Transformation to add a ”missing” numeric feature

Sometimes our models can't fit the data because some key feature is missing.

This was the case for our "curvy" data and Linear model: the polynomial term was missing.

The model

$$\mathbf{y} = \Theta_0 + \Theta_1 * \mathbf{x}_1$$

was not a good match for the data, but

$$\mathbf{y} = \Theta_0 + \Theta_1 * \mathbf{x}_1 + \Theta_2 * \mathbf{x}_1^2$$

was a much better fit.

Both models are linear, but the linearity of the relationship between target and features did not become clear until the missing feature $\mathbf{x}_1^2$ was added.

An identical transformation works for a Classification task:

By adding polynomial features

- We achieve separability
- The separating boundary is linear in *transformed features*
- But clearly not linear in raw features

In [4]:

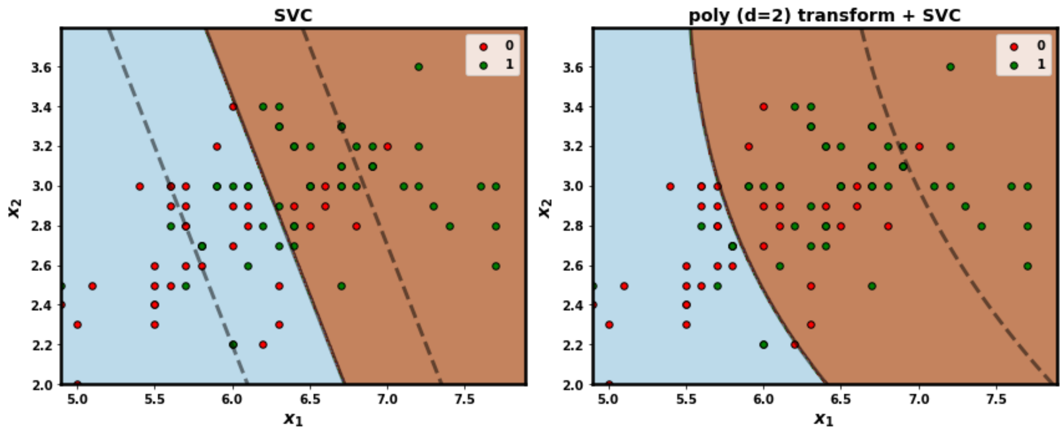
```
svmh = svm_helper.SVM_Helper()

_= svmh.create_kernel_data()

gamma=1
C=0.1

linear_kernel_svm = svm.SVC(kernel="linear", gamma=gamma)

# Pipelines
feature_map_poly2 = PolynomialFeatures(2)
poly2_approx = pipeline.Pipeline( [ ("feature map", feature_map_poly2),
                                    ("svm", svm.LinearSVC())
                                  ])

classifiers =  [ ("SVC", linear_kernel_svm),
                ("poly (d=2) transform + SVC", poly2_approx)
               ]
_= svmh.create_kernel_data(classifiers=classifiers)
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)
```

In [5]:

```
fig
```

Out[5]:



SVC
poly (d=2) transform + SVC
0
1
x_1
x_2
5.0
5.5
6.0
6.5
7.0
7.5
2.0
2.2
2.4
2.6
2.8
3.0
3.2
3.4
3.6

- Left plot shows a boundary that is linear in raw features
- Right plot show a boundary that is linear in transformed features
    - plotted in the dimensions of raw features

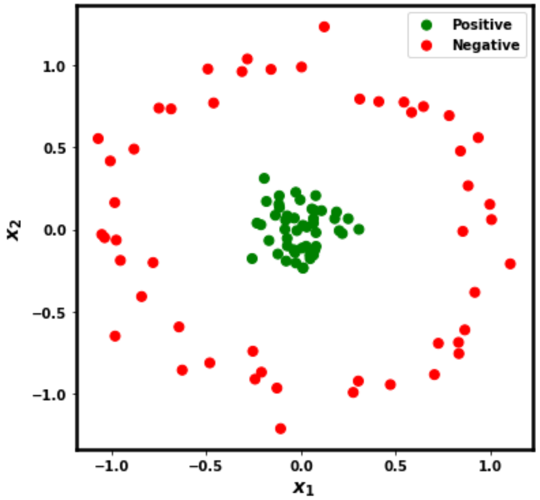
The transformation results in a boundary shape with greater flexibility.

Here is another common transformation that adds a feature to facilitate linear separability.

Consider the follow examples

- Which, to the eye, are separable
- But are not linearly separable

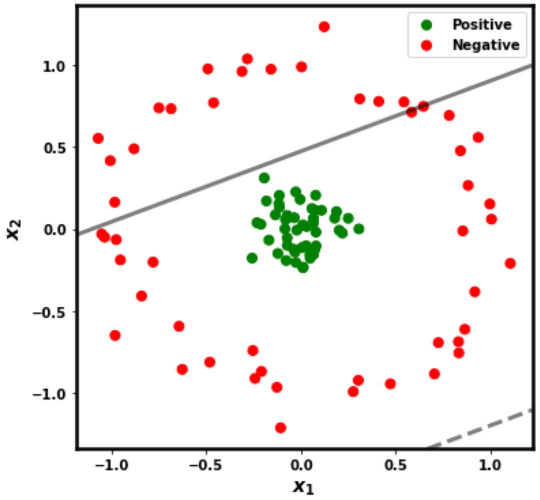
In [6]:
```
fig, ax = plt.subplots(1,1, figsize=(6,6) )
Xc, yc = svmh.make_circles(ax=ax, plot=True)
```



Visually, we can see that the classes are separable, but clearly not by a line.

Here's what one linear classifier (an SVC, which we will study later) produces

In [7]:
```
fig, ax = plt.subplots(1,1, figsize=(6,6) )
svm_clf = svmh.circles_linear(Xc, yc, ax=ax)
```

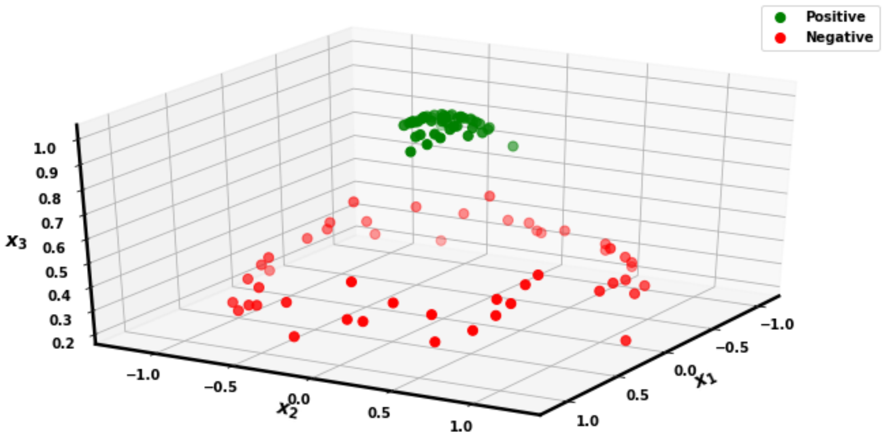


Let's add a new feature defined by the (Gaussian) *Radial Basis Function (RBF)*

$$\mathbf{x}_3 = e^{- \sum_j \mathbf{x}_j^2}$$

Our features are now 3 dimensional; let's look at the plot:

In [8]:
```
X_w_rbf = svmh.circles_rbf_transform(Xc)
_= svmh.plot_3D(X=X_w_rbf, y=yc )
```



Magic ! The new feature enables a plane that is parallel to the $\mathbf{x}_1, \mathbf{x}_2$ plane to separate the two classes.

We can write the RBF transformation in a more general form:

$$\text{RBF}(\mathbf{x}) = e^{-||\mathbf{x}-\mathbf{x}_c||}$$

- $||\mathbf{x} - \mathbf{x}_c||$ is a measure of the distance between example $\mathbf{x}$ and reference point $\mathbf{x}_c$
- In our case
    - $||\mathbf{x} - \mathbf{x}_c||$ is the L2 (Euclidean) distance
    - $\mathbf{x}_c$ is the origin $(0, 0)$

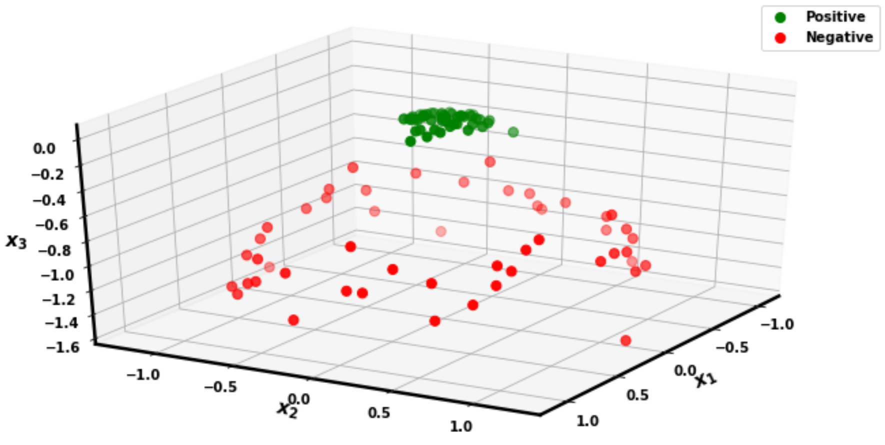
There is an even simpler transformation we could have used.

$$\mathbf{x}_3 = -\sum_j \mathbf{x}_j^2$$

That is: the (negative) of the L2 distance.

The advantage of the RBF is that it has little effect on points far from the reference point.

In [9]:
```
X_w_rad = svmh.circles_radius_transform(Xc)
_= svmh.plot_3D(X=X_w_rad, y=yc )
```



Although this transformation seems magical, we must be skeptical of magic

- There should be some *logical* justification for the added feature
- Without such logic: we are in danger of overfitting and will fail to generalize to test examples

For example:

- Perhaps $\mathbf{x}_1, \mathbf{x}_2$ are geographic coordinates (latitude/longitude)
- There is a distinction (different classes) based on distance from the city center $(\mathbf{x}_1, \mathbf{x}_2) = (0, 0)$
    - e.g. Urban/Suburban

Transformation to add a ”missing” categorical feature

Here is a less obvious case of a missing feature.

Suppose we obtain examples

- At different points in time
- Or in distinct geographies

What we often observe is that the examples

- From the same time/same place are similar to one another
- From different times/places are quite different

That is: the data naturally partitions into self-similar "groups".

How do we pool data that is similar intra-group but different across groups ?

Here is an artificial data set (Price as a function of Size) sampled at two different dates.

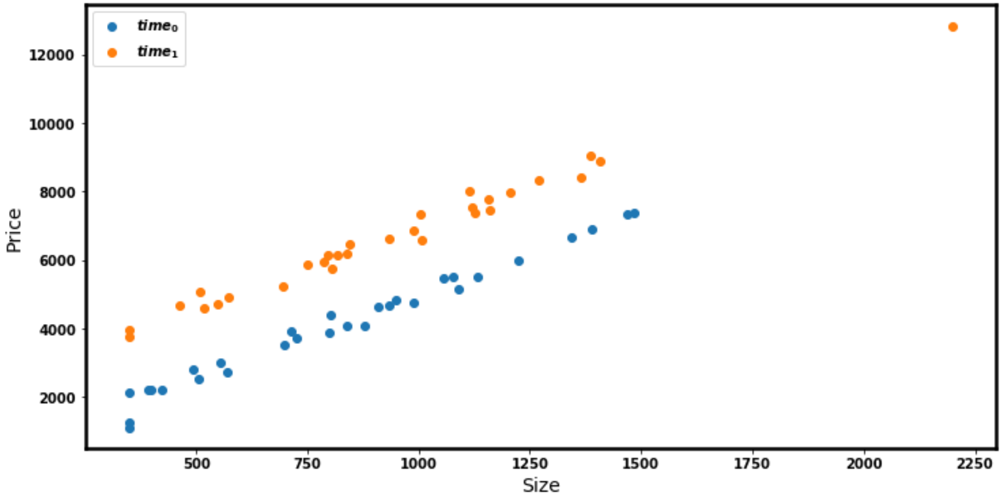
We will refer to the data at each date as a "group".

In [10]:

```
sph = transform_helper.ShiftedPrice_Helper()
series_over_time = sph.gen_data(m=30)

fig, ax = plt.subplots(1,1, figsize=(12,6) )
_= sph.plot_data(series_over_time, ax=ax)

plt.close(fig)
```

In [11]:
```
fig
```

Out[11]:



It appears that

- The groups are defined by examples gathered at different times: time_0, time_1
- There is a linear relationship *in each group* in isolation
- There slope of the relationship is *the same* across time
- But the intercept differs across groups
    - Perhaps this reflects a tax or rebate that is independent of price.

If we are correct in hypothesizing that each group is from the same distribution *except for* different intercepts

- Here's a set of equations that describes the data (separately for each of the two groups):

$$
\begin{aligned}
\mathbf{y}_{(\text{time}_0)} &= \Theta_{(\text{time}_0)} + \Theta_1 * \mathbf{x} \\
\mathbf{y}_{(\text{time}_1)} &= \Theta_{(\text{time}_1)} + \Theta_1 * \mathbf{x}
\end{aligned}
$$

Trying to fit a line (Linear Regression) as a function of the combined data will be disappointing.

We can try a transformation that compensates for the different intercepts.

Consider the transformation

- That adds a *categorical feature* `time` with two discrete values
- Encoded via OHE as two binary *indicators*

$$\text{Is}_j^{(\mathbf{i})} = \begin{cases} 1 & \text{if } \mathbf{x}^{(\mathbf{i})} \text{ is in group } j \\ 0 & \text{if } \mathbf{x}^{(\mathbf{i})} \text{ is NOT in group } j \end{cases}$$

For example:

- if example i is from the time 0 group

$$\mathrm{Is}_0^{\mathbf{(i)}} = 1$$

$$\mathrm{Is}_1^{\mathbf{(i)}} = 0$$

Because Is_0 and Is_1 are complementary

- The following single equation combines the two groups without losing the distinction

$$\mathbf{y} \quad = \quad \Theta_{(\mathrm{time}_0)} * \mathrm{Is}_0 + \Theta_{(\mathrm{time}_1)} * \mathrm{Is}_1 + \Theta_1 * \mathbf{x}$$

Effectively, the equation allows each group to have its own intercept !

This is equivalent to

- Fitting one line per group, with the same slope
- But different intercepts

Here's what the design matrix $\mathbf{X}''$ looks like when we add the two indicators:

$$\mathbf{X}'' = \begin{array}{c} \\ \\ \\ \end{array} \begin{array}{l} \begin{array}{ccc} \mathbf{Is}_0 & \mathbf{Is}_1 & \textbf{other features} \end{array} \\ \begin{pmatrix} 1 & 0 & \ldots \\ 0 & 1 & \ldots \\ \vdots & & \end{pmatrix} \begin{array}{l} \text{time}_0 \\ \text{time}_1 \\ \end{array} \end{array}$$

- Examples from the first time period look similar to the first row
- Examples from the second time period look similar to the second row

Notice that there is no "constant" feature in the design matrix

- Would correspond to the "intercept" term in linear regression

The reason for this is

- We already have **two** intercept-like terms Is_0 and Is_1
- The constant feature would be equal to the sum of these two features, for each example
    - Creates *dummy variable* trap for Linear Regression

Alternate method: non-homogeneous groups similar

Given our hypothesis that

$$\begin{aligned}\mathbf{y}_{(\text{time}_0)} &= \Theta_{(\text{time}_0)} + \Theta_1 * \mathbf{x} \\ \mathbf{y}_{(\text{time}_1)} &= \Theta_{(\text{time}_1)} + \Theta_1 * \mathbf{x}\end{aligned}$$

it would be natural to try to make the two groups appear similar by subtracting each group's intercept term from each example's target.

Unfortunately: we don't know these intercept terms a priori.

Here is a simple trick.

If

$$
\begin{array}{lll}
\mathbf{y^{(i)}} & = & \Theta_0 + \Theta_1 * \mathbf{x^{(i)}} & \text{hypothesize lin} \\
\frac{1}{m}\sum_i \mathbf{y^{(i)}} & = & \frac{1}{m}\sum_i (\Theta_0 + \Theta_1 * \mathbf{x^{(i)}}) & \text{sum over all ex} \\
\mathbf{\bar{y}} & = & \Theta_0 + \Theta_1 * \mathbf{\bar{x}} & \text{definition of av} \\
\Theta_0 = \mathbf{\bar{y}} - \Theta_1 * \mathbf{\bar{x}} & \text{re-arrange terms}
\end{array}
$$

This still doesn't allow us to know the value for Θ_0 because we don't know Θ_1 a priori.

But: if we demean each $\mathbf{x^{(i)}}$

$$\mathbf{x^{(i)}} = \mathbf{x^{(i)}} - \bar{\mathbf{x}}$$

then $\bar{\mathbf{x}} = 0$ and

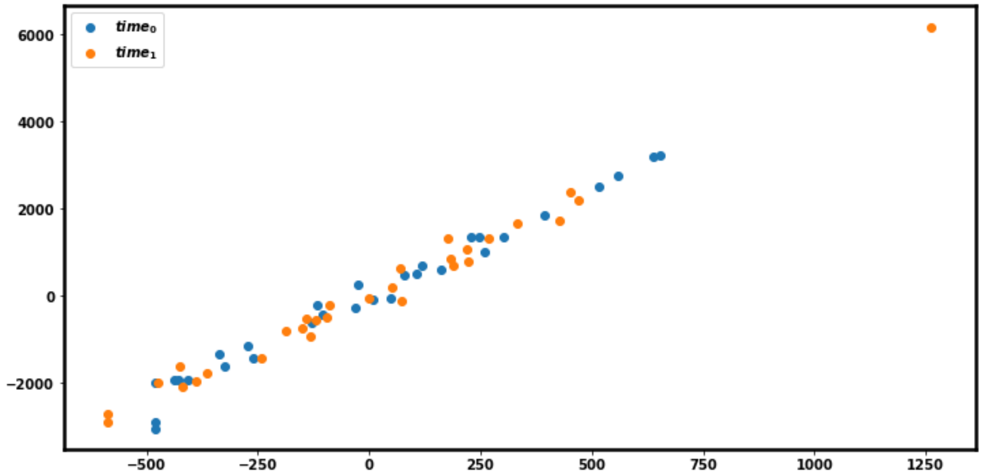
$$\Theta_0 = \bar{\mathbf{y}}$$

So

- if we demean the feature of each group separately
- we know the intercept for each group is the mean of the group's target
- we can make the two groups similar by subtracting the group mean from the target of each group

Here is a little code showing the effect

In [12]:
```
fig, ax = plt.subplots(1,1, figsize=(12,6) )

demean_x0 = sph.x0 - sph.x0.mean()
demean_x1 = sph.x1 - sph.x1.mean()

_= ax.scatter(demean_x0, sph.y0 - sph.y0.mean(), label="$time_0$")
_= ax.scatter(demean_x1, sph.y1 - sph.y1.mean(), label="$time_1$")
_= ax.legend()
```



Now it looks like each group comes from the same distribution.

- We can pool the observations from the two groups

Cross features

We have already seen a number of examples where adding a simple indicator succeeded in making our data linearly separable.

Sometimes though, an indicator on a *single* feature won't suffice

- But a synthetic feature that is the *product* of indicators will
- Can indicate an example's presence in the *intersection* of two groups

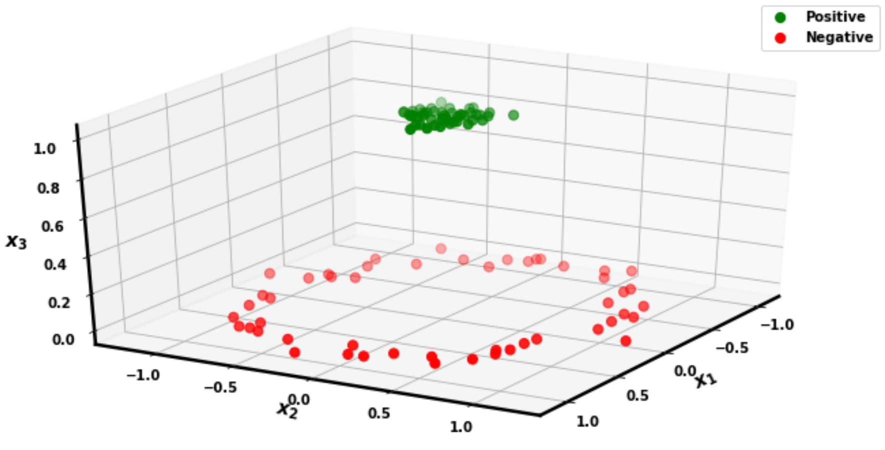
A synthetic feature created by combining (multiplying) two or more simple features is called a *cross term*

Let's revisit our circle classification dataset.

Here we create a cross feature that is `True` if two simpler features hold simultaneously

- $\mathbf{x}_1$ indicator: the horizontal offset from the origin $(0, 0)$ is "small"
- $\mathbf{x}_2$ indicator: the vertical offset from the origin $(0, 0)$ is "small"

In [13]:
```
X_w_sq = svmh.circles_square_transform(Xc)
_= svmh.plot_3D(X=X_w_sq, y=yc )
```



Here's the code to create the new feature "r" as a cross term

```
r = np.zeros( X.shape[0] ) r[ np.all(np.abs(X) <= 0.5, axis=1) ] = 1
```

We created a single "cross product" indicator $\text{Is}_{\text{in area}}$ as the product of two indicators, one per feature

$$\begin{aligned}
\text{Is}_{\text{in } \mathbf{x}_1 \text{ range}} &= (|\mathbf{x}_1| \le 0.5) \\
\text{Is}_{\text{in } \mathbf{x}_2 \text{ range}} &= (|\mathbf{x}_2| \le 0.5) \\
\text{Is}_{\text{in area}} &= \text{Is}_{\text{in } \mathbf{x}_1 \text{ range}} * \text{Is}_{\text{in } \mathbf{x}_2 \text{ range}}
\end{aligned}$$

The *cross term* $\text{Is}_{\text{in area}}$ indicates being in the *intersection* of $\text{Is}_{\text{in } \mathbf{x}_1 \text{ range}}$ and $\text{Is}_{\text{in } \mathbf{x}_2 \text{ range}}$.

Note that only the single $\mathrm{Is}_{\text{in area}}$ indicator is included in the equation and design matrix $\mathbf{X}''$

The isolated indicators

$$\mathrm{Is}_{\text{in } \mathbf{x}_1 \text{ range}}, \mathrm{Is}_{\text{in } \mathbf{x}_2 \text{ range}}$$

don't appear in the final regression equation -- they are used only to define $\mathrm{Is}_{\text{in area}}$

Cross terms are very tempting but can be abused when over-used.

To illustrate potential for abuse, it is possible to

- Create *one indicator per example*
- Create a cross term of the example indicator with each parameter in Θ
- This results in a completely separate set of parameters for *each* example
    - We "memorize" the data !

Here's a picture of the "per example" indicator

First, construct an indicator which is true

- if an example's feature j value is equal to the feature j value of example i:
$$\text{Is}_{\mathbf{x}_j^{(\mathbf{i})}} = (\mathbf{x}_j = \mathbf{x}_j^{(\mathbf{i})})$$

Now construct a cross feature that combines the indicators for all j and a single example i:

$$\text{Is}_{\text{example } i} \quad = \quad (\mathbf{x}_1 = \mathbf{x}_1^{(\mathbf{i})}) * (\mathbf{x}_2 = \mathbf{x}_2^{(\mathbf{i})})$$

This cross feature will be true on example i.

We can construct such a cross feature that recognizes any single example.

And here's the design matrix $\mathbf{X}''$ with a separate intercept per example.

$\mathbf{X}''$ has m intercept columns, one for each example, forming a diagonal of 1's

$$\mathbf{X}'' = \begin{pmatrix} \mathbf{const} & \text{Is}_{\text{example 1}} & \text{Is}_{\text{example 2}} & \text{Is}_{\text{example 3}} & \ldots & \mathbf{other\ features} \\ 1 & 1 & 0 & 0 & \ldots & \\ 1 & 0 & 1 & 0 & \ldots & \\ 1 & 0 & 0 & 1 & \ldots & \\ \vdots & & & & & \end{pmatrix}$$

We can do the same for $\Theta_1, \Theta_2, ..., \Theta_n$ resulting in a design matrix $\mathbf{X}''$ with $m * n$ indicators

- One per example per parameter

Here's a design matrix $\mathbf{X}''$ with one set of parameters per example: \

$$\mathbf{X}'' = \begin{pmatrix} \mathbf{const} & \text{Is}_{\text{example 1}} & (\text{Is}_{\text{example 1}} * \mathbf{x}_1) & (\text{Is}_{\text{example 1}} * \mathbf{x}_2) & \dots & \text{Is}_{\text{example 2}} & (\text{Is}_{\text{ex}} \\ 1 & 1 & \mathbf{x}_1^{(1)} & \mathbf{x}_2^{(1)} & \dots & 0 & \\ 1 & 0 & 0 & 0 & \dots & 1 & \\ \vdots & & & & & & \end{pmatrix}$$

Using this as the design matrix in Linear Regression

- Will get a perfect fit to training examples
- Would likely **not generalize** well to out of sample test examples.

When truly justified a small number of complex cross terms are quite powerful.

In [14]:

```
print("Done")
```

Done