Model Evaluation and Hyperparameter Tuning

Hsi-Pin Ma 馬席彬

http://lms.nthu.edu.tw/course/40724

Department of Electrical Engineering

National Tsing Hua University

Outline

- Streaming Workflows with Pipelines
- Using k-fold Cross-Validation to Assess Model Performance
- Debugging Algorithms with Learning and Validation Curves
- Fine-Tuning Machine Learning Models via Grid Search
- Looking at Different Performance Evaluation Metrics
- Dealing with Class Imbalance

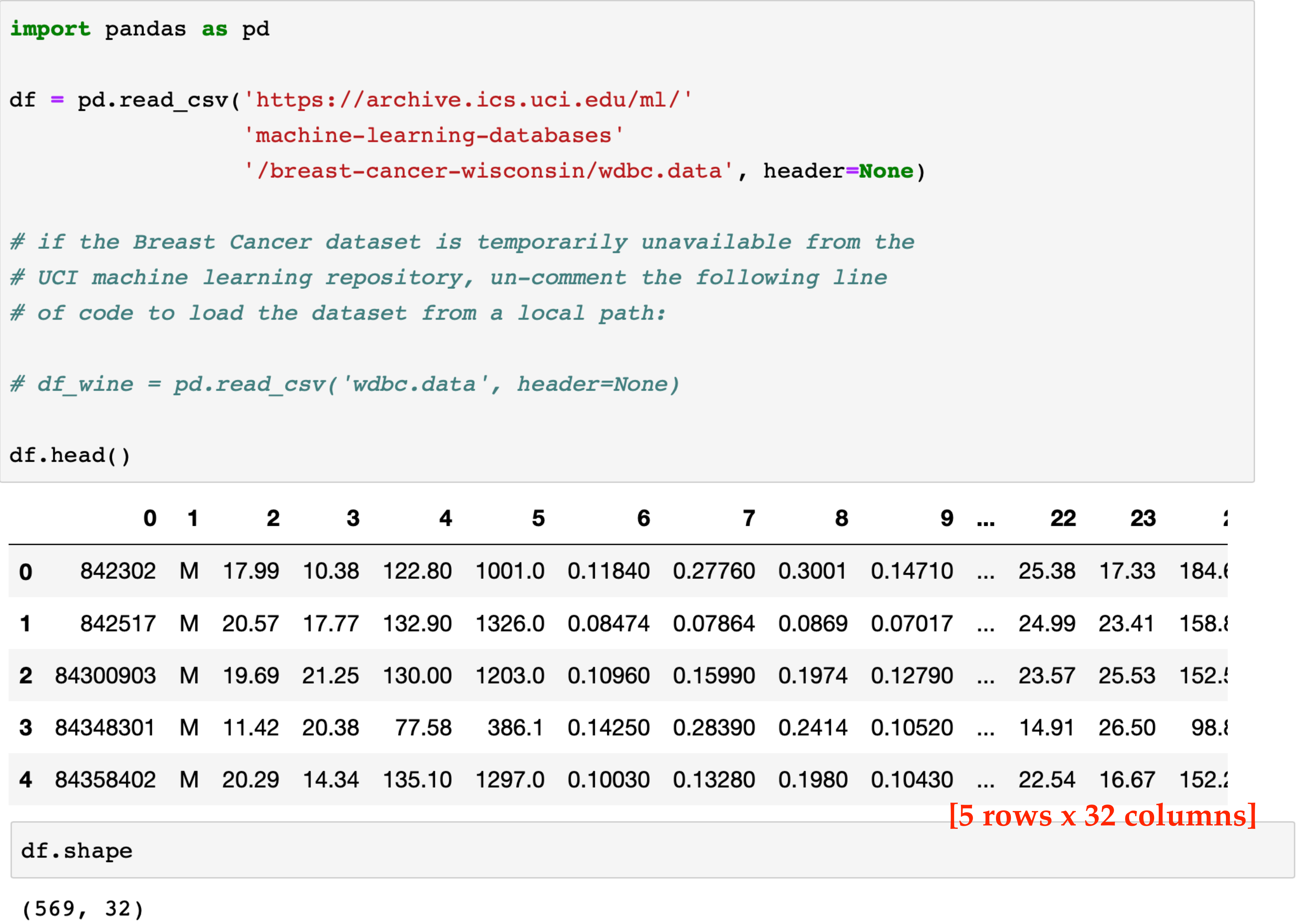
Streamlining Workflows with Pipelines

Pipeline Class

- In preprocessing, the parameters obtained during the fitting of the training data should be reused in the separate test dataset
- **Pipeline** class in scikit-learn allows to fit a model including an arbitrary number of transformation steps and apply it to make predictions about new data.

Breast Cancer Wisconsin Dataset

Load dataset



```
import pandas as pd

df = pd.read_csv('https://archive.ics.uci.edu/ml/'
                 'machine-learning-databases'
                 '/breast-cancer-wisconsin/wdbc.data', header=None)

# if the Breast Cancer dataset is temporarily unavailable from the
# UCI machine learning repository, un-comment the following line
# of code to load the dataset from a local path:

# df_wine = pd.read_csv('wdbc.data', header=None)

df.head()
```

	0	1	2	3	4	5	6	7	8	9	...	22	23	2
0	842302	M	17.99	10.38	122.80	1001.0	0.11840	0.27760	0.3001	0.14710	...	25.38	17.33	184.
1	842517	M	20.57	17.77	132.90	1326.0	0.08474	0.07864	0.0869	0.07017	...	24.99	23.41	158.
2	84300903	M	19.69	21.25	130.00	1203.0	0.10960	0.15990	0.1974	0.12790	...	23.57	25.53	152.
3	84348301	M	11.42	20.38	77.58	386.1	0.14250	0.28390	0.2414	0.10520	...	14.91	26.50	98.
4	84358402	M	20.29	14.34	135.10	1297.0	0.10030	0.13280	0.1980	0.10430	...	22.54	16.67	152.

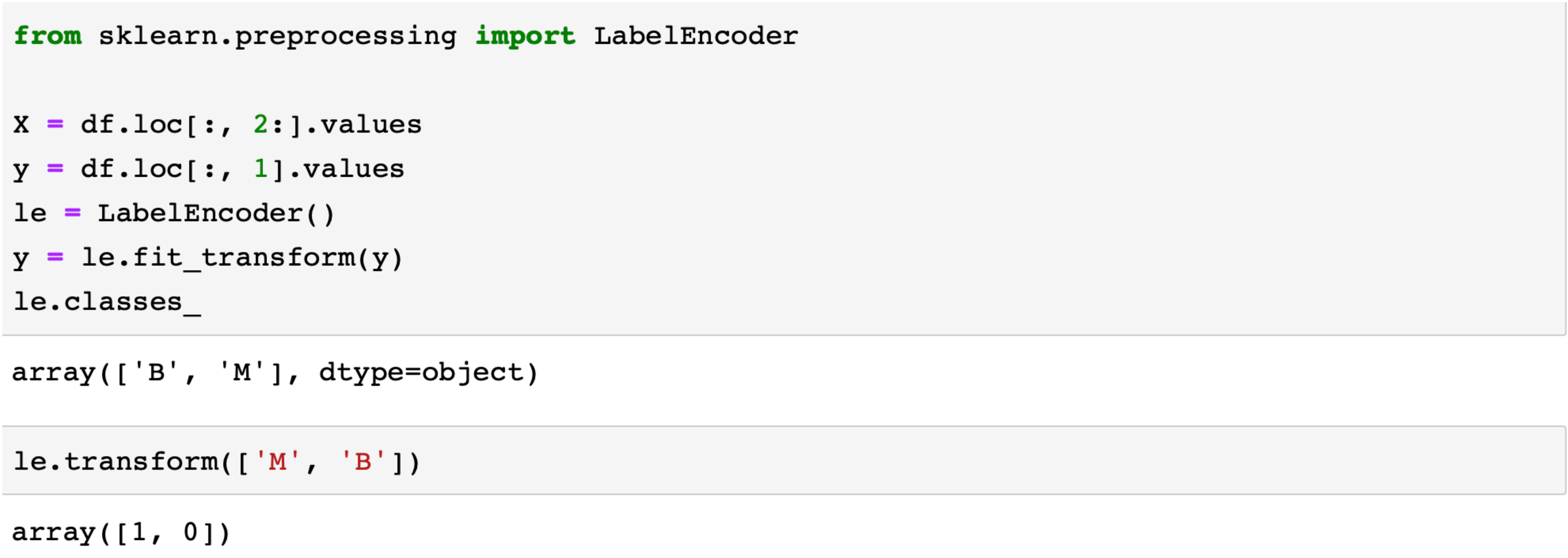
[5 rows x 32 columns]

```
df.shape
```

```
(569, 32)
```

Breast Cancer Wisconsin Dataset

- Assign 30 features to X and use LabelEncoder to transform the class label



```
from sklearn.preprocessing import LabelEncoder

X = df.loc[:, 2:].values
y = df.loc[:, 1].values
le = LabelEncoder()
y = le.fit_transform(y)
le.classes_
```

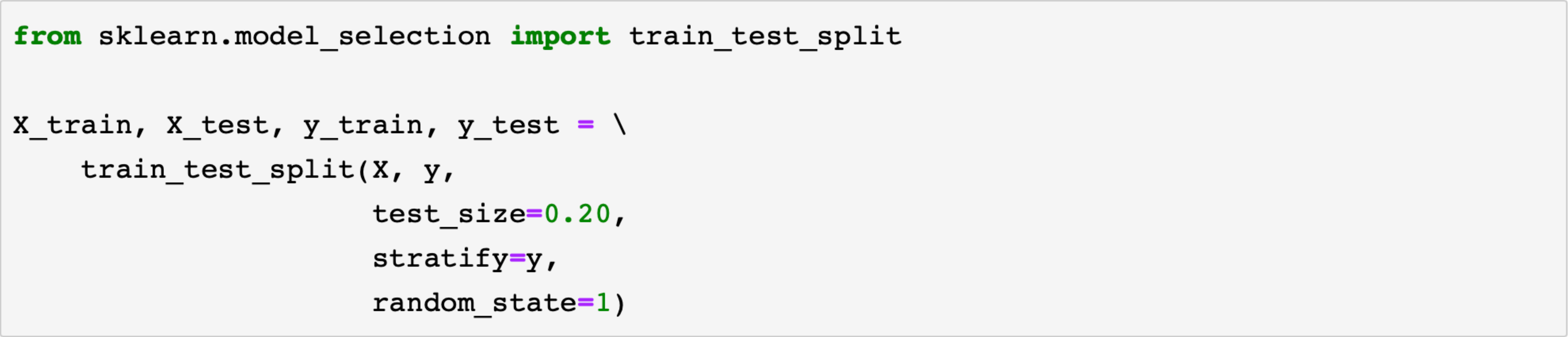
```
array(['B', 'M'], dtype=object)
```

```
le.transform(['M', 'B'])
```

```
array([1, 0])
```

Breast Cancer Wisconsin Dataset

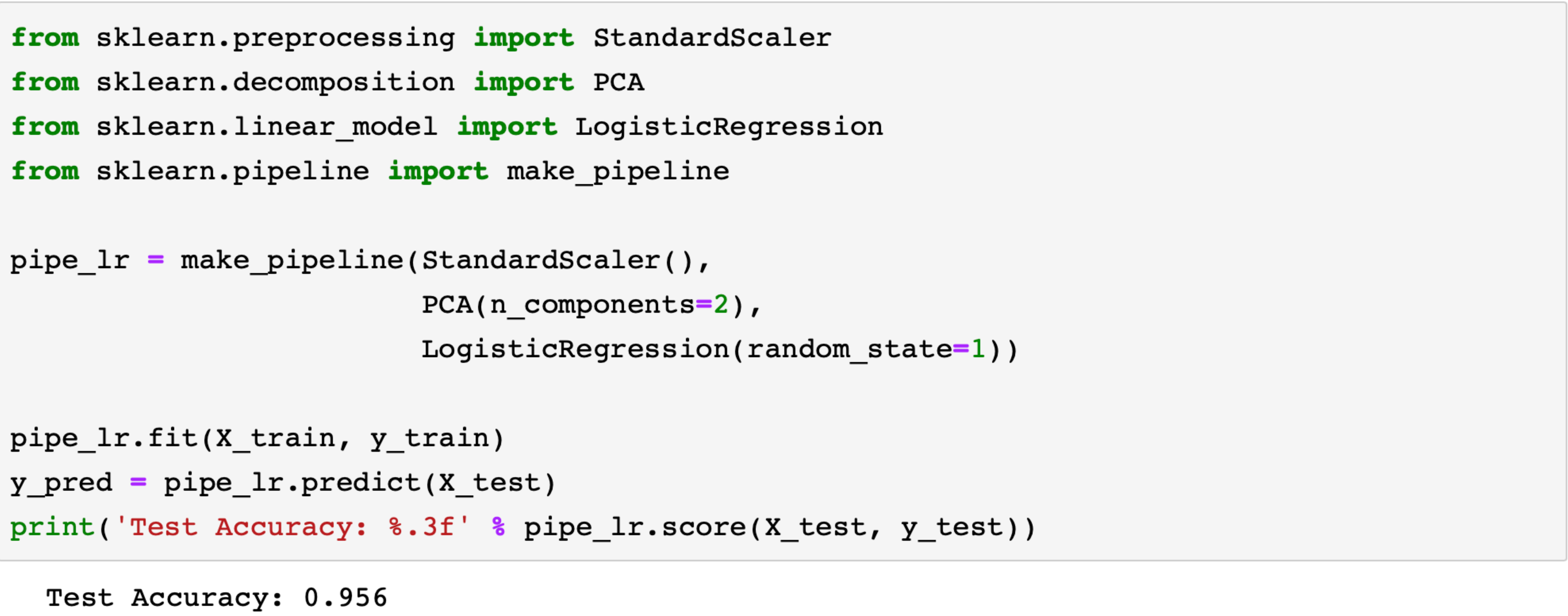
- Split dataset with 80:20



```
from sklearn.model_selection import train_test_split

X_train, X_test, y_train, y_test = \
    train_test_split(X, y,
                     test_size=0.20,
                     stratify=y,
                     random_state=1)
```

Combining Transformers and Estimators in a Pipeline

- Put standardization, dimensionality reduction of features (PCA), logistic regression in a pipeline

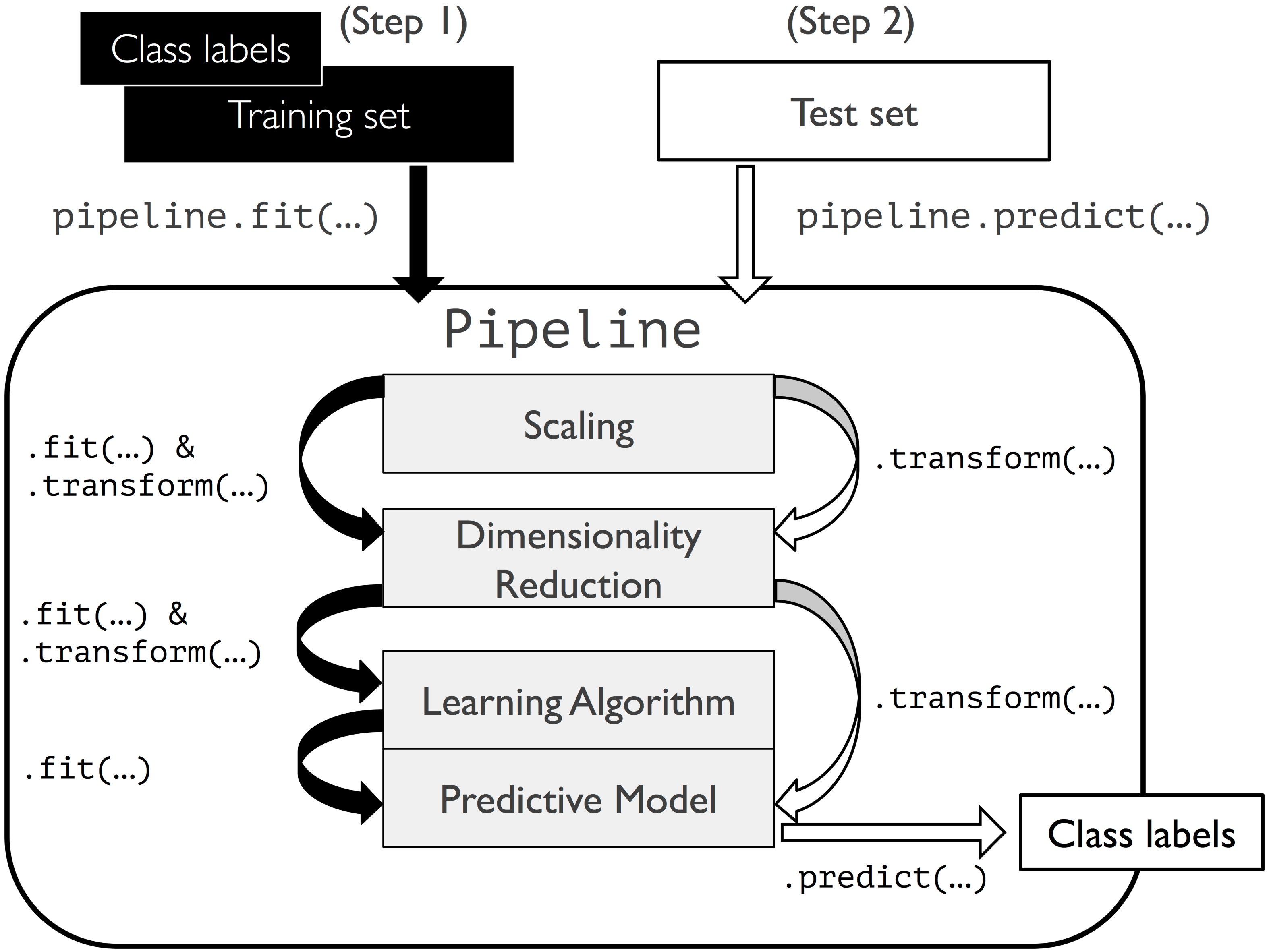


```
from sklearn.preprocessing import StandardScaler
from sklearn.decomposition import PCA
from sklearn.linear_model import LogisticRegression
from sklearn.pipeline import make_pipeline

pipe_lr = make_pipeline(StandardScaler(),
                        PCA(n_components=2),
                        LogisticRegression(random_state=1))

pipe_lr.fit(X_train, y_train)
y_pred = pipe_lr.predict(X_test)
print('Test Accuracy: %.3f' % pipe_lr.score(X_test, y_test))
```

```
Test Accuracy: 0.956
```

Combining Transformers and Estimators in a Pipeline

- **make_pipeline** function takes ***an arbitrary number of scikit-learn transformers*** that support the **fit** and **transform** method, followed by ***a scikit-learn estimator*** that implements **fit** and **predict** methods



Using k-fold Cross Validation to Assess Model Performance

Model Selection

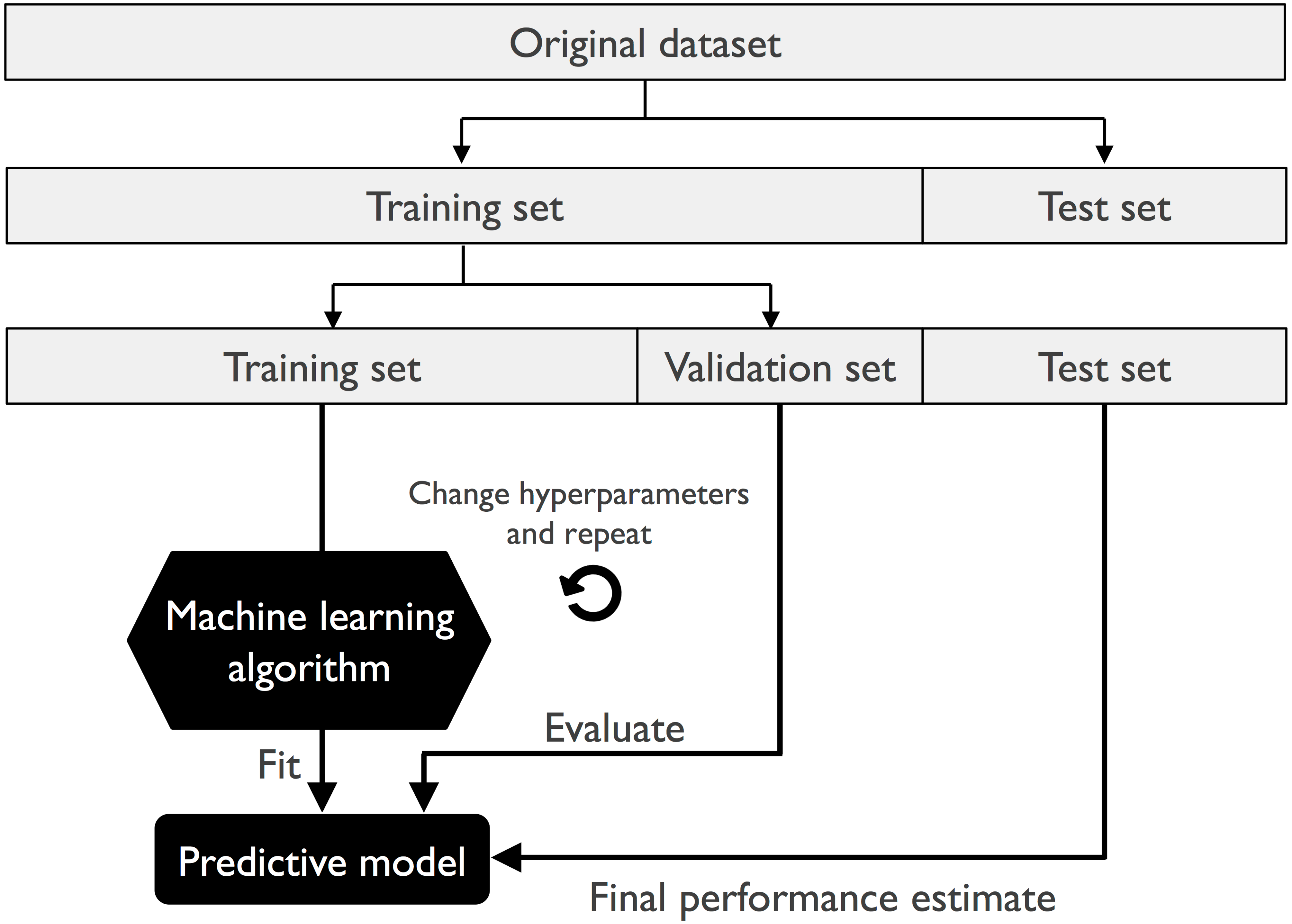
- How to obtain an unbiased estimate of models' performance
  - Estimate model performance on **unseen** data
- Model selection
  - For a given classification, to select the optimal values of tuning parameters (i.e., hyperparameters) to further improve the performance of predicting unseen data.
- There are two general methods
  - holdout cross-validation method
  - k-fold cross-validation method

The Holdout Method

- Split the initial dataset into a separating training and test dataset
  - The former is for model training and the latter is to estimate its generalization performance
- However, during model selection, if the same test dataset is reused over and over again, the test dataset will become part of the training data, and the model may be overfitted.

The Holdout Method

- A better way



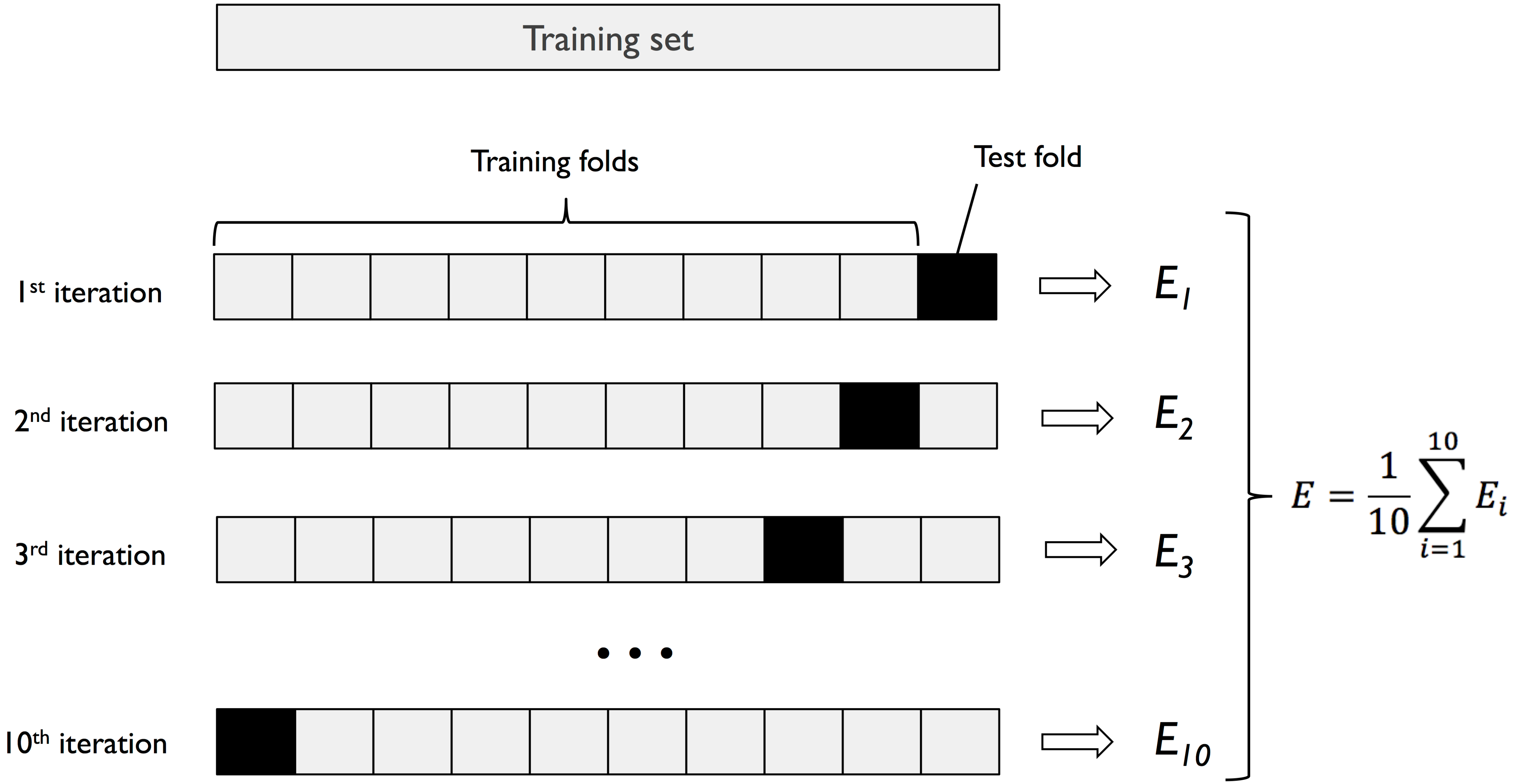
- However, performance estimate will be sensitive on how to partition the training set into training and validation subsets.

k-fold Cross-Validation

- Repeat the holdout method k times on k subset of the training data
  - Randomly split the training dataset into k folds without replacement (resampling without replacement)
    - k-1 fold for model training, and one fold for performance evaluation
  - Repeat k times => k models and performance estimates
  - A configuration of hyperparameters is selected when its average performance is the best.
  - Low-variance estimate of model performance than holdout method
    - Each sample point will be used for training and validation exactly once

k-fold Cross-Validation

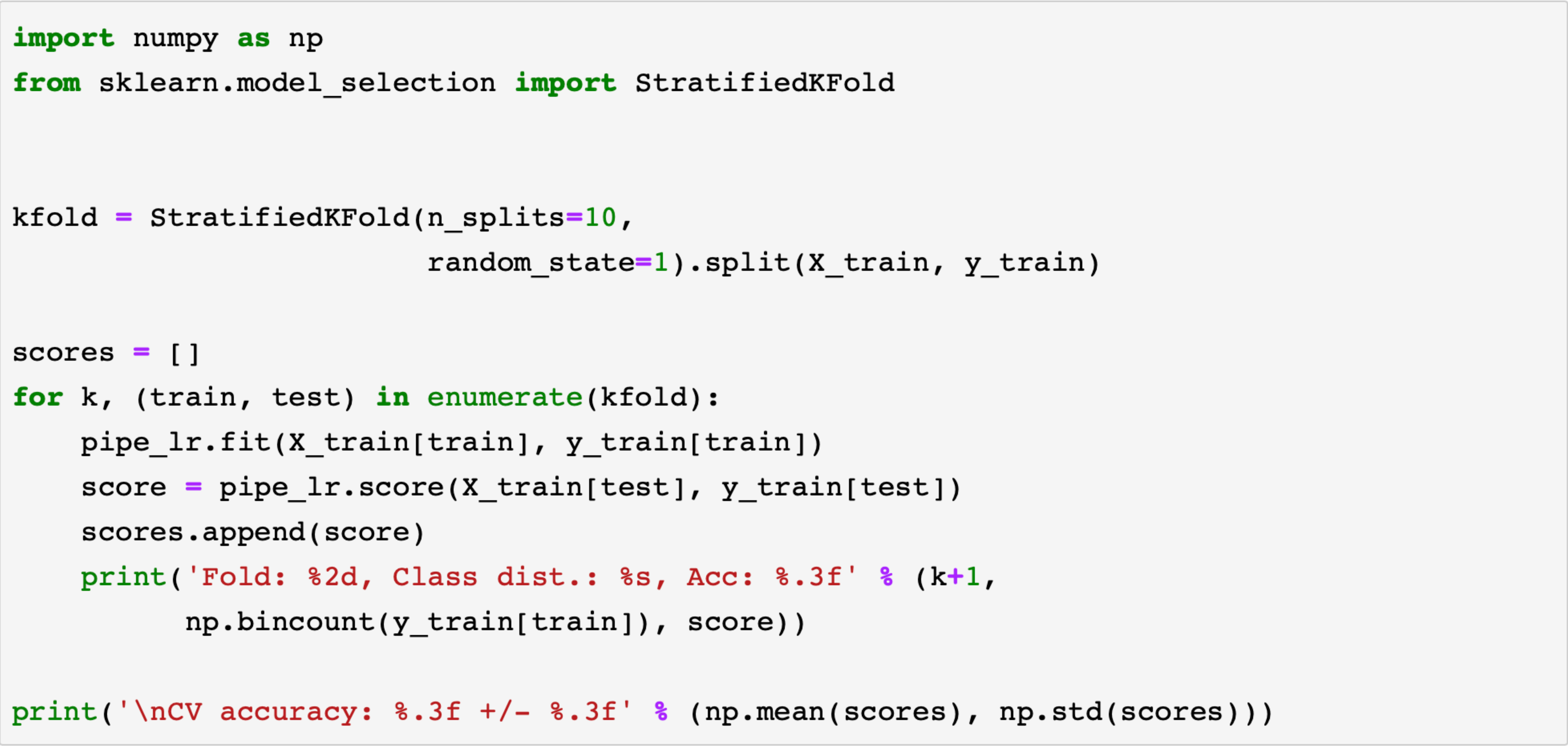
- A good standard value of k is 10
  - k increases as the training dataset is relatively small
  - k decreases as the training dataset is relatively large



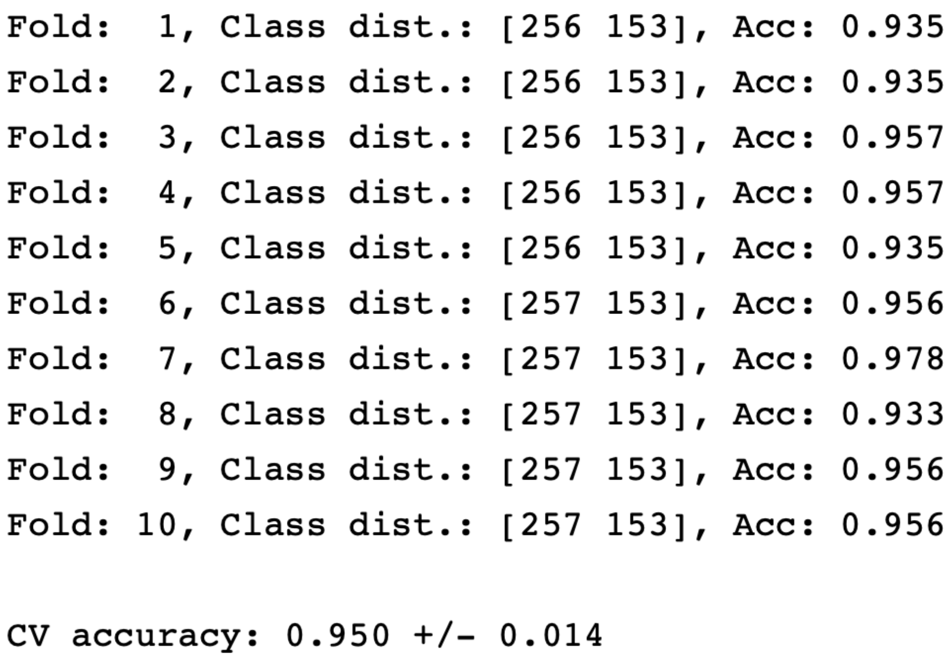
Variations of k-fold Cross-Validation

- Leave-one-out cross-validation
  - Set the number of folds equal to the number of training samples
  - Only a single training sample used for testing during each iteration
  - Recommended approach for very small dataset
- Stratified k-fold cross-validation
  - Class proportions preserved in each fold
    - each fold is representative of the class proportions in the training set
  - Better performance estimates for imbalanced data

Stratified k-fold Cross-Validation



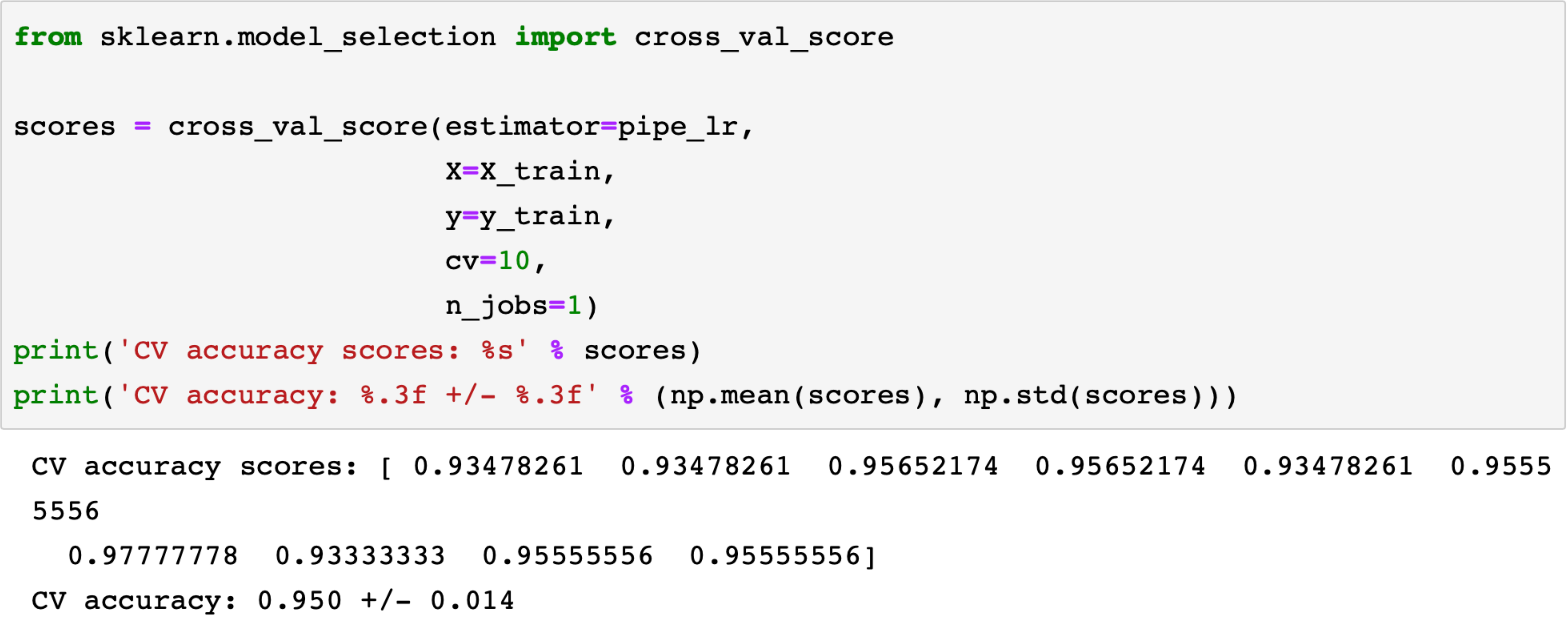
```python
import numpy as np
from sklearn.model_selection import StratifiedKFold


kfold = StratifiedKFold(n_splits=10,
                        random_state=1).split(X_train, y_train)

scores = []
for k, (train, test) in enumerate(kfold):
    pipe_lr.fit(X_train[train], y_train[train])
    score = pipe_lr.score(X_train[test], y_train[test])
    scores.append(score)
    print('Fold: %2d, Class dist.: %s, Acc: %.3f' % (k+1,
          np.bincount(y_train[train]), score))

print('\nCV accuracy: %.3f +/- %.3f' % (np.mean(scores), np.std(scores)))
```



```
Fold:  1, Class dist.: [256 153], Acc: 0.935
Fold:  2, Class dist.: [256 153], Acc: 0.935
Fold:  3, Class dist.: [256 153], Acc: 0.957
Fold:  4, Class dist.: [256 153], Acc: 0.957
Fold:  5, Class dist.: [256 153], Acc: 0.935
Fold:  6, Class dist.: [257 153], Acc: 0.956
Fold:  7, Class dist.: [257 153], Acc: 0.978
Fold:  8, Class dist.: [257 153], Acc: 0.933
Fold:  9, Class dist.: [257 153], Acc: 0.956
Fold: 10, Class dist.: [257 153], Acc: 0.956

CV accuracy: 0.950 +/- 0.014
```

Stratified k-fold Cross-Validation

- scikit-learn implements a k-fold cross-validation scorer (**cross_val_score**)
  - **n_jobs** provides parallel processing capability



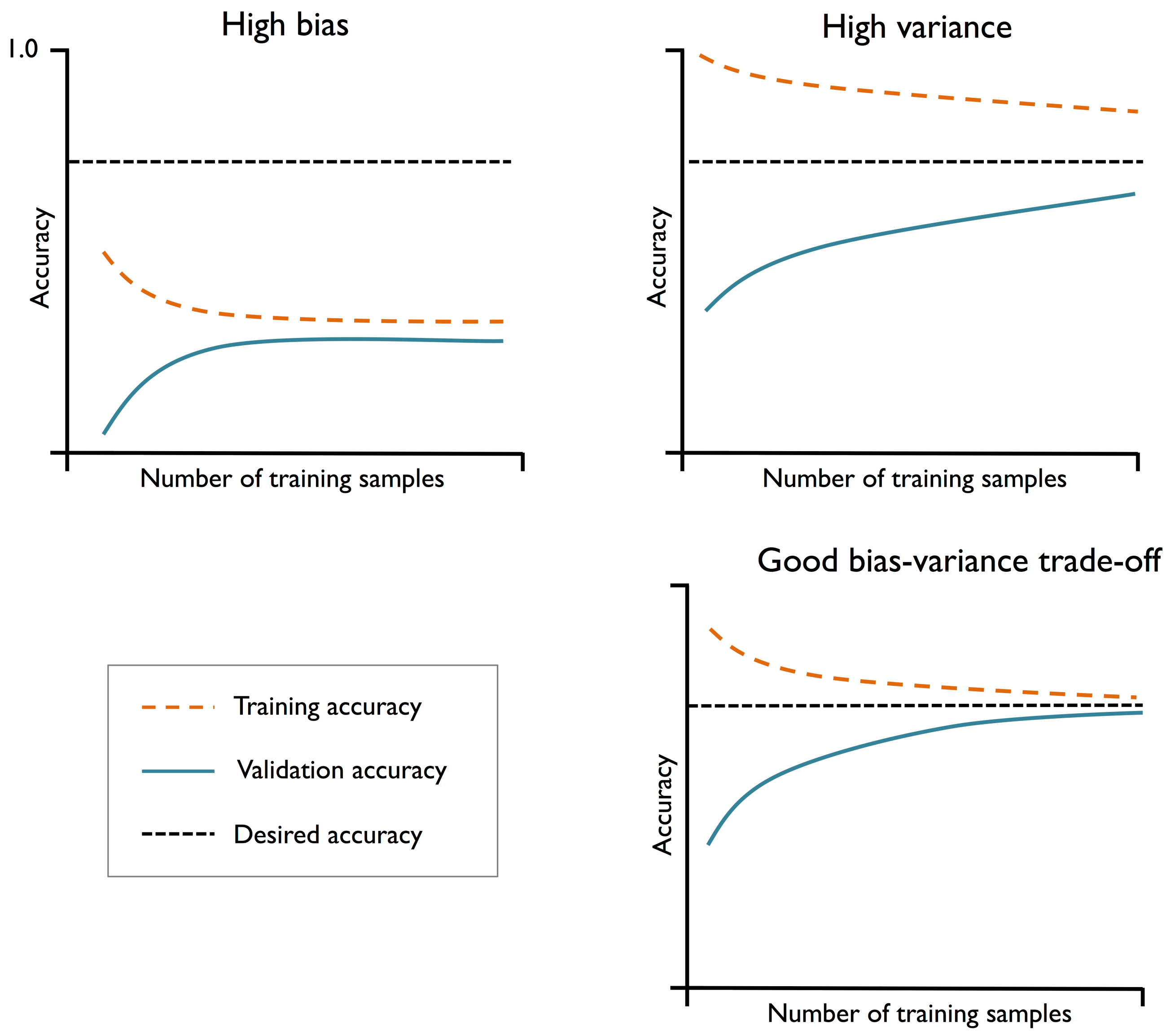
```
from sklearn.model_selection import cross_val_score

scores = cross_val_score(estimator=pipe_lr,
                         X=X_train,
                         y=y_train,
                         cv=10,
                         n_jobs=1)
print('CV accuracy scores: %s' % scores)
print('CV accuracy: %.3f +/- %.3f' % (np.mean(scores), np.std(scores)))
```

```
CV accuracy scores: [ 0.93478261  0.93478261  0.95652174  0.95652174  0.93478261  0.9555
5556
  0.97777778  0.93333333  0.95555556  0.95555556]
CV accuracy: 0.950 +/- 0.014
```

Debugging Algorithms with Learning and Validation Curves

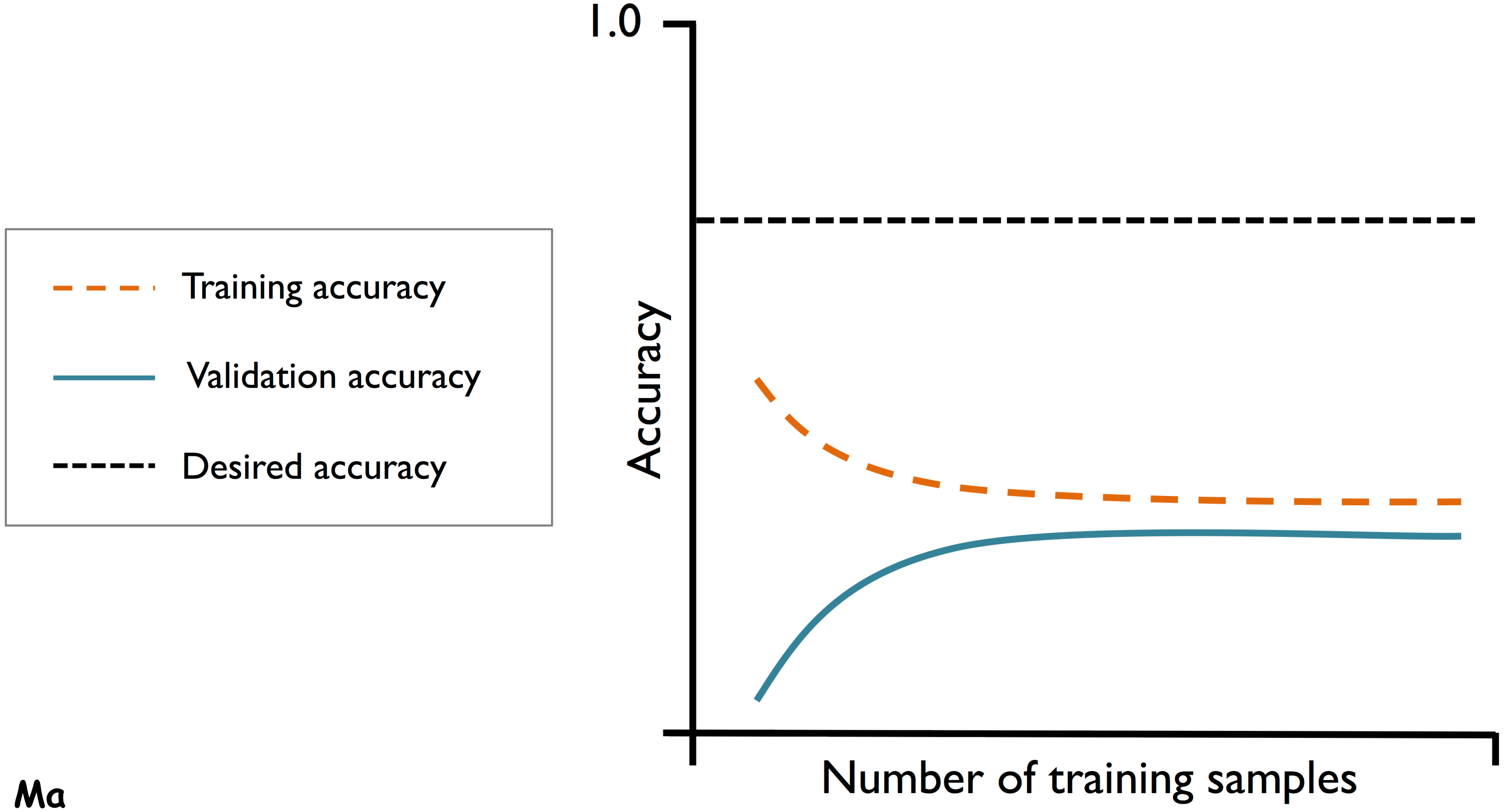
Diagnosing Bias and Variance Problems with Learning Curves (1/3)

- By plotting the model ***training*** and ***validation accuracies*** as functions of the training set size, we can easily detect whether the model suffers from high variance or high bias, and whether the collection of more data could help address the problem



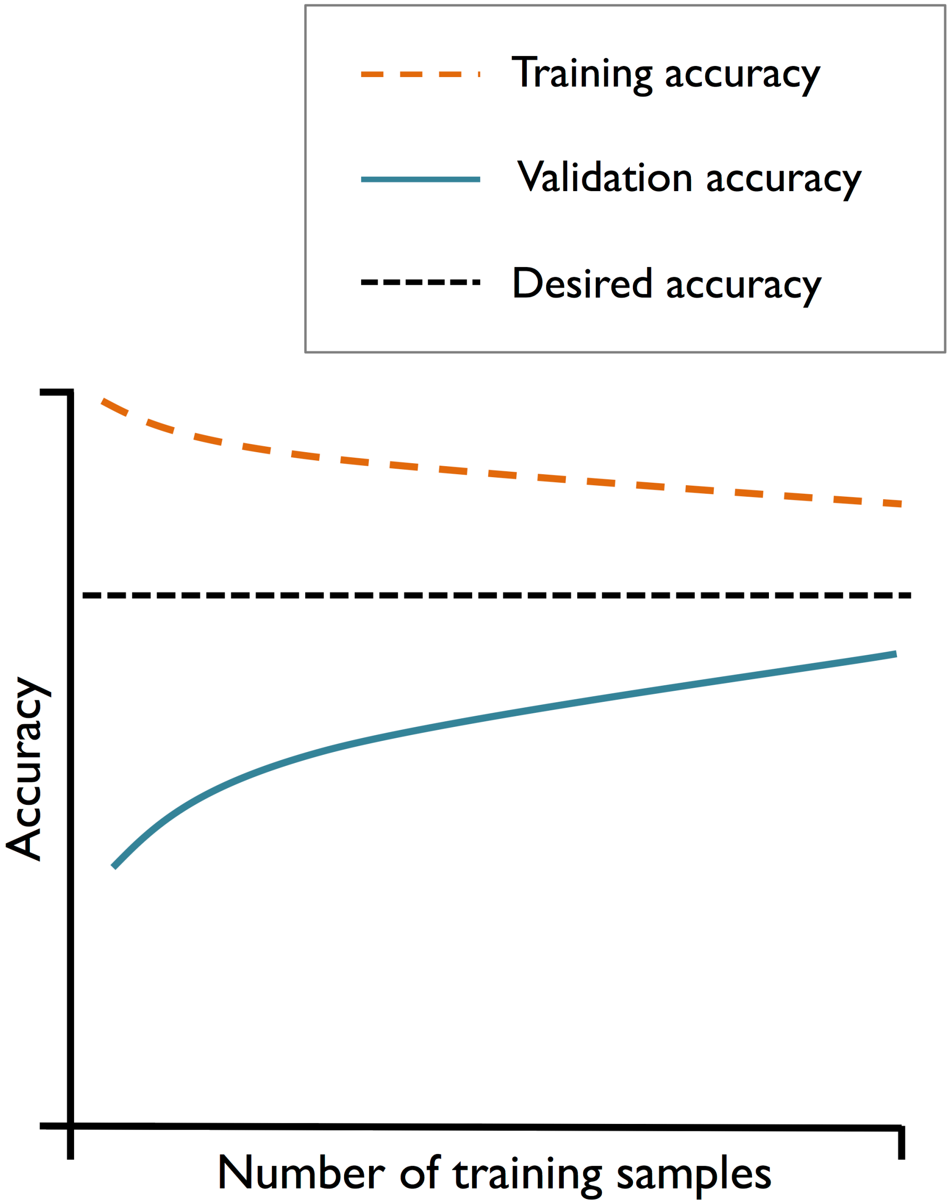
Diagnosing Bias and Variance Problems with Learning Curves (2/3)

- High bias
  - Model underfits the training data
  - Possible solutions
    - Increase the number of parameters of the model
    - Decrease the degree of regularization

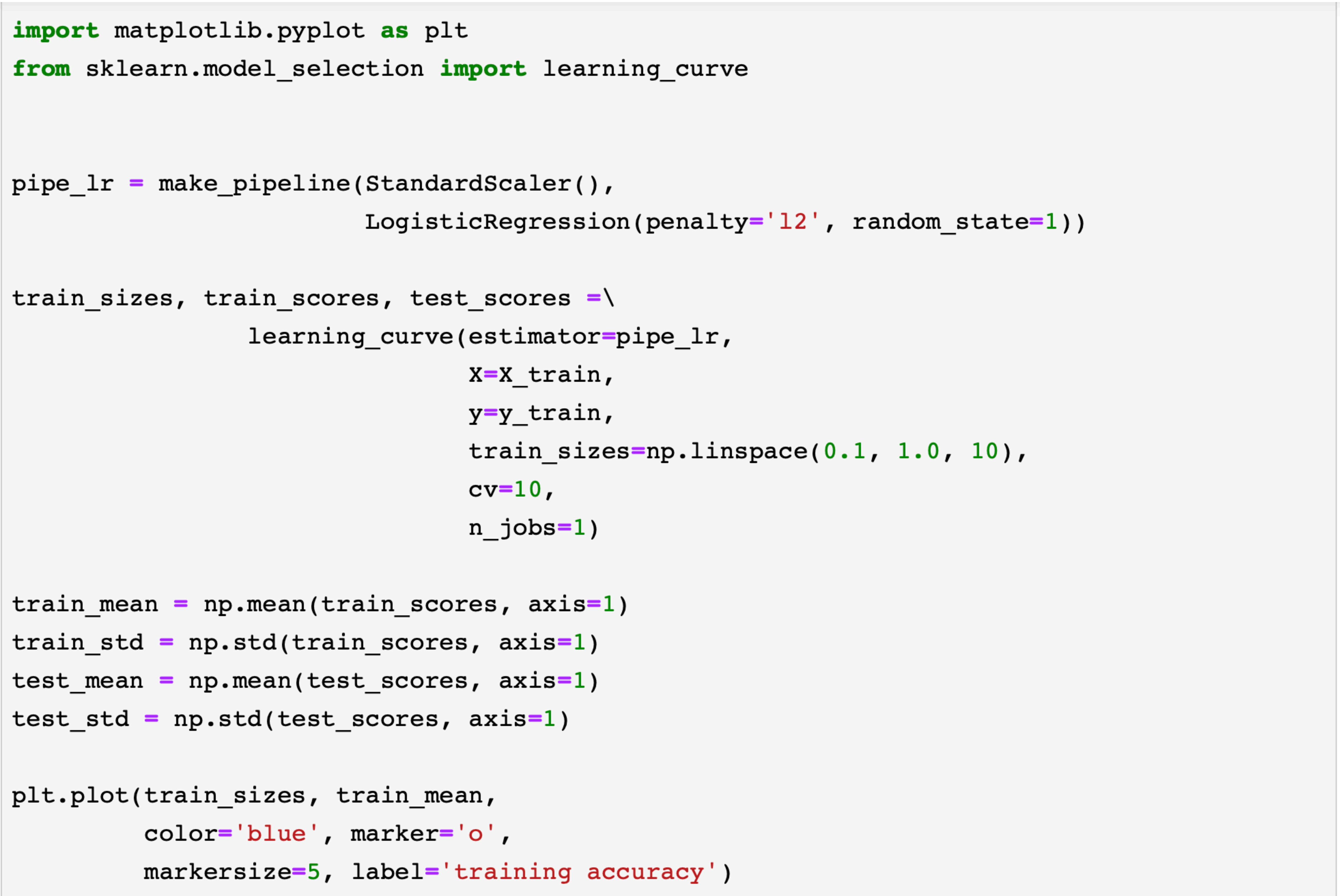


Diagnosing Bias and Variance Problems with Learning Curves (3/3)

- High variance
  - Large gap between training and cross-validation accuracy
  - Possible solutions
    - Collect more training data, reduce the complexity of the model
    - Increase the regularization parameter
    - For unregularized models, decrease the number of features via feature selection or feature extraction
    - However, collect more training data may not always help (noisy training data)

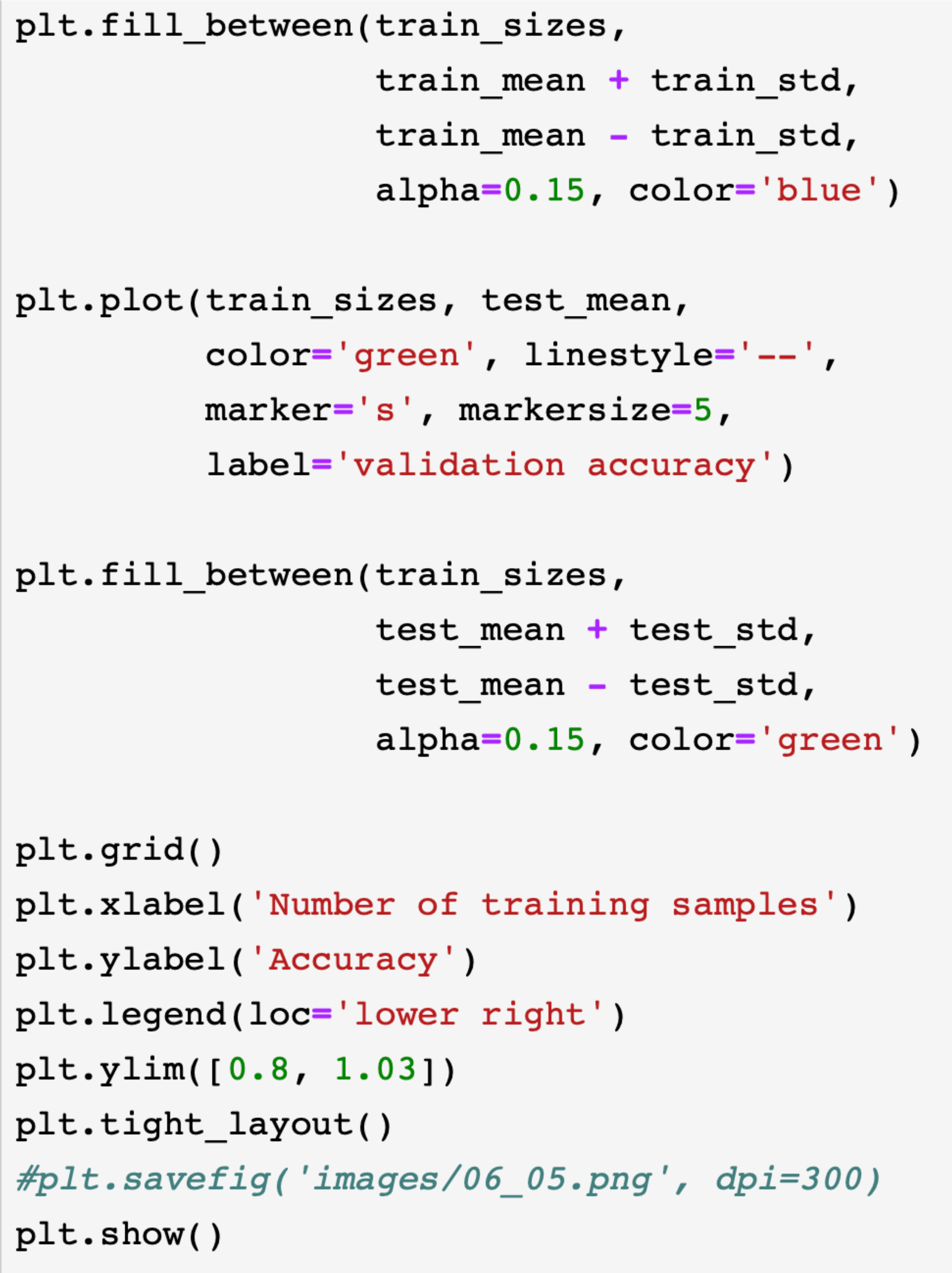


Learning Curves with Scikit-learn (1/2)

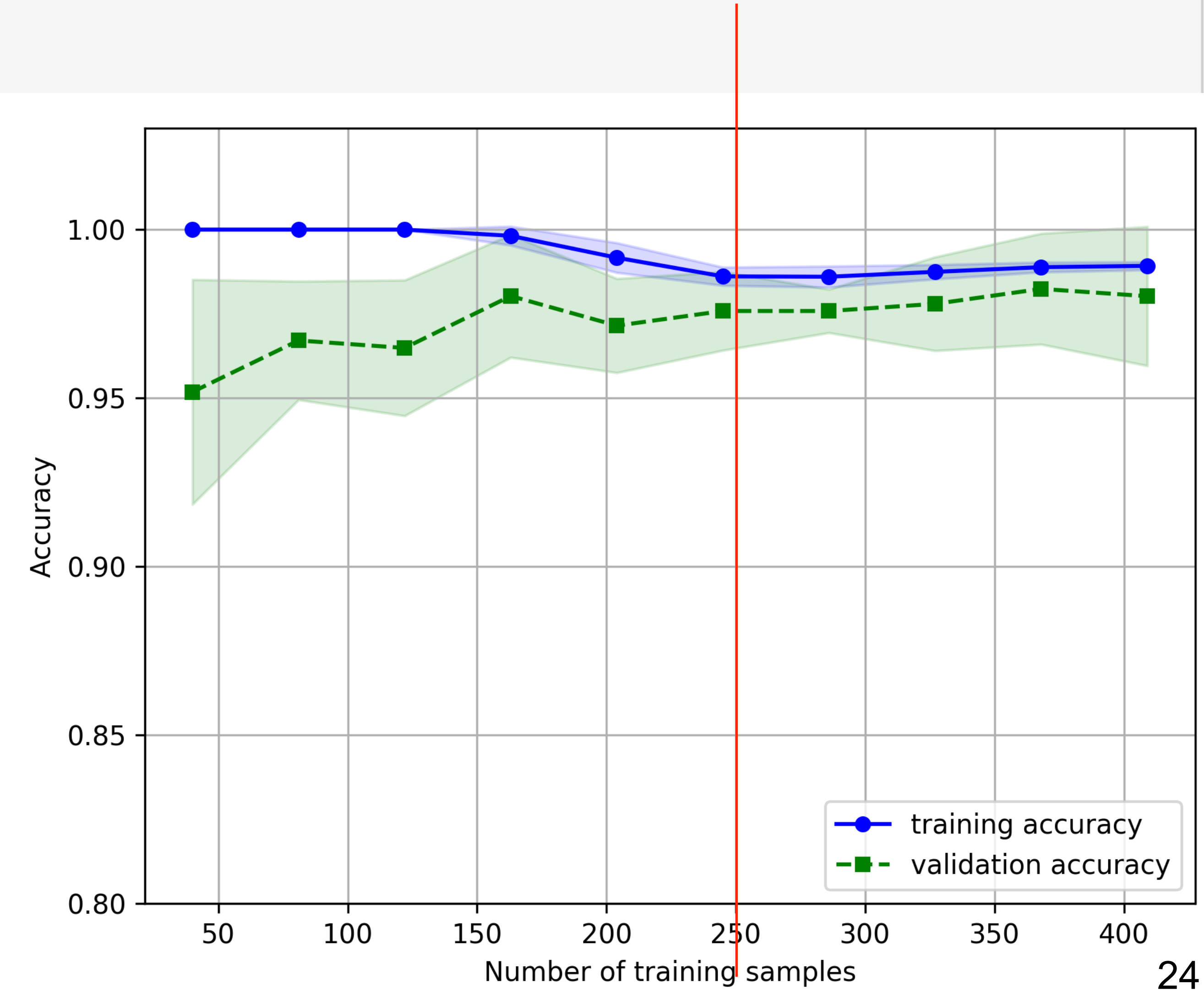


```
import matplotlib.pyplot as plt
from sklearn.model_selection import learning_curve



pipe_lr = make_pipeline(StandardScaler(),
                        LogisticRegression(penalty='l2', random_state=1))

train_sizes, train_scores, test_scores =\
                learning_curve(estimator=pipe_lr,
                               X=X_train,
                               y=y_train,
                               train_sizes=np.linspace(0.1, 1.0, 10),
                               cv=10,
                               n_jobs=1)

train_mean = np.mean(train_scores, axis=1)
train_std = np.std(train_scores, axis=1)
test_mean = np.mean(test_scores, axis=1)
test_std = np.std(test_scores, axis=1)

plt.plot(train_sizes, train_mean,
         color='blue', marker='o',
         markersize=5, label='training accuracy')
```

Learning Curves with Scikit-learn (2/2)

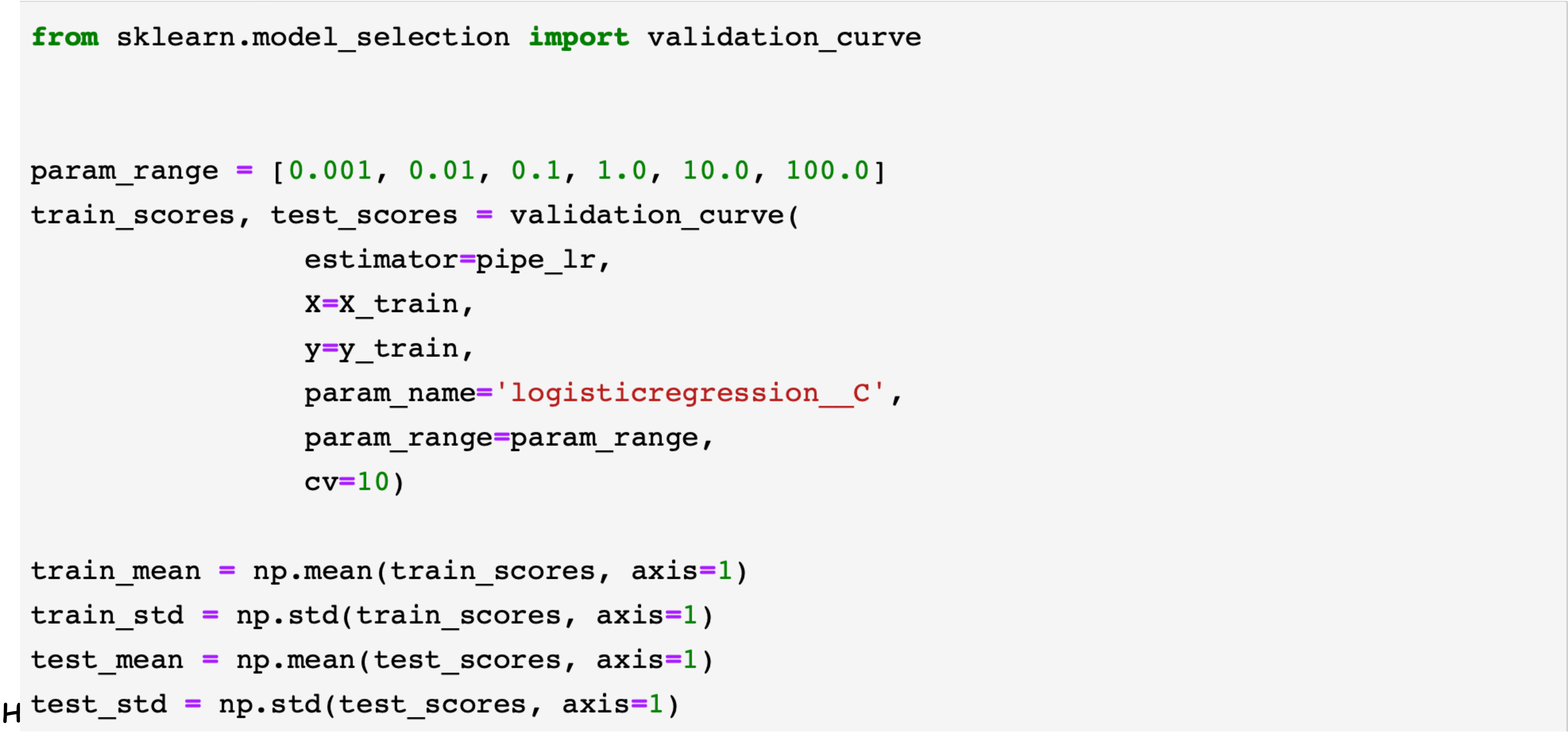


```
plt.fill_between(train_sizes,
                 train_mean + train_std,
                 train_mean - train_std,
                 alpha=0.15, color='blue')

plt.plot(train_sizes, test_mean,
         color='green', linestyle='--',
         marker='s', markersize=5,
         label='validation accuracy')

plt.fill_between(train_sizes,
                 test_mean + test_std,
                 test_mean - test_std,
                 alpha=0.15, color='green')

plt.grid()
plt.xlabel('Number of training samples')
plt.ylabel('Accuracy')
plt.legend(loc='lower right')
plt.ylim([0.8, 1.03])
plt.tight_layout()
#plt.savefig('images/06_05.png', dpi=300)
plt.show()
```



Address Over- and Under-fitting with Validation Curves (1/2)

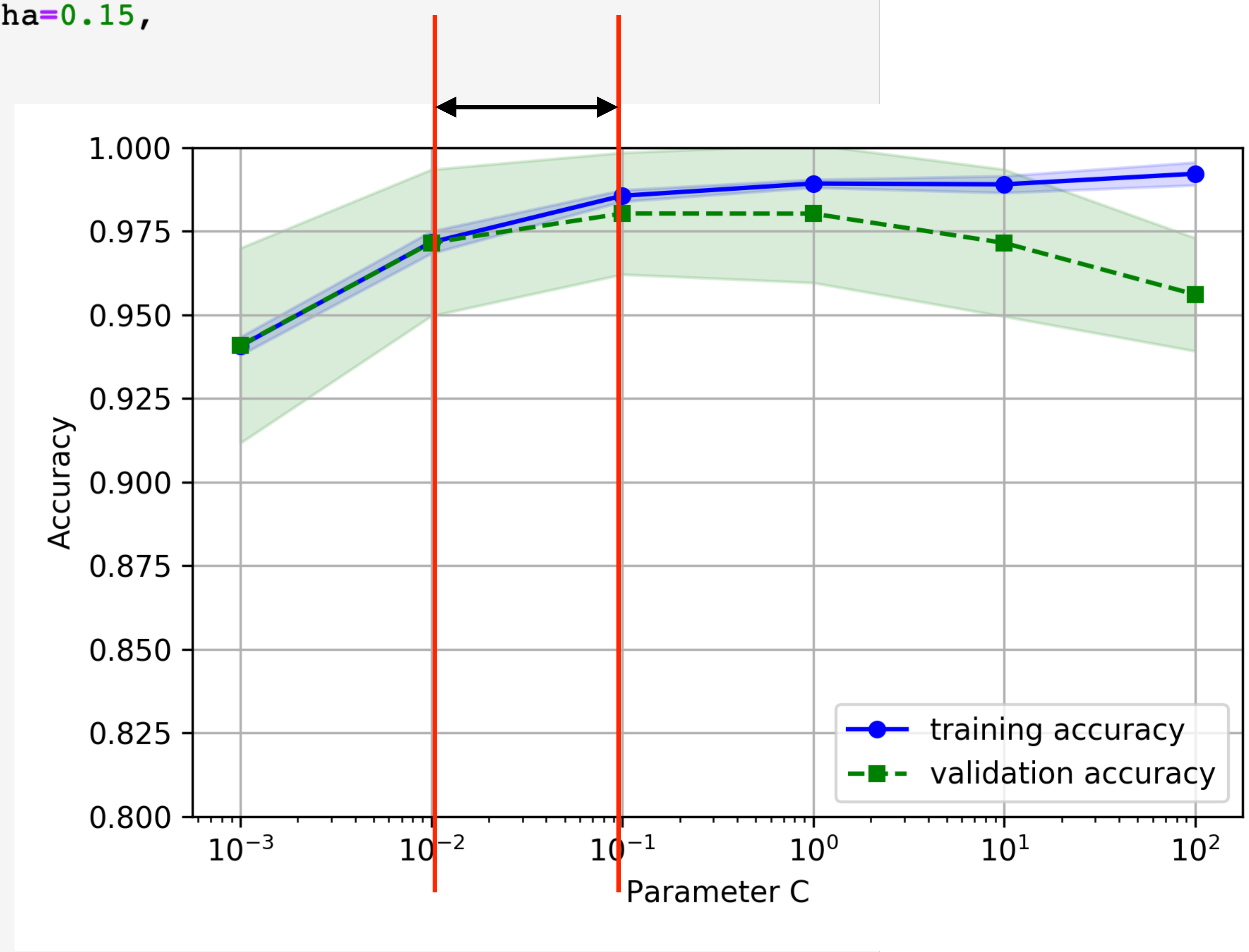
- The validation curves are the figures of the model training and validation accuracies as functions of the *model parameters*



```
from sklearn.model_selection import validation_curve



param_range = [0.001, 0.01, 0.1, 1.0, 10.0, 100.0]
train_scores, test_scores = validation_curve(
                estimator=pipe_lr,
                X=X_train,
                y=y_train,
                param_name='logisticregression__C',
                param_range=param_range,
                cv=10)

train_mean = np.mean(train_scores, axis=1)
train_std = np.std(train_scores, axis=1)
test_mean = np.mean(test_scores, axis=1)
test_std = np.std(test_scores, axis=1)
```

Address Over- and Under-fitting with Validation Curves (2/2)

```
plt.plot(param_range, train_mean,
         color='blue', marker='o',
         markersize=5, label='training accuracy')

plt.fill_between(param_range, train_mean + train_std,
                 train_mean - train_std, alpha=0.15,
                 color='blue')

plt.plot(param_range, test_mean,
         color='green', linestyle='--',
         marker='s', markersize=5,
         label='validation accuracy')

plt.fill_between(param_range,
                 test_mean + test_std,
                 test_mean - test_std,
                 alpha=0.15, color='green')

plt.grid()
plt.xscale('log')
plt.legend(loc='lower right')
plt.xlabel('Parameter C')
plt.ylabel('Accuracy')
plt.ylim([0.8, 1.0])
plt.tight_layout()
# plt.savefig('images/06_06.png', dpi=300)
plt.show()
```

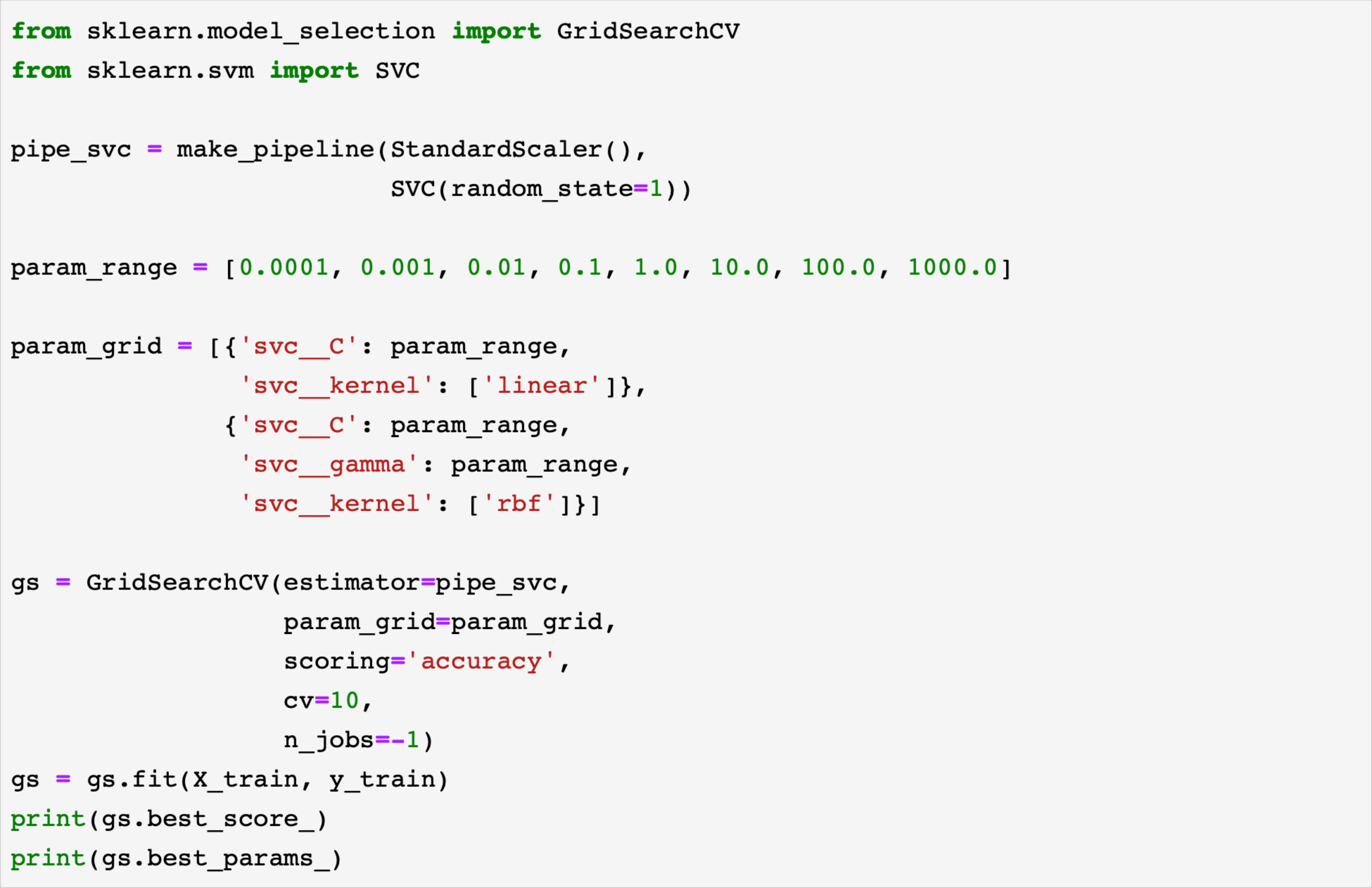


Fine-Tuning Machine Learning Models via Grid Search

Tuning Hyperparameters via Grid Search

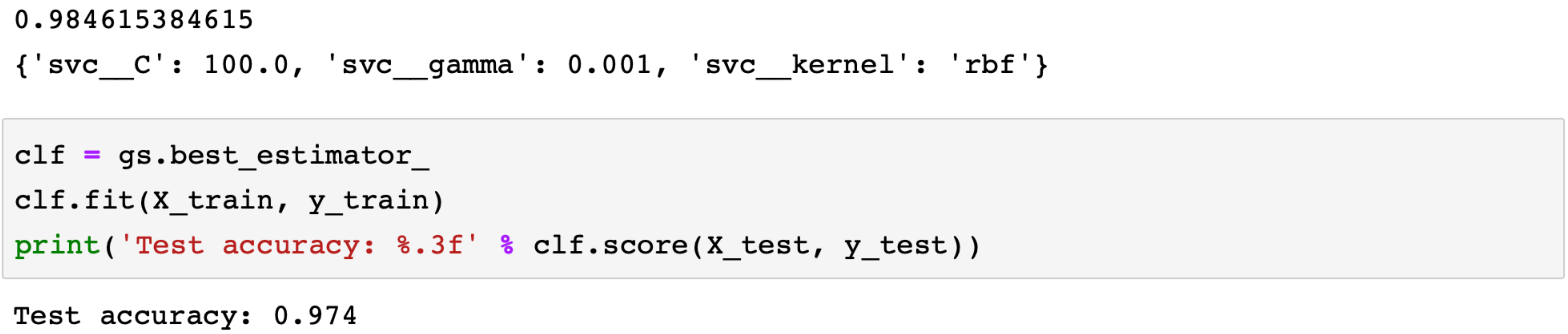
- Grid search is a brute-force exhaustive search paradigm
- A list of values is specified for each hyperparameter
- We evaluate the model performance for each possible combination of those listed values to obtain the optimal combination of values

Tuning Hyperparameters via Grid Search



```
from sklearn.model_selection import GridSearchCV
from sklearn.svm import SVC

pipe_svc = make_pipeline(StandardScaler(),
                         SVC(random_state=1))

param_range = [0.0001, 0.001, 0.01, 0.1, 1.0, 10.0, 100.0, 1000.0]

param_grid = [{'svc__C': param_range,
               'svc__kernel': ['linear']},
              {'svc__C': param_range,
               'svc__gamma': param_range,
               'svc__kernel': ['rbf']}]

gs = GridSearchCV(estimator=pipe_svc,
                  param_grid=param_grid,
                  scoring='accuracy',
                  cv=10,
                  n_jobs=-1)
gs = gs.fit(X_train, y_train)
print(gs.best_score_)
print(gs.best_params_)
```

Tuning Hyperparameters via Grid Search



```
0.984615384615
{'svc__C': 100.0, 'svc__gamma': 0.001, 'svc__kernel': 'rbf'}
```

```
clf = gs.best_estimator_
clf.fit(X_train, y_train)
print('Test accuracy: %.3f' % clf.score(X_test, y_test))
```

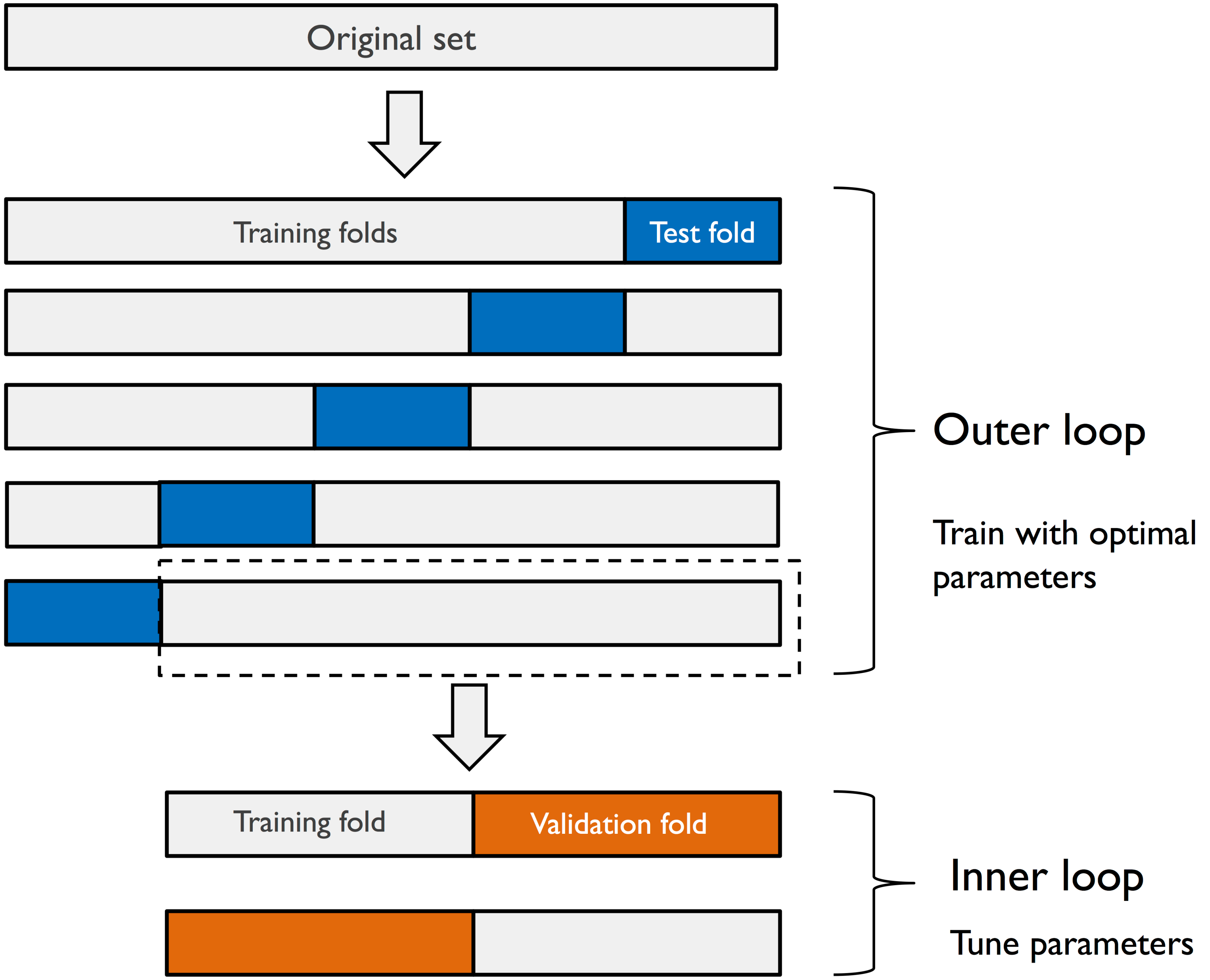
```
Test accuracy: 0.974
```

Algorithm Selection with Nested Cross-Validation

- In a nested cross-validation, we have two loops
  - one outer *k*-fold cross-validation loop to split the data into training and test folds
  - one inner *k*-fold cross-validation loop to select the model using *k*-fold cross-validation on the training folds,
  - after model selection, use the test fold to evaluate model performance
  - the two loops may have different value of *k*
- Used to evaluate the generalization performance of different classification algorithms in order to select the best one

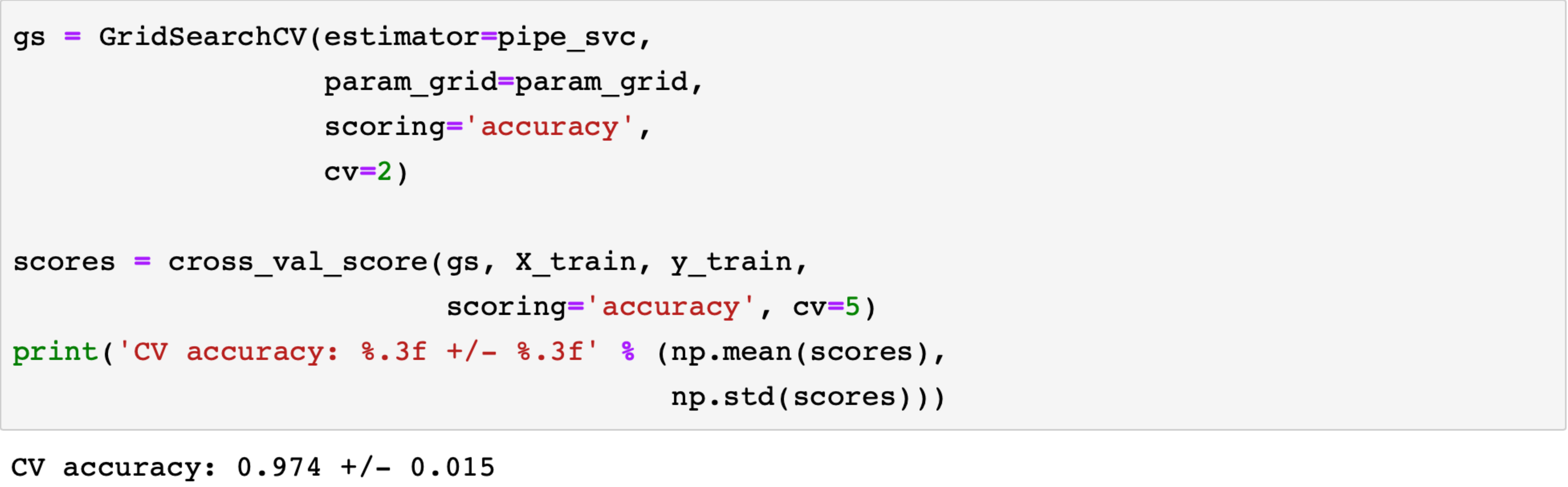
Algorithm Selection with Nested Cross-Validation

- 5x2 cross-validation



Algorithm Selection with Nested Cross-Validation

- Compare SVM and decision tree classifier with only depth parameter
- SVM classifier

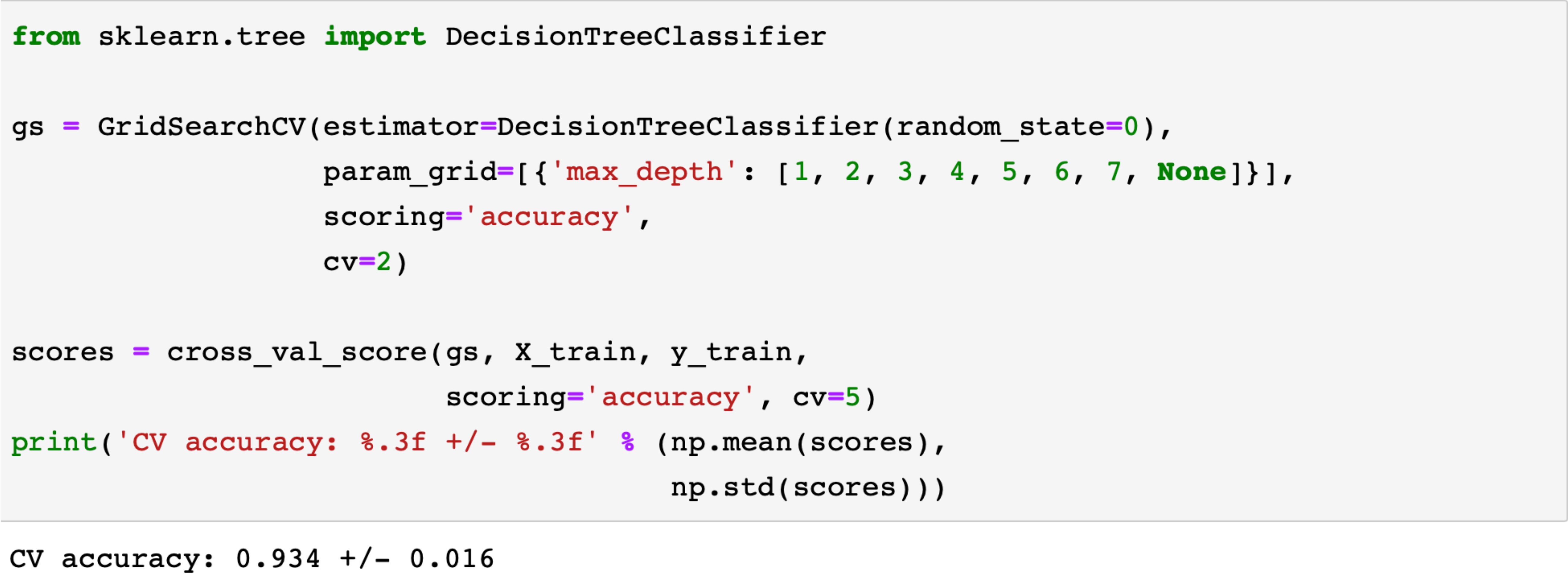


```
gs = GridSearchCV(estimator=pipe_svc,
                  param_grid=param_grid,
                  scoring='accuracy',
                  cv=2)

scores = cross_val_score(gs, X_train, y_train,
                         scoring='accuracy', cv=5)
print('CV accuracy: %.3f +/- %.3f' % (np.mean(scores),
                                      np.std(scores)))
```

```
CV accuracy: 0.974 +/- 0.015
```

Algorithm Selection with Nested Cross-Validation

- Decision tree classifier



```
from sklearn.tree import DecisionTreeClassifier

gs = GridSearchCV(estimator=DecisionTreeClassifier(random_state=0),
                  param_grid=[{'max_depth': [1, 2, 3, 4, 5, 6, 7, None]}],
                  scoring='accuracy',
                  cv=2)

scores = cross_val_score(gs, X_train, y_train,
                         scoring='accuracy', cv=5)
print('CV accuracy: %.3f +/- %.3f' % (np.mean(scores),
                                      np.std(scores)))
```

```
CV accuracy: 0.934 +/- 0.016
```

- SVM model (97.4%) is better then decision tree model (93.4%)

Looking at Different Performance Evaluation Metrics

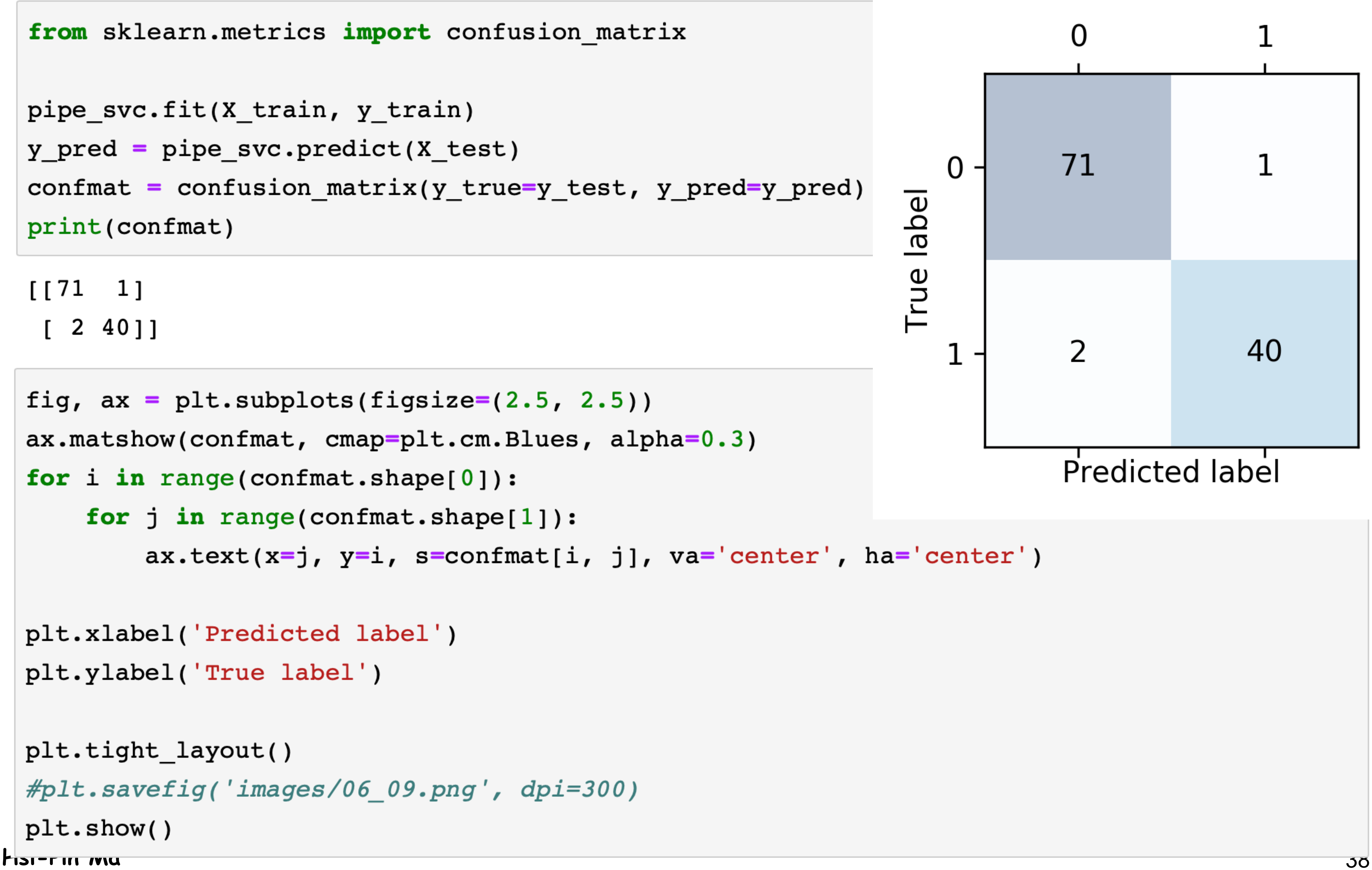
Confusion Matrix

- Accuracy can be misleading for imbalanced datasets
- Need ways to compute performance for a specific class
- Confusion matrix helps to visualize different types of errors a classifier can make by reporting the counts of these errors

Confusion Matrix

		Predicted class	
		P	N
Actual class	P	True positives (TP)	False negatives (FN) Type II error
	N	False positives (FP) Type I error	True negatives (TN)

Confusion Matrix

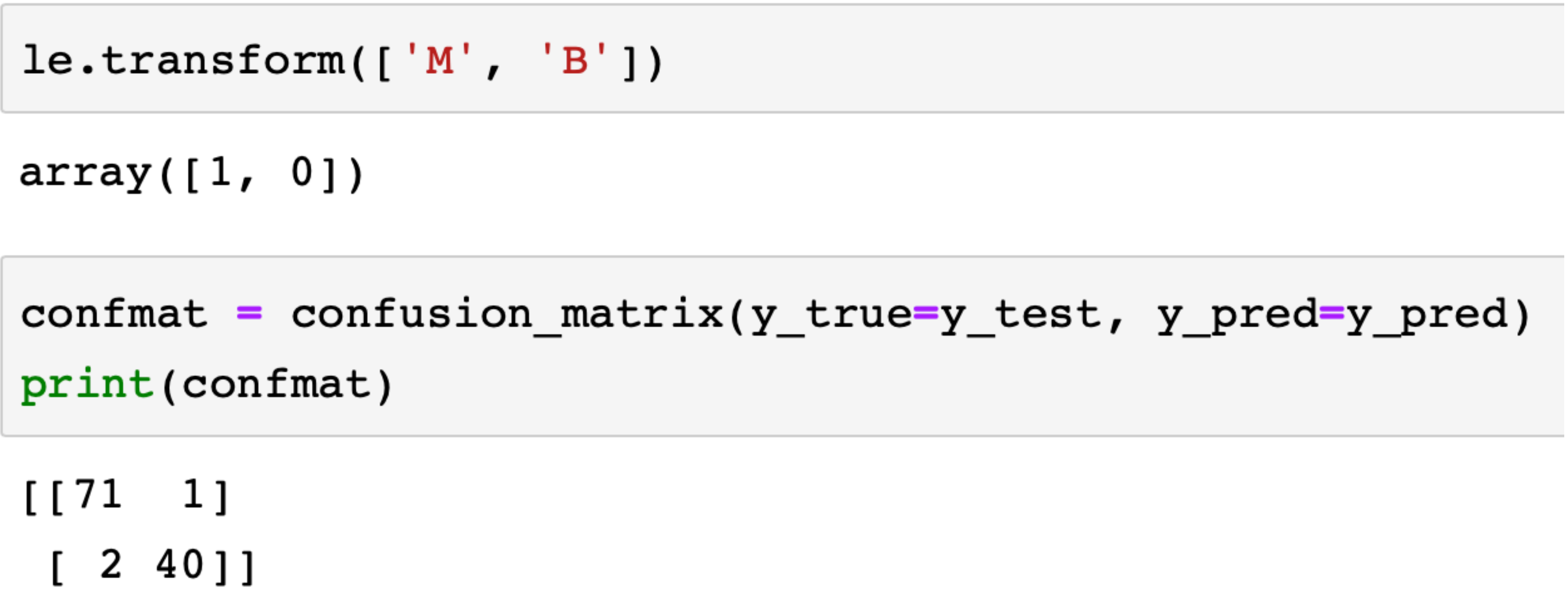


```
from sklearn.metrics import confusion_matrix

pipe_svc.fit(X_train, y_train)
y_pred = pipe_svc.predict(X_test)
confmat = confusion_matrix(y_true=y_test, y_pred=y_pred)
print(confmat)
```

```
[[71  1]
 [ 2 40]]
```

```
fig, ax = plt.subplots(figsize=(2.5, 2.5))
ax.matshow(confmat, cmap=plt.cm.Blues, alpha=0.3)
for i in range(confmat.shape[0]):
    for j in range(confmat.shape[1]):
        ax.text(x=j, y=i, s=confmat[i, j], va='center', ha='center')

plt.xlabel('Predicted label')
plt.ylabel('True label')

plt.tight_layout()
#plt.savefig('images/06_09.png', dpi=300)
plt.show()
```

Confusion Matrix

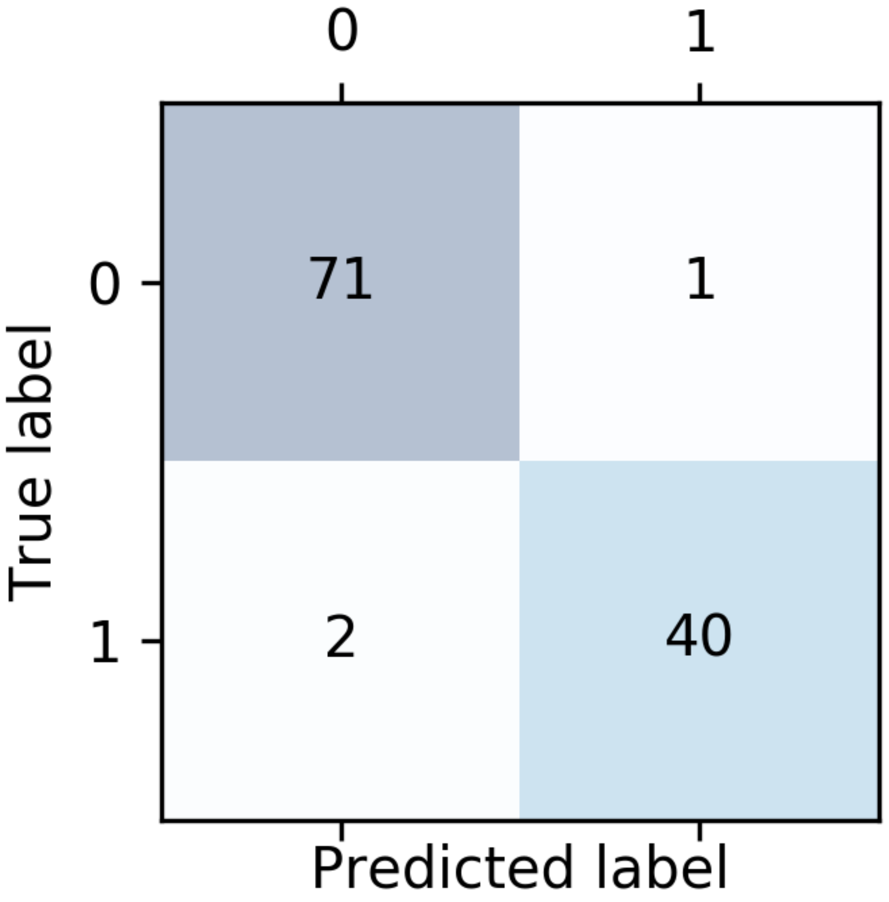


```
le.transform(['M', 'B'])
```

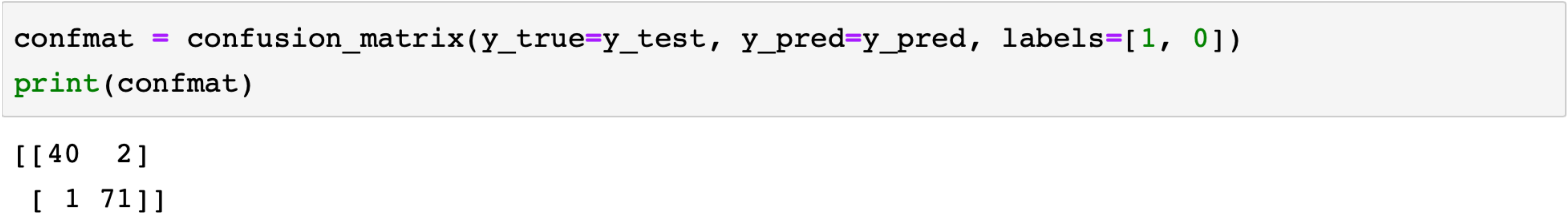
```
array([1, 0])
```

```
confmat = confusion_matrix(y_true=y_test, y_pred=y_pred)
print(confmat)
```

```
[[71  1]
 [ 2 40]]
```



- Change the order of the class label



```
confmat = confusion_matrix(y_true=y_test, y_pred=y_pred, labels=[1, 0])
print(confmat)
```

```
[[40  2]
 [ 1 71]]
```

Optimizing the Precision and Recall of a Classification Model

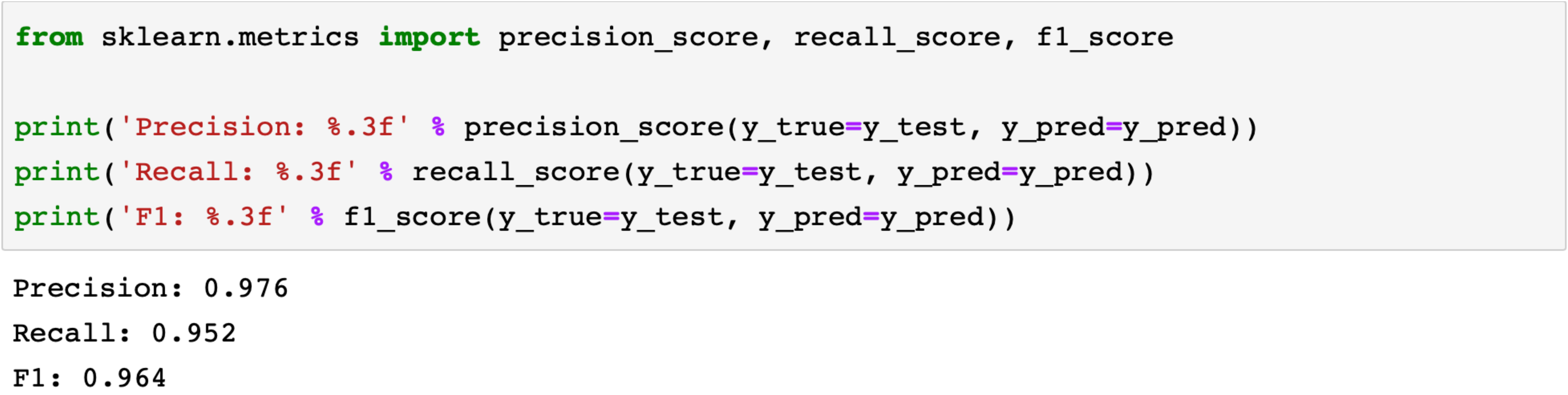
- Still quite a few different performance evaluation metrics
  - Prediction error (ERR) $ERR = \frac{FP + FN}{FP + FN + TP + TN}$
  - Accuracy (ACC) $ACC = \frac{TP + TN}{FP + FN + TP + TN} = 1 - ERR$
  - True positive rate (TPR) $TPR = \frac{TP}{P} = \frac{TP}{FN + TP}$
  - False positive rate (FPR) $FPR = \frac{FP}{N} = \frac{FP}{FP + TN}$ **1-specificity**
  - Precision (PRE) $PRE = \frac{TP}{TP + FP}$
  - Recall (REC) $REC = TPR = \frac{TP}{P} = \frac{TP}{FN + TP}$ **sensitivity**
  - F1-score (F1) $F1 = 2\frac{PRE \times REC}{PRE + REC}$

$$\text{Specificity} = \frac{\text{TN}}{\text{FP} + \text{TN}}$$

$$\text{Prevalence} = \frac{\text{TP} + \text{FN}}{\text{Total N}}$$

Optimizing the Precision and Recall of a Classification Model

- Scoring metrics are all implemented in scikit-learn

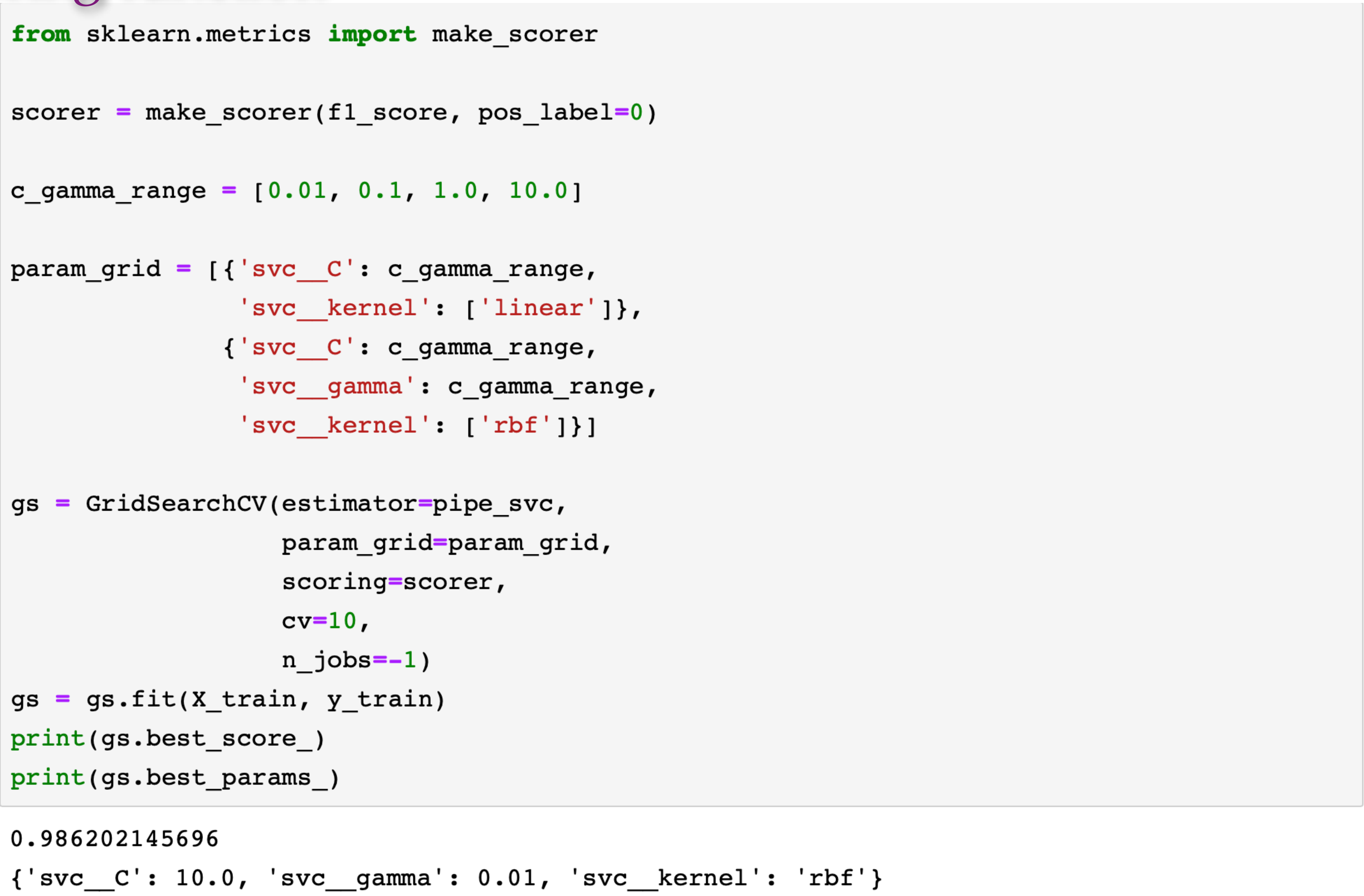


```python
from sklearn.metrics import precision_score, recall_score, f1_score


print('Precision: %.3f' % precision_score(y_true=y_test, y_pred=y_pred))
print('Recall: %.3f' % recall_score(y_true=y_test, y_pred=y_pred))
print('F1: %.3f' % f1_score(y_true=y_test, y_pred=y_pred))
```

```
Precision: 0.976
Recall: 0.952
F1: 0.964
```

Optimizing the Precision and Recall of a Classification Model

- We can use the **make_scorer** function in scikit-learn's **metrics** module to designate our own positive label and scoring function



```
from sklearn.metrics import make_scorer

scorer = make_scorer(f1_score, pos_label=0)

c_gamma_range = [0.01, 0.1, 1.0, 10.0]

param_grid = [{'svc__C': c_gamma_range,
               'svc__kernel': ['linear']},
              {'svc__C': c_gamma_range,
               'svc__gamma': c_gamma_range,
               'svc__kernel': ['rbf']}]

gs = GridSearchCV(estimator=pipe_svc,
                  param_grid=param_grid,
                  scoring=scorer,
                  cv=10,
                  n_jobs=-1)
gs = gs.fit(X_train, y_train)
print(gs.best_score_)
print(gs.best_params_)
```

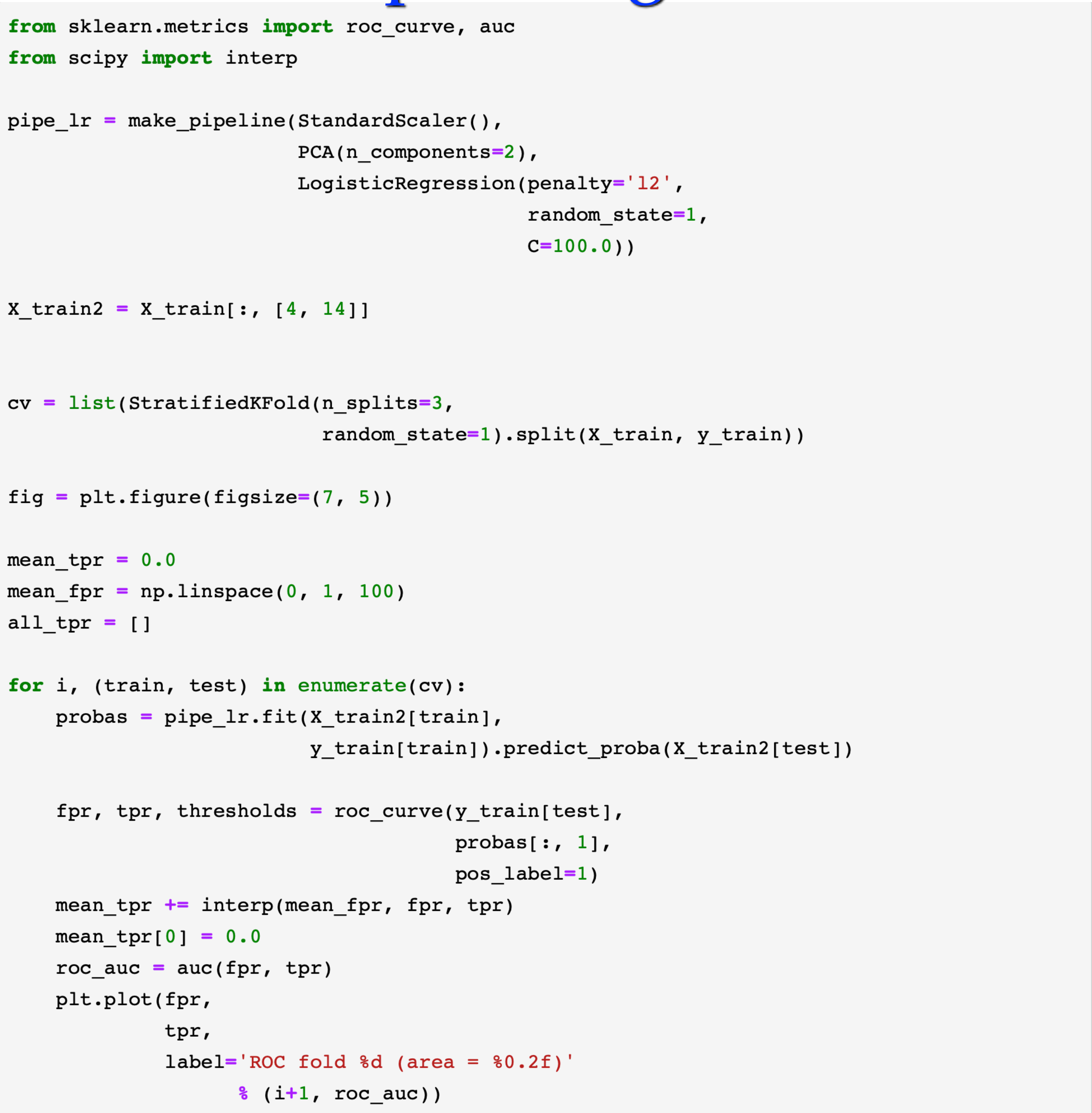
```
0.986202145696
{'svc__C': 10.0, 'svc__gamma': 0.01, 'svc__kernel': 'rbf'}
```

Receiving Operating Characteristic (ROC)

- A tool to select models for classification based on performance with respect to FPR and TPR
  - FPR and TPR are computed by shifting the decision threshold of the classifier
- The diagonal of an ROC graph can be interpreted as *random guessing*
  - Classification models that fall below the diagonal are considered as worse than random guessing
- A perfect classifier would fall into the top left corner
- ROC Area Under the Curve (ROC AUC) to characterize the performance

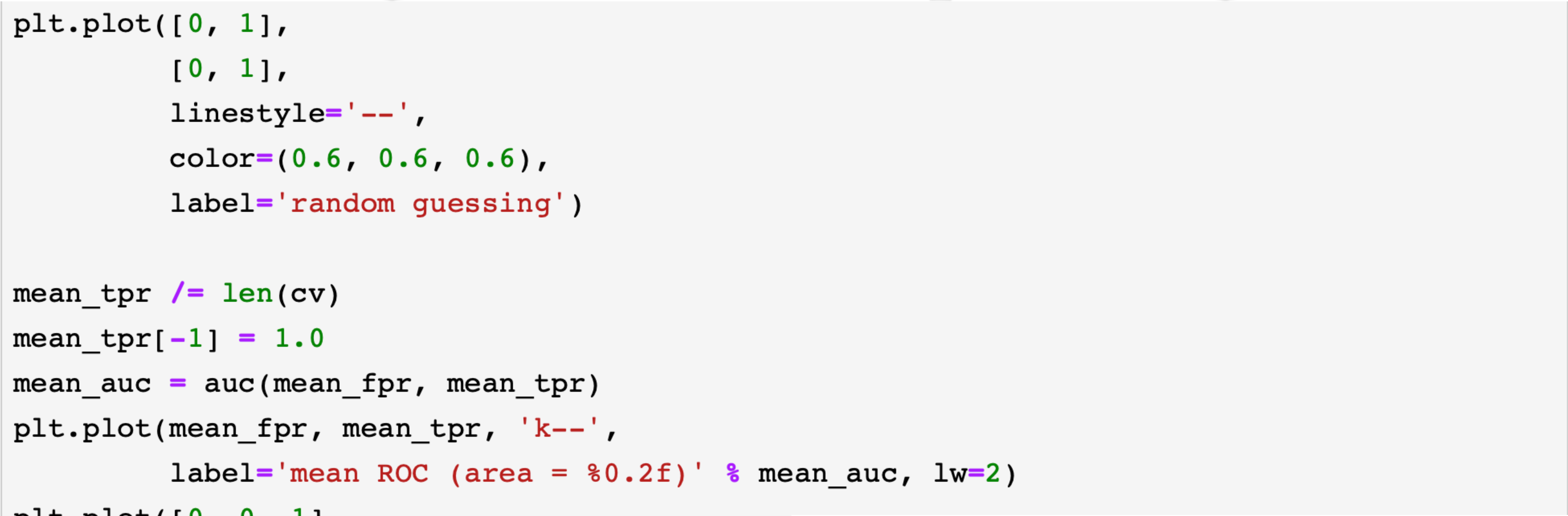
Plotting a Receiver Operating Characteristic

- TPR vs. FPR
  - ROC AUC

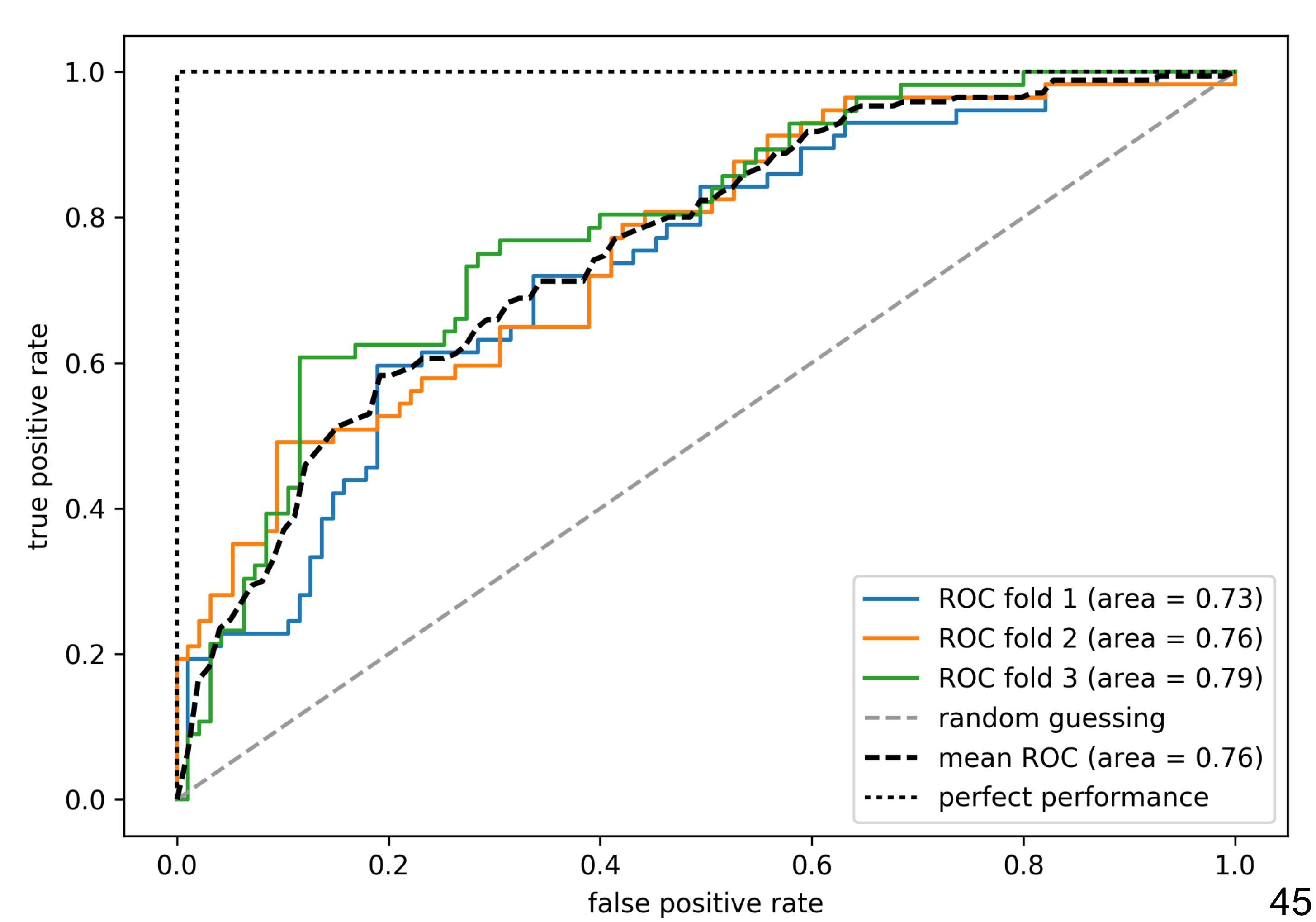


```
from sklearn.metrics import roc_curve, auc
from scipy import interp

pipe_lr = make_pipeline(StandardScaler(),
                        PCA(n_components=2),
                        LogisticRegression(penalty='l2',
                                           random_state=1,
                                           C=100.0))

X_train2 = X_train[:, [4, 14]]


cv = list(StratifiedKFold(n_splits=3,
                          random_state=1).split(X_train, y_train))

fig = plt.figure(figsize=(7, 5))

mean_tpr = 0.0
mean_fpr = np.linspace(0, 1, 100)
all_tpr = []

for i, (train, test) in enumerate(cv):
    probas = pipe_lr.fit(X_train2[train],
                         y_train[train]).predict_proba(X_train2[test])

    fpr, tpr, thresholds = roc_curve(y_train[test],
                                     probas[:, 1],
                                     pos_label=1)
    mean_tpr += interp(mean_fpr, fpr, tpr)
    mean_tpr[0] = 0.0
    roc_auc = auc(fpr, tpr)
    plt.plot(fpr,
             tpr,
             label='ROC fold %d (area = %0.2f)'
                   % (i+1, roc_auc))
```

Plotting a Receiver Operating Characteristic



```
plt.plot([0, 1],
         [0, 1],
         linestyle='--',
         color=(0.6, 0.6, 0.6),
         label='random guessing')

mean_tpr /= len(cv)
mean_tpr[-1] = 1.0
mean_auc = auc(mean_fpr, mean_tpr)
plt.plot(mean_fpr, mean_tpr, 'k--',
         label='mean ROC (area = %0.2f)' % mean_auc, lw=2)
plt.plot([0, 0, 1],
         [0, 1, 1],
         linestyle=':',
         color='black',
         label='perfect performance')

plt.xlim([-0.05, 1.05])
plt.ylim([-0.05, 1.05])
plt.xlabel('false positive rate')
plt.ylabel('true positive rate')
plt.legend(loc="lower right")

plt.tight_layout()
# plt.savefig('images/06_10.png', dpi=300)
plt.show()
```



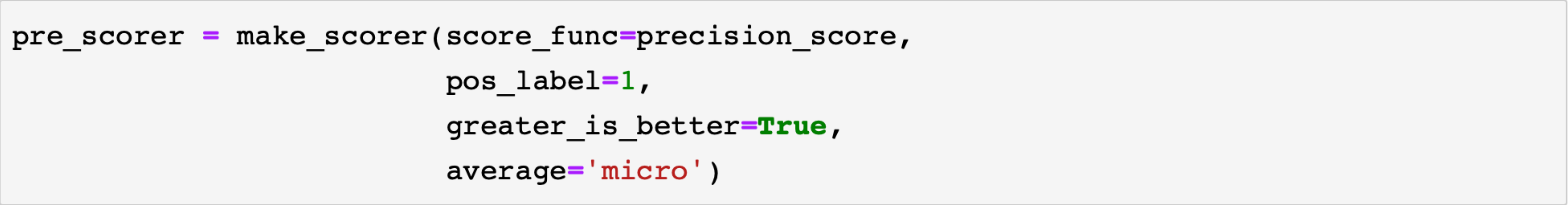
Scoring Metrics for Multiclass Classification

- Scikit-learn implements micro and macro averaging methods to extend the previous scoring metrics to multi class problems via One-versus-All (OvA) classification
- The micro-average of the precision score
  - Calculated from individual TPs, TNs, FPs, and FNs

$$PRE_{micro} = \frac{TP_1 + \cdots + TP_k}{TP_1 + \cdots + TP_k + FP_1 + \cdots + FP_k}$$

- The macro-average of the precision score
  - Calculated as the average scores of different systems

$$PRE_{macro} = \frac{PRE_1 + \cdots + PRE_k}{k}$$



```python
pre_scorer = make_scorer(score_func=precision_score,
                         pos_label=1,
                         greater_is_better=True,
                         average='micro')
```

Dealing with Class Imbalance

Class Imbalance

- Samples from one class or multiple classes are over-represented in a dataset
  - A quite common problem
- For breast cancer dataset with 90% healthy patients
  - If achieve 90% accuracy on the test dataset by just predicting the majority class, without supervised learning, the model does not learn anything from dataset features
- Focus on other metrics than accuracy

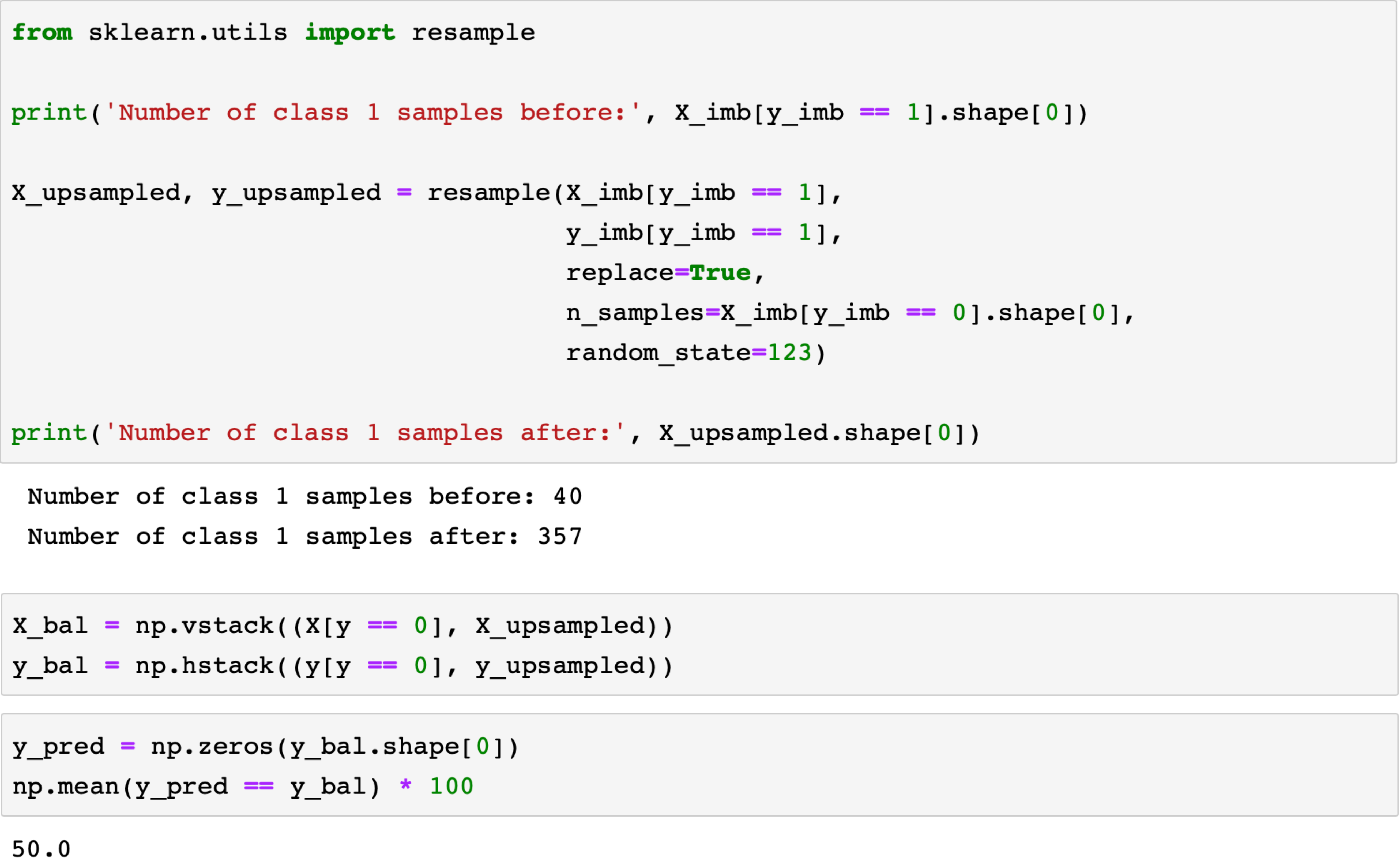
Class Imbalance Examples

- For breast cancer, the priority might be to identify the majority of patients with malignant cancer patients to recommend an additional screening
  - Recall should be the metric of choice
- In spam filtering, where we don't want to label emails as spam if the system is not very certain
  - Precision might be a more appropriate metric

Strategies for Dealing with Class Imbalance

- No universally best solutions
- Possbile solutions
  - Assign a large penalty to wrong predictions on minority class during model fitting
    - Set the **class_weight** parameter to **class_weight=‘balanced’**
  - Upsampling the minority class
    - Use **resample** function
  - Downsampling the majority class
    - Use **resample** function
  - Generation of synthetic training samples

Resample Example



```
from sklearn.utils import resample

print('Number of class 1 samples before:', X_imb[y_imb == 1].shape[0])

X_upsampled, y_upsampled = resample(X_imb[y_imb == 1],
                                    y_imb[y_imb == 1],
                                    replace=True,
                                    n_samples=X_imb[y_imb == 0].shape[0],
                                    random_state=123)

print('Number of class 1 samples after:', X_upsampled.shape[0])
```

```
Number of class 1 samples before: 40
Number of class 1 samples after: 357
```

```
X_bal = np.vstack((X[y == 0], X_upsampled))
y_bal = np.hstack((y[y == 0], y_upsampled))
```

```
y_pred = np.zeros(y_bal.shape[0])
np.mean(y_pred == y_bal) * 100
```

```
50.0
```