MLDL **SVM** Coding Assignment Report

20225092 손희경(Son Hui-gyoeng)
Github ID : gyoenge

[Report1] Hard margin SVM

1. Instruction

Draw a decision boundary that perfectly separates the two datasets. Implement the process of finding the optimal decision boundary using **hinge loss** and **coordinate gradient descent**(What you learned in the lesson!). The report should include a reasoned explanation of how the decision boundary was drawn. The **size of the margin does not necessarily have to be the maximum**, and will only be evaluated for **complete separation** of the data relative to the decision boundary.

*Hint : The hinge loss can be used to find the optimal decision boundary using the gradient descent method. **Compute LOSS for all data points** to find the optimized weight and bias. The **constant of hinge loss can be modified to determine the margin of the decision boundary**. Make sure the decision boundary divides all data points well.

2. Load Dataset

```python
def load_HSVM_dataset():
    iris = datasets.load_iris()
    X = iris.data[:100, :2]
    y = iris.target[:100]
    # 레이블을 -1, 1로 변환
    labels = np.unique(y)
    y = np.where(y == labels[1], 1, -1)
    return X, y
```

Code. HSVM iris dataset load

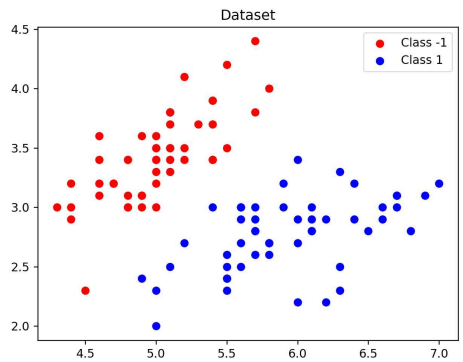


Figure. HSVM iris dataset

HSVM(Hard margin SVM)에 사용할 iris 데이터셋을 시각화하면 위와 같다. 총 100개의 데이터포인트에, class -1에 50개 데이터, class 1에 50개 데이터를 가진다. 데이터의 feature dimension은 2이다. 특히 데이터셋이 **linearly separable함**을 확인할 수 있다.

3. Implement

(1) hard margin SVM과 hinge loss

hard margin SVM은 선형 분리 가능한 데이터에 대해서 마진을 최대화하는 초평면을 찾는 기법이다. 마진을 최대화하기 위해 제약조건을 힌지 손실(hinge loss)의 형태로 표현하고 최적화 문제를 해결할 수 있다.

Hard Margin SVM의 목표는 다음과 같이 제약 조건 하에서 margin을 최대화하는 것

이다:

$$\min_{w,b} \quad \frac{1}{2}\|w\|^2$$
$$\text{subject to} \quad y_i(w \cdot x_i + b) \geq 1, \quad \forall i$$

이때, 각 데이터 포인트에 대한 제약조건을 hinge loss로 변환하여 다음과 같이 손실 함수로 포함시킬 수 있다:

$$\text{loss}_i = \max(0, 1 - y_i(\mathbf{w} \cdot \mathbf{x}_i + b))$$

그러면 이를 통해 다음과 같이 최적화 문제를 정할 수 있는데, 이때 C는 정규화 파라미터로, 이 값은 Hard Margin SVM의 경우 매우 크게 설정된다. 이는 Hinge loss가 0이 되도록 C를 무한대로 설정하는 것의 의미라고 볼 수 있다. 따라서 제약 조건을 직접 포함하여 최적화 문제를 해결할 수 있다:

$$\min_{w,b} \quad \frac{1}{2}\|w\|^2 + C\sum_{i=1}^{n} L(y_i, f(x_i))$$

(2) hinge loss와 sub gradient descent

```python
def hinge_loss_gradient(X, y, weights, bias):
    n_samples, n_features = X.shape

    # init dw, db
    dw = np.zeros(n_features)
    db = 0

    # calculate gradient for given datapoints
    for i in range(n_samples):
        # check condition for sub-gradient with hinge loss
        condition = y[i] * (np.dot(weights, X[i]) + bias)
        if condition < 1:
            dw -= y[i] * X[i]
            db -= y[i]
        else:
            pass

    # normalize dw, db with n_samples
    dw /= n_samples
    db /= n_samples

    return dw, db
```

Code. calculate hinge loss sub gradient

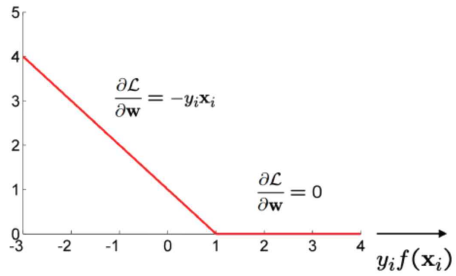


Figure. hinge loss의 sub-gradient 값

Hinge Loss는 y*f(x)=1을 경계로 불연속이다. 따라서 해당점은 미분불가능한데, 이를 sub-gradient descent 방법으로 해결할 수 있다. 위의 코드와 같이 미분불가능 점을 기준으로 두 범위에 대해 따로 미분 값을 계산하여 합한다.

(3) optimizer : SGD, CGD, ADAM

```python
def train_svm(X, y, learning_rate=0.01, epochs=1000, C=1000, optimizer='CGD'):
    n_samples, n_features = X.shape

    # Normalize the features
    mean = X.mean(axis=0)
    std = X.std(axis=0)
    X = (X - mean) / std

    # init weights, bias
    weights = np.zeros(n_features)
    bias = 0
```

Code. train HSVM – initialize part

위와 같이 최적화를 시작하기 전 X를 정규화하여 모델 학습이 안정적으로 이루어지게 하였고, 최적화할 파라미터인 weights와 bias를 0으로 초기화하였다.

Hard margin SVM 최적화를 수행할 때, 총 세 가지 optimizer를 시도하였다. 특히 각

각의 optimizer를 이용해 최적화할 때 모두 hinge_loss_gradient 함수를 사용하였다.

첫 번째로, SGD(Stochastic Gradient descent)를 사용하였다. SGD는 각 샘플에 대해 계산된 경사도를 사용하여 매번 모델 매개변수를 업데이트하는 방법이다. 이는 빠른 수렴을 가능하게 하지만, 소음에 민감할 수 있다.

```
# 1) Stochastic sub-gradient descent (SGD)
if optimizer == 'SGD':
    for epoch in range(epochs):
        # for i in range(n_samples):
        indices = np.random.permutation(n_samples)
        for i in indices:
            # Calculate the gradient for each sample
            dw, db = hinge_loss_gradient(X[i:i+1], y[i:i+1], weights, bias)

            # Update weights and bias with regularization term
            weights -= learning_rate * (C * dw + weights)
            bias -= learning_rate * C * db
```

Code. train HSVM – optimize part 1 : SGD

두 번째로, Instruction의 요구 조건에 해당하는 CGD(Coordinate Gradient descent)는 각 반복마다 하나의 좌표(특징)에 대해 경사도를 계산하여 그 축을 따라 최적화하는 방법이다. 이는 고차원 데이터에서 효율적일 수 있으며, 각 특징이 독립적일 때 효과적이다.

```
# 2) Coordinate sub-gradient descent (CGD)
elif optimizer == 'CGD':
    for epoch in range(epochs):
        for j in range(n_features):
            gradient_w = 0
            gradient_b = 0
            # Calculate the gradient for each weight
            dw, db = hinge_loss_gradient(
                X[:, j:j+1], y, weights[j:j+1], bias)

            # Update weights and bias with regularization term
            gradient_w += C * dw + weights[j:j+1]
            gradient_b += C * db

            # gradient_w /= n_samples
            # gradient_b /= n_samples

            # Update weights and bias
            weights[j] -= learning_rate * gradient_w
            bias -= learning_rate * gradient_b

            # Avoid overflow by capping the updates
            if np.abs(weights[j]) > 1e5:
                weights[j] = np.sign(weights[j]) * 1e5
            if np.abs(bias) > 1e5:
                bias = np.sign(bias) * 1e5
```

Code. train HSVM – optimize part 2 : CGD

세 번째로, ADAM(Adaptive Moment Estimation)을 사용하였다. ADAM은 모멘텀과 RMSProp의 장점을 결합하여 각 매개변수에 대한 개별 학습률을 적용하는 방법이다. 이는 경사도의 첫 번째와 두 번째 모멘트 추정치를 사용하여 학습률을 적응적으로 조정하며, 빠르고 안정적인 수렴을 보장한다.

```
# 3) Adam (ADAM)
elif optimizer == 'ADAM':
    beta1 = 0.9
    beta2 = 0.999
    epsilon = 1e-8
    m_w = np.zeros(n_features)
    v_w = np.zeros(n_features)
    m_b = 0
    v_b = 0
    for epoch in range(epochs):
        dw, db = hinge_loss_gradient(X, y, weights, bias)

        # Update biased first moment estimate
        m_w = beta1 * m_w + (1 - beta1) * dw
        m_b = beta1 * m_b + (1 - beta1) * db

        # Update biased second raw moment estimate
        v_w = beta2 * v_w + (1 - beta2) * (dw ** 2)
        v_b = beta2 * v_b + (1 - beta2) * (db ** 2)

        # Compute bias-corrected first moment estimate
        m_w_hat = m_w / (1 - beta1 ** (epoch + 1))
        m_b_hat = m_b / (1 - beta1 ** (epoch + 1))

        # Compute bias-corrected second raw moment estimate
        v_w_hat = v_w / (1 - beta2 ** (epoch + 1))
        v_b_hat = v_b / (1 - beta2 ** (epoch + 1))

        # Update weights and bias
        weights -= learning_rate * m_w_hat / (np.sqrt(v_w_hat) + epsilon)
        bias -= learning_rate * m_b_hat / (np.sqrt(v_b_hat) + epsilon)
```

Code. train HSVM – optimize part 3 : ADAM

4. Visualize result & Analyze

(1) Visualize the decision boundary

사용한 parameter는 **C=1000, epoches=1000, learning_rate=0.01** 으로 통일하였다.

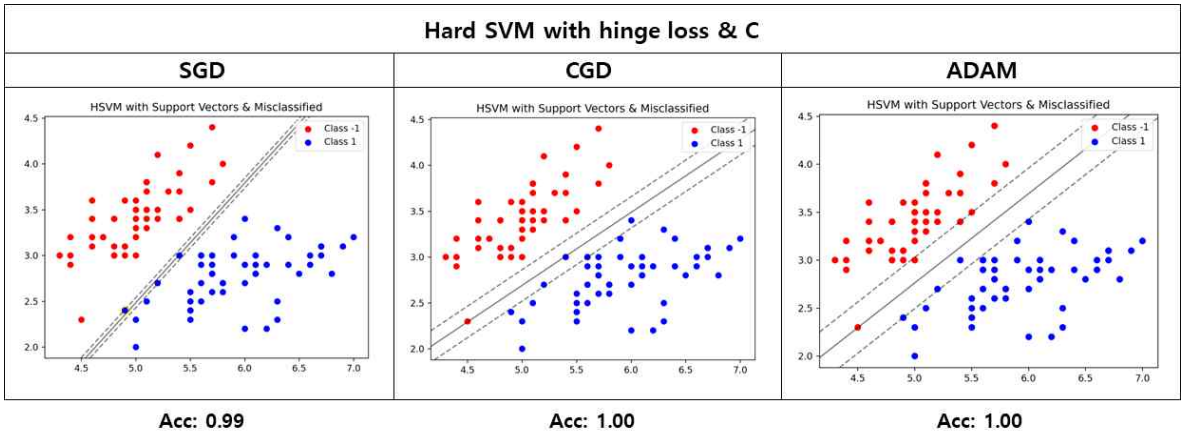


Figure. Hard SVM, optimizer 종류별 Decision boundary 및 정확도 결과

(2) Analyze

C=1000으로 하여 Hard margin SVM을 구현할 수 있었다. CGD와 ADAM의 경우 정확도 100%로 데이터의 complete separation이 완료되었다. 그러나 SGD의 경우 확률적 특성으로 인해 결정 경계에 약간의 불안정성이 생길 수 있어 0.94~1.00 사이의 정확도로 매번 변화하는 것을 확인할 수 있었다.

[Report2] Soft margin SVM

1. Instruction

Find a decision boundary of two classes by solving **dual problem**. Also, find a decision boundary of two classes by solving **primal problem**. Compare two solutions of each problem and **analyze differences**. Visualize the decision boundary of an SVM and **analyze how slack variables allow misclassification**. You can also **adjust the hyperparameters** to find best accuracy

2. Load Dataset

```
def load_SSVM_dataset():
    iris = datasets.load_iris()
    X = iris.data[50:, 2:]
    y = iris.target[50:]
    # 레이블을 -1, 1로 변환
    labels = np.unique(y)
    y = np.where(y == labels[1], 1, -1)
    return X, y
```

Code. SSVM iris dataset load

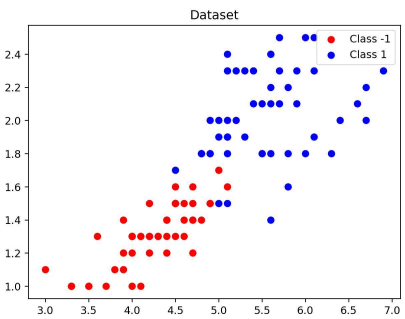


Figure. SSVM iris dataset

SSVM(Soft margin SVM)에 사용할 iris 데이터셋을 시각화하면 위와 같다. 총 100개의

데이터포인트에, class -1에 50개 데이터, class 1에 50개 데이터를 가진다. 각 데이터포인트의 feature dimension은 2이다. 특히 이 데이터셋은 linearly separable 하지 않다.

3. Implement

linearly separable 하지 않은 데이터셋을 분류하기 위해, soft margin SVM을 구현한다. 두 가지 방식으로 구현할 수 있는 데, 1. primal problem을 푸는 방식 2. dual problem을 푸는 방식이다.

(1) Solve with Primal problem

```python
class PrimalSSVM:
    def __init__(self, C=1.0, epoches=1000, learning_rate=0.01):
        # model weights & bias
        self.weights = None
        self.bias = None
        # learning hyperparameters
        self.epoches = epoches
        self.lr = learning_rate  # learning rate
        self.C = C

    def _normalizeX(self, X):
        return (X - self.mean) / self.std

    def fit(self, X, y, optimize="SGD"):
        if optimize == "SGD":
            self._SGD(X, y)
        elif optimize == "CGD":
            self._CGD(X, y)
        else:
            raise NameError(f"Unsupported optimization method: {optimize}")
```

Code. Primal SSVM – initialize part

```python
def _SGD(self, X, y):
    """Stochastic GD"""

    # normalize X (fit 시에만)
    self.mean = X.mean(axis=0)
    self.std = X.std(axis=0)
    X = self._normalizeX(X)

    n_samples, n_features = X.shape
    # init weights, bias with 0
    self.weights = np.zeros(n_features)
    self.bias = 0

    # optimize
    for epoch in range(self.epoches):
        for i in range(n_samples):
            condition = y[i] * \
                (np.dot(X[i], self.weights) + self.bias) >= 1
            if condition:
                self.weights -= self.lr * self.weights
            else:
                self.weights -= self.lr * (
                    self.weights - self.C * np.dot(X[i], y[i])
                )
                self.bias += self.lr * self.C * y[i]

    # 가중치와 편향을 원래 스케일로 조정
    self.weights = self.weights / self.std
    self.bias = self.bias - np.sum(self.weights * self.mean)

    return self.weights, self.bias
```

Code. Primal SSVM – SGD

```python
def _CGD(self, X, y):
    """Coordinate GD"""

    # normalize X (fit 시에만)
    self.mean = X.mean(axis=0)
    self.std = X.std(axis=0)
    X = self._normalizeX(X)

    n_samples, n_features = X.shape
    # init weights, bias with 0
    self.weights = np.zeros(n_features)
    self.bias = 0

    # optimize
    for epoch in range(self.epoches):
        for j in range(n_features):
            gradient_w = 0
            gradient_b = 0
            for i in range(n_samples):
                condition = y[i] * \
                    (np.dot(X[i], self.weights) + self.bias) >= 1
                if condition:
                    gradient_w += self.weights[j]
                else:
                    gradient_w += self.weights[j] - self.C * X[i][j] * y[i]
                    gradient_b -= self.C * y[i]

            gradient_w /= n_samples
            gradient_b /= n_samples

            # 가중치와 바이어스 업데이트
            self.weights[j] -= self.lr * gradient_w
            self.bias -= self.lr * gradient_b
```

Code. Primal SSVM – CGD

Soft Margin SVM은 더 나은 일반화를 위해 일부 오분류를 허용한다. 이러한 오분류에 대한 패널티(term)는 C라는 매개변수를 통해 조절된다. Primal Soft Margin SVM 모델은 hinge loss과 regularization term을 결합한 손실 함수를 최소화하도록 매개변수를 조정한다. 이때, optimizer로 SGD와 CGD를 사용하였다.

$$\min_{w,b} \frac{1}{2}||w||^2 + C\sum_{i=1}^{N}\xi_i$$

$$\text{subject to } y_i(w^T x_i + b) \geq 1 - \xi_i, \quad \xi_i \geq 0$$

(2) Solve with Dual problem

```python
class DualSSVM:
    def __init__(self, C=1.0, epoches=1000, learning_rate=0.01):
        # model weights & bias
        self.weights = None
        self.bias = None
        # learning hyperparameters
        self.epoches = epoches
        self.lr = learning_rate  # learning rate
        self.C = C
        self.alpha = None
```

Code. Dual SSVM – initialize part

```python
def _normalizeX(self, X):
    return (X - self.mean) / self.std

def fit(self, X, y):
    # normalize X (fit 시에만)
    self.mean = X.mean(axis=0)
    self.std = X.std(axis=0)
    X = self._normalizeX(X)

    n_samples, n_features = X.shape
    # init alpha, weights, bias with 0
    self.alpha = np.zeros(n_samples)
    self.weights = np.zeros(n_features)
    self.bias = 0

    for epoch in range(self.epoches):
        for i in range(n_samples):
            # Gradient of the dual objective with respect to alpha_i
            gradient = np.dot(self.weights, y[i] * X[i]) - 1

            # Update alpha_i using the projected gradient method
            alpha_i_old = self.alpha[i]
            # 0~C 사이 값으로 alpha제한
            alpha_i_new = np.clip(
                alpha_i_old - gradient / (np.dot(X[i], X[i])), 0, self.C)
            self.alpha[i] = alpha_i_new

            # Update the weight vector
            self.weights += (alpha_i_new - alpha_i_old) * y[i] * X[i]

        # Calculate bias
        self.bias = np.mean(y - np.dot(X, self.weights))

    # 가중치와 편향을 원래 스케일로 조정
    self.weights = self.weights / self.std
    self.bias = self.bias - np.sum(self.weights * self.mean)

    return self.weights, self.bias
```

Code. Dual SSVM – optimize part

Dual Soft Margin SVM은 primal SSVM과 달리 듀얼 형태로 문제를 풀며, 라그랑주 승수법을 이용하여 최적화한다. 이는 라그랑주 승수인 알파(α)를 최적화하는 문제로 정의된다:

$$\max_{\alpha} \sum_{i=1}^{N} \alpha_i - \frac{1}{2} \sum_{i=1}^{N} \sum_{j=1}^{N} \alpha_i \alpha_j y_i y_j x_i^T x_j$$

$$\text{subject to } \sum_{i=1}^{N} \alpha_i y_i = 0, \quad 0 \leq \alpha_i \leq C$$

The solution is again given by

$$\mathbf{w} = \sum_{i=1}^{N_S} \alpha_i y_i \mathbf{x}_i.$$

이때 특히 알파는 C보다 작아야한다는 constraint가 생긴다. 이러한 constrained 최적화 문제에 대해, constraint를 벗어나면 projection하여 다시 범위 내로 조정시키는 projected gradient descent를 이용하여 최적화 과정을 구현하였다.

또한, Primal Soft Margin SVM와 Dual Soft Margin SVM에서 모두 최적화를 시작하기 전 X를 정규화하여 모델 학습이 안정적으로 이루어지게 하였고, 최적화할 파라미터인 weights와 bias를 0으로 초기화하였다.

4. Visualize result & Analyze

(1) Visualize the decision boundary

사용한 parameter는 **C=1.0, epoches=1000, learning_rate=0.01** 으로 통일하였다.

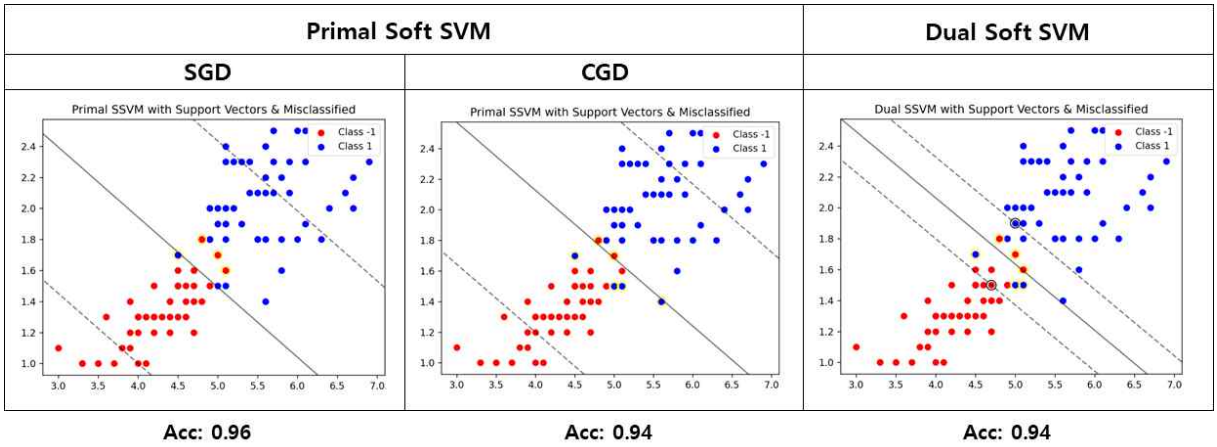


Figure. Soft SVM 종류별 Decision boundary 및 정확도 결과

(2) Analyze

위의 결과에 따르면, 모든 Soft SVM에서 약 95%내외의 높은 정확도를 보이는 것을 확인할 수 있었다. 그래프의 노란 표시는 오분류된 점이다. 최상의 정확도를 찾기 위해 C, learning rate, epochs를 적절히 조절하였고 위의 결과에 나온 파라미터가 결과가 잘 나오는 것을 확인할 수 있었다.

이때 Primal과 Dual을 비교하면, 같은 에폭과 C에 대해 Dual에서 더 적은 마진을 가지는 것을 확인할 수 있었다. 이를 통해 Dual이 Primal보다 더 빨리 수렴한다는 것을 확인할 수 있다. Dual 문제의 최적화가 상대적으로 덜 복잡하고, 더 적은 마진을 가지기 때문이다.

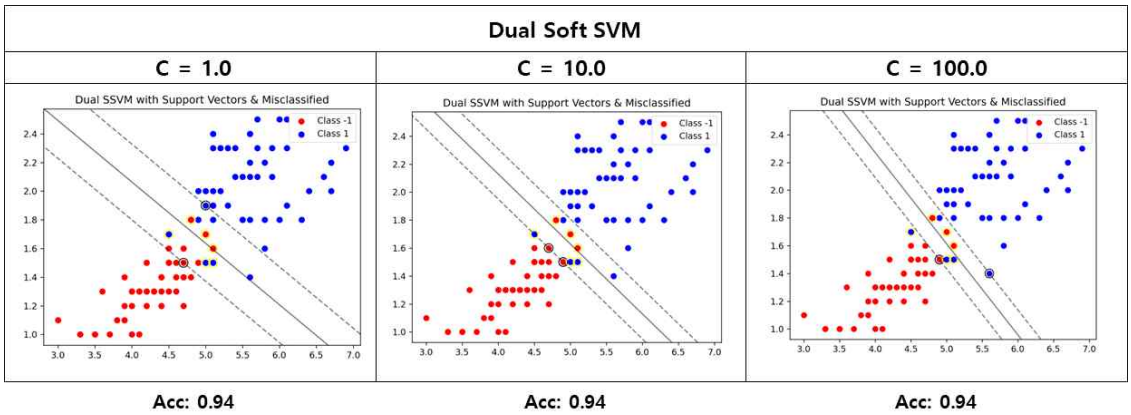


Figure. Dual Soft SVM, C증가에 따른 Decision boundary 및 마진 변화

뿐만 아니라 slack variable이 misclassification을 일부 허용하는 것을 시각적으로 확인할 수 있었다. 일부 점(오분류된 점)은 결정 경계의 반대편에 존재하며 오분류가 허용된다. 슬랙 변수는 엄격한 분리를 강제하지 않고 더 나은 일반화를 이루게 한다. 이를 통해 일부 오류를 허용하여 전체 정확도를 높일 수 있다. C값이 커질수록 slack variable의 패널티를 증가시켜 misclassification이 덜 허용되고 마진이 줄어드는 것을 확인할 수 있다.

[Report3] Kernel Tricks

1. Instruction

Apply **various kernel filters** to SVM and **compare** their performance. Also, you have to visualize the decision boundaries and support vectors of SVM with different kernel filters. The report should indicate which kernel used in the SVM **performed best**, **including reasons** based on visualized data.

2. Load Dataset

```
def load_KSVM_dataset():
    # you can change noise and random_state where noise >= 0.15
    dataset = datasets.make_moons(
        n_samples=300, noise=0.3, random_state=20)
    X, y = dataset
    # 레이블을 -1, 1로 변환
    labels = np.unique(y)
    y = np.where(y == labels[1], 1, -1)
    return X, y
```

Code. KSVM moon dataset load

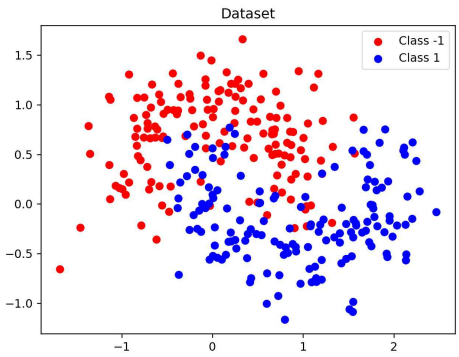


Figure. KSVM moon dataset

KSVM(Kernel trick SVM)에 사용할 iris 데이터셋을 시각화하면 위와 같다. 총 300개의 데이터포인트에, class -1에 150개 데이터, class 1에 150개 데이터를 가진다. 각 데이터포인트의 feature dimension은 2이다. 특히 이 데이터셋은 linearly separable 하지 않다. (moon 데이터셋은 linearly non-separable정도가 특히 심하다)

3. Implement

(1) parameter

```
class KSVM:
    def __init__(self, C=1.0, epoches=2000, learning_rate=0.001):
        self.C = C
        self.epoches = epoches
        self.learning_rate = learning_rate
        self.kernel = None
        self.alpha = None
        self.b = None
```

Code. KSVM parameters

Kernel trick SVM 모델에서 최적화되는 parameter에는 alpha(라그랑주 승수)와 b(바이어스)가 있다. 이는 dual Soft SVM에서의 것과 같은 의미이다. Kernel trick SVM은 dual Soft SVM에 Kernel trick을 사용한 것이기 때문이다. 결정 경계를 정의하는 데 기여하는 훈련 샘플들, 즉 서포트 벡터를 결정하는 데 alpha가 사용된다. b는 결정경계의 위치를 조정하여, 최적의 마진을 갖도록 한다.

정해야하는 hyperparameter에는 C, epoches, learning_rate가 있다. C는 Soft SVM에서와 같이 정규화 매개변수로, 결정 경계와 마진 사이의 균형을 설정한다. epoches와 learning_rate는 훈련 속도와 양을 결정하는 매개변수이다.

커널 함수	수식
Linear	$K(x_i, x_j) = x_i^T x_j$
Polynomial	$K(x_i, x_j) = \exp(-\gamma \|x_i - x_j\|^2)$
RBF (Radial Basis Function) / Gaussian	$K(x_i, x_j) = (\gamma x_i^T x_j + r)^d$
Sigmoid	$K(x_i, x_j) = \tanh(\gamma x_i^T x_j + r)$

Figure. 사용한 Kernel filter 목록

Kernel trick에서 핵심적인 역할을 하는 kernel은 데이터를 원래의 특징 공간에서 보다 높은 차원의 공간으로 매핑하여, 원래 공간에서 선형적으로 분리 불가능한 데이터를 고차원에서 선형적으로 분리할 수 있게 한다. 사용한 kernel의 종류에는 linear, rbf, polynomial, sigmoid가 있다.

(2) model fit

```
def _normalizeX(self, X):
    return (X - self.mean) / self.std

def fit(self, X, y, kernel="linear"):
    # kernels init
    self.kernel = {
        "linear": lambda X1, X2: np.dot(X1, X2.T),
        # rbf : gamma = 1.0
        "rbf": lambda X1, X2: np.exp(-1*np.linalg.norm(X1[:, np.newaxis] - X2[np.newaxis, :], axis=2) ** 2),
        # polynomial : degree = 3
        "polynomial": lambda X1, X2: (1 + np.dot(X1, X2.T)) ** 3,
        # sigmoid : gamma = 1, coef0 = 1
        "sigmoid": lambda X1, X2: np.tanh(np.dot(X1, X2.T) + 1)
    }[kernel]

    # normalize X (fit 시에만)
    self.mean = X.mean(axis=0)
    self.std = X.std(axis=0)
    X = self._normalizeX(X)
```

Code. KSVM model fit part 1

```
    # init parameters
    n_samples, n_features = X.shape
    self.alpha = np.zeros(n_samples)
    self.X_fit = X
    self.y_fit = y
    self.b = 0

    # Compute kernel matrix
    K = self.kernel(X, X)

    # optimize
    for _ in range(self.epoches):
        for i in range(n_samples):
            gradient = np.sum(self.alpha * y * K[:, i] * y[i]) - 1
            self.alpha[i] -= self.learning_rate * gradient / K[i, i]
            self.alpha[i] = np.clip(self.alpha[i], 0, self.C)

    # find Support Vectors
    self.support_vector_indices = np.where(
        (self.alpha > 1e-4) & (self.alpha < self.C))[0]
    self.b = np.mean([y[i] - np.sum(self.alpha * y * K[:, i])
                      for i in self.support_vector_indices])
```

Code. KSVM model fit part 2

Kernel SVM model fit 함수를 살펴보면, 특히 part 1에서 X를 정규화하여 데이터의 각 특성이 동일한 스케일을 갖게 함으로써 수렴이 더 잘되도록 하였다. 실제로 정규화 전후를 비교하였을 때 정규화 후에 더 잘 수렴하는 것을 확인할 수 있었다.

part 2의 optimize 부분에서는 dual Soft SVM 목적함수와 progection GD에 Kernel 함수 K를 적용하여 최적화를 진행한다. 각 데이터 포인트에 대해 alpha를 업데이트 하면서 동시에 0과 C 사이로 제한(클리핑)하여 오분류를 일부 허용하도록 한다.

Support vector는 Lagrange 멀티플라이어 alpha 값이 0보다 크고 정규화 파라미터 C보다 작은 데이터 포인트들로 결정된다. 이 값들은 결정 경계 근처에 위치하며, 클래스 간 마진을 정의하는 데 중요한 역할을 한다.

4. Visualize result & Analyze

(1) Visualize deicision boundaries and support vectors, for different kernel filters

사용한 parameter는 **C=1.0, epoches=2000, learning_rate=0.001** 으로 통일하였고, kernel filter로 linear, rbf, polynomial, ksvm을 사용한 결과는 각각 아래와 같다.

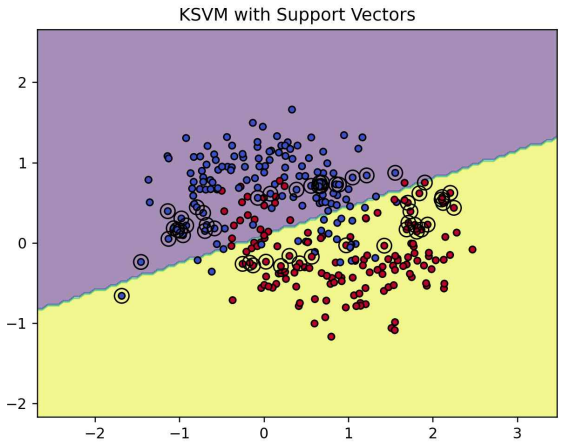


Figure. **Linear** kernel SVM (acc: **0.85**)

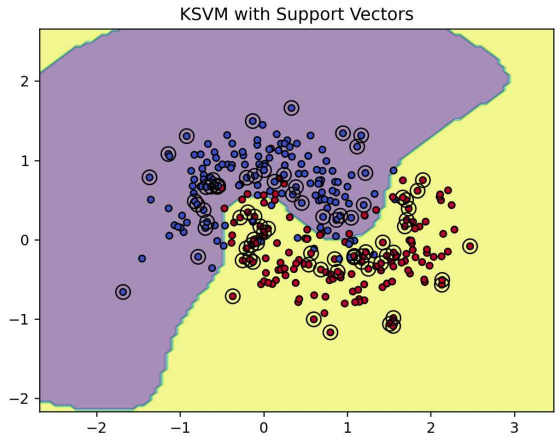


Figure. **Rbf** kernel SVM (acc: **0.90**)

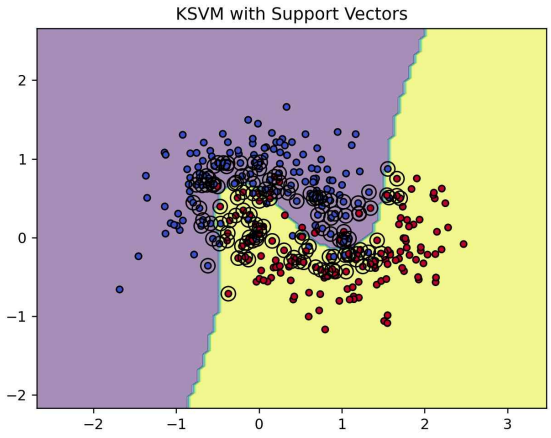


Figure. **Polynomial** kernel SVM (acc: **0.91**)

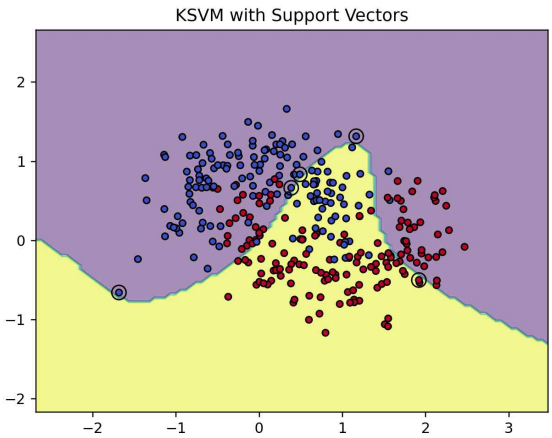


Figure. **Sigmoid** kernel SVM (acc: **0.63**)

(2) Analyze

위 결과에 따르면, best performed kernel은 Polynomial kernel이다. 아주 근소한 차이도 Rbf kernel도 best라고 할 수 있겠다. 이들은 복잡한 결정 경계를 통해 높은 정확도를 달성하며, 고차원 공간에서 데이터의 비선형 관계를 효과적으로 학습하여 다른 커널 대비 우수한 성능을 보여준다. Linear kernel은 그 구조가 상대적으로 단순하여 주로 선형적으로 분리 가능한 데이터셋에서 더 잘 작동하며, 복잡한 비선형 패턴을 가진 데이터셋

에서는 한계를 가져 약간 뒤떨어지는 성능을 보인다.

Sigmoid 커널은 복잡한 결정 경계를 가짐에도 이 상황에서 그다지 좋지못한 성능을 보이는데, sigmoid 함수는 다차원 공간에서 비선형적인 경계를 모델링하는 데 있어 다른 커널(예: Polynomial이나 Rbf)만큼 유연하지 않은 것이 원인이 될 수 있다. 뿐만 아니라, Sigmoid 커널은 매개변수(예: gamma와 coef0)에 대해 상대적으로 민감하며, 이들 매개변수의 조정이 적절히 이루어지지 않으면 모델의 성능에 크게 영향을 미칠 수 있어 커널 매개변수가 적절히 선택되지 않은 것이 좋지 못한 성능에 요인이 되었을 것이다.

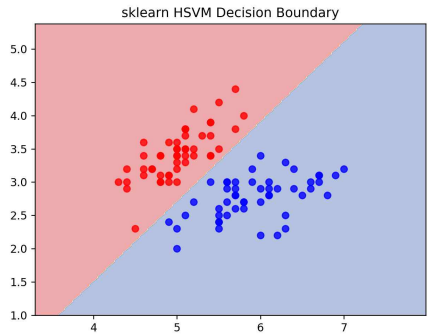
[Report4] Discussion

1. Instruction

Compare your implementation **with sklearn library** with **same hyper-parameters**.

2. Hard margin SVM : Implementation vs sklearn

사용한 parameter는 **C=1000** 이다.

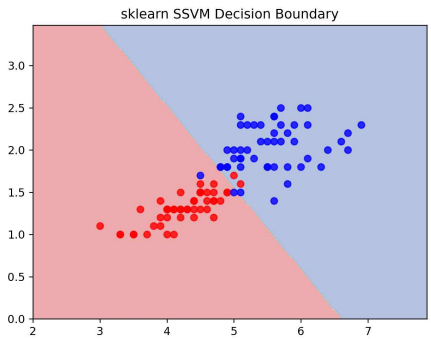


Hard SVM (acc: **1.00**)

sklearn Hard SVM의 결과, 직접 구현한 Hard SVM과 비슷한 결정 경계를 형성하여 완전히 데이터를 이진분류하는 것을 확인할 수 있었다.

3. Soft margin SVM : Implementation vs sklearn

사용한 parameter는 **C=1.0** 이다.

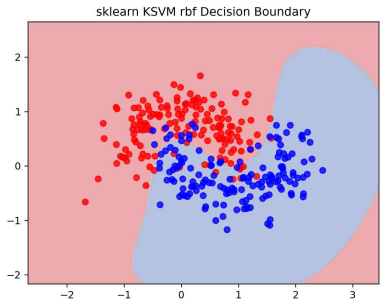


Soft SVM (acc: **0.95**)

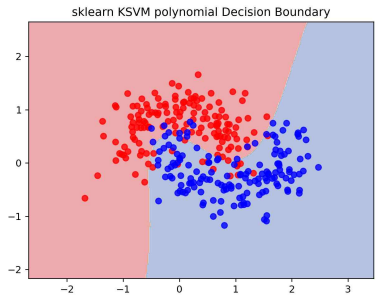
sklearn Soft SVM의 결과, 직접 구현한 Soft SVM과 비슷한 정확도와 결정 경계를 형성하여 데이터를 분류하는 것을 확인할 수 있었다.

4. Kernel Tricks : Implementation vs sklearn

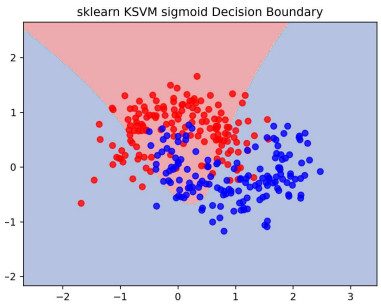
사용한 parameter는 **C=1.0, kernel 각각 맞게, gamma/degree/coef0 각각 맞게** 이다.



rbf (acc: **0.90**)



polynomial (acc: **0.90**)



sigmoid (acc: **0.66**)

위의 sklearn 라이브러리의 SVM 결과를 직접 구현한 rbf, polynomial, sigmoid kernel SVM들과 비교하였을 때, 정확도 수치에서 상당히 비슷한 결과를 보이는 것을 확인할 수 있다. 특히 직접 구현한 커널 SVM과 scikit-learn 라이브러리를 사용한 커널 SVM 모두 RBF와 다항식 커널에서 높은 정확도를 보여주며, 시그모이드 커널은 두 구현 모두에서 상대적으로 낮은 성능을 보이고 있다. 이처럼 일관된 결과를 통해, 커널 선택이 데이터셋의 특성에 따라 중요한 영향을 미칠 수 있다는 것을 알 수 있다.

한편, 직접 구현한 커널 SVM과 sklearn 라이브러리를 사용한 SVM 사이에서 정확도가 비슷하게 나타나면서도 결정 경계가 다르게 보이는 현상을 확인할 수 있었다. 이러한 현상은 최적화 알고리즘의 차이, 수치 안정성과 정밀도, 랜덤성의 영향 등과 같은 요인들에서 기인할 수 있다. 이러한 차이들을 통해, 동일한 데이터셋과 매개변수 설정을 사용하더라도 다른 결정 경계를 생성할 수 있음을 알 수 있다.