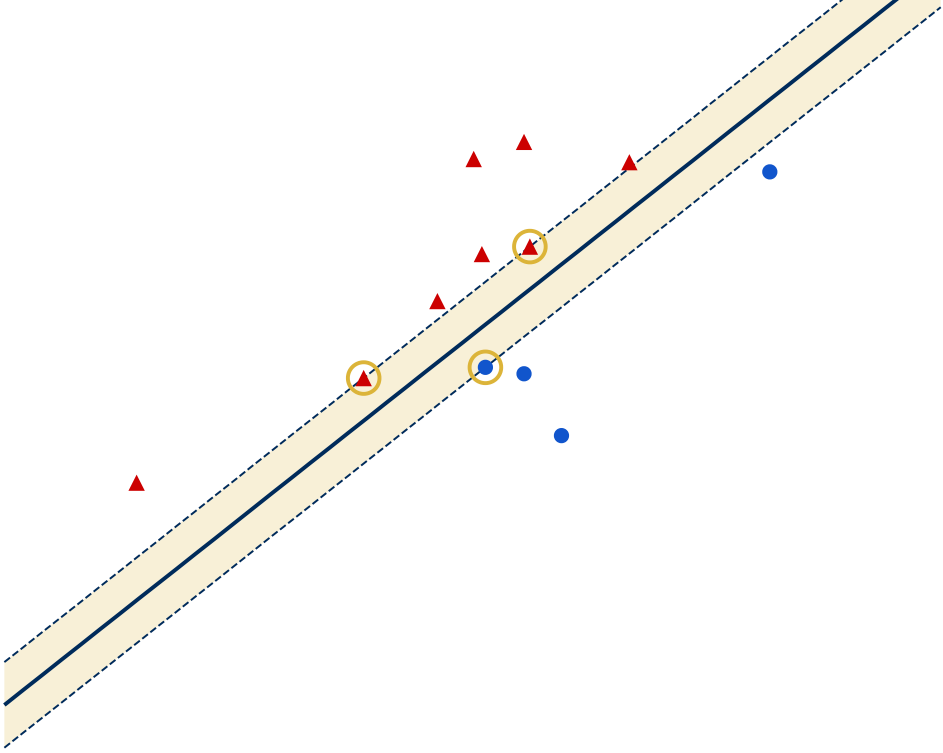


UNIT 02 · CHAPTER 9

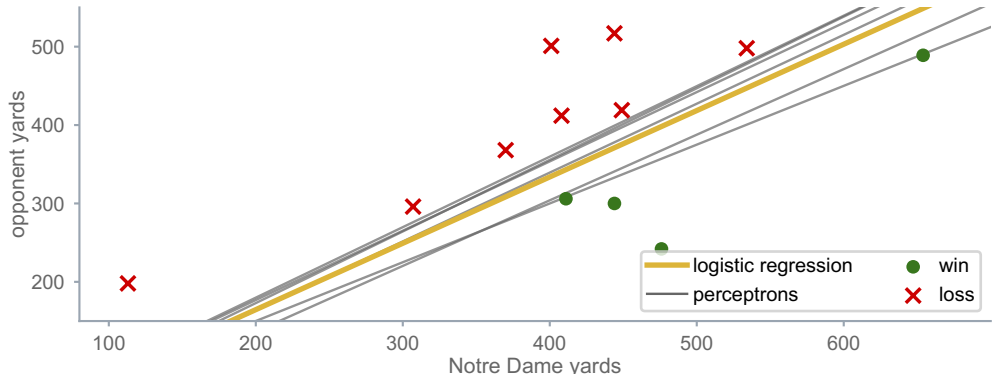
Support Vector Machines

CSE 30124 · Introduction to Artificial Intelligence
William Theisen · University of Notre Dame

Linear Classifiers

Given our dataset of Notre Dame football games, we've been trying to answer one question: **"Did we win?"** Our last two models, the **perceptron** and **logistic regression**, both answer it with a straight line.

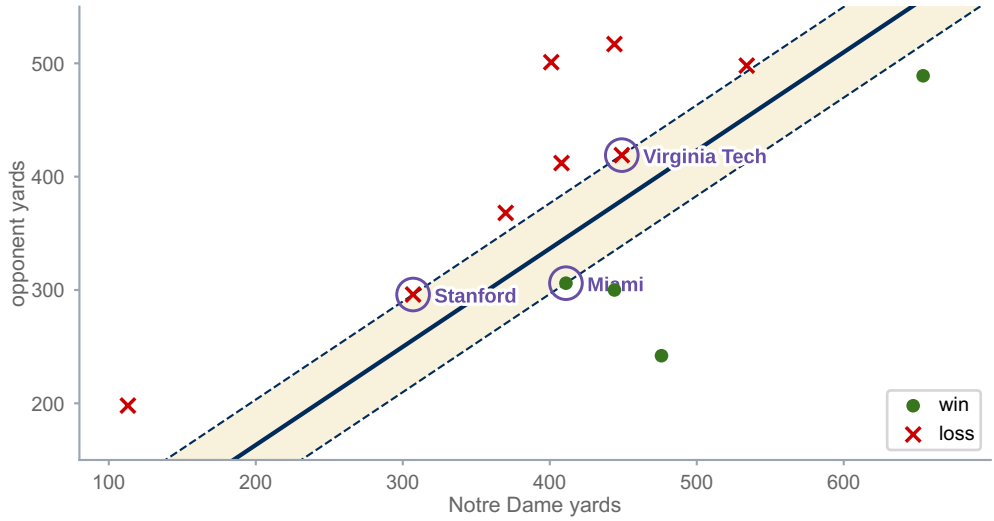
Pictured below is the `2016` season, `4` wins and `8` losses (not a vintage year), plotted by Notre Dame's yards against the opponent's. Each grey line is a perceptron trained on a different shuffle of the same `12` games, and the gold line is a logistic regression. Every one of them gets all `12` games right.



So which line should we use for the next game? Most people would pick one through the middle of the gap, leaving some room on both sides. The training data can't help us here, every line scores `100%` on it. As we saw at the end of Logistic Regression, the games that matter most are the ones nearest the line, so **a computer needs some way to prefer the line that stays furthest from them**.

Margins

One way to say "the middle" is to measure how much room a line leaves. Slide a copy of the line toward the wins until it touches a game, and another copy toward the losses until it touches a game. The empty street between the two copies is called the **margin**. A line with a wider street is less likely to be fooled by the next game, which will land a little off from anywhere we've seen a game before.



Pictured above is the widest street we can fit through the `2016` season. It's about `60` yards wide, and the line we want runs straight down the middle of it. A classifier that picks its line this way is called a **maximum margin classifier**.

Support Vectors

Notice that only three games actually touch the street: losses to Stanford and Virginia Tech, and a win over Miami. (Stanford is an odd one. Notre Dame out-gained them 307 yards to 296 and still lost 10 to 17.) These games are called **support vectors**, since they hold up the edges of the street, and they are **the only games that decide where the line goes**.

If we threw away the other 9 games and found the widest street again, we'd get the same line. Move any of those 9 games around, as long as they stay off the street, and nothing changes. Move a support vector and the street moves with it.

This is the idea behind our next classifier. A **Support Vector Machine (SVM)** is a binary classifier built around the margin. It picks the line with the widest street, and to use that line it only needs to remember the support vectors. Similarly to our decision tree, it boils the training data down to a small model, while kNN has to keep every game it has ever seen.

The Decision Boundary

To find the widest street, a computer needs it written down. Similarly to our perceptron, an SVM scores a game by multiplying its features by weights and adding a bias:

[9.1] the score

$$\hat{y} = \mathbf{x}\,\boldsymbol{\omega}^{\mathsf{T}} + b$$

Here $\mathbf{x}$ is a game's two yard totals (in hundreds), $\boldsymbol{\omega}$ holds one weight per feature, and b plays the part ω_0 played in the perceptron and Linear Regression. A positive score predicts a win and a negative score a loss, so the line itself, the **decision boundary**, is everywhere the score is exactly 0. The edges of the street are where the score is $+1$ and -1.

One Rule for Both Classes

SVMs label the classes $y = +1$ (win) and $y = -1$ (loss) instead of 1 and 0. That allows a handy trick: **the score times the label is positive whenever the game is on the correct side**, whichever class it's in. Here are five 2016 games scored by the SVM from the last page:

game	y	$\hat{y}$	$y\hat{y}$
Army (won)	+1	+4.01	4.01
Miami (won)	+1	+1.00	1.00
Duke (lost)	−1	−1.13	1.13
Stanford (lost)	−1	−1.00	1.00
Texas (lost)	−1	−3.56	3.56

Every 2016 game has a $y\hat{y}$ of at least 1, and the support vectors have exactly 1. So "every game on or past its edge of the street" is one rule for both classes, $y_i(x_i w^T + b) \geq 1$.

Margin Width

Why a score of 1, and not 5? If we double w and b every score doubles, but the boundary, where the score is 0, doesn't move at all. So the 1 is just a convention. What really matters is **how fast the score grows** as we walk away from the boundary. The faster it grows, the sooner a game reaches a score of 1, and the narrower the street.

Walking a distance t straight across the street (in the direction of w) changes the score by $t\|w\|$. The two edges are 2 apart in score, so the width of the street is:

[9.2]

$$\text{margin} = \frac{2}{\|w\|}$$

the width of the street

Our 2016 SVM has w = `[2.16, -2.50]`, so $\|w\|$ = `3.30` and the street is `2/3.30` = `0.61` hundred yards, the 60 or so yards we saw. **Smaller weights mean a wider street**, which should sound familiar from Optimization and Regularization.

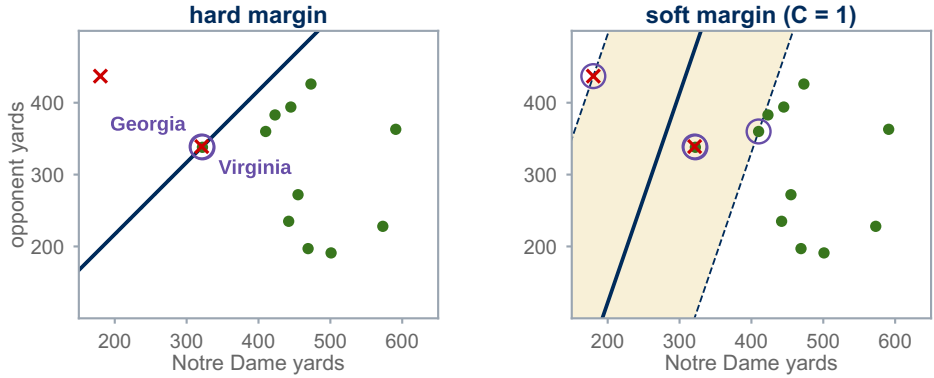
Hard-Margin SVMs

Putting the last two pages together, the widest street comes from the smallest $\|\mathbf{w}\|$ that still keeps every game on or past its edge. This is called a **hard-margin SVM**:

the hard-margin SVM

$$\min_{\mathbf{w},b} \frac{1}{2}\|\mathbf{w}\|^2 \quad \text{subject to} \quad y_i(\mathbf{x}_i\mathbf{w}^T + b) \geq 1 \quad \text{for every } i$$

(Minimizing $\frac{1}{2}\|\mathbf{w}\|^2$ picks the same line as minimizing $\|\mathbf{w}\|$, and is nicer to take derivatives of.) Much like a perceptron, a hard-margin SVM allows no misclassifications. Unfortunately, that makes it **very fragile**. Only `6` of our `11` seasons can be split by a straight line at all, and even when one can, a single game can wreck it.



In `2019` Notre Dame lost to Georgia gaining `321` yards to Georgia's `339`, and beat Virginia gaining `322` to `338`. The hard margin (on the left) has to squeeze between two nearly identical box scores, and its street ends up `1.4` yards wide.

Soft-Margin SVMs

The fix is to let some games **cheat a little**. We give every game a **slack** value ε_i, how far it's allowed to fall short of its edge of the street, and charge for it at a price C:

the soft-margin SVM

$$\min_{w,b,\varepsilon} \ \frac{1}{2}\|w\|^2 + C\sum_i \varepsilon_i$$

$$\downarrow \text{ subject to}$$

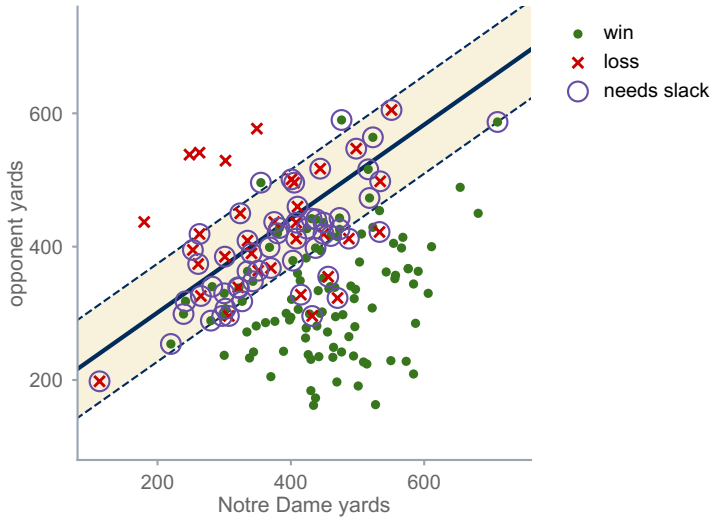
$$y_i(x_i w^\top + b) \geq 1 - \varepsilon_i, \qquad \varepsilon_i \geq 0$$

A game on or past its edge needs no slack, a game inside the street needs a slack between 0 and 1, and a game on the wrong side of the line needs more than 1 (that's exactly what being misclassified means). This is called a **soft-margin SVM**, and it **allows some misclassifications in exchange for a more robust boundary**.

On the right of the last figure, a soft margin with $C = 1$ gives up on Georgia and gets a street 243 yards wide, still getting 12 of the 13 games in 2019 right. Georgia is now the one game on the wrong side of the line, and its slack is what it costs us.

Slack

Slack also means we no longer need a season that a line can split. No line splits all `141` games, and pictured below is a soft margin with $C = 1$ fit to every one of them. The ringed games are the ones that need some slack.



The street is about `120` yards wide and `86%` of games land on the correct side of the line. Of the `141` games, `81` need no slack, `40` sit inside the street but on the correct side, and `20` are on the wrong side. A ringed game is exactly a game with a say in where the line goes, so this SVM has `60` support vectors. **Slack makes a lot more games support vectors**, since a game inside the street is touching it too.

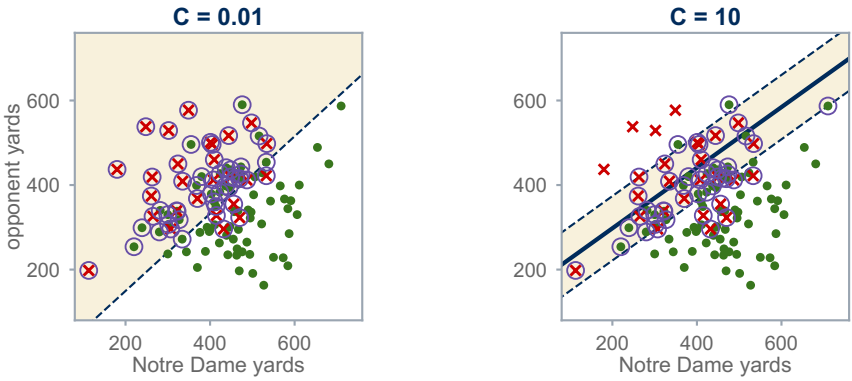
Choosing C

C decides what we care about more, a wide street or fewer games inside it. To see what it does on games the SVM hasn't seen, we can split the games by time like we did in Linear Regression, training on the `103` games from `2012` to `2019` and testing on the `38` from `2020` to `2022`.

C	street (yards)	support vectors	training accuracy	test accuracy
0.01	2,234	55	73.8%	78.9%
0.1	252	54	77.7%	84.2%
1	191	49	80.6%	86.8%
10	124	47	84.5%	**89.5%**

Both accuracies climb as C grows and the street narrows, and past C = 10 nothing changes at all.

Bias and Variance in SVMs



With `C = 0.01` slack is so cheap that the street is over `2,000` yards wide (a football field is `120`) and the SVM predicts a win for every game. That's still right `74%` of the time on the training games, which says more about Notre Dame's schedule than about the model. In Linear Regression's terms it has a **high bias**, it's underfit, and no amount of extra data would help it.

Raising `C` makes slack expensive, so the SVM narrows the street to get more training games right. A straight line in two features can only go so far. With a more flexible boundary, though, a large `C` acts like a hard margin and squeezes the street around single games like Georgia. A model like that does very well on its training games and poorly on new ones, a **high variance**. Underfitting and overfitting are symptoms of the two, and `C` is a dial between them that we set using games the model didn't train on.

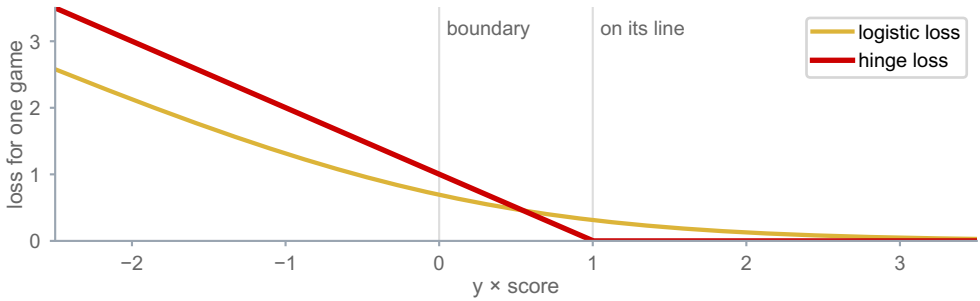
Hinge Loss

For any line, the smallest slack a game needs is how far its $y\hat{y}$ falls short of 1, or nothing if it doesn't. This is called **hinge loss**, $\varepsilon_i = \max(0,\ 1 - y_i\hat{y}_i)$, and with it the soft-margin SVM loses its "subject to":

the same SVM, with hinge loss

$$\min_{w,b} \ \frac{1}{2}\|w\|^2 + C\sum_i \max(0,\ 1 - y_i\hat{y}_i)$$

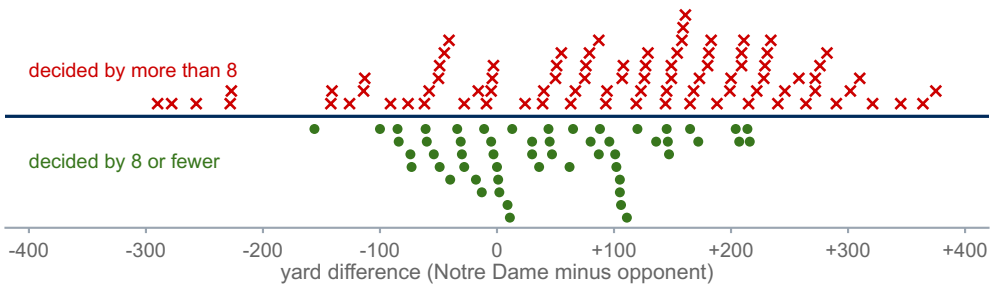
That's an L2 penalty on the weights plus a loss on every game, the same recipe as a regularized logistic regression, just with a different loss. One twist is that C multiplies the loss instead of the penalty, so **a big C means weak regularization**.



The difference between the two losses is on the right of the plot. Logistic regression keeps charging a game a little even when it's well past the line (`0.13` for a $y\hat{y}$ of 2), while hinge loss charges it exactly nothing. So in an SVM **games past their edge of the street have no say at all**, which is why only the support vectors matter.

Non-Linearly Separable Data

A soft margin handles a few awkward games, but sometimes a straight line is the wrong shape altogether. Say instead of "Did we win?" we ask "Was it close?", meaning decided by `8` points (one score) or fewer. `55` of our `141` games were. Pictured below is every game placed by its yard difference, Notre Dame's yards minus the opponent's.

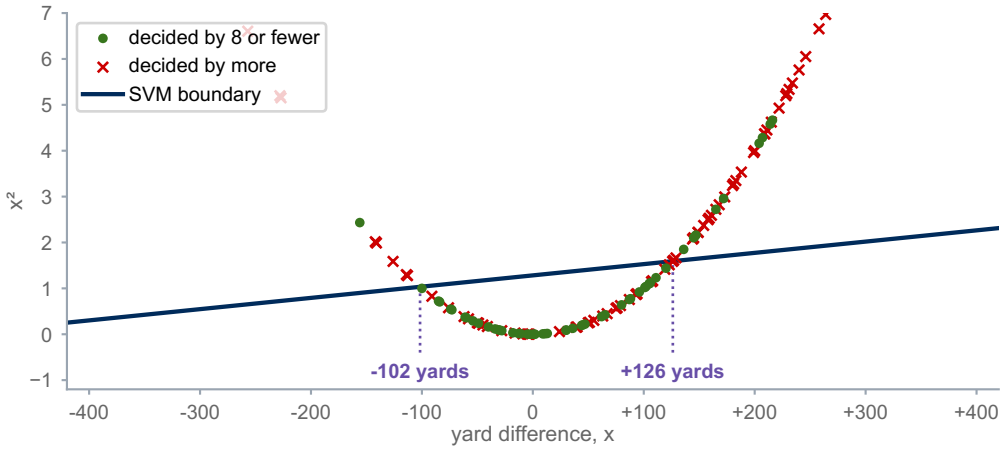


Close games pile up in the middle, and the ends of the line are all blowouts (out-gaining someone by `300` yards rarely ends in a one-score game, and neither does being out-gained by `300`). With one feature a line is just a single cut, and **one cut can't isolate the middle**. A linear SVM ends up with a weight of `0`, predicting "not close" for every game, which is right `61%` of the time.

Slack can't fix this either, because the trouble is the shape of the data itself.

Changing the Space

What if we add a second feature, the yard difference squared? It's the same trick as polynomial regression in Linear Regression, building new features out of old ones. Each game now sits on a parabola, the close games near the bottom, and in this new space a straight line does the job:



With x in hundreds of yards, the SVM's line in the new space is $\hat{y} = 0.19x - 0.78x^2 + 1.00$, and back on the original number line that's two cuts, at -102 and $+126$ yards. Anything in between is predicted to be close, and accuracy goes from 61% to 69%. (Football is noisy, a game where the yards come out even can still be a blowout.) **A straight line in the new space is a curve in the old one**, and changing your space is a very powerful idea.

Too Many Features

So why not add features every time? Unfortunately, they multiply quickly. In Linear Regression, every square and product of seven box-score stats was already `35` features, and going up to cubes made `119`. Our dataset has `26` box-score stats (`13` for each team):

features we start with	up to squares	up to cubes	up to 5th powers
1 (yard difference)	2	3	5
7 (Linear Regression)	35	119	791
26 (every box-score stat)	377	3,653	169,910

Every one of those features has to be built and stored for every game, and every one needs its own weight. What we'd really like is **the effect of the extra features without ever building them**. It turns out an SVM can do this, because its math can be rewritten so that the games only ever appear in pairs, as dot products. Doing the rewrite takes one more tool.

Lagrangian Multipliers

Our SVM is a minimization with rules attached, one "subject to" per game, and rules are awkward to optimize. **Lagrangian multipliers** are a strategy for finding the local maxima and minima of a function subject to constraints. Essentially, they **wrap the "subject to" into a single cost function**. Each rule gets its own multiplier α, and breaking the rule costs us something. To minimize $f(x)$ subject to $g(x) \geq 0$:

minimize f subject to g ≥ 0

$$L(x, \alpha) = f(x) - \alpha g(x), \qquad \alpha \geq 0$$

We have actually seen this before. In Optimization and Regularization we wrote ridge regression two ways, as a budget (minimize the SSE subject to $\Sigma w_j^2 \leq t$) and as a cost (minimize SSE + $\lambda \Sigma w_j^2$), and claimed that every budget has a λ that gives the same answer. That λ is a Lagrangian multiplier, and the cost version is the budget with its "subject to" wrapped in (the rule becomes $-\lambda(t - \Sigma w_j^2)$, and the leftover $-\lambda t$ doesn't move the best weights).

The SVM Lagrangian

For the hard-margin SVM, game i's rule is $y_i(x_i w^T + b) - 1 \geq 0$, so it gets a multiplier $\alpha_i \geq 0$. Similarly to OLS, the best weights are where the slopes of L in w and b are zero:

one multiplier per game

$$L = \frac{1}{2}\|w\|^2 - \sum_i \alpha_i \left[y_i (x_i w^T + b) - 1 \right]$$

↓ set the slopes in w and b to 0

$$w = \sum_i \alpha_i y_i x_i, \qquad \sum_i \alpha_i y_i = 0$$

The first result says **the best weights are a weighted sum of the games**. A game that clears its edge of the street with room to spare gets $\alpha_i = 0$ and adds nothing, so only the support vectors count. The second says the pushes from the two classes balance out.

For `2016` the three support vectors get multipliers of `2.98` (Stanford), `2.47` (Virginia Tech) and `5.46` (Miami). The pushes balance (`2.98` + `2.47` = `5.46`, losses against the win), and adding up $\alpha_i\ y_i\ x_i$ over just those three games gives w = `[2.16, −2.50]`, the same weights as before.

The Dual Form

Now we plug $w = \Sigma \alpha_i\, y_i\, x_i$ back into L. The b term vanishes (it's multiplied by $\Sigma \alpha_i\, y_i = 0$), and the other two pieces become:

$$\frac{1}{2}\| w\|^2 = \frac{1}{2}\sum_i \sum_j \alpha_i \alpha_j y_i y_j\, x_i x_j^T$$

$$\sum_i \alpha_i y_i\, x_i w^T = \sum_i \sum_j \alpha_i \alpha_j y_i y_j\, x_i x_j^T$$

The second sum is twice the first, so L is left with $\Sigma \alpha_i$ minus one copy of the double sum. The weights and the bias are gone, and L depends only on the α's. The best multipliers make what's left as big as possible, so we maximize. Adding slack back only puts a cap of C on each multiplier. This is the **dual form** of the SVM:

the dual form

$$\max_{\alpha} \sum_i \alpha_i - \frac{1}{2}\sum_i \sum_j \alpha_i \alpha_j\, y_i y_j\, x_i x_j^T$$

↓ subject to

$$0 \le \alpha_i \le C, \qquad \sum_i \alpha_i y_i = 0$$

Notice that the games only ever show up in pairs, as dot products $x_i\, x_j^T$, so **the dual never needs a game's features on their own**.

Solving the Dual by Hand

To see the dual work, let's shrink the problem to two games and one feature, the yard difference rounded to the nearest hundred. In `2016` Notre Dame lost to Michigan State while being out-gained by `100` yards ($x_1 = -1$, $y_1 = -1$) and beat Miami while out-gaining them by `105` ($x_2 = +1$, $y_2 = +1$).

The balance rule says $-\alpha_1 + \alpha_2 = 0$, so both games get the same multiplier, call it a. With one feature the double sum is just $(\sum \alpha_i y_i x_i)^2$, and here each $y_i x_i$ is `1`:

the dual with both multipliers equal to a

$$q(a) = 2a - \frac{1}{2}(a + a)^2 = 2a - 2a^2$$

↓ set the derivative $2 - 4a$ to 0

$$a = 0.5, \qquad w = \sum_i \alpha_i y_i x_i = 0.5 + 0.5 = 1$$

↓ Miami sits on its edge, $y_2(x_2 w + b) = 1$

$$b = 1 - x_2 w = 0$$

Our SVM is $\hat{y} = x$. The boundary is at a yard difference of `0`, right between the two games, and the street runs from `-100` to `+100` yards, a margin of $2/\|w\| = 2$ hundred yards. Both games sit exactly on its edges, so both are support vectors, and the dual's best value ($q(0.5) = 0.5$) is exactly $\frac{1}{2}\|w\|^2$, the cost of the original problem.

The Kernel Trick

Back to our close games. To add x^2 we replaced each game's x with a longer list of features, call it $\phi(x)$. The dual only ever needs $\phi(x_i)\phi(x_j)^T$, so if some function computed that dot product straight from x_i and x_j, we'd never have to build ϕ at all. A function like that is called a **kernel**, and swapping it in for every dot product is the **kernel trick**:

[9.3] a kernel

$$K(x, x') = \phi(x)\,\phi(x')^T$$

For example, $K(x, x') = (xx' + 1)^2$ is the dot product of $\phi(x) = [1, \sqrt{2}x, x^2]$ with $\phi(x')$. Take Boston College in `2020` (Notre Dame out-gained them by `200`, $x = 2$) and Michigan State in `2016` ($x = -1$). Building the features gives $[1, 2.83, 4]$ and $[1, -1.41, 1]$, with a dot product of $1 - 4 + 4 = 1$. The kernel gets there in one step, $(2 \cdot (-1) + 1)^2 = 1$.

What makes SVMs so powerful is this ability to **implicitly increase the dimensionality of our data**. To classify a new game we put the kernel into the score as well, $\hat{y}(x) = b + \Sigma \alpha_i\, y_i\, K(x, x_i)$, and since every other game has $\alpha_i = 0$, that sum only runs over the support vectors.

Common Kernels

There are three common kernels:

linear

$$K(x, x') = x\,x'^{T}$$

polynomial

$$K(x, x') = (\gamma\, x\, x'^{T} + r)^{p}$$

RBF

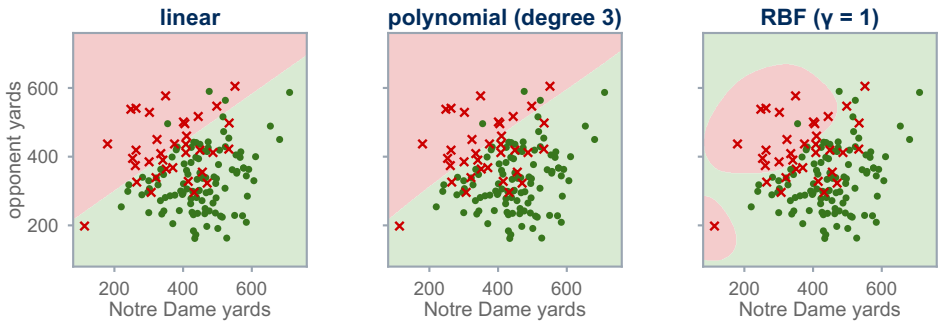
$$K(x, x') = \exp(-\gamma \| x - x' \|^{2})$$

The **linear kernel** is just the dot product we started with, so it gives back the plain SVM from the start of the chapter. The **polynomial kernel** of degree p covers every product of up to p features. For our `26` box-score stats at degree `3` that's the `3,653` features from the table, for the price of one `26`-long dot product and a cube.

The **radial basis function (RBF) kernel** measures how close two games are instead. Two identical games score `1`, and the score falls toward `0` as they move apart. With yards in hundreds and $\gamma = 1$, two games `50` yards apart score `0.78`, `100` yards apart `0.37`, and `200` yards apart `0.02`. It turns out the RBF kernel is a dot product in a space with **infinitely many dimensions**, which we could never build, and it costs one subtraction and an exponential. γ sets how far each game's influence reaches, so a bigger γ gives a wigglier boundary.

Comparing Kernels

Back to "Did we win?" on all 141 games, with the same two features our kNN and decision tree used (both teams' yards, in hundreds). Below are SVMs with each of the three kernels ($C = 1$):



model	accuracy on unseen games	games it keeps
kNN (k = 5)	81.6%	141
decision tree (depth 3)	82.3%	0
linear SVM	83.0%	60
RBF SVM (γ = 1)	82.9%	70

The boundaries look quite different, and they all land at about 82% to 83% (the polynomial SVM too, at 82.2%). With only yards to go on, **no kernel can find information that isn't in the features**. An RBF with $\gamma = 10$ and $C = 100$ gets 98.6% of training games right and only 68.8% of unseen ones, worse than always guessing a win (75.2%).

SVMs in sklearn

We'll never solve the dual by hand for real data. sklearn's `SVC` does it for us, with the kernel, C and γ as arguments. Here it is trained on `2012` to `2019` and tested on the three seasons after:

```
import pandas as pd
from sklearn.svm import SVC

games = pd.read_csv("nd_football.csv")
X = games[["nd_total_yards", "opp_total_yards"]] / 100
y = games["win"]
train = games["season"] <= 2019     # 103 games

for kernel in ["linear", "rbf"]:
    svm = SVC(kernel=kernel, C=1)
    svm.fit(X[train], y[train])
    print(kernel, len(svm.support_),
          round(svm.score(X[~train], y[~train]), 3))
```

OUTPUT

```
linear 49 0.868
rbf 55 0.895
```

`support_` holds the positions of the support vectors (the games with $\alpha_i > 0$), so the linear SVM keeps `49` of its `103` training games and the RBF keeps `55`. Our labels can stay `0` and `1`, sklearn turns them into `-1` and `+1` itself. Notice that we still divide the yards by `100`. The RBF kernel works with distances, so **the units of each feature change the answer**, the same reason we rescaled features before LASSO.