Using Kernels

Michael Jayasuriya
University of California, Berkeley
mjayasur@berkeley.edu

Anthony Patitucci
University of California, Berkeley
apatitucci1@berkeley.edu

Kevin Merino
University of California, Berkeley
kmerino@berkeley.edu

Rachel Wang
University of California, Berkeley
rachel.pj.wang@berkeley.edu

Sam Yuen
University of California, Berkeley
yuens@berkeley.edu

1 Introduction

Kernels are an extremely useful concept that is ubiquitous in machine learning. We will take a look at kernels in multiple different scenarios throughout the notes and the homework to get you comfortable with both the theory and application of the concept.

1.1 Motivations

Let's take a look at Figure 1. Let's imagine that we are trying to place a line in the plane to divide up the points by color (class). As you probably remember from the SVM unit, this data is not **linearly separable**, and a single line cannot divide up the points properly. Each data point has two features, x_1 and x_2, and a label y. This space is not "rich" enough to give us data that is separable. Let's try to fix this issue. Suppose that you applied the feature map $\phi(\vec{x})$ to the data, where ϕ is defined as follows for a datapoint $\vec{x}$

$$\phi(\vec{x}) = [x_1, x_2, x_1^2 + x_2^2] \tag{1}$$

Let's visualize this data and see how our feature space has changed in Figure 2. In this richer space, our data is linearly separable! This is called the **lifting trick**, and is very advantageous in many situations. However, as we lift our data into higher and higher dimensions, instead of just two to three, computing the lifted features becomes very computationally expensive and becomes infeasible. **Kernels** present a solution to that, and we'll show you why.

2 Ridge Regression

Earlier in this course, you looked at Ridge Regression. Given feature matrix $X \in \mathbb{R}^{n \times d}$ and vector $y \in \mathbb{R}^{n \times 1}$, to find the weights w such that $Xw = y$ we find the following for solution for w.

$$w^* = (X^T X + \lambda I)^{-1} X^T y \tag{2}$$

As we just discussed, sometimes it can be advantageous to "lift" our features to a higher dimension d' such that instead of the matrix X we now use $\Phi \in \mathbb{R}^{n \times d'}$ where $d' > d$. Therefore we find the same solution as above in this "lifted" feature space but now with the Φ replacing the X.

$$w^* = (\Phi^T \Phi + \lambda I)^{-1} \Phi^T y \tag{3}$$

Figure 1: Data without lifting.

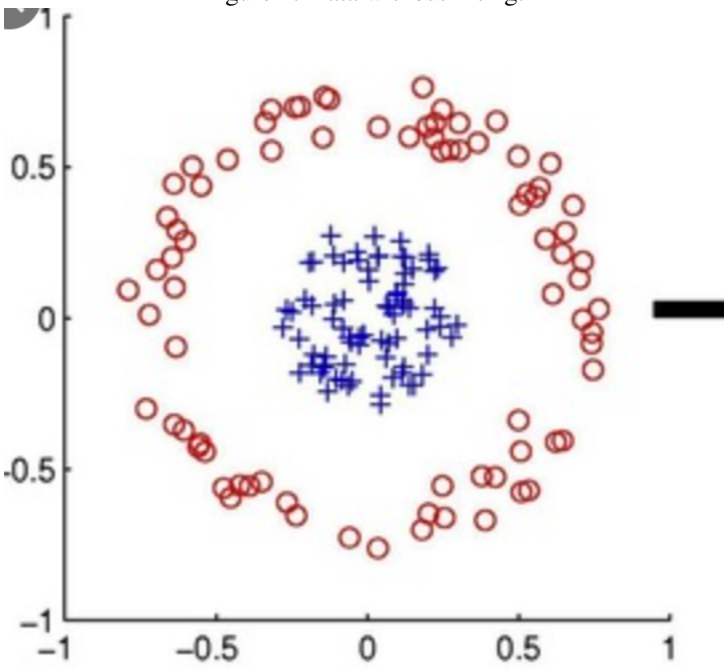
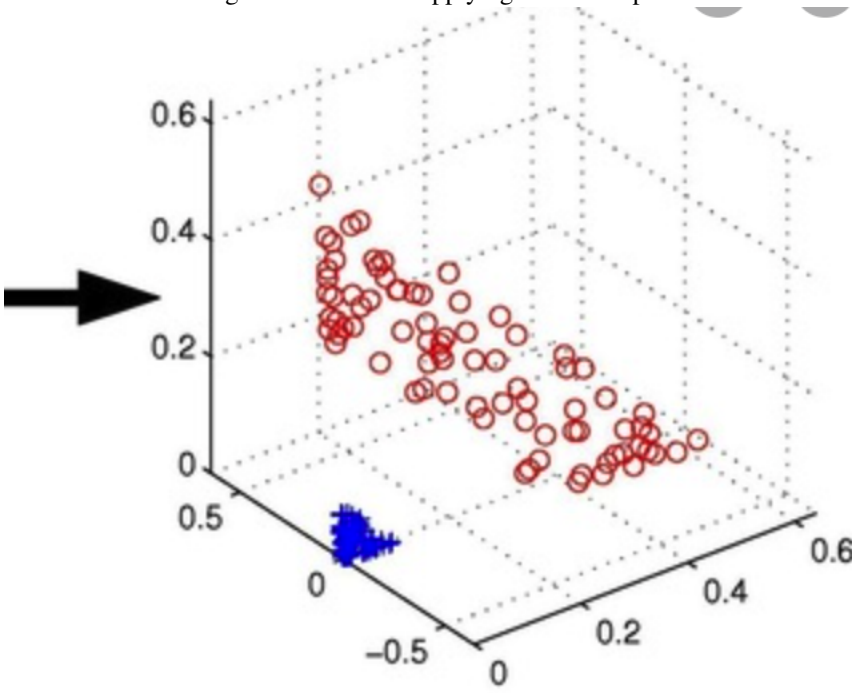


Figure 2: Data after applying feature map.



To compute this solution for ridge regression, it takes $O(d'^2 n)$ to compute the product $\Phi^T \Phi$, and another $O(d'^3)$ to invert. As we lift our feature space to a higher d' it starts to become impractical to compute this solution. Well, suppose we could express our Ridge Regression solution as this:

$$w^* = \Phi^T (\Phi \Phi^T + \lambda I)^{-1} y \tag{4}$$

Then, it would take $O(n^3)$ time to invert as our matrix $\Phi \Phi^T$ would be an $n \times n$ matrix. That would be awesome, wouldn't it? We could lift our features to any dimension without worrying about the time it would take to actually compute our solution. Luckily this is not a pipe dream! Let's explore how we got from the ridge regression solution you're familiar with to this alternate formulation.

2.1 Alternate Ridge Regression Formula Derivation

Let's assume that we are seeking a solution to the equation

$$Xw = y \tag{5}$$

Starting from the ridge formula and going forward, we can see the following.

$$\hat{w} = (X^T X + \lambda I)^{-1} X^T y$$

$$(X^T X + \lambda I)\hat{w} = X^T y$$

$$X^T X \hat{w} + \lambda \hat{w} = X^T y$$

$$\lambda \hat{w} = X^T y - X^T X \hat{w}$$

$$\hat{w} = \frac{X^T y - X^T X \hat{w}}{\lambda}$$

$$\hat{w} = X^T \frac{y - X\hat{w}}{\lambda}$$

First note that $\frac{y - X\hat{w}}{\lambda}$ is a n element vector. We can see from here that $\hat{w}$ is equal to X^T multiplied by some n element vector, and therefore $\hat{w}$ is some linear combination of the datapoints (if you're a little lost here check your notes from EE16A: https://eecs16a.org/lecture/Note3.pdf). We can write w as a linear combination of the datapoints as

$$w = X^T a$$

where a is a vector with n elements. From there, we can see that

$$a = \frac{y - X\hat{w}}{\lambda}$$

$$\lambda a = y - X\hat{w} = y - XX^T a$$

$$y = \lambda a + XX^T a$$

$$y = (\lambda I + XX^T)a$$

$$a = (\lambda I + XX^T)^{-1} y$$

or, for a form more familiar

$$a = (XX^T + \lambda I)^{-1} y$$

From here because we know that $w = X^T a$, we can left multiply both sides by X^T to get our solution.

$$w = X^T a = X^T (XX^T + \lambda I)^{-1} y$$

This is exactly what we wanted! A solution for ridge regression that allows us to lift our features to any dimension without worrying about the time complexity of the inversion.

2.2 Gram matrix and kernel functions

Specifically, in the earlier derivation we got XX^T into our solution. Let's take a closer look at this matrix.

$$X = \begin{bmatrix} — & x_1 & — \\ — & x_2 & — \\ & \vdots & \\ — & x_n & — \end{bmatrix}$$

$$X^T = \begin{bmatrix} | & | & & | \\ x_1 & x_2 & \dots & x_n \\ | & | & & | \end{bmatrix}$$

$$XX^T = \begin{bmatrix} — & x_1 & — \\ — & x_2 & — \\ & \vdots & \\ — & x_n & — \end{bmatrix} \begin{bmatrix} | & | & & | \\ x_1^T & x_2^T & \dots & x_n^T \\ | & | & & | \end{bmatrix} = \begin{bmatrix} \langle x_1, x_1 \rangle & \langle x_1, x_2 \rangle & \dots & \langle x_1, x_n \rangle \\ \langle x_1, x_n \rangle & \ddots & & \\ \vdots & & & \\ \langle x_n, x_1 \rangle & \dots & & \langle x_n, x_n \rangle \end{bmatrix}$$

Notice that the matrix XX^T contains the pairwise inner products of all the data points, or rows, in the data matrix X. We denote this matrix XX^T the **Gram matrix** and denote it as $K = XX^T$. Note that if we know the Gram matrix, we never really even need to know the actual underlying data matrix X. Instead, we can work entirely with the inner products between data points. Define the following function

$$k(x_i, x_j) = \langle x_i, x_j \rangle$$

We call $k(x_i, x_j)$ a **kernel function**. Therefore, we see that the Gram matrix is

$$K = \begin{bmatrix} k(x_1, x_1) & k(x_1, x_2) & \dots & k(x_1, x_n) \\ k(x_1, x_n) & \ddots & & \\ \vdots & & & \\ k(x_n, x_1) & \dots & & k(x_n, x_n) \end{bmatrix}$$

By defining a kernel function k, we can basically do all of our learning in this space instead of working directly with the datapoints. This is useful because we can work in a d dimensional space, where d can essentially be infinite. There are a wide array of kernel functions that are useful to map different decision boundaries.

2.3 Polynomial kernels and ridge regression

A good place to see the usefulness of kernelizing learning is with polynomials. We saw in the treatment of feature engineering that adding polynomials over the original variables induces a richer space in which we can learn non-linear models. Assume that we have a data matrix X with 2 features. We can lift a given data point we can lift it to a higher dimension by defining the following feature map.

$$\phi(\vec{x}) = \begin{bmatrix} x_1^2 & x_2^2 & \sqrt{2}x_1x_2 & \sqrt{2}x_1 & \sqrt{2}x_2 & 1 \end{bmatrix}$$

The polynomial kernel is the kernel approach to polynomial regression. The use of a polynomial kernel speeds up the learning process whenever the polynomial extends our data matrix from a tall matrix to a wide matrix, i.e more features than data points. Let's define the following kernel function for two d dimensional vectors.

$$k(\phi(\vec{x}), \phi(\vec{z})) = \langle \phi(\vec{x}), \phi(\vec{z}) \rangle = (\vec{x}^T \vec{z} + 1)^d \tag{6}$$

Why is this true?

$$k(\phi(\vec{x}), \phi(\vec{z})) = \langle \phi(\vec{x}), \phi(\vec{z}) \rangle$$

$$= \begin{bmatrix} x_1^2 & x_2^2 & \sqrt{2}x_1x_2 & \sqrt{2}x_1 & \sqrt{2}x_2 & 1 \end{bmatrix}^T \begin{bmatrix} z_1^2 & z_2^2 & \sqrt{2}z_1z_2 & \sqrt{2}z_1 & \sqrt{2}z_2 & 1 \end{bmatrix}$$

$$= x_1^2z_1^2 + x_2^2z_2^2 + 2x_1x_2z_1z_2 + 2x_1z_1 + 2x_2z_2 + 1$$

$$= (x_1^2 z_1^2 + 2x_1 z_1 x_2 z_2 + x_2^2 z_2^2) + 2x_1 z_1 + 2x_2 z_2 + 1$$
$$= (x_1 z_1 + x_2 z_2)^2 + 2(x_1 z_1 + x_2 z_2) + 1$$
$$= (\vec{x}^T \vec{z})^2 + 2\vec{x}^T \vec{z} + 1$$
$$= (\vec{x}^T \vec{z} + 1)^2$$

This trick allows us to express the inner product of two polynomial feature vectors with a kernel function. Instead of taking the inner product of the datapoints in the lifted feature space, we can work with the original data. By making our kernel function's complexity in terms of the original feature space and not the lifted feature space, we don't have to worry about what dimension we lift our data too.

3 Mercer's Theorem

At this point, you may be wondering what we can even use as a kernel function. There are obviously quite a few kernel functions, but what makes a kernel function use able and valid? In order for a kernel function to be valid, it must satisfy the following condition:

- The gram matrix is **positive semi definite** and symmetric

A symmetric matrix $K = K^T$ is positive semi definite if all of its eigenvalues are non-negative, i.e. $\lambda_i \geq 0$ for all eigenvalues λ_i of K.

If a function satisfies the condition, then it expresses an inner product of two vectors

$$k(\vec{x}, \vec{z}) = \langle \phi(\vec{x}), \phi(\vec{v}) \rangle \tag{7}$$

where phi is some mapping of the original features. As we've been stressing this whole time, we don't ever really need to compute the mapping of the original features $\phi(\cdot)$, we can just work with our kernel function.

4 Types of Kernels

There are many different kernel functions, all of which useful in different situations. Designing and choosing which kernel function to use is something of an art, and often involves quite a bit of domain knowledge. That being said, here are some extremely common kernel functions you'll likely see.

4.1 Polynomial kernel

The polynomial kernel is the kernel approach to polynomial regression. The use of a polynomial kernel speeds up the learning process whenever the polynomial extends our data matrix from a tall matrix to a wide matrix, i.e more features than data points.

$$k(\phi(\vec{x}), \phi(\vec{z})) = (\vec{x}^T \vec{z} + 1)^d \tag{8}$$

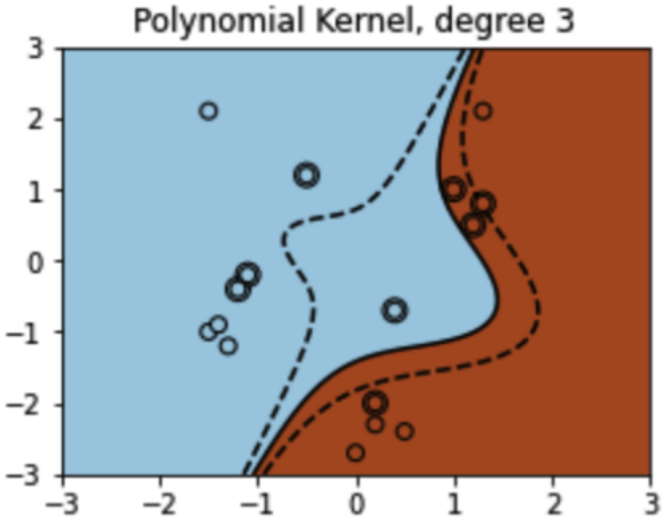
Similar to polynomial regression, the degree d can be tuned by validation techniques and represents the highest degree any product term will have. This kernel should be used whenever polynomial features are expressive enough to separate the data and our degree is larger than the number of data points ($n >= m$).

To use Sci-Kit SVM with a polynomial kernel we can use the kernel parameter set equal to 'poly', additionally, we can set the degree parameter equal to d. All that's needed to be modified is the svm.SVC function call:

```
# Fit a SVM on data (X, Y)
clf = svm.SVC(kernel='linear', degree=4)
clf.fit(X, Y)
```

You can see the result in Figure 5.

Figure 3: SVM with a polynomial kernel.



4.2 RBF kernel

The RBF kernel is a kernel that expresses a feature space with essentially infinite dimensions. The kernel function for the RBF kernel is defined as follows:

$$k(\vec{x}, \vec{z}) = exp(-\frac{||\vec{x} - \vec{z}||^2}{2\sigma}) \tag{9}$$

We can see that the function involves the euclidean distance squared and is divided by a hyperparameter σ. The formula is sometimes written with a different parameter γ called "bandwidth", where

$$\gamma = \frac{1}{2\sigma} \tag{10}$$

and therefore

$$k(\vec{x}, \vec{z}) = exp(-\gamma||\vec{x} - \vec{z}||^2) \tag{11}$$

Notice that as the distance between x and z go to infinity, the kernel function goes to zero. As the distance between x and z becomes small, the kernel function grows in value. Therefore we can see the RBF kernel function as a sort of measure of similarity between two vectors. This function is very useful to use with our data as it allows infinite dimension and a nonlinear decision boundary that is great when you don't have any prior information about your data. It is built into sci-kit learn, and can be done by setting kind to 'rbf' for the SVC classifier object. With the following code, we can get a decision boundary like the one in Figure 4.

```
clf = svm.SVC(kernel='rbf', gamma=2)
clf.fit(X, Y)
```

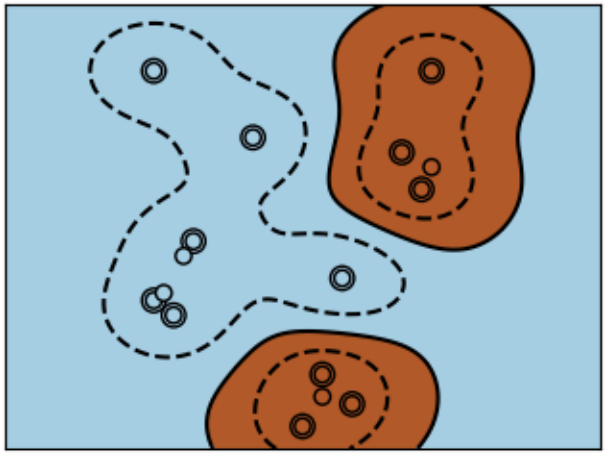
4.3 Exponential Kernel and Custom Kernels in sklearn

The Exponential Kernel is similar to the RBF kernel, but instead of using the L2 Norm squared it uses the L1 Norm.

$$k(\vec{x}, \vec{z}) = exp(-\gamma|\vec{x} - \vec{z}|_1) \tag{12}$$

where gamma is the same as in the RBF kernel. Notice that this kernel is not differentiable because the norm is an L1 norm. Because of this, the RBF kernel will work best for approximating smooth functions, whereas this kernel will perform better with non-differentiable decision boundaries like bends in a decision boundary.
sci-kit learn does not have the exponential kernel built in, so instead we get to implement a custom kernel function. Look to the following code for an example.

Figure 4: Decision boundary of RBF kernel



```
gamma = 0.2
def exponential_kernel(x,z):
    return np.exp(-np.linalg.norm(x - z, ord=1) / gamma)
clf = svm.SVC(kernel=exponential_kernel)
clf.fit(X, Y)
```

5 Applying kernels to Support Vector Machines

If you remember from the SVM unit, in order to maximize the margin for our separating hyperplane we maximized this objective function:

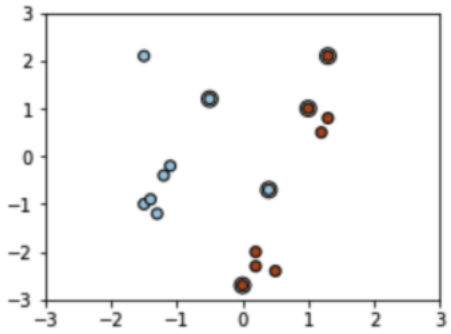
$$\alpha^T \mathbf{1} - \frac{1}{2}\alpha^T Q \alpha \tag{13}$$

Where $\mathbf{Q} = (\text{diag } \mathbf{y})\ XX^T\ (\text{diag } \mathbf{y})$. Immediately, we can see that the Gram matrix is present in the objective function, as $\mathbf{Q} = (\text{diag } \mathbf{y})\ \mathbf{K}\ (\text{diag } \mathbf{y})$. Therefore, we can see that given data we can construct a Gram matrix with a kernel function k and have our SVM find a max margin solution in the feature space of the kernel. Like in Ridge Regression, we can essentially lift our data to any dimension without worrying about time complexity and find decision boundaries that working with the raw features might not be able to find.

5.1 Visualizing SVM kernels

Recall the work of the Support Vector Machine in classification problems is to find a decision boundary. Such a boundary exists as a subset of the space in which our data lies and SVM uses two approaches when searching for this curve to separate our data. Ideally, we would find a hyper-plane which separates all data classes by sending each to one particular side of the hyper-plane. It's easy to see that this may not always be possible in the ambient space of our data as we saw previously with the circle decision boundary. We also saw in note 2 that the two approaches taken by SVM are hard-margin and soft-margin. The hard-margin approach uses the assumption of linearly separable data and seeks to maximize the space between the classes and the boundary (for uniqueness). The soft-margin approach is only different in that it allows for the miss-classification of a few data points. The soft-margin approach is specially useful with a non-linear decision boundary. In such a case, we may additionally benefit from featurizing our data and that may also benefit from kernelizing and using the kernel trick on the SVM. Using Sci-Kit we can visualize the decision boundary for our choice of kernel on our SVM with the following code. You can see the results in Figures 1, 2, and 3.

Figure 5: Support vectors along with data points.




```
fignum = 1

# Fit a SVM on data (X, Y)
clf = svm.SVC(kernel='linear')
clf.fit(X, Y)

# Plot the line, the points, and the nearest vectors to the plane
plt.figure(fignum, figsize=(4, 3))
plt.clf()

# Show scatter plot of Support Vectors
plt.scatter(clf.support_vectors_[:, 0], clf.support_vectors_[:, 1], s=80, \
                facecolors='none', zorder=10, edgecolors='k')

# Show scatter plot of Data Points
plt.scatter(X[:, 0], X[:, 1], c=Y, zorder=10, cmap=plt.cm.Paired, edgecolors='k')

x_min, x_max = -3, 3
y_min, y_max = -3, 3

# Sample the decision function
XX, YY = np.mgrid[x_min:x_max:200j, y_min:y_max:200j]
Z = clf.decision_function(np.c_[XX.ravel(), YY.ravel()])

# Put the result into a color plot
Z = Z.reshape(XX.shape)
plt.figure(fignum, figsize=(4, 3))


# Color Mesh Faces
plt.pcolormesh(XX, YY, Z > 0, cmap=plt.cm.Paired)

# Draw decision boundary and margins
plt.contour(XX, YY, Z, colors=['k', 'k', 'k'], linestyles=['--', '-', '--'],
                levels=[-.5, 0, .5])

plt.xlim(x_min, x_max)
plt.ylim(y_min, y_max)
```

Figure 6: Decision boundary (solid line) and margins (dashed), Colored Mesh Faces

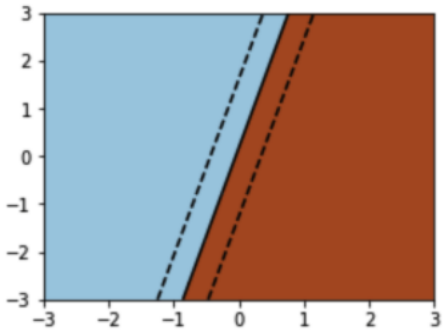
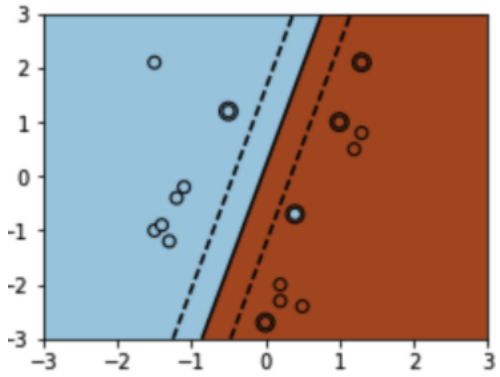


Figure 7: Soft-Margin SVM normally plotted with both plots overlaid.



References

[1] A Comprehensive Guide to Machine Learning, Nasiriany, S., Thomas, G., Wang, W., Yang, A., Listgarten, J.; Sahai, A. (2019). http://snasiriany.me/files/ml-book.pdf [2] Scikit-learn: Machine Learning in Python, Pedregosa et al., JMLR 12, pp. 2825-2830, 2011. [3] The Kernel Trick, Michael I. Jordan, Romain Thibaux (2004). https://people.eecs.berkeley.edu/ jordan/courses/281B-spring04/lectures/lec3.pdf