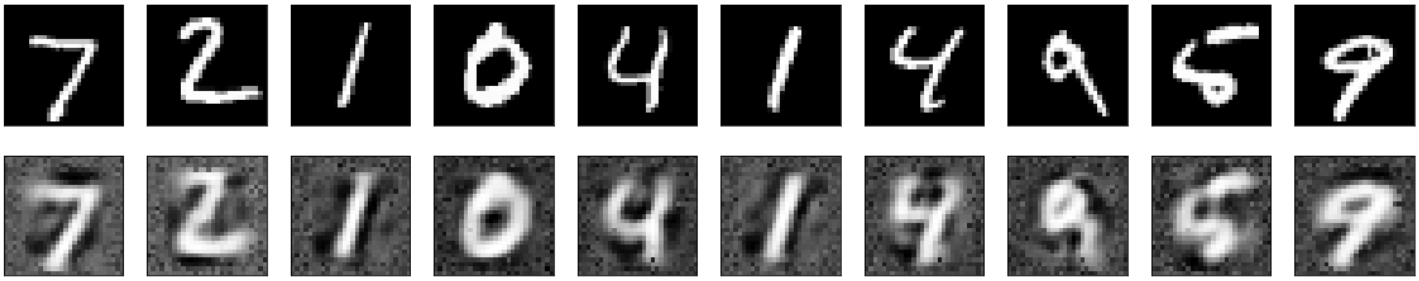
All of the image data has to pass through this low, 32-dimensional space. This represents a compression of the original data. This is fairly easy to imagine. There's pixels with no information around the edges.

The output layer is set to have the same number of neurons as the input. Then the input is compared to the output and this is the error.

The code is below. Even in this simple model there are many hyperparameters to choose from.

```
model.add(Dense(n_inputs, bias_initializer='glorot_uniform', activation='sigmoid'))

# Compile the model
model.compile(loss='binary_crossentropy', optimizer='adadelta')
```

These are my results for a simple autoencoder.



- Not used as much as you'd expect
- Custom solutions already do "good enough". For example
  - image compression
  - t-SNE

But more importantly

- Should never replace standard feature analysis
- Very useful for data denoising
- Area of active research. Waiting for a breakthrough.

???

Interestingly, despite their obvious promise, autoencoders aren't used that much.

For tasks that require compression, e.g. images, image compressors already do a pretty good job. It's hard to train an autoencoder that does a better job that jpeg compression, for example.

Secondly, there is little use in replacing standard feature analysis and enhancement with autoencoders because the manual step usually provides insights into the data that surpasses any value in the time saved by using autoencoders.

Thirdly, other compression techniques exist that at least do as well and are a lot simpler to build. E.g. t-SNE.

But there are some uses. It is useful for data denoising. I.e. if you have masses of complex noisy data, there's not many algorithms that will automatically try and generalise the data model and produce a better output.

This is still an active area of research and we have only touched the surface. Keras has a great blog on the topic: https://blog.keras.io/building-autoencoders-in-keras.html

Workshop: 24-autoencoders

Now you try. Feel free to try a variation of this. See who can get the lowest validation score.

Convolutional Neural Networks

- CNN's are a mix of neural networks and convolution (an old signal processing technique)
- Inspired by the visual cortex
- CNN's form most of the excitement (because it most tasks performance scores are better than humans)

???

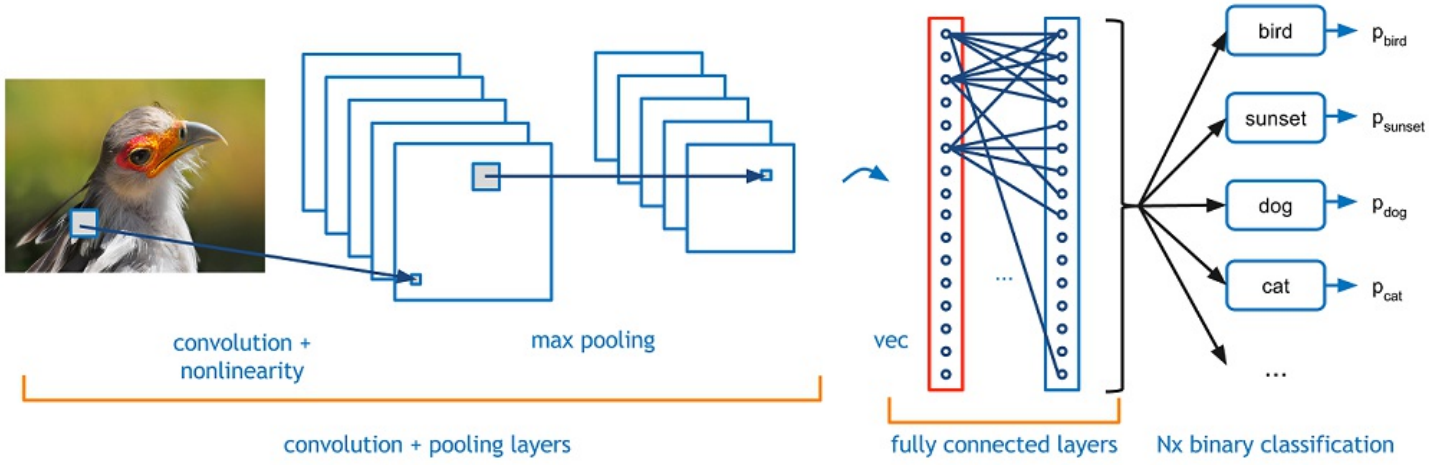
Neural networks began by emulating biological structures.

Even today, most advances take hints from animals, where natural selection has produced exquisitely specialised structures to perform everyday jobs.

The design of convolutional neural networks (CNN) takes inspiration from the visual cortex.

And CNNs form the bulk of the excitement in the news today.

Key difference: Connectivity is limited spatially. Parts of the CNN will only look at subset of the image



???

The implementation will seem quite similar to what we have seen already. Each layer feeds into the next with ever decreasing numbers of neurons.

But one key difference is that the connectivity between nodes is limited spatially. In other words a parts of the CNN will only look at a small area of the image (or sound, or whatever).

The first step in the process is convolution.

.center[

]

The main parameters in this process are:

1. The number of filters (depth) (i.e the size of the square)
2. The "stride"
3. Zero padding (generally not used)

.bottom[http://cs.nyu.edu/~fergus/tutorials/deep_learning_cvpr12/]

???

Convolution is the first step in the process. See the image for a nice visual description of how convolution works.

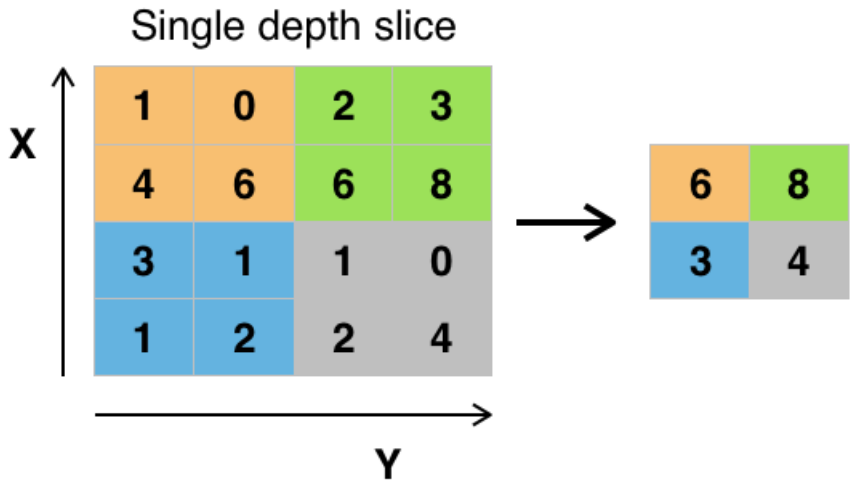
Next there is an activation layer. Often this is the "ReLU" activation function, the rectified linear unit.

This is required to add a nonlinearity to the data, so we can learn nonlinear functions.

ReLU's have been shown to work well with images. (because black is "off")

- Pooling is just another word for downsampling

The pooling algorithm can be altered, but often "max pooling" is chosen. This picks the largest value in a region.

.center[

]

???

Next, there is the so called "pooling" step. Which is basically just downsampling.

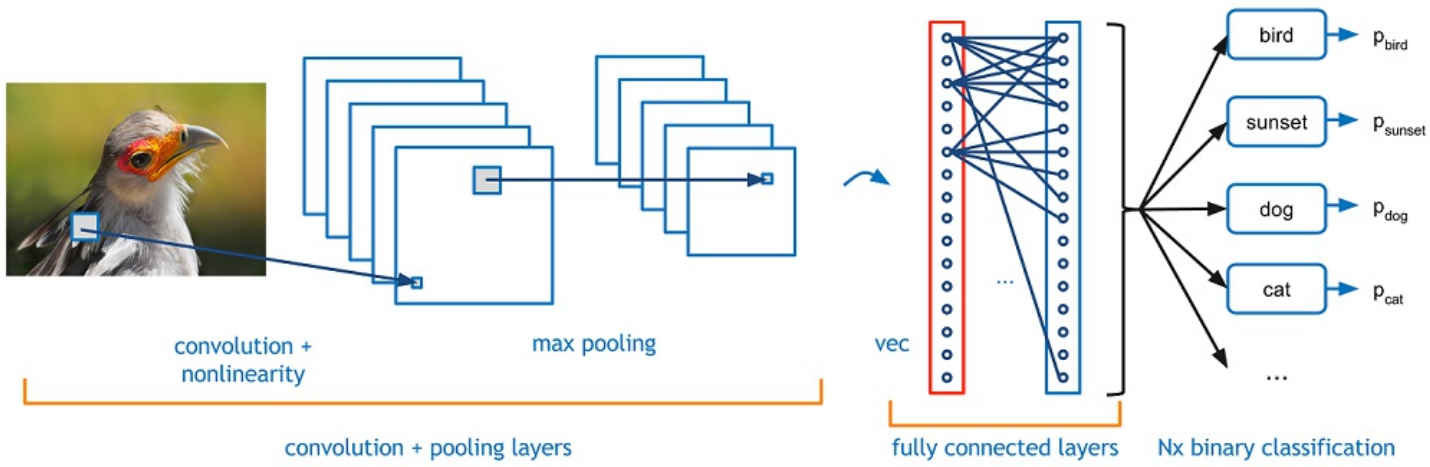
- "Fully connected" layer at the output

This is just a standard multi-layer perceptron (MLP). The outputs of the MLP are the targets of the supervised

task.

???

Finally, after a number of convolution, activation and pooling layers, there is a "fully connected layer".



???

So there we have it. Let's look at a really simple implementation...

First we create a model and add a `Conv2D` layer. This is the convolution layer.

```
model = Sequential()
model.add(Conv2D(32, kernel_size=(3, 3),
                 activation='relu',
                 input_shape=input_shape))
```

Next `MaxPooling2D` downsamples the data with the "max pooling" algorithm.

```
model.add(MaxPooling2D(pool_size=(2, 2)))
```

Then we add a `Dropout` layer. We haven't talked about this yet. But a dropout layer attempts to remove neurons with low weights. It's an anti-overfitting measure.

```
model.add(Dropout(0.25))
```

We then need to `Flatten` the data so that the 2-D image data can squeeze into the MLP.

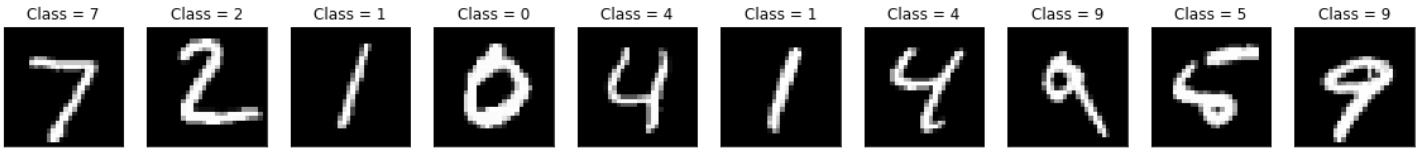
```
model.add(Flatten())
```

Then we add the MLP (32 neurons in the hidden layer?) with a `softmax` activation on the output to predict the (absolute) classes.

```
model.add(Dense(32, activation='relu'))
model.add(Dropout(0.25))
model.add(Dense(num_classes, activation='softmax'))

model.compile(loss='categorical_crossentropy',
              optimizer='adadelta',
```

```
              metrics=['accuracy'])
```

The results are below. Perfect!



Now it's your turn to get some experience with CNNs.

Workshop: 25-convnets

Other Neural Net Structures

Recurrent Neural Networks

- RNNs have a similar feed-forward structure as MLPs.
- But a delayed feedback path is introduced

Similar to ARIMA models that depended on previous inputs

- Layers are dependent on the past outputs of other layers
- Issue is that we can't train layers individually (because of the feedback)
- Very hard to converge

In practice, truncated RNNs have been developed

Most people use *Long-Short Term Memory* (LSTM) RNNs.

???

Recurrent Neural Networks (RNN) have the same feed-forward structure as MLPs. But the introduce a delayed feedback path. I.e. the outputs are fed back to the inputs with a time delay.

The idea here is similar to the ARIMA models we saw earlier. The next output depends not only on the current inputs, but also the previous inputs.

The issue with RNNs is that the we can't pre-train layers individually, because layers have a temporal dependence which influences other layers. This makes it very hard to converge.

In practice, truncated RNNs have been developed to overcome this. The most popular is Long-Short Term Memory (LSTM) RNNs.

These work very well for any inputs that have a temporal nature. E.g. audio, video, writing, etc.

Reinforcement Learning

RL is interesting because it changes the goals, rather than the structure

- RL's goal is to maximise a final state (some measure of performance)...
- based upon a reward for the current state...
- and making predictions about the future state.

I.e. pick a set of actions that produces the most reward

The underlying structure can vary significantly. E.g. video games use some sort of CNN.

RL is based upon Markov chains of expected states

???

This is slightly different, in that it alters the learning process and goals, rather than just changing architectures.

The idea is to maximise a final state, usually derived from some measure of performance, like a score in a video game.

To achieve a high score, the models are rewarded according to the current state, and the predicted future state.

Hence, the model is trained to pick the best action that would result in a future reward at any given point in time.

The NN structure doesn't really matter, but most often we see examples with video games, which means a CNN is usually used.

But the important thing to note is the change from a static cost function, to a stochastic prediction of future states (via a Markov Chain).

Adversarial Learning

Again, an interesting model because it changes the goal

- Aversarial networks are actually two models competing against each other
- The first generates data (the generator)
- The second evaluates data (the evaluator)

The goal of the generator is to *increase the error rate of the evaluator*

In other words, the generator is trying to fool the evaluator

???

Continuing the idea of altering how a model learns, adversarial learning is a setup where you have two competing models.

One model is the "generator". The other is the "evaluator". This is called a Generative Adversarial Network (GAN), but the idea can be generalised.

The job of the evaluator is to verify and evaluate the data that has been generated by the generator.

The goal of the generator is to increase the error rate in the evaluator, yes, *increase*.

In other words, the generator is trying to fool the evaluator into thinking that the data is real.

This can be thought of as a type of auto-encoding. If the generator can accurately model the underlying data, then it should be able to generate data that perfectly mimics the original.

You will see this type of learning in "style transfer" type examples. But I've seen some really cool examples

generating music (e.g. @feynmanliang's PhD on recreating Bach)

Boltzmann machine, Restricted Boltzmann Machine and Deep belief networks

Some books start with BMs, RBMs and then DBNs. They're not used that often and I think that they confuse matters.

- RBMs are structures of *undirected* neurons. I.e. information can flow in either direction.

The beauty is that they are unsupervised and trained to model the input data.

Key point: They don't scale past a handful of nodes.

And even then, they take a lifetime to converge.

???

Some books start with BMs, RBMs and then DBNs. They're not used that often and I think that they confuse matters.

With that said, they are important as they are generally believed to be the "best" NN you can define, but are impractically difficult to train.

RBMs are structures of neurons, like always, but the connections are undirected. In the most basic form, the BM, all nodes are connected to all other nodes.

In the restricted BM, layers are formed, but the connections are still undirected. DBNs are deeper versions of RBMs.

The beauty is that they are unsupervised. I.e. they are a form of auto-encoder, or GAN. They are trained so that they are able to model the input data, no matter what the input data is. This makes it a great general purpose computational medium.

The problem is that they don't scale. To the point where it's very hard to get anything but a trivial example working. It takes a crazy amount of time for convergence due to the knock-on effects caused by undirected connections. Also, they don't perform well in the presence of noise (since noise perturbates all other neurons due to the undirected connections, again).

And there we have it. Zero to Deep learning in half a day.

I've barely scratched the surface. But hopefully you have a good idea of what deep learning is and when you would use it. Here are some tips on using DL:

- You need lots of data
- Hyperparameter tuning is vitally important
- It's computationally expensive

But:

- It produces groundbreaking results in complex datasets

Generally:

- Start with simpler methods. E.g. t-SNE, some simpler classification method, etc.

Working with Text

- A significant portion of data isn't encoded. E.g. text.

How do we work with text?

- Like before, feature engineering is vitally important
- And the greatest problem is converting text into features

We can't use the same techniques that we learnt earlier

???

Previously we worked with image data. It's very complex, but is provided in a digitised form. I.e. pixels represent colour and intensity numerically.

What about data that isn't provided in a numerical form? Data like text?

In this section we're going to be talking about working with textual data. Indeed one of the greatest sources of information on the planet, the internet, is all text.

It turns out that the biggest hurdle is converting text into features that can be understood by traditional algorithms

As you might expect, the techniques we've discussed to manage features don't apply to text. So we need to develop new techniques for cleaning and transforming text.

Let's look at cleaning first.

Cleaning Text

- How many people speak perfect English?
- What about the raw text formats (e.g. HTML)

Many problems are lucky to have data that is corrupted by noise. Text is corrupted by humans!

- This section will use a "trolling" dataset

???

Imagine all the forums out there. All the tweets. All the reviews.

How many of them do you think use perfect English?

What's more, text data can often be wrapped in coded messages. For example you might have text that has unicode values or embedded html.

In this section we'll be using a dataset from a Kaggle competition where the goal was to create a model that accurately detects insults.

https://www.kaggle.com/c/detecting-insults-in-social-commentary

Those that have weak hearts should look away now.

The data comprises of three columns. A target variable stating whether the message was an insult or not, a date

and the message. Some examples can be seen below. (Questionable classifications on both sides.)

Negative examples:

```
0,,"""The FBI,CIA,DOJ let 6 gaylords bury this country, Thet suck hot dogs"""
0,20120530031921Z,""""""pretty sure"" don't cut it pal. \xa0Read his first book, published long before he dreamed of
being president."""
0,,"""my screen is stuck in the black screen help somebody"""
0,20120619010607Z,""".............RON PAUL....is SCARY WRONG....."""
0,20120618222716Z,"""WE ALL KNOW THEM JESUS FREAKS ARE THE BIGGEST FREAKS IN THE WORLD! (SEXUALLY)"""
0,20120528081016Z,"""At least you're sticking to your guns.<br>"""
```

Positive examples:

```
1,20120619213024Z,"""It was his idiot parents, idiot."""
1,20120527213529Z,"""You were born a bald-headed freak."""
1,20120609181200Z,"""Rob...you're one pathetic asshole...and ""libs"" aren't responsible for that."""
1,20120502053501Z,"""Its kind of frightening knowing dbags like you exist."""
1,20120618192155Z,"""You fuck your dad."""
1,,"""shut the fuck up. you and the rest of your faggot friends should be burned at the stake"""
```

- Classifications are subjective
- Step back: Never be perfect because some of the targets are wrong

Aside: If we were collecting the data, would a measure of how bad an insult was help?

- Look at all the junk

???

So, firstly, it seems like the classifications are quite subjective. Depending on who classifies the data, you will get a different result.

This is going to affect the baseline. We're never going to be able to perform this task perfectly because some of the targets are wrong.

If this was our dataset, maybe we could mitigate against that by having a classification of *how bad* each insult was.

Secondly, you can see all kinds of junk in there. Some of it is bad punctuation. Some is html. Some have no date. Some have unicode. Etc.

So how do we clean all that data?

Removing Rubbish

```
0,20120619094753Z,"""C\\xe1c b\\u1ea1n xu\\u1ed1ng \\u0111\\u01b0\\u1eddng bi\\u1ec3u t\\xecnh 2011 c\\xf3 \\xf4n
ho\\xe0 kh\\xf4ng ? \\nC\\xe1c ng\\u01b0 d\\xe2n ng\\u1ed3i cu\\xed \\u0111\\u1ea7u chi\\u1ee5 nh\\u1ee5c c\\xf3
\\xf4n ho\\xe0 kh\\xf4ng ?\\nC\\xe1c n\\xf4ng d\\xe2n gi\\u1eef \\u0111\\u1ea5t \\u1edf V\\u0103n Giang, C\\u1ea7n
Th\\u01a1 c\\xf3 \\xf4n ho\\xe0 kh\\xf4ng ?\\n.................\\nR\\u1ed1t cu\\u1ed9c \\u0111\\u01b0\\u1ee3c
g\\xec\\xa0 th\\xec ch\\xfang ta \\u0111\\xe3 bi\\u1ebft !\\nAi c\\u0169ng y\\xeau chu\\u1ed9ng ho\\xe0 b\\xecnh,
nh\\u01b0ng \\u0111\\xf4i khi ho\\xe0 b\\xecnh ch\\u1ec9 th\\u1eadt s\\u1ef1 \\u0111\\u1ebfn sau chi\\u1ebfn tranh
m\\xe0 th\\xf4i.\\nKh\\xf4ng c\\xf2n con \\u0111\\u01b0\\u1eddng n\\xe0o ch\\u1ecdn kh\\xe1c \\u0111\\xe2u,
\\u0111\\u1eebng m\\u01a1 th\\xeam n\\u01b0\\xe3."""
```

- Regular expressions (regex) are very powerful. They allow you to specify a pattern to which text must match.
- Be very careful about what you remove. It could be informative.
- Test every removal
- Generally prefer keeping as much as possible

???

Look at this one: Looks like Chinese or something.

Regular expressions (regex) are very powerful. They allow you to specify a pattern to which text must match.

One of the simplest, and most effective, things we can do is systematically use regexs to find a remove rubbish data.

But by removing data, you could be removing information, so we have to be careful.

Here's a batch of test data:

```
1      "i really don't understand your point.\\xa0 It ...
4      "C\\xe1c b\\u1ea1n xu\\u1ed1ng \\u0111\\u01b0\\u1edd...
5      "@SDL OK, but I would hope they'd sign him to ...
19        "Your a retard go post your head up your #%&*"
21          "http://www.youtube.com/watch?v=tLYLLPHKRU4"
113    "POLITICAL CORRECTNESS:\\n\\nPolitical Correctne...
174    "GALLUP DAILY\\nMay 24-26, 2012 \\u2013 Updates ...
183    "@nilbymouth \\\\n\\\\nYou are right concerning Ca...
Name: Comment, dtype: object
```

First, we can make everything lowercase. This reduces the number of characters. BUT WE MIGHT BE LOSING SOME INFORMATION ABOUT CAPS LOCK RAGE!

```
def clean(X):
    # Lowercase everything
    X = X.str.lower()
```

Next we should bet rid of those duplicate backslashes.

```
    # Get rid of those duplicate backslashes
    X = X.str.replace(r'\\\\', r'\\', case=False)
```

Before:

```
1      "i really don't understand your point.\\xa0 It ...
4      "C\\xe1c b\\u1ea1n xu\\u1ed1ng \\u0111\\u01b0\\u1edd...
5      "@SDL OK, but I would hope they'd sign him to ...
19        "Your a retard go post your head up your #%&*"
21          "http://www.youtube.com/watch?v=tLYLLPHKRU4"
113    "POLITICAL CORRECTNESS:\\n\\nPolitical Correctne...
174    "GALLUP DAILY\\nMay 24-26, 2012 \\u2013 Updates ...
183    "@nilbymouth \\\\n\\\\nYou are right concerning Ca...
Name: Comment, dtype: object
```

The result:

```
1      "i really don't understand your point.\\xa0 it ...
4      "c\\xe1c b\\u1ea1n xu\\u1ed1ng \\u0111\\u01b0\\u1edd...
5      "@sdl ok, but i would hope they'd sign him to ...
19        "your a retard go post your head up your #%&*"
21          "http://www.youtube.com/watch?v=tlyllphkru4"
113    "political correctness:\\n\\npolitical correctne...
174    "gallup daily\\nmay 24-26, 2012 \\u2013 updates ...
183    "@nilbymouth \\n\\nyou are right concerning camp...
```

Next we can remove all the whitespace and ditch any remaining unicode.

Not that you might need to use unicode if your working with data that doesn't have an ascii equivalent.

```
    # Remove whitespace
    X = X.str.replace('"', ' ')
    X = X.str.replace('_', ' ')
    X = X.str.replace('-', ' ')
    X = X.str.replace(r'\n', ' ')
    X = X.str.replace(r'\\n', ' ')
    X = X.str.replace(r'\t', ' ')
    X = X.str.replace(r'\\t', ' ')
    X = X.str.replace(r"\\xa0", ' ') # A space
    X = X.str.replace(r"\\xc2", ' ') # A space
    X = X.str.replace(' +', ' ')

    # Ditch all other unicode
    X = X.str.decode("unicode_escape").str.encode('ascii', 'ignore').str.decode("utf-8")
```

Before:

```
1      "i really don't understand your point.\\xa0 it ...
4      "c\\xe1c b\\u1ea1n xu\\u1ed1ng \\u0111\\u01b0\\u1edd...
5      "@sdl ok, but i would hope they'd sign him to ...
19        "your a retard go post your head up your #%&*"
21          "http://www.youtube.com/watch?v=tlyllphkru4"
113    "political correctness:\\n\\npolitical correctne...
174    "gallup daily\\nmay 24-26, 2012 \\u2013 updates ...
183    "@nilbymouth \\n\\nyou are right concerning camp...
```

After:

```
1       i really don't understand your point. it seem...
4       cc bn xung ng biu tnh 2011 c n ho khng ? cc n...
5       @sdl ok, but i would hope they'd sign him to ...
19         your a retard go post your head up your #%&*
21           http://www.youtube.com/watch?v=tlyllphkru4
113     political correctness: political correctness ...
174     gallup daily may 24 26, 2012  updates daily a...
183     @nilbymouth you are right concerning campbell...
```

Let's try to remove all the contractions:

```
    # Remove contractions
    X = X.str.replace("won't", "will not")
    X = X.str.replace("can't", "can not")
```

```
    X = X.str.replace("don't", "do not")
    X = X.str.replace("i'm", "i am")
    X = X.str.replace(" im", " i am")
    X = X.str.replace("ain't", "is not")
    X = X.str.replace("'ll", " will")
    X = X.str.replace("'t", " not")
    X = X.str.replace("'ve", " have")
    X = X.str.replace("'re", " are")
    X = X.str.replace("'d", " would")
```

Before:

```
1       i really don't understand your point. it seem...
4       cc bn xung ng biu tnh 2011 c n ho khng ? cc n...
5       @sdl ok, but i would hope they'd sign him to ...
19         your a retard go post your head up your #%&*
21           http://www.youtube.com/watch?v=tlyllphkru4
113     political correctness: political correctness ...
174     gallup daily may 24 26, 2012  updates daily a...
183     @nilbymouth you are right concerning campbell...
```

After:

```
1       i really do not understand your point. it see...
4       cc bn xung ng biu tnh 2011 c n ho khng ? cc n...
5       @sdl ok, but i would hope they would sign him...
19         your a retard go post your head up your #%&*
21           http://www.youtube.com/watch?v=tlyllphkru4
113     political correctness: political correctness ...
174     gallup daily may 24 26, 2012  updates daily a...
183     @nilbymouth you are right concerning campbell...
```

And finally convert some of the raw text into tokens. For example urls, swearwords and smileys.

```
    # Create tokens of interest
    X = X.str.replace(r"([#%&\*\$]{2,})(\w*)", r"_SW") # Swearword obfuscations
    X = X.str.replace(r" [8x;:=]-?(?:\)|\}|\]|>){2,}", " _BS") # Big smileys
    X = X.str.replace(r" (?:[;:=]-?[\)\}\]d>])|(?:<3)", " _S") # Smileys
    X = X.str.replace(r" [x:=]-?[\(\[\|\\/\{<]', r", " _F") # Sad faces
    X = X.str.replace(r" [x:=]-?(?:\(|\[|\||\\|/|\{|<){2,}", " _BF") # Big Sad faces
    X = X.str.replace(r"(@[a-z]+)", r"_AT") # Directed at someone
    X = X.str.replace(r"[\w\-][\w\-\.]+@[\w\-][\w\-\.]+[a-zA-Z]{1,4}", r"_EM") # Email
    X = X.str.replace(r"\w+:\/\/\S+", r"_U") # URL
```

Before:

```
1       i really do not understand your point. it see...
4       cc bn xung ng biu tnh 2011 c n ho khng ? cc n...
5       @sdl ok, but i would hope they would sign him...
19         your a retard go post your head up your #%&*
21           http://www.youtube.com/watch?v=tlyllphkru4
113     political correctness: political correctness ...
174     gallup daily may 24 26, 2012  updates daily a...
```

```
183     @nilbymouth you are right concerning campbell...
```

After:

```
1       i really do not understand your point. it see...
4       cc bn xung ng biu tnh 2011 c n ho khng ? cc n...
5       _AT ok, but i would hope they would sign him ...
19          your a retard go post your head up your _SW
21                                                  _U
113     political correctness: political correctness ...
174     gallup daily may 24 26, 2012  updates daily a...
183     _AT you are right concerning campbell. my bad...
```

- There are a few things left that we could have done. That foreign unicode example is pretty useless. We could have wrote something to detect text with too many unicode characters. But I'm hoping that the result is so random that it won't actually affect the results that much.

It might actually be worth removing this

- There's probably more manual tokens to find within the data. E.g. dates. I would think that a date might be a reasonable predictor of not trolling.
- More deep thought about what we're trying to do. E.g. I think it would be very easy to create a dictionary of bad troll words. Like ones used on simple message boards to filter out swearwords. Or really bad "I can't say that to my mum" words. These are probably a great indicator of trolling.
- Languages. Damn those Americans/English for their language 🇦🇳🇦🇳.

Workshop: 31-text-cleaning

Text Representation

This section discusses a range of techniques to transform the text into features that can be parsed by algorithms. First let's define the terminology:

- Corpus: a collection of documents
- Document: a single instance of text. It could be a tweet or a 100-page document.
- Token (or term): a document is comprised of individual tokens, usually words or symbols representing features.

Bag of Words

The simplest way to represent the data is to have one feature, one column, that represents the individual token.

Each row represents a document.

Each cell represents whether a word appears in a document.

and	apples	are	dad	do	fuck	it	oranges	point	really	seems	that	understand
0	0	0	1	0	1	0	0	0	0	0	0	0
1	1	1	0	1	0	1	1	1	1	1	1	1

This is the same as one-hot encoding

Term Frequency

A step up from bag of words is counting the number of times that a word occurs in a document.

and	are	at	be	friends	now	of	rest	should	shut	stake	the	up	where	yea
1	1	0	0	0	1	0	0	0	0	0	0	0	1	1
1	0	1	1	1	0	1	1	1	1	1	3	1	0	0

This is the same as dictionary vectorisation

Stemming

You will see in this data that we have a plural. If we also saw a singular version, it would be counted as a new token.

Quite often we perform *stemming* to reduce a word down to its base. There are better stemmers from `nltk` but let's keep it simple for a moment:

```
def stemming(X):
    X = X.str.replace("ies( |$)", "y ")  # Plurals
    X = X.str.replace("s( |$)", " ")  # Plurals
    X = X.str.replace("ing( |$)", " ")  # adverbs
    X = X.str.replace("ed( |$)", " ")  # Past tense
    X = X.str.replace("your( |$)", "you ") # Personal
    X = X.str.replace("our( |$)", "us ") # Personal
    return X
```

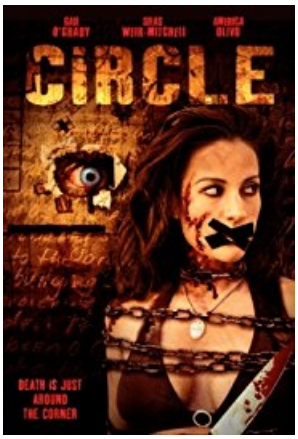
There's probably lots more we can think of, but we don't know how they will affect the result yet.

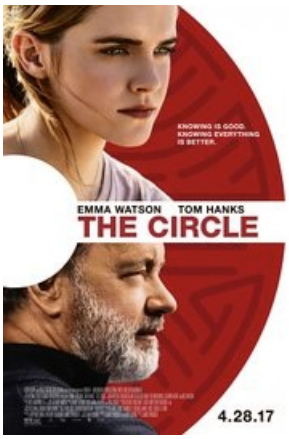
and	are	at	be	friend	now	of	rest	should	shut	stake	the	up	where	yeal
1	1	0	0	0	1	0	0	0	0	0	0	0	1	1
1	0	1	1	1	0	1	1	1	1	1	3	1	0	0

Stopword Removal

Stopwords are words used to segment a sentence. They are very common in english and do not provide much information, except in certain circumstances.

E.g. Imagine you were performing sentiment analysis on films...

.center[



]

```
def stopwords(X):
    X = X.str.replace("the( |$)", " ")
    X = X.str.replace("and( |$)", " ")
    X = X.str.replace("of( |$)", " ")
    X = X.str.replace("on( |$)", " ")
    return X
```

Normalising

Some algorithms prefer normalised values rather than counts (e.g. neural networks).

So often the frequencies are divided by the total number of words.

Importance of Words

Depending on your task, words that occur very often or very rarely may not be of interest.

For example, if you were tasked with document classification, then a single instance of a token in a corpus adds no value. They won't cluster with any other documents.

Similarly, tokens that appear too frequently will be common between many documents and add no discriminatory power.

So quite often, lower and upper limits are imposed on a corpus.

Inverse Document Frequency

If some tokens occur repeatedly in only a few documents, then it is likely that this token is important to that document.

For example, the word prehensile probably doesn't occur much in normal literature, but it is vitally important to the studiers of Apes (a Primatologist, apparently!).

This quality is represented by the *Inverse Document Frequency*

$$IDF(t) = 1 + \log\left(\frac{\text{total number of documents}}{\text{number of documents containing t}}\right)$$

I.e. rare terms get a boost.

TF-IDF

What has emerged as the holy grail of text feature encoding is the combination of term frequencies and inverse document frequencies.

$$TFIDF(t,d) = TF(t,d) \times IDF(t)$$

```
tfidf_vect = TfidfVectorizer(analyzer = "word", tokenizer=None,
    preprocessor=None, stop_words=None, max_features=5000)
X_train_counts = tfidf_vect.fit_transform(X[6:8])
```

are	at	be	friend	now	rest	should	shut	stake	up	where	yeah	you
0.47	0.0	0.0	0.0	0.47	0.0	0.0	0.0	0.0	0.0	0.47	0.47	0.34
0.0	0.28	0.28	0.28	0.0	0.28	0.28	0.28	0.28	0.28	0.0	0.0	0.39

It's interesting to note that TF-IDF is a measure of entropy (the TDIDF equation is the same as the equation for entropy).

Classification

```
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.20, random_state=42)
tfidf_vect = TfidfVectorizer(analyzer = "word", tokenizer=None,
    preprocessor=None, stop_words=None, max_features=500)
X_train_counts = tfidf_vect.fit_transform(X_train)
X_test_counts = tfidf_vect.transform(X_test)
```

- Note that I've prevented the default `TfidfVectorizer` `tokenizer`, `preprocessor` and `stop_words` because we've already done that.

If we wanted to, we could have plugged out methods in here. In the future this is probably a better idea.

- We set the `analyzer` to word to split the strings into words.
- And use a test-train split. Don't cheat!

???

Let's take a quick stab at classification to see how we're doing.

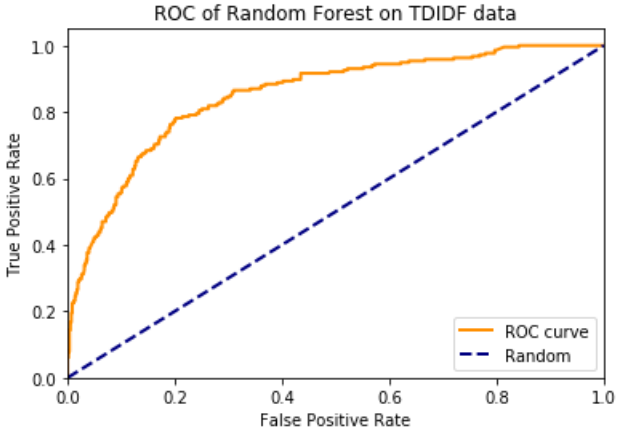
For this classifier, I pick my trusty Random Forest and simply play with a few values. At this point in time I just want to know if I'm in the right ball-park.

```
clf = RandomForestClassifier(n_estimators=10, max_depth=None,
    min_samples_split=50, random_state=42)
clf.fit(X_train_counts, y_train)
scores = cross_val_score(clf, X_train_counts, y_train)
scores.mean()
```

Next we fit the forest and calculate a cross validation score on the training data.

It reports `0.80`. Not too bad!

Let's take a closer look at the ROC curve...



.center[ ]

And the confusion matrix reports:

```
[[ 99  37]
 [116 538]]
```

So we can see that the data is very biased and we're reporting a lot of false positives.

So if we want to use accuracy as a score, we should balance the dataset (recall that the ROC curve takes the bias into account)

After balancing we get the following results (accuracy `0.76`):

So, still not too bad, but lots more can be done.

For now, let's do an exercise where you create your own cleaners and try your own classification algorithm.

(Mini competition time!)

Workshop: 32-representing-text

N-Gram Sequences

- We haven't considered *context*

For instance, in our troll example, we could have the words `["lucky", "bugger"]`. In a bag of words representation, the word bugger would probably flag up as trolling.

In most of the UK and elsewhere, the word bugger is a naughty word. (Don't ask me to define it).

But where I'm from, in Yorkshire, the word bugger isn't really a swearword. Even my Mum uses it.

The phrase "lucky bugger", in Yorkshire, means

> *a lucky person whom doesn't deserve it*

The phrase "little bugger", means:

> *a small child that is not very polite*

You see how the context changes the meaning?

???

You might have thought that a bag of words representation does not represent information well.

The meaning of words is often defined by the context, not just the word.

So instead of a bag of words, we can merge multiple words and place them in one feature.

In a 2-gram sequence, this would form the feature "lucky_bugger" and "little_bugger".

They are disjoint, and the terms will not be considered equal.

```
count_vect = CountVectorizer(ngram_range=(1,2))
X_train_counts = count_vect.fit_transform(X[8:10])
print(X_train_counts.shape)
print(count_vect.get_feature_names())
```

`CountVectorizer` and `TfidfVectorizer` have a parameter to do this for us. The result looks like:

```
['an', 'an idiot', 'are', 'are an', 'are fake', 'both', 'either', 'either you', 'extremely', 'extremely stupid',
'fake', 'fake or', 'health', 'idiot', 'idiot who', 'maybe', 'maybe both', 'neither', 'neither taxation', 'nor', 'nor
women', 'or', 'or extremely', 'stupid', 'stupid maybe', 'taxation', 'taxation nor', 'that', 'that you', 'understands',
'understands neither', 'who', 'who understands', 'women', 'women health', 'you', 'you are']
```

Interestingly, returning to our trolls person, it only increases accuracy by about 1%.

Here we plot the ROC curves for n-grams from 1-3. We can see that it does make a subtle difference, but probably not worth the extra complexity. (That is, unless the classifier isn't able to represent the complexity and therefore no improvements in features would improve the score! This is a tricky problem in general, and can only be proven through experimentation.)

Workshop: 33-ngrams

Named Entity Extraction

- Tokenisation for names of products/people/companies/films/etc.
- Often hand crafted
- Increasing use of automated solutions

???

Often we want to extract tokens with more sophistication. For example, we might want to tokenise the name of a product, a company or a film, etc. And replace this with a UID.

A basic bag-of-words or even n-grams might not capture this type of information and could be crucial to your task.

Often, they are hand-coded dictionaries.

There are automated solutions too. For example I've seen various neural networks and advanced statistical techniques being used. These are approaching the performance of a human. So as long as you have enough data to train upon, this is probably the way to go.

How?

Named Entity Extraction follows the following steps, some of which you are already familiar with:

1. Tokenise
2. Tag words
3. Combine tags

- Tagging is just a mapping of terms to tags, using a dictionary
- Once tagged, they need to be combined

E.g. "Williams Martini Racing"

is very different from

"Martini Racing"

???

The tagging process is often just a dictionary of words. The dictionary has usually been trained on a large corpus. It returns categories of words, like whether it's a verb, proper noun, etc.

Once the words have been tagged, they need to be "chunked" to group proper nouns together. For example, to combine "Williams Martini Racing" into a group, rather than just the individual words.

Indeed, I'd love to know what "Martini Racing" is! :-)

Let's look at some code.

```
def extract_NE(sample, debug=False):
    # Split sentences
    sentences = nltk.sent_tokenize(sample)
    # Split words
    tokenized_sentences = [nltk.word_tokenize(sentence) for sentence in sentences]
    # Tag words
    tagged_sentences = [nltk.pos_tag(sentence) for sentence in tokenized_sentences]
    # Combine tags
    chunked_sentences = nltk.ne_chunk_sents(tagged_sentences, binary=False)

    def extract_entity_names(t):
        entity_names = []
        if hasattr(t, 'label') and t.label:
            if t.label() in ["NE", "ORGANIZATION", "PERSON", "LOCATION"]:
                entity_names.append(' '.join([child[0] for child in t]))
            else:
                for child in t:
                    entity_names.extend(extract_entity_names(child))
        return entity_names

    entity_names = []
    for tree in chunked_sentences:
        entity_names.extend(extract_entity_names(tree))

    return set(entity_names)
```

```
input:
 "Quote from Teresa May "That is why we have been trying to deport him to Jordan, his home country." TRYING - Since
when does a Sovereign have problems TRYING to deport diseased minded scum like him. Stuff the EU laws and if the UK is
\\\\'fined\\\\' by the EUSSR for doing this then don\\\\'t pay it ... and then stop all financial payments to this
corrupt experiment. It doesn\\\\'t work, its broke so listen to the PEOPLE Camoron and have an in or out referendum ..
NOW!"

sentences:
 ['"Quote from Teresa May "That is why we have been trying to deport him to Jordan, his home country."', 'TRYING -
Since when does a Sovereign have problems TRYING to deport diseased minded scum like him.', "Stuff the EU laws and if
the UK is \\\\\\\\'fined\\\\\\\\' by the EUSSR for doing this then don\\\\\\\\'t pay it ... and then stop all
financial payments to this corrupt experiment.", "It doesn\\\\\\\\'t work, its broke so listen to the PEOPLE Camoron
and have an in or out referendum ..", 'NOW!"']

tokens:
 [['``', 'Quote', 'from', 'Teresa', 'May', '``', 'That', 'is', 'why', 'we', 'have', 'been', 'trying', 'to', 'deport',
'him', 'to', 'Jordan', ',', 'his', 'home', 'country', '.', "'"], ['TRYING', '-', 'Since', 'when', 'does', 'a',
'Sovereign', 'have', 'problems', 'TRYING', 'to', 'deport', 'diseased', 'minded', 'scum', 'like', 'him', '.'],
['Stuff', 'the', 'EU', 'laws', 'and', 'if', 'the', 'UK', 'is', "\\\\\\\\'fined\\\\\\\\", "'", 'by', 'the', 'EUSSR',
```

```
'for', 'doing', 'this', 'then', "don\\\\\\\\'t", 'pay', 'it', '...', 'and', 'then', 'stop', 'all', 'financial',
'payments', 'to', 'this', 'corrupt', 'experiment', '.'], ['It', "doesn\\\\\\\\'t", 'work', ',', 'its', 'broke', 'so',
'listen', 'to', 'the', 'PEOPLE', 'Camoron', 'and', 'have', 'an', 'in', 'or', 'out', 'referendum', '..'], ['NOW', '!',
"'"]]
```

```
tagged:
 [[('```', '```'), ('Quote', 'NN'), ('from', 'IN'), ('Teresa', 'NNP'), ('May', 'NNP'), ('```', '```'), ('That', 'DT'),
('is', 'VBZ'), ('why', 'WRB'), ('we', 'PRP'), ('have', 'VBP'), ('been', 'VBN'), ('trying', 'VBG'), ('to', 'TO'),
('deport', 'VB'), ('him', 'PRP'), ('to', 'TO'), ('Jordan', 'NNP'), (',', ','), ('his', 'PRP$'), ('home', 'NN'),
('country', 'NN'), ('.', '.'), ("'", "'")], [('TRYING', 'SYM'), ('-', ':'), ('Since', 'IN'), ('when', 'WRB'),
('does', 'VBZ'), ('a', 'DT'), ('Sovereign', 'NNP'), ('have', 'VBP'), ('problems', 'NNS'), ('TRYING', 'VBP'), ('to',
'TO'), ('deport', 'VB'), ('diseased', 'VBN'), ('minded', 'JJ'), ('scum', 'NNS'), ('like', 'IN'), ('him', 'PRP'), ('.',
'.')], [('Stuff', 'NNP'), ('the', 'DT'), ('EU', 'NNP'), ('laws', 'NNS'), ('and', 'CC'), ('if', 'IN'), ('the', 'DT'),
('UK', 'NNP'), ('is', 'VBZ'), ("\\\\\\\\\'fined\\\\\\\\\", 'NNP'), ("'", 'POS'), ('by', 'IN'), ('the', 'DT'), ('EUSSR',
'NNP'), ('for', 'IN'), ('doing', 'VBG'), ('this', 'DT'), ('then', 'RB'), ("don\\\\\\\\\'t", 'VBZ'), ('pay', 'VB'),
('it', 'PRP'), ('...', ':'), ('and', 'CC'), ('then', 'RB'), ('stop', 'VB'), ('all', 'DT'), ('financial', 'JJ'),
('payments', 'NNS'), ('to', 'TO'), ('this', 'DT'), ('corrupt', 'JJ'), ('experiment', 'NN'), ('.', '.')], [('It',
'PRP'), ("doesn\\\\\\\\\'t", 'VBZ'), ('work', 'NN'), (',', ','), ('its', 'PRP$'), ('broke', 'VBD'), ('so', 'RB'),
('listen', 'JJ'), ('to', 'TO'), ('the', 'DT'), ('PEOPLE', 'NNP'), ('Camoron', 'NNP'), ('and', 'CC'), ('have', 'VBP'),
('an', 'DT'), ('in', 'IN'), ('or', 'CC'), ('out', 'IN'), ('referendum', 'NN'), ('..', 'NN')], [('NOW', 'NN'), ('!',
'.'), ("'", "'")]]
```

```
chunked:
 [Tree('S', [('```', '```'), ('Quote', 'NN'), ('from', 'IN'), Tree('PERSON', [('Teresa', 'NNP'), ('May', 'NNP')]),
('```', '```'), ('That', 'DT'), ('is', 'VBZ'), ('why', 'WRB'), ('we', 'PRP'), ('have', 'VBP'), ('been', 'VBN'),
('trying', 'VBG'), ('to', 'TO'), ('deport', 'VB'), ('him', 'PRP'), ('to', 'TO'), Tree('GPE', [('Jordan', 'NNP')]),
(',', ','), ('his', 'PRP$'), ('home', 'NN'), ('country', 'NN'), ('.', '.'), ("'", "'")]), Tree('S', [('TRYING',
'SYM'), ('-', ':'), ('Since', 'IN'), ('when', 'WRB'), ('does', 'VBZ'), ('a', 'DT'), ('Sovereign', 'NNP'), ('have',
'VBP'), ('problems', 'NNS'), ('TRYING', 'VBP'), ('to', 'TO'), ('deport', 'VB'), ('diseased', 'VBN'), ('minded', 'JJ'),
('scum', 'NNS'), ('like', 'IN'), ('him', 'PRP'), ('.', '.')]), Tree('S', [('Stuff', 'NNP'), ('the', 'DT'), Tree('GPE',
[('EU', 'NNP')]), ('laws', 'NNS'), ('and', 'CC'), ('if', 'IN'), ('the', 'DT'), Tree('ORGANIZATION', [('UK', 'NNP')]),
('is', 'VBZ'), ("\\\\\\\\\'fined\\\\\\\\\", 'NNP'), ("'", 'POS'), ('by', 'IN'), ('the', 'DT'), Tree('ORGANIZATION',
[('EUSSR', 'NNP')]), ('for', 'IN'), ('doing', 'VBG'), ('this', 'DT'), ('then', 'RB'), ("don\\\\\\\\\'t", 'VBZ'),
('pay', 'VB'), ('it', 'PRP'), ('...', ':'), ('and', 'CC'), ('then', 'RB'), ('stop', 'VB'), ('all', 'DT'),
('financial', 'JJ'), ('payments', 'NNS'), ('to', 'TO'), ('this', 'DT'), ('corrupt', 'JJ'), ('experiment', 'NN'), ('.',
'.')]), Tree('S', [('It', 'PRP'), ("doesn\\\\\\\\\'t", 'VBZ'), ('work', 'NN'), (',', ','), ('its', 'PRP$'), ('broke',
'VBD'), ('so', 'RB'), ('listen', 'JJ'), ('to', 'TO'), ('the', 'DT'), Tree('ORGANIZATION', [('PEOPLE', 'NNP'),
('Camoron', 'NNP')]), ('and', 'CC'), ('have', 'VBP'), ('an', 'DT'), ('in', 'IN'), ('or', 'CC'), ('out', 'IN'),
('referendum', 'NN'), ('..', 'NN')]), Tree('S', [('NOW', 'NN'), ('!', '.'), ("'", "'")])]

{'UK', 'Teresa May', 'EUSSR', 'PEOPLE Camoron'}
```

Workshop: 34-named-entity-extraction

Latent Information

There is a broad set of tasks which require the formation of *latent information* from the data.

Latent information can be thought of as a type of metadata. Information that isn't inherently part of the data, but provides generalisations.

It's an unobserved layer of information that lies hidden between the inputs and outputs.

One common example is topics and tags.

For example, raw words relate to topics and topics map to documents.

How?

Latent information is usually obtained via some unsupervised process. Clustering for example.

Documents are parsed in the normal way and clustered to form distinct clusters or types of information.

The clusters would then form the basis for topics or tags.

As you can imagine there are any number of ways to achieve this, so I'm not going to show specifics here.

Very similar to hidden Markov Models

Stock Market Prediction From News Stories

(Your alarm bells should be ringing by the time you read this!)

- Market prediction requires huge leverage and is very complex

Let's simplify the problem

- A tool to alert us to some news stories that are predicted to make a significant impact on the market.

???

Ok, not really stock market *prediction*. As we've already seen stock market prediction is probably possible to some extent, but requires massive leverage (the amount of money you place on each trade, which is usually borrowed from someone) and is hugely complex.

However, something that we could do, given our new text skils, is create a tool to alert us to news stories that are predicted to make a significant impact on the market.

Then you might be able to perform a trade based upon the news that is recommended.

Still Too Complex

But this is still too complex.

1. How do news stories affect future prices? How far into the future?
2. What do we mean by significant?
3. What about prices?
4. How do stories affect the overall market? How does one piece of news affect all businesses?

Let's simplify the problem further:

1. Only use same day prices

2. We will threshold the data to only work on *large* movements. This is likely to be easier to predict. (Specifically, anything +/- 1% is significant.)
3. Prices are hard to predict. We will only result in a binary decision. `No Change` and `Significant Change`.
4. Predicting how a single news story will affect all companies is intractable. We will only consider news that specifically mentions a company.

Data

Unfortunately, because of the obvious value in such data, there aren't that many freely available datasets.

So, ideally, we would scrape the airwaves for stock information. We could trigger on known stock symbols, for example.

I managed to massage one dataset, from the UCI repository, into something vaguely useful.

```
2014-03-25 13:49:25.852,"Netflix, Inc. (NFLX) news: Is Apple Going To Be The Death Of Netflix?"
2014-03-25 13:49:30.742,"Netflix, Inc. (NFLX) Plummets on Potential Competition from Apple Inc (AAPL)"
2014-03-25 13:49:32.212,"Apple Inc. (AAPL) news: WSJ's Apple / Comcast Story Not Accurate, News Being  ..."
2014-03-25 13:49:41.917,Why Arris Group (ARRS) Is Down Today
2014-03-25 13:49:49.600,Apple Inc. (AAPL) news: More on Apple: Skepticism follows Comcast report
2014-03-25 13:49:50.128,Apple Inc. (AAPL) news: More on Apple: Skepticism follows Comcast report; new  ...
```

(`data/news.csv`)

We can see that there isn't much information there. Furthermore, there isn't many examples either. For the label `(AAPL)` there's only just over 200 samples.

Ploughing On

But let's keep going anyway. It's a good example.

First we load the CSV data.

```
news = pd.read_csv("data/news.csv", index_col=0, header=None)
news.columns = ["text"]
news.index = pd.to_datetime(news.index)
start_date, end_date = (min(news.index), max(news.index))
```

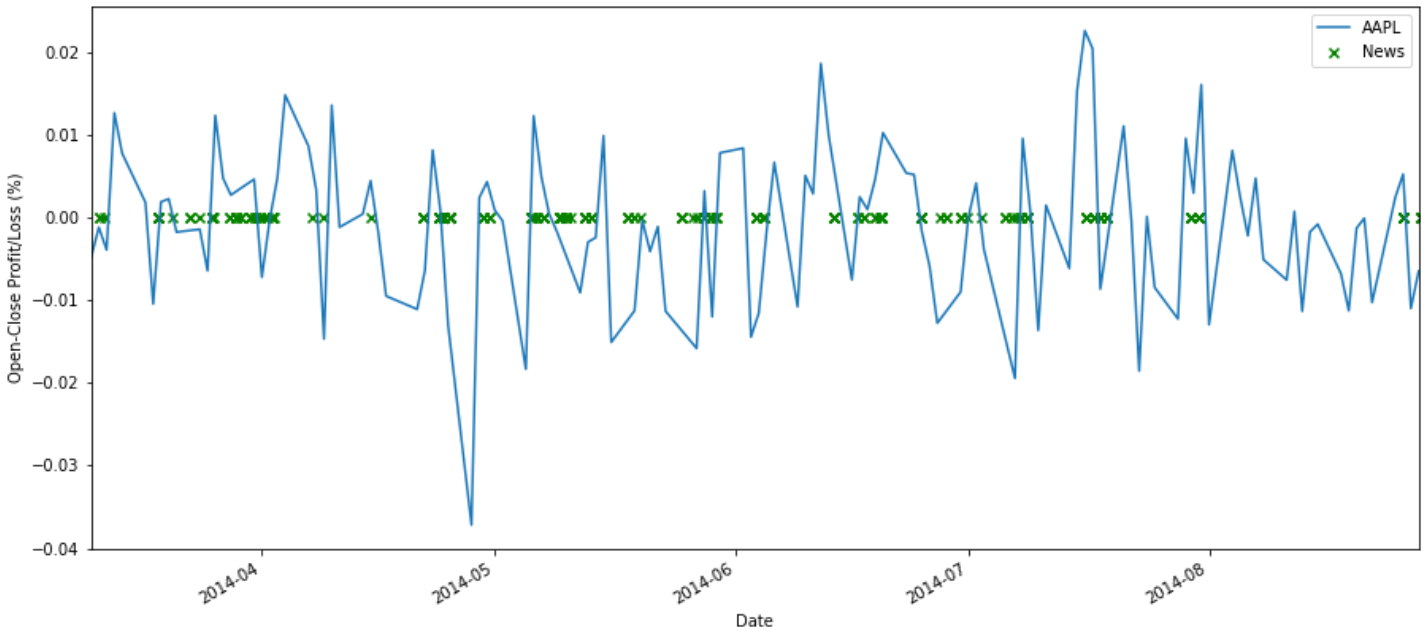
Next, for now, we only pick the Apple stock.

```
stock = "AAPL"
source = "yahoo"
apple_news = news[news["text"].str.contains(stock)]
```

Then use the `pandas_datareader` library to pull the stock prices for that date range:

```
apple_stock = data.DataReader(stock, source, start_date.date(), end_date.date())
c = apple_stock['Close']
o = apple_stock['Open']
pl_ratio = (o - c)/o
```

Then we can plot the two data...

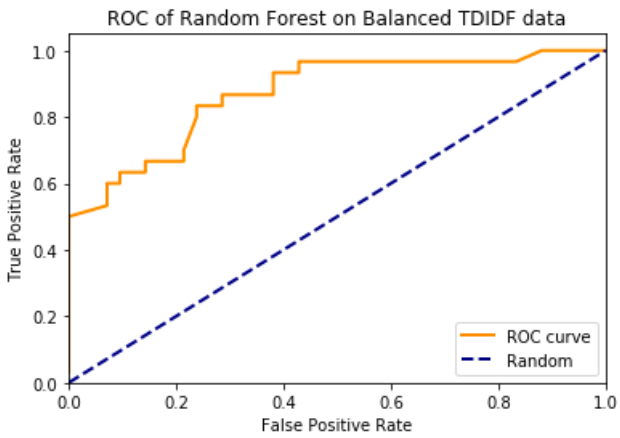


Now let's set a threshold and create our X/y dataset:

```
target = abs(pl_ratio) > 0.01
X = []
y = []
for d, t in zip(apple_news.index.date, apple_news["text"]):
    try:
        val = target[str(d)]
        y.append(val)
        X.append(t)
    except KeyError:
        continue

X = pd.Series(X)
```

Then after cleaning, td-idf'ing and fitting a RandomForest, we get...

A cross-validation score of `0.65` and the following roc curve:



The confusion matrix for the test data looks like:

```
[[16  3]
 [14 39]]
```

Workshop: 36-stock-market

I'd take that with a huge pinch of salt, given the complexity of the tast and the small number of samples.

Now it's your turn to use all your text prowess to try a recreate (or better?!) those results.

Ensemble Methods

Ensemble methods combine different classifiers to attempt to utilise the good parts of each model.

> *"This is how you win ML competitions: you take other peoples' work and ensemble them together" - Vitaly Kuznetsov, NIPS2014.*

???

All of our training has concentrated on one thing at a time. We've taken one dataset, created one model and one set of results.

Ensemble methods combine different classifiers to attempt to utilise the good parts of each model.

For example, if you create highly complex NN-based models, or sets of trees, then it's likely to overfit. Ensemble methods usually have better generalisation performance than any one classifier alone.

Ensemble also have practical ramifications. The majority of the highest scoring models in Kaggle competitions are ensemble models. I like this sarcastic quote:

> *"This is how you win ML competitions: you take other peoples' work and ensemble them together" - Vitaly Kuznetsov, NIPS2014.*

Types of Ensemble

Ensemble techniques can be generalised into three catagories:

1. Averaging methods - develop models in parallel and combine them with averaging or voting.
2. Stacking (or blending) methods - a weighted combination of multiple classifiers are fed into the input of another layer
3. Boosting methods - Building models in sequence where each added model aims to improve the score of the combined estimator

We'll leave the workshop until the end of this section because it isn't very long.

Averaging methods

Averaging methods take the results from individual classifiers and combine them to produce a final classification.

There are a few different ways to combine the results:

1. Majority voting - Pick the class that has the most votes
2. Averaging - Combine the prediction by averaging - this is especially useful for models that produce scores/probabilities at the output.
3. Plurality voting - A generalisation of majority voting to multiple classes.

Majority voting

Thankfully, once again, there is a class in sklearn called `VotingClassifier` that allows us to combine the outputs from several classifiers.

```
clf1 = LogisticRegression(random_state=42)
clf2 = RandomForestClassifier(random_state=42)
clf3 = KNeighborsClassifier(n_neighbors=1, p=2, metric='minkowski')

eclf = VotingClassifier(estimators=[('lr', clf1), ('rf', clf2), ('gnb', clf3)], voting='hard')
```

The `voting` parameter, when set to hard, means it picks the classification with the most votes.

Average Voting

When the `voting` parameter, is set to `soft`, this uses the `predict_proba` methods to get the scores from each individual classifier and average them together.

```
eclf = VotingClassifier(estimators=[('lr', clf1), ('rf', clf2), ('gnb', clf3)], voting='soft', weights=[2, 2, 1])
```

The `weights` parameter allows you to specify how much each classifier contributes to the final result.

Example

When we use cross validation to fit on the digits dataset (pixels are flattened), we get the following results.

```
# Majority
Accuracy: 0.94 (+/- 0.03) [Logistic Regression]
Accuracy: 0.93 (+/- 0.03) [Random Forest]
Accuracy: 0.96 (+/- 0.02) [k-NN]
Accuracy: 0.96 (+/- 0.02) [Ensemble]

# Average
Accuracy: 0.94 (+/- 0.03) [Logistic Regression]
Accuracy: 0.93 (+/- 0.03) [Random Forest]
Accuracy: 0.96 (+/- 0.02) [k-NN]
Accuracy: 0.97 (+/- 0.02) [Ensemble]
```

???

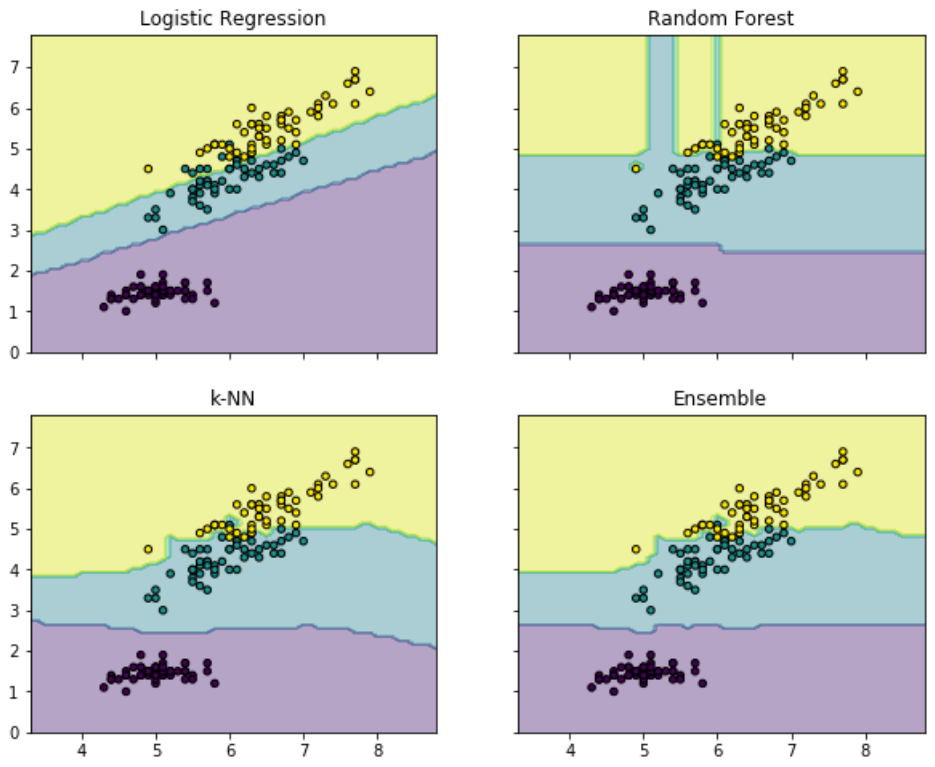
Note that we're measuring accuracy here, because the target is multi-class. The ensemble is performing as well

as the best classifier because it's making the same classification as the best classifier.

It would be better to use a multi-class F1 or AUC score, which would show better performance on the ensemble.

Also, if we had a larger dataset, the decisions would be slightly different, so again, we'd see the ensemble performing better.

Plotting the decision boundary for Iris data...

.center[

]

Bagging

- Each classifier is trained on different parts of the dataset

```
clf = BaggingClassifier(KNeighborsClassifier(),max_samples=0.5, max_features=0.5, random_state=42)
```

???

Bagging is another form of averaging. Instead of using all the training data for each model, each model gets its own segment of the dataset.

In other words, the different classifiers are trained upon different parts of the dataset.

This is how we can use it...

Unfortunately this only allows us to provide one classifier. If you wanted to use different classifiers, then we'd need to implement our own.

The main benefit of bagging is that it is an automated (but random) way to average out problems with noisy data. E.g. imagine if you had several samples that were outliers?

- Alternatively pick a random subset of features

Similar to random forests

- No excuse not to do data analysis

???

An alternative, but similar method is to pick a subset of features, rather than a subset of samples.

This allows us to fit different classifiers to different parts of the dataset.

This works very well, as it tends to create classifiers for different types of data. Basically it's an automated (but again, random) method of segmenting the data into more solvable problems.

You've probably realised that we've seen this before, and we have. If we used this approach with Decision Trees, then we'd have a Random Forest classifier.

Boosting

Instead of trying to create one model that is able to model a complex domain, create lots and lots of tiny classifiers that model a small part of the complex domain.

Then we can recombine the results with averaging or majority voting.

The really cool part is the way the algorithm slices up the problem.

1. Given observations
2. Create a model
3. Fit the model
4. Find all the misclassified observations and feed back to 1.

One type of boosting is implemented in `sklearn` with `AdaBoostClassifier`.

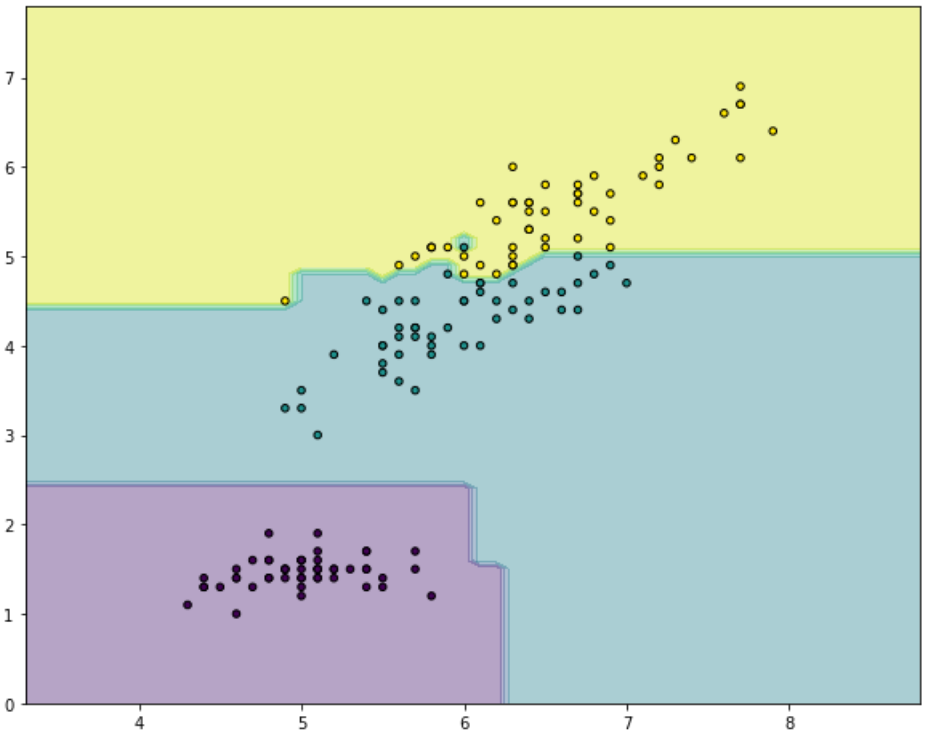
???

This is another really interesting angle.

```
stump = DecisionTreeClassifier(max_depth=8, min_samples_leaf=5)
clf = AdaBoostClassifier(stump,n_estimators=100,learning_rate=0.1)
scores = cross_val_score(estimator=clf,X=X,y=y,cv=10,scoring='accuracy')
print("Accuracy: %0.2f (+/- %0.2f) [%s]" % (scores.mean(), scores.std(), "Boosting"))
```

```
Accuracy: 0.92 (+/- 0.03) [Boosting]
```

One problem with boosting is that in the extreme, it can overfit. This is the iris data.



.center[ ]

Gradient Boosting

Gradient boosting is essentially the same as the previous boosting algorithm, except that it allows a customisable loss function.

Hence, this can be generalised to regression problems and even be used in a profit/lost function.

Stacking

A more complicated way of combining models is to feed the outputs of several models into the input of another.

In the averaging section we saw a parameter called `weights` that controlled the weighted combination of the averaged output.

Think of stacking as a method to *learn* those weights.

This implies that it is best to provide as much information to the second layer as possible. I.e. pass the scores, not the class predictions.

I'm using iris data here to be able to plot the resultant decision boundary.

```
clf1 = LogisticRegression(random_state=42)
clf2 = SVC(random_state=42, probability=True)
clf3 = GaussianNB()
clf4 = DecisionTreeClassifier(max_depth=1, criterion='entropy',random_state=0)
clf5 = KNeighborsClassifier(n_neighbors=1, p=2, metric='minkowski')

clfs = [clf1, clf2, clf3, clf4, clf5]
names = ["Logistic", "SVM", "NB", "Tree", "k-NN"]
for i, (clf, n) in enumerate(zip(clfs, names)):
    scores = cross_val_score(estimator=clf,X=X,y=y,cv=10,scoring='accuracy')
    print("Accuracy: %0.2f (+/- %0.2f) [%s]" % (scores.mean(), scores.std(), n))
```

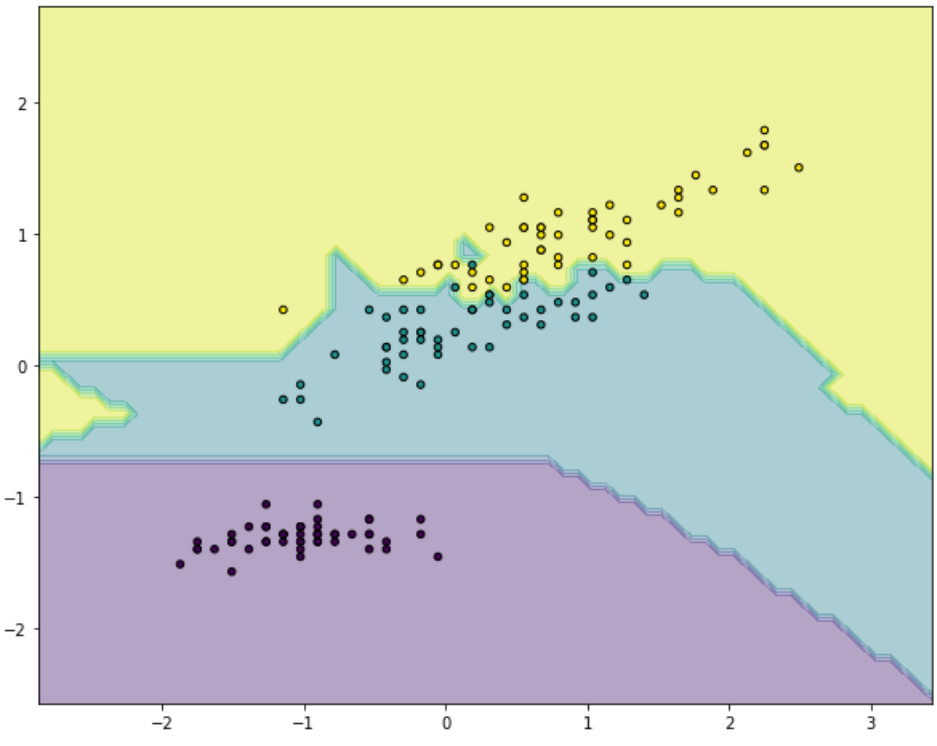
```
    clf.fit(X, y)

def create_blend(clfs, X):
    output = np.empty((len(X), 0))
    for clf in clfs:
        y_pred = clf.predict_proba(X)
        output = np.append(output, y_pred, axis=1)
    return output

blend = create_blend(clfs, X)

second_clf = KNeighborsClassifier(n_neighbors=1, p=2, metric='minkowski')
second_clf.fit(blend, y)
scores = cross_val_score(estimator=second_clf,X=blend,y=y,cv=10,scoring='accuracy')
print("Accuracy: %0.2f (+/- %0.2f) [%s]" % (scores.mean(), scores.std(), "Stacked"))
```

The result:

```
Accuracy: 0.87 (+/- 0.09) [Logistic]
Accuracy: 0.93 (+/- 0.06) [SVM]
Accuracy: 0.91 (+/- 0.08) [NB]
Accuracy: 0.67 (+/- 0.00) [Tree]
Accuracy: 0.92 (+/- 0.06) [k-NN]
Accuracy: 0.99 (+/- 0.02) [Stacked]
```

Let's look at the decision boundary...

Look at how much it's overfitting!

.center[

]

Ensemble Conclusion

- Reasonable extension to "traditional" machine learning

The only way to score highly in public competitions

- Try to mix classifiers that produce *orthogonal* results
- Obvious overfitting problems

???

Ensemble methods are pretty simple, but are usually more work. You need extra feature generation time, extra tuning time, etc.

Ensembles really are the only way you can score highly on public competitions. So, (hint, hint) if you're doing that then definitely use Ensembles.

More complex problems will require improving individual classifiers as well as the ensembled result.

Generally, you want to mix classifiers that generate orthogonal results. I.e. if one classifier get's some types of classes correct and another does well at another type, they are good for combination.

Ensembles can actually be used as a form of automated feature generation, much like auto-encoders.

Combinations of ensembles, like bagging classifiers whilst stacking can also be highly performant.

But like usual, there is no one best structure of any given model. And remember that features are still important!

- Beware of using them for work/production

Now it's your turn to get some experience with Ensembles.

???

If you're using them for work, then I'd definitely consider whether they are worth the effort.

Remember that any extra layer usually means making a nonlinear transformation (that is unless you picked layers of linear classifiers). Which means that it makes it next to impossible to draw and theories or conclusions about the classifications that were made.

If you are in an industry where algorithms must make understandable decisions, Ensembles are probably not for you.

Also be wary of overfitting. The extra modelling power that ensembles bring mean it's very easy to overfit.

Workshop: 40-ensembles

Grand Challenge: Text

Now it's time for you to have some practise.

There's no point you listening to me all day. You need to get your hands dirty!

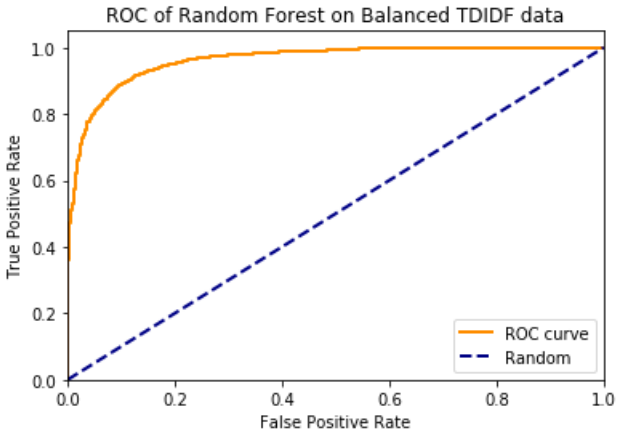
This challenge is based upon Kaggle's IMDB movie dataset.

```
"9057_3"    0  "<br /><br />What an absolutely crappy film this is. How or why this movie was made and what the hell
Billy Bob Thornton and Charlize Theron were doing signing up for this mediocre waste of time is beyond me. Strong
advise for anyone sitting down to catch a flick: DO NOT waste your time on this 'film'."
```

The goal is to correctly classify textual movie reviews as either being good (1) or bad (0).

The winner will get to show his results to the rest of the class!

As a baseline, I managed to get 90% accuracy, but only after a while of messing around with lots of different algorithms. Here's the roc curve:



.center[ ]

- Don't prioritise cleaning the text. The data are already pretty clean.
- Make sure you cross-validate to get a proper score. I will be checking the winner!
- Reduce the amount of data whilst iterating. Don't waste time using the whole dataset, do that at the end.
- You should be able to get 70% with any old thing. If you're getting more than 90% you're either not testing properly or you're a superstar.
- I have provided a test set that should be used to generate your final score. You shouldn't need this until the end. Don't cheat! ;-)

Tips:

- Features? (Try any crazy idea. E.g. total number of characters? Smilies?)
- Try different classifiers (this seems to be more important that improving the text?)
- Neural networks for binary classification?
- Ensembles?

Review

Today:

- Deep Learning
- Text
- Ensemble methods

All very advanced topics

- Be wary, it's easy to over-complicate

Previous two days:

- A rapid tour of data science
- Very fast. You probably feel fuzzy.
  - Take a couple of days to let it all sink in
  - Then take a fresh look at your work
  - Can you see any value in your data?
  - How can you use data to improve your products?
- Then come back and start investigating in more depth

---

class: middle

Thank You!

- If you need any help at all, get in touch.
- Know anyone that would like this training?

 @DrPhilWinder

 DrPhilWinder

 http://WinderResearch.com

 http://TrainingDataScience.com

.center[Winder Research]
