Lecture 6. Data preprocessing

Real-world machine learning pipelines

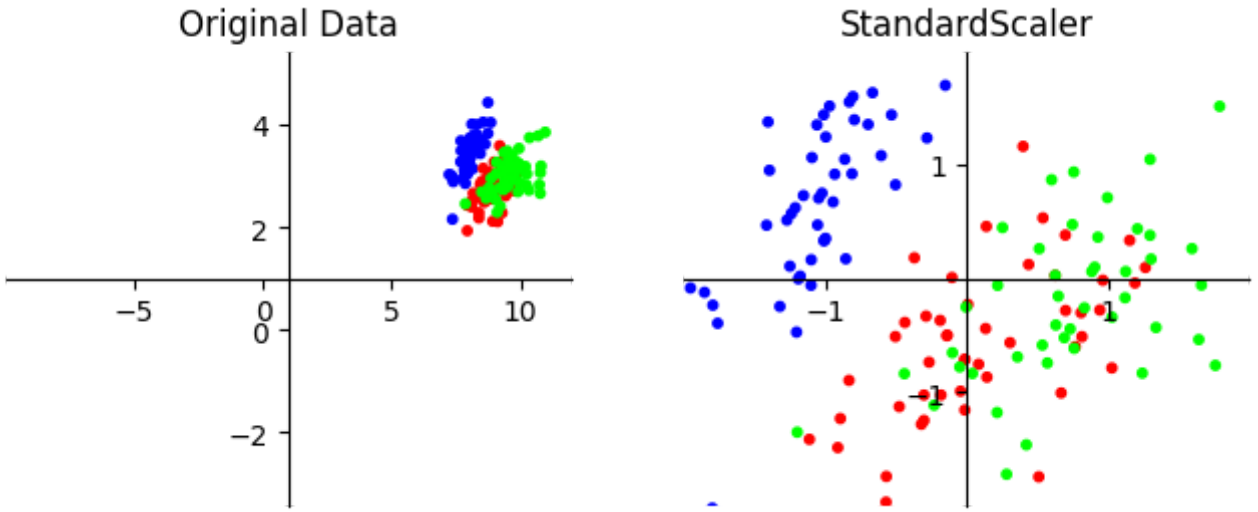
Joaquin Vanschoren

Data transformations

- Machine learning models make a lot of assumptions about the data
- In reality, these assumptions are often violated
- We build *pipelines* that *transform* the data before feeding it to the learners
    - Scaling (or other numeric transformations)
    - Encoding (convert categorical features into numerical ones)
    - Automatic feature selection
    - Feature engineering (e.g. binning, polynomial features,...)
    - Handling missing data
    - Handling imbalanced data
    - Dimensionality reduction (e.g. PCA)
    - Learned embeddings (e.g. for text)
- Seek the best combinations of transformations and learning methods
    - Often done empirically, using cross-validation
    - Make sure that there is no data leakage during this process!

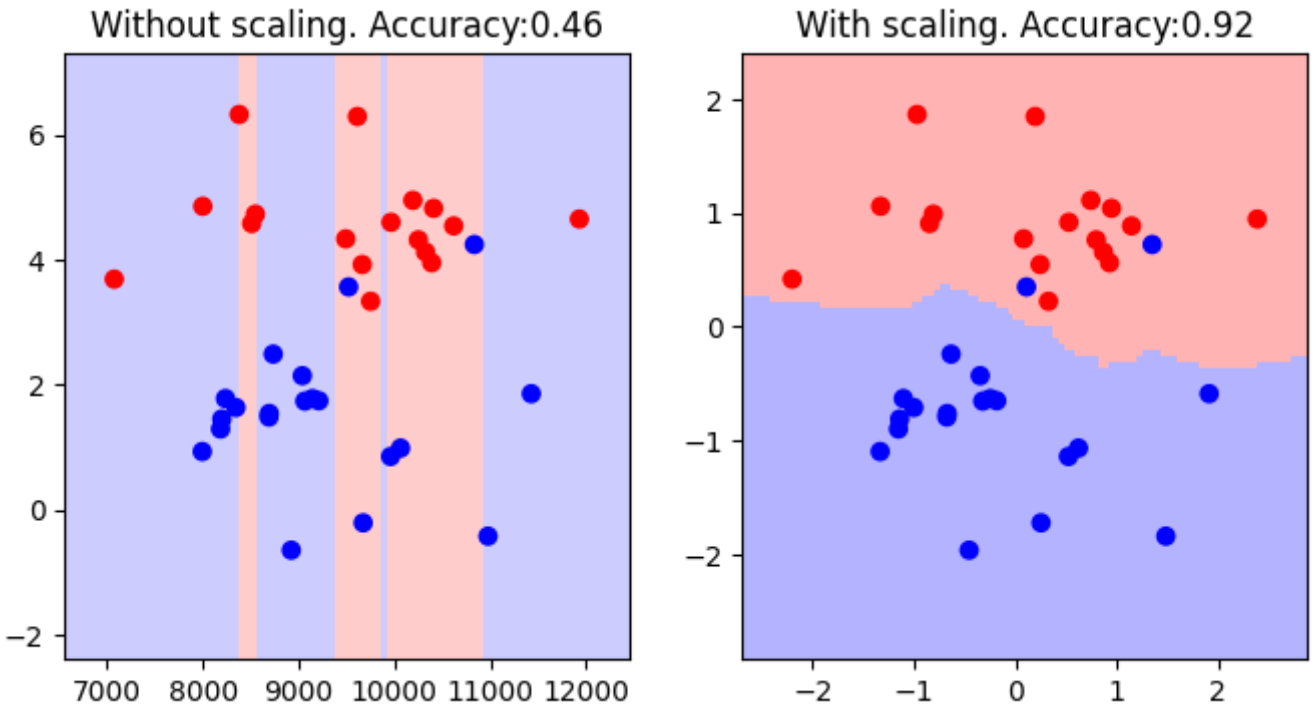
Scaling

- Use when different numeric features have different scales (different range of values)
    - Features with much higher values may overpower the others
- Goal: bring them all within the same range
- Different methods exist



Why do we need scaling?

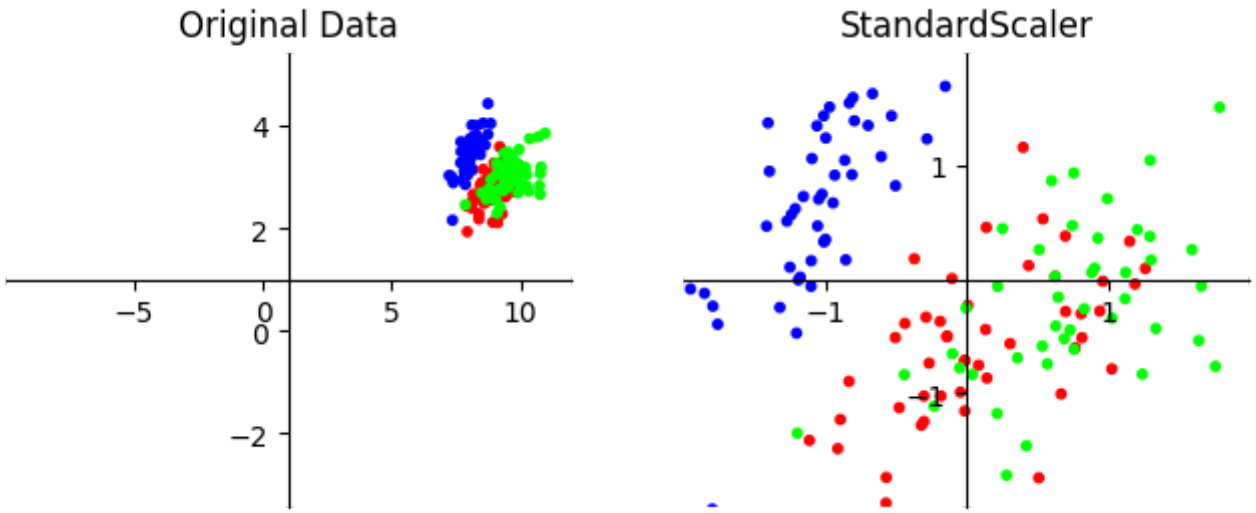
- KNN: Distances depend mainly on feature with larger values
- SVMs: (kernelized) dot products are also based on distances
- Linear model: Feature scale affects regularization
    - Weights have similar scales, more interpretable



Standard scaling (standardization)

- Generally most useful, assumes data is more or less normally distributed
- Per feature, subtract the mean value μ, scale by standard deviation σ
- New feature has $\mu = 0$ and $\sigma = 1$, values can still be arbitrarily large

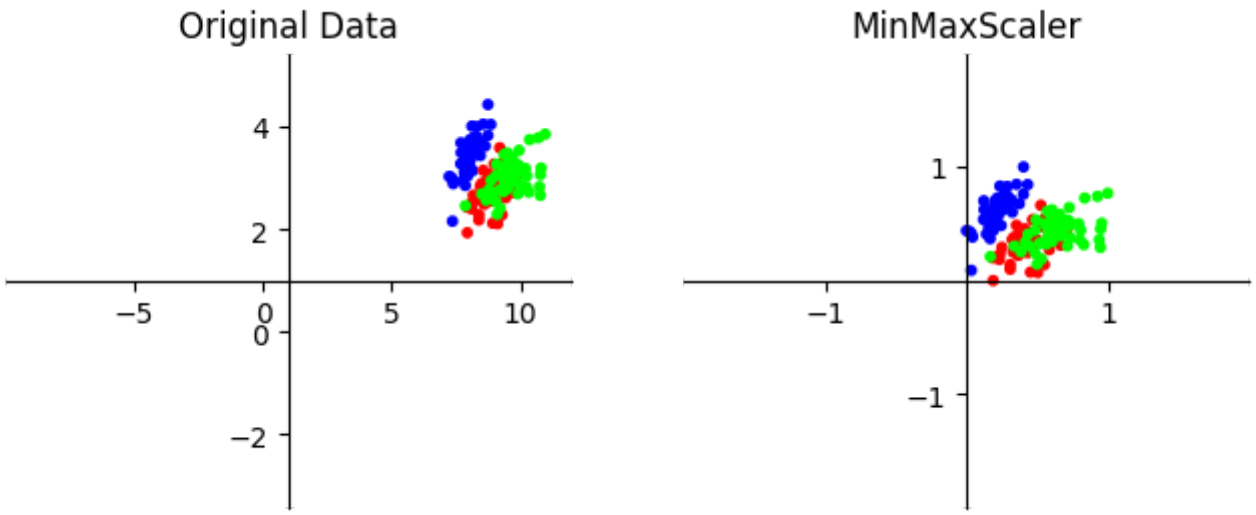
$$\mathbf{x}_{new} = \frac{\mathbf{x} - \mu}{\sigma}$$



Min-max scaling

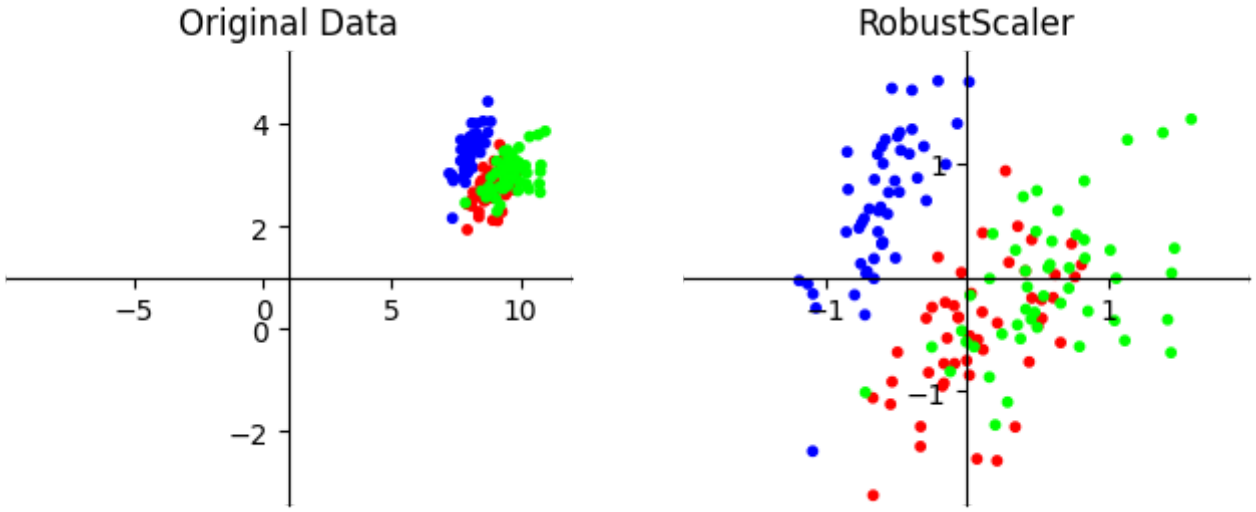
- Scales all features between a given min and max value (e.g. 0 and 1)
- Makes sense if min/max values have meaning in your data
- Sensitive to outliers

$$\mathbf{x}_{new} = \frac{\mathbf{x} - x_{min}}{x_{max} - x_{min}} \cdot (max - min) + min$$



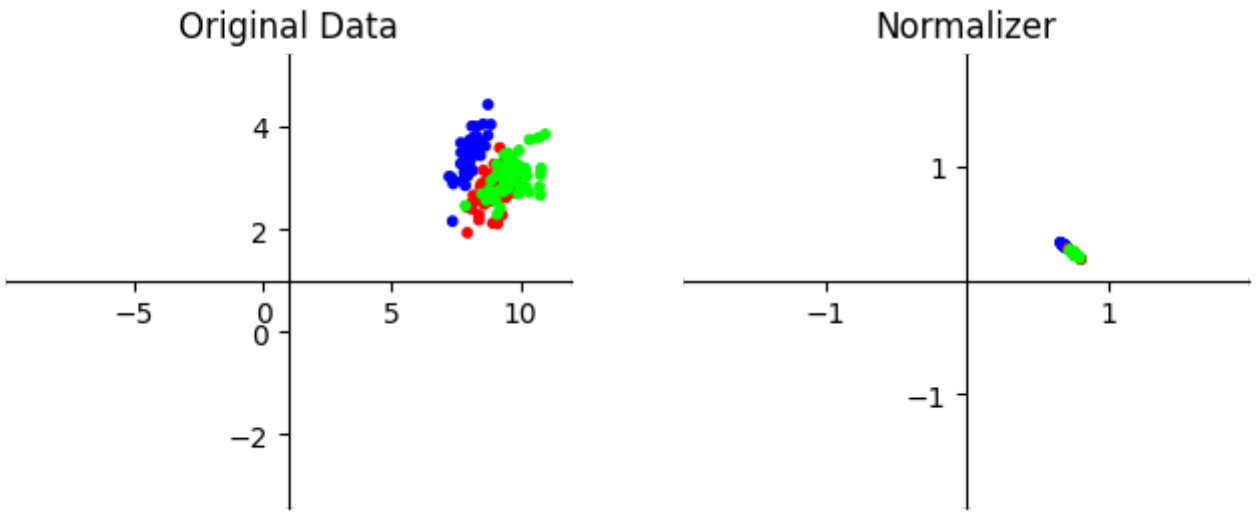
Robust scaling

- Subtracts the median, scales between quantiles q_{25} and q_{75}
- New feature has median 0, $q_{25} = -1$ and $q_{75} = 1$
- Similar to standard scaler, but ignores outliers



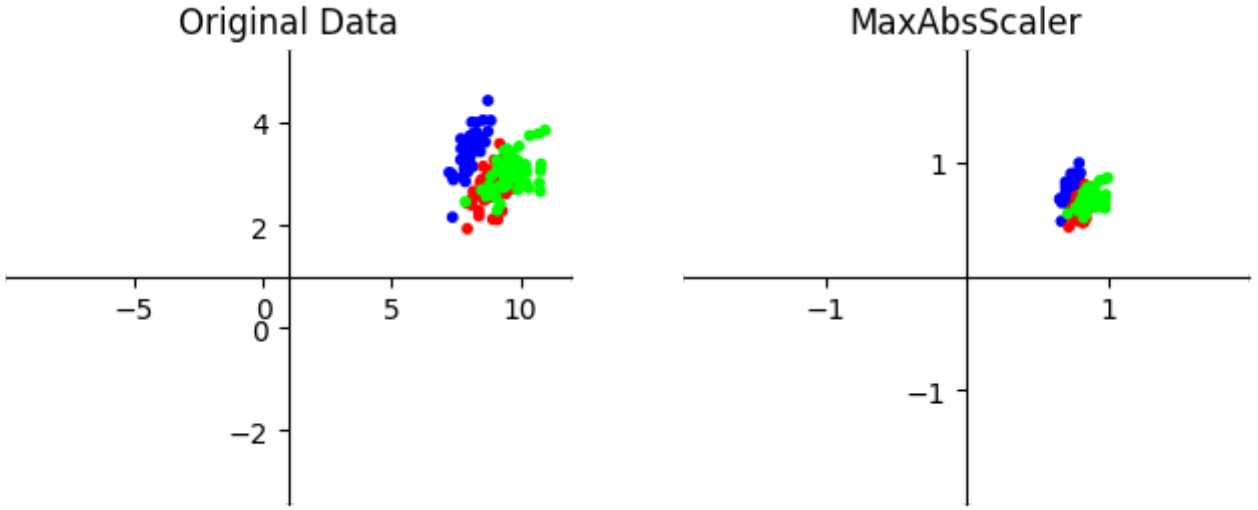
Normalization

- Makes sure that feature values of each point (each row) sum up to 1 (L1 norm)
    - Useful for count data (e.g. word counts in documents)
- Can also be used with L2 norm (sum of squares is 1)
    - Useful when computing distances in high dimensions
    - Normalized Euclidean distance is equivalent to cosine similarity



Maximum Absolute scaler

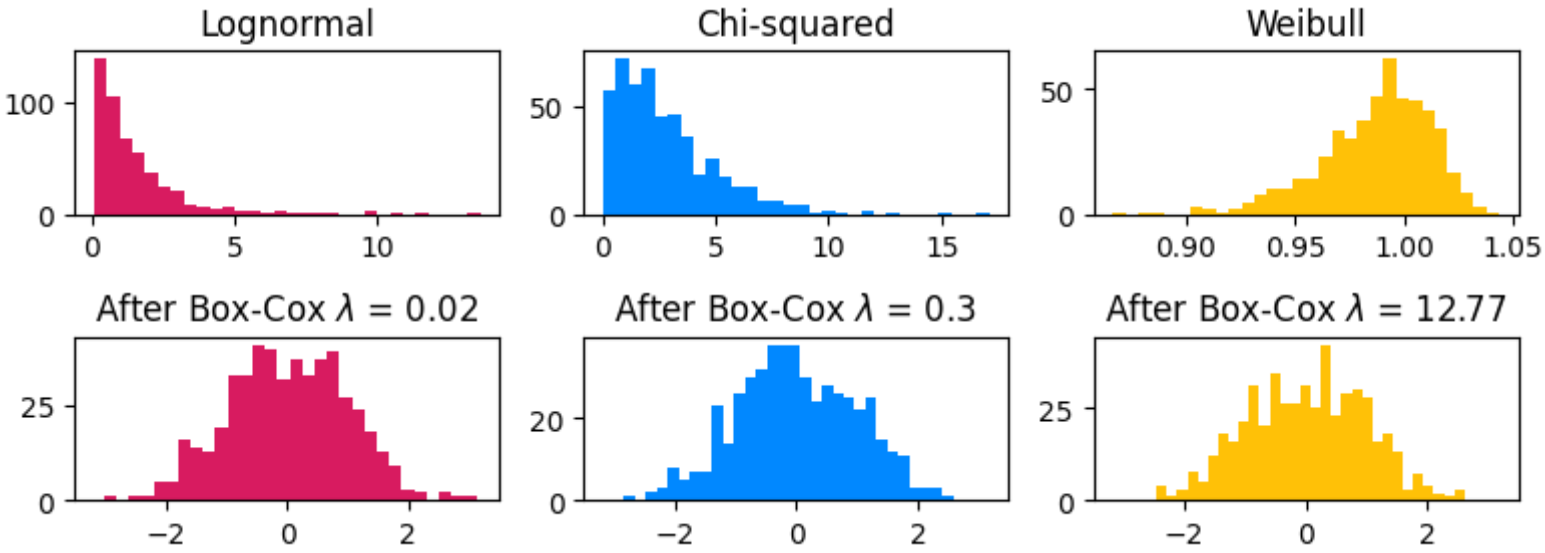
- For sparse data (many features, but few are non-zero)
    - Maintain sparseness (efficient storage)
- Scales all values so that maximum absolute value is 1
- Similar to Min-Max scaling without changing 0 values



Power transformations

- Some features follow certain distributions
    - E.g. number of twitter followers is log-normal distributed
- Box-Cox transformations transform these to normal distributions (λ is fitted)
    - Only works for positive values, use Yeo-Johnson otherwise

$$bc_\lambda(x) = \begin{cases} log(x) & \lambda = 0 \\ \frac{x^\lambda - 1}{\lambda} & \lambda \neq 0 \end{cases}$$



Categorical feature encoding

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

	boro	salary	vegan
0	Manhattan	103	0
1	Queens	89	0
2	Manhattan	142	0
3	Brooklyn	54	1
4	Brooklyn	63	1
5	Bronx	219	0

Ordinal encoding

- Simply assigns an integer value to each category in the order they are encountered
- Only really useful if there exist a natural order in categories
    - Model will consider one category to be 'higher' or 'closer' to another

	boro	boro_ordinal	salary
0	Manhattan	2	103
1	Queens	3	89
2	Manhattan	2	142
3	Brooklyn	1	54
4	Brooklyn	1	63
5	Bronx	0	219

One-hot encoding (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
- What if test set contains a new category not seen in training data?
    - Either ignore it (just use all 0's in row), or handle manually (e.g. resample)

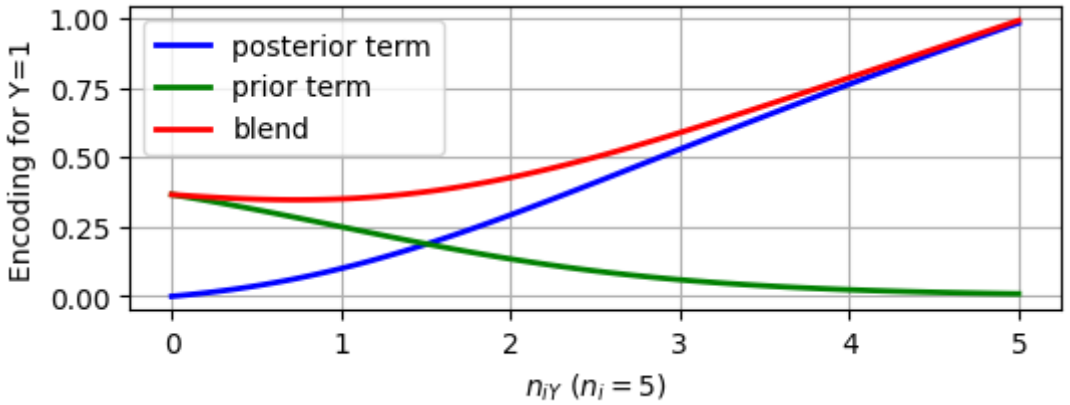
	boro	boro_Bronx	boro_Brooklyn	boro_Manhattan	boro_Queens	salary
0	Manhattan	0	0	1	0	103
1	Queens	0	0	0	1	89
2	Manhattan	0	0	1	0	142
3	Brooklyn	0	1	0	0	54
4	Brooklyn	0	1	0	0	63
5	Bronx	1	0	0	0	219

Target encoding

- Value close to 1 if category correlates with class 1, close to 0 if correlates with class 0
- Blends posterior probability of the target $\frac{n_{iY}}{n_i}$ and prior probability $\frac{n_Y}{n}$.
    - n_{iY}: nr of samples with category i and class Y=1, n_i: nr of samples with category i
    - Blending is 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 $\frac{n_{iY}}{n_i}$: average target value with category i, $\frac{n_Y}{n}$: overall mean
- Preferred when you have lots of category values. It only creates a few new features (1 per class)



Example:

- For Brooklyn, $n_{iY} = 2, n_i = 2, n_Y = 2, n = 6$
- Would be closer to 1 if there were more examples, all with label 1

$$Enc(Brooklyn) = \frac{1}{1+e^{-1}}\frac{2}{2} + (1 - \frac{1}{1+e^{-1}})\frac{2}{6} = 0,82$$

	boro	boro_encoded	salary	vegan
0	Manhattan	0.09	103	0
1	Queens	0.33	89	0
2	Manhattan	0.09	142	0
3	Brooklyn	0.82	54	1
4	Brooklyn	0.82	63	1
5	Bronx	0.33	219	0

Applying data transformations

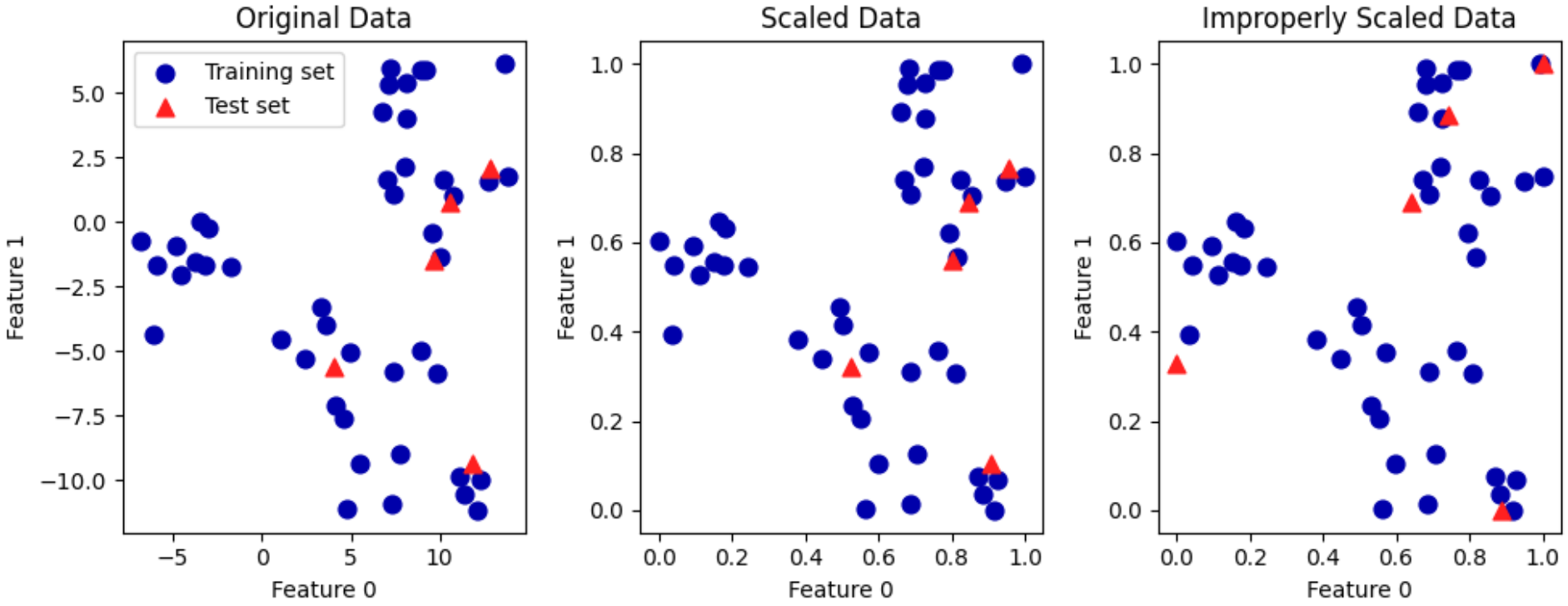
- Data transformations should always follow a fit-predict paradigm
    - Fit the transformer on the training data only
        - E.g. for a standard scaler: record the mean and standard deviation
    - Transform (e.g. scale) the training data, then train the learning model
    - Transform (e.g. scale) the test data, then evaluate the model
- Only scale the input features (X), not the targets (y)
- If you fit and transform the whole dataset before splitting, you get data leakage
    - You have looked at the test data before training the model
    - Model evaluations will be misleading
- If you fit and transform the training and test data separately, you distort the data
    - E.g. training and test points are scaled differently

In practice (scikit-learn)

```
# choose scaling method and fit on training data
scaler = StandardScaler()
scaler.fit(X_train)

# transform training and test data
X_train_scaled = scaler.transform(X_train)
X_test_scaled = scaler.transform(X_test)


# calling fit and transform in sequence
X_train_scaled = scaler.fit(X_train).transform(X_train)
# same result, but more efficient computation
X_train_scaled = scaler.fit_transform(X_train)
```

Test set distortion example

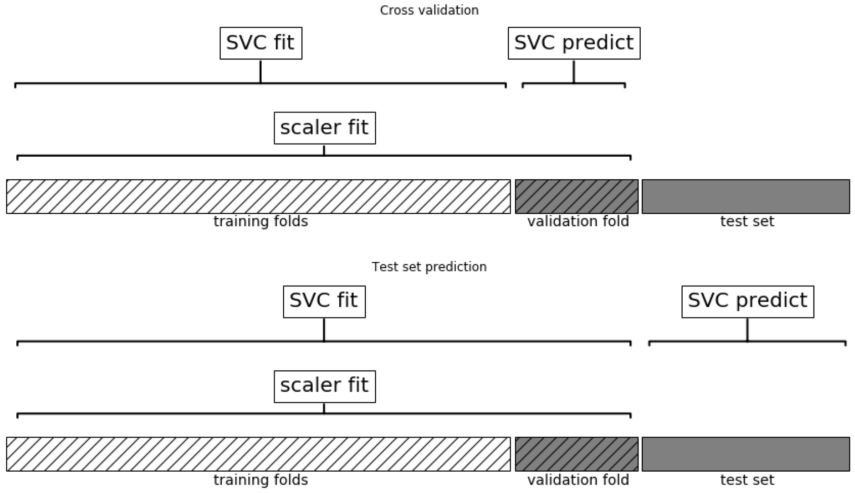
- Properly scaled: `fit` on training set, `transform` on training and test set
- Improperly scaled: `fit` and `transform` on the training and test data separately
    - Test data points nowhere near same training data points



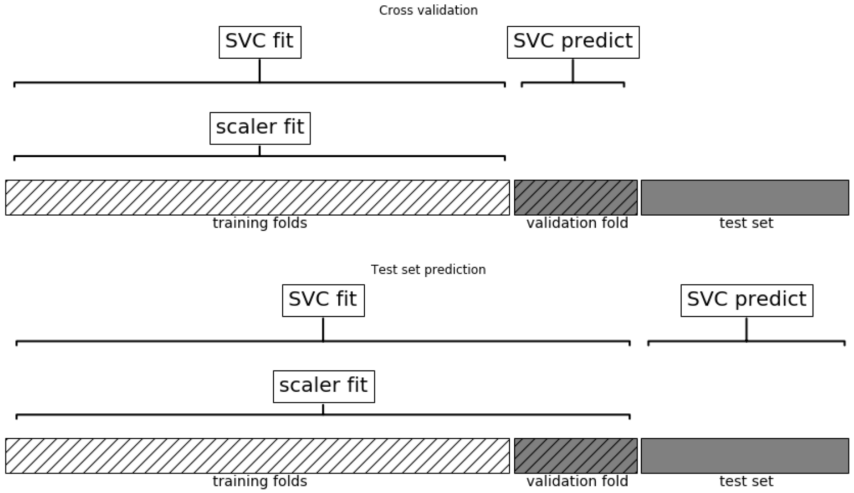
Data leakage example

- Cross-validation: training set is split into training and validation sets for model selection
- Incorrect: Scaler is fit on whole training set before doing cross-validation
    - Data leaks from validation folds into training folds, selected model may be optimistic
- Right: Scaler is fit on training folds only

Information Leak



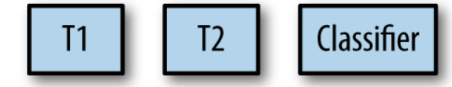
No Information leakage



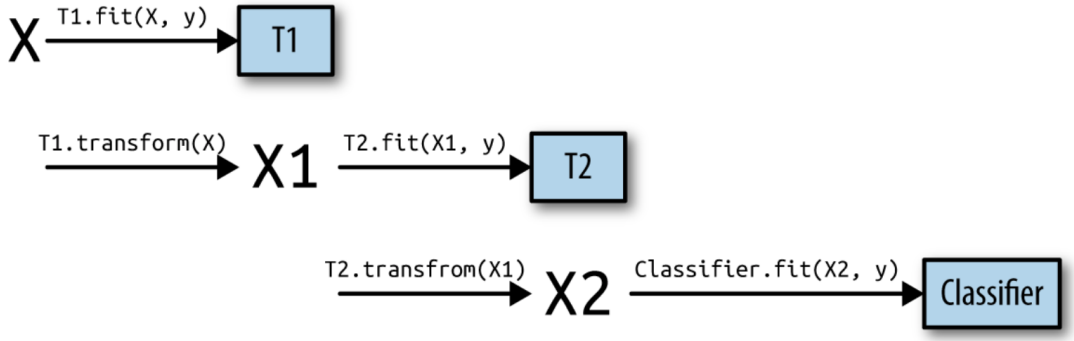
Pipelines

- A pipeline is a combination of data transformation and learning algorithms
- It has a `fit`, `predict`, and `score` method, just like any other learning algorithm
    - Ensures that data transformations are applied correctly

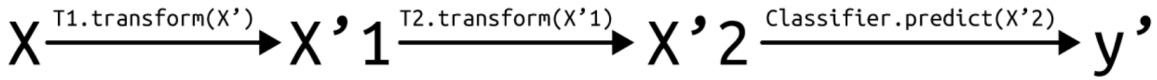
```
pipe = make_pipeline(T1(), T2(), Classifier())
```




```
pipe.fit(X, y)
```




```
pipe.predict(X')
```



In practice (scikit-learn)

- A `pipeline` combines multiple processing *steps* in a single estimator
- All but the last step should be data transformer (have a `transform` method)

```
# Make pipeline, step names will be 'minmaxscaler' and 'linearsvc'
pipe = make_pipeline(MinMaxScaler(), LinearSVC())
# Build pipeline with named steps
pipe = Pipeline([("scaler", MinMaxScaler()), ("svm", LinearSVC())])

# Correct fit and score
score = pipe.fit(X_train, y_train).score(X_test, y_test)
# Retrieve trained model by name
svm = pipe.named_steps["svm"]


# Correct cross-validation
scores = cross_val_score(pipe, X, y)
```

In practice (scikit-learn), continued

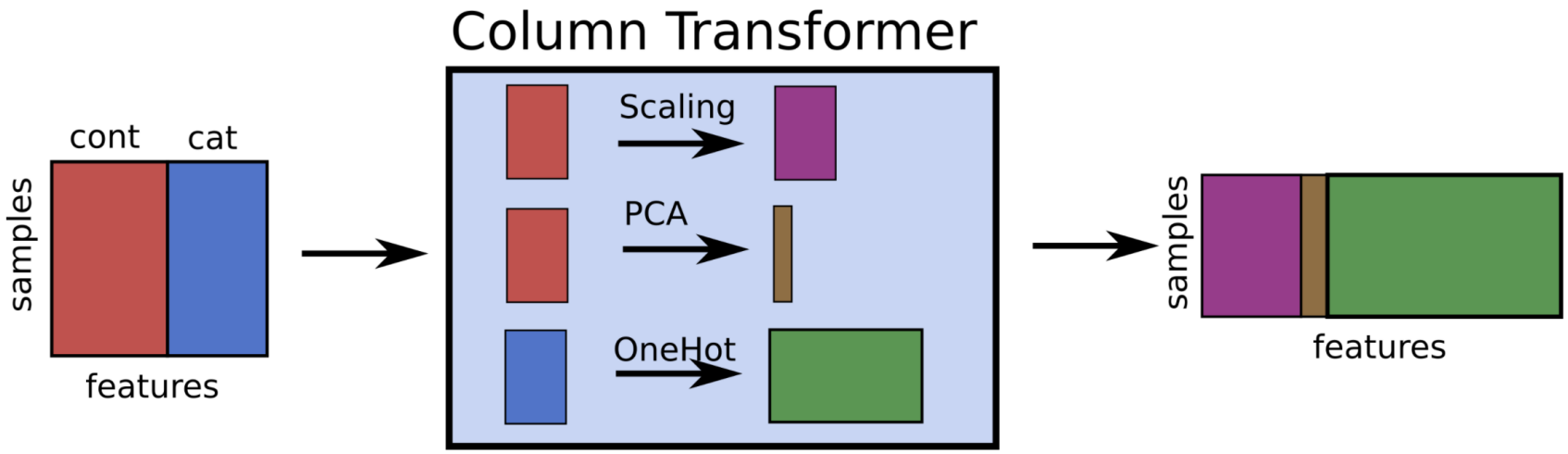
- If you want to apply different preprocessors to different columns, use `ColumnTransformer`
- If you want to merge pipelines, you can use `FeatureUnion` to concatenate columns

```
# 2 sub-pipelines, one for numeric features, other for categorical ones
numeric_pipe = make_pipeline(SimpleImputer(),StandardScaler())
categorical_pipe = make_pipeline(SimpleImputer(),OneHotEncoder())

# Using categorical pipe for features A,B,C, numeric pipe otherwise
preprocessor = make_column_transformer((categorical_pipe,["A","B","C"]),
                                       remainder=numeric_pipe)

# Combine with learning algorithm in another pipeline
pipe = make_pipeline(preprocess, LinearSVC())


# Feature union of PCA features and selected features
union = FeatureUnion([("pca", PCA()), ("selected", SelectKBest())])
pipe = make_pipeline(union, LinearSVC())
```

In practice (scikit-learn), continued

- `ColumnTransformer` concatenates features in order

```
pipe = make_column_transformer((StandardScaler(),numeric_features),
                               (PCA(),numeric_features),
                               (OneHotEncoder(),categorical_features))
```



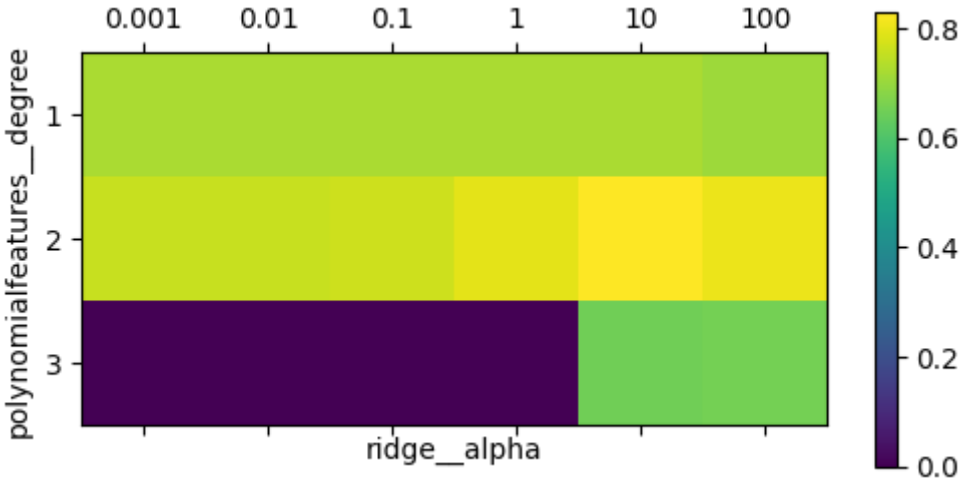
Model selection (scikit-learn)

- We can safely use pipelines in model selection (e.g. grid search)
- Use `'__'` to refer to the hyperparameters of a step, e.g. `svm__C`

```
# Correct grid search (can have hyperparameters of any step)
param_grid = {'svm__C': [0.001, 0.01],
              'svm__gamma': [0.001, 0.01, 0.1, 1, 10, 100]}
grid = GridSearchCV(pipe, param_grid=param_grid).fit(X,y)
# Best estimator is now the best pipeline
best_pipe = grid.best_estimator_

# Tune pipeline and evaluate on held-out test set
grid = GridSearchCV(pipe, param_grid=param_grid).fit(X_train,y_train)
grid.score(X_test,y_test)
```

Example: Tune multiple steps at once

```
pipe = make_pipeline(StandardScaler(),PolynomialFeatures(), Ridge())
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).fit(X_train, y_train)
```



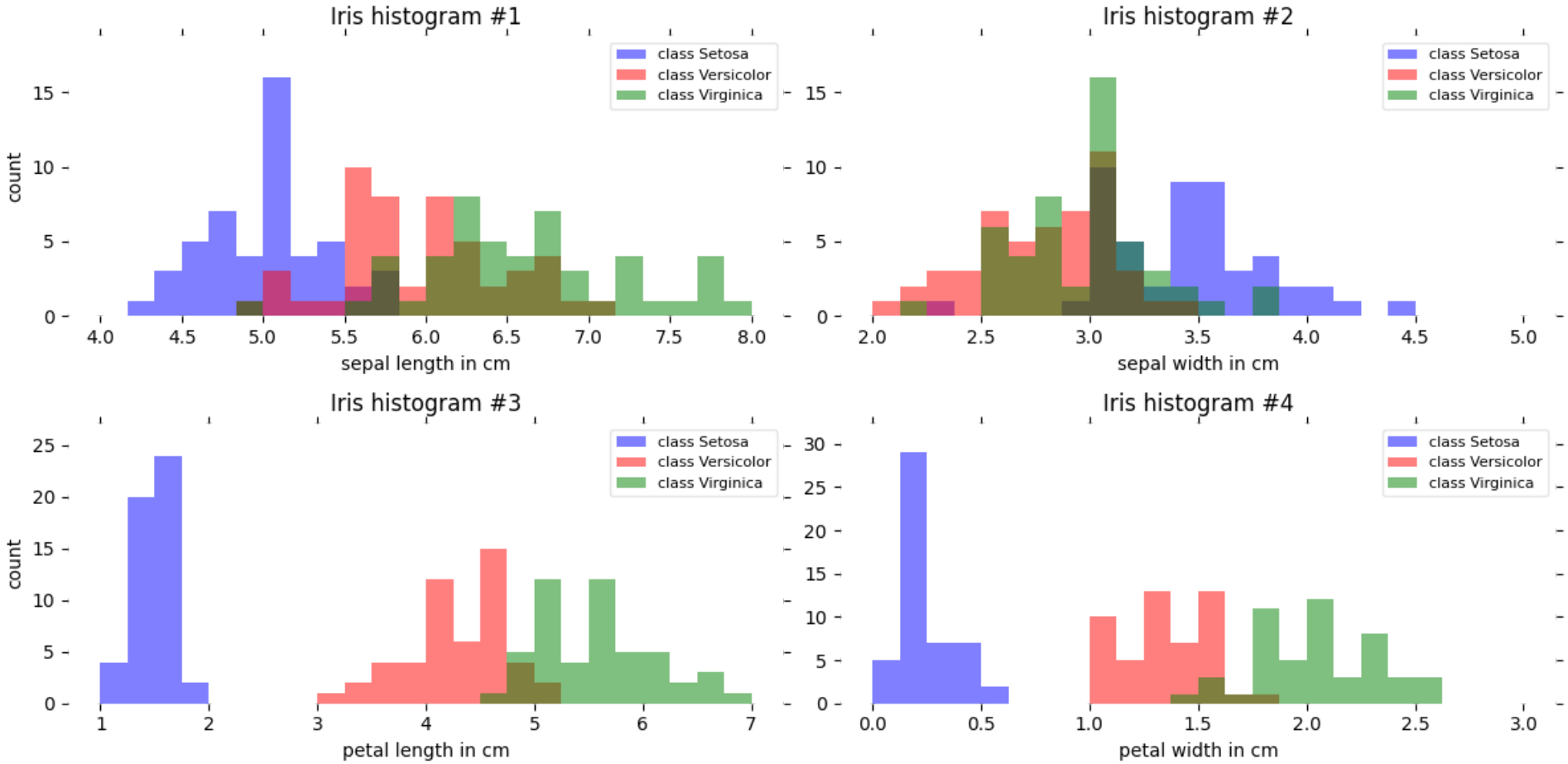
Automatic Feature Selection

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

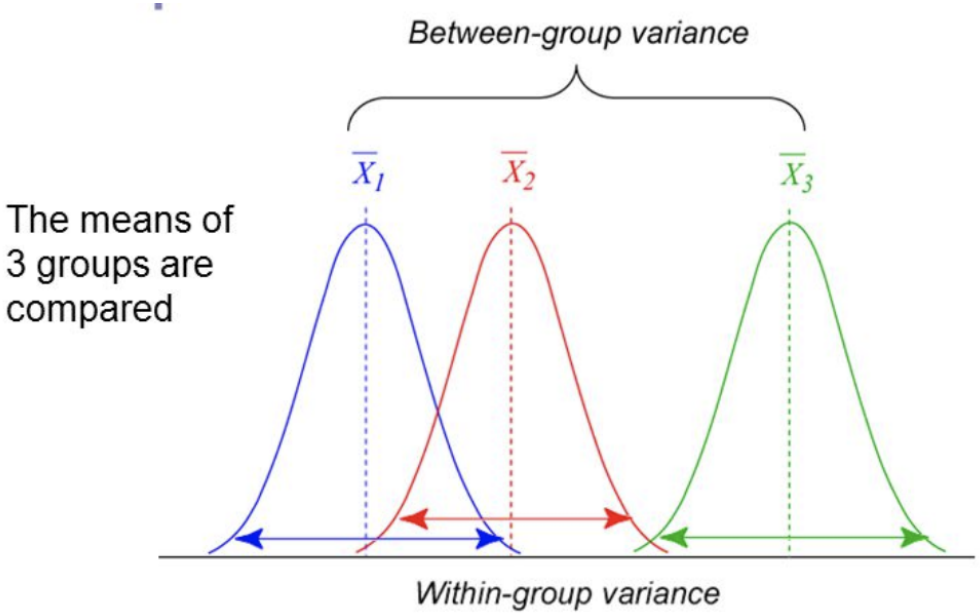
- Keep features for which there is a **statistically significant relationship** between it and the target.
- Consider each feature individually (univariate), independent of the model that you might want to apply afterwards.
- We can use different tests to measure how informative a feature is:

`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.

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

- Both the F-statistic and χ^2 methods use the p-value under the F- and χ^2 distribution, respectively.
- 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.



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

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

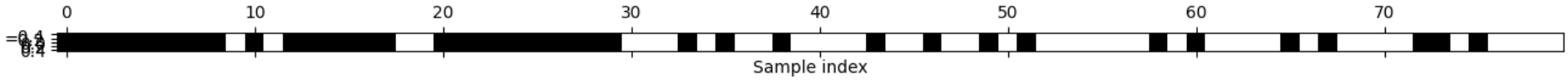
where O_{ij} is the number of observations of feature category i and class j, and E_{ij} is the expected number of observations of category i and class j if there was no relationship between the feature and the target (number of samples of category i * ratio of class j).

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.

```
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.

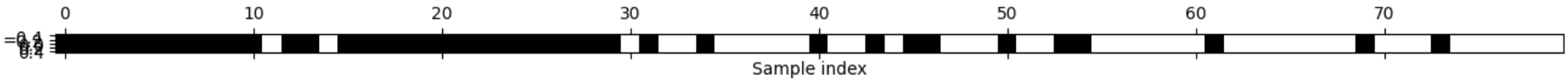
The supervised model that is used for feature selection doesn't need to be the same model that is used for the final supervised modeling, it only needs to be able to measure the (perceived) importance for each feature:

- 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
    - 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)
```

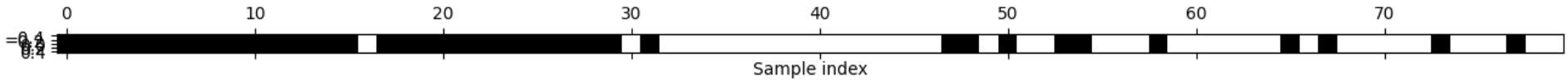
Vice versa, we could also ask it to iteratively add one feature at a time. This is called *forward selection*.

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



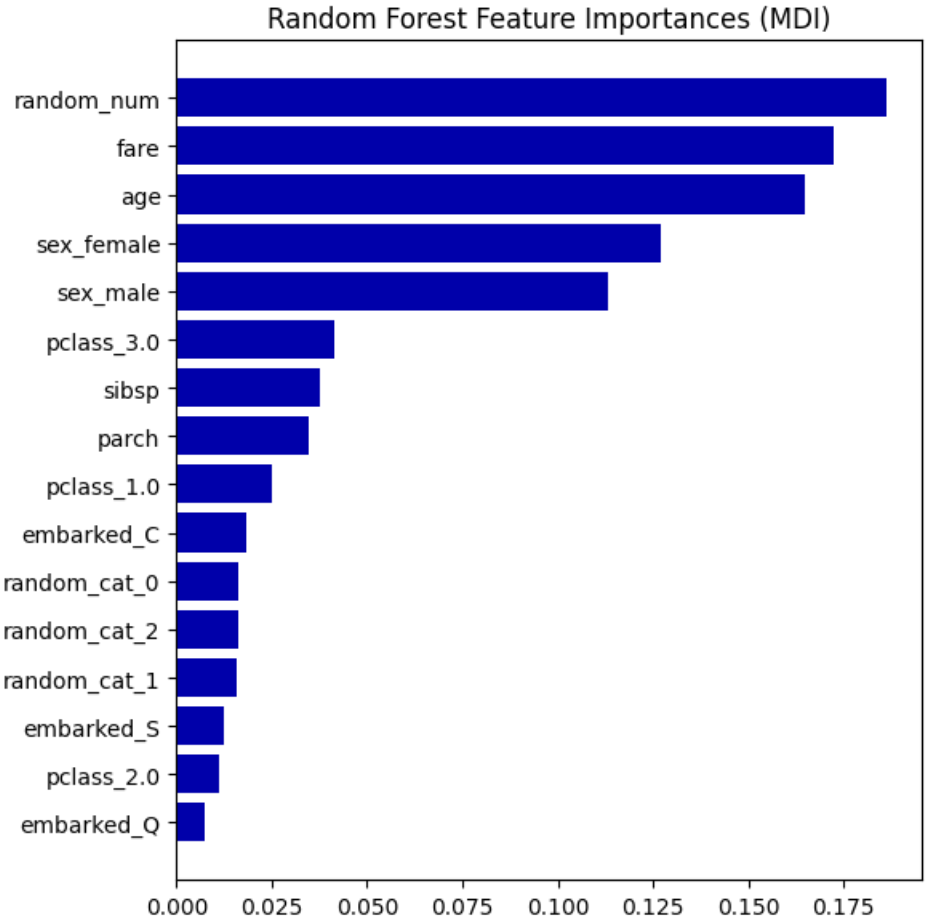

```
LogisticRegression Test score: 0.930
```

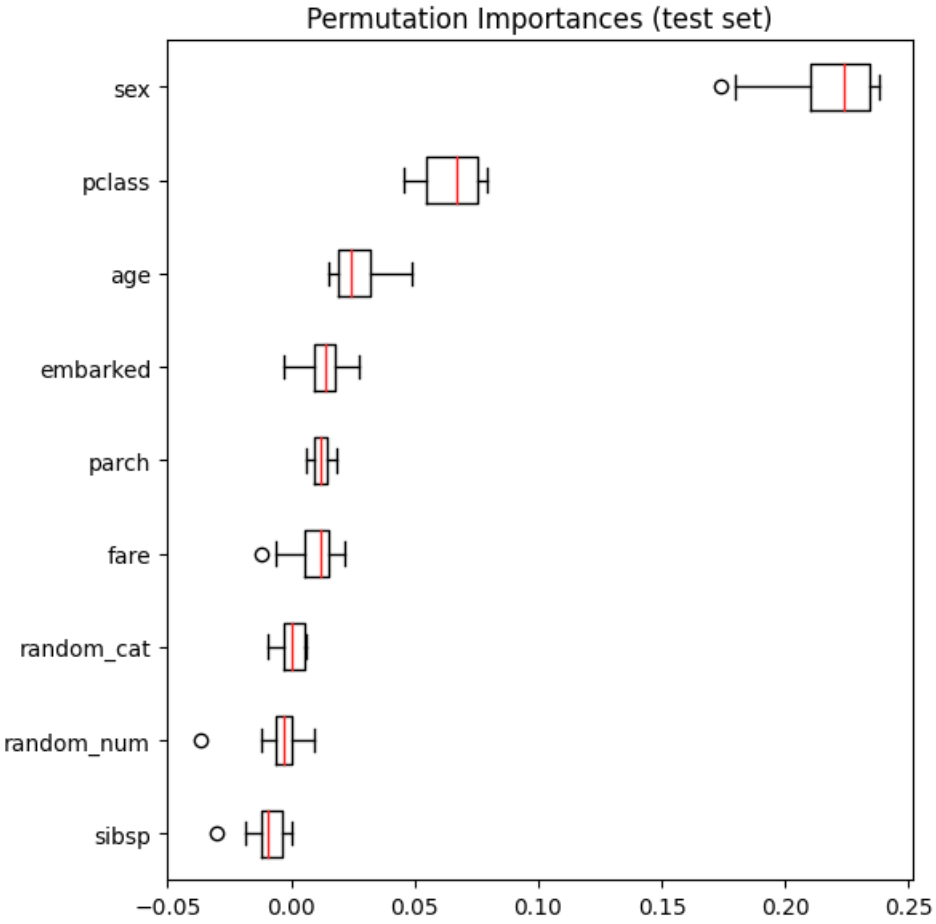
Permutation feature importance

- 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)

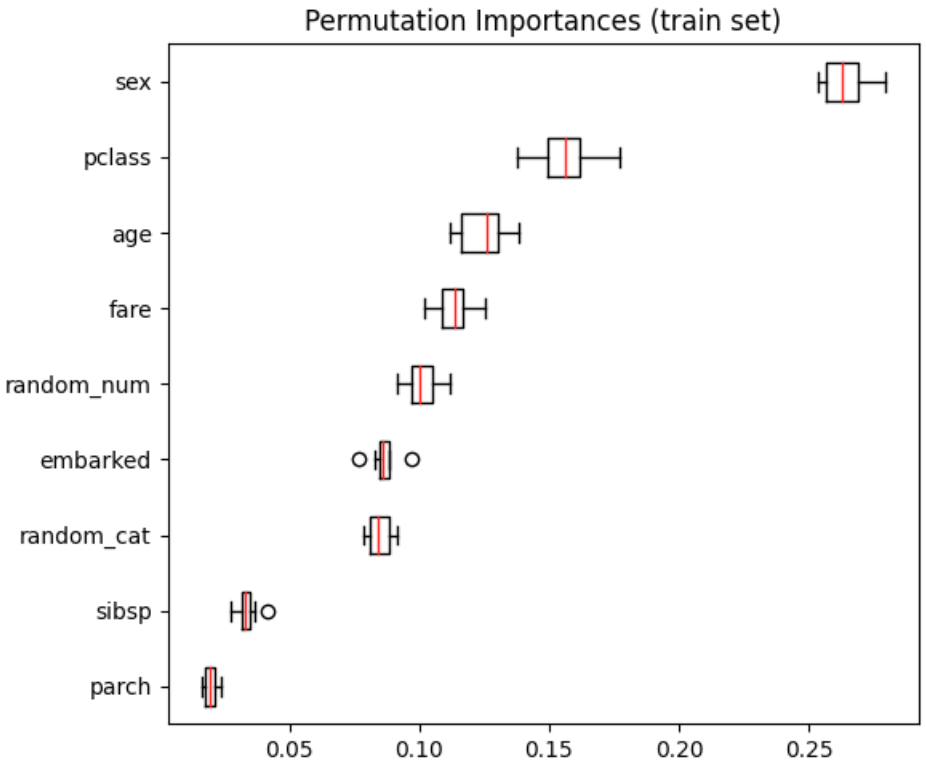
- We add a random feature as well: Random Forest deems it important!
- Low cardinality feature `sex` and `pclass` are actually more important
- Note: this requires the development version of sklearn

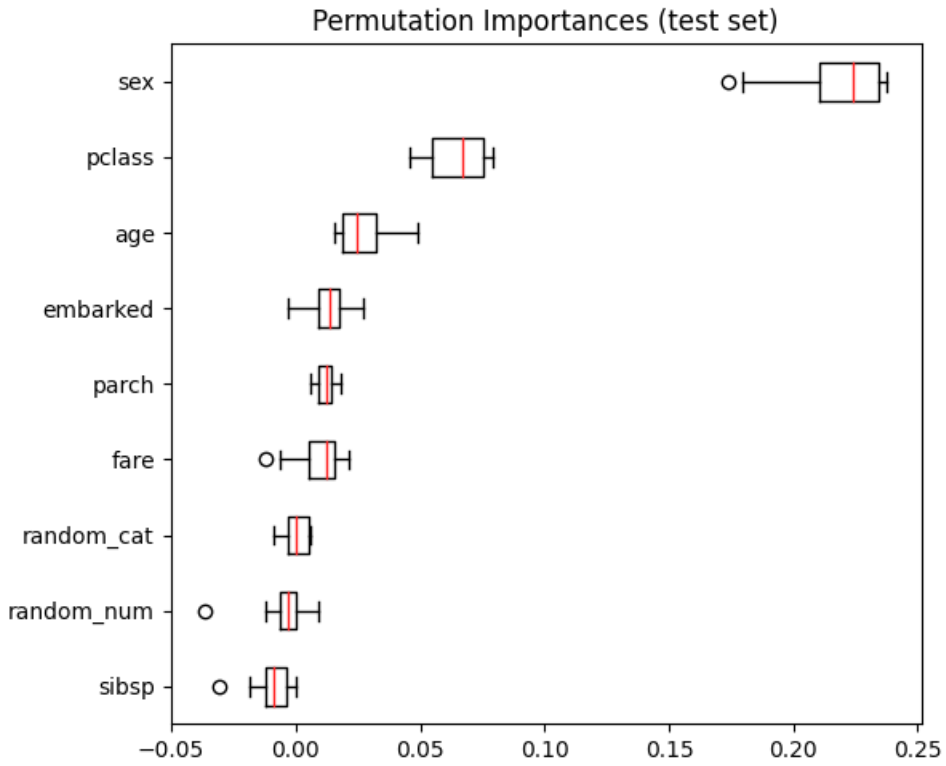




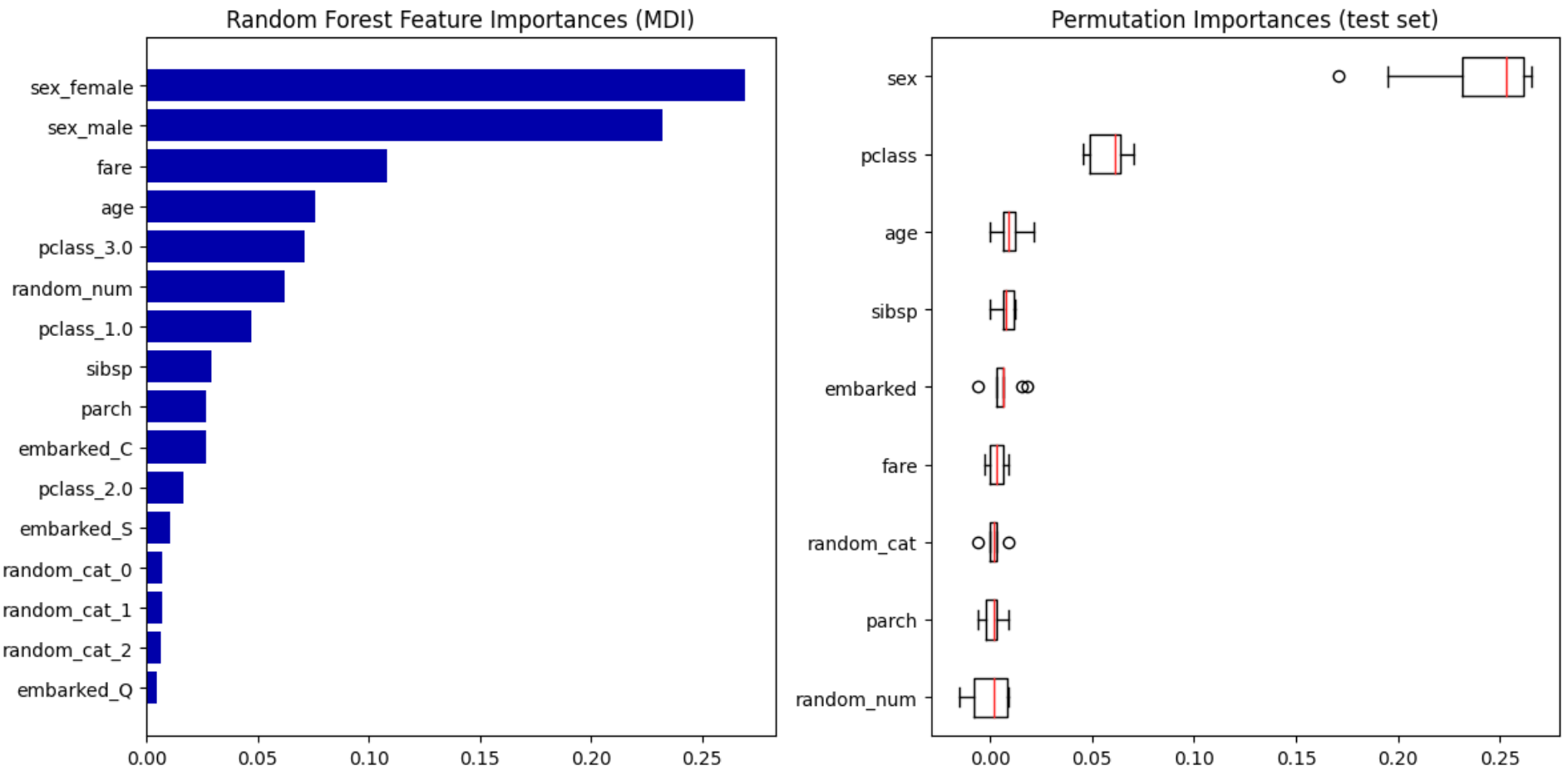
We can also compute the permutation importances on the training set.

- `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`):



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)

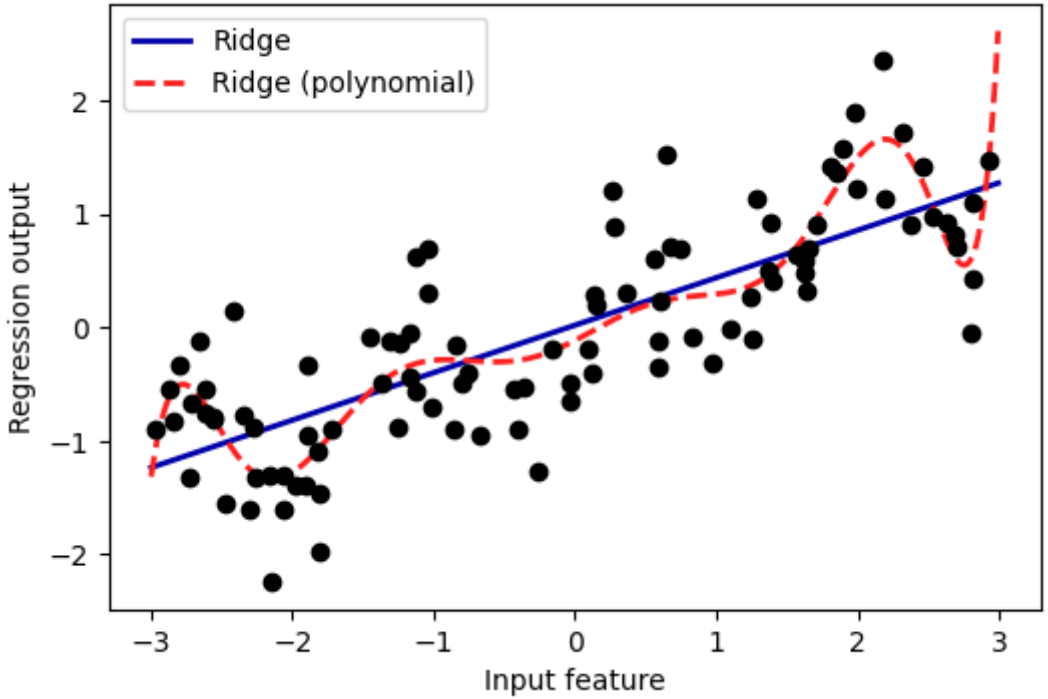
Feature Engineering

- Create new features based on existing ones
    - Polynomial features
    - Interaction features
    - Binning
- Mainly useful for simple models (e.g. linear models)
    - Other models can learn interations themselves
    - But may be slower, less robust than linear models

Polynomials

- Add all polynomials up to degree d and all products
    - Equivalent to polynomial basis expansions

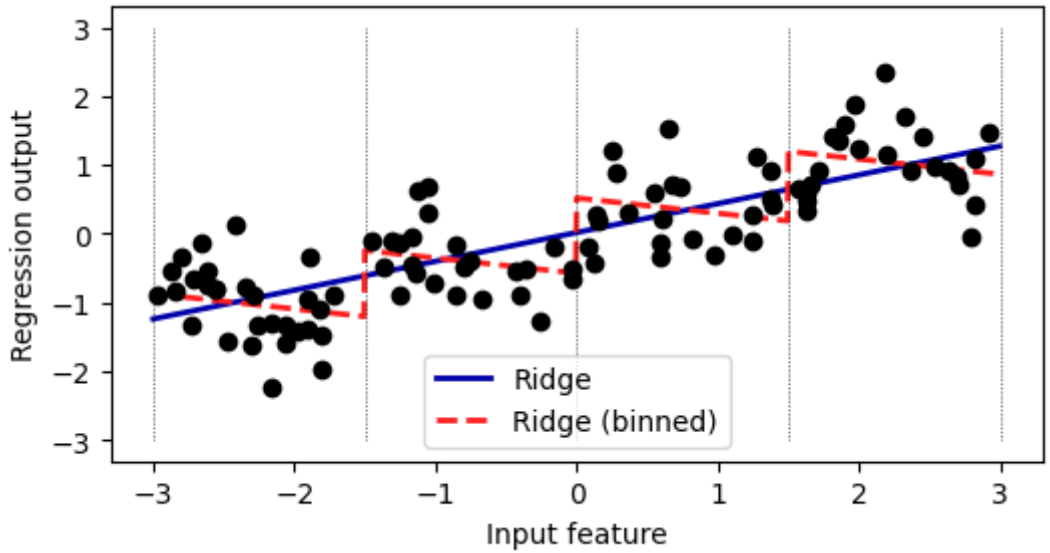
$$[1, x_1, \ldots, x_p] \rightarrow [1, x_1, \ldots, x_p, x_1^2, \ldots, x_p^2, \ldots, x_p^d, x_1 x_2, \ldots, x_{p-1} x_p]$$



Binning

- Partition numeric feature values into n intervals (bins)
- Create n new one-hot features, 1 if original value falls in corresponding bin
- Models different intervals differently (e.g. different age groups)

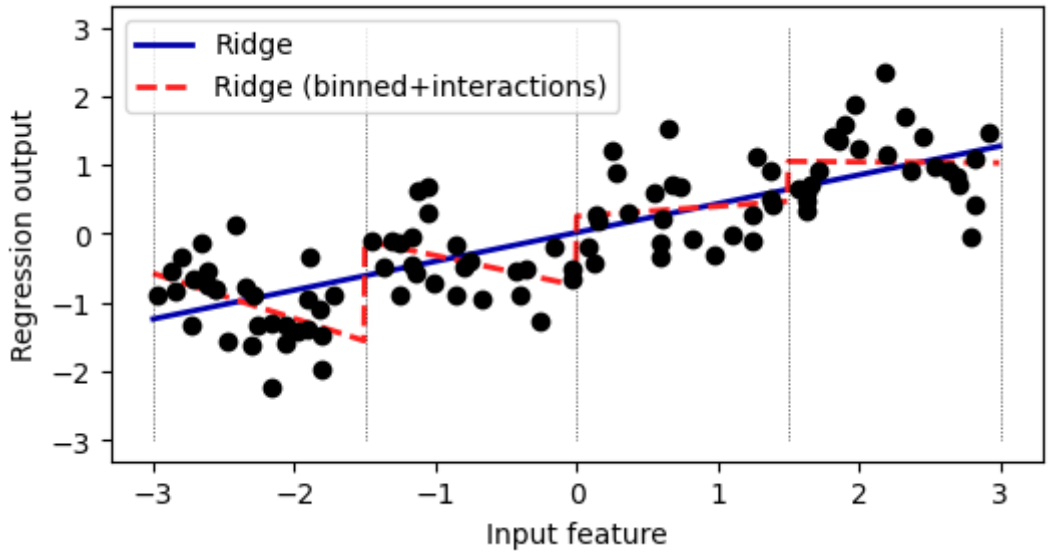
	orig	[-3.0,-1.5]	[-1.5,0.0]	[0.0,1.5]	[1.5,3.0]
0	-0.75	0.0	1.0	0.0	0.0
1	2.70	0.0	0.0	0.0	1.0
2	1.39	0.0	0.0	1.0	0.0



Binning + interaction features

- Add *interaction features* (or *product features*)
    - Product of the bin encoding and the original feature value
    - Learn different weights per bin

	orig	b0	b1	b2	b3	X*b0	X*b1	X*b2	X*b3
0	-0.75	0.0	1.0	0.0	0.0	-0.0	-0.75	-0.00	-0.0
1	2.70	0.0	0.0	0.0	1.0	0.0	0.00	0.00	2.7
2	1.39	0.0	0.0	1.0	0.0	0.0	0.00	1.39	0.0



Categorical feature interactions

- One-hot-encode categorical feature
- Multiply every one-hot-encoded column with every numeric feature
- Allows to built different submodels for different categories

	gender	age	pageviews	time_online
0	M	14	70	269
1	F	16	12	1522
2	M	12	42	235

	age_M	pageviews_M	time_online_M	gender_M_M	age_F	pageviews_F	time_online_F	gender_F_F
0	14	70	269	1	0	0	0	0
1	0	0	0	0	16	12	1522	1
2	12	42	235	1	0	0	0	0

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]]
```

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

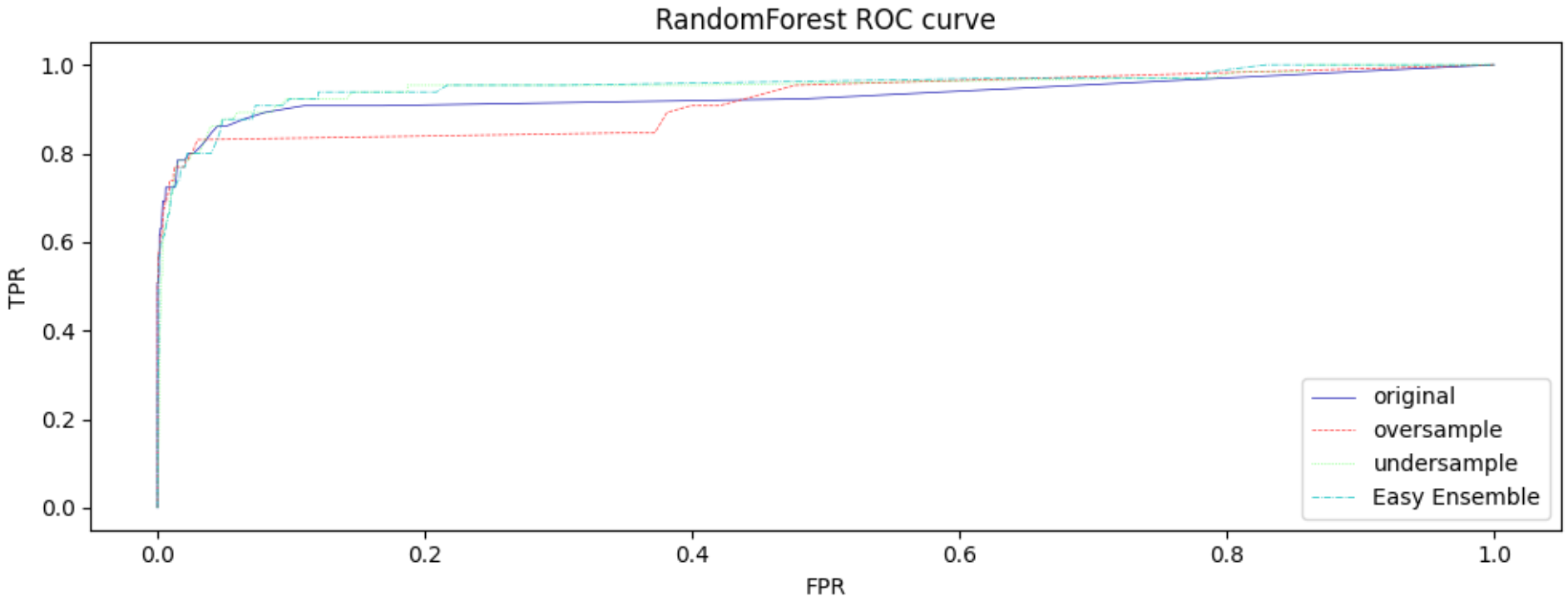
Handling imbalanced data

- 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.

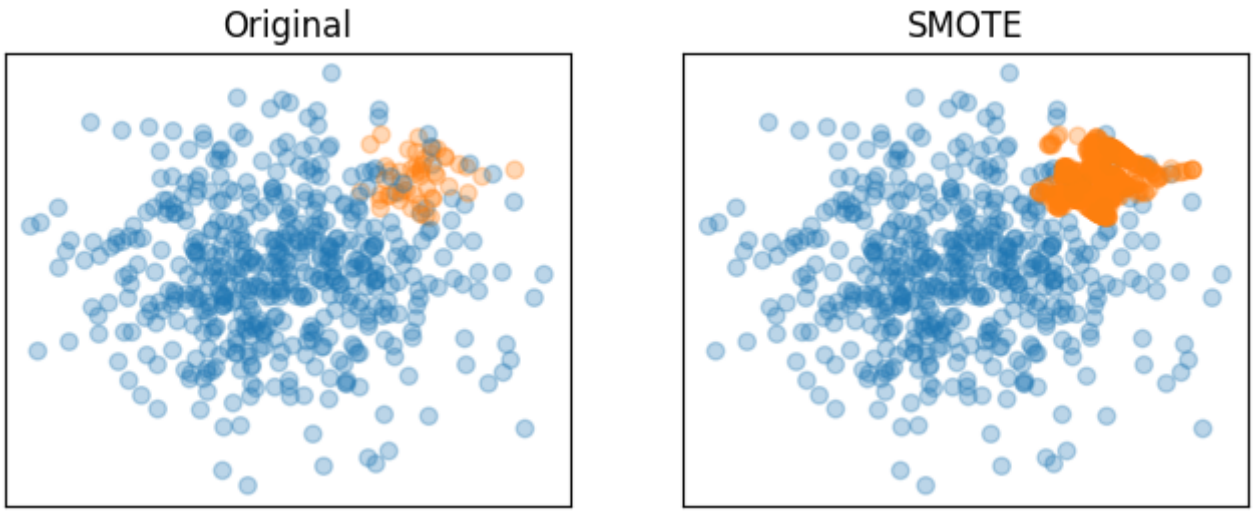
Handling imbalanced data in practice

- http://imbalanced-learn.org
- 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


RandomForest ROC curve
TPR
FPR
0.0
0.2
0.4
0.6
0.8
1.0
original
oversample
undersample
Easy Ensemble

SMOTE



Summary

- Transforming the features can drastically impact performance
    - Constructing new features makes linear models more powerful
    - Selecting features can reduce overfitting and improve performance
    - Scaling is important for many distance-based methods (e.g. kNN, SVM)
- 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.