Feature Engineering

Handling imperfect data

Joaquin Vanschoren, Eindhoven University of Technology

Feature engineering

- In reality, data won't be as nicely represented as we've seen thus far
- Many algorithms are greatly affected by *how* data is represented
- 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.
    - E.g. some models handle a large amount of features better than others

Overview

- Feature generation
- Automated feature selection
- Scaling
- Missing value imputation
- Categorical feature encoding
- Handling imbalanced data
- Practical advice

Feature generation

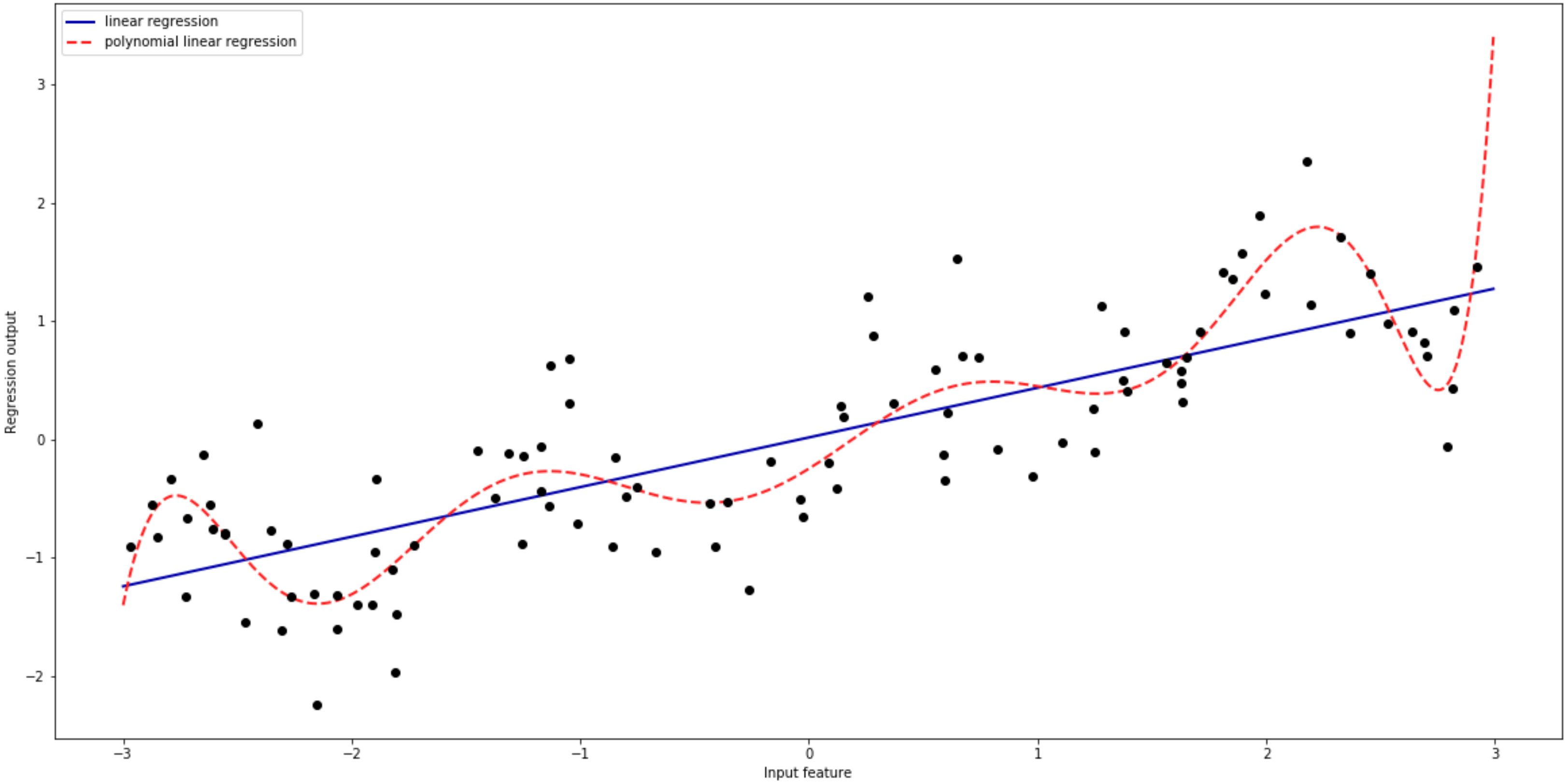
Polynomial features

- For a given feature x, we might want to consider adding x^2, x^3, x^4, and so on.
- This happens implictly in *kernelization* (SVMs, GLMs, Gaussian Processes,...)
- We can do it explictly, choosing a specific degree:
    - `degree` is the max. degree of the polynomials, requires some searching
    - `include_bias` adds the 0-th polynomial, i.e. x^0

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

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

- We added all polynomials up to the 10th degree.
- The model is still a hyperplane (in a 10-dimensional space)
- The predictions are now a non-linear function of the original variable X


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

Binning (Discretization)

Sometimes we need a different model for different *ranges* of the input data

- Partition (numeric) feature values into *n* 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

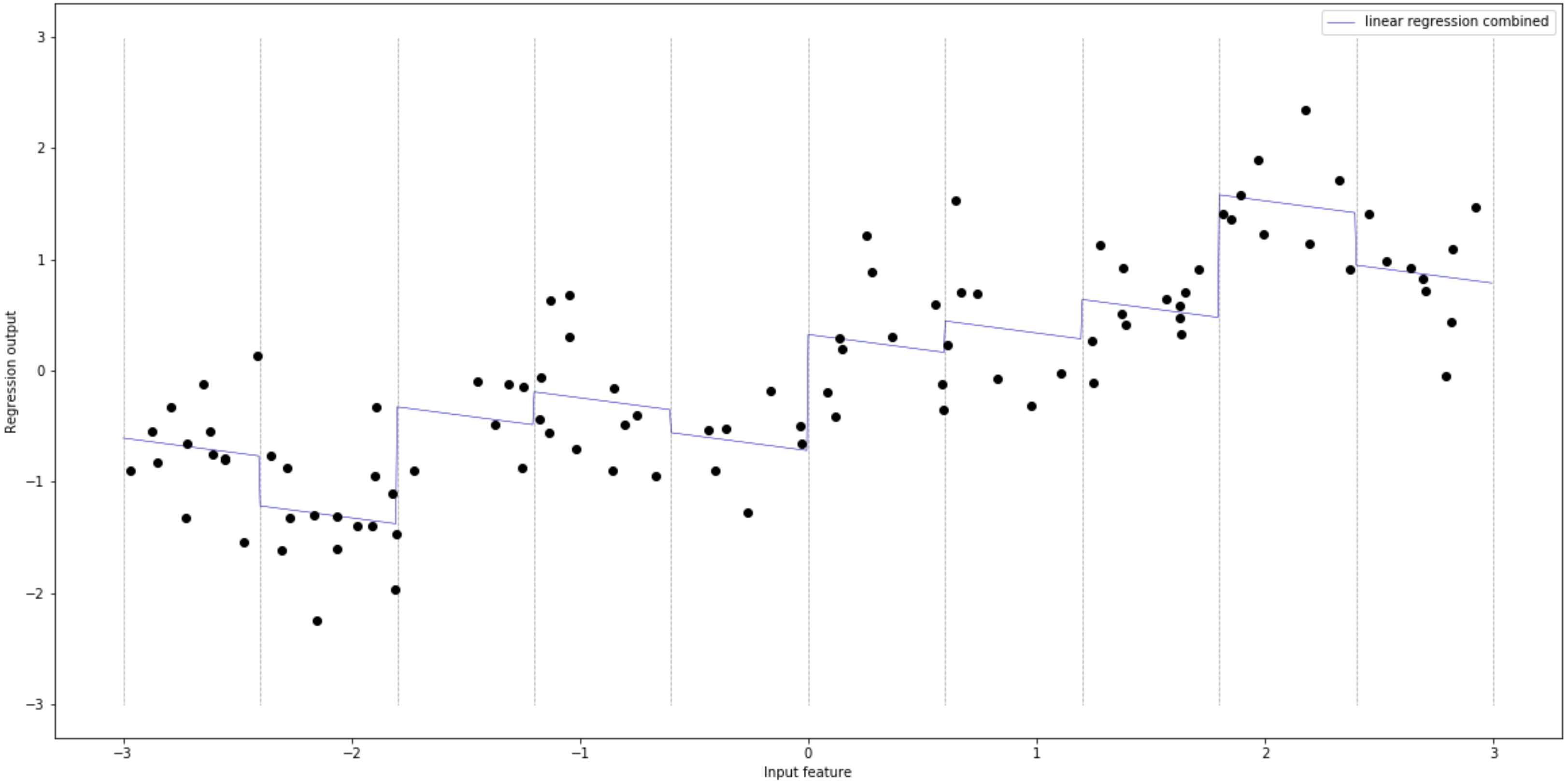
- No handy sklearn function. You need to use:
    - `numpy.digitize` to assign values to bins
    - `OneHotEncoder` to convert a feature of *n* numbers to *n* binary features
    - `numpy.stack` to add the original feature

```
which_bin = numpy.digitize(X, bins=np.linspace(-3, 3, 11))
X_binned = OneHotEncoder(sparse=False).fit_transform(which_bin)
X_combined = np.hstack([X, X_binned])
```

Out[6]:

	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
0	-0.75	0.00	0.00	0.00	1.00	0.00	0.00	0.00	0.00	0.00
1	2.70	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00
2	1.39	0.00	0.00	0.00	0.00	0.00	0.00	0.00	1.00	0.00
3	0.59	0.00	0.00	0.00	0.00	0.00	1.00	0.00	0.00	0.00
4	-2.06	0.00	1.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00

- Our linear model learned a fixed slope for each bin in the wave dataset
  - Because we learn a single weight for feature x



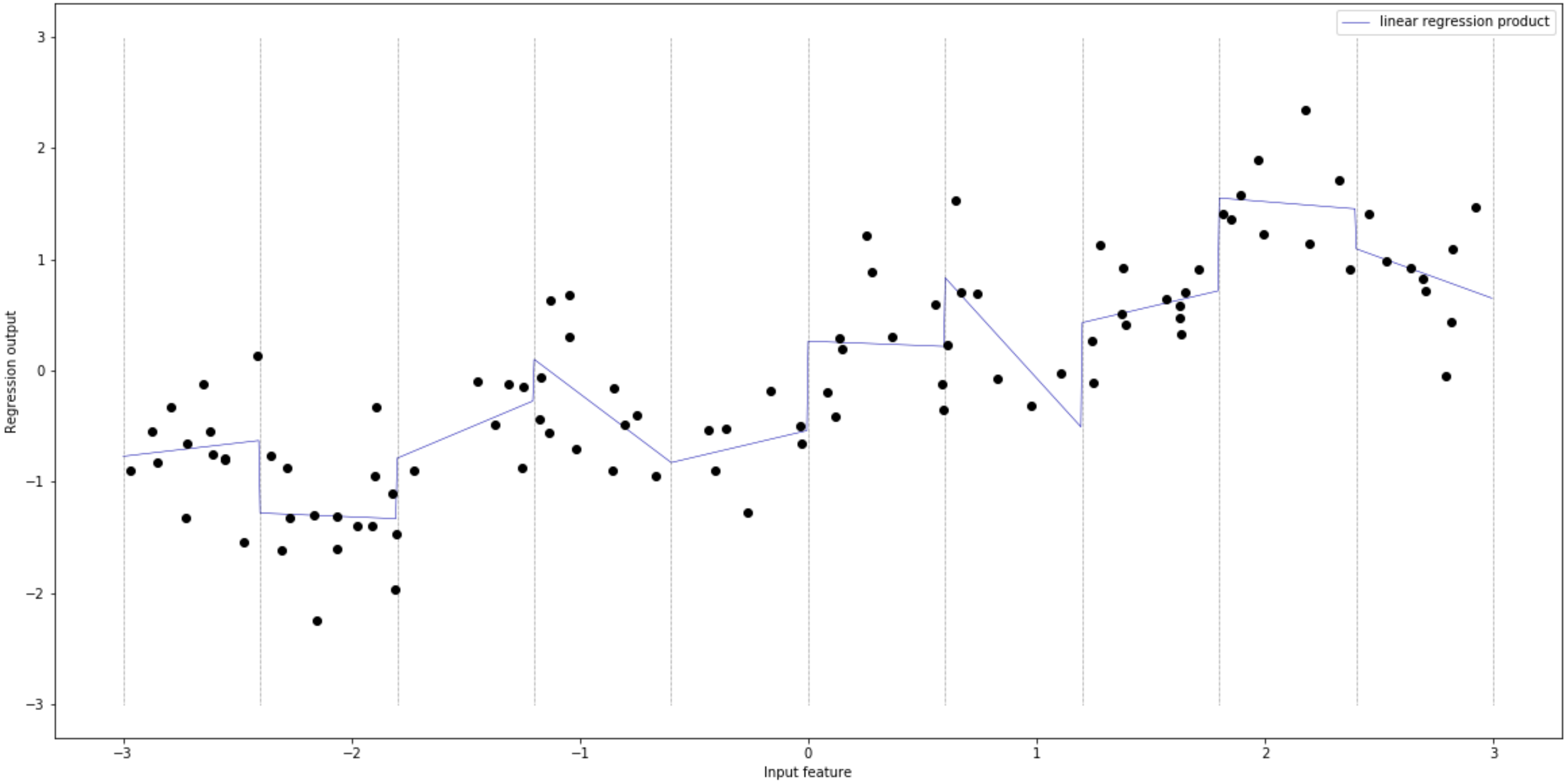
Interaction features

- If we want a different slope per bin, we need *interaction features* (or *product features*)
- These are the product of the bin encoding and the original feature value

Out[8]:

	b0	b1	b2	b3	b4	b5	b6	b7	b8	b9	X*b0	X*b1	X*b2	X*b3	X*b4	X*
0	0.00	0.00	0.00	1.00	0.00	0.00	0.00	0.00	0.00	0.00	-0.00	-0.00	-0.00	-0.75	-0.00	-0.
1	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	1.00	0.00	0.00	0.00	0.00	0.00	0.0
2	0.00	0.00	0.00	0.00	0.00	0.00	0.00	1.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.0
3	0.00	0.00	0.00	0.00	0.00	1.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.5
4	0.00	1.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	-0.00	-2.06	-0.00	-0.00	-0.00	-0.

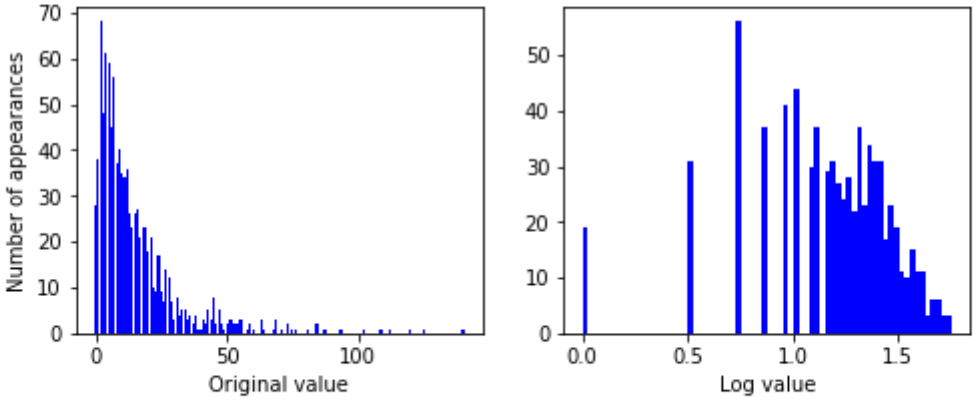
The interaction features allow the linear model to learn a weight (slope) per bin



Non-linear transformations

- There are other transformations that often prove useful for transforming certain features.
- Commonly, feature values follow a power law (or Poisson distribution), descreasing rapidly
    - E.g. the number of friends in a social network
- Models sensitive to feature scales (linear models, SVMs, neural networks, kNN,...) handle this badly
- Transform them to more Gaussian-like value distributions (somewhat of an art)

Taking log(x+1) of all numeric features yields *Poisson regression*.



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

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

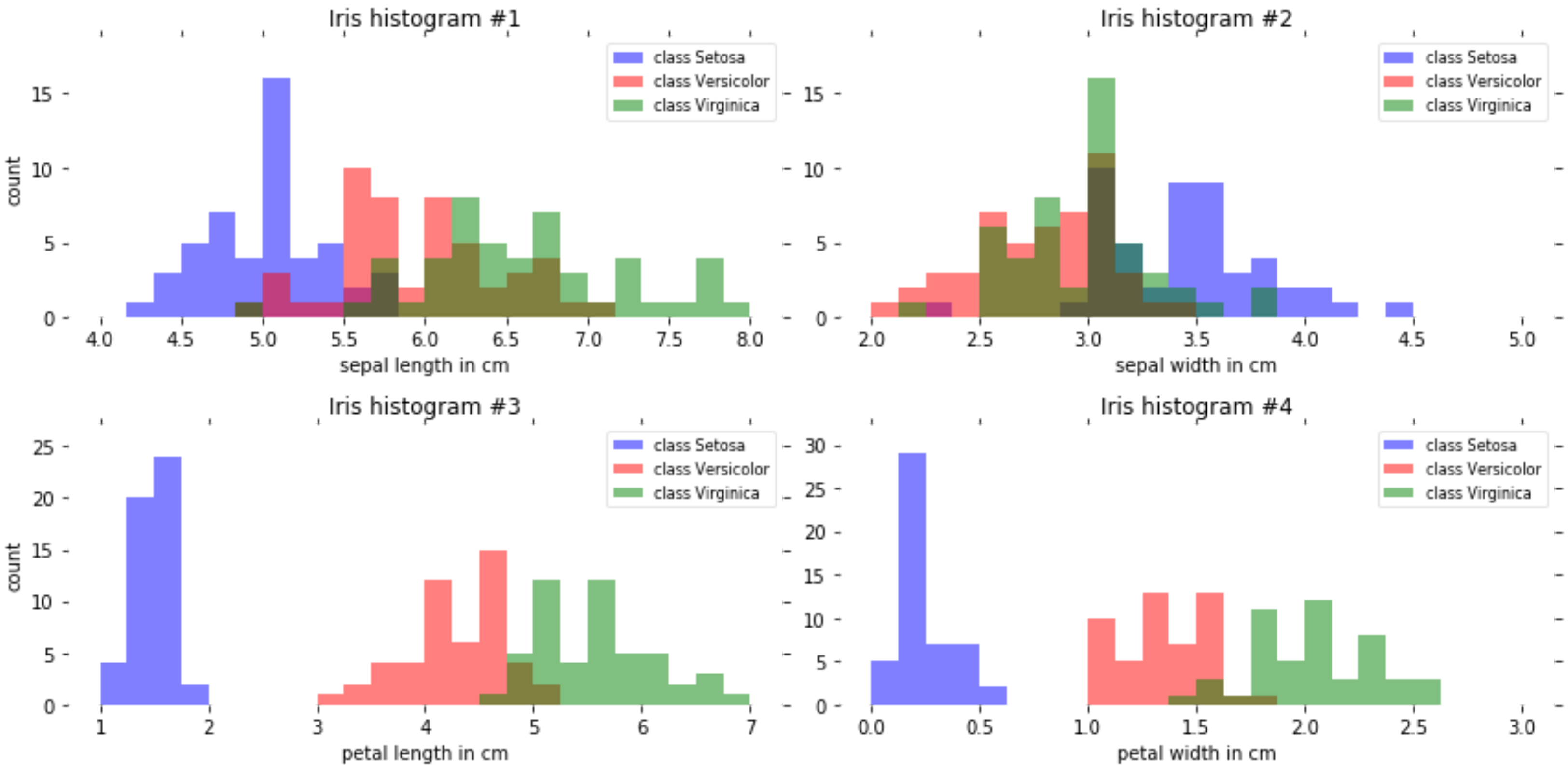
Automatic Feature Selection

It can be a good idea to reduce the number of features to only the most useful ones

- Simpler models that generalize better (less overfitting)
    - Even models such as RandomForest can benefit from this
- Help algorithms that are sensitive to the curse of dimensionality
    - e.g. kNN and many other distance-based methods
- Sometimes it is one of the main methods to improve models (e.g. gene expression data)

Example: Iris

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



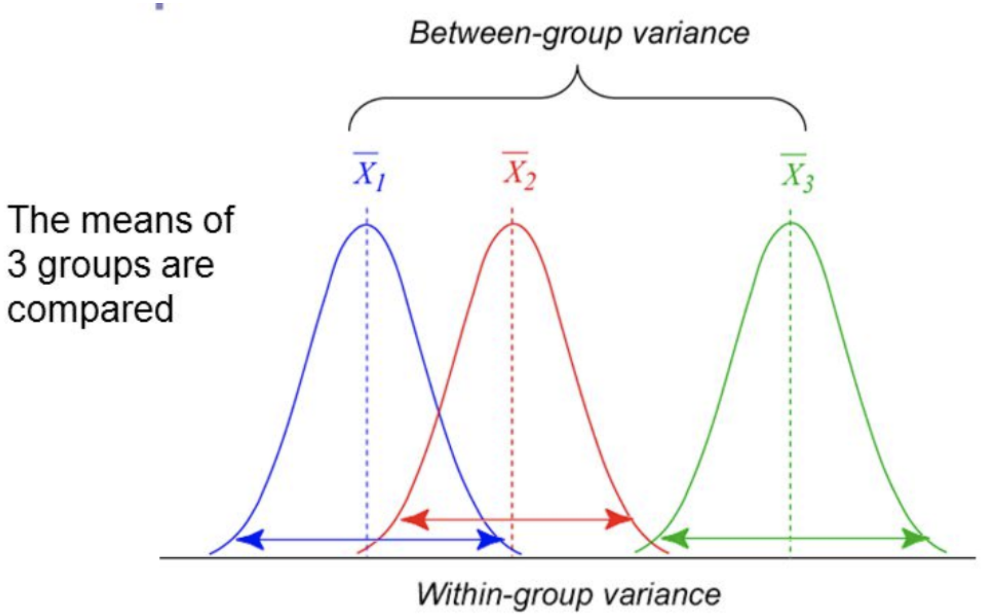
Univariate statistics

- Keep features for which there is a **statistically significant relationship** between it and the target.
- Consider each feature individually (univariate), independent of the model that you might want to apply afterwards.
- 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 within-class variance explained by one feature.

- F-statistic = $\frac{var(\mu_i)}{\overline{var}(X_i)}$ (higher is better)
  - X_i: all samples with class i.
  - Better if per-class distributions separate well: means are far apart and variance is small.



`chi2` : For categorical features and targets. Performs the chi-square (χ^2) statistic. Similar results as F-statistic, but less sensitive to nonlinear relationships.

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

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

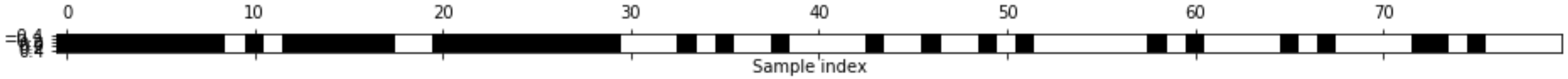
where O_{ij} is the number of observations of feature category i and class j, and E_{ij} is the expected number of observations of category i and class j if there was no relationship between the feature and the target (number of samples of category i * ratio of class j).

In practice

- Given a feature ranking, sklearn has two general ways to remove features :
- `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()`

Visualization:

- 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.

- Univariate statistics are not always very useful

```
LogisticRegression score with all features: 0.916
LogisticRegression score with only selected features: 0.919
```

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.

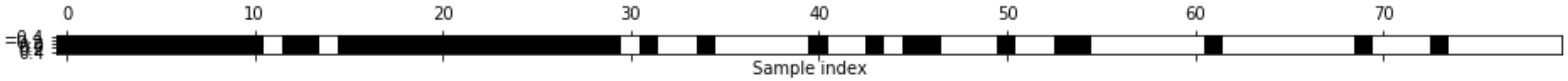
The supervised model that is used for feature selection doesn't need to be the same model that is used for the final supervised modeling, it only needs to be able to measure the (perceived) importance for each feature:

- Decision tree–based models return a `feature_importances_` attribute
- Linear models return coefficients (`coef_`), 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")
```

- Random Forests are known to produce good estimates of feature importance
    - Based on how often a feature is used high up in the trees
    - Based on Information Gain or Mean Decrease in Impurity (MDI)
    - Use with care: Beware Default Random Forest Importances (https://explained.ai/rf-importance/index.html)
    - Tune the RandomForest (e.g. `min_samples_leaf`)
    - Use permutation importance (coming up)
- In our example, all but two of the original features were selected, and most of the noise features removed.
- Our logistic regression model improves further



```
LogisticRegression test score: 0.930
```

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)
```

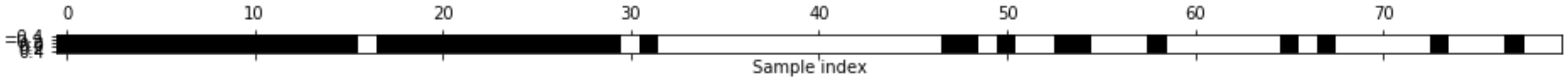
Vice versa, we could also ask it to iteratively add one feature at a time. This is called *forward selection*.

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.

Can be rather slow.

RFE result:

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



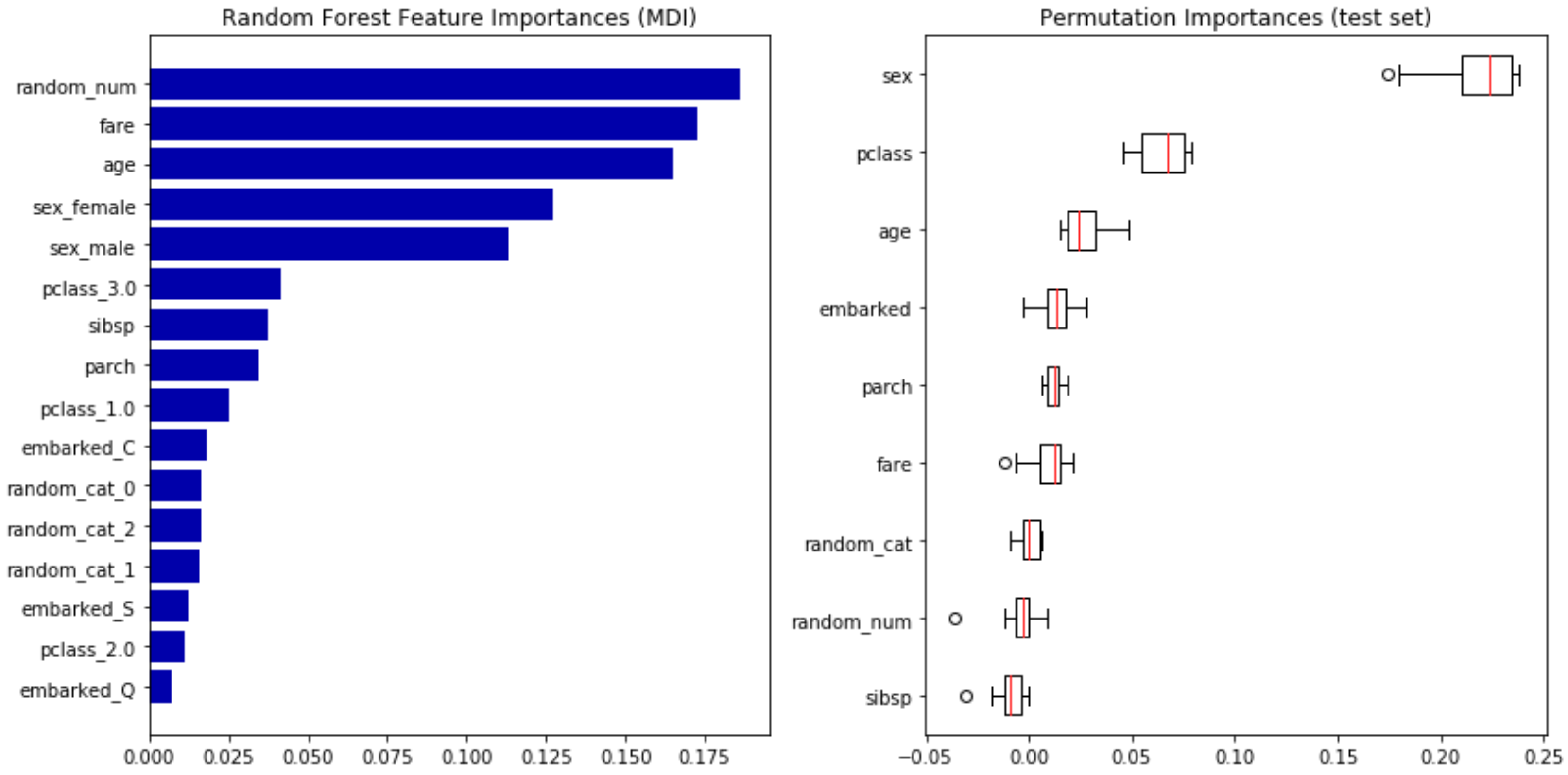
```
LogisticRegression Test score: 0.930
```

Permutation feature importance

- Model inspection technique, especially useful for non-linear or opaque estimators.
- Defined as **the decrease in a model score when a single feature value is randomly shuffled**.
- This breaks the relationship between the feature and the target, thus the drop in the model score is indicative of how much the model depends on the feature.
- Model agnostic, metric agnostic, and can be calculated many times with different permutations.
- The problem with impurity based techniques (e.g. Random Forest)
    - Gives importance to features not predictive on unseen data.
        - Permutation feature importance can be applied to unseen data.
    - Strong bias towards high cardinality features (e.g. numerical features).
        - Permutation feature importances do not exhibit such a bias.

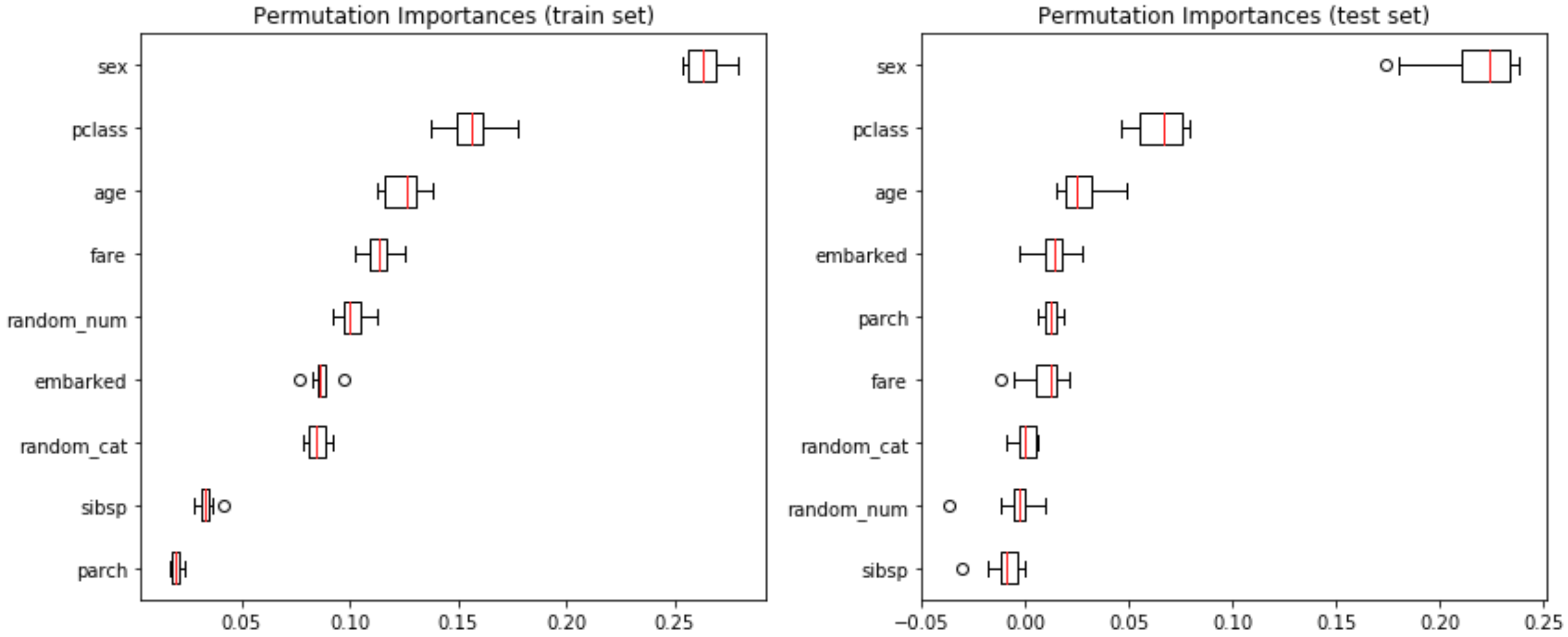
Example (Titanic dataset)

- We add a random feature as well: Random Forest deems it important!
- Low cardinality feature `sex` and `pclass` are actually more important



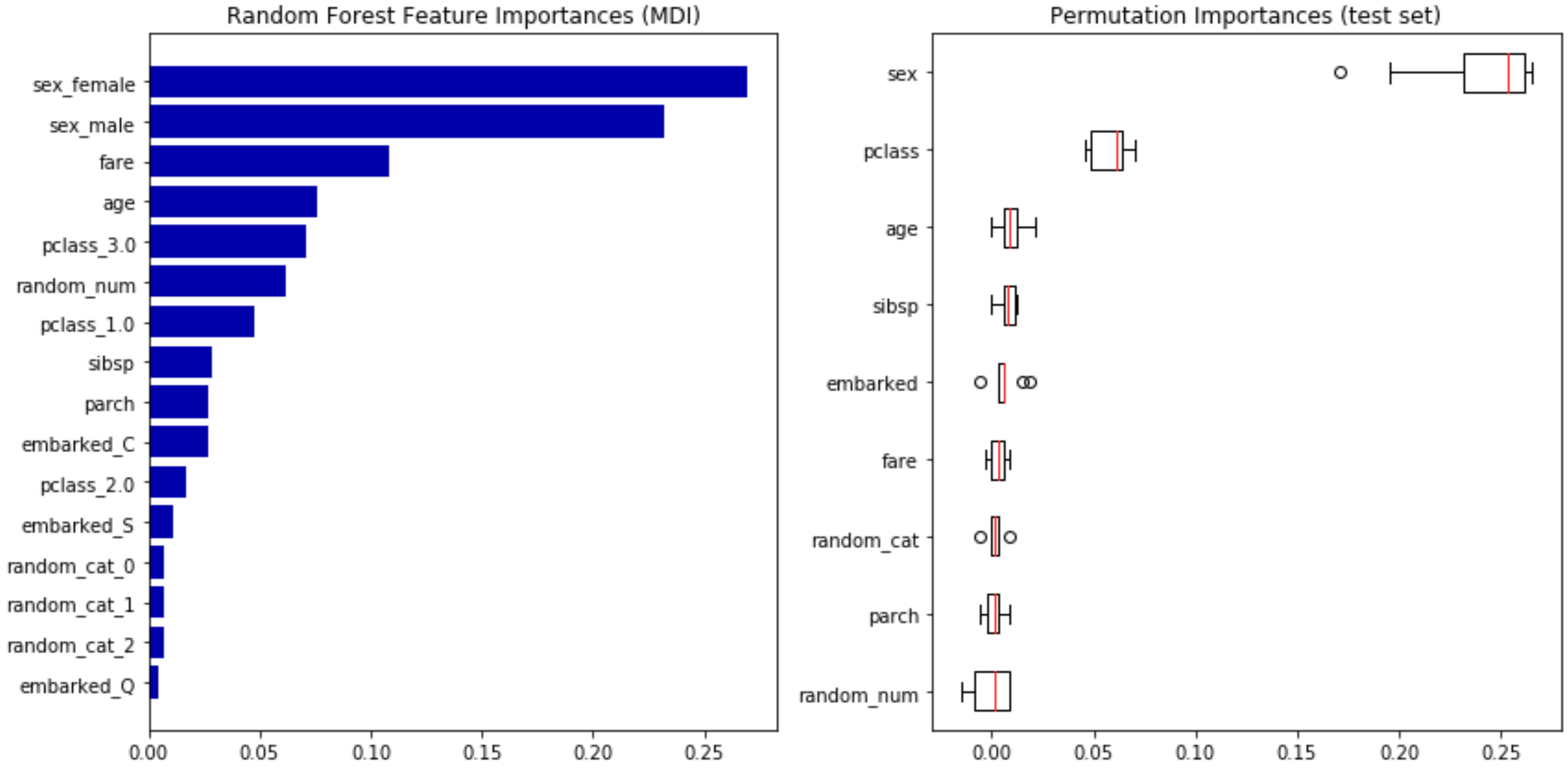
We can also compute the permutation importances on the training set.

- `random_num` gets a significantly higher importance ranking than when computed on the test set.
- This shows that the RF model has enough capacity to use that random numerical feature to overfit.



Let's rerun this with a more regularized Random Forest (`min_samples_leaf=10`)

- A well-tuned random forest has less tendency to choose the random features



Feature selection wrap-up

Automatic feature selection can be helpful when:

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

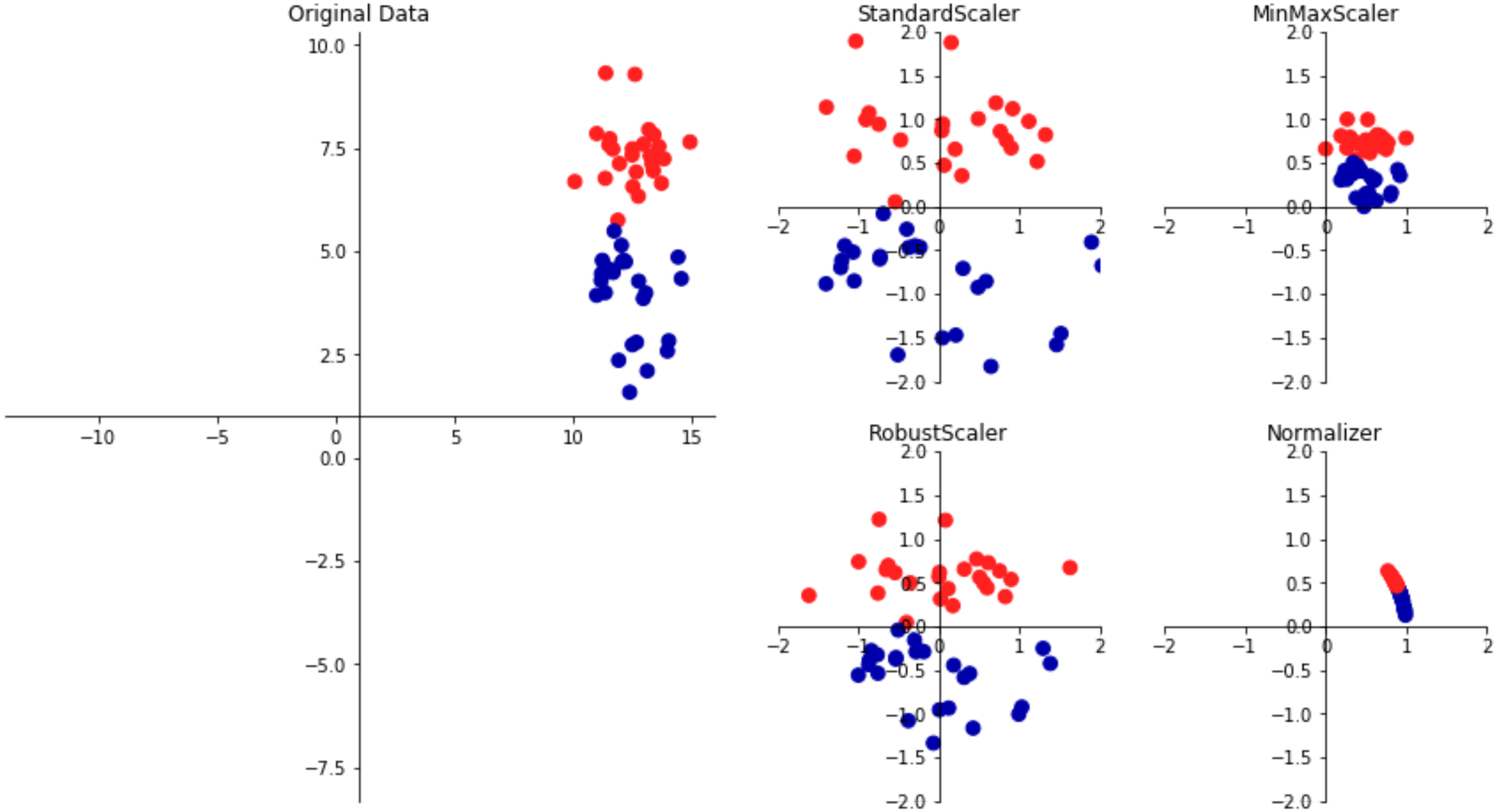
Scaling

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
10.0
7.5
5.0
2.5
0.0
−2.5
−5.0
−7.5
−10
−5
0
5
10
15
StandardScaler
MinMaxScaler
RobustScaler
Normalizer
2.0
1.5
1.0
0.5
0.0
−0.5
−1.0
−1.5
−2.0
−2
−1
0
1
2

Applying scaling transformations

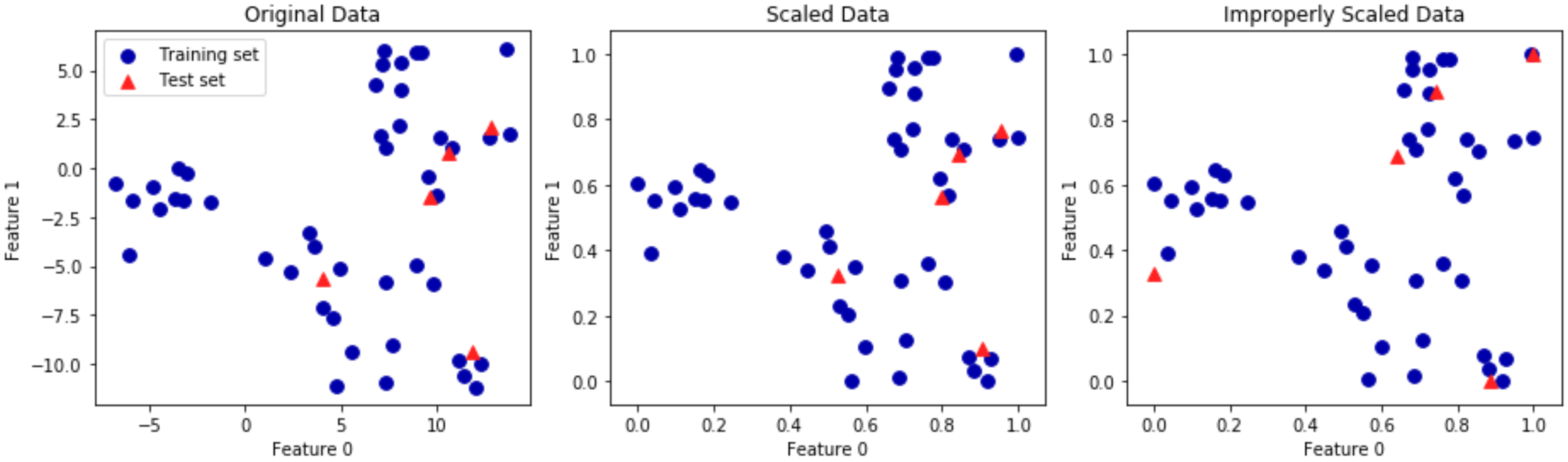
- Lets apply a scaling transformation *manually*, then use it to train a learning algorithm
- First, split the data in training and test set
- Next, we `fit` the preprocessor on the **training data**
    - This computes the necessary transformation parameters
    - For `MinMaxScaler`, these are the min/max values for every feature
- After fitting, we can `transform` the training and test data

```
scaler = MinMaxScaler()
scaler.fit(X_train)
X_train_scaled = scaler.transform(X_train)
X_test_scaled = scaler.transform(X_test)
```

- 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: 0.95
```

- 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 = SimpleImputer(missing_values=np.nan, strategy='mean')
imp.fit_transform(X1_train)
```

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

Categorical feature encoding

- Many algorithms can only handle numeric features, so we need to encode the categorical ones

```
Out[40]:
```

	boro	salary	vegan
0	Manhattan	103	No
1	Queens	89	No
2	Manhattan	142	No
3	Brooklyn	54	Yes
4	Brooklyn	63	Yes
5	Bronx	219	No

Ordinal encoding

- Simply assigns an integer value to each category in the order they are encountered
- Often bad: a model will think that one category is 'higher' or 'closer' to another

Out[41]:

	boro	boro_ordinal	vegan
0	Manhattan	2	No
1	Queens	3	No
2	Manhattan	2	No
3	Brooklyn	1	Yes
4	Brooklyn	1	Yes
5	Bronx	0	No

Dummy encoding

- Simply adds a new 0/1 feature for every category, having 1 (hot) if the sample has that category
- Can explode if a feature has lots of values, causing issues with high dimensionality

```
Out[42]:
```

	boro_ordinal	vegan	boro_Bronx	boro_Brooklyn	boro_Manhattan	boro_Queens
0	2	No	0	0	1	0
1	3	No	0	0	0	1
2	2	No	0	0	1	0
3	1	Yes	0	1	0	0
4	1	Yes	0	1	0	0
5	0	No	1	0	0	0

Target encoding

- Calculates the posterior probability for each class *given a certain categorical value*
    - For regression, it calculates the expected value of the target given a categorical value.
- Features are replaced with a blend of posterior probability of the target $\frac{n_{iY}}{n_i}$ and its prior probability $\frac{n_Y}{n}$.
    - n_{iY} is the number of samples with category i and class Y=1
    - Blending is typically done using the logit function (S-curve)

$$Enc(i) = \frac{1}{1+e^{-(n_i-1)}}\frac{n_{iY}}{n_i} + (1 - \frac{1}{1+e^{-(n_i-1)}})\frac{n_Y}{n}$$

    - Same for regression, using the expected value
        - $\frac{n_{iY}}{n_i}$ is average target value with category i, $\frac{n_Y}{n}$ the overall average
- Preferred when you have lots of levels. It only creates a few new features (1 per class)

Example:

- For Manhattan, $n_{iY} = 0, n_i = 2, n_Y = 2, n = 6$

$$Enc(Manhattan) = 0 + (1 - \frac{1}{1 + e^{-1}})\frac{2}{6} = 0, 0896$$

Out[43]:

	boro	salary	boro_original	vegan
0	0.09	103	Manhattan	No
1	0.33	89	Queens	No
2	0.09	142	Manhattan	No
3	0.82	54	Brooklyn	Yes
4	0.82	63	Brooklyn	Yes
5	0.33	219	Bronx	No

Handling imbalanced data

- Randomly oversample the minority class
  - Sample the same points over and over again
- Randomly undersample the majority class
  - Faster, but may lose information
- Add class weights to the classifier loss function
  - Contribution of every wrongly predicted point is weighted by its class's imbalance
  - Most algorithms allow this (`class_weight` hyperparameter)
- Ensemble resampling
  - Create an ensemble by iteratively applying random under-sampling

Handling imbalanced data

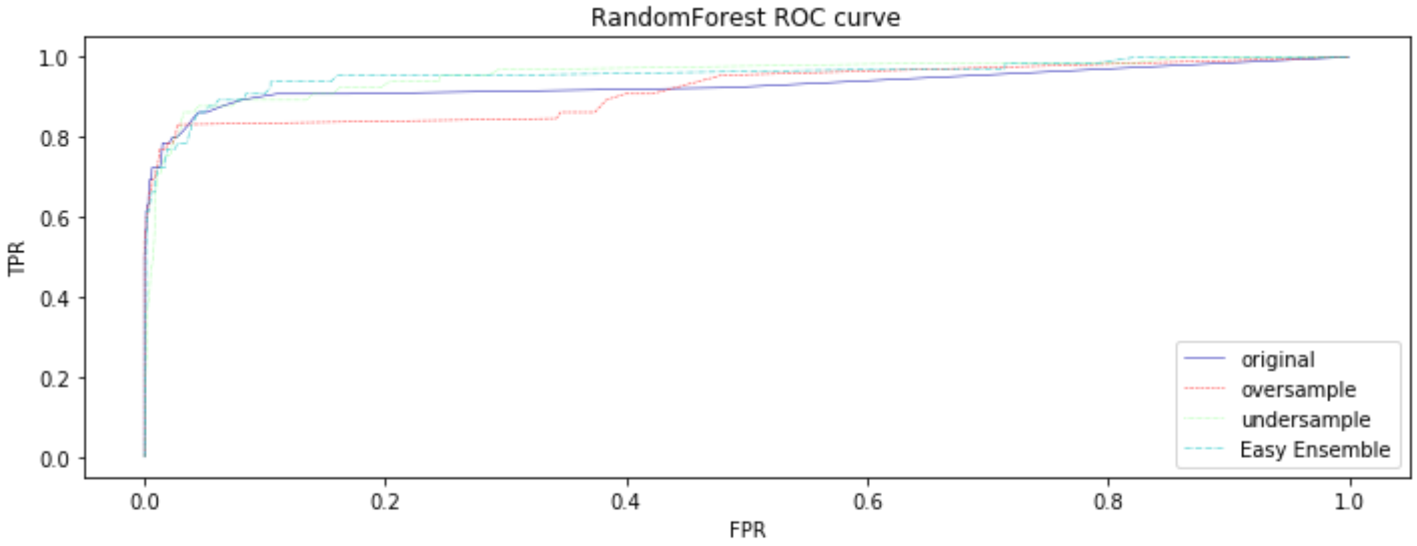
- Edited Nearest Neighbors
    - Remove all majority samples that are misclassified by KNN (mode) or that have a neighbor from the other class (all).
    - Remove their influence on the minority samples
- Condensed Nearest Neighbors
    - Remove all majority samples that are *not* misclassified by KNN
    - Focus on only the hard samples.
- Synthetic Minority Oversampling Technique (SMOTE)
    - Choose a minority point and a neighboring minority point
    - Add a new, artificial point on the line between them
    - May bias the data. Be careful never to use artificial points in the test set.

Handling imbalanced data in practice

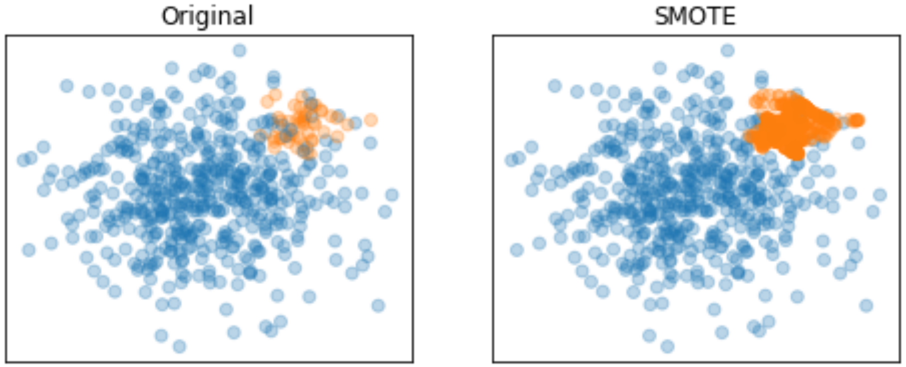
- http://imbalanced-learn.org (http://imbalanced-learn.org)
- Always build *pipelines* of a sampler and a learner
    - The test set should never be affected by resampling
- Methods:
    - RandomUnderSampler
    - RandomOverSampler
    - BalancedBaggingClassifier
    - EditedNearestNeighbours
    - CondensedNearestNeighbour
    - SMOTE

Comparison

```
Using TensorFlow backend.
```



SMOTE



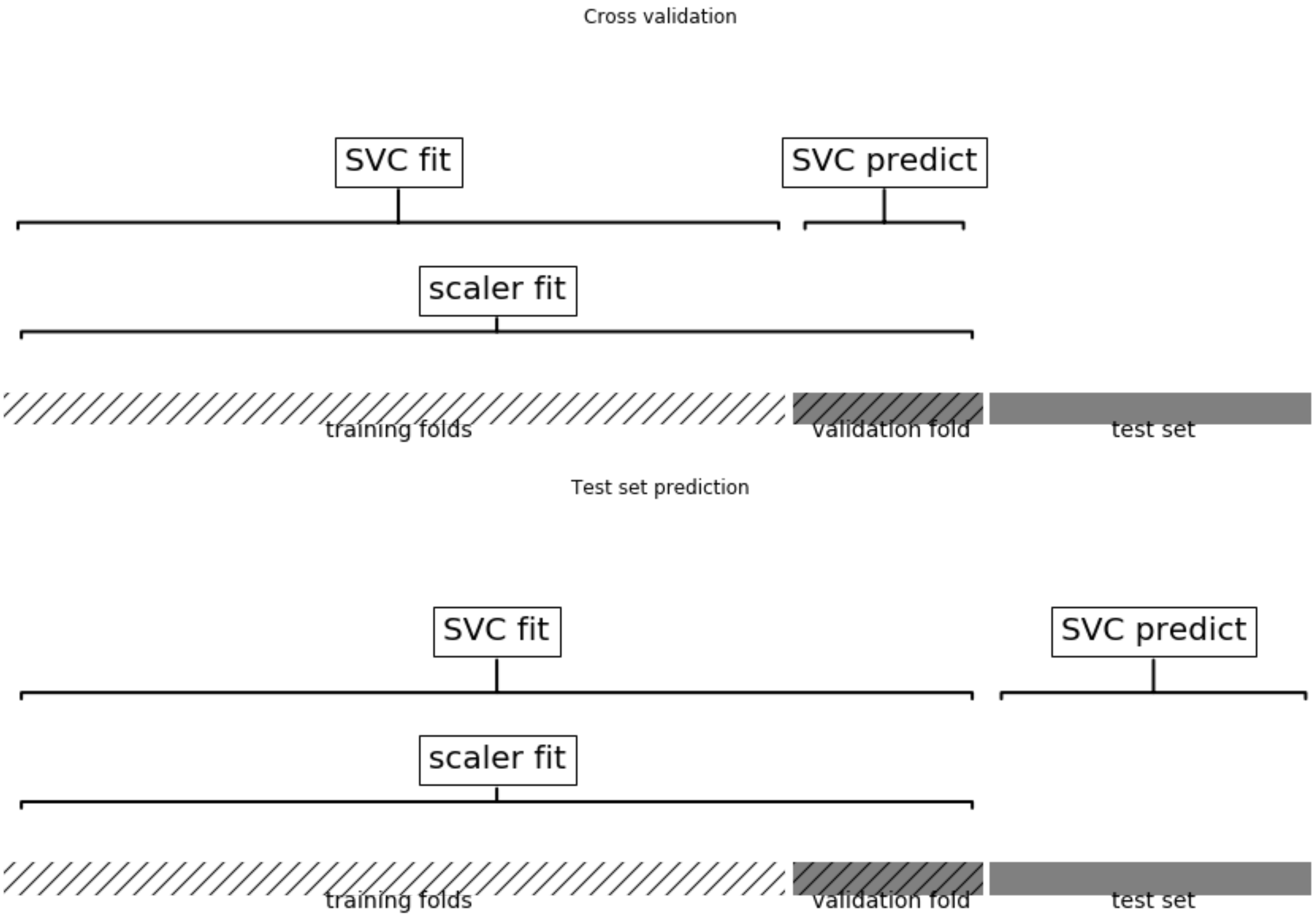
Hyperparameter Selection with Preprocessing

- 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)

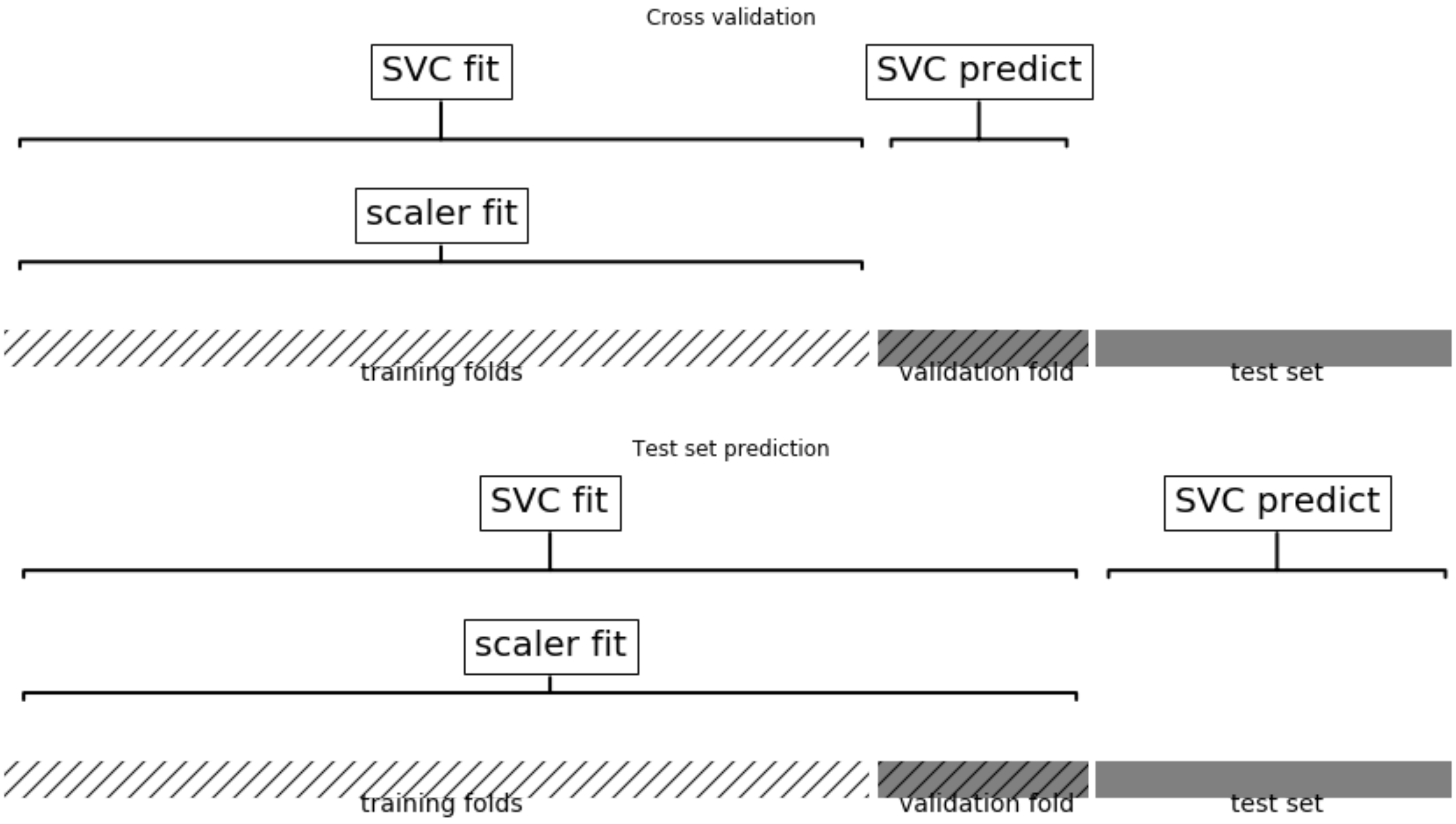
```
Best cross-validation accuracy: 0.98
Best set score: 0.97
Best parameters:  {'C': 1, 'gamma': 1}
```

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

- We need to include both preprocessor and learning in a *pipeline* and handle it as a single estimator
- Now, the preprocessors are refit with only the training data in each cross-validation split.



Summary

- Feature engineering
  - Make the problem easier by adding polynomials, interactions, binning
  - Correct data distributions (e.g. Poisson regression)
- Automated feature selection
  - Univariate techniques give mixed results
  - Model based results often better (but biased)
  - Permutation importance often useful (yet more expensive)
- Scaling
  - Crucial for any distance-based algorithm (SVM, LogReg, Neural nets)
  - Standard/Robust scaling works well, but no one-fits-all

Summary

- Missing value imputation
    - Removing rows/columns may destroy information
    - Simple imputation may introduce bias
    - Better but more expensive: model-based, matrix factorization,...
- Categorical feature encoding
    - One-hot-encoding is useful, but may explode the number of features
    - Target encoding useful for multi-category features, but mixed results
    - Many techniques, no one-fits-all
- Handling imbalanced data
    - Above all, choose a good evaluation measure
    - Under- or oversampling and ensembles work well
    - SMOTE constructs artificial points, mixed results