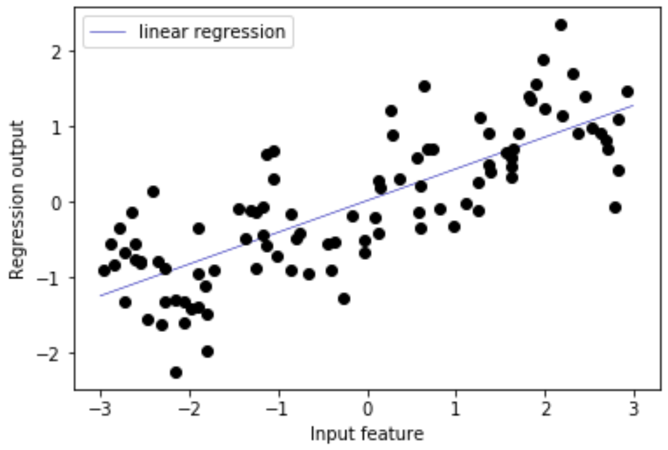
Preprocessing

- Many of the algorithms that we've seen are greatly affected by *how* you represent the training data
- Scaling, numeric/categorical values, missing values, feature selection/construction
- We typically need chain together different algorithms
    - Many preprocessing steps
    - Possibly many models
- This is called a *pipeline* (or *workflow*)
- The best way to represent data depends not only on the semantics of the data, but also on the kind of model you are using.
- For instance, linear models can learn better models by using different feature representations.


linear regression
Regression output
Input feature
2
1
0
−1
−2
−3
−2
−1
0
1
2
3

Polynomials

We can also make linear models behave more flexibly by adding polynomials of the original continuous features.

For a given feature x, we might want to consider x^2, x^3, x^4, and so on. In scikit-learn, this is implemented in `PolynomialFeatures` in the preprocessing module

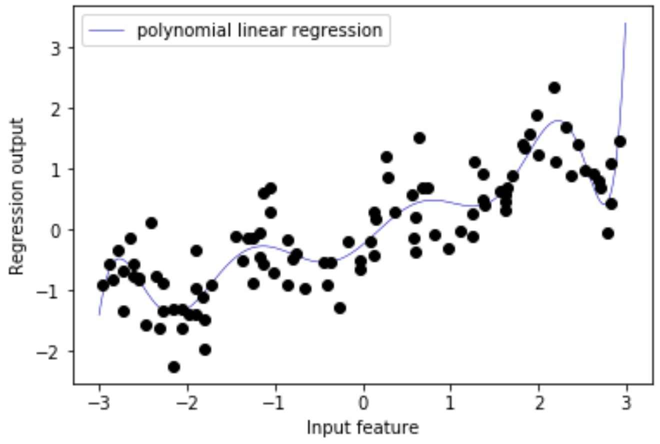
Using a degree of 10 yields 10 features, with the original value raised to the n-th power.

```
X_poly.shape: (100,
10)

Entries of X:
[[-0.753]
 [ 2.704]
 [ 1.392]
 [ 0.592]
 [-2.064]]
Entries of X_poly:
[[    -0.753      0.567     -0.427      0.321     -0.242
0.182
      -0.137      0.103     -0.078      0.058]
 [     2.704      7.313     19.777     53.482    144.632    39
1.125
    1057.714   2860.36    7735.232  20918.278]
 [     1.392      1.938      2.697      3.754      5.226
7.274
      10.125     14.094     19.618     27.307]
 [     0.592      0.35       0.207      0.123      0.073
0.043
       0.025      0.015      0.009      0.005]
 [    -2.064      4.26      -8.791     18.144    -37.448     7
7.289
    -159.516    329.222   -679.478   1402.367]]
```

```
Polynomial feature names:
['x0', 'x0^2', 'x0^3', 'x0^4', 'x0^5', 'x0^6', 'x0^7', 'x0^8', 'x0^9',
'x0^10']
```

Using polynomial features together with a linear regression model yields *polynomial regression*.



Binning (Discretization)

Make linear models more powerful by splitting up a feature into multiple artificial features:

- Partition the feature values into a fixed number of bins
- A data point will then be represented by which bin it falls into.
- Each bin becomes a new dimension

```
bins: [-3.  -2.4 -1.8 -1.2 -0.6  0.   0.6  1.2  1.8  2.4
3. ]
```

Numpy's `digitize` maps each value to its corresponding bin. E.g. the first sample goes to bin nr. 4.

```
Data points:
 [[-0.753]
 [ 2.704]
 [ 1.392]
 [ 0.592]
 [-2.064]]

Bin membership for data poi
nts:
 [[ 4]
 [10]
 [ 8]
 [ 6]
 [ 2]]
```

Encoding data with scikit-learn (`OneHotEncoder`)

- Scikit-learn offers a convenient `OneHotEncoder`
- Call `fit` to compute the internal parameters of the transformation
- Call `transform` to produce the transformed data. You can also run `fit_transform` to do both at once.

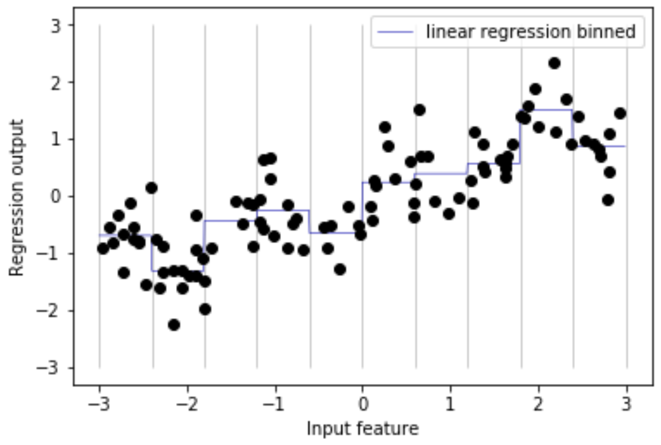
!! Test data should *never* be used to compute the preprocessing, because information about the test data will *leak* into the training data, invalidating your model evaluation. Use a nested loop.

```
[[ 0.  0.  0.  1.  0.  0.  0.  0.  0.
0.]
 [ 0.  0.  0.  0.  0.  0.  0.  0.  0.
1.]
 [ 0.  0.  0.  0.  0.  0.  0.  1.  0.
0.]
 [ 0.  0.  0.  0.  0.  1.  0.  0.  0.
0.]
 [ 0.  1.  0.  0.  0.  0.  0.  0.  0.
0.]]

X_binned.shape: (100,
10)
```

Now we build a new linear regression model and a new decision tree model on the one-hot-encoded data.

```
Out[39]:  Text(0.5,0,'Input feature')
```



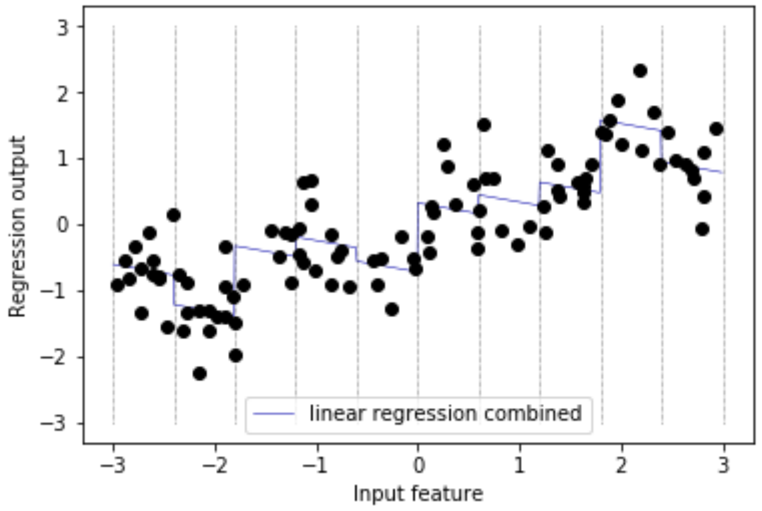
Interaction features

Another way to enrich a feature representation, particularly for linear models, is adding interaction features and polynomial features of the original data.

For instance: our linear model learned a constant value for each bin in the wave dataset. If we want a sloped linear model we need to allow interaction with another feature, e.g., the original feature. So let's add the original feature (the x-axis in the plot) back in. This leads to an 11-dimensional dataset:

```
(100,
11)
```

```
Out[41]:  [<matplotlib.lines.Line2D at 0x1c184a15f
          8>]
```

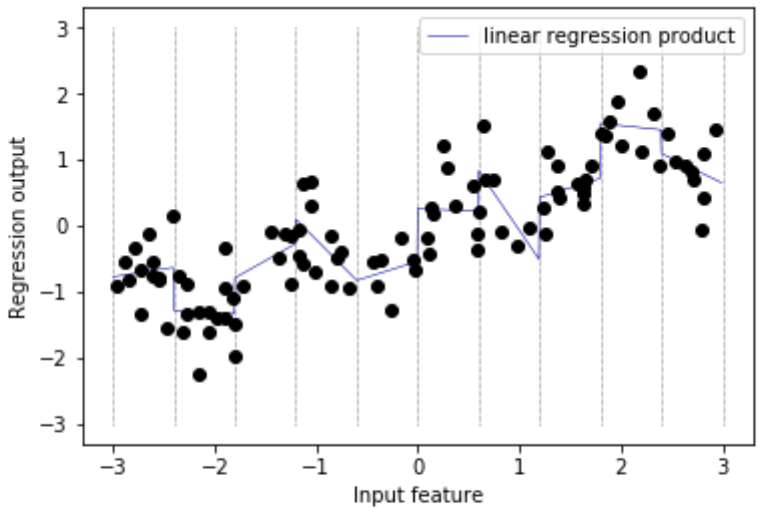


Product features

If we want a different slope per bin, we need a new *interaction feature* (or *product feature*) that indicates in which bin a data point is in **and** where it lies on the x-axis.

```
(100,
20)
```

```
Out[43]:  <matplotlib.legend.Legend at 0x1c186daf
          60>
```



Non-linear transformations

There are other transformations that often prove useful for transforming certain features.

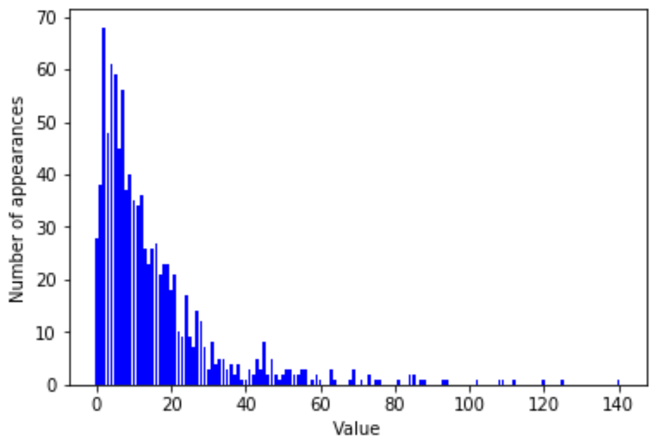
For instance, `log` or `exp` are very useful to better scale your data. This is useful for models that are sensitive to feature scales, such as linear models, SVMs and neural networks.

The functions log and exp can help by adjusting the relative scales, tranforming them to more Gaussian-like value distributions.

Here we generate some data that has a very non-normal (Poisson) distribution:

```
Number of feature appearances:
[28 38 68 48 61 59 45 56 37 40 35 34 36 26 23 26 27 21 23 23 18 21 10
9 17
  9  7 14 12  7  3  8  4  5  5  3  4  2  4  1  1  3  2  5  3  8  2  5
2  1
  2  3  3  2  2  3  3  0  1  2  1  0  0  3  1  0  0  0  1  3  0  1  0
2  0
  1  1  0  0  0  0  1  0  0  2  2  0  1  1  0  0  0  0  1  1  0  0  0
0  0
  0  0  1  0  0  0  0  0  1  1  0  0  1  0  0  0  0  0  0  0  1  0  0
0  0
  1  0  0  0  0  0  0  0  0  0  0  0  0  0  0  1]
```

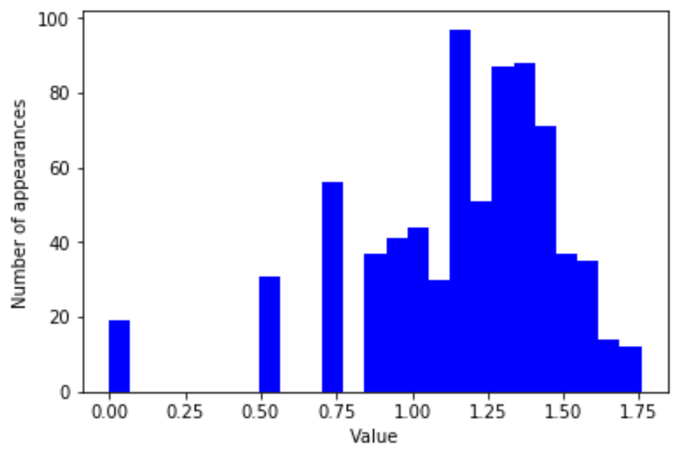
```
Out[46]:  Text(0.5,0,'Value')
```



This is something most linear models can’t handle very well:

```
Test score:
0.622
```

Applying a logarithmic transformation can help to create a more normal (Gaussian) distribution



And our Ridge regressor now performs a lot better.

```
Test score:
0.875
```

Finding the transformation that works best for each combination of dataset and model is somewhat of an art.

Automatic Feature Selection

When adding new features, or with high-dimensional datasets in general, it can be a good idea to reduce the number of features to only the most useful ones, and discard the rest.

- Simpler models that generalize better
- Help algorithms that are sensitive to the number of features (e.g. kNN).

Univariate statistics (ANOVA)

We want to keep the features for which there is statistically significant relationship between it and the target. In the case of classification, this is also known as analysis of variance (ANOVA). These test consider each feature individually (they are univariate), and are completely independent of the model that you might want to apply afterwards. The result will be a p-value for each feature (lower is better).

In scikit-learn":

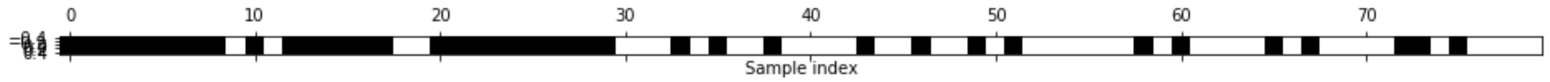
- `SelectKBest` will only keep the *k* features with the lowest p values.
- `SelectPercentile` selects a fixed percentage of features.

To test these methods, we'll take the `breast_cancer` dataset, and add 50 random noise features. The feature selector should be able to remove at least these noise features.

```
X_train.shape: (284, 80)
X_train_selected.shape: (284,
40)
```

We can retrieve which features were selected with `get_support`, and visualize the selected (black) and removed (white) features. `SelectPercentile` removed most of the noise features, but not perfectly.

```
Out[53]:  Text(0.5,0,'Sample index')
```



As usual, we need to check how the transformation affects the performance of our learning algorithms.

```
Score with all features: 0.930
Score with only selected features:
0.940
```

Model-based Feature Selection

Model-based feature selection uses a supervised machine learning model to judge the importance of each feature, and keeps only the most important ones. Compared to ANOVA, they consider all features together, and are thus able to capture interactions: a feature may be more (or less) informative in combination with others.

The supervised model that is used for feature selection doesn't need to be the same model that is used for the final supervised modeling, it only needs to be able to measure the (perceived) importance for each feature:

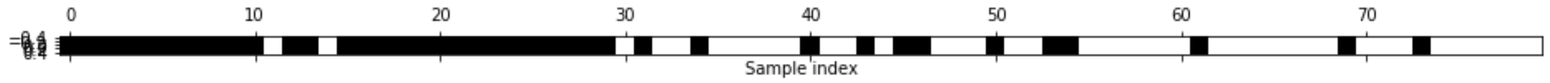
- Decision tree–based models return a `feature_importances_` attribute
- Linear models return coefficients, whose absolute values also reflect feature importance

In scikit-learn, we can do this using `SelectFromModel`. It requires a model and a threshold. Threshold='median' means that the median observed feature importance will be the threshold, which will remove 50% of the features.

We've seen before how RandomForests return good estimates of feature importance:

```
X_train.shape: (284, 8
0)
X_train_l1.shape: (284,
40)
```

```
Out[57]:  Text(0.5,0,'Sample index')
```



All but two of the original features were selected, and most of the noise features removed. Our linear model trained on the selected features also performs quite a bit better.

```
Test score:
0.951
```

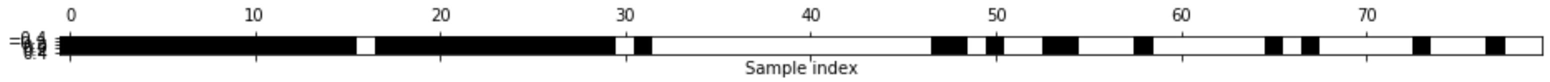
Iterative feature selection

Instead of building a model to remove many features at once, we can also just ask it to remove the worst feature, then retrain, remove another feature, etc. This is known as *recursive feature elimination* (RFE).

Vice versa, we couls also ask it to iteratively add one feature at a time. This is called *forward selection*.

In both cases, we need to define beforehand how many features to select. When this is unknown, one often considers this as an additional hyperparameter of the whole process (pipeline) that needs to be optimized.

```
Out[59]: Text(0.5,0,'Sample index')
```



```
Test score:
0.951
```

```
Test score:
0.951
```

Automatic feature selection can be helpful when:

- You expect some inputs to be uninformative, and your model does not select features internally (as tree-based models do)
- You need to speed up prediction without loosing much accuracy
- You want a more interpretable model (with fewer variables)

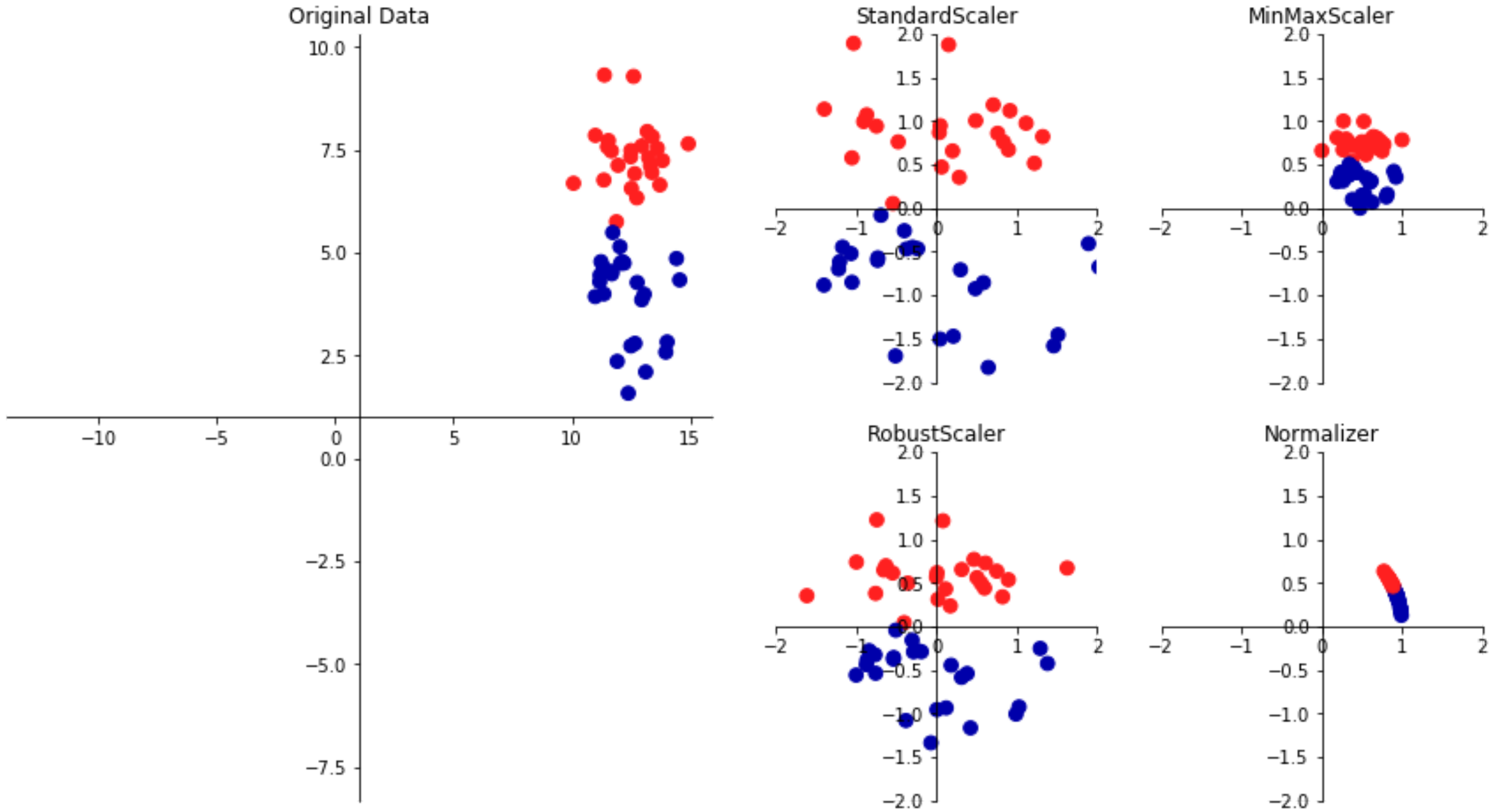
Scaling

When the features have different scales (their values range between very different minimum and maximum values), it makes sense to scale them to the same range. Otherwise, one feature will overpower the others, expecially when raised to the *n*th power.

- We can rescale features between 0 and 1 using MinMaxScaler.
- Remember to `fit_transform` the training data, then `transform` the test data

Several scaling techniques are available:

- `StandardScaler` rescales all features to mean=0 and variance=1
  - Does not ensure and min/max value
- `RobustScaler` uses the median and quartiles
  - Median m: half of the values < m, half > m
  - Lower Quartile lq: 1/4 of values < lq
  - Upper Quartile uq: 1/4 of values > uq
  - Ignores *outliers*, brings all features to same scale
- `MinMaxScaler` brings all feature values between 0 and 1
- `Normalizer` scales data such that the feature vector has Euclidean length 1
  - Projects data to the unit circle
  - Used when only the direction/angle of the data matters


Original Data
StandardScaler
MinMaxScaler
RobustScaler
Normalizer

Applying scaling transformations

- Lets apply a scaling transformation *manually*, then use it to train a learning algorithm
- First, split the data in training and test set

- Next, we `fit` the preprocessor on the **training data**
  - This computes the necessary transformation parameters
  - For `MinMaxScaler`, these are the min/max values for every feature

```
Out[65]:  MinMaxScaler(copy=True, feature_range=(0,
          1))
```

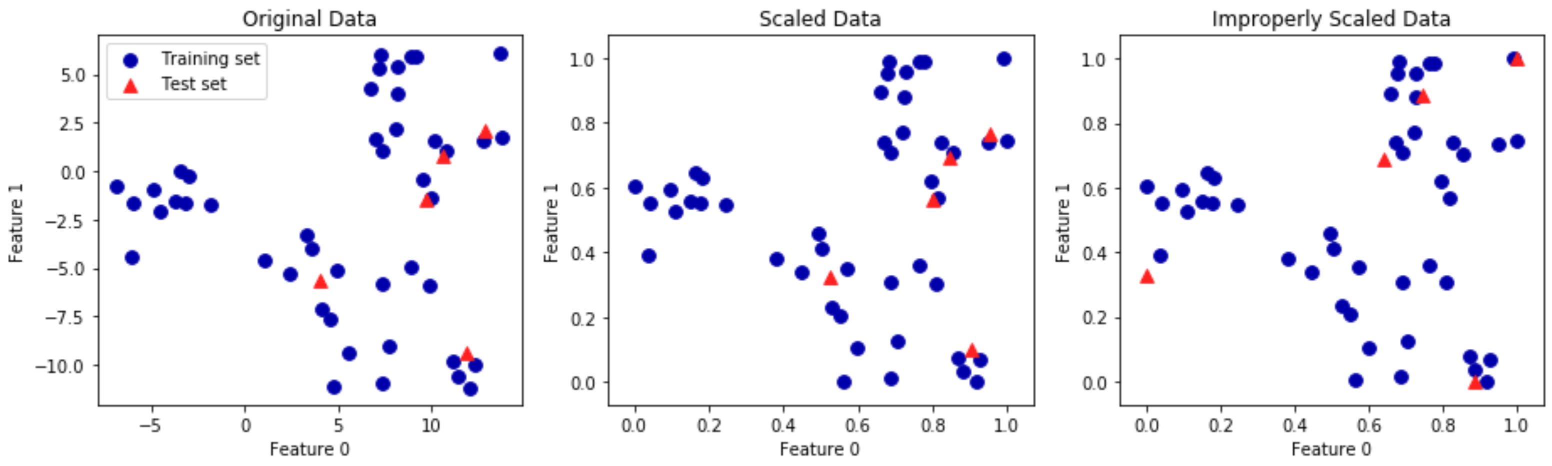
- After fitting, we can `transform` the training and test data

```
per-feature minimum before scaling:
 [   6.981    9.71    43.79   143.5      0.053    0.019    0.
0.
    0.106    0.05     0.115    0.36     0.757    6.802    0.002    0.
002
    0.       0.       0.01     0.001    7.93    12.02    50.41   185.
2
    0.071    0.027    0.       0.       0.157    0.055]
per-feature maximum before scaling:
 [   28.11     39.28    188.5    2501.        0.163     0.287     0.4
27
     0.201     0.304     0.096     2.873     4.885    21.98    542.2
     0.031     0.135     0.396     0.053     0.061     0.03      36.04
    49.54    251.2    4254.         0.223     0.938     1.17      0.29
1
     0.577     0.149]
per-feature minimum after scaling:
 [ 0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.
  0.
  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.]
per-feature maximum after scaling:
 [ 1.  1.  1.  1.  1.  1.  1.  1.  1.  1.  1.  1.  1.  1.  1.  1.  1.
  1.
  1.  1.  1.  1.  1.  1.  1.  1.  1.  1.  1.  1.]
```

```
per-feature minimum after scaling:
[ 0.034  0.023  0.031  0.011  0.141  0.044  0.     0.     0.154 -0.
006
 -0.001  0.006  0.004  0.001  0.039  0.011  0.     0.    -0.032  0.
007
  0.027  0.058  0.02   0.009  0.109  0.026  0.     0.    -0.    -0.
002]
per-feature maximum after scaling:
[ 0.958  0.815  0.956  0.894  0.811  1.22   0.88   0.933  0.932  1.
037
  0.427  0.498  0.441  0.284  0.487  0.739  0.767  0.629  1.337  0.
391
  0.896  0.793  0.849  0.745  0.915  1.132  1.07   0.924  1.205  1.
631]
```

- After scaling the test data, the values are not exactly between 0 and 1
- This is correct: we used the min/max values from the training data only
- We are still interested in how well our preprocessing+learning model generalizes from the training to the test data

- Remember to `fit` and `transform` on the training data, then `transform` the test data
- 2nd figure: `fit` on training set, `transform` on training and test set
- 3rd figure: `fit` and `transform` on the training data
  - Test data points nowhere near same training data points
  - Trained model will have a hard time generalizing correctly



- Note: you can fit and transform the training together with `fit_transform`
- To transform the test data, you always need to `fit` on the training data and `transform` the test data

How great is the effect of scaling?

- First, we train the (linear) SVM without scaling

```
Test set accuracy:
0.94
```

- With scaling, we get a much better model

```
Scaled test set accuracy:
0.97
```

Scaling for polynomial regression

After scaling, we extract polynomial features and interactions up to a degree of 2. Note how we `fit` the `PolynomialFeatures` only on the training data and then apply it (`transform`) on both the training and test data.

`PolynomialFeatures` will add a new features for each possible interaction (product) of up to 2 input features, including the products of a feature with itself (the squares). Hence, $\frac{13!}{11!2!}$ + 13 + 13 features total.

```
X_train.shape: (379, 13)
X_train_poly.shape: (379,
105)
```

The exact correspondence between input and output features can be found using the get_feature_names method:

```
Polynomial feature names:
['1', 'x0', 'x1', 'x2', 'x3', 'x4', 'x5', 'x6', 'x7', 'x8', 'x9', 'x1
0', 'x11', 'x12', 'x0^2', 'x0 x1', 'x0 x2', 'x0 x3', 'x0 x4', 'x0 x5',
'x0 x6', 'x0 x7', 'x0 x8', 'x0 x9', 'x0 x10', 'x0 x11', 'x0 x12', 'x1^
2', 'x1 x2', 'x1 x3', 'x1 x4', 'x1 x5', 'x1 x6', 'x1 x7', 'x1 x8', 'x1
x9', 'x1 x10', 'x1 x11', 'x1 x12', 'x2^2', 'x2 x3', 'x2 x4', 'x2 x5',
'x2 x6', 'x2 x7', 'x2 x8', 'x2 x9', 'x2 x10', 'x2 x11', 'x2 x12', 'x3^
2', 'x3 x4', 'x3 x5', 'x3 x6', 'x3 x7', 'x3 x8', 'x3 x9', 'x3 x10', 'x3
x11', 'x3 x12', 'x4^2', 'x4 x5', 'x4 x6', 'x4 x7', 'x4 x8', 'x4 x9', 'x
4 x10', 'x4 x11', 'x4 x12', 'x5^2', 'x5 x6', 'x5 x7', 'x5 x8', 'x5 x9',
'x5 x10', 'x5 x11', 'x5 x12', 'x6^2', 'x6 x7', 'x6 x8', 'x6 x9', 'x6 x1
0', 'x6 x11', 'x6 x12', 'x7^2', 'x7 x8', 'x7 x9', 'x7 x10', 'x7 x11',
'x7 x12', 'x8^2', 'x8 x9', 'x8 x10', 'x8 x11', 'x8 x12', 'x9^2', 'x9 x1
0', 'x9 x11', 'x9 x12', 'x10^2', 'x10 x11', 'x10 x12', 'x11^2', 'x11 x1
2', 'x12^2']
```

Let’s compare the performance of a linear model (Ridge regression) on the data with and without interactions:

```
Score without interactions:
0.621
Score with interactions: 0.75
3
```

Clearly, the interactions and polynomial features gave us a good boost in performance when using Ridge. When using a more complex model like a random forest, the story is a bit different, though:

```
Score without interactions:
0.796
Score with interactions: 0.76
5
```

The random forest does not benefit from the interaction features, in fact, performance decreases...

Adding polynomials is typically good for linear models, but not a cure-for-all. Always evaluate the performance of models when adding preprocessing steps.

- What if we want the cross-validated evaluation?
    - Apply scaling on every fold independently?

Building Pipelines

- In scikit-learn, a `pipeline` combines multiple processing *steps* in a single estimator
- All but the last step should be transformer (have a `transform` method)
  - The last step can be a transformer too (e.g. Scaler+PCA)
- It has a `fit`, `predict`, and `score` method, just like any other learning algorithm
- Pipelines are built as a list of steps, which are (name, algorithm) tuples
  - The name can be anything you want, but can't contain `'__'`
  - We use `'__'` to refer to the hyperparameters, e.g. `svm__C`
- Let's build, train, and score a `MinMaxScaler` + `LinearSVC` pipeline:

```
Test score:
0.97
```

- Now with cross-validation:

```
Cross-validation scores: [ 0.984  0.953  0.
979]
Average cross-validation score: 0.97
```

- We can retrieve the trained SVM by querying the right step indices

```
SVM component: LinearSVC(C=1.0, class_weight=None, dual=True, fit_inter
cept=True,
     intercept_scaling=1, loss='squared_hinge', max_iter=1000,
     multi_class='ovr', penalty='l2', random_state=None, tol=0.0001,
     verbose=0)
```

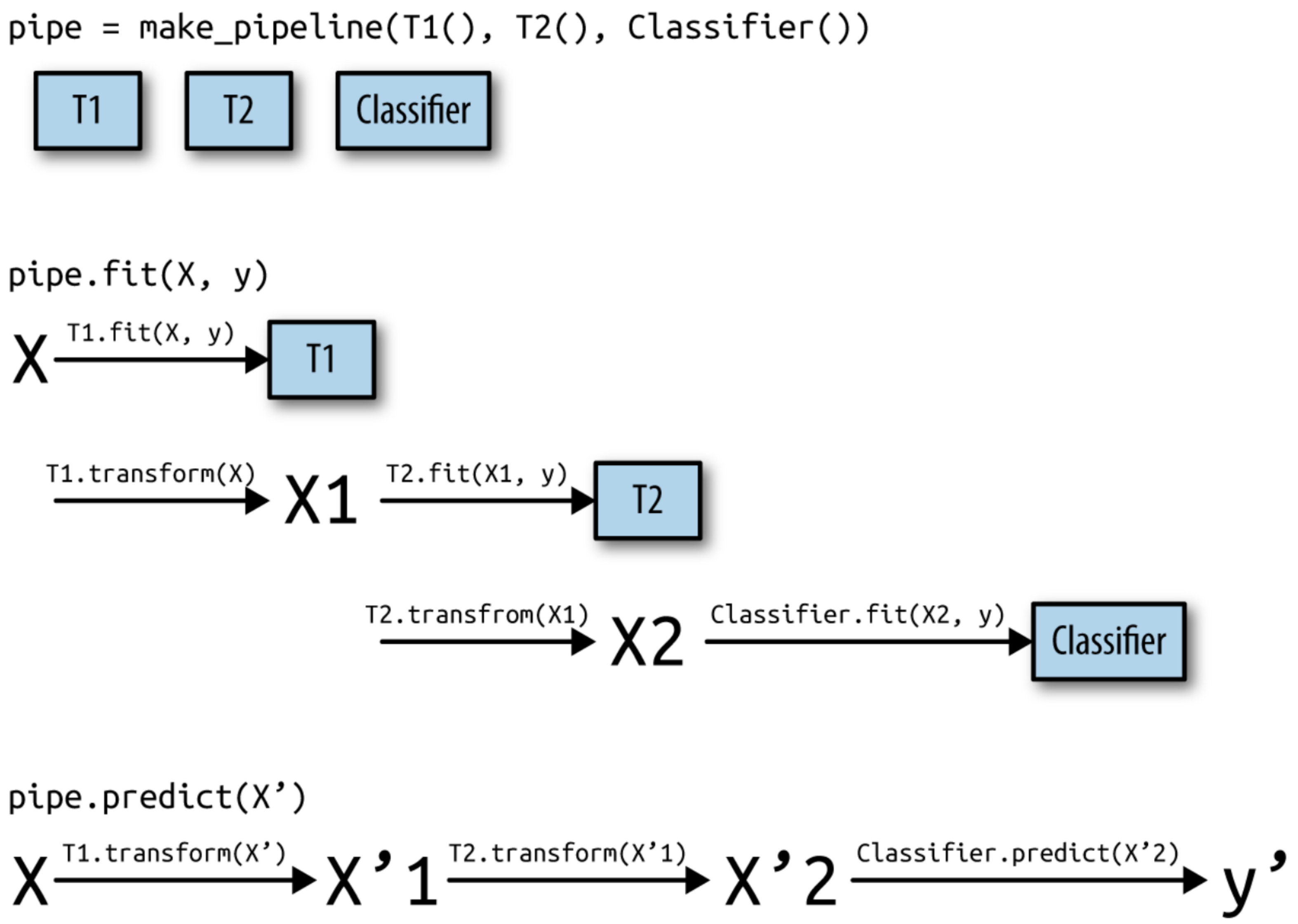
- Or we can use the `named_steps` dictionary

```
SVM component: LinearSVC(C=1.0, class_weight=None, dual=True, fit_inter
cept=True,
     intercept_scaling=1, loss='squared_hinge', max_iter=1000,
     multi_class='ovr', penalty='l2', random_state=None, tol=0.0001,
     verbose=0)
```

- When you don't need specific names for specific steps, you can use `make_pipeline`
    - Assigns names to steps automatically

```
Pipeline steps:
[('minmaxscaler', MinMaxScaler(copy=True, feature_range=(0, 1))), ('lin
earsvc', LinearSVC(C=100, class_weight=None, dual=True, fit_intercept=T
rue,
     intercept_scaling=1, loss='squared_hinge', max_iter=1000,
     multi_class='ovr', penalty='l2', random_state=None, tol=0.0001,
     verbose=0))]
```

Visualization of a pipeline `fit` and `predict`


pipe = make_pipeline(T1(), T2(), Classifier())
T1
T2
Classifier
pipe.fit(X, y)
X
T1.fit(X, y)
T1
T1.transform(X)
X1
T2.fit(X1, y)
T2
T2.transfrom(X1)
X2
Classifier.fit(X2, y)
Classifier
pipe.predict(X’)
X
T1.transform(X’)
X’1
T2.transform(X’1)
X’2
Classifier.predict(X’2)
y’