In [76]:
```
using MultivariateStats, CSV, DataFrames, KernelFunctions, LinearAlgebra, Printf
```

Introduction

In this Jupyter Notebook, we apply three different kernel methods in order to classify handwritten digits from the MNIST data set. To do this, we first apply principal component analysis to reduce the dimension of our training data to 154, preserving 95% of the variance. We then build classifiers via a kernel ridge regression with three different choices of kernels: linear, polynomial, and RBF (Gaussian). We do this for the pairs of digits $(1, 9)$, $(3, 8)$, $(1, 7)$, and $(5, 2)$. We find that the pairs $(3, 8)$ and $(5, 2)$ are the hardest to classify correctly and that in all cases, polynomial and RBF kernels when tuned well outperform the linear kernel by a large margin.

Let's start by reading the training and test data in, converting them to matrices, and separating the labels from the actual data.

In [2]:
```
t_data = Matrix(CSV.read("mnist_train.csv", DataFrame))
training_data = t_data[:,2:end]
training_labels = t_data[:,1];
```

In [3]:
```
test_data = Matrix(CSV.read("mnist_test.csv", DataFrame))
testing_data = test_data[:,2:end]
testing_labels = test_data[:,1];
```

Principal Component Analysis

We begin by using principal component analysis (PCA) to reduce the dimension of our input but preserve 95% of the variance. In doing this, we use the Julia package MultivariateStats.jl. This package is pretty black box, so we begin with a brief explanation.

PCA can be understood through either a covariance matrix decomposition or the SVD; however, since the dimension is smaller than the number of observations, MultivariateStats defaults to the former. The broad idea of this approach is that a covariance matrix of our data is constructed, and an eigendecomposition of this matrix is computed. We then select the eigenvectors in decreasing order of the size of their eigenvalues, stopping when at least 95% of the variance is preserved. We find that this requires 154 prinicpal components.

In [190…
```
M = fit(PCA,training_data';pratio=0.95);
@printf("Number of modes needed: %i\n",size(M,2))
```

```
Number of modes needed: 154
```

Now, we project both our training data and testing data onto the span of the selected eigenvectors. This is what we will use as our data input for the rest of our analysis.

In [34]:
```
Xte = predict(M, training_data')' |> Array;
Yte = predict(M, testing_data')' |> Array;
```

Kernel regression

For ease of implimentation, we focus solely on kernel ridge regression. The general framework for this is outlined in the course notes from lecture 8. Assuming $\mathcal{H}$ is an RKHS with kernel K, we look to solve

$$\min_{f\in\mathcal{H}} = \sum_{j=1}^{n} |f(x_j) - y_j|^2 + \frac{\lambda}{2}\|f\|^2.$$

The representer theorem allows us to write this as the problem

$$\min_{z\in\mathbb{R}^n} \|z - y\|^2 + \frac{\lambda}{2} z^T K(X,X)^{-1} z.$$

We can write the optimal solution in a slight different from than expressed in class as

$$z^* = K(X^*, X)(K(X,X) + \lambda I)^{-1} y,$$

which we impliment in the following function.

In what follows, we'll try three choices of the kernel K which we obtain from the KernelFunctions.jl package.

In [185…
```
function kernel_ridge_regression(k, X, y, Xstar, lambda)
    K = kernelmatrix(k, X)
    kstar = kernelmatrix(k, Xstar, X)
    return kstar * ((K + lambda * I) \ y)
end;
```

Now, we write functions which given an output, check what percentage of observations were classified correctly.

In [186…
```
classify(x) = x>0.5 ? 1. : -1.;
```

In [187…
```
function pct_classified(output,labels)
    class = classify.(output)
    incorrect = sum(abs.(classify.(output)-labels))/2
    @printf("percent classified correctly: %f%%\n",(length(labels)-incorrect)/length(labe
end;
```

Let's collect the data corresponding to the digits 1 and 9 and attempt to classify with kernel ridge regression with linear, polynomial, and RBF kernels.

In [53]:
```
index1 = findall(training_labels.==1)
index9 = findall(training_labels.==9)
X₁ = Xte[[index1; index9],:]
y₁ = [ones(size(index1)); -ones(size(index9))];
index1t = findall(testing_labels.==1)
index9t = findall(testing_labels.==9)
Xstar₁ = Yte[[index1t; index9t],:];
ystar₁ = [ones(size(index1t)); -ones(size(index9t))];
```

We first try a linear kernel

$$K(x, x') = x^T x'.$$

KernelFunctions allows us to add a centering term as well but uses zero by default. Choosing $\lambda = 0.1$ seems to give decent results here.

In [142…
```
output1 = kernel_ridge_regression(LinearKernel(),X₁',y₁,Xstar₁',0.1);
pct_classified(output1,ystar₁)
```

```
percent classified correctly: 99.067164%
```

Now, we use a polynomial kernel

$$K(x, x') = (x^T x' + c)^\alpha,$$

where $\alpha \in \mathbb{N}$. By default, KernelFunctions uses $\alpha = 2$ and $c = 0$, but using a quartic polynomial ($\alpha = 4$) seems to give better results here. $c = 0$ and $\lambda = 0.1$ seem to still work here.

In [140…
```
output2 = kernel_ridge_regression(PolynomialKernel(degree=4),X₁',y₁,Xstar₁',0.1);
pct_classified(output2,ystar₁)
```

```
percent classified correctly: 99.720149%
```

Finally, we try an RBF kernel

$$K(x, x') = \exp\left(-\frac{\|x - x'\|_2^2}{2\sigma^2}\right).$$

KernelFunctions only allows for $\sigma = 1$, so we have to rescale our data to adjust this parameter. We find that $\sigma = 1000$, $\lambda = 0.1$ gives good results here.

In [136…
```
output3 = kernel_ridge_regression(RBFKernel(),X₁'/1000,y₁,Xstar₁'/1000, 0.1);
pct_classified(output3,ystar₁)
```

```
percent classified correctly: 99.813433%
```

For this case, it seems that all three kernels perform well, but polynomial and RBF perform a bit better than linear.

Now, let's repeat this experiment for 3s and 8s, which should be a bit more difficult.

In [160…
```
index3 = findall(training_labels.==3)
index8 = findall(training_labels.==8)
X₂ = Xte[[index3; index8],:]
y₂ = [ones(size(index3)); -ones(size(index8))];
index3t = findall(testing_labels.==3)
index8t = findall(testing_labels.==8)
Xstar₂ = Yte[[index3t; index8t],:];
ystar₂ = [ones(size(index3t)); -ones(size(index8t))];
```

We see that the linear kernel performs significantly worse than in the previous case, but we can only tinker with the regularization term which doesn't affect too much. $\lambda = 0.1$ seems to be about optimal.

In [153…
```
output1 = kernel_ridge_regression(LinearKernel(),X₂',y₂,Xstar₂',0.1);
pct_classified(output1,ystar₂)
```

```
percent classified correctly: 90.473790%
```

Reducing the degree of our polynomial (using $\alpha = 3$ instead of $\alpha = 4$) produces better results, but we still aren't nearly as accurate as the previous case.

In [156…
```
output2 = kernel_ridge_regression(PolynomialKernel(degree=3),X₂',y₂,Xstar₂',0.1);
pct_classified(output2,ystar₂)
```

```
percent classified correctly: 98.135081%
```

As before, RBF performs just slightly better than polynomial but much worse than in the previous case. The same hyperparameters ($\sigma = 1000$, $\lambda = 0.1$) still seem to be optimal.

In [157…
```
output3 = kernel_ridge_regression(RBFKernel(),X2'/1000,y2,Xstar2'/1000,0.1);
pct_classified(output3,ystar2)
```

```
percent classified correctly: 98.538306%
```

Now, we try to classify 1s and 7s.

In [159…
```
index7 = findall(training_labels.==7)
X3 = Xte[[index1; index7],:]
y3 = [ones(size(index1)); -ones(size(index7))];
index7t = findall(testing_labels.==7)
Xstar3 = Yte[[index1t; index7t],:];
ystar3 = [ones(size(index1t)); -ones(size(index7t))];
```

Linear does decently again, but is still not as accurate as we'd like.

In [162…
```
output1 = kernel_ridge_regression(LinearKernel(),X3',y3,Xstar3',0.1);
pct_classified(output1,ystar3)
```

```
percent classified correctly: 98.289413%
```

Now, a degree 4 polynomial seems like the optimal choice.

In [164…
```
output2 = kernel_ridge_regression(PolynomialKernel(degree=4),X3',y3,Xstar3',0.1);
pct_classified(output2,ystar3)
```

```
percent classified correctly: 99.722607%
```

Choosing $\sigma = 1000$ still seems optimal. Interestingly, we get the same accuracy for RBF and polynomial.

In [175…
```
output3 = kernel_ridge_regression(RBFKernel(),X3'/1000,y3,Xstar3'/1000,0.1);
pct_classified(output3,ystar3)
```

```
percent classified correctly: 99.722607%
```

Finally, we look to classify 5s and 2s. This should also be a bit more difficult.

In [167…
```
index5 = findall(training_labels.==5)
index2 = findall(training_labels.==2)
X4 = Xte[[index5; index2],:]
y4 = [ones(size(index5)); -ones(size(index2))];
index5t = findall(testing_labels.==5)
index2t = findall(testing_labels.==2)
Xstar4 = Yte[[index5t; index2t],:];
ystar4 = [ones(size(index5t)); -ones(size(index2t))];
```

The linear kernel once again does poorly, but there's not much that we can do to amend this other than use a different kernel.

In [169…
```
output1 = kernel_ridge_regression(LinearKernel(),X4',y4,Xstar4',0.1);
pct_classified(output1,ystar4)
```

```
percent classified correctly: 93.139293%
```

Interestingly, the degree 3 polynomial seems to be the right choice here. It seems that $\alpha = 3$ is optimal when the digits are difficult to distinguish between while $\alpha = 4$ works better when the digits are easier to tell apart.

In [172…
```
output2 = kernel_ridge_regression(PolynomialKernel(degree=3),X4',y4,Xstar4',0.1);
pct_classified(output2,ystar4)
```

```
percent classified correctly: 99.272349%
```

Once again, $\sigma = 1000$ seems to be the optimal choice for RBF, and it performs comparably to the polynomial kernel.

In [171…

```
output3 = kernel_ridge_regression(RBFKernel(),X4'/1000,y4,Xstar4'/1000,0.1);
pct_classified(output3,ystar4)
```

```
percent classified correctly: 99.324324%
```

Overall, we find that when well-tuned, the polynomial and RBF kernels perform fairly well, while the linear kernel performs much worse. This suggests that threshold which divides our data is simply nonlinear, so nonlinear functions will never be able to capture it well. A linear kernel is really just equivalent to OLS, so this says that in order to fit MNIST accurately, we need to do something fancier than linear regression; however, this is a simple enough data set that just adding some basic nonlinearities improves our classification substantially. Hence, polynomial and RBF kernels perform quite well, even when distinguishing between similar-looking digits.

In [ ]: