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

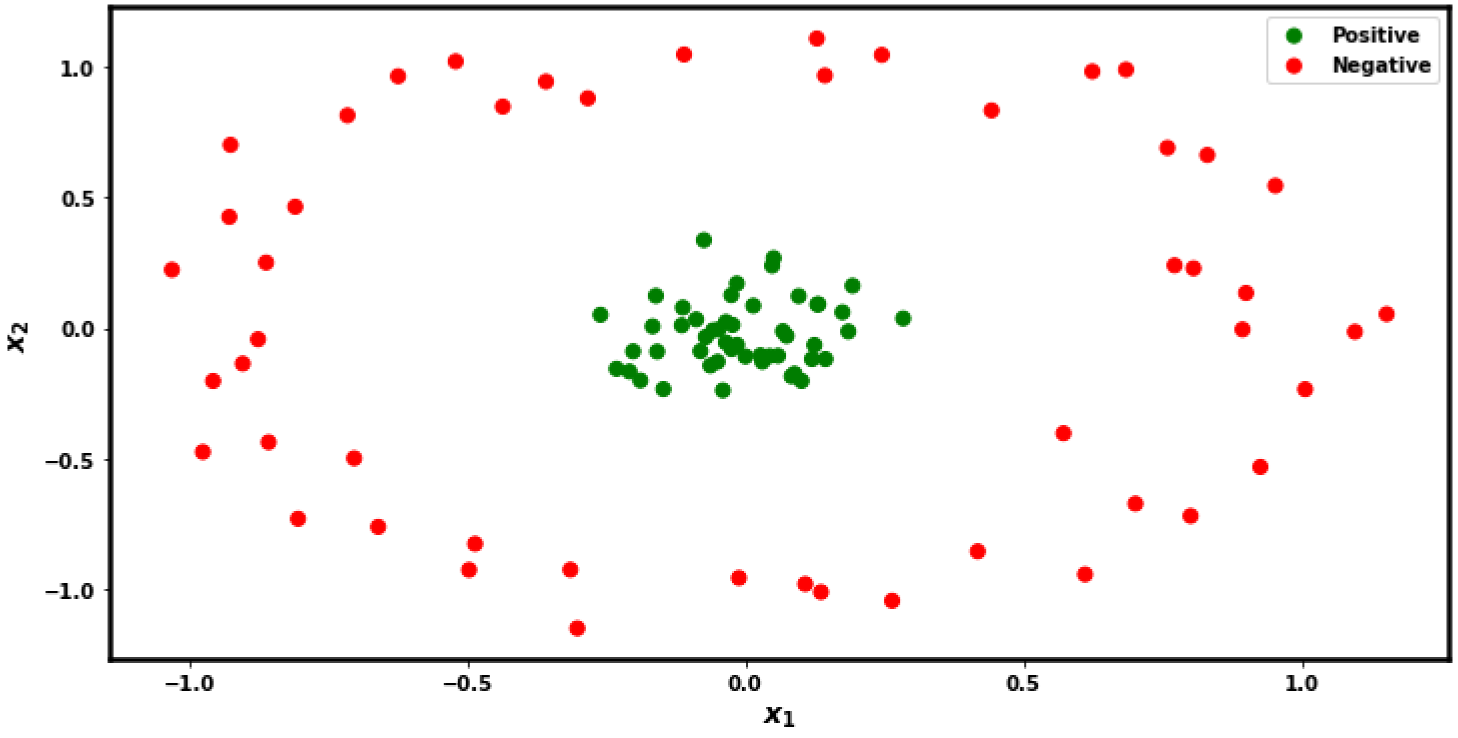
As we will see, non-linear decision boundaries are a key part of Deep Learning.

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

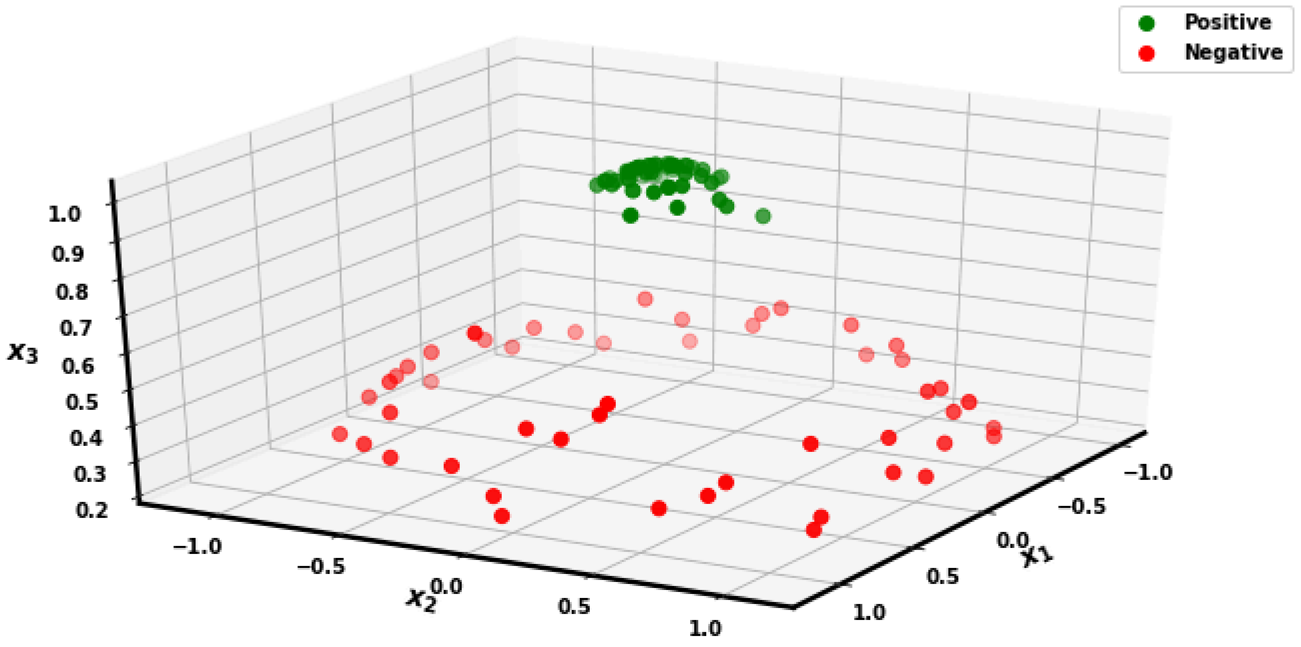
```
In [4]: Xc, yc = svmh.make_circles(plot=True)
```



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

The issue with transformations

Transformations may increase the number of features

- increased computational cost for the optimizer

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 ϕ 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 ϕ

- 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

- incur no greater computational cost
- even when $n' \gg n$

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

How do we achieve this "free lunch" ?

Let's examine the behavior of the quadratic optimizer used to solve SVC.

The SVC optimization problem minimizes

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

where

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

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

- 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)

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

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

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

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

That is

- **it is not necessary** to implement the transformation !

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

Example of a transformation with a Kernel function

In the optimization objective, we have terms of the form

$$\phi(\mathbf{x}) \cdot \phi(\underline{\mathbf{x}})$$

for two examples $\mathbf{x}, \underline{\mathbf{x}}$

Notation alert

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

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

Suppose we want a transformation

- to square each feature in vector $\mathbf{x}$

For $n = 2$:

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

This gives rise to terms of the form

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

This expression

- requires exponentiating n terms (for each of the n features of $\mathbf{x}, \underline{\mathbf{x}}$)

It would be desirable to have a Kernel function

- to compute $\phi(\mathbf{x}) \cdot \phi(\underline{\mathbf{x}})$
- *without* requiring n exponentiations.

There may not (is not ?) such a Kernel function for the transformation

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

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

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

- that **does** have a Kernel function
- that avoids invoking (expensive) transformation ϕ

We can prove that this transformation

- has Kernel function
$$K(\mathbf{x}, \underline{\mathbf{x}}) = \left(\begin{pmatrix} \mathbf{x}_1 \\ \mathbf{x}_2 \end{pmatrix} \cdot \begin{pmatrix} \underline{\mathbf{x}}_1 \\ \underline{\mathbf{x}}_2 \end{pmatrix} \right)^2$$

Note that

- there is a single exponentiation
- of a *scalar* (the result of the dot product)
- rather than n exponentiations

Much cheaper !

To see that this Kernel function computes

$$\phi(\mathbf{x}) \cdot \phi(\underline{\mathbf{x}})$$

let's write out the steps.

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

By using a transformation slightly different than our original goal

- we facilitate computing terms in the optimization
- that are much cheaper
- because they use a Kernel function
    - single exponentiation of a *scalar* (result of $\mathbf{x} \cdot \underline{\mathbf{x}}$
    - rather than n exponentiations

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

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 ϕ 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 σ.

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

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

Conversely, when σ is large, the drop-off is much smoother.

Note the presence of the *hyper parameter* σ.

How do we choose the right value ?

We don't need to !

- Consider σ to be just like Θ: a parameter that the optimizer solves for
    - Let the optimizer simultaneously find the best combination of Θ and σ

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

```
Done
```