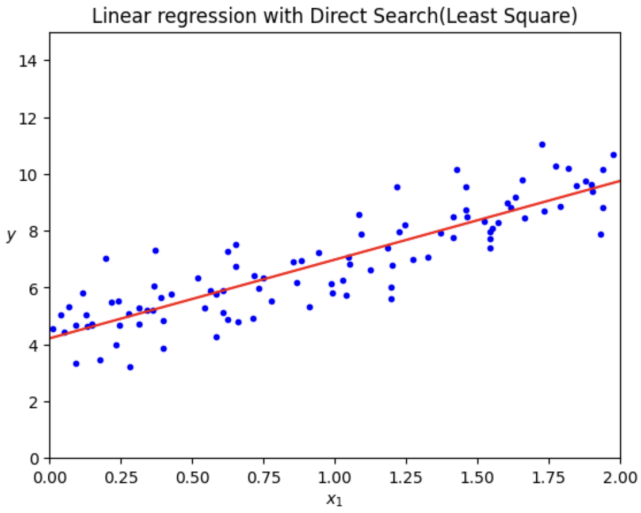
경사하강법 보고서

> 19011810 이용빈

1. Linear Regression (선형 회귀)

인위적으로 만든 랜덤 노이즈 데이터를 사용하여 선형회귀로 데이터의 추세를 분석하는 코드입니다.

scikit-learn에서 이미 구현된 Linear Regression클래스와, python 코드로 직접 구현한 선형회귀 모델을 사용하여 차이를 비교합니다.



scikit-learn의 `Linear Regression`

```
from sklearn.linear_model import LinearRegression

model = LinearRegression() # 모델
model.fit(X_train, y_train) # 학습
y_predict = model.predict(X_test) # 예측
```

scikit-learn은 선형 회귀와 같은 머신러닝 알고리즘을 쉽게 사용할 수 있는 파이썬 라이브러리입니다. scikit-learn은 선형 회귀를 위한 LinearRegression 클래스를 제공하며, 이 클래스에 데이터를 입력하여 모델을 학습시킬 수 있습니다. scikit-learn의 LinearRegression 클래스는 최소 제곱법(OLS)을 사용하여 선형 회귀 모델을 구현합니다. 내부적으로 최적화 알고리즘을 사용하여 회귀 계수를 추정합니다. 또한 아래의 하이퍼 파라미터를 통해 모델의 기능을 변경 가능합니다.

scikit learn LinearRegression의 파라미터는 아래와 같습니다.

`{'copy_X': True, 'fit_intercept': True, 'n_jobs': None, 'positive': False}`

추가로 선형회귀의 정규화 알고리즘을 다르게 구현한 모델도 있습니다.

- Ridge 회귀는 일반 최소 제곱법(Ordinary Least Squares)의 문제를 해결하기 위해 l2 정규화를 통해 계수의 크기에 패널티를 부여합니다.
- Lasso는 l1 정규화를 사용하여 희소한(coefficient가 0인) 계수를 추정하는 선형 모델입니다.
- ElasticNet은 l1 및 l2 -norm 정규화를 모두 사용하여 계수를 추정하는 선형 회귀 모델입니다.

Gradiant Desent

모델의 오차를 수정하기 위해서는 가장 작은 오차를 찾아내어 weight와 bias를 추정하는 경사하강법이 필수적입니다. 모델을 최적화 하기 위해서 사용하는 알고리즘은 매우 많지만, 가장 기초적인 알고리즘으로는 경사하강법, 확률적 경사 하강법, mini batch 경사하강법 등이 있습니다. 각 경사하강법의 특징과 optimizer로서의 차이, 성능에 대해 비교하고 sklearn에서 구현된 선형 회귀 모델과는 어떤 차이가 있는지 알아볼 예정입니다.

GD (Batch Gradiant Descent)

```
beta_bgd_path = []
eta = 0.1 # 학습률
n_epochs = 1000 # epoch 수
n = 100 # 샘플수

beta_bgd = np.random.randn(2,1)  # 무작위로 beta 초깃값 설정

####### Empty Module.2 #######
# Batch Gradient Descent를 통해서 최적의 Beta를 찾아보세요
for iteration in range(n_epochs):
    # 배치 경사 하강법에 해당하는 gradient를 계산하세요
    gradients = 2 / n * Xb.T.dot(Xb.dot(beta_bgd) - y) # 배치 경사 하강법에 해당하는 gradient를 계산하세요
    beta_bgd = beta_bgd - eta * gradients  # update rule에 따라서 beta_bgd를 update 하세요
    beta_bgd_path.append(beta_bgd) # beta_bgd_path에 beta_bgd 추가

y_predict_bgd = X_new * beta_bgd[1] + beta_bgd[0]  # 얻은 회귀 계수를 가지고 X_new에 대해서 y값을 예측하세요
##############################
```

가장 기본적인 경사하강법인 GD는 모든 데이터를 다 고려하여 weight와 bias를 찾는다는 방법론입니다. 무작위로 선정한 초기 weight와 bias에서 학습률을 곱하여 나온 예측값을 기존의 학습 데이터와 비교하여 오차를 추정합니다. 모든 데이터의 정보를 epochs만큼 학습을 진행하여 오차를 최소화 하는 방법론입니다.

특징으로는 모든 데이터를 사용하기 때문에 정확한 성능을 기대할 수 있으나, 그만큼 학습에 시간을 많이 소요하고, 타 optimizer에 비해서 성능이 좋지 않습니다. 또한 local minima등에도 빠지기 쉽습니다.

SGD (Stochastic Gradient Descent)

```
beta_sgd_path = []
n_epochs = 50
t0, t1 = 5, 50  # 학습 스케줄 하이퍼파라미터
n = 100         # 샘플 수

def learning_schedule(t): # 학습 스케줄 함수
    return t0 / (t + t1) # 시간이 갈수록 learning rate가 줄어들도록 설정
beta_sgd = np.random.randn(2,1)  # beta 무작위 초기화

####### Empty Module.3 #########
# Stochastic Gradient Descent를 통해서 최적의 Beta를 찾아보세요
for epoch in range(n_epochs):
    for i in range(n):
        eta = learning_schedule(epoch * n + i) # 학습 스케줄에 따라서 learning rate 줄어들도록 설정

        random_idx = np.random.randint(n) # 0 ~ n-1 까지 랜덤하게 인덱스 선택
        tx = random_idx # random_idx를 활용해 샘플 하나 선택
        ty = random_idx # random_idx를 활용해 샘플 하나 선택
        gradients = 2 * Xb[tx:tx+1].T.dot(Xb[tx:tx+1].dot(beta_sgd) - y[ty:ty+1]) # 확률적 경사 하강법에 해당하는 gradient를 계산하세요
        beta_sgd = beta_sgd - eta * gradients # update rule에 따라서 beta_sgd를 update 하세요

        beta_sgd_path.append(beta_sgd)

y_predict_sgd = X_new * beta_sgd[1] + beta_sgd[0] # 얻은 회귀 계수를 가지고 X_new에 대해서 y값을 예측하세요
##############################
```

확률적 경사하강법은 경사하강법을 보안한 버전의 방법론입니다. 특징은 모든 데이터를 학습에 이용하지 않고, 랜덤으로 샘플을 추출하여 학습에 사용한다는 점 입니다. 후의 오차를 추정하여 최소오차를 찾는 지점은 경사 하강법과 동일힙니다. 경사 하강법에 비해 성능이 높습니다. 확률적으로 추출하기에 local minima도 어느정도는 예방 가능합니다.

MSGD (Mini batch Stochastic Gradiant Descent)

```
beta_mgd_path = []

n_epochs = 50 # epoch 수
minibatch_size = 20 # batch size

np.random.seed(42) # 일정한 결과를 위해 랜덤 시드 고정
beta_mgd = np.random.randn(2,1)  # 랜덤 초기화

t0, t1 = 200, 1000 # 학습 스케줄 하이퍼파라미터
def learning_schedule(t):
    return t0 / (t + t1)

t = 0
```

```
####### Empty Module.4 #########
# Mini Batch Gradient Descent를 통해서 최적의 Beta를 찾아보세요.
for epoch in range(n_epochs):
    shuffled_indices = np.random.permutation(n) # 0 ~ n-1 까지 랜덤하게 인덱스 선택
    Xb_shuffled = Xb[shuffled_indices] # 랜덤하게 섞인 인덱스를 활용해 Xb 인덱싱
    y_shuffled = y[shuffled_indices] # 랜덤하게 섞인 인덱스를 활용해 y 인덱싱
    for i in range(0, n, minibatch_size):
        t += 1
        eta = learning_schedule(t) # 학습 스케줄에 따라서 learning rate 줄어들도록 설정

        tx = Xb_shuffled[i:i+minibatch_size] # minibatch_size를 활용해 batch 단위로 샘플링
        ty = y_shuffled[i:i+minibatch_size] # minibatch_size를 활용해 batch 단위로 샘플링
        gradients = 2 / minibatch_size * tx.T.dot(tx.dot(beta_mgd) - ty) # 확률적 경사 하강법에 해당하는 gradient를 계산하세요
        beta_mgd = beta_mgd - eta * gradients # update rule에 따라서 beta_mgd를 update 하세요

        beta_mgd_path.append(beta_mgd)

y_predict_mgd = beta_mgd[0] + beta_mgd[1] * X_new # 얻은 회귀 계수를 가지고 X_new에 대해서 y값을 예측하세요
##############################
```

mini batch 경사 하강법은 학습에 사용하는 데이터의 양을 batch_size를 이용해 조절하여 사용한다는 점입니다. 코드에서는 총 100개의 데이터 중에서 20개를 batch_size로 정하여 중복 추출하고 있습니다. 추출한 데이터를 학습에 사용하는데, 장점은 역시 원본 데이터를 모두 사용하는 것 보다 오차 수정이 빠르고 슈팅이 발생하기 때문이 어느정도의 local minima는 회피 가능합니다.

sklearn의 `LinearRegression`과 코드로 구현한 모델의 비교

sklearn은 인공지능 관련 tool을 간편하게 사용할수 있도록 구현된 라이브러리 입니다. 안에는 다양한 도구들이 있고, 많이 사용하는 데이터, 연구된 모델 등도 수학적인 지식 없이 쉽게 불러와서 사용할 수 있습니다.

sklearn.linear_model의 LinearRegression도 같은 의미 입니다. 이미 라이브러리 상에서 class의 형태로 구현된 모델을 손쉽게 불러와서 파라미터를 수정하고, 학습시킬 수 있습니다. 또한 오차를 계산하는 MSE, MAE, SSE등의 수식도 조정 가능합니다.

sklearn의 LinearRegression의 회귀계수 추정은 최소 제곱법(Ordinary Least Squares, OLS)를 사용합니다. OLS는 주어진 데이터와 예측값 간의 오차를 최소화하는 회귀 계수를 추정하는 방법입니다. OLS의 수식은 아래와 같습니다.

OLS

$$(X^T X)^{-1} X^T y \ : \ \text{coef}$$

이와 비교하여 코드로 구현한 경사 하강법의 회귀 계수 추정 방법입니다.

GD

$$\nabla_\beta L(\beta) = 2x_i^{\mathrm{T}}(x_i^{\mathrm{T}}\beta - y) \ : \ \text{gradient}$$
$$\beta^{(t+1)} \Leftarrow \beta^{(t)} - \eta \nabla_\beta L(\beta) \ : \ \text{update rule}$$

SGD

$$\nabla_\beta L(\beta) = \frac{2}{m}\sum_{i=1}^{m} x_i^{\mathrm{T}}(x_i^{\mathrm{T}}\beta - y) \ : \ \text{gradient}$$
$$\beta^{(t+1)} \Leftarrow \beta^{(t)} - \eta \nabla_\beta L(\beta) \ : \ \text{update rule}$$

Mini batch

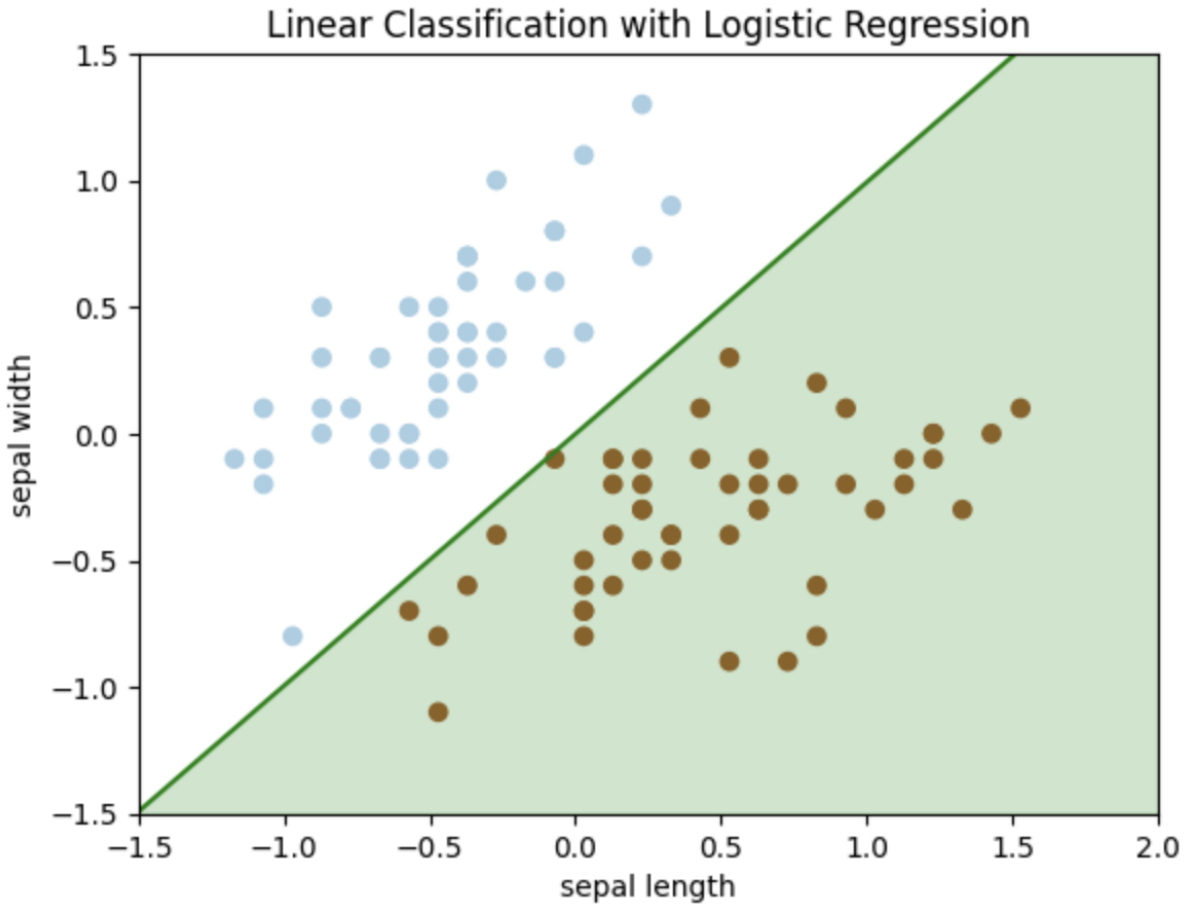
$$\nabla_\beta L(\beta) = \frac{2}{n} X^{\mathrm{T}}(X\beta - y) \ : \ \text{gradient}$$
$$\beta^{(t+1)} \Leftarrow \beta^{(t)} - \eta \nabla_\beta L(\beta) \ : \ \text{update rule}$$

반면 코드로 구현하면, 모두 베이스 부터 시작해야해서, 파라미터도 하나부터 열까지 선언하고 default값을 적어주어야 합니다. 기본적으로 학습 반복수인 epoches 값을 설정해주고 learning rate, batch가 필요하다면 batch_size 또한 적어주어야 합니다. 또한 기존의 수식을 참고하여 코드로 변환과정을 거쳐야하고, 데이터의 feature에 따라서 코드 또한 동적으로 반응하도록 설계해주어야 합니다. 그리고 모델의 학습과 예측 과정, 점수에 관한 함수또한 설계가 필요하여 sklearn으로 구현된 모델을 사용하는 것에 대비하여 많은 작업이 요구됩니다.

2. Logistic Regression (선형 분류)

아이리스 데이터를 바탕으로 2가지 꽃받침의 높이와 너비 데이터를 사용하였습니다.

데이터를 label에 맞게 classification하고, 각 경사하강법에 따라서 다른 특징을 확인하고 sklearn의 LogisticRegression와 코드로 작성한 모델간의 차이를 비교합니다.



로지스틱 회귀 모델을 선형 분류 모델으로 시그모이드 함수의 특성을 이용해 데이터를 분류합니다.

scikit-learn의 `LogisticRegression`

```
from sklearn.linear_model import LogisticRegression # 선형 분류 모델

model = LogisticRegression(fit_intercept=False,random_state=42) # 모델
model.fit(X_train,y_train) # 학습
y_predict = model.predict(X_test) # 예측
```

scikit-learn의 로지스틱 회귀(Logistic Regression) 모델은 이진 분류(Binary Classification) 문제를 다루는 선형 분류 모델입니다. 로지스틱 회귀는 선형 회귀 모델을 기반으로 하지만, 출력을 확률 값으로 변환하여 클래스의 확률을 추정합니다.

로지스틱 회귀는 S자 모양의 로지스틱 함수 또는 시그모이드 함수(Sigmoid Function)를 사용하여 입력 변수의 가중치와 절편을 조정합니다. 이 함수는 입력 값을 0과 1 사이의 확률 값으로 변환합니다. 이를 통해 어떤 클래스에 속할 확률을 추정할 수 있습니다.

Logistic Regression 모델의 파라미터

```
{'C': 1.0,
'class_weight': None,
'dual': False,
'fit_intercept': True,
'intercept_scaling': 1,
'l1_ratio': None,
'max_iter': 100,
'multi_class': 'auto',
'n_jobs': None,
'penalty': 'l2',
```

```
'random_state': None,
'solver': 'lbfgs',
'tol': 0.0001,
'verbose': 0,
'warm_start': False}
```

GD

```
beta_bgd_path = list()
eta = 0.1 # 학습률
n_epochs = 500 # epoch 수
n = 100 # 샘플수

# seed 고정
np.random.seed(42)
beta_bgd = np.random.randn(2,1)

for iteration in range(n_epochs):
    gradients = 1/n * X.T.dot(sigmoid(X.dot(beta_bgd)) - y) # 배치 경사 하강법에 해당하는 gradient를 계산하세요
    beta_bgd = beta_bgd - eta * gradients # update rule에 따라서 beta_bgd를 update 하세요

beta_bgd_path.append(beta_bgd)
```

데이터를 sigmoid함수에 넣어서 0인지 1인지 분류합니다. 그리고 gradiant에 learning rate를 곱하여 오차를 추청합니다. 선형 회귀와 마찬가지로 batch Gradiant Desecent에서는 모델의 데이터 전부를 학습에 사용하기 때문에 이론상으로 정확한 결과를 낼수 있지만 학습이 오래 걸리고, Local minima 탈출이 어렵습니다.

SGD

```
beta_sgd_path = []
n_epochs = 500
t0, t1 = 5, 10  # 학습 스케줄 하이퍼파라미터
n = 100         # 샘플 수

def learning_schedule(t):
    return t0/(t+t1)

np.random.seed(42)
beta_sgd = np.random.randn(2,1)  # beta 무작위 초기화

for epoch in range(n_epochs):
    for i in range(n):
        eta = learning_schedule(epoch * n + i)

        random_idx = np.random.randint(n) # 0 ~ n-1 까지 랜덤하게 인덱스 선택
        tx = random_idx # random_idx를 활용해 샘플 하나 선택
        ty = random_idx # random_idx를 활용해 샘플 하나 선택
        # 확률적 경사 하강법에 해당하는 gradient를 계산하세요
        #print(X[tx].shape, beta_sgd.shape, y[ty].shape)
        gradients = X[tx].reshape(2, 1).dot((sigmoid(X[tx].T.dot(beta_sgd)) - y[ty]).reshape(1, 1)) # 배치 경사 하강법에 해당하는 gradient를
        # (2, 1) * (1, 1) = (2, 1)
        beta_sgd = beta_sgd - eta * gradients # update rule에 따라서 beta_mgd를 update 하세요

        beta_sgd_path.append(beta_sgd)
```

확률적 경사하강법도 random으로 인덱스를 추출하여 랜덤 하게 추출된 데이터 X를 sigmoid함수에 넣고 오차를 계산하여 경사 하강법을 진행합니다. 특징은 랜덤한 X 데이터 추출입니다. 주의해야 할 점은 데이터의 shape에 따라서 행렬 연산이 진행 되기 때문에 올바른 shape인지 확인하는것 이 중요합니다.

Mini batch

```
beta_mgd_path = []

n_epochs = 50
minibatch_size = 20

# seed 고정
np.random.seed(42)
beta_mgd = np.random.randn(2,1)  # 랜덤 초기화

t0, t1 = 200, 1000 # 학습 스케줄 하이퍼파라미터
def learning_schedule(t):
    return t0 / (t + t1)
```

```
t = 0

for epoch in range(n_epochs):
    shuffled_indices = np.random.permutation(n) # 0 ~ n-1 까지 랜덤하게 인덱스를 섞어주세요
    X_shuffled = X[shuffled_indices] # X 랜덤 인덱싱
    y_shuffled = y[shuffled_indices] # y 랜덤 인덱싱
    for i in range(0, n, minibatch_size): # 0 ~ n-1 까지 minibatch_size 단위로 샘플링
        t += 1
        eta = learning_schedule(t)

        tx = X_shuffled[i:i+minibatch_size] # minibatch_size를 활용해 batch 단위로 샘플링
        ty = y_shuffled[i:i+minibatch_size] # minibatch_size를 활용해 batch 단위로 샘플링
        gradients = 1 / minibatch_size * tx.T.dot(sigmoid(tx.dot(beta_mgd)) - ty) # 미니 배치 경사 하강법에 해당하는 gradient를 계산하세요
        beta_mgd = beta_mgd - eta * gradients # update rule에 따라서 beta_mgd를 update 하세요
        beta_mgd_path.append(beta_mgd)
```

mini batch는 랜덤한 인덱스 범위를 정해서 추출하여 그 범위의 데이터만 학습에 이용하는 것입니다. 위의 코드에서는 20개의 데이터만 랜덤 추출하여 학습에 사용하고 오차를 수정하며 다시 20개를 추출 총 50회의 epochs가 끝날 때 까지 gradiant를 업데이트 해줍니다.

sklearn의 `LogisticRegression`과 코드로 구현한 모델의 비교

sklearn.linear_model의 LogisticRegression은 회귀 모델을 이용하여 분류 모델을 구현한 것입니다.

로지스틱 회귀모델의 가장 큰 특징은 activation을 sigmoid 함수를 사용하여 분류를 하는 것입니다. 코드 상으로 구현할때도 sigmoid함수를 사용하여 label 데이터를 예측하고 분류하였습니다. sklearn라이브러리의 LogisticRegression의 특징은 다양한 하이퍼 파라미터를 사용하여 모델의 파라미터 튜닝을 가능하게 한다는 점입니다. 기본적으로 penalty, C, fit_intercept, solver, class_weight 등의 파라미터가 있고, 규제 강도 조절, 최적화 알고리즘 반복횟수 조절, 과적합 방지등의 성능 향상을 위한 튜닝을 수행할 수 있습니다. 또한 이미 구현된 sklearn의 다른 도구들도 사용 가능하여, 모델의 성능 확인과 지표, 반복적인 파라미터 튜닝 작업도 쉽게 수행할 수 있습니다.