

Transformations

We now spend more time on transforming data.

This may be *the most crucial* step in Classical Machine Learning. This is the art of Feature Engineering

- transforming features
- adding features all in the service of finding good predictors

Once we study Deep Learning, we will see how Neural Nets "learn" transformations that are useful.

We have already seen some rudimentary transformations

- polynomial transformation of our "curvy" data set
- "to categorical" for some variables in the Titanic problem
- the sigmoid transformation, used in the target of classification
- price to return for stock prediction

Why might we need Synthetic Variables (transformed variables)

- Compatibility
 - non-numeric to number (one-hot)
 - text to categorical
- To squeeze your problem into the model
 - need to make relationships linear to use a Linear model
- Because your model may *demand* it
 - Linear Regression demands
 - homoskedastic residuals
 - normality assumption
 - Some models need all features on similar scale
 - KNN distance metric

- To remove an extraneous relationship
 - "lumpy" data
 - clusters of data groups (e.g., sampled at different times) with differences
 - may violate model assumptions (e.g., independent residuals)
- To find the "right" relationship
 - may be a ratio/product between two features, or target and feature
 - area vs (length times width)
 - buying power (Price/GDP)

Transformations: introduction

- Many kinds of transformation
- A transformation may be applied for several reason

This makes it hard to provide a clean taxonomy

- I'll classify the type of transformation based on the example that introduces it.

Be aware however that there may be other reaons for using the transformation.

Linearity inducing transformations: Making data fit your model

Many models for Regression and Classification are based on *linear* relationships between targets and features.

One of the main uses for transformations is to create linearity where it does not naturally exist in the raw examples.

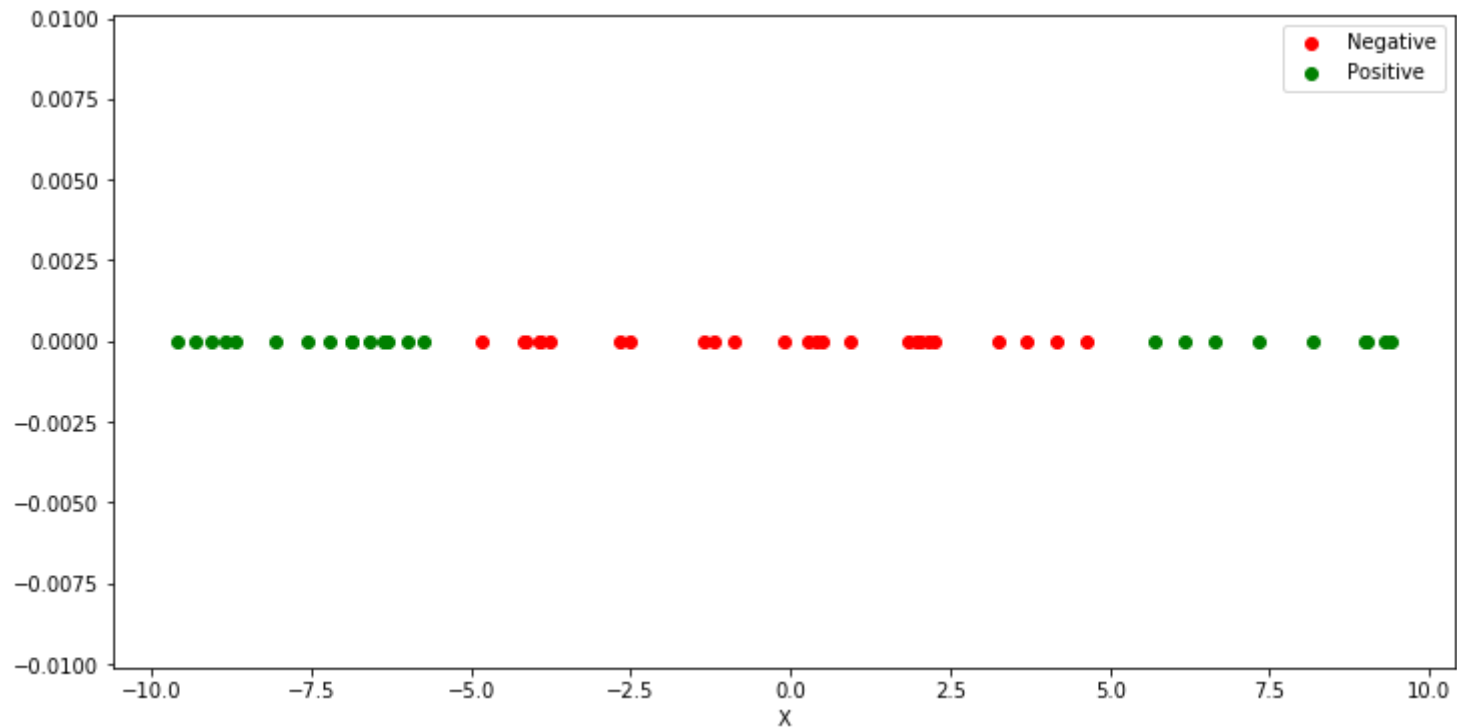
Transforming features to induce a linear relationship

As we saw in our introduction, the following Binary Classification task does not have an obvious linear separating boundary.

```
In [9]: th = transform_helper.Transformation_Helper()
fig_raw, ax_raw, fig_trans, ax_trans = th.LinearSeparate_1d_example(max_val=10,
num_examples=50, visible=False)

fig_raw
```

Out[9]:



The transformation that replaces the single feature \mathbf{x}_1 with x_1^2 achieves linear separability.

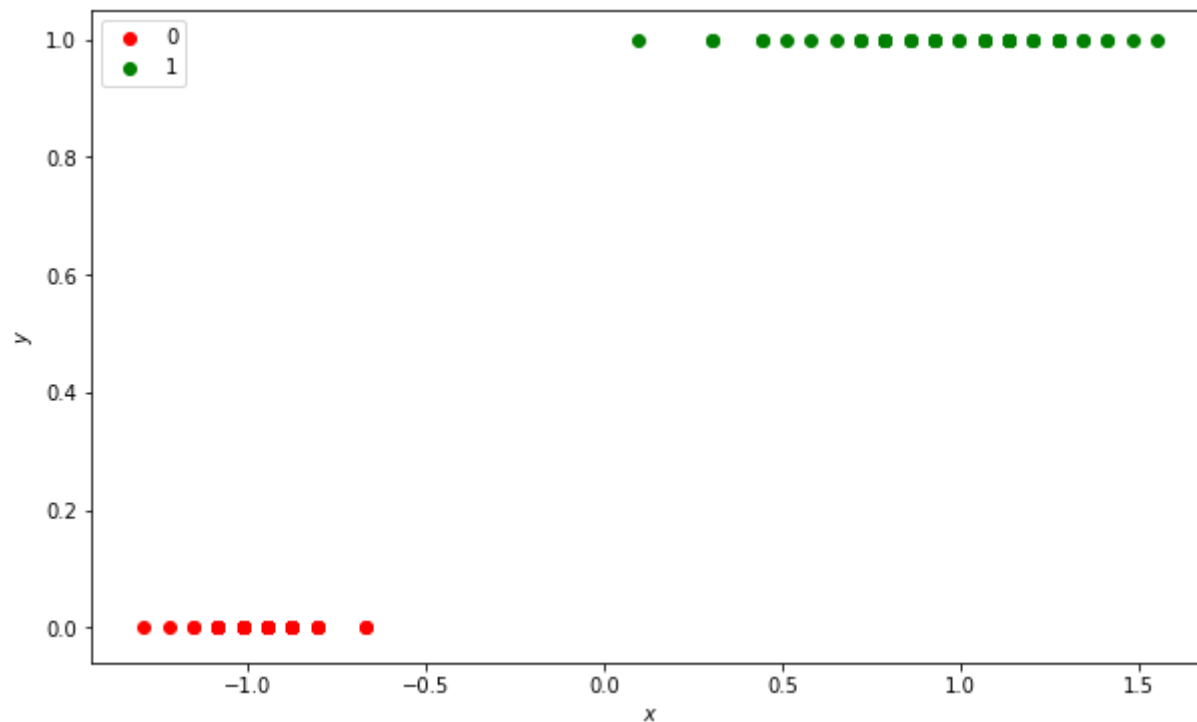
Transforming targets to induce a linear relationship

As we saw in our module on Binary Classification:

- discrete values are not fit well by a linear model

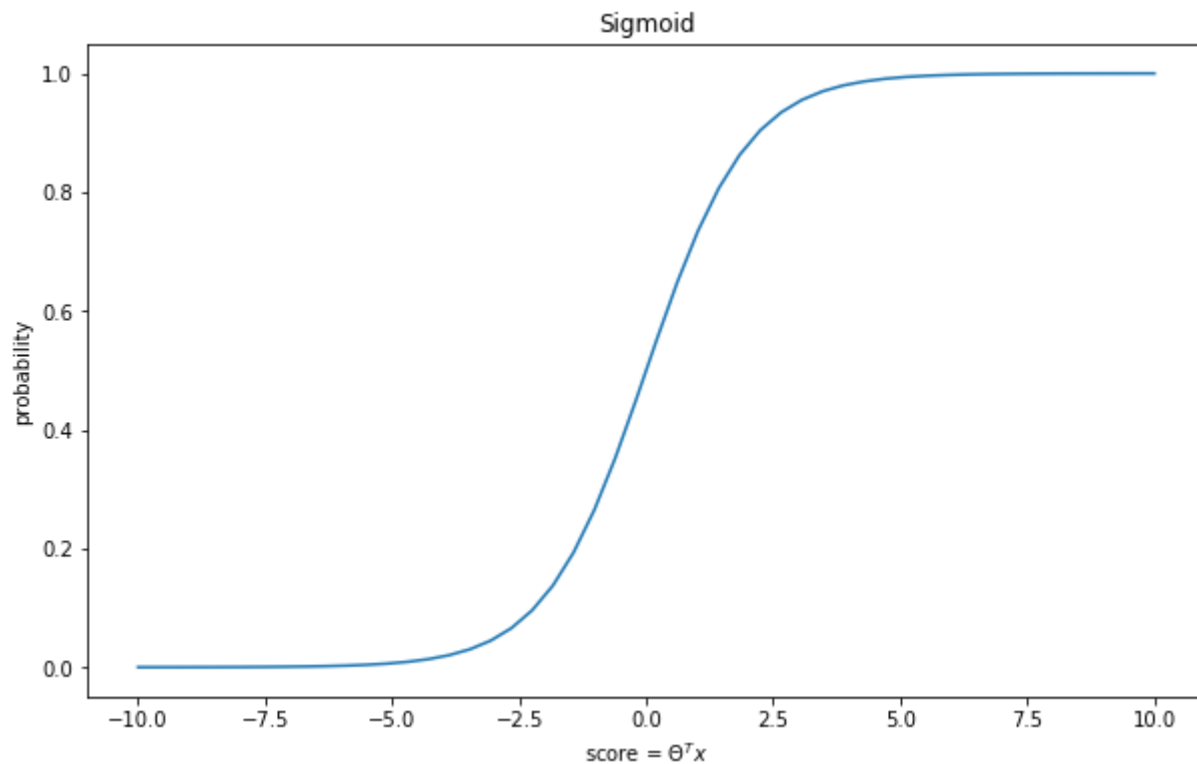
```
In [10]: lsh = class_helper.LinearSep_Helper()
X_ls, y_ls = lsh.load_iris()

fig, ax = plt.subplots(figsize=(10,6))
_ = lsh.plot_y_vs_x(ax, X_ls[:,0], y_ls)
```



The functional form of a sigmoid is a much better match

```
In [11]: fig, ax = plt.subplots(figsize=(10,6))  
         _ = lsh.plot_sigmoid(ax)
```



If we transform the probability into log odds, we make the targets linear in the features.

```
In [13]: s = np.linspace(-10,10, 50)
sigma_s = 1/(1 + np.exp(- s))

p = sigma_s
eps = 1e-8

log_odds = np.log(p/(1 -p + eps))

fig, axs = plt.subplots(1,2, figsize=(12,6))

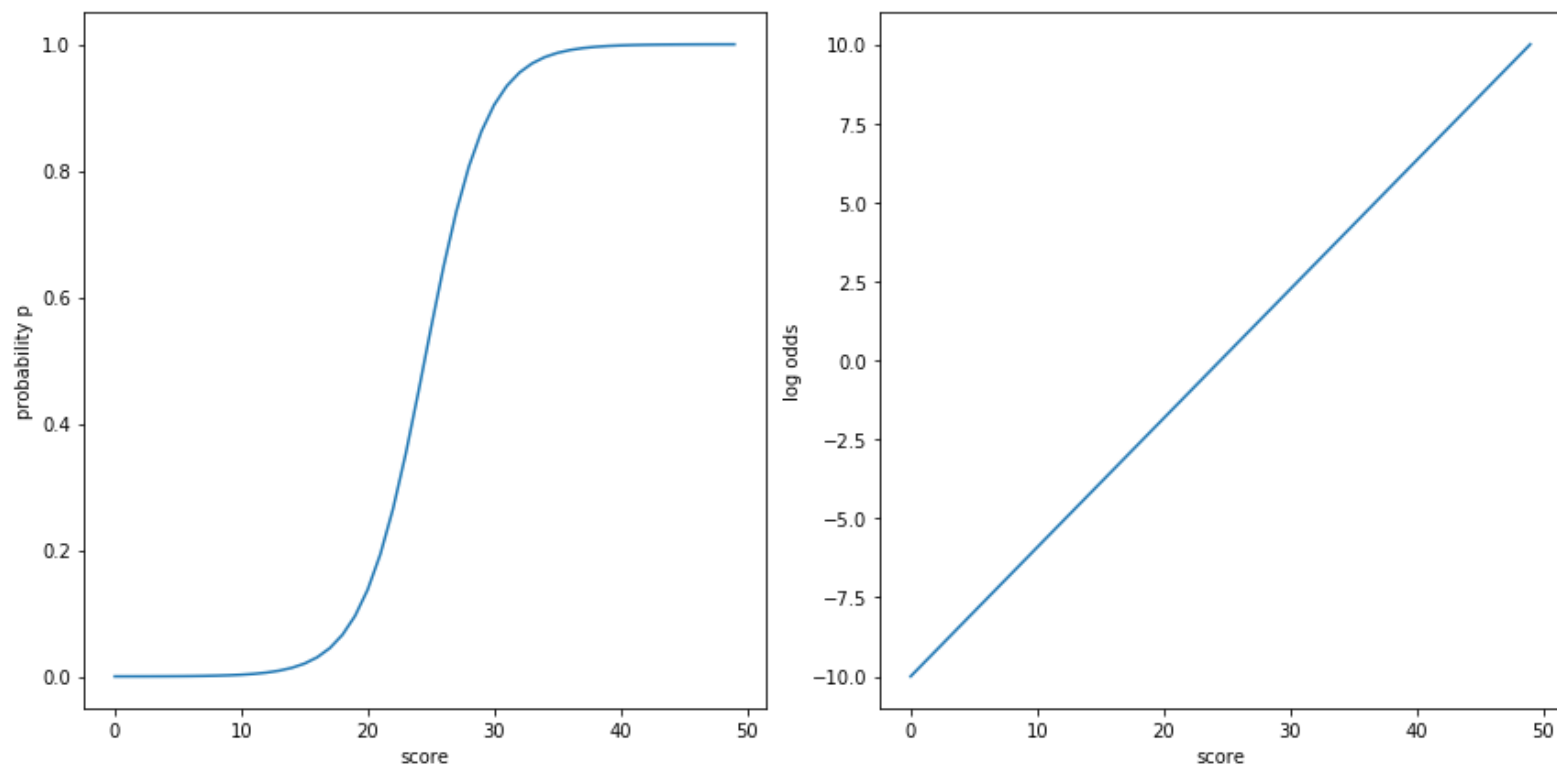
_ = axs[0].plot(p)
_ = axs[0].set_xlabel("score")
_ = axs[0].set_ylabel("probability p")

_ = axs[1].plot(log_odds)
_ = axs[1].set_xlabel("score")
_ = axs[1].set_ylabel("log odds")

fig.tight_layout()
plt.close(fig)
```

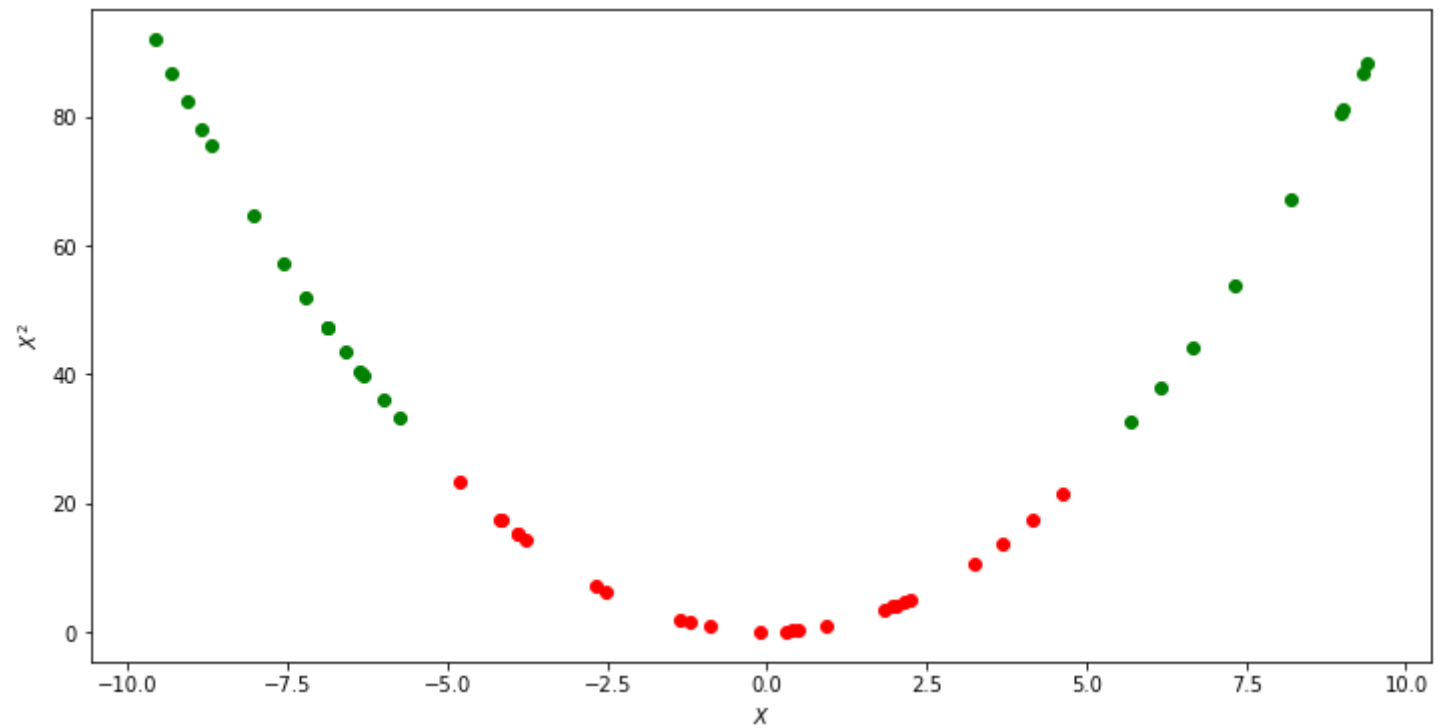
In [14]: fig

Out[14]:



In [15]: fig_trans

Out[15]:



Non-linear transformation + Linear Classifier = Non-linear boundary

To see the power of transformations, consider a transformation to induce linear separability.

After running this transformation, let's run a binary Linear Classifier and plot the boundary separating classes:

```

In [11]: 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())
                                   ])

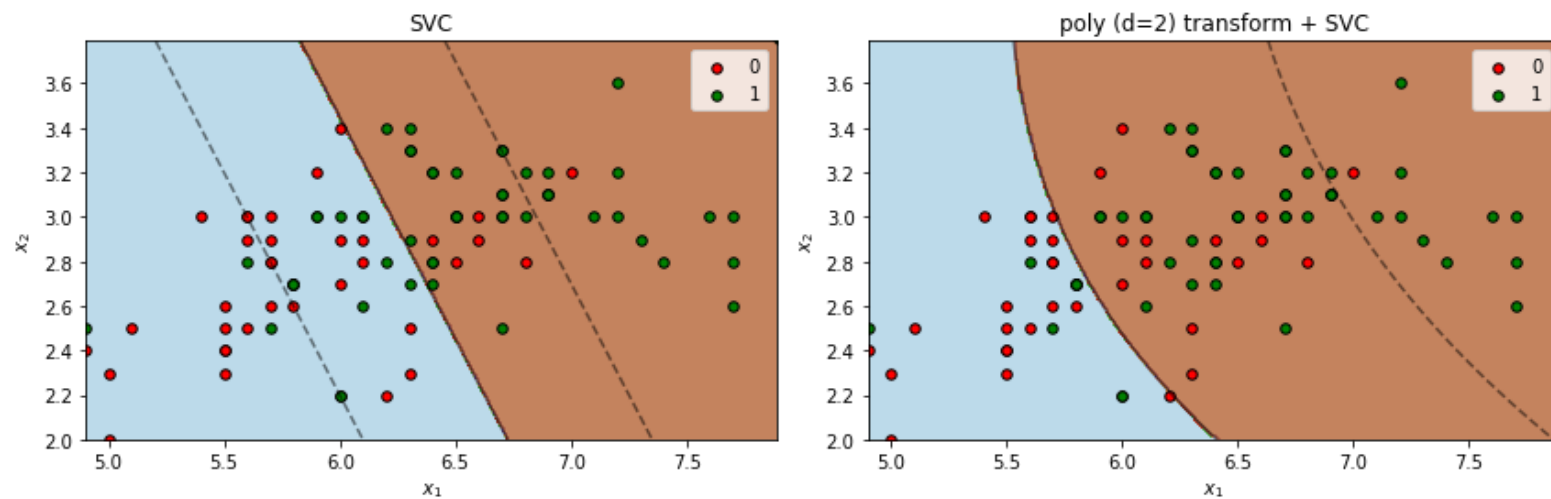
classifiers = [ ("SVC", linear_kernel_svm),
                 ("poly (d=2) transform + SVC", poly2_approx)
               ]
_ = svmh.create_kernel_data(classifiers=classifiers)
fig, axs = svmh.plot_kernel_vs_transform()
plt.close()

```

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

In [12]: fig

Out[12]:



The boundary on the right is no longer linear in the original features !

It **is** linear in the *transformed* features.

Thus, transformations are one way to create non-linear separating boundaries.



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

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

was not a good match for the data, but

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

was a much better fit.

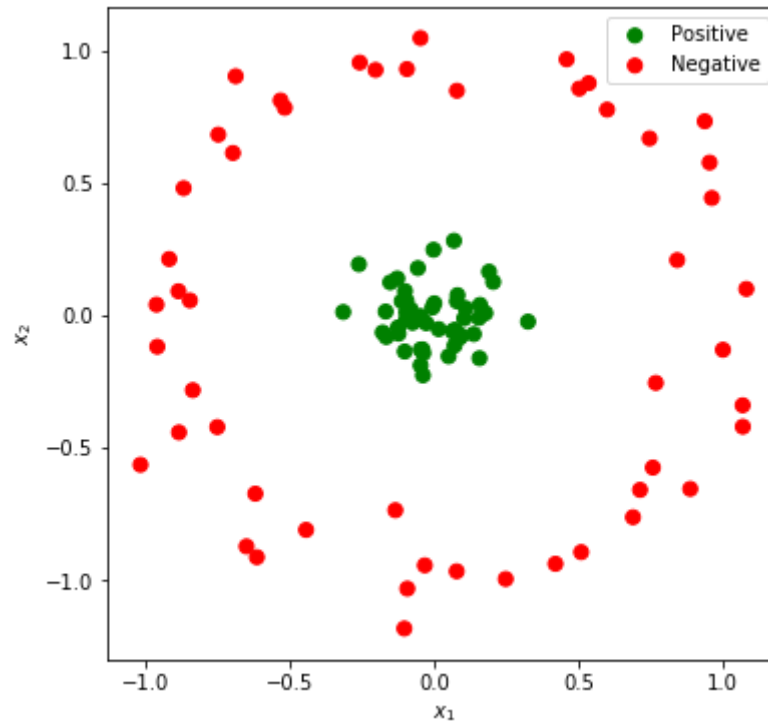
Both models are linear, but the linearity of the relationship between target and features did not become clear until the missing feature \mathbf{x}^2 was added.

Many classifiers attempt to create a linear boundary separating classes.

If the raw data is not linearly separable, sometimes adding a feature will make it so.

Here's a set of examples from two classes (Positive/Negative) that are not linearly separable.


```
In [16]: 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 [ ]: fig, ax = plt.subplots(1,1, figsize=(6,6) )  
        svm_clf = svmh.circles_linear(Xc, yc, ax=ax)
```

It tries to fit a linear model (solid line) to separate the data, but is only partially successful:

The SVC can isolate *some* of the red points to one side of the line, but not all.

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

$$x_3 = e^{-\sum_j \mathbf{x}_j^2}$$

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

```
In [ ]: 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.

The Radial Basis Function is a common transformation

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

- $\|\mathbf{x} - \mathbf{x}_c\|$ 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.

$$x_3 = - \sum_j \mathbf{x}_j^2$$

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 [ ]: 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 some 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
 - 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

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

The general problem is that we have multiple groups, each differing in some aspect ("lumpy" data)

Can we pool the examples across groups into one set, or must we keep them separate ?

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

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

```
In [ ]: sph = transform_helper.ShiftedPrice_Helper()
series_over_time = sph.gen_data()

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

plt.close(fig)
```

```
In [ ]: fig
```

It appears that

- At either time, there is a linear relationship between price and the independent variable
- The slope is the same at both times
- But the intercept is different
 - Perhaps this reflects a tax or rebate that is independent of price.

If we fit the combined raw examples, the difference in intercept will corrupt an obvious relationship.

Here's an equation that describes the data for each of the two groups:

$$y_{(\text{time}_0)} = \Theta_{(\text{time}_0)} + \Theta_1 * x$$

$$y_{(\text{time}_1)} = \Theta_{(\text{time}_1)} + \Theta_1 * x$$

Consider the transformation

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

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

For example:

- if example i is from the time 0 group

$$Is_0^{(i)} = 1$$

$$Is_1^{(i)} = 0$$

Because I_{s_0} and I_{s_1} are complementary

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

$$y = \Theta_{(\text{time}_0)} * I_{s_0} + \Theta_{(\text{time}_1)} * I_{s_1} + \Theta_1 * x$$

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

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

$$\mathbf{X}'' = \begin{pmatrix} \mathbf{Is}_0 & \mathbf{Is}_1 & \text{other features} \\ 1 & 0 & \dots \\ 0 & 1 & \dots \\ \vdots & & \end{pmatrix} \begin{matrix} \text{time}_0 \\ \text{time}_1 \end{matrix}$$

- 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

Note that the two groups don't have to be distinct because of time

- geography
- some other property with a constant impact

Adding the indicator allows the groups to share a common Θ_1 (slope) but have different Θ_0 (intercepts).



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

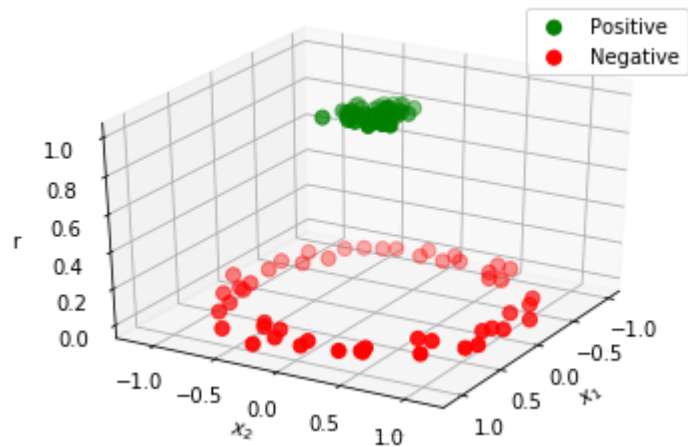
A synthetic feature created by combining (multiplying) two or more simple features is called a *cross term*

Let's revisit our circle classification dataset.

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

- x_1 indicator: the horizontal offset from the origin $(0, 0)$ is "small"
- x_2 indicator: the vertical offset from the origin $(0, 0)$ is "small"


```
In [105]: 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

$$\text{Is}_{\text{in } x_1 \text{ range}} = (|\mathbf{x}_1| \leq 1)$$

$$\text{Is}_{\text{in } x_2 \text{ range}} = (|\mathbf{x}_2| \leq 1)$$

$$\text{Is}_{\text{in area}} = \text{Is}_{\text{in } x_1 \text{ range}} * \text{Is}_{\text{in } x_2 \text{ range}}$$

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

- $\text{Is}_{\text{in } x_1 \text{ range}}, \text{Is}_{\text{in } x_2 \text{ range}}$
used only to define $\text{Is}_{\text{in area}}$

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

We can

- 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 !

Create indicator $I^{(i')}$ to match example i :

- create indicator $I_j^{(i')}$ to match feature j of example i
 - $I_j^{(i')} = \text{True}$ if $\mathbf{x}_j^{(i')} = \mathbf{x}_j^{(i)}$
- create such an indicator for each feature $1 \leq j \leq n$
- The n-fold cross feature $I^{(i')} = \prod_{1 \leq j \leq n} I_j^{(i')}$
 - is True when $i' = i$
 - it is the "Is example i " indicator

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

$$I_{\text{example 1}} = (\mathbf{x}_1 = \mathbf{x}_1^{(1)}) * (\mathbf{x}_2 = \mathbf{x}_2^{(1)})$$

$$I_{\text{example 2}} = (\mathbf{x}_1 = \mathbf{x}_1^{(2)}) * (\mathbf{x}_2 = \mathbf{x}_2^{(2)})$$

\vdots

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

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

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

- one per example per parameter

The associated regression equation has $m * n$ new parameters.

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

$$\mathbf{X}'' = \begin{pmatrix} \mathbf{const} & \mathbf{Is}_{\text{example 1}} & (\mathbf{Is}_{\text{example 1}} * \mathbf{x}_1) & (\mathbf{Is}_{\text{example 1}} * \mathbf{x}_2) & \dots & \mathbf{Is}_{\text{example 2}} & (\mathbf{Is}_{\text{example 2}} * \mathbf{x}_1) \\ 1 & 1 & x_1^{(1)} & x_2^{(1)} & \dots & 0 & 0 \\ 1 & 0 & 0 & 0 & \dots & 1 & 0 \\ \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots \end{pmatrix}$$

This would give a model with perfect results in the training set but which would likely generalize poorly.

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



Feature scaling

Sometimes it is necessary to restrict the *scale* of our data

- features and targets

There are several reasons

- Some models are sensitive to scale
 - features with large differences in scale may affect the model
- Some cost functions are sensitive to scale

Some scalers

Scaling transformations are relatively simple mathematically.

Let's discuss the *how* of scaling before delving into the *why*.

Note Scaling may be applied to a variable regardless of whether it is a feature or target.

MinMax

Covert to $[0, 1]$ range.

$$\hat{\mathbf{x}}_j^{(i)} = \frac{\mathbf{x}_j^{(i)} - \min_{1 \leq i \leq m} (\mathbf{x}_j^{(i)})}{\min_{1 \leq i \leq m} (\mathbf{x}_j^{(i)}) - \min_{1 \leq i \leq m} (\mathbf{x}_j^{(i)})}$$

New feature is measured in units of "fraction of range".

Standardize

$$\hat{x}_j^{(i)} = \frac{x_j^{(i)} - \bar{x}_j}{\sigma_{x_j}}$$

The new feature has a mean of 0, i.e., is *zero centered*.

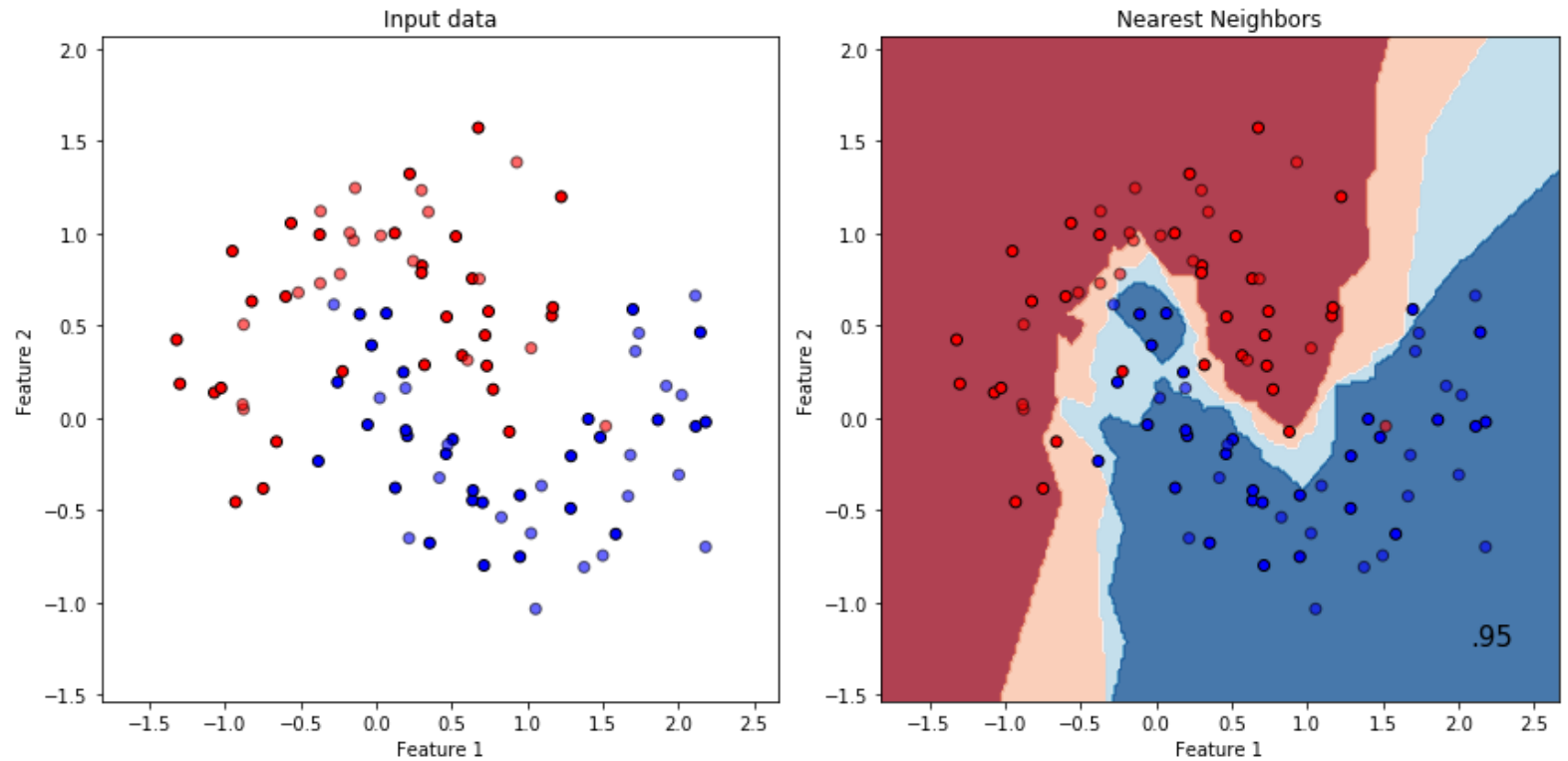
New feature is measured in units of "number of standard deviations".

Models sensitive to scale

One reason for scaling features is the mathematics of the model.

Let's illustrate using the Loss function (Euclidean distance) used by K Nearest Neighbors for classification.

```
In [52]: _ = kn. plot_classifiers(scale=False)
```



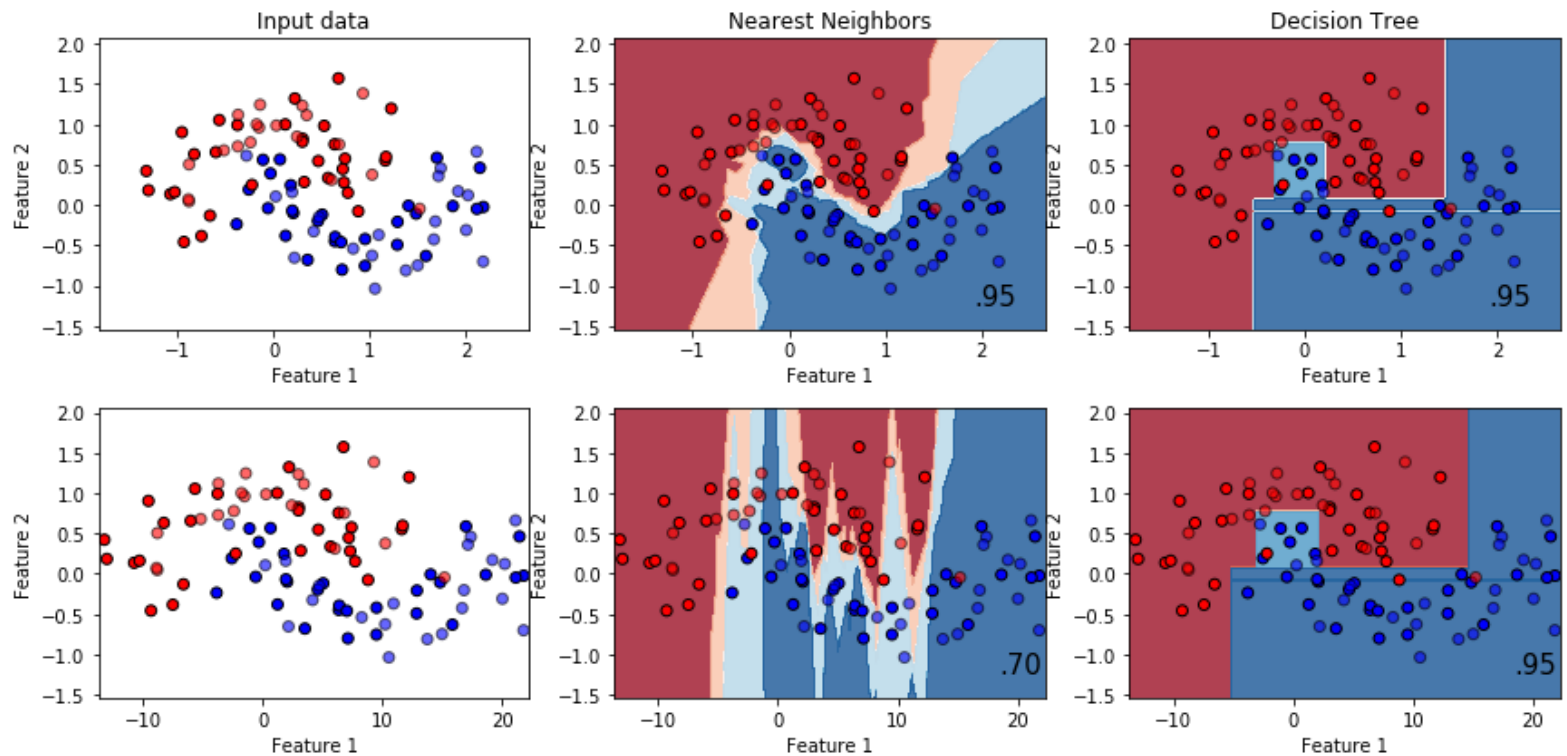
The two features are roughly on the same scale (min to max value).

What would happen if the scales were very different for the two features ?

Let's "stretch" the first feature by multiplying it by a factor of 10, leaving the second feature unchanged.

(n.b., accuracy is the number in the lower right of the prediction plot)

```
In [53]: kn = tmh.KNN_Helper()  
_ = kn.plot_classifiers(scale=False, num_ds=2)
```



The first row shows the original dataset; the second the "stretched" data set.

Note the horizontal scale of the lower plot is 10 times the upper.

- The plots look the same because the plotting routine keeps the plot sizes the same.

As you can see, the KNN classifier produces drastically different results on the two datasets.

Arguably KNN's predictions on the stretched dataset feel overfit.

- even with overfitting: accuracy drops from 95% to 70%

What is the reason for this ?

If you recall our brief introduction to KNN:

- the classifier measures the distance between features in a test example and feature in (each) training example
- the distance measure (L2, sum of squared feature-wise differences) is sensitive to scale

$(\mathbf{x}_1^{(i)}, \mathbf{x}_2^{(i)})$ training example i

$(10 * \mathbf{x}_1^{(i)}, \mathbf{x}_2^{(i)})$ stretched training example i

$(\mathbf{x}_1^{(test)}, \mathbf{x}_2^{(test)})$ test example

$(\mathbf{x}_1^{(test)} - \mathbf{x}_1^{(i)})^2 + (\mathbf{x}_2^{(test)} - \mathbf{x}_2^{(i)})^2$ distance from test

$(\mathbf{x}_1^{(test)} - 10 * \mathbf{x}_1^{(i)})^2 + (\mathbf{x}_2^{(test)} - \mathbf{x}_2^{(i)})^2$ stretched distance from test

Differences in \mathbf{x}_1 are much more important than differences in \mathbf{x}_2 in the stretched data.

Arguably, the visual plot (which preserved aspect ratio by de-scaling) shows the relationship didn't change.

- just imagine if the units changed from inches to millimeters

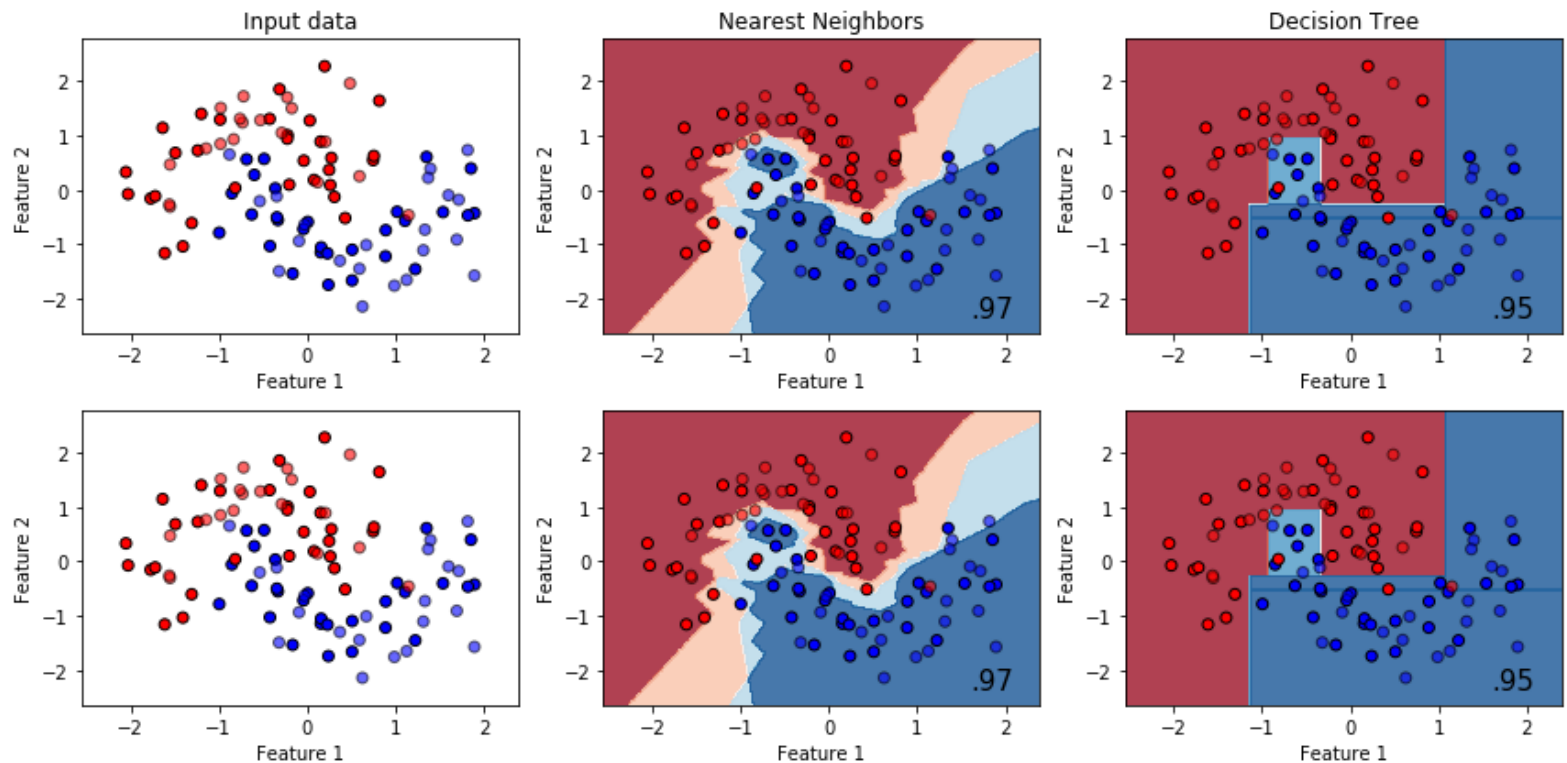
In contrast, the Decision Tree classifier (which will be the topic of a subsequent lecture) is *insensitive* to the changed scale.

- decision boundaries unchanged
- accuracy unchanged

This motivates Scaling transformations: putting features on similar scales.

Let's see how our classifiers perform if we re-scale the data (using standardization) prior to fitting

```
In [54]: kn = tmh.KNN_Helper()  
_ = kn.plot_classifiers(scale=True, num_ds=2)
```



Standardizing *both* features puts them on similar scale.

The KNN classifier's new behavior

- decision boundaries and accuracy is unaffected by stretching the data
- accuracy on original (unstretched) data *improves* from 95% to 97%
 - so even original data was not completely insensitive

Bottom line

- Know the math behind your model
 - KNN Loss is scale sensitive
 - Decision Tree Loss is *not* scale sensitive

To come:

- Principal Components, an Unsupervised Learning task
 - very different results with unscaled versus scaled features
 - Covariance (of features) matrix versus Correlation matrix

Parameters sensitive to scale

Consider the linear model

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

If we inflate \mathbf{x}_1 by a factor of 10 then Θ_1 will mathematically need to decrease by the same factor

$$\begin{aligned}\mathbf{y} &= \frac{\Theta_1}{10} * (10 * \mathbf{x}_1) \\ &= \Theta'_1 * \mathbf{x}'_1\end{aligned}$$

Changing the scale of the feature *also* changes the scale of the corresponding parameter.

Cost functions sensitive to scale: regularization penalty

There is another, easily overlooked place where scale of features matter

- Regularization penalty

Because scaling the feature may also scale the corresponding parameter

- the regularization penalty will change too

[Ridge Regression \(external/PythonDataScienceHandbook/notebooks/05.06-Linear-Regression.ipynb#Ridge-regression-%28\\$L_2\\$-Regularization%29\)](https://external/PythonDataScienceHandbook/notebooks/05.06-Linear-Regression.ipynb#Ridge-regression-%28L_2-Regularization%29)

Penalty is: sum (over parameters) of squared parameter value

$$P = \alpha \sum_{n=1}^N \Theta_n^2$$

In our example

- Inflating the sole parameter \mathbf{x}_1 by a factor of 10
- Decreases the corresponding parameter Θ_1 by a factor of 10
- Decreases the corresponding penalty by a factor of 100

Note that even something as "benign" as changing units from meters to kilometers has a big effect.

[Lasso Regression \(external/PythonDataScienceHandbook/notebooks/05.06-Linear-Regression.ipynb#Lasso-regression-\(L1-regularization\)\)](https://external/PythonDataScienceHandbook/notebooks/05.06-Linear-Regression.ipynb#Lasso-regression-(L1-regularization))

Penalty is: sum (over parameters) of parameter's absolute value

$$P = \alpha \sum_{n=1}^N |\Theta_n|$$

In our example

- Increasing the sole \mathbf{x}_1 parameter by a factor of 10
- Decreases the corresponding parameter Θ_1 by a factor of 10
- Decreases the corresponding penalty by a factor of 10

Feature scaling: summary

- A very simple operation
- With subtle but significant impact

Preview:

- In the Deep Learning part of the course
- We will see the need to scale features
 - Small magnitude



Normalization

Our goal, as usual, is to find a relationship between targets and features.

But suppose our set of examples is not uniform

- Examples drawn at different times
- Examples drawn from different geographies

The examples may describe the *same* fundamental relationship, but the units used to measure the target or features might be different.

Here is an artificial data set relating house Price to size.

```
In [46]: rph = transform_helper.RelativePrice_Helper()
series = rph.gen_data(attrs=["area", "price"])

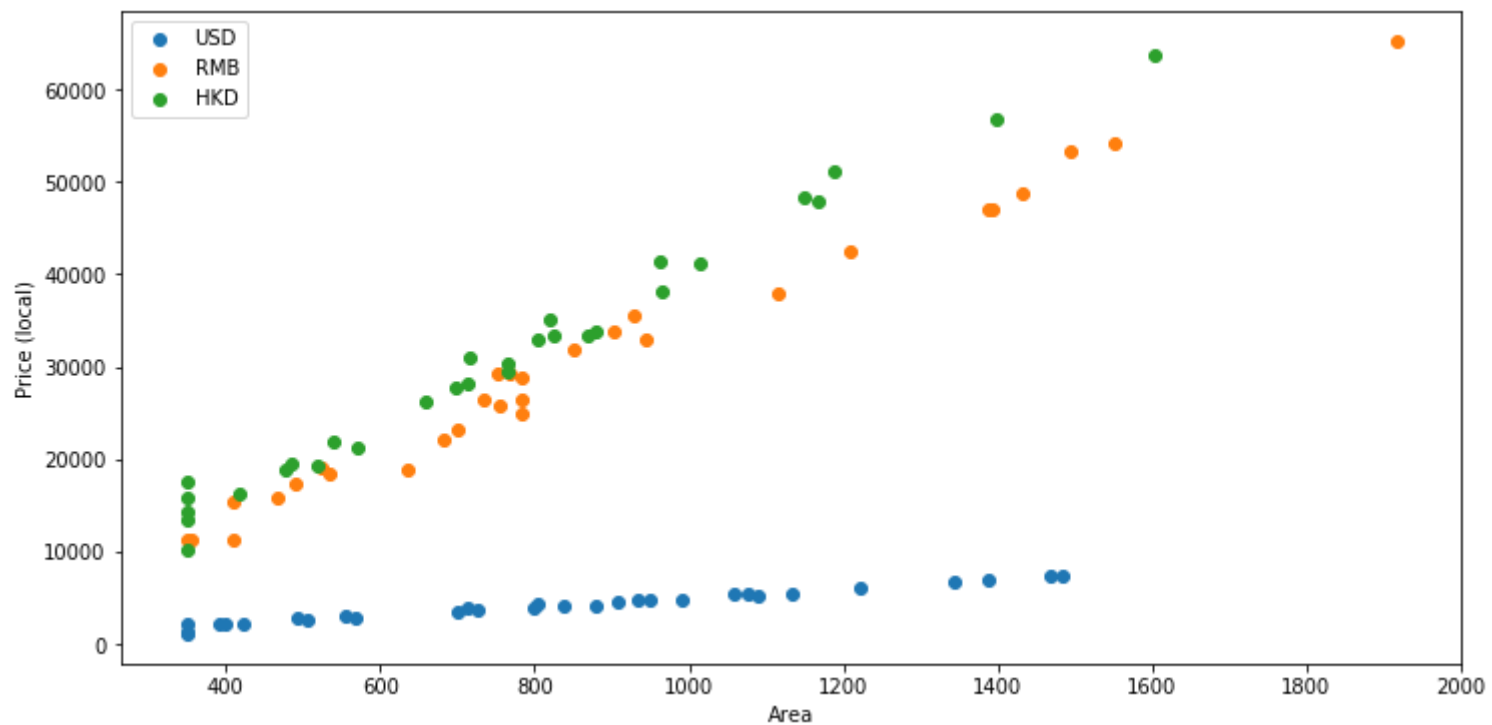
labels=[ "USD", "USD", "RMB", "HKD" ]

fig, ax = plt.subplots(1,1, figsize=(12,6))
_= rph.plot_data(series, ax=ax, labels=labels, xlabel="Area", ylabel="Price (local)")

plt.close(fig)
```

In [47]: fig

Out[47]:



Our examples are split into multiple groups, identified by color

- Price denominated in different currencies

The apparant diversity in the target may obscure a simple relationship

```
In [49]: # Relative price levels
rel_price = rph.relative_price()

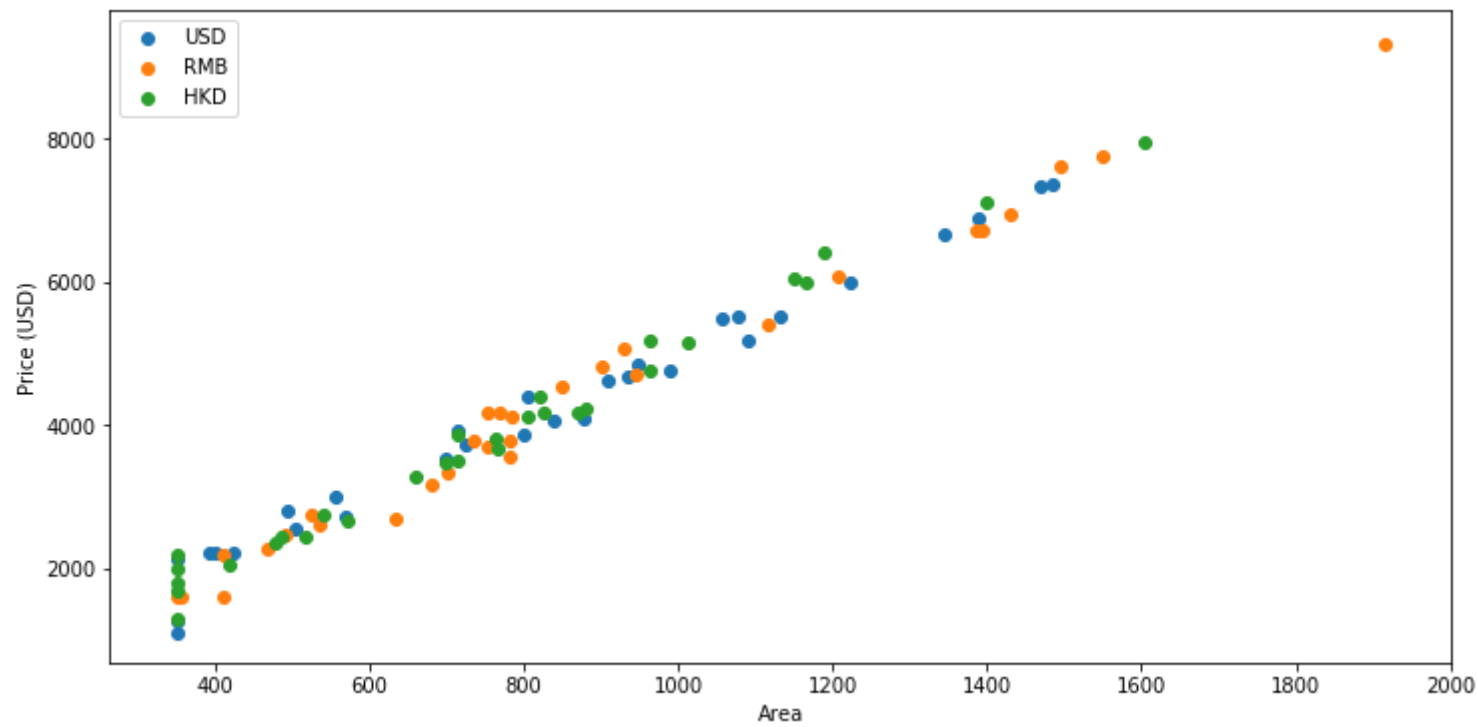
# Normalize the price of each series by the relative price
series_normalized = [ series[i]/(1,rel_price[i]) for i in range(len(series))]

fig, ax = plt.subplots(1,1, figsize=(12,6))
_ = rph.plot_data(series_normalized, ax=ax, labels=labels, xlabel="Area", ylabel=
"Price (USD)")

plt.close(fig)
```

```
In [50]: fig
```

Out[50]:



Once all prices are expressed in the same units, it becomes clear that the relationship is uniform across currencies.

We performed a transformation that "normalized" the units of a variable across different examples,

- Creating a linear relationship.

Here's an equation that describes the data for each of the first two groups:

$$y^{\text{group}_0} = \beta_0^{\text{group}} * (\Theta_0 + \Theta_1 * x)$$

$$y^{\text{group}_1} = \beta_1^{\text{group}} * (\Theta_0 + \Theta_1 * x)$$

And the regression equation for the transformed data:

$$\frac{y^{\text{group}_0}}{\beta_0^{\text{group}}} = \Theta_0 + \Theta_1 * x$$

$$\frac{y^{\text{group}_1}}{\beta_1^{\text{group}}} = \Theta_0 + \Theta_1 * x$$

In []: The phenomenon of variables denominated **in** different units between groups **is not** uncommon

- Groups of examples observed at different times (e.g., adjust **for** time-varying general price level)
- Groups of examples expressed **in** dollars **and** others **in** cents

In our case, the only variable that needed to be normalized was the target.

You can imagine situations **in** which features (e.g., Area) need to be normalized

- Meters versus feet

You don't have to get too fancy when adjusting price levels

- consider redominating in units of "number of McDonald's burgers" !
- may work across time and currency

As long as all commodity prices change the same, this should work.

Categorical transformation

We encountered the "to categorical" transformation in the lecture on Classification when talking about Categorical variables.

The recommended solution was to use One Hot Encoding for Categorical variables.

Normality inducing transformations

Adding missing feature as a normality inducing transformation

The Linear Regression model is

$$\mathbf{y} = \Theta^T \mathbf{x} + \epsilon$$

As explained before, Regression produces a conditional probability

$$p(\hat{\mathbf{y}}|\mathbf{x})$$

where $\hat{\mathbf{y}}$ and ϵ are *Normally distributed variables*.

Assumptions of the Linear Regression model are violated if

- ϵ is not Normal
- the individual $\epsilon^{(i)}$ display a pattern
- the individual $\epsilon^{(i)}$ have different variances (heteroskedastic)

One reason for failure of these assumptions is a missing feature

- "curvy" data set and Linear model
 - we saw pattern of errors: larger in tails
 - variances increased in tail

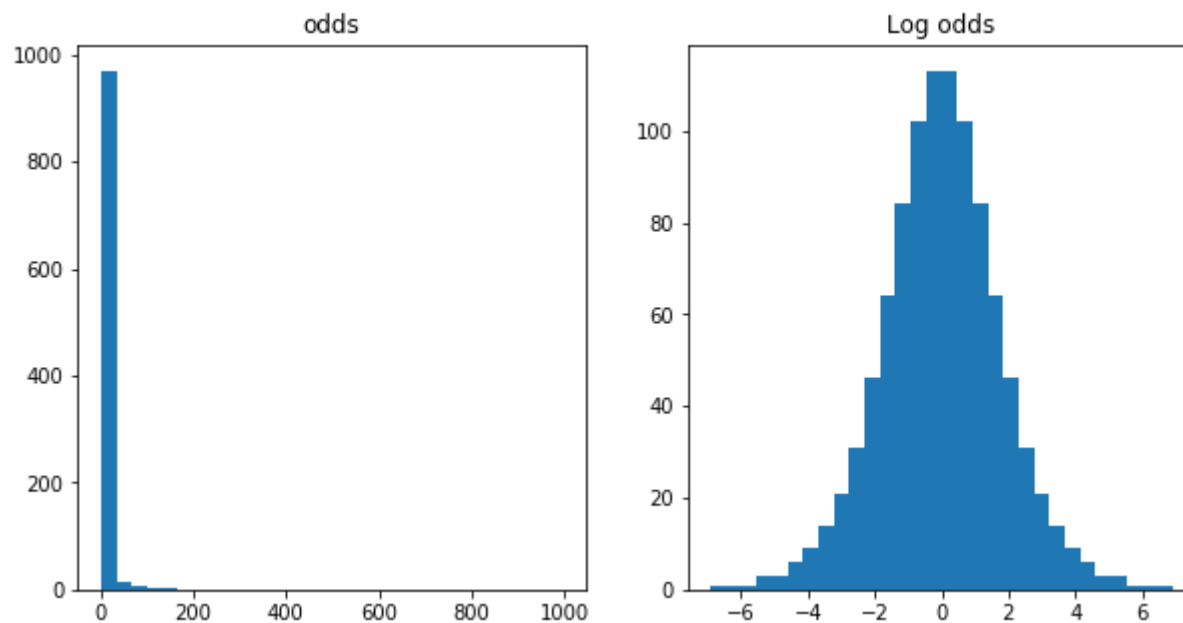
Adding a feature (e.g., second order polynomial term for the curvy data set) can be seen as a normality inducing transformation.

Log transformation

We've seen this in our lecture on Logistic Regression

- the probabilities are *not* normally distributed
- the odds are *not* normally distributed
- the *log odds* is normally distributed

```
In [21]: tf = tmh.TransformHelper()  
tf.plot_odds()
```



$$\begin{aligned}
 \frac{\hat{p}}{1-\hat{p}} &= \frac{\frac{1}{1+e^{-s}}}{1-\frac{1}{1+e^{-s}}} \\
 &= \frac{\frac{1}{1+e^{-s}}}{\frac{1+e^{-s}-1}{1+e^{-s}}} \\
 &= \frac{1}{e^{-s}} \\
 &= e^s
 \end{aligned}$$

So LogisticRegression is really just a LinearRegression with a transformed target

$$\log\left(\frac{\hat{p}}{1-\hat{p}}\right) = \Theta^T \cdot x$$

Other transformations

Centering

Transforming a feature to have mean 0.

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

- low values now become negative
 - more clearly indicates deletrious effect than a low, positive number
 - example: Star Ratings for movies
- some algos (PCA) need centered data

Bucketing/Binning

- Target may be linear in a feature only in broad ranges of the feature
 - income vs age
 - very young (below working age) all income is identical (0)
 - very old (above retirement) - no job related income
 - Latitude/Longitude
 - small changes matter MUCH less than big changes
- Converts numerical feature
 - into categorical **Is bucket 1, Is bucket 2, ...**
 - ordinal: replace value with center value of bin

Bucket size choices:

- Equal spaced buckets
- Equal quantile buckets

Lesson Don't fit a square peg (non-linear response) into a round hole (linear model)

Outliers

Pull in extreme values to reduce their influence on the fit.

- Clipping, Winsorization

Feature engineering example: Geron Housing Data

[Geron housing data \(external/hands-on-ml/02_end_to_end_machine_learning_project.ipynb#Get-the-data\)](#)

Data: California Housing Prices Data (AG Chapt 2)

```
In [23]: housing = tm.load_housing_data()  
housing.head()
```

Out[23]:

	longitude	latitude	housing_median_age	total_rooms	total_bedrooms	population	households	median_income	median_house_value
0	-122.23	37.88	41.0	880.0	129.0	322.0	126.0	8.3252	452600.0
1	-122.22	37.86	21.0	7099.0	1106.0	2401.0	1138.0	8.3014	358500.0
2	-122.24	37.85	52.0	1467.0	190.0	496.0	177.0	7.2574	352100.0
3	-122.25	37.85	52.0	1274.0	235.0	558.0	219.0	5.6431	341300.0
4	-122.25	37.85	52.0	1627.0	280.0	565.0	259.0	3.8462	342200.0

Each row represents a district

Goal is to predict Median House Value in the district.

- Features
 - total_rooms: number of rooms in the district
 - total_bedrooms: number of bedrooms in the district
 - households; number of households in the district
 - population: number of people in the district
 - median_income: median income of people in the district
 -
- Target
 - median_house_value: median value of a house in the district

What potential issues pop out ?

Some *raw* features depend on the *size* of the district, so can't compare the same feature across districts.

- total_rooms, total_bedrooms, households, population

Feature engineering: Normalization

The first transformation is a kind of normalization

- adjusting a feature for the size of the district
 - divide district aggregate by number of items in the district
- makes the feature comparable across districts of different sizes

```
In [24]: housing["rooms_per_household"] = housing["total_rooms"]/housing["households"]  
housing["bedrooms_per_room"] = housing["total_bedrooms"]/housing["total_rooms"]  
housing["population_per_household"]=housing["population"]/housing["households"]
```

By synthesizing ratios:

- rooms/household
- population/household
- bedrooms/room

we have transformed the raw features into processed features that will probably be better predictors

Feature engineering: Bucketing and Clipping

- Will \$1 more of income *really* predict higher housing prices in the district ?
 - create income buckets
 - Disclaimer
 - In the book this is **not** a feature but something used to "stratify" the sample
 - We are taking a bit of artistic license to make a point

```
In [25]: print("Median income ranges from {min:.1f} to {max:.1f}".format(min=housing["median_income"].min(),
                                                                    max=housing["median_income"].max()))
        # Divide by 1.5 to limit the number of income categories
        housing["income_cat"] = np.ceil(housing["median_income"] / 1.5)

        print("Income buckets")
        housing["income_cat"].value_counts()
```

Median income ranges from 0.5 to 15.0

Income buckets

```
Out[25]: 3.0      7236
         2.0      6581
         4.0      3639
         5.0      1423
         1.0       822
         6.0       532
         7.0       189
         8.0       105
         9.0        50
        11.0        49
        10.0        14
        Name: income_cat, dtype: int64
```

Still a lot of buckets. One theory is that incomes above bucket 5 don't predict housing prices, so clip.

```
In [26]: # Label those above 5 as 5  
housing["income_cat_clipped"] = housing["income_cat"].where(housing["income_cat"  
] < 5, 5.0)  
print("Income buckets")  
housing["income_cat_clipped"].value_counts()
```

Income buckets

```
Out[26]: 3.0    7236  
        2.0    6581  
        4.0    3639  
        5.0    2362  
        1.0     822  
        Name: income_cat_clipped, dtype: int64
```

In [27]: `print("Done")`

Done