Support Vector Machines

DS 6410 | Spring 2025

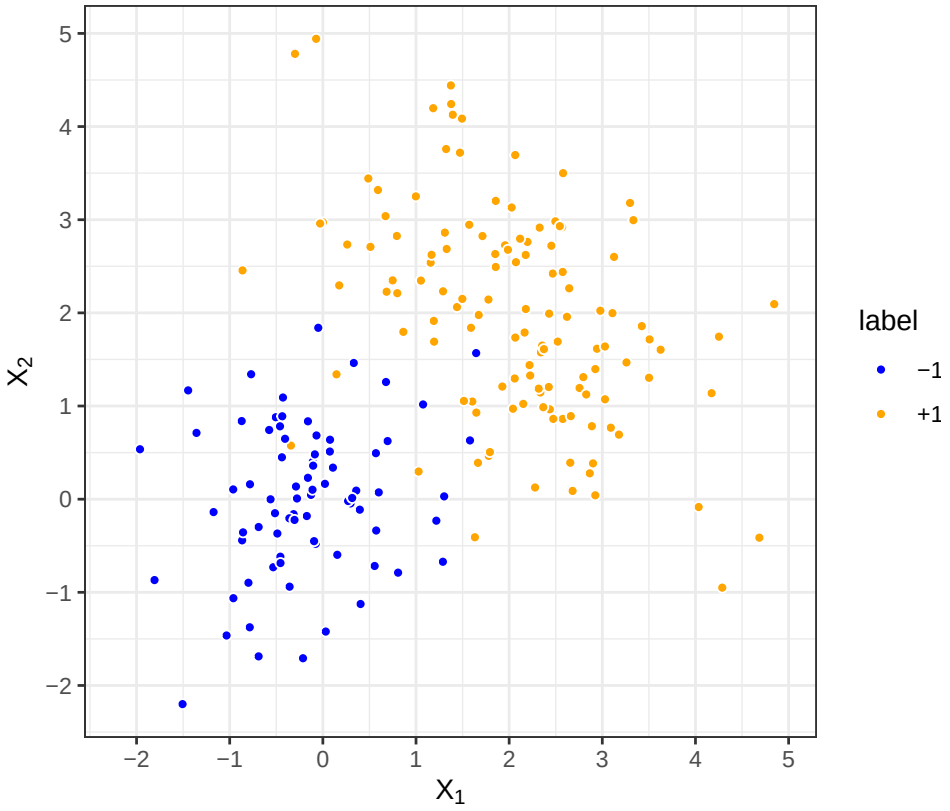
svm.pdf

Contents

1 Support Vector Machines (SVM) Introduction

1.1 Example

Goal: find *best* line(s)/curve(s) to separate the two classes.



1.2 Support Vector Machines (SVM) formulation

Training Data: $\{(\tilde{y}_i, \mathbf{x}_i)\}_{i=1}^n$

- $\tilde{y}_i \in \{-1, +1\}$
- $\mathbf{x}_i^\intercal = [x_{i1}, x_{i2}, \ldots, x_{ip}] \in \mathbb{R}^p$

Predictor Function: $f(x)$

- Linear: $f(\mathbf{x}; \beta) = \beta_0 + \sum_{j=1}^p x_j \beta_j$
- Basis Expansion: $f(\mathbf{x}; \beta) = \beta_0 + \sum_{j=1}^d h_j(\mathbf{x}) \beta_j$
  - $h_j(\mathbf{x})$ transforms the raw x vector
  - Polynomial example: $h_1(\mathbf{x}) = x_1$, $h_2(\mathbf{x}) = x_1^2$, $h_3(\mathbf{x}) = x_2$, $h_4(\mathbf{x}) = x_2^2$, $h_5(\mathbf{x}) = x_1 x_2$
  - This is the connection to *kernels* that we will discuss later
- Let $f_i = f(\mathbf{x}_i)$

Classification:

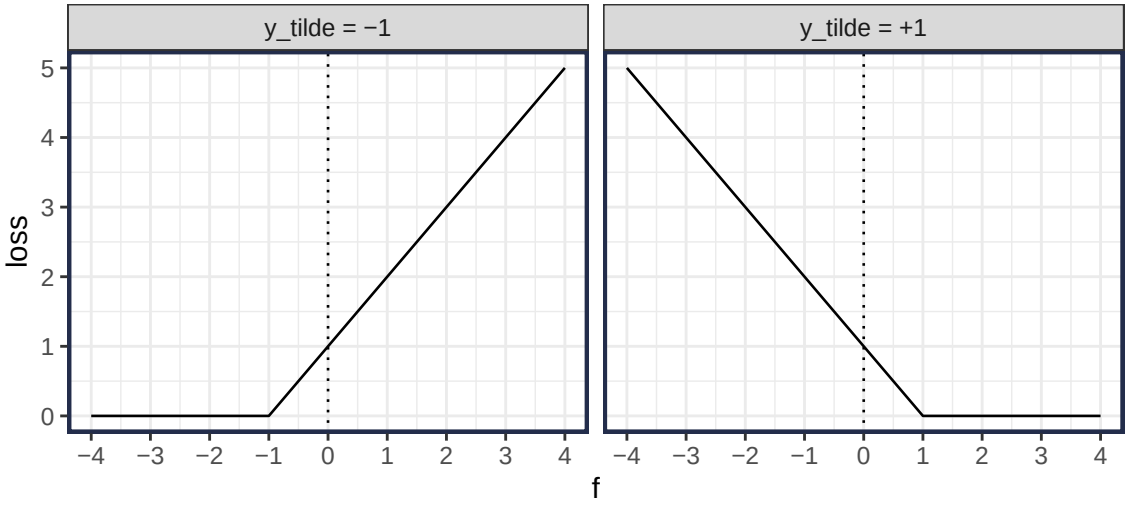
- If $\hat{f}_i \geq 0$, then label as class $+1$
- If $\hat{f}_i < 0$, then label as class -1

Note

The threshold of 0 implies an equal cost of mis-classification. It is possible to use different thresholds.

Loss Function: Hinge Loss

$$L(\tilde{y}_i, f_i) = \max\{0, 1 - \tilde{y}_i f_i\}$$



Penalty Function: Ridge Penalty

$$P_\lambda(\beta) = \frac{\lambda}{2}\sum_{j=1}^{d}\beta_j^2$$
$$= \lambda \|\beta\|^2 /2$$

- This should have you thinking that the $\mathbf{x}$'s should be scaled!

Summary of SVM

1. Estimate *model parameters* β

$$\hat{\beta}_\lambda = \arg\min_\beta \left\{ \sum_{i=1}^{n} \max\{0, 1 - \tilde{y}_i f_i(\beta)\} + \lambda \|\beta\|^2 /2 \right\}$$
$$= \arg\min_\beta \{\text{Hinge Loss}(\beta) + \lambda \text{ Penalty}(\beta)\}$$

2. Label Class $+1$ if $\hat{f}_i \geq 0$, else label Class -1
   - This implicitly assumes a 0-1 loss (or equal cost FP and FN)
   - More generally, use threshold t that considers the costs of FP and FN
3. **Tuning Parameters**: besides the λ, SVM will also have tuning parameters related to the kernels (more to come on this).

1.3 SVM vs. Logistic Regression

The *linear* SVM is actually very similar to Logistic Regression (with Ridge Penalty).

	SVM	Logistic Regression
Model Output	$f_i(\beta) = \beta_0 + \sum_{j=1}^{p} x_j\beta_j$	$f_i(\beta) = \beta_0 + \sum_{j=1}^{p} x_j\beta_j$ (logit formulation)
Loss Function	Hinge Loss(β) $\sum_{i=1}^{n} \max\{0, 1 - \tilde{y}_i f_i(\beta)\}$	Log Loss(β) $\sum_{i=1}^{n} \log\left(1 + \exp\left(-\tilde{y}_i f_i(\beta)\right)\right)$
Parameter Estimation $\hat{\beta}_\lambda$	$\arg\min_\beta \{\text{Hinge Loss}(\beta) + \lambda \text{ Penalty}(\beta)\}$	$\arg\min_\beta \{\text{Log Loss}(\beta) + \lambda \text{ Penalty}(\beta)\}$

Log-Loss

A fitted *logistic regression* model specifies the log odds as:

$$\hat{f}_i = \hat{\beta}_0 + \sum_{j=1}^{p} x_j\hat{\beta}_j$$

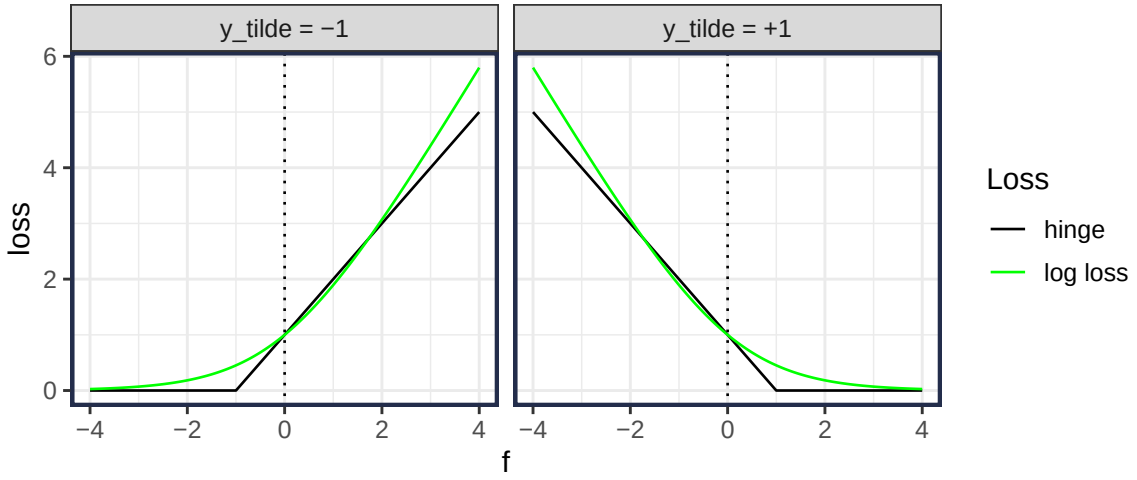
From this, we can calculate the estimated probabilities:

$$\begin{aligned}\hat{p}_i &= \frac{e^{\hat{f}_i}}{1+e^{\hat{f}_i}} \\ &= \frac{1}{1+e^{-\hat{f}_i}} \\ &= \left(1+e^{-\hat{f}_i}\right)^{-1} \\ 1-\hat{p}_i &= \left(1+e^{\hat{f}_i}\right)^{-1}\end{aligned}$$

and the log-loss function can be written:

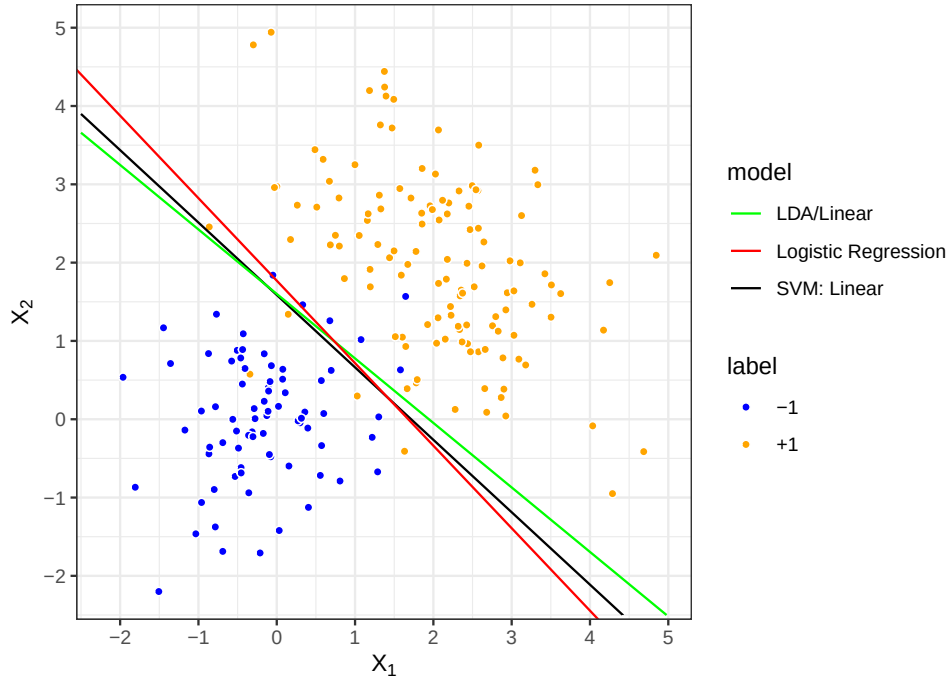
$$\begin{aligned}\text{log-Loss} &= L(y_i, \hat{p}_i) \\ &= -\left[y_i \log \hat{p}_i + (1-y_i)\log(1-\hat{p}_i)\right] \\ &= \begin{cases} -\log \hat{p}_i & y_i = 1 \\ -\log 1-\hat{p}_i & y_i = 0 \end{cases} \\ &= \begin{cases} \log(1+e^{-\hat{f}_i}) & y_i = 1 \\ \log(1+e^{\hat{f}_i}) & y_i = 0 \end{cases} \\ &= \log(1+e^{-\tilde{y}_i \hat{f}_i}) \qquad \text{where } \tilde{y}_i \in \{-1,+1\}\end{aligned}$$

Using the notation: $y_i \in \{0,1\}$ and $\tilde{y}_i = 2y_i - 1$.



The Log−Loss has been scaled so it equals the Hinge Loss at f=0.

1.4 Example: Compare Linear Classifiers



1.5 Three Representations of the Optimization Problem

1. **Penalized optimization**

$$\begin{aligned}\hat{\beta} &= \arg\min_{\beta} \{\text{Loss}(\beta) + \lambda \text{ Penalty}(\beta)\} \\ &= \arg\min_{\beta} \{C \text{ Loss}(\beta) + \text{Penalty}(\beta)\}\end{aligned}$$

   - where $C = \frac{1}{\lambda} > 0$ is an alternative strength of penalty
   - In SVM, C is referred to as the *cost*

2. **Constraint on Penalty**

$$\begin{aligned}\hat{\beta} &= \arg\min_{\beta} \ \text{Loss}(\beta) \qquad \text{subject to } \ \text{Penalty}(\beta) \leq t \\ &= \arg\min_{\beta:\ \text{Penalty}(\beta) \leq t} \ \text{Loss}(\beta)\end{aligned}$$

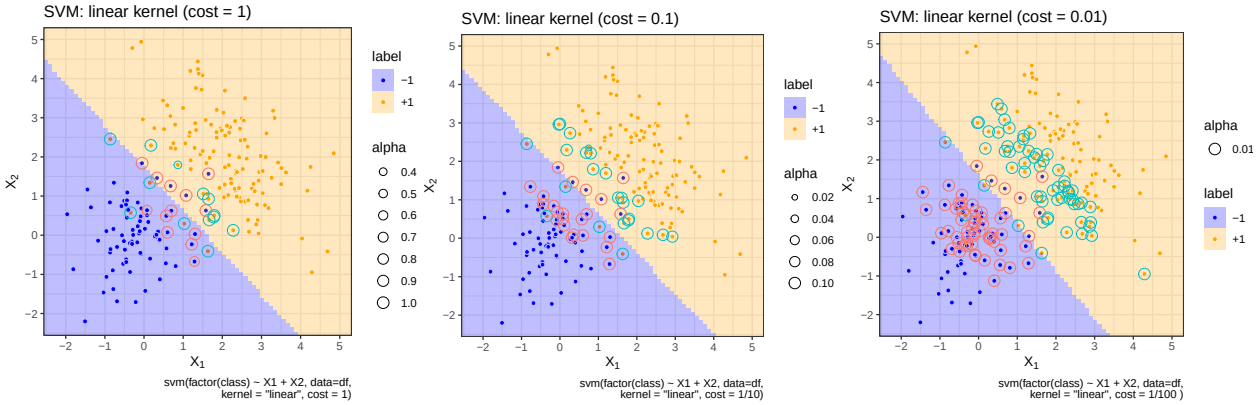
3. **Constraint on Loss**

$$\begin{aligned}\hat{\beta} &= \arg\min_{\beta} \ \text{Penalty}(\beta) \qquad \text{subject to } \ \text{Loss}(\beta) \leq M \\ &= \arg\min_{\beta:\ \text{Loss}(\beta) \leq M} \ \text{Penalty}(\beta)\end{aligned}$$

Note

There is a one-to-one relationship between all tuning parameters: λ, C, t, M.

1.6 Linear SVM



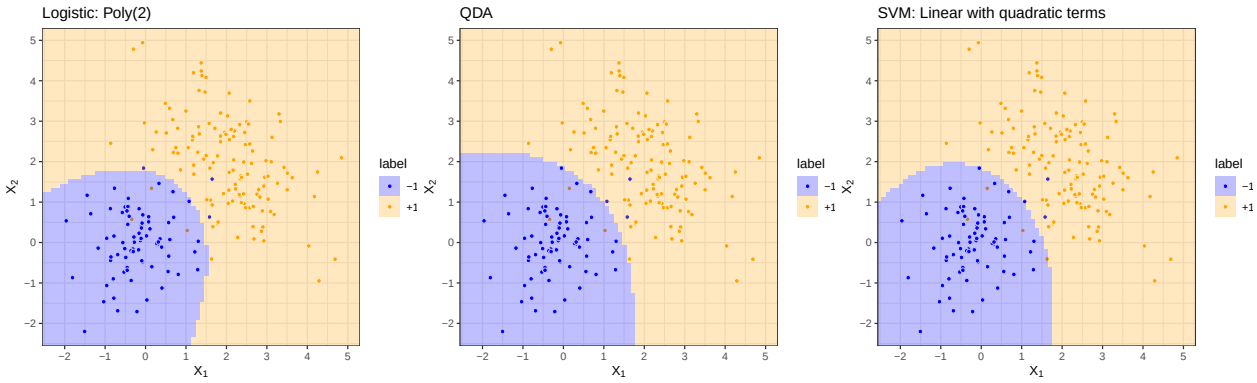
2 Kernels and Non-linear SVM

2.1 Basis Expansion

- Linear: $f(\mathbf{x};\beta) = \beta_0 + \sum_{j=1}^{p} x_j\beta_j$
- Basis Expansion: $f(\mathbf{x};\beta) = \beta_0 + \sum_{j=1}^{d} h_j(\mathbf{x})\beta_j$
  - $h_j(\mathbf{x})$ transforms the raw $\mathbf{x}$ vector

2.1.1 Polynomial Expansion: Quadratic Terms

- Polynomial example: $h_1(\mathbf{x}) = x_1$, $h_2(\mathbf{x}) = x_1^2$, $h_3(\mathbf{x}) = x_2$, $h_4(\mathbf{x}) = x_2^2$, $h_5(\mathbf{x}) = x_1x_2$
- $f(\mathbf{x};\beta) = \beta_0 + \sum_{j=1}^{5} h_j(\mathbf{x})\beta_j$



2.2 Alternative (Dual) Formulation

It turns out that the SVM solution for the model coefficients β can be written as

$$\hat{\beta}_j = \sum_{i=1}^{n} \hat{\alpha}_i \, \tilde{y}_i \, h_j(\mathbf{x}_i)$$

- See ESL Eq. 12.17 if you are interested in the details.
- In this reformulation, the $\{\hat{\alpha}_i\}_{i=1}^n$ become the model parameters.
  - There is one model parameter for each observation!
  - $0 \leq \alpha_i \leq C$ (or $0 \leq \alpha_i \leq \frac{1}{\lambda}$)
  - Most α values will be set to 0 (like what lasso does).
  - The observations with $\alpha_i > 0$ are called the *support vectors*
    - So the entire SVM model is a function of the support vectors only!
    - The support vectors are the observations on the wrong side of the margin.
- The decision function $\hat{f}(\mathbf{x})$ can be re-written

$$\begin{aligned}
\hat{f}(\mathbf{x}) &= f(\mathbf{x}; \hat{\beta}) \\
&= \hat{\beta}_0 + \sum_{j=1}^{d} h_j(\mathbf{x}) \hat{\beta}_j \\
&= \hat{\beta}_0 + \sum_{j=1}^{d} h_j(\mathbf{x}) \left[\sum_{i=1}^{n} \hat{\alpha}_i \, \tilde{y}_i \, h_j(\mathbf{x}_i) \right] \\
&= \hat{\beta}_0 + \sum_{i=1}^{n} \hat{\alpha}_i \, \tilde{y}_i \left[\sum_{j=1}^{d} h_j(\mathbf{x}) h_j(\mathbf{x}_i) \right] \\
&= \hat{\beta}_0 + \sum_{i=1}^{n} \hat{\alpha}_i \, \tilde{y}_i \, K(\mathbf{x}, \mathbf{x}_i)
\end{aligned}$$

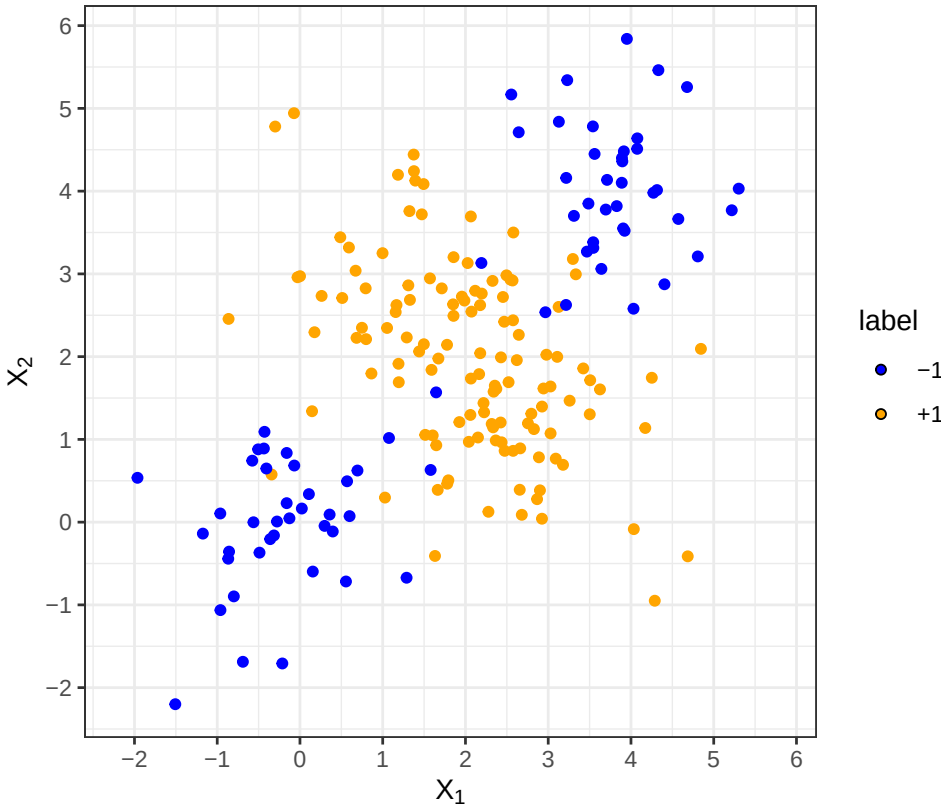
  where

$$K(\mathbf{x}, \mathbf{x}_i) = \sum_{j=1}^{d} h_j(\mathbf{x}) h_j(\mathbf{x}_i) = \langle h(\mathbf{x}), h(\mathbf{x}_i) \rangle$$

  is called a **kernel** and measures the inner product, or *similarity* between x and x_i (observation i).

2.3 Kernels

To help illustrate the difference between different kernels, let's look at slightly different data that won't be easy to classify using linear classifiers.

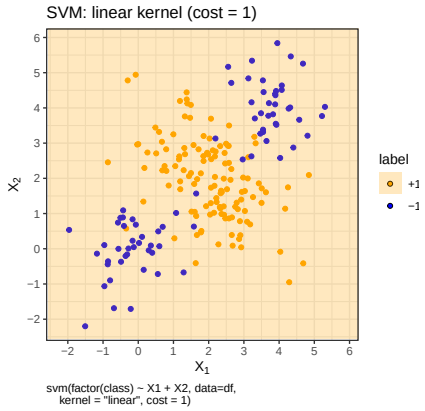


Three popular kernels (but there are many more) are the *linear*, *polynomial*, and *radial basis*

2.3.1 Linear Kernel

The *linear kernel* is

$$K(\mathbf{x}, \mathbf{u}) = \sum_{j=1}^{p} \mathbf{x}_j \mathbf{u}_j = \langle \mathbf{x}, \mathbf{u} \rangle$$



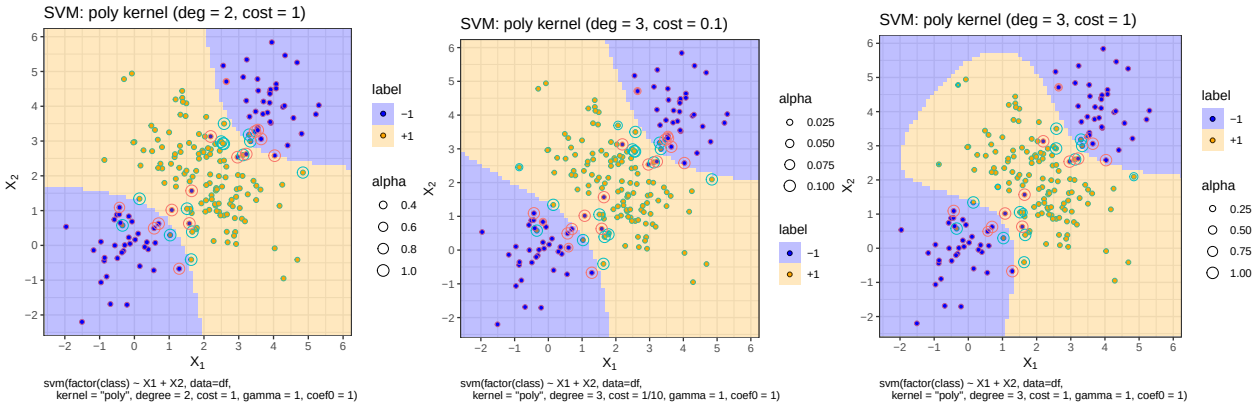
$$\hat{f}_{\text{linear}}(\mathbf{u}) = \hat{\beta}_0 + \sum_{i=1}^{n} \hat{\alpha}_i \, \tilde{y}_i \, K(\mathbf{x}_i, \mathbf{u})$$
$$= \hat{\beta}_0 + \sum_{i=1}^{n} \hat{\alpha}_i \, \tilde{y}_i \left(\sum_{j=1}^{p} \mathbf{x}_{ij} \mathbf{u}_j \right)$$

2.3.2 Polynomial Kernel

The *polynomial kernel* of degree `deg` is

$$K(\mathbf{x}, \mathbf{u}) = \left(1 + \sum_{j=1}^{p} \mathbf{x}_j \mathbf{u}_j \right)^{\text{deg}}$$

Note: in R, the 'svm()' function from 'e1071' package includes two other tuning parameters (`gamma` and `coef0`)



$$\hat{f}_{\text{poly}}(\mathbf{u}) = \hat{\beta}_0 + \sum_{i=1}^{n} \hat{\alpha}_i \, y_i \, K(\mathbf{x}_i, \mathbf{u})$$
$$= \hat{\beta}_0 + \sum_{i=1}^{n} \hat{\alpha}_i \, y_i \left(1 + \sum_{j=1}^{p} \mathbf{x}_j \mathbf{u}_j \right)^{\text{deg}}$$

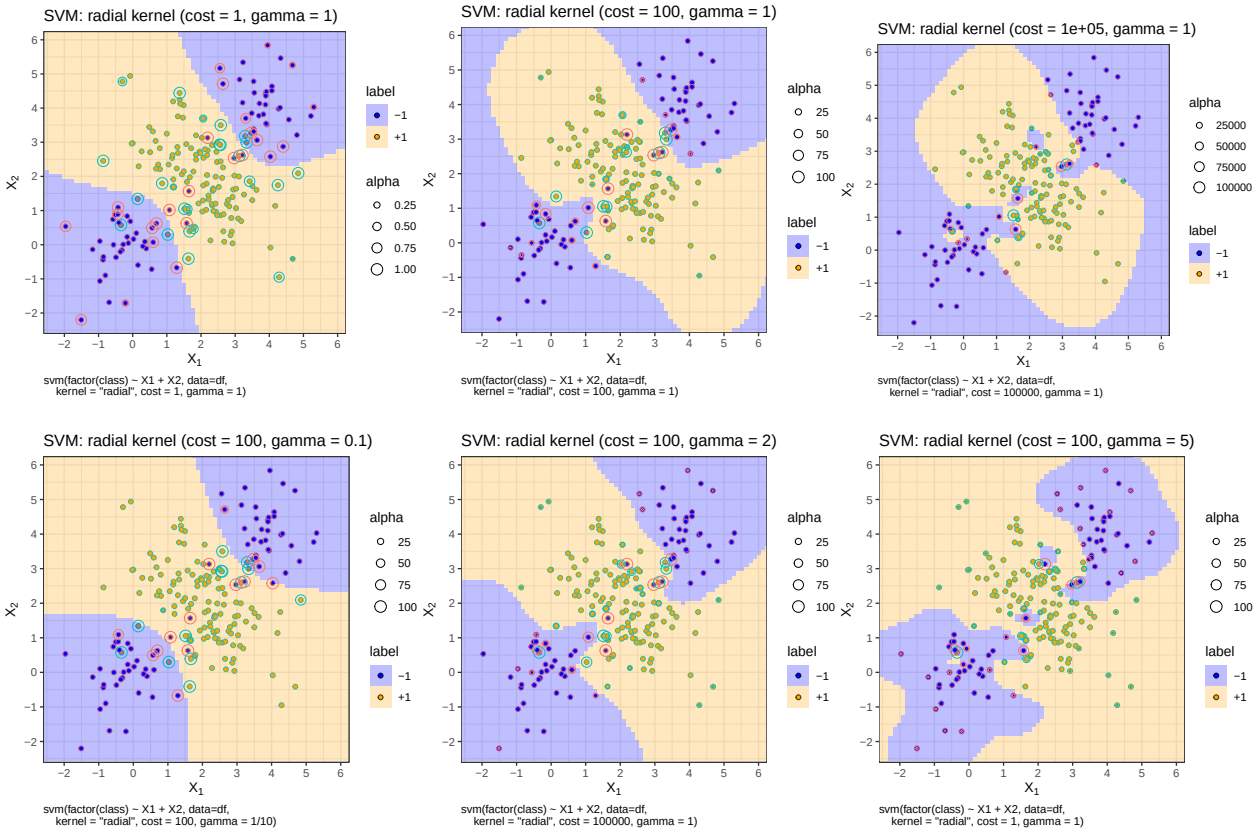
2.3.3 Radial Basis Kernel

The *Radial Basis Kernel* with parameter γ (`gamma`) is

$$K(\mathbf{x}, \mathbf{u}) = \exp\left(-\gamma \sum_{j=1}^{p} (\mathbf{x}_j - \mathbf{u}_j)^2 \right)$$
$$= \exp\left(-\gamma \, dist^2(\mathbf{x}, \mathbf{u}) \right)$$

where $dist^2(\mathbf{x}, \mathbf{u})$ is the *squared Euclidean distance* between $\mathbf{x}$ and $\mathbf{u}$.

- The Radial Basis kernel is large for test observations close to training observations.
- Notice that both `cost` and `gamma` are influential tuning parameters.



$$\hat{f}_{\text{radial}}(\mathbf{u}) = \hat{\beta}_0 + \sum_{i=1}^{n} \hat{\alpha}_i \, \tilde{y}_i \, K(\mathbf{x}_i, \mathbf{u})$$

$$= \hat{\beta}_0 + \sum_{i=1}^{n} \hat{\alpha}_i \, \tilde{y}_i \, \exp\left(-\gamma \, dist^2(\mathbf{x}_i, \mathbf{u})\right)$$

2.4 Imbalanced Data and Unequal costs of misclassification

Imbalanced data (i.e., the number of $y = -1$ is very different than the number of $y = +1$; or the prior probability $\Pr(Y = +1)$ is not close to 1/2) can be a problem for SVM models. This is because the loss function is constructed to separate classes, not estimate the probability. Notice that the optimal Hinge (SVM) loss solution minimizes sign[Pr(Y = +1) - 1/2] (see ESL (Table 12.1) for details). This will also be a problem if the costs of misclassification are not equal (i.e., $C_{\text{FP}} \neq C_{\text{FN}}$).

Three primary approaches to purse with unbalanced outcomes and/or unequal costs of misclassification:

1. Use weights corresponding to prior class probabilities
   - See the `class.weights` argument in `?svm`
   - Ensure the resulting score is what you want for unadjusted prediction.

$$\hat{\beta}_\lambda = \arg\min_\beta \left\{ \sum_{i=1}^{n} w_i \max\{0, 1 - \tilde{y}_i f_i(\beta)\} + \lambda \|\beta\|^2 / 2 \right\}$$

2. Use a threshold on the score
   - E.g., choose label = +1 if: $\hat{f}(\mathbf{x}) > t$
   - The $\hat{f}$ are the `decision.values` output of `predict.svm` (this is an *attribute* of the returned svm object).
   - Thresholding may not work well with SVM since the loss is encouraging a hard classification (at threshold 1/2).
3. Sampling
   - under or over sample training data from one class to balance the label distribution.
   - The SVM (hinge) objective function is optimized at sign $[\Pr(Y = +1|X = x) - 1/2]$.
   - Ensure the resulting score is what you want for unadjusted prediction.

3 Appendix: R Code

3.1 Required R Packages

We will be using the R packages of:

- `tidyverse` for data manipulation and visualization
- `e1071` for the `svm()` functions

```
library(e1071)        # svm() functions
library(yardstick)    # for evaluation
library(tidyverse)    # hello tidyverse
```

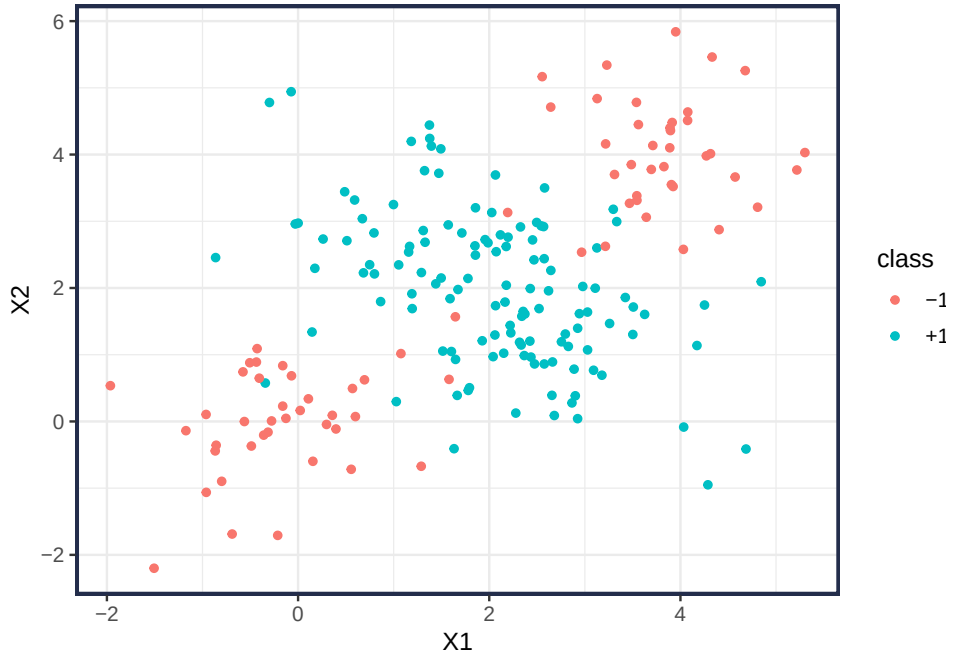
3.2 SVM in R

- The `svm()` function from the `e1071` package can implement SVM.
  - There is also a helpful `tune.svm()` to help you select the tuning parameters.
- The ISLR Lab in Section 9.6 has example R code.
- The ISLR `tidymodels` lab has example R code.

3.3 Data

```
#-------------------------------------------------------------------------#
# Create Data
#-------------------------------------------------------------------------#
library(mvtnorm)

prior = c(.60, .40)
mu1 = c(2, 2)
mu2 = c(0, 0)
mu2.B = c(4, 4)
sigma1 = .5*matrix(c(2,-1, -1, 2), nrow=2)
sigma2 = .5*matrix(c(1,0,0,1), nrow=2)


set.seed(2020)
n = 200
n1 = rbinom(1, size=n, prob=prior[1])
n2 = n-n1
n2 = round(c(n2/2, n2/2))
X1 = mvtnorm::rmvnorm(n1, mean=mu1, sigma = sigma1)
X2 = mvtnorm::rmvnorm(n2[1], mean=mu2, sigma = sigma2)
X2 = rbind(X2, mvtnorm::rmvnorm(n2[2], mean=mu2.B, sigma = sigma2))
labels = c("+1", "-1")


data_mix = bind_rows(
  !!labels[1] := as_tibble(X1, .name_repair = ~str_c("X", seq_along(.))),
  !!labels[2] := as_tibble(X2, .name_repair = ~str_c("X", seq_along(.))),
  .id = "class"
)


data_mix %>%
```

```
  ggplot(aes(X1, X2, color = class)) +
  geom_point()
```



3.3.1 Set up Cross-validation

```
#------------------------------------------------------------------------#
# Create Cross-Validation Folds
#------------------------------------------------------------------------#
set.seed(2023)
n.folds = 10
fold = sample(rep(1:n.folds, length=nrow(data_mix)))
```

3.4 SVM Implementation

```
#------------------------------------------------------------------------#
# Polynomial SVM Example
#------------------------------------------------------------------------#
library(e1071)
svm_poly = svm(factor(class) ~ X1 + X2,
               data = data_mix[fold != 1, ],   # hold out fold 1
               #: tuning parameters
               kernel = "poly",
               degree = 2,
               cost = 1
              )

# Notes:
# - svm() requires that the outcome variable be a "factor" for classification
#   problems.
# - the polynomial kernel only has cost and degree parameters
```

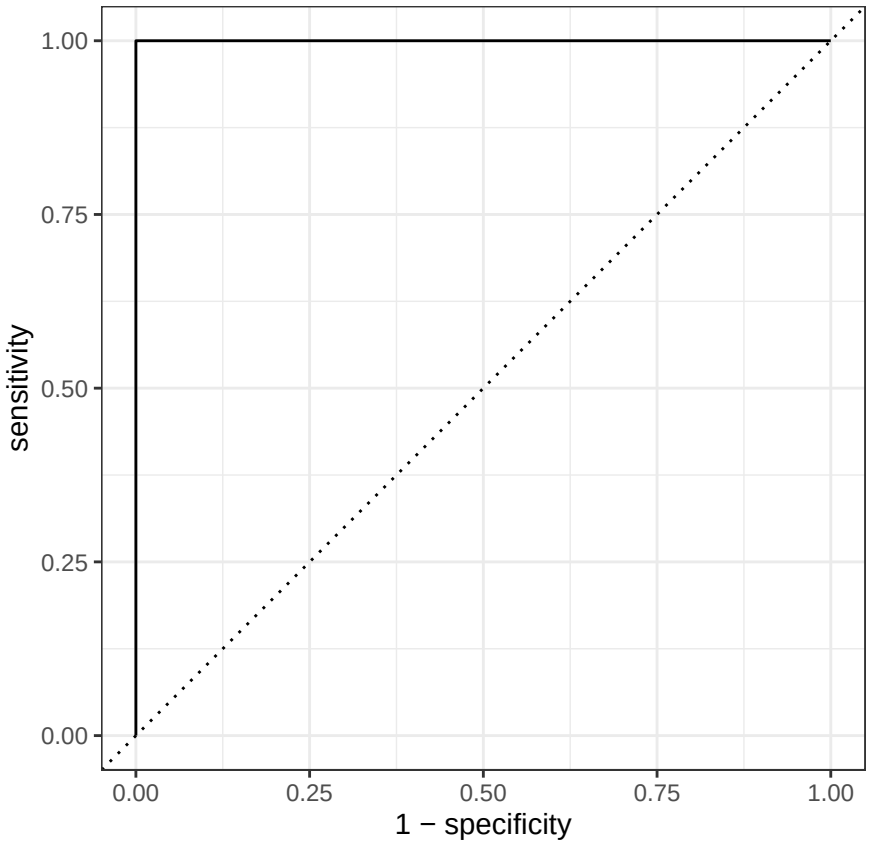
```
#: basic model summary
summary(svm_poly)
#>
```

```
#> Call:
#> svm(formula = factor(class) ~ X1 + X2, data = data_mix[fold != 1,
#>     ], kernel = "poly", degree = 2, cost = 1)
#>
#>
#> Parameters:
#>    SVM-Type:  C-classification
#>  SVM-Kernel:  polynomial
#>        cost:  1
#>      degree:  2
#>      coef.0:  0
#>
#> Number of Support Vectors:  30
#>
#>  ( 15 15 )
#>
#>
#> Number of Classes:  2
#>
#> Levels:
#>  -1 +1
```

```
#: predictions
pred = predict(svm_poly, data_mix[fold == 1,], decision.values = TRUE)

eval_data = tibble(
  outcome = data_mix$class[fold == 1],
  pred_hard = c(pred),
  pred_soft = as.numeric(attr(pred, "decision.values"))
)

# Notes:
# - the standard output of predict.svm() is the hard labels {-1, +1}
# - To get the decision values,
#     a. Set the `decision.values = TRUE` argument.
#     b. extract the decision.values *attribute*
# - the decision values/scores can be used to create rank or
#   threshold-based metrics like ROC curves
# - There is also a way to get probability estimates by setting
#   svm(..., probability = TRUE) and predict(..., probability = TRUE).
#   This attempts to convert scores to probabilities using Platt's method.
```

```
#: evaluation
library(yardstick)
levs = c("+1", "-1")  # set outcome of interest at first level

#: update the eval data to ensure the categorical outcomes have correct levels
eval_data_tidy = eval_data %>%
  mutate(
    outcome = factor(outcome, levels=levs),
    pred_hard = factor(pred_hard, levels=levs)
  )

#: Make ROC curve
eval_data_tidy %>%
  yardstick::roc_curve(outcome, pred_soft) %>%
  autoplot()
```

```
#: Calculate AUC
eval_data_tidy %>%
  yardstick::roc_auc(outcome, pred_soft)
#> # A tibble: 1 x 3
#>   .metric .estimator .estimate
#>   <chr>   <chr>          <dbl>
#> 1 roc_auc binary             1


#: Calculate accuracy
eval_data_tidy %>%
  yardstick::accuracy(outcome, pred_hard)
#> # A tibble: 1 x 3
#>   .metric  .estimator .estimate
#>   <chr>    <chr>          <dbl>
#> 1 accuracy binary           0.9


# Notes:
# - the yardstick packages wants outcome variables and hard predictions to be
#   "factors" with the outcome of interest as the first level
```



3.5 SVM Evaluation

Here we'll switch to a radial basis function kernel which has `gamma` and `cost` tuning parameters.

```
#-------------------------------------------------------------------#
# Radial basis SVM tuning parameter selection
#-------------------------------------------------------------------#
```

```r
#: create function to fit, predict, and evaluate
eval_svm <- function(data_train, data_test, cost, gamma){

  #: fit
  fit = svm(factor(class) ~ X1 + X2,
            data = data_train,
            #: tuning parameters
            kernel = "radial",
            gamma = gamma,
            cost = cost
            )

  #: predict
  pred = predict(fit, data_test, decision.values = TRUE)

  #: evaluate
  levs = c("+1", "-1")  # set outcome of interest at first level
  eval_data = tibble(
    outcome = data_test$class %>% factor(levels=levs),
    pred_hard = c(pred) %>% factor(levels = levs),
    pred_soft = as.numeric(attr(pred, "decision.values"))
  )

  #: output
  auroc = eval_data %>%
    yardstick::roc_auc(outcome, pred_soft)

  accuracy = eval_data %>%
      yardstick::accuracy(outcome, pred_hard)

  bind_rows(auroc, accuracy) %>%
    mutate(cost, gamma, .before=1)  # add tuning parameters

}
```

```r
#: test it out with single fold
eval_svm(
  data_train = data_mix[fold != 1,],
  data_test = data_mix[fold == 1,],
  cost = 1,
  gamma = .1
)
#> # A tibble: 2 x 5
#>    cost gamma .metric  .estimator .estimate
#>   <dbl> <dbl> <chr>    <chr>          <dbl>
#> 1     1   0.1 roc_auc  binary         0.989
#> 2     1   0.1 accuracy binary         0.9
```

3.6 SVM Tuning (single fold)

Set up grid of tuning parameters

```r
# Set-up grid of tuning parameters
svm_grid = expand_grid(
  cost = 10^(1:5),
  gamma = 10^(seq(-4, 4, length=5))
)
```

Function that implements for a single set of tuning parameters

```
# function that implements for tuning grid
library(purrr)
svm_tune_fold <- function(grid, k = 1){

  eval_function <- function(cost, gamma){
    eval_svm(
      data_train = data_mix[fold != k,],
      data_test =  data_mix[fold == k,],
      cost = cost, gamma = gamma
    )
  }

 pmap(svm_grid, function(cost, gamma) eval_function(cost, gamma)) %>%
    bind_rows()
}

# Note: the pmap() function passes each row to the evaluation function
```

Run function over all tuning parameters in grid

```
# run function over all tuning parameters (fits model 25 times)
results = svm_tune_fold(svm_grid)

results %>%
  filter(.metric == "roc_auc") %>%
  arrange(-.estimate, gamma, cost)
#> # A tibble: 25 x 5
#>     cost  gamma .metric .estimator .estimate
#>    <dbl>  <dbl> <chr>   <chr>          <dbl>
#> 1    100 0.01   roc_auc binary         1
#> 2   1000 0.01   roc_auc binary         1
#> 3  10000 0.01   roc_auc binary         1
#> 4 100000 0.01   roc_auc binary         1
#> 5    100 0.0001 roc_auc binary         0.989
#> 6   1000 0.0001 roc_auc binary         0.978
#> # i 19 more rows

results %>%
  filter(.metric == "accuracy") %>%
  arrange(-.estimate, gamma, cost)
#> # A tibble: 25 x 5
#>     cost gamma .metric  .estimator .estimate
#>    <dbl> <dbl> <chr>    <chr>          <dbl>
#> 1    100  0.01 accuracy binary           0.9
#> 2   1000  0.01 accuracy binary           0.9
#> 3  10000  0.01 accuracy binary           0.9
#> 4 100000  0.01 accuracy binary           0.9
#> 5     10  1    accuracy binary           0.9
#> 6    100  1    accuracy binary           0.9
#> # i 19 more rows
```

3.6.1 Alternative with explicit loop

```
#: More explicit loop over grid of tuning parameters
k = 1
out = tibble()
for(i in 1:nrow(svm_grid)){
```

```
  metrics = eval_svm(
        data_train = data_mix[fold != k,],
        data_test = data_mix[fold == k,],
        cost = svm_grid$cost[i],
        gamma = svm_grid$gamma[i]
  )

  out = bind_rows(out, metrics %>% mutate(fold = k) )
}
```

3.7 SVM Tuning (cross-validation)

```
#: cross-validation (loop over folds and tuning grid)
# Note: this will take some time since it implements svm many times
#  (# grid x # folds), which is 25x10 = 250.
library(purrr)
svm_tune_cv <- function(grid, foldid){
  map(foldid, ~svm_tune_fold(grid, k = .x)) %>%
    bind_rows(.id = "fold")
}
```

```
#: get results for all folds and grid
results_cv = svm_tune_cv(svm_grid, foldid = fold)
```

```
#: calculate average performance over all folds
results_cv_avg  = results_cv %>%
  group_by(cost, gamma, .metric) %>%
    summarize(avg = mean(.estimate), n = n(), sd = sd(.estimate)) %>%
  ungroup()

results_cv_avg %>%
  filter(.metric == "roc_auc") %>%
  arrange(-avg, gamma, cost)
#> # A tibble: 25 x 6
#>     cost  gamma .metric   avg     n     sd
#>    <dbl>  <dbl> <chr>   <dbl> <int>  <dbl>
#> 1   1000 0.01   roc_auc 0.984   200 0.0278
#> 2  10000 0.01   roc_auc 0.984   200 0.0244
#> 3    100 0.01   roc_auc 0.982   200 0.0337
#> 4 100000 0.01   roc_auc 0.980   200 0.0363
#> 5     10 0.01   roc_auc 0.974   200 0.0386
#> 6 100000 0.0001 roc_auc 0.974   200 0.0357
#> # i 19 more rows

results_cv_avg %>%
  filter(.metric == "accuracy") %>%
  arrange(-avg, gamma, cost)
#> # A tibble: 25 x 6
#>     cost gamma .metric    avg     n     sd
#>    <dbl> <dbl> <chr>    <dbl> <int>  <dbl>
#> 1    100  0.01 accuracy 0.95    200 0.0448
#> 2   1000  0.01 accuracy 0.945   200 0.0569
#> 3  10000  0.01 accuracy 0.94    200 0.0540
#> 4 100000  0.01 accuracy 0.935   200 0.0504
#> 5     10  1    accuracy 0.925   200 0.0560
#> 6    100  1    accuracy 0.925   200 0.0462
#> # i 19 more rows
```

3.7.1 Alternative with explicit loop

```
#: loop over folds and grids
out = tibble()

for(k in unique(fold)){

  for(i in 1:nrow(svm_grid)){

    metrics = eval_svm(
          data_train = data_mix[fold != k,],
          data_test = data_mix[fold == k,],
          cost = svm_grid$cost[i],
          gamma = svm_grid$gamma[i]
    )

    out = bind_rows(out, metrics %>% mutate(fold = k) )
  }
}
```

3.8 Tidymodels

The tidymodels set of packages implements the same steps.

3.8.1 Resampling splits

First, set up the cross-validation splits.

```
#: using tidymodels
library(rsample)
set.seed(2023)
cv_folds = rsample::vfold_cv(data_mix, v = 10)
```

3.8.2 Using the custom `eval_svm()` function

```
#: function to search over grid
eval_svm_grid <- function(data, grid){
  pmap_df(grid, function(cost, gamma) eval_svm(training(data), testing(data),
                                               cost = cost, gamma = gamma))
}

# try it out on single fold
data = cv_folds$splits[[1]]
eval_svm_grid(data, head(svm_grid))
#> # A tibble: 12 x 5
#>    cost  gamma .metric  .estimator .estimate
#>   <dbl>  <dbl> <chr>    <chr>          <dbl>
#> 1    10 0.0001 roc_auc  binary         0.879
#> 2    10 0.0001 accuracy binary         0.35
#> 3    10 0.01   roc_auc  binary         0.978
#> 4    10 0.01   accuracy binary         0.7
#> 5    10 1      roc_auc  binary         0.934
#> 6    10 1      accuracy binary         0.9
#> # i 6 more rows

#: loop over folds
map_df(cv_folds$splits, ~eval_svm_grid(data=.x, grid=svm_grid), .id="fold")
```

```
#> # A tibble: 500 x 6
#>   fold   cost  gamma .metric  .estimator .estimate
#>   <chr> <dbl>  <dbl> <chr>    <chr>          <dbl>
#> 1 1        10 0.0001 roc_auc  binary         0.879
#> 2 1        10 0.0001 accuracy binary         0.35
#> 3 1        10 0.01   roc_auc  binary         0.978
#> 4 1        10 0.01   accuracy binary         0.7
#> 5 1        10 1      roc_auc  binary         0.934
#> 6 1        10 1      accuracy binary         0.9
#> # i 494 more rows
```

3.8.3 Using `parsnip`, `tune`

First step is to specify the model.

```
library(tidymodels)

svm_model = svm_rbf(mode = "classification") %>%
  set_args(cost = tune(), rbf_sigma = tune())
```

Here we specify the tuning parameters at `cost=1` and `rbf_sigma = .1` and fit to all resamples.

- Note that in the `svm_rbf()` function the argument corresponding to gamma is `rbf_sigma`.

```
fit_resamples(
  object = svm_model %>% set_args(cost = 1, rbf_sigma = .1),
  preprocessor = factor(class) ~ X1 + X2,
  resamples = cv_folds
) %>%
  collect_metrics()
#> # A tibble: 3 x 6
#>   .metric     .estimator   mean     n std_err .config
#>   <chr>       <chr>       <dbl> <int>   <dbl> <chr>
#> 1 accuracy    binary     0.94      10  0.0245 Preprocessor1_Model1
#> 2 brier_class binary     0.0482    10  0.0135 Preprocessor1_Model1
#> 3 roc_auc     binary     0.980     10  0.0109 Preprocessor1_Model1
```

The function `tune_grid()` will loop over both resamples and grid.

```
svm_tuned = tune_grid(
  object = svm_model,
  preprocessor = factor(class) ~ X1 + X2,
  resamples = cv_folds,
  grid = svm_grid %>% rename(rbf_sigma = gamma)
)
```

```
svm_tuned %>% show_best()
#> Warning in show_best(.): No value of `metric` was given; "roc_auc" will be
#> used.
#> # A tibble: 5 x 8
#>     cost rbf_sigma .metric .estimator  mean     n std_err .config
#>    <dbl>     <dbl> <chr>   <chr>      <dbl> <int>   <dbl> <chr>
#> 1   1000      0.01 roc_auc binary     0.984    10 0.00923 Preprocessor1_Model12
#> 2  10000      0.01 roc_auc binary     0.984    10 0.00811 Preprocessor1_Model17
#> 3    100      0.01 roc_auc binary     0.982    10 0.0112  Preprocessor1_Model07
#> 4 100000      0.01 roc_auc binary     0.980    10 0.0121  Preprocessor1_Model22
#> 5     10      0.01 roc_auc binary     0.974    10 0.0128  Preprocessor1_Model02
```

3.9 Probabilites from SVM Decision Values

SVMs don't naturally output a probability estimate, but a calibration approach called Platt scaling is often used convert the SVM output (called *decision values*) to a probability. This approach uses the sigmoid/logistic function to transform the raw SVM output to a number between 0 and 1. In other words, use the output from the SVM as the predictor variable in logistic regression.

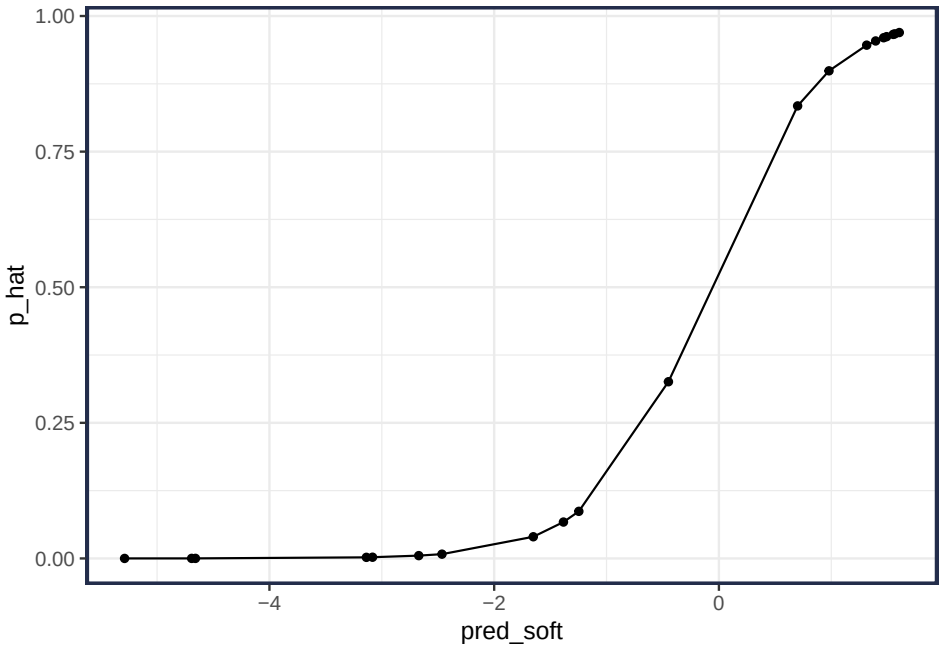
- Because the hinge loss function that is used to estimate the SVM model parameters is zero for observations on the correct side of the margin, we may not expect the probability conversion to work perfectly. However, because the hinge loss and log-loss are not too different, we can have hope it gives something reasonable.
- The original implementation of Platt uses a weighted logistic regression. He also suggested fitting the logistic regression using the hold-predictions from 3-fold cross-validation.
- Because extra modeling steps need to be taken, most SVM implementations require you to specify that you want probability outputs.

3.9.1 Using e1071::svm()

Three steps:

1. Set `probability = TRUE` in the call to `svm()`
2. Also set `probability = TRUE` in the call to `predict()`
3. Extract the probabilities from the "decision.values" attribute

```
data_train = data_mix[fold != 1, ]
data_test =  data_mix[fold == 1, ]

library(e1071)
svm_fit = svm(factor(class) ~ X1 + X2,
          data = data_train,
          probability = TRUE, # ** Need to set to TRUE to get probabilities
          #: tuning parameters
          kernel = "radial",
          gamma = .01,
          cost = 1000
          )

#: predict
pred = predict(svm_fit, data_test,
               decision.values = TRUE,
               probability = TRUE # ** Need to set to TRUE to get probabilities
               )

#: evaluation data
levs = c("+1", "-1")  # set outcome of interest at first level
eval_data = tibble(
  outcome = data_test$class %>% factor(levels=levs),
  pred_hard = c(pred) %>% factor(levels = levs),
  pred_soft = as.numeric(attr(pred, "decision.values")),
  p_hat = attr(pred, "probabilities")[,"+1"] # get probs corresponding to +1 class.
)


eval_data %>%
  ggplot(aes(pred_soft, p_hat)) + geom_point() + geom_line()
```

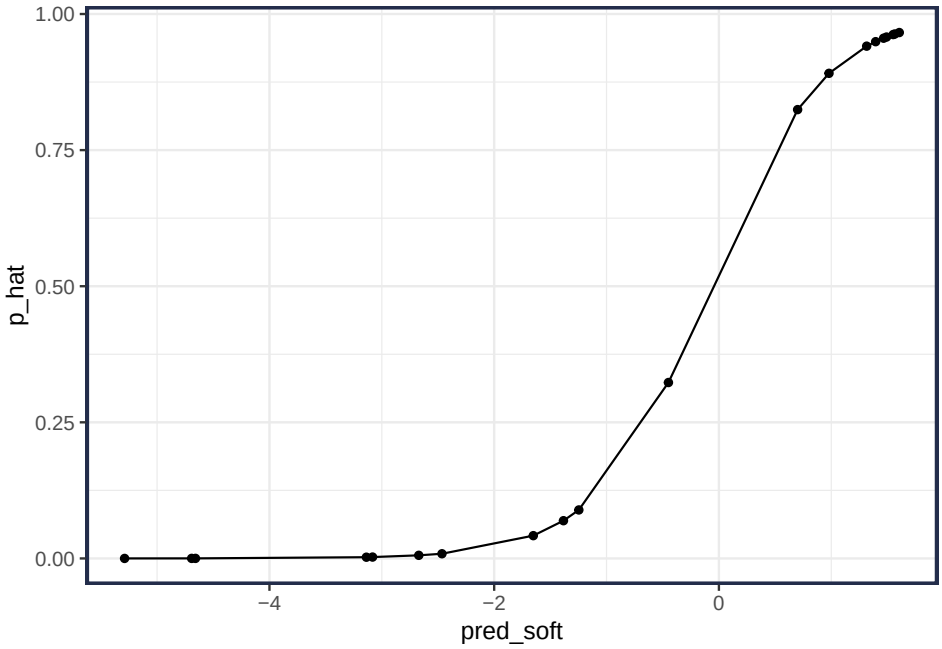


3.9.2 Using kernlab::ksvm()

Two steps:

1. Set `prob.model = TRUE` in the call to `svm()`
2. Use `type = "probabilities"` in the call to `predict()`

```
data_train = data_mix[fold != 1, ]
data_test =  data_mix[fold == 1, ]

library(kernlab)
svm_fit = ksvm(factor(class) ~ X1 + X2,
          data = data_train,
          prob.model = TRUE, # ** Need to set to TRUE to get probabilities
          #: tuning parameters
          kernel = "rbfdot",
          kpar = list(sigma = .01), # called gamma in the notes and e1071::svm()
          C = 1000
          )


#: evaluation data
levs = c("+1", "-1")  # set outcome of interest at first level
eval_data = tibble(
  outcome = data_test$class %>% factor(levels=levs),
  pred_hard = predict(svm_fit, data_test),
  pred_soft = predict(svm_fit, data_test, type = "decision")[,1],
  p_hat = predict(svm_fit, data_test, type = "probabilities")[,"+1"]
)

eval_data %>%
  ggplot(aes(pred_soft, p_hat)) + geom_point() + geom_line()
```


1.00
0.75
0.50
0.25
0.00
p_hat
−4
−2
0
pred_soft