scikit-learn

One of the most prominent Python libraries for machine learning:

- Contains many state-of-the-art machine learning algorithms
- Wide range of evaluation measures and techniques
- Offers comprehensive documentation (http://scikit-learn.org/stable/documentation) about each algorithm
- Widely used, and a wealth of tutorials (http://scikit-learn.org/stable/user_guide.html) and code snippets are available
- Works well with numpy, scipy, pandas, matplotlib,...

Algorithms

See the Reference (http://scikit-learn.org/dev/modules/classes.html)

Supervised learning:

- Linear models (Ridge, Lasso, Elastic Net, ...)
- Support Vector Machines
- Tree-based methods (Classification/Regression Trees, Random Forests,...)
- Nearest neighbors
- Neural networks
- Gaussian Processes
- Feature selection

Unsupervised learning:

- Clustering (KMeans, ...)
- Matrix Decomposition (PCA, ...)
- Manifold Learning (Embeddings)
- Density estimation
- Outlier detection

Model selection and evaluation:

- Cross-validation
- Grid-search
- Lots of metrics

Data import

Multiple options:

- A few toy datasets are included in `sklearn.datasets`
- Import 1000s of datasets via `sklearn.datasets.fetch_openml`
- You can import data files (CSV) with `pandas` or `numpy`

```
from sklearn.datasets import load_iris, fetch_openml
iris_data = load_iris()
dating_data = fetch_openml(name="SpeedDating")
```

Building models

All scikitlearn *estimators* follow the same interface

```
class SupervisedEstimator(...):
    def __init__(self, hyperparam, ...):

    def fit(self, X, y):   # Fit/model the training data
        ...                # given data X and targets y
        return self

    def predict(self, X):  # Make predictions
        ...                # on unseen data X
        return y_pred

    def score(self, X, y): # Predict and compare to true
        ...                # labels y
        return score
```

Training and testing data

To evaluate our classifier, we need to test it on unseen data.
`train_test_split`: splits data randomly in 75% training and 25% test data.

```
X_train, X_test, y_train, y_test = train_test_split(
    iris_data['data'], iris_data['target'], random_state=0)
```

```
X_train shape: (11
2, 4)
y_train shape: (11
2,)
X_test shape: (38,
4)
y_test shape: (38,)
```

Fitting a model

The first model we'll build is a k-Nearest Neighbor classifier.
kNN is included in `sklearn.neighbors`, so let's build our first model

```
knn = KNeighborsClassifier(n_neighbors=1)
knn.fit(X_train, y_train)
```

```
Out[8]: KNeighborsClassifier(algorithm='auto', leaf_size=30, metric='minkowsk
        i',
                    metric_params=None, n_jobs=None, n_neighbors=1, p=2,
                    weights='uniform')
```

Making predictions

Let's create a new example and ask the kNN model to classify it

```
X_new = np.array([[5, 2.9, 1, 0.2]])
prediction = knn.predict(X_new)
class_name = iris_data['target_names'][prediction]
```

```
Prediction: [0]
Predicted target name: ['seto
sa']
```

Evaluating the model

Feeding all test examples to the model yields all predictions

```
y_pred = knn.predict(X_test)
```

```
Test set predictions:
 [2 1 0 2 0 2 0 1 1 1 2 1 1 1 1 0 1 1 0 0 2 1 0 0 2 0 0 1 1 0 2 1 0 2 2
1 0
 2]
```

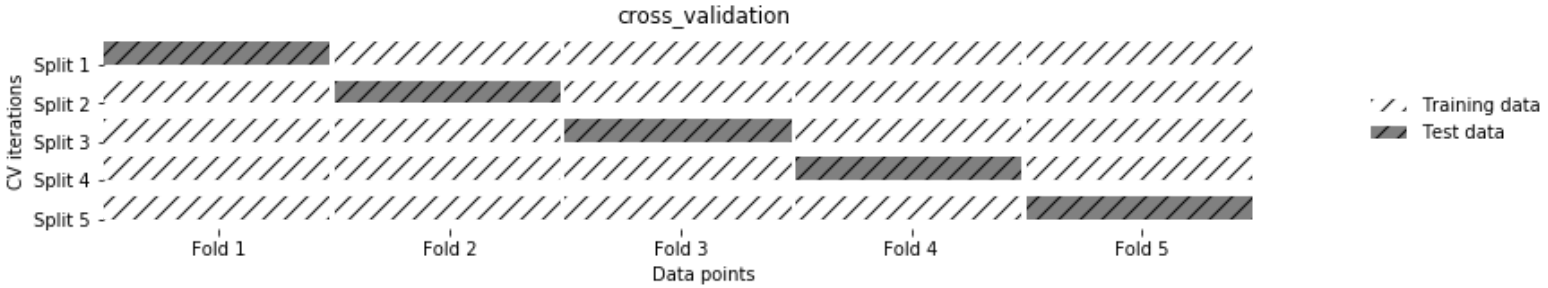
The `score` function computes the percentage of correct predictions

```
knn.score(X_test, y_test)
```

```
Score:
0.97
```

Cross-validation

- More stable, thorough way to estimate generalization performance
- *k-fold cross-validation* (CV): split (randomized) data into *k* equal-sized parts, called *folds*
  - First, fold 1 is the test set, and folds 2-5 comprise the training set
  - Then, fold 2 is the test set, folds 1,3,4,5 comprise the training set
  - Compute *k* evaluation scores, aggregate afterwards (e.g. take the mean)



Cross-validation in scikit-learn

- `cross_val_score` function with learner, training data, labels
- Returns list of all scores
  - Does 3-fold CV by default, can be changed via `cv` hyperparameter
  - Default scoring measures are accuracy (classification) or R^2 (regression)
- Even though models are built internally, they are not returned

```
knn = KNeighborsClassifier(n_neighbors=1)
scores = cross_val_score(knn, iris.data, iris.target, cv=5)
print("Cross-validation scores: {}".format(scores))
print("Average cross-validation score: {:.2f}".format(scores.mean()))
print("Variance in cross-validation score: {:.4f}".format(np.var(scores
)))
```

```
Cross-validation scores: [0.98  0.922 1.
]
Average cross-validation score: 0.97
Variance in cross-validation score: 0.00
11
```

More variants

- Stratified cross-validation: for inbalanced datasets
- Leave-one-out cross-validation: for very small datasets
- Shuffle-Split cross-validation: whenever you need to shuffle the data first
- Repeated cross-validation: more trustworthy, but more expensive
- Cross-validation with groups: Whenever your data contains non-independent datapoints, e.g. data points from the same patient
- Bootstrapping: sampling with replacement, for extracting statistical properties

Avoid data leakage

- Simply taking the best performing model based on cross-validation performance yields optimistic results
- We've already used the test data to evaluate each model!
- Hence, we don't have an independent test set to evaluate these hyperparameter settings
  - Information 'leaks' from test set into the final model
- Solution: Set aside part of the training data to evaluate the hyperparameter settings
  - Select best model on validation set
  - Rebuild the model on the training+validation set
  - Evaluate optimal model on the test set


training set
validation set
test set
Model fitting
Parameter selection
Evaluation

Model selection and Hyperparameter tuning

- There are many algorithms to choose from
- Most algorithms have parameters (hyperparameters) that control model complexity
- Now that we know how to evaluate models, we can improve them selecting by `tuning` algorithms for your data

We can basically use any optimization technique to optimize hyperparameters:

- **Grid search**
- **Random search**

More advanced techniques:

- Local search
- Racing algorithms
- Bayesian optimization
- Multi-armed bandits
- Genetic algorithms

Grid Search

- For each hyperparameter, create a list of interesting/possible values
  - E.g. For kNN: k in [1,3,5,7,9,11,33,55,77,99]
  - E.g. For SVM: C and gamma in $[10^{-10}..10^{10}]$
- Evaluate all possible combinations of hyperparameter values
  - E.g. using cross-validation
- Split the training data into a training and validation set
- Select the hyperparameter values yielding the best results on the validation set

Grid search in scikit-learn

- Create a parameter grid as a dictionary
  - Keys are parameter names
  - Values are lists of hyperparameter values

```
param_grid = {'C': [0.001, 0.01, 0.1, 1, 10, 100],
              'gamma': [0.001, 0.01, 0.1, 1, 10, 100]}
print("Parameter grid:\n{}".format(param_grid))
```

```
Parameter grid:
{'C': [0.001, 0.01, 0.1, 1, 10, 100], 'gamma': [0.001, 0.01, 0.1, 1, 1
0, 100]}
```

- `GridSearchCV`: like a classifier that uses CV to automatically optimize its hyperparameters internally
  - Input: (untrained) model, parameter grid, CV procedure
  - Output: optimized model on given training data
  - Should only have access to training data

```
grid_search = GridSearchCV(SVC(), param_grid, cv=5)
grid_search.fit(X_train, y_train)
```

```
Out[15]: GridSearchCV(cv=5, error_score='raise-deprecating',
             estimator=SVC(C=1.0, cache_size=200, class_weight=None, coef0=0.
0,
  decision_function_shape='ovr', degree=3, gamma='auto_deprecated',
  kernel='rbf', max_iter=-1, probability=False, random_state=None,
  shrinking=True, tol=0.001, verbose=False),
             fit_params=None, iid='warn', n_jobs=None,
             param_grid={'C': [0.001, 0.01, 0.1, 1, 10, 100], 'gamma': [0.00
1, 0.01, 0.1, 1, 10, 100]},
             pre_dispatch='2*n_jobs', refit=True, return_train_score='warn',
             scoring=None, verbose=0)
```

The optimized test score and hyperparameters can easily be retrieved:

```
grid_search.score(X_test, y_test)
grid_search.best_params_
grid_search.best_score_
grid_search.best_estimator_
```

```
Test set score: 0.97
Best parameters: {'C': 100, 'gamma': 0.01}
Best cross-validation score: 0.97
Best estimator:
SVC(C=100, cache_size=200, class_weight=None, coef0=0.0,
  decision_function_shape='ovr', degree=3, gamma=0.01, kernel='r
bf',
  max_iter=-1, probability=False, random_state=None, shrinking=T
rue,
  tol=0.001, verbose=False)
```

Nested cross-validation

- Note that we are still using a single split to create the outer test set
- We can also use cross-validation here
- Nested cross-validation:
  - Outer loop: split data in training and test sets
  - Inner loop: run grid search, splitting the training data into train and validation sets
- Result is a just a list of scores
  - There will be multiple optimized models and hyperparameter settings (not returned)
- To apply on future data, we need to train `GridSearchCV` on all data again

```
scores = cross_val_score(GridSearchCV(SVC(), param_grid, cv=5),
                         iris.data, iris.target, cv=5)


Cross-validation scores:  [0.967 1.    0.967 0.967 1.
]
Mean cross-validation score:  0.9800000000000001
```

Random Search

- Grid Search has a few downsides:
  - Optimizing many hyperparameters creates a combinatorial explosion
  - You have to predefine a grid, hence you may jump over optimal values
- Random Search:
  - Picks `n_iter` random parameter values
  - Scales better, you control the number of iterations
  - Often works better in practice, too
    - not all hyperparameters interact strongly
    - you don't need to explore all combinations

- Executing random search in scikit-learn:
  - `RandomizedSearchCV` works like `GridSearchCV`
  - Has `n_iter` parameter for the number of iterations
  - Search grid can use distributions instead of fixed lists

```
param_grid = {'C': expon(scale=100),
              'gamma': expon(scale=.1)}
random_search = RandomizedSearchCV(SVC(), param_distributions=param_grid
,
                                   n_iter=20)
random_search.fit(X_train, y_train)
random_search.best_estimator_
```

```
Out[18]: SVC(C=6.109237791897481, cache_size=200, class_weight=None, coef0=
         0.0,
           decision_function_shape='ovr', degree=3, gamma=0.047236266339034
         1,
           kernel='rbf', max_iter=-1, probability=False, random_state=None,
           shrinking=True, tol=0.001, verbose=False)
```

Pipelines

- Many learning algorithms 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.

Example: Speed dating data

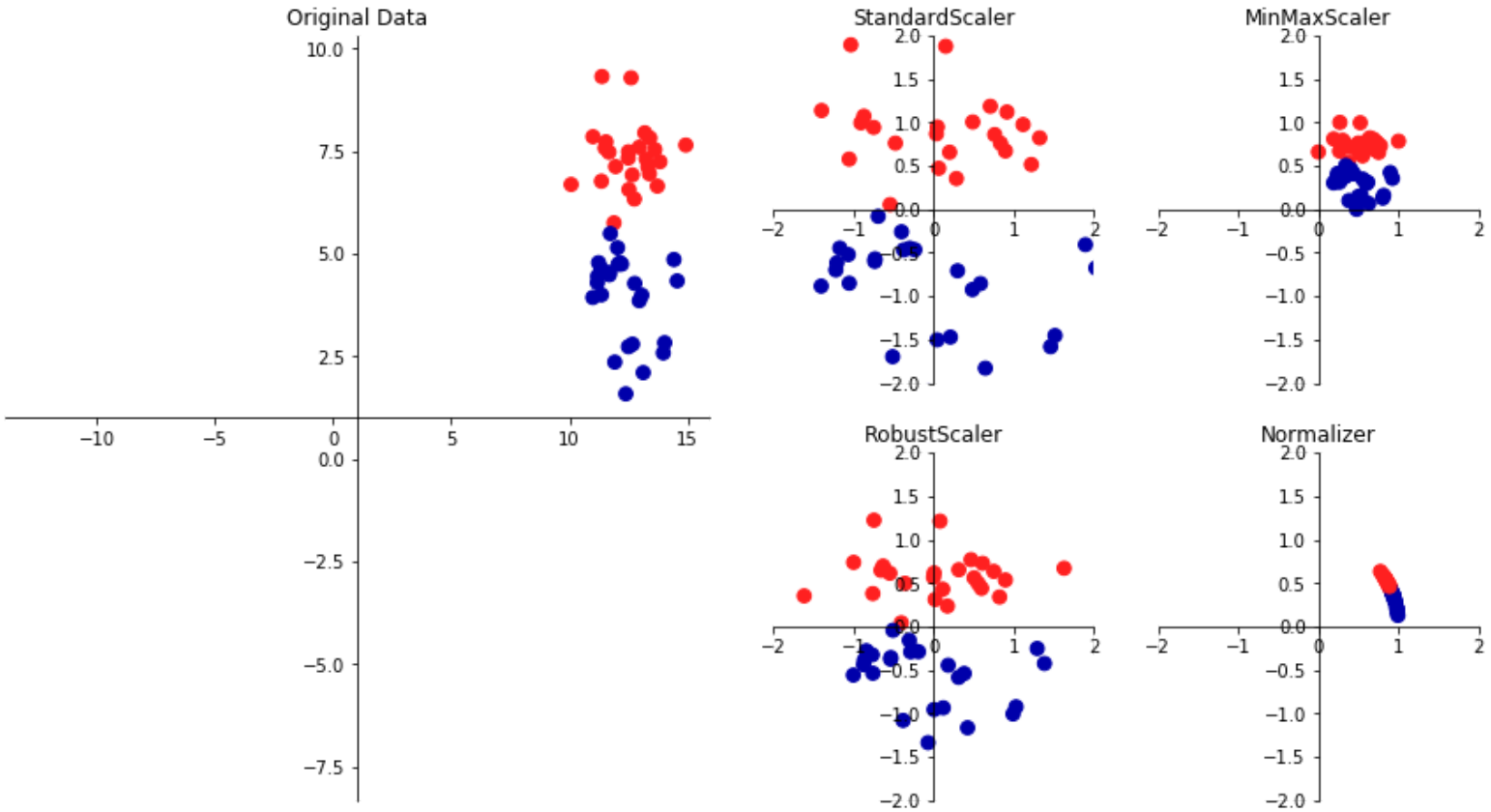
- Data collected from speed dating events
- See https://www.openml.org/d/40536 (https://www.openml.org/d/40536)
- Could also be collected from dating website or app
- Real-world data:
  - Different numeric scales
  - Missing values
  - Likely irrelevant features
  - Different types: Numeric, categorical,...
  - Input errors (e.g. 'lawyer' vs 'Lawyer')

```
dating_data = fetch_openml("SpeedDating")
```

Scaling

When the features have different scales (their values range between very different minimum and maximum values), one feature will overpower the others. Several scaling techniques are available to solve 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


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

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

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

Use it 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)

Univariate feature selection

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.

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")
```

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.

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)
```

Feature encoding

- scikit-learn classifiers only handle numeric data. If your features are categorical, you need to encode them first
- `LabelEncoder` simply replaces each value with an integer value
- `OneHotEncoder` converts a feature of *n* values to *n* binary features
  - Provide `categories` as array or set to 'auto'

```
X_enc = OneHotEncoder(categories='auto').fit_transform(X)
```

- `ColumnTransformer` can apply different transformers to different features
- Transformers can be pipelines doing multiple things

```
numeric_features = ['age', 'pref_o_attractive']
numeric_transformer = Pipeline(steps=[
    ('imputer', SimpleImputer(strategy='median')),
    ('scaler', StandardScaler())])

categorical_features = ['gender', 'd_d_age', 'field']
categorical_transformer = Pipeline(steps=[
    ('imputer', SimpleImputer(strategy='constant', fill_value='missing'
)),
    ('onehot', OneHotEncoder(handle_unknown='ignore'))])

preprocessor = ColumnTransformer(
    transformers=[
        ('num', numeric_transformer, numeric_features),
        ('cat', categorical_transformer, categorical_features)])
```

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_inter
cept=True,
     intercept_scaling=1, loss='squared_hinge', max_iter=1000,
     multi_class='ovr', penalty='l2', random_state=None, tol=0.0001,
     verbose=0)
```

- Or we can use the `named_steps` dictionary

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

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

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

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

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`

```
param_grid = {'svm__C': [0.001, 0.01, 0.1, 1, 10, 100],
              'svm__gamma': [0.001, 0.01, 0.1, 1, 10, 100]}
pipe = pipeline.Pipeline([("scaler", MinMaxScaler()), ("svm", SVC(C=100
))])
grid = GridSearchCV(pipe, param_grid=param_grid, cv=5)
grid.fit(X_train, y_train)
```

```
Best cross-validation accuracy: 0.97
Test set score: 0.97
Best parameters: {'svm__C': 10, 'svm__gamm
a': 1}
```