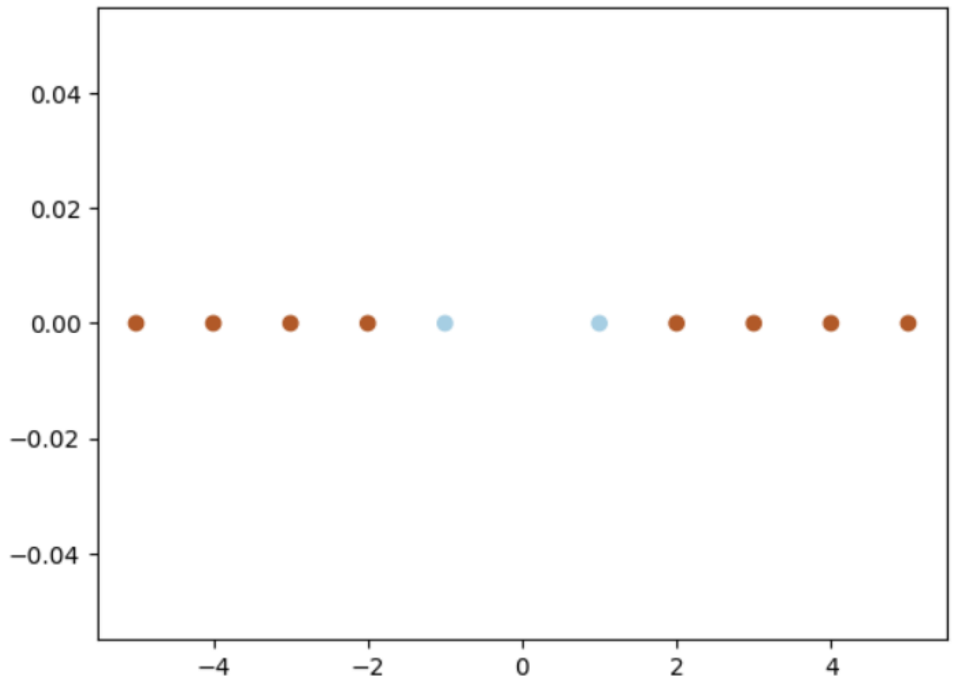
Assignment 4

For all cases I have taken a simple example on x=0 of linearly non-separable points. It will way be easier to analyze why dual is better to solve than primal (because of kernels).



Above pic shows set of points, ideally only non –linear boundary like ellipse, parabola can separate it.

Solving in primal

Lingo Model - non_separable_primal_1

```
min = 1*(w1*w1 + w2*w2) + 0.001*(q1 + q2 + q3 + q4 + q5 + q6 + q7 + q8 + q9 +q0) ;

-1 * (w1*1 + w2*0 + b) +q1 >= 1;
1 * (w1*2 + w2*0 + b) +q2>= 1;
1 * (w1*3 + w2*0 + b) +q3>= 1;
1 * (w1*4 + w2*0 + b) +q4>= 1;
1 * (w1*5 + w2*0 + b) +q5>= 1;

-1 * (w1*-1 + w2*0 + b) +q6>= 1;
1 * (w1*-2 + w2*0 + b) +q7>= 1;
1 * (w1*-3 + w2*0 + b) +q8>= 1;
1 * (w1*-4 + w2*0 + b) +q9>= 1;
1 * (w1*-5 + w2*0 + b) +q0>= 1;

w1*w1 + w2*w2 =1;
@free(b);
@free(w2);
@free(w1);
```

Result: It is a line passing through, y=-1, so it just ignores blue points, and tries to only make boundary for red points.

Variable	Value	Reduced Cost
W1	0.000000	0.000000
W2	0.9999996	0.000000
Q1	2.000000	0.000000
Q2	0.000000	0.1000000E-02
Q3	0.000000	0.1250000E-03
Q4	0.000000	0.1000000E-02
Q5	0.000000	0.1000000E-02
Q6	2.000000	0.000000
Q7	0.000000	0.000000
Q8	0.000000	0.1000000E-02
Q9	0.000000	0.1000000E-02
Q0	0.000000	0.8750000E-03
B	1.000000	0.000000

Solving in Dual

As we can see a parabola should be able to classify the points, so I take kernel Z=X^2.

X=[x1, x2] (where x2 is 0, for points lying on x1 only)

So [z1, z2] = [x1, x1^2+x2]

Why have I added x2?

Transform [x1, x2] to [x1, x1^2]:

- This transformation maps the data to a two-dimensional feature space where the second feature is the square of the first feature.
- It captures quadratic relationships, making it suitable for problems where the relationship between x1 and the target variable is expected to be quadratic.
- If you need to recover the original feature x2, you cannot do so directly from this transformation, as the information in x2 is lost.

Transform [x1, x2] to [x1, x1^2 + x1]:

- This transformation maps the data to a two-dimensional feature space where the second feature is the square of the first feature plus the first feature itself.
- It captures quadratic relationships (like the first transformation) but also includes a linear component.
- This transformation allows you to recover the original feature x2 by subtracting x1 from the transformed feature. It retains more information from the original data.

Points now are, projected on parabola.

X= ([-1, 1], [-2, 4], [-3, 9], [-4, 16], [-5, 25], [1, 1], [2, 4], [3, 9], [4, 16], [5, 25])

Solving in dual with kernel

```
min = 0.5*(d0*d0*-1*-1*2+d0*d1*-1*1*6+d0*d2*-1*1*12+d0*d3*-1*1*20+d0*d4*-1*1*30+d0*d5*-1*-1*0+d0*d6*-1*1*2+d0*d7*-1*1*6+d0*d8*-1*1*12+d0*d9*-1*1*20+d1*d0*1*-1*6+d1*d1*1*1*20+d1*d2*1*1*42+d1*d3*1*1*72+d1*d4*1*1*110+d
      -(d0+d1+d2+d3+d4+d5+d6+d7+d8+d9);

d0*-1+d1*1+d2*1+d3*1+d4*1+d5*-1+d6*1+d7*1+d8*1+d9*1=0;

d0>=0;
d1>=0;
d2>=0;
d3>=0;
d4>=0;
d5>=0;
d6>=0;
d7>=0;
d8>=0;
d9>=0;

d0<=100;
d1<=100;
d2<=100;
d3<=100;
d4<=100;
d5<=100;
d6<=100;
d7<=100;
d8<=100;
d9<=100;


d0+n0=0;
d1+n1=0;
d2+n2=0;
d3+n3=0;
d4+n4=0;
d5+n5=0;
d6+n6=0;
d7+n7=0;
d8+n8=0;
d9+n9=0;


@free(d0);
@free(d1);
@free(d2);
@free(d3);
@free(d4);
@free(d5);
@free(d6);
@free(d7);
@free(d8);
@free(d9);
```

Variable	Value	Reduced Cost
D0	0.1111111	0.000000
D1	0.1111111	0.1280823E-08
D2	0.000000	0.2881851E-08
D3	0.000000	0.5123292E-08
D4	0.000000	0.8005145E-08
D5	0.1111111	0.000000
D6	0.1111111	0.1280825E-08
D7	0.000000	0.2881853E-08
D8	0.000000	0.5123290E-08
D9	0.000000	0.8005149E-08
N0	-0.1111111	0.000000
N1	-0.1111111	0.000000
N2	0.000000	0.000000
N3	0.000000	0.000000
N4	0.000000	0.000000
N5	-0.1111111	0.000000
N6	-0.1111111	0.000000
N7	0.000000	0.000000
N8	0.000000	0.000000
N9	0.000000	0.000000

By using python code for W and b , I calculated.

W= [0, 0.6666666] and b =-1.66664

Weight

```
X = np.array([[-1, 1], [-2, 4], [-3, 9], [-4, 16], [-5, 25], [1, 1], [2, 4], [3, 9], [4, 16], [5, 25]])
y = np.array([-1, 1, 1, 1, 1, -1, 1, 1, 1, 1])
d=np.array([0.1111111,0.1111111,0,0,0,0.1111111,0.1111111,0,0,0])
w = np.zeros(X.shape[1])
st=""
for i in range(0,10):
       w+=(d[i]*y[i]*X[i])

```

```
[18] w
```

```
array([0.        , 0.6666666])
```

Bias

```
def calculate_b(X, y, alphas, kernel, support_vector_indices):
    b_sum = 0
    num_support_vectors = len(support_vector_indices)

    for i in support_vector_indices:
        prediction = 0
        for j in support_vector_indices:
            prediction += alphas[j] * y[j] * kernel(X[i], X[j])
        b_sum += y[i] - prediction

    b = b_sum / num_support_vectors
    return b

support_vector_indices = [0, 1,2,3]
alphas = [0.1111111, 0.1111111,0.1111111,0.1111111]
X = np.array([[-1, 1], [1, 1],[2, 4],[-2, 4]])
y = np.array([-1, -1,1,1])

def kernel(x1, x2):
    return np.dot(x1, x2)

b = calculate_b(X, y, alphas, kernel, support_vector_indices)
print("Bias (b):", b)

```

```
Bias (b): -1.6666664999999998
```

Which gives us 0*z1 + 0.666666z2 - 1.6666664 =0

Which is line passing through z2=2.49

When converted to original form of [x1, x2], we get (x1^2+x2) –2.49 =0, parabola having roots at 1.58 and –1.58 and cutting x2(y-axis) at 2.49, it is downward opening parabola. So Blue points have got separated now.

Sample 2

Now , I will solve the same example using polynomial Kernel based on Mercer theorem.

(1+pT * q) ^ d

```python
import numpy as np

X = np.array([[-1, -1], [-2, 2], [-3, 3], [-4, 4], [-5, 5], [1, 1], [2, 2], [3, 3], [4, 4], [5, 5]])
y = np.array([-1, 1, 1, 1, 1, -1, 1, 1, 1, 1])

# Define the degree parameter for the polynomial kernel
degree = 2

# Initialize the kernel matrix
kernel_matrix = np.zeros((X.shape[0], X.shape[0]))

# Compute the kernel values for the polynomial kernel of degree 2
for i in range(X.shape[0]):
    for j in range(X.shape[0]):
        kernel_matrix[i, j] = (1 + np.dot(X[i], X[j])) ** degree

st = ""
for i in range(0, 10):
    for j in range(0, 10):
        st += str(f"d{i}*d{j}*{y[i]}*{y[j]}*{kernel_matrix[i][j]}+")

```

```
st
```

```
'd0*d0*-1*-1*4.0+d0*d1*-1*1*9.0+d0*d2*-1*1*16.0+d0*d3*-1*1*25.0+d0*d4*-1*1*36.0+d0*d5*-1*-1*0.0+d0*d6*-1*1*1.0+d0*d7*-1*1*4.0+d0*d8*-1*1*9.0+d0*d9*-1*1*16.0+d1*d0*1*-1*9.0+d1*d1*1*1*25.0+d1*d2*1*1*49.0+d1*d3*1*1*81.0+d1*d4*1*1*121.0+d1*d5*1*-
0+d1*d7*1*1*25.0+d1*d8*1*1*49.0+d1*d9*1*1*81.0+d2*d0*1*-1*16.0+d2*d1*1*1*49.0+d2*d2*1*1*100.0+d2*d3*1*1*169.0+d2*d4*1*1*256.0+d2*d5*1*-1*4.0+d2*d6*1*1*25.0+d2*d7*1*1*64.0+d2*d8*1*1*121.0+d2*d9*1*1*196.0+d3*d0*1*-1*25.0+d3*d1*1*1*81.0+d3*d2*1*
89.0+d3*d4*1*1*441.0+d3*d5*1*-1*9.0+d3*d6*1*1*49.0+d3*d7*1*1*121.0+d3*d8*1*1*225.0+d3*d9*1*1*361.0+d4*d0*1*-1*36.0+d4*d1*1*1*121.0+d4*d2*1*1*256.0+d4*d3*1*1*441.0+d4*d4*1*1*676.0+d4*d5*1*-1*16.0+d4*d6*1*1*81.0+d4*d7*1*1*196.0+d4*d8*1*1*361.0+
d0*-1*-1*0.0+d5*d1*-1*1*1.0+d5*d2*-1*1*4.0+d5*d3*-1*1*9.0+d5*d4*-1*1*16.0+d5*d5*-1*-1*4.0+d5*d6*-1*1*9.0+d5*d7*-1*1*16.0+d5*d8*-1*1*25.0+d5*d9*-1*1*36.0+d6*d0*1*-1*1.0+d6*d1*1*1*9.0+d6*d2*1*1*25.0+d6*d3*1*1*49.0+d6*d4*1...'
```

We know how primal solves it.

Using Dual

```
min =0.5*(d0*d0*-1*-1*4.0+d0*d1*-1*1*9.0+d0*d2*-1*1*16.0+d0*d3*-1*1*25.0+d0*d4*-1*1*36.0+d0*d5*-1*-1*0.0+d0*d6*-1*1*1.0+d0*d7*-1*1*4.0+d0*d8*-1*1*9.0+d0*d9*-1*1*16.0+d1*d0*1*-1*9.0+d1*d1*1
     -(d0+d1+d2+d3+d4+d5+d6+d7+d8+d9);

d0*-1+d1*1+d2*1+d3*1+d4*1+d5*-1+d6*1+d7*1+d8*1+d9*1=0;

d0>=0;
d1>=0;
d2>=0;
d3>=0;
d4>=0;
d5>=0;
d6>=0;
d7>=0;
d8>=0;
d9>=0;

d0<=100;
d1<=100;
d2<=100;
d3<=100;
d4<=100;
d5<=100;
d6<=100;
d7<=100;
d8<=100;
d9<=100;


d0+n0=100;
d1+n1=100;
d2+n2=100;
d3+n3=100;
d4+n4=100;
d5+n5=100;
d6+n6=100;
d7+n7=100;
d8+n8=100;
d9+n9=100;


@free(d0);
@free(d1);
@free(d2);
@free(d3);
```

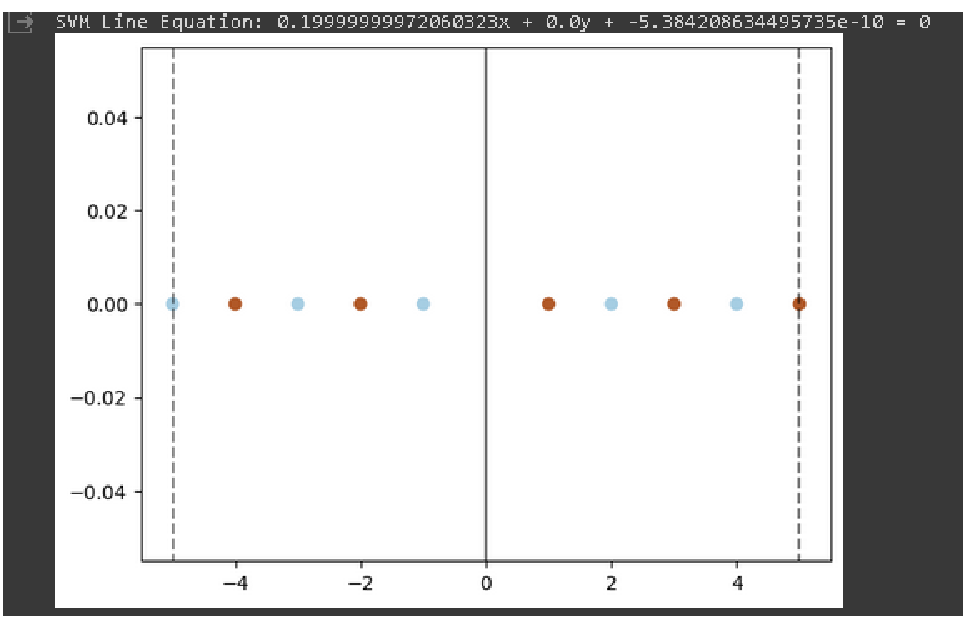
C value taken was 100, as we don’t worry about just maximizing margin, we need zero error using kernel.

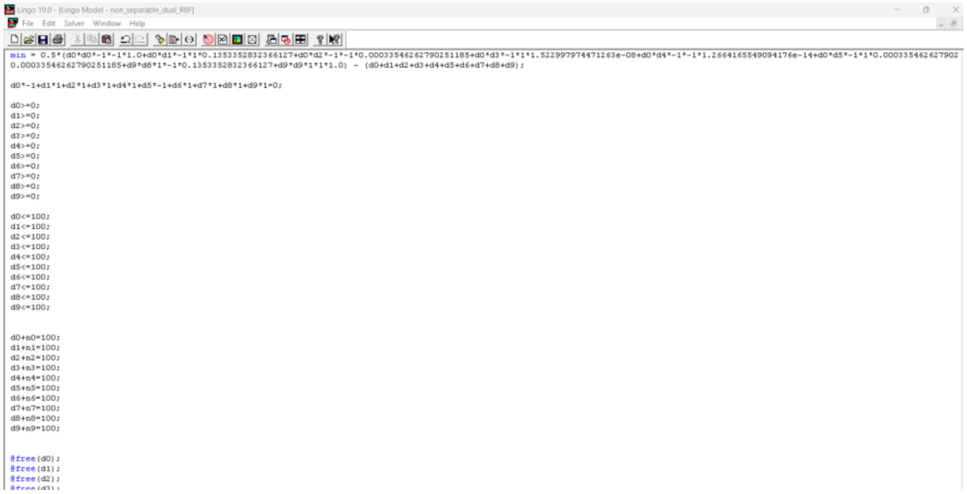
Variable	Value	Reduced Cost
D0	0.1111111	0.000000
D1	0.1111111	0.000000
D2	0.1027579E-06	0.000000
D3	0.8911439E-07	0.000000
D4	0.8525468E-07	0.000000
D5	0.1111111	0.000000
D6	0.1111111	0.000000
D7	0.1027579E-06	0.000000
D8	0.8911439E-07	0.000000
D9	0.8525468E-07	0.000000
N0	99.88889	0.000000
N1	99.88889	0.000000
N2	100.0000	0.000000
N3	100.0000	0.000000
N4	100.0000	0.000000
N5	99.88889	0.000000
N6	99.88889	0.000000
N7	100.0000	0.000000
N8	100.0000	0.000000
N9	100.0000	0.000000

We can see D0, D1, D5,D6 are support vectors, rest negligible. So, a hyperplane in dimension Z is dividing the points.

Sample 3

Now I will solve alternatively occurring points of different class using infinite RBF Kernel.






```
min = 0.5*(d0*d0*-1*-1*1.0+d0*d1*-1*1*0.1353352832366127+d0*d2*-1*-1*0.00033546262790251185+d0*d3*-1*1*1.522997974471263e-08+d0*d4*-1*-1*1.2664165549094176e-14+d0*d5*-1*1*0.000335462627902
0.00033546262790251185+d9*d8*1*-1*0.1353352832366127+d9*d9*1*1*1.0) - (d0+d1+d2+d3+d4+d5+d6+d7+d8+d9);

d0*-1+d1*1+d2*1+d3*1+d4*1+d5*-1+d6*1+d7*1+d8*1+d9*1=0;

d0>=0;
d1>=0;
d2>=0;
d3>=0;
d4>=0;
d5>=0;
d6>=0;
d7>=0;
d8>=0;
d9>=0;

d0<=100;
d1<=100;
d2<=100;
d3<=100;
d4<=100;
d5<=100;
d6<=100;
d7<=100;
d8<=100;
d9<=100;


d0+n0=100;
d1+n1=100;
d2+n2=100;
d3+n3=100;
d4+n4=100;
d5+n5=100;
d6+n6=100;
d7+n7=100;
d8+n8=100;
d9+n9=100;


@free(d0);
@free(d1);
@free(d2);
@free(d3);
```

Variable	Value	Reduced Cost
D0	1.783944	0.000000
D1	0.5970695	0.000000
D2	0.4319660	0.000000
D3	0.4031758	0.000000
D4	0.3517329	0.000000
D5	1.783944	0.000000
D6	0.5970695	0.000000
D7	0.4319660	0.000000
D8	0.4031758	0.000000
D9	0.3517329	0.000000
N0	98.21606	0.000000
N1	99.40293	0.000000
N2	99.56803	0.000000
N3	99.59682	0.000000
N4	99.64827	0.000000
N5	98.21606	0.000000
N6	99.40293	0.000000
N7	99.56803	0.000000
N8	99.59682	0.000000
N9	99.64827	0.000000

D is non-zero everywhere , so all are support vectors .

W=[4.06333286,0] , b=0

WT[x] + b -> x1=0. It is a y-axis line; RBF automatically converted the points to oscillating dimension.

Bias

```python
def calculate_b(X, y, alphas, kernel, support_vector_indices):
    b_sum = 0
    num_support_vectors = len(support_vector_indices)

    for i in support_vector_indices:
        prediction = 0
        for j in support_vector_indices:
            prediction += alphas[j] * y[j] * kernel(X[i], X[j])
        b_sum += y[i] - prediction

    b = b_sum / num_support_vectors
    return b

support_vector_indices = [0, 1,2,3,4,5,6,7,8,9]
alphas = [1.783944,0.5970695,0.4319660,0.4031758,0.3517329,1.783944,0.5970695,0.4319660 ,0.4031758,0.3517329]
X = np.array([[-1, 0], [-2, 0], [-3, 0], [-4, 0], [-5, 0], [1, 0], [2, 0], [3, 0], [4, 0], [5, 0]])
y = np.array([-1, 1, -1, 1, -1, 1, -1, 1, -1, 1])

def kernel(x1, x2):
    return np.dot(x1, x2)

b = calculate_b(X, y, alphas, kernel, support_vector_indices)
print("Bias (b):", b)
```

```
Bias (b): 0.0
```

Weight

```python
[25]
X = np.array([[-1, 0], [-2, 0], [-3, 0], [-4, 0], [-5, 0], [1, 0], [2, 0], [3, 0], [4, 0], [5, 0]])
y = np.array([-1, 1, -1, 1, -1, 1, -1, 1, -1, 1])
d=np.array([1.783944,0.5970695,0.4319660,0.4031758,0.3517329,1.783944,0.5970695,0.4319660 ,0.4031758,0.3517329])
w = np.zeros(X.shape[1])
st=""
for i in range(0,10):
       w+=(d[i]*y[i]*X[i])

[26]
w
```

```
array([4.0633286, 0.        ])
```

Link to collab - https://colab.research.google.com/drive/16-hwrBa3w9WHNfVyvjWMs81xgxS0kFTm?usp=sharing

Lindo files are in Zip folder containing this.