Model Evaluation and Hyperparameter Optimization

Dr. Tran Anh Tuan

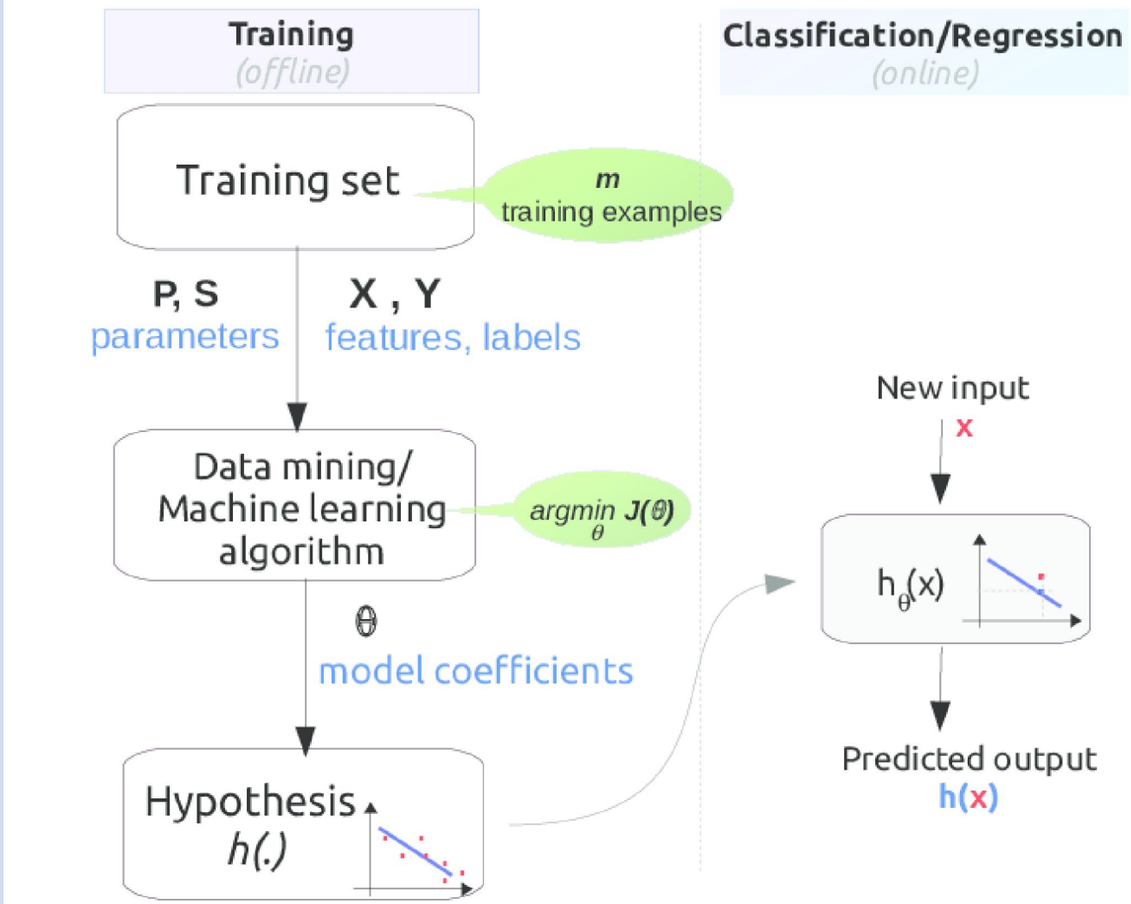
THEORY: Basic Recipe

The basic recipe for applying a supervised machine learning model:

1. Choose a class of model
2. Choose model hyper parameters
3. Fit the model to the training data
4. Use the model to predict labels for new data

➔ The choice of model and choice of hyper parameters—are perhaps the most important part of using these tools and techniques effectively

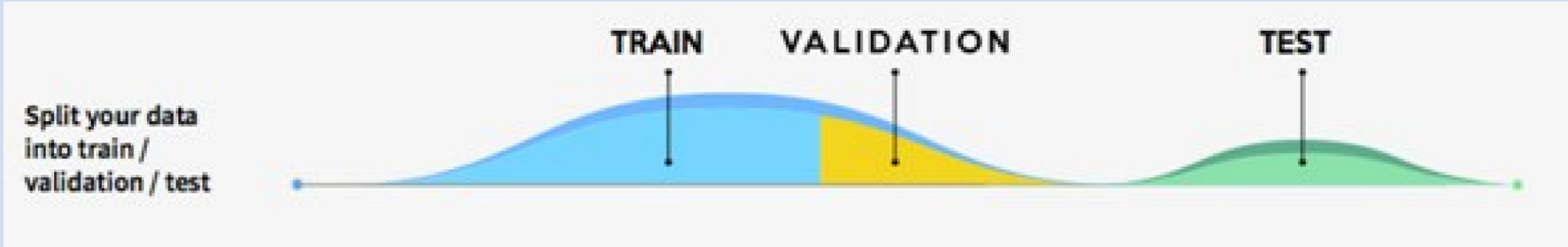
➔ In order to make an informed choice, we need a way to *validate* that our model and our hyperparameters are a good fit to the data

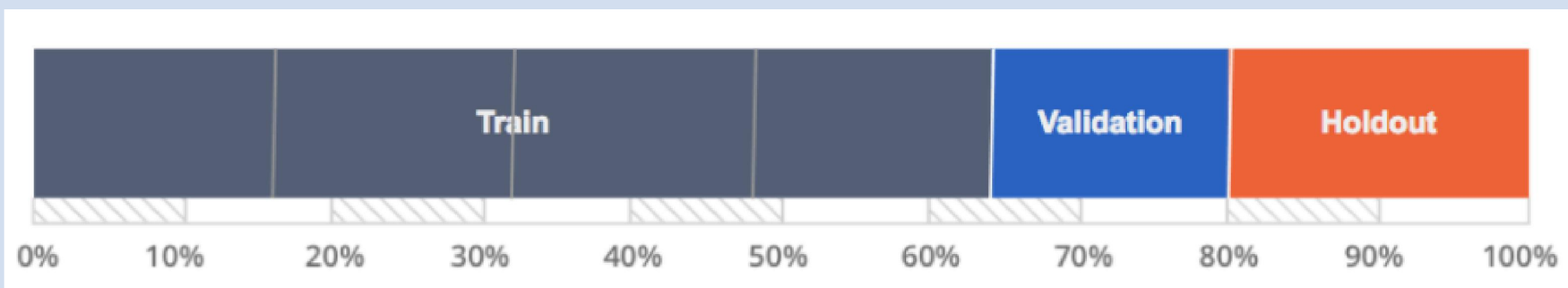


THEORY: Model Validation

In principle, model validation is very simple: after choosing a model and its hyperparameters, we can estimate how effective it is by applying it to some of the training data and comparing the prediction to the known value

Wrong approach : this approach contains a fundamental flaw: *it trains and evaluates the model on the same data*.
Right approach (Holdout sets) : That is, we hold back some subset of the data from the training of the model, and then use this holdout set to check the model performance.

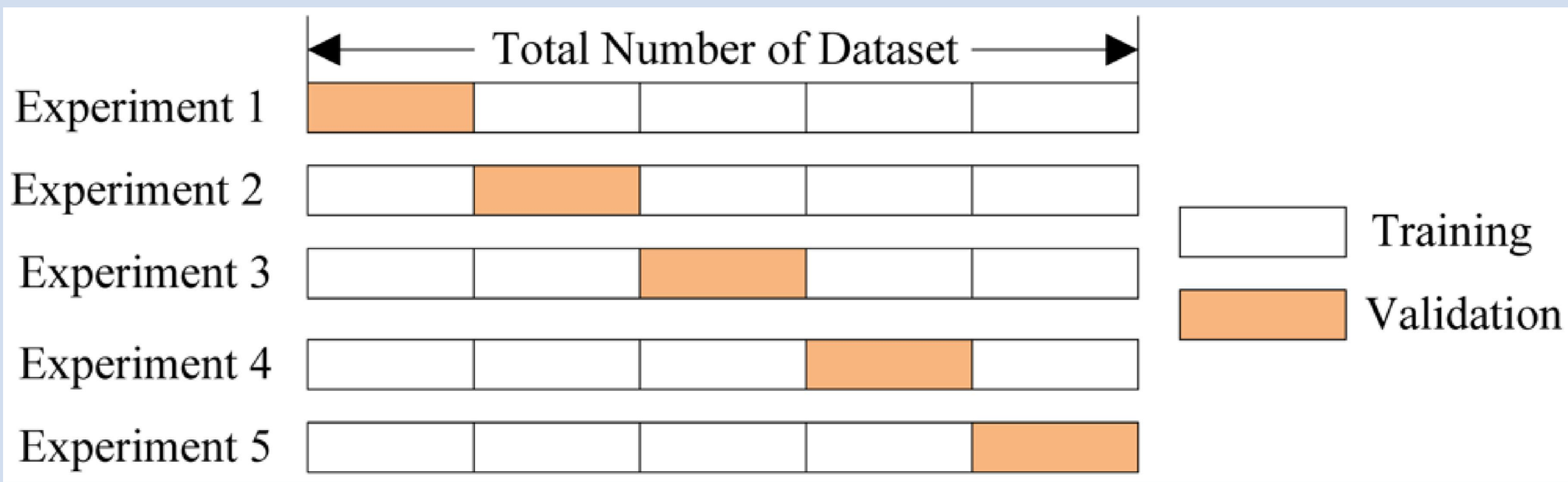




THEORY: Model Validation

Optimal approach (cross-validation) :

1. One disadvantage of using a holdout set for model validation is that we have lost a portion of our data to the model training. Half the dataset does not contribute to the training of the model.
2. One way to address this is to use *cross-validation*; that is, to do a sequence of fits where each subset of the data is used both as a training set and as a validation set
3. This particular form of cross-validation is a *two-fold cross-validation*—that is, one in which we have split the data into two sets and used each in turn as a validation set.



THEORY: Model Validation

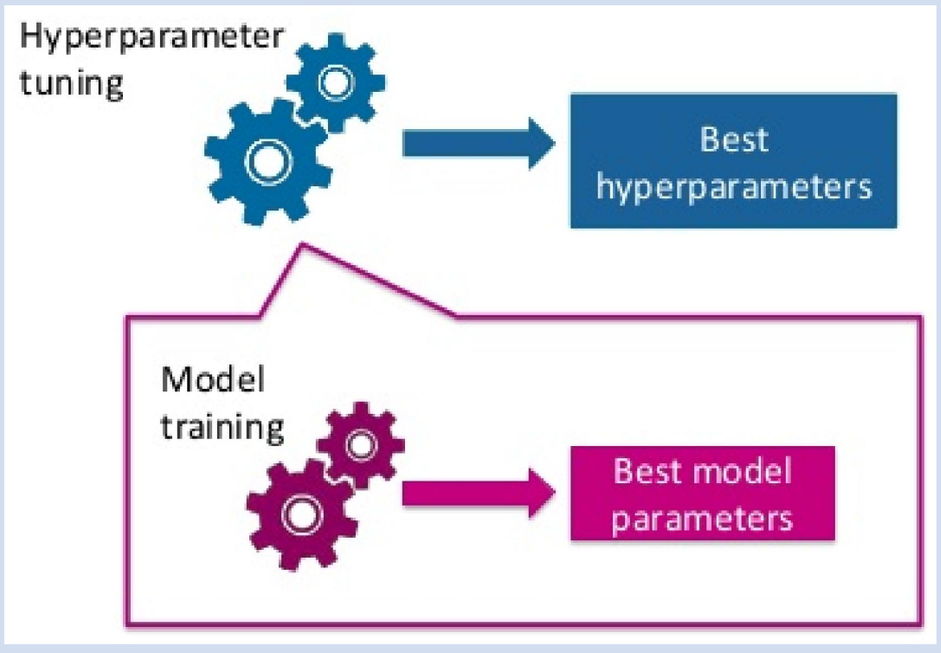
Leave-one-out cross-validation:
SVM classification with leave-one-out cross-validation, independent validations were conducted 80 times. In each independent validation, only one sample was selected as the testing set, and the remaining 79 samples were the training set. The average of the 80 SVM classifiers' accuracies was regarded as the final classification accuracy of the SVM method.

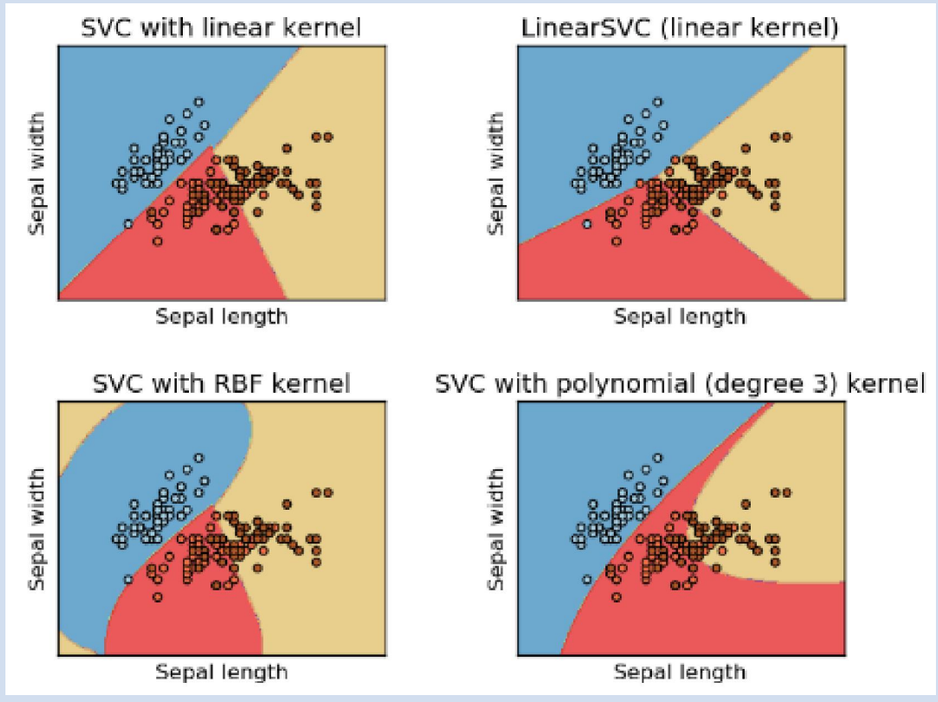
Diagram of leave-one-out cross-validation (k-fold cross-validation with k = 80).

THEORY: Selecting the Best Model

The core importance is the following question: *if our estimator is underperforming, how should we move forward?* There are several possible answers:

- Use a more complicated/more flexible model
- Use a less complicated/less flexible model
- Gather more training samples
- Gather more data to add features to each sample

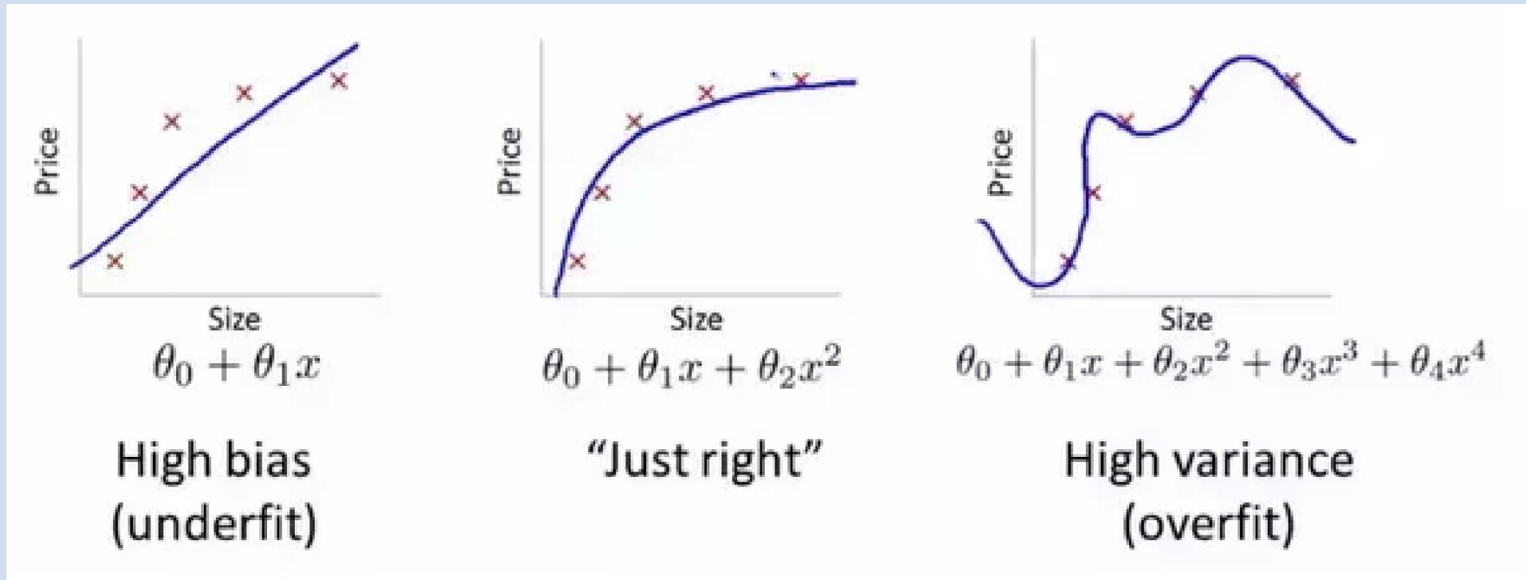




THEORY: The Bias-variance trade-off

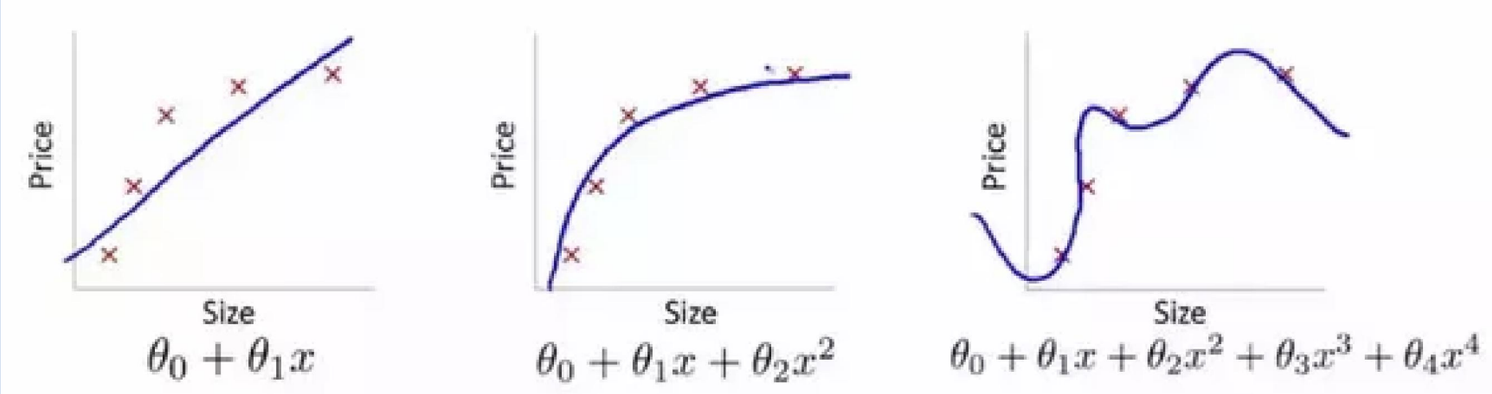
The question of "the best model" is about finding a sweet spot in the tradeoff between bias and variance.

1. (High Bias Model) : The model attempts to find a straight-line fit through the data. Because the data are intrinsically more complicated than a straight line, the straight-line model will never be able to describe this dataset well. Such a model is said to *underfit* the data. It does not have enough model flexibility to suitably account for all the features in the data; another way of saying this is that the model has high *bias*.



THEORY: The Bias-variance trade-off

The question of "the best model" is about finding a sweet spot in the tradeoff between bias and variance.
2. (High Variance Model): The model attempts to fit a high-order polynomial through the data. Here the model fit has enough flexibility to nearly perfectly account for the fine features in the data, but even though it very accurately describes the training data, its precise form seems to be more reflective of the particular noise properties of the data rather than the intrinsic properties of whatever process generated that data. Such a model is said to overfit the data. It has so much model flexibility that the model ends up accounting for random errors as well as the underlying data distribution; another way of saying this is that the model has high variance.



THEORY: R^2 Score

The score here is the R^2 score, or coefficient of determination, which measures how well a model performs relative to a simple mean of the target values. R^2=1 indicates a perfect match, R^2=0 indicates the model does no better than simply taking the mean of the data, and negative values mean even worse models.

From the scores associated with these two models, we can make an observation that holds more generally:

- For high-bias models, the performance of the model on the validation set is similar to the performance on the training set.
- For high-variance models, the performance of the model on the validation set is far worse than the performance on the training set.

- The total sum of squares (proportional to the variance of the data):

$$SS_{\text{tot}} = \sum_i (y_i - \bar{y})^2,$$

- The regression sum of squares, also called the explained sum of squares:

$$SS_{\text{reg}} = \sum_i (f_i - \bar{y})^2,$$

- The sum of squares of residuals, also called the residual sum of squares:

$$SS_{\text{res}} = \sum_i (y_i - f_i)^2 = \sum_i e_i^2$$

The most general definition of the coefficient of determination is

$$R^2 \equiv 1 - \frac{SS_{\text{res}}}{SS_{\text{tot}}}.$$

BAD MODEL

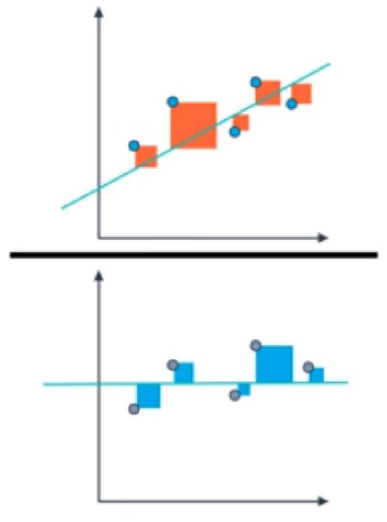
The errors should be similar.
R2 score should be close to 0.

GOOD MODEL

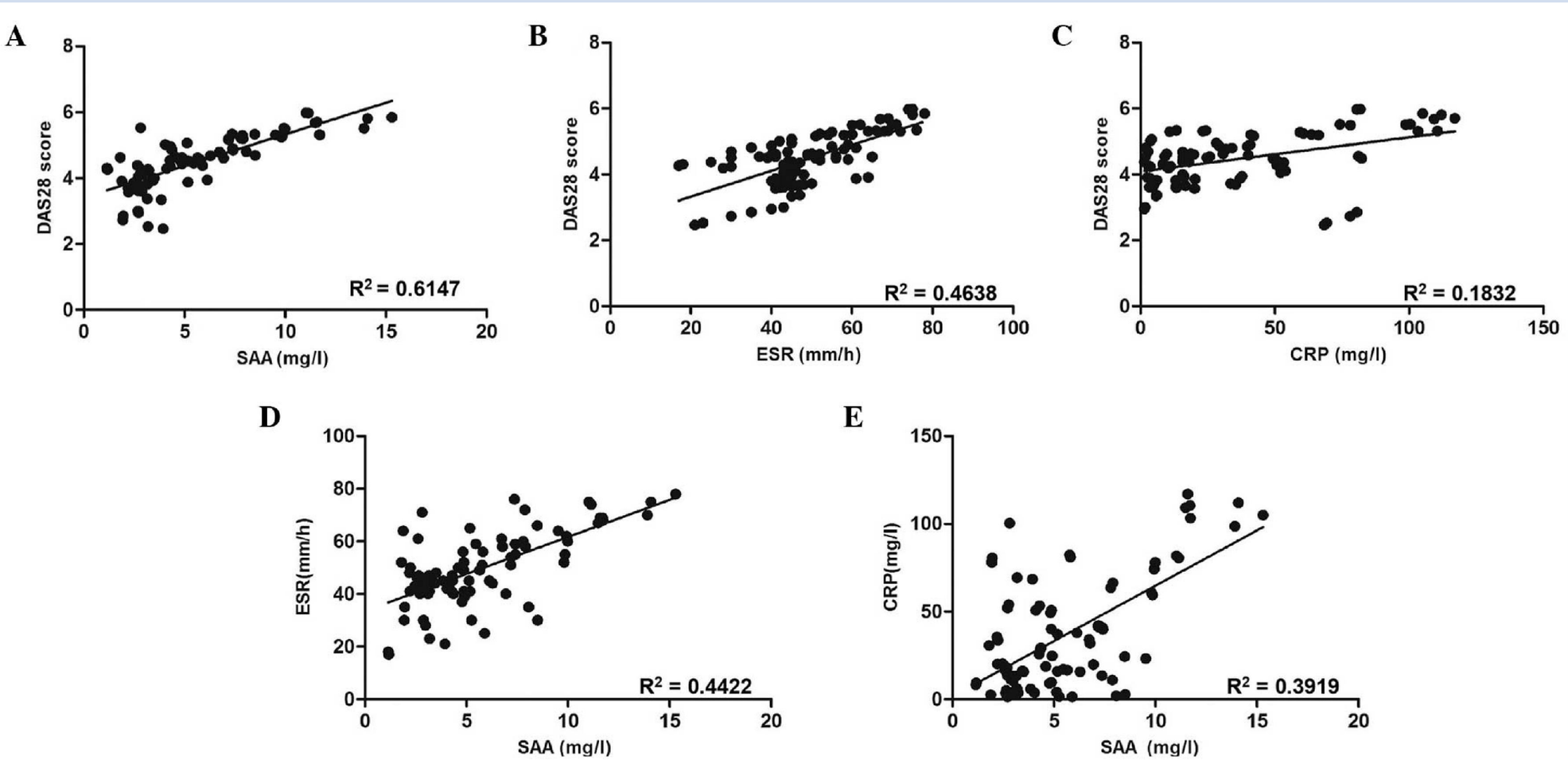
The mean squared error for the linear regression model should be a lot smaller than the mean squared error for the simple model.

R2 score should be close to 1.

R2 = 1 -



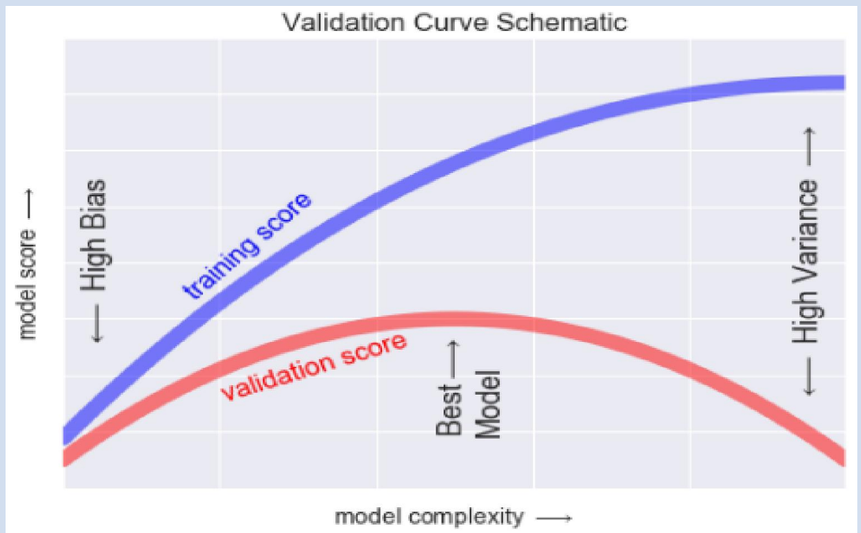
THEORY: R^2 Score

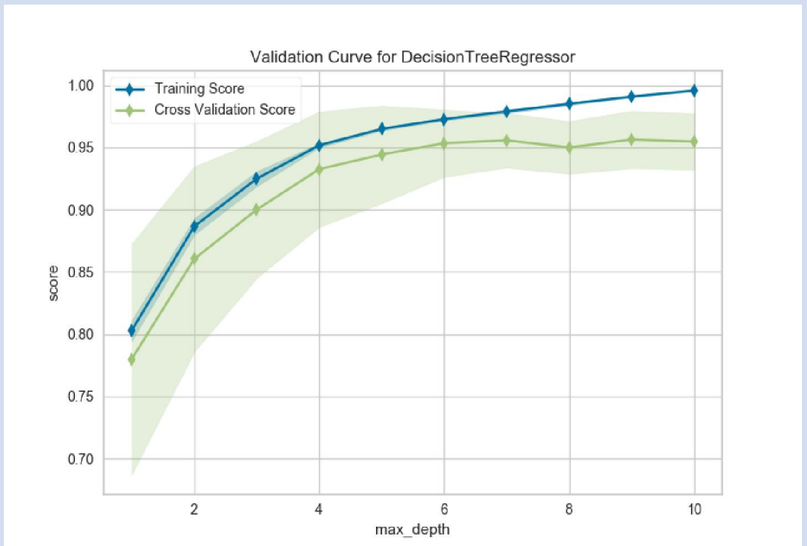


THEORY: Validation Curve

We see the following essential features:

1. The training score is everywhere higher than the validation score. This is generally the case: the model will be a better fit to data it has seen than to data it has not seen.
2. For very low model complexity (a high-bias model), the training data is under-fit, which means that the model is a poor predictor both for the training data and for any previously unseen data.
3. For very high model complexity (a high-variance model), the training data is over-fit, which means that the model predicts the training data very well, but fails for any previously unseen data.
4. For some intermediate value, the validation curve has a maximum. This level of complexity indicates a suitable trade-off between bias and variance.

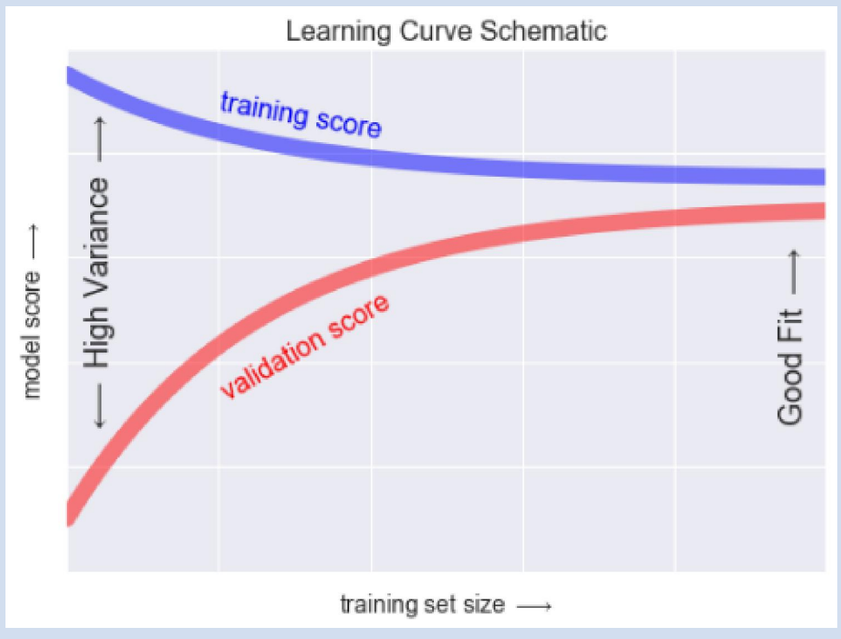




THEORY: Learning Curves

The general behavior we would expect from a learning curve is this:

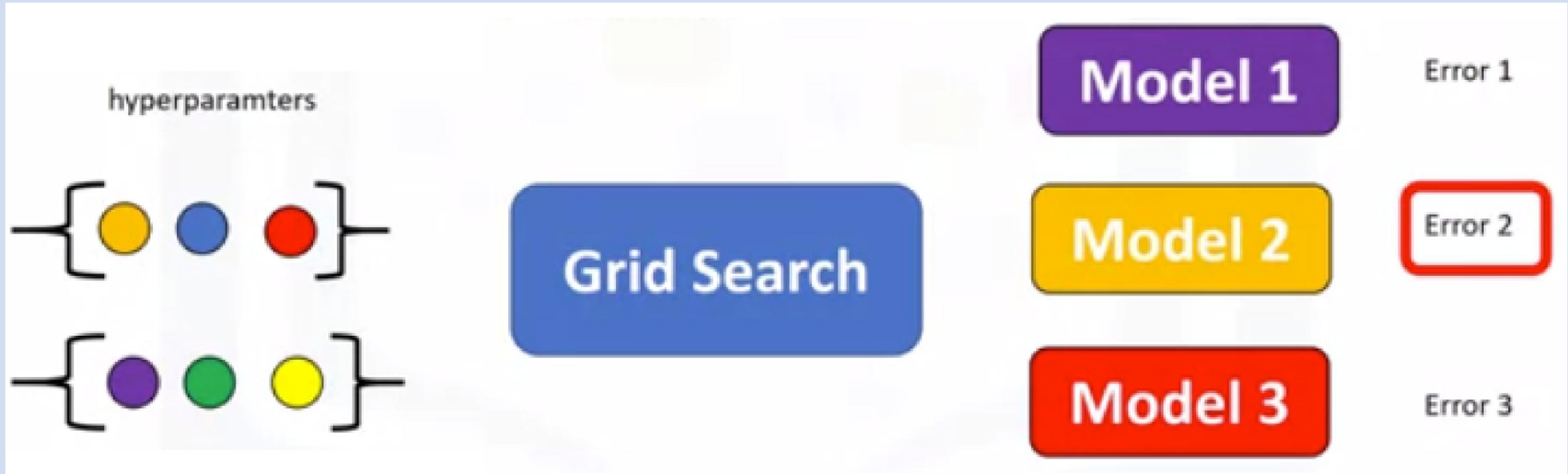
1. A model of a given complexity will *overfit* a small dataset: this means the training score will be relatively high, while the validation score will be relatively low.
2. A model of a given complexity will *underfit* a large dataset: this means that the training score will decrease, but the validation score will increase.
3. A model will never, except by chance, give a better score to the validation set than the training set: this means the curves should keep getting closer together but never cross.



One important aspect of model complexity is that the optimal model will generally depend on the size of your training data. It is often useful to explore the behavior of the model as a function of the number of training points, which we can do by using increasingly larger subsets of the data to fit our model.

THEORY: Grid Search

Grid Search
The traditional way of performing hyperparameter optimization has been ***grid search***, or a parameter sweep, which is simply an exhaustive searching through a manually specified subset of the hyperparameter space of a learning algorithm. A grid search algorithm must be guided by some performance metric, typically measured by cross-validation on the training set or evaluation on a held-out validation set.



THEORY: Grid Search

parameters = [{ 'alpha' : [1, 10, 100, 1000] }]

Alpha	1	10	100	1000

Ridge()

Scoring

Number of Folds

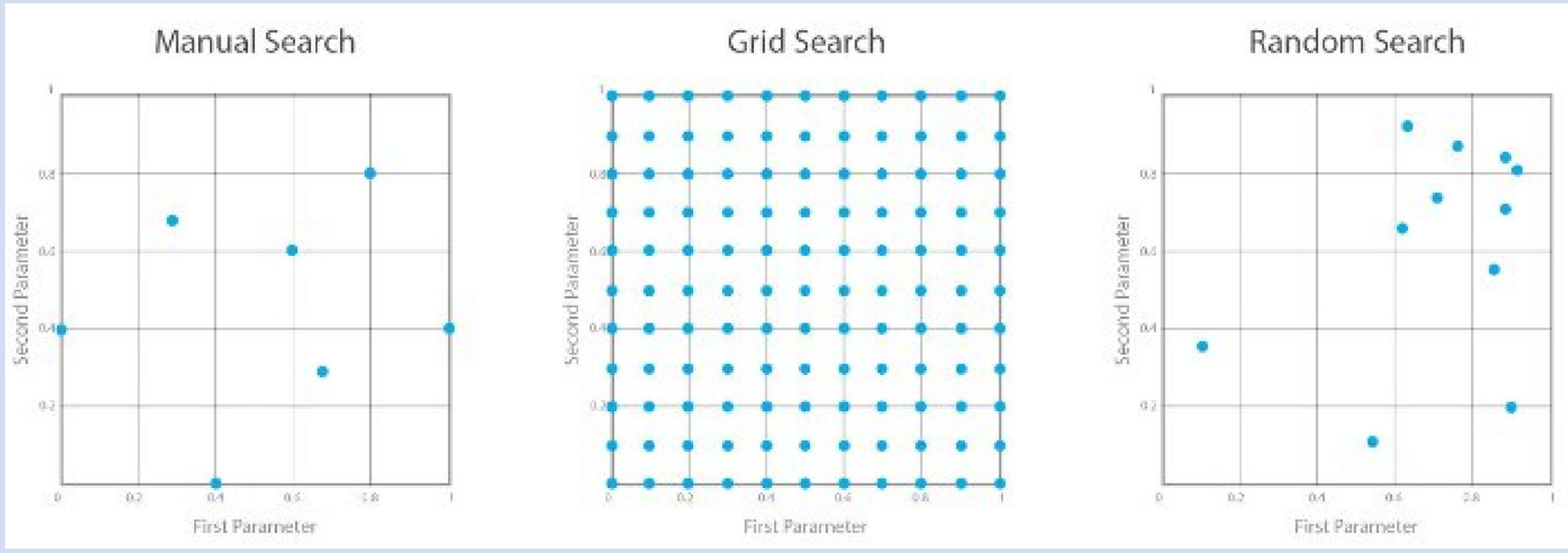
Grid Search CV

Alpha	1	10	100	1000

Alpha	1	10	100	1000
R^2	0.74	0.35	0.073	0.008

THEORY: *Random Search*

Random Search replaces the exhaustive enumeration of all combinations by selecting them randomly. This can be simply applied to the discrete setting described above, but also generalizes to continuous and mixed spaces. It can outperform Grid search, especially when only a small number of hyperparameters affects the final performance of the machine learning algorithm

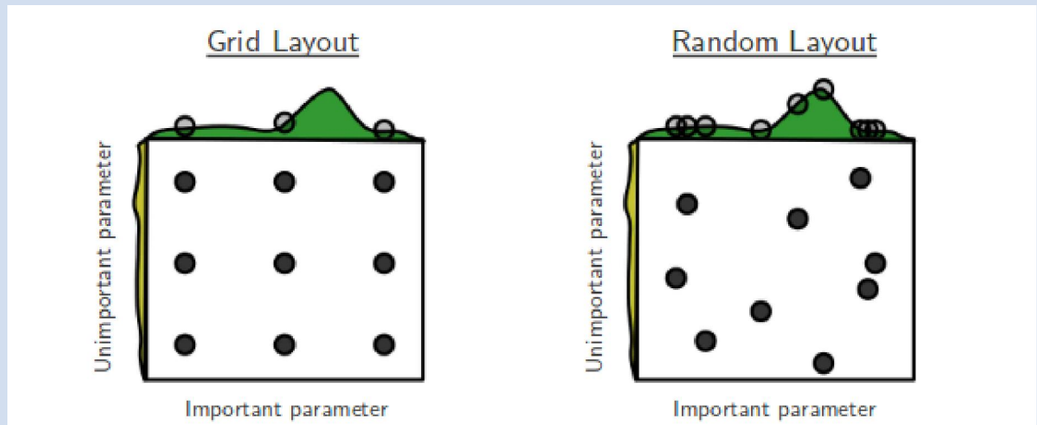


THEORY: *Random Search*

Algorithm (below) provides a pseudocode listing of the Random Search Algorithm for minimizing a cost function.

```
Input: NumIterations, ProblemSize, SearchSpace
Output: Best
Best ← ∅
For (iter_i ∈ NumIterations)
  candidate_i ← RandomSolution(ProblemSize, SearchSpace)
  If (Cost(candidate_i) < Cost(Best))
    Best ← candidate_i
  End
End
Return (Best)
```

Pseudocode for Random Search.



THEORY: Hyperparameter Optimization

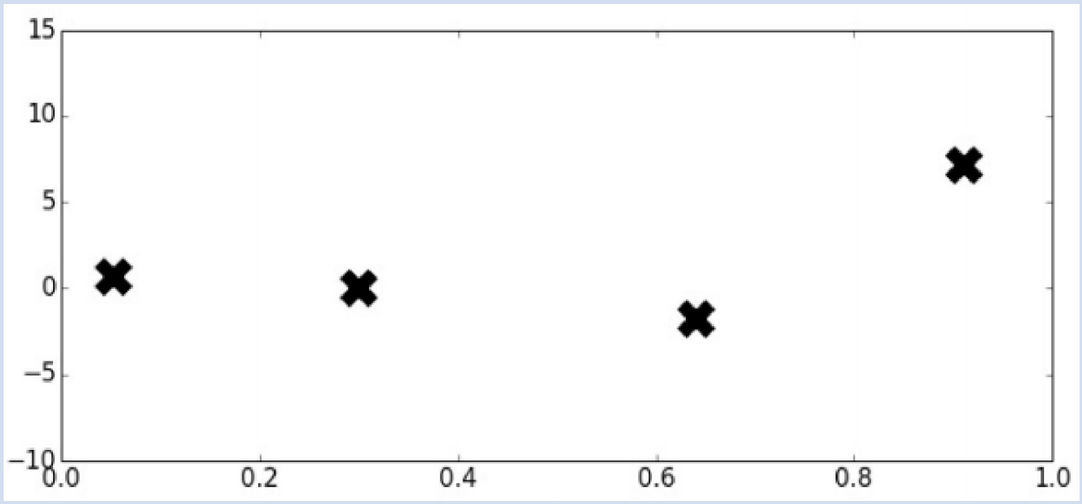
Bayesian optimization:

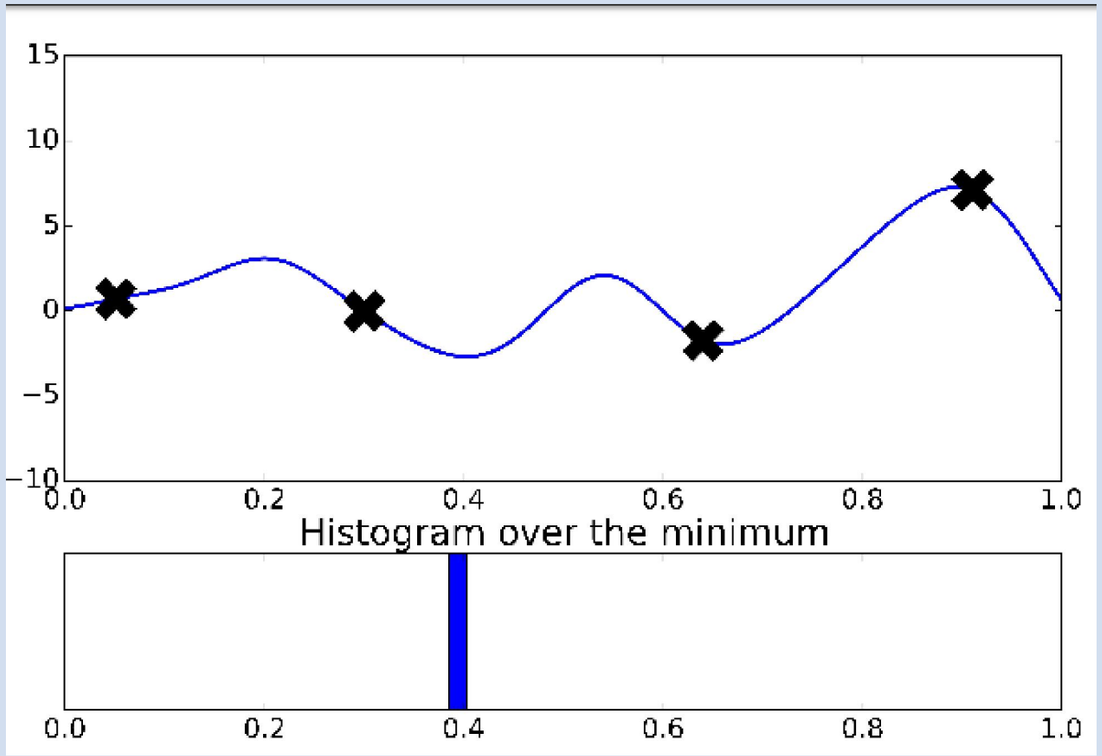
- Bayesian Optimization is a method that uses some kind of approximation. Imagine, if we don't know a function, what we usually do? Of course, we will try to guess or approximate it with some know prior knowledge. The same idea is behind the **posteriori probability**. The criteria here is that we have **observations**, where the data are coming records by records (**online-learning**), so we need to train this model. The trained model will obviously obey a function. That function that we don't know will absolutely depend on the **learnt data**. So our task is to find the **hyper-parameters** that **maximizes** the learning schema.

Bayesian optimization Process:

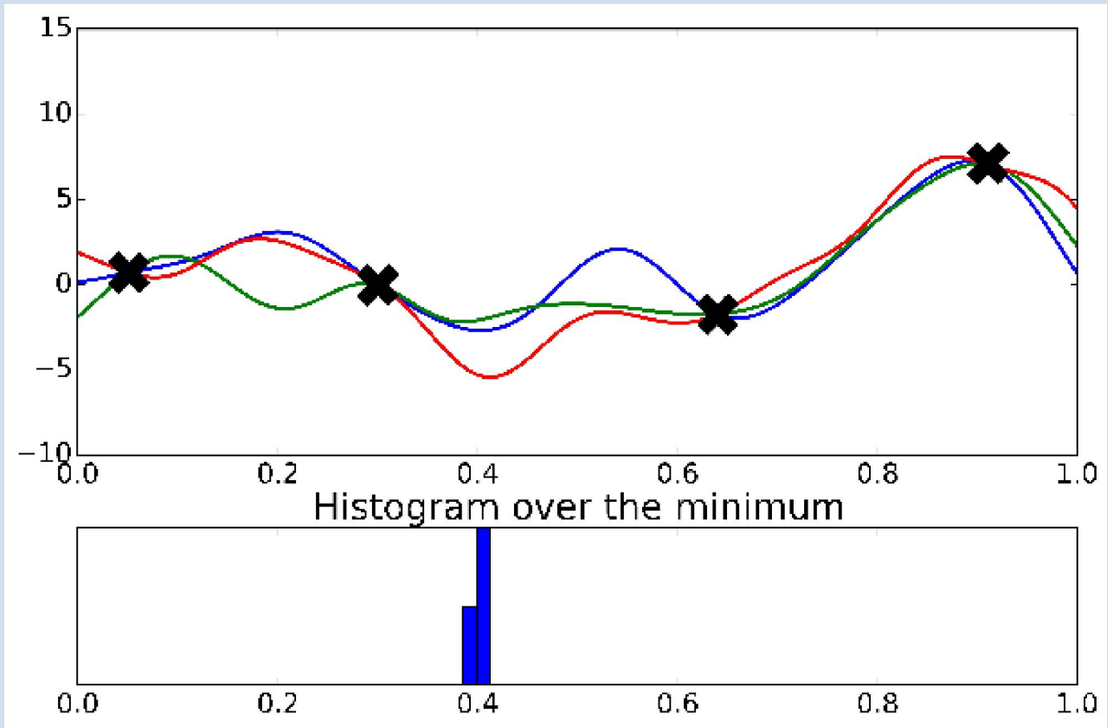
The *first observed point* comes to the model. we draw a line. Then the next point follows. We joint those 2 points and draw an adjusted line from previous curve. A third point comes, so we draw a non-linear curve. When the number of observed points increases, the number of combination for possible curves increases. *It is same like sampling theorem in Statistics where we estimate a population parameter from sample parameter*

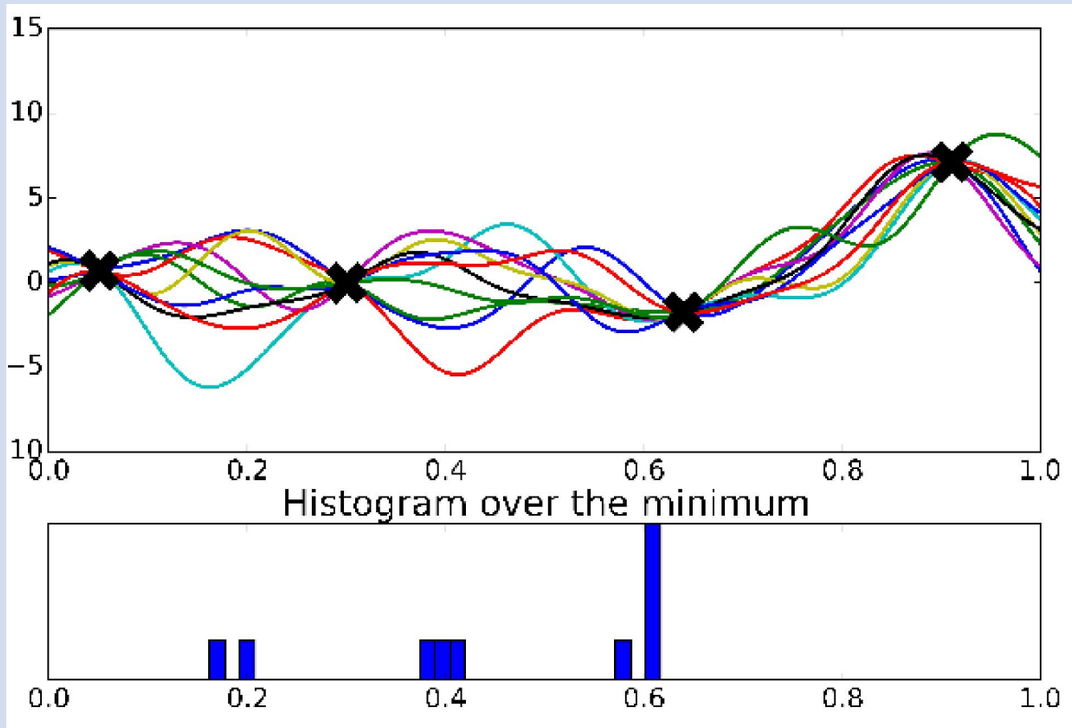
THEORY: Hyperparameter Optimization



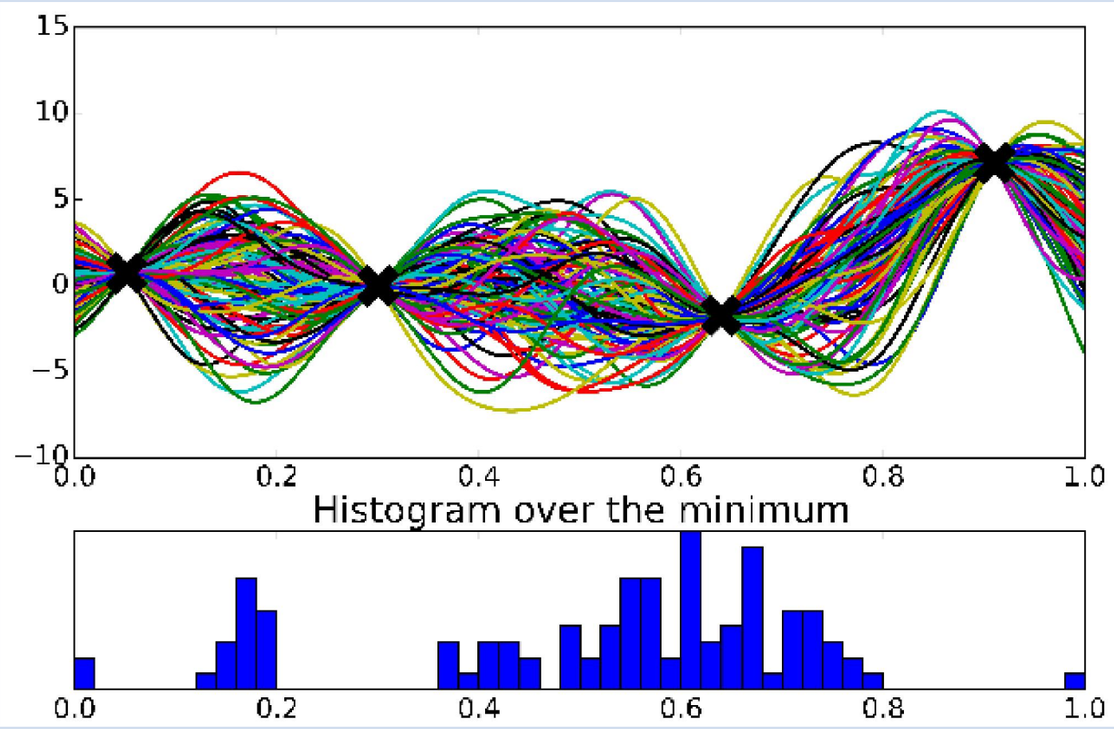


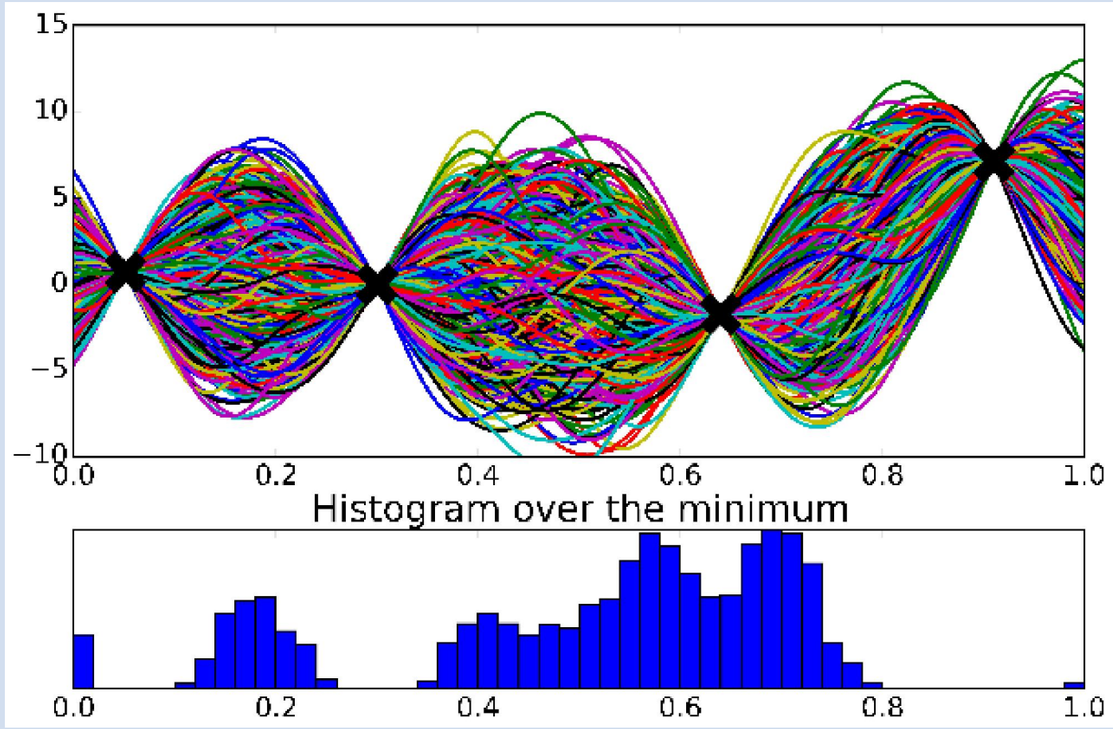
THEORY: Hyperparameter Optimization



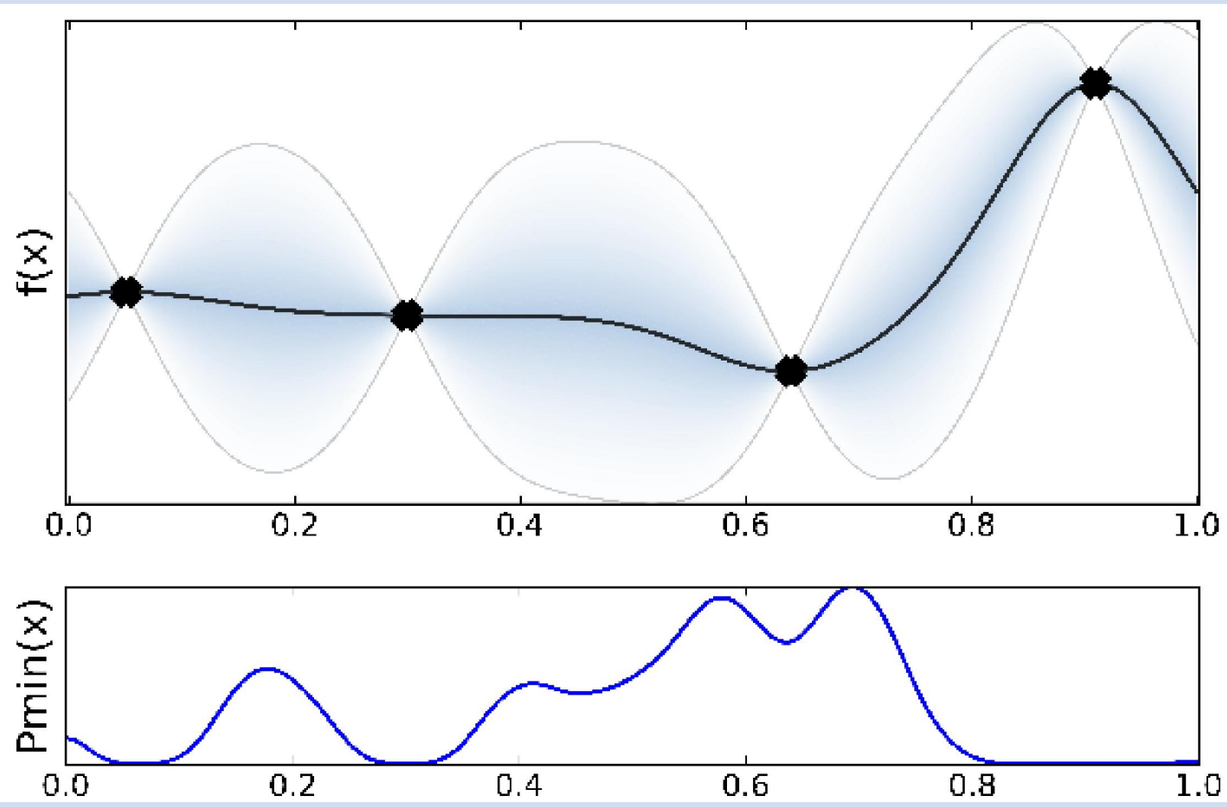


THEORY: Hyperparameter Optimization





THEORY: Hyperparameter Optimization



1. Use a surrogate model of f to carry out the optimization.

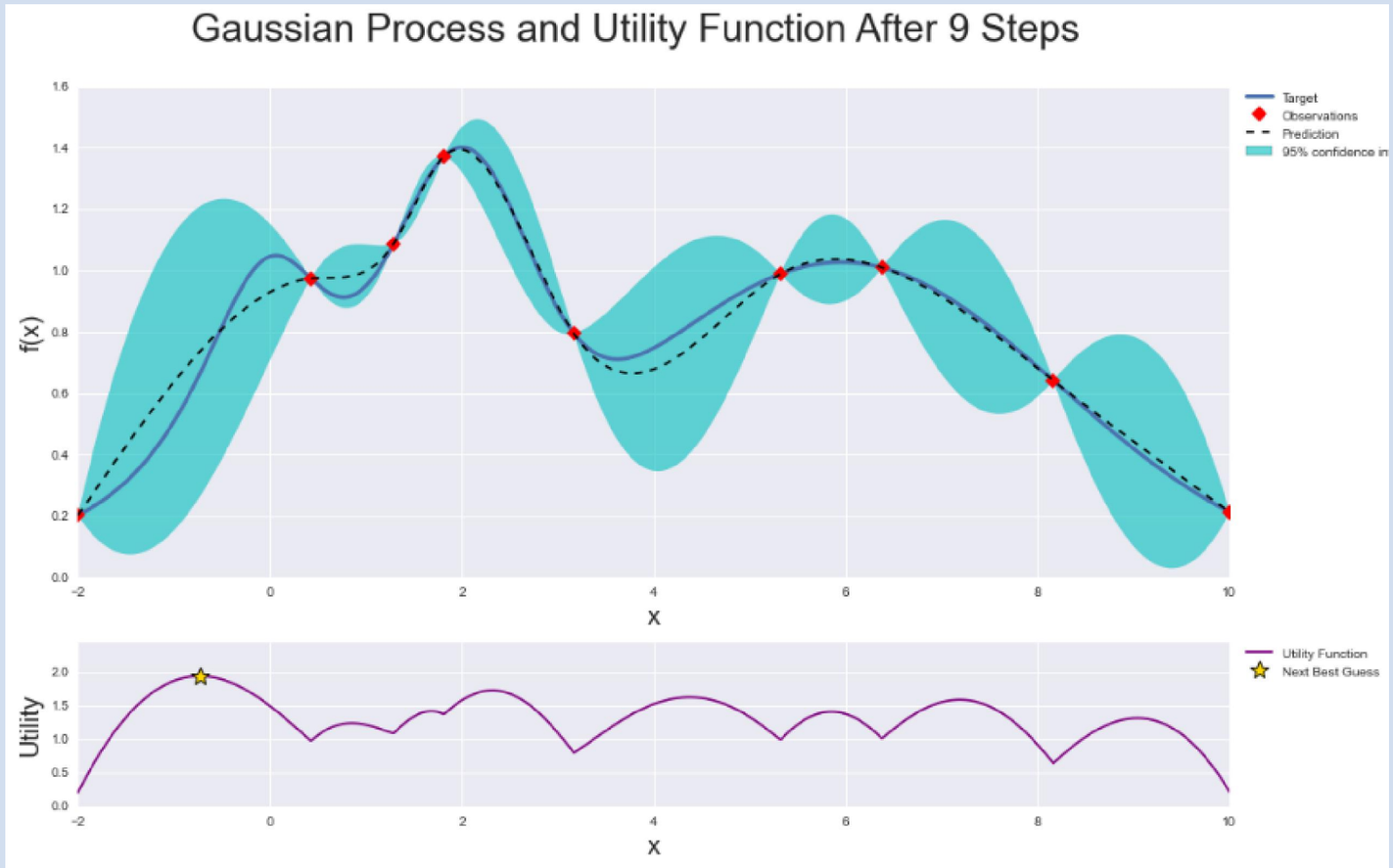
2. Define an utility function to collect new data points satisfying some optimality criterion: *optimization* as *decision*.

3. Study *decision* problems as *inference* using the surrogate model: use a probabilistic model able to calibrate both, epistemic and aleatoric uncertainty.

Uncertainty Quantification

THEORY: Hyperparameter Optimization

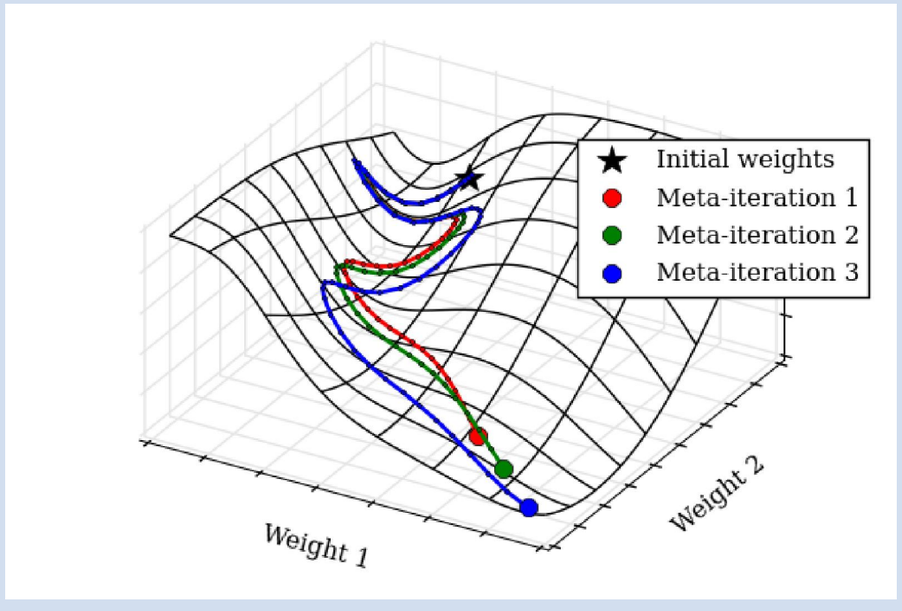
Bayesian optimization works by constructing a posterior distribution of functions (gaussian process) that best describes the function you want to optimize. As the number of observations grows, the posterior distribution improves, and the algorithm becomes more certain of which regions in parameter space are worth exploring and which are not, as seen in the picture below.



THEORY: Hyperparameter Optimization

Gradient-based optimization

- For specific learning algorithms, it is possible to compute the gradient with respect to hyperparameters and then optimize the hyperparameters using gradient descent. The first usage of these techniques was focused on neural networks. Since then, these methods have been extended to other models such as support vector machines[11] or logistic regression.
- A different approach in order to obtain a gradient with respect to hyperparameters consists in differentiating the steps of an iterative optimization algorithm using automatic differentiation.



$$\arg\min_{\lambda \in \mathcal{D}} \underbrace{f(\lambda) \triangleq g(X(\lambda), \lambda)}_{\text{loss on test set}}$$

$$\text{s.t.} \quad \underbrace{X(\lambda)}_{\text{model parameters}} \in \arg\min_{x \in \mathbb{R}^p} \underbrace{h(x, \lambda)}_{\text{loss on train set}}$$

THEORY: Hyperparameter Optimization

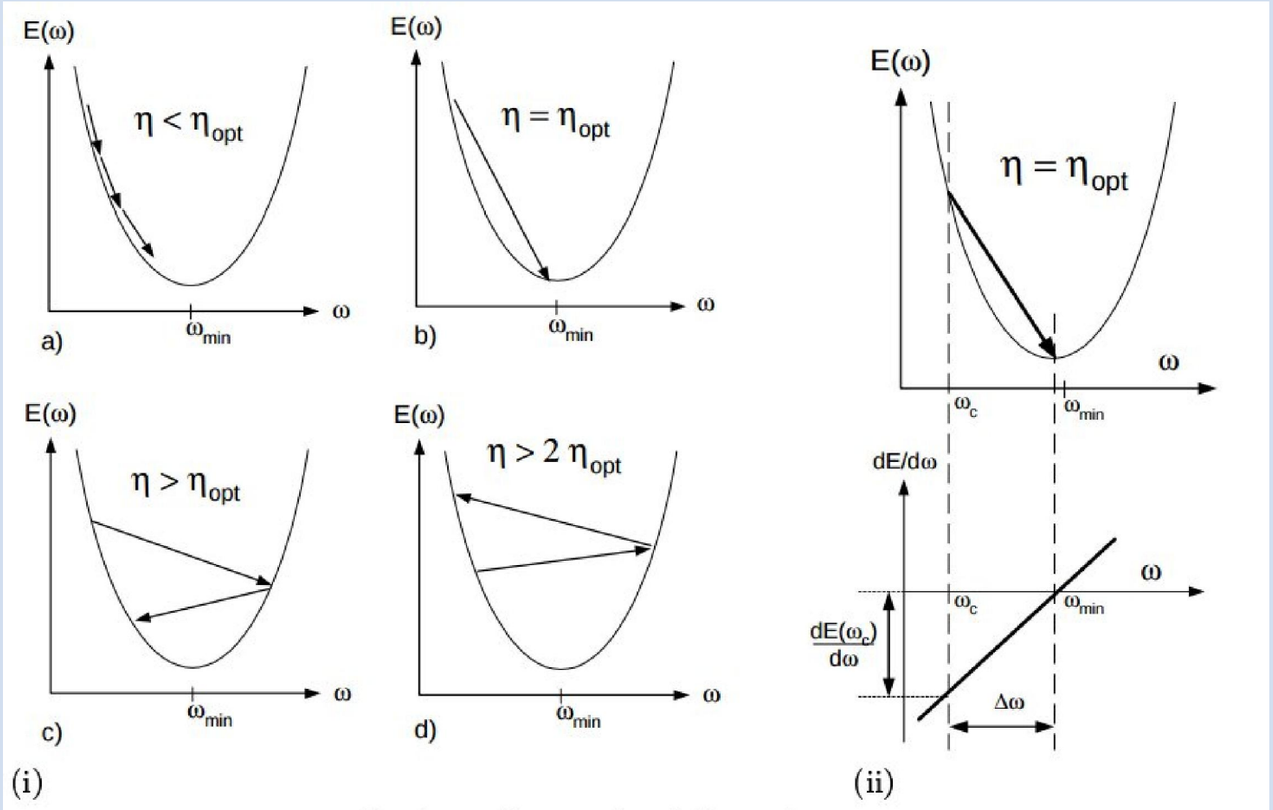


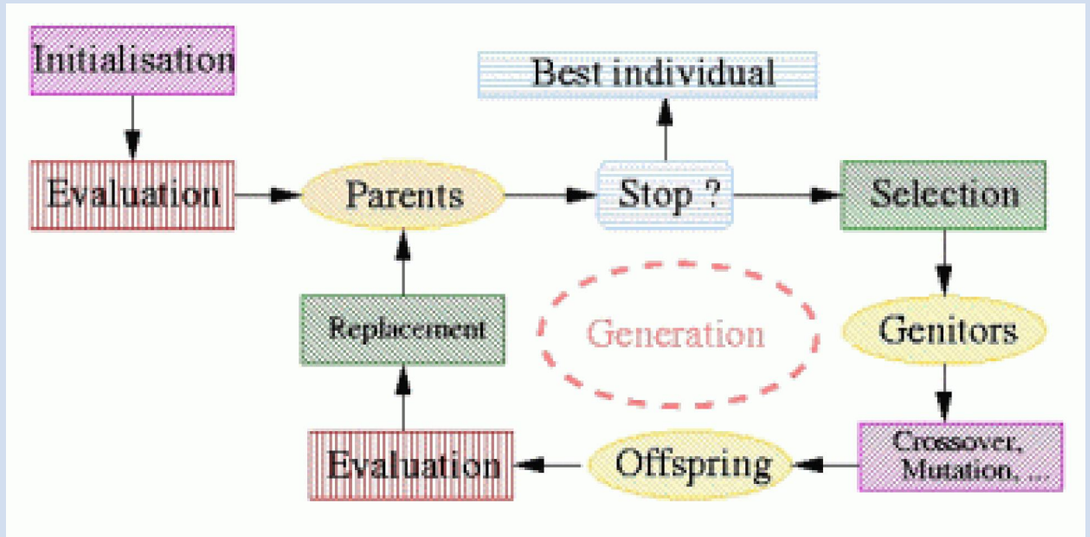
Fig. 6. Gradient descent for different learning rates.

Tuning hyperparameters of learning algorithms is hard because gradients are usually unavailable. We compute exact gradients of cross-validation performance with respect to all hyperparameters by chaining derivatives backwards through the entire training procedure. These gradients allow us to optimize thousands of hyperparameters, including step-size and momentum schedules, weight initialization distributions, richly parameterized regularization schemes, and neural network architectures. We compute hyperparameter gradients by exactly reversing the dynamics of stochastic gradient descent with momentum.

THEORY: Hyperparameter Optimization

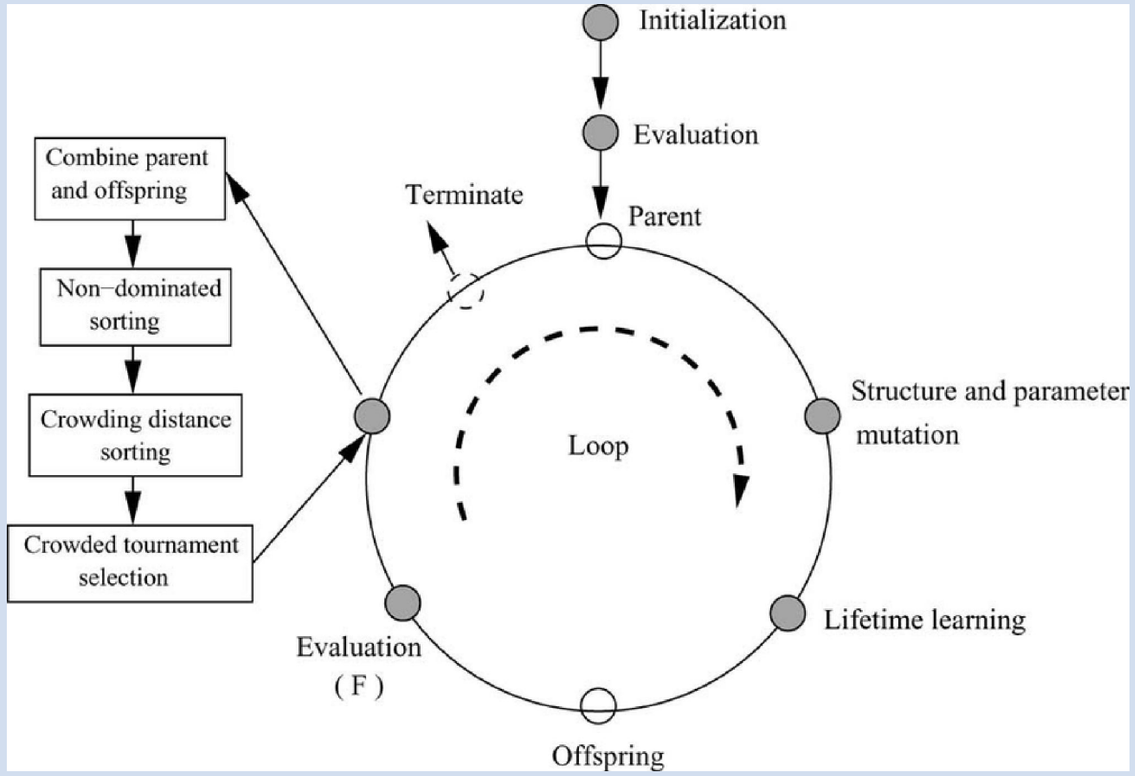
Evolutionary optimization

- Evolutionary optimization is a methodology for the global optimization of noisy black-box functions. In hyperparameter optimization, evolutionary optimization uses evolutionary algorithms to search the space of hyperparameters for a given algorithm.Evolutionary hyperparameter optimization follows a process inspired by the biological concept of evolution
- Evolutionary optimization has been used in hyperparameter optimization for statistical machine learning algorithms, automated machine learning, deep neural network architecture search,[17][18] as well as training of the weights in deep neural networks.



THEORY: Hyperparameter Optimization

1. Create an initial population of random solutions (i.e., randomly generate tuples of hyperparameters, typically 100+)
2. Evaluate the hyperparameters tuples and acquire their fitness function (e.g., 10-fold cross-validation accuracy of the machine learning algorithm with those hyperparameters)
3. Rank the hyperparameter tuples by their relative fitness
4. Replace the worst-performing hyperparameter tuples with new hyperparameter tuples generated through crossover and mutation
5. Repeat steps 2-4 until satisfactory algorithm performance is reached or algorithm performance is no longer improving



Practice Sections

The Boston housing data

As a reminder, we are using three features from the Boston housing dataset: `'RM'`, `'LSTAT'`, and `'PTRATIO'`. For each data point (neighborhood):

- `'RM'` is the average number of rooms among homes in the neighborhood.
- `'LSTAT'` is the percentage of homeowners in the neighborhood considered "lower class" (working poor).
- `'PTRATIO'` is the ratio of students to teachers in primary and secondary schools in the neighborhood.

	RM	LSTAT	PTRATIO	MEDV
2	6.575	4.98	15.3	504000.0
3	6.421	9.14	17.8	453600.0
4	7.185	4.03	17.8	728700.0
5	6.998	2.94	18.7	701400.0
6	7.147	5.33	18.7	760200.0
7	6.43	5.21	18.7	602700.0
8	6.012	12.43	15.2	480900.0
9	6.172	19.15	15.2	569100.0

Practice Sections

```
import numpy as np
import pandas as pd
from sklearn.cluster import KMeans

OrigData = pd.read_csv('../InputData/housing.csv')
data = OrigData
features = data.drop('MEDV', axis = 1)
prices = data['MEDV']


kmeans = KMeans(n_clusters=3)
model = kmeans.fit(features)
classes = model.predict(features)
OrigData['class'] = classes
print(OrigData)
```

	RM	LSTAT	PTRATIO	MEDV	class
0	6.575	4.98	15.3	504000.0	1
1	6.421	9.14	17.8	453600.0	1
2	7.185	4.03	17.8	728700.0	1
3	6.998	2.94	18.7	701400.0	1
4	7.147	5.33	18.7	760200.0	1
5	6.430	5.21	18.7	602700.0	1
6	6.012	12.43	15.2	480900.0	2
7	6.172	19.15	15.2	569100.0	2
8	5.631	29.93	15.2	346500.0	0
9	6.004	17.10	15.2	396900.0	2
10	6.377	20.45	15.2	315000.0	2
11	6.009	13.27	15.2	396900.0	2
12	5.889	15.71	15.2	455700.0	2
13	5.949	8.26	21.0	428400.0	1
14	6.096	10.26	21.0	382200.0	1
15	5.834	8.47	21.0	417900.0	1
16	5.935	6.58	21.0	485100.0	1
17	5.990	14.67	21.0	367500.0	2
18	5.456	11.69	21.0	424200.0	2

Practice Sections

```
from sklearn.neighbors import KNeighborsClassifier
model = KNeighborsClassifier(n_neighbors=1)

from sklearn.model_selection  import train_test_split
# split the data with 50% in each set
X1, X2, y1, y2 = train_test_split(features, classes, random_state=0,
                                  train_size=0.5)

from sklearn.metrics import accuracy_score
# fit the model on one set of data
model.fit(X1, y1)
# evaluate the model on the second set of data
y2_model = model.predict(X2)
accuracy01 = accuracy_score(y2, y2_model)

print("Accuracy 1:" + str(accuracy01))
```

```
Accuracy 1:0.987755102040816
```

Practice Sections

```
y2_model = model.fit(X1, y1).predict(X2)
y1_model = model.fit(X2, y2).predict(X1)
accuracy02 = accuracy_score(y1, y1_model)
accuracy03 = accuracy_score(y2, y2_model)
print("Accuracy 2:" + str(accuracy02))
print("Accuracy 3:" + str(accuracy03))


from sklearn.model_selection import cross_val_score
accuracy04 = cross_val_score(model, features, classes, cv=5)

print("Accuracy 4:" + str(accuracy04))


from sklearn.model_selection import LeaveOneOut
scores = cross_val_score(model, features, classes, cv=LeaveOneOut())
print(scores)

print("Score Mean:" + str(scores.mean()))
```

```
Accuracy 2:0.9877049180327869
Accuracy 3:0.9918367346938776
Accuracy 4:[0.98989899 0.94897959 0.97959184 0.95876289 1.
```

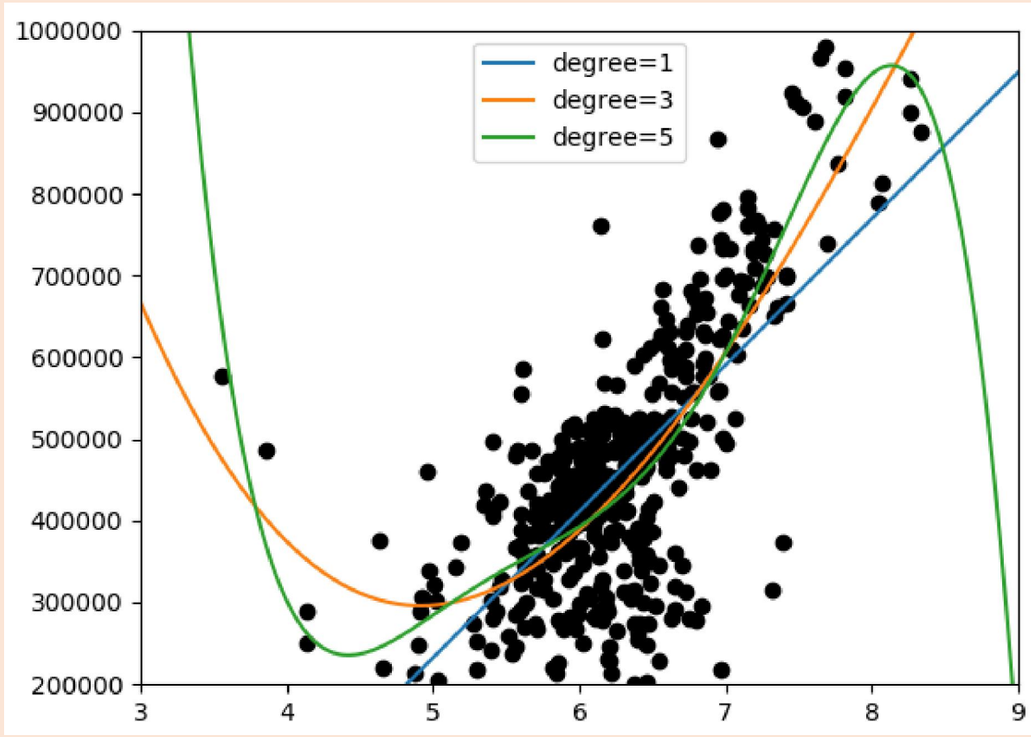
```
[1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1.
 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1.
 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1.
 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1.
 0. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1.
 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 0. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1.
 1. 1. 1. 1. 1. 0. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 0. 1. 1. 1. 1. 1. 1.
 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1.
 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1.
 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1.
 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1.
 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1.
 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1.
 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1.
 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1.
 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 0. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1.
 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1.
 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1.
 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1.
 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1.
 1. 1. 1. 1. 1. 1. 1. 1. 1.]
```

```
Score Mean:0.9897750511247444
```

Practice Sections

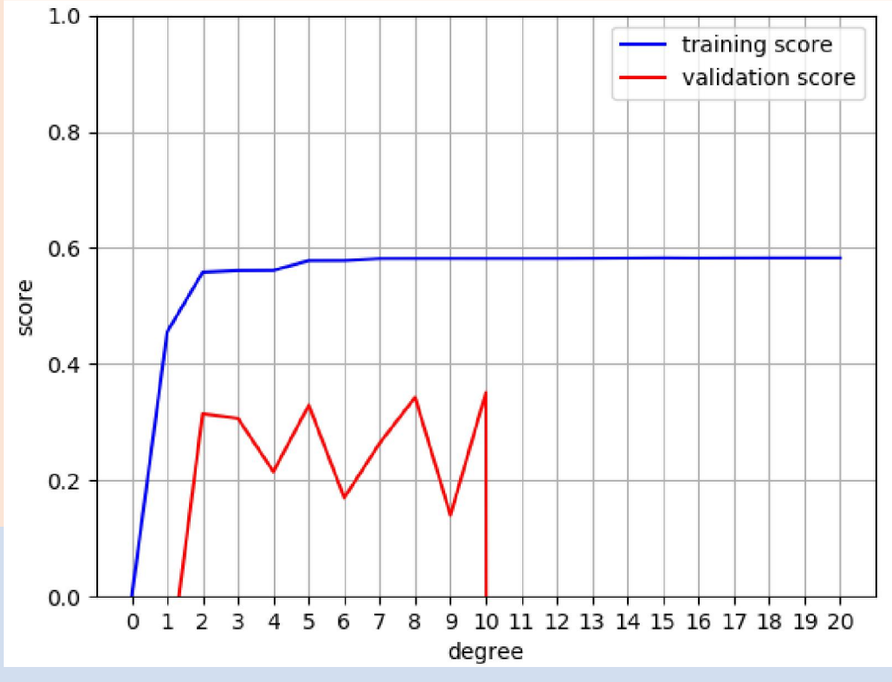
```
import numpy as np
X = features['RM'].values
y = prices.values
import matplotlib.pyplot as plt
import numpy as np

plt.figure()
plt.scatter(X.ravel(), y, color='black')
axis = plt.axis()
for degree in [1, 3, 5]:
    coeffs = np.polyfit(X,y,degree)
    xPoly = np.arange(min(X)-1, max(X)+1, .01)
    yPoly = np.polyval(coeffs, xPoly)
    plt.plot(xPoly, yPoly, label='degree={0}'.format(degree))
plt.xlim(3, 9)
plt.ylim(200000, 1000000)
plt.legend()
plt.show()
```



Practice Sections

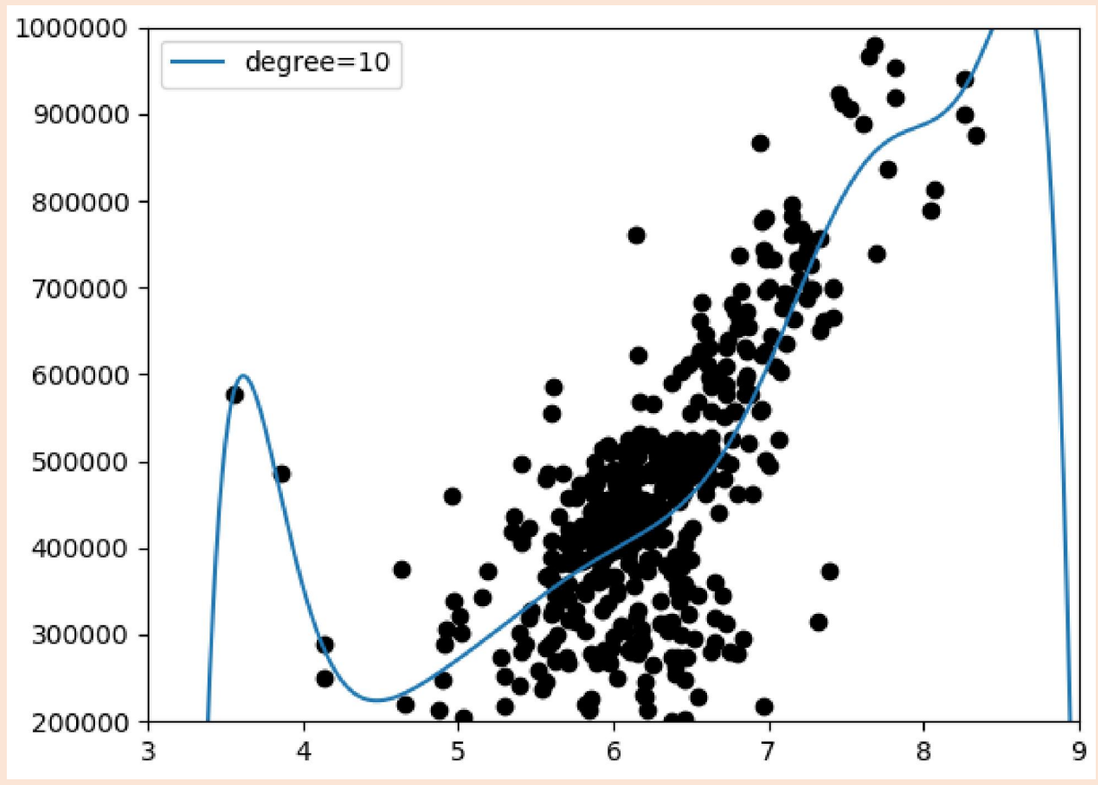
```
from sklearn.preprocessing import PolynomialFeatures
from sklearn.linear_model import LinearRegression
from sklearn.pipeline import make_pipeline
from sklearn.model_selection import validation_curve
degree = np.arange(0, 21)
model = make_pipeline(PolynomialFeatures(), LinearRegression())
train_score, val_score = validation_curve(model, X[:, np.newaxis], y,
                                          param_name='polynomialfeatures__degree',param_range=degree)
plt.plot(degree, np.median(train_score, 1), color='blue', label='training score')
plt.plot(degree, np.median(val_score, 1), color='red', label='validation score')
plt.legend(loc='best')
plt.ylim(0, 1)
plt.xticks(range(21))
plt.xlabel('degree')
plt.ylabel('score')
plt.grid()
plt.show()
```



Practice Sections

From the validation curve, we can read-off that the optimal trade-off between bias and variance is found for a
tenth-order polynomial; we can compute and display this fit over the original data as follows:

```
plt.figure()
plt.scatter(X.ravel(), y, color='black')
axis = plt.axis()
degree = 10
coeffs = np.polyfit(X,y,degree)
#use more points for a smoother plot
xPoly = np.arange(min(X)-1, max(X)+1, .01)
#Evaluates the polynomial for each xPoly value
yPoly = np.polyval(coeffs, xPoly)
plt.plot(xPoly, yPoly, label='degree={0}'.format(degree))
plt.xlim(3, 9)
plt.ylim(200000, 1000000)
plt.legend()
plt.show()
```



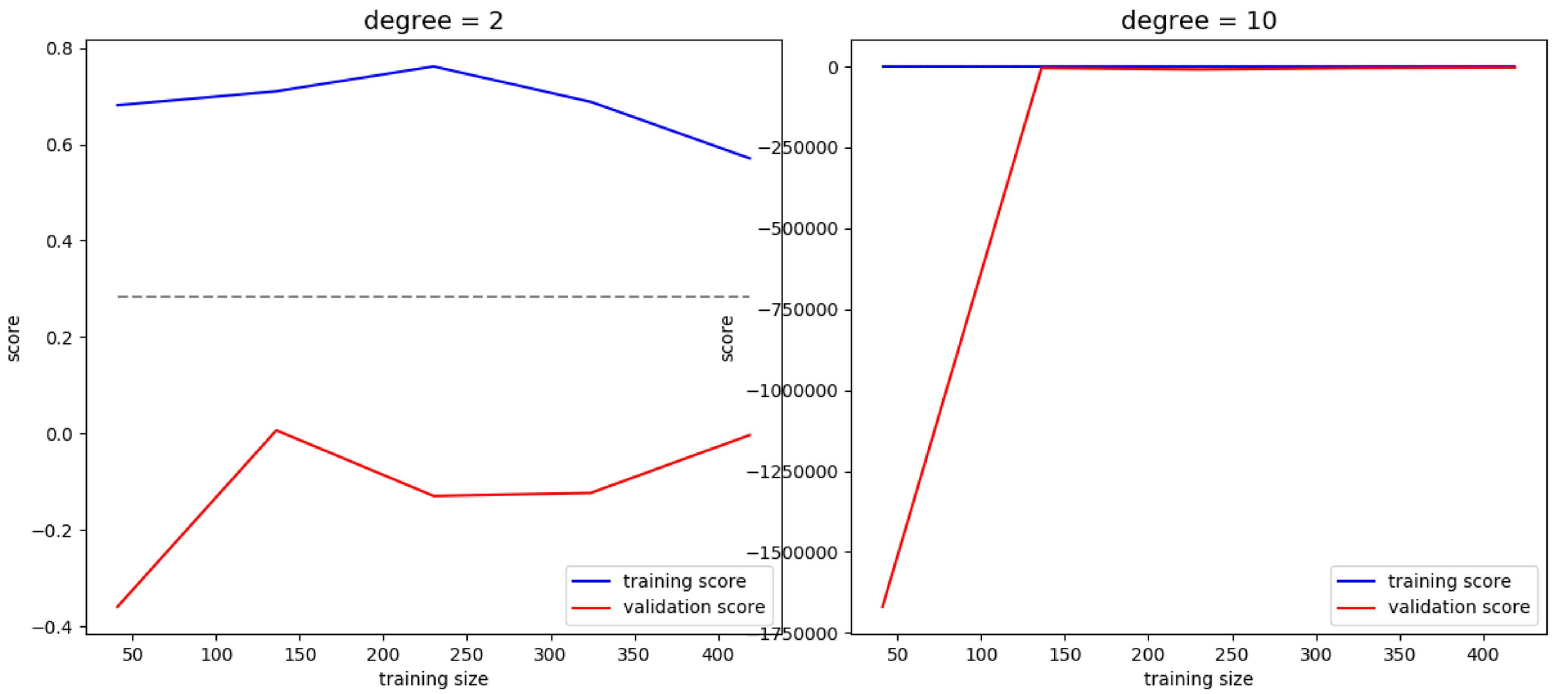
Practice Sections

```
from sklearn.model_selection import learning_curve
fig, ax = plt.subplots(1, 2, figsize=(16, 6))
fig.subplots_adjust(left=0.0625, right=0.95, wspace=0.1)
for i, degree in enumerate([2, 10]):
    model = make_pipeline(PolynomialFeatures(degree), LinearRegression())
    N, train_lc, val_lc = learning_curve(model, X[:, np.newaxis], y, cv=7)

    ax[i].plot(N, np.mean(train_lc, 1), color='blue', label='training score')
    ax[i].plot(N, np.mean(val_lc, 1), color='red', label='validation score')
    ax[i].hlines(np.mean([train_lc[-1], val_lc[-1]]), N[0], N[-1],
                 color='gray', linestyle='dashed')

    # ax[i].set_ylim(0, 1)
    # ax[i].set_xlim(N[0], N[-1])
    ax[i].set_xlabel('training size')
    ax[i].set_ylabel('score')
    ax[i].set_title('degree = {0}'.format(degree), size=14)
    ax[i].legend(loc='best')
plt.show()
```

Practice Sections



Practice Sections

```
# TODO: Import 'make_scorer', 'DecisionTreeRegressor', and 'GridSearchCV'
from sklearn.metrics import make_scorer
from sklearn.tree import DecisionTreeRegressor
from sklearn.model_selection import GridSearchCV
from sklearn.model_selection import ShuffleSplit

# TODO: Import 'r2_score'
from sklearn.metrics import r2_score

def performance_metric(y_true, y_predict):
    """ Calculates and returns the performance score between
        true and predicted values based on the metric chosen. """

    # TODO: Calculate the performance score between 'y_true' and 'y_predict'
    score = r2_score(y_true, y_predict)

    # Return the score
    return score
```

Practice Sections

```
def fit_model(X, y):
    """ Performs grid search over the 'max_depth' parameter for a decision tree regressor trained on the input data [X, y]. """
    # Create cross-validation sets from the training data. ShuffleSplit works iteratively compared to KFOLD. It saves computation
    # time when your dataset grows. X.shape[0] is the total number of elements. n_iter is the number of re-shuffling & splitting
    # iterations.
    cv_sets = ShuffleSplit(X.shape[0], test_size = 0.20, random_state = 0)

    # TODO: Create a decision tree regressor object: Instantiate
    regressor = DecisionTreeRegressor(random_state=0)
# TODO: Create a dictionary for the parameter 'max_depth' with a range from 1 to 10
dt_range = range(1, 11)
params = dict(max_depth=dt_range)
# TODO: Transform 'performance_metric' into a scoring function using 'make_scorer'
# We initially created performance_metric using R2_score
scoring_fnc = make_scorer(performance_metric)
# TODO: Create the grid search object
# You would realize we manually created each, including scoring_func using R^2
grid = GridSearchCV(regressor, params, cv=cv_sets, scoring=scoring_fnc)
# Fit the grid search object to the data to compute the optimal model
grid = grid.fit(X, y)
return grid.best_estimator_
```

Practice Sections

```
# Import RandomizedSearchCV
from sklearn.model_selection import RandomizedSearchCV

# Create new similar function
def fit_model_2(X, y):
    cv_sets = ShuffleSplit(X.shape[0], test_size = 0.20, random_state = 0)
    # Instantiate
    regressor = DecisionTreeRegressor(random_state=0)
    # TODO: Create a dictionary for the parameter 'max_depth' with a range from 1 to 10
dt_range = range(1, 11)
params = dict(max_depth=dt_range)
# TODO: Transform 'performance_metric' into a scoring function using 'make_scorer'
# We initially created performance_metric using R2_score
scoring_fnc = make_scorer(performance_metric)
# TODO: Create the grid search object
# You would realize we manually created each, including scoring_func using R^2
rand = RandomizedSearchCV(regressor, params, cv=cv_sets, scoring=scoring_fnc)
# Fit the grid search object to the data to compute the optimal model
rand = rand.fit(X, y)
# Return the optimal model after fitting the data
return rand.best_estimator_
```

Practice Sections

```
# Fit the training data to the model using grid search
# TODO: Import 'train_test_split'
from sklearn.model_selection import train_test_split
# TODO: Shuffle and split the data into training and testing subsets
X_train, X_test, y_train, y_test = train_test_split(features, prices, test_size=0.2, random_state=10)
reg = fit_model(X_train, y_train)
# Produce the value for 'max_depth'
print("Parameter 'max_depth' is {} for the optimal model.".format(reg.get_params()['max_depth']))
# Explanation of how we got the 'max_depth' param
# First we fit the model
# Then we use get_params() to get the optimal parameters
# As you can see here, it's a dictionary
reg = fit_model(X_train, y_train)
print(reg.get_params())
# We can access our value from reg.get_params(), a dictionary, using dict['key']
print(reg.get_params()['max_depth'])
# Fit the training data to the model using grid search
reg_2 = fit_model_2(X_train, y_train)
# Produce the value for 'max_depth'
print("Parameter 'max_depth' is {} for the optimal model.".format(reg_2.get_params()['max_depth']))
```

```
Parameter 'max_depth' is 4 for the optimal model.
```

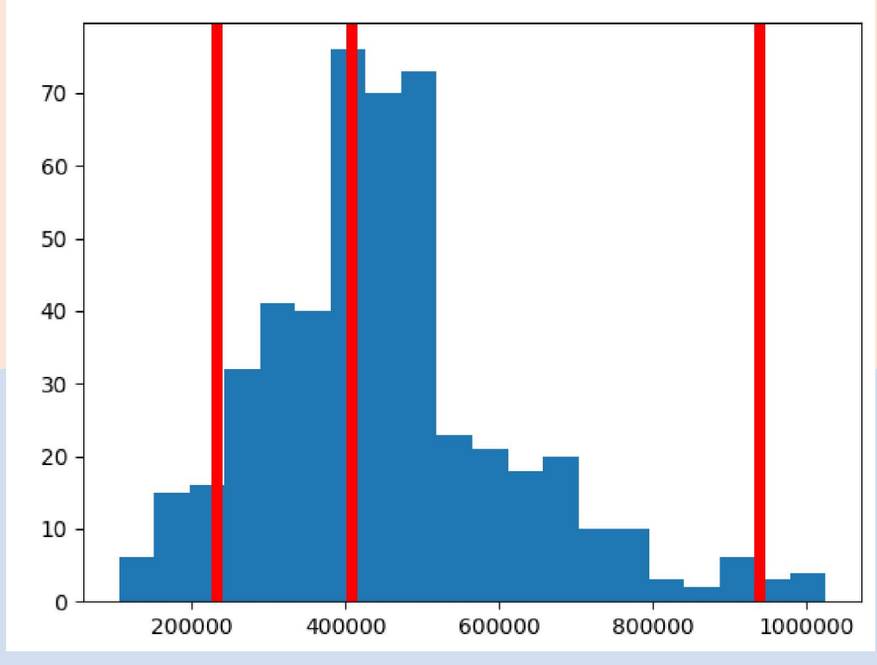
```
{'criterion': 'mse',
'max_depth': 4,
'max_features': None,
'max_leaf_nodes': None,
'min_impurity_decrease': 0.0,
'min_impurity_split': None,
'min_samples_leaf': 1,
'min_samples_split': 2,
'min_weight_fraction_leaf': 0.0,
'presort': False,
'random_state': 0,
'splitter': 'best'}
```

Practice Sections

```
# Produce a matrix for client data
client_data = [[5, 17, 15], # Client 1
               [4, 32, 22], # Client 2
               [8, 3, 12]]  # Client 3
# Show predictions
for i, price in enumerate(reg.predict(client_data)):
    print("Predicted selling price for Client {}'s home: ${:,.2f}".format(i+1, price))
```

```
import matplotlib.pyplot as plt
plt.figure()
plt.hist(prices, bins = 20)
for price in reg.predict(client_data):
    plt.axvline(price, lw = 5, c = 'r')
plt.show()
```

```
Predicted selling price for Client 1's home: $406,933.33
Predicted selling price for Client 2's home: $232,200.00
Predicted selling price for Client 3's home: $938,053.85
```



THANK YOU