Notes for ISL Chapter 9: Support Vector Machines

Justin Burruss

2026-01-19

Background

The Python code and notes below are for the *Introduction to Statistical Learning* (ISL) study group. This is to try out some of the concepts from Chapter 9. The book illustrates the concepts from this chapter using a mix of synthetic data and the `Heart` data set, and as we already used `Heart` for chapter 2, for this chapter we'll go with some synthetic blobs in two dimensions. The document was created in RMarkdown with the Python code running via the `reticulate` library plus a little LaTeX.

Our ground rule: when implementing concepts from the chapter, just use basic Python + Pandas + NumPy. It's OK to use more when visualizing or evaluating results.

Notation

The text uses this notation for hyperplanes:

$$\beta_0 + \beta_1 X_1 + \beta_2 X_2 + \cdots + \beta_p X_p = 0$$

So that we're not typing out all those betas, we'll go with a more compact vector notation with `w` being our vector of weights. We won't do the "vector of ones" thing this time, so we keep an intercept term `b` on its own, like this:

$$\mathbf{w} \cdot \mathbf{x} + b = 0$$

The book (and some other materials) all refer to our "slack variable" as `C`, and we will follow that here.

Sequential Minimal Optimization (SMO)

This chapter is all about support vector machines (SVMs), an extension of support vector classifiers (SVCs) for non-linear decision boundaries.

On page 379 we're given:

> We have not discussed exactly how the support vector classifier is computed because the details become somewhat technical.

We'll look outside of *Introduction to Statistical Learning* for the details. If we look at chapter 12 in the second edition of *Elements of Statistical Learning* we see that we'll need to tackle a constrained optimization problem with second order terms. As of version 1.26.4, NumPy does *not* have a a built-in optimizer for this class of problem. Our ground rule of using just basic Python + Pandas + NumPy means we will want to look outside the text for some algorithm suited to support vector machines that we can implement using just

Python + NumPy. We'll go with the SMO algorithm described by John Platt in the 1998 technical report "Sequential Minimal Optimization: A Fast Algorithm for Training Support Vector Machines". The report lays out all the relevant mathematics and even includes pseudo-code.

Upshot here is we'll follow SMO as described in the technical report: this means coding up a big while loop that sequentially fits just two Lagrange multipliers at a time.

Sketching a plan:

- Code up a few kernels from the text
- SMO outer loop examines training points sequentially
- SMO `examineExample` uses heuristics the select second point
- SMO `takeStep` computes the Lagrange multiplier `alpha`
- Write a function to visualize the possibly non-linear boundaries

First, the kernels.

Kernels

We'll implement the kernels listed on page 380-381. As noted in the source materials, we can use non-linear kernels to give as a way to support a non-linear boundary between classes.

Linear

This is given on page 380 in equation 9.21. With Python and NumPy, the linear kernel is just the `@` operator, so we're just doing `a @ b`.

```
lambda a, b: a @ b
```

Polynomial

See equation 9.22 in the text. Now we have `(a @ b + 1)**d`. Let's go our own way here and parameterize the hard-coded +1 part of the kernel so we can express it as follows.

```
lambda a, b: (a @ b + c) ** d
```

So for instance, if we want a cubic polynomial we set `d=3`.

Radial

This follows from equation 9.24. Now we have a γ (lowercase Greek letter gamma) parameter which needs to be a positive constant. We'll use `@` again and add `np.exp` into the mix to express the kernel like this:

```
lambda a, b: np.exp(-gamma * (a - b) @ (a - b))
```

A larger `gamma` will give us a tighter shape.

SVMs using SMO

The technical report gives us everything we need. To make it easier to follow along with the algorithm, let's use the same variable names to the extent possible.

The code has backup heuristics that only get exercised in unusual circumstances, so it's not clear that a given test will really be validating the full code. We'll add a little lightweight logging to our implementation so we can see if we're hitting the edge cases. In Python this is straightforward with a list and `.append()`.

One other thing is the report makes no mention of normalizing the inputs as part of SMO. The algorithm should work provided we set the various parameters correctly, but leaving the inputs unscaled is going to make it harder for us to get these parameters dialed in for each input, especially in light of the fact that we're going to have three kinds of kernels. Let's add a little preprocessing so that our mean is recentered to zero and standard deviation is rescaled to 1. Then for example we can simply set the `C` parameter to 0.1 or 1 or 10 and already have some expectation about how that's supposed to look.

Karush-Kuhn-Tucker (KKT) conditions

The thing we're working toward in the algorithm follows from a generalization of Lagrange multipliers with the upshot that the problem is solved when for all i the Lagrange multipliers α_i satisfy:

$$\begin{aligned} \alpha_i = 0 &\iff y_i u_i \geq 1, \\ 0 < \alpha_i < C &\iff y_i u_i = 1, \\ \alpha_i = C &\iff y_i u_i \leq 1 \end{aligned}$$

where u_i is the output for the ith training example. We're working with floating point operations here, so the algorithm allows for a tolerance for "close enough to zero". We also have a *second* "close enough to zero" setting used elsewhere in the algorithm.

See the appendix for a refresher on Lagrange multipliers.

Second derivative η

The appendix of the technical report provides the derivation of η (lowercase Greek Eta), the second derivative of the objective function Ψ. The bottom line is we get to:

$$\eta = K(x_1, x_1) + K(x_2, x_2) - 2K(x_1, x_2)$$

where $K()$ is our kernel function. This lets us—in most cases—calculate the second alpha as just the old alpha plus $\frac{1}{\eta} y_2(E_1 - E_2)$. This makes it easier for the computer, but we can see that it's important we keep our error cache accurate, otherwise our new alphas will drift off.

Error cache

The technical report doesn't spell out *exactly* how best to update the error cache at the end of every step where we've updated alphas. The approach we'll take here is to update the errors *incrementally* rather than running a full `predict`. In other words, we're going to increment the error by the calculated change in error based on the alphas we just changed. This is how that will look for the other alphas that are not being updated.

```
        for i in other_alpha:
            self.E[i] += (
                y1 * (a1 - alph1) * self.kernel(self.point[i1], self.point[i]) +
                y2 * (a2 - alph2) * self.kernel(self.point[i2], self.point[i]) -
                (self.b - b_old))
```

We ought to acknowledge the risk that repeated incremental updates could end up accumulating floating point errors over time, but we likely will never notice anything with our small datasets that converge in just a few dozen passes. The benefit is this will run much faster.

`SV` class

Here's our plan for this class.

- one class for all three kernels, just make kernel a parameter in `__init__`
- parameters for `__init__` include:
  - `C`
  - kernel name: linear, polynomial, or radial
  - `c` and `d` for polynomial kernels
  - `gamma` for radial kernels
  - `tol` for a tolerance, report used 0.001, that will be our default
  - `eps` for a different tolerance epsilon (ϵ)
  - `seed` for our random seed for `np.random.default_rng`
- use `fit`, `predict`, and `decision_function` method names for convenient comparisons with sklearn implementations
  - normalize our inputs
- use same variable names as pseudo-code from technical report
- SMO outer loop goes in `fit` method
- SMO `examineExample` procedure implemented in `__examineExample` method
- SMO `takeStep` procedure implemented in `__take_step` method
- log occurrence of "non-positive η" heuristic, are we really exercising that code?

An implementation is given below. Comments like "equations 13 and 14 (page 7)" refer to the equation numbers and page numbers in the technical report.

```python
import numpy as np
import matplotlib.pyplot as plt
from matplotlib.lines import Line2D
from sklearn.datasets import make_blobs
from sklearn.svm import SVC

class SV:
    """Train a support vector classifier, i.e., max "soft margin" hyperplane"""
    def __init__(self,
                 C: float = 1.0,
                 kernel_name: str = 'linear',
                 c: float | None = None,
                 d: float | None = None,
                 gamma: float | None = None,
                 tol: float = 1e-3,
                 eps: float = 3e-10,
                 seed: int = 2026):
        if kernel_name not in {'linear', 'polynomial', 'radial'}:
            raise ValueError(f"Unknown kernel_name: {kernel_name!r}")
        self.C = C
        self.kernel_name = kernel_name
        self.c = c
        self.d = d
        self.gamma = gamma
        self.tol = tol
        self.rng = np.random.default_rng(seed)
        self.eps = eps
        self.kernels_ = {
            'linear': lambda a, b: a @ b,
            'polynomial': lambda a, b: (a @ b + self.c) ** self.d,
            'radial': lambda a, b: np.exp(-self.gamma * (a - b) @ (a - b))}

        # adding for normalization
        self.mean_ = 0.0
        self.scale_ = 1.0

    def kernel(self,
               a: np.ndarray,
               b: np.ndarray) -> float:
        """Return K(a, b)"""
        return self.kernels_[self.kernel_name](a, b)

    def decision_function(self,
                          X: np.ndarray) -> np.ndarray:
        """Return the signed distance to the separating hyperplane"""

        # rescale to match our fit (which was on rescaled X)
        X = (X - self.mean_) / self.scale_

        # for linear SVM, we can use weights and bias directly
        if self.kernel_name == 'linear':
            return X @ self.w - self.b

        # for all other cases, use a kernel from our kernel function and support vectors
        K = np.array([[self.kernel(xi, xj) for xj in self.support_vectors_]for xi in X])
        return K @ self.dual_coef_ - self.b

    def predict(self, X: np.ndarray) -> np.ndarray:
        """Predict classes (+1 or -1) based on input X"""
        return np.sign(self.decision_function(X))

    def fit(self,
            X: np.ndarray,
            Y: np.ndarray) -> None:
        """Fit the X and Y where Y has classes 1 and -1"""

        # rescale
        self.mean_ = X.mean(axis=0)
        self.scale_ = X.std(axis=0)
        self.scale_[self.scale_ == 0.0] = 1.0
        X = (X - self.mean_) / self.scale_

        # defaults for c and d if not already set
        if self.kernel_name == 'polynomial':
            if self.c is None:
                self.c = 1.0 # book uses 1, see page 380
            if self.d is None:
                self.d = 3.0 # default to a cubic

        # default for gamma if not already set
        if self.kernel_name == 'radial' and self.gamma is None:
            self.gamma = 1.0 / X.shape[1] # default to 1 / num_dimensions

        # always re-initialize with any new fit
        self.point = X
        self.target = Y
        self.alpha = np.zeros(len(Y))
        self.b = 0
        if self.kernel_name == 'linear':
            self.w = np.zeros(X.shape[1])
        else:
            self.w = None
        self.E = -Y.copy().astype(float)
        self.support_ = None
        self.support_vectors_ = None
        self.dual_coef_ = None
        self.fit_log = []
        pass_cnt = 0

        # after one pass through all training data, iterate over training cases
        # whose Lagrange multiplier are neither 0 nor C
        num_changed = 0
        examine_all = True
        while (num_changed > 0 or examine_all):
            pass_cnt += 1
            num_changed = 0
            if examine_all:
                for i2 in range(len(self.target)):
                    num_changed += self.__examineExample(i2)
            else:
                idx = np.nonzero((self.alpha > 0) & (self.alpha < self.C))[0]
                for i2 in idx:
                    num_changed += self.__examineExample(i2)
            if examine_all:
                examine_all = False
            elif num_changed == 0:
                examine_all = True

            # log the history to see how quickly we fit
            self.fit_log.append(f"Pass {pass_cnt} changed {num_changed}")

        self.support_ = np.nonzero(self.alpha > self.eps)[0]
        self.support_vectors_ = self.point[self.support_]
        self.dual_coef_ = (self.alpha[self.support_] * self.target[self.support_])
        self.fit_log.append(f"Ending after {pass_cnt} passes")

    def __take_step(self,
                    i1: int,
                 i2: int) -> bool:
        if (i1 == i2):
            return False

        alph1, alph2 = self.alpha[i1], self.alpha[i2]
        y1, y2 = self.target[i1], self.target[i2]
        E1, E2 = self.E[i1], self.E[i2]
        s = y1*y2

        # compute L, H via equations 13 and 14 (page 7)
        if y1 != y2:
            L = max(0, alph2 - alph1)
            H = min(self.C, self.C + alph2 - alph1)
        else:
            L = max(0, alph2 + alph1 - self.C)
            H = min(self.C, alph2 + alph1)

        if L == H:
            return False

        k11 = self.kernel(self.point[i1], self.point[i1])
        k12 = self.kernel(self.point[i1], self.point[i2])
        k22 = self.kernel(self.point[i2], self.point[i2])
        eta = k11 + k22 - 2*k12
        if eta > self.eps:

            # compute minimum along the direction of the constraint (equations 16 and 17)
            a2 = np.clip(alph2 + y2*(E1 - E2)/eta, L, H)

        else:

            # SMO will work even when eta is not positive, in which case
            # the objective function Psi should be evaluated at each end
            # of the line segment (equation 19)
            f1 = y1 * (E1 + self.b) - alph1 * k11 - s * alph2 * k12
            f2 = y2 * (E2 + self.b) - s * alph1 * k12 - alph2 * k22
            L1 = alph1 + s * (alph2 - L)
            H1 = alph1 + s * (alph2 - H)
            Psi_L = L1 * f1 + L * f2 + 0.5 * L1**2 * k11 + 0.5 * L**2 * k22 + s * L * L1 * k12
            Psi_H = H1 * f1 + H * f2 + 0.5 * H1**2 * k11 + 0.5 * H**2 * k22 + s * H * H1 * k12

            if Psi_L < Psi_H - self.eps:
                a2 = L
                self.fit_log.append("Had non-positve eta, set a2 to L")
            elif Psi_L > Psi_H + self.eps:
                a2 = H
                self.fit_log.append("Had non-positve eta, set a2 to H")
            else:
                a2 = alph2
                self.fit_log.append("Had non-positve eta, set a2 to alph2")

        if abs(a2 - alph2) < self.eps * (a2 + alph2 + self.eps):
            return False
        a1 = alph1 + s * (alph2 - a2)

        # update threshold b to reflect change in Lagrange multipliers.
        b_old = self.b
        if 0 < a1 < self.C:

            # equation 20: the following threshold b1 is valid when the new
            # alpha1 is not at the bounds
            self.b = E1 + y1 * (a1 - alph1) * k11 + y2 * (a2 - alph2) * k12 + self.b

        elif 0 < a2 < self.C:

            # equation 21: the following threshold b2 is valid when the new
            # alpha2 is not at the bounds
            self.b = E2 + y1 * (a1 - alph1) * k12 + y2 * (a2 - alph2) * k22 + self.b

        else:

            # if both new Lagrange multipliers are at bound and if L is not equal
            # to H, then SMO chooses the threshold halfway between the two
            self.b = 0.5 * (
                E1 + y1 * (a1 - alph1) * k11 + y2 * (a2 - alph2) * k12 +
                E2 + y1 * (a1 - alph1) * k12 + y2 * (a2 - alph2) * k22 +
                2 * self.b)

        # for linear kernels, update weight vector to reflect change in a1 & a2
        if self.kernel_name == 'linear':
            self.w += y1 * (a1 - alph1) * self.point[i1] + y2 * (a2 - alph2) * self.point[i2]

        # store new Lagrange multipliers
        self.alpha[i1] = a1
        self.alpha[i2] = a2

        # incrementally update error cache using new Lagrange multipliers
        for i, alph in zip([i1, i2], [a1, a2]):
            if 0.0 < alph < self.C:
                self.E[i] = 0.0
        other_alpha = np.nonzero(~np.isin(np.arange(len(self.E)), i))[0]
        for i in other_alpha:
           self.E[i] += (
               y1 * (a1 - alph1) * self.kernel(self.point[i1], self.point[i]) +
               y2 * (a2 - alph2) * self.kernel(self.point[i2], self.point[i]) -
               (self.b - b_old))

        return True

    def __examineExample(self,
                         i2: int) -> int:
        y2 = self.target[i2]
        alph2 = self.alpha[i2]
        E2 = self.E[i2]
        r2 = E2 * y2
        if ((r2 < -self.tol and alph2 < self.C) or
                (r2 > self.tol and alph2 > 0)):
            bound_idx = np.nonzero((self.alpha > 0) & (self.alpha < self.C))[0]

            if len(bound_idx) > 0:

                # second choice heuristic: approximately maximize the step size
                # by pairing i2 with the i1 that has the largest error
                i1 = np.argmax(np.abs(self.E - self.E[i2]))
                if self.__take_step(i1, i2):
                    return 1

            # second choice heuristic, continued: if the first approach fails
            # to make progress, iterate through the non-bound examples randomly
            self.rng.shuffle(bound_idx)
            for i1 in bound_idx:
                if self.__take_step(i1, i2):
                    return 1

            # second choice heuristic, continued: if the second approach fails
            # to make progress, iterate through the bound examples, too
            non_bound_idx = np.nonzero(
                ~np.isin(np.arange(len(self.E)), bound_idx))[0]
            self.rng.shuffle(non_bound_idx)
            for i1 in non_bound_idx:
                if self.__take_step(i1, i2):
                    return 1
        return 0
```

There's a lot that can go wrong with the various parameters and edge cases, so let's make sure we have a clean way of visualizing boundaries and start with some linear cases where a good boundary is more or less obvious.

These boundaries are a perfect place to use `contour` to plot contour lines. Let's code that up.

`viz_blob_boundaries` function

We define a contour plot function to compare the hand-coded SVM versus sklearn.

```
def viz_blob_boundaries(X: np.ndarray,
                        Y: np.ndarray,
                        m1,
                        m2,
                        m1_label: str = 'Hand-coded',
                        m2_label: str = 'sklearn') -> None:
    """Contour plot to compare hand-code and sklearn versions"""

    # mesh grid in original feature space
    xx, yy = np.meshgrid(
        np.linspace(X[:, 0].min() - 1.5, X[:, 0].max() + 1.5, 300),
        np.linspace(X[:, 1].min() - 1.5, X[:, 1].max() + 1.5, 300)
    )
    grid = np.c_[xx.ravel(), yy.ravel()]
    # decision values
    Z_hand = m1.decision_function(grid).reshape(xx.shape)
    Z_sklearn = m2.decision_function(grid).reshape(xx.shape)

    fig, ax = plt.subplots(figsize=(6, 6))

    # decision boundaries
    ax.contour(xx, yy, Z_hand, levels=[0],
               colors='k', linewidths=2,
               linestyles='-')
    ax.contour(xx, yy, Z_sklearn, levels=[0],
               colors='r', linewidths=2, linestyles='--')

    # data
    ax.scatter(X[:, 0], X[:, 1], c=Y, cmap='Paired',
               s=12, alpha=0.8)

    ax.set_aspect('equal', adjustable='box')
    ax.set_title(f"SVM Decision Boundary Comparison ({m1.kernel_name})")
    ax.set_xticks([])
    ax.set_yticks([])

    # manual legend (contour doesn't auto-label)
    legend_lines = [
        Line2D([0], [0], color='k', lw=2, linestyle='-',
               label='Hand-coded'),
        Line2D([0], [0], color='r', lw=2, linestyle='--',
               label='sklearn'),
    ]
    ax.legend(handles=legend_lines)

    plt.show()
```

Now let's test it on a case that should have an obvious place to put a linear boundary.

Testing

We will use `make_blobs` to synthesize training data similar to what we see in figures 9.5 or 9.8. If we get similar training accuracy as sklearn we probably are doing something right, keeping in mind that there are many parameters we could be setting.

```
# synthetic blobs: two X features, two classes in Y
X, Y = make_blobs(n_samples=200,
                  n_features=2,
                  centers=2,
                  cluster_std=150,
                  random_state=2026,
                  center_box=(100, 900))
Y = np.where(Y == 0, -1, 1)

m1 = SV()
m1.fit(X, Y)
```

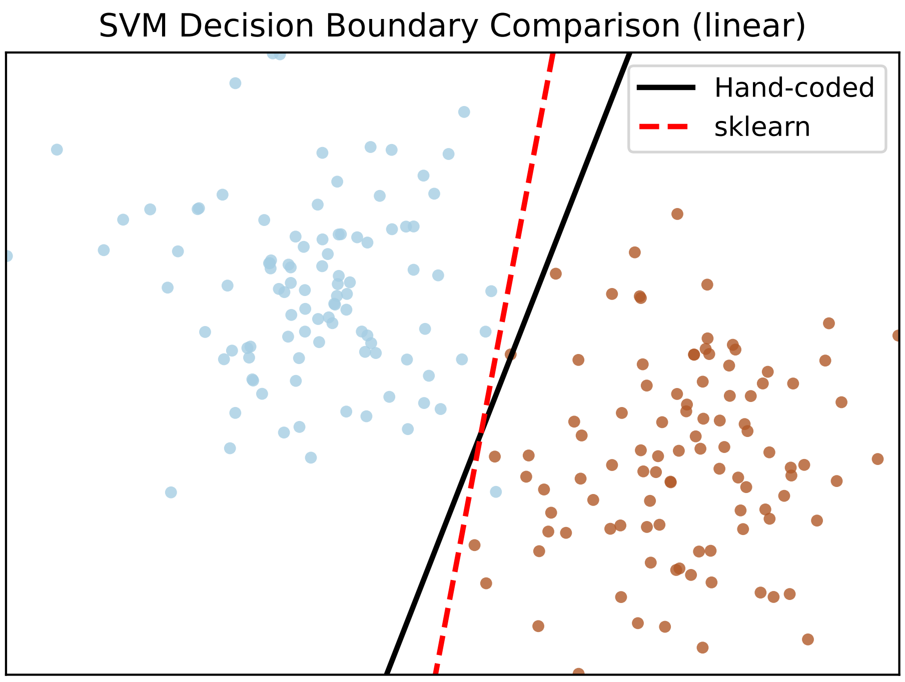
```
m2 = SVC(kernel='linear', C=1)
est = m2.fit(X, Y)

# similar training accuracy
print(f"Hand code {np.mean(m1.predict(X) == Y)}, sklearn {np.mean(m2.predict(X) == Y)}")
```

```
## Hand code 0.99, sklearn 0.995
```

Almost identical—so far, so good.

We kept it 2-dimensional, so we ought to be able to quickly tell if the class is working by plotting the boundary from the hand-coded class against sklearn.

```
viz_blob_boundaries(X, Y, m1, m2)
```



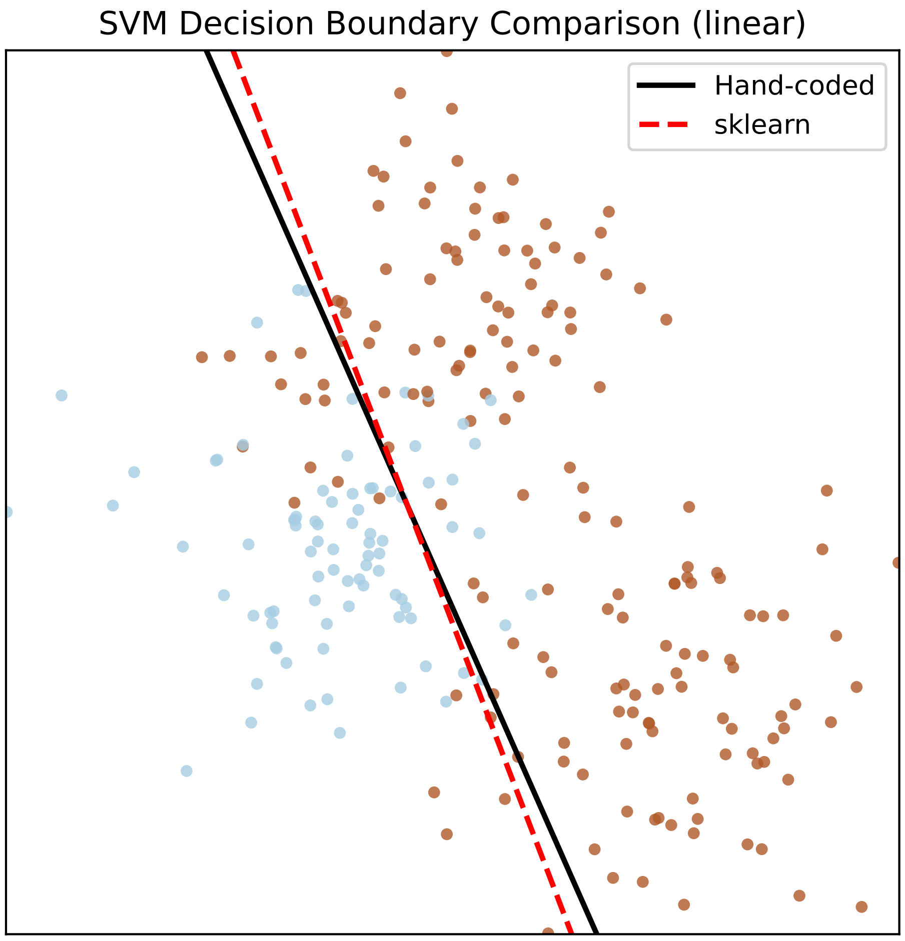
Both lines are in reasonable places. This one is kind of easy, we ought to try something harder to put both codes to work.

```python
# have them overlap more with more centers
X, Y = make_blobs(n_samples=250,
                  n_features=2,
                  centers=3,
                  cluster_std=190,
                  random_state=2026,
                  center_box=(100, 900))
Y = np.where(Y == 0, -1, 1)

m1 = SV()
m1.fit(X, Y)

m2 = SVC(kernel='linear', C=1)
est = m2.fit(X, Y)

# similar training accuracy
print(f"Hand code {np.mean(m1.predict(X) == Y)}, sklearn {np.mean(m2.predict(X) == Y)}")
```

```
## Hand code 0.872, sklearn 0.872
```

Identical training accuracy in this case. How's it look?

```python
viz_blob_boundaries(X, Y, m1, m2)
```



Looks very similar.

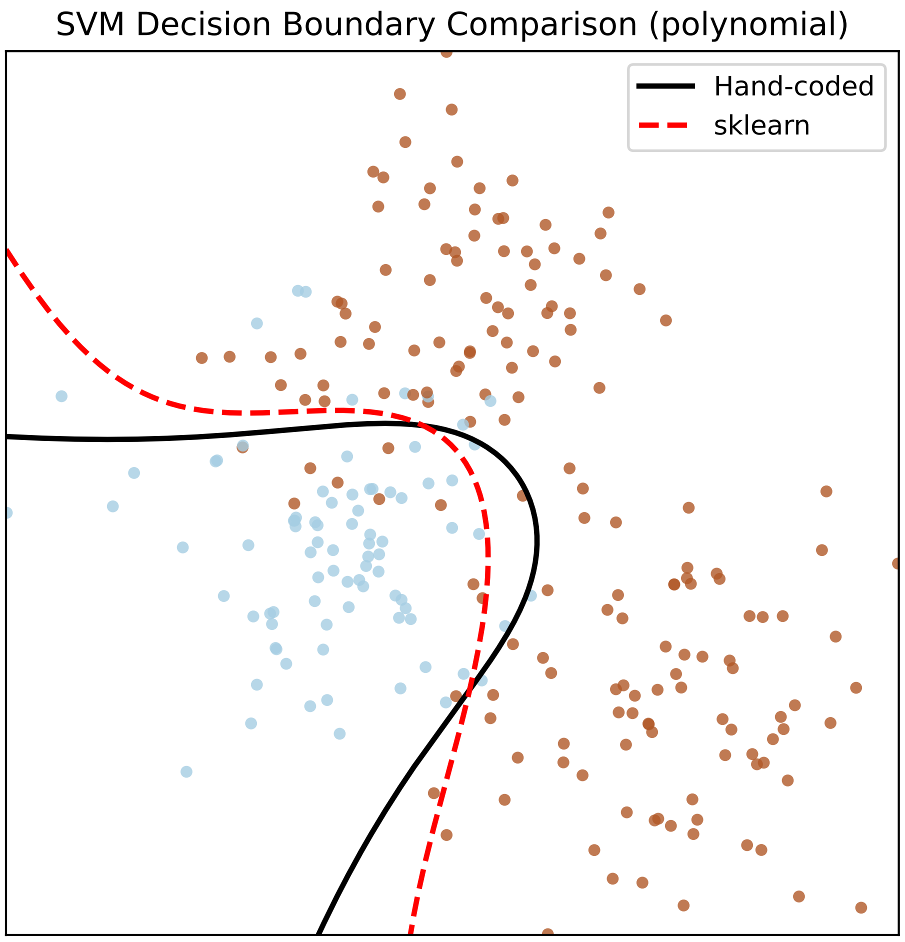
This also looks like a better scenario to put a non-linear kernel to the test. We ought to be able to beat the 0.872 training accuracy with something non-linear. Let's try a polynomial kernel.

```
# polynomial fit with a cubic, dial the epsilon back a bit
m1 = SV(kernel_name='polynomial', eps=1e-7)
m1.fit(X, Y)

m2 = SVC(kernel='poly', C=1, degree=3)
est = m2.fit(X, Y)

# similar training accuracy
print(f"Hand code {np.mean(m1.predict(X) == Y)}, sklearn {np.mean(m2.predict(X) == Y)}")
```

```
## Hand code 0.912, sklearn 0.908
```

As expected, we do indeed see a better fit using a polynomial kernel. Training accuracy is once again similar between the methods. Let's see the boundaries.

```
viz_blob_boundaries(X, Y, m1, m2)
```



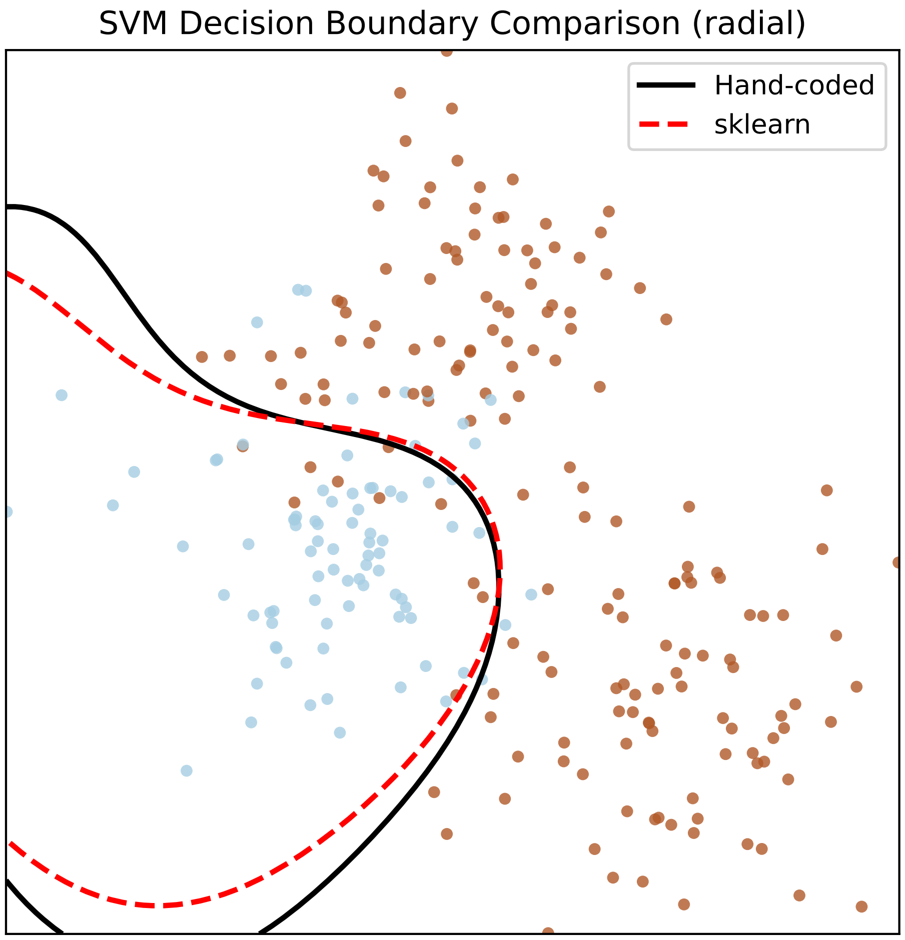
Looking good. Now let's try radial.

```
# radial kernel?
m1 = SV(kernel_name='radial')
m1.fit(X, Y)

m2 = SVC(kernel='rbf')
est = m2.fit(X, Y)

# identical training accuracy
print(f"Hand code {np.mean(m1.predict(X) == Y)}, sklearn {np.mean(m2.predict(X) == Y)}")
```

```
## Hand code 0.908, sklearn 0.908
```

Identical training accuracy. Now let's see the boundaries.

```
viz_blob_boundaries(X, Y, m1, m2)
```



Once again very similar.

Contours and `C`

Let's see if the `C` parameter is working as intended. Here it will be helpful to display more contour lines. Let's wrap it in a function for ease-of-use.

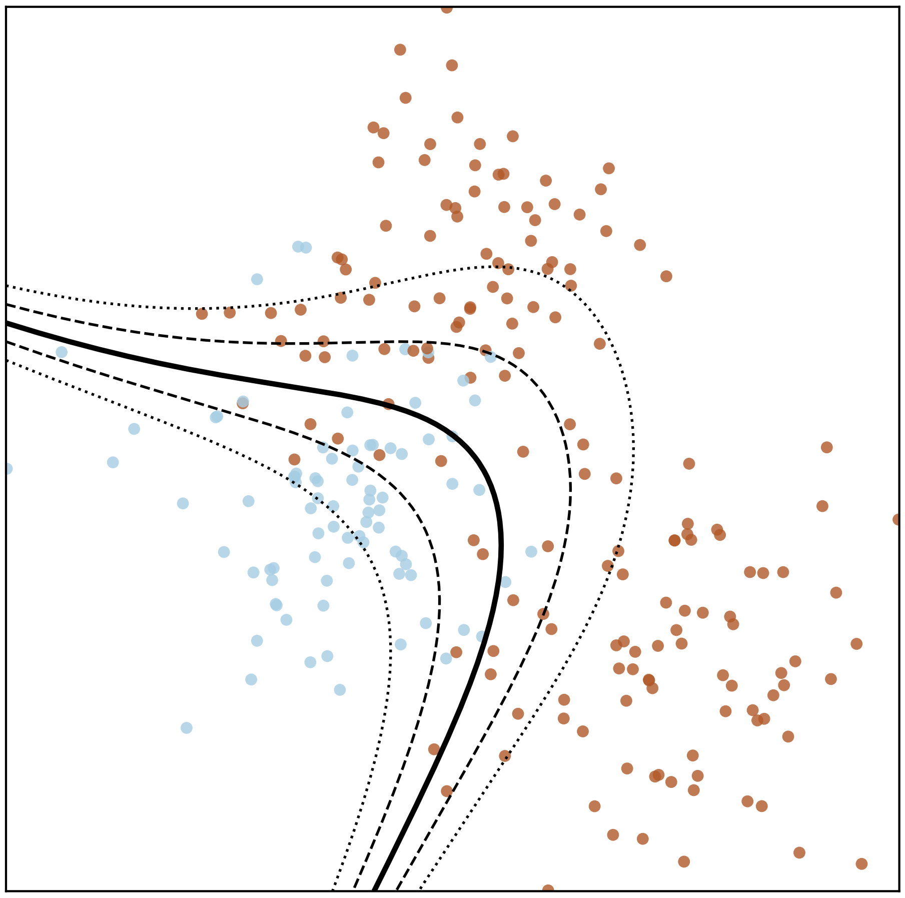
```python
def viz_blob_contours(X: np.ndarray,
                      Y: np.ndarray,
                      model) -> None:
    """Contour plot to compare hand-code and sklearn versions"""
    # training accuracy
    acc = np.mean(model.predict(X) == Y)

    # mesh grid in original feature space
    xx, yy = np.meshgrid(
        np.linspace(X[:, 0].min() - 1.5, X[:, 0].max() + 1.5, 300),
        np.linspace(X[:, 1].min() - 1.5, X[:, 1].max() + 1.5, 300)
    )
    grid = np.c_[xx.ravel(), yy.ravel()]

    # decision values
    Z = model.decision_function(grid).reshape(xx.shape)

    fig, ax = plt.subplots(figsize=(6, 6))

    # decision boundaries
    ax.contour(xx, yy, Z, levels=[-2, -1, 0, 1, 2],
               colors='k', linewidths=[1, 1, 2, 1, 1],
               linestyles=[':', '--', '-', '--', ':'])

    # data
    ax.scatter(X[:, 0], X[:, 1], c=Y, cmap='Paired',
               s=12, alpha=0.8)

    ax.set_aspect('equal', adjustable='box')
    ax.set_title(f"SVM Decision Boundary ({model.kernel_name}, C={model.C}, Acc={acc})")
    ax.set_xticks([])
    ax.set_yticks([])

    plt.show()
```

If `C` is working, then using a small `C` should give us a simpler boundary with wider margins, whereas a larger `C` should be less smooth but with narrower margins.

First, let’s use a small `C` value of 0.1—we should see a simple boundary.

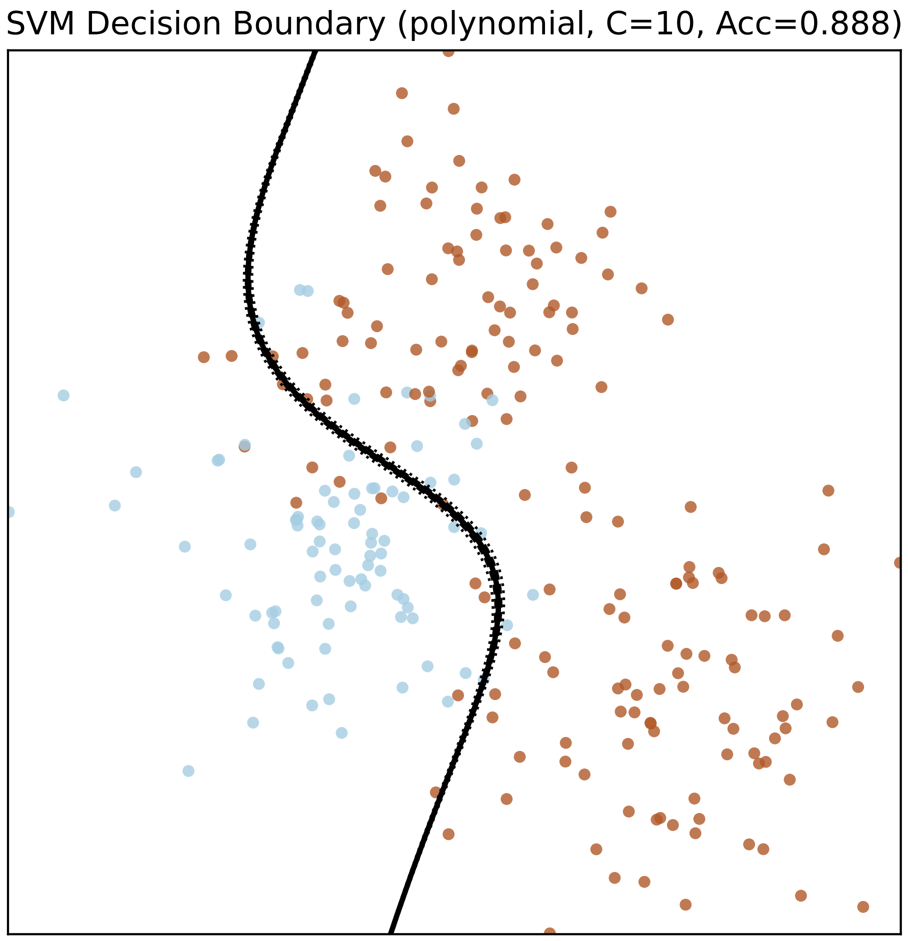
```
m1 = SV(kernel_name='polynomial', eps=1e-7, C=0.1)
m1.fit(X, Y)

viz_blob_contours(X, Y, m1)
```

SVM Decision Boundary (polynomial, C=0.1, Acc=0.912)



It does indeed look simple. Now let's try `C=10`.

```
m1 = SV(kernel_name='polynomial', eps=1e-7, C=10)
m1.fit(X, Y)

viz_blob_contours(X, Y, m1)
```

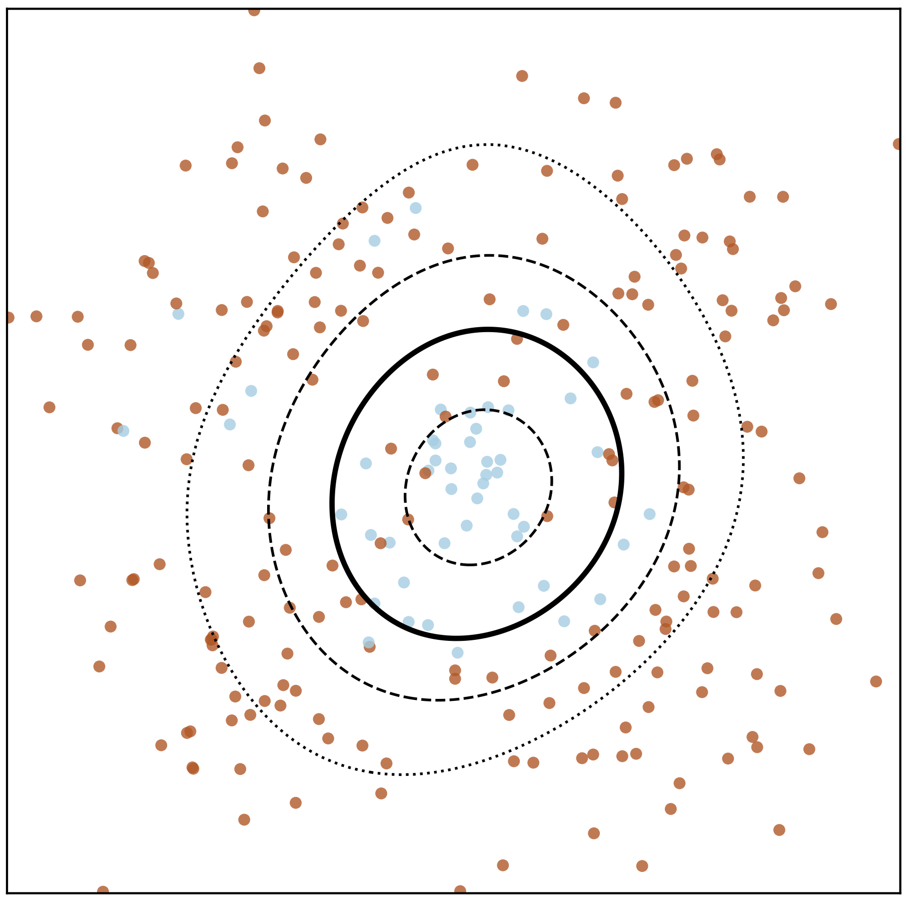


Now we have a more complicated boundary with much narrower margins. This is a good sign the `C` parameter is working as intended.

How will it look for a radial kernel? Let's craft an example with one central blob and give it a try.

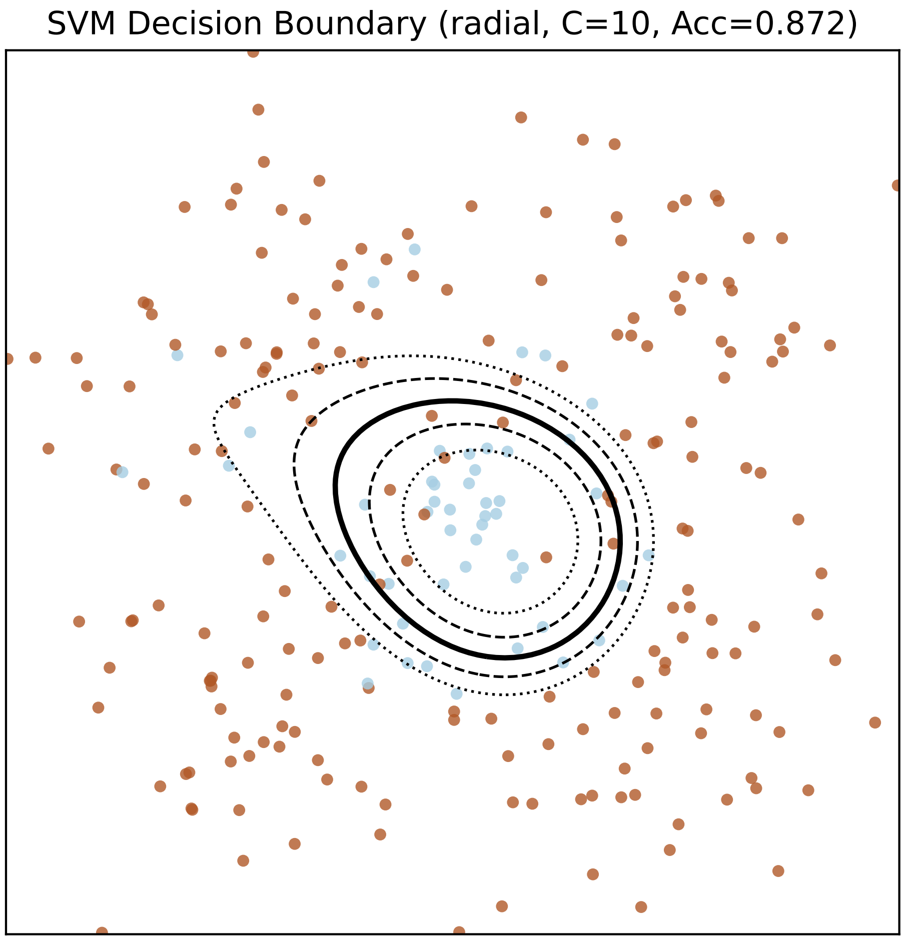
```python
# one central blob, four satellite blobs
X, Y = make_blobs(n_samples=250,
                  centers=[(250,250), (50, 50), (450, 450), (450, 50), (50, 450)],
                  cluster_std=100,
                  random_state=2026)
Y = np.where(Y == 0, -1, 1)

m1 = SV(kernel_name='radial', C=1)
m1.fit(X, Y)
viz_blob_contours(X, Y, m1)
```

SVM Decision Boundary (radial, C=1, Acc=0.892)



Looks reasonable. Will margins tighten at `C=10` as they did in the previous setup?

```
m1 = SV(kernel_name='radial', C=10)
m1.fit(X, Y)
viz_blob_contours(X, Y, m1)
```

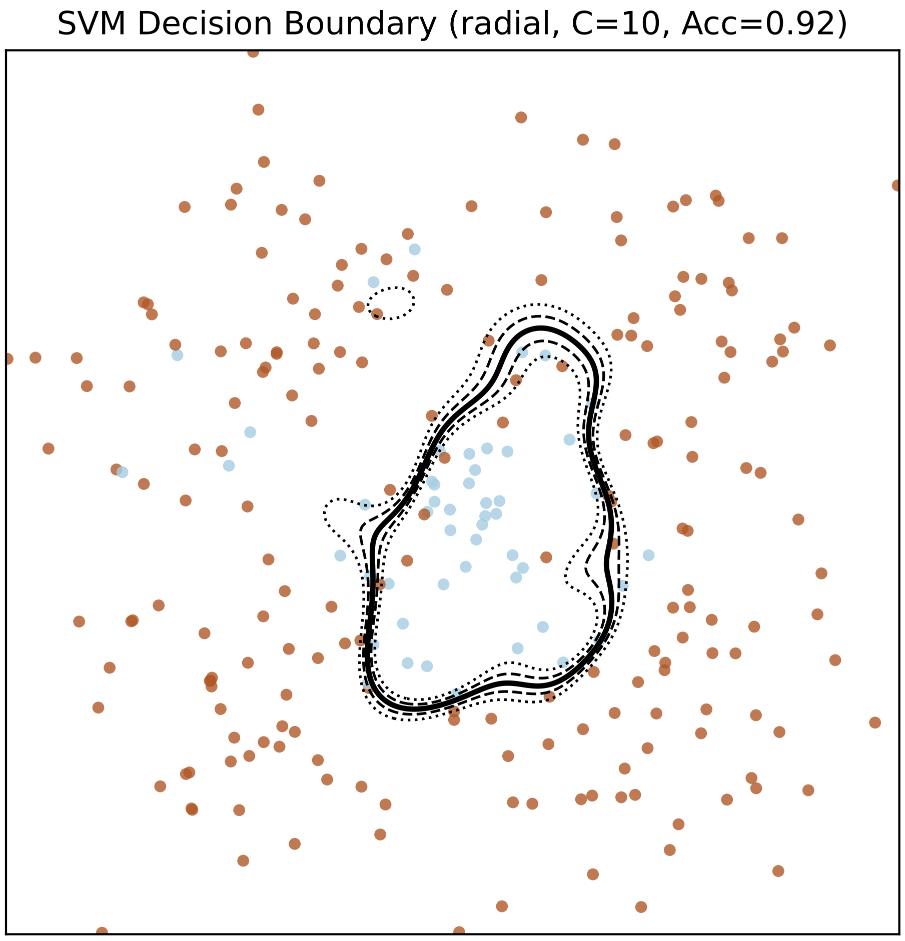


Success.

Radial Kernel and γ

One more parameter to test. Our γ (`gamma`) parameter, if it's working correctly, ought to let us magnify the squared errors so that local errors matter more and we fit to a more complicated boundary. Let's give it a try by increasing `gamma` from the default 0.5 up to 10.

```
m1 = SV(kernel_name='radial', C=10, gamma=10)
m1.fit(X, Y)
viz_blob_contours(X, Y, m1)
```



The shape is much more irregular, exactly as it required. It's likely that `gamma` is also working properly.

The more complicated boundary made it possible to boost training accuracy, though of course we're just overfitting to how this specific blob turned out.

A check of `fit_log` (not printed here) shows that none of these fits ever exercised the part of the code that is supposed to work on non-positive η, so we don't have any hints one way or the other that it was implemented correctly. With that caveat, we've seen that results seem to match expectations.

Appendix: Lagrange Multipliers

Say we're working with $\mathbf{x} = (x_1, x_2, \dots, x_D)$ and have some function $f(\mathbf{x})$ we want to maximize subject to some constraint $g(\mathbf{x}) = 0$. Our constraint $g(\mathbf{x}) = 0$ is a $(D-1)$-dimensional surface in $\mathbf{x}$-space, like how $\mathbf{w} \cdot \mathbf{x} + b = 0$ defines a boundary subspace for SVMs. For example, if $D = 3$, then $f(\mathbf{x})$ is a surface and $g(\mathbf{x}) = 0$ is a curve on that surface.

Suppose f and g are differentiable in the regions of interest. In that case, we'll have a gradient vectors, which (assuming they are not zero) will by definition be perpendicular to the surfaces. Our gradient vector $\nabla g(\mathbf{x})$

will be normal to the constraint surface $g(\mathbf{x}) = 0$, and our maximized $f(\mathbf{x})$ along $g(\mathbf{x}) = 0$ will likewise have $\nabla f(\mathbf{x})$ orthogonal to the constraint surface (otherwise, we could have increased $f(\mathbf{x})$ a little by moving along the constraint surface).

Since ∇f and ∇g can only differ by magnitude (including the possibility of being in opposite direction), there must exist a $\lambda \neq 0$ such that $\nabla f + \lambda \nabla g = 0$.

So the idea is to define a Lagrangian function $\mathcal{L}(\mathbf{x}, \lambda) = f(\mathbf{x}) + \lambda g(\mathbf{x})$ that captures both f and g then solve for the partial derivatives equal to zero.

Here's an example taken from appendix E of *Pattern Recognition and Machine Learning* by C. M. Bishop to illustrate. Let's say:

- we're working in $\mathbf{x} = (x_1, x_2)$
- we want to maximize $f(\mathbf{x}) = 1 - x_1^2 - x_2^2$
- subject to constraint $g(\mathbf{x}) = x_1 + x_2 - 1 = 0$

Then:

$$\begin{aligned}
\mathcal{L}(\mathbf{x}, \lambda) &= f(\mathbf{x}) + \lambda g(\mathbf{x}) && \text{definition} \\
&= 1 - x_1^2 - x_2^2 + \lambda g(\mathbf{x}) && \text{plug in } f \\
&= 1 - x_1^2 - x_2^2 + \lambda(x_1 + x_2 - 1) && \text{plug in } g \\
&= 1 - x_1^2 - x_2^2 + \lambda x_1 + \lambda x_2 - \lambda
\end{aligned}$$

Then our first partial is:

$$\begin{aligned}
\mathcal{L}_{x_1} &= \frac{\partial \mathcal{L}}{\partial x_1}[1 - x_1^2 - x_2^2 + \lambda x_1 + \lambda x_2 - \lambda] \\
&= \frac{\partial \mathcal{L}}{\partial x_1}[1] - \frac{\partial \mathcal{L}}{\partial x_1}[x_1^2] - \frac{\partial \mathcal{L}}{\partial x_1}[x_2^2] + \frac{\partial \mathcal{L}}{\partial x_1}[\lambda x_1] + \frac{\partial \mathcal{L}}{\partial x_1}[\lambda x_2] - \frac{\partial \mathcal{L}}{\partial x_1}[\lambda] && \text{differentiation is linear} \\
&= 0 - \frac{\partial \mathcal{L}}{\partial x_1}[x_1^2] - 0 + \frac{\partial \mathcal{L}}{\partial x_1}[\lambda x_1] + 0 - 0 && \text{derivatives of constant terms are each 0} \\
&= 0 - 2x_1 - 0 + \frac{\partial \mathcal{L}}{\partial x_1}[\lambda x_1] + 0 - 0 && \text{power rule} \\
&= 0 - 2x_1 - 0 + \lambda + 0 - 0 && \text{power rule} \\
&= \lambda - 2x_1
\end{aligned}$$

Our next partial:

$$\begin{aligned}
\mathcal{L}_{x_2} &= \frac{\partial \mathcal{L}}{\partial x_1}[1 - x_1^2 - x_2^2 + \lambda x_1 + \lambda x_2 - \lambda] \\
&= 0 - 0 - 2x_2 + 0 + \lambda - 0 && \text{linearity, constants, power rule} \\
&= \lambda - 2x_2
\end{aligned}$$

Our last partial:

$$\begin{aligned}
\mathcal{L}_{\lambda} &= \frac{\partial \mathcal{L}}{\partial \lambda}[1 - x_1^2 - x_2^2 + \lambda x_1 + \lambda x_2 - \lambda] \\
&= 0 - 0 - 0 + x_1 + x_2 - 1 && \text{linearity, constants, power rule} \\
&= x_1 + x_2 - 1
\end{aligned}$$

This approach gives us $D + 1$ conditions, and we were working in $D = 2$, so we end up with 3 conditions.

We can code that up as a an augmented matrix and solve.

$$\begin{array}{ccc|c} x_1 & x_2 & \lambda & \\ 1 & 1 & 0 & 1 \\ -2 & 0 & 1 & 0 \\ 0 & -2 & 1 & 0 \end{array} \longrightarrow \left[\begin{array}{ccc|c} 1 & 1 & 0 & 1 \\ 0 & 2 & 1 & 2 \\ 0 & -2 & 1 & 0 \end{array}\right] \qquad r_2 + 2r_1 \to r_2$$

$$\longrightarrow \left[\begin{array}{ccc|c} 1 & 1 & 0 & 1 \\ 0 & 2 & 1 & 2 \\ 0 & 0 & 2 & 2 \end{array}\right] \qquad r_3 + r_2 \to r_3$$

$$\longrightarrow \left[\begin{array}{ccc|c} 1 & 1 & 0 & 1 \\ 0 & 2 & 1 & 2 \\ 0 & 0 & 1 & 1 \end{array}\right] \qquad \text{mult } r_3 \text{ by } \tfrac{1}{2}$$

$$\longrightarrow \left[\begin{array}{ccc|c} 1 & 1 & 0 & 1 \\ 0 & 2 & 0 & 1 \\ 0 & 0 & 1 & 1 \end{array}\right] \qquad r_2 - r_3 \to r_2$$

$$\longrightarrow \left[\begin{array}{ccc|c} 1 & 1 & 0 & 1 \\ 0 & 1 & 0 & \frac{1}{2} \\ 0 & 0 & 1 & 1 \end{array}\right] \qquad \text{mult } r_2 \text{ by } \tfrac{1}{2}$$

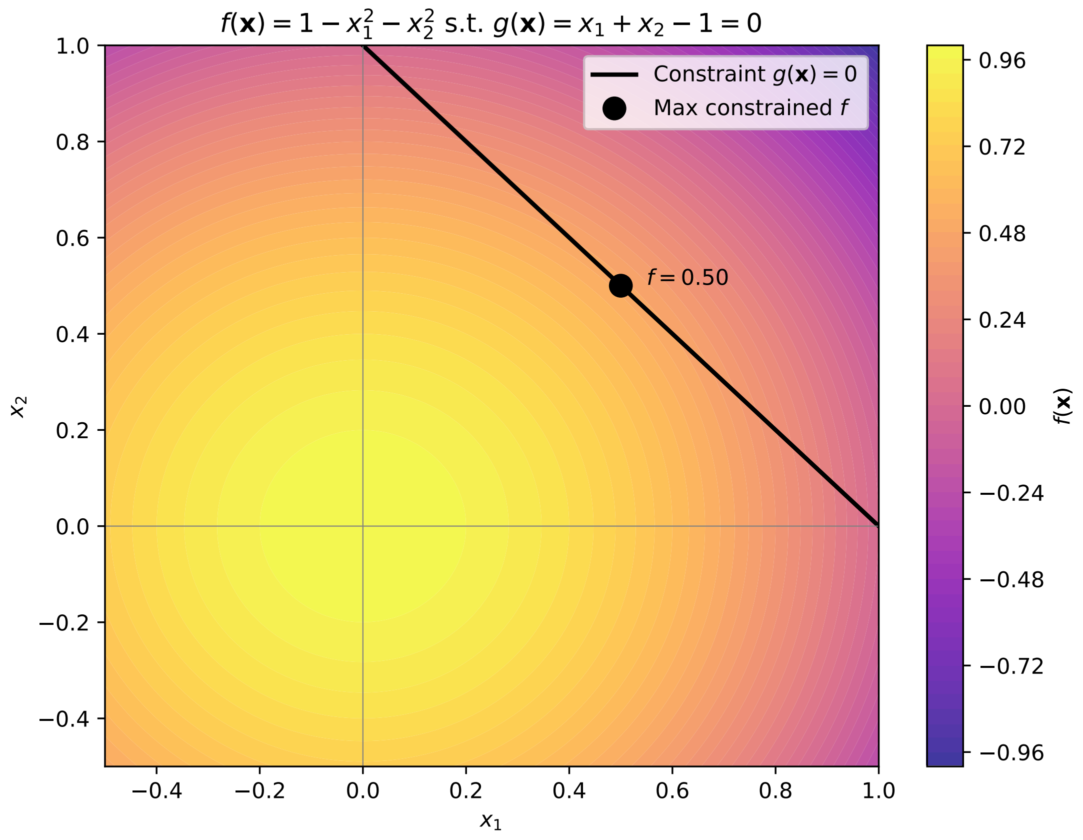
$$\longrightarrow \left[\begin{array}{ccc|c} 1 & 0 & 0 & \frac{1}{2} \\ 0 & 1 & 0 & \frac{1}{2} \\ 0 & 0 & 1 & 1 \end{array}\right] \qquad r_1 - r_2 \to r_1$$

Let's see how that looks with a contour plot.

```python
# define a function to illustrate
def f(x1, x2):
    return 1 - x1**2 - x2**2

# create a meshgrid for x and plot f(x) as a contour with colors
x1 = np.linspace(-0.5, 1, 100)
x2 = np.linspace(-0.5, 1, 100)
X1, X2 = np.meshgrid(x1, x2)
Z = f(X1, X2)
fig, ax = plt.subplots(figsize=(8, 6)) # a little wider for colorbar on left
contour = ax.contourf(X1, X2, Z, cmap='plasma', levels=50, alpha=0.8)

# add a colorbar so we recognize the slope directions
fig.colorbar(contour, ax=ax, label='$f(\mathbf{x})$')
```

```
## <matplotlib.colorbar.Colorbar object at 0x000002828F9EA860>
```

```python
# add the constraint g(x) = 0  →  x2 = 1 - x1
x1_constraint = np.linspace(-1, 1, 100)
x2_constraint = 1 - x1_constraint
ax.plot(
    x1_constraint,
    x2_constraint,
    color='black',
    linewidth=2,
    label='Constraint $g(\mathbf{x}) = 0$'
)

# call out the max value at (1/2, 1/2)
x1_point = 0.5
x2_point = 0.5
z_point = f(x1_point, x2_point)
ax.plot(
    x1_point,
    x2_point,
    'o',
    color='black',
    markersize=10,
    label='Max constrained $f$'
)
ax.text(
    x1_point + 0.05,
    x2_point,
    f"$f = {z_point:.2f}$",
    color='black',
    fontsize=10
)

# label axes, put equations in the title
ax.set_xlim(-0.5, 1.0);
ax.set_ylim(-0.5, 1.0);
ax.set_xlabel('$x_1$')
ax.set_ylabel('$x_2$')
ax.set_title(
    '$f(\mathbf{x}) = 1 - x_1^2 - x_2^2$ '
    's.t. $g(\mathbf{x}) = x_1 + x_2 - 1 = 0$'
)

# legend will show the constraint and the point of interest at 1/2, 1/2
ax.legend()

# lightly call attention to the 0 lines (which intersect at the unconstrained max f)
ax.axhline(0, color='grey', lw=0.5)
ax.axvline(0, color='grey', lw=0.5)

plt.show()
```



This was a simple case, whereas often we would have multiple critical points and want to check to see which was the maximum. Note that we can also have more than one equality constraint. For instance, we could have constraints $g(\mathbf{x}) = 0$ and $h(\mathbf{x}) = 0$ in which case we're solving for $\nabla f(\mathbf{x}) = \lambda \nabla g(\mathbf{x}) + \mu \nabla h(\mathbf{x})$ where μ is our second Lagrange multiplier.

Where the Karush-Kuhn-Tucker (KKT) conditions enter into the picture is when we generalize for inequality constraints $g(\mathbf{x}) \geq 0$.