6. Evaluation & Tuning

COMP3314
Machine Learning

Motivation

- Evaluation
- Tuning
- Learn about the best practices of building models by fine-tuning the model and evaluating its performance
  - Obtain unbiased estimates of a model's performance
  - Diagnose the common problems of machine learning algorithms
  - Fine-tune machine learning models
  - Evaluate predictive models using different performance metrics

Outline

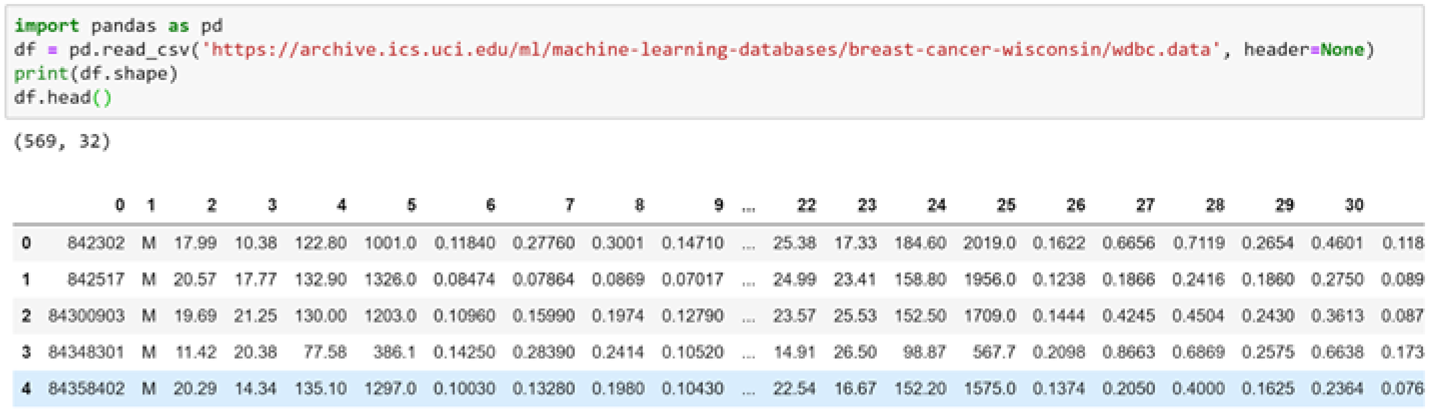
- Pipelining Transformers
- Validation
  - Holdout
  - Cross-Validation
- Learning and Validation Curve
- Hyperparameter Search
- Performance Evaluation Metrics
  - Precision
  - Recall
  - F1-score
  - Receiver Operating Characteristic
- Scoring Metrics for Multiclass Classification
- Class Imbalance

Code - EvaluationAndTuning.ipynb

- Available here on CoLab

Breast Cancer Wisconsin Dataset (BCWD)

- The BCWD contains 569 samples of malignant and benign tumor cells
  - The first two columns in the dataset store the unique ID numbers of the samples and the corresponding diagnoses (M = malignant, B = benign)
  - Columns 3-32 contain 30 real-valued features that have been computed from digitized images of the cell nuclei
- The BCWD has been deposited in the UCI Machine Learning Repository
  - https://archive.ics.uci.edu/ml/datasets/Breast+Cancer+Wisconsin+(Diagnostic)
- Let’s read in the dataset



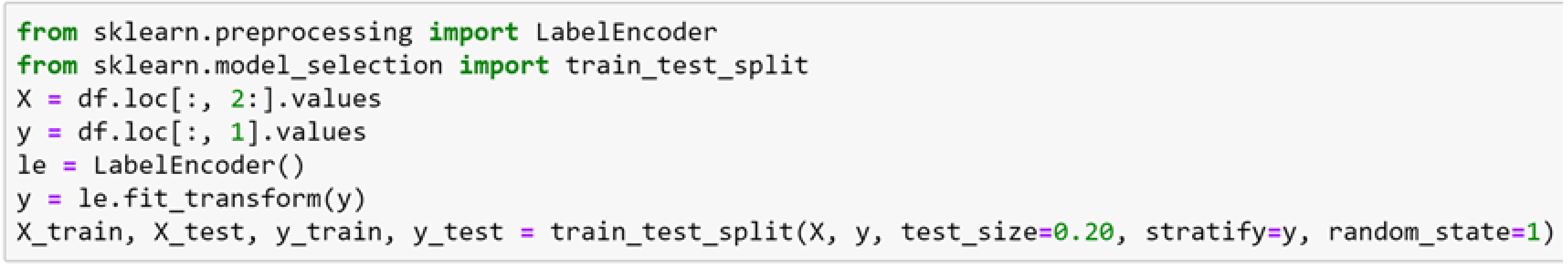
```
import pandas as pd
df = pd.read_csv('https://archive.ics.uci.edu/ml/machine-learning-databases/breast-cancer-wisconsin/wdbc.data', header=None)
print(df.shape)
df.head()
```

```
(569, 32)
```

	0	1	2	3	4	5	6	7	8	9	...	22	23	24	25	26	27	28	29	30	
0	842302	M	17.99	10.38	122.80	1001.0	0.11840	0.27760	0.3001	0.14710	...	25.38	17.33	184.60	2019.0	0.1622	0.6656	0.7119	0.2654	0.4601	0.118
1	842517	M	20.57	17.77	132.90	1326.0	0.08474	0.07864	0.0869	0.07017	...	24.99	23.41	158.80	1956.0	0.1238	0.1866	0.2416	0.1860	0.2750	0.089
2	84300903	M	19.69	21.25	130.00	1203.0	0.10960	0.15990	0.1974	0.12790	...	23.57	25.53	152.50	1709.0	0.1444	0.4245	0.4504	0.2430	0.3613	0.087
3	84348301	M	11.42	20.38	77.58	386.1	0.14250	0.28390	0.2414	0.10520	...	14.91	26.50	98.87	567.7	0.2098	0.8663	0.6869	0.2575	0.6638	0.173
4	84358402	M	20.29	14.34	135.10	1297.0	0.10030	0.13280	0.1980	0.10430	...	22.54	16.67	152.20	1575.0	0.1374	0.2050	0.4000	0.1625	0.2364	0.076

BCWD - Preprocessing

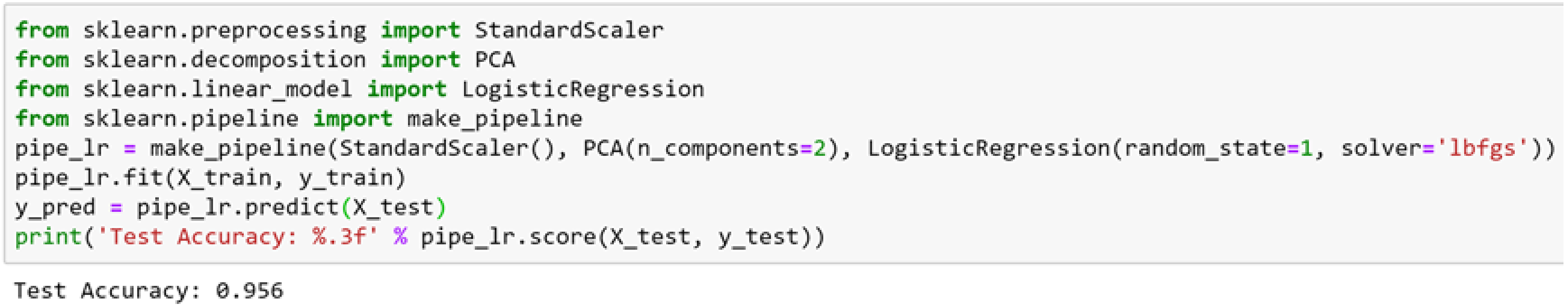
- Next, we assign the 30 features to a NumPy array X
- Using a LabelEncoder object, we transform the class labels from their original string representation ('M' and 'B') into integers
- Then we divide the dataset into a separate training dataset (80%) and a separate test dataset (20%)



```
from sklearn.preprocessing import LabelEncoder
from sklearn.model_selection import train_test_split
X = df.loc[:, 2:].values
y = df.loc[:, 1].values
le = LabelEncoder()
y = le.fit_transform(y)
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.20, stratify=y, random_state=1)
```

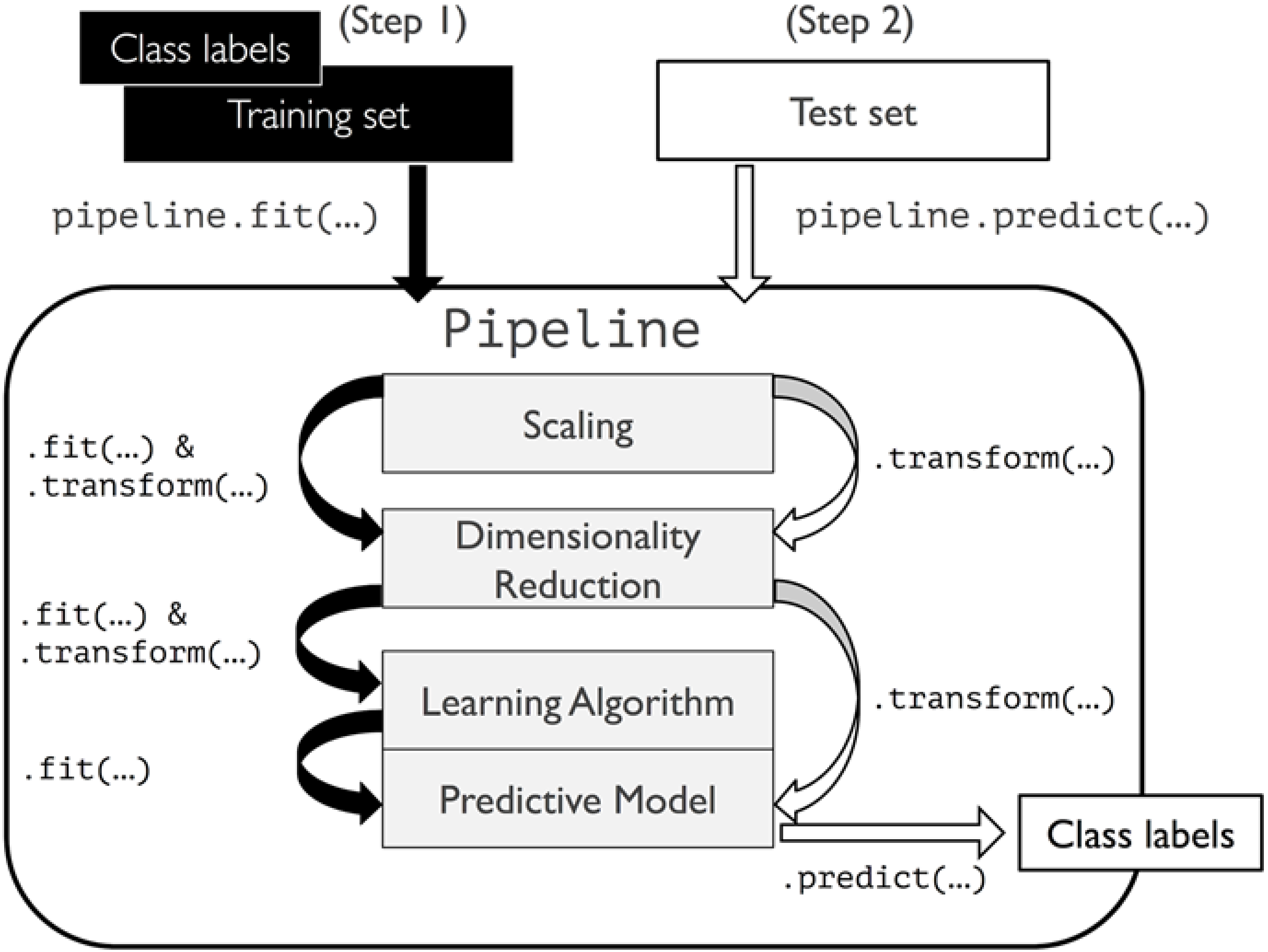
Pipelining Transformers

- Let’s standardize and compress our data from the initial 30 dimensions onto a lower two-dimensional subspace via PCA before feeding it into a logistic regression classifier
- Instead of going through the fitting and transformation steps for the training and test datasets separately, we can chain the StandardScaler, PCA, and LogisticRegression objects in a pipeline
- The Pipeline class in scikit-learn allows us to fit a model including an arbitrary number of transformation steps and apply it to make predictions about new data



```
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, solver='lbfgs'))
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
```


(Step 1)
Class labels
Training set
pipeline.fit(...)
(Step 2)
Test set
pipeline.predict(...)
Pipeline
Scaling
Dimensionality Reduction
Learning Algorithm
Predictive Model
.fit(...) & .transform(...)
.fit(...) & .transform(...)
.fit(...)
.transform(...)
.transform(...)
.predict(...)
Class labels

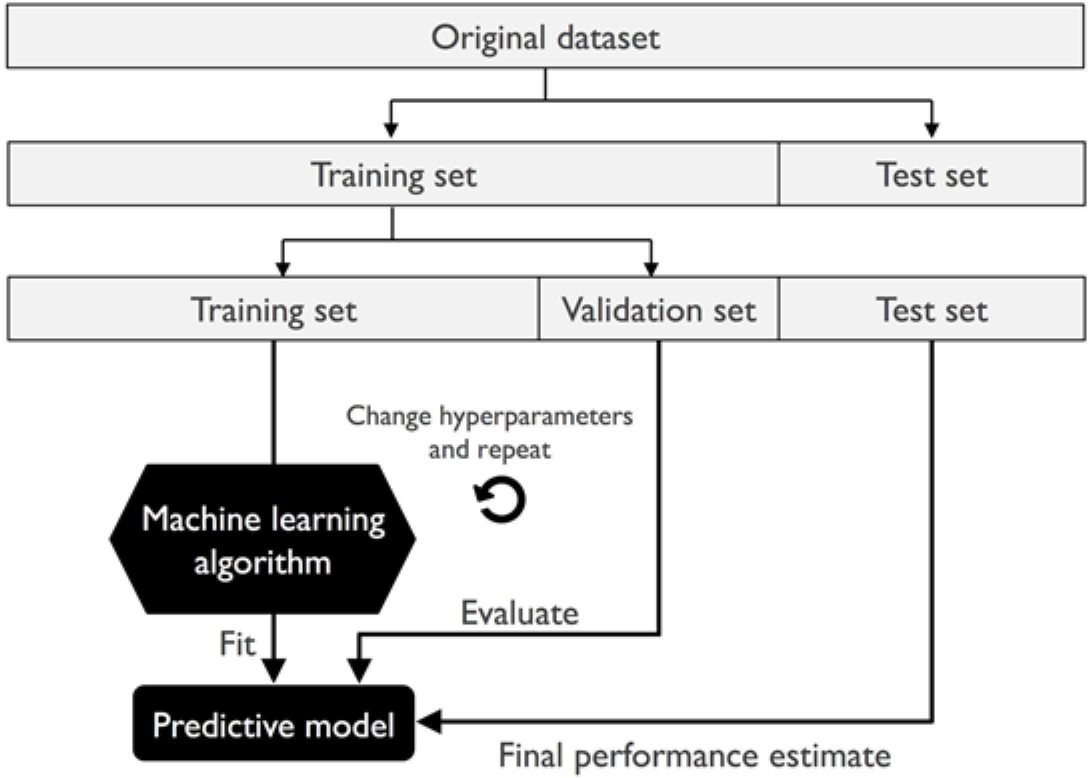
Validation

- One of the key steps in building a machine learning model is to estimate its performance on data that the model hasn't seen before
- To find an acceptable bias-variance trade-off, we need to evaluate our model carefully
- Validation can help us obtain reliable estimates of the model's generalization performance, that is, how well the model performs on unseen data
- We will learn about the common cross-validation techniques
    - Holdout cross-validation
    - K-fold cross-validation

The Holdout Method

- A classic and popular approach
- Split our initial dataset into a separate training and test dataset
    - The former is used for model training, and the latter is used to estimate its generalization performance
- To tune and compare different parameter settings, i.e. to figure out optimal values of hyperparameters, we further split the training set into training and validation
- Advantage of test set that the model hasn't seen before during the training and model selection step:
    - We can obtain a less biased estimate of its ability to generalize to new data

The Holdout Method



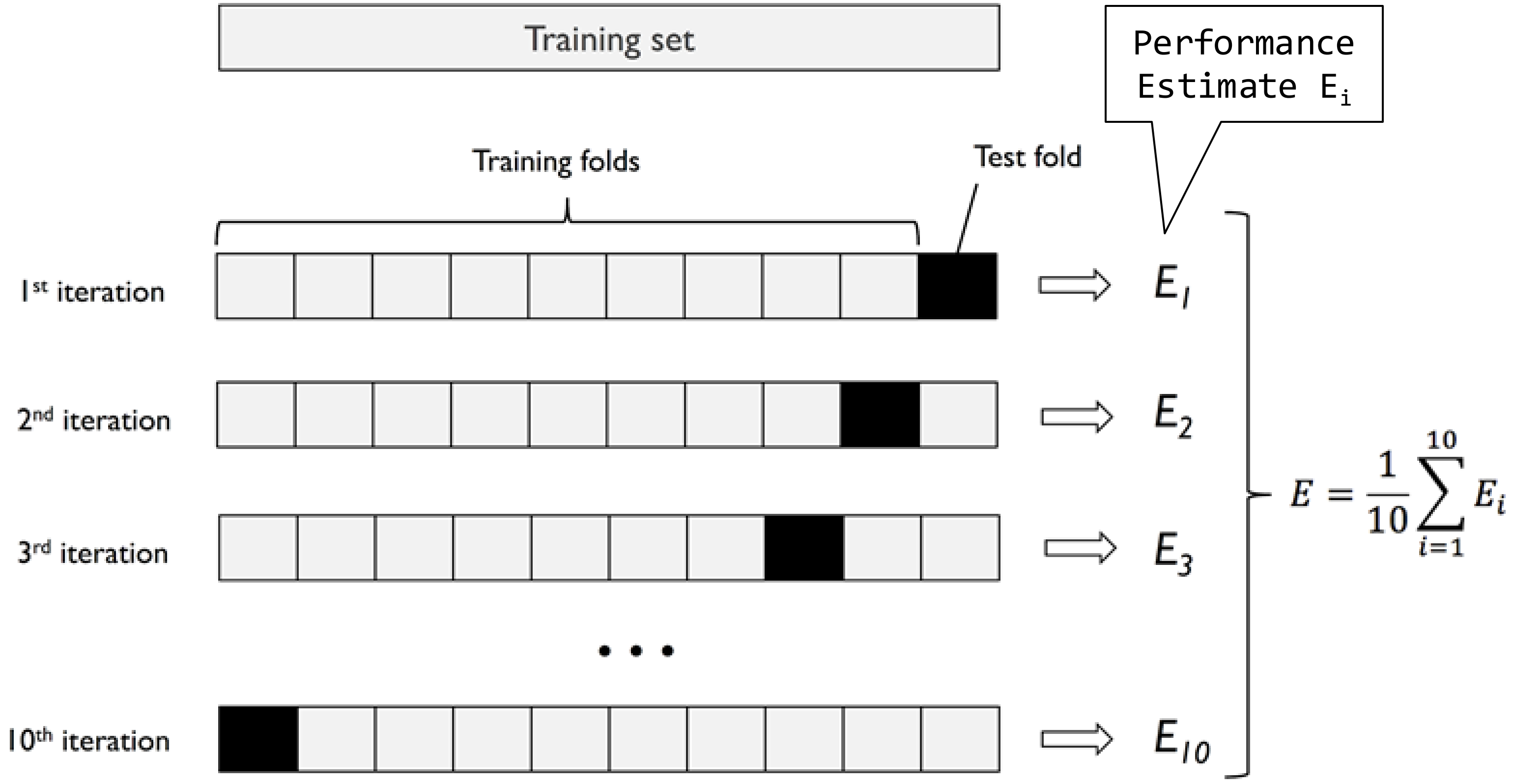
The Holdout Method

- A disadvantage of the holdout method is that the performance estimate may be very sensitive to how we partition the training set into the training and validation subsets
    - The estimate may vary for different samples of the data

K-Fold Cross-Validation

- Randomly split the training dataset into k folds without replacement
- Use k - 1 folds for training, and one fold for evaluation
- Repeated k times
  - This will give us k models and performance estimates
- Calculate the average performance of the models based on the different, independent folds
  - Obtain a performance estimate that is less sensitive to the sub-partitioning of the training data compared to the holdout method
- Once we have found satisfactory hyperparameter values, we can retrain the model on the complete training set and obtain a final performance estimate using the independent test set

K-Fold Cross-Validation



K-Fold Cross-Validation

- A good value for k in is 10, as empirical evidence shows
  - For relatively small training sets, it can be useful to increase k
    - More training data will be used in each iteration
      - Results in a lower bias towards estimating the generalization performance by averaging the individual model estimates
    - Runtime will increase
      - Yields estimates with higher variance, since the training folds will be more similar to each other
  - For larger datasets, we can choose a smaller value for k
    - Still obtain an accurate estimate of the average performance of the model while reducing the computational cost of refitting and evaluating the model on the different folds

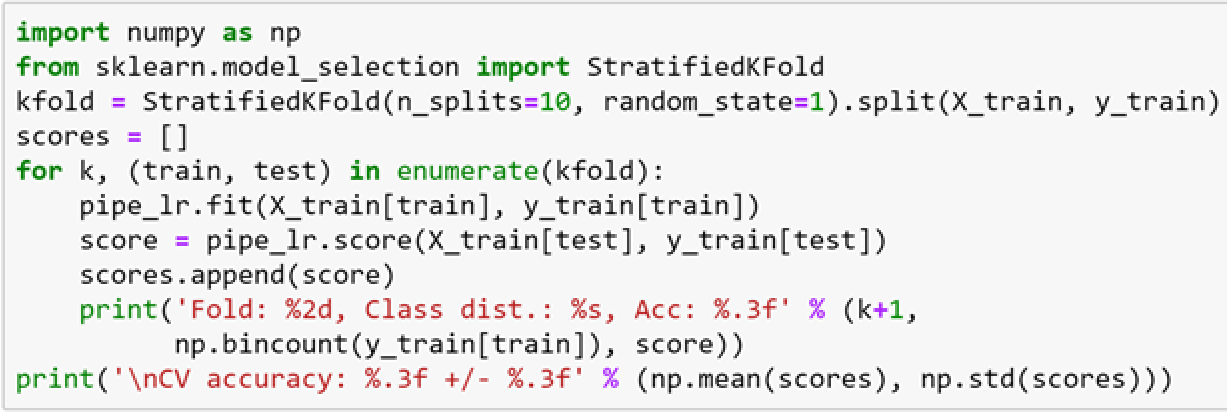
Leave-One-Out Cross-Validation

- A special case of k-fold cross-validation is the Leave-one-out cross-validation (LOOCV) method
- Set the number of folds equal to the number of training samples (k = n) so that only one training sample is used for testing during each iteration
  - Recommended approach for working with very small datasets

Stratified K-Fold Cross-Validation

- In stratified cross-validation, the class proportions are preserved in each fold to ensure that each fold is representative of the class proportions in the training dataset
  - Can yield better bias and variance estimates, especially in cases of unequal class proportions
- Scikit-learn provides the StratifiedKFold iterator class for this

Stratified K-Fold Cross-Validation



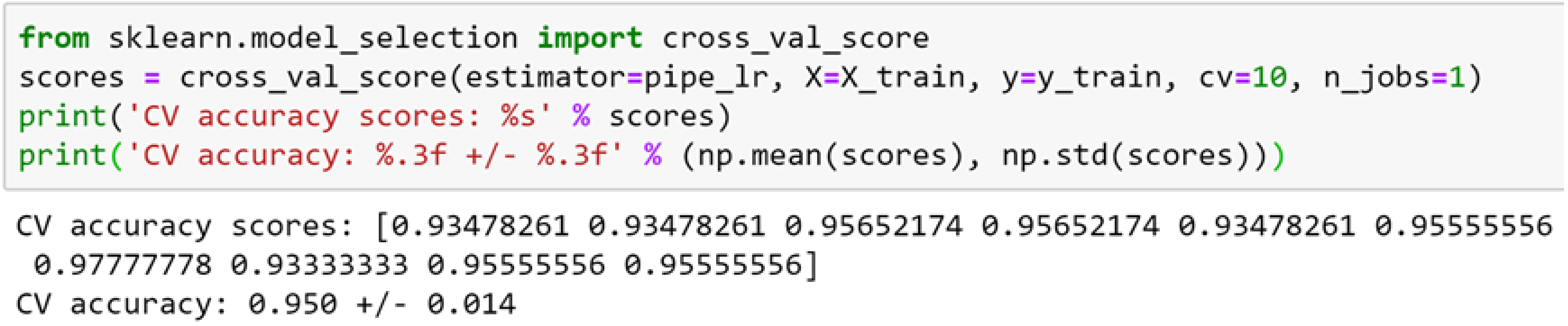
```
import numpy as np
from sklearn.model_selection import StratifiedKFold
kfold = StratifiedKFold(n_splits=10, random_state=1).split(X_train, y_train)
scores = []
for k, (train, test) in enumerate(kfold):
    pipe_lr.fit(X_train[train], y_train[train])
    score = pipe_lr.score(X_train[test], y_train[test])
    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

- The previous code example was useful to illustrate how k-fold cross-validation works
- Scikit-learn also implements a k-fold cross-validation scorer
    - Allows us to evaluate our model less verbosely



```python
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 0.95555556
 0.97777778 0.93333333 0.95555556 0.95555556]
CV accuracy: 0.950 +/- 0.014
```

Outline

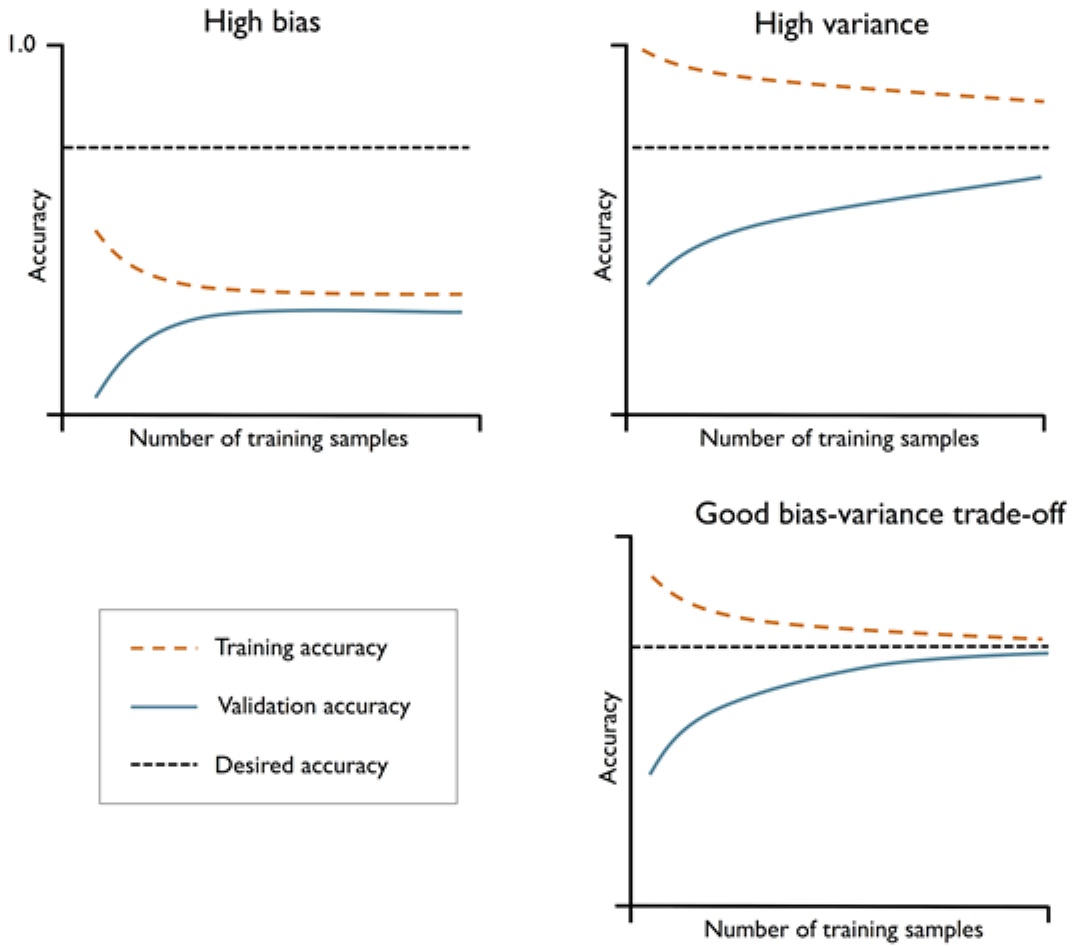
- Pipelining Transformers
- Validation
    - Holdout
    - Cross-Validation
- Learning and Validation Curve
- Hyperparameter Search
- Performance Evaluation Metrics
    - Precision
    - Recall
    - F1-score
    - Receiver Operating Characteristic
- Scoring Metrics for Multiclass Classification
- Class Imbalance

Learning and Validation Curve

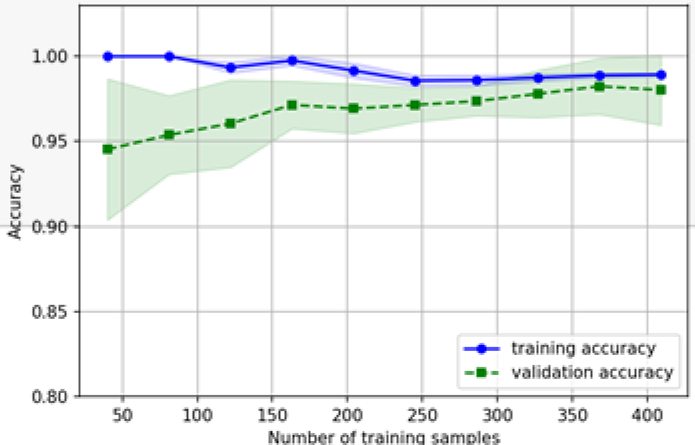
- Simple yet powerful diagnostic tools
- Learning curve
  - Can help diagnose overfitting (high variance) or underfitting (high bias)
- Validation curve
  - Can help us address some common issues of a learning algorithm

Learning Curve

- Plot the model training and validation accuracies as functions of the training set size
- Can detect if
  - model suffers from high variance or high bias
  - more data could help address this problem

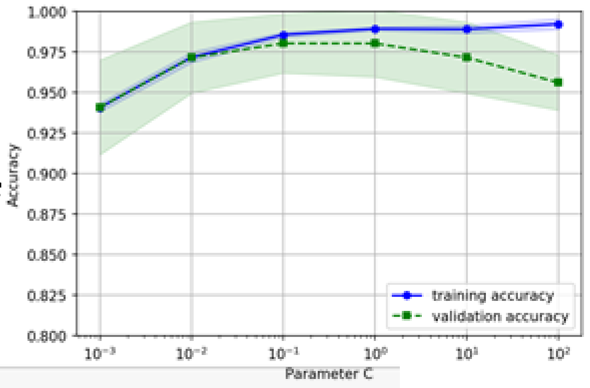


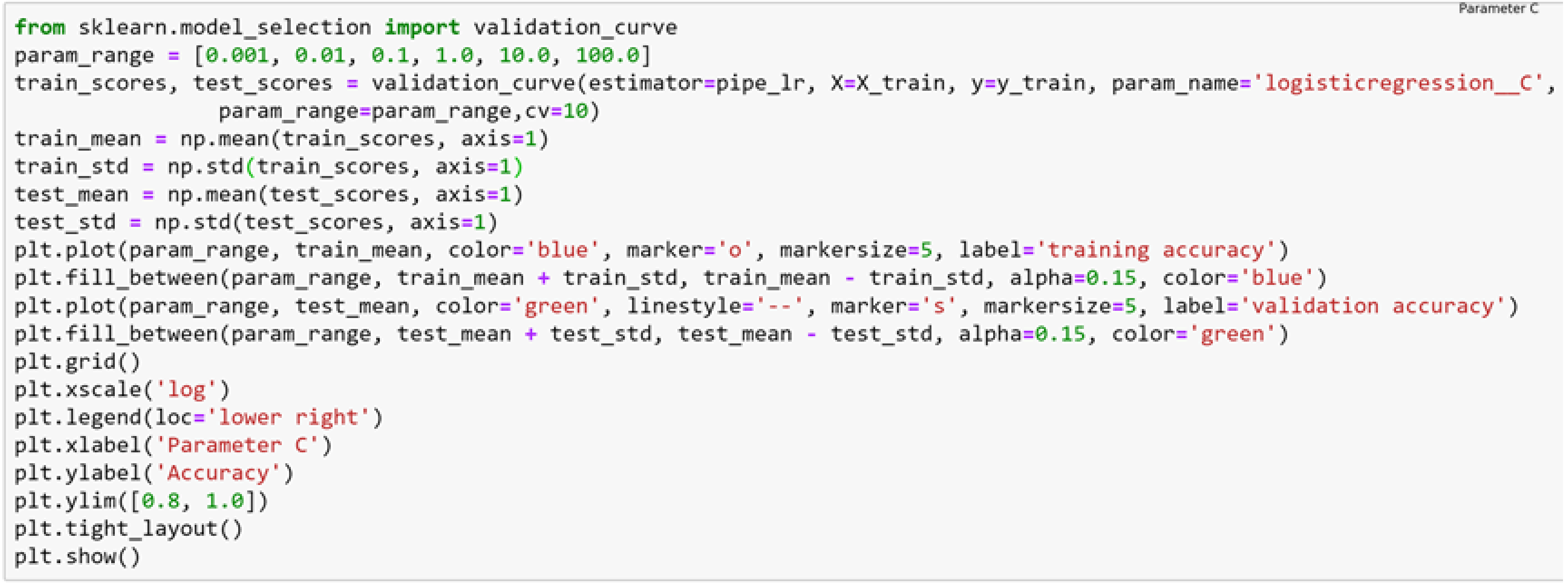
```
import matplotlib.pyplot as plt
from sklearn.model_selection import learning_curve
import matplotlib as mpl
mpl.rcParams['figure.dpi'] = 150
pipe_lr = make_pipeline(StandardScaler(), LogisticRegression(penalty='l2', random_state=1, solver='liblinear'))
train_sizes, train_scores, test_scores = learning_curve(estimator=pipe_lr, 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')
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.show()
```



Validation Curve

- Validation curves are very similar to learning curves
- Instead of plotting the train and test accuracies as functions of the sample size, we vary the values of the model parameters
  - E.g., the regularization parameter C



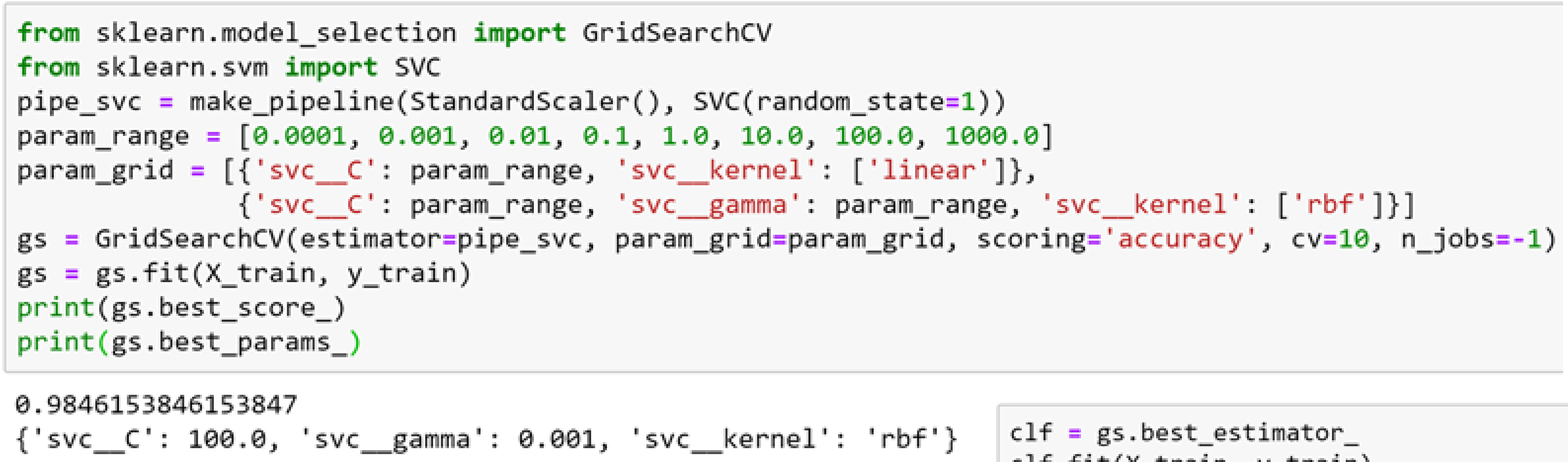


```python
from sklearn.model_selection import validation_curve
param_range = [0.001, 0.01, 0.1, 1.0, 10.0, 100.0]
train_scores, test_scores = validation_curve(estimator=pipe_lr, X=X_train, y=y_train, param_name='logisticregression__C',
                param_range=param_range,cv=10)
train_mean = np.mean(train_scores, 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)
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.show()
```

doc

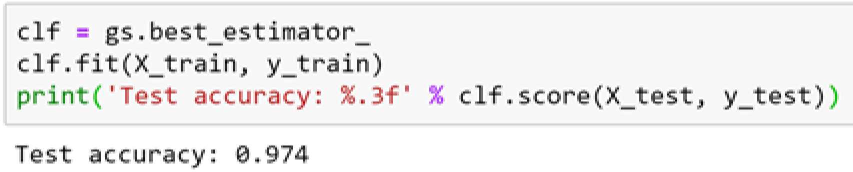
Hyperparameter Grid Search

- Finds the optimal combination of hyperparameter values
- Approach
  - Brute-force exhaustive search
  - Specify list of values for all hyperparameters
  - Evaluate the model performance for all combinations



```python
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_)
```

```
0.9846153846153847
{'svc__C': 100.0, 'svc__gamma': 0.001, 'svc__kernel': 'rbf'}
```



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

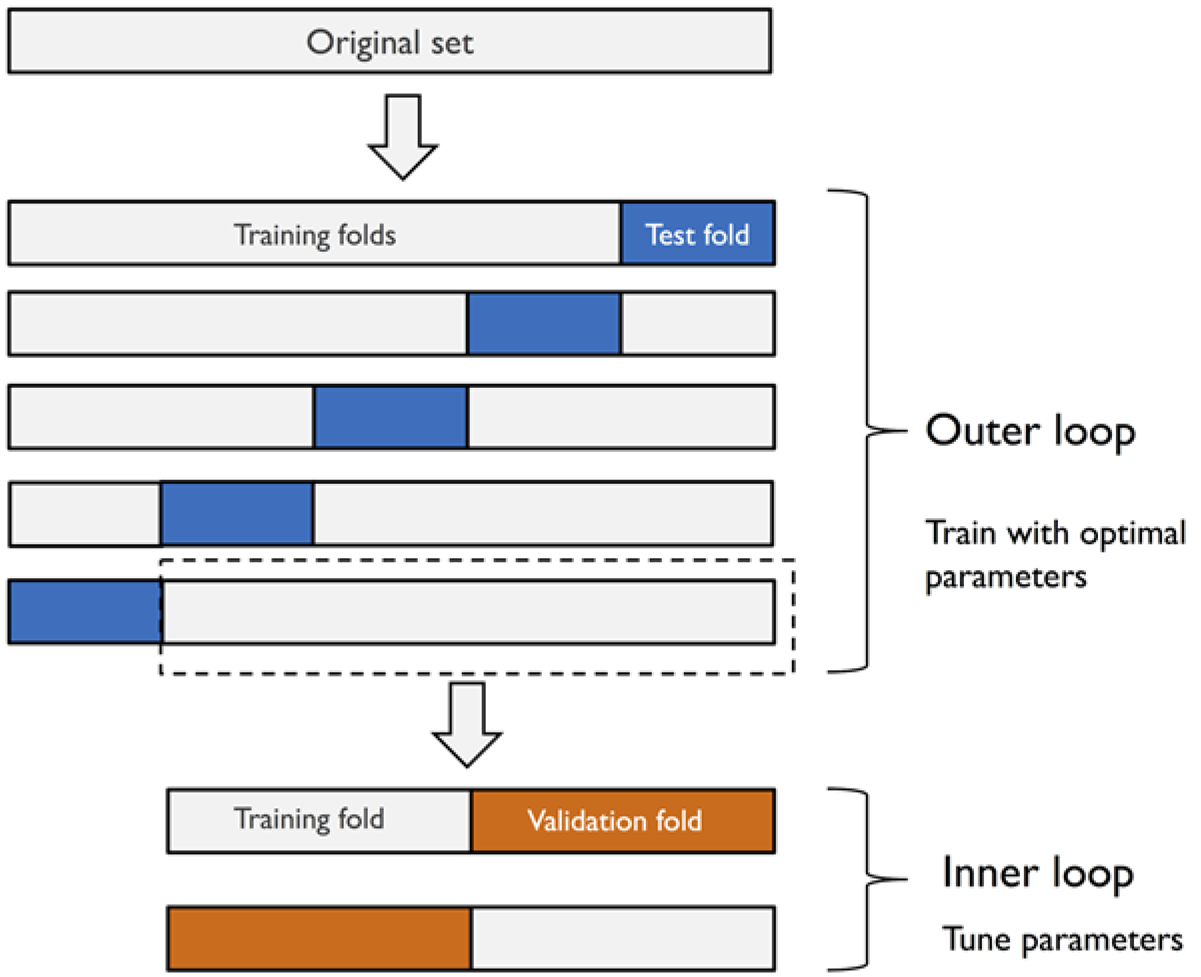
```
Test accuracy: 0.974
```

Hyperparameter Grid Search

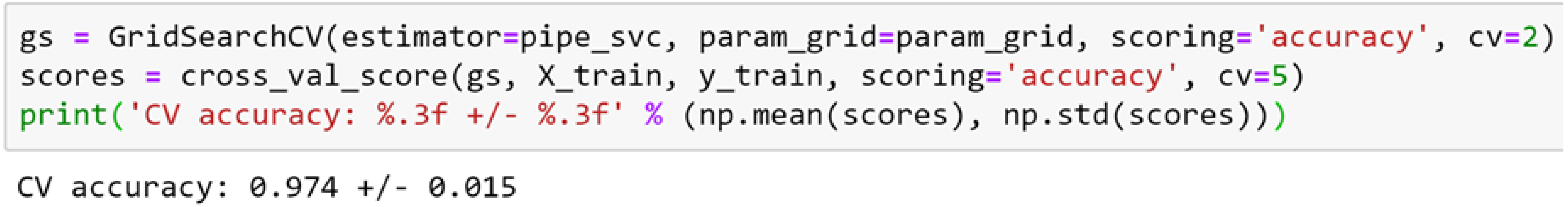
- Grid search is a powerful approach for finding the optimal set of parameters
  - Evaluation of all possible parameter combinations is however computationally very expensive
- An alternative approach to sampling different parameter combinations using scikit-learn is randomized search
  - Using the RandomizedSearchCV class in scikit-learn, we can draw random parameter combinations from sampling distributions with a specified budget

Nested Cross-Validation

- Previously we used k-fold cross-validation in combination with grid search
- We optimized the hyperparameters based on a validation score, the validation score is biased and not a good estimate of the generalization any longer
    - To get a proper estimate of the generalization we should compute the score on another validation set
    - The recommended approach is nested cross-validation
- In nested cross-validation, we have an outer k-fold cross-validation loop to split the data into training and test folds
    - An inner loop is then used to select the model using k-fold cross-validation on the training fold
- After model selection, the test fold is then used to evaluate the model performance


Original set
Training folds
Test fold
Outer loop
Train with optimal parameters
Training fold
Validation fold
Inner loop
Tune parameters

Nested Cross-Validation



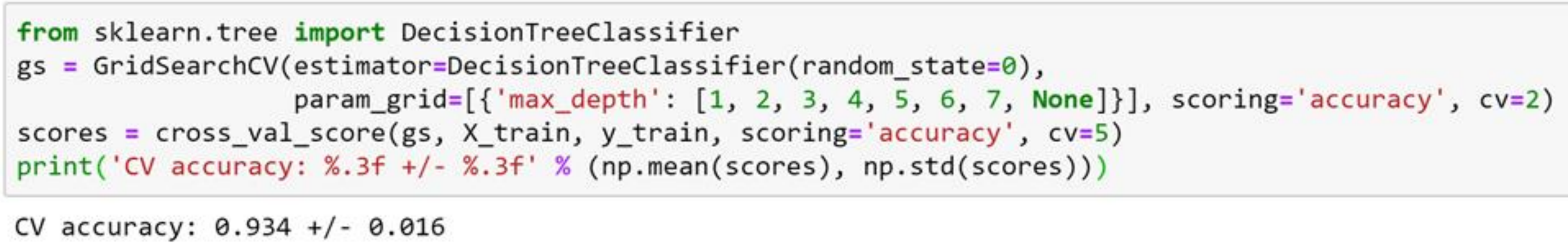
```
gs = GridSearchCV(estimator=pipe_svc, param_grid=param_grid, scoring='accuracy', cv=2)
scores = cross_val_score(gs, X_train, y_train, scoring='accuracy', cv=5)
print('CV accuracy: %.3f +/- %.3f' % (np.mean(scores), np.std(scores)))
```

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

- The returned average cross-validation accuracy gives us a better estimate of what to expect if we tune the hyperparameters of a model and use it on unseen data

Nested Cross-Validation

- We could use the nested cross-validation approach to compare an SVM model to a simple decision tree classifier; for simplicity, we will only tune its depth parameter



```
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
```

Outline

- Pipelining Transformers
- Validation
  - Holdout
  - Cross-Validation
- Learning and Validation Curve
- Hyperparameter Search
- Performance Evaluation Metrics
  - Precision
  - Recall
  - F1-score
  - Receiver Operating Characteristic
- Scoring Metrics for Multiclass Classification
- Class Imbalance

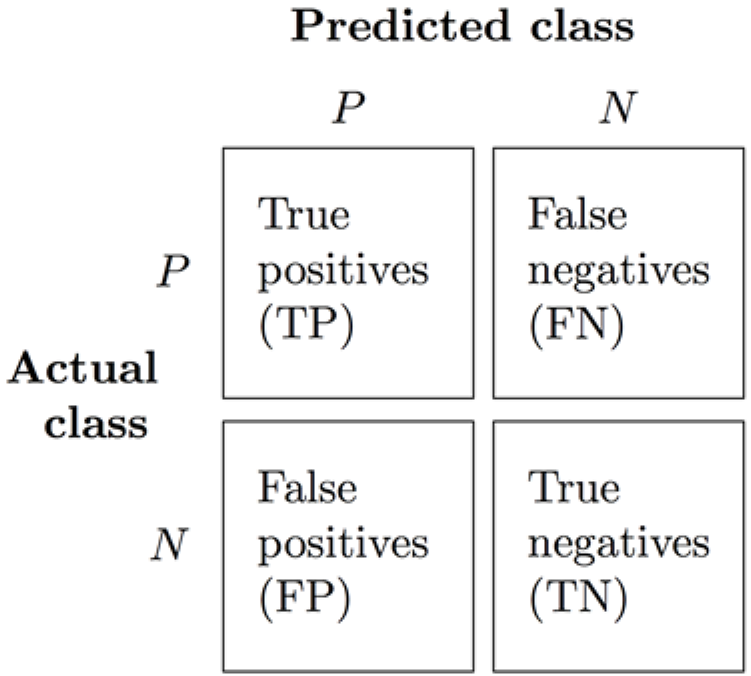
Different Performance Evaluation Metrics

- In general, accuracy is a useful metric to quantify the performance of a model
- However, there are several other performance metrics that can be used to measure a model's relevance, such as
  - Precision
  - Recall
  - F1-score

Confusion Matrix

- A matrix that lays out the performance of a learning algorithm
- The confusion matrix is simply a square matrix that reports the counts of the
  - True Positive (TP)
  - True Negative (TN)
  - False Positive (FP)
  - False Negative (FN)

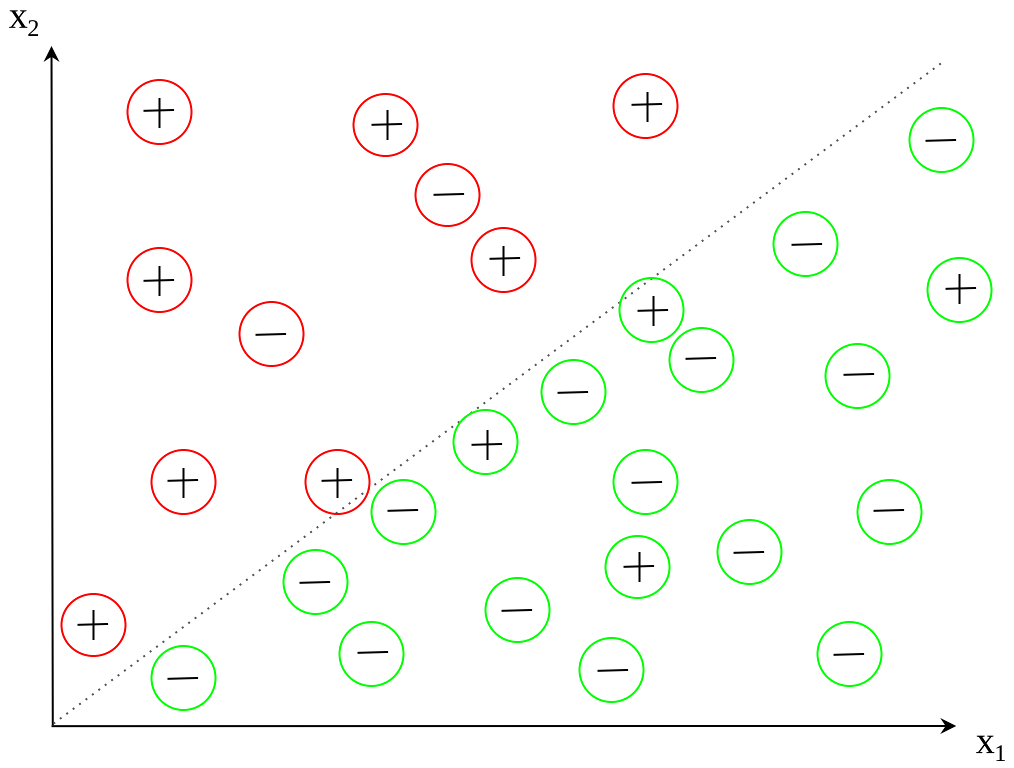
  predictions of a classifier

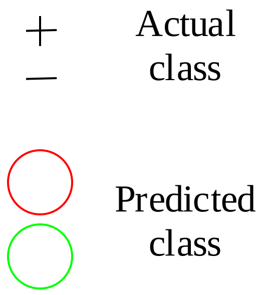


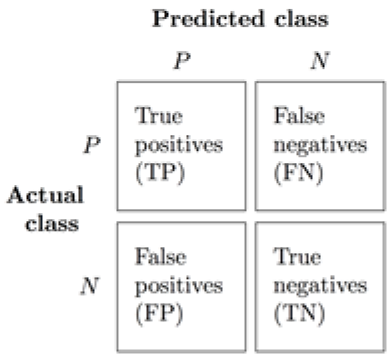
		Predicted class	
		P	*N*
Actual class	*P*	True positives (TP)	False negatives (FN)
	N	False positives (FP)	True negatives (TN)

Quiz

- Calculate the confusion matrix for the following example





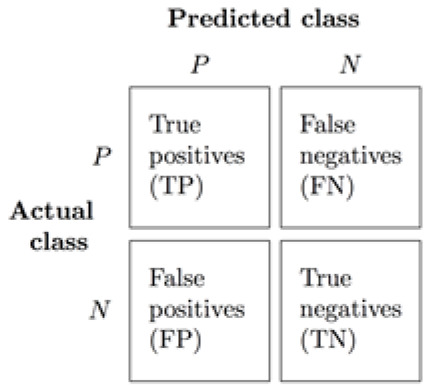


		Predicted class	
		P	*N*
Actual class	*P*	True positives (TP)	False negatives (FN)
	N	False positives (FP)	True negatives (TN)

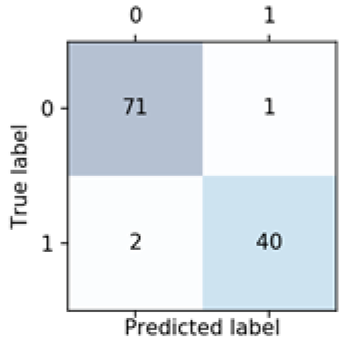
Confusion Matrix

```
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]]
```

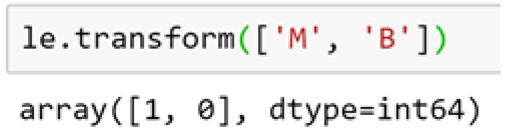


```
fig, ax = plt.subplots(figsize=(2.5, 2.5))
ax.matshow(confmat, cmap=plt.cm.Blues, alpha=0.3)
for i in range(confmat.shape[0]):
    for j in range(confmat.shape[1]):
        ax.text(x=j, y=i, s=confmat[i, j], va='center', ha='center')
plt.xlabel('Predicted label')
plt.ylabel('True label')
plt.tight_layout()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



Confusion Matrix

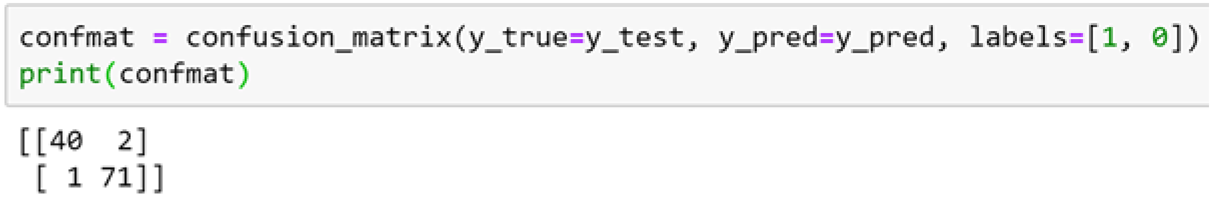
- Note that we previously encoded the class labels so that malignant samples are the "positive" class (1), and benign samples are the "negative" class (0):



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

```
array([1, 0], dtype=int64)
```

- Note that the (true) class 0 samples that are correctly predicted as class 0 (true negatives) are in the upper left corner of the matrix on the previous slide
- To change the ordering so that the true negatives are in the lower right corner (index 1,1) and the true positives are in the upper left, we can use the labels argument

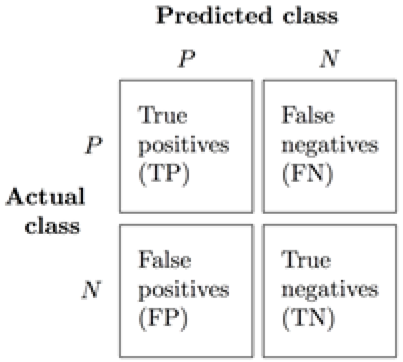


```
confmat = confusion_matrix(y_true=y_test, y_pred=y_pred, labels=[1, 0])
print(confmat)
```

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

Error and Accuracy

- Both the prediction error (ERR) and accuracy (ACC) provide general information about how many samples are misclassified
  - Error can be understood as the sum of all false predictions divided by the number of total predictions



		Predicted class	
		P	N
Actual class	P	True positives (TP)	False negatives (FN)
	N	False positives (FP)	True negatives (TN)

$$ERR = \frac{FP + FN}{FP + FN + TP + TN}$$

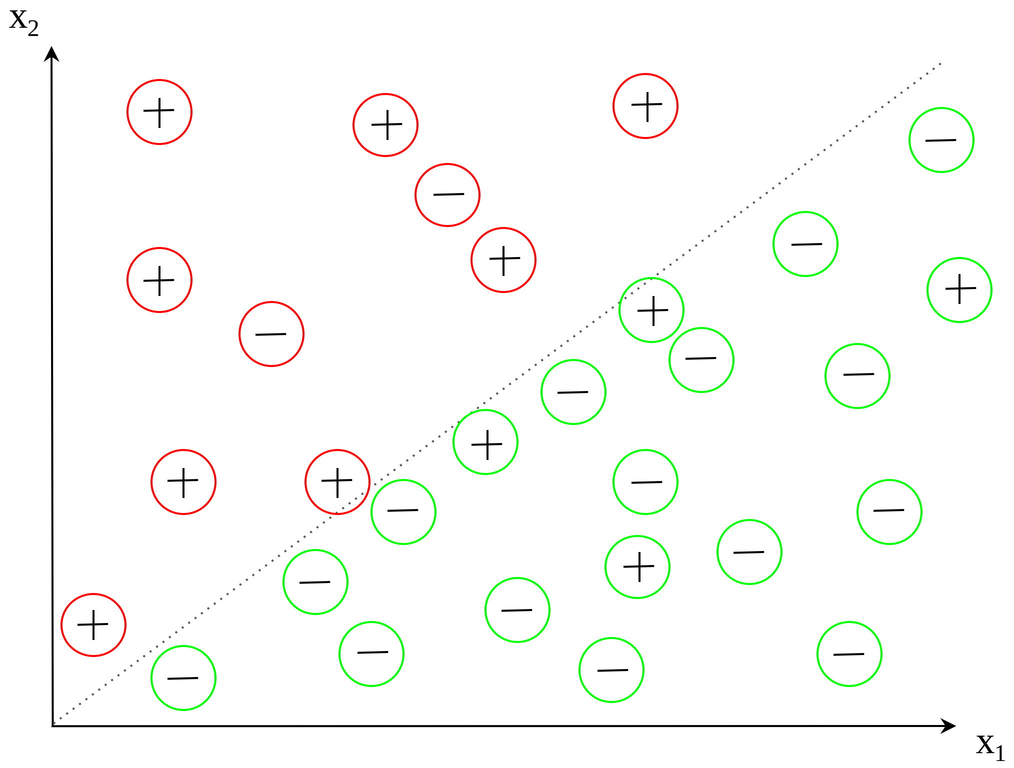
  - Accuracy is calculated as the sum of correct predictions divided by the total number of prediction

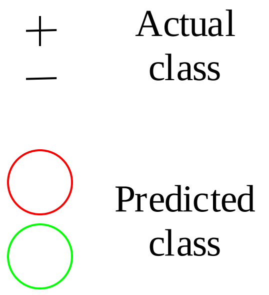
$$ACC = \frac{TP + TN}{FP + FN + TP + TN} = 1 - ERR$$

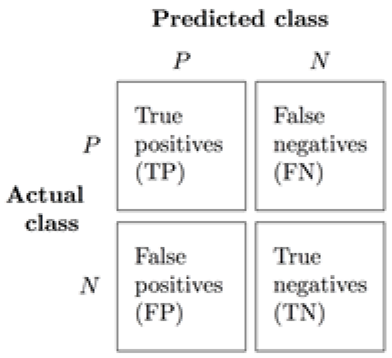
Quiz

$$ERR = \frac{FP + FN}{FP + FN + TP + TN} \qquad ACC = \frac{TP + TN}{FP + FN + TP + TN} = 1 - ERR$$

- Calculate the error and accuracy for the following example







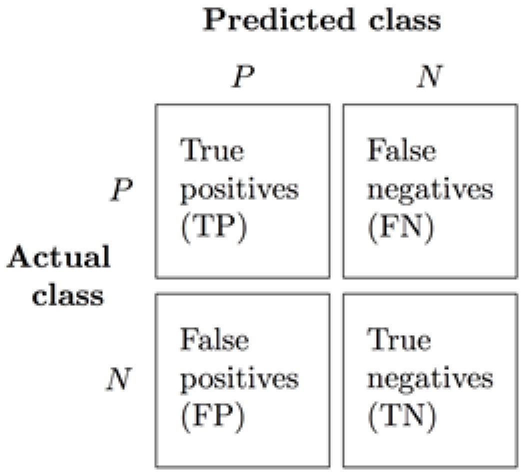
		Predicted class	
		P	N
Actual class	P	True positives (TP)	False negatives (FN)
	N	False positives (FP)	True negatives (TN)

True/False Positive Rate

- The True positive rate (TPR) and False positive rate (FPR) are performance metrics that are especially useful for imbalanced class problems

$$FPR = \frac{FP}{N} = \frac{FP}{FP + TN}$$

$$TPR = \frac{TP}{P} = \frac{TP}{FN + TP}$$



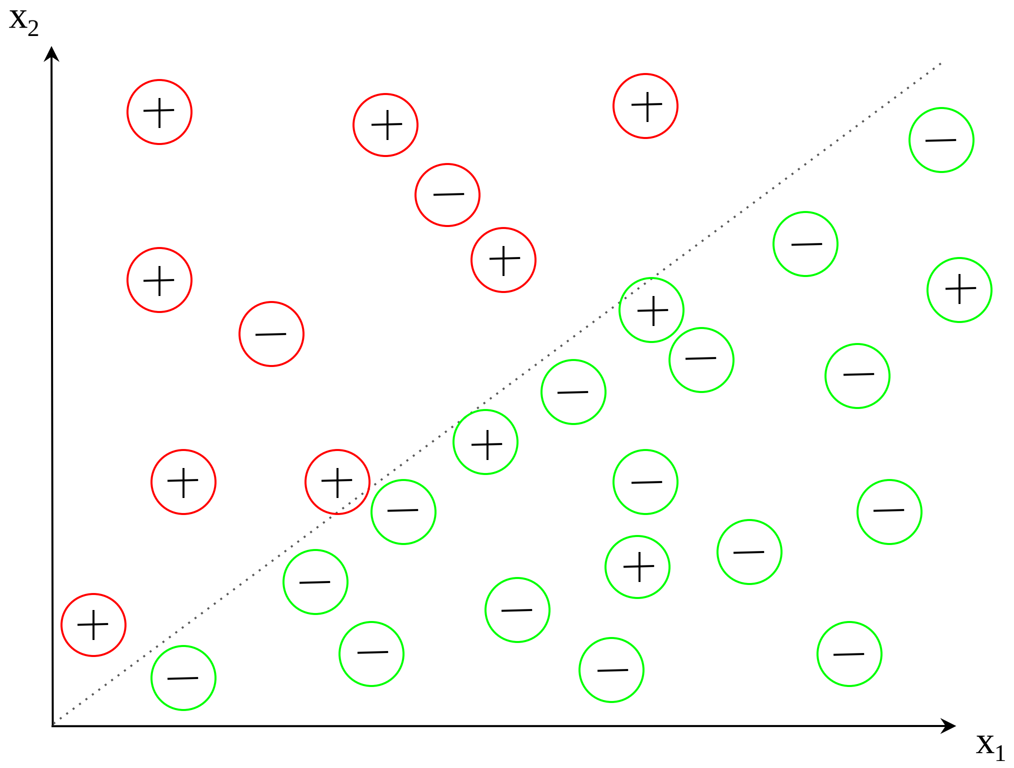
		Predicted class	
		P	N
Actual class	P	True positives (TP)	False negatives (FN)
	N	False positives (FP)	True negatives (TN)

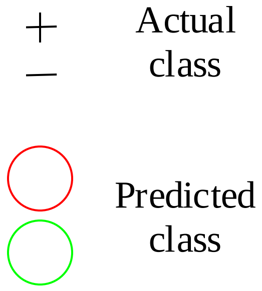
Quiz

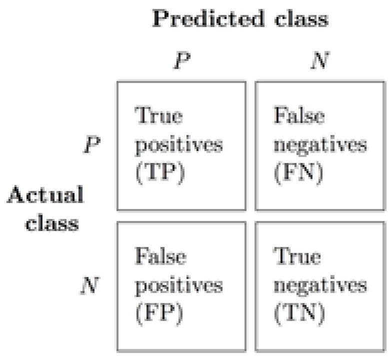
$$FPR = \frac{FP}{N} = \frac{FP}{FP + TN}$$

$$TPR = \frac{TP}{P} = \frac{TP}{FN + TP}$$

- Calculate the TPR and FPR for the following example







Predicted class

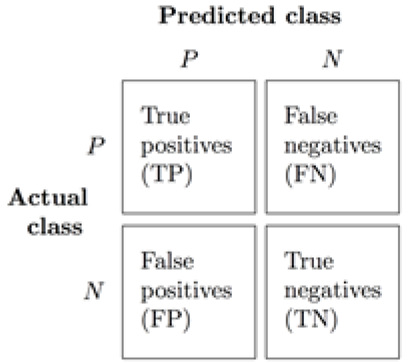
		P	N
Actual class	P	True positives (TP)	False negatives (FN)
	N	False positives (FP)	True negatives (TN)

Precision and Recall

- The performance metrics precision (PRE) and recall (REC) are related to those true positive and negative rates, and in fact, REC is synonymous with TPR

$$PRE = \frac{TP}{TP + FP}$$

$$REC = TPR = \frac{TP}{P} = \frac{TP}{FN + TP}$$



		Predicted class	
		P	N
Actual class	P	True positives (TP)	False negatives (FN)
	N	False positives (FP)	True negatives (TN)

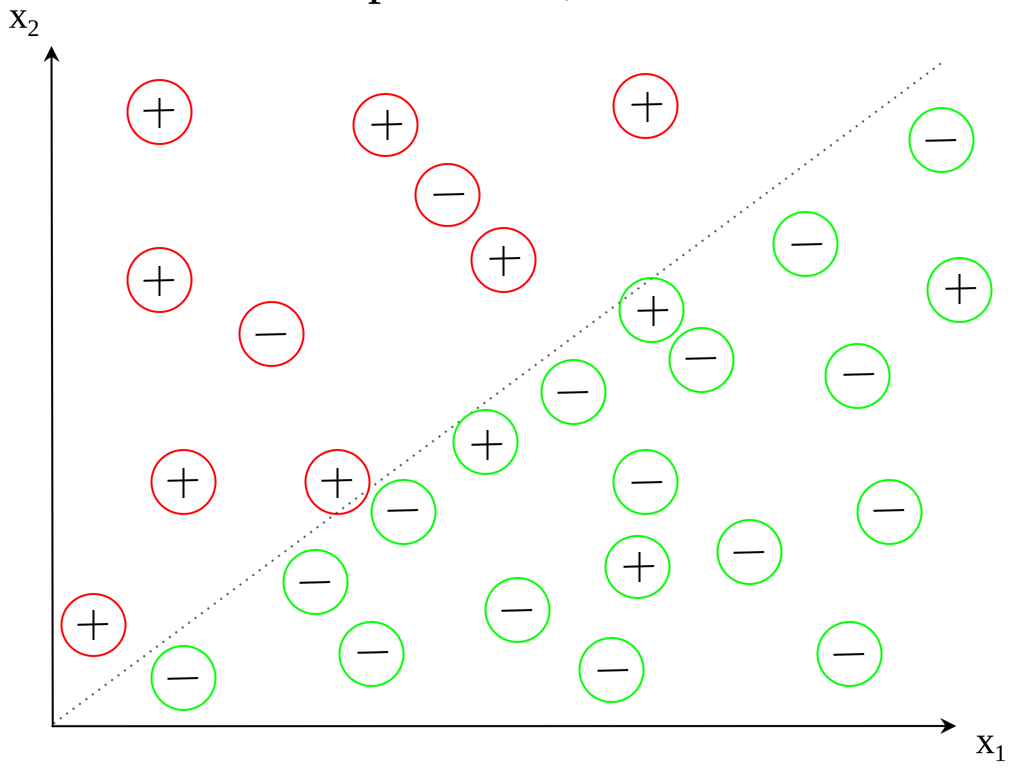
- In practice, often a combination of PRE and REC is used, the so-called F1-score

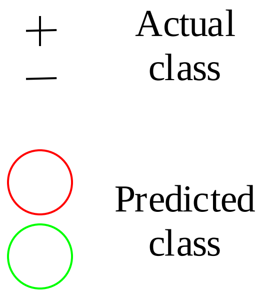
$$F1 = 2\frac{PRE \times REC}{PRE + REC}$$

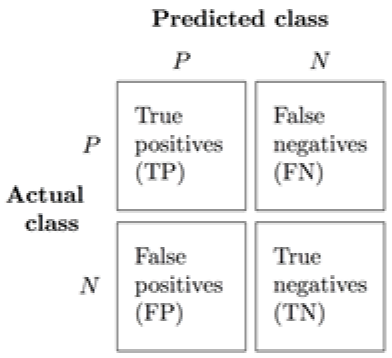
Quiz

$$PRE = \frac{TP}{TP + FP} \qquad REC = TPR = \frac{TP}{P} = \frac{TP}{FN + TP} \qquad F1 = 2\frac{PRE \times REC}{PRE + REC}$$

- Calculate the precision, recall and F1-score for the following example






Predicted class
P
N
Actual class
P
N
True positives (TP)
False negatives (FN)
False positives (FP)
True negatives (TN)

Scoring Metrics in scikit-learn

- Those scoring metrics are all implemented in scikit-learn and can be imported from the sklearn.metrics module

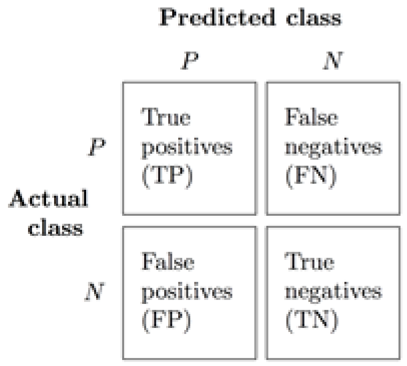
```python
from sklearn.metrics import precision_score, recall_score, f1_score
print('Precision: %.3f' % precision_score(y_true=y_test, y_pred=y_pred))
print('Recall: %.3f' % recall_score(y_true=y_test, y_pred=y_pred))
print('F1: %.3f' % f1_score(y_true=y_test, y_pred=y_pred))
```

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

$$PRE = \frac{TP}{TP + FP}$$

$$F1 = 2\frac{PRE \times REC}{PRE + REC}$$

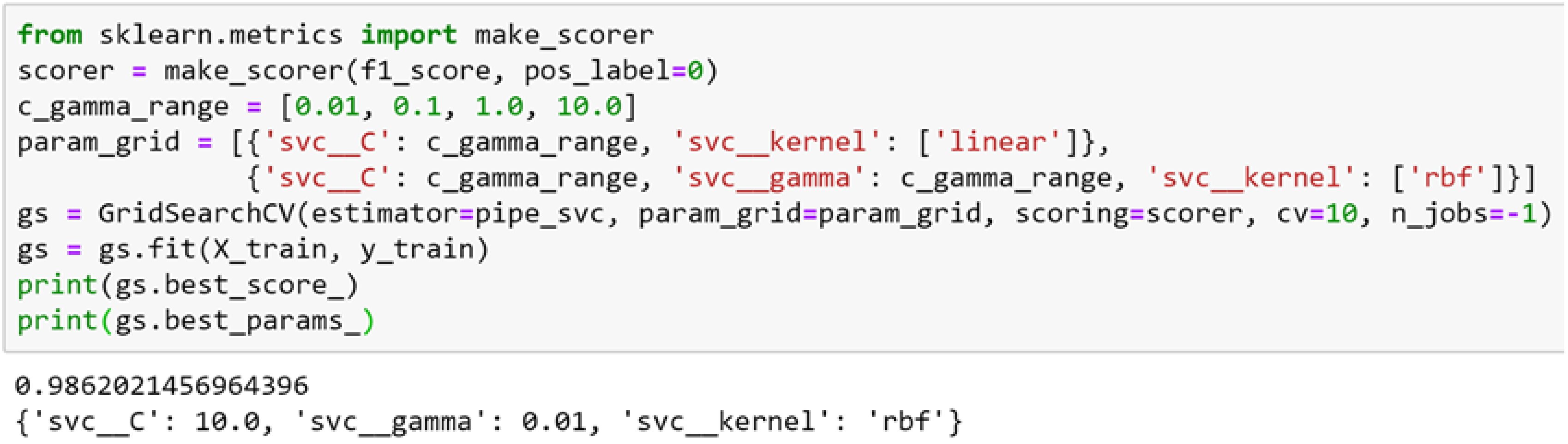
$$REC = TPR = \frac{TP}{P} = \frac{TP}{FN + TP}$$



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

Scoring Metric in GridSearchCV

- We can use a different scoring metric than accuracy in the GridSearchCV
  - Via the scoring parameter
- Remember that the positive class in scikit-learn is the class that is labeled as class 1
- Specify a different positive label by construct scorer via the make_scorer function



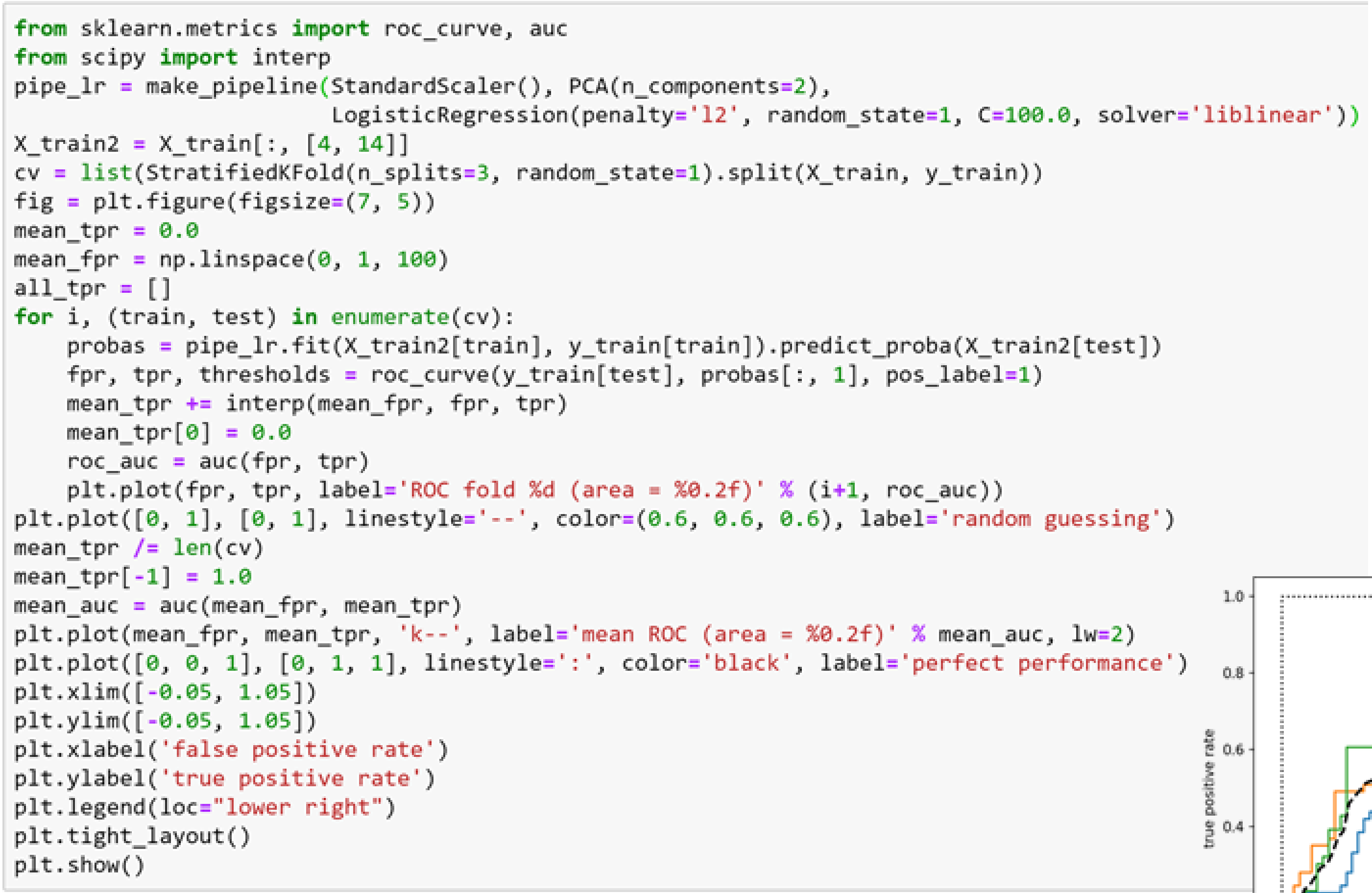
```
from sklearn.metrics import make_scorer
scorer = make_scorer(f1_score, pos_label=0)
c_gamma_range = [0.01, 0.1, 1.0, 10.0]
param_grid = [{'svc__C': c_gamma_range, 'svc__kernel': ['linear']},
              {'svc__C': c_gamma_range, 'svc__gamma': c_gamma_range, 'svc__kernel': ['rbf']}]
gs = GridSearchCV(estimator=pipe_svc, 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_)
```

```
0.9862021456964396
{'svc__C': 10.0, 'svc__gamma': 0.01, 'svc__kernel': 'rbf'}
```

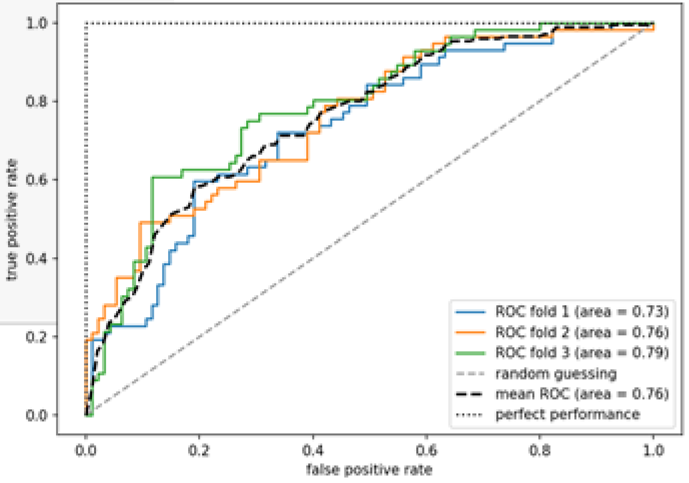
Receiver Operating Characteristic ROC

- ROC graphs are useful tools to select models for classification based on their performance with respect to the FPR and TPR
    - Computed by shifting the decision threshold of the classifier
- The diagonal of an ROC graph can be interpreted as random guessing, and classification models that fall below the diagonal are considered as worse than random guessing
- A perfect classifier would fall into the top left corner of the graph with a TPR of 1 and an FPR of 0
- Based on the ROC curve, we can then compute the so-called ROC Area Under the Curve (ROC AUC) to characterize the performance of a classification model

$$FPR = \frac{FP}{N} = \frac{FP}{FP + TN} \qquad TPR = \frac{TP}{P} = \frac{TP}{FN + TP}$$



```python
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, solver='liblinear'))
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))
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.show()
```



Scoring Metrics for Multiclass Classification

- The scoring metrics that we discussed in this section are specific to binary classification systems
- Macro and micro averaging methods exist to extend those scoring metrics to multiclass problems via One-versus-All (OvA) classification
- The micro-average is calculated from the individual TPs, TNs, FPs, and FNs of the system
    - For example, the micro-average of the precision score in a k-class system can be calculated as

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

- The macro-average is simply calculated as the average scores of the different systems

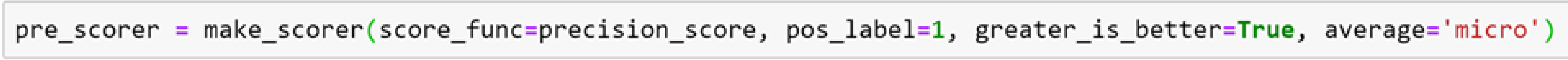
$$PRE_{macro} = \frac{PRE_1 + \cdots + PRE_k}{k}$$

Micro-averaging vs. Macro-averaging

- Micro-averaging is useful if we want to weight each instance or prediction equally
- Macro-averaging weights all classes equally to evaluate the overall performance of a classifier with regard to the most frequent class labels

Micro/Macro-averaging in scikit-learn

- If we are using binary performance metrics to evaluate multiclass classification models in scikit-learn, a normalized or weighted variant of the macro-average is used by default
- The weighted macro-average is calculated by weighting the score of each class label by the number of true instances when calculating the average
- The weighted macro-average is useful if we are dealing with class imbalances, that is, different numbers of instances for each label
- While the weighted macro-average is the default for multiclass problems in scikit-learn, we can specify the averaging method via the average parameter inside the different scoring functions that we import from the sklearn.metrics module, for example, the precision_score or make_scorer functions



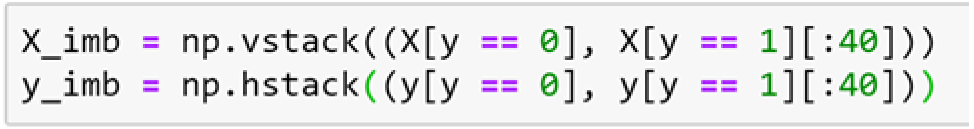
```
pre_scorer = make_scorer(score_func=precision_score, pos_label=1, greater_is_better=True, average='micro')
```

Class Imbalance

- Common problem when working with real-world
  - Samples from one class or multiple classes are over-represented in a dataset

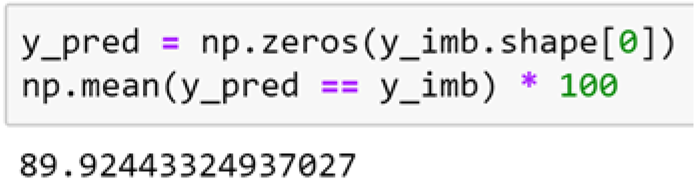
Dealing with Class Imbalance

- Let's create an imbalanced dataset from our breast cancer dataset, which originally consisted of 357 benign tumors (class 0) and 212 malignant tumors (class 1)



```
X_imb = np.vstack((X[y == 0], X[y == 1][:40]))
y_imb = np.hstack((y[y == 0], y[y == 1][:40]))
```

- If we were to compute the accuracy of a model that always predicts the majority class (benign, class 0), we would achieve a prediction accuracy of approximately 90 percent



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

```
89.92443324937027
```

Dealing with Class Imbalance

- Thus, when we fit classifiers on such datasets, it would make sense to focus on other metrics than accuracy when comparing different models, such as precision, recall, the ROC curve
    - Whatever we care most about in our application
        - Our priority might be to identify the majority of patients with malignant cancer patients to recommend an additional screening, then recall should be our 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

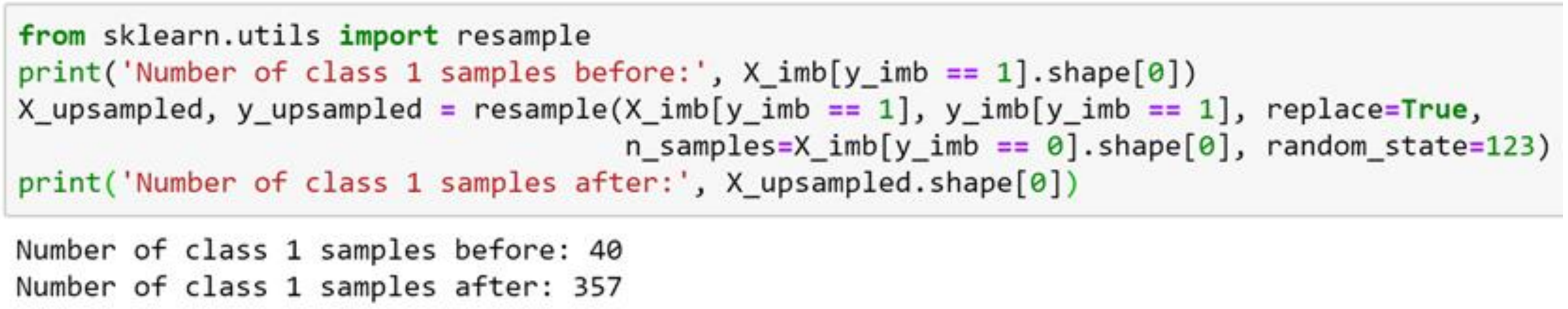
Dealing with Class Imbalance

- Aside from evaluating machine learning models, class imbalance influences a learning algorithm during model fitting itself
- Since machine learning algorithms typically optimize a reward or cost function that is computed as a sum over the training examples that it sees during fitting, the decision rule is likely going to be biased towards the majority class
    - I.e., the algorithm implicitly learns a model that optimizes the predictions based on the most abundant class in the dataset, in order to minimize the cost or maximize the reward during training
- One way to deal with imbalanced class proportions during model fitting is to assign a larger penalty to wrong predictions on the minority class. Via scikit-learn, adjusting such a penalty is as convenient as setting the class_weight parameter to class_weight='balanced', which is implemented for most classifiers.

Dealing with Class Imbalance

- Other popular strategies for dealing with class imbalance include upsampling the minority class, downsampling the majority class, and the generation of synthetic training samples
  - Unfortunately, there's no universally best solution, no technique that works best across different problem domains
  - In practice, it is recommended to try out different strategies on a given problem, evaluate the results, and choose the technique that seems most appropriate
- The scikit-learn library implements a simple resample function that can help with the upsampling of the minority class by drawing new samples from the dataset with replacement
- Similarly, we could downsample the majority class by removing training examples from the dataset
  - To perform downsampling using the resample function, we could simply swap the class 1 label with class 0

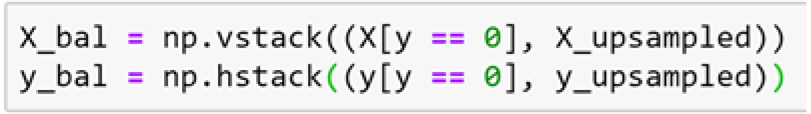
Dealing with Class Imbalance



```
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])
```

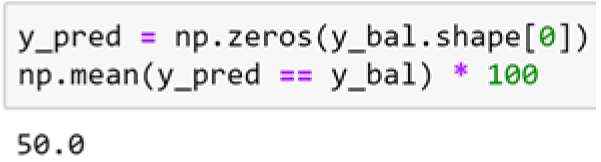
```
Number of class 1 samples before: 40
Number of class 1 samples after: 357
```

- After resampling, we can then stack the original class 0 samples with the upsampled class 1 subset to obtain a balanced dataset as follows



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

- Consequently, a majority vote prediction rule would only achieve 50 percent accuracy



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

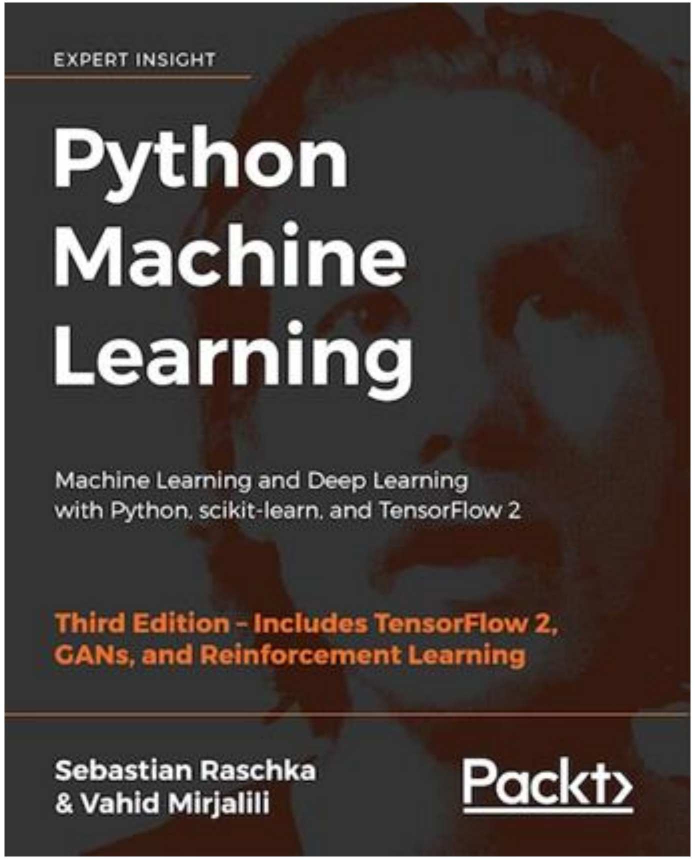
```
50.0
```

Dealing with Class Imbalance

- Another technique for dealing with class imbalance is the generation of synthetic training samples, which is beyond the scope of this course
- A widely used algorithm for synthetic training sample generation is Synthetic Minority Over-sampling Technique (SMOTE)
- More techniques to deal with the curse of imbalance can be found here

References

- Most materials in this chapter are based on
  - Book
  - Code



EXPERT INSIGHT

Python Machine Learning

Machine Learning and Deep Learning with Python, scikit-learn, and TensorFlow 2

Third Edition – Includes TensorFlow 2, GANs, and Reinforcement Learning

Sebastian Raschka & Vahid Mirjalili

Packt>

References

- A Study of Cross-Validation and Bootstrap for Accuracy Estimation and Model Selection, International Joint Conference on Artificial Intelligence (IJCAI), 14 (12): 1137-43, 1995
- Model Evaluation, Model Selection, and Algorithm Selection in Machine Learning, Sebastian Raschka
- Analysis of Variance of Cross-validation Estimators of the Generalization Error, M. Markatou, H. Tian, S. Biswas, and G. M. Hripcsak, Journal of Machine Learning Research, 6: 1127-1168, 2005
- Bias in Error Estimation When Using Cross-validation for Model Selection, BMC Bioinformatics, S. Varma and R. Simon, 7(1): 91, 2006