Support Vector Machines

Emma Lathouwers, Mahdieh Malekian, Ish Mukul, Ulas Ozdemir

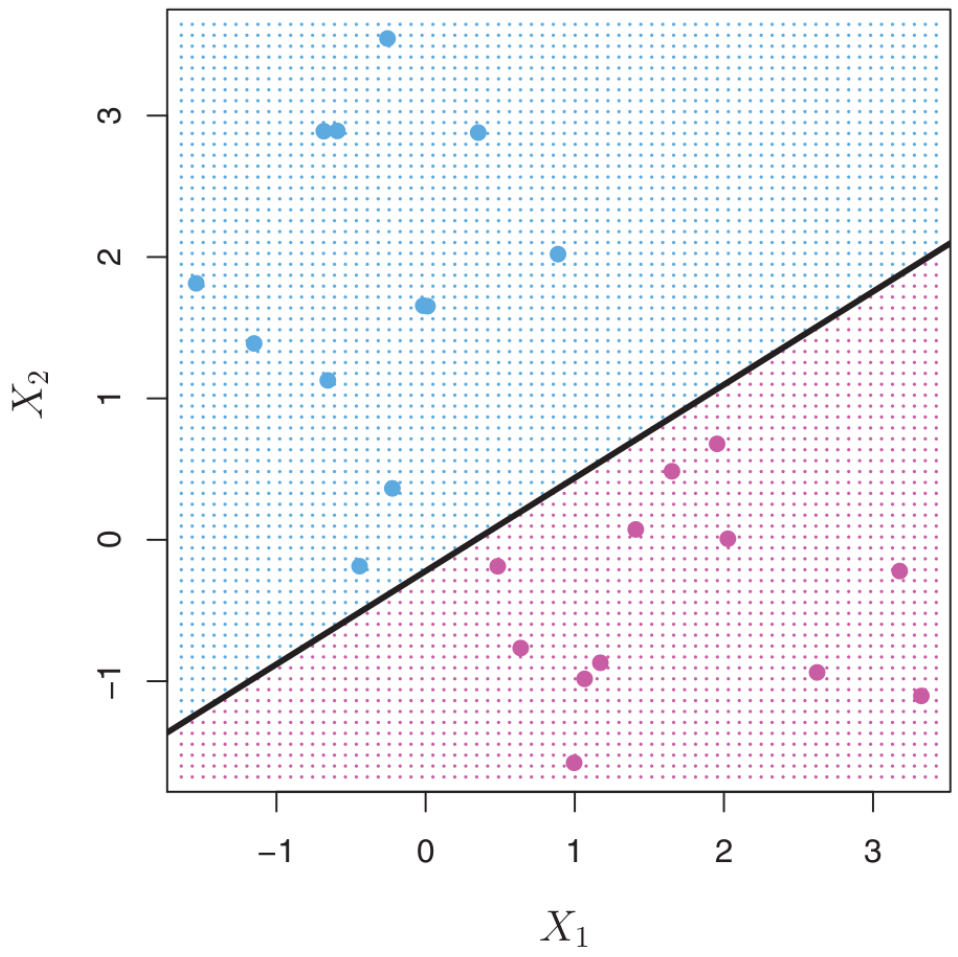
AQM

March 4, 2020

Outline

- SVM Theory (Ulas)
- Code Walkthrough (Ish)
- Results and Discussion (Emma)

Hyperplane Classification



- N training examples

$$\{(\mathbf{x}_1, y_1), (\mathbf{x}_2, y_2), \ldots, (\mathbf{x}_N, y_N)\}$$

$$y_i \in \{\pm 1\}$$

- Separating hyperplane

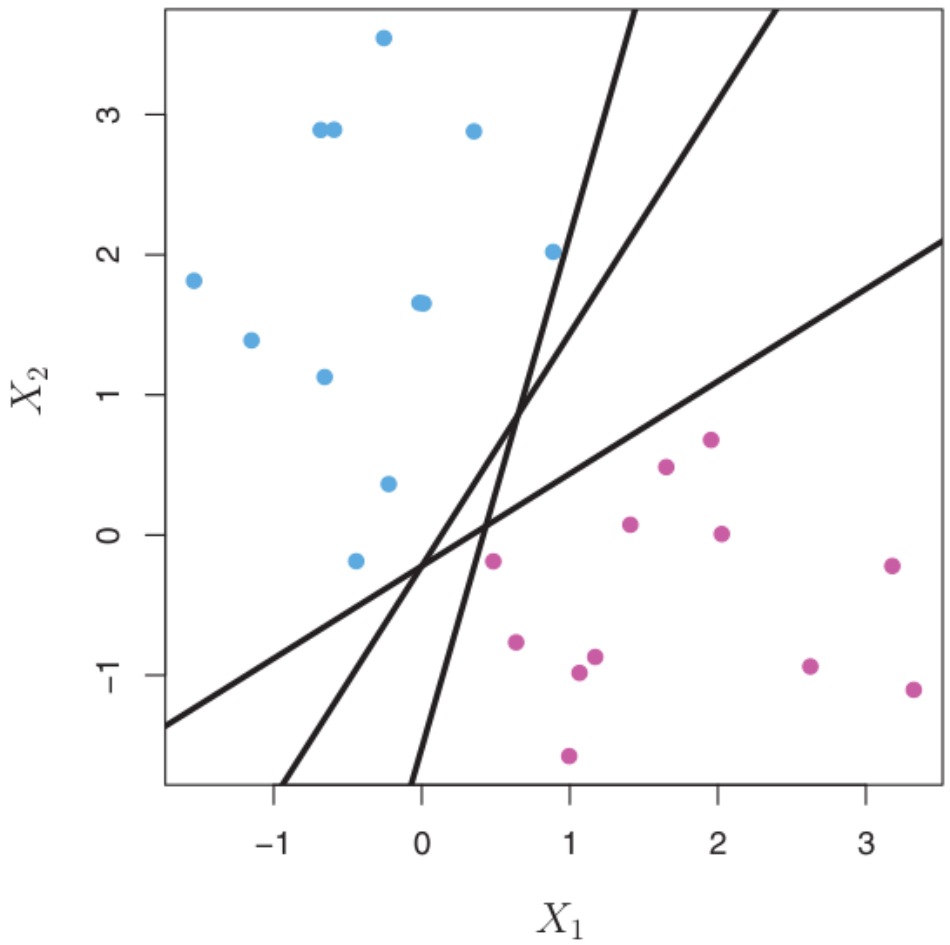
$$f(x) = x^T\beta + \beta_0 = 0$$

- Classification

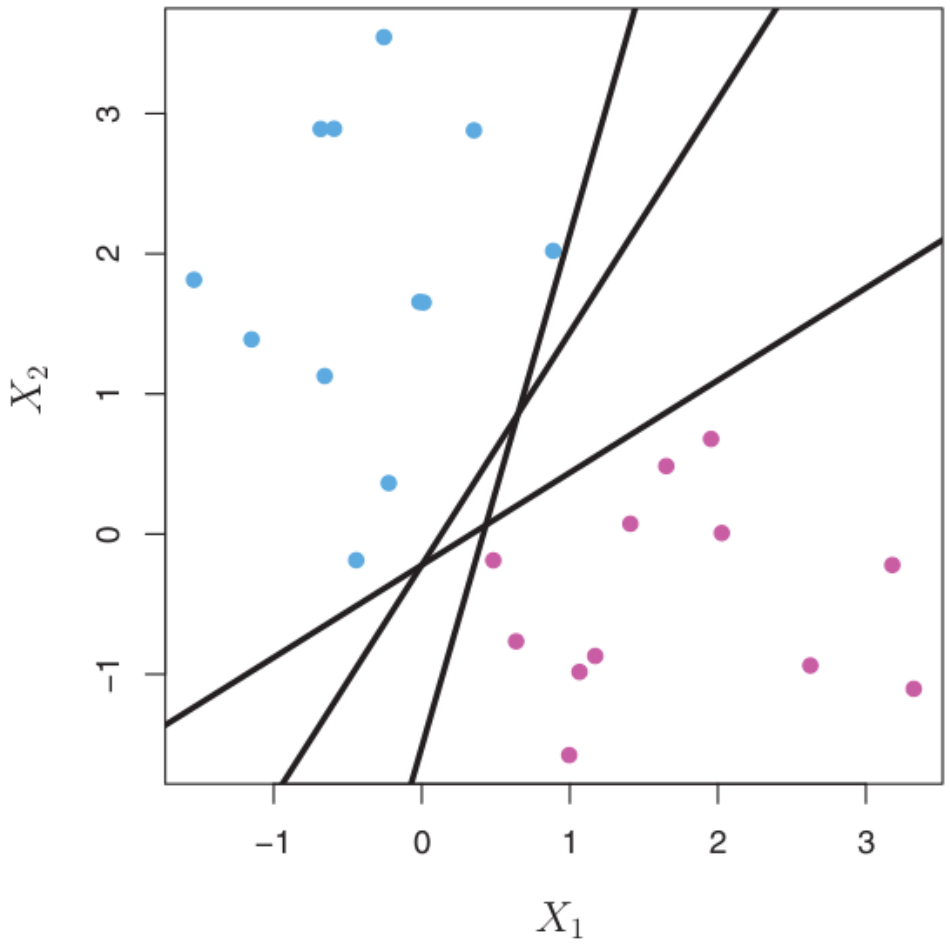
$$\beta_0 + \beta_1 x_{i1} + \beta_2 x_{i2} + \ldots + \beta_p x_{ip} > 0 \text{ if } y_i = 1$$

$$\beta_0 + \beta_1 x_{i1} + \beta_2 x_{i2} + \ldots + \beta_p x_{ip} < 0 \text{ if } y_i = -1$$

Hyperplane Classification



Hyperplane Classification

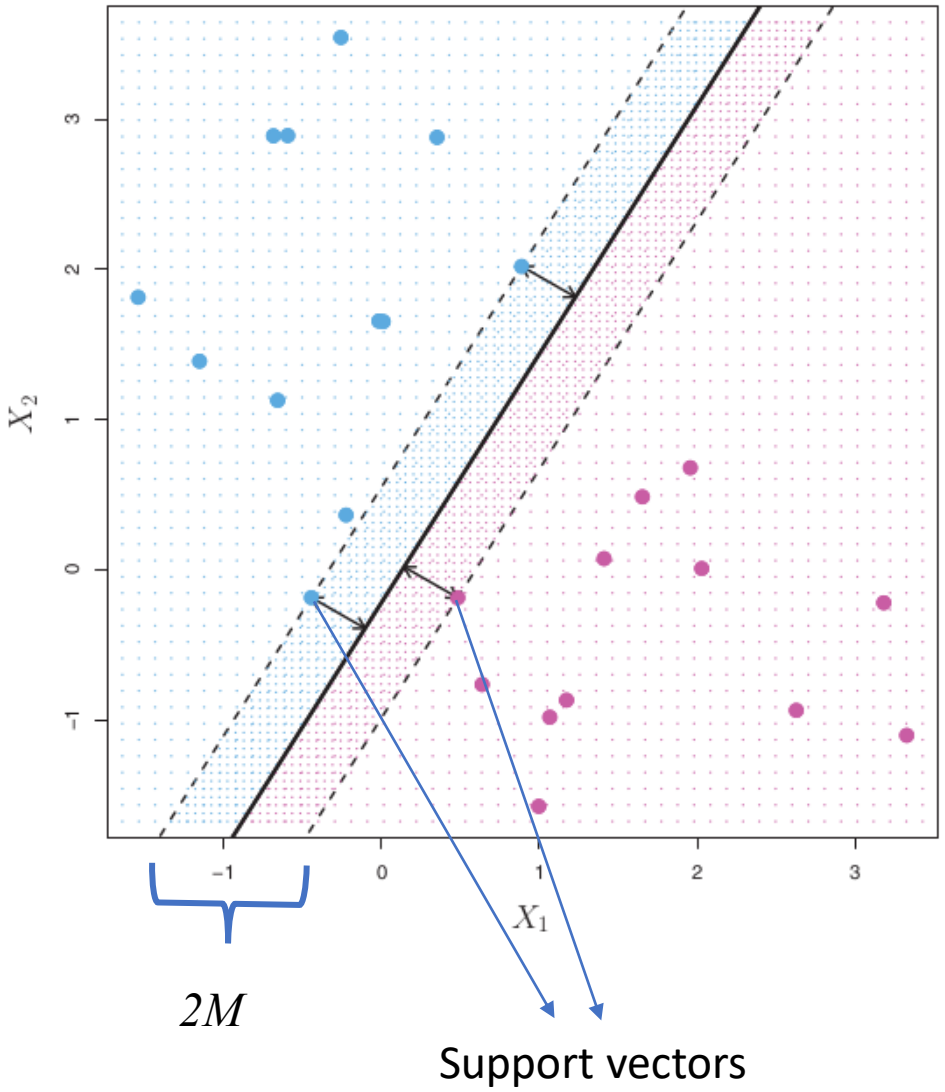


- Infinite numbers of hyperplanes.
  - Hard to predict the best separating plane.

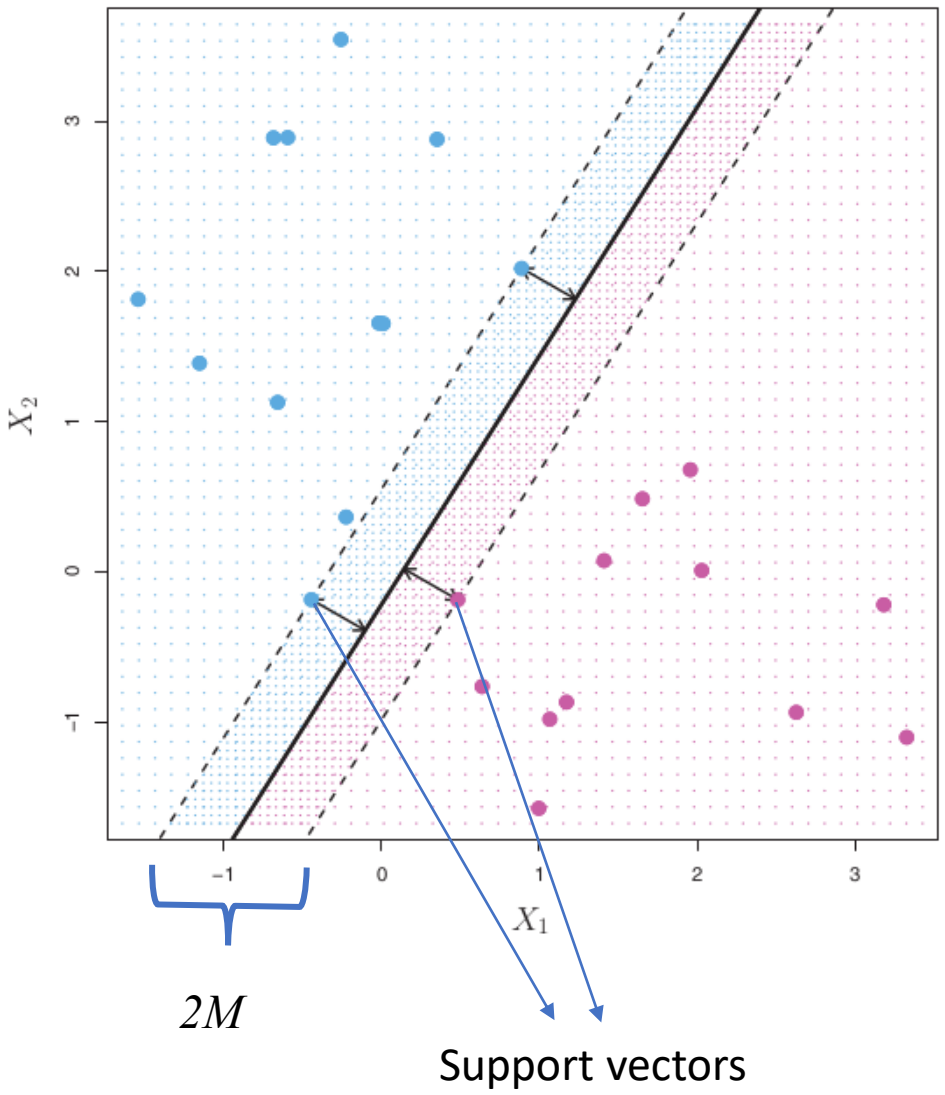
$$f(x) = x^T\beta + \beta_0 \text{ with } y_i f(x_i) > 0 \forall i$$

- Does not work well for the test set.
  - Solution only exists for separable case.

Maximal Margin Classifier (Hard Margin)



Maximal Margin Classifier (Hard Margin)

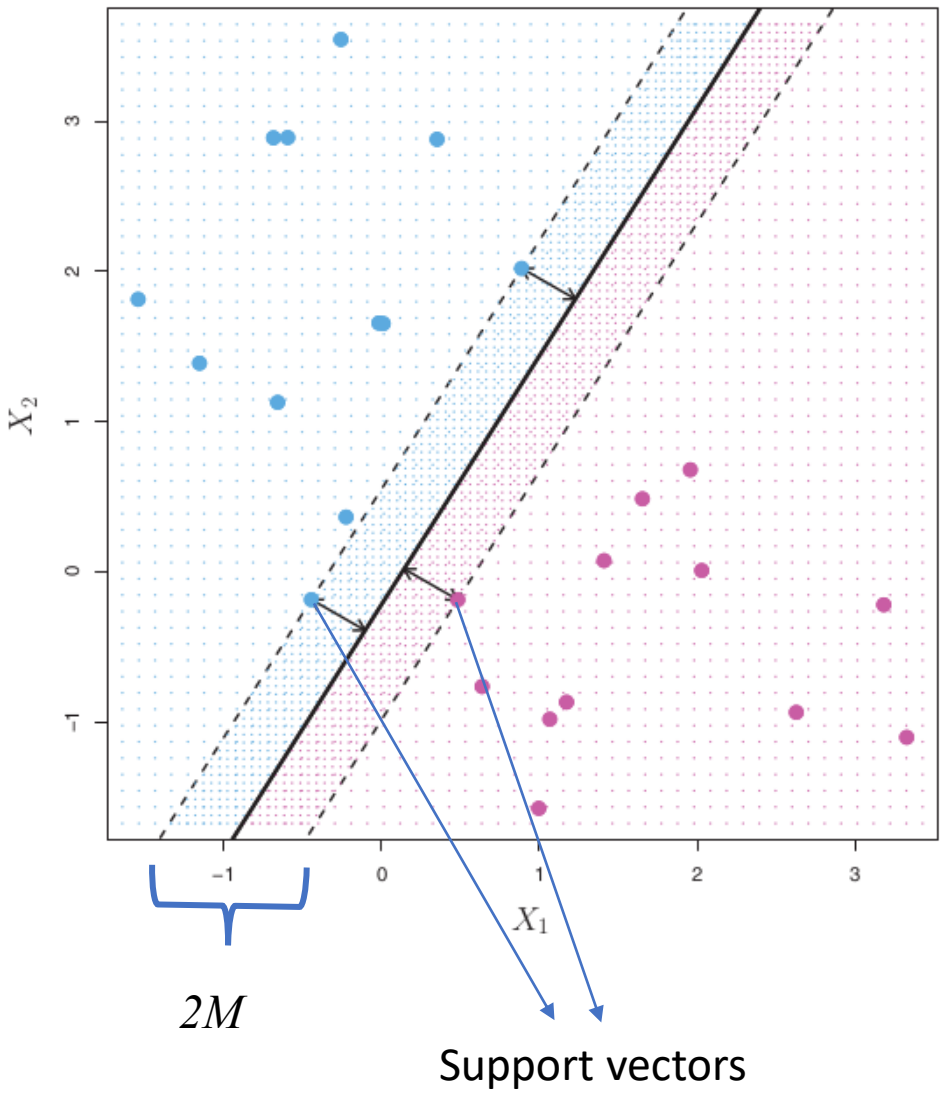


- A special case of SVM
- Separating hyperplane is the farthest from the training set

$$\max_{\beta,\beta_0,\|\beta\|=1} M$$

$$\text{subject to } y_i\left(x_i^T\beta+\beta_0\right)\geq M, i=1,\ldots,N$$

Maximal Margin Classifier (Hard Margin)

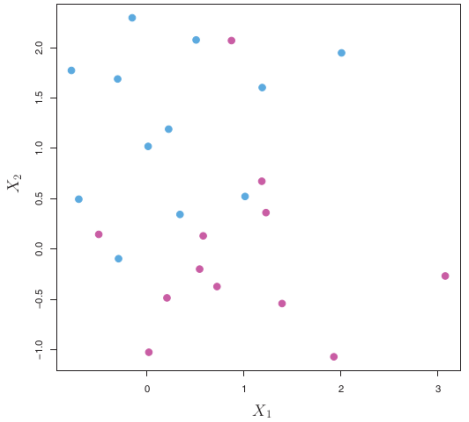


- A special case of SVM
- Separating hyperplane is the farthest from the training set

$$\max_{\beta,\beta_0,\|\beta\|=1} M$$

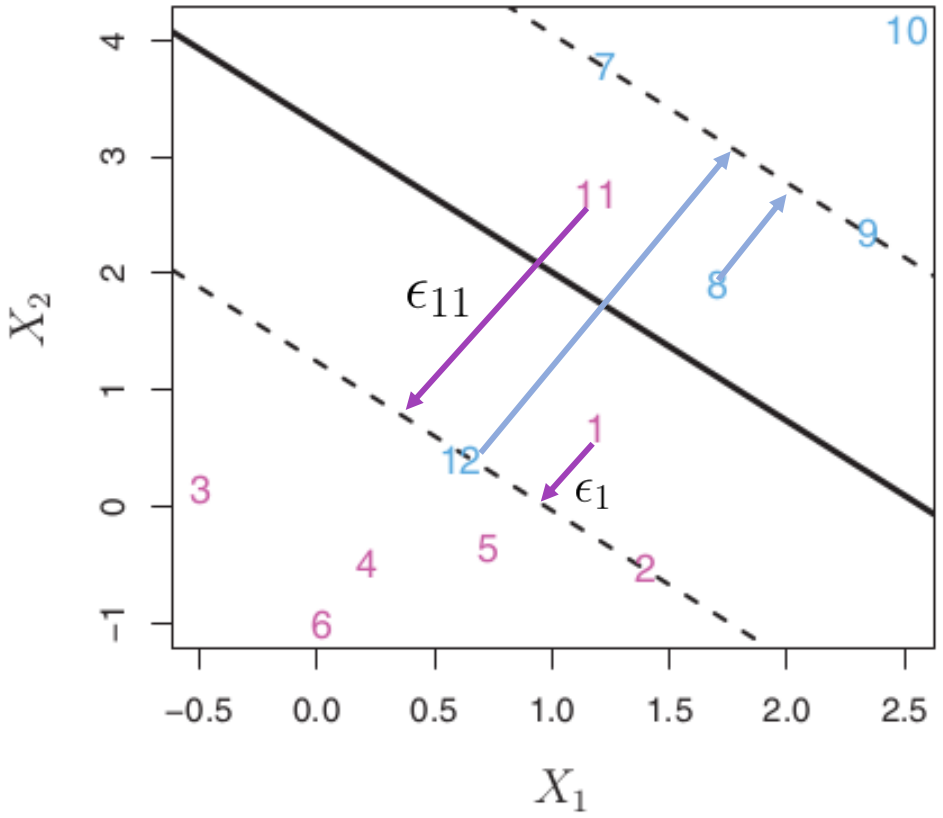
$$\text{subject to } y_i\left(x_i^T\beta+\beta_0\right)\geq M, i=1,\ldots,N$$

- Sensitive to individual observations
- No solution for non-separable case.



- Need a more robust classifier for each observation
- Better classification of most of the training observations

Support Vector Classifier (Soft Margin)



- Allow some points to be classified incorrectly.

$$\underset{\beta_0,\beta_1,\ldots,\beta_p,\epsilon_1,\ldots,\epsilon_n,M}{\text{maximize}} \; M$$

$$\text{subject to} \sum_{j=1}^{p} \beta_j^2 = 1$$

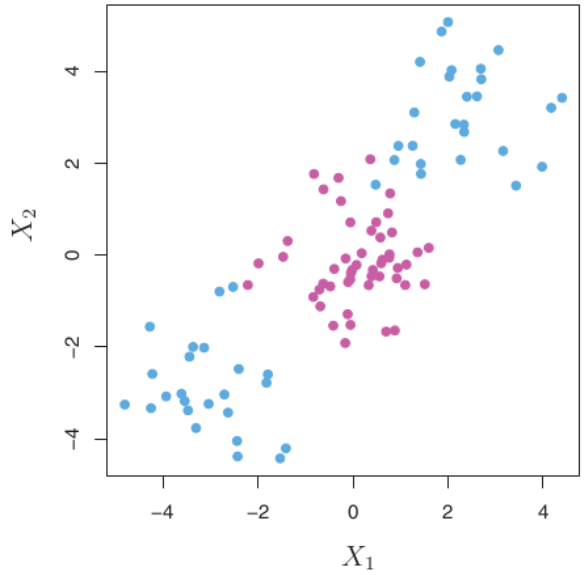
$$y_i(\beta_0 + \beta_1 x_{i1} + \beta_2 x_{i2} + \ldots + \beta_p x_{ip}) \geq M(1 - \epsilon_i)$$

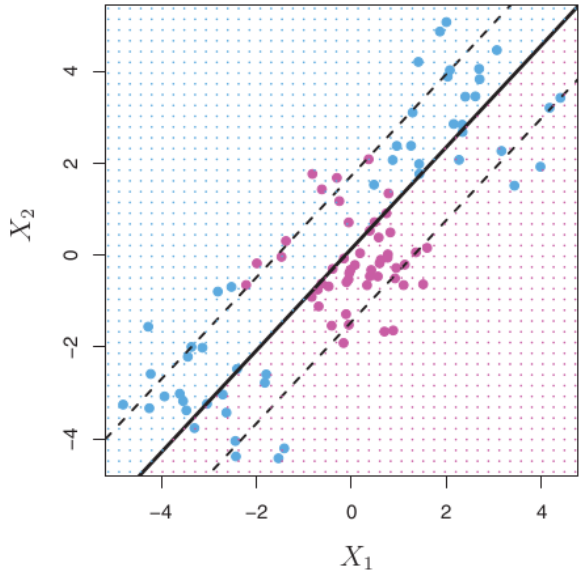
$$\epsilon_i \geq 0, \sum_{i=1}^{n} \epsilon_i \leq C$$

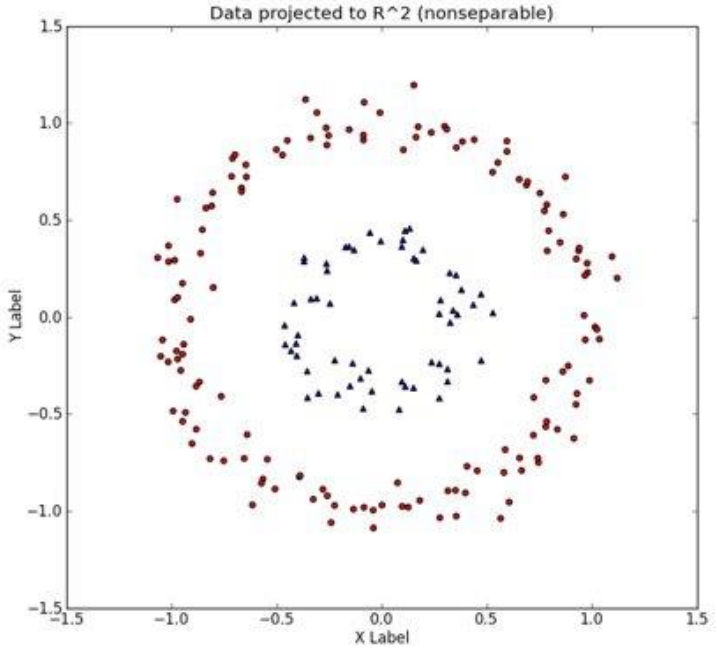
ϵ_i = Slack variable

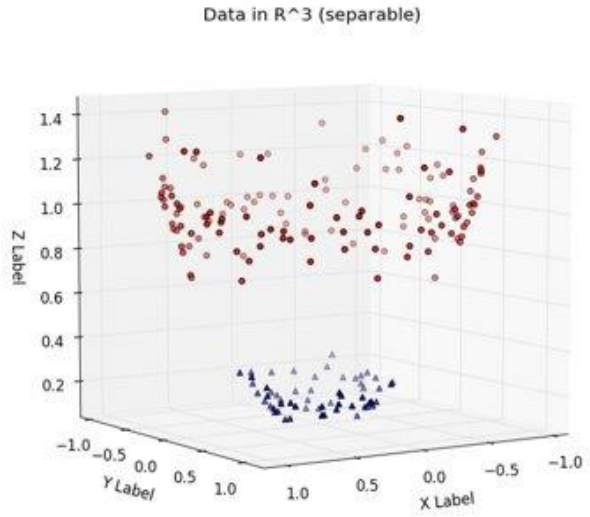
C = Tuning parameter

Non Linear Decision Boundary









- Enlarge the feature space

$$X_1, X_2, \ldots, X_p \longrightarrow X_1, X_1^2, X_2, X_2^2, \ldots, X_p, X_p^2$$

- Map all decision boundary to a space where linear separation is possible.
- Computationally expensive (Large feature set)
- Can lead to overfitting
- Not standardized -> Different combinations features

Use a kernel function

- Can be precomputed
- Smaller feature size
- Polynomial kernel

$$K(\mathbf{x}, \mathbf{y}) = (\mathbf{x} \cdot \mathbf{y} + 1)^p$$

- Gaussian kernel

$$K(\mathbf{x}, \mathbf{y}) = \exp\left\{-||x - y|^2 / 2\sigma^2\right\}$$

Incorporating Kernels into code

- The main difference is that we need to solve for α from modified Lagrangian dual:

$$\begin{aligned} \underset{\alpha}{\text{maximize}} \quad & L_D = \sum_{i=1}^{N} \alpha_i - \frac{1}{2} \sum_{i=1}^{N} \sum_{j=1}^{N} \alpha_i \alpha_j y_i y_j K(x_i, x_j) \\ \text{subject to} \quad & 0 \leq \alpha_i \leq C, \\ & \sum_{i=1}^{N} \alpha_i = 0 \end{aligned}$$

- This is a quadratic programming which cvxopt solves, except that cvxopt solves equations of form:

$$\begin{aligned} \underset{\alpha}{\text{minimize}} \quad & \frac{1}{2} \alpha^T P \alpha + q^T \alpha \\ \text{subject to} \quad & G\alpha \preceq h, \\ & A\alpha = b \end{aligned}$$

- Need to choose P, q, G, h, A, b so that Lagrangian dual turns into the cvxopt form.

$$P = \begin{pmatrix} \vdots & \\ \cdots & y_i y_j K(x_i, x_j) \quad \cdots \\ \vdots & \end{pmatrix}_{N \times N} \qquad q = \begin{pmatrix} -1 \\ \vdots \\ -1 \end{pmatrix}_{N \times 1}$$

$$G = \begin{pmatrix} -I_{N \times N} \\ \hline I_{N \times N} \end{pmatrix}_{2N \times N} \qquad h = \begin{pmatrix} 0 \\ \vdots \\ 0 \\ \hline C \\ \vdots \\ C \end{pmatrix}_{2N \times 1}$$

$$A = y^T \quad b = 0$$

Code Walkthrough

```
def compute_cost(ww, xx, yy):
    # calculate hinge loss
    distances = 1 - yy * (xx @ ww)
    distances[distances < 0] = 0  # equivalent to max(0, distance)
    hinge_loss = C * (distances.sum() / xx.shape[0])

    # calculate cost
    cost = 0.5 * (ww @ ww) + hinge_loss
    return cost


def calculate_cost_gradient(ww, xx, yy):
    dw = np.zeros(len(ww))
    distance = 1 - (yy * (xx @ ww))
    if distance <= 0:
        di = ww
    else:
        di = ww - (C * yy * xx)
    dw += di
    return dw
```

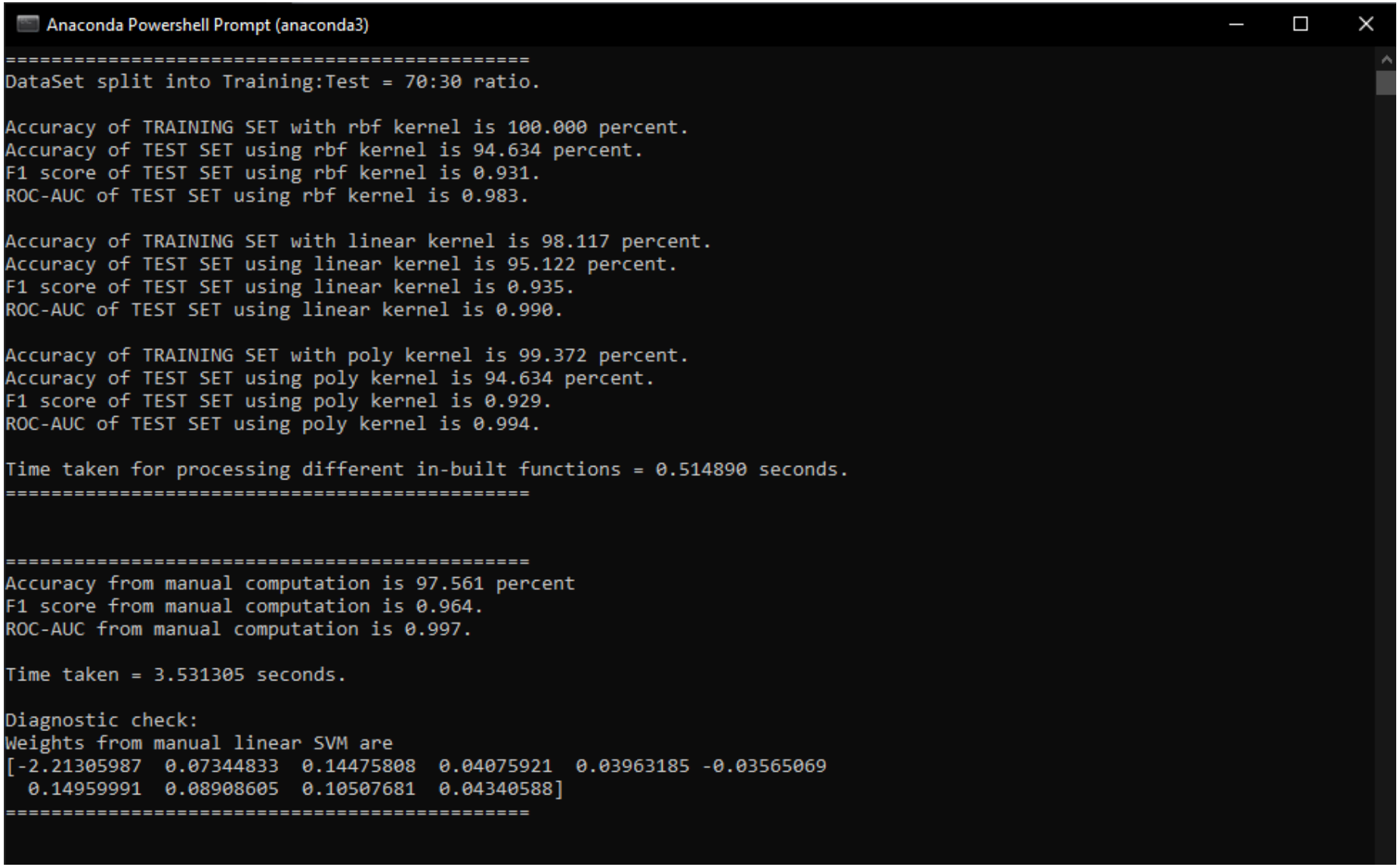
Code Walkthrough

```
def gradient_descent(xx, yy, ww, max_iters):
    costs = []
    # stochastic gradient descent
    for epoch in range(1, max_iters):
        for i in range(xx.shape[0]):
            ascent = calculate_cost_gradient(ww, xx[i], yy[i])
            ww = ww - (learning_rate * ascent)

        cost = compute_cost(ww, xx, yy)
        costs.append(cost)

    return costs, ww
```

Code Walkthrough

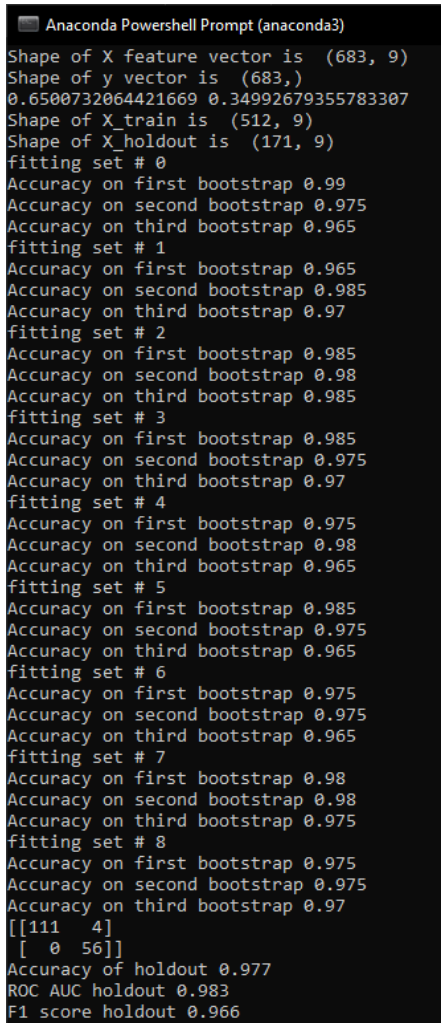
```
start_time = timer()
print('================================================')
# Cost function, Gradient calculation and Gradient descent functions defined on the top
# Search for weights start here
Costs, Weight = gradient_descent(X_train, y_train, weights, max_epochs)

# Y prediction from fitted parameter
#  numpy sign function gives sign of output result. Since we are interested the sign of the dot product product of
#  X@w, not the actual value. Remember, +1 is 1 and -1 is 0 in our convention.
y_score_man = (X_test @ Weight)
y_test_pred_man = np.sign(X_test @ Weight)
acc_man = acc(y_test, y_test_pred_man)  # Accuracy from confusion matrix
fpr_man, tpr_man, _ = roc_curve(y_test, y_score_man)  # false positive rate, true positive rate
auc_man = roc_auc_score(y_test, y_score_man)  # ROC area under curve
f1_man = f1_score(y_test, y_test_pred_man)

end_time = timer()
time_manual = end_time - start_time
print('Accuracy from manual computation is %4.3f percent' % (100 * acc_man))
print('F1 score from manual computation is %4.3f.' % f1_man)
print('ROC-AUC from manual computation is %4.3f. \n' % auc_man)
print('Time taken = %f seconds. \n' % time_manual)
# print('Diagnostic check:')
# print('Weights from manual linear SVM are ')
# print(Weight)
print('================================================\n\n')
```

Code Walkthrough

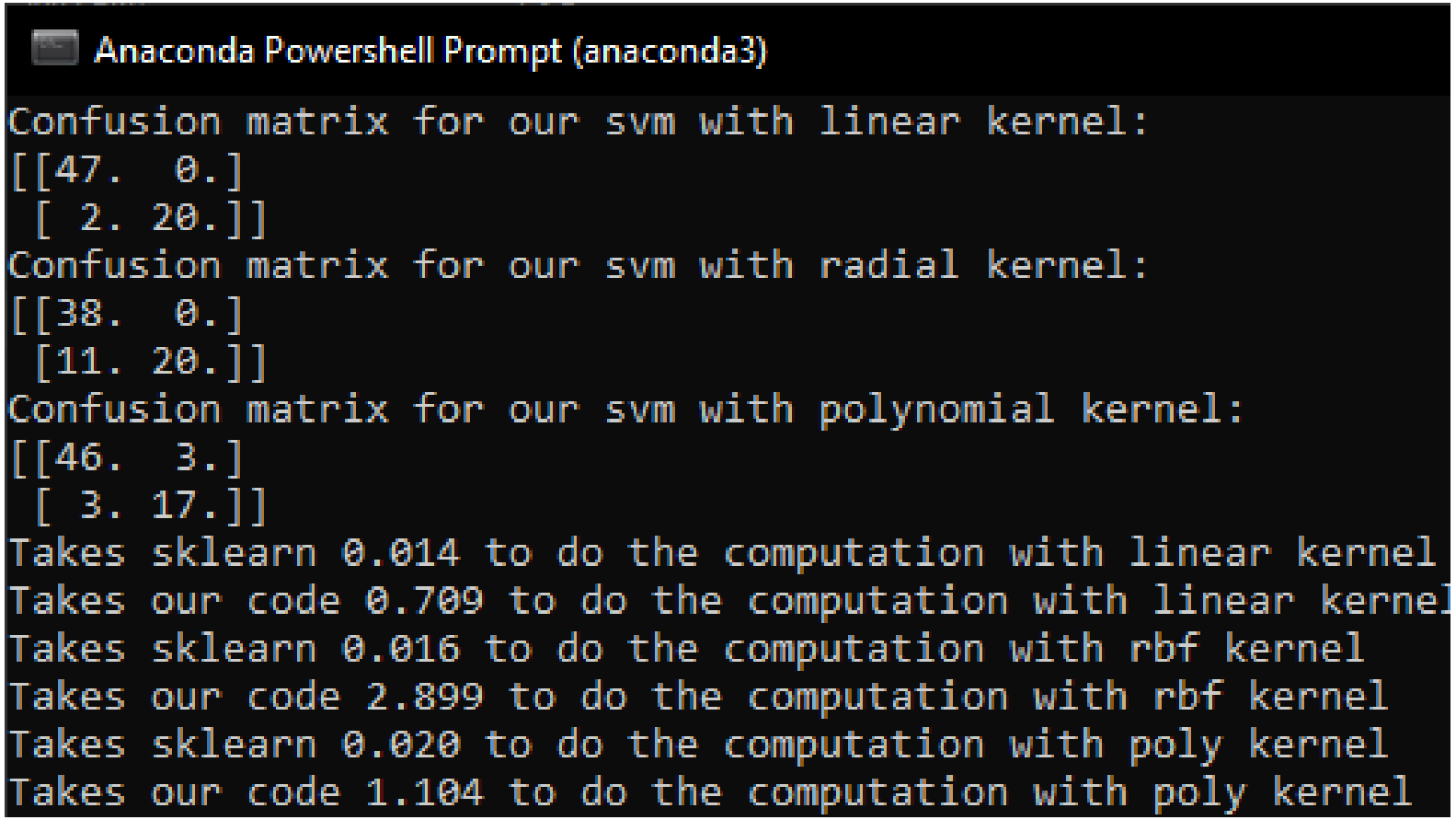


Code Walkthrough

Bootstraping



CVXOPT



Results

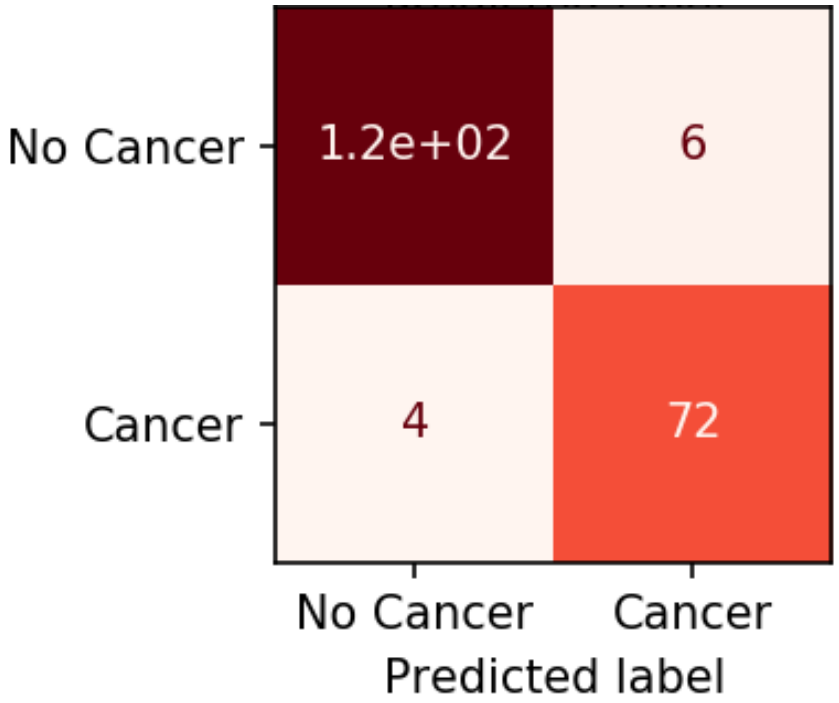
Breast cancer data set (Binary classification)

- 9 attributes
- 700 samples

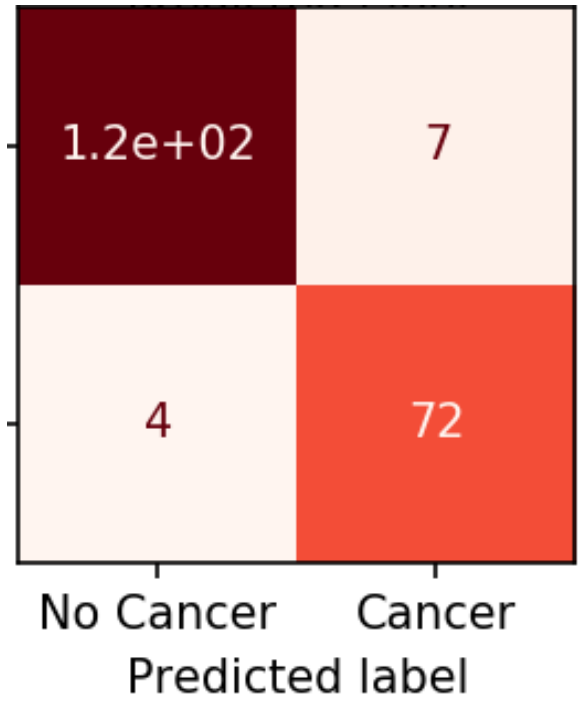
- Homebrewed code
  - Linear decision boundary
  - Stochastic gradient descent minimization
  - CVXOPT Quadratic Equation Solver
  - 70:30 validation set
- Scikit-learn
  - 3 different kernels -> RBF, Linear and Polynomial

Confusion Matrices

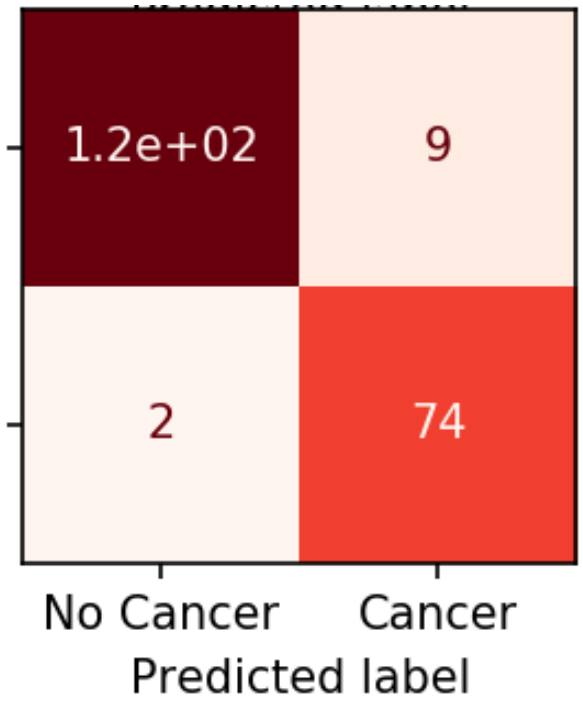
Linear Kernel



Polynomial

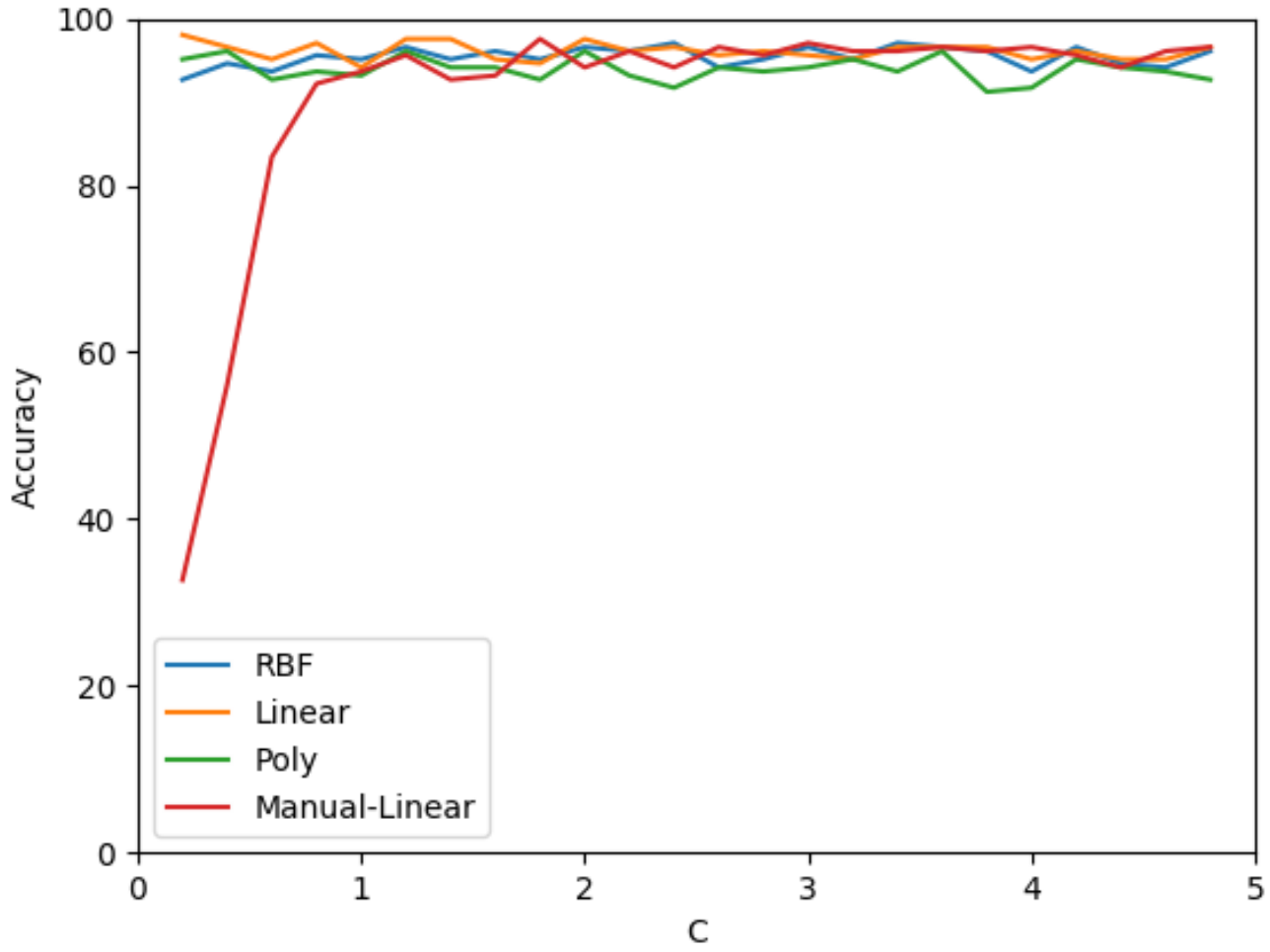


RBF



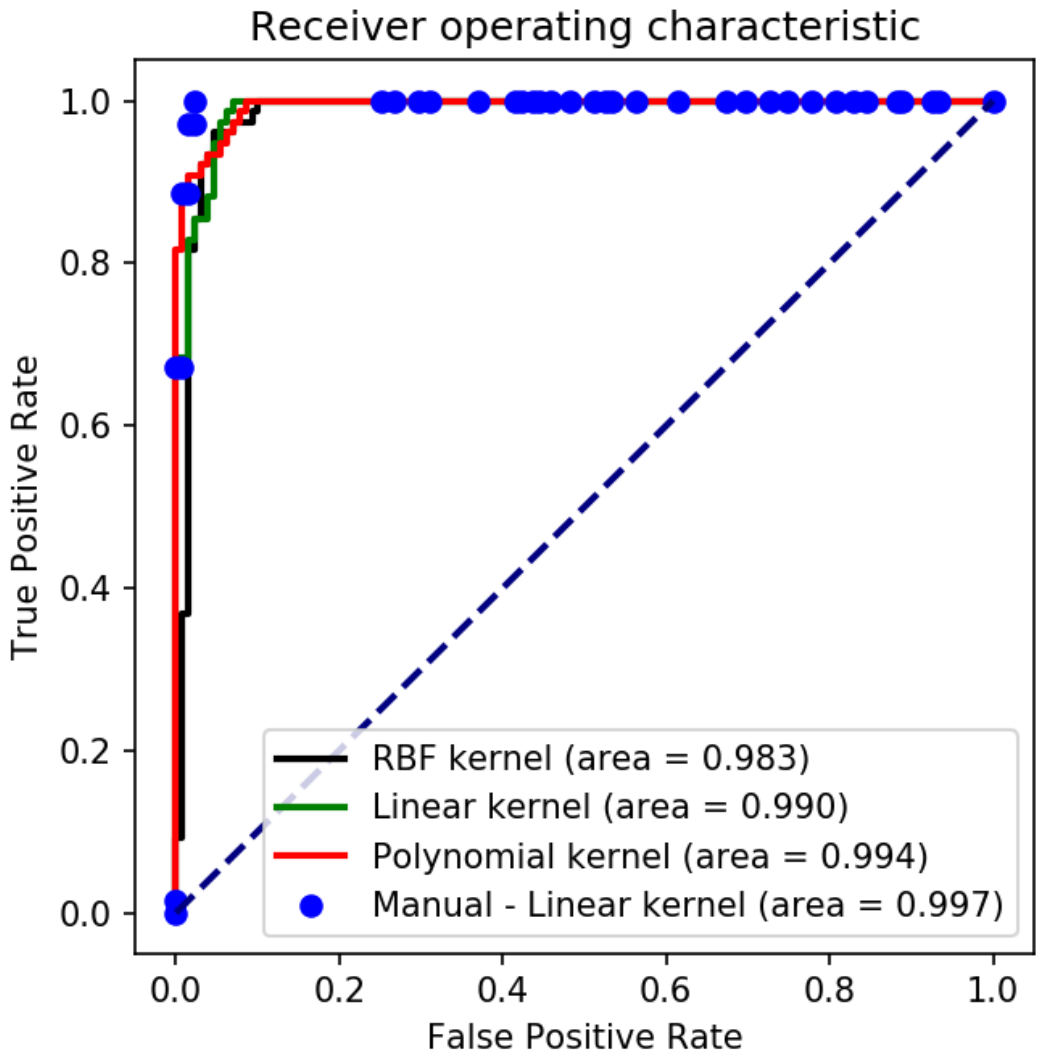
RBF classified two more data points as cancer

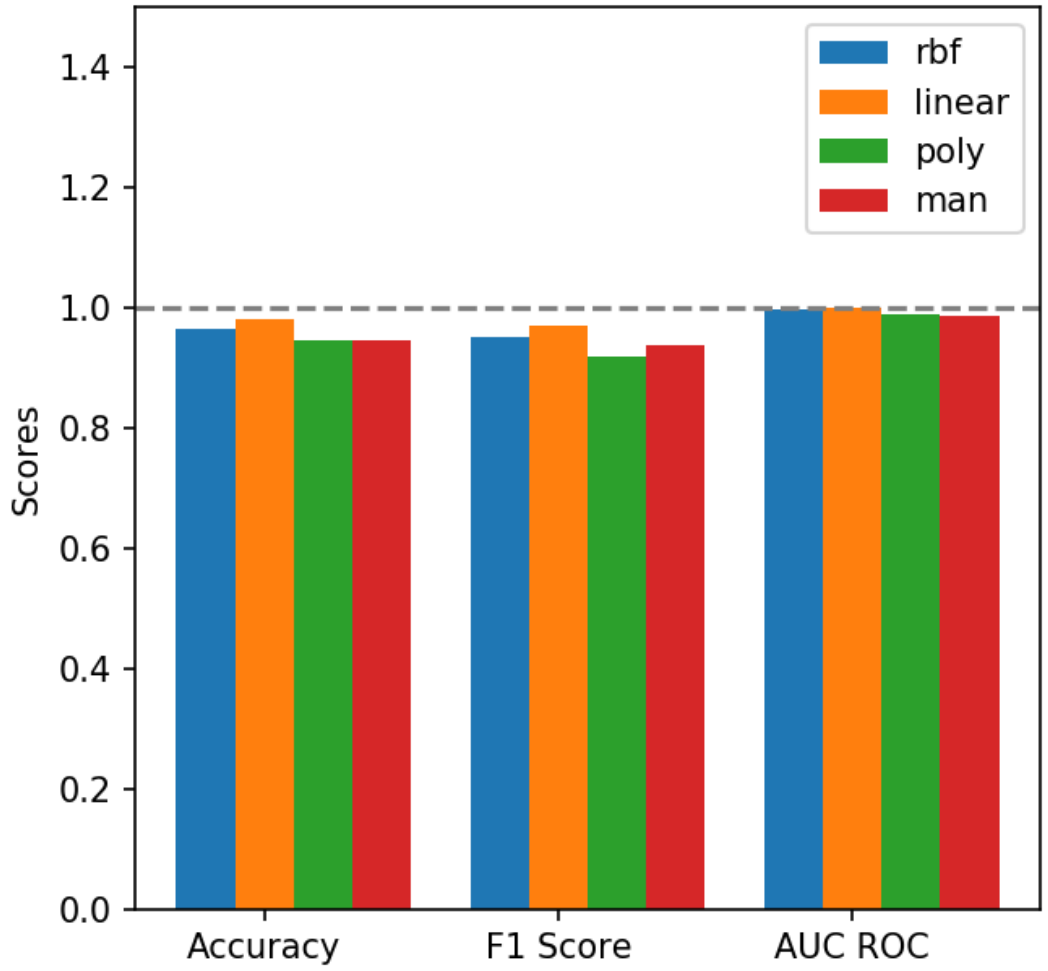
The Effect of Margin Width on Accuracy



Using a very small C value leads to lower accuracy in the homebrewed code.

Classifier Output Quality Tests





Accuracy of homebrewed linear SVM classifies as accurate as blackbox algorithms.

Shortcomings of SVM

- Non probabilistic classification
- Large number of hyperparameters
- Choosing a "good" kernel function
- Not accurate when the signal to noise ratio is low
- Long training time
- Not suitable for multiclass classification
- Extensive memory requirement of quadratic programming solver for large datasets

Technical Difficulties

- Optimization algorithm requires dual equation solver
- Kernel trick is tricky