Assignment 2

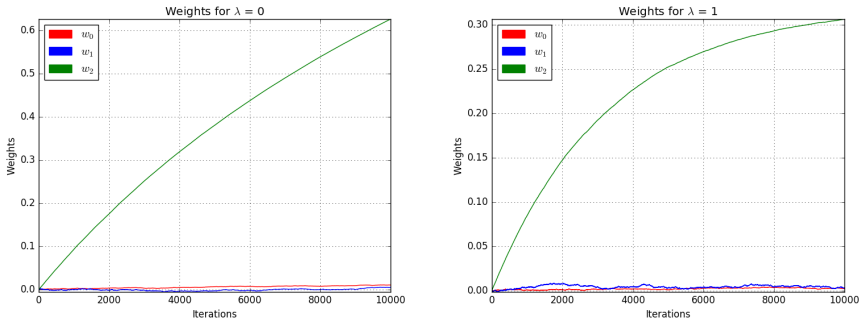
Anonymous

October 25, 2016

1 Logistic Regression

We attempt to classify four datasets using logistic regression. We try using both a L_1 and L_2 regularizer, where the subscript indicates the exponent of the 2-norm of the weight vector w. Our objective function is $E(w, w_0) = NLL(w, w_0) + \lambda||w||_2^{obj}$ where $obj \in \{1, 2\}$ for L_1 and L_2 respectively and NLL is the negative log likelihood.

We first run a gradient descent method with L_2-regularization to optimize the logistic regression objective. With $\lambda = 0$, there is no regularization and the weights are free to become as large as they want (Fig. 1.1a). Using $\lambda = 1$, we have a gradual tapering off of the weights as they large weights are penalized (Fig. 1.1b). Note that the value of w_3 after 10000 iterations for $\lambda = 1$ is only $\frac{1}{2}$ of the value for $\lambda = 0$.



(a) $\lambda = 0$ (b) $\lambda = 1$

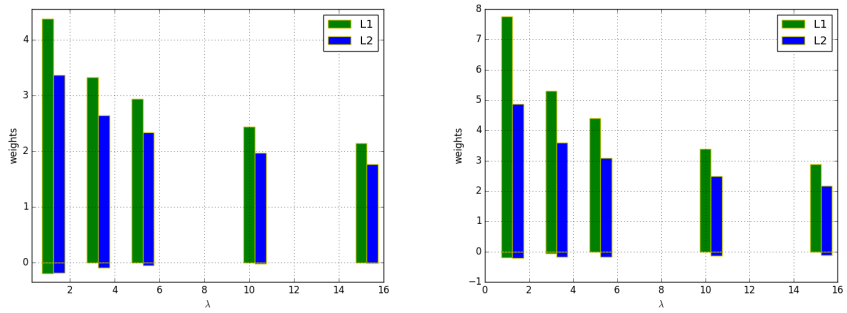
Figure 1.1: The value of the weights for fitting a L_2-regularized logistic regression to dataset 1 as the number of iterations increases.

When λ increases in L_1 regularization, this makes the weights slowly drop out (feature selection), while in L_2, this makes the weights slowly tend to 0. In both cases, with higher λ, $|w|$ is smaller (Fig. 1.2). In all cases, the weights from L_2 regularization are lower than their corresponding weights with L_1 regularization, and one of the weights drops out for $\lambda \geq 3$ (dataset 1) and $\lambda \geq 5$ (dataset 3). Datasets 2 and 4 are not shown but have similar qualitative behavior.

λ also affects the decision boundary and hence the classification error rate for the datasets (Figure 1.3). The number of misclassifications (Figure 1.4) does not always have a unique minimum (e.g. Fig. 1.4a), but in picking the best hyperparameters for each dataset, we opted to prefer lower λ values to prevent our weights from becoming too small. Table 1.1 shows the performance of our best models on the testing set. Our training-validation-testing data ratio was 2:1:1.

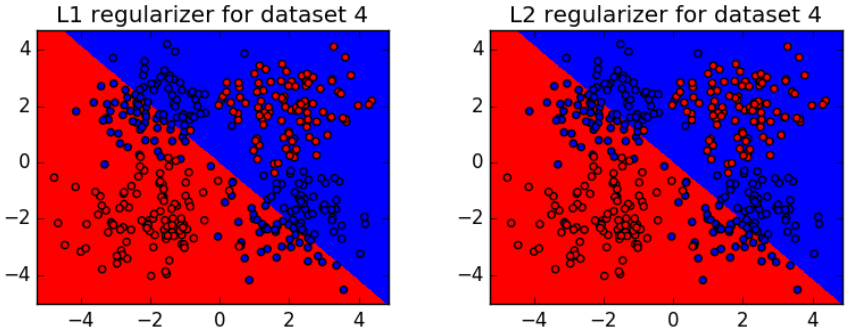
Table 1.1: The performance of our models using the optimal λ and choice of L_1/L_2 on the test set.

Dataset	L1 or L2	λ	Errors
1	L1	1	0
2	L1	1.1	38
3	L2	2.4	5
4	L1	11.0	191

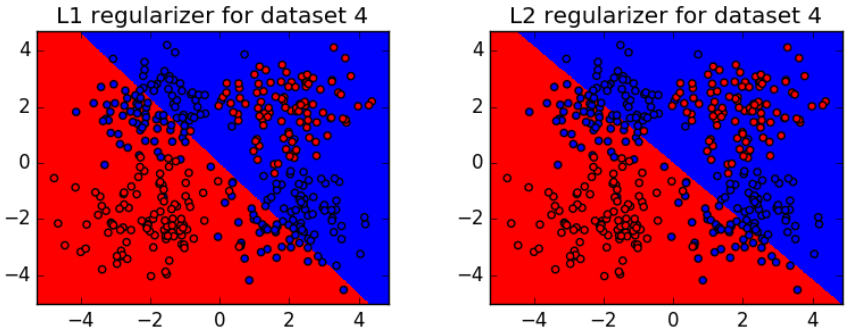


(a) Dataset 1 (b) Dataset 3

Figure 1.2: The value of the weights for L_1 (green) and L_2 (blue) regularization for a variety of λ values. Note that the two components of w are opposite in sign. Only two datasets are shown here for brevity, but all datasets show the same decreasing trend for $|w_i|$ as λ increases.

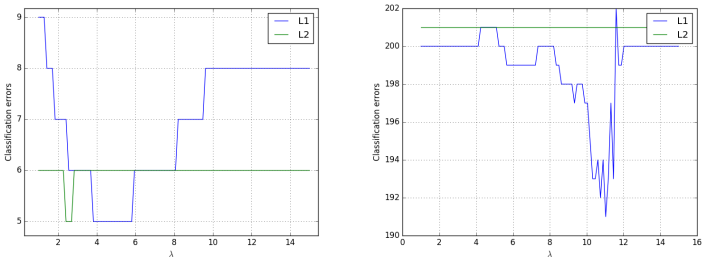


(a) L_1, $\lambda = 1$ (b) L_2, $\lambda = 1$



(c) L_1, $\lambda = 10$ (d) L_2, $\lambda = 10$

Figure 1.3: The effect of λ and regularization type on the decision boundary in dataset 4. The differences are minor but noticeable if you compare the x-intercept of the decision line.



(a) Dataset 3 (b) Dataset 4

Figure 1.4: The number of classification errors as a function of λ. We see the minimum number is near $\lambda = 2.4$ with L_2 for dataset 3 and near $\lambda = 11$ for dataset 4 with L_1. Note that the optimal hyperparameters for dataset 3 are not unique; to tie-break, we chose the lowest λ value.

2 Support Vector Machines

For our next learning classifier we implemented the dual form of linear support vector machines (SVM) with slack variables. The goal of support vector machines is to find a hyperplane which separates the data in different categories with the largest margin possible. New examples are then mapped to the space and categorized based on which side of the hyperplane they fall on. In order to find this hyperplane we implement the following dual optimization problem:

$$\arg\max_\alpha \quad \sum_{i=1}^{n} \alpha_i - \frac{1}{2}\sum_{i=1}^{n}\sum_{j=1}^{n} y_i y_j \alpha_i \alpha_j x_i^T x_j \tag{2.1}$$

$$\text{s.t.} \quad 0 \leq \alpha_i \leq C, \forall i \quad \text{and} \quad \sum_{i=1}^{n} y_i \alpha_i = 0, \forall i$$

To solve this quadratic programing problem we utilized the *cvxopt* package in python.The quadratic programming package we used framed the quadratic programming problem as:

$$\text{minimize } (1/2)\alpha^T P \alpha + q^T \alpha \tag{2.2}$$

$$\text{subject to } G\alpha < h; y\alpha = b \tag{2.3}$$

Using this framework for our optimization problem:

$$q = \begin{pmatrix} -1 \\ \vdots \\ -1 \end{pmatrix} \quad ; b = 0 \tag{2.4}$$

$$G = \begin{pmatrix} \mathbb{1}_n * -1 \\ \mathbb{1}_n \end{pmatrix} \quad ; h = \begin{pmatrix} 0 \\ \vdots \text{ n times} \\ 0 \\ C \\ \vdots \text{ n times} \\ C \end{pmatrix} \tag{2.5}$$

$$P = \begin{pmatrix} y_1 y_1 x_1^T x_1 & y_1 y_2 x_1^T x_2 & \dots & y_1 y_n x_1^T x_n \\ y_2 y_1 x_2^T x_1 & \dots & \dots & y_2 y_n x_2^T x_n \\ \vdots & & & \\ y_n y_1 x_n^T x_1 & \dots & \dots & y_n y_n x_n^T x_n \end{pmatrix} \tag{2.6}$$

Fig. 2.1 illustrates the optimized margin on sample data.

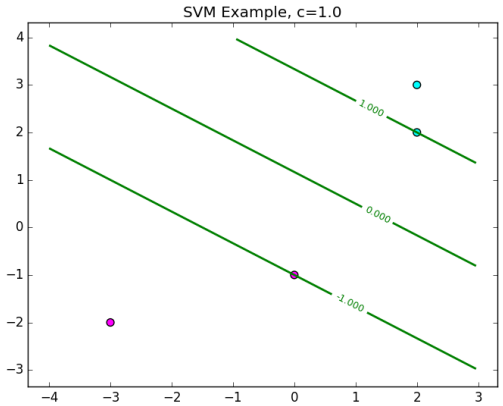


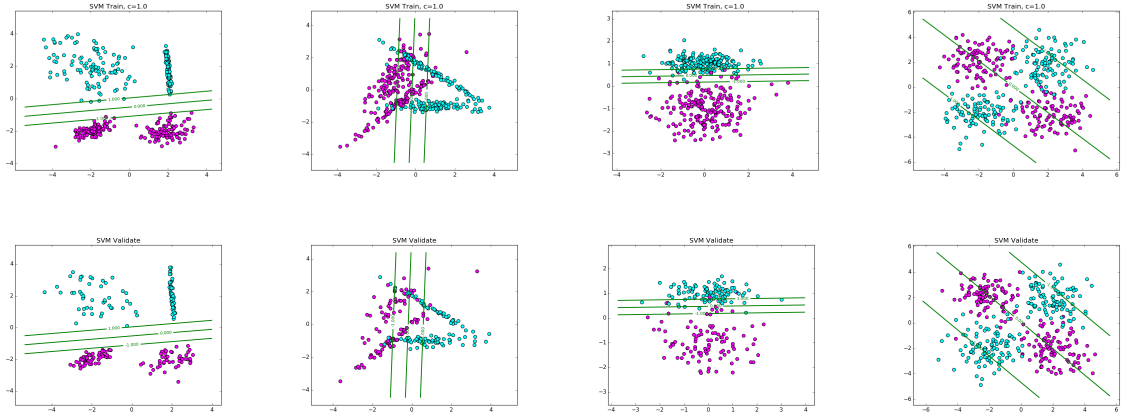
Figure 2.1: SVM with 4 sample points in 2 different categories. The support vectors are the points (2,2) and (0,-1).

Next, we kept C = 1 and tried our SVM implementation on four datasets. Some of the datasets were linearly separable while others were not (Fig 2.2). Notice that Figure 2.2a is linearly separable, but because we are using a soft threshold some points fall within the margin. Clearly, the linear separator has some limitations as illustrated in Figure 2.2d which has a large classification error rate.

A remedy for this high misclassification rate is to use the kernel trick. To use a kernel, instead of having $x_i^T x_j$ in equation 2.1, we use $k(x_i, x_j)$. The kernels we experimented with were the Gram kernel and Gaussian RBF kernel. Graph 2.3 illustrates the Gram kernel on dataset 3 for different C values.

As shown in Figure 2.3 and Table 2.1, as C increases the geometric margin decreases as the machine attempts to minimize misclassifications. This will continue to happen as we increase C. Dataset 1 is linearly separable so once C gets large enough the number of support vectors will not change. In cases where the data is not linearly separable like in datasets 2 and 4, increases in C do not decrease the number of support vectors due to the high misclassification rate.

The Gaussian kernel is: $K(X, X') = exp(-\gamma||X - X'||^2)$



(a) Dataset 1 (b) Dataset 2 (c) Dataset 3 (d) Dataset 4

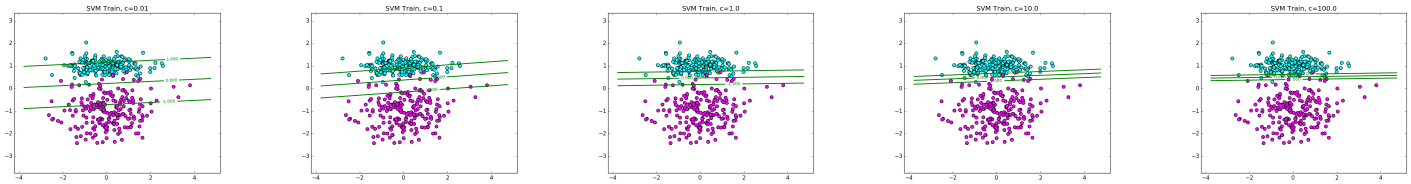


Figure 2.2: SVM with Gram Kernel Applied to Dataset 3 with different size C

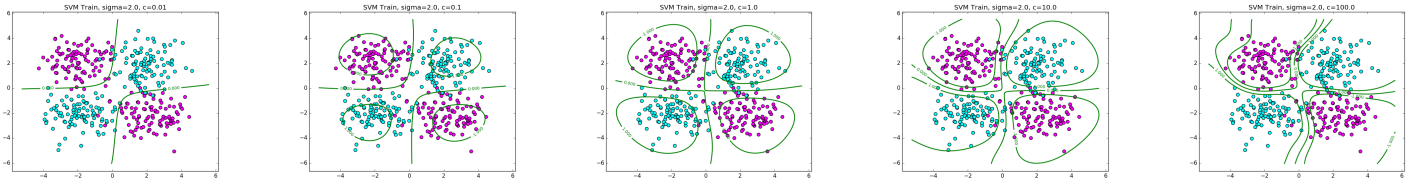


Figure 2.3: SVM with Gaussian Kernel Applied to Dataset 4 with different size C

Table 2.1: Values are displayed as $\frac{1}{\|\omega\|}$ / Number of Support Vectors Using Gram Kernel

C	.01	.1	1	10	100
Dataset 1	1.380 / 75	.5437 / 20	.2979 / 4	.2523 / 3	.2523 / 3
Dataset 2	.3998 / 253	.0542 /186	.0058 / 176	.0005 / 322	5.8×10^{-5} / 242
Dataset 3	.5436 / 184	.1493 / 69	.0318 / 33	.0052 / 20	.0006 / 17
Dataset 4	.2512 / 399	.02552 / 394	.0025 / 395	.0002 / 396	2.5×10^{-5}/ 400

Figure 2.4 shows using a Gaussian kernel with different sizes of C. As with the Gram kernel, the geometric margin decreases as C increases. When using a Gaussian kernel it can be very easy to overfit the data. This is because the feature space of the kernel has infinite dimensions. An important feature to look at is how many support vectors were found. As shown in Table 2.3 for dataset 4, the number of support vectors get smaller until C = 100, at which point the number of support vectors jumps to 308. This is an indication that for C = 100 the training data is being overfit.

The way to find the best C for the dataset is to test the SVM of the validation set of the dataset. We used the training set to train the SVM to get the weights and tested on the validation set to confirm we were choosing an appropriate C. Using this method confirms that using a Gaussian kernel with C = 100 over fits the data. The error rate on the validation for C = 10 is 5.24% while the error rate for C = 100 is 5.5 % .

When using the Gaussian kernel we also need to find the γ that best fits the data. Table 2.3 shows our findings when experimenting with different γ values.

Table 2.2: Values are displayed as $\frac{1}{\|\omega\|}$ / Number of Support Vectors Using Gaussian Kernel

C	.01	.1	1	10	100
Dataset One	.2512 / 398	.1250 / 86	.0611 / 20	.0401 / 8	.0401 / 10
Dataset Two	.2564 / 391	.0438 / 243	.0063 / 173	.0008 / 249	.0001 / 261
Dataset Three	.2604 / 388	.0694 / 146	.01864 /55	.0036 / 31	.0005 / 33
Dataset Four	.2500 / 400	.0553 / 188	.0127 / 88	.0021 / 59	.0002/ 308

Table 2.3: Values are displayed as $\frac{1}{\|\omega\|}$ / Number of Support Vectors Using Gaussian Kernel

γ	.25	.5	1	2	4
Dataset One	.0072 / 22	.0661 / 36	.0517 / 52	.0360 / 87	.0239 / 153
Dataset Two	.0072 / 144	.0086 / 128	.0095 / 135	.0104 / 231	.0106 / 162
Dataset Three	.02165 / 50	.0234 / 48	.0245 /61	.0229/ 153	.01965 / 244
Dataset Four	.0136 / 83	.0137 / 145	.0129 / 271	.0109 / 212	.0083/ 329

3 Pegasos

An alternative to SVM is using Pegasos, which is like C-SVM if $C = \frac{1}{n\lambda}$. The objective function to minimize over w is

$$\frac{\lambda}{2}||w||^2 + \frac{1}{n}\sum_{i=1}^{n}\max\{0, 1 - y_i(w^T x_i)\}$$

We expect that as λ increases, C decreases and the margin increases. This can be seen in Figure 3.1.

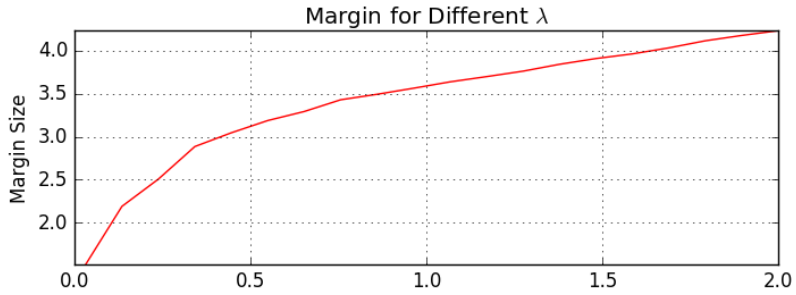


Figure 3.1: The margin ($\frac{1}{|w|}$) as a function of λ.

This method can be generalized, like with SVMs, to use kernels. This will have similar sparsity properties as the dual SVM solution, in that non-support vectors will have Lagrangian coefficients $\alpha_i = 0$. Once we have a model, to predict the another x, the prediction rule is given by

$$\text{Pred}(x) = sgn(\sum_{i\in \text{SV}}^{M} \alpha_i K(x, x_i))$$

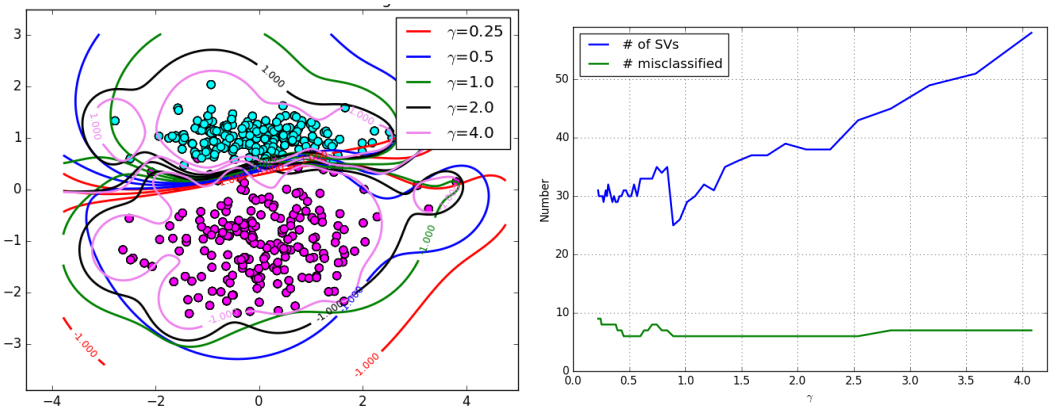
for M support vectors and a kernel function K. As there is no longer a constraint that $\alpha_i \geq 0$ for $\forall i$, our prediction rule will not use y_i, otherwise all the predictions will be positive. In Figure 3.2a the effect of γ choice (the bandwith of the kernel) is shown. The number of support vectors increases as γ increases, as this means that points have a more localized area of influence. We can see that $\gamma = 4$ starts to overfit as it assigns a closed loop around one data point by itself near $(-2.2, 1.5)$.

Comparing to the previous section, the Pegasos method has generally fewer support vectors than the SVM method. They both show the same trend - with increasing γ (decreasing σ), there are more support vectors.

4 Handwritten Digit Recognition with MNIST

Finally, to test the methods described on real world data we classified the MNIST dataset of hand written digits. The goal was to use logistic regression, SVM and Pegasos SVM to classify hand written digits into a binary classification of different subsets of digits. Each digit has 500 images: we split 200 of the images for each digit into training data, 150 to validation data and 150 into test data.

We began with comparing logistic regression with a linear SVM. We looked at the classification of: 1 vs 7, 3 vs 5, 4 vs 9 and (0 ,2 , 4, 6, 8) vs (1, 3, 5, 7, 9). In order to find the best C for the linear SVM we used the validation dataset. We noticed that normalizing the data produces better accuracy. This is because each feature has equal importance and



(a) Decision boundaries

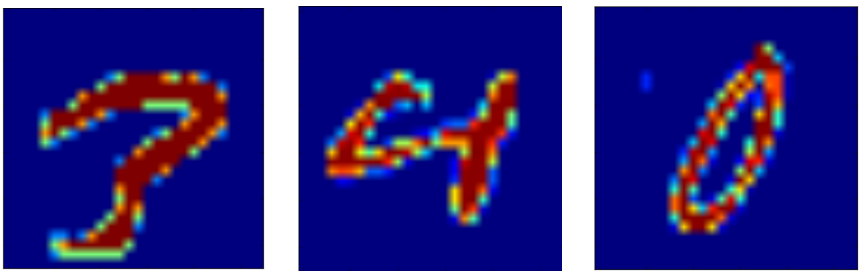
(b) # of support vectors and misclassifications

Figure 3.2: The performance of Pegasos on dataset 1 in terms of (a) decision boundaries and (b) number of support vectors and classification errors, on the validation set using a variety of γ values for the Gaussian kernel with a fixed $\lambda = 0.02$.

by not normalizing we were giving some features more importance than others. Table 4.1 summarizes our finding: the linear SVM consistently out preformed logistic regression. Figure 4.1 shows some examples of misclassified digits. These are clearly sloppily written and we can reasonably expect them to be misclassified.

Table 4.1: The number of misclassified test examples out of 150

	1 vs 7	3 vs 5	4 vs 9	Even vs Odd
Logistic Regression	0	6	4	67
Linear SVM	0	5	3	44



(a) 3 vs 5: Example mislabeled as 5

(b) 4 vs 9: Example mislabeled as 9

(c) Even vs Odd: Example mislabeled as Odd

Figure 4.1: Examples of the misclassified digits

Next, we used a Gaussian kernel SVM to classify the digits. We tried out a series of γ and C to find the combination that had the fewest misclassifications with the validation dataset. Just as in linear SVM, normalizing the data allowed for better accuracy, especially as the exponential nature of Gaussian often distorted non-normalized values. Table 4.2 shows the results of using Gaussian kernel SVM. This produced better results than the linear SVM.

Table 4.2: The number of misclassified test examples out 150

	1 vs 7	3 vs 5	4 vs 9	Even vs Odd
Gaussian SVM	1	3	2	15

We also tried using the Pegasos algorithm to see if we could get a high classification rate in less time than the standard SVM. We set the number of epochs to 10 and searched over different λ and γ to get the best fit. The results were right in the same area of the Gaussian SVM. When the number of training points was 200, Gaussian SVM was quicker at 2.67 seconds compared to Pegasos at 3.1 seconds. Once we got to 400 training examples they were about equal with Gaussian SVM at 12.32 seconds and Pegasos at 12.23 seconds. Once we continued past 400 training examples, Pegasos continued to out-perform Gaussian SVM in terms of speed.