SVM Dual Problem and Kernel Methods

Yaniv Tal

18-Jun-2021

Agenda

- SVM – The Primal problem
- SVM – The Dual problem
- Linear Inseparability and Kernels
- Common Kernels
- Implementation
- Results on Datasets

SVM Primal Problem

SVM – The Primal Problem

- Linear separator, optimizes for largest margin

We have n data points, p features

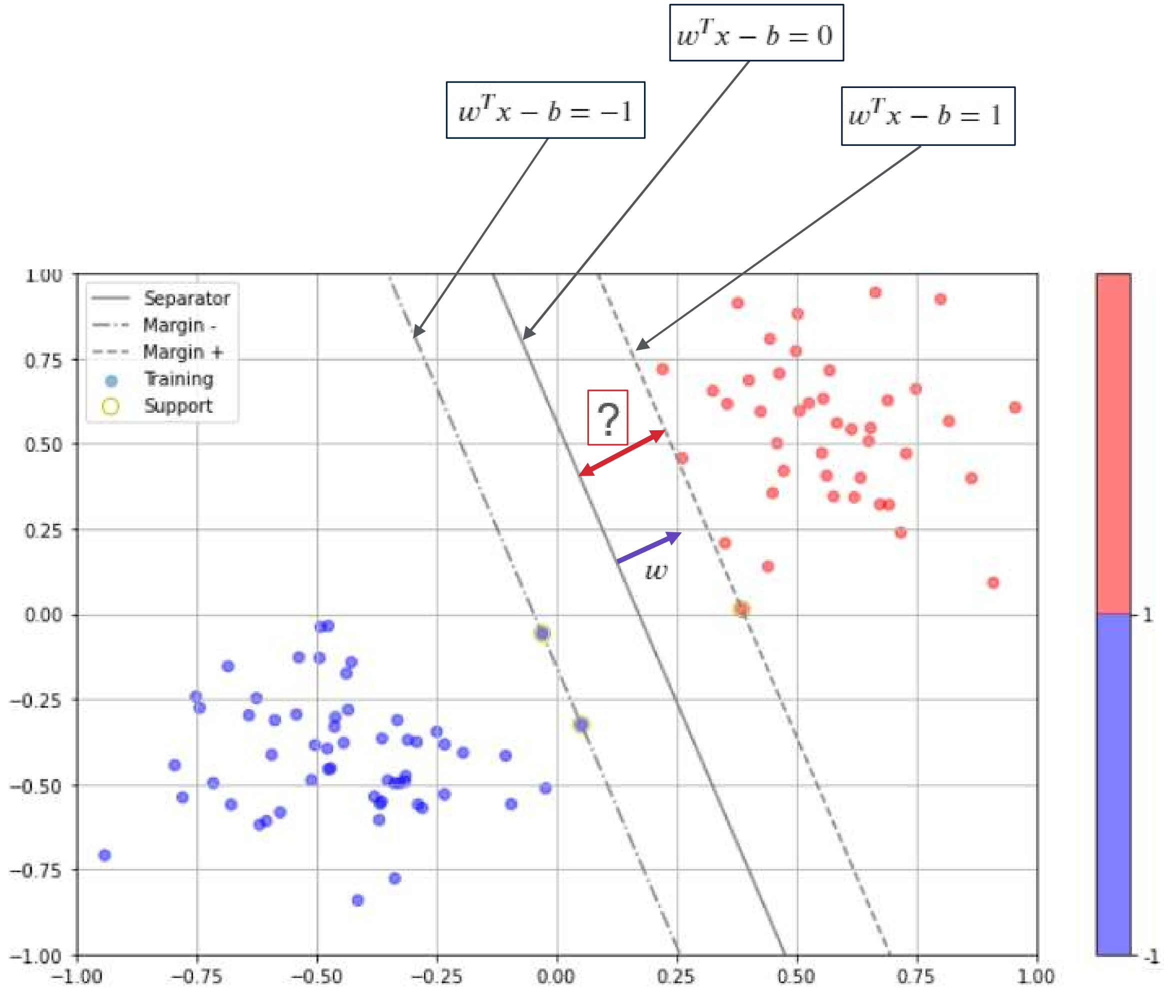
$x_i, w \in \mathbb{R}^p$

$b \in \mathbb{R}$

$y_i \in \{-1, 1\}$

$x \in \mathbb{R}^{nxp}$

$y \in \mathbb{R}^n$



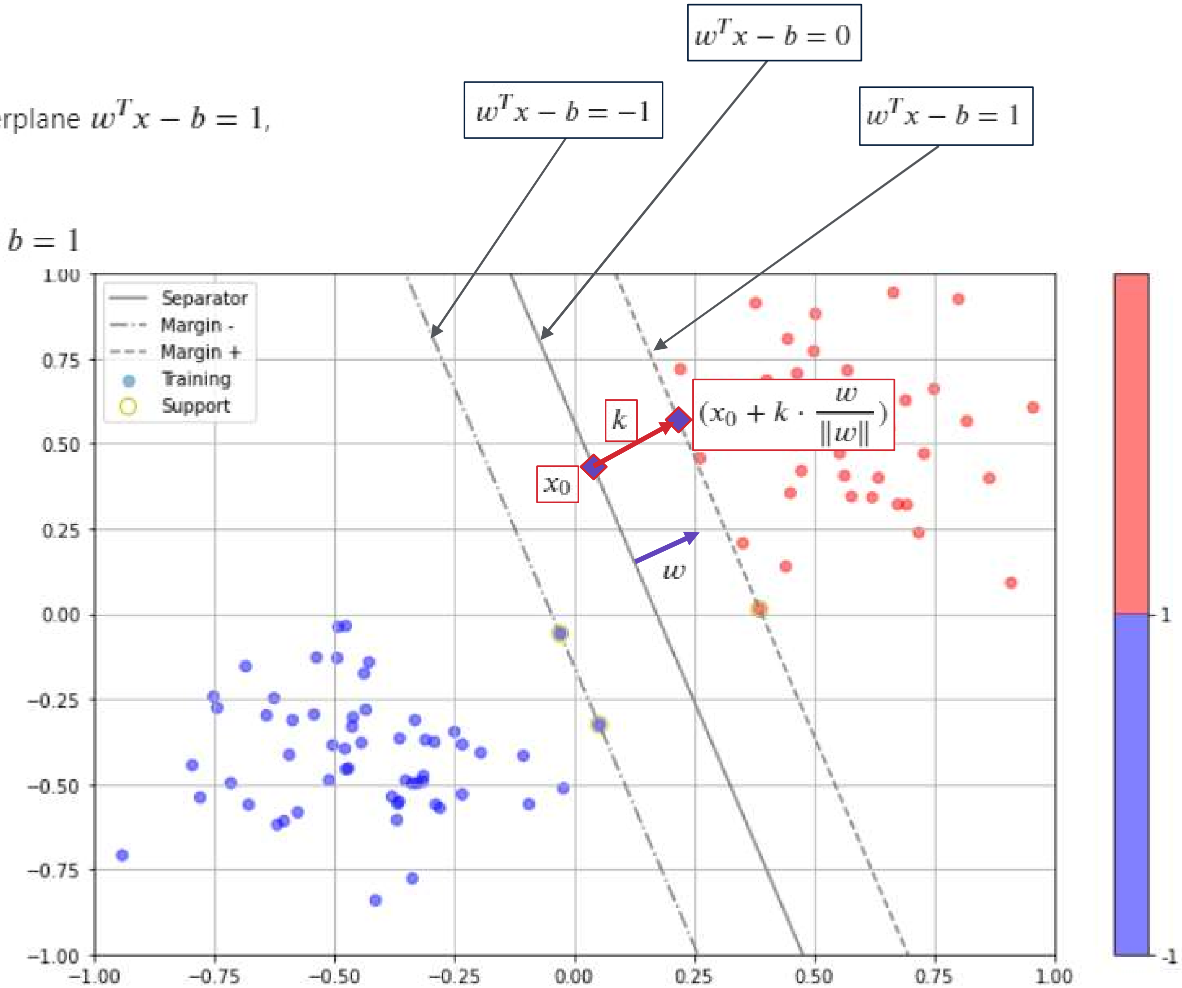
SVM – The Primal Problem

- Margin size formulation :

Let x_0 be a point on the hyperplane, so: $w^T x_0 - b = 0$.
If we move from x_0 a distance k in the direction of w to the hyperplane $w^T x - b = 1$,
we get a point: $x_0 + k \cdot \frac{w}{\|w\|}$.

The new point is on the upper plane, so: $w^T \cdot (x_0 + k \cdot \frac{w}{\|w\|}) - b = 1$

And so we get: $k = \frac{1}{\|w\|}$

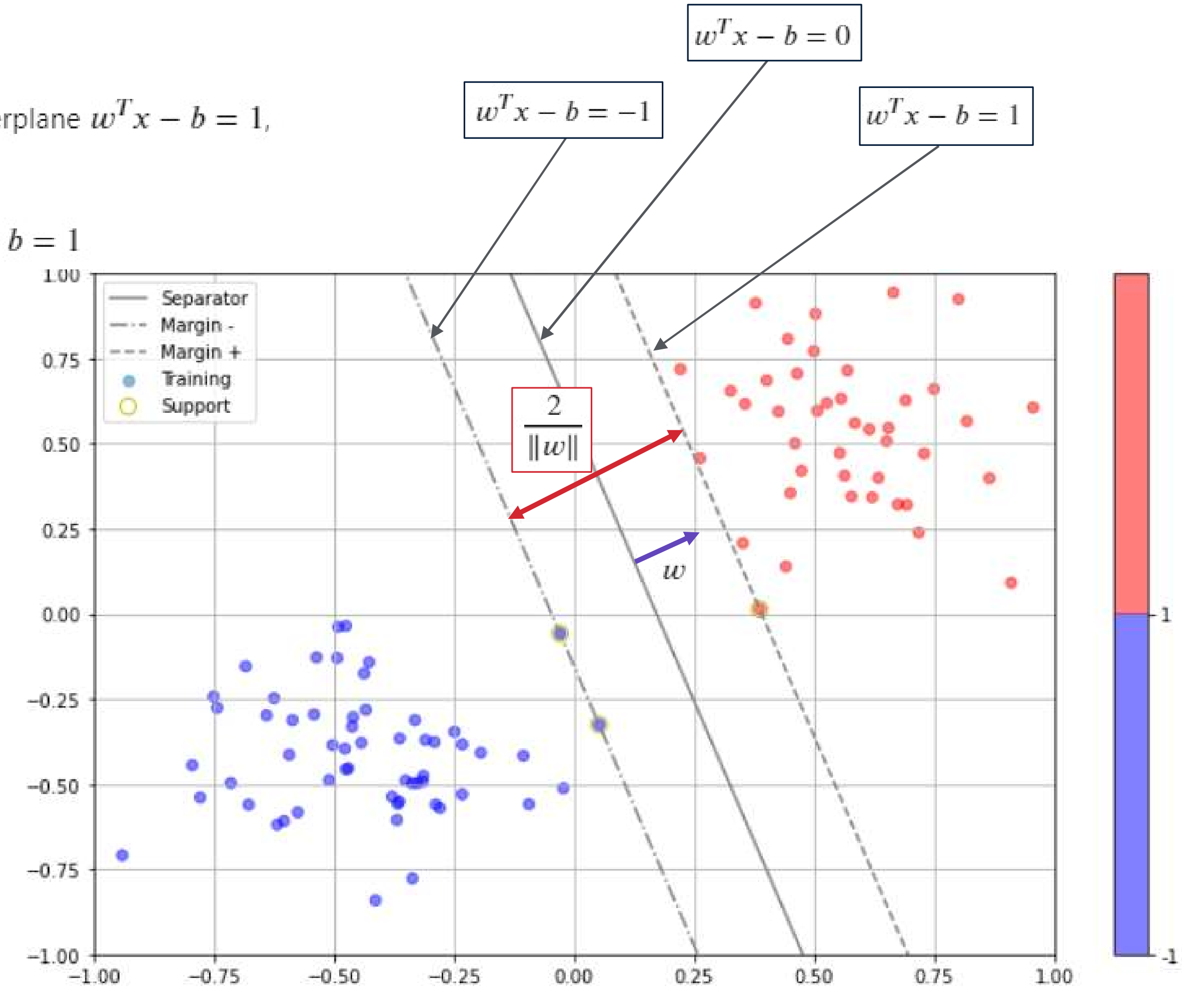


SVM – The Primal Problem

- Margin size formulation :

Let x_0 be a point on the hyperplane, so: $w^T x_0 - b = 0$.
If we move from x_0 a distance k in the direction of w to the hyperplane $w^T x - b = 1$,
we get a point: $x_0 + k \cdot \frac{w}{\|w\|}$.
The new point is on the upper plane, so: $w^T \cdot (x_0 + k \cdot \frac{w}{\|w\|}) - b = 1$
And so we get: $k = \frac{1}{\|w\|}$

- Hence, the margin we want to <u>maximize</u> is: $\frac{2}{\|w\|}$
- Which is the same as <u>minimizing</u> $\frac{1}{2}\|w\|^2$



SVM – The Primal Problem

- **Constraints**: All points should be on the correct sides of the lines:

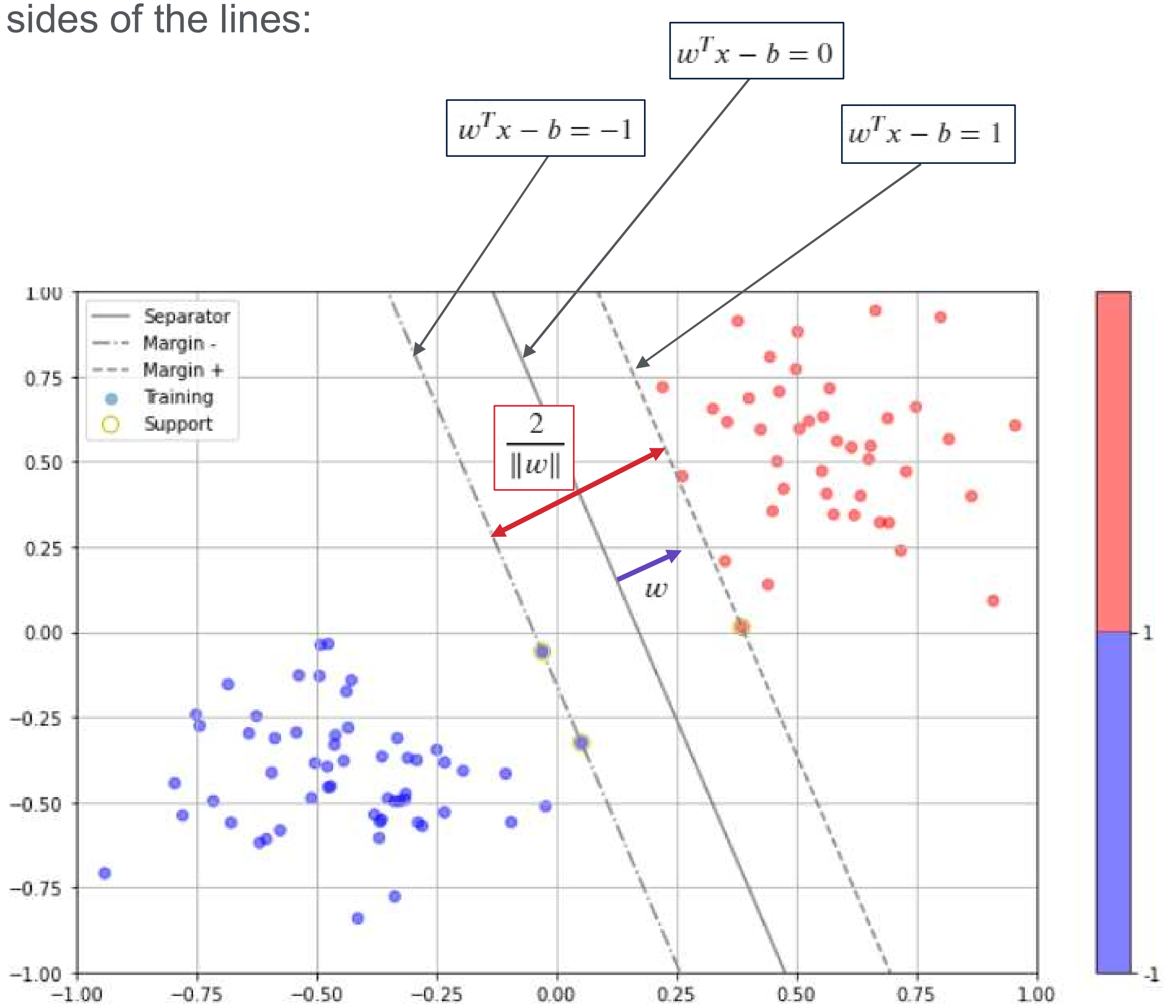
$$y_i(x_i^T w + b) \geq 1$$
$$i = 1..n$$

- **And so, we get the primal SVM problem:**

$$\min_{w,b} \frac{1}{2}\|w\|^2$$

s.t.

$$y_i(x_i^T w + b) \geq 1$$
$$i = 1..n$$



SVM – The Primal Problem

- Primal SVM problem:

$$\min_{w,b} \frac{1}{2}\|w\|^2$$

s.t.

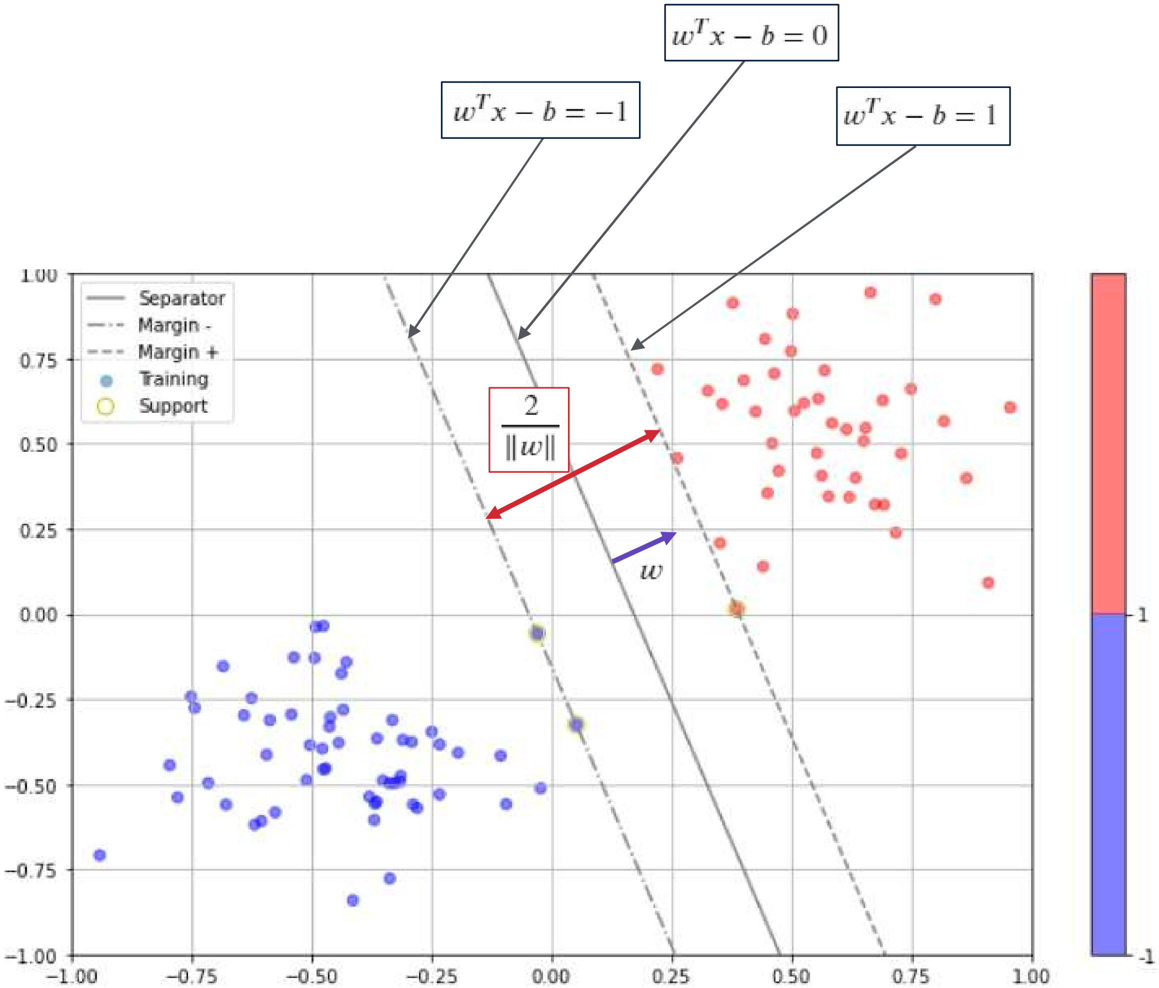
$$y_i(x_i^T w + b) \geq 1$$

$$i = 1..n$$

- Lagrangian:

$$\mathcal{L}_p(w, b, \alpha) = \frac{1}{2}\|w\|^2 - \sum_{i=0}^{n} \alpha_i(y_i(x_i^T w + b) - 1)$$

  - Quadratic problem, solved with KKT conditions
  - Note that the only active constraints are the points that define the margins -
    from KKT – all other Alphas are zero!



SVM Dual Problem

SVM – The Dual Problem (Strong duality)

- Formulation:

$$Max_\alpha Min_{w,b} \mathcal{L}(w, b, \alpha)$$

$$\mathcal{L}(w, b, \alpha) = \frac{1}{2} w^T w - \sum_{i=1}^{n} \alpha_i \left(y_i (w^T x_i - b) - 1 \right)$$

$$g(\alpha) = Min_{w,b} \mathcal{L}(w, b, \alpha)$$

$$\frac{\partial \mathcal{L}}{\partial w} = 0 \implies w = \sum_i^n \alpha_i y_i x_i$$

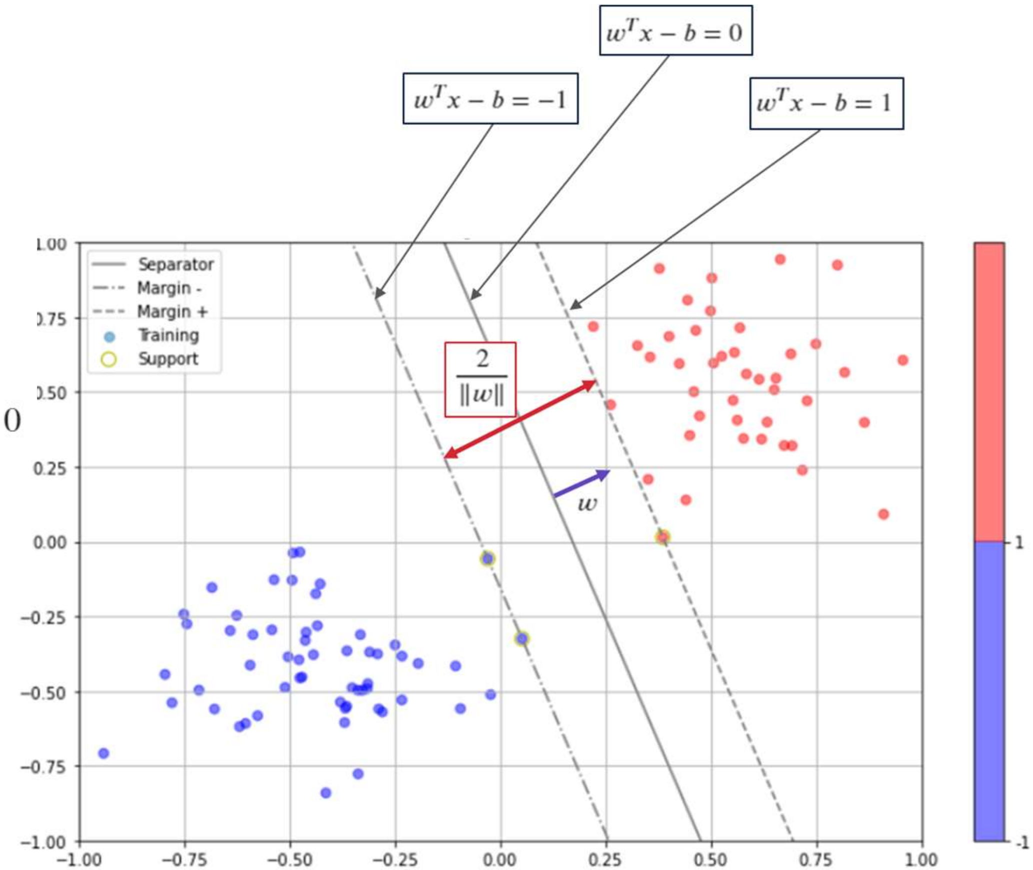
$$\frac{\partial \mathcal{L}}{\partial b} = 0 \implies \alpha^T y = 0$$

$$g(\alpha) = \sum_{i=0}^{n} \alpha_i - \frac{1}{2} \sum_{i=1}^{n} \sum_{j=1}^{n} \alpha_i \alpha_j y_i y_j x_i^T x_j, \quad \alpha^T y = 0, \quad \alpha_i \geq 0$$

$$Max_\alpha g(\alpha)$$

s.t.

$$\alpha^T y = 0, \quad \alpha_i \geq 0$$



SVM – The Dual Problem (Strong duality)

- Formulation:

$$Max_{\alpha} g(\alpha)$$

s.t.

$$\alpha^T y = 0, \quad \alpha_i \geq 0$$

- Which is same as:

$$Min_{\alpha}\left(\frac{1}{2}\sum_{i=1}^{n}\sum_{j=1}^{n}\alpha_i\alpha_j y_i y_j x_i^T x_j\right) - \sum_{i=1}^{n}\alpha_i$$

s.t.

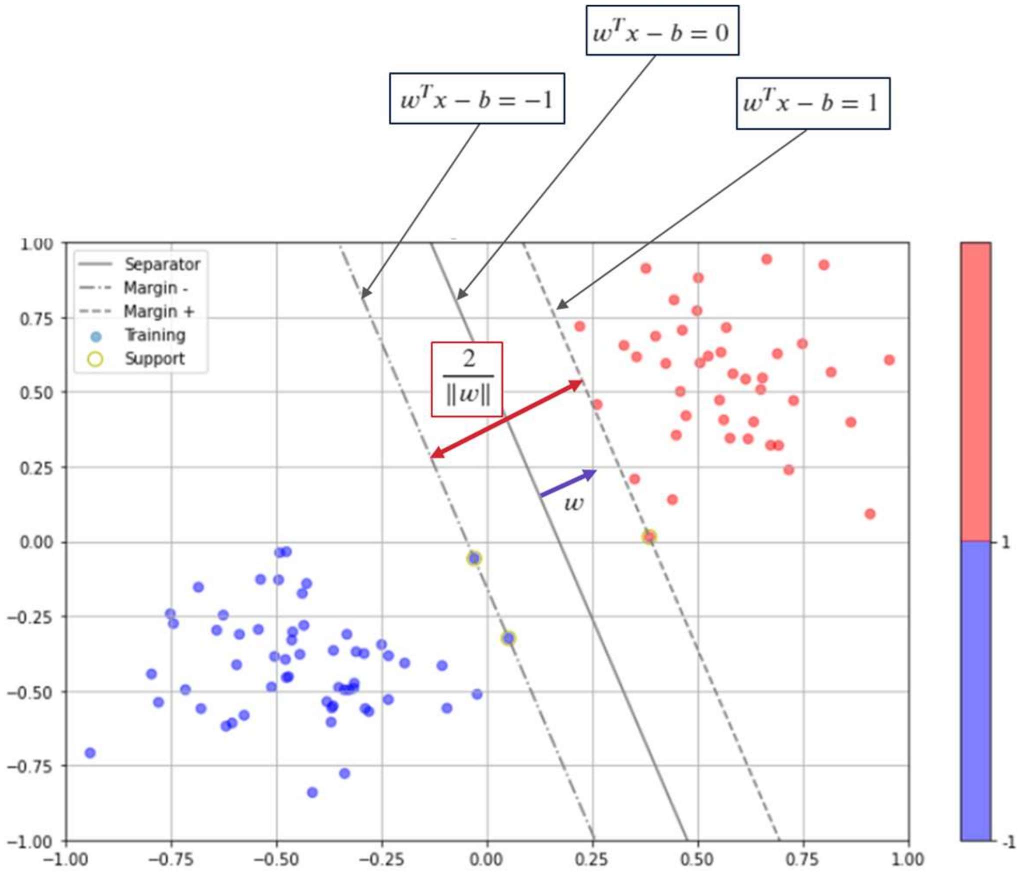
$$\alpha^T y = 0, \quad \alpha_i \geq 0$$

- Remember - Alpha=0 for non-support vectors

$$Min_{\alpha}\left(\frac{1}{2}\sum_{SV\,pairs(i,j)}\alpha_i\alpha_j y_i y_j x_i^T x_j\right) - \sum_{SV\,alphas}\alpha_i$$

s.t.

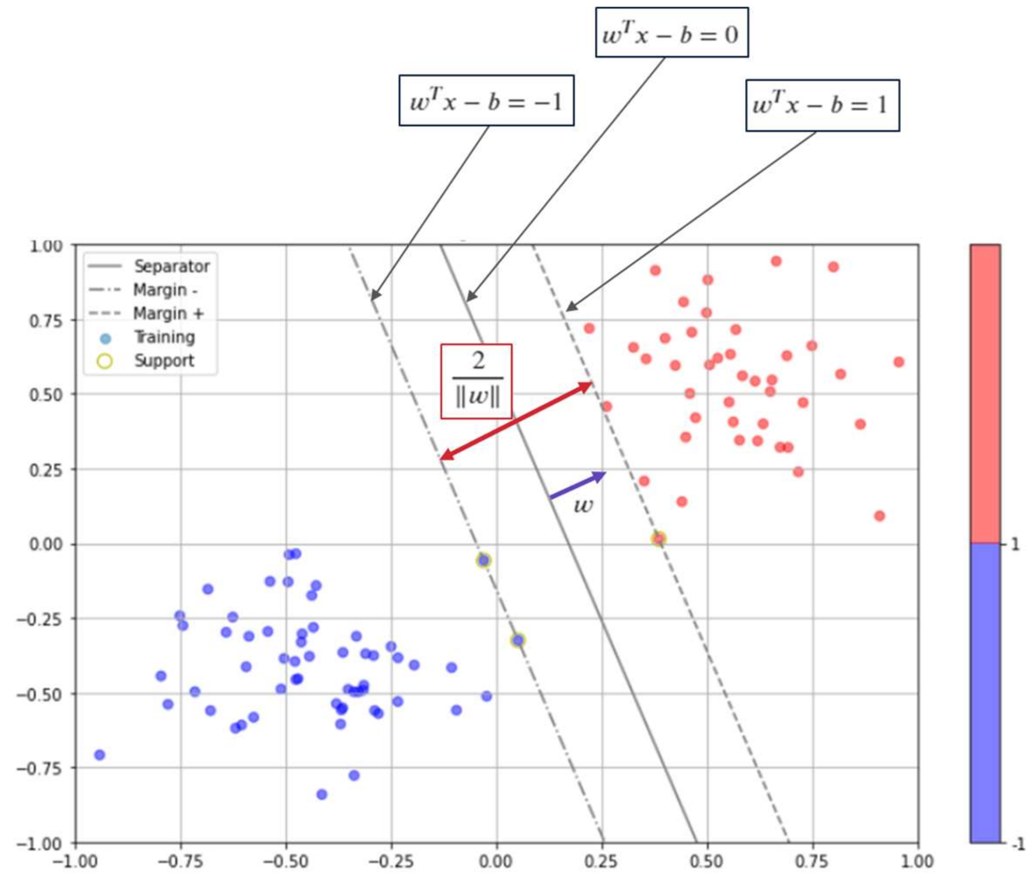
$$\alpha^T y = 0, \quad \alpha_i \geq 0$$



Why go Dual?

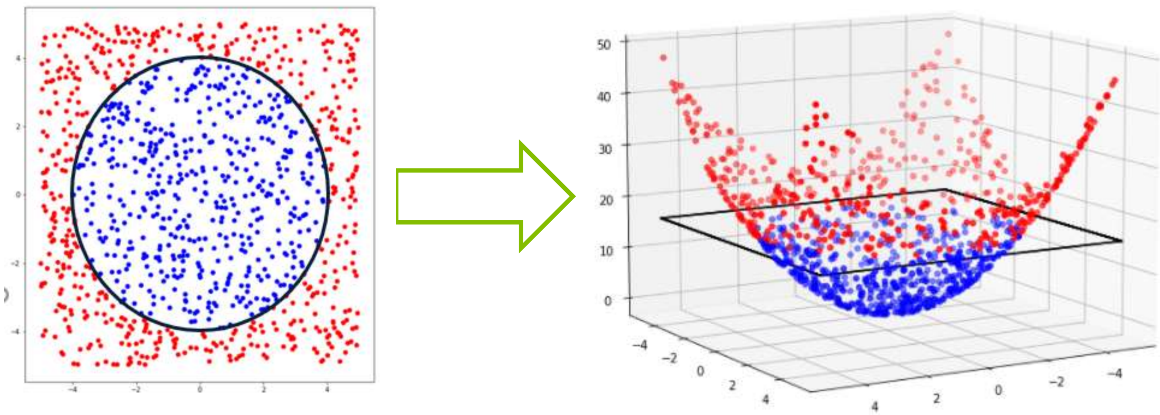
- "Wide" datasets - High dimensional data (p>n):
  - Primal problem size – n x p (Size of X)
  - Dual problem size – n x n (Size of $X^T X$)

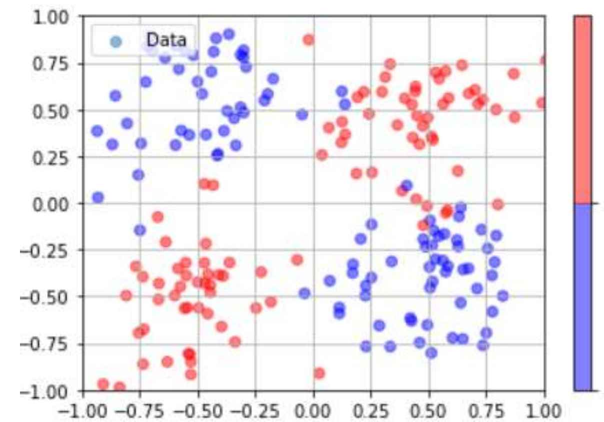
- Kernels!

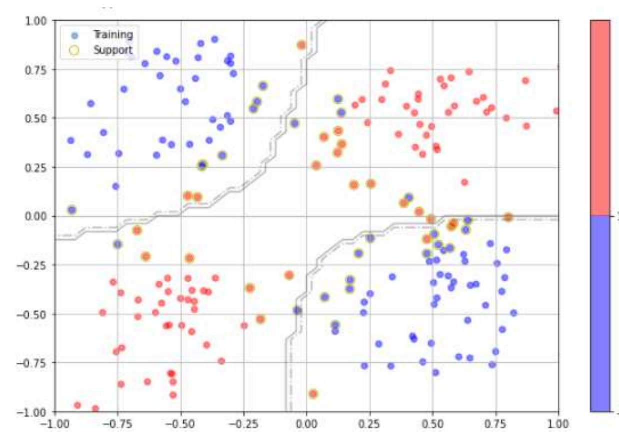


SVM with Nonlinearly Separable Data, Kernel Functions

Non-linearly separable data







- Cover's theorem – Linear separability is more likely in higher dimensions

- The good news: Given enough dimensions, Everything is linearly separable!
- The bad news: We get a VERY wide matrices….

Adding Features the Standard Way

- Reminder – The dual problem:

$$Min_\alpha\left(\frac{1}{2}\sum_{i=1}^{n}\sum_{j=1}^{n}\alpha_i\alpha_j y_i y_j \underbrace{x_i^T x_j}\right)-\sum_{i=1}^{n}\alpha_i$$

s.t.

$$\alpha^T y = 0, \quad \alpha_i \geq 0$$

- Define:

$$\varphi(x):\mathbb{R}^p \to \mathbb{R}^P, \quad P \geq p$$

$$k(x,y)=\varphi(x)\cdot\varphi(y), \quad k(x,y):\mathbb{R}^p x \mathbb{R}^p \to \mathbb{R}$$

- So, the problem becomes:

$$Min_\alpha\left(\frac{1}{2}\sum_{i=1}^{n}\sum_{j=1}^{n}\alpha_i\alpha_j y_i y_j \underbrace{k(x_i,x_j)}\right)-\sum_{i=1}^{n}\alpha_i$$

s.t.

$$\alpha^T y = 0, \quad \alpha_i \geq 0$$

- Dot product between vectors of dimension P instead of p.

Adding features – the Kernel Trick

- A function $k : \mathbb{R}^p x \mathbb{R}^p \rightarrow \mathbb{R}$ is called a kernel
  if there exists a mapping function $\varphi : \mathbb{R}^p \rightarrow \mathbb{R}^P$
  so that the following always holds: $\forall x, y : k(x, y) = \varphi(x) \cdot \varphi(y)$

- A mapping function $\varphi : \mathbb{R}^p \rightarrow \mathbb{R}^P$ is said to afford a kernel if such k exists

- Example – Inhomogeneous polynomial kernel of degree 2:

$$
\begin{aligned}
&x, y \in \mathbb{R}^2 \\
&k(x, y) = (x^T y + 1)^2 \\
&\varphi(x) = [x_1^2, \sqrt{2}x_1, \sqrt{2}x_1x_2, \sqrt{2}x_2, x_2^2, 1]
\end{aligned}
$$

$$
\begin{aligned}
&\varphi : \mathbb{R}^2 \rightarrow \mathbb{R}^6 \\
&\varphi(x, y) \cdot \varphi(x, y) : \mathbb{R}^6 x \mathbb{R}^6 \rightarrow \mathbb{R} \\
&k : \mathbb{R}^2 x \mathbb{R}^2 \rightarrow \mathbb{R}
\end{aligned}
$$

- Less computation, same result!

Some Common Useful Kernels

- Homogeneous Polynomial kernel: $k(x, y) = (x^T y)^d$
- Inhomogeneous Polynomial kernel: $k(x, y) = (x^T y + c)^d$
- Radial Basis Function (RBF) kernel: $k(x, y) = e^{-\frac{\|x-y\|^2}{2\sigma^2}}$

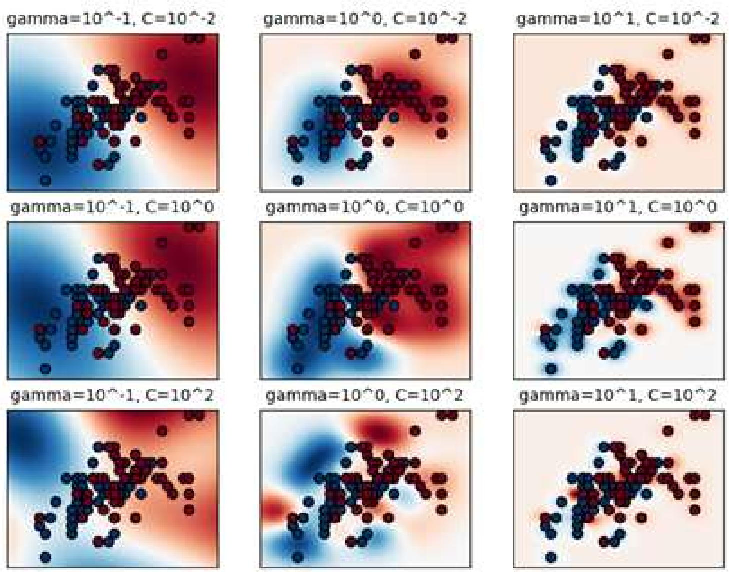
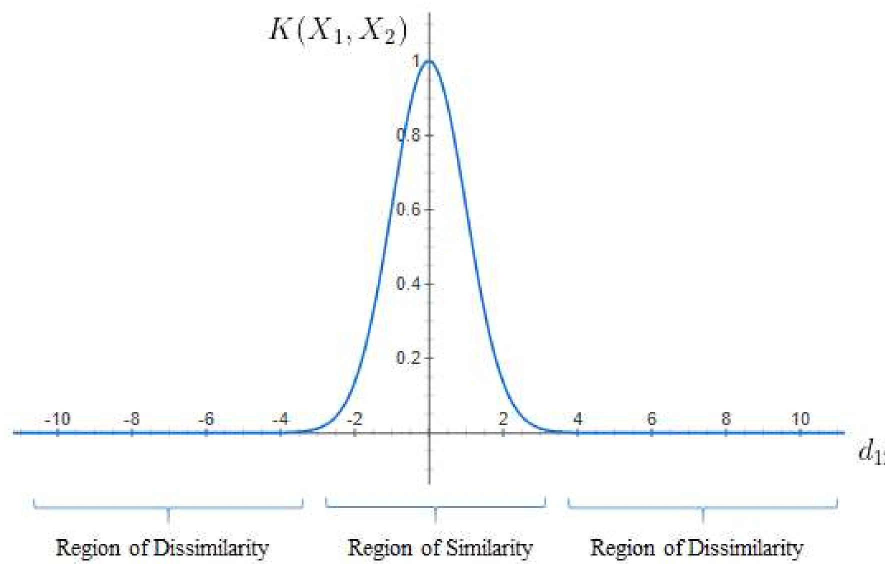


Fig 6: RBF Kernel SVM for Iris Dataset [Image Credits: https://scikit-learn.org/]



Additional notes

- Some kernel arithmetic:

$$
\begin{aligned}
&(1)\ k(x,y) = k_1(x,y) + k_2(x,y) \\
&(2)\ k(x,y) = a k_1(x,y) \quad \text{where } a > 0 \\
&(3)\ k(x,y) = f(x) \cdot f(y) \text{ for any function } f \text{ on } x \\
&(4)\ k(x,y) = k_1(x,y) \ \cdot \ k_2(x,y) \\
&(5)\ k(x,y) = \frac{k_1(x,y)}{\sqrt{k_1(x,x)}\sqrt{k_1(y,y)}}
\end{aligned}
$$

- Not all kernels are useful

Implementation

Implementation – SVM Kernel Classifier

- Classifier class usage example:

```python
print(f'[Main] Info: Building SVM model with C={C}, sigma={sigma}')
model = KernelSvmClassifier(C=C, kernel=RBF)
model.fit(X_train, y_train, DEBUG=DEBUG)

print('[Main] Info: Test model on test data')
test_prediction = model.predict(X_test, DEBUG=DEBUG)

print('[Main] Info: Finished prediction!  Test results:')
run_results_df = data_helpers.assess_accuracy(test_prediction, y_test)
```

- Kernel function:

```python
def RBF(x1, x2):
    """"RBF kernel """
    diff = x1 - x2
    return np.exp(-1 * (diff @ diff) / (2 * (sigma**2)))
```

```python
def poly(x1, x2):
    """"Poly kernel"""
    return (x1 @ x2 + 1) ** poly_rank
```

Implementation – SVM Kernel Classifier (Constrained)

- Fit() method – cost function:

```
# Create Gram matrix of k(x) y:
gramXX = np.apply_along_axis(lambda x1: np.apply_along_axis(lambda x2: self.kernel(x1, x2), 1, X), 1, X)  # 2 for loops.
yp = y.reshape(-1, 1)
yy = yp @ yp.T
gramXXyy = gramXX * yy
```

```
# Lagrange dual objective function (to maximize!)
def ld_obj(gram, alpha):
    return alpha.sum() - 0.5 * alpha @ (alpha @ gram)

# Partial derivative of ld_obj on alpha
def d_ldobj_d_alpha(gram, alpha):
    return np.ones_like(alpha) - alpha @ gram

# cost function and its gradient - to minimizes
def cost_func(a):
    return -ld_obj(gramXXyy, a)

def cost_grad(a):
    return -d_ldobj_d_alpha(gramXXyy, a)
```

$$Min_{\alpha}\left(\frac{1}{2}\sum_{i=1}^{n}\sum_{j=1}^{n}\underbrace{\alpha_i\alpha_j}\underbrace{y_i y_j k(x_i, x_j)}\right) - \underbrace{\sum_{i=1}^{n}\alpha_i}$$

Matrix writing:

$$Min_{\alpha}\left(\frac{1}{2}\underbrace{\alpha^T(\alpha^T}\cdot\underbrace{Gram_k)}\right) - \underbrace{\alpha^T\cdot\begin{bmatrix}1\\ \vdots\\ 1\end{bmatrix}}$$

Implementation – SVM Kernel Classifier (Constrained)

- Fit() method – constraints and (scipy) solver

```
# Constraints:
A = np.vstack((-np.eye(N), np.eye(N)))
b = np.hstack((np.zeros(N), self.C * np.ones(N)))
constraints = ({'type': 'eq', 'fun': lambda a: np.dot(a, y), 'jac': lambda a: y},          # y @ alpha = 0
               {'type': 'ineq', 'fun': lambda a: b - np.dot(A, a), 'jac': lambda a: -A})   # 0 <= alpha <= C
```

```
# Minimizing the **negative** dual
results = optimize.minimize(fun=cost_func,
                            x0=np.ones(N),
                            method='SLSQP',
                            jac=cost_grad,
                            constraints=constraints)
self.alpha = results.x
```

```
# store support vectors and y*alpha vals to use in inference
epsilon = 1e-8  # Max distance to become a support vector
supportIndices = self.alpha > epsilon
self.supportVectors = X[supportIndices]
self.supportAlphaY = y[supportIndices] * self.alpha[supportIndices]
```

Implementation – SVM Kernel Classifier (Constrained)

- Predict() method – note it only uses the support vectors:

$$Sign\left(\sum_{SV} k(x, SV_i) \cdot \alpha_i y_i \right)$$

```python
def predict(self, X, DEBUG=False):
    """ Predict y values in {-1, 1} """
    if DEBUG:
        print(f'[KernelSvmClassifier] Debug: predict() started --------------------')
        print(f'[KernelSvmClassifier] Debug: X.shape: {X.shape}')

    def predict_sample(x):
        x1 = np.apply_along_axis(lambda s: self.kernel(s, x), 1, self.supportVectors)
        # Calc kernel of the sample with each support vector, return vectorized results
        x2 = x1 * self.supportAlphaY # Multiply by alpha*y of each vector
        return np.sum(x2)

    d = np.apply_along_axis(predict_sample, 1, X) # for each vector in X, run predict_sample
    return 2 * (d > 0) - 1
```

Some Results

Results on Actual Datasets – Epileptic Seizure Predictin

- Dataset:
  - Identify epileptic seizure based on brainwave data from 178 sensors
  - 178 attributes, binary target (Seizure / no seizure)
  - Train size: 800, Test size: 15,000

- Linear kernel with slack: Failed to converge

- RBF Kernel

Column1	C	sigma	% Success	% False neg	% False pos
3	0.1	0.1	89.64	0.28	10.08
10	1	0.1	89.64	0.28	10.08
17	10	0.1	89.64	0.28	10.08
18	10	0.5	85.08	0.06	14.86
11	1	0.5	82.79	0.03	17.19

- Polynomial kernel

Column1	C	Polynom rank	% Success	% False neg	% False pos
1	100	3	79.95	0.00	20.05
2	100	5	79.95	0.00	20.05
3	100	10	79.95	0.00	20.05
4	100	15	79.95	0.00	20.05
6	10	3	79.95	0.00	20.05

Results on Actual Datasets – Network Attack Prediction

- Dataset:
  - Predict whether a network transaction is an attack based on technical attributes – protocol, ports, packet attributes and more
  - 41 attributes, binary target (attack true / false - **Union of all attack types**)
  - Train size: 800, Test size: 15,000

- Linear kernel with slack:

Column1	C	Polynom rank	% Success	% False neg	% False pos
3	0.1		96.82	2.45	0.73
4	0.01		96.63	2.61	0.75
2	1		87.81	12.06	0.13
1	10		82.65	17.29	0.07
0	100		80.23	19.77	0.00

- RBF Kernel

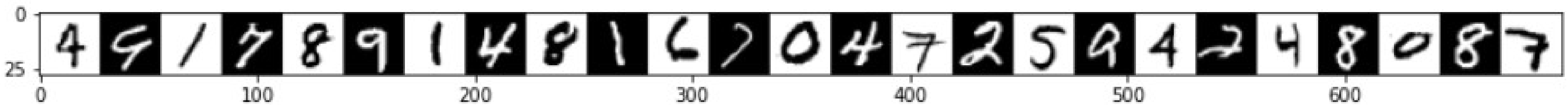
Column1	C	sigma	% Success	% False neg	% False pos
24	1	0.1	99.21	0.42	0.37
31	10	0.1	99.21	0.43	0.35
18	0.1	0.5	99.18	0.04	0.78
26	1	0.75	99.17	0.11	0.71
34	10	1	99.05	0.14	0.81

- Polynomial kernel

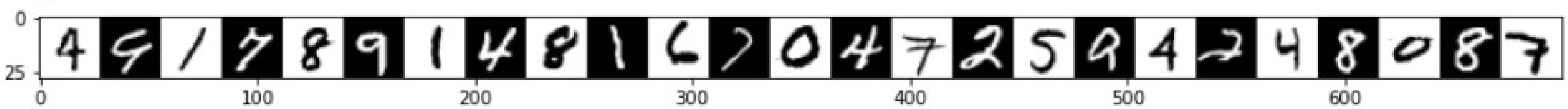
Column1	C	Polynom rank	% Success	% False neg	% False pos
1	100	3	99.27	0.41	0.32
8	10	10	98.84	0.29	0.87
18	0.1	10	98.63	0.27	1.10
15	0.1	1	96.81	2.45	0.73
20	0.01	1	96.63	2.61	0.75

Altered Digits Dataset – One vs All

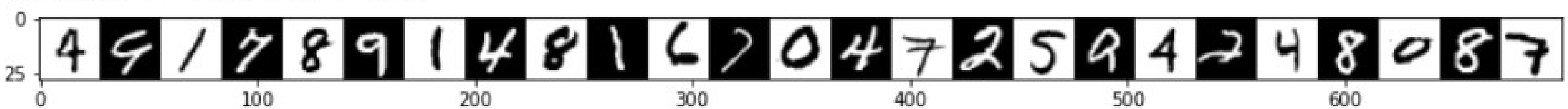
```
Linear kernal results:
# support vectors: (2896, 784)
Training set:  {'Success': 1.0, 'False Positive': 0.0, 'False Negative': 0.0}
Test set:      {'Success': 0.1864406779661017, 'False Positive': 0.8135593220338984, 'False Negative': 0.8135593220338984}
<Figure size 640x480 with 0 Axes>
```




```
[7 3 3 7 4 0 1 9 3 1 6 9 2 0 3 0 0 1 5 3 9 6 8 0 7]
Polinomial kernal results:
# support vectors: (2054, 784)
Training set:  {'Success': 0.8536666666666667, 'False Positive': 0.14633333333333334, 'False Negative': 0.14633333333333334}
Test set:      {'Success': 0.7867381214902947, 'False Positive': 0.21326187850970538, 'False Negative': 0.21326187850970538}
<Figure size 640x480 with 0 Axes>
```




```
[4 9 1 1 8 9 1 4 8 1 6 1 0 4 4 2 5 9 4 1 4 8 0 8 7]
rbf kernal results:
# support vectors: (2071, 784)
Training set:  {'Success': 0.95, 'False Positive': 0.05, 'False Negative': 0.05}
Test set:      {'Success': 0.9067411984922691, 'False Positive': 0.09325880150773097, 'False Negative': 0.09325880150773097}
<Figure size 640x480 with 0 Axes>
```




```
[4 9 1 7 8 9 1 4 8 1 6 2 0 4 4 2 5 9 4 2 4 8 4 8 7]
```