Feature Engineering and pipelines

Feature engineering

- Many of the algorithms that we've seen are greatly affected by *how* you represent the training data
- Examples: 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.

Overview

- Feature generation
  - **Polynomial features**: turn linear regression into polynomial regression
  - **Binning**: split a feature into multiple dimensions
  - **Product features**: model interactions between features
  - **Non-linear transformations**: log, sin,...
- Feature selection
  - ANOVA
  - Model-based feature selection
  - Forward/backward selection
- Scaling
- Missing value imputation
- Pipelines
- Practical advice

Polynomials

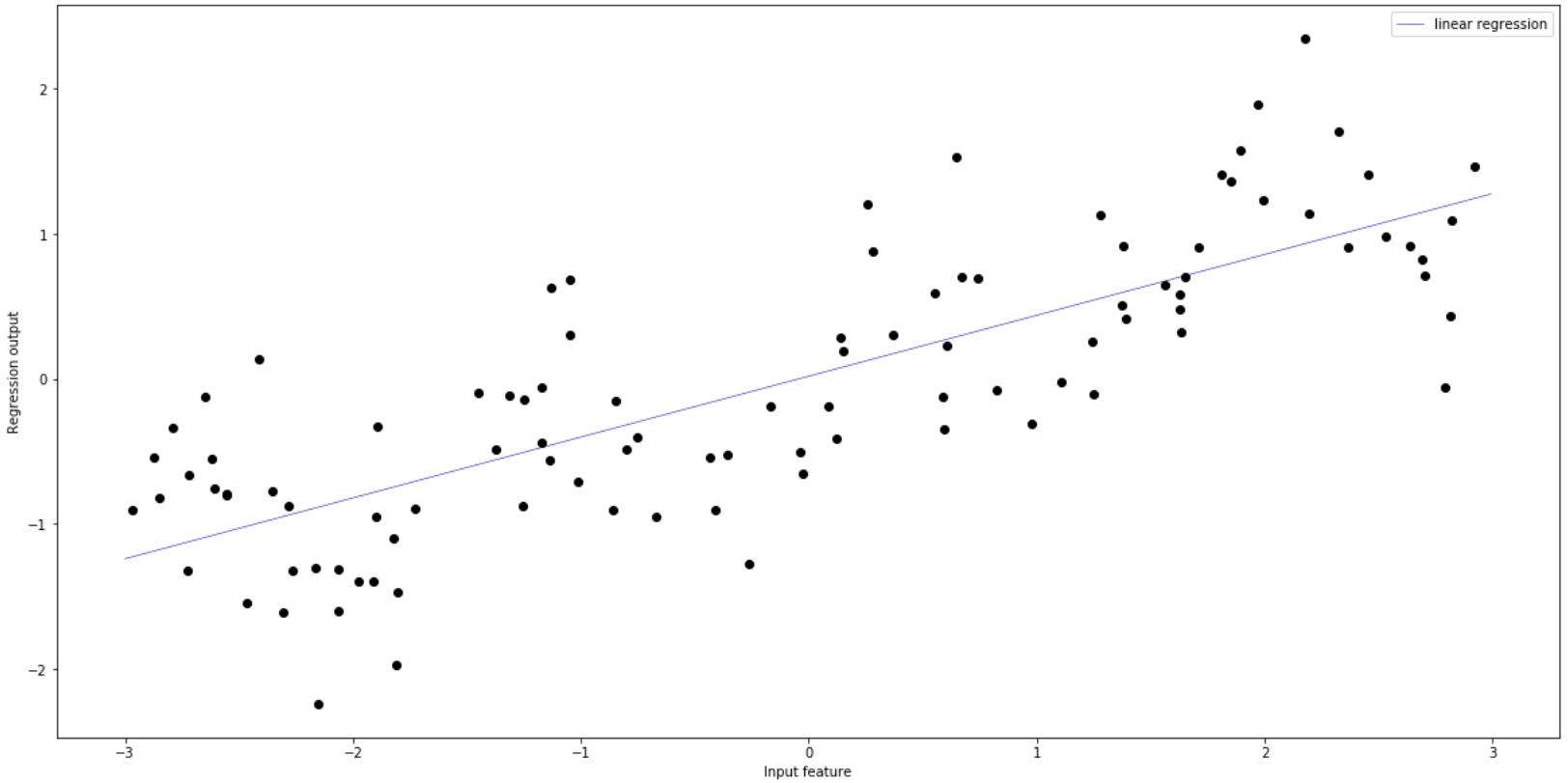
We can make linear models behave more flexibly by adding polynomials of the original continuous features.

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

```
poly = PolynomialFeatures(degree=10, include_bias=False)
X_poly = poly.fit(X).transform(X)
```

- `degree` is the max. degree of the polynomilas, requires some searching
- `include_bias` adds the 0-th polynomial, i.e. x^0

Linear regression can't capture the 'wave' in this data



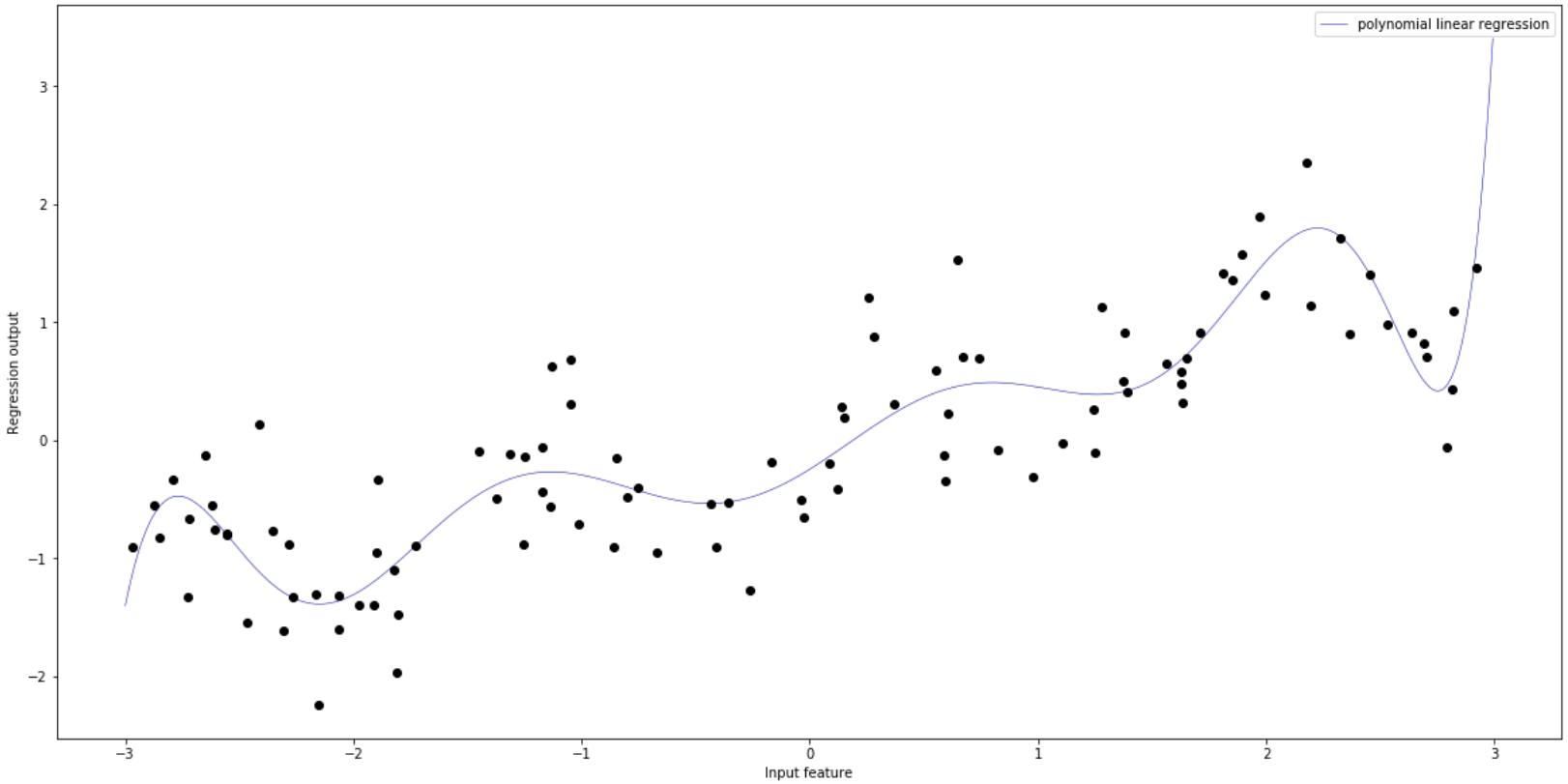
Using a degree of 10 yields 10 features, with the original value raised to the n-th power.

Out[4]:

	x0	x0^2	x0^3	x0^4	x0^5	x0^6	x0^7	x0^8	x0^9	x0^10
0	-0.75	0.57	-0.43	0.32	-0.24	0.18	-0.14	0.1	-0.078	0.058
1	2.7	7.3	20	53	1.4e+02	3.9e+02	1.1e+03	2.9e+03	7.7e+03	2.1e+04
2	1.4	1.9	2.7	3.8	5.2	7.3	10	14	20	27
3	0.59	0.35	0.21	0.12	0.073	0.043	0.025	0.015	0.0089	0.0053
4	-2.1	4.3	-8.8	18	-37	77	-1.6e+02	3.3e+02	-6.8e+02	1.4e+03

modelling polynomial features with linear regression yields *polynomial regression*.

- The model is still a hyperplane (in a 10-dimensional space)
- The predictions are now a non-linear function of the original variable X
- Fits the wave shape a lot better


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

Binning (Discretization)

Split a numeric feature into multiple artificial features:

- Partition the (numeric) feature values into *n* equal intervals (bins)
- Replace each feature with *n* features
  - Value is 1 if the original feature value falls in the corresponding bin
  - Value is 0 otherwise
- Train the model on those *n* features

- No handy sklearn function. You need to use:
  - `numpy.digitize` to assign values to bins
  - `OneHotEncoder` to convert a feature of n numbers to n binary features

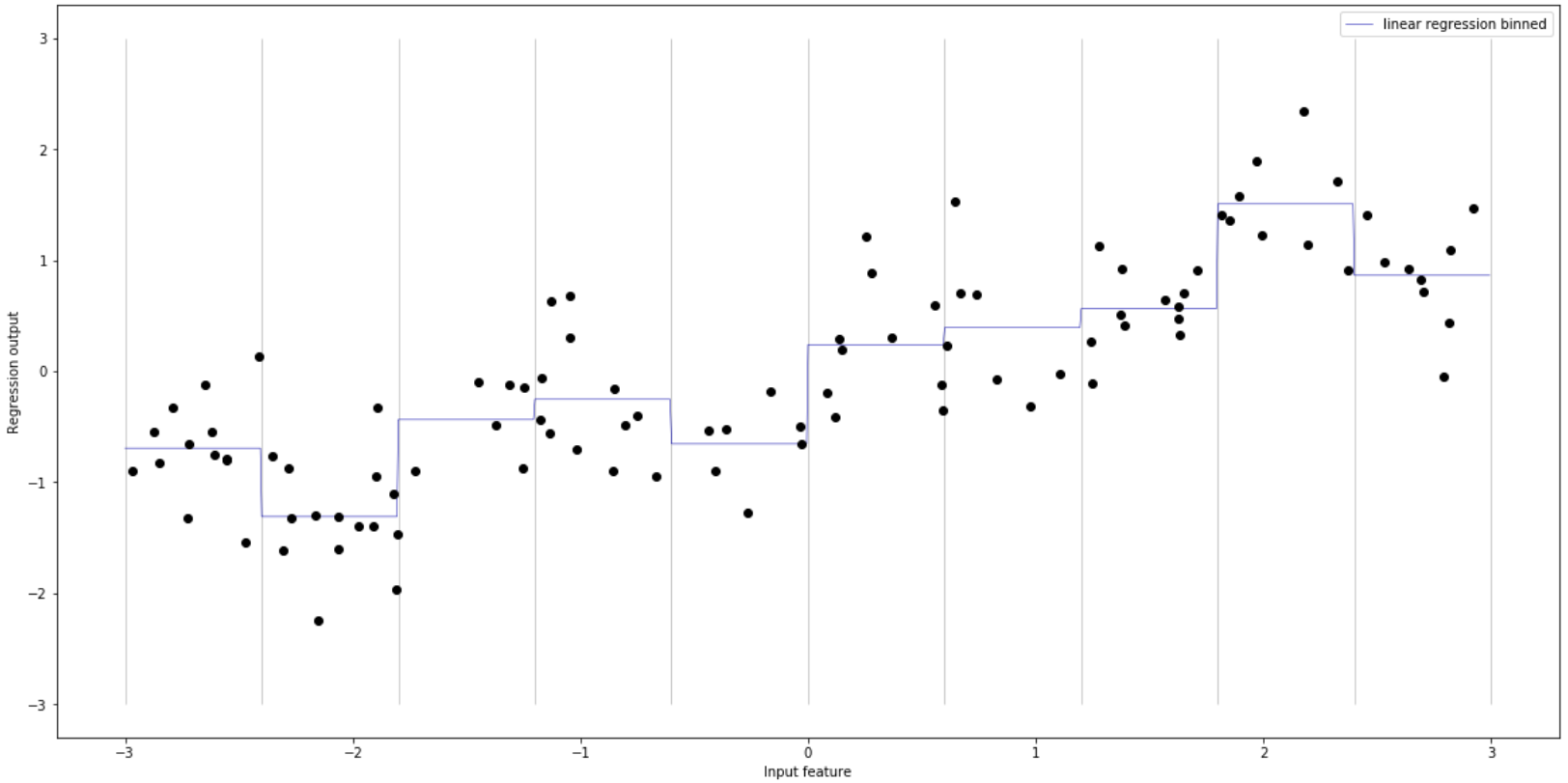
```
which_bin = numpy.digitize(X, bins=np.linspace(-3, 3, 1
1))
X_binned = OneHotEncoder(sparse=False).fit_transform(wh
ich_bin)
```

	orig
0	-0.75
1	2.7
2	1.4
3	0.59
4	-2.1

	which_bin
0	4
1	10
2	8
3	6
4	2

	[-3.0,-2.4]	[-2.4,-1.8]	[-1.8,-1.2]	[-1.2,-0.6]	[-0.6,0.0]	[0.0,0.6]	[0.6,1.2]	[1.2,1.8]	[1.8,2.4]	[2.4,3.0]
0	0	0	0	1	0	0	0	0	0	0
1	0	0	0	0	0	0	0	0	0	1
2	0	0	0	0	0	0	0	1	0	0
3	0	0	0	0	0	1	0	0	0	0
4	0	1	0	0	0	0	0	0	0	0

Training a linear regression model on the one-hot-encoded data:



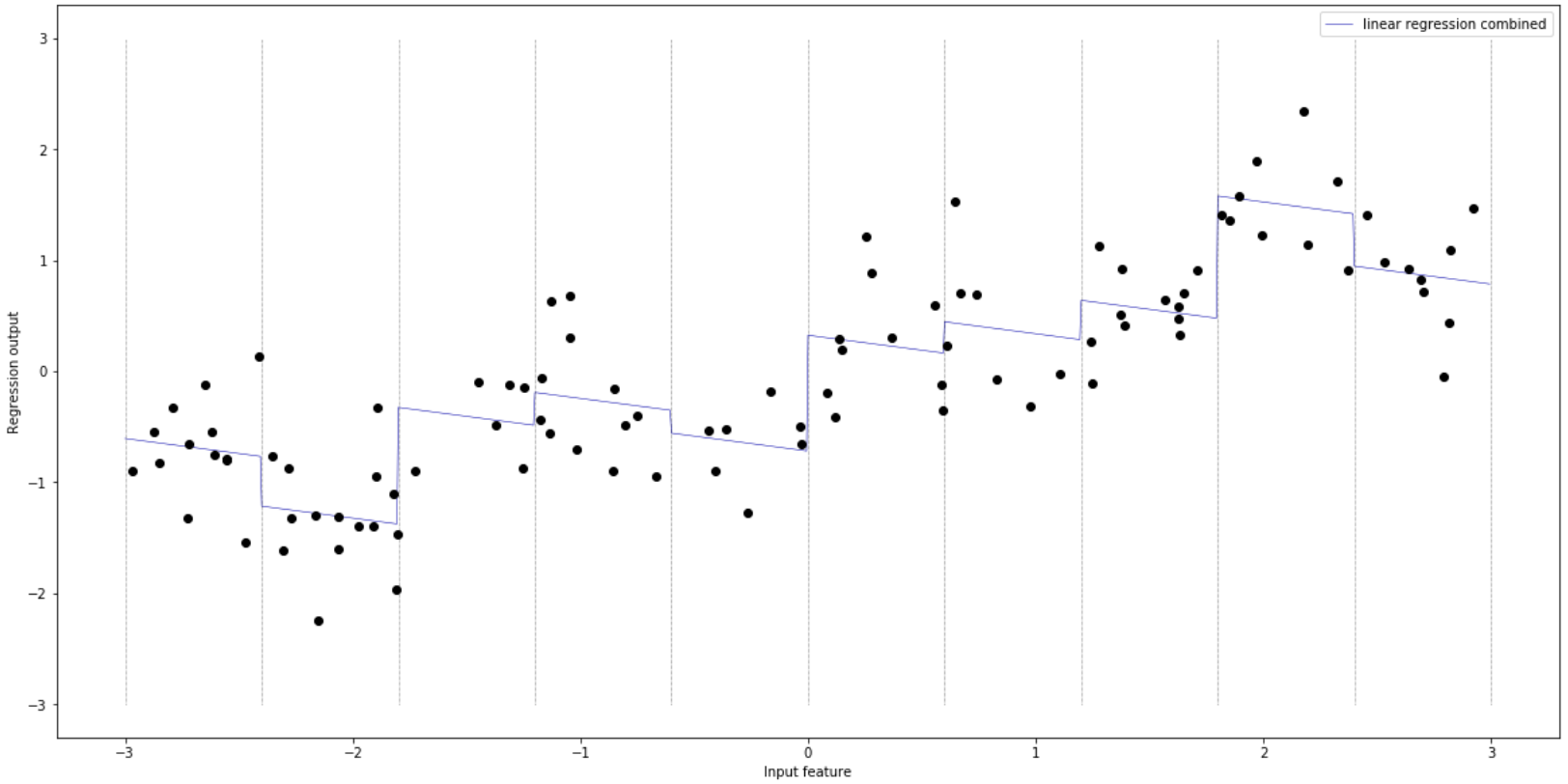
Interaction features

- our linear model learned a constant value for each bin in the wave dataset
- allow interaction with the original feature by simply adding it to the data

Out[9]:

	orig	[-3.0,-2.4]	[-2.4,-1.8]	[-1.8,-1.2]	[-1.2,-0.6]	[-0.6,0.0]	[0.0,0.6]	[0.6,1.2]	[1.2,1.8]	[1.8,2.4]	[2.4,3.0]
0	-0.75	0	0	0	1	0	0	0	0	0	0
1	2.7	0	0	0	0	0	0	0	0	0	1
2	1.4	0	0	0	0	0	0	0	1	0	0
3	0.59	0	0	0	0	0	1	0	0	0	0
4	-2.1	0	1	0	0	0	0	0	0	0	0

The interaction allows the linear model to learn the slope (in function of the original feature)



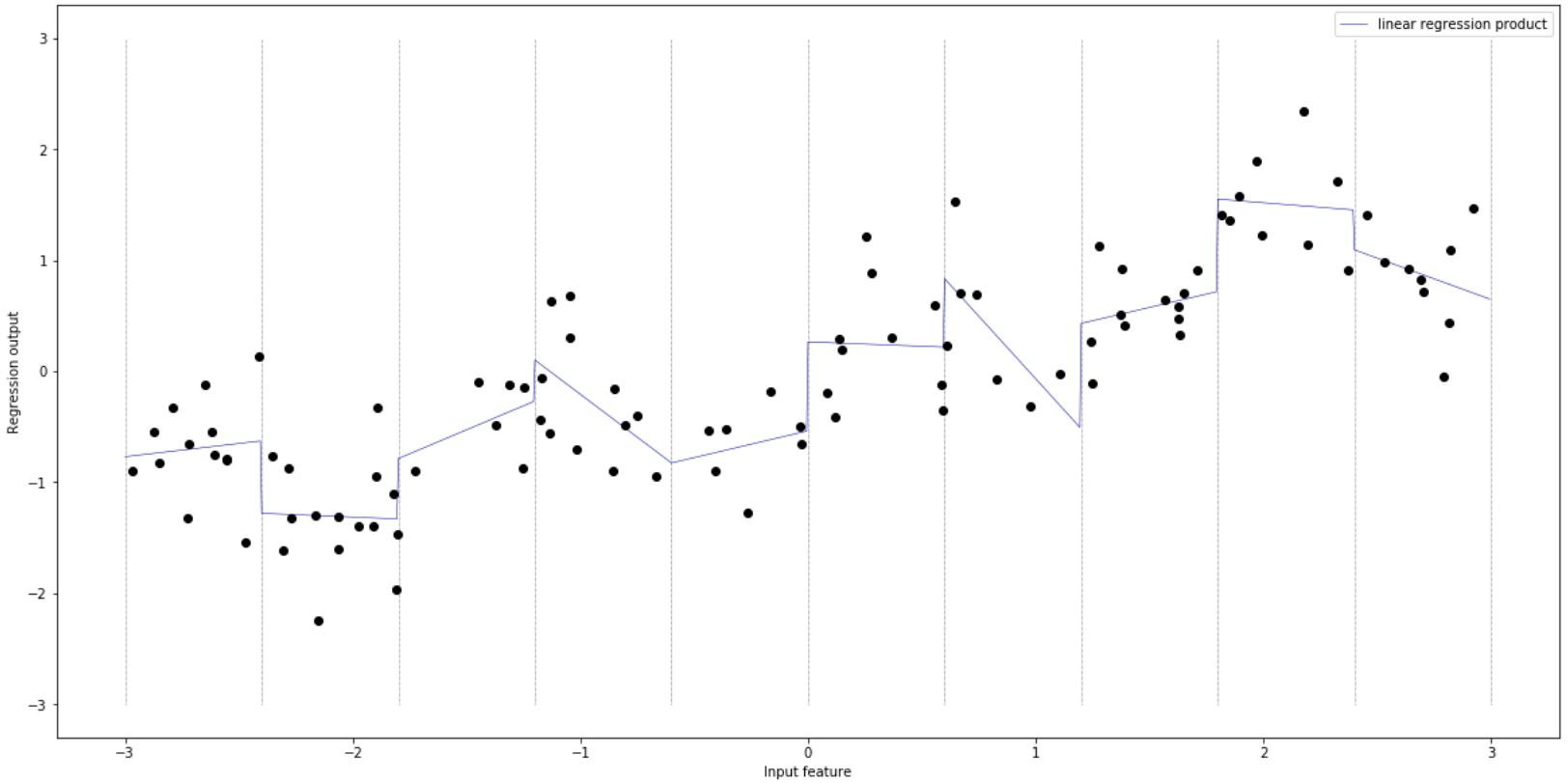
Interaction features

- 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.
- This requires adding the product of the original feature with every discretized feature

Out[11]:

	b0	b1	b2	b3	b4	b5	b6	b7	b8	b9	X*b0	X*b1	X*b2	X*b3	X*b4	X*b5	X*b6	X*b7	X*b8	X*b9
0	0	0	0	1	0	0	0	0	0	0	-0	-0	-0	-0.75	-0	-0	-0	-0	-0	-0
1	0	0	0	0	0	0	0	0	0	1	0	0	0	0	0	0	0	0	0	2.7
2	0	0	0	0	0	0	0	1	0	0	0	0	0	0	0	0	0	1.4	0	0
3	0	0	0	0	0	1	0	0	0	0	0	0	0	0	0	0.59	0	0	0	0
4	0	1	0	0	0	0	0	0	0	0	-0	-2.1	-0	-0	-0	-0	-0	-0	-0	-0

The interaction features allow the linear model to learn a slope per bin



Non-linear transformations

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

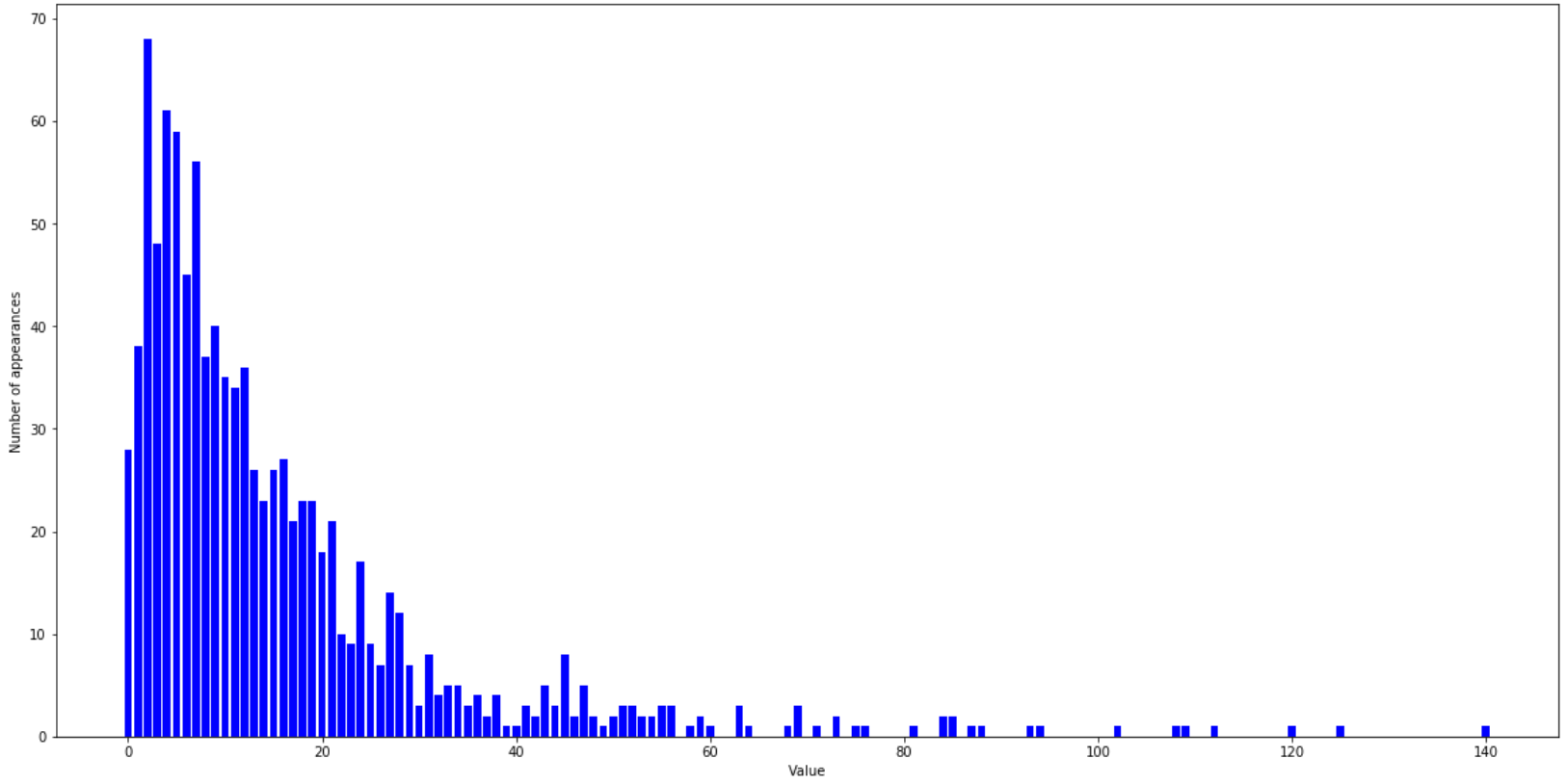
For instance, `log` or `exp` are very useful to better scale your data. This is useful for models that are sensitive to feature scales, such as linear models, SVMs and neural networks.

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

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

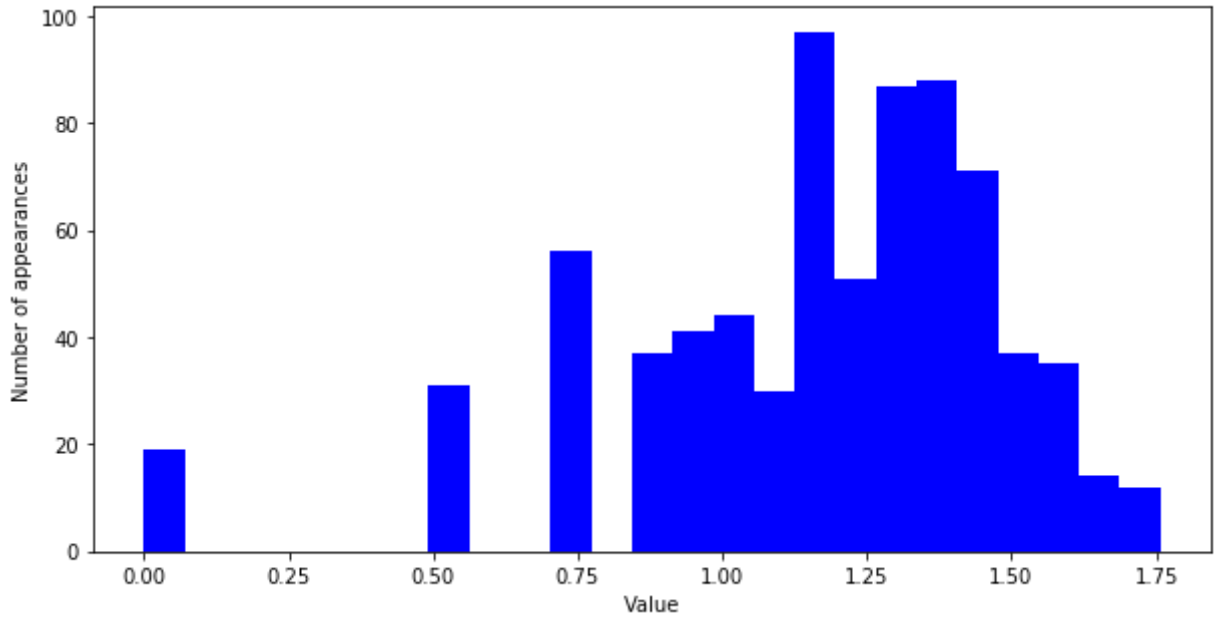
Example: Poisson distributions

Imagine a process where many outcomes are rare (only occur 1 or 2 times) while other occur hundreds of times



Applying a logarithmic transformation can help to create a more normal distribution

```
X_train_log = np.log(X_train + 1)
X_test_log = np.log(X_test + 1)
```



Most linear models can't handle non-normal distributions well:

```
score = Ridge().fit(X_train, y_train).score(X_test, y_test)
```

```
Ridge regression (original data) test score (R2): 0.622
```

But do a lot better if the data is transformed:

```
score = Ridge().fit(X_train_log, y_train).score(X_test_log, y_test)
```

```
Ridge regression (transformed data) test score (R2): 0.875
```

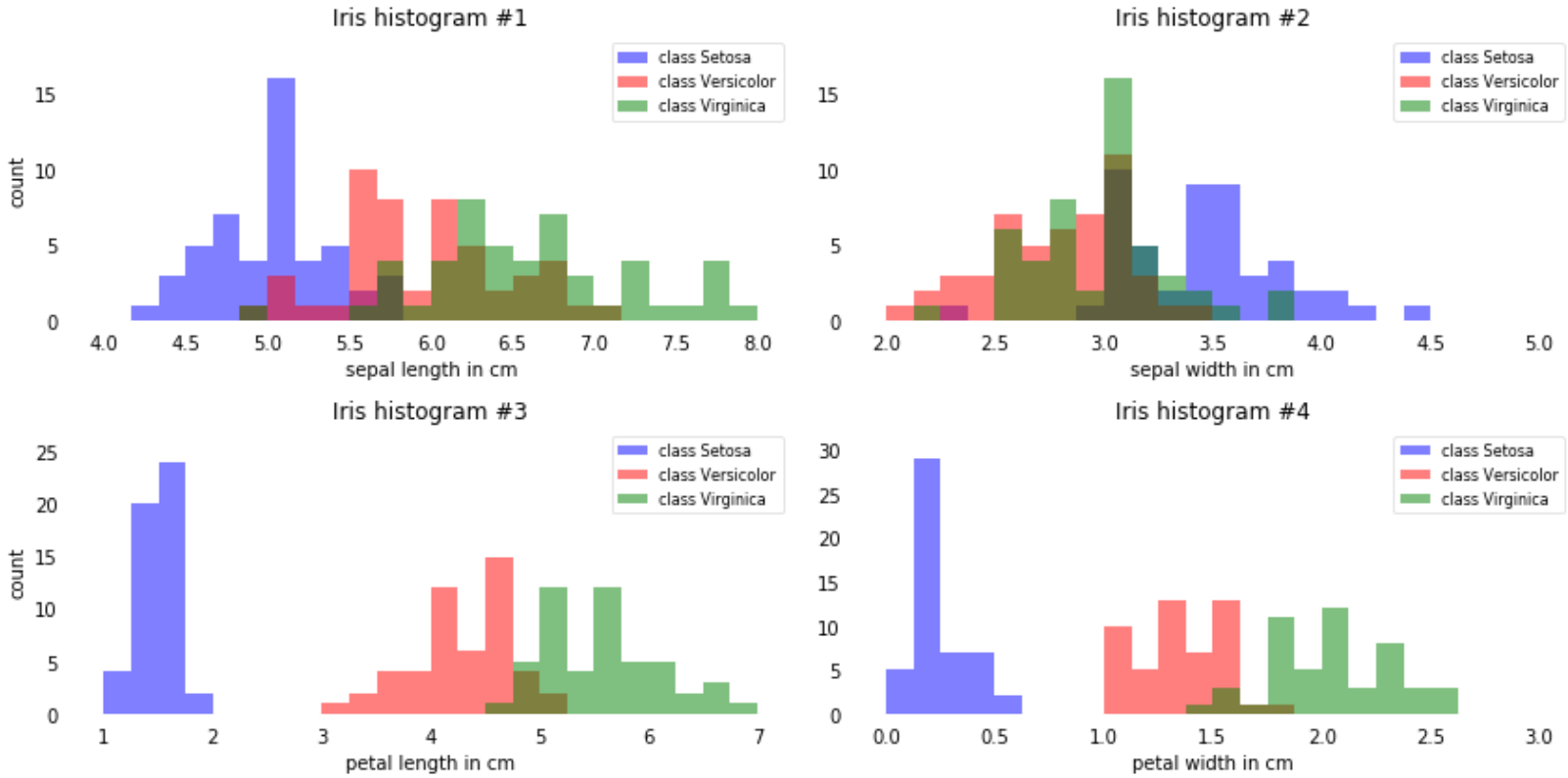
Automatic Feature Selection

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).

Example: Iris

Below are the distributions (histograms) of every class according to every feature.
Which of the four features is most informative?



Univariate statistics

We want to keep the features for which there is statistically significant relationship between it and the target. These test consider each feature individually (they are univariate), and are completely independent of the model that you might want to apply afterwards.

In scikit-learn we have two options:

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

Retrieve the selected features with `get_support()`

We can use different tests to measure how informative a feature is:

`f_regression`: For numeric targets. Measures the performance of a linear regression model trained on only one feature.

`f_classif`: For categorical targets. Measures the *F-statistic* from one-way Analysis of Variance (ANOVA), or the proportion of total variance explained by one feature.

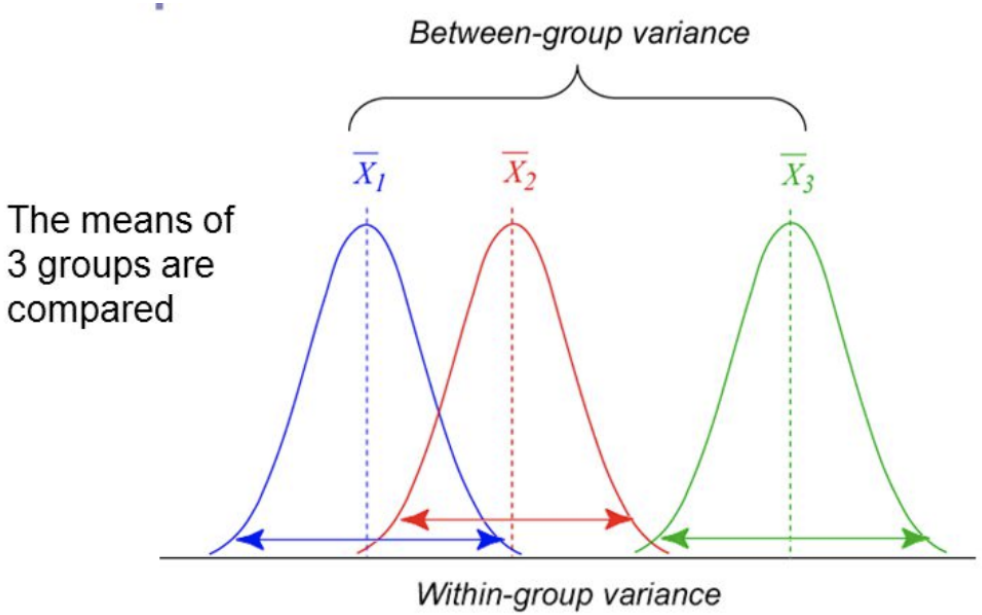
`chi2`: For categorical features and targets. Performs the chi-square statistic. Similar results as F-statistic, but less sensitive to nonlinear relationships.

For both the F-statistic and χ^2, we actually obtain the p-value under the F- and χ^2 distribution, respectively.

F-statistic = variation between sample means / mean variation within the samples (higher is better)

X_i: all samples with class i.

Better is samples means are far apart and variation within samples is small.



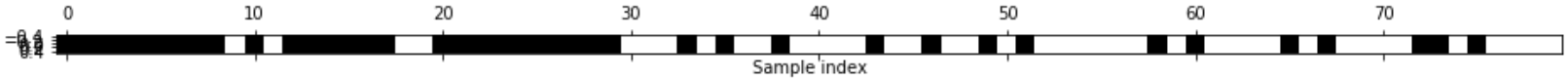
Chi-squared for a feature with c categories and k classes:

$$\chi^2 = \sum_{i=0}^{c} \sum_{j=0}^{k} \frac{(O_{ij} - E_{ij})^2}{E_{ij}}$$

where O_{ij} is the number of observations of feature category i and class j, and E_{ij} is the expected number of observations of category i and class j if there was no relationship between the feature and the target (number of samples of category i * ratio of class j).

Comparison

- To compare feature selection methods, we take a classification dataset with 30 real features, and add 50 random noise features.
- Ideally, the feature selection removes at least the last 50 noise features.
- Selected features in black, removed features in white
- Results for `SelectPercentile` with `f_classif` (ANOVA):
  - OK, but fails to remove several noise features



Impact on performance: check how the transformation affects the performance of our learning algorithms.

```
LogisticRegression score with all features: 0.930
LogisticRegression 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. They consider all features together, and are thus able to capture interactions: a feature may be more (or less) informative in combination with others.

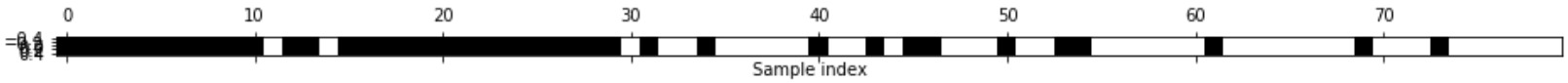
The supervised model that is used for feature selection doesn't need to be the same model that is used for the final supervised modeling, it only needs to be able to measure the (perceived) importance for each feature:

- 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.

```
select = SelectFromModel(
    RandomForestClassifier(n_estimators=100, random_state=42),
    threshold="median")
```

- RandomForests (see later) is known to good estimates of feature importance
- All but two of the original features were selected, and most of the noise features removed.
- Logistic regression mdoel improves further




```
LogisticRegression 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).

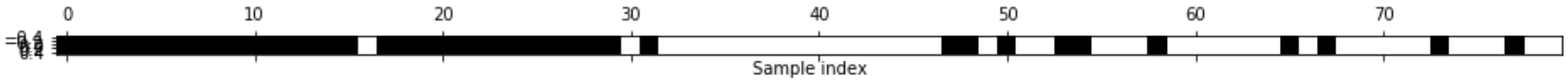
```
select = RFE(RandomForestClassifier(n_estimators=100, random_state=42),
             n_features_to_select=40)
```

Vice versa, we could 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.

RFE result:

- Fewer noise features, only 1 original feature removed
- LogisticRegression performance about the same




```
LogisticRegression 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)

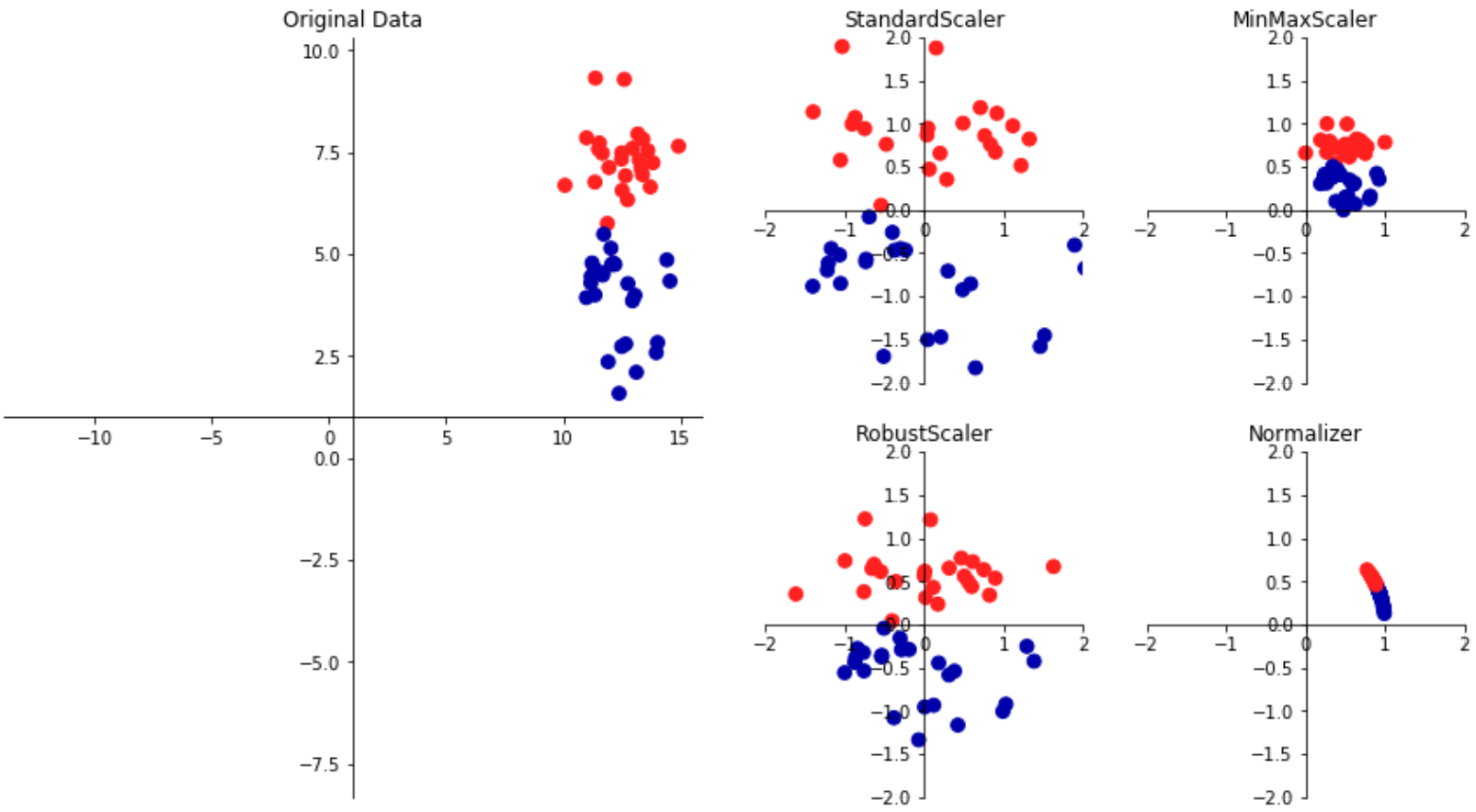
Scaling

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

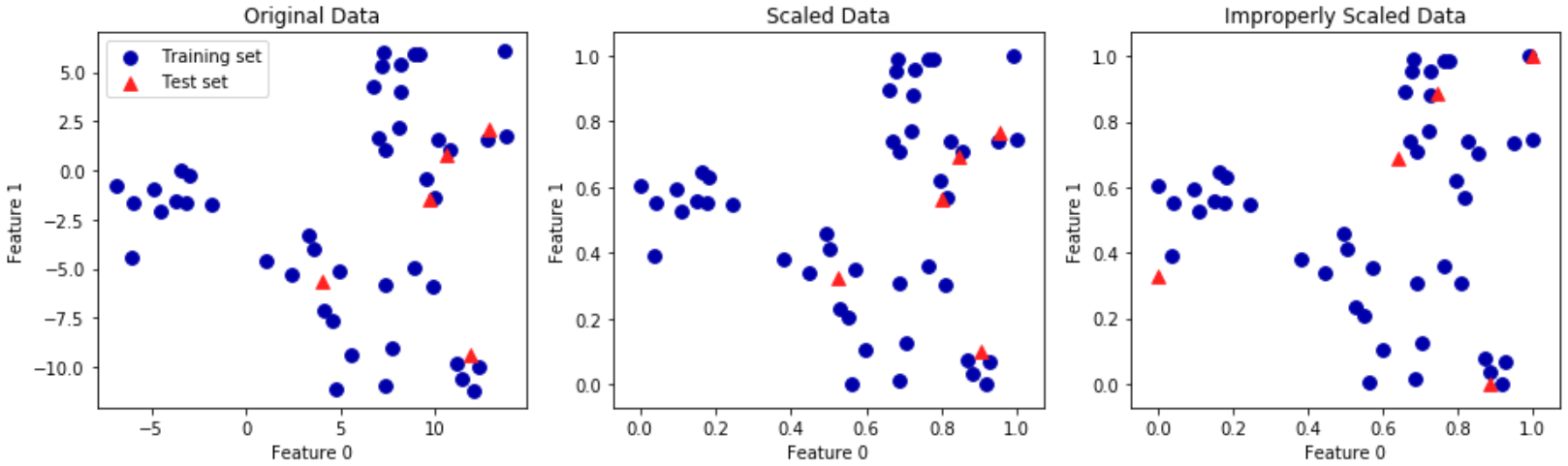
Out[28]: MinMaxScaler(copy=True, feature_range=(0, 1))

```
Training data 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.]
Training data 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.]
```

- 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

```
Test data 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]
Test data 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]
```

- 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

```
scaler = StandardScaler()
# calling fit and transform in sequence (using method chaining)
X_scaled = scaler.fit(X).transform(X)
# same result, but more efficient computation
X_scaled_d = scaler.fit_transform(X)
```

How great is the effect of scaling?

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

```
LinearSVC test set accuracy: 0.87
```

- With scaling, we get a much better model

```
LinearSVM (with scaling) test set accuracy: 0.97
```

Missing value imputation

- Many sci-kit learn algorithms cannot handle missing value
- `Imputer` replaces specific values
  - `missing_values` (default 'NaN') placeholder for the missing value
  - `strategy`:
    - `mean`, replace using the mean along the axis
    - `median`, replace using the median along the axis
    - `most_frequent`, replace using the most frequent value
- Many more advanced techniques exist, but not yet in scikit-learn
  - e.g. low rank approximations (uses matrix factorization)

```
imp = Imputer(missing_values='NaN', strategy='mean', axis=0)
imp.fit_transform(X1_train)
```

```
Missing data:
 [[ 1.  2.]
 [nan  3.]
 [ 7. nan]]
Imputed data:
 [[1.  2. ]
 [4.  3. ]
 [7.  2.5]]
```

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:

```
pipe = Pipeline([("scaler", MinMaxScaler()), ("svm", LinearSVC())])
pipe.fit(X_train, y_train).score(X_test, y_test)
```

```
Test score: 0.97
```

- Now with cross-validation:

```
scores = cross_val_score(pipe, cancer.data, cancer.target)
```

```
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

```
pipe.steps[1][1]
```

```
SVM component: LinearSVC(C=1.0, class_weight=None, dual=True, fit_intercep
t=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

```
pipe.named_steps['svm']
```

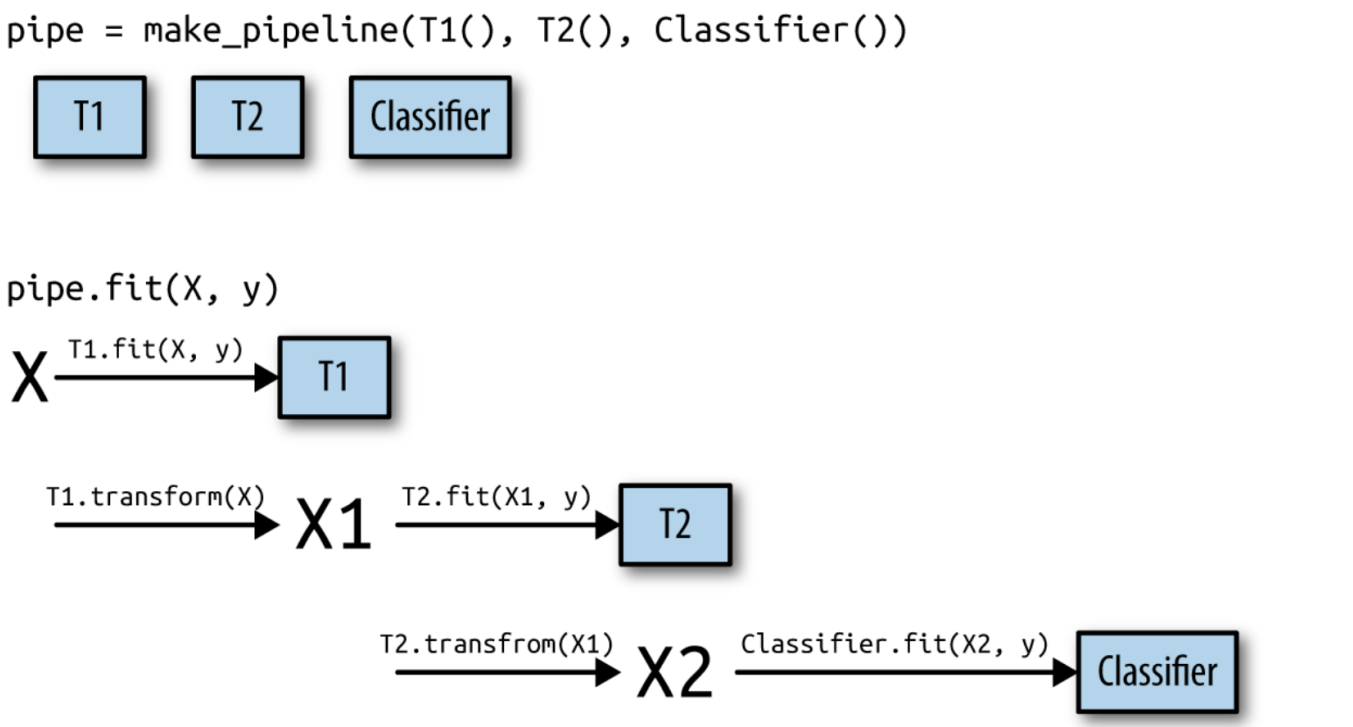
```
SVM component: LinearSVC(C=1.0, class_weight=None, dual=True, fit_intercep
t=True,
     intercept_scaling=1, loss='squared_hinge', max_iter=1000,
     multi_class='ovr', penalty='l2', random_state=None, tol=0.0001,
     verbose=0)
```

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

```
pipe_short = make_pipeline(MinMaxScaler(), LinearSVC(C=
100))
print("Pipeline steps:\n{}".format(pipe_short.steps))
```

```
Pipeline steps:
[('minmaxscaler', MinMaxScaler(copy=True, feature_range=(0, 1))), ('linear
svc', LinearSVC(C=100, class_weight=None, dual=True, fit_intercept=True,
     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`



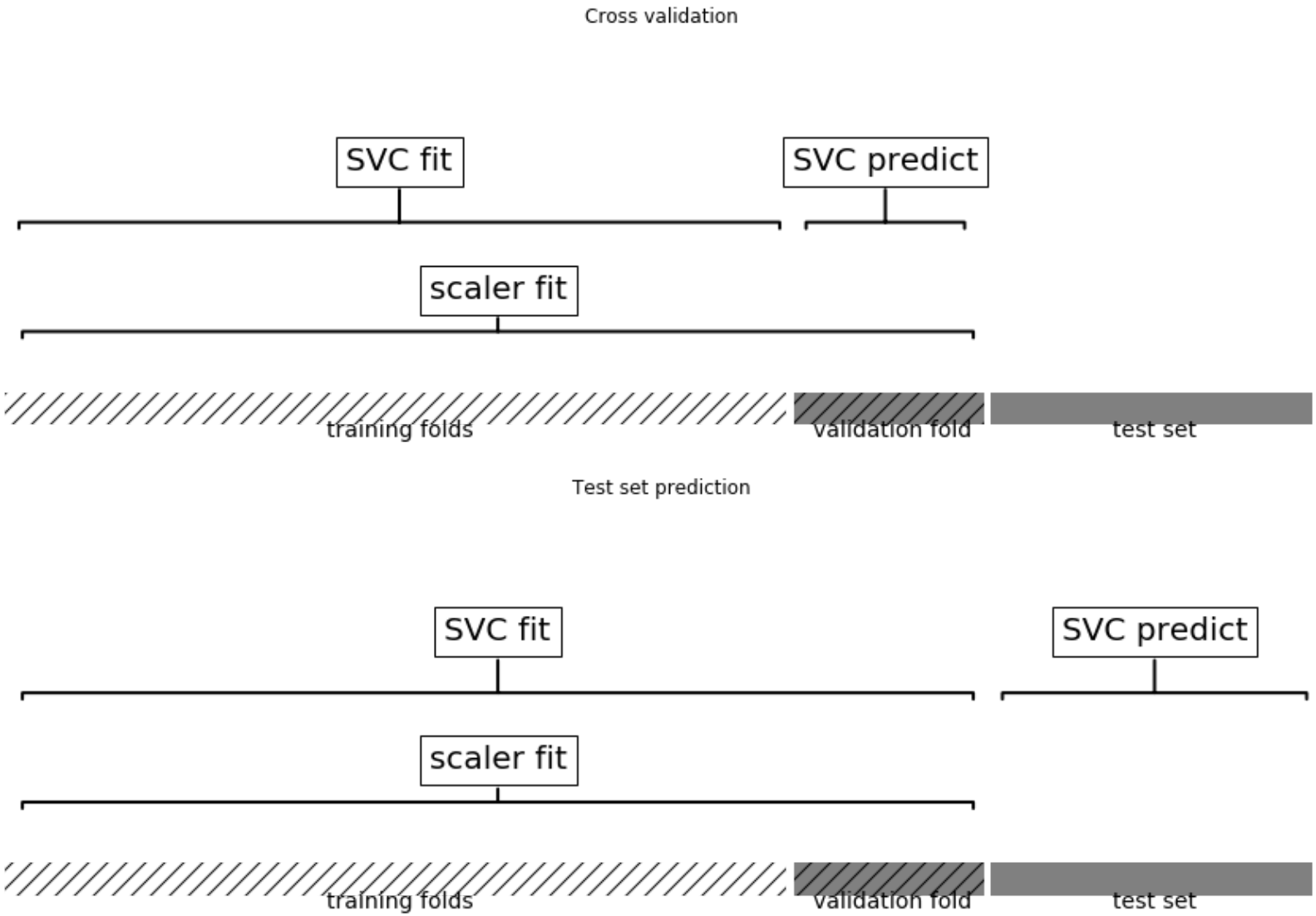
Hyperparameter Selection with Preprocessing

- If we also want to optimize our hyperparameters, things get more complicated
- Indeed, when we `fit` the preprocessor (`MinMaxScaler`), we used *all* the training data.
- The cross-validation splits in GridSearchCV will have training sets preprocessed with information from the test sets (data leakage)

```
Best cross-validation accuracy: 0.98
Best set score: 0.97
Best parameters:  {'C': 1, 'gamma': 1}
```

Visualization of what happens in this code

- During cross-validation (grid search) we evaluate hyperparameter settings on a validation set that was preprocessed with information in that validation set
- This will lead to overly optimistic results during cross-validation
- When we want to use the optimized hyperparameters on the held-out test data, the selected hyperparameters may be suboptimal.
- To solve this, we need to *glue* the preprocessing and learning algorithms together by building a pipeline

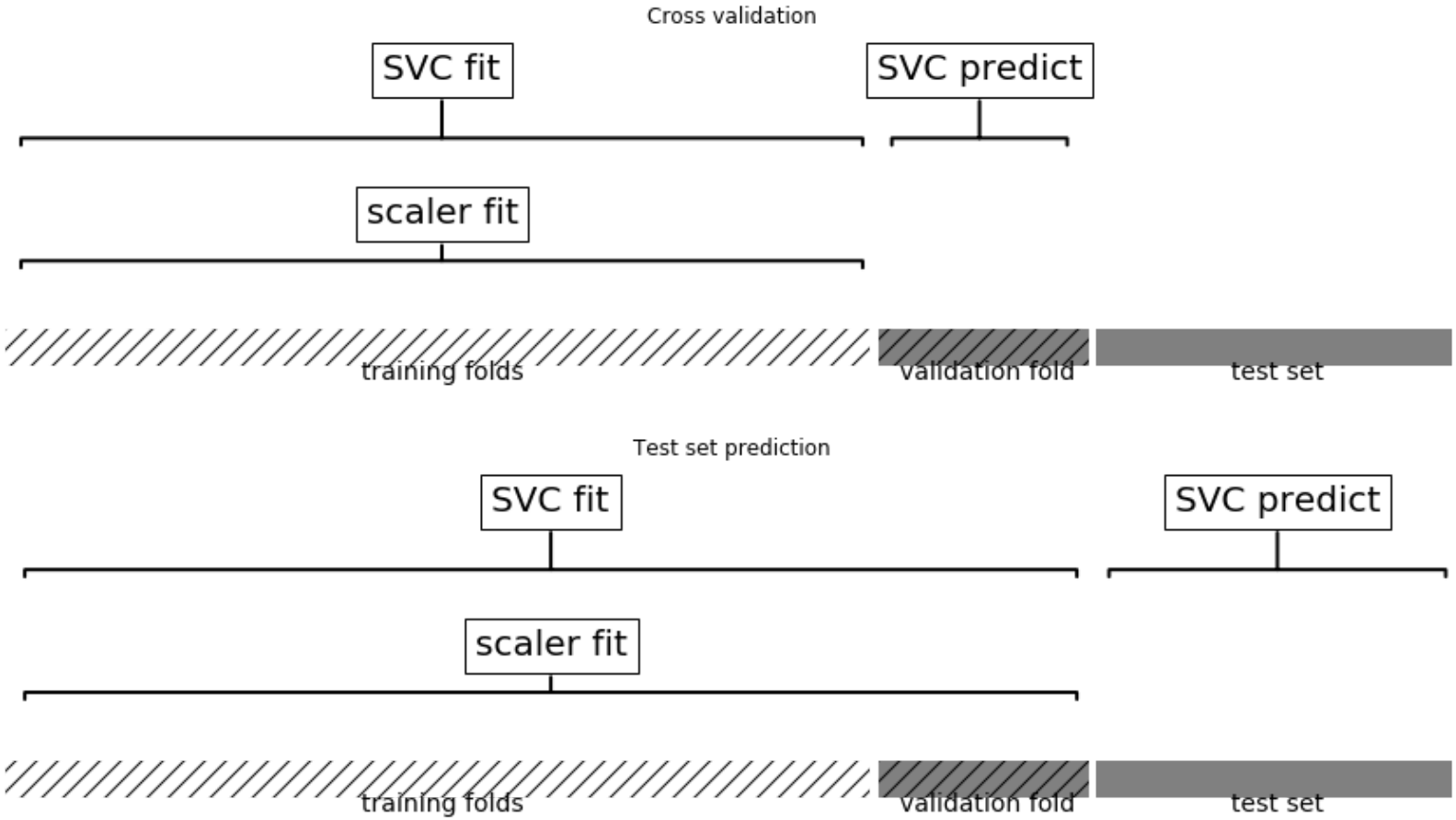

Cross validation
SVC fit
SVC predict
scaler fit
training folds
validation fold
test set
Test set prediction
SVC fit
SVC predict
scaler fit
training folds
validation fold
test set

Using Pipelines in Grid-searches

- We can use the pipeline as a single estimator in `cross_val_score` or `GridSearchCV`
- To define a grid, refer to the hyperparameters of the steps
  - Step `svm`, parameter `C` becomes `svm__C`

```
Best cross-validation accuracy: 0.98
Test set score: 0.97
Best parameters: {'svm__C': 1, 'svm__gamma': 1}
```

- Now, the preprocessors are refit with only the training data in each cross-validation split.



- When we request the best estimator of the grid search, we'll get the best pipeline

```
grid.best_estimator_
```

```
Best estimator:
Pipeline(memory=None,
     steps=[('scaler', MinMaxScaler(copy=True, feature_range=(0, 1))), ('s
vm', SVC(C=1, cache_size=200, class_weight=None, coef0=0.0,
  decision_function_shape='ovr', degree=3, gamma=1, kernel='rbf',
  max_iter=-1, probability=False, random_state=None, shrinking=True,
  tol=0.001, verbose=False))])
```

- And we can drill down to individual components and their properties

```
grid.best_estimator_.named_steps["svm"]
```

```
SVM step:
SVC(C=1, cache_size=200, class_weight=None, coef0=0.0,
  decision_function_shape='ovr', degree=3, gamma=1, kernel='rbf',
  max_iter=-1, probability=False, random_state=None, shrinking=True,
  tol=0.001, verbose=False)
```

Grid-searching preprocessing steps and model parameters

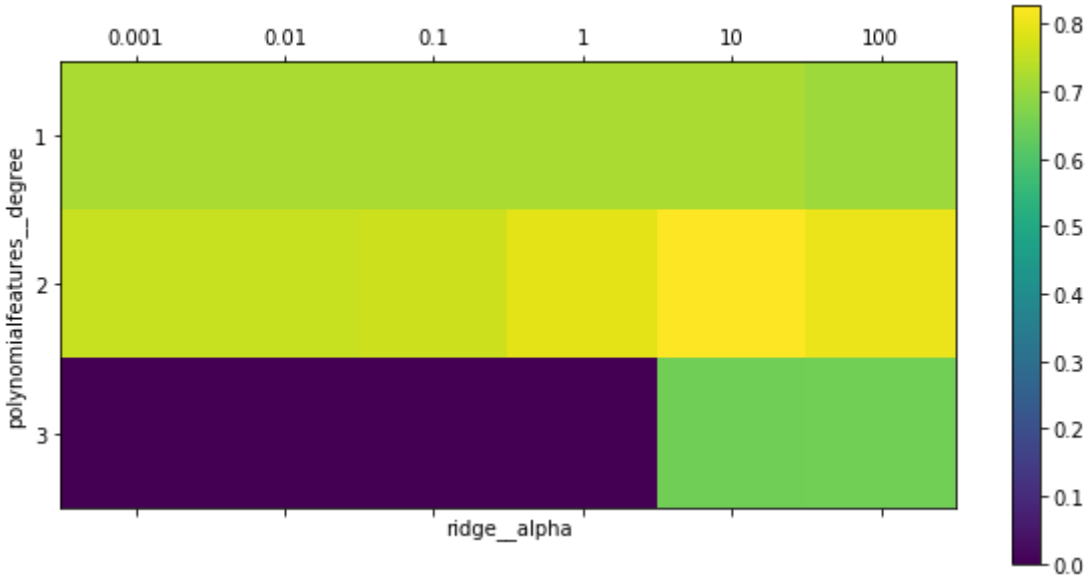
- We can use grid search to optimize the hyperparameters of our preprocessing steps and learning algorithms at the same time
- Consider the following pipeline:
  - `StandardScaler`, without hyperparameters
  - `PolynomialFeatures`, with the max. *degree* of polynomials
  - `Ridge` regression, with L2 regularization parameter *alpha*

- We don't know the optimal polynomial degree or alpha value, so we use a grid search (or random search) to find the optimal values

```
param_grid = {'polynomialfeatures__degree': [1, 2, 3],
              'ridge__alpha': [0.001, 0.01, 0.1, 1, 10, 100]}
grid = GridSearchCV(pipe, param_grid=param_grid, cv=5, n_jobs=1)
grid.fit(X_train, y_train)
```

Out[56]:
```
GridSearchCV(cv=5, error_score='raise-deprecating',
       estimator=Pipeline(memory=None,
     steps=[('standardscaler', StandardScaler(copy=True, with_mean=True,
with_std=True)), ('polynomialfeatures', PolynomialFeatures(degree=2, incl
ude_bias=True, interaction_only=False)), ('ridge', Ridge(alpha=1.0, copy_
X=True, fit_intercept=True, max_iter=None,
   normalize=False, random_state=None, solver='auto', tol=0.001))]),
       fit_params=None, iid='warn', n_jobs=1,
       param_grid={'polynomialfeatures__degree': [1, 2, 3], 'ridge__alph
a': [0.001, 0.01, 0.1, 1, 10, 100]},
       pre_dispatch='2*n_jobs', refit=True, return_train_score='warn',
       scoring=None, verbose=0)
```

- Visualing the R^2 results as a heatmap:



- Here, degree-2 polynomials help (but degree-3 ones don't), and tuning the alpha parameter helps as well.
- Not using the polynomial features leads to suboptimal results (see the results for degree 1)

```
Best parameters: {'polynomialfeatures__degree': 2, 'ridge__alpha': 10}
Test-set score: 0.77
```

Approaching machine learning problems

- Running your favourite algorithm on every problem is usually not a great way to start
- Consider the problem at large
  - Do you want exploratory analysis or (black box) modelling?
  - How to define and measure success? Are there costs involved?
  - Do you have the right data? How can you make it better?
- Build prototypes early-on to evaluate the above.
- Analyse your model's mistakes
  - Should you collect more, or additional data?
  - Should the task be reformulated?
  - Often a higher payoff than endless grid searching
- Technical debt
  - Very complex machine learning prototypes are hard/impossible to put into practice
  - There is a creation-maintenance trade-off
  - See 'Machine Learning: The High Interest Credit Card of Technical Debt'

Real world evaluations

- Accuracy is seldomly the right measure. Usually, costs are involved.
- Don't just evaluate your predictions themselves, also evaluate how the outcome improves *after* you take actions based on them
- Beware of non-representative samples. You often don't have the data you really need.
- Adversarial situations (e.g. spam filtering) can subvert your predictions
- Data leakage: the signal your model found was just an artifact of your data
    - See 'Why Should I Trust You?' by Marco Ribeiro et al.
- A/B testing to evaluate algorithms in the wild
    - More advanced: bandit algorithms

Summary

- Transforming the features can drastically impact performance
  - Scaling is important for many distance-based methods (e.g. kNN, SVM)
  - Constructing new features makes linear models more powerful
- Pipelines allow us to encapsulate multiple steps into a single estimator
  - Has `fit`, `transform`, and `predict` methods
- Avoids data leakage, hence crucial for proper evaluation
- Choosing the right combination of feature extraction, preprocessing, and models is somewhat of an art.
- Pipelines + Grid/Random Search help, but search space is huge
  - Smarter techniques are being researched (see later)
- Real world applications require careful thought, prototyping, and tireless evaluation.