6.867: Problem Set 2

October 25, 2016

1 Logistic Regression

We implemented logistic regression with L_2 regularization. Recall that the objective function for L_2 regularization takes the form

$$E_{LR}(w, w_0) = \text{NLL}(w, w_0) + \lambda ||w||_2^2 \tag{1}$$

where NLL is the logistic loss

$$\text{NLL}(w, w_0) = \sum_i \log(1 + \exp(-y^{(i)}(w \cdot x^{(i)} + w_0))) \tag{2}$$

and the $y^{(i)}$ are labels ± 1. We minimized the objective function by gradient descent; note that the gradient of E_{LR} with respect to w is

$$\frac{dE_{LR}}{dw} = \sum_{i=1}^{n} \left(s(w \cdot x^{(i)}) - y^{(i)} \right) x^{(i)} - 2\lambda w \tag{3}$$

where $s(x) = 1/(1 + e^{-x})$ is the sigmoid function.

We tested our implementation against four artificial datasets, the first three with 400 training samples, 200 validation samples, and 200 testing samples, and fourth with 400 training, validation, and testing samples. The data consisted of two-dimensional vectors $x^{(i)}$ labeled by $y^{(i)} = \pm 1$.

With both $\lambda = 0$ and $\lambda = 1$, $||w||$ decreased and eventually converged as gradient descent progressed. When $\lambda = 1$, $||w||$ was penalized, and so the final $||w||$ was slightly smaller than it was when $\lambda = 0$.

1.1 L_1 and L_2 Regularization

We compared the effects of L_1 and L_2 regularization. Recall that the objective function with L_1 regularization is

$$E_{LR}(w, w_0) = \text{NLL}(w, w_0) + \lambda ||w||_1 \tag{4}$$

where $||w||_1 = \sum_{i=1}^{n} |w_i|$. While L_2 regularization maintained small values throughout w, L_1 allowed for sparse w, especially when λ increased. L_1 also tended to be less stable, causing more fluctuations in w across different λ. At high λ (e.g. $\lambda = 8$), the sparser L_1 weights led to higher classification error than the L_2 weights. However, both regularization schemes produced similar decision boundaries, as drawn in Figure 1.

We selected the optimal hyperparameters for each dataset by rate of misclassification on the validation set. Some datasets did not have a unique optimal parameter. For these, we arbitrarily selected one. We evaluated these optimal models' performance on our test sets; we present our results below.

dataset	λ	regularization	accuracy
1	1.0	L_2	1.000
2	4.0	L_2	0.810
3	4.0	L_1	0.975
4	1.0	L_1	0.500

2 Support Vector Machines (SVMs)

2.1 Dual Soft-SVM

We implemented the dual form of soft-SVM with third-party convex optimization software. Recall that the soft-SVM dual optimization problem takes the form

$$\begin{aligned} \text{maximize} \quad & -\frac{1}{2}\sum_{i,j} \alpha_i \alpha_j y^{(i)} y^{(j)} \left\langle x^{(i)}, x^{(j)} \right\rangle + \sum_i \alpha_i \\ \text{subject to} \quad & 0 \le \alpha_i \le C \\ & \sum_{i=1}^{n} \alpha_i y^{(i)} = 0 \end{aligned} \tag{5}$$

for training data $x^{(i)}$ labeled by $y^{(i)} = \pm 1$. Hyperparameter C controls the amount of slack we permit. Given a new data point x, we predict its label y as

$$y = \text{sgn}\left(\sum_i \alpha_i^* y^{(i)} \left\langle x^{(i)}, x \right\rangle \right) \equiv \text{sgn}(t(x)) \tag{6}$$

where α_i^* is the optimal value of α_i from Equation 5.

For instance, on a toy dataset with positive examples $(2, 2), (2, 3)$ and negative examples $(0, -1), (-3, -2)$, our optimization problem takes the form

$$\begin{aligned} \text{maximize} \quad & -\frac{1}{2}\alpha^T \begin{bmatrix} 8 & 10 & 2 & 10 \\ 10 & 13 & 3 & 12 \\ 2 & 3 & 1 & 2 \\ 10 & 12 & 2 & 13 \end{bmatrix} \alpha + \sum_{i=1}^{4} \alpha_i \\ \text{subject to} \quad & 0 \le \alpha_i \le C \\ & \alpha_1 + \alpha_2 = \alpha_3 + \alpha_4 \end{aligned} \tag{7}$$

Upon inspection, it is clear that for large C (i.e. in the hard-SVM limit), the support vectors are $(2, 2)$ and $(0, -1)$.

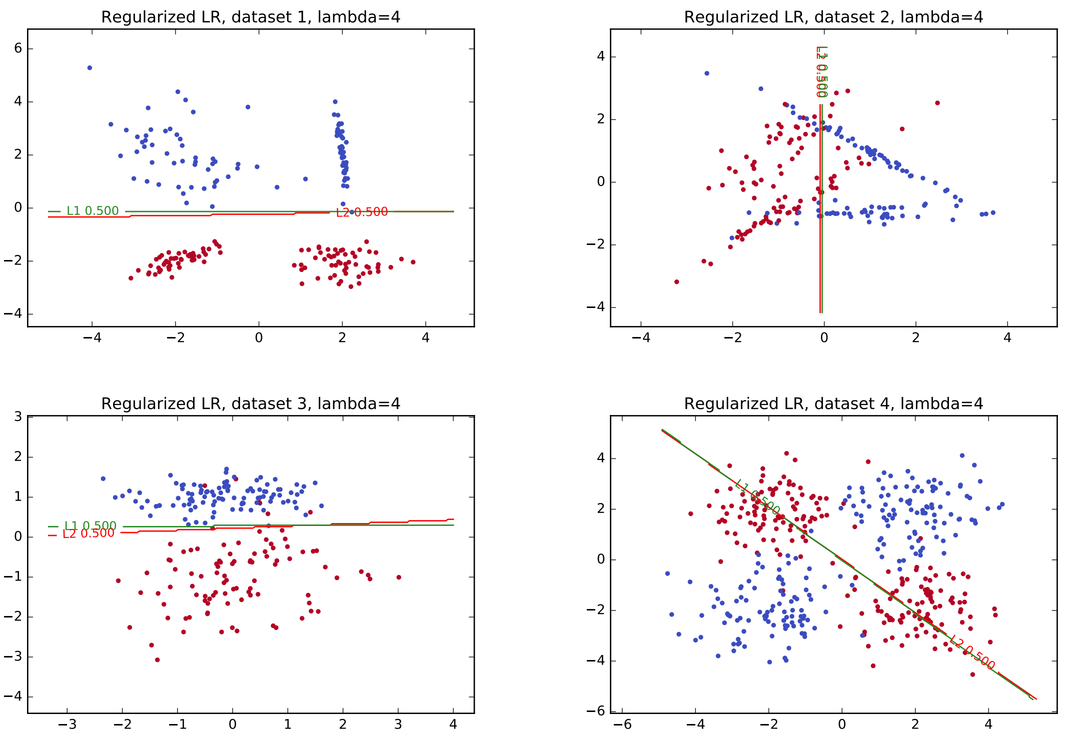

Figure 1: Decision boundaries from L_2/L_1 normalization. The red line is the L_2 boundary; the green, L_1. Datasets 1 and 3 are more or less separable, unlike datasets 2 and 4. Classification accuracy for L_2/L_1 regularized logistic regression were respectively, in order of dataset, 1.0 and 0.995; 0.81 and 0.81; 0.97 and 0.975; 0.4975 and 0.5.

We tested our soft-SVM implementation on the same four 2D datasets from the previous section. Setting regularization parameter $C = 1$, we get the following support vector counts and misclassification rates on validation data:

Dataset	Support vectors	Misclassification
1	4/400	0/200
2	174/400	36/200
3	33/400	6/200
4	392/400	122/400

Our model clearly performs better in datasets 1 and 3. Indeed, these datasets are linearly separable (or very nearly so), as is apparent from Figure 2, where the test data from each dataset are plotted. Observe also that the number of support vectors generally tracks the misclassification rate.

In the next section, we will kernelize our SVM routine to handle nonlinearities in our data (for instance, the XOR data in dataset 4 is roughly separable, but only nonlinearly).

2.2 Kernelization

It is often useful to map our raw data with some nonlinear feature map ϕ into a higher dimensional feature space. However, it is often infeasible to compute and store in memory all the values $\phi(x^{(i)})$.

Therefore, we recall the "kernel trick": the insight that because Equation 5 and 6 only refer to values $\phi(x^{(i)})$ inside inner products with other $\phi(x^{(j)})$, we only need the kernel function

$$k(x, x') = \langle \phi(x), \phi(x') \rangle, \tag{8}$$

which is usually much easier to compute than the feature map ϕ.

By replacing the inner products in Equations 5 and 6 with the corresponding kernels, we arrive at the kernelized soft-SVM optimization problem. Thus, we kernelized our SVM routine and tested it with a linear kernel

$$k(x, x') = x \cdot x' \tag{9}$$

and with Gaussian RBF kernels

$$k(x, x') = \exp\left(-\frac{|x - x'|^2}{2\sigma^2}\right) \tag{10}$$

with bandwidths $\sigma = 0.25, 0.5, 1.0, 2.0$ on the same four datasets as before. We took

$$C \in \{0.01, 0.1, 1, 10, 100\}.$$

In particular, we performed model selection as follows: within both the family of linear SVMs (parametrized by C) and the family of RBF SVMs (parametrized by C and σ). we took the hyperparameters that produced the best model by validation error (using misclassification as our error metric). Finally, we took the best of the best two models (again by

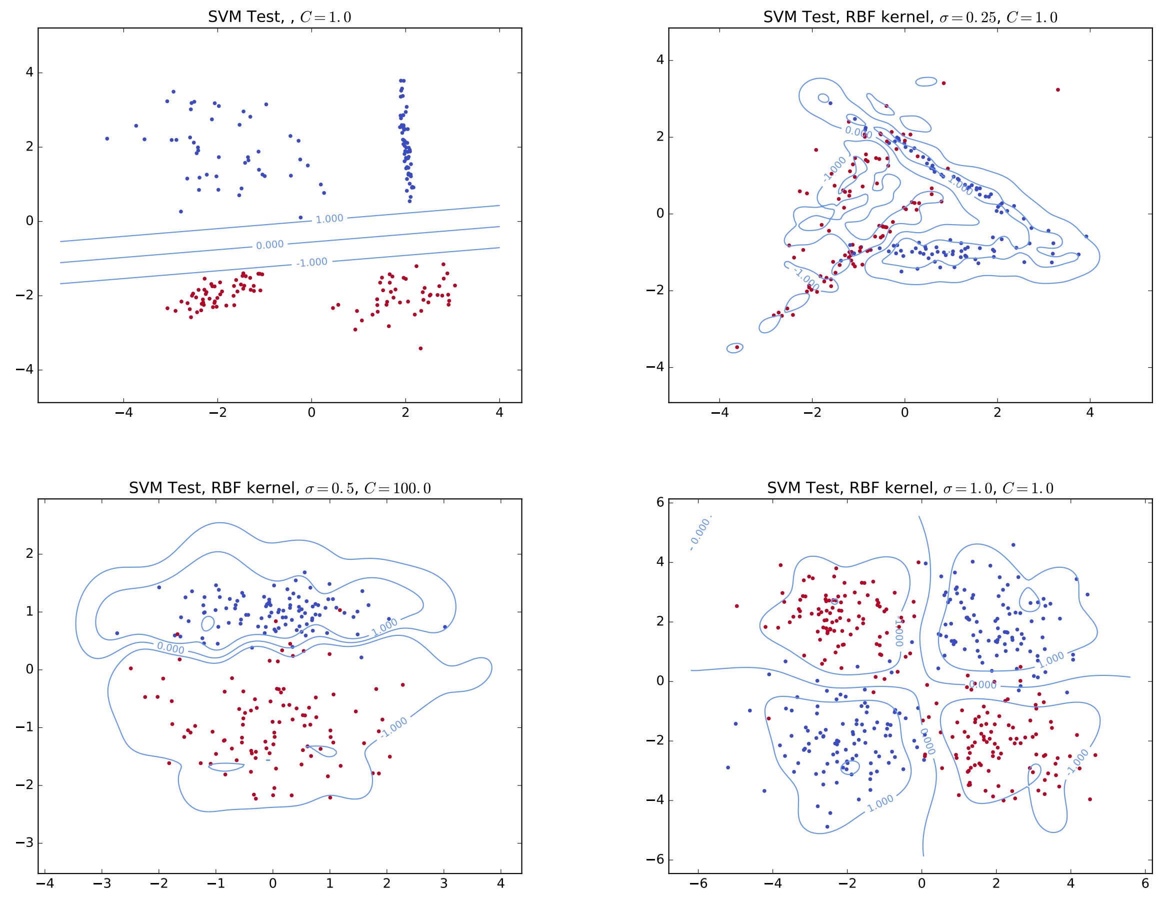

Figure 2: Decision boundaries (with contours $t = \pm 1$) on test data for best models produced by model selection. Our best models for each dataset had misclassification rates of 0, 6, 3.5, 4.25%, respectively.

validation error). We show these best models on test data in Figure 2 along with the corresponding test errors.

In the linearly separable case, with a linear kernel, we found that increasing C led to tighter margins and fewer support vectors. The model made no misclassifications on the validation data for C in the range we tested.

Gaussian kernels worked well for bandwidths σ commensurate to the size of the spatial features of the dataset. Excessively small σ created decision boundaries that overfit to training data. On the other hand, models with σ too large were unable to discern tight structure. We illustrate these findings in Figure 3.

In general, Gaussian kernel SVMs have lower bias, but introduce greater variance, since using RBF kernels is equivalent to mapping the raw data into an infinite-dimensional feature space. This lower bias is manifested in the larger number of support vectors and in the tighter shapes of the decision boundaries.

We make some observations about hyperparameter C. From Equation 5, larger C corresponds to less tolerance for slack. For small C, the model favors maximizing the margin at the expense of slack, so we expect increasing C shrinks the geometric margin, as our experiments confirm.

Consequently, there are fewer support vectors with large C: the margin becomes tighter, and the model grows increasingly averse to margin errors, which make up the bulk of the support vectors for small C.

The size of the margin is not a good metric for choosing C, as can make the margin arbitrarily large by taking $C \to 0$ (indeed, $C = 0$ allows arbitrary amounts of slack with no penalty on the objective function). We'd instead like a metric that evaluates the performance of a given C on unseen data.

We could use classification error on the validation set, for example. Another metric is the hinge loss incurred by our classifier on a validation dataset:

$$\sum_i \max\left(0, 1 - y^{(i)} t\left(x^{(i)}\right)\right) \tag{11}$$

where t is the prediction function from Equation 6 and where the sum is taken over all validation data.

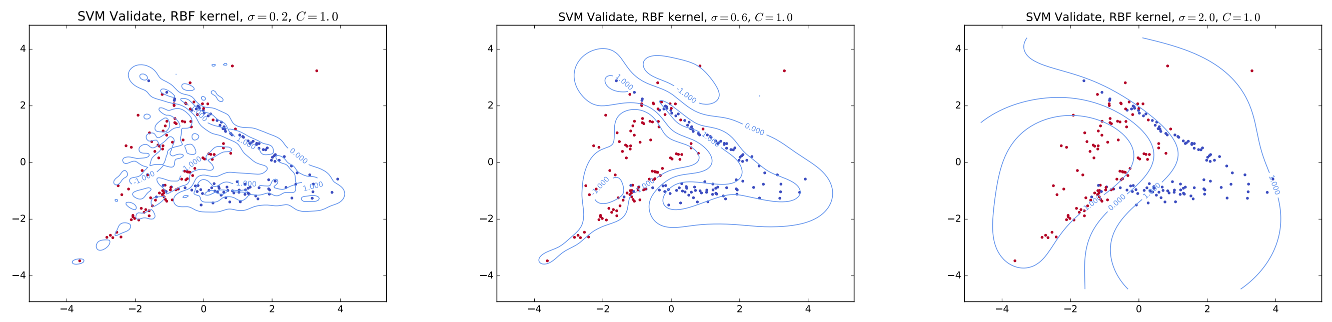

Figure 3: SVM decision boundaries (with contours $t = \pm 1$) on validation data for dataset 2, with $C = 1$ and Gaussian RBF kernels with bandwidths $\sigma = 0.2, 0.6, 2.0$, in that order.

3 SVMs with Pegasos

While generally effective, convenient, and efficient, SVMs become unwieldy to solve as they become large. Instead, we may consider solving the following soft-SVM using the Pegasos algorithm (primal estimated sub-gradient solver for SVM):

$$\min_w \left(\frac{\lambda}{2} ||w||^2 + \frac{1}{n} \sum_{i=1}^{n} \max \left(0, 1 - y^{(i)} (w^T x^{(i)}) \right) \right) \tag{12}$$

This problem is equivalent to C-SVM for $C = \frac{1}{n\lambda}$. Given a regularization parameter λ and some maximum number of "epochs," Pegasos stochastically updates our weight vector w as follows.

1. Initialize t and w_0 to 0.

2. At each iteration, increment t and set a decaying step size $\eta_t = (t\lambda)^{-1}$.

3. Choose some training example i and update

$$w_{t+1} \leftarrow (1 - \eta_t \lambda) w_t + \begin{cases} \eta_t y^{(i)} x^{(i)} & y^{(i)} (w_t^T x^{(i)}) < 1 \\ 0 & \text{otherwise} \end{cases}$$

4. Repeat until t reaches the maximum number of epochs.

We implemented the Pegasos algorithm and included a bias term w_0 that we did not penalize by regularization. (We simply update w_0 as $w_0 = w_0 + \eta_t y^{(i)}$.)

Now recall that we define the margin as $||w||^{-1}$. The regularization constant λ limits the magnitude of w, so as we increase λ, the margin grows. We show this effect in Figure 4 on a sample dataset.

3.1 Kernelized Pegasos

We can extend the Pegasos algorithm to solve the following kernelized soft-SVM problem:

$$\min_w \left(\frac{\lambda}{2} ||w||^2 + \frac{1}{n} \sum_{i=1}^{n} \max \left(0, 1 - y^{(i)} \left(w^T \phi(x^{(i)}) \right) \right) \right) \tag{13}$$

where we map x to some transformation $\phi(x)$ (in the previous section, ϕ was simply the identity). The following kernelized Pegasos algorithm takes in a Gram matrix, where entry $k_{ij} = K(x^{(i)}, x^{(j)}) = \phi(x^{(i)}) \cdot \phi(x^{(j)})$.

1. Initialize t and $\alpha_{t=0}$ to 0.

2. At each iteration, increment t and set a decaying step size $\eta_t = (t\lambda)^{-1}$.

3. Choose some training example i and update

$$\begin{aligned} w_{t+1} \leftarrow & (1 - \eta_t \lambda) \alpha_i \\ & + \begin{cases} \eta_t y^{(i)} & y^{(i)} \left(\sum_j \alpha_j K(x^{(j)}, x^{(i)}) \right) < 1 \\ 0 & \text{otherwise} \end{cases} \end{aligned}$$

4. Repeat until t reaches the maximum number of epochs.

Note that kernelized Pegasos differs from vanilla Pegasos primarily in the use of an n-dimensional α instead of a d-dimensional weight vector w. We also replace $x^{(i)}$ with $\phi(x^{(i)})$.

The vector α may be sparse, with non-zero entries for support vectors (as it was in the dual soft-SVM optimization problem). Given optimal α_i and a new data point x, we predict its label y as

$$y = \text{sgn} \left(\sum_i a_i K(x, x^{(i)}) \right), \tag{14}$$

in analogy with Equation 6.

We implemented kernelized Pegasos and tested it with Gaussian RBF kernels, fixing $\lambda = 0.02$ and training for 10000

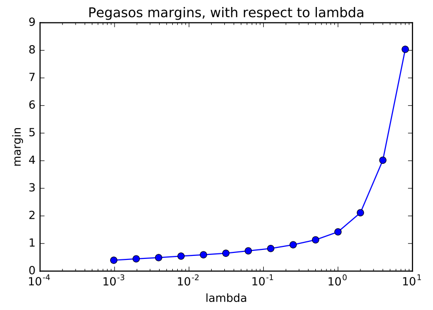

Figure 4: Increasing λ increases the margin $||w||^{-1}$.

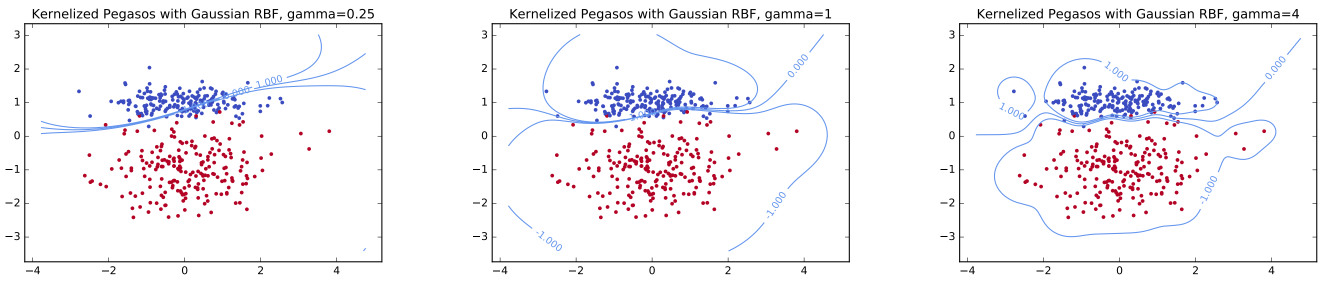

Figure 5: Larger γ increases the fit to data. On the test sets, classification errors were 28/200, 14/200, and 11/200, respectively. These results are in analogy with those presented in Figure 3.

epochs, varying the bandwidth $\gamma = \{2^{-2}, \dots, 2^2\}$ (where $\gamma = \frac{1}{2}\sigma^{-2}$). We found that smaller γ resulted in fewer support vectors:

γ	2^{-2}	2^{-1}	1	2	4
SV	1/400	1/400	2/400	45/400	100/400

The decision boundary also becomes tighter as we decrease γ. We illustrate this effect in Figure 5 for a sample dataset.

4 MNIST Digit Recognition

We tested the previous classifiers on the MNIST dataset of 28×28 grayscale images of handwritten digits, given as a labeled dataset of 28^2 dimensional vectors with integer entries $[0, 255]$ representing the grayscale luminescence of each pixel.

We optionally normalized input pixel features to the range $[-1, 1]$, so that an input vector x is mapped to $\frac{2x}{255} - 1$, with all operations applied componentwise.

For all our following experiments, we chose pairs of disjoint subsets of digits to use as our positive and negative classes for binary classification. Training sets contained 250 samples of each digit in either of the two classes; validation and test sets each contained 150 samples of each digit.

4.1 Linear Models

We begin by exploring the efficacy of our linear models on MNIST. In particular, we compare logistic regression from section 1 with the linear SVM classifier from section 2. We experimented with the following binary classification problems; the results of our experiments (optimal hyperparameters and resulting test set misclassification rates) are listed next to them.

+	-	λ	LR error	C	SVM error
1	7	0.5	3/300	0.01	4/300
3	5	0.5	18/300	0.01	16/300
4	9	0.5	12/300	0.01	17/300
even	odd	0.5	225/1500	0.01	245/1500

We used unnormalized data for our experiments above. Normalization did not significantly affect test set accuracy for either model. Neither model performed significantly better than the other.

We visually inspected some classification errors (Figure 6). The images were generally very ambiguous, even to the human eye, explaining their misclassification. For example, a roundish 4 may look like a 9.

4.2 SVMs with RBF kernels

Next, we introduced nonlinearity into our models by using Gaussian RBF kernels with our SVM classifiers. We performed model selection as in section 2.2 with only RBF

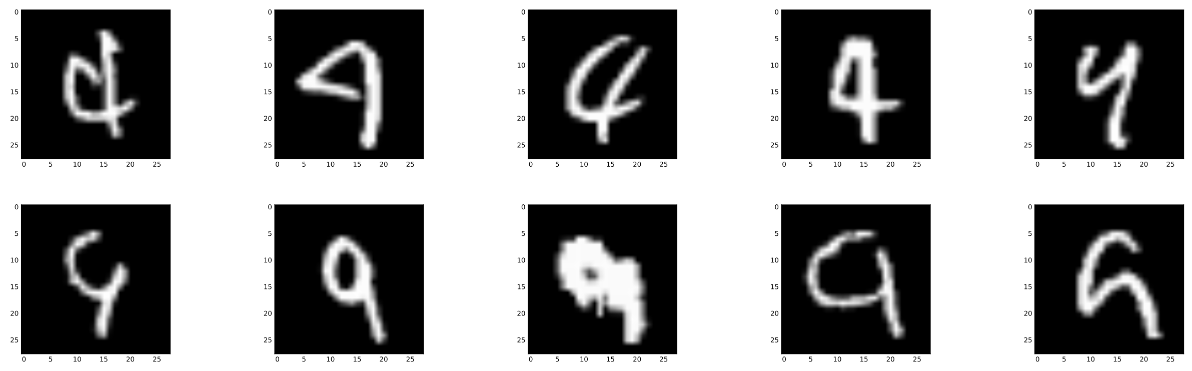

Figure 6: Misclassified images. The top row contains images of 4's misclassified as 9's, while the bottom row contains 9's misclassified as 4's.

kernels (with the same values of C and with bandwidths $\sigma = 0.03, 0.1, 0.3, \ldots, 300$ spaced log-uniformly).

Unlike with linear SVMs, kernelized SVM classifiers performed significantly better when trained on normalized data. Indeed, validation and test set classification errors uniformly dropped by roughly a factor of 10 when trained on normalized data.

Normalizing, our optimal hyperparameters and their corresponding test set error rates are below:

+	-	σ	C	SVM error
1	7	30	10	4/300
3	5	10	100	5/300
4	9	30	10	16/300
even	odd	-	-	-

The even/odd classification problem was too expensive to perform with our quadratic-programming-based SVM implementation.

Comparing with our results from the previous section, we find that kernelizing our SVMs can improve some results (as was the case in the 3/5 classification problem). In other cases, kernelization had little effect.

4.3 Pegasos and Quadratic Programming

We now run kernelized Pegasos (section 3.1) on the same four classification problems as above. On normalized data, we found optimal values for λ and number of training epochs, and evaluated the resulting models on the respective test set. Our optimal hyperparameters and error rates were as follows:

+	-	λ	epochs	Pegasos error
1	7	0.25	100 000	2/300
3	5	0.50	100 000	20/300
4	9	2.00	100 000	24/300
even	odd	1.00	500 000	240/1500

We see that Pegasos achieves comparable accuracy as our SVM classifier. Given this parity, we consider the runtimes of our quadratic-programming based approach and Pegasos. For the 1/7 classification problem, using Pegasos with $\lambda = 0.25$ and 100000 epochs (chosen because they produced comparably accurate results as our QP approach), the training times of the two approaches increased as:

training set size	QP time (s)	Pegasos time (s)
200	18.1	2.1
300	30.6	2.1
400	46.3	2.0
500	67.8	2.1

While QP runtime grows rapidly as training set size increases, Pegasos runtime largely remains fast and constant. Because Pegasos algorithm stochastically updates on single vectors, increasing the training set does not affect how fast the algorithm trains. Indeed, stochastic optimization methods are much more computationally tractable for large datasets.