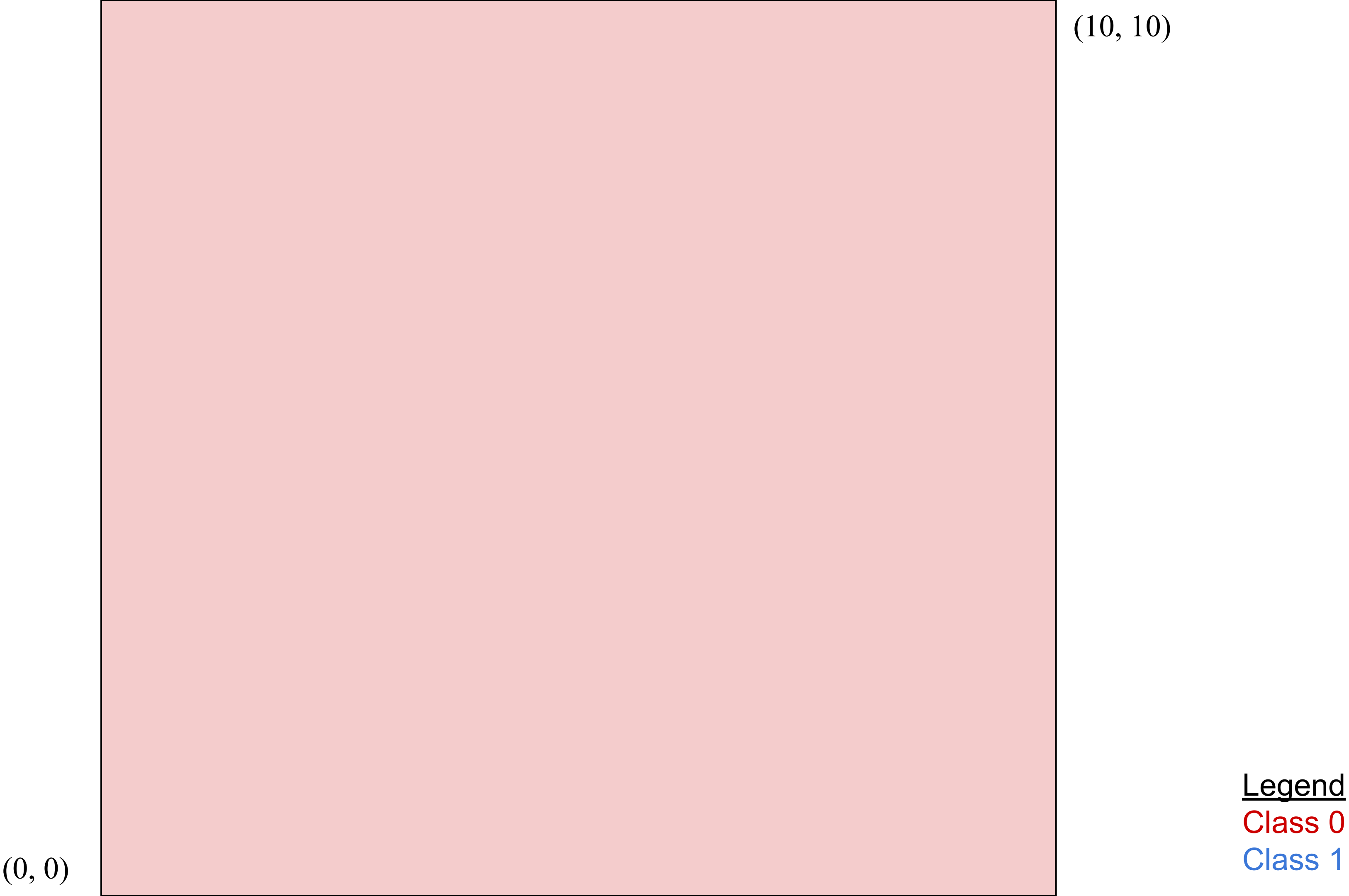
# General Case

## Segment Splitting Logic

(10, 10)
(0, 0)


(10, 10)
(0, 0)
Legend
Class 0
Class 1

# Obtaining *Prev*

## Query (*SET*)

$0 \leq \text{in}_1 \leq 10$

$0 \leq \text{in}_2 \leq 10$

$\forall\, \text{in}_1,\ \forall\, \text{in}_2,\ \text{output} \in \text{Class } 0?$

# Obtaining *Prev*

## Query (*SET*)

$0 \leq \text{in}_1 \leq 10$

$0 \leq \text{in}_2 \leq 10$

output $\geq \boldsymbol{\epsilon}$ ?

# Obtaining *Prev*

## Query (*SET*)

$0 \le \text{in}_1 \le 10$

$0 \le \text{in}_2 \le 10$

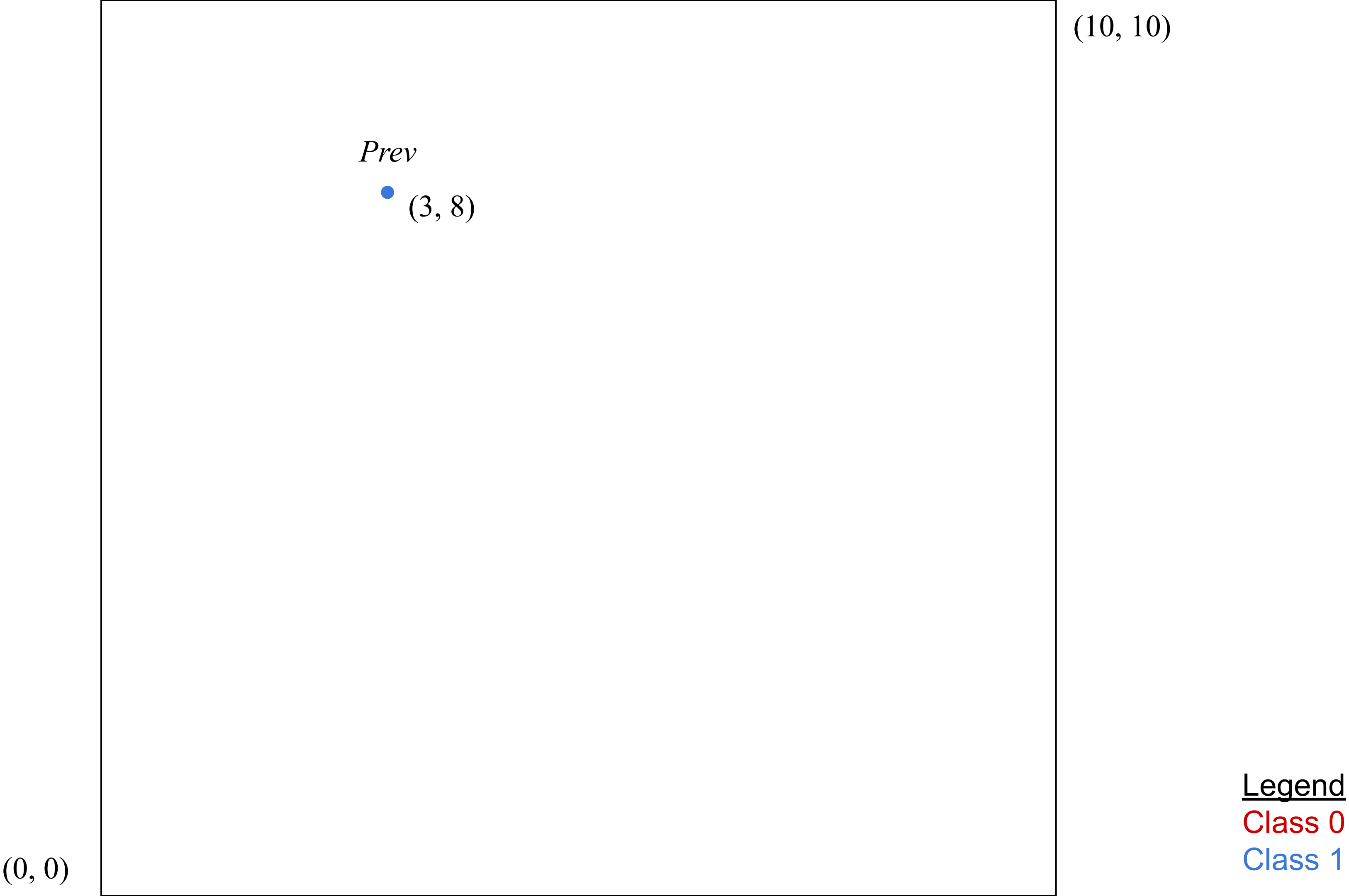
$\text{output} \ge \boldsymbol{\epsilon}$ ?

## Result

SAT

$\text{in}_1 = 3, \text{in}_2 = 8$

$\text{output} = 100$



(10, 10)
Prev
(3, 8)
(0, 0)
Legend
Class 0
Class 1

# Query (*SPLIT*)

$0 \leq \text{in}_1 \leq 10$

$0 \leq \text{in}_2 \leq 10$

$\forall \text{in}_1, \forall \text{in}_2, \text{output} \in \text{Class 1?}$

# **Obtaining *Curr***

# Obtaining *Curr*

## Query (*SPLIT*)

$0 \leq \text{in}_1 \leq 10$

$0 \leq \text{in}_2 \leq 10$

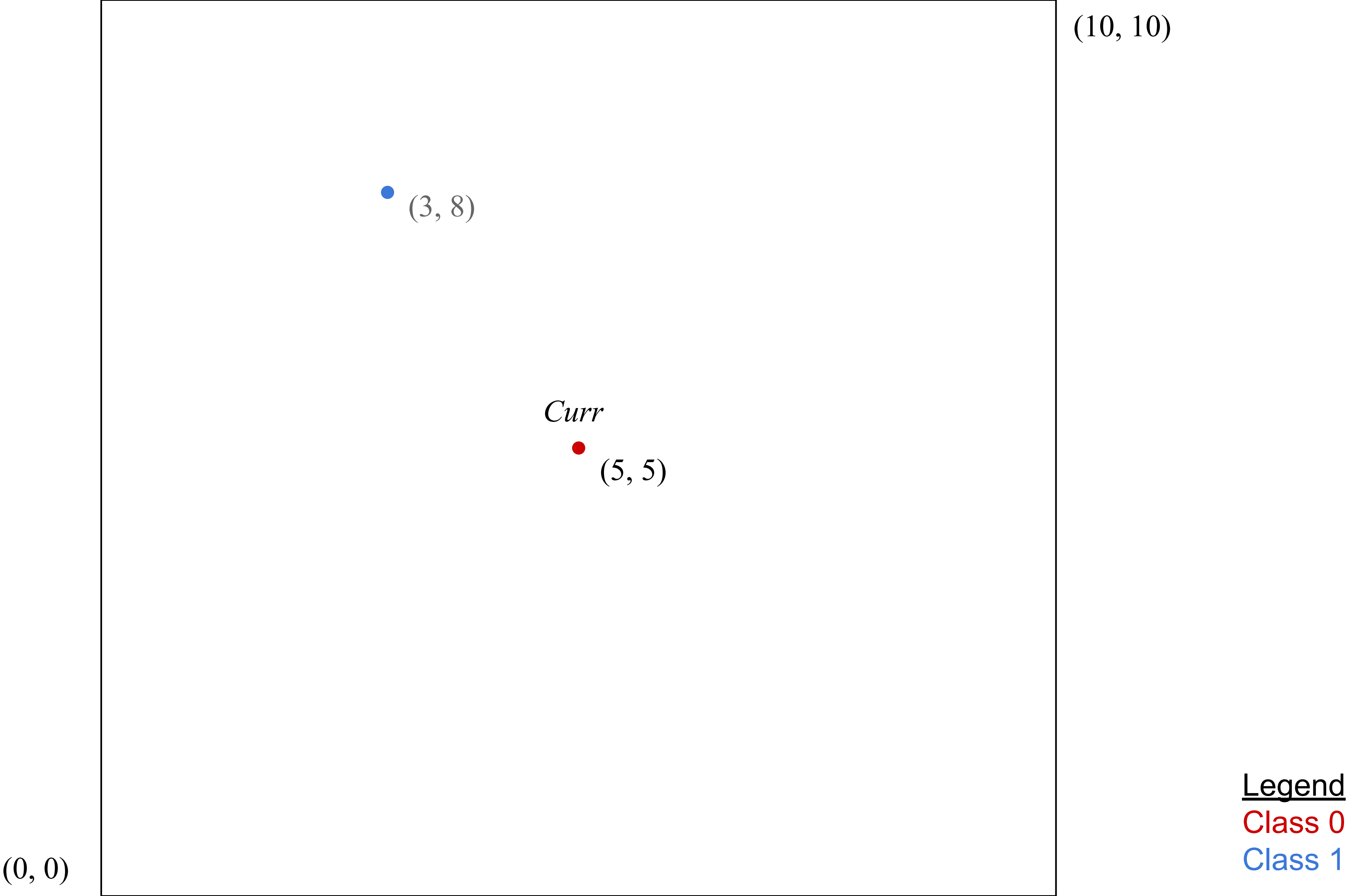
output ≤ 0 ?

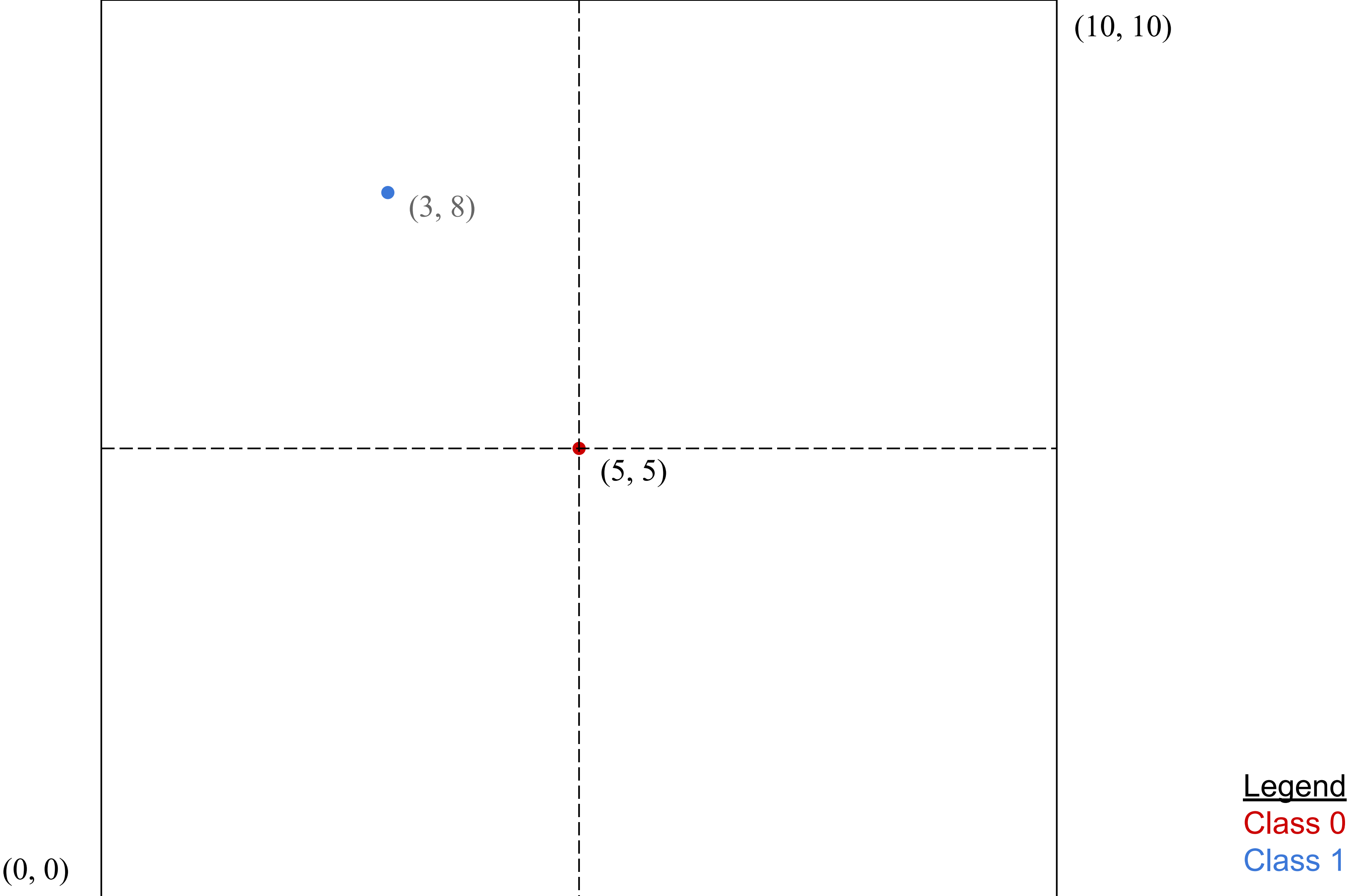
## Result

SAT

$\text{in}_1 = 5, \text{in}_2 = 5$

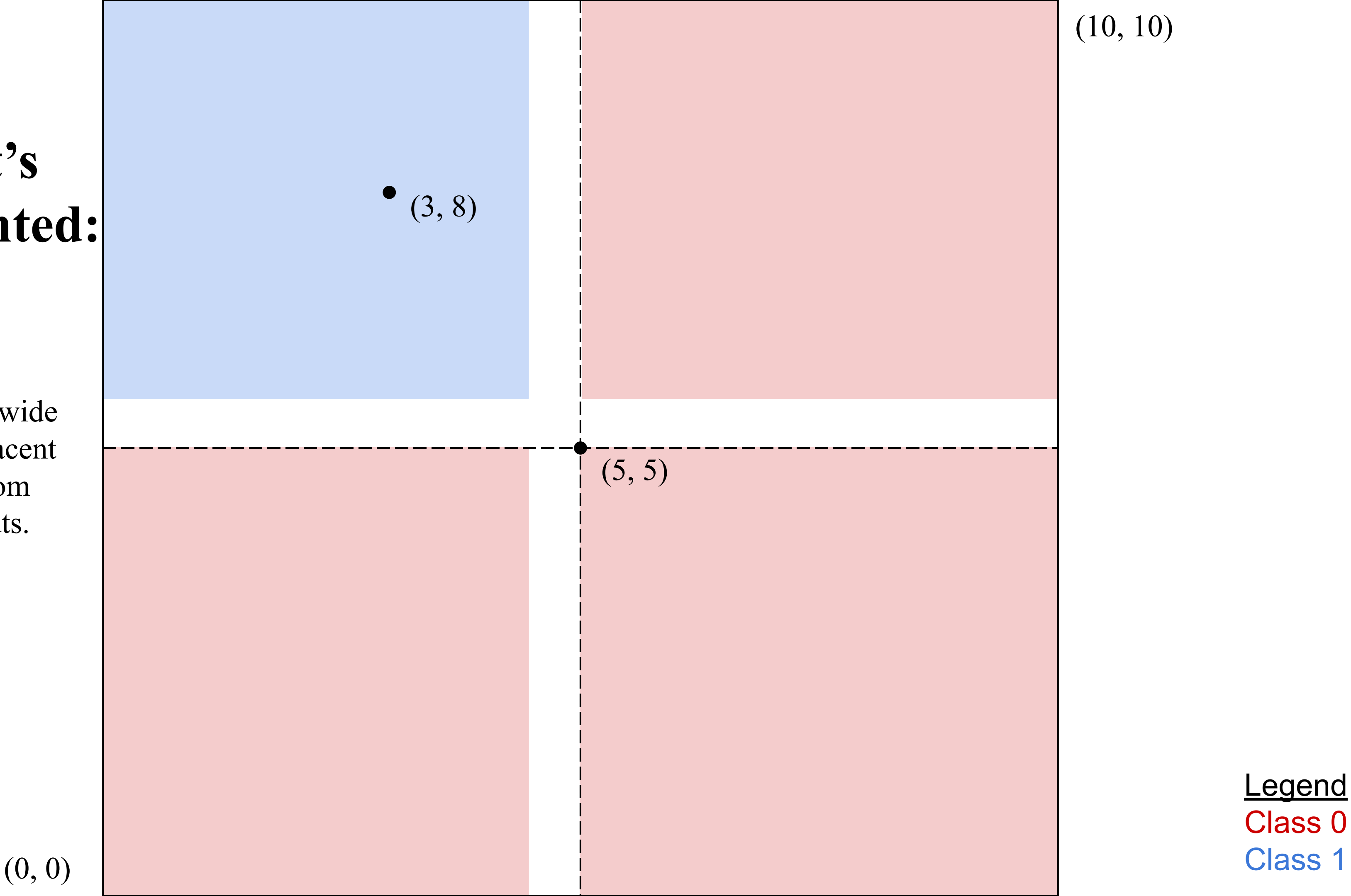
output = 0


(10, 10)
(3, 8)
Curr
(5, 5)
(0, 0)
Legend
Class 0
Class 1


(10, 10)
(3, 8)
(5, 5)
(0, 0)
Legend
Class 0
Class 1

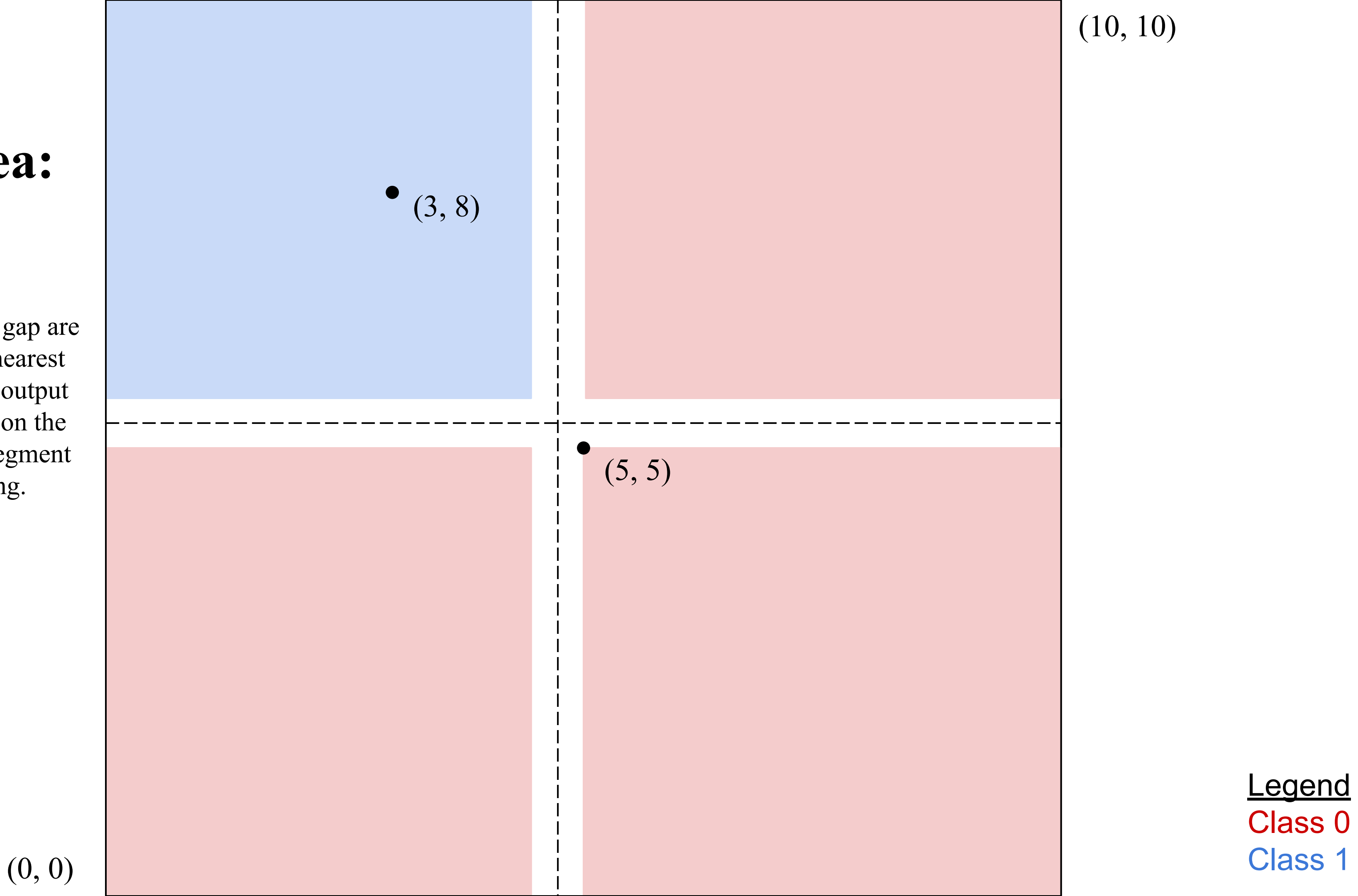
# How it's implemented:

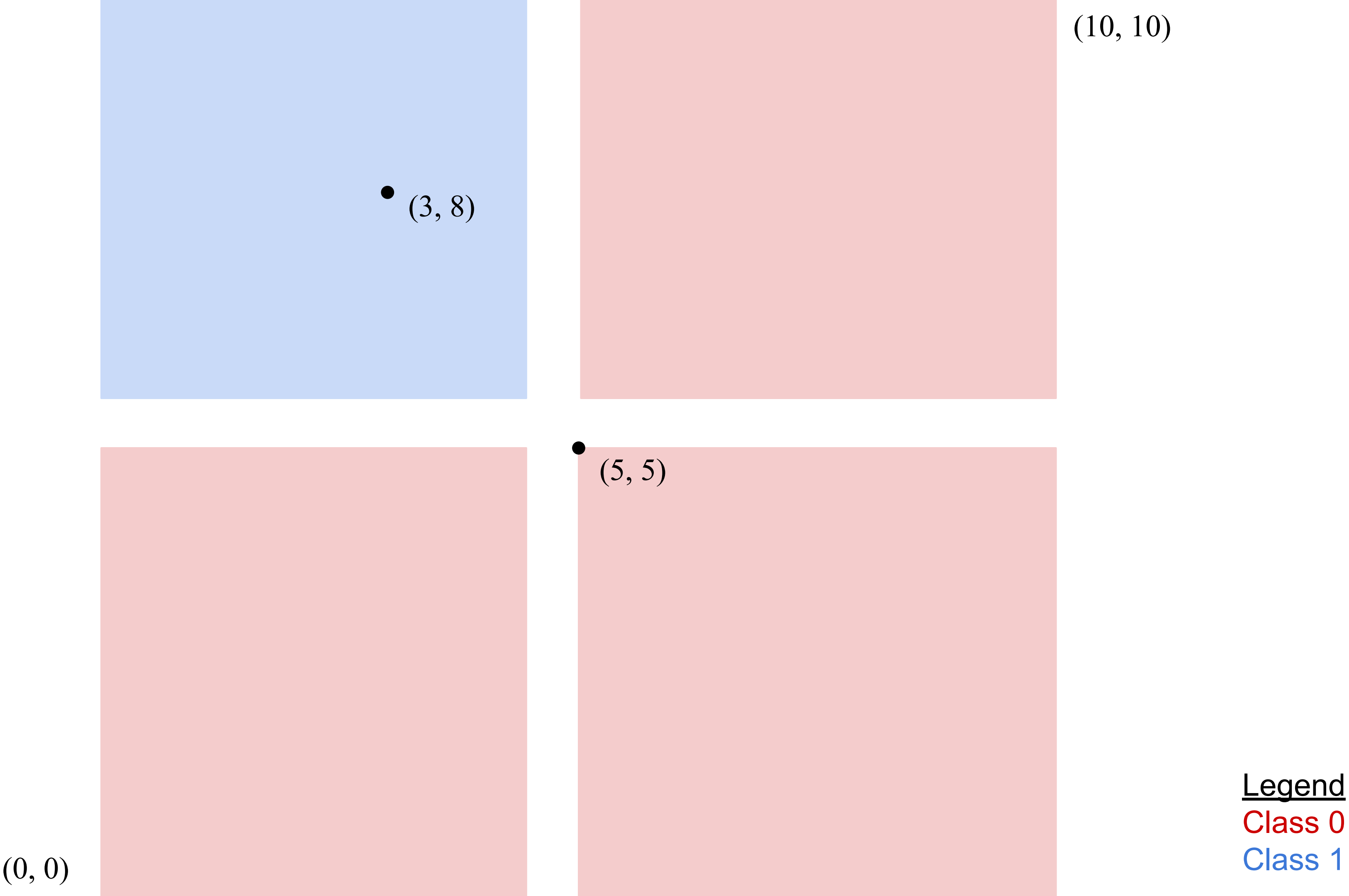
All gaps are $\epsilon$ wide to prevent adjacent segments from sharing inputs.



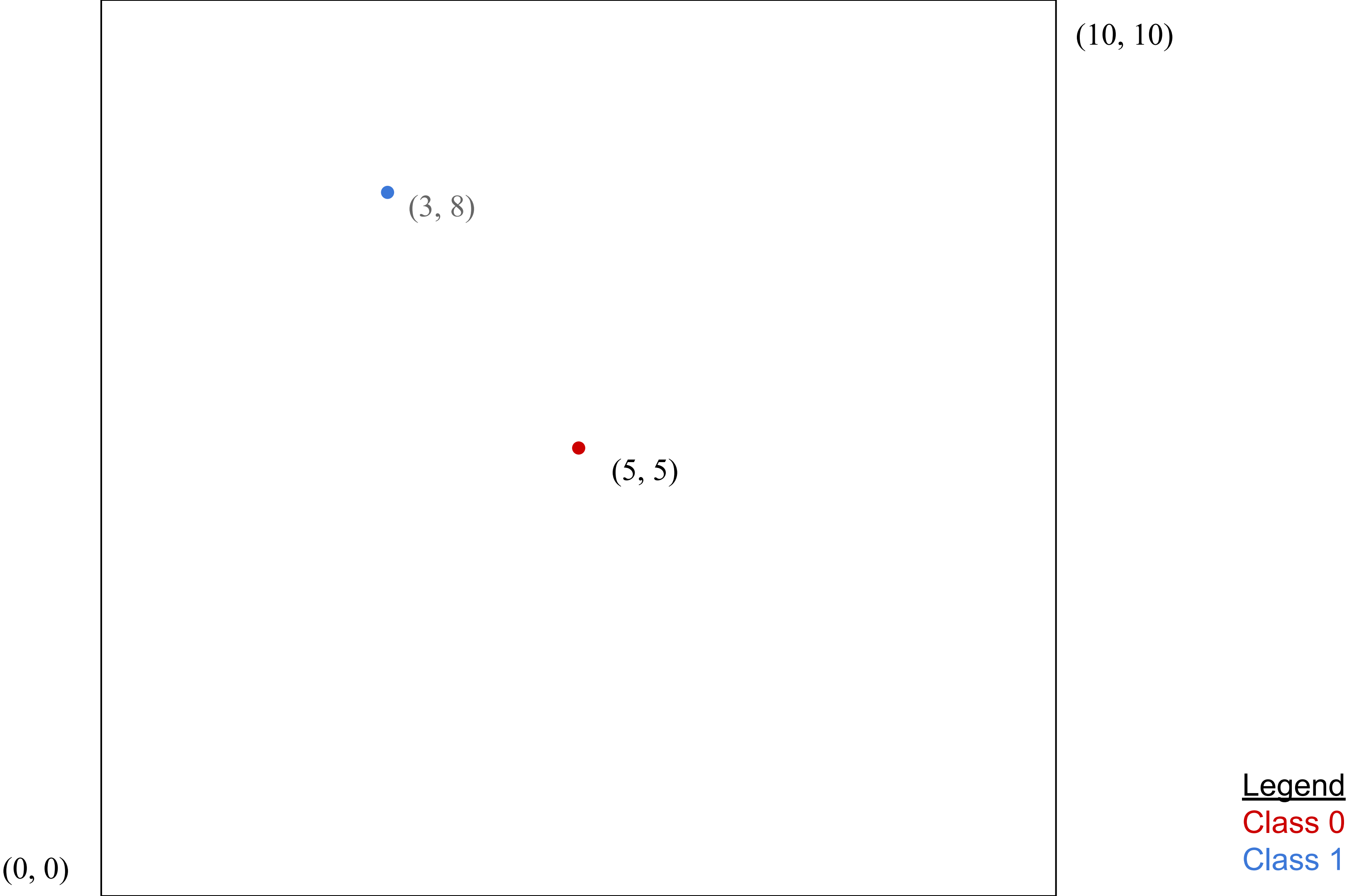
# The idea:

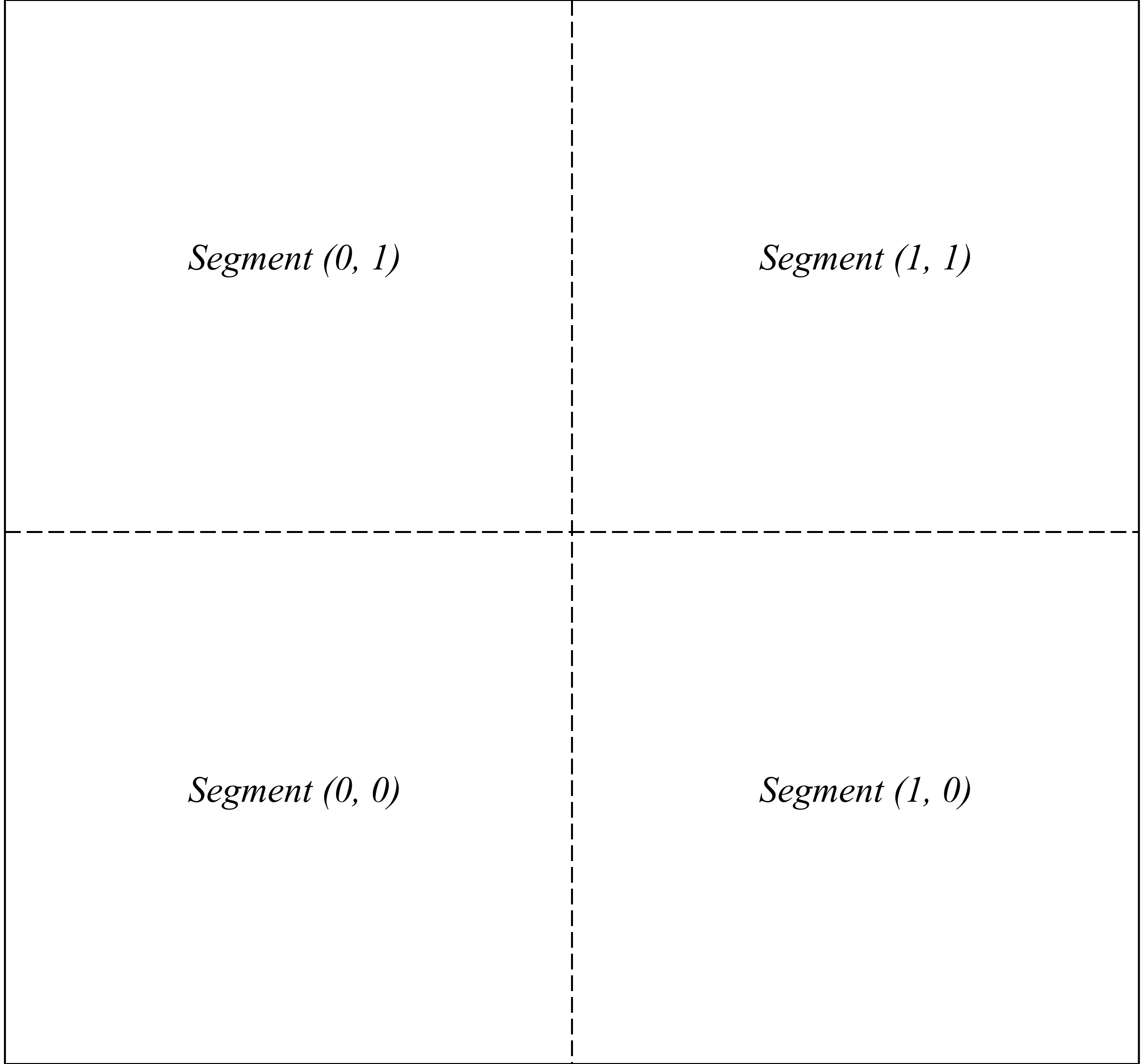
Inputs within the gap are rounded to the nearest epsilon, and the output would be based on the corresponding segment after rounding.

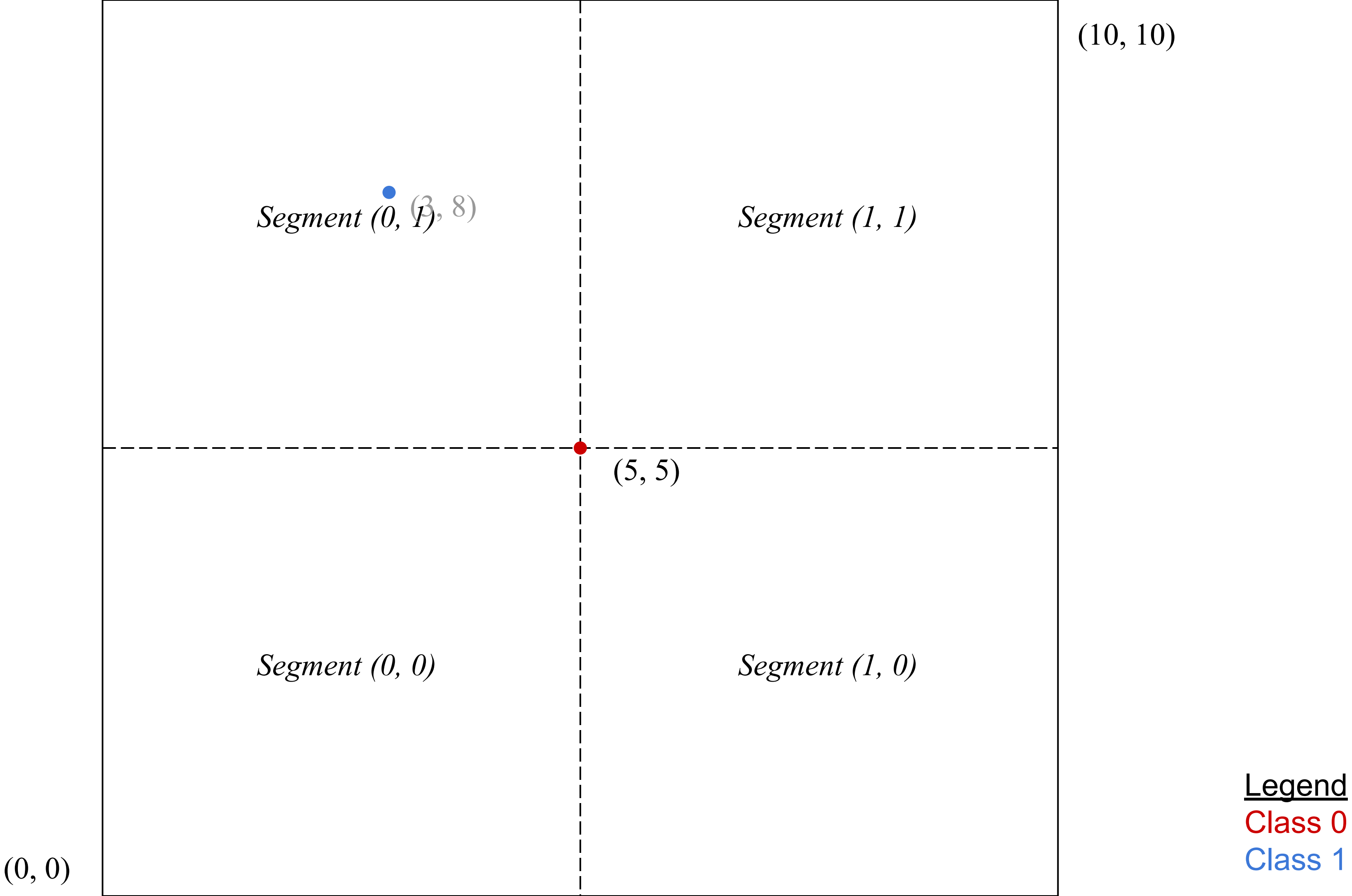




(10, 10)
(3, 8)
(5, 5)
(0, 0)
Legend
Class 0
Class 1


(10, 10)
(3, 8)
(5, 5)
(0, 0)
Legend
Class 0
Class 1


Segment (0, 1)
Segment (1, 1)
Segment (0, 0)
Segment (1, 0)



(10, 10)

*Segment (0, 1)*

(3, 8)

*Segment (1, 1)*

(5, 5)

*Segment (0, 0)*

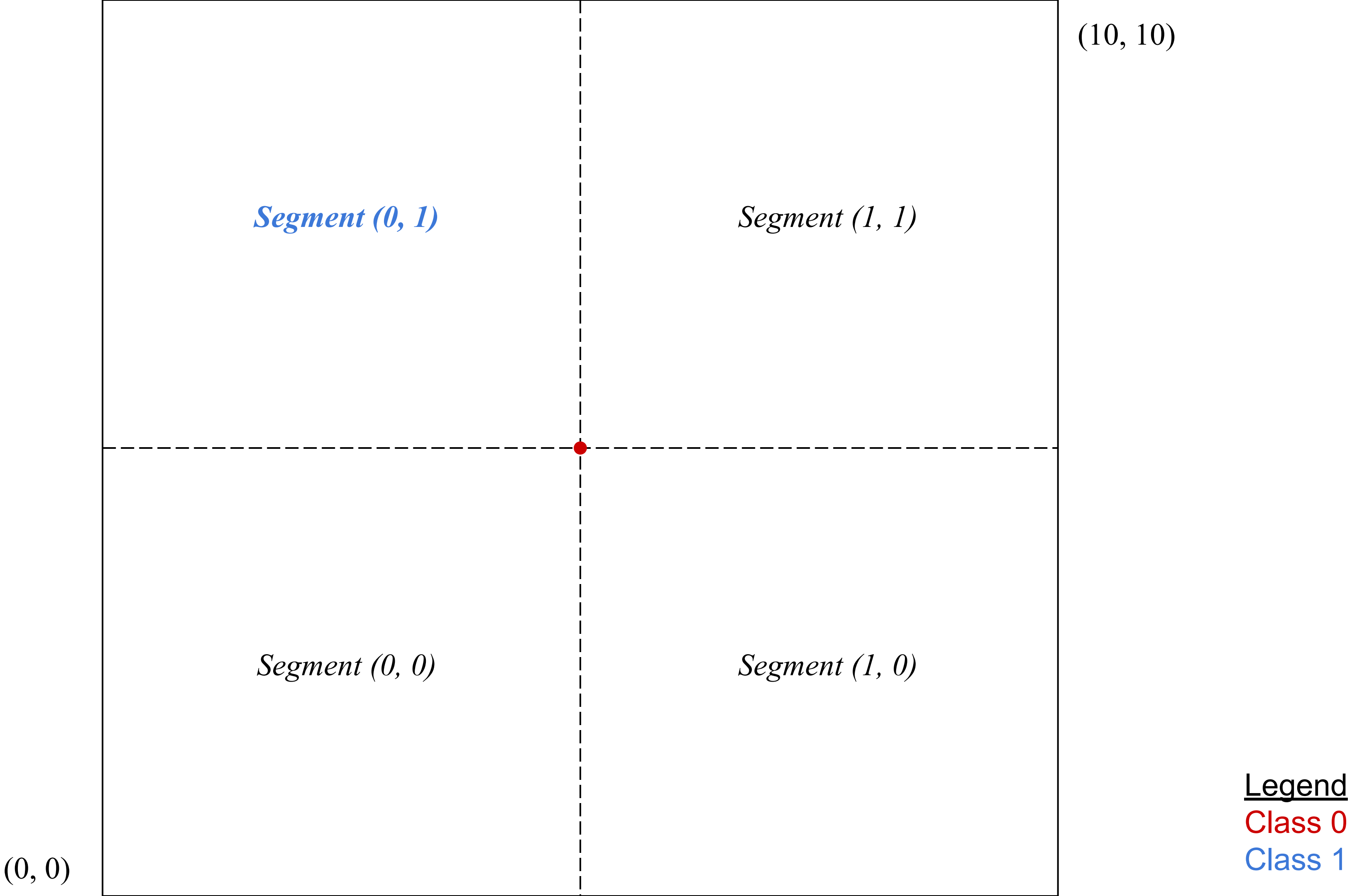
*Segment (1, 0)*

(0, 0)

Legend

Class 0

Class 1



(10, 10)

*Segment (0, 1)*

*Segment (1, 1)*

*Segment (0, 0)*

*Segment (1, 0)*

(0, 0)

Legend

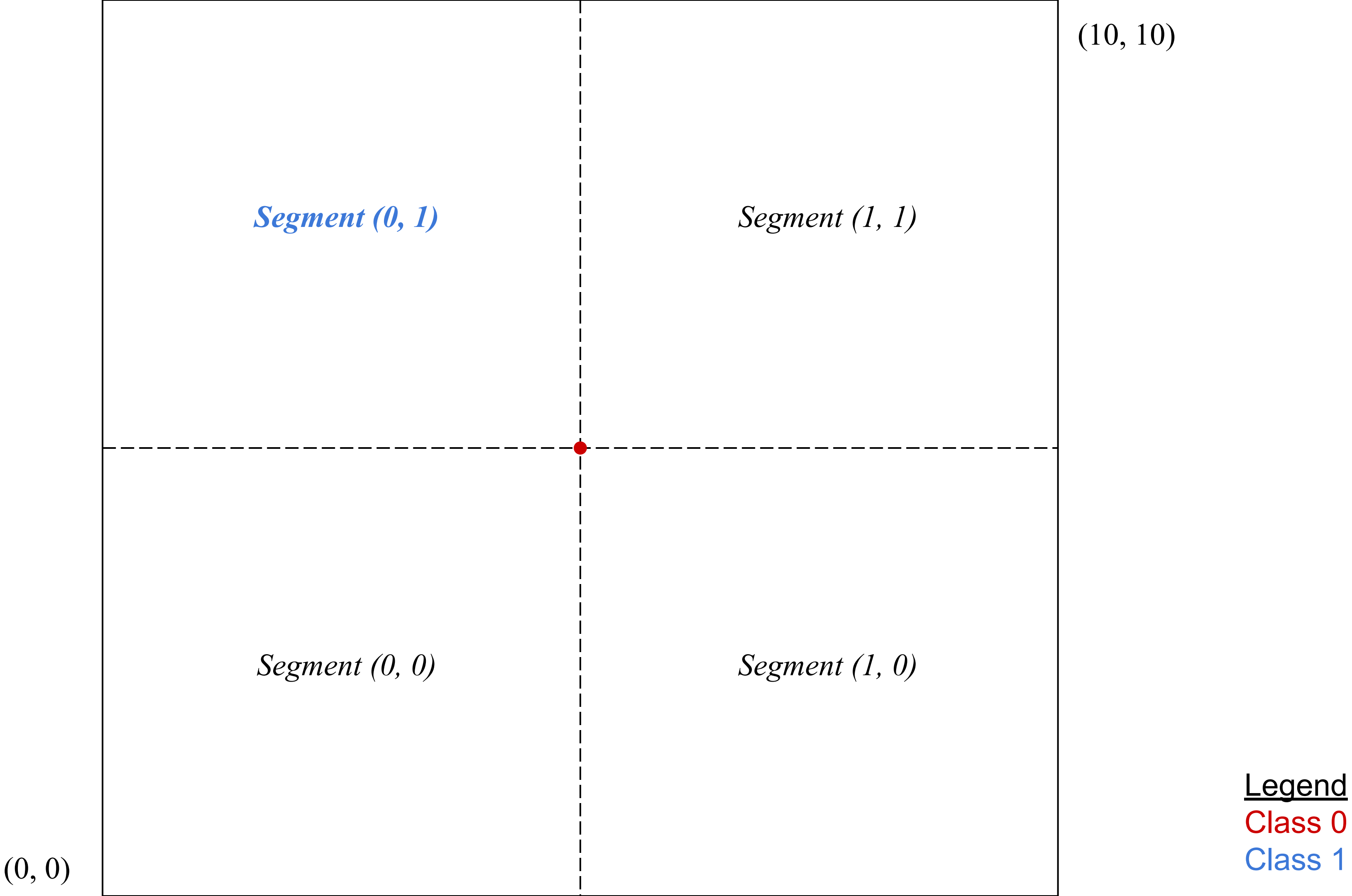
Class 0

Class 1

```
# Compute representation of deviation between new counterexample and previous counterexample (during set attempt) to later determine how to assign output values to split segments
split_idxs = []
prev_counterex_relative_segment = [] # 2D input case: [0, 0] is bottom-left, [0, 1] is top-left, [1, 0] is bottom-right, and [1, 1] is top-right
```

```python
# Compute representation of deviation between new counterexample and previous counterexample
(during set attempt) to later determine how to assign split segments' output value(s)
split_idxs = []
prev_counterex_relative_segment = [] # 2D input case: [0, 0] is bottom-left, [0, 1] is top-left,
[1, 0] is bottom-right, and [1, 1] is top-right
for key in inputVars:
    value = customRound(vals[key], awayFrom = prev_cntr_ex[key])
    split_idxs.append(value)
```

```python
# Compute representation of deviation between new counterexample and previous counterexample
(during set attempt) to later determine how to assign split segments' output value(s)
split_idxs = []
prev_counterex_relative_segment = [] # 2D input case: [0, 0] is bottom-left, [0, 1] is top-left,
[1, 0] is bottom-right, and [1, 1] is top-right
for key in inputVars:
    value = customRound(vals[key], awayFrom = prev_cntr_ex[key])
    split_idxs.append(value)

    diff = prev_cntr_ex[key] - value
    prev_counterex_relative_segment.append(int(diff > 0))
```

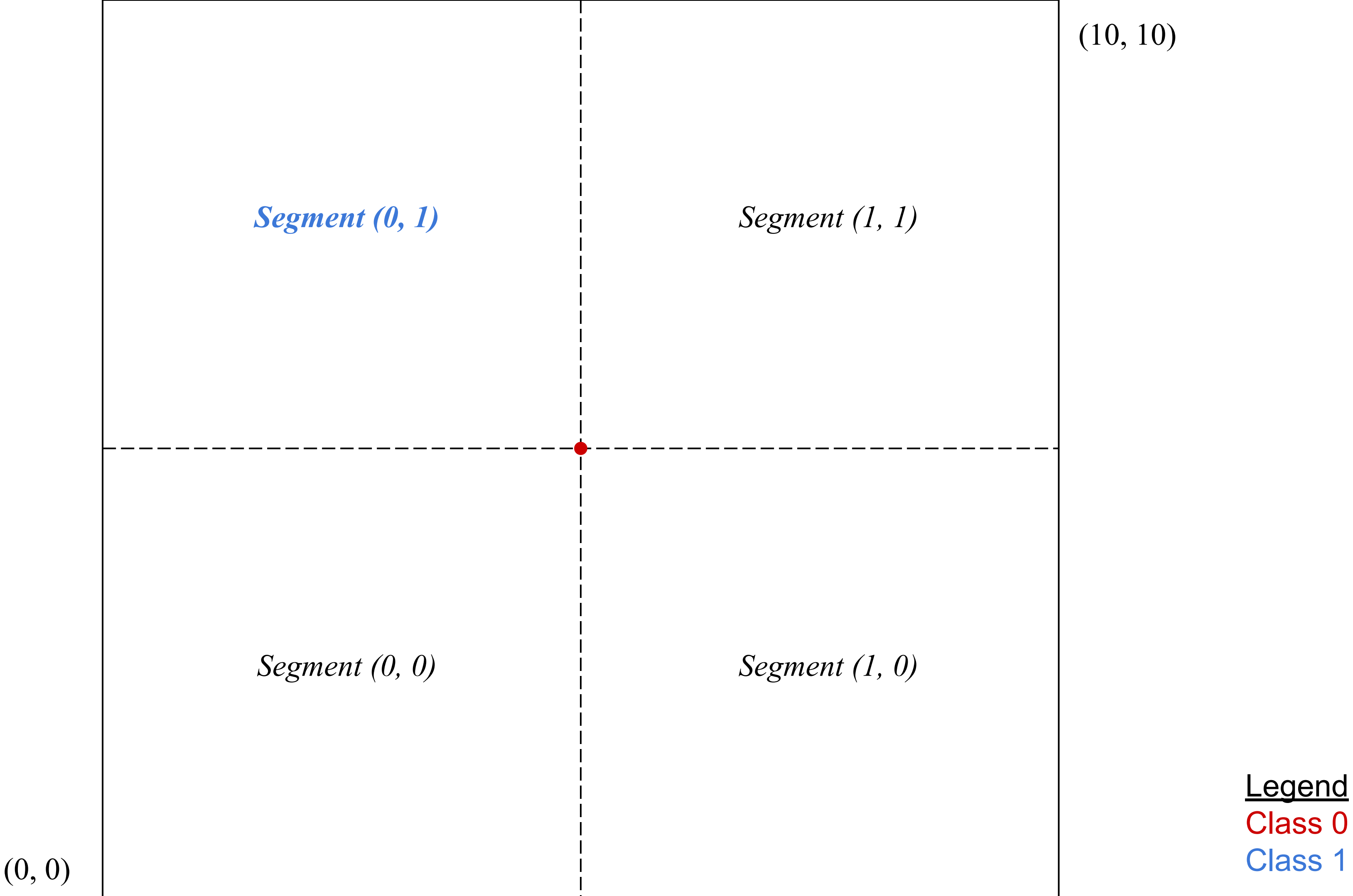

(10, 10)
Segment (0, 1)
Segment (1, 1)
Segment (0, 0)
Segment (1, 0)
(0, 0)
Legend
Class 0
Class 1

```
for segment in ([0, 0], [0, 1], [1, 0], [1, 1]):
```

```python
for segment in itertools.product(*[[0, 1] for idx in range(2)]):
```

```python
for segment in itertools.product(*[[0, 1] for idx in range(len(inputVars))]):
```

```
for segment in itertools.product(*[[0, 1] for idx in range(len(inputVars))]):
    bottom_left = []
    top_right = []

    for dim_idx, dim_val in enumerate(segment): # (0, 1), (1, 0) for segment [1, 0]
        # Iterate through each dimension of segment
```



(10, 10)

*Segment (0, 1)*

*Segment (1, 1)*

*Segment (0, 0)*

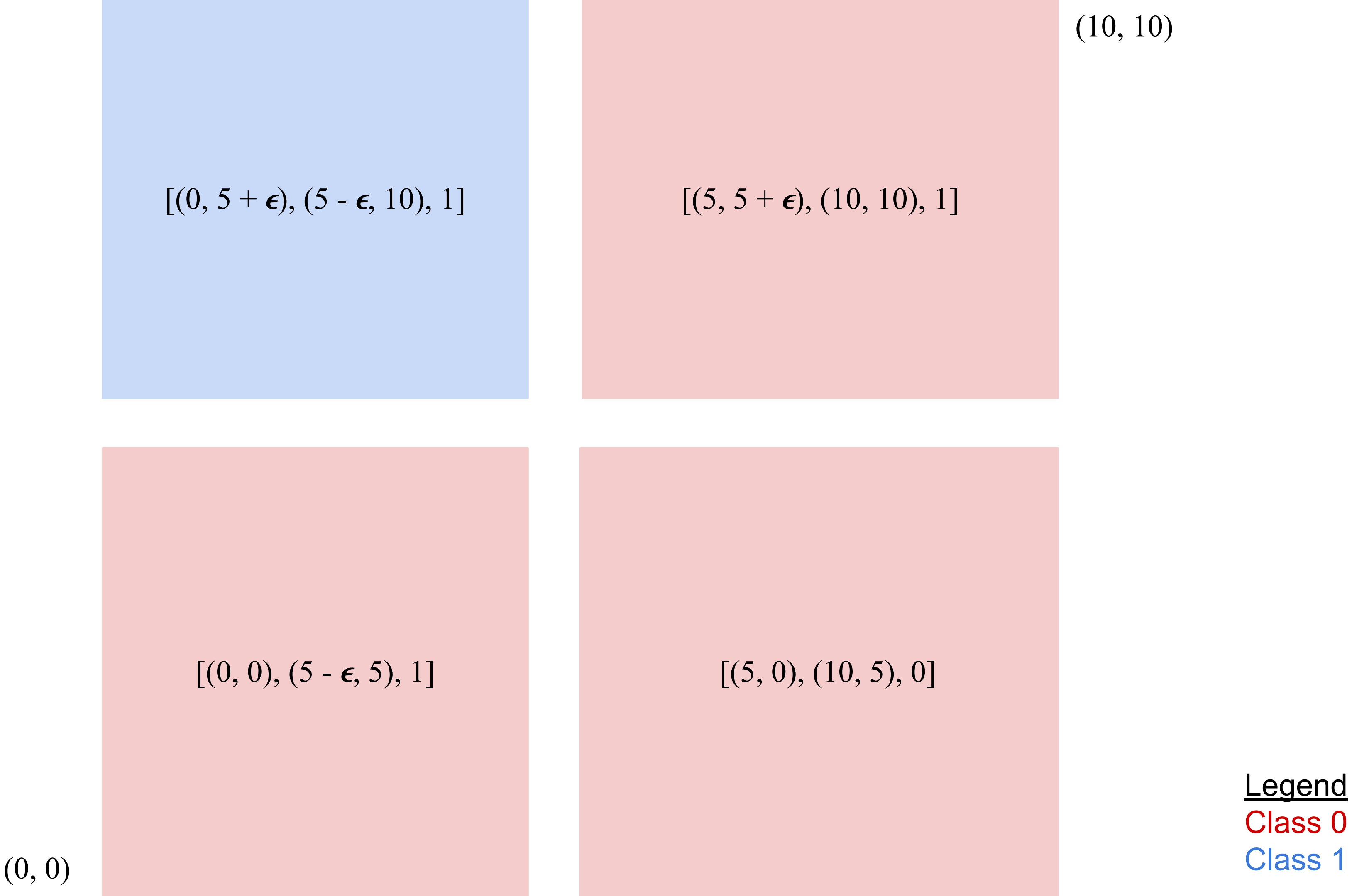
*Segment (1, 0)*

(0, 0)

Legend

Class 0

Class 1



(10, 10)

$[(0, 5 + \epsilon), (5 - \epsilon, 10), 1]$

$[(5, 5 + \epsilon), (10, 10), 1]$

$[(0, 0), (5 - \epsilon, 5), 1]$

$[(5, 0), (10, 5), 0]$

(0, 0)

Legend

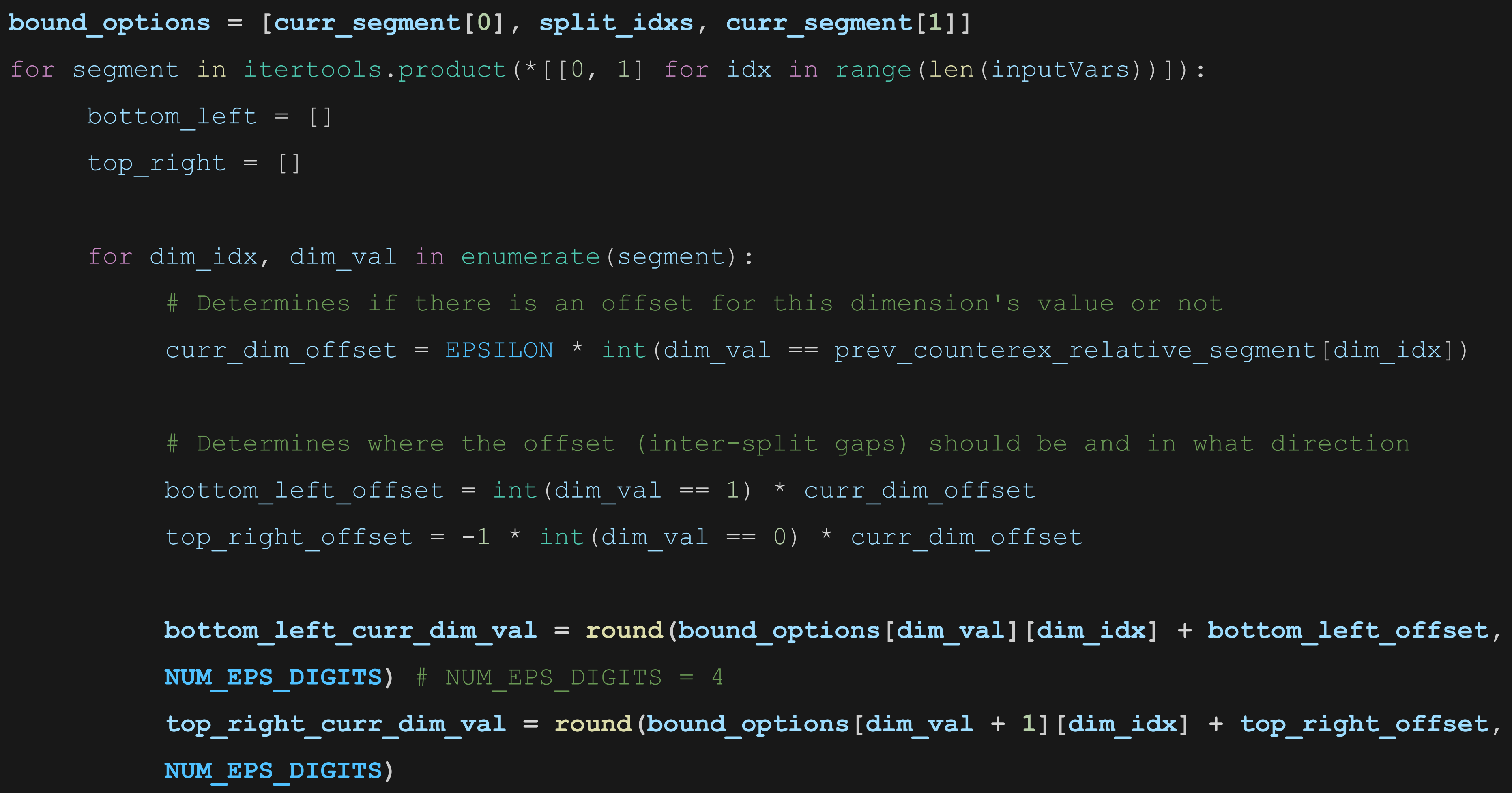
Class 0

Class 1

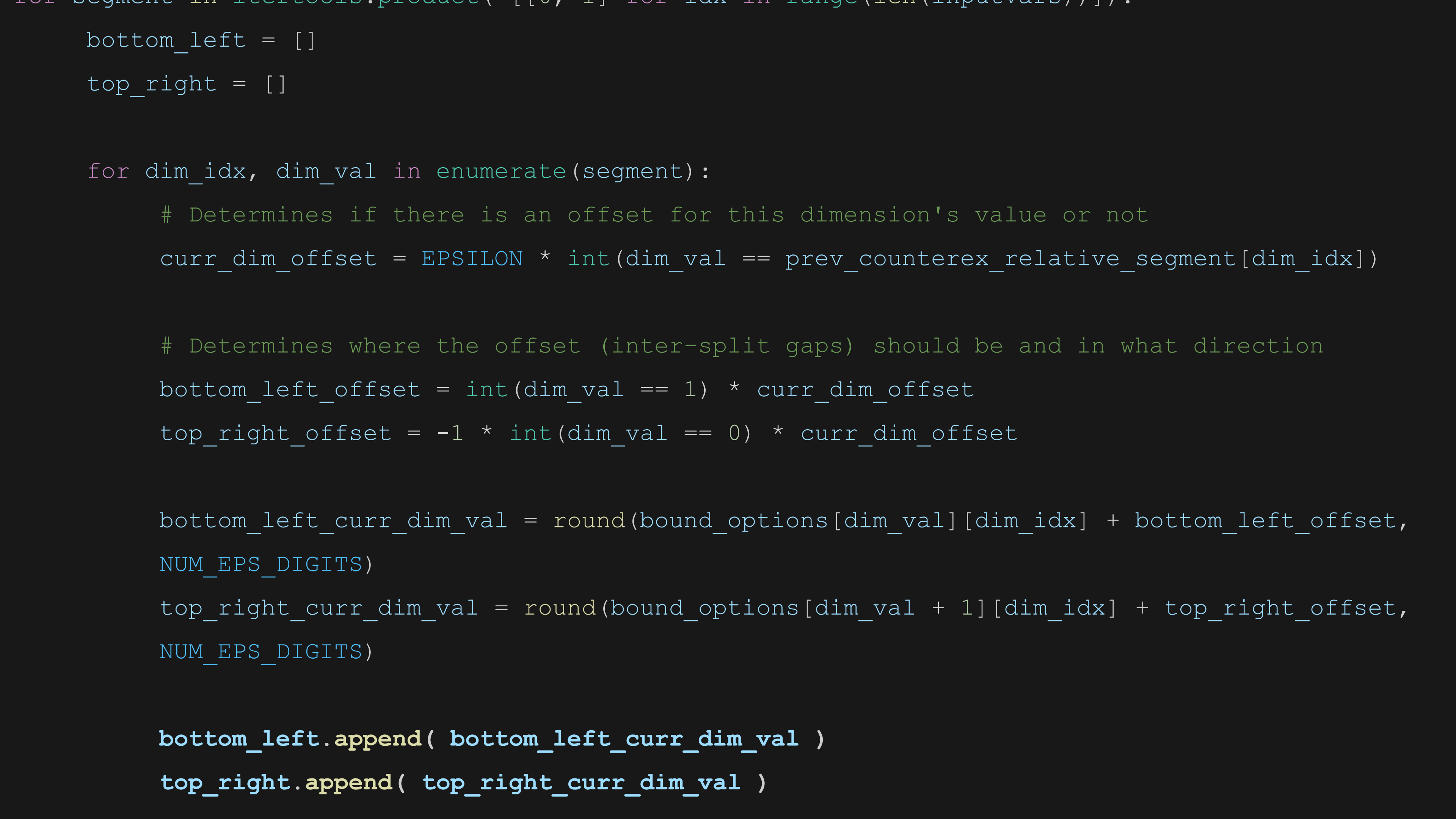
```
for segment in itertools.product(*[[0, 1] for idx in range(len(inputVars))]):
    bottom_left = []
    top_right = []

    for dim_idx, dim_val in enumerate(segment):
        # Determines if there is an offset for this dimension's value or not
        curr_dim_offset = EPSILON * int(dim_val == prev_counterex_relative_segment[dim_idx])
```

```python
for segment in itertools.product(*[[0, 1] for idx in range(len(inputVars))]):
    bottom_left = []
    top_right = []

    for dim_idx, dim_val in enumerate(segment):
        # Determines if there is an offset for this dimension's value or not
        curr_dim_offset = EPSILON * int(dim_val == prev_counterex_relative_segment[dim_idx])

        # Determines where the offset (inter-split gaps) should be and in what direction
        bottom_left_offset = int(dim_val == 1) * curr_dim_offset
        top_right_offset = -1 * int(dim_val == 0) * curr_dim_offset
```

```
bound_options = [curr_segment[0], split_idxs, curr_segment[1]]
for segment in itertools.product(*[[0, 1] for idx in range(len(inputVars))]):
    bottom_left = []
    top_right = []

    for dim_idx, dim_val in enumerate(segment):
        # Determines if there is an offset for this dimension's value or not
        curr_dim_offset = EPSILON * int(dim_val == prev_counterex_relative_segment[dim_idx])

        # Determines where the offset (inter-split gaps) should be and in what direction
        bottom_left_offset = int(dim_val == 1) * curr_dim_offset
        top_right_offset = -1 * int(dim_val == 0) * curr_dim_offset

        bottom_left_curr_dim_val = bound_options[dim_val][dim_idx] + bottom_left_offset
        top_right_curr_dim_val = bound_options[dim_val + 1][dim_idx] + top_right_offset
```

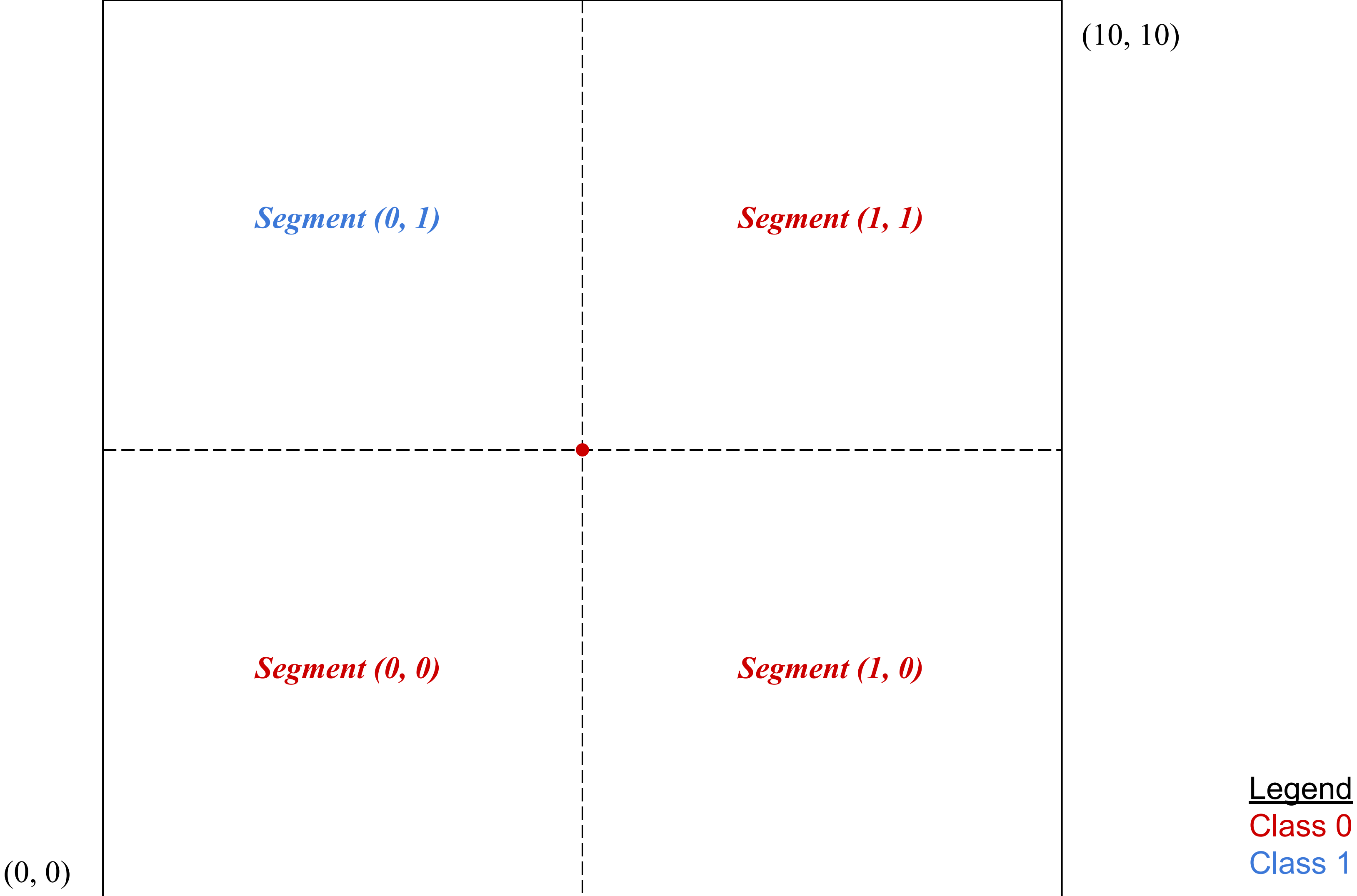


```
bound_options = [curr_segment[0], split_idxs, curr_segment[1]]
for segment in itertools.product(*[[0, 1] for idx in range(len(inputVars))]):
    bottom_left = []
    top_right = []

    for dim_idx, dim_val in enumerate(segment):
        # Determines if there is an offset for this dimension's value or not
        curr_dim_offset = EPSILON * int(dim_val == prev_counterex_relative_segment[dim_idx])

        # Determines where the offset (inter-split gaps) should be and in what direction
        bottom_left_offset = int(dim_val == 1) * curr_dim_offset
        top_right_offset = -1 * int(dim_val == 0) * curr_dim_offset

        bottom_left_curr_dim_val = round(bound_options[dim_val][dim_idx] + bottom_left_offset,
        NUM_EPS_DIGITS) # NUM_EPS_DIGITS = 4
        top_right_curr_dim_val = round(bound_options[dim_val + 1][dim_idx] + top_right_offset,
        NUM_EPS_DIGITS)
```



```
for segment in itertools.product( [[0, 1] for idx in range(len(inputvars))]):
    bottom_left = []
    top_right = []

    for dim_idx, dim_val in enumerate(segment):
        # Determines if there is an offset for this dimension's value or not
        curr_dim_offset = EPSILON * int(dim_val == prev_counterex_relative_segment[dim_idx])

        # Determines where the offset (inter-split gaps) should be and in what direction
        bottom_left_offset = int(dim_val == 1) * curr_dim_offset
        top_right_offset = -1 * int(dim_val == 0) * curr_dim_offset

        bottom_left_curr_dim_val = round(bound_options[dim_val][dim_idx] + bottom_left_offset,
        NUM_EPS_DIGITS)
        top_right_curr_dim_val = round(bound_options[dim_val + 1][dim_idx] + top_right_offset,
        NUM_EPS_DIGITS)

        bottom_left.append( bottom_left_curr_dim_val )
        top_right.append( top_right_curr_dim_val )
```

```
for segment in itertools.product(*[[0, 1] for idx in range(len(inputVars))]):
     bottom_left = []
     top_right = []

     for dim_idx, dim_val in enumerate(segment):
          # Iterate through each dimension of segment

     usePrevCounterExOutput = (list(segment) == prev_counterex_relative_segment)
     output = int(prev_cntr_ex[outputVarIdx] > 0) if usePrevCounterExOutput
               else int(prev_cntr_ex[outputVarIdx] <= 0)
```



(10, 10)

*Segment (0, 1)*

*Segment (1, 1)*

*Segment (0, 0)*

*Segment (1, 0)*

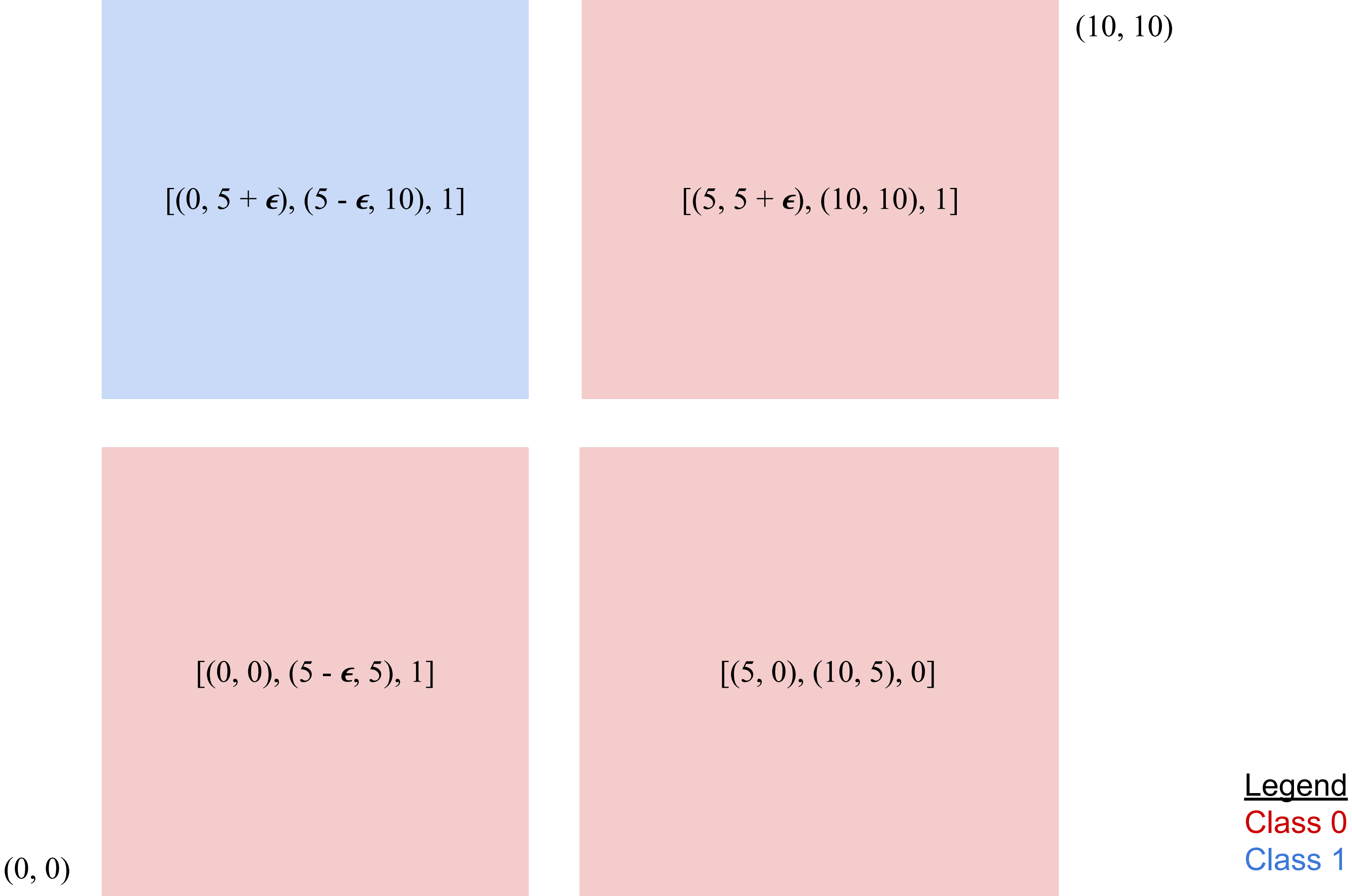
(0, 0)

Legend

Class 0

Class 1

```
for segment in itertools.product(*[[0, 1] for idx in range(len(inputVars))]):
     bottom_left = []
     top_right = []

     for dim_idx, dim_val in enumerate(segment):
          # Iterate through each dimension of segment

output = ...
new_segment = [bottom_left, top_right, output]
stack.append(new_segment) # Keep track of split segment
```



(10, 10)

[(0, 5 + $\epsilon$), (5 - $\epsilon$, 10), 1]

[(5, 5 + $\epsilon$), (10, 10), 1]

[(0, 0), (5 - $\epsilon$, 5), 1]

[(5, 0), (10, 5), 0]

(0, 0)

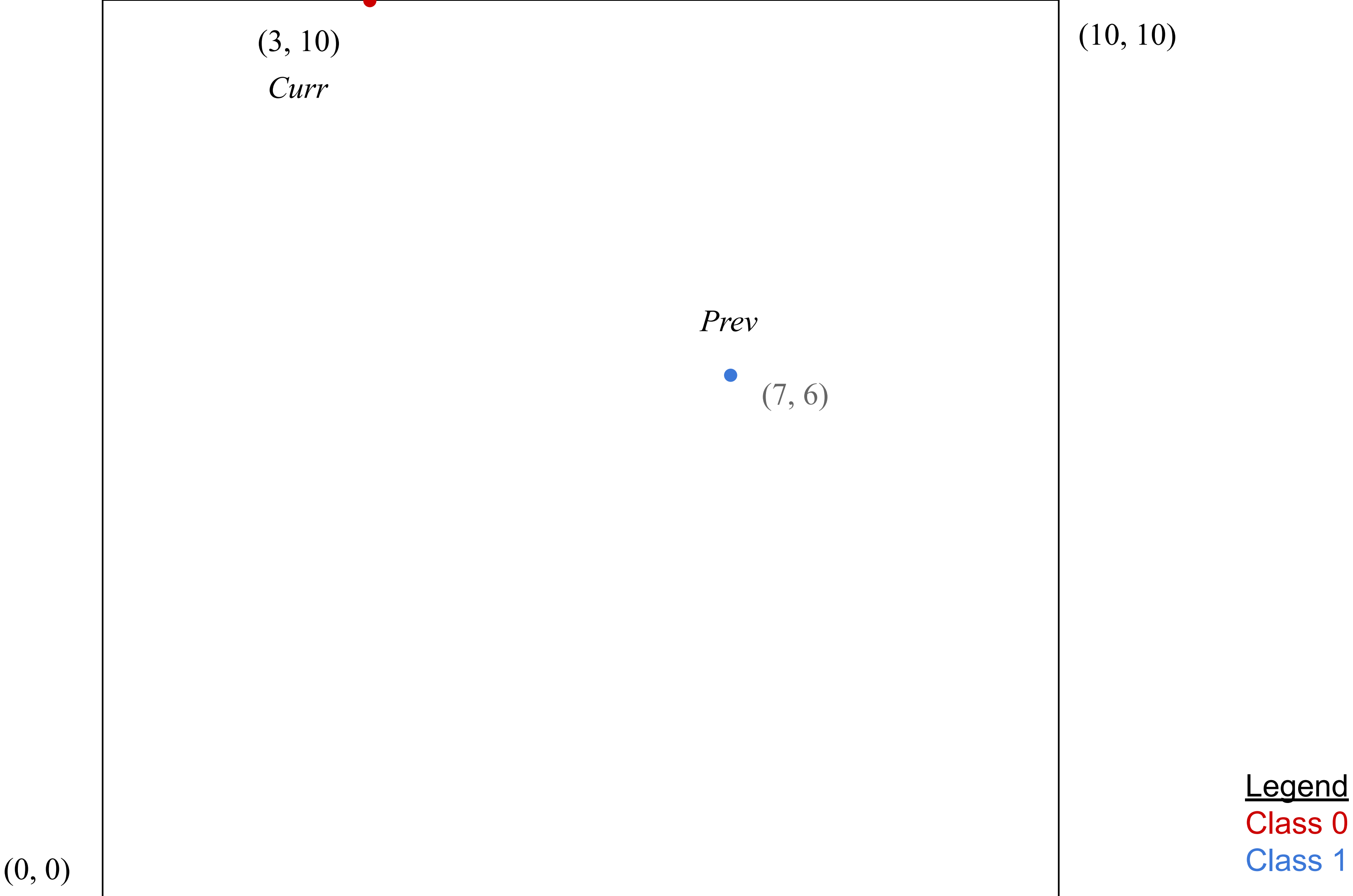
Legend

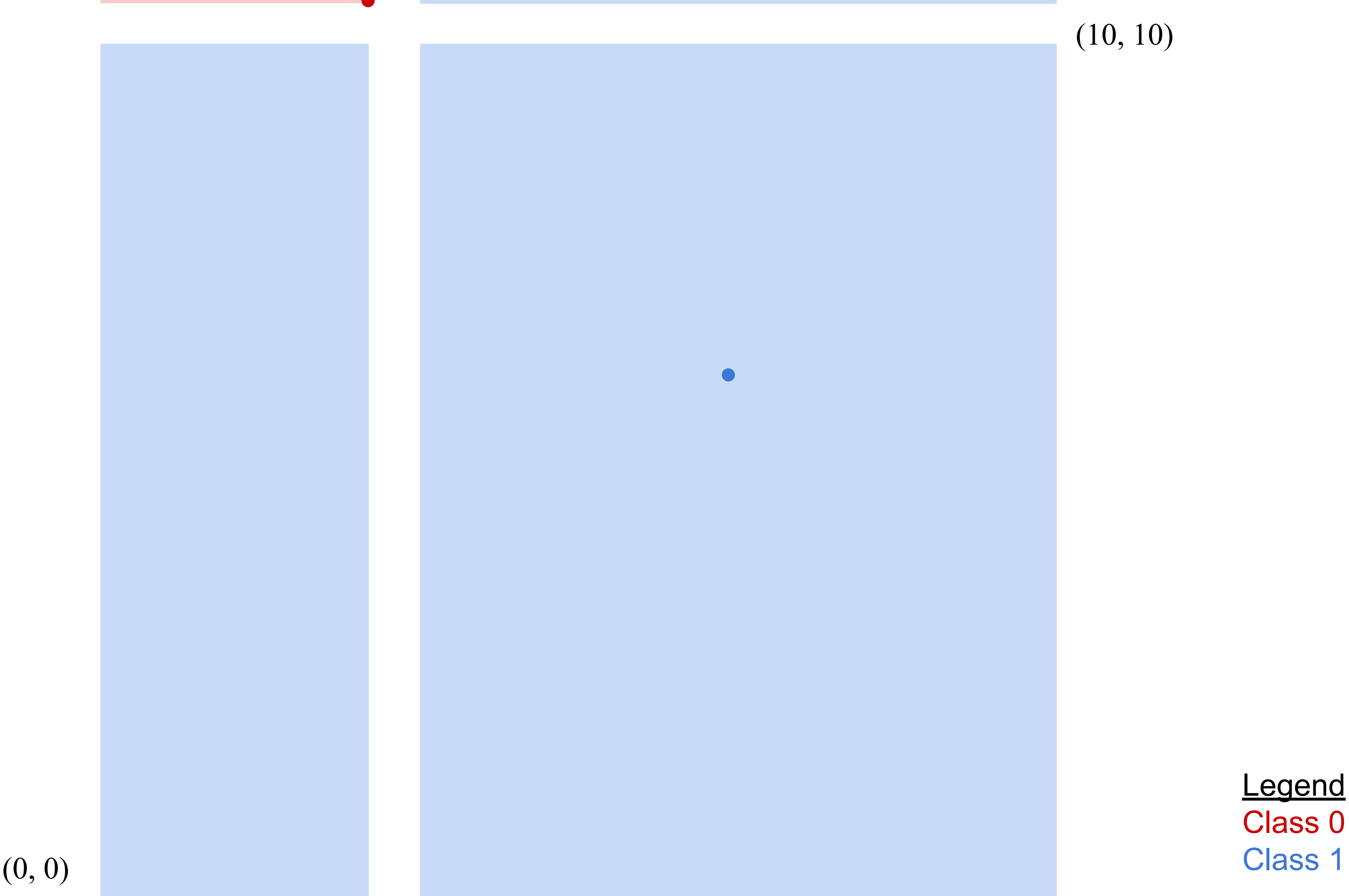
Class 0

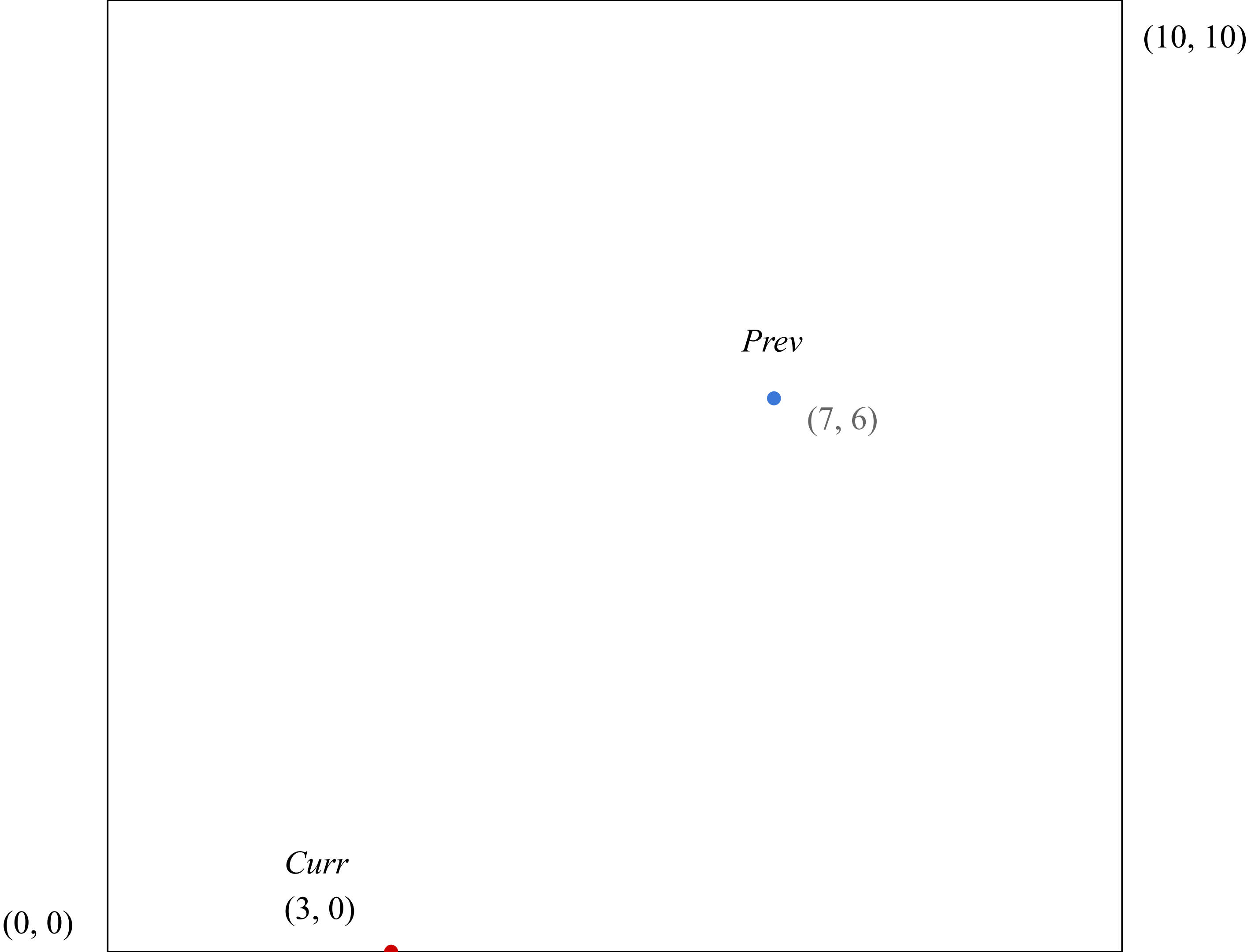
Class 1

# Edge Case:
# *Curr* on edge, *Prev* inside

**Segment Splitting Logic**


(3, 10)
*Curr*
(10, 10)
*Prev*
(7, 6)
(0, 0)
Legend
Class 0
Class 1

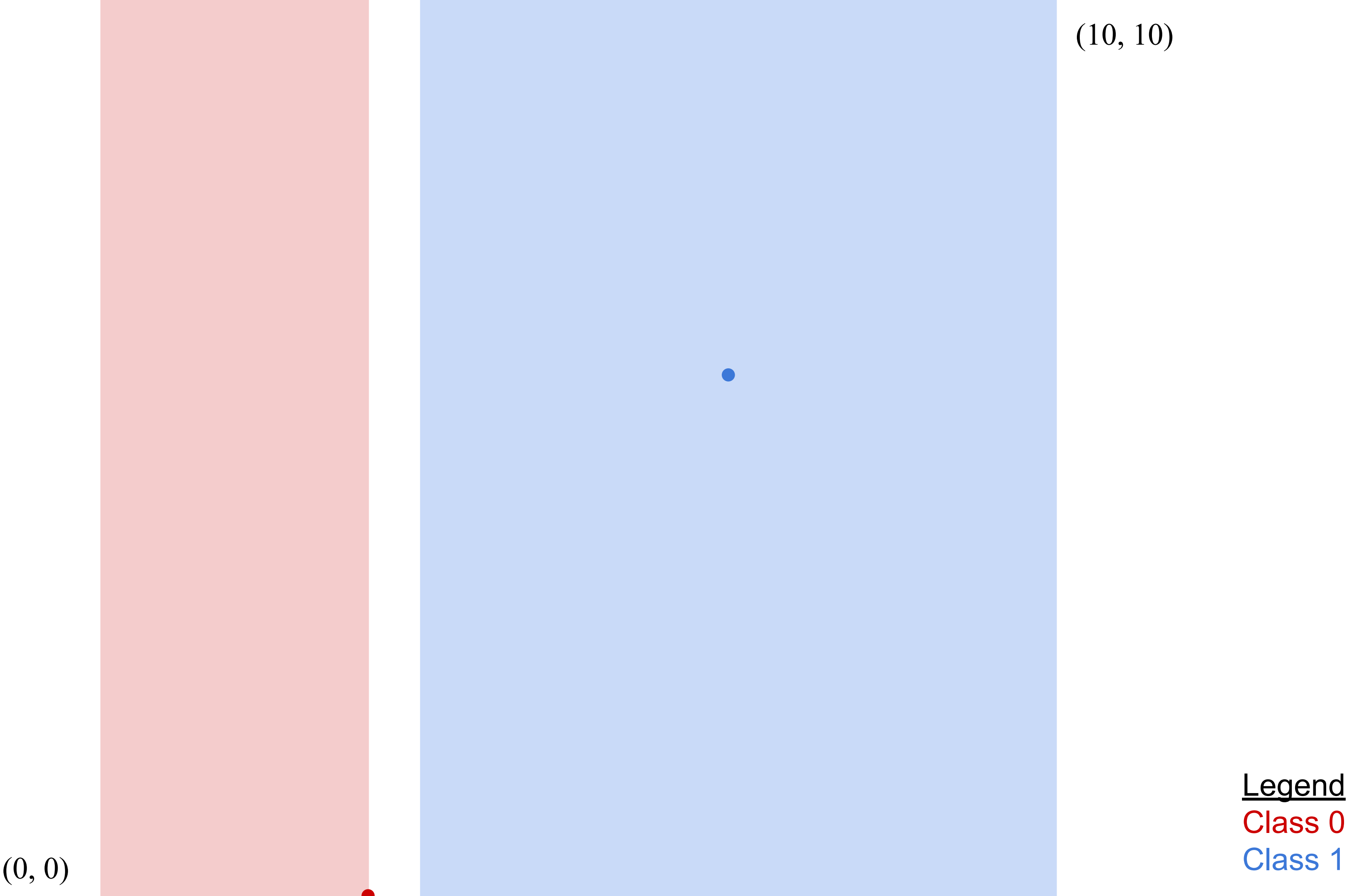

(10, 10)
(0, 0)
Legend
Class 0
Class 1



Legend

Class 0

Class 1



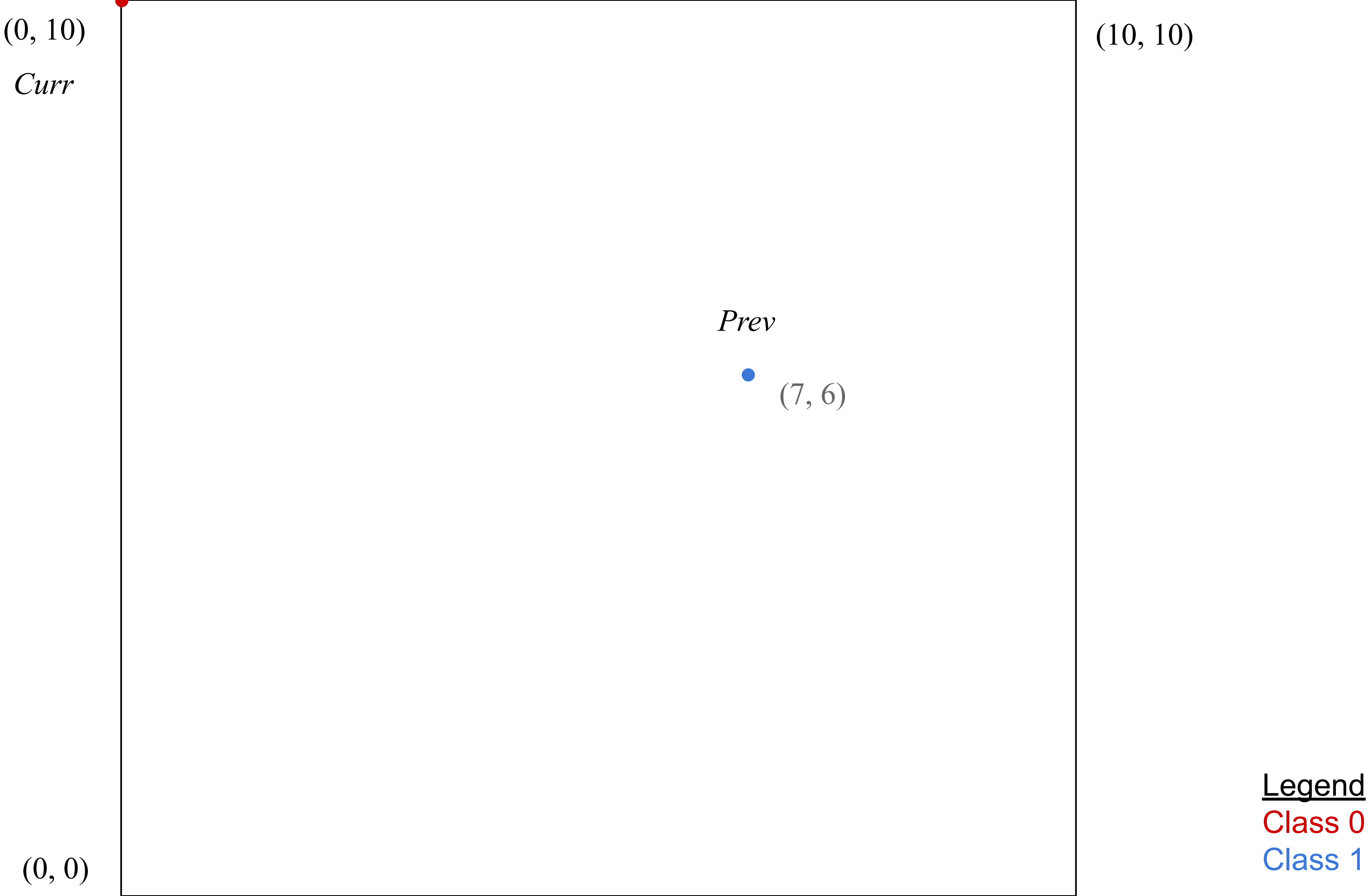
(10, 10)
(0, 0)
Legend
Class 0
Class 1

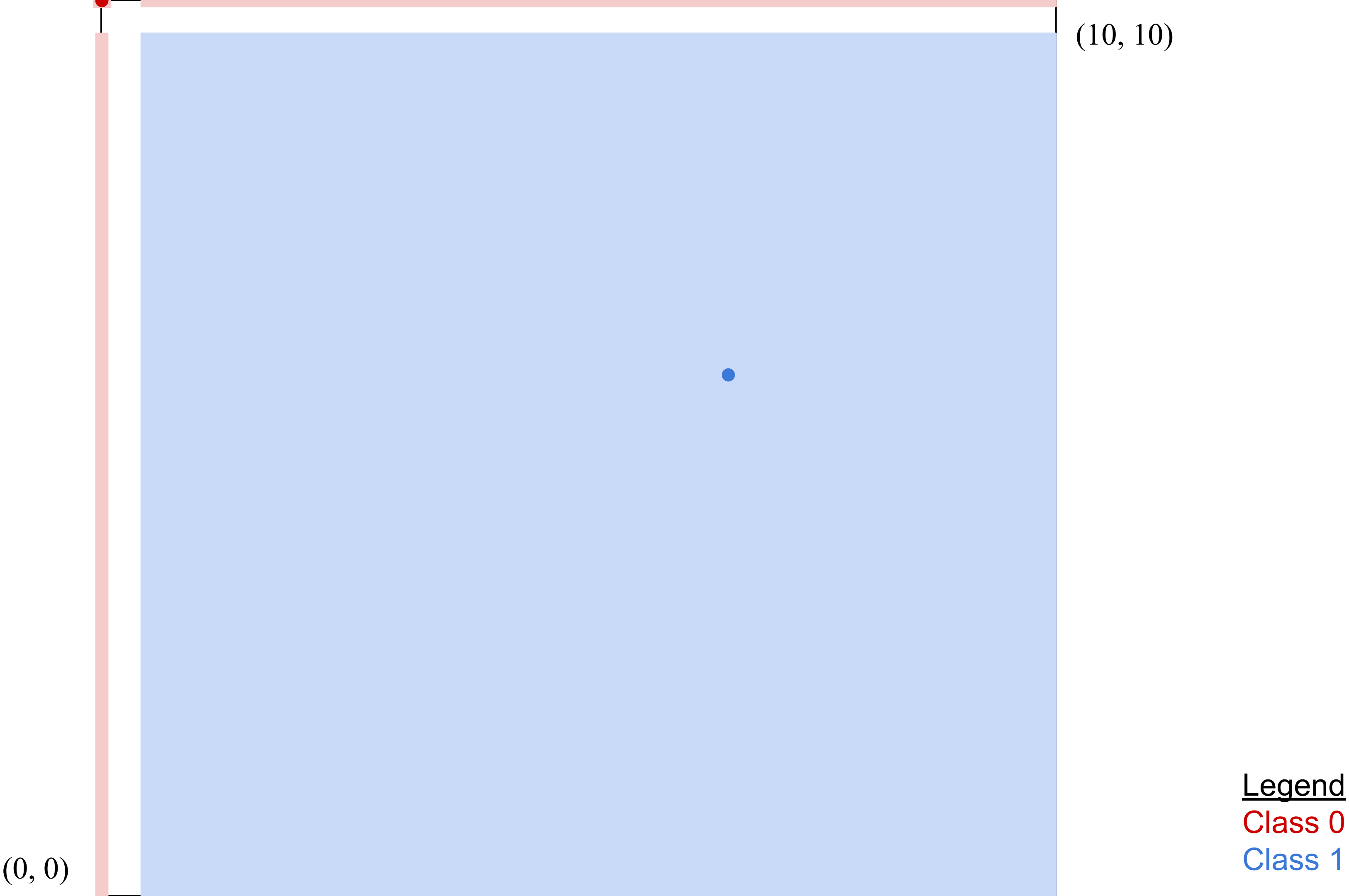
```
for segment in itertools.product(*[[0, 1] for idx in range(len(inputVars))]):
    bottom_left = []
    top_right = []
    skipSegment = False

    for dim_idx, dim_val in enumerate(segment): # (0, 1), (1, 0) for segment [1, 0]
        # Iterate through each dimension of segment

    if skipSegment:
        continue

    output = int( (prev_cntr_ex[outputVarIdx] > 0) ^ isNotPrevCounterExSegment )
    new_segment = [bottom_left, top_right, output]
    stack.append(new_segment) # Keep track of split segment
```

```
for dim_idx, dim_val in enumerate(segment):
    # Determines if there is an offset for this dimension's value or not
    curr_dim_offset = EPSILON * int(dim_val == prev_counterex_relative_segment[dim_idx])

    # Determines where the offset (inter-split gaps) should be and in what direction
    bottom_left_offset = int(dim_val == 1) * curr_dim_offset
    top_right_offset = -1 * int(dim_val == 0) * curr_dim_offset

    bottom_left_curr_dim_val = round(bound_options[dim_val][dim_idx] + bottom_left_offset,
    NUM_EPS_DIGITS)
    top_right_curr_dim_val = round(bound_options[dim_val + 1][dim_idx] + top_right_offset,
    NUM_EPS_DIGITS)

    if bottom_left_curr_dim_val > bound_options[2][dim_idx] \
        or bottom_left_curr_dim_val < bound_options[0][dim_idx] \
        or top_right_curr_dim_val > bound_options[2][dim_idx] \
        or top_right_curr_dim_val < bound_options[0][dim_idx]:
        skipSegment = True
        break
```

# Edge Case:
# *Curr* on corner, *Prev* inside

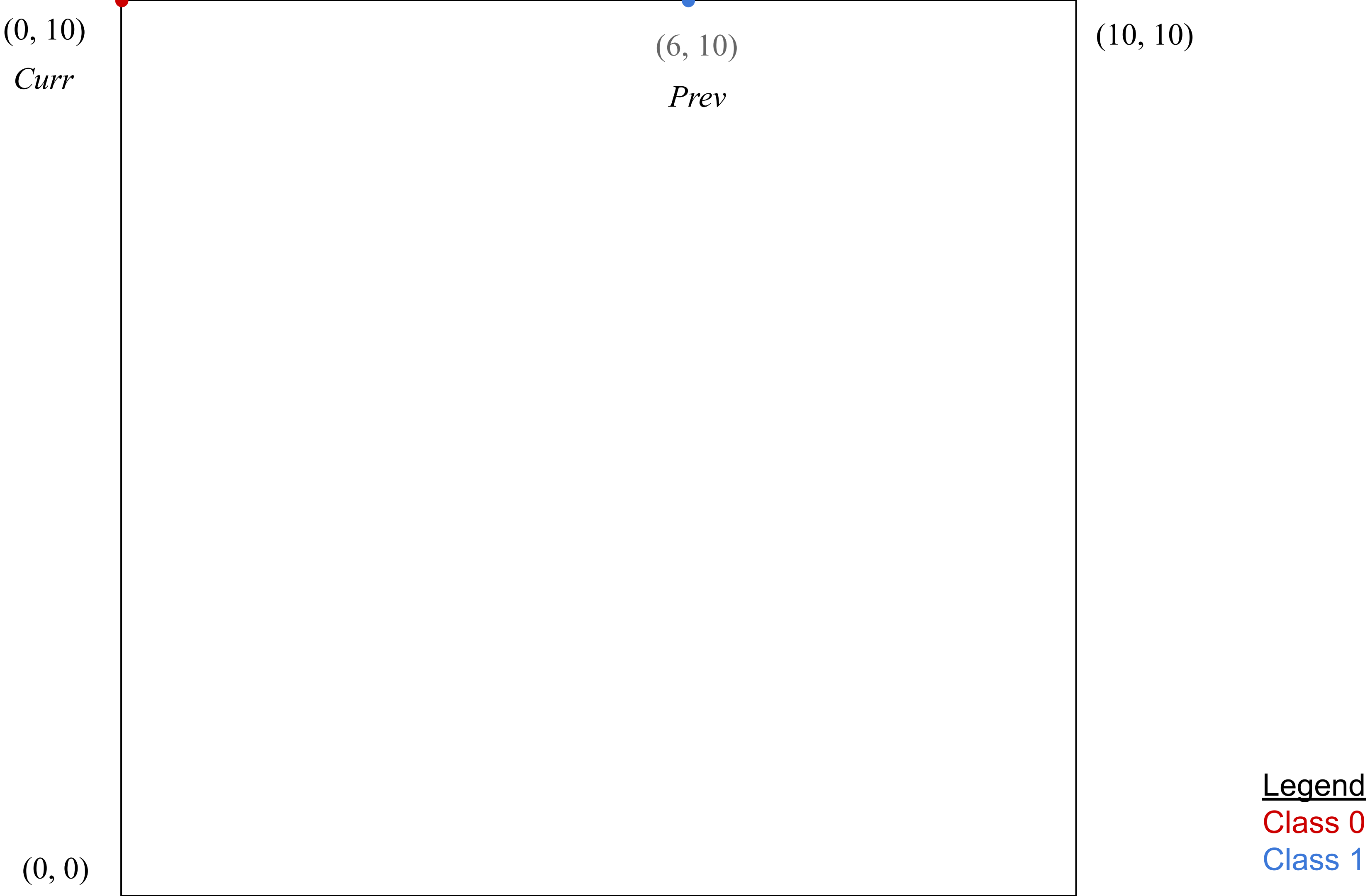
**Segment Splitting Logic**

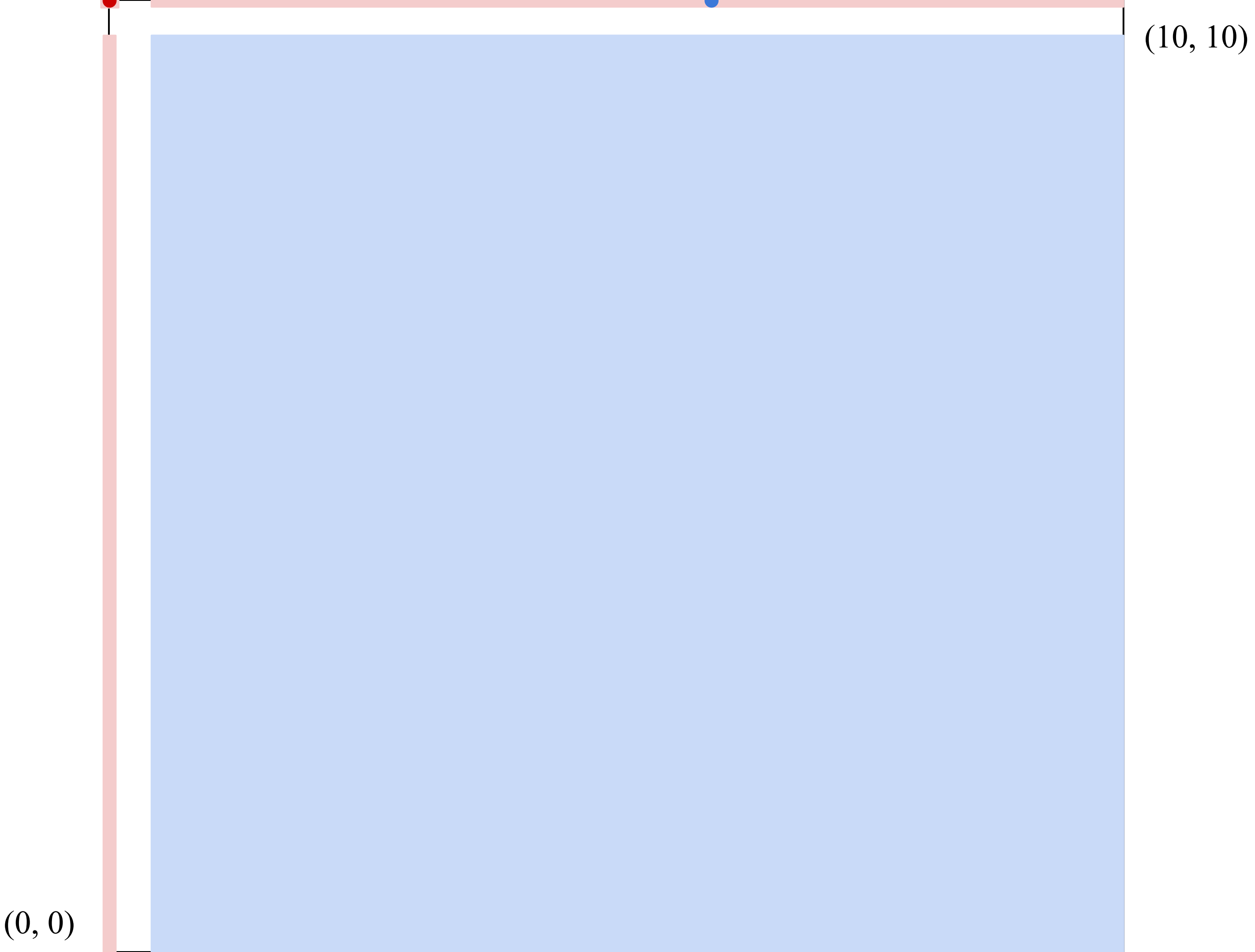

(0, 10)
Curr
(10, 10)
Prev
(7, 6)
(0, 0)
Legend
Class 0
Class 1


(10, 10)
(0, 0)
Legend
Class 0
Class 1

# Edge Case:
# *Curr* on corner, *Prev* on edge

**Segment Splitting Logic**


(0, 10)
*Curr*
(6, 10)
*Prev*
(10, 10)
(0, 0)
Legend
Class 0
Class 1



Legend

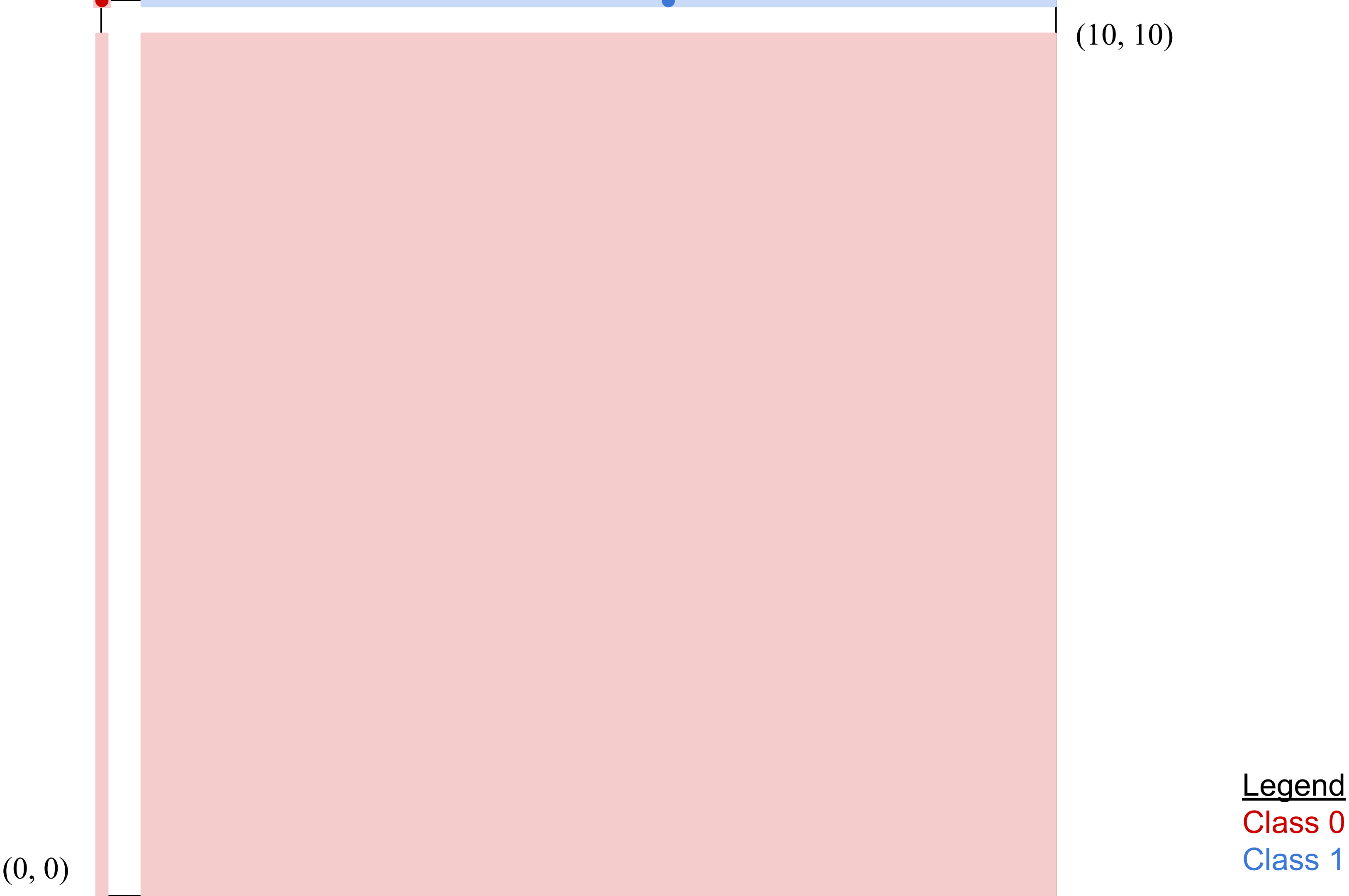
Class 0

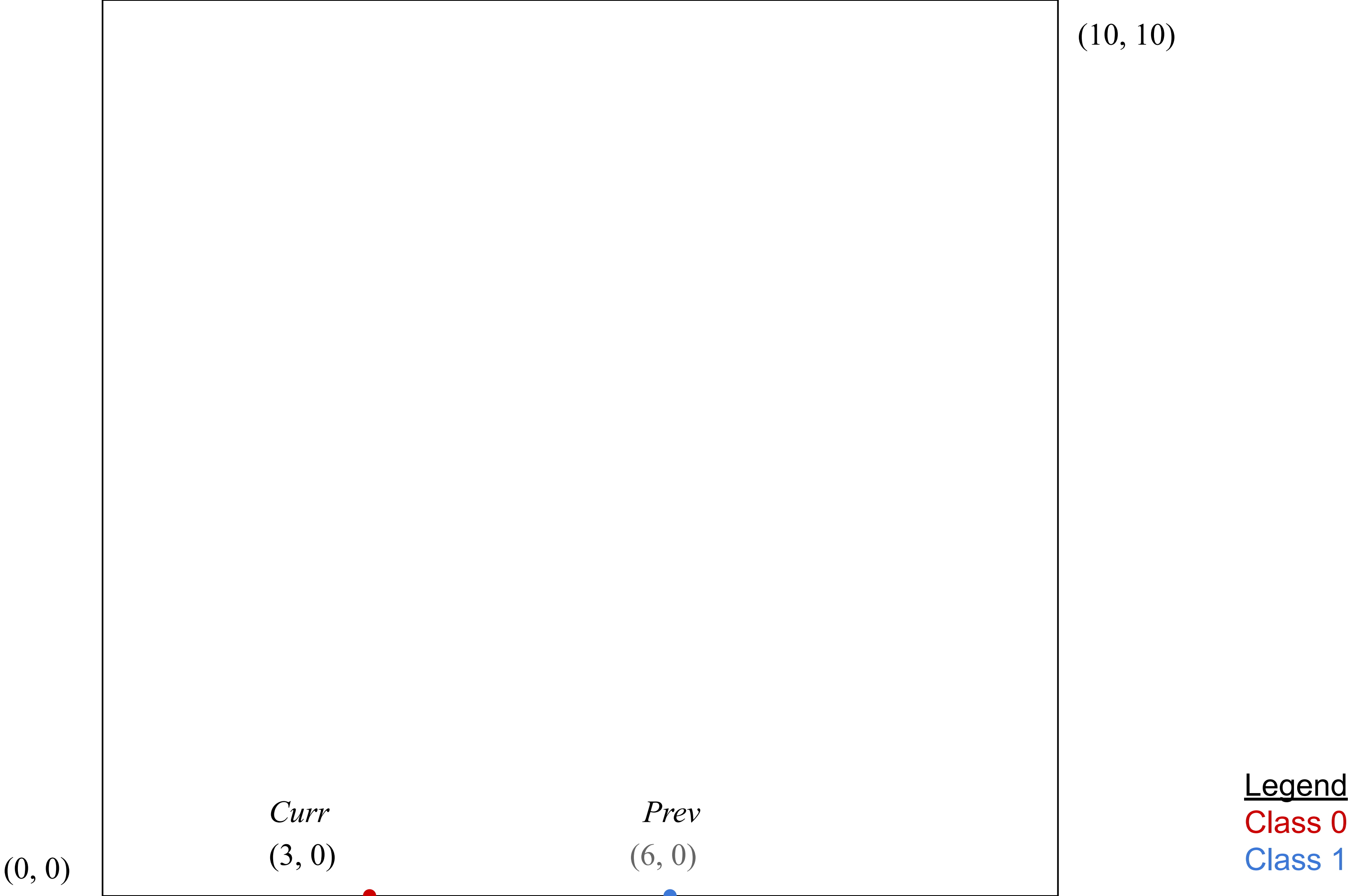
Class 1

```
# Compute representation of deviation between new counterexample and previous counterexample
(during set attempt) to later determine how to assign split segments' output value(s)
split_idxs = []
prev_counterex_relative_segment = [] # 2D input case: [0, 0] is bottom-left, [0, 1] is top-left,
[1, 0] is bottom-right, and [1, 1] is top-right
prev_curr_counterex_delta_signs = [] # Direction of previous counterex. (e.g. [-1, 0], [0, 1])

for key in inputVars:
     value = customRound(vals[key], awayFrom = prev_cntr_ex[key])
     split_idxs.append(value)


     diff = prev_cntr_ex[key] - value
     prev_counterex_relative_segment.append(int(diff > 0))


     delta_signs = int(diff / abs(diff)) if diff != 0 else 0
     prev_curr_counterex_delta_signs.append(delta_signs)
isAlignedCase = ( len(prev_curr_counterex_delta_signs) - prev_curr_counterex_delta_signs.count(0) ) == 1
```

```python
for segment in itertools.product(*[[0, 1] for idx in range(len(inputVars))]):
    bottom_left = []
    top_right = []

    for dim_idx, dim_val in enumerate(segment):
        if isAlignedCase:
            aligned_case_prev_counterex_segment.append(
                min(prev_curr_counterex_delta_signs[dim_idx] + 1, 1) )

        # Determines if there is an offset for this dimension's value or not
        curr_dim_offset = EPSILON * int(dim_val == prev_counterex_relative_segment[dim_idx])
```

```
for segment in itertools.product(*[[0, 1] for idx in range(len(inputVars))]):
    bottom_left = []
    top_right = []
    aligned_case_prev_counterex_segment = []

    for dim_idx, dim_val in enumerate(segment):
        # Iterate through each dimension of segment

    usePrevCounterExOutput = (list(segment) == aligned_case_prev_counterex_segment) \
        if isAlignedCase else (list(segment) == prev_counterex_relative_segment)
    output = int(prev_cntr_ex[outputVarIdx] > 0) if usePrevCounterExOutput
            else int(prev_cntr_ex[outputVarIdx] <= 0)
```


(10, 10)
(0, 0)
Legend
Class 0
Class 1



(10, 10)

*Curr*

(3, 0)

*Prev*

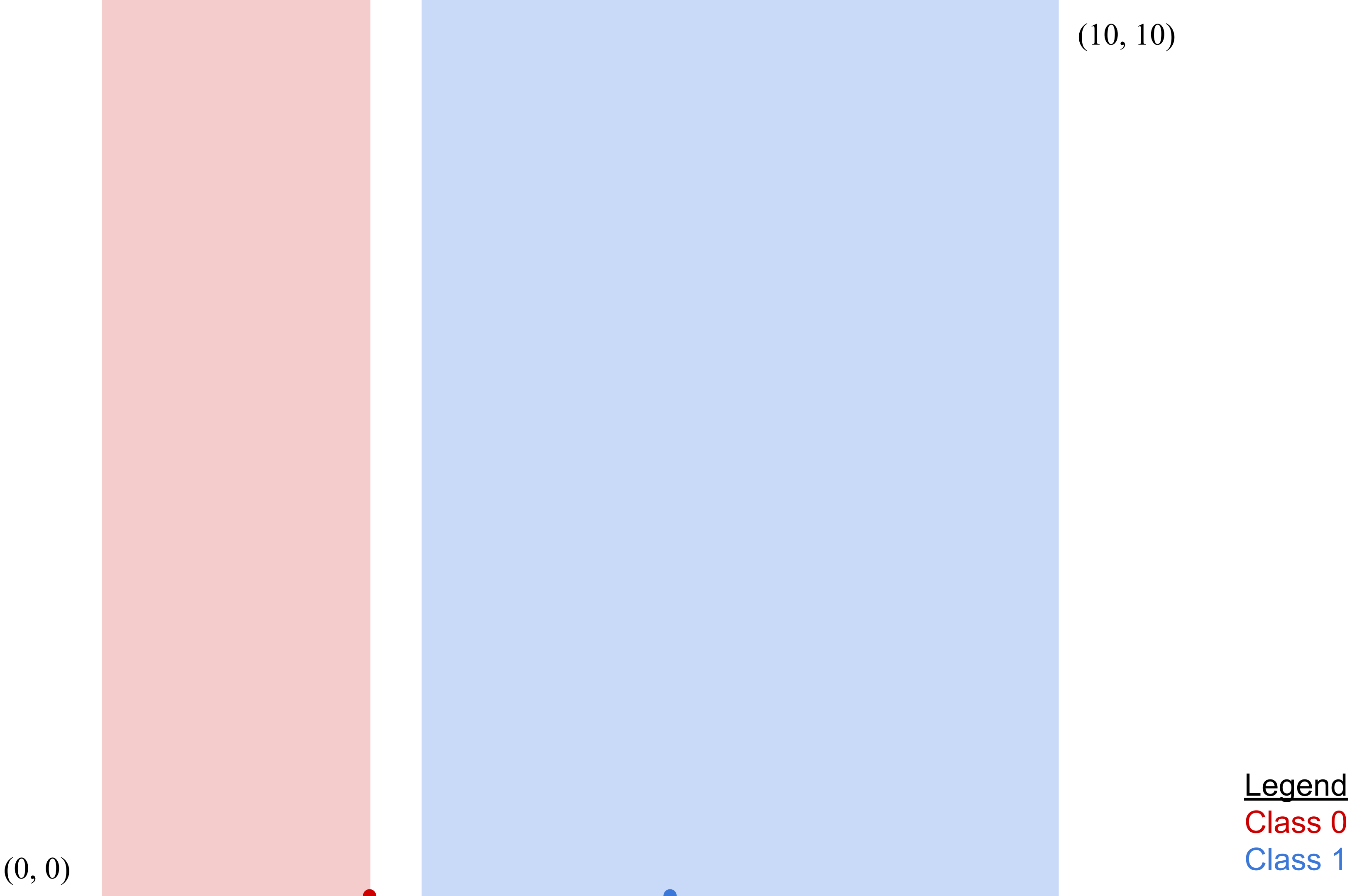
(6, 0)

(0, 0)

Legend

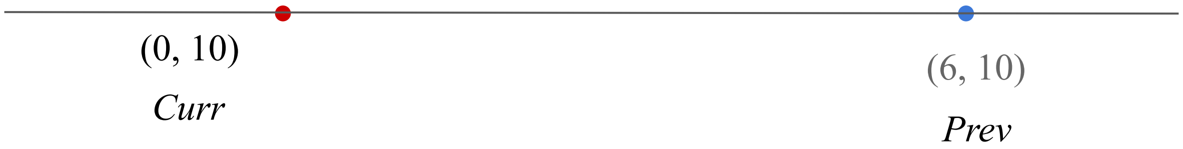
Class 0

Class 1


(10, 10)
(0, 0)
Legend
Class 0
Class 1

# Edge Case:
# *Line*

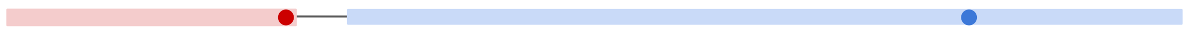
## **Segment Splitting Logic**



Legend

Class 0

Class 1



Legend

Class 0

Class 1