### **1.3 접착용 에폭시 고분자 개발**

|  |  |
| --- | --- |
| **학습 내용** | |
| [문제] | 에폭시 최적 생산조건 탐색 |
| [방법] | 접착용 에폭시 생산 모사 함수 구축 |
| [응용] | 베이지안 최적화 기반 최적 생산 조건 도출 |
| [요약] | * 접착용 에폭시 생산 반응 이해 * 생산 모사 함수 구축 * 베이지안 최적화를 사용하여 최적의 생산 조건 도출 |

### **[이론] - 능동학습을 사용한 고분자 생산조건 최적화.**

2010년대에 들어서부터 능동학습 프레임워크를 활용한 기능성 신물질 탐색이 점점 각광을 받고 있다. 기계학습을 통한 능동학습(active learning)의 프레임 워크의 하에서의 의사결정 과정의 장점이 기존의 의사결정 과정들과 비교해서 여러 장점이 있기 때문이다. 인간의 의사결정보다는 의사결정과정이 더욱 정교하기 때문에 탐험-탐사(exploration-exploitation)사이의 균형을 고차원의 의사결정에서도 잘 맞출 수 있고, 지도학습 기반 의사결정 보다는 필요한 초기 데이터의 수가 훨씬 적고 또한, 데이터 생성이 시간이 많이 걸리고 비용이 드는 신물질 개발에서 능동학습은 점점 중요도가 올라가고 있다.

능동학습 기반 의사과정은 다음과 같은 세가지 과정들을 거친다. 첫째로는, 현재 존재하는 소수의 데이터를 바탕으로 모델을 구축하여 input-output 사이의 상관관계를 기계가 학습을 한다 (modeling). 둘째로는 기계가 직접 다음 실험 지점을 추천한다 (proposal). 이때 추천이 되는 점은 모델에서의 이해를 바탕으로 우리가 원하는 물성이 최대화가 될 수 있는 지점이면서, 또한 탐색되지 않은 지점의 정보를 얻어낼 수 있는, 즉 탐험-탐사 간의 균형이 맞은 지점이다. 셋째로는 추천된 실험 후보를 사람이 직접 실험을 하여 얻어진 결과를 확인하는 것 이다. 이때 얻어진 결과가 목적에 부합하는지 여부에 따라서 실험이 종결되거나 지속된다.

탐험-탐사간의 균형이 맞은 의사결정을 내리는 방법론중에서 가장 대표적인 방법은 베이지안 최적화이다. 따라서 베이지안 최적화를 접착용 에폭시 개발 문제에 직접 적용하는 것이 이번 실습의 목적이다. 베이지안 최적화의 이론 관한 자세한 설명은 챕터 8, 기계학습과 최적화에 서술되어있다.

베이지안 최적화를 실제 실험 적용해 보면 가장 좋겠지만, 교과서적인 한계로 인해 고분자를 실제로 합성해 나가며 물질 개발을 할 수 없기에, 접착용 에폭시 고분자의 input – output의 상관관계를 Sum-squares function이라는 기준 함수(benchmark function)으로 정의한다. 이때, 함수를 모른다고 가정한 뒤, 초기의 특정한 실험 데이터 (spot data)만을 시작으로 베이지안 최적화 기반의 의사결정을 해 나가면서 의사결정이 효율적으로 이루어짐을 확인한다. 효율성을 보이기 위해, 베이지안 최적화를 진행한 결과와, 불확실성에 대한 고려 없는 greedy-policy (대체 모델의 평균값이 최대화 되는 지점을 실험. 즉, 지도학습 기반 의사결정) 기반 최적화를 진행한 퍼포먼스를 비교한다.

### **[문제] 새로운 기능성 물질을 효율적으로 개발하기 위한 베이지안 최적화 기반의 의사결정 방법론을 접착용 에폭시 고분자 개발이라는 예시에 적용하여, 블랙박스 최적화에서의 효율적인 의사결정을 가이드 해 보자.**

- “part3\_접작용 에폭시 고분자 개발.py 코드를 활용하여라.

- sklearn, scipy 라이브러리를 설치 하여라

- 챕터 8을 읽고 베이지안 최적화의 이론을 이해하여라.

### **[방법] 접착용 에폭시 생산 모사 함수 구축**

2019년에 science and technology of advanced materials 출간된 Sirawit P.의 논문에서는 접착용 에폭시 고분자를 DGEBA 레진, propylene glycol(경화제)의 두가지 재료를 사용하여 접작용 에폭시 고분자를 생성한다. 이러한 시스템에서 조작변인은 4가지로써, 레진과 글리콜의 비율, 레진의 분자량, 경화제의 분자량, 그리고 반응 온도이다. 논문에서는 이러한 상황 하에서 어떠한 비율, 분자량, 반응 온도를 설정해야 가장 접착력이 높은 에폭시 폴리머를 만들 수 있는지를 베이지안 최적화를 사용하여 찾아낸다(실험 가능한 조합의 수: 256). 이를 간단하게 모사하기 위하여 본 실습에서는 2차원의 input을 받아들이는 Real\_epoxy 함수를 정의하여 베이지안 최적화를 진행한다. 와 같은 Sum-squares 함수를 이번 실습의 에폭시 폴리머의 input-output 상관관계라고 정의한다. 이때 최적해 (global maximum)는 (0,0)에서 0의 꼴로 존재하며, 과의 가능한 실험 후보는 -30 ~ 30까지 0.1의 단위로 존재한다 (실험 가능한 조합의 수: ).

#### 관련 라이브러리를 설치하자.

1. 베이지안 최적화를 진행하기 위해선 numpy, sklearn, scipy등의 라이브러리와 GaussianProcessRegressor, WhiteKernel, RBF 등의 함수가 요구되기에, 이를 불러온다.

|  |
| --- |
| import numpy as np from sklearn.gaussian\_process import GaussianProcessRegressor from sklearn.gaussian\_process.kernels import WhiteKernel, RBF import itertools from scipy.stats import norm |

#### 문제 조건대로 에폭시 접착제의 생산 모사 class를 정의해보자.

1. 문제 상황에서 정의한대로의 코드를 구성해보면 다음과 같이 구성할 수가 있다. 이때 이상적인 상황을 가정하여 실험 결과가 특정 input을 넣었을 때 같은 output이 매번 나온다면 .real의 함수를 사용할 수 있고, 현실적인 상황을 가정하여 실험 결과가 noise에 의해 오염된 상황을 실험해보고 싶다면 .noisy 함수를 사용할 수 있다.

|  |
| --- |
| class Real\_epoxy:  def \_\_init\_\_(self):  pass   def real(self, x):  x1 = x[:,0]  x2 = x[:,1]  return -x1\*\*2 - 2\*x2\*\*2   def noisy(self, x, noise\_var):  x1 = x[0]  x2 = x[1]  return -x1\*\*2 - 2\*x2\*\*2 + np.sqrt(noise\_var) \* np.random.standard\_normal()  test\_func = Real\_epoxy() noise\_var = 0.01 GP\_lengthscale = np.sqrt(0.1) search\_space\_org = np.array(list(itertools.product(np.arange(-30,30,0.1), np.arange(-30,30,0.1)))) y\_max\_real = 0 action\_min = np.array([-30, -30]) action\_max = np.array([30, 30]) |

### **[응용] 베이지안 최적화 기반 최적 생산조건 도출**

모사된 에폭시 input-output 함수를 바탕으로 초기 데이터 확보, 데이터 전처리, 획득함수 정의, 그리고 베이지안 최적화를 진행해보자

#### 의사 결정을 위한 초기 데이터셋을 구축해보자.

1. 초기 데이터셋은 전체 가능한 실험 조합인 360,000개의 점들 중에서 20개를 알고 있다고 가정을 한다. 이때, 어떠한 초기 데이터를 안다고 가정 하는지에 따라 베이지안 최적화의 성능이 많이 다를 수 있다. 따라서 베이지안 최적화의 평균적인 성능을 보기 위하여 랜덤하게 20 실험 포인트의 결과를 알고있는 상태(num\_initial\_data)에서 20번 최적화를 진행을 한 뒤(num\_experiments), 이러한 과정을 초기 데이터를 바꾸어가며 10번 진행을 하게끔 세팅을 한다(num\_runs). 이때 매 의사결정마다 최적해로부터 얼마나 떨어진 지점을 탐색했는지 성과를 수치화 하기 위해 regret\_store라는 빈 list를 만들어 둔다.

|  |
| --- |
| np.random.seed(1) num\_runs = 10 num\_experiments = 20 num\_initial\_data = 20 regret\_store = [] initial\_indices\_store = [] for i in range(num\_runs):  random\_indices = np.random.choice(search\_space\_org.shape[0], size=num\_initial\_data, replace=False)  initial\_indices\_store.append(random\_indices) |

#### 데이터 전처리를 위한 class를 설정해보자.

1. 기계학습을 위하여 input-output 로우 데이터(raw data)를 사용하는 것 보다 데이터를 표준화하여 스케일 된 데이터를 사용하면 기계학습의 성능이 더 나아진다는 것은 널리 알려진 사실이다. 따라서 에폭시 모사 시스템의 대체모델과 획득함수를 정의 하기에 앞서 로우 데이터를 표준화된 데이터로 스케일링 할 수 있는 함수와 표준화된 형태로 나타내어진 데이터를 우리에게 친숙한 로우 데이터의 형태로 다시 바꾸는 함수를 정의 하여야 한다.

|  |
| --- |
| class Scaler:  def \_\_init\_\_(self):  pass   def fit(self, x):  self.mu = np.mean(x,0)  self.std = np.std(x,0)   def transform(self, x):  return (x-self.mu)/self.std   def inverse\_transform\_mean(self, x):  return x\*self.std + self.mu   def inverse\_transform\_std(self, x):  return x\*self.std |

#### Expected improvement 기반 획득함수 class를 설정해보자.

1. 챕터 8에서 소개된 expected improvement의 식은 와 같고, 이를 예측 평균치(와 표준편차(를 활용하여 표현하면 다음과 같이 나타낼 수 있다: . 이때 는 기존에 관찰된 데이터 중에서 가장 성능이 좋았던 값이고, 는 표준 정규 누적 분포 함수이고, 는 표준 정규 분포 함수이다. 이를 코드로 구현하면 다음과 같다.

|  |
| --- |
| class Base\_EI:  def \_\_init\_\_(self):  self.scaler\_EI = Scaler()   def evaluate(self, x, model, y\_max, y\_train):  self.scaler\_EI.fit(y\_train)   mean, std = model.predict(x, return\_std=True)  mean = self.scaler\_EI.inverse\_transform\_mean(mean)  std = self.scaler\_EI.inverse\_transform\_std(std)   a = (mean - y\_max).reshape(-1,)  z = a / std  ei = a \* norm.cdf(z) + std \* norm.pdf(z)   index\_next = np.argmax(ei)  X\_next = x[index\_next]  return X\_next, index\_next |

#### 초기 데이터를 사용한 베이지안 최적화를 통해 의사결정을 진행해보고 의사결정들의 평균 퍼포먼스를 print해보자.

1. 사용하는 획득함수를 Q5에서 정의한 expected improvement로 정의한다. 이후, 실험이 진행됨에 따라 초기 데이터에 가우시안 프로세스 모델을 사용하여 대체모델을 구축한다. 이를 바탕으로 획득함수가 최대화 되는 점을 찾고, real\_epoxy 함수에 넣어서 결과값을 얻고, 얻어진 결과값을 최적해와 비교한 성과지표(regret)를 기록하는것을 하나의 에피소드가 끝날 때까지 20번 반복을 하고(num\_experiments), 초기 데이터를 바꾸어 가며 총 10번의 에피소드(num\_run)을 반복하여 베이지안 최적화를 진행한다.

|  |
| --- |
| acq\_func = Base\_EI() kernel = 1.0 \* RBF(length\_scale=GP\_lengthscale) + WhiteKernel(noise\_level=noise\_var) gpr = GaussianProcessRegressor(kernel=kernel, alpha=0.0, optimizer=None, random\_state=0) scaler = Scaler() for i in range(num\_runs):  print(**"run: "** + str(i))  regrets = []  random\_indices = initial\_indices\_store[i]  print(random\_indices)  X\_train = search\_space\_org[random\_indices,:]  y\_train = test\_func.real(X\_train).reshape(-1,1)  search\_space = np.delete(search\_space\_org, random\_indices, axis=0)   y\_max = np.max(y\_train)  regret = y\_max\_real - y\_max  regrets.append(regret)  for j in range(num\_experiments):  print(**"experiment: "** + str(j))  scaler.fit(y\_train)  y\_train\_scaled = scaler.transform(y\_train)  gpr.fit(X\_train, y\_train\_scaled)   X\_next, index\_next = acq\_func.evaluate(search\_space, gpr, y\_max, y\_train)  print(X\_next)  y\_next = test\_func.noisy(X\_next, 0)  search\_space = np.delete(search\_space, index\_next, axis=0)   X\_train = np.vstack((X\_train, X\_next))  y\_train = np.vstack((y\_train, y\_next))   y\_max = np.max(y\_train)  regret = y\_max\_real - y\_max  print(regret)  regrets.append(regret)   regret\_store.append(regrets)  regret\_store = np.array(regret\_store) avg\_regret = np.mean(regret\_store, axis=0) print(avg\_regret) |

이때 print(avg\_regret)의 결과로는 다음과 같은 list가 프린트 된다.

[101.293 99.311 98.47 98.47 97.717 97.717 96.724 96.125 95.136 92.865 92.254 90.46 88.852 87.793 86.667 85.133 84.711 82.068 81.889 80.708 79.627]

즉, 평균적으로 매 time-step마다 베이지안 최적화 기반의 의사결정이 최적해에 근접한 값들을 제시해 나갔다는 것을 알 수 있다.

#### 초기 데이터를 사용한 greedy-policy기반 최적화를 통해 의사결정을 진행해보고 의사결정의 효율성을 베이지안 기반 최적화와 비교해보자.

1. Q6의 성능 지표가 다른 의사결정 방법론보다 좋은지 확인할 대조군 제작 위해, 대체 모델의 불확실성에 대한 고려 없이 예측값의 최댓값만을 실험해 나가는 greedy policy 기반의 의사결정을 진행한 뒤(supervised learning기반 의사결정), 그 결과(avg\_regret)를 베이지안 최적화의 결과와 비교하여 베이지안 최적화의 의사결정 효율성을 인식한다.

|  |
| --- |
| class Base\_EI:  def \_\_init\_\_(self):  self.scaler\_EI = Scaler()   def evaluate(self, x, model, y\_max, y\_train):  self.scaler\_EI.fit(y\_train)   mean, std = model.predict(x, return\_std=True)  mean = self.scaler\_EI.inverse\_transform\_mean(mean)  std = self.scaler\_EI.inverse\_transform\_std(std)   a = (mean - y\_max).reshape(-1,)  z = a / std  ei = a \* norm.cdf(z) + std \* norm.pdf(z)   index\_next = np.argmax(mean)  X\_next = x[index\_next]  return X\_next, index\_next |

코드상으론 기존의 베이지안 최적화 기반 코드와 대부분이 똑같지만, 획득함수를 정의함에 있어서 index\_next = np.argmax(ei)가 아닌, index\_next = np.argmax(mean)로 바꾸어주면 예측값의 최대치를 추천하는 greedy policy 기반의 의사결정 실험이 된다. 이때 결과를 print해보면 다음과 같은 list가 출력된다.

[101.293 100.358 99.959 99.959 98.678 98.678 98.678 97.699 97.348 96.92 96.236 95.887 95.27 94.678 94.312 93.779 93.037 92.675 92.164 91.426 91.083]

Q6의 결과와 비교하여 불확실성을 고려한 베이지안 최적화 기반 의사결정의 데이터 효율성을 입증할 수 있다.

### **[결론]**

본 장에서는 베이지안 최적화 기반 의사결정의 효율성을 모사된 접착용 에폭시 폴리머 생산 공정에 적용하여 능동학습을 실제로 진행해보았고, 이러한 의사결정을 탐욕정책(greedy policy)기반 의사결정과정과 성능을 비교하여 불확실성을 고려한 의사결정과정의 효율성을 몸소 체험해보는 공부를 하였다.

### **학습 결과**

* 학습 내용

접착용 에폭시 폴리머 생산 공정을 모사해보기.

* 학습 결과 확인하기

베이지안 최적화 기반 의사결정을 접착용 에폭시 폴리머 공정에 적용해보기.

* 학습 결과 응용하기

베이지안 최적화 기반 의사결정과 탐욕정책 기반 의사결정의 데이터 효율성 비교해보기