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

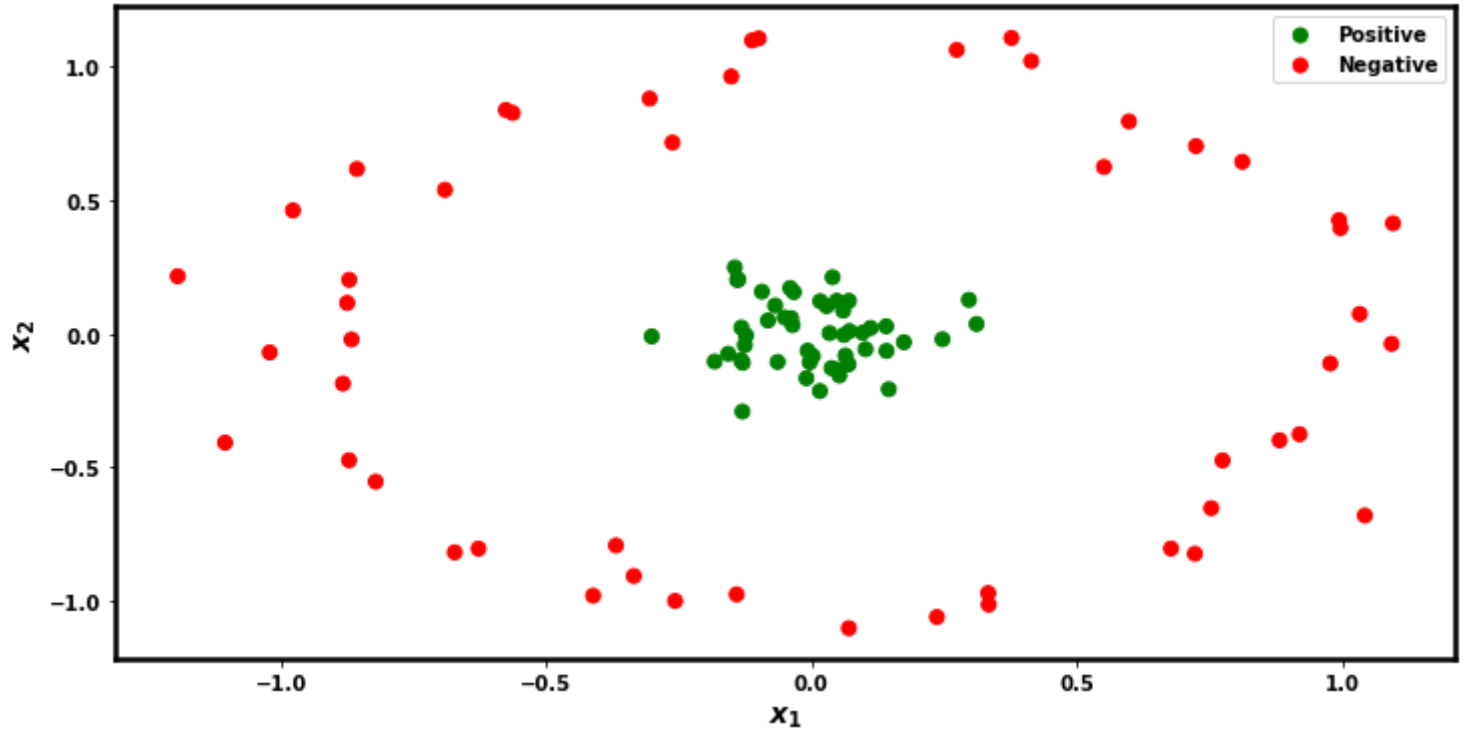
As we will see, non-linear decision boundaries are a key part of Deep Learning.

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

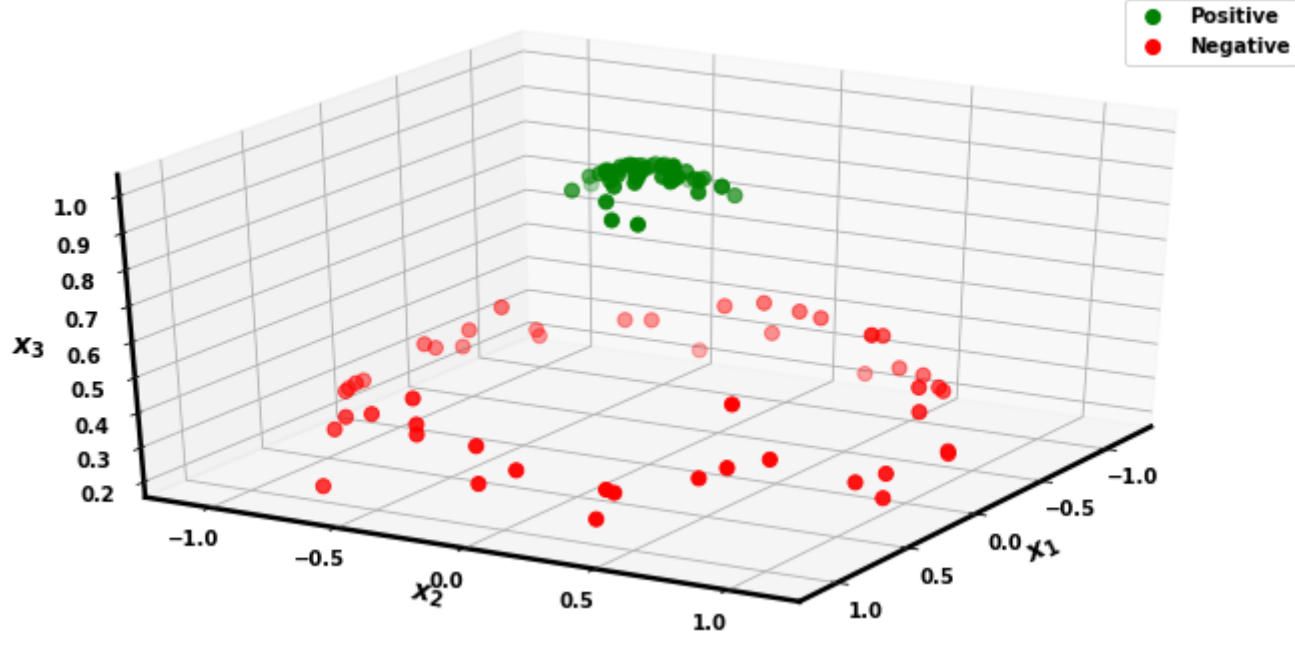
In [4]:
```
Xc, yc = svmh.make_circles(plot=True)
```



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

Nothing we have said above 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.

So rather than applying transformations as a preprocessing step (as above), the SVM integrates these special transformations into its basic algorithm.

These special class of transformations are known as *kernels*.

Kernels and the Kernel Trick

Kernels are a powerful form of transformation because they can be computed especially efficiently when used in conjunction with an SVC.

Consider a transformation Φ applied to a feature vector $\mathbf{x^{(i)}}$ of length n: $\phi(\mathbf{x^{(i)}})$

Quite often, $\phi(\mathbf{x^{(i)}})$ creates a vector $\tilde{\mathbf{x}}^{\mathbf{(i)}}$ of length $n' \gg n$.

For example, the "circle within a circle" example used a transformation that added one new feature.

Adding features may be costly (in fact, the RBF can add an infinite number of features !)

A *kernel transformation* allows the SVC to achieve the *effect* of having performed the transformation **without actually performing it**!

It is able to do this by relating the transformation to the mathematics of algorithms that solve the SVC optimization problem (minimization of Loss function).

We present, without proof, some properties of efficient solvers for the SVM problem.

The SVC optimization problem can be formulated as a Quadratic Programming problem.

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 $\mathbf{x^{(i)}} \cdot \mathbf{x}^{(i')} = \mathbf{x^{(i)}}^T \mathbf{x}^{(i')}$

Thus, if we transform $\mathbf{x^{(i)}}$ to $\tilde{\mathbf{x}}^{\mathbf{(i)}} = \Phi(\mathbf{x^{(i)}})$ our optimization problem will have sub-expressions of the form

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

A *kernel function* K is a **shortcut** for computing the *value* of this product **without** having to perform transformation ϕ.

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

This is particularly valuable

- If ϕ is expensive to compute
- If ϕ adds many features (i.e., $n' \gg n$)
    - Because the value of the dot product is a *scalar*, it is independent of n'

Aside

In order for a function to be a Kernel Function, it must satisfy Mercer's Theorem.

Not all transformations have corresponding kernel functions.

Rather

- There are a number of kernel functions
- Each kernel function corresponds to a transform

For this reason, although it seems backwards

- We will define a kernel function K
- And derive the ϕ implied by K

As we will see, there are many kernel functions whose derived ϕ's are similar to natural transformations.

To make this concrete:

- Consider the function K that compares two examples with a second order polynomial transformation

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

The ϕ corresponding to K (illustrated for $\mathbf{x}$ with 2 features) is

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

Notice that the Polynomial kernel above is not the same as the "natural" transformation that simply squares each feature.

- It adds a third "cross" term

We can prove that K is a kernel function by computing the dot product of $\phi(\mathbf{x^{(i)}}) \cdot \phi(\mathbf{x}^{(i')})$ and showing it to be equal to $K((\mathbf{x^{(i)}}, \mathbf{x}^{(i')})$

$$
\begin{aligned}
\phi(\mathbf{x^{(i)}})^T \phi(\mathbf{x}^{(i')}) &= \begin{pmatrix} \mathbf{x_1^{(i)}}^2 \\ \sqrt{2}\,\mathbf{x_1^{(i)}}\mathbf{x_2^{(i)}} \\ \mathbf{x_2^{(i)}}^2 \end{pmatrix}^T \begin{pmatrix} \mathbf{x_1}^{(i')^2} \\ \sqrt{2}\,\mathbf{x_1}^{(i')}\mathbf{x_2}^{(i')} \\ \mathbf{x_2}^{(i')^2} \end{pmatrix} \\
&= \mathbf{x_1^{(i)}}^2 \mathbf{x_1}^{(i')^2} + 2\mathbf{x_1^{(i)}}\mathbf{x_1}^{(i')}\mathbf{x_2^{(i)}}\mathbf{x_2}^{(i')} + \mathbf{x_2^{(i)}}^2 \mathbf{x_2}^{(i')^2} \\
&= \left(\mathbf{x_1^{(i)}}\mathbf{x_1}^{(i')} + \mathbf{x_2^{(i)}}\mathbf{x_2}^{(i')}\right)^2 \\
&= \left(\begin{pmatrix} \mathbf{x_1^{(i)}} \\ \mathbf{x_2^{(i)}} \end{pmatrix}^T \begin{pmatrix} \mathbf{x_1}^{(i')} \\ \mathbf{x_2}^{(i')} \end{pmatrix}\right)^2 \\
&= (\mathbf{x^{(i)}}^T \mathbf{x}^{(i')})^2
\end{aligned}
$$

You might try to simulate kernel functions with their natural counterparts

- Transform the data with the "natural transformation", e.g., square each feature
- Apply a linear SVC to the transformed data

As with the polynomial kernel above, the results won't be exactly the same.

For example, consider $\mathbf{x} = [\mathbf{x}_1, \mathbf{x}_2]$

- The Polynomial kernel (with degree 2) transforms $\mathbf{x}$ to
$$[\mathbf{x}_1^2, \mathbf{x}_2^2, \sqrt{2}\mathbf{x}_1\mathbf{x}_2]$$
- The Polynomial transform (with degree 2) in `sklearn` transforms $\mathbf{x}$ to
$$[1, \mathbf{x}_1, \mathbf{x}_2, \mathbf{x}_1^2, \mathbf{x}_2^2, \mathbf{x}_1\mathbf{x}_2]$$

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

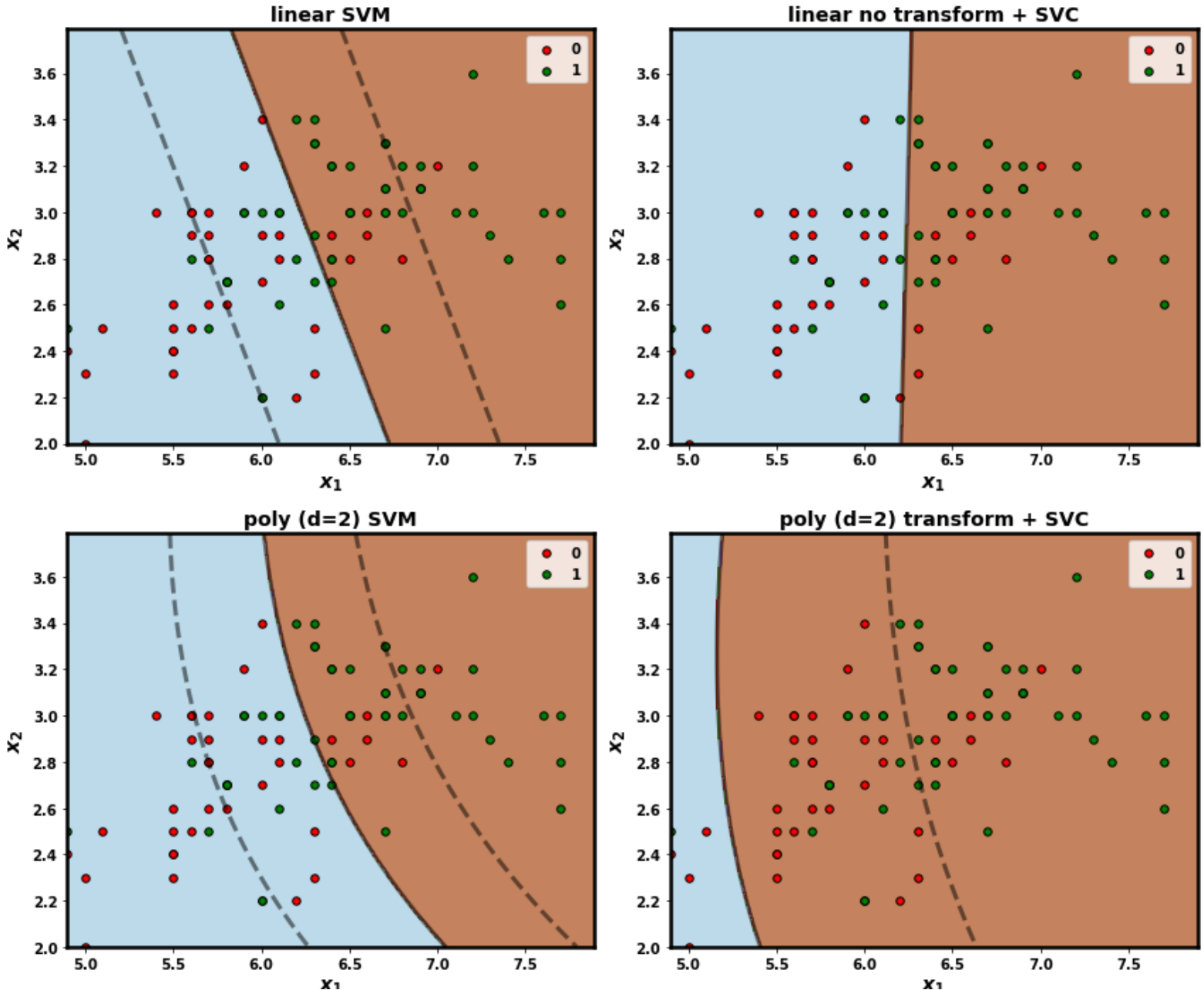
In [6]:

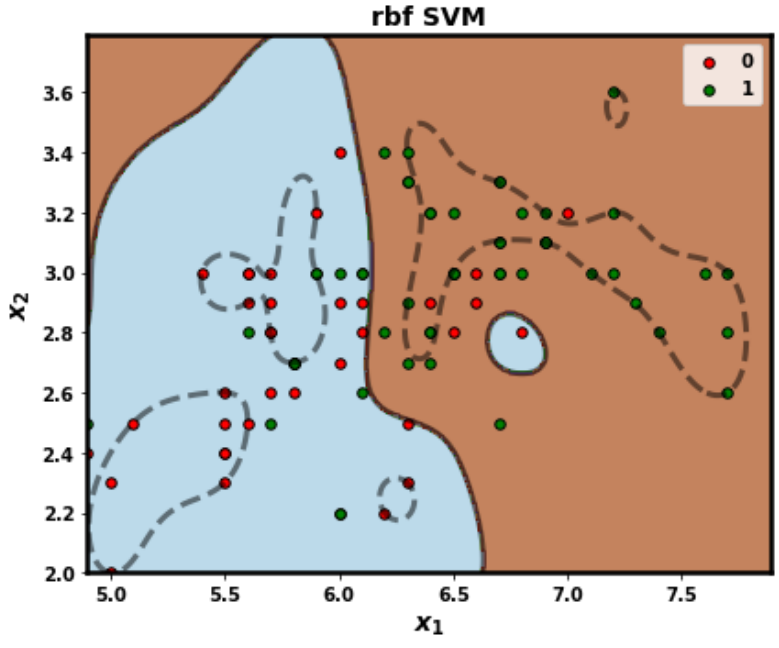
```
svmh = svm_helper.SVM_Helper()

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

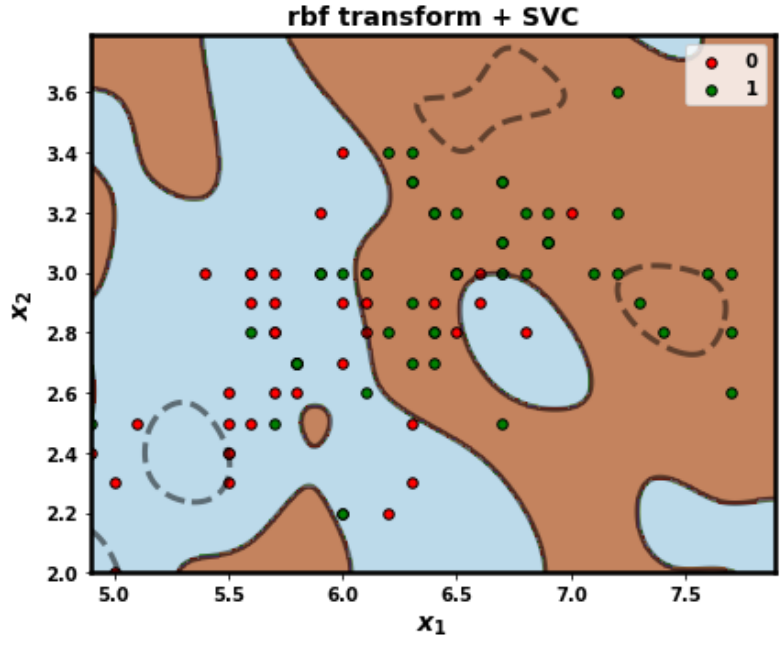
```
/home/kjp/anaconda3/lib/python3.7/site-packages/sklearn/svm/base.py:929: Conve
rgenceWarning: Liblinear failed to converge, increase the number of iteration
s.
  "the number of iterations.", ConvergenceWarning)
/home/kjp/anaconda3/lib/python3.7/site-packages/sklearn/svm/base.py:929: Conve
rgenceWarning: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


rbf SVM
0
1
3.6
3.4
3.2
3.0
2.8
2.6
2.4
2.2
2.0
x_2
5.0
5.5
6.0
6.5
7.0
7.5
x_1


rbf transform + SVC
0
1
3.6
3.4
3.2
3.0
2.8
2.6
2.4
2.2
2.0
x_2
5.0
5.5
6.0
6.5
7.0
7.5
x_1

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
```