```
[1]: # Global imports and settings
     from preamble import *
     %matplotlib inline
     plt.rcParams['savefig.dpi'] = 120 # Use 300 for PDF, 100 for slides
     #InteractiveShell.ast_node_interactivity = "all"
     HTML('''<style>html, body{overflow-y: visible !important} .CodeMirror{min-w
```

```
<IPython.core.display.HTML object>
```

Machine learning pipelines

Preprocessing

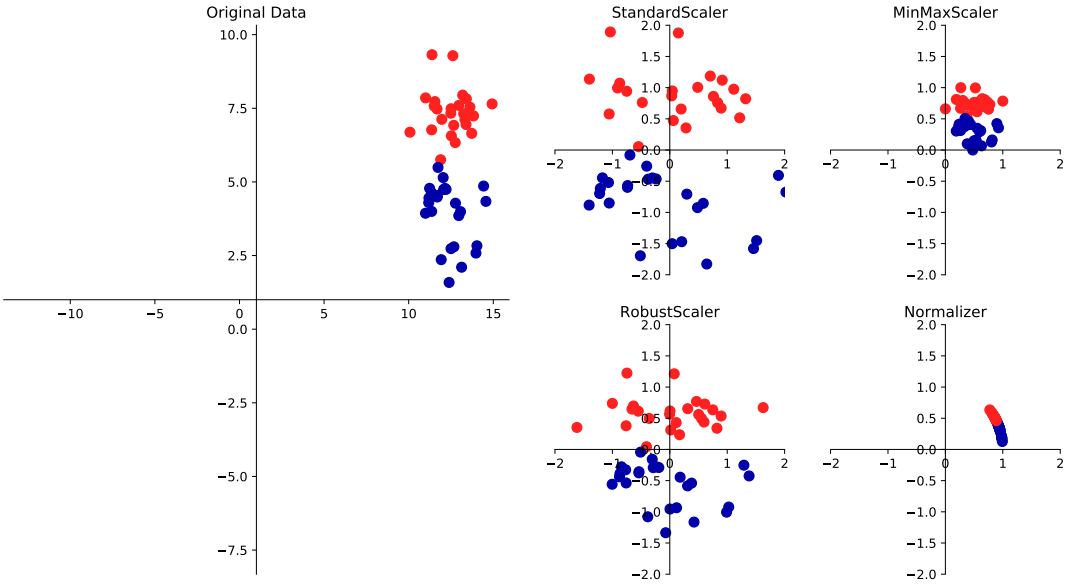
- Many of the algorithms that we've seen are greatly affected by *how* you represent the training data
- Scaling, numeric/categorical values, missing values, feature selection/construction
- We typically need chain together different algorithms
  - Many preprocessing steps
  - Possibly many models
- This is called a *pipeline* (or *workflow*)

Outline: * Examples of which preprocessing steps are best combined with learning algorithm * How to apply preprocessing techniques * How to build pipelines * How to design and optimize pipelines * Some practical advice

Example: Scaling

- SVMs, kNN, neural networks are very sensitive to the scaling of the data
- Solution: simple per-feature rescaling and shift of the data
- There are several ways to do this:
  - `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

```
[2]: mglearn.plots.plot_scaling()
```



Applying data transformations

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

```
[3]: from sklearn.datasets import load_breast_cancer
     from sklearn.model_selection import train_test_split
     cancer = load_breast_cancer()

     X_train, X_test, y_train, y_test = train_test_split(cancer.data, cancer.tar
                                                         random_state=1)
```

- 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

```
[4]: from sklearn.preprocessing import MinMaxScaler

     scaler = MinMaxScaler()
     scaler.fit(X_train)
```

```
MinMaxScaler(copy=True, feature_range=(0, 1))
```

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

```
[5]: # transform data
     X_train_scaled = scaler.transform(X_train)
     # print dataset properties before and after scaling
     print("per-feature minimum before scaling:\n {}".format(X_train.min(axis=0)
     print("per-feature maximum before scaling:\n {}".format(X_train.max(axis=0)
     print("per-feature minimum after scaling:\n {}".format(
```

```
            X_train_scaled.min(axis=0)))
    print("per-feature maximum after scaling:\n {}".format(
            X_train_scaled.max(axis=0)))
```

```
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     0.427
     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      0.291
     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.]
```

```
[6]: # transform test data
     X_test_scaled = scaler.transform(X_test)
     # print test data properties after scaling
     print("per-feature minimum after scaling:\n{}".format(X_test_scaled.min(axi
     print("per-feature maximum after scaling:\n{}".format(X_test_scaled.max(axi
```

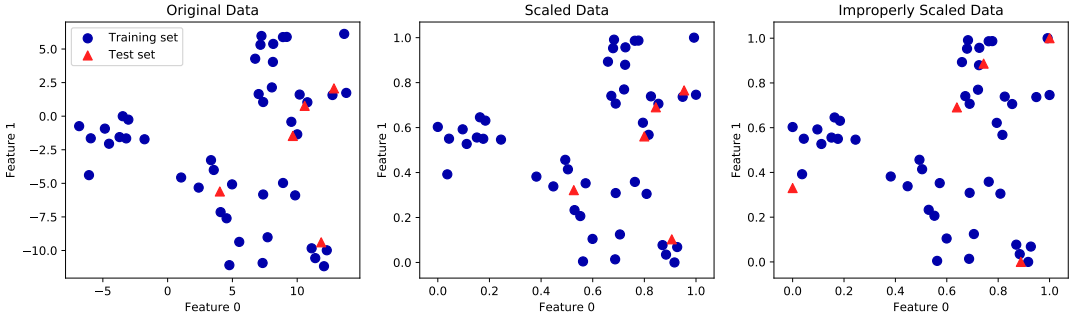
```
per-feature minimum after scaling:
[ 0.034  0.023  0.031  0.011  0.141  0.044  0.     0.     0.154 -0.006
 -0.001  0.006  0.004  0.001  0.039  0.011  0.     0.    -0.032  0.007
  0.027  0.058  0.02   0.009  0.109  0.026  0.     0.    -0.    -0.002]
per-feature maximum after scaling:
[ 0.958  0.815  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

What happens if we transform the test data separately? * 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

```
[7]: from sklearn.datasets import make_blobs
     # make synthetic data
     X, _ = make_blobs(n_samples=50, centers=5, random_state=4, cluster_std=2)
     # split it into training and test set
```

```
X_train, X_test = train_test_split(X, random_state=5, test_size=.1)

# plot the training and test set
fig, axes = plt.subplots(1, 3, figsize=(13, 4))
axes[0].scatter(X_train[:, 0], X_train[:, 1],
                c=mglearn.cm2(0), label="Training set", s=60)
axes[0].scatter(X_test[:, 0], X_test[:, 1], marker='^',
                c=mglearn.cm2(1), label="Test set", s=60)
axes[0].legend(loc='upper left')
axes[0].set_title("Original Data")

# scale the data using MinMaxScaler
scaler = MinMaxScaler()
scaler.fit(X_train)
X_train_scaled = scaler.transform(X_train)
X_test_scaled = scaler.transform(X_test)

# visualize the properly scaled data
axes[1].scatter(X_train_scaled[:, 0], X_train_scaled[:, 1],
                c=mglearn.cm2(0), label="Training set", s=60)
axes[1].scatter(X_test_scaled[:, 0], X_test_scaled[:, 1], marker='^',
                c=mglearn.cm2(1), label="Test set", s=60)
axes[1].set_title("Scaled Data")

# rescale the test set separately
# so that test set min is 0 and test set max is 1
# DO NOT DO THIS! For illustration purposes only
test_scaler = MinMaxScaler()
test_scaler.fit(X_test)
X_test_scaled_badly = test_scaler.transform(X_test)

# visualize wrongly scaled data
axes[2].scatter(X_train_scaled[:, 0], X_train_scaled[:, 1],
                c=mglearn.cm2(0), label="training set", s=60)
axes[2].scatter(X_test_scaled_badly[:, 0], X_test_scaled_badly[:, 1],
                marker='^', c=mglearn.cm2(1), label="test set", s=60)
axes[2].set_title("Improperly Scaled Data")

for ax in axes:
    ax.set_xlabel("Feature 0")
    ax.set_ylabel("Feature 1")
fig.tight_layout()
```



- 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

```
[8]: from sklearn.preprocessing import StandardScaler
     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 on SVMs?

- First, we train the SVM without scaling

```
[9]: from sklearn.svm import SVC

     X_train, X_test, y_train, y_test = train_test_split(cancer.data, cancer.tar
                                                         random_state=0)

     svm = SVC()
     svm.fit(X_train, y_train)
     print("Test set accuracy: {:.2f}".format(svm.score(X_test, y_test)))
```

```
Test set accuracy: 0.63
```

- With scaling, we get a much better model

```
[10]: # preprocessing using 0-1 scaling
      scaler = MinMaxScaler()
      scaler.fit(X_train)
      X_train_scaled = scaler.transform(X_train)
      X_test_scaled = scaler.transform(X_test)

      # learning an SVM on the scaled training data
      svm.fit(X_train_scaled, y_train)
      # scoring on the scaled test set
      print("Scaled test set accuracy: {:.2f}".format(svm.score(X_test_scaled, y
```

```
Scaled test set accuracy: 0.95
```

Parameter Selection with Preprocessing

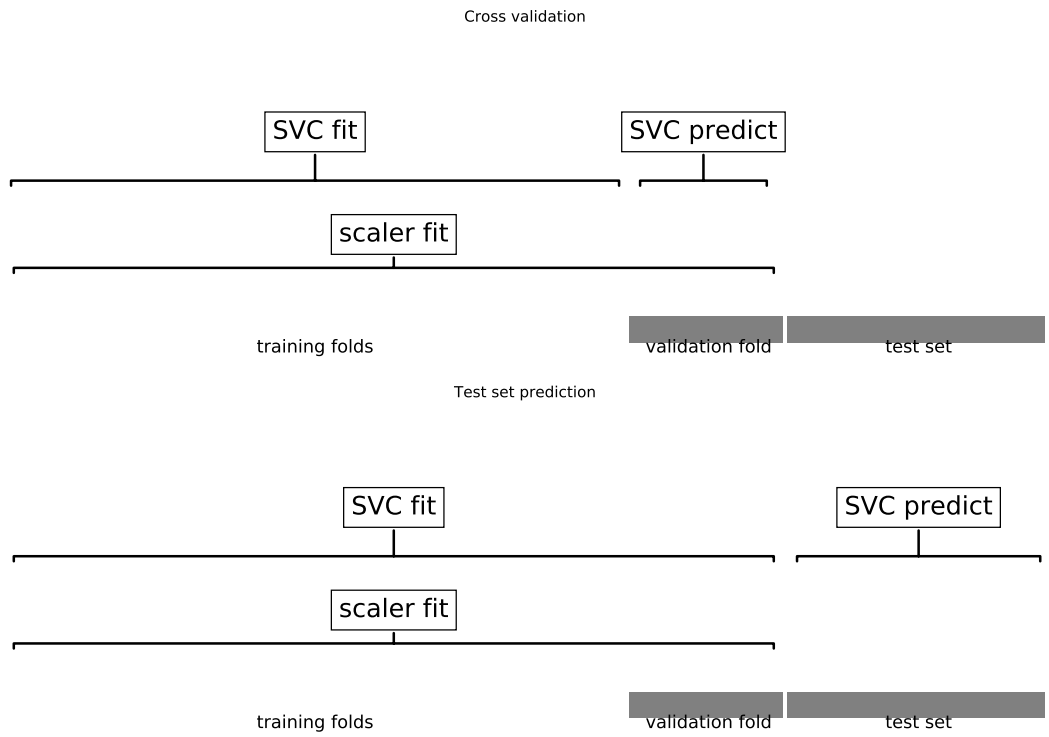
- If we also want to optimize our SVM's hyperparameters, things get even more complicated
- Indeed, when we `fit` the preprocessor (`MinMaxScaler`), we used *all* the training data.
- The cross-validation splits in GridSearchCV will have training sets preprocessed with information from the test sets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

```
[11]: from sklearn.model_selection import GridSearchCV
      # illustration purposes only, don't use this code
      param_grid = {'C': [0.001, 0.01, 0.1, 1, 10, 100],
                    'gamma': [0.001, 0.01, 0.1, 1, 10, 100]}
      grid = GridSearchCV(SVC(), param_grid=param_grid, cv=5)
      grid.fit(X_train_scaled, y_train)
      print("Best cross-validation accuracy: {:.2f}".format(grid.best_score_))
      print("Best set score: {:.2f}".format(grid.score(X_test_scaled, y_test)))
      print("Best parameters: ", grid.best_params_)
```

```
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.

```
[12]: mglearn.plots.plot_improper_processing()
```



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` + `SVC` pipeline:

```
[13]: from sklearn.pipeline import Pipeline
      pipe = Pipeline([("scaler", MinMaxScaler()), ("svm", SVC())])
      pipe.fit(X_train, y_train)
      print("Test score: {:.2f}".format(pipe.score(X_test, y_test)))
```

```
Test score: 0.95
```

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

```
[14]: print("SVM component: {}".format(pipe.steps[1][1]))
```

```
SVM component: SVC(C=1.0, cache_size=200, class_weight=None, coef0=0.0,
  decision_function_shape=None, degree=3, gamma='auto', kernel='rbf',
  max_iter=-1, probability=False, random_state=None, shrinking=True,
  tol=0.001, verbose=False)
```

- Or we can use the `named_steps` dictionary

```
[15]: print("SVM component: {}".format(pipe.named_steps['svm']))
```

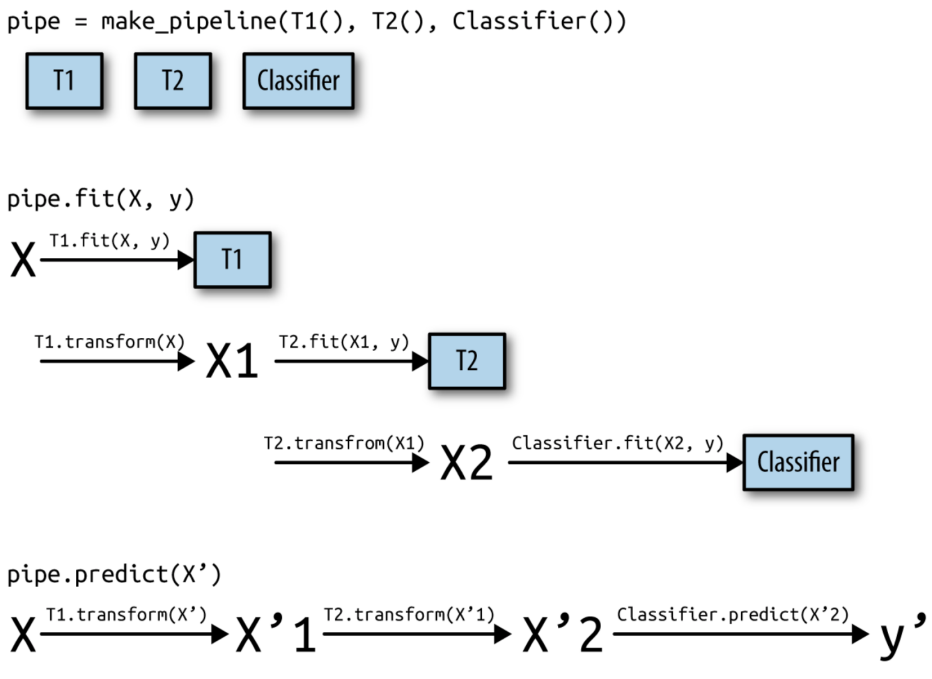
```
SVM component: SVC(C=1.0, cache_size=200, class_weight=None, coef0=0.0,
  decision_function_shape=None, degree=3, gamma='auto', kernel='rbf',
  max_iter=-1, probability=False, random_state=None, shrinking=True,
  tol=0.001, verbose=False)
```

Visualization of a pipeline `fit` and `predict`

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

```
[16]: from sklearn.pipeline import make_pipeline
      # standard syntax
      pipe_long = Pipeline([("scaler", MinMaxScaler()), ("svm", SVC(C=100))])
      # abbreviated syntax
      pipe_short = make_pipeline(MinMaxScaler(), SVC(C=100))
      print("Pipeline steps:\n{}".format(pipe_short.steps))
```

```
Pipeline steps:
[('minmaxscaler', MinMaxScaler(copy=True, feature_range=(0, 1))), ('svc', SVC(C=
  decision_function_shape=None, degree=3, gamma='auto', kernel='rbf',
  max_iter=-1, probability=False, random_state=None, shrinking=True,
  tol=0.001, verbose=False))]
```



pipeline_illustration

Pipelines and OpenML

- You can share pipelines (and their results) on OpenML

```
[44]: from sklearn import pipeline, ensemble, preprocessing
      from openml import tasks, runs, datasets
      task = tasks.get_task(59)
      pipe = pipeline.Pipeline(steps=[
                  ('Imputer', preprocessing.Imputer(strategy='median')),
                  ('OneHotEncoder', preprocessing.OneHotEncoder(sparse=False, ha
                  ('Classifier', ensemble.RandomForestClassifier())
                ])
      run = runs.run_task(task, pipe)
      myrun = run.publish()
      print("Uploaded to http://www.openml.org/r/" + str(myrun.run_id))
```

```
Uploaded to http://www.openml.org/r/1849085
```

Other transformation techniques

We've covered many other transformation techniques before. These can all be used in pipelines. * `OneHotEncoder`: convert categorical to numeric features * `np.digitize`: discretization (binning) of numeric features * `PolynomialFeatures`: construct all interactions with polynomials up to a given degree * `SelectPercentile`: use ANOVA to select most informative features * `SelectFromModel(RandomForestClassifier())`: model-based feature selection * `RFE`: recursive feature elimination * `VarianceThreshold`: removes low-variance (e.g. constant) features

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

```
[58]: from sklearn.preprocessing import Imputer
      X_train = [[1, 2], [np.nan, 3], [7, 6]];
      imp = Imputer(missing_values='NaN', strategy='mean', axis=0)
      imp.fit(X_train)
      X = [[np.nan, 2], [6, np.nan], [7, 6]]
      print("Missing data: {}".format(X) )
      print("Imputed data:\n {}".format(imp.transform(X))  )
```

```
Missing data: [[nan, 2], [6, nan], [7, 6]]
Imputed data:
 [[ 4.     2.   ]
 [ 6.     3.667]
 [ 7.     6.   ]]
```

Dimensionality reduction techniques

- Many techniques
  - `decomposition.PCA` (Principal Component Analysis)
  - `manifold.MDS` (Multi-Dimensional Scaling)
  - `manifold.Isomap`
  - `random_projection.GaussianRandomProjection`
  - `random_projection.johnson_lindenstrauss_min_dim`
  - ...
- See coming lectures
- Reducing the number of features greatly helps distance-based algorithms (kNN, clustering,...)
  - Curse of dimensionality (Bellman's curse): for every new feature, we need exponentially more data
- Very useful in their own right
- Note: not all dimensionality reduction techniques can be used as a transformer.
  - They have a `fit`, but no `transform` method, and can't be applied on test data.

Using Pipelines in Grid-searches

- Let's return to our scaler + SVM pipeline
- We can now 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`

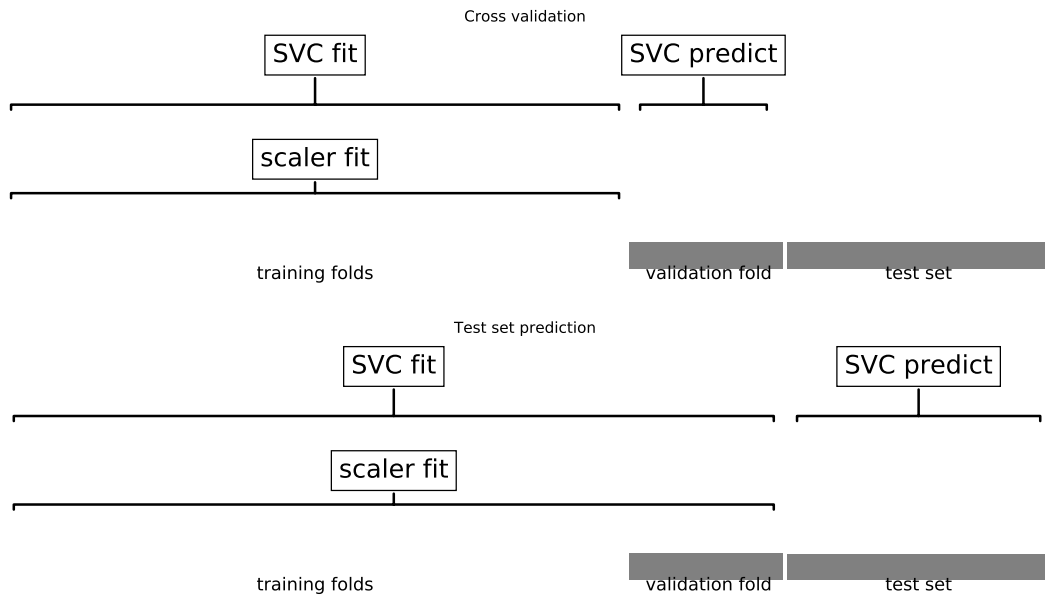
```
[17]: param_grid = {'svm__C': [0.001, 0.01, 0.1, 1, 10, 100],
                    'svm__gamma': [0.001, 0.01, 0.1, 1, 10, 100]}
```

```
[18]: grid = GridSearchCV(pipe, param_grid=param_grid, cv=5)
      grid.fit(X_train, y_train)
      print("Best cross-validation accuracy: {:.2f}".format(grid.best_score_))
      print("Test set score: {:.2f}".format(grid.score(X_test, y_test)))
      print("Best parameters: {}".format(grid.best_params_))
```

```
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.

```
[19]: mglearn.plots.plot_proper_processing()
```



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

```
[20]: print("Best estimator:\n{}".format(grid.best_estimator_))
```

```
Best estimator:
Pipeline(steps=[('scaler', MinMaxScaler(copy=True, feature_range=(0, 1))), ('svm
  decision_function_shape=None, 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

```
[21]: # Get the SVM
      print("SVM step:\n{}".format(
            grid.best_estimator_.named_steps["svm"]))
```

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

```
[22]: # Get the SVM dual coefficients (support vector weights)
      print("SVM support vector coefficients:\n{}".format(
            grid.best_estimator_.named_steps["svm"].dual_coef_))
```

```
SVM support vector coefficients:
[[-1.    -1.    -1.    -1.    -1.    -1.    -1.    -1.    -0.822 -0.609
  -0.93  -1.    -0.665 -1.    -0.793 -1.    -1.    -1.    -1.    -1.    -1.
  -0.63  -1.    -1.    -1.    -0.166 -0.575 -1.    -0.119 -1.    -0.38  -1.
  -0.55  -0.048 -0.606 -1.    -0.071 -1.    -0.126 -0.106 -0.285 -1.    -1.
  -1.    -0.46   1.     0.932  1.     1.     0.901  1.     0.911  1.     1.
   1.     0.653  1.     1.     0.953  0.577  1.     1.     1.     1.     1.
   0.659  1.     1.     0.757  1.     1.     1.     1.     1.     1.     1.
   0.096  0.274  0.329  1.     1.     1.     0.491  1.     0.41 ]]
```

Information leakage

- See 'Elements of statistical learning' for a great example of data leakage
- Consider a synthetic regression task with 100 samples and 10,000 features with data and labels independently sampled from a Gaussian distribution
  - Hence, there should be no relation between the data `X` and target `y`

```
[23]: rnd = np.random.RandomState(seed=0)
      X = rnd.normal(size=(100, 10000))
      y = rnd.normal(size=(100,))
```

- First, we select the 5% most informative features with `SelectPercentile`, and then evaluate a `Ridge` regressor

```
[24]: from sklearn.feature_selection import SelectPercentile, f_regression

      select = SelectPercentile(score_func=f_regression, percentile=5).fit(X, y)
      X_selected = select.transform(X)
      print("X_selected.shape: {}".format(X_selected.shape))
```

```
X_selected.shape: (100, 500)
```

```
[25]: from sklearn.model_selection import cross_val_score
      from sklearn.linear_model import Ridge
      print("Cross-validation accuracy (cv only on ridge): {:.2f}".format(
            np.mean(cross_val_score(Ridge(), X_selected, y, cv=5))))
```

```
Cross-validation accuracy (cv only on ridge): 0.91
```

- The R^2 performance is 0.91 (a very good model). This can't be right given that our data is random.
- Our feature selection picked out some of the random features which (by chance) correlated with the random target.
- Because we selected the features *outside* of the cross-validation, it could find features that are correlated on the samples in the test folds.
- Hence, information leaked from the test set into the training set (through the selection of features).

- Now, let's do a proper cross-validation using a pipeline:

```
[26]: pipe = Pipeline([("select", SelectPercentile(score_func=f_regression,
                                                   percentile=5)),
                       ("ridge", Ridge())])
      print("Cross-validation accuracy (pipeline): {:.2f}".format(
            np.mean(cross_val_score(pipe, X, y, cv=5))))
```

```
Cross-validation accuracy (pipeline): -0.25
```

- We get a negative R^2 score (a very poor model), as expected.
- The feature selection now happens *inside* the cross-validation loop, using only the training folds.
- It will still find features that correlate with the labels in the training data, but not with those in the test data.

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*

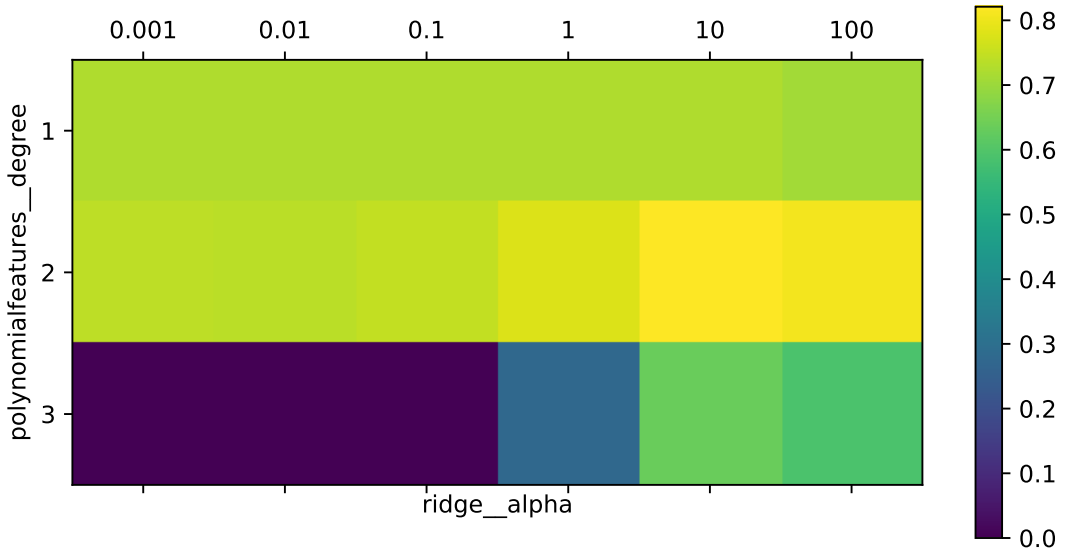
```
[38]: from sklearn.datasets import load_boston
      boston = load_boston()
      X_train, X_test, y_train, y_test = train_test_split(boston.data, boston.ta
                                                          random_state=0)
      from sklearn.preprocessing import PolynomialFeatures
      pipe = make_pipeline(
          StandardScaler(),
          PolynomialFeatures(),
          Ridge())
```

- 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

```
[40]: param_grid = {'polynomialfeatures__degree': [1, 2, 3],
                    'ridge__alpha': [0.001, 0.01, 0.1, 1, 10, 100]}
      # Note: I had to use n_jobs=1. (n_jobs=-1 stalls on my machine)
      grid = GridSearchCV(pipe, param_grid=param_grid, cv=5, n_jobs=1)
      grid.fit(X_train, y_train)

GridSearchCV(cv=5, error_score='raise',
       estimator=Pipeline(steps=[('standardscaler', StandardScaler(copy=True, wi
   normalize=False, random_state=None, solver='auto', tol=0.001))]),
       fit_params={}, iid=True, n_jobs=1,
       param_grid={'ridge__alpha': [0.001, 0.01, 0.1, 1, 10, 100], 'polynomialfe
       pre_dispatch='2*n_jobs', refit=True, return_train_score=True,
       scoring=None, verbose=0)
```

- Visualing the R^2 results as a heatmap:

```
[41]: plt.matshow(grid.cv_results_['mean_test_score'].reshape(3, -1),
                  vmin=0, cmap="viridis")
      plt.xlabel("ridge__alpha")
      plt.ylabel("polynomialfeatures__degree")
      plt.xticks(range(len(param_grid['ridge__alpha'])), param_grid['ridge__alph
      plt.yticks(range(len(param_grid['polynomialfeatures__degree'])),
                 param_grid['polynomialfeatures__degree'])

      plt.colorbar();
```



- 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)

```
[32]: print("Best parameters: {}".format(grid.best_params_))
      print("Test-set score: {:.2f}".format(grid.score(X_test, y_test)))
```

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

Algorithm selection

- It is also possible to use a grid search to consider various alternative algorithms for a specific step, and tune the pipelines as well
  - StandardScaler or MinMaxScaler?
  - RandomForest or SVM?
- Note that the search space quickly becomes huge
- As an example, let's consider a pipeline that explores:
  - StandardScaler + SVM, varying `gamma` and `C`
  - None + RandomForest, varying `max_features`
- We instantiate a general pipeline, and define everything else in the grid
  - We need to define a list of 2 grids because of dependencies.

```
[34]: pipe = Pipeline([('preprocessing', StandardScaler()), ('classifier', SVC()
```

```
[35]: from sklearn.ensemble import RandomForestClassifier

      param_grid = [
          {'classifier': [SVC()], 'preprocessing': [StandardScaler(), None],
           'classifier__gamma': [0.001, 0.01, 0.1, 1, 10, 100],
           'classifier__C': [0.001, 0.01, 0.1, 1, 10, 100]},
          {'classifier': [RandomForestClassifier(n_estimators=100)],
           'preprocessing': [None], 'classifier__max_features': [1, 2, 3]}]
```

The Scaling+SVM pipeline wins!

```
[36]: X_train, X_test, y_train, y_test = train_test_split(
          cancer.data, cancer.target, random_state=0)

      grid = GridSearchCV(pipe, param_grid, cv=5)
      grid.fit(X_train, y_train)

      print("Best params:\n{}\n".format(grid.best_params_))
      print("Best cross-validation score: {:.2f}".format(grid.best_score_))
      print("Test-set score: {:.2f}".format(grid.score(X_test, y_test)))
```

```
Best params:
{'preprocessing': StandardScaler(copy=True, with_mean=True, with_std=True), 'cla
  decision_function_shape=None, degree=3, gamma=0.01, kernel='rbf',
  max_iter=-1, probability=False, random_state=None, shrinking=True,
  tol=0.001, verbose=False), 'classifier__gamma': 0.01}

Best cross-validation score: 0.99
Test-set score: 0.98
```

Algorithm selection: the road ahead

Automating the construction of machine learning pipelines is a hot topic of research, with different possible solutions:

- Model-based optimization: search large algorithm+hyperparameter spaces more effectively (limited to the learning step)
  - E.g. `Auto-SKLearn` and `Hyperopt-SKLearn`
- Multi-armed bandits and Racing: train algorithms first on a small sample, train the best 50% on a larger sample, continue until full dataset
- Genetic programming: very good at exploring large spaces
  - E.g. `sklearn-deap` or `TPOT` (builds pipelines as well)
- Meta-learning: build *meta-models* to predict which algorithms/hyperparameter ranges are most useful on new datasets
  - Requires large amounts of observations on previous datasets, as well as measureable dataset properties (meta-features) -> OpenML
- Combinations of the above
  - Especially meta-learning with all others
  - Contact me if interested :)

Approaching machine learning problems

- Just running your favourite algorithm on every new 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'

Concept drift

- Data is often a stream, and model building is often part of a feedback cycle
  - Collect new data, cleaning, build models, analyse
  - Continually check that your model is still working
- Concept drift: sudden or gradual changes in the phenomenon that you want to model
  - External: market/weather evolutions, human behavior

  - Feedback: your predictions (and actions) may change future data
    - E.g. movie recommendations, new drugs, treatments, ...

- Different ways to tackle this:
  - Repeated retraining (but often not sure when)
  - Stream mining algorithms (learn on fast streaming data)
  - Change detection techniques (retrain when model starts failing)
  - Meta-learning (switch algorithms depending on data properties)

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

Further reading (Theory)

- The Elements of Statistical Learning (Hastie, Tibshirani, Friedman)
- Deep Learning (Goodfellow, Bengio, Courville) - see later
- Gaussian Processes for Machine Learning (Rasmussen, Williams)
- Machine Learning: An Algorithmic Perspective (Marsland)
- Pattern Recognition and Machine Learning (Bishop)
- Machine Learning: A Probabilistic Perspective (Murphy)
- Foundations of Data Science (Blum, Hopcroft, Kannan)

Other packages

- Deep learning: `TensorFlow`, `Theano`, `Torch`, `Caffe`,...
  - `Keras`, `Lasagne` provide simpler interfaces
- R libraries often provide a richer variation of techniques
  - Powerful statistical analysis and visualization tools
  - mlr (Machine Learning in R)
- `vowpal wabbit` (vw): C++ library for large datasets and streaming data
- MOA (Massive Online Analysis): Java library for streaming data
- MLLib: Scala library on top of Spark, for large distributed systems
- PyMC/Stan: Probabilistic programming, allows to model how likely each data point is correct. Simple, elegant way to build models.

Summary

- 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
- Real world applications require careful thought, prototyping, and tireless evaluation.