ELEMENTS OF DATA SCIENCE AND STATISTICAL LEARNING

SPRING 2017

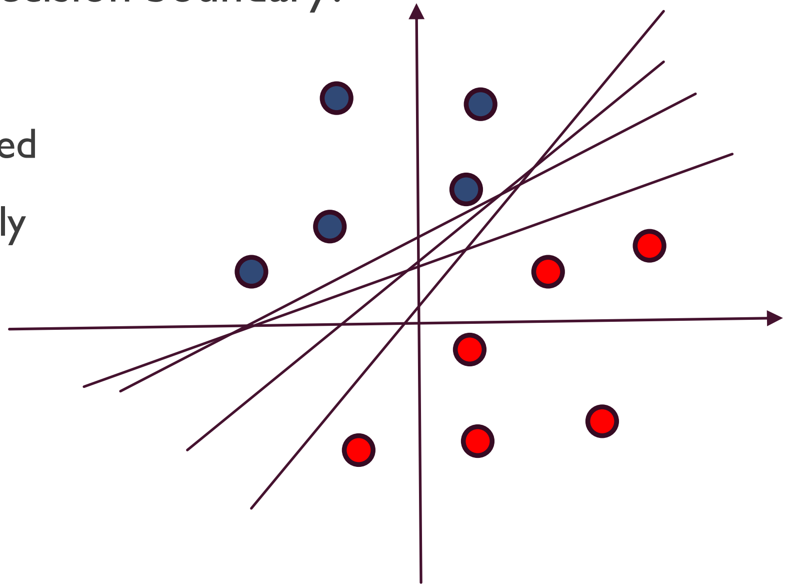
Week 11

OUTLINE

- Support vector machines:
  - Maximal margin classifier
  - Support vector classifier
  - Support vector machine (support vector classifier + non-linear kernel)
- SVM model fitting examples

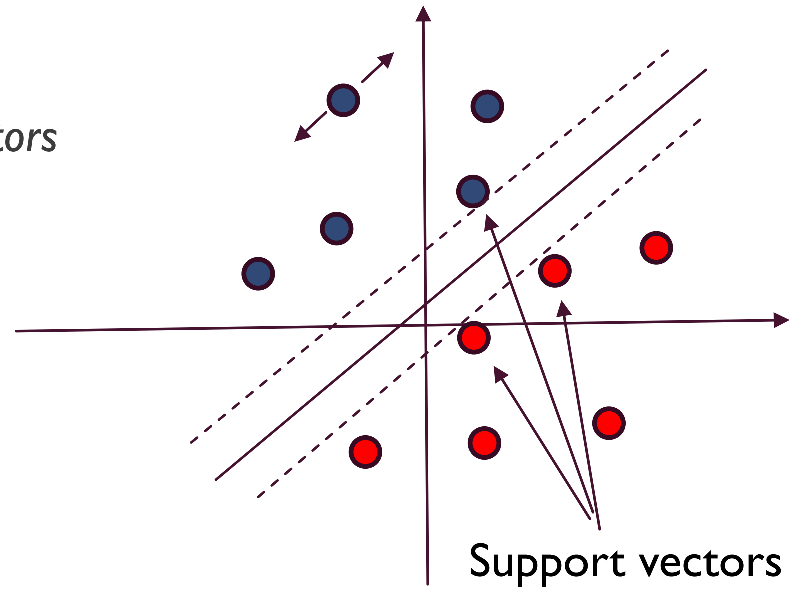
DECISION BOUNDARY

- We have seen a number of different classifiers by now
- All of them have a decision boundary
  - "Watershed" P(y=1) = 0.5;
  - Or simply a (series of) cut(s) in a decision tree (as the latter is not based on calculating probabilities directly)
- What if we try a purely geometric interpretation then, by *starting* from decision boundary?
  - Consider the diagram on the right. The classes are *linearly separable*
  - Classification: point is on one side of the boundary → blue, on the other → red
  - In fact, many lines (linear boundaries) exist that separate these classes perfectly
  - Which of those lines is "the best one"?



MAXIMAL MARGIN CLASSIFIER

- Let us choose a line (or hyperplane in higher-dimensional space) that is as far as possible from the training observations
  - Compute perpendicular distance, d, from each point to the line
  - min(d) is the *margin* (computed separately on either side)
  - Choose the separating line (hyperplane) that has the largest margin
  - This is *maximal margin classifier*
  - Prediction: point on one side = blue, on the other = red
- The data points that are defining the maximal margin are called *support vectors*
- Important property: the maximal margin hyperplane is determined by support vectors only:
  - Imagine that data points that are NOT support vectors are wiggled
  - Will it affect the maximal margin line?

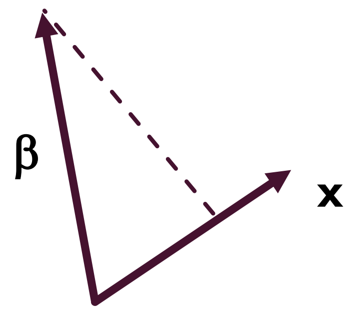


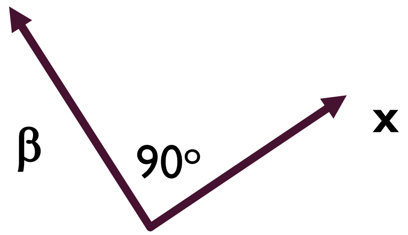
DIGRESSION: SCALAR PRODUCT

- Scalar product of two vectors **β** =(β$_1$, β$_2$,…) and **x** = (x$_1$,x$_2$,…):

  $$\boldsymbol{\beta}^T \boldsymbol{x} = \sum_{i=1}^{p} \beta_i \; x_i$$

- Scalar product ~ length of the projection of one vector onto the other (|β| |x| cosθ)
  - Also, scalar product between variables x, y is the correlation (if x and y are centered and standardized)
- If the two vectors are perpendicular, the projection of one onto the other = 0
  - This is in fact the *definition*: orthogonal directions (vectors) are those with scalar product = 0





HYPERPLANES

- *All* vectors orthogonal to the given direction (vector) β : if we draw such vectors from origin, we get a line (in 2D), plane (in 3D) or hyperplane (in pD). Condition of orthogonality immediately results in

$$\beta_1 x_1 + \beta_2 x_2 + \cdots + \beta_p x_p = 0$$

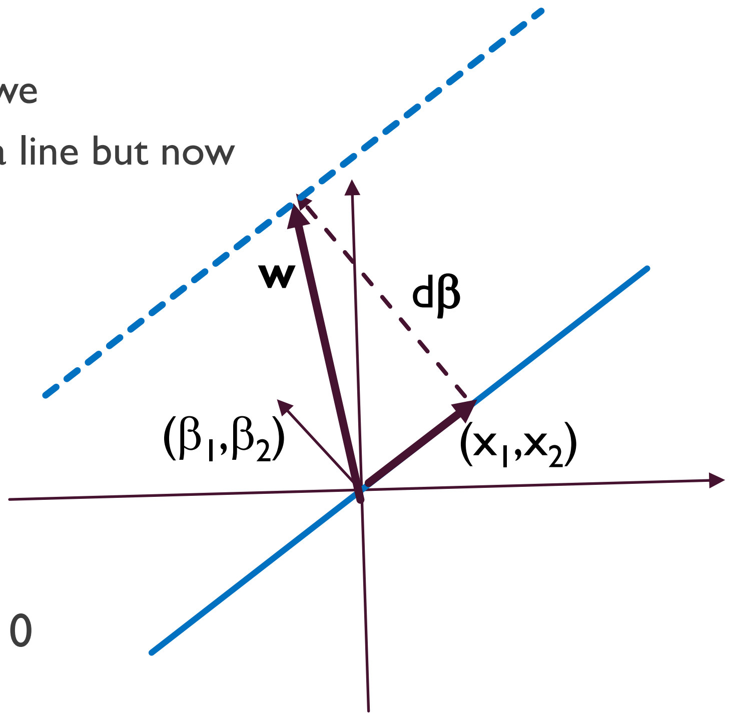
(this is what the equation of a line/hyperplane means: vector of coefficients is the vector normal to the line it defines)

- But must this line/hyperplane go through the origin?
  - If we have a line as the one depicted on the right, and to each point (vector) on that line we add a vector $d \cdot \boldsymbol{\beta}$ (i.e. shift each point by the same amount, **w=x**+d β), we will still get a line but now $0 = \boldsymbol{\beta}^T \boldsymbol{x} = \boldsymbol{\beta}^T (\boldsymbol{w} - d\, \boldsymbol{\beta}\,) = \boldsymbol{\beta}^T \boldsymbol{w} - d\|\beta\|_2^2$
  - In other words, the line/hyperplane shifted from the origin by d β is described by:

$$\beta_1 x_1 + \beta_2 x_2 + \cdots + \beta_p x_p = d\, \|\beta\|_2^2$$, where 2-norm $\|\beta\|_2 = \sqrt{\beta_1^2 + \cdots + \beta_p^2}$

(RHS = distance from the origin)

or $\beta_0 + \beta_1 x_1 + \beta_2 x_2 + \cdots + \beta_p x_p = 0$. Half-space: is defined as LHS > 0 or LHS < 0



MAXIMAL MARGIN CLASSIFIER: SPECIFICATION

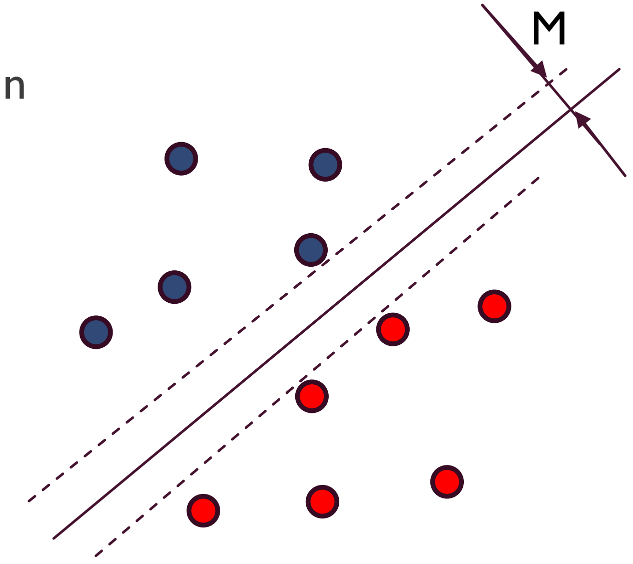
- We can now formulate the conditions on the maximal margin classifier:

  Maximize M over $\beta_0, \beta_1, \ldots, \beta_p$

  Subject to $\sum_{j=1}^{p} \beta_j^2 = 1$ (just for convenience, then M is the actual width of the margin)

  and $y_i\,(\beta_0 + \beta_1 x_{1i} + \beta_2 x_{2i} + \cdots + \beta_p x_{pi}) \geq M$ for all $i=1\ldots n$

- The latter is the principal condition that ensures that all points are outside of the margin
  - We encode y as {-1,1} here, also for convenience (it does not matter which two particular numbers/labels we choose to represent the two classes so we choose the values that make formulas simpler).



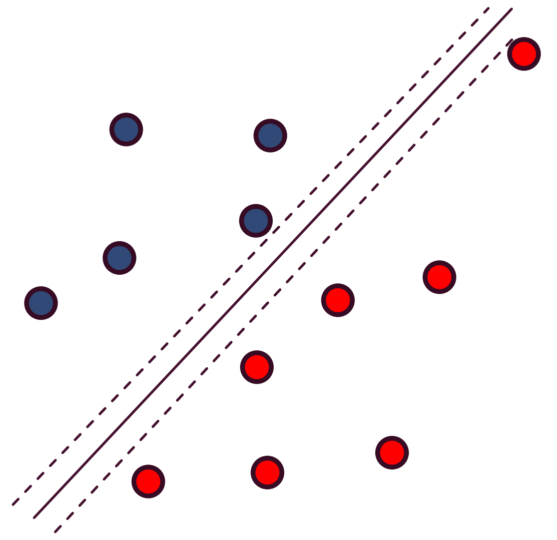
SUPPORT VECTOR CLASSIFIER

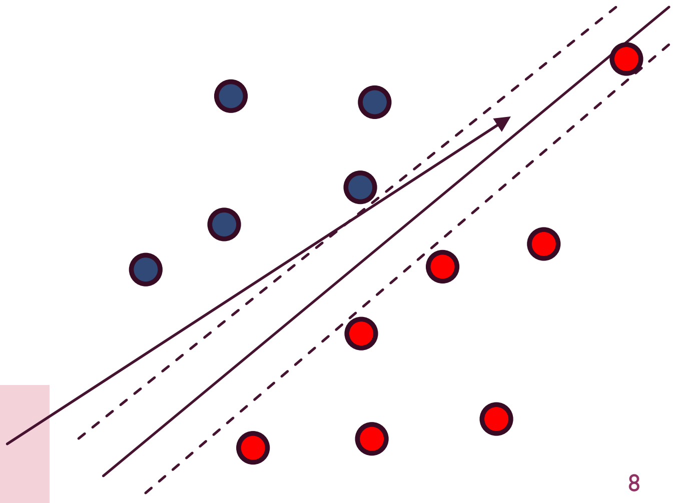
- Maximal margin classifier is very intuitive but
  - Can deal only with separable cases
  - Has very high variance (overfits easily)
- With a simple modification we can allow some points to be within the margin or even on the wrong size: maximize M in

$$y_i\,(\beta_0 + \beta_1 x_{1i} + \beta_2 x_{2i} + \cdots + \beta_p x_{pi}) \geq M(1-\epsilon_i) \quad \text{where} \quad \textstyle\sum_{i=1}^{n} \epsilon_i \leq A$$

… (all the rest is the same)

- What does this modification achieve?
  - Any point x$_i$ is allowed to be inside the margin ($\epsilon_i > 0$) or even on the wrong side of the separating hyperplane ($\epsilon_i > 1$), but the further it ventures from its half-space, the higher the cost ϵ_i. The constrained optimization with $\sum_{i=1}^{n} \epsilon_i \leq A$ endures that we choose the separating hyperplane in such a way that they don't overspend the allowance A for being in the margin/on the wrong side





If we have sufficient allowance to pay for placing one point inside the margin, we can get a wider margin!

SV LOSS FUNCTION (HINGE LOSS)

- Can the function that is the subject for optimization be written down?
- For instance, in linear regression – we minimize RSS=$\sum_{i=1}^{n}(y_i - f(\boldsymbol{x}_i))^2$ where

  $f(\boldsymbol{x}) = \beta_0 + \beta_1 x_1 + \beta_2 x_2 + \cdots + \beta_p x_p$

  - Related to maximum likelihood (normally distributed noise → quadratic RSS)
- It turns out that in the max. margin classifier we are minimizing:

$$\sum_{i=1}^{n} \ell(y_i, x_i) + \lambda \sum_{j=1}^{p} \beta_j^2$$

, where loss function $\ell(y_i, x_i) = \max[0, 1 - y_i\, f(x_i)]$

Margin, rescaled

  - Parameter λ is related to A: small λ = small A (few margin violations)
  - Yet another equivalent form (C is the *cost* of margin violation or misclassification; high cost = small lambda in the formula above):

$$C \sum_{i=1}^{n} \ell(y_i, x_i) + \sum_{j=1}^{p} \beta_j^2$$

C is what the parameter "cost" is in e1071 implementation of the SVM

LOGISTIC LOSS FUNCTION

- We can write down a loss function for logistic regression as well.
- Remember that in LR we are using a linear model for log odds Z: $Z = f(x) = \beta_0 + \beta_1 x_1 + \cdots + \beta_p x_p$
- While the outcome probability is linked to Z via non-linear logistic function (NOTE: here we use y={-1,1}):

$$\left.\begin{aligned} P(y=1) &= \frac{e^{f(x)}}{1+e^{f(x)}} = \frac{1}{1+e^{-f(x)}} \\ P(y=-1) &= 1 - P(y=1) = \frac{1}{1+e^{f(x)}} \end{aligned}\right\} \Longrightarrow \quad P(y) = \frac{1}{1+e^{-yf(x)}}$$

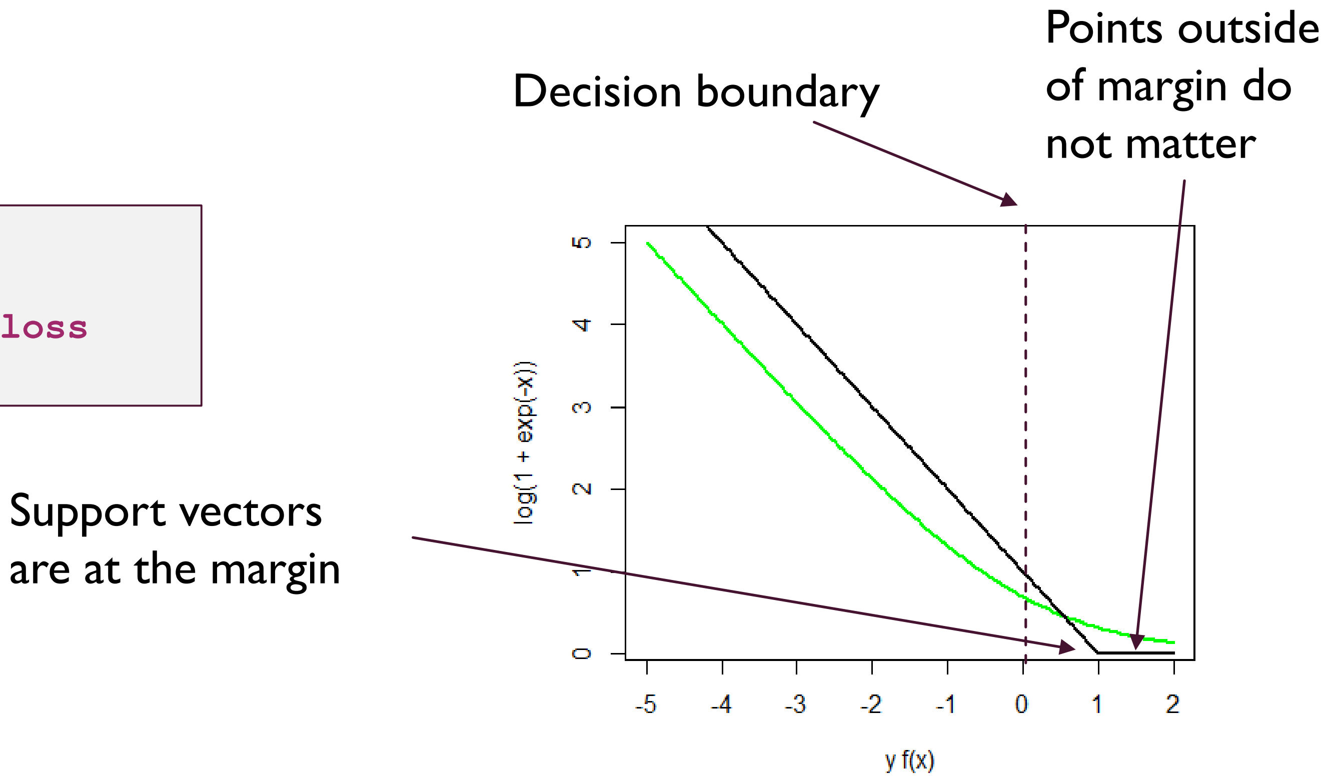
- The likelihood (probability to observe the data) is $\prod_{i=1}^{n} P(y = y_i|\theta)$ (binomial probability), hence

$$-\log L = -\log P(D|\theta) = \sum_{i=1}^{N} \log\left(1 + e^{-y_i f(x_i)}\right)$$

LOGISTIC LOSS VS HINGE LOSS

- We can compare loss functions for the support vector classifier and for LR (as function of the distance from the classification boundary, *y f(x)*):
  - Note that they are very similar. The SV loss ("hinge loss") does not have any contribution for correctly classified points that do not fall inside the margin

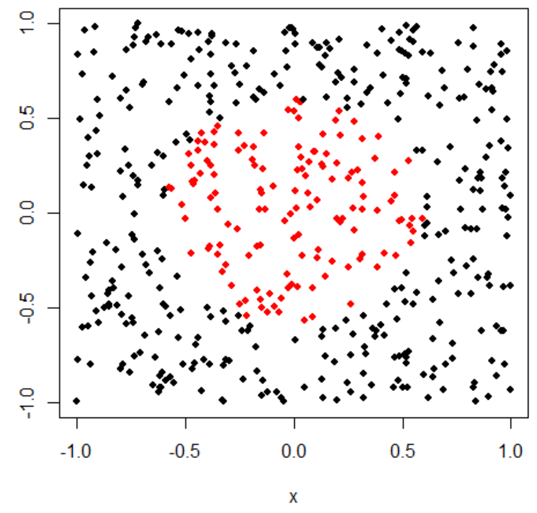
```
x=seq(-5,2,by=0.05)
plot(x,log(1+exp(-x)),type='l',col='green',
     xlab=expression("y f(x)"),lwd=2) # logistic loss
points(x,pmax(0,1-x),type='l',lwd=2)  # SV loss
```



A MOTIVATING EXAMPLE

- Let us consider a simple example of a dataset that is not linearly separable (CLASS=1 if $X^2+Y^2<0.36$, 0 otherwise)
  - Clearly it might become separable if we consider "higher order terms" in a model…

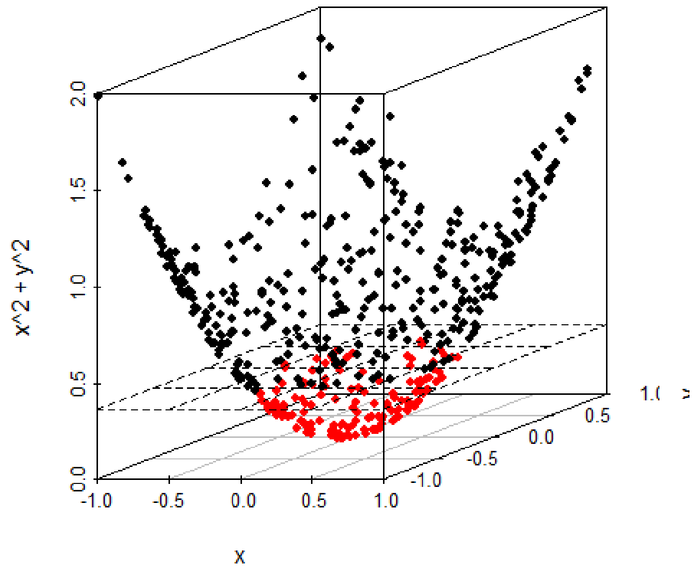
```
library(scatterplot3d)
set.seed(1234)
x=runif(500,min=-1,max=1)
y=runif(500,min=-1,max=1)
cl=numeric(500)
cl[x^2+y^2 < 0.36 ] = 1
plot(x,y,col=cl+1,pch=19,cex=0.8)
d = data.frame(CLASS=as.factor(cl),X=x,Y=y)
Mglm = glm(CLASS ~ X+Y+I(X^2)+I(Y^2) , data = d, family="binomial")
assess.prediction(d$CLASS, as.numeric(predict(Mglm,type="response") > 0.5))
Total cases that are not NA: 500
Correct predictions (accuracy): 500(100%)
TPR (sensitivity)=TP/P: 100%
TNR (specificity)=TN/N: 100%
PPV (precision)=TP/(TP+FP): 100%
FDR (false discovery)=1-PPV: 0%
FPR =FP/N=1-TNR: 0%
```



A MOTIVATING EXAMPLE

- Let us explicitly introduce new variable $Z = X^2+Y^2$ and generate a plot in 3D
- Clearly the data are perfectly separable now!
  - Note that this is not a new fact/observation compared to what we did in the previous slide; the fact is the same: additional/higher order terms can improve the model
  - This is a different visualization of the same fact: we explicitly consider our data points in the higher-dimensional space. Reflect on it.

```
library(scatterplot3d)
sp=scatterplot3d(x,y,x^2+y^2,color=cl+1,cex.symbols=0.8,
    pch=19,angle=20)
sp$plane3d(0.36,0,0)
```



SUPPORT VECTOR MACHINE

- A simple and naïve way to achieve non-linear decision boundary: use higher orders, transformations and combinations of the variables (just like with other models): X^2, log X, XY^3 etc
- A smarter (and more efficient!) way exists for support vector classifiers:
- The solution to the SV classifier problem involves only scalar products between the observations, $\boldsymbol{x}_i^T \boldsymbol{x}_j = \sum_{k=1}^{p} x_{ik} x_{jk}$, and the classifier function *f*(***x***) for new observation ***x*** can be written down in terms of projections of ***x*** onto the training observations:

  $f(x) = \beta_0 + \sum_{i=1}^{n} \alpha_i \; \boldsymbol{x}^T \boldsymbol{x}_i$

  (α are in fact non-zero only for the support vectors, so most of α_i =0).
- We can (somewhat formally) replace all scalar products $\boldsymbol{x}_i^T \boldsymbol{x}_j = \sum_{k=1}^{p} x_{ik} x_{jk}$ with (arbitrary) *kernel* function $K(\boldsymbol{x}_i, \boldsymbol{x}_j)$. Such model is what is known, strictly speaking, as Support Vector Machine (SVM).
- Using a kernel other than the conventional scalar product is equivalent to fitting support vector classifier in a higher-dimensional space (as if we manually added higher powers/combinations of the predictor variables!).

SOME POPULAR KERNELS

- Polynomial kernel of degree *d* (amounts to using polynomials of degree d as variables):

$$K(x_i, x_j) = \left(1 + \sum_{k=1}^{p} x_{ik} x_{jk}\right)^d$$

- Radial kernel. Note that only "nearby" points x_i , x_j will result in relatively large kernels (how close they should be is determined by γ: when γ is large, only very close points matter). The feature space is *infinite*-dimensional! The classifier is also somewhat related to KNN in this case, but not the same and in SVM formulation one can perform regularization relatively easily too:

$$K(x_i, x_j) = \exp\left[-\gamma \sum_{k=1}^{p} (x_{ik} - x_{jk})^2\right]$$

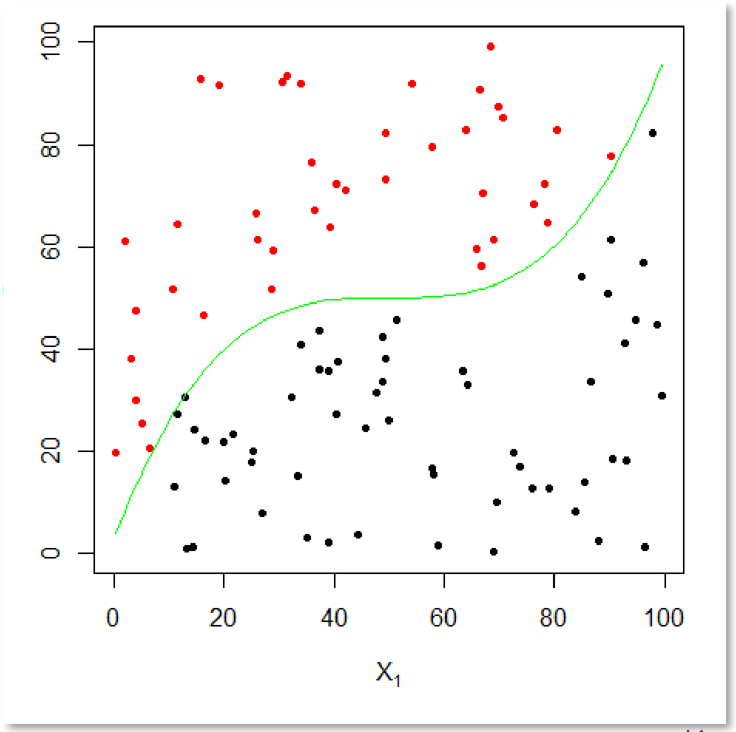
- The classifier function (that defines the separating plane) is defined in general as

$$f(x) = \beta_0 + \sum_{i=1}^{n} \alpha_i \; K(\boldsymbol{x}, \boldsymbol{x}_i)$$

STUDY: SIMULATED DATASET

- Let us start with simulating another simple 2D dataset that we can visualize and study
  - Note that we are creating here a separable dataset (but the boundary is not a straight line!) – we don't add any noise
  - Note also that we start with generating not too many data points, the limited amount of data that we have is likely insufficient for restoring the correct shape of the separating curve. Even straight boundary might work well (the usual question – do we even have enough data to be able to detect any details beyond the simplest possible linear trend?)

```
x1=runif(100,min=0,max=100)
x2=runif(100,min=0,max=100)
bound=function(x) { 50+(x-2*(x-20)+3*(x-50)^3)/8000 }
y=ifelse(x2 > bound(x1),1,0)
plot(x1,x2,pch=19,cex=0.7,col=y+1,xlab=expression(X[1]),
     ylab=expression(X[2]))
points(sort(x1),bound(sort(x1)),type='l',col='green')
```

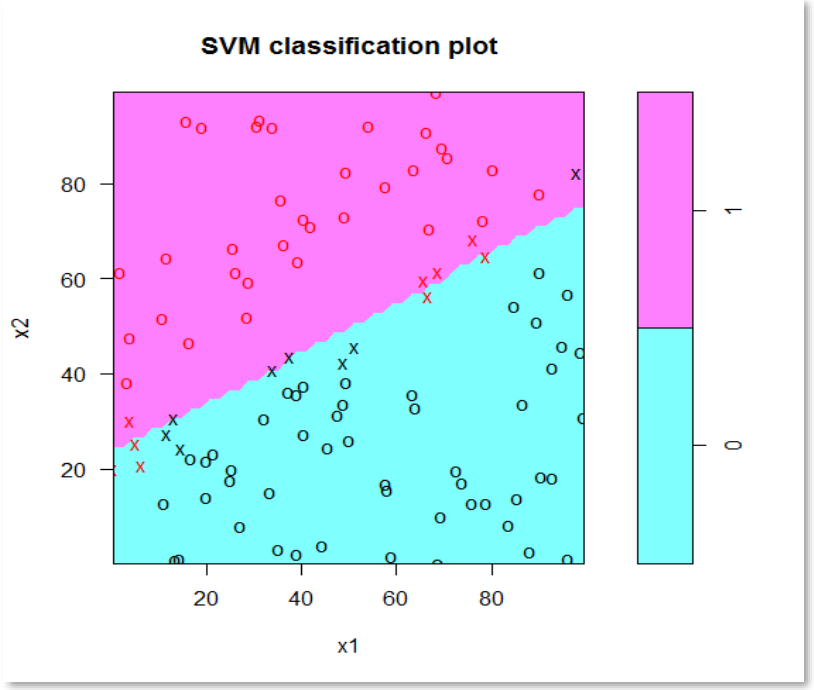


LINEAR SVM

- Let us try fitting an SVM to the simulated data. We will use the function `svm()` from the package `e1071`.
  - Just like decision trees, SVM can in fact perform both regression and classification. We will make sure we pass the outcome variable as a *factor* which will automatically make `svm()` to perform classification rather than regression.
  - Note that we explicitly chose the order of the predictor variables as x2+x1 in the formula. This is done *only* to make the plot command generate the plot with x1 and x2 being the horizontal and vertical axises , respectively (in order to match the plot in the previous slide). Otherwise we could use the formula y~ . , as usual, in order to fit against all available predictors.

```
library(e1071)
dat = data.frame(y=as.factor(y),x1=x1,x2=x2)
# "cost" is related to the parameter A (the allowance for points
# on the wrong side of the margin), but it reflects the *penalty*:
# large A means that we allow many points on the wrong side, i.e.
# the *cost* of misclassification is *low*. And vice versa, small
# A means that we want the cost of misclassification C to be *high*.
svm.fit.l=svm(y~x2+x1 ,data=dat,kernel="linear",cost=1,scale=FALSE)
plot(svm.fit.l,dat) # requires model AND data
```

The plot() command, when applied to an svm fitted model object draws the data points in different colors according to the actual observed class label; in addition, the fitted separating hyperplane is shown with areas on different sides filled with different colors; the support vectors are indicated with crosses, all the remaining data points are circles. We can see that linear boundary provides a very decent fit in this case. There are only a few misclassified data points (red points in cyan area and black points in magenta area).



PREDICTING WITH SVM

- Generic function predict() is overloaded to work with svm fitted objects, just like with many other models we have seen before. Let us assess our model predictions on the training data as well as on the new, independent test dataset generated according to the same rules as the training data:

```
assess.prediction(y,predict(svm.fit.l,dat))
Total cases that are not NA: 100
Correct predictions (accuracy): 94(94%)
TPR (sensitivity)=TP/P: 94.1%
TNR (specificity)=TN/N: 93.9%
PPV (precision)=TP/(TP+FP): 94.1%
FDR (false discovery)=1-PPV: 5.88%
FPR =FP/N=1-TNR: 6.12%
```

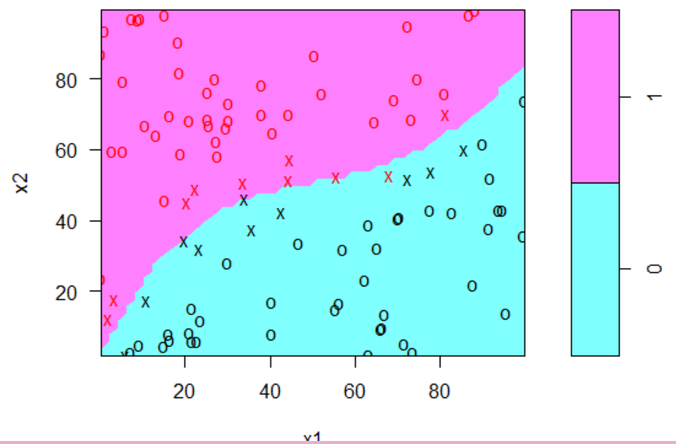
Conclusion: we can see that our model does not overfit as it shows nearly the same performance on an independently drawn dataset. The accuracy is in fact pretty good: it turns out that just a simple linear boundary provides a good approximation to the actual one (which is really just the consequence of how we simulated the data and illustration of the fact that we discussed many times: a simple(r) dependence can sometimes approximate "real" one rather well!)

```
x1.test=runif(100,min=0,max=100)
x2.test=runif(100,min=0,max=100)
y.test=ifelse(x2.test > bound(x1.test),1,0)
dat.test = data.frame(x1=x1.test,x2=x2.test)
assess.prediction(y.test,
     predict(svm.fit.l,newdata=dat.test) )
Total cases that are not NA: 100
Correct predictions (accuracy): 94(94%)
TPR (sensitivity)=TP/P: 95.9%
TNR (specificity)=TN/N: 92.2%
PPV (precision)=TP/(TP+FP): 92.2%
FDR (false discovery)=1-PPV: 7.84%
FPR =FP/N=1-TNR: 7.84%
```

USING KERNELS

- An example of using SVM with a nonlinear kernel (use `kernel=` option ("polynomial", "radial")).
  - Additional options (coef0. gamma, degree) set the parameters: polynomial kernel is (coef0 + gamma*x_i*x_j)^{degree} , radial kernel is exp(-gamma*$(x_i-x_j)^2$) (note that both use parameter 'gamma'). Defaults: gamma=1, coef0=0
  - Let us see if we even have enough data in our synthetic dataset in order to recover the "correct" separating curve:

```
# here we scale data (which is often a good idea);
# without scaling SVM iterative search for solution
# does not converge well in our case! (a warning
# to that effect would be printed)
svm.fit.3=svm(y~x2+x1 ,data=dat,
   kernel="polynomial",d=3,coef0=1,cost=1,scale=T)
plot(svm.fit.3,dat)
assess.prediction(y,predict(svm.fit.3,dat))
Total cases that are not NA: 100
Correct predictions (accuracy): 98(98%)
TPR (sensitivity)=TP/P: 98%
TNR (specificity)=TN/N: 98%
PPV (precision)=TP/(TP+FP): 98%
FDR (false discovery)=1-PPV: 1.96%
FPR =FP/N=1-TNR: 2.04%
```



As choppy as our sparse dataset looks, we seem to be able to recover the nonlinear boundary quite well. The accuracy increased and, most importantly, the *test set* prediction accuracy increases too (do this as an exercise: assess the accuracy of this new model on the test dataset we generated earlier – see previous slide)

OPTIMIZING PARAMETERS

- With such a good fit we were able to obtain outright on our simplistic simulated dataset this might look theoretical and irrelevant (but in practice it is not!), but how does one choose the "right" values of the cost C and the kernel parameters, such as degree, gamma, etc?
  - Just like we did it when we tried to compare different linear (or other) models, with different numbers of predictor variables included and/or with different degrees and combinations of those variables: compare prediction accuracy across multiple models!
  - More parameters (or model that's more "flexible" for any reason in general) always leads to better fit on the training set, but this could be overfitting. We need to consider, as usual, prediction accuracy on a test dataset. Hence, ***cross-validation or bootstrap***.
  - We do know how to do it manually, but e1071 package includes a convenience function `tune()`. Just specify the function (model) you want to cross-validate (we use 'svm', obviously), the fixed parameters to be passed to that model, and the *grid* of parameter(s) of the model we want to evaluate. A separate model will be fitted for each set of parameter values from the grid and then cross-validated. The result of the calculation will be the cross-validation error for each of these models.

EXAMPLE OF SVM PARAMETER TUNING

- Here's the example of how the function tune() is used:
  - We are going to tune both C and degree, so we have a 2D grid of model parameter values. You can of course tune just one parameter, or you can tune more than two (just be mindful on the number of models that needs to be fitted and cross-validated on large multi-dimensional parameter grids – this can become prohibitive on large datasets!)
  - It looks like we were pretty much on the money (nearly the same error as in the best case) with the polynomial SVM we chose in the previous slide!

```
tune.out=tune(svm,y~.,data=dat,kernel="poly",scale=T,
    ranges=list(cost=c(0.001,0.01,0.1,1,5,10),degree=c(2,3)))
summary(tune.out)

Parameter tuning of 'svm':

- sampling method: 10-fold cross validation

- best parameters:
 cost degree
    5      3

- best performance: 0.12
```

```
# <output continued>
- Detailed performance results:
    cost degree error dispersion
1  1e-03      2  0.53 0.13374935
2  1e-02      2  0.53 0.13374935
3  1e-01      2  0.48 0.14757296
4  1e+00      2  0.42 0.07888106
5  5e+00      2  0.42 0.07888106
6  1e+01      2  0.41 0.07378648
7  1e-03      3  0.46 0.24129281
8  1e-02      3  0.25 0.25055494
9  1e-01      3  0.17 0.14181365
10 1e+00      3  0.13 0.09486833
11 5e+00      3  0.12 0.09189366
12 1e+01      3  0.12 0.11352924
```

EXAMPLE WITH REAL DATA (DIGITS)

- Let us use the handwritten digits dataset to see how SVM performs in more realistic settings
  - Load dataset using the same code we used in previous lecture (decision trees).
  - Do not forget to take a subset of the data (again, the same way we did it last time), computation on the full dataset (60K observations) will likely choke your laptop (or you can get on a more serious computational server if you want to play with a full dataset!)
  - First we perform a single one-vs-all fit (digit=0 vs digit is mot 0), same way we did it last time, for direct comparison:

```
m=read.digit.images("train-images-idx3-ubyte.gz") # 60K images
l=read.digit.labels("train-labels-idx1-ubyte.gz")
m.test=read.digit.images("t10k-images-idx3-ubyte.gz") # test set: 10K images
l.test=read.digit.labels("t10k-labels-idx1-ubyte.gz")
set.seed(1234)
sample.idx=sample(60000,5000)
m1=m[sample.idx,]; l1=l[sample.idx]
k=0  # let's choose a digit to look for
Lk=ifelse(l1==k,1,0) # the two-level class variable we are going to fit
Lk.test=ifelse(l.test==k,1,0)  # outcome in the test set
digit.0.svm.fit=svm(D~.,data=data.frame(D=as.factor(Lk),m=m1),kernel="poly",C=1, degree=3,scale=T)
```

DIGIT PREDICTION ACCURACY

- Applying the SVM model we fitted in the previous slide (polynomial kernel) to the test set, we obtain very high prediction accuracy, better than we have seen in the last lecture using the same two-class (one vs all) labels:

```
svm.pred.0=predict(digit.0.svm.fit,newdata = data.frame(m=m.test))
assess.prediction(Lk.test,svm.pred.0)
Total cases that are not NA: 10000
Correct predictions (accuracy): 9940(99.4%)
TPR (sensitivity)=TP/P: 97.2%
TNR (specificity)=TN/N: 99.6%
PPV (precision)=TP/(TP+FP): 96.7%
FDR (false discovery)=1-PPV: 3.35%
FPR =FP/N=1-TNR: 0.366%
```

MULTI-LEVEL OUTCOME

- Last time we had to build a panel of one-vs all models in order to build a multi-level classifier; then we picked the model with the "strongest" (most confident) prediction as the actual digit predicted by the panel.
- Note: that "trick" is very general and can be applied to different types of model
- SVM implementation does have multi-level classification built in: just fit the model against multi-level outcome!

```
digit.svm.fit=svm(D~.,data=data.frame(D=as.factor(l1),m=m1),kernel="poly",C=1, degree=3,scale=T)
svm.pred=predict(digit.svm.fit,newdata = data.frame(m=m.test))
table(l.test,svm.pred)
      svm.pred
l.test    0    1    2    3    4    5    6    7    8    9
     0  963    0    1    1    0   10    3    1    1    0
     1    0 1109    2    2    0    0    5    0   16    1
     2   14   19  956    6    2    6    7   13    7    2
     3    1    9   13  940    1   18    1    9   11    7
     4    1   10    2    0  925    0   12    1    3   28
     5    4    4    3   27    4  822   10    1   11    6
     6    8    8    1    0    3   13  923    2    0    0
     7    1   28   14    4    6    2    0  954    3   16
     8   11    4    6   11    6   13    9    2  898   14
     9    8    9    4    6   21    3    2   12    3  941
sum(diag(table(l.test,svm.pred)))  # result is as good as the best result we could get with random forest!
[1] 9431
```

WHICH MODEL TO USE

- How does SVM perform compared to other models? Which one to choose?
  - Short (but not very encouraging) answer is "it depends"
  - Definitely depends on the structure of the data
  - Also depends on the goals of the analysis (e.g. more interpretable model such as logistic regression can be preferred)
- As we have seen, the loss function of SVM is somewhat similar to that of logistic regression (LR). Some considerations regarding these two models:
  - SVM may perform better on the data that are (almost) separable – note that the separating boundary can be very complex if we are using nonlinear kernels
  - LR with higher order, non-linear terms might offer similar performance (theoretically!) but note that (1) by using the kernel trick SVM does not need to use the high-dimensional space of all the variables and all their higher degrees explicitly (which would result in combinatorial complexity and/or we might not even have enough data points to explicitly fit a model like LR in that space); (2) SVM can be more efficient (only support vectors matter); (3) LR may in fact become unstable when the data are (almost) perfectly separable – the solution might still make sense and work fine but needs to be carefully checked
  - LR explicitly calculates the *probability* of the outcome (label) and in some cases might perform better on poorly separable datasets (with large "grey areas" where both outcomes might occur). The "allowance" A an SVM gives towards the points that fall into the margin/onto the wrong side might become a procedure that's too crude/limited

REFINEMENTS

- It is possible to use “ensemble”/”slow learning” approaches with SVM too (note that all such methods are computationally expensive; the gain might be not worth the improvement achieved)
- Boosting of a classifier: in the last lecture we briefly described boosting and defined it in terms of a regression model
  - We used residuals: the next model in the ensemble (partially) fits the residuals, the next model fits what’s left etc
  - Is it possible to define boosting procedure for a categorical outcome? There is no continuous residual that we can gradually decrease: we either predict correct label (e.g. {0,1}), or not!
  - Solution: introduce *weights*. The misclassification error is the sum of all misclassified examples, each taken with weight w_i
  - We start with all weights equal (we are equally eager to correctly predict each of the training examples)
  - Fit a model and update the weights: for the observations that are predicted correctly, the weights are decreased
  - The “remaining error” now has the misclassified examples weighted stronger. This is what, in a sense, the “updated residual” is (what’s still “remains to be fitted”). Repeat the fitting, update the residual error, etc.
  - The boosted solution is the sum of such iteratively built models.
  - This is a general procedure that can be applied to a number of models (e.g. glm). See the package ‘ada’. Definitely possible with SVM (some R packages do exist too), but too complex for us right now.