Lecture 11: Model Evaluation Part 2 – model diagnose, parameter tuning

(textbook chapter 6)

Dr. Huiping Cao

Diagnostic tools

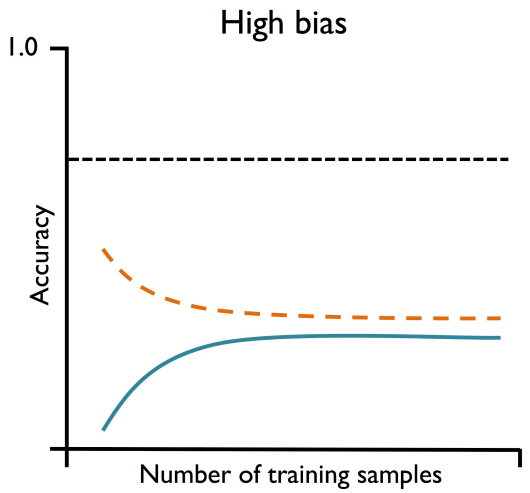
- Learning curves
- Validation curves

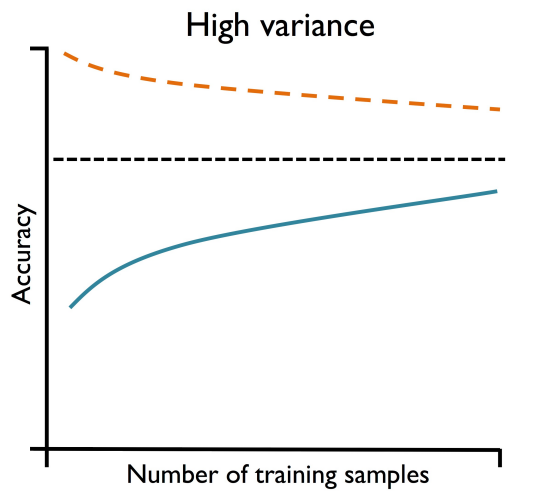
Learning curves

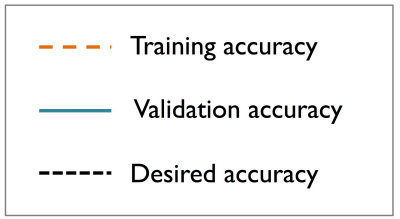
- **Learning curves**: can help us to diagnose whether a learning algorithm has a problem of overfitting (high variance) or underfitting (high bias).
- When a model is unnecessarily complicated, it tends to overfit the training data and does not generalize well to unseen data.
- Plotting the model training and validation accuracies as functions of the training set size can help
  - (1) detect whether a model suffers from high variance or high bias, and
  - (2) whether collecting more data will help address the issue.

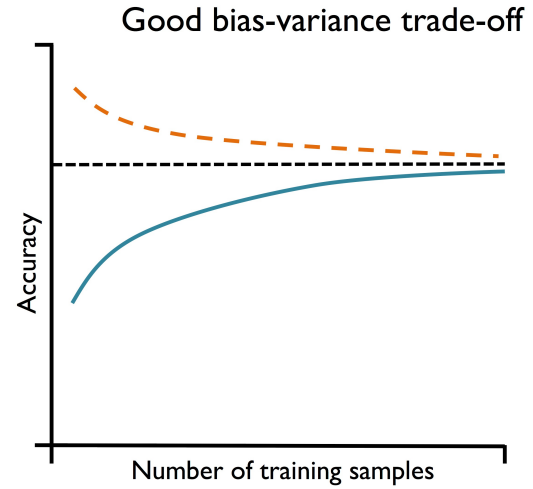
Learning curves

- Low training and cross-validation accuracy. The model underfits the data (high bias)
  - Address the issue: (1) increase the number of parameters (e.g., collect additional features), and/or (2) decrease the degree of regularization.
- Higher training accuracy and lower validation accuracy. Overfitting (high variance).
  - Address the issue: (1) collect more data, (2) reduce the model complexity, (3) increase the regularization, or (4) reduce features.









Example

- The **learning_curve** function uses stratified k-fold cross-validation

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

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

train_sizes, train_scores, test_scores =\
                learning_curve(estimator=pipe_svc,
                               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)
```

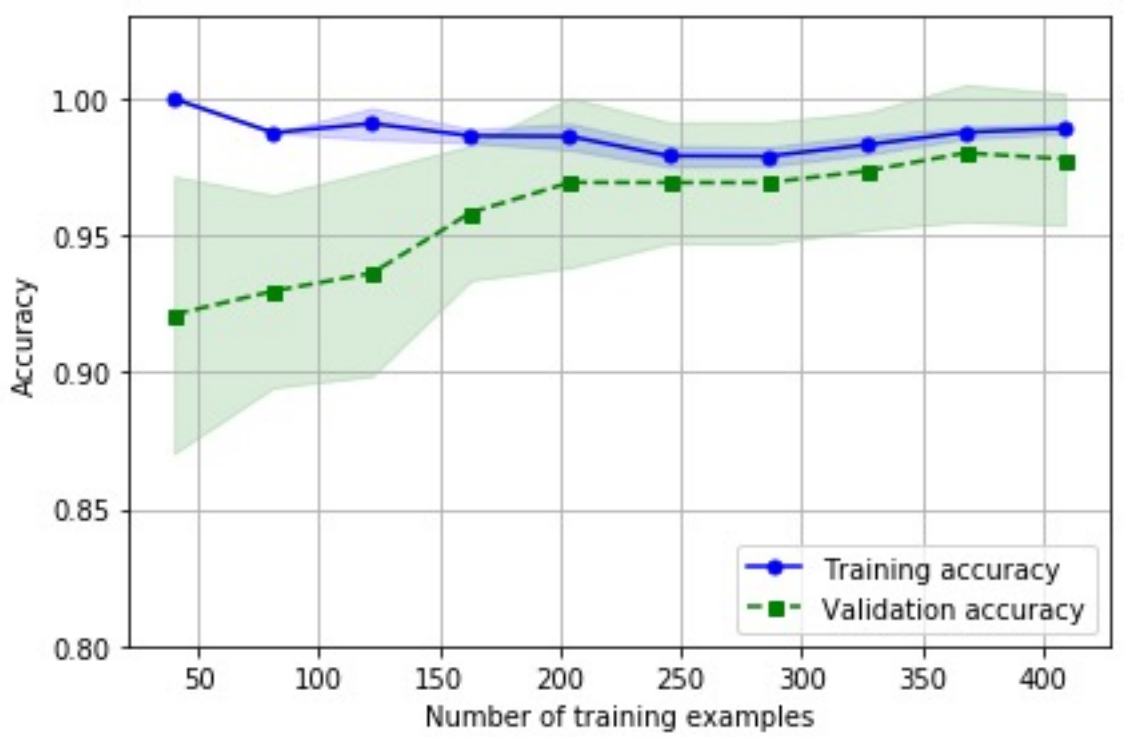
```
plt.plot(train_sizes, train_mean, color='blue', marker='o',
        markersize=5, label='Training accuracy')

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 examples')
plt.ylabel('Accuracy')
plt.legend(loc='lower right')
plt.ylim([0.8, 1.03])
plt.tight_layout()
plt.show()
```



Validation curves

- **Validation curves**: can help address common issues of a learning algorithm. Vary the values of the model parameters.
- Similar to learning curves, it gets both training and validation accuracies.
- Different from learning curves, it does not vary the size of training samples. Instead, we vary **the values of the model parameters** to get validation curves.

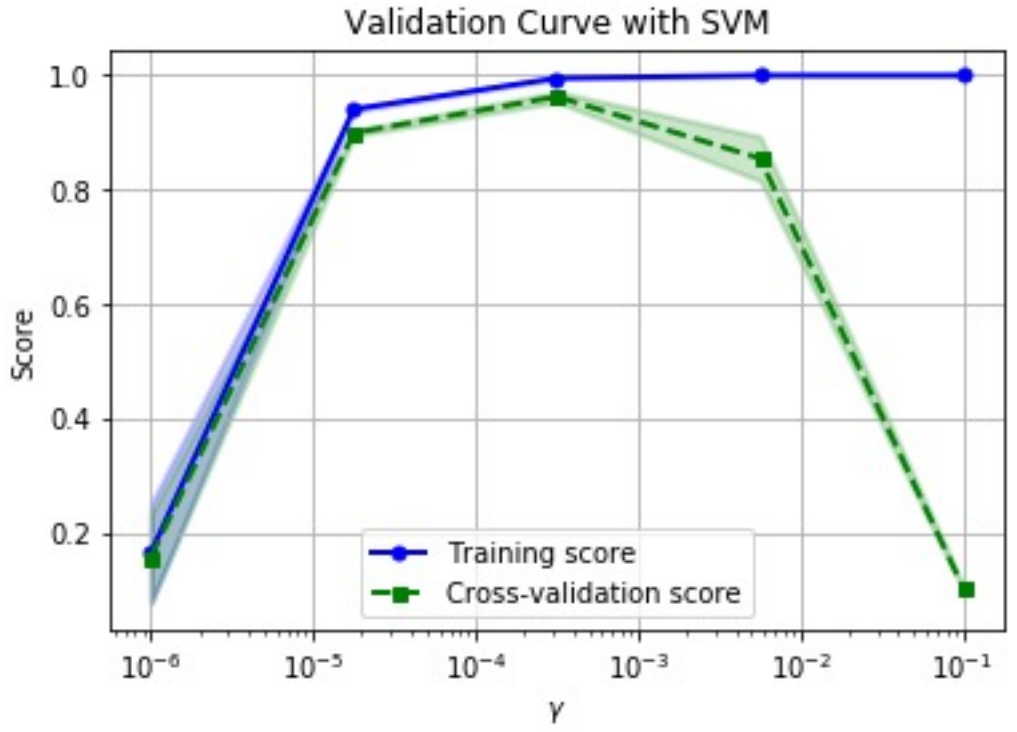
Validation curves

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

param_range = np.logspace(-6, -1, 5)
train_scores, test_scores = validation_curve(
    SVC(), X=X_train,y=y_train,
    param_name="gamma", param_range=param_range,
    scoring="accuracy", n_jobs=1)

train_scores_mean = np.mean(train_scores, axis=1)
train_scores_std = np.std(train_scores, axis=1)
test_scores_mean = np.mean(test_scores, axis=1)
test_scores_std = np.std(test_scores, axis=1)
```

Code reference: here



Parameter tuning via grid search

- Two types of parameters
  - Parameters learned from the training data (e.g., weights)
  - **Hyperparameters**: parameters of a learning algorithm that are optimized separately. E.g.,
    - The learning rate of perceptron and Adaline models
    - The C and γ parameters in the SVC function.
- Validation curves can improve a model's performance by tuning one hyperparameter.
- **Grid search technique** is another popular hyperparameter optimization technique. It can help improve the performance of a model by finding the optimal combination of hyperparameter values.

Grid search technique

- Grid search is a brute-force exhaustive search paradigm.
- We specify a list of values for different hyperparameters. The computer evaluates the model performance for each combination of those to obtain the optimal combination of parameter values.
- Grid search, combined with k-fold cross-validation, is a useful approach for fine-tuning the performance of a machine learning model by varying the hyperparameter values.

```
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',
                  refit=True,
                  cv=10,
                  n_jobs=-1)
gs = gs.fit(X_train, y_train) #WDBC data

print(gs.best_score_)
print(gs.best_params_)
```

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

- **GridSearchCV** function
- Tune 3 hyperparameters (C, γ, kernel) of SVC function
- **best_score_** and **best_param_** attribute

GridSearchCV

- The **best_estimator_** attribute gives us the model with the best performance. We can directly use it to make predictions.

```
clf = gs.best_estimator_

print('Test accuracy: %.3f' % clf.score(X_test, y_test))
```

```
Test accuracy: 0.974
```

Randomized hyperparameter search

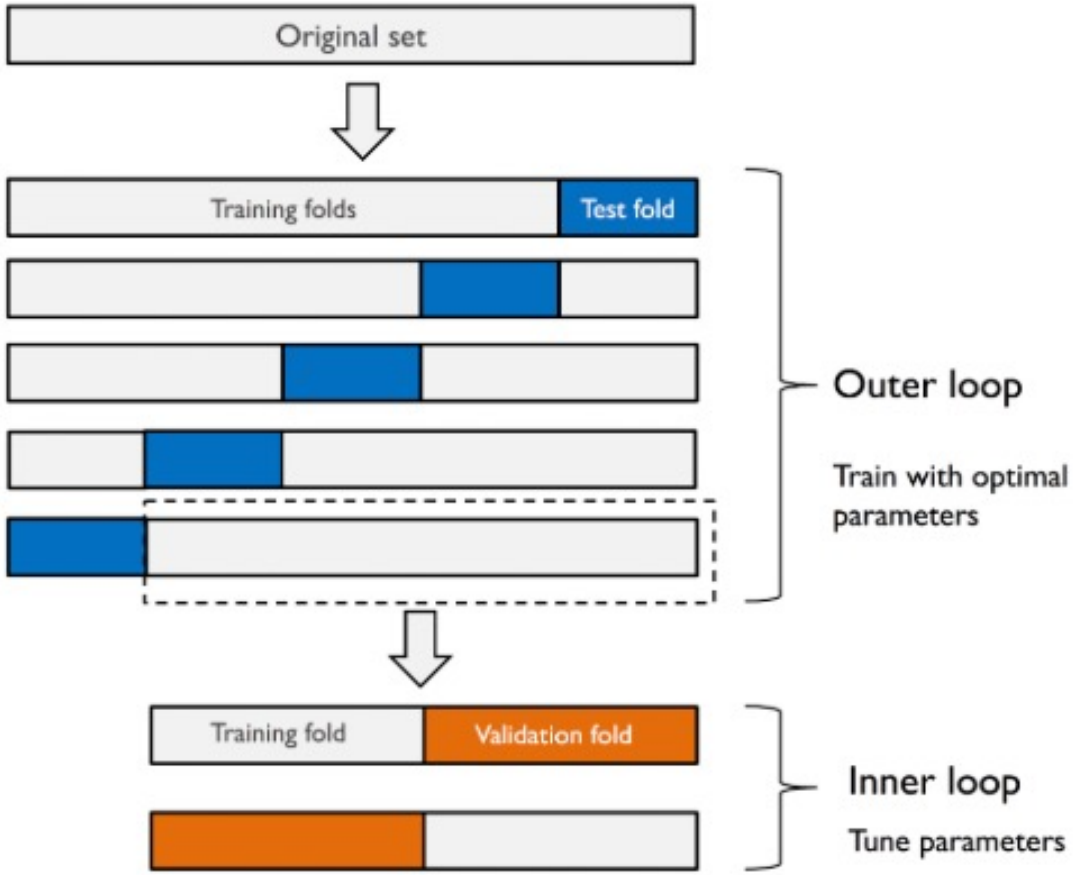
- **Disadvantage of grid search**: the evaluation of all possible parameter combinations is computationally very expensive.
- An alternative approach is to use **randomized hyperparameter search**. We can draw random parameter combinations from sampling distributions with a specified budget.
  - **RandomizedSearchCV** function

Nested cross-validation

- If we want to select among different machine learning algorithms, **nested cross-validation** is more often used.
- We specify a list of values for different hyperparameters. The computer evaluates the model performance for each combination of those to obtain the optimal combination of parameter values.

Nested cross-validation

- Outer loop: k-fold cross-validation loop to split the data into training and test folds
- Inner loop: select the model using k-fold cross-validation on the training fold
- 5×2 cross-validation: five outer and two inner folds



Example

```
from sklearn.model_selection import cross_val_score

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
```

```
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
```