# Kernel functions

The classifiers that we have studied are Linear: they create boundaries that are linear in the features in an attempt to separate classes.

That is: they work well when the classes are (nearly) Linearly Separable.

What if our classes don't appear to be Linearly Separable ?

We can try transforming the features so that, in the transformed features, the classes are separable.

# Transformations to induce Linear Separability

Transformations to induce linearity are interesting because

- They make our data linear in the new dimensions (features)
- when mapped back to our original dimensions, they introduce non-linearity, which can be powerful

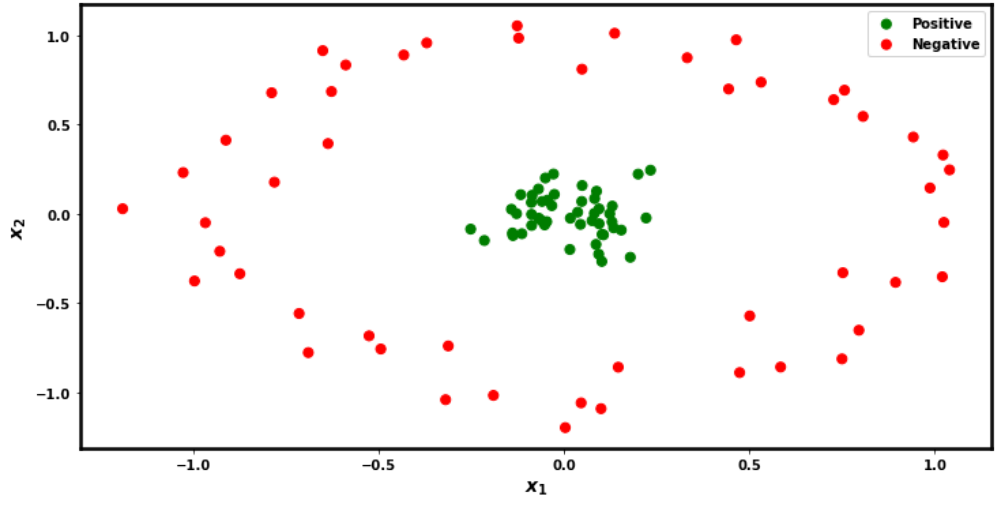
As we will see, non-linear decision boundaries are a key part of Deep Learning.

In Classical Machine Learning, these transformations serve a similar role.

Let's explore a classification problem.

- The colors denote different classes.
- We won't name the $\mathbf{x}_1$, $\mathbf{x}_2$ features

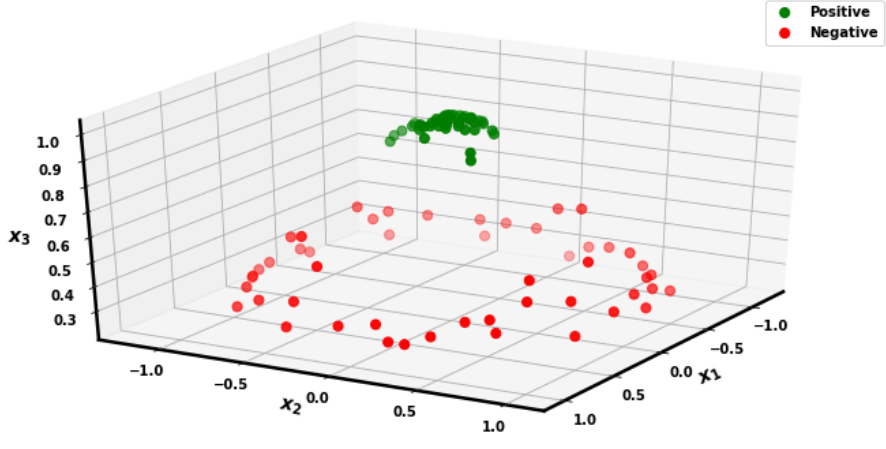
In [4]:
```
Xc, yc = svmh.make_circles(plot=True)
```



Doesn't appear to be linearly separable.

But, as we saw in the Transformation lecture, an RBF transformation does the trick:

In [5]:
```
X_w_rbf = svmh.circles_rbf_transform(Xc)
_= svmh.plot_3D(X=X_w_rbf, y=yc )
```



Magic ! The new feature separates the two classes.

- Just like the polynomial feature make the curvy data set linear

The particular transformation is called Radial Basis Function (RBF) Transformation.

Here's the code that created the new feature "r".

```
r = np.exp( -(Xc ** 2).sum(1) )
```

Simple.

Basically, the transformation creates a scalar measure (*similarity function*)

- The distance between the existing features $\mathbf{x}^{(\mathbf{i})}$ of example $i$ and the features of a reference point.

In this case the reference point ("landmark") is the origin $(0, 0)$ so we don't write it explicitly.

In general the distance would be coded as `Xc - x_ref`

Examples that are very close to the reference point have high values, and the values fall off sharply as the distance increases.

# Support Vector Machines: SVC's with integrated transformations

## Motivation

Nothing we have said about Linearity Inducing transformations should be new to you.

It is just like any other problem:

- Apply transformations
- Apply a model (Classifier)

A *Support Vector Machine* is an SVC that has been integrated with a *special class of transformations*.

These transformations are intimately tied to the mathematics of the SVC.

As a result, these transformations

- are specified as arguments to the model
- rather than separate statements implementing the transformation

So, rather than

- applying transformation $\phi$ to each example $\mathbf{x}$
  $$\tilde{\mathbf{x}} = \phi(\mathbf{x})$$
  and then fitting the SVC to transformed examples $\tilde{\mathbf{X}}$
  $$\text{SVC.fit}(\tilde{\mathbf{X}}, \mathbf{y})$$

you specify the *name* of the transformation as argument to the model

- *without* computing $\tilde{\mathbf{X}}$
  $$\text{SVC.fit}(\mathbf{X}, \mathbf{y}, \text{kernel='name of kernel'})$$

What makes these "integrated transformations" special ?

Consider a transformation $\phi$

- applied to a feature vector $\mathbf{x^{(i)}}$ of length $n$

$$\phi(\mathbf{x^{(i)}})$$

- resulting in transformed vector $\tilde{\mathbf{x}}$ of length $n'$

Some transformations

- result in much bigger $n' \gg n$
    - increase space
- are computationally intensive

The integrated transformations have the property

- we *achieve the effect* of fitting the model on transformed $\tilde{\mathbf{X}} = \phi(\mathbf{X})$
- **without** having to incur the cost of computing $\phi(\mathbf{X})$

A free lunch !

- no increase in space of time complexity

## Mathematics of the optimization problem

The SVC optimization problem minimizes

$$\mathcal{L} = \frac{1}{2}\Theta_{-0}^T \cdot \Theta_{-0} + C * \frac{1}{m}\sum_{i=1}^{m} \max\left(0, 1 - \dot{\mathbf{y}}^{(\mathbf{i})} * s(\hat{\mathbf{x}}^{(\mathbf{i})})\right)$$

where

$$\hat{s}\left(\mathbf{x}^{(\mathbf{i})}\right) = \Theta^T \cdot \mathbf{x}^{(\mathbf{i})}$$

Because of the term $\frac{1}{2}\Theta_{-0}^T \cdot \Theta_{-0}$

- the Loss minimization problem is one of *quadratic optimization* with linear constraints

These are solvable via a technique (beyond the scope of the course) of Quadratic Programming.

A Quadratic Programming problems can be reformulated into a twin problem called its *dual form*. The dual form for the SVC optimization problem is

**Dual form of the linear SVM objective (similar to Geron Equation 5-6)**

$$\underset{\alpha}{\text{minimize}} \frac{1}{2} \sum_{i=1}^{m} \sum_{i'=1}^{m} \alpha^{(i)} \alpha^{(i')} \dot{\mathbf{y}}^{(i)} \dot{\mathbf{y}}^{(i')} \mathbf{x^{(i)}}^T \mathbf{x}^{(i')} \quad - \quad \sum_{i=1}^{m} \alpha^{(i)}$$

$$\text{subject to} \quad \alpha^{(i)} \geq 0 \quad \text{for } i = 1, 2, \ldots, m$$

It is beyond the scope of this course to explain the equation.

The key observations to take away from the above equation is

- The only sub-expression involving an example is when examples occur in pairs : $\mathbf{x^{(i)}}, \mathbf{x}^{(i')}$
- The only computation on the pair is the dot product $\mathbf{x^{(i)}} \cdot \mathbf{x}^{(i')} = \mathbf{x^{(i)}}^T \mathbf{x}^{(i')}$

**Key point**

If we transform each $\mathbf{x}$ into $\tilde{\mathbf{x}} = \phi(\mathbf{x})$

- the above equation requires us to compute
$$\phi(\mathbf{x^{(i)}}) \cdot \phi(\mathbf{x}^{(i')})$$

# Kernels and the Kernel Trick

The integrated transformations are

- a way to compute
  $\phi(\mathbf{x^{(i)}}) \cdot \phi(\mathbf{x}^{(i')})$

using only the dot product of the *untransformed* examples

$$\mathbf{x^{(i)}} \cdot \mathbf{x}^{(i')}$$

As the computation of the transformations

- are expensive
- space intensive
- longer $\tilde{\mathbf{x}}$ results in more expensive dot product

it is very desirable to use these integrated transformations.

These integrated transformations are based on special functions called *kernels*.

A *kernel* is a function

$$K(\mathbf{x^{(i)}}, \mathbf{x}^{(i')})$$

that is able to compute

$$\phi(\mathbf{x^{(i)}}) \cdot \phi(\mathbf{x}^{(i')})$$

while only computing

$$\mathbf{x^{(i)}} \cdot \mathbf{x}^{(i')}$$

So, in the optimization

- we replace the expensive term
  $$\phi(\mathbf{x^{(i)}}) \cdot \phi(\mathbf{x}^{(i')})$$
- with inexpensive term
  $$K(\mathbf{x^{(i)}}, \mathbf{x}^{(i')})$$

The "integrated transformations"

- are restricted
- so that they have kernels that can compute them

For example

- suppose we have examples $\mathbf{x}$ and $\underline{\mathbf{x}}$ as vectors of length $2$

$$\begin{aligned} \mathbf{x} &= (\mathbf{x}_1, \mathbf{x}_2) \\ \underline{\mathbf{x}} &= (\underline{\mathbf{x}}_1, \underline{\mathbf{x}}_2) \end{aligned}$$

**Notation alert**

- we use $\mathbf{x}$ and $\underline{\mathbf{x}}$
- rather than our usual $\mathbf{x}^{(\mathbf{i})}$ and $\mathbf{x}^{(i')}$

because we will need superscripts to denote exponentiation and desire a transformation to square each feature

We want a transformation to square each feature

$$\phi(\mathbf{x}_1, \mathbf{x}_2) = (\mathbf{x}_1^2, \mathbf{x}_2^2)$$

This will result in a term

$$\phi(\mathbf{x}) \cdot \phi(\underline{\mathbf{x}}) = (\mathbf{x}_1^2, \mathbf{x}_2^2) \cdot (\underline{\mathbf{x}}_1^2, \underline{\mathbf{x}}_2^2)$$

in the optimization.

That is

- we **must** raise *each* of $n$ features to the power of 2
    - $n = 2$ for illustration, but imagine the cost for a much larger $n$
- before computing the dot product

There may not (is not ?) a kernel to compute

$$(\mathbf{x}_1^2, \mathbf{x}_2^2) \cdot (\underline{\mathbf{x}}_1^2, \underline{\mathbf{x}}_2^2)$$

**But**

- there is a *similar* (not identical) transformation

$$\phi(\mathbf{x}) = \phi\left(\begin{pmatrix}\mathbf{x}_1\\ \mathbf{x}_2\end{pmatrix}\right) = \begin{pmatrix}{\mathbf{x}_1}^2\\ \sqrt{2}\,\mathbf{x}_1\mathbf{x}_2\\ {\mathbf{x}_2}^2\end{pmatrix}$$

- that **does** have a kernel

$$K(\mathbf{x^{(i)}}, \mathbf{x}^{(i')}) = (\mathbf{x^{(i)}} \cdot \mathbf{x}^{(i')})^2$$

- only the *scalar* that results from the dot product is exponentiated
  - **not** each of $n$ features

We can prove that this transformation has $K(\mathbf{x}, \underline{\mathbf{x}}) = (\mathbf{x} \cdot \underline{\mathbf{x}})^2$ as its kernel

- by computing the dot product of $\phi(\mathbf{x}) \cdot \phi(\underline{\mathbf{x}})$
- and showing it to be equal to kernel $K(\mathbf{x}, \underline{\mathbf{x}}) = (\mathbf{x} \cdot \underline{\mathbf{x}})^2$

$$
\begin{aligned}
\phi(\mathbf{x})^T \phi(\underline{\mathbf{x}}) &= \begin{pmatrix} {\mathbf{x}_1}^2 \\ \sqrt{2}\,\mathbf{x}_1\mathbf{x}_2 \\ {\mathbf{x}_2}^2 \end{pmatrix}^T \begin{pmatrix} {\underline{\mathbf{x}}_1}^2 \\ \sqrt{2}\,\underline{\mathbf{x}}_1\underline{\mathbf{x}}_2 \\ {\underline{\mathbf{x}}_2}^2 \end{pmatrix} \\
&= {\mathbf{x}_1}^2{\underline{\mathbf{x}}_1}^2 + 2\mathbf{x}_1\underline{\mathbf{x}}_1\mathbf{x}_2\underline{\mathbf{x}}_2 + {\mathbf{x}_2}^2{\underline{\mathbf{x}}_2}^2 \\
&= (\mathbf{x}_1\underline{\mathbf{x}}_1 + \mathbf{x}_2\underline{\mathbf{x}}_2)^2 \\
&= \left( \begin{pmatrix} \mathbf{x}_1 \\ \mathbf{x}_2 \end{pmatrix}^T \begin{pmatrix} \underline{\mathbf{x}}_1 \\ \underline{\mathbf{x}}_2 \end{pmatrix} \right)^2 \\
&= (\mathbf{x}^T \mathbin{\dot{-}} \mathbf{x})^2 \\
&= K(\mathbf{x}, \underline{\mathbf{x}})
\end{aligned}
$$

By using a slightly different transformation

- adding a cross-product feature
- we facilitate replacing an expensive operation
    - transformation plus dot product of transformed terms
    - with a much cheaper operation
        - exponentiate only a single term (the scalar resulting from the dot product)
        - rather than $n$ features

Here are some examples of kernel transformation, paired with their "natural" counterparts.

Why the differences ?

- The kernel transformation is not identical to the natural (e.g., Polynomial)
- Possible differences in parameters

In [6]:

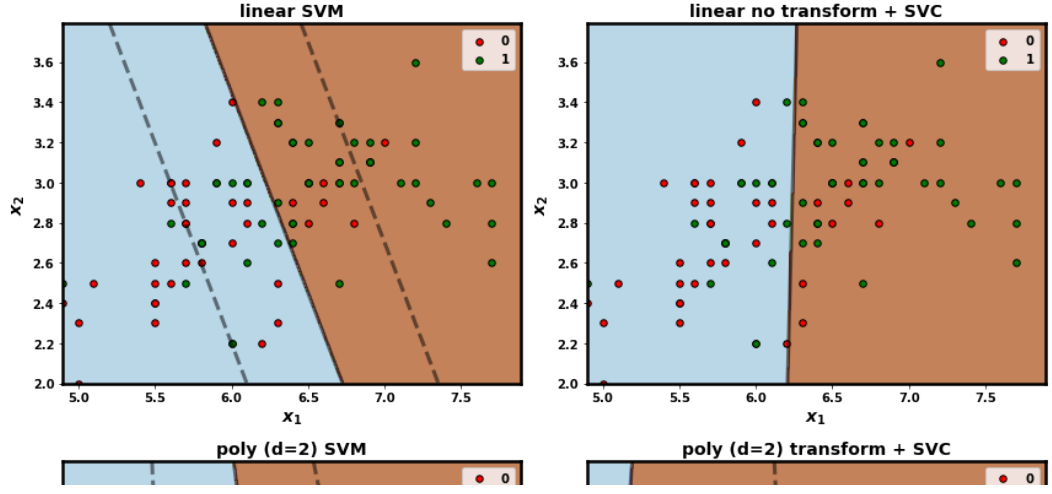
```
svmh = svm_helper.SVM_Helper()

_ = svmh.create_kernel_data()
fig, axs = svmh.plot_kernel_vs_transform()
plt.close()
```

```
/home/kjp/anaconda3/lib/python3.7/site-packages/sklearn/svm/base.py:929: Conve
rgenceWarning: Liblinear failed to converge, increase the number of iteration
s.
  "the number of iterations.", ConvergenceWarning)
/home/kjp/anaconda3/lib/python3.7/site-packages/sklearn/svm/base.py:929: Conve
rgenceWarning: Liblinear failed to converge, increase the number of iteration
s.
  "the number of iterations.", ConvergenceWarning)
```

In [7]:
```
fig
```

Out[7]:


linear SVM
linear no transform + SVC
poly (d=2) SVM
poly (d=2) transform + SVC
0
1
$x_1$
$x_2$

## Some Kernel functions

kernels (http://crsouza.com/2010/03/17/kernel-functions-for-machine-learning-applications/#linear)

There are a number of common kernel functions. We investigate just a handful:

### Linear

The linear kernel

$$K(\mathbf{x}, \mathbf{x}^{(i')}) = \mathbf{x}^T \cdot \mathbf{x}^{(i')} + c$$

corresponds to a $\phi$ that is the identity transformation $\phi(\mathbf{x}) = \mathbf{x}$.

The linear kernel results in something very close to plain logistic regression.

## Gaussian

The Gaussian Radial Basis Function (RBF)

$$K(\mathbf{x}, \mathbf{x}^{(i')}) = \exp(-\frac{1}{2\sigma^2} * || \mathbf{x} - \mathbf{x}^{(i')} ||^2)$$

The form of this function is that of a Gaussian distribution with mean $0$ and standard deviation $\sigma$.

The similarity is maximized when $\mathbf{x}$ and $\mathbf{x}^{(i')}$ are close in Euclidean space (due to the 2-norm $|| \ ||^2$)

When $\sigma$ is small, there is a sharp drop-off from the maximum at $\mathbf{x}^{(i')}$ to near-by points.

Conversely, when $\sigma$ is large, the drop-off is much smoother.

Note the presence of the *hyper parameter* $\sigma$.

How do we choose the right value ?

We don't need to !

- Consider $\sigma$ to be just like $\Theta$: a parameter that the optimizer solves for
    - Let the optimizer simultaneously find the best combination of $\Theta$ and $\sigma$

In [8]:
```
print("Done")
```

```
Done
```