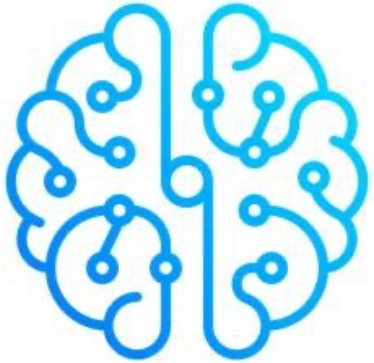


Model Evaluation

Machine Learning for Engineering Applications

Fall 2023

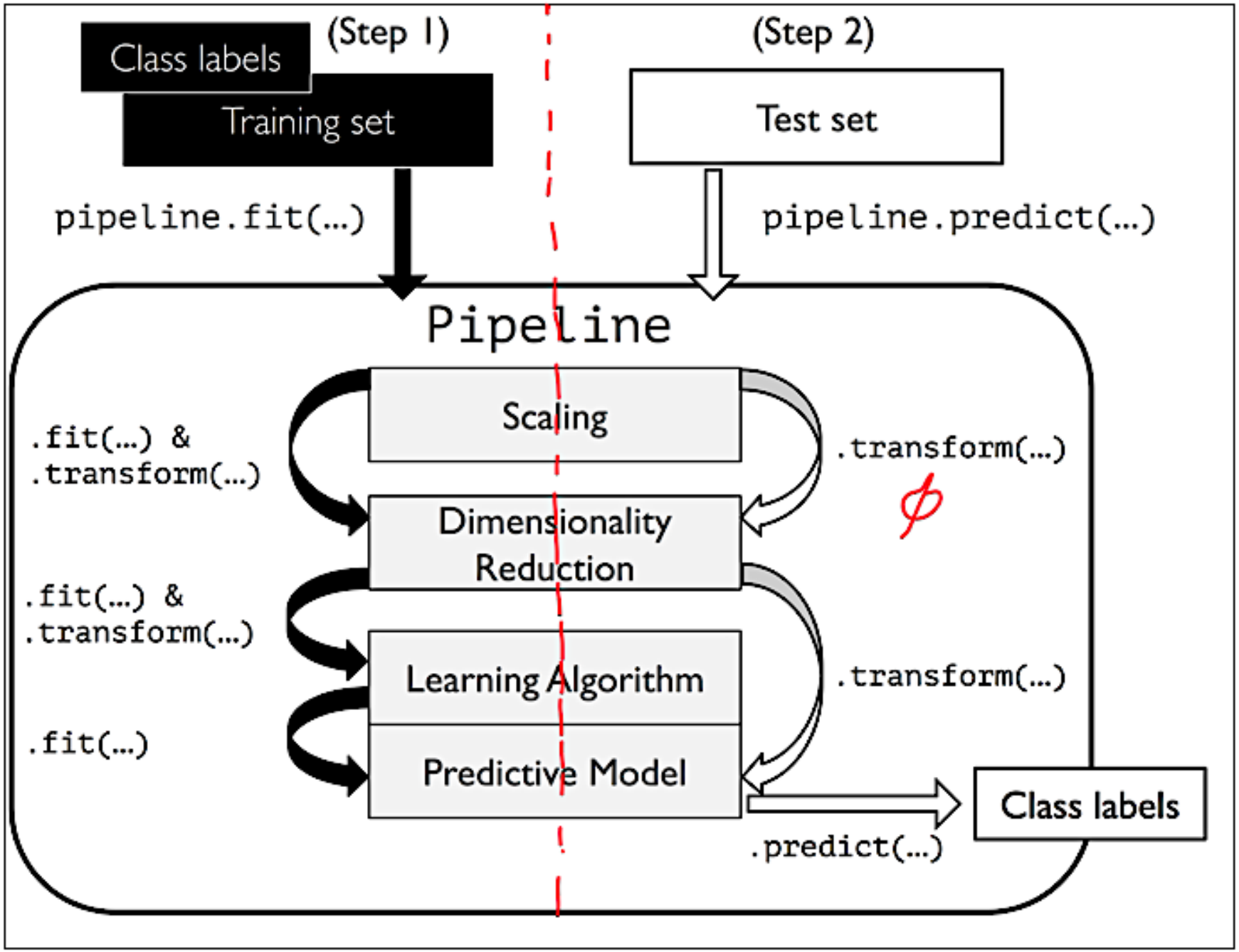
Python Pipelines

- Scikit-learn has a **Pipeline library** that help process the training and prediction methods in a specified order

- The **pipeline** will execute:
  - The training
  - The transformations
  - Provide the prediction of the model with the test data split

```
import pandas as pd
from sklearn.preprocessing import LabelEncoder
from sklearn.model_selection import train_test_split

#standard import and split code

df = pd.read_csv('your favorite dataset', header=None)
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)
```

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

#This is the setup; nothing gets executed yet…
pipe_lr = make_pipeline(StandardScaler(),
          ... PCA(n_components=2),
          ... LogisticRegression(random_state=1))

#This will evoke the pipeline
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
```


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

Model Performance

- Issues with models:
  - It can overfit (*training data is too variant*)
  - It can underfit (*high biasing*)

- **Cross-validation** techniques help in finding the right balance between high variance & high biasing

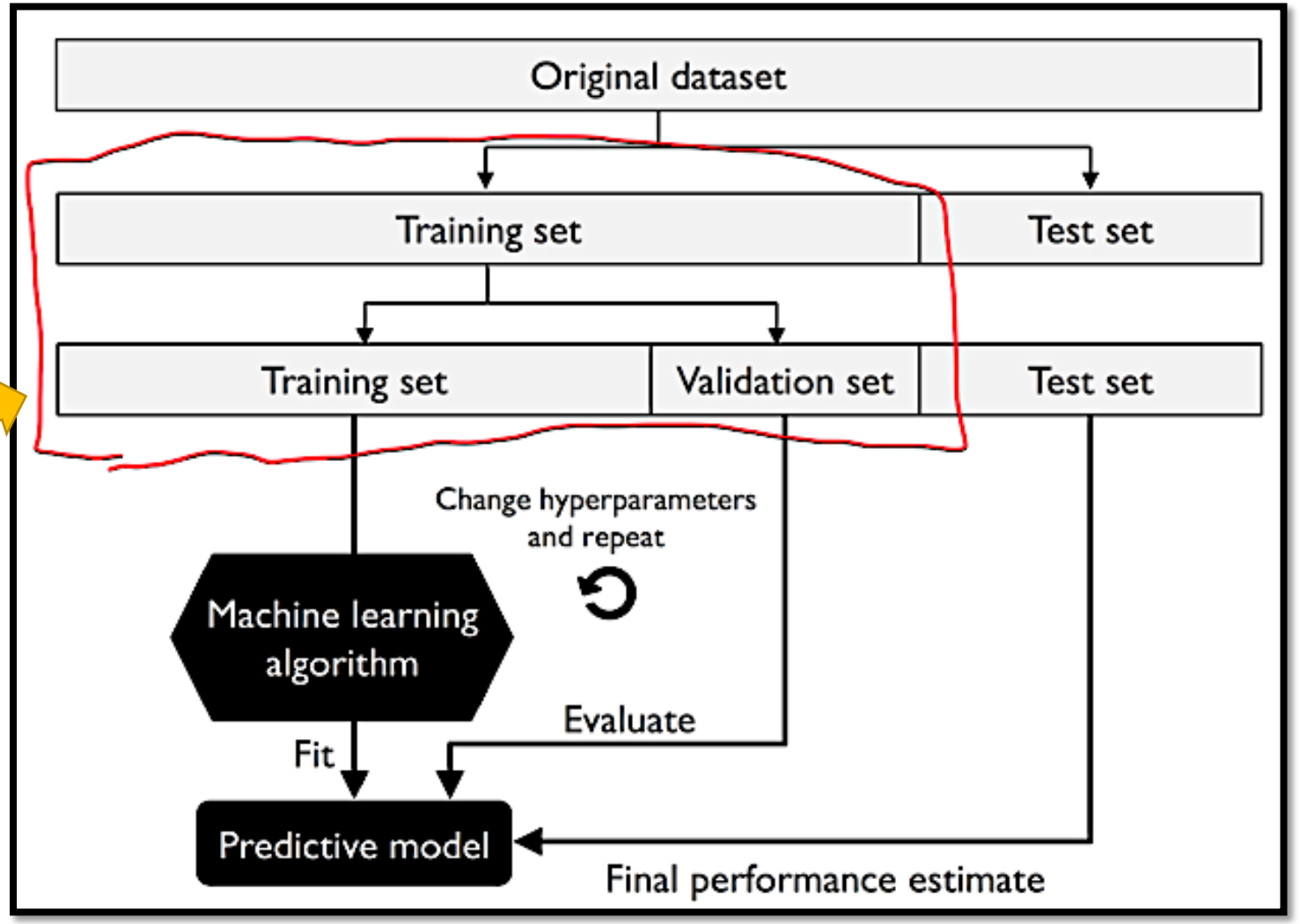
- **Techniques**:
  - Holdout Cross-Validation
  - K-fold Cross-Validation
  - Stratified K-fold Cross-Validation

- **Holdout Cross-Validation**

- The validation comes in how we split the data

- **The split consist**:
  - Training split
  - Validation split
  - Test split

- Why?
- What's the benefit?
- Disadvantage?



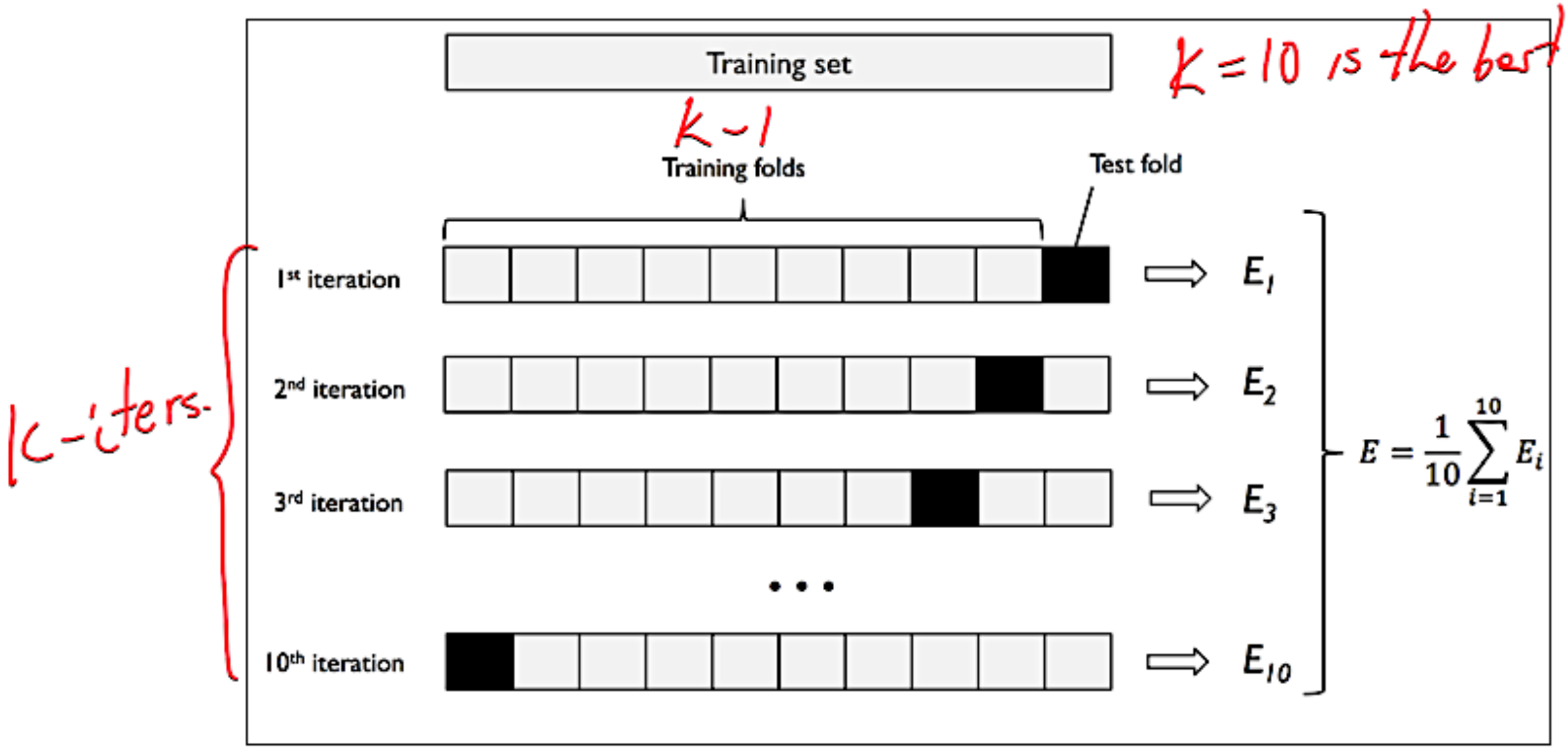
- **K-fold Cross-Validation**

- **Different approach**:
  - Randomly sort the training split & test split
  - The training split is folded *k*-times
  - Use *k-1* folds for training, k^{th} for the test
  - Then, repeat the training/test *k-times*

- You end up with *k* models, *k* evaluations

- The average is calculated and is less sensitive to the splitting of the data

- Great to find the optimal hyperparameter values


Training set
k=10 is the best!
k-1
Training folds
Test fold
k-iters
1st iteration
2nd iteration
3rd iteration
10th iteration
E1
E2
E3
E10
$E = \frac{1}{10}\sum_{i=1}^{10} E_i$

- **K-fold Cross-Validation**

- **Small datasets**:
  - When dealing with smaller dataset -> Increase the number of folds
  - Increasing the number of folds -> increases you're the number of "*different models*"

- **Increasing the *k*-value**:
  - Increase the runtime of the cross-validation
  - The average outcomes can start to have large variances (*overfit*)

- **Stratified K-fold Cross-Validation**

- Same as K-fold

- The folding part is stratified for all iterations
  - **Again**: it will evenly distribute the training and test fold for all class labels

  - Make the splitting less sensitive to the split decisions; even when random

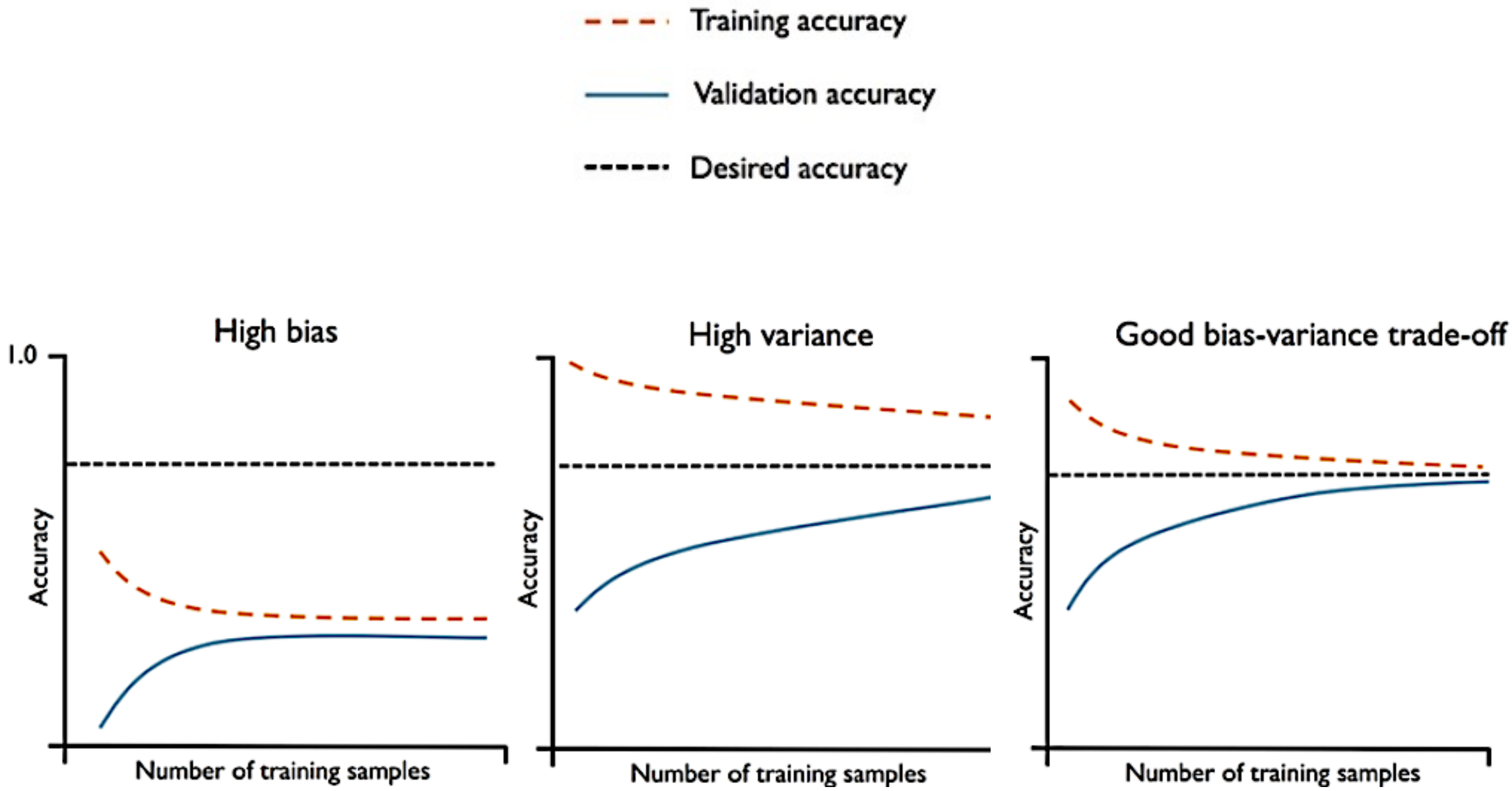
```
import numpy as np
from sklearn.model_selection import StratifiedKFold

kfold = StratifiedKFold(n_splits=10, random_state=1).split(X_train,
                                                           ... y_train)
scores = []  #initialize the object vector
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))

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
```

```
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))

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

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

CV accuracy: 0.950 +/- 0.014
```

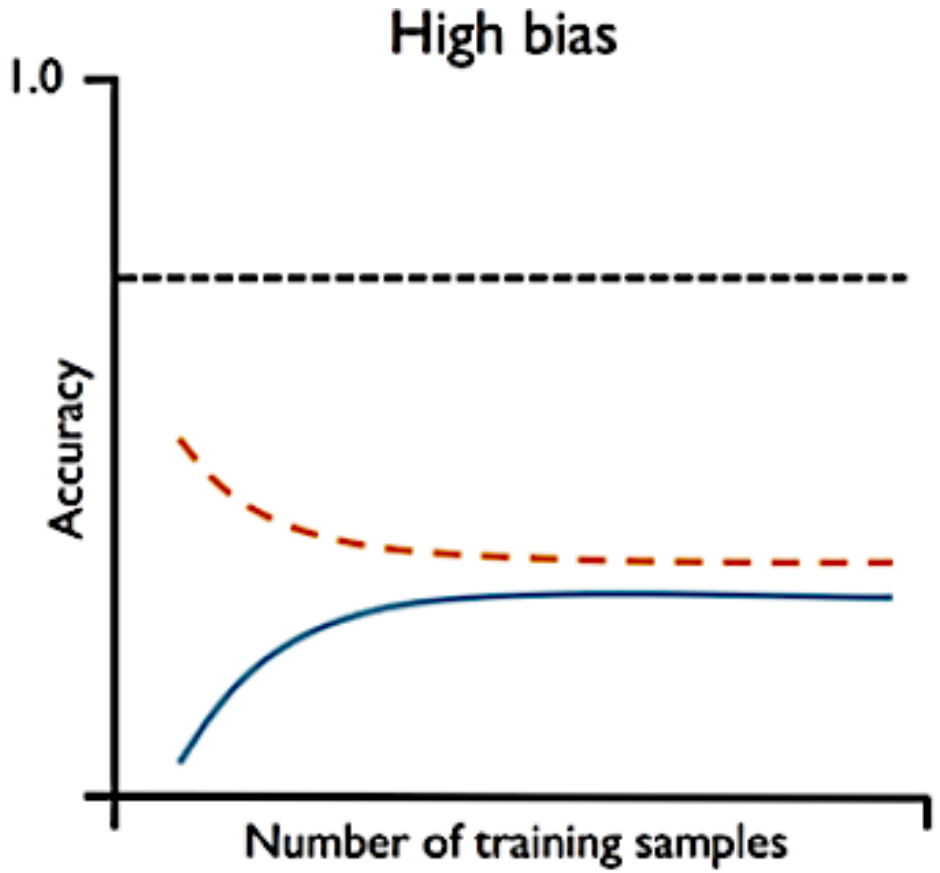
```
from sklearn.model_selection import cross_val_score

scores = cross_val_score(estimator=pipe_lr,
      ... X=X_train,
      ... y=y_train,
      ... cv=10,
      ... n_jobs=-1)  #use all available cores in the machine

print('CV accuracy scores: %s' % scores)
      CV accuracy scores: [ 0.93478261 0.93478261 0.95652174
      0.95652174 0.93478261 0.95555556
      0.97777778 0.93333333 0.95555556
      0.95555556]

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

      CV accuracy: 0.950 +/- 0.014
```

Learning Curves

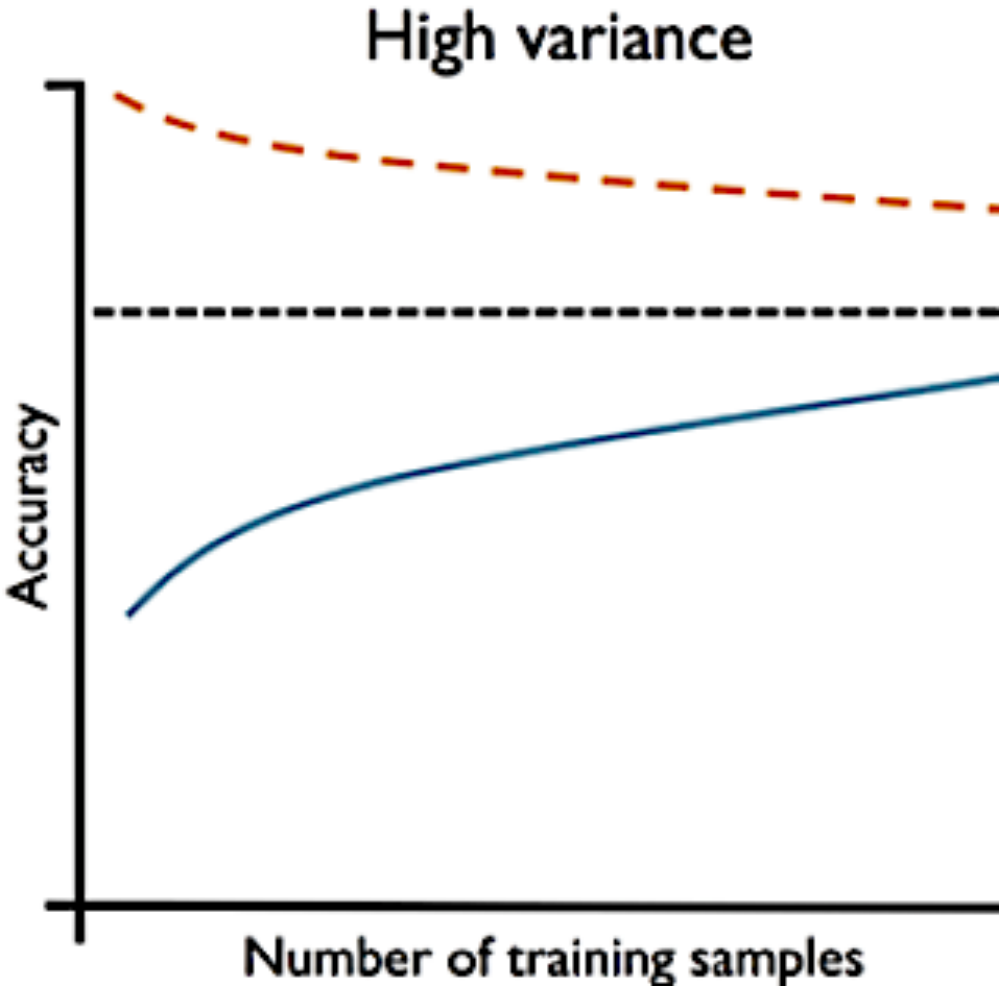
- Learning curves help to visualize how the model is "*learning*" through the training, and
- How the model is optimizing its "*learning*" through the prediction
- Overfitting & Underfitting can be spotted through the learning curves
  - These issues can sometimes be address by collecting more data
  - Learning curves can help you identify if you do need more data

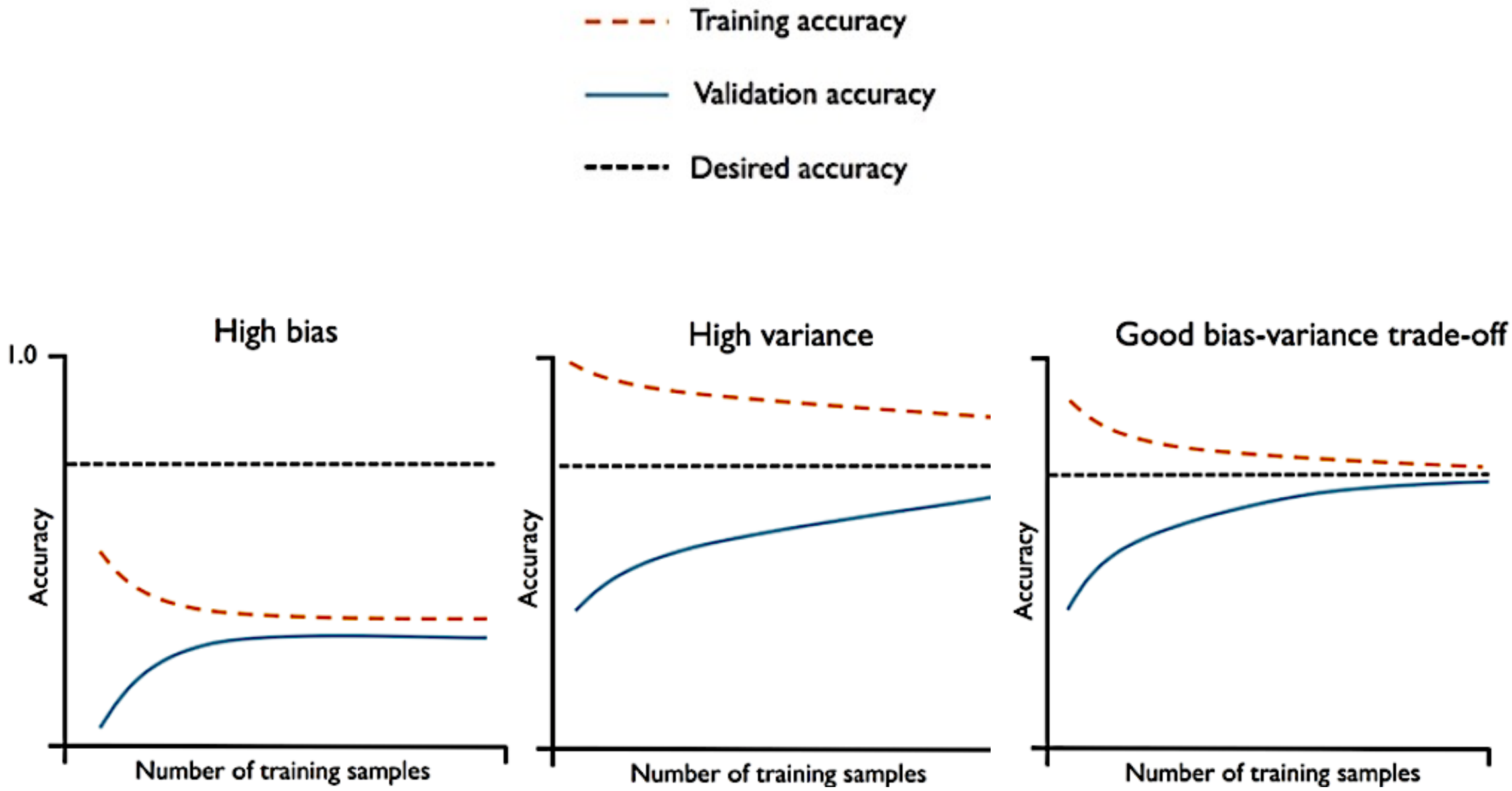

Training accuracy
Validation accuracy
Desired accuracy
High bias
High variance
Good bias-variance trade-off
1.0
Accuracy
Number of training samples
Accuracy
Number of training samples
Accuracy
Number of training samples

- Underfitting (*High Bias*)
- **How to fight this disease**:
  - Increase the number of parameters (*features*)
  - Decrease the degree of penalty (*regularization*)
    - Don’t over-penalize your model
  - (*Always*) collect more data



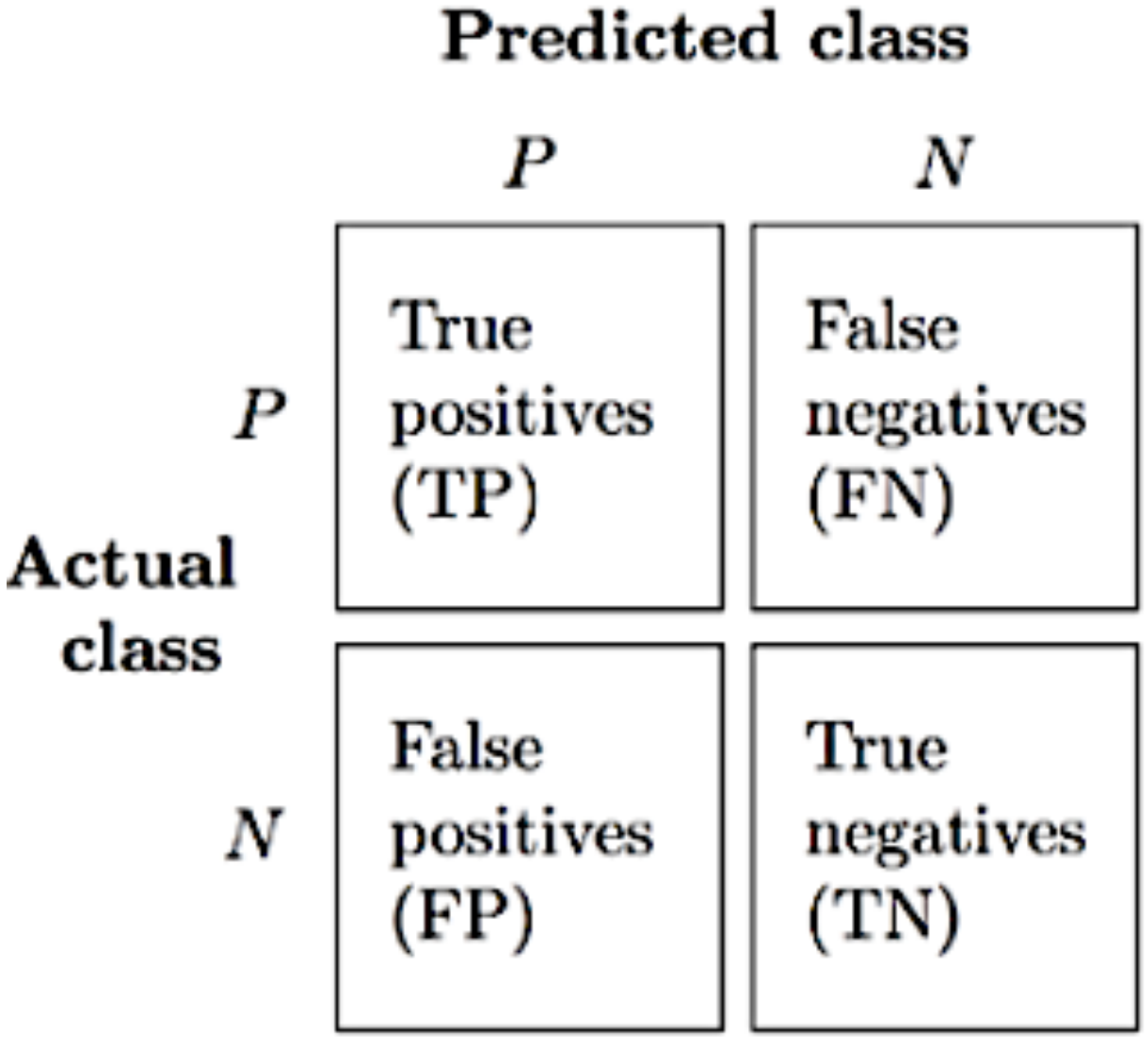
- Overfitting (*High Variance*)
- **How to fight this disease**:
  - Increase the number of parameters (*features*)
  - Decrease the degree of penalty (*regularization*)
    - Don’t over-penalize your model
  - (*Always*) collect more data



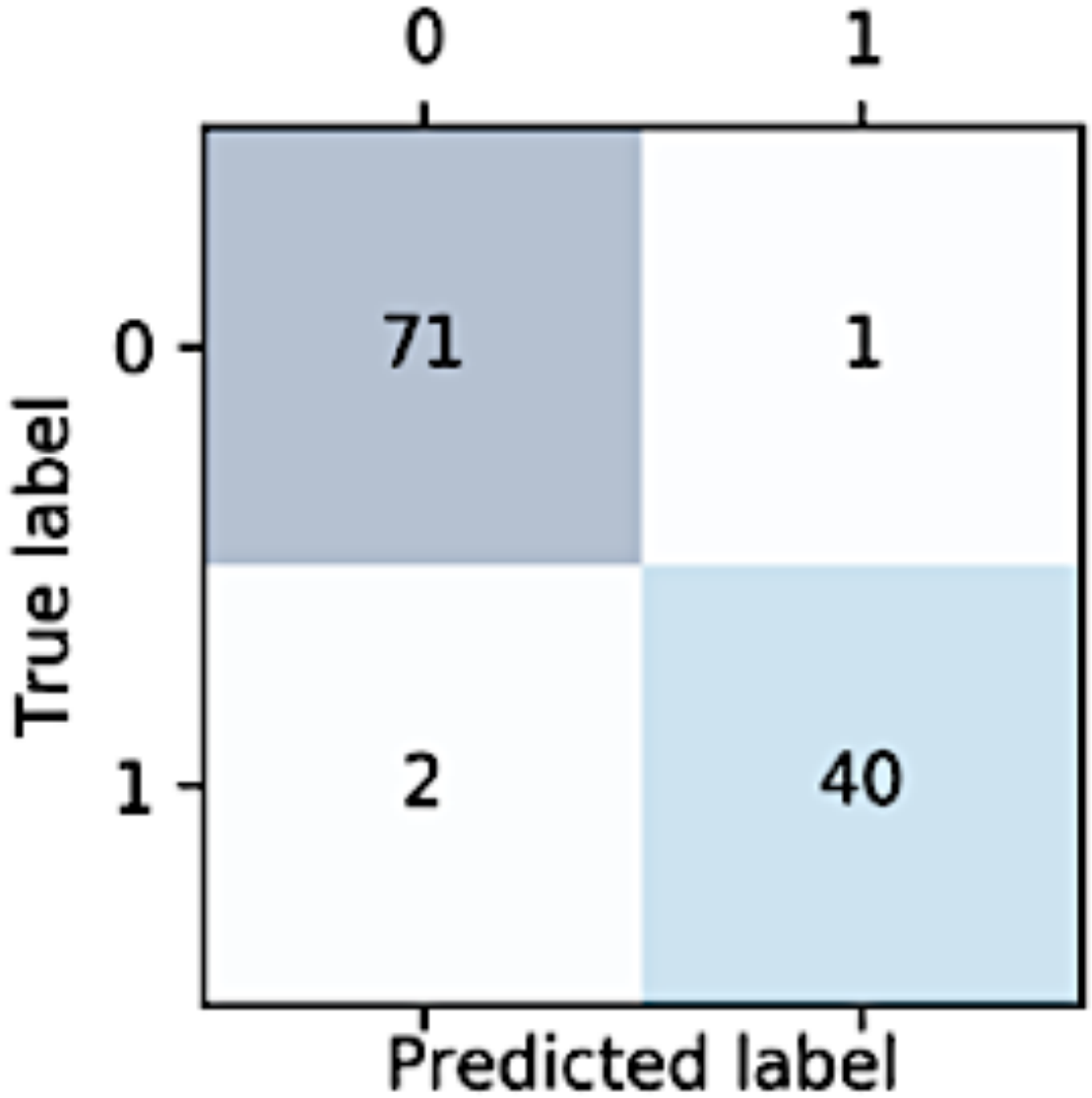

Training accuracy
Validation accuracy
Desired accuracy
High bias
High variance
Good bias-variance trade-off
1.0
Accuracy
Number of training samples
Accuracy
Number of training samples
Accuracy
Number of training samples

Confusion Matrix

- A graphical matrix that show the number or percentage of the what is:
  - A true-positive (TP)
  - A false-positive (FP)
  - A false-negative (FN)
  - A true-negative (TN)



```
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))
#Matplotlib's matshow
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.show()
```


0
1
0
1
True label
71
1
2
40
Predicted label

Precision & Recall

- Models are often graded on the accuracy value
- Sometimes, accuracy doesn't provide the entire picture about the dataset, the training, and how the accuracy was obtained.
- Error is also utilized in how to grade a model

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

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

- Rates (*fractions*) of the results also help in identifying the increases or decreases of certain classes

- **True-Positive Rate** (TPR) &
- **False-Positive Rate** (FPR)

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

- TPR – fraction of positive samples correctly ID'd from all positives

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

- FPR – if this is high then it bad!
  Meaning: your true-negatives are missing the mark

- Other (*more utilized*) metrics:
  - Precision (PRE)
  - Recall (REC)
  - F1-Score (mix of PRE & REC)

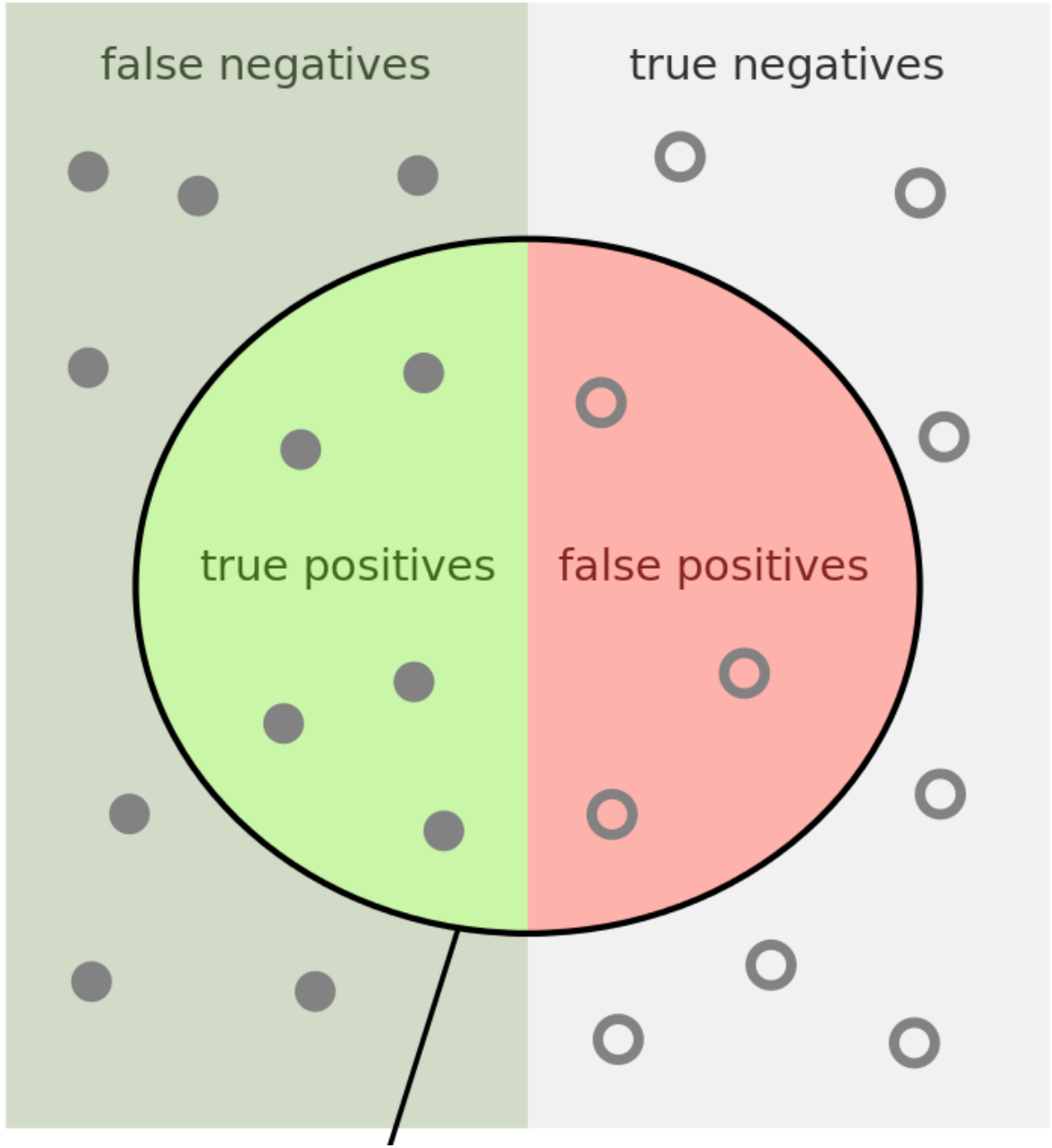
$$PRE = \frac{TP}{TP + FP}$$

- What if the F1-Score?

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

[Wiki: *The F1 score is the harmonic mean of the precision and recall, where an F1 score reaches its best value at 1 (perfect precision and recall) and worst at 0.*]

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



Precision = 

Recall = 

Ways to compute each metric:

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

print('Precision: %.3f' % precision_score(
                ... y_true=y_test, y_pred=y_pred))
Precision: 0.976

print('Recall: %.3f' % recall_score(
                ... y_true=y_test, y_pred=y_pred))
Recall: 0.952

print('F1: %.3f' % f1_score(
                ... y_true=y_test, y_pred=y_pred))
F1: 0.964
```

Receiver Operating Characteristic

- Receiver Operating Characteristic (ROC) is a graphical analysis to determine how a model's rate aligns with other models and extremes

- It uses the idea of the FPR & TPR (*the rate in how positive the model is*)

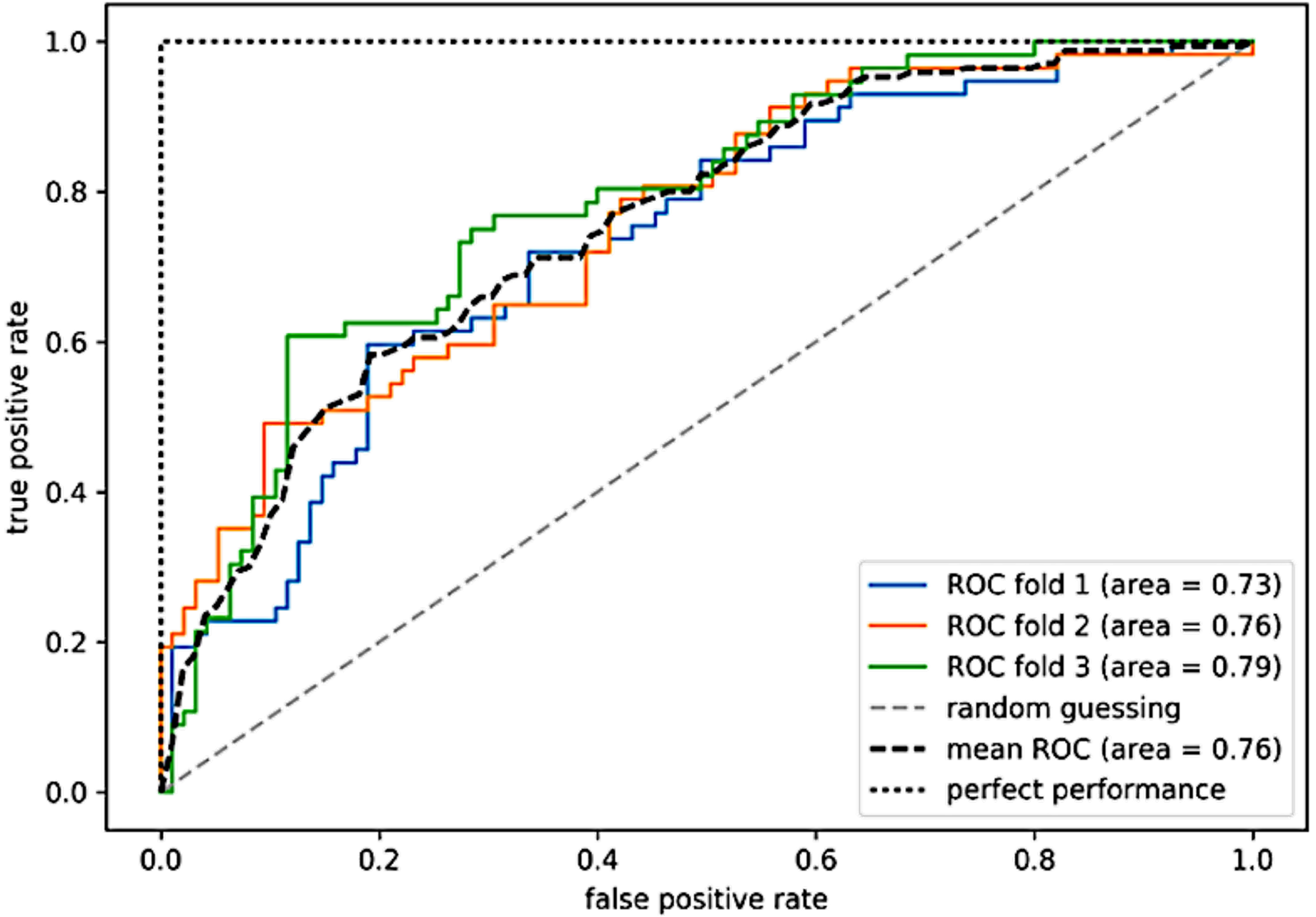
- **The extremes**:
  - The perfect model = 1
  - The random guess = 0

- The curve(s) is important, but the Area Under the Curve (AUC) is more critical

```
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))
```

**** Rest of the plotting code in the textbook***


1.0
0.8
0.6
0.4
0.2
0.0
true positive rate
0.0
0.2
0.4
0.6
0.8
1.0
false positive rate
ROC fold 1 (area = 0.73)
ROC fold 2 (area = 0.76)
ROC fold 3 (area = 0.79)
random guessing
mean ROC (area = 0.76)
perfect performance