1 Introduction

We can build linear binary classifiers for linearly separable datasets. The problem is that often we don't have linearly separable datasets and so we either have utilize a different method such as a non-linear classifier or linearize the data. In the field of kernel methods we can achieve the latter utilizing the *kernel trick*. A *kernel function* is given by a fixed non-linear *feature space* mapping $\phi(\mathbf{x})$. The kernel function $k(\mathbf{x}, \mathbf{x}')$ is given by:

$$k(\mathbf{x}, \mathbf{x}') = \phi(\mathbf{x})^T \phi(\mathbf{x}') \tag{1.0.1}$$

We note that $\Phi = \phi(\mathbf{x})$ is actually a matrix called the *design matrix*. The concept of the kernel is built around the idea that we can substitute the kernel (the kernel trick) by replacing the input vector $\mathbf{x}$. In general, we aim to map our feature vector to a linear space for use in a linear classifier.

2 Dual Representation

The problem of classification often lies in the minimization of a fixed cost function which takes a weight vector $\mathbf{w}$ as its parameter. This means that we are fitting a function to a some particular points in hyper-dimensional space. Here we use the cost function $J(\mathbf{w})$ with *Tikhonov regularization*:

$$J(\mathbf{w}) = \frac{1}{2}\sum_{n=1}^{N}\{\mathbf{w}^T\phi(\mathbf{x}_n) - t_n\}^2 + \frac{\lambda}{2}\mathbf{w}^T\mathbf{w} \tag{2.0.1}$$

If we want to minimize this function we take its gradient with respect to and set it equal to zero. We then solve for $\mathbf{w}$. In symbols, this looks like solving the following equation:

$$\nabla J(\mathbf{w}) = 0 \tag{2.0.2}$$

The solution to this equation is:

$$\mathbf{w} = -\frac{1}{\lambda}\sum_{n=1}^{N}\{\mathbf{w}^T\phi(\mathbf{x}_n) - t_n\}\phi(\mathbf{x}_n) \tag{2.0.3}$$

If we set

$$a_n = -\frac{1}{\lambda}\{\mathbf{w}^T\phi(\mathbf{x}_n) - t_n\} \tag{2.0.4}$$

and substitute into our solution then it looks like:

$$\mathbf{w} = \sum_{n=1}^{N} a_n\phi(\mathbf{x}_n) = \Phi^T\mathbf{a} \tag{2.0.5}$$

This gives us the *dual representation* of the minimization problem. We can represent the cost function in terms of $\mathbf{a}$:

$$J(\mathbf{a}) = \frac{1}{2}\mathbf{a}^T\Phi\Phi^T\Phi\Phi^T\mathbf{a} - \mathbf{a}^T\Phi\Phi^T\mathbf{t} + \frac{1}{2}t^Tt + \frac{\lambda}{2}\mathbf{a}^T\Phi\Phi^T\mathbf{a} \tag{2.0.6}$$

We can define the *Gram matrix* as $\mathbf{K} = \Phi\Phi^T$ and substitute into equation (2.0.6):

$$J(\mathbf{a}) = \frac{1}{2}\mathbf{a}^T\mathbf{K}\mathbf{K}\mathbf{a} - \mathbf{a}^T\mathbf{K}\mathbf{t} + \frac{1}{2}\mathbf{t}^T\mathbf{t} + \frac{\lambda}{2}\mathbf{a}^T\mathbf{K}\mathbf{a} \tag{2.0.7}$$

Now as with equation (2.0.1) we can minimize it by setting the gradient of $J(\mathbf{a})$ to zero and solve for $\mathbf{a}$. The solution given by:

$$\mathbf{a} = (\mathbf{K} + \lambda\mathbf{I}_N)^{-1}\mathbf{t} \tag{2.0.8}$$

Finally, assuming $t_n \in \{-1, 1\}$, we predict the class of a new input $\mathbf{x}$ with:

$$\hat{y}(\mathbf{x}) = sgn(\mathbf{w}^T\phi(\mathbf{x}) + \frac{1}{2}) = sgn(k(\mathbf{x}_n, \mathbf{x})^T\mathbf{a} + \frac{1}{2}) \tag{2.0.9}$$

3 Example

We are tasked to classify the dataset plotted in Figure 1. We view the dataset as a 800×2 matrix which means there are 2 feature columns and 800 samples. This dataset is clearly not linearly separable so it is up to us to find an appropiate feature mapping. In this case we know the appropiate mapping because the dataset was generated using a function which generates circles. The code is:

```
# Generate concentric circles
X1, Y1 = make_circles(n_samples=800, noise=0.2, factor=0.2)
Y1[np.where(Y1 == 0)] = -1.  # change label from 0 to -1
```

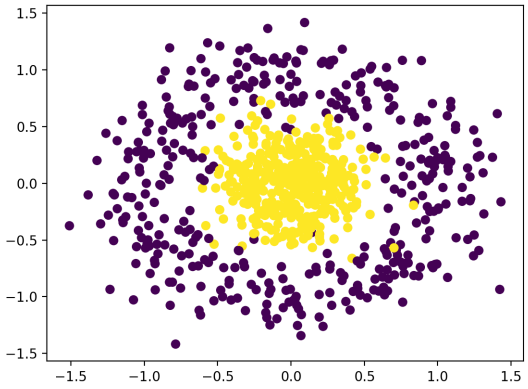

Figure 1: Dataset - Yellow dots have class label -1. Purple dots have class label 1

Thus, we suggest the mapping

$$\phi : (x1, x2) \to (x1, x2, x1^2 + x^2) \tag{3.0.1}$$

In code this looks like:

```
def phi(x):
    """
    The feature space mapping.
    """
    x1 = x[:, 0]
    x2 = x[:, 1]
    return np.array([x1, x2, np.power(x1, 2) + np.power(x2, 2)])
```

We also define the kernel function:

```
def k(x0, x1):
    """
    The kernel function.
    """
    return np.dot(phi(x0).T, phi(x1))
```

Next we set everything up for minimizing the function ($\lambda = 1$):

```
# Set up parameters for training
N = len(X1)  # number of samples
lamb = 1  # lambda
K = k(X1, X1)  # kernel matrix
Id = np.identity(N)  # identity matrix
t = Y1  # targets
```

Lastly, we minimize by inverting the matrix and taking the dot product:

```
# Minimize
a = np.linalg.inv(K + lamb * Id)
a = np.dot(a, t)
```

4 Results

We calculated the accuracy of the classifier by taking 400 random entries of the dataset and comparing them to their actual labels. The reader can check the code used in Appendix A. We got an accuracy of 89.5% for this particular dataset. We can see that the dataset of Figure 1 has overlapping classes. If our kernel is appropiate we should expect 100% accuracy on non-overlapping classes. To quickly test this hypothesis we give a non-overlapping dataset generated by the code:

```
# Generate concentric circles with little noise
X1, Y1 = make_circles(n_samples=800, noise=0.1, factor=0.2)
Y1[np.where(Y1 == 0)] = -1.  # change label from 0 to -1
```

Which we then plot and can see in Figure 2.

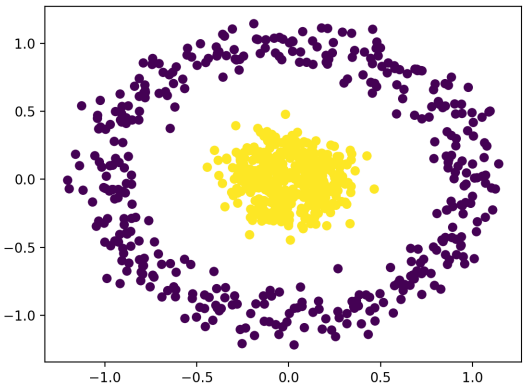

Figure 2: Non-overlapping Dataset - Yellow dots have class label -1. Purple dots have class label 1

When running the algorithm again we indeed get a 100% accuracy.

A Scoring Code

```
# Check accuracy
sample_count = 400
total = 0.
for i in range(sample_count):
    idx = np.random.randint(0, N)

    # Predict
    x = np.array([X1[idx]])
    y = np.dot(k(x, X1), a) + 0.5

    if np.sign(y) == t[idx]:
        total += 1.

acc = total / sample_count

print 'Accuracy: ', acc * 100, '%'
```