Machine Learning HW 2: Classification

1 Logistic Regression

In this section, we consider an implementation of Logistic Regression and tweak factors such as the regularization function and the regularization parameter λ. We are using Sklearn's *LogisticRegression* class, which uses stochastic gradient descent methods to solve this kind of problems. More specifically, we will consider the problem of classifying 2-dimensional data into two sets, one denoted by value $+1$ and the other by -1. We seek to find weight parameters $w_0, w_1, \ldots, w_n$ (set $w = (w_1, \ldots, w_n)$, where n is the number of features ("dimensions"), so that subsequently, given a new input x, we predict $sgn(\sigma(w_0 + w^T x))$ as the classification sign of x. Here $\sigma(x)$ is the sigmoid function. To fit the model to our data, we wish to minimize the error function $E_{LR}(w, w_0) = \sum_i \log(1 + \exp(-y^{(i)}(wx^{(i)} + w_0))) + \lambda||w||_2^2$, where $\lambda||w||_2^2$ is an L_2 regularization term. First, suppose that there is no regularization, or $\lambda = 0$. Since Sklearn uses $C = 1/\lambda$ as a parameter instead, we will set C to be a large value, namely 10^{-9}. In Figure 1 we compare the weight vector produced for various values of the number of iterations. Note that the magnitudes of the coefficients seem to grow a lot, almost linearly in the number of iterations. In contrast, Figure 2 illustrates the results of the same measurements, but with an L_2 regularizer with $\lambda = 1$. Now the coefficients grow to become much lower, and this is because of the L_2 penalty. Furthermore, note that the regularized version converges much faster, intuiutively since it looks for a "simpler" solution.

Now, we consider L_1 regularization, for which the penalty term becomes $\lambda||w||_1$. In Figure 3 we consider both L_2 and L_1 regularization, as well as various values of λ, and observe the resulting weight vectors. Note that, especially for larger λ, L_1 regularization yields a sparse vector. Generally, we see that the vectors produced by L_1 have larger l_∞ norm than that of L_2, which is normal because L_2 penalizes large terms

iter	w_0	w_1	w_2
1	0.002357	-0.001910	0.917126
6	2.176299	-0.375040	4.77914
11	6.151912	-1.216057	11.734891
16	10.713165	-2.081250	20.676475
21	15.426343	-3.003549	30.143699
26	17.379323	-3.561438	34.698793

Figure 1: Logistic regression (no regularizer): Number of iterations vs weight vector

iter	w_0	w_1	w_2
1	0.002352	-0.001905	0.914969
6	1.132461	-0.178799	3.349396
11	1.13831	-0.182987	3.368100
16	1.13831	-0.182987	3.368100
21	1.13831	-0.182987	3.368100
26	1.13831	-0.182987	3.368100

Figure 2: Logistic regression (L2 regularizer with $\lambda = 1$): Number of iterations vs weight vector

regul, λ	w_0	w_1	w_2
l2,0.01	3.869962	−0.7834012	7.840805
l2,0.1	2.375524	−0.4642034	5.306355
l2,1	1.138297	−0.1829372	3.367916
l2,10	0.3483440	−0.01716593	1.966549
l1,0.01	5.075894	−0.9876057	10.46214
l1,0.1	3.330449	−0.6064500	7.135366
l1,1	1.493504	−0.1892744	4.3710
l1,10	0.0	0.0	2.433784

Figure 3: Logistic regression: Regularization vs weight vector

while L_1 does not.

Figures 4,5,6,7 illustrate the classification model for L_2, L_1 regularization and for different values of λ. As expected, when λ grows we get more misclassifications, and also the classification line becomes "simpler": We note that its slope becomes close to zero, as we increase λ. Furthermore, the classification error percentages for the four cases are $0, 0, 0, 1$ respectively in the first and second datasets, and $1.25, 3.75, 1.25, 3.75$ in the third dataset.

Finally, we use the validation test set to pick the best regularizer and value for λ for each dataset. For the first dataset, we note that it is an "easy" datasets, since for most values of λ and for any regularizer (except for L_1 and $\lambda > 100$) we get no classification errors. So for L_2 regularization and $\lambda = 1$, we get no errors on the test dataset. For the second dataset, we get optimal error ratio for $\lambda \leq 1$, for any regularizer. Picking L_2 regularizer and $\lambda = 1$ gives a 19.5% error for the test dataset. For the third dataset, the optimal error is achieved if we have L_2 with $\lambda \in \{0.01, 1\}$, or L_1 with $\lambda = 0.1$. Picking L_2 with $\lambda = 1$ gives us also a 3% error for the test dataset. It seems that an L_2 regularizer and $\lambda = 1$ is a consistent optimal option for all three datasets.

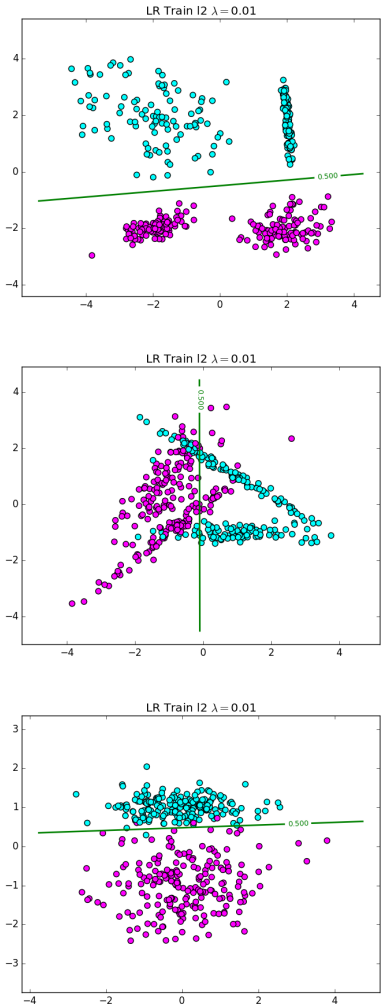
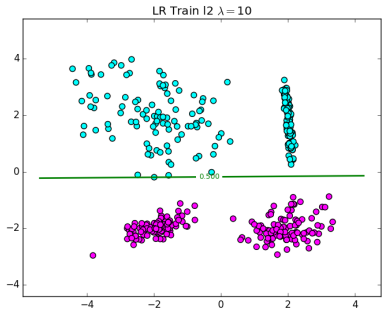
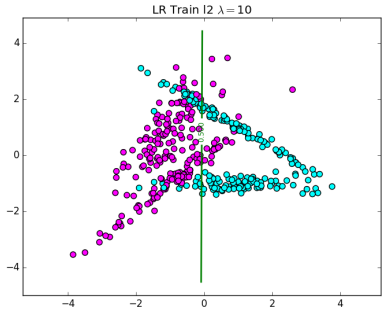


Figure 4: $L_2, \lambda = 0.01$

2 SVM

We implement the dual form of SVM with slack variables. So we have to solve $\max_a\{-\frac{1}{2}||\sum_i a_i y^{(i)} x^{(i)}||^2 + \sum_i a_i\}$, subject to $a_i \geq 0$, $\sum_i y^{(i)} a_i = 0$. This can be equivalently written as the following quadratic program: $min_x \frac{1}{2} x^T P x + q^T x$, subject to $Gx \prec h$, $Ax = b$, where $\prec$ is element-wise inequality. Setting $P_{ij} = y^{(i)} y^{(j)} x^{(i)T} x^{(j)}$, $q_i = -1$, $G = diag(-1)$, $h = 0$, $A_{0i} = y^{(i)}$, and $b = 0$, solving the quadratic





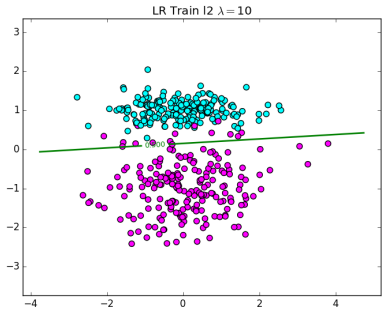
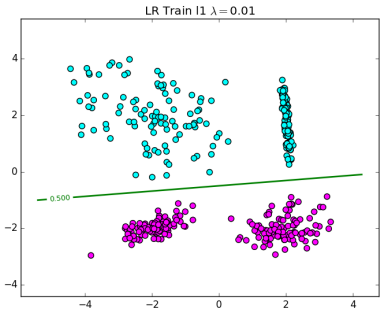
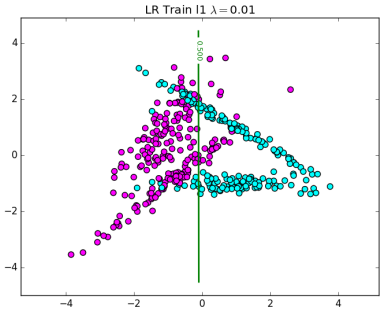


Figure 5: $L_2, \lambda = 10$





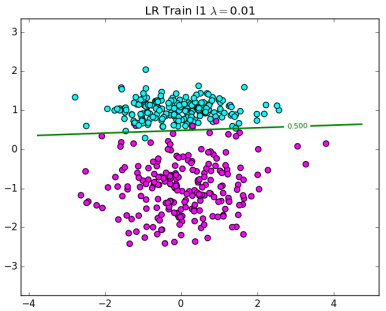


Figure 6: $L_1, \lambda = 0.01$

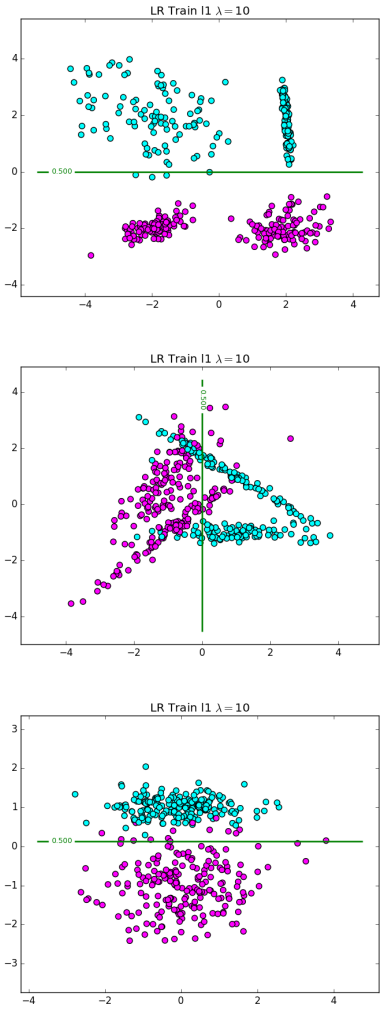


Figure 7: $L_1, \lambda = 10$

problem is exactly equivalent to solving the initial program. Specifically, for $x^0 = (2,2)$, $x^1 = (2,3)$, $x^2 = (0,-1)$, $x^3 = (-3,-2)$ and $y^{(0)} = y^{(1)} = 1$, $y^{(2)} = y^{(3)} = -1$, the program that we have to solve is $\max_a\{-\frac{1}{2}||a_0(2,2)^T + a_1(2,3)^T - a_2(0,-1)^T - a_3(-3,-2)^T + \sum a_i\}$, subject to $a_i \geq 0$ and $a_0 + a_1 - a_2 - a_3 = 0$.

We solve the quadratic program and get that the support vectors are $(2,2)$ and $(0,-1)$ and both their a_i values are 0.154. The other two a_i values are negligibly small.

Now, we go back to the datasets of the first section and classify those. Now we use a C-SVM with $C = 1$, which yields the extra constraint $a_i \leq C$ to the quadratic program. The predictor that we use for input x is $\sum a_i y^{(i)} x^T x^{(i)} + b$. Here $b = \sum_i (y^{(i)} - \sum_j a_j y^{(j)} x^{(i)T} x^{(j)})/N$, where N is the number of support vectors and both sums run only on support vectors. As can be seen in Figure **??**, for the first dataset, we get no error on the training and a 18% error on the validation dataset, for the second dataset we get 17% and 18% errors respectively, and for the third dataset we get 1.25% and 5.5% errors. The large number of errors on both the validation dataset and the training dataset for the second dataset suggests that non-linear features should be used to capture the dataset in a better way.

Now we do the same, but this time using kernels. Linear kernels are exactly what we had in the previous questions. We experiment for $C = 0.01, 0.1, 1, 10, 100$. We note that as C increases, in the first dataset it is easy to fit, so we get zero error for both the training and validation dataset. The only case that we have a bad fit is $C = 1$ where we have a 18% error on the validation dataset. For the second dataset, we note that the optimal value is $C = 0.01$, since for larger value the model overfits and has worse performance on the validation dataset. For this C, the number of errors is 16.5%, which is good considering that for the training dataset it is 17.25%. For the third dataset, the best value of C is again 0.01 because it overfits the least. The errors for the training and validation dataset are 2.75% and 3.5%, while for example for $C = 100$ they are 1.75% and 5.5%, a much worse combination.

Now we repeat the same experiments for Gaussian RBF kernels, where $K(x, x') = e^{-\gamma \cdot ||x - x'||_2^2}$ for various values of γ. We get that for the first dataset, the best value for γ is 1, since for any $C \in \{1, 10, 100\}$ it gives zero errors in both the training and the validation dataset. For the second dataset, the difference

with the linear kernel is evident. For $\gamma = 10$, we get 7.5% error in the validation dataset for $C = 0.1$ (5% for training dataset). This verifies our hypothesis that a linear kernel was inherently unable to model this dataset well enough. For the third dataset, again for $\gamma = 10$ and $C = 10$, we get a 5% error for the validation dataset (0.5% for training dataset), which is slightly better than the linear case.

With these experiments we can conclude that as C increases, the geometric margin $1/||w||$ decreases, i.e the margin lines come closer to the classification line. This can be seen because for example $1/C$ penalizes $||w||$, so increasing C we penalize $||w||$ less, so $||w||$ increases and $1/||w||$ decreases. This decrease will not necessarily always happen, it also depends on the data itself. Furthermore, as C increases, the number of support vectors greatly decreases.

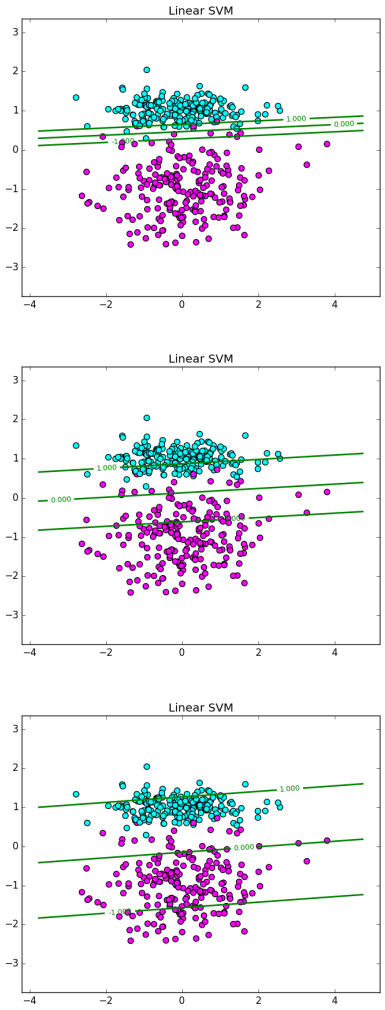


Figure 8: $\lambda = 2^{-10}, 2^{-3}, 2^{0}$

3 SVM with Pegasos

Next, we use Pegasos for a faster SVM implementation, which uses the objective function $\min_w \frac{\lambda}{2}||w||^2 + \frac{1}{n}\sum_i \max\{0, 1 - y_i(w^T x_i)\}$.

First, we implement a variant of Pegasos as a linear classifier. It uses weights w_i that it updates iteratively. We also add the possibility for a bias term b, which we update as follows: If $y_i(w^T x_i) < 1$, then we set $b := b + \eta \cdot y_i$, where η is the current step size. We run this on the third training dataset, and experiment with $\lambda \in \{2^{-10}, 2^{-9}, \ldots, 2\}$. We observe that as λ increases, the margin increases. This can also be seen in Figure 8. This further, validates our claim in the previous section, namely that increasing C decreases the margin. Since λ is inversely proportionate to C, this is true.

Now, we add support for kernels to our Pegasos implementation. In order to make a prediction for a new input x, we can do it exactly like in the usual dual SVM version, but keeping in mind that a_i's have the factor $y^{(i)}$ inside them. So we have $\sum a_i k(x, x^{(i)})$. Pegasos achieves similar sparsity to that of dual SVM. Basically, for fixed $\lambda = 0.02$ and for $\gamma = 0.25, 0.5, 1, 2, 4$ respectively we get error percentages of $3\%, 2.5\%, 1.75\%, 1\%, 1.25\%$. By the arithmetical results it is easy to see that as γ grows, the number of support vectors becomes larger (equivalently: the number of non-zero a_i's). Also, as can be seen in Figure 9, as γ grows, the decision boundary comes closer to the points (becomes tighter).

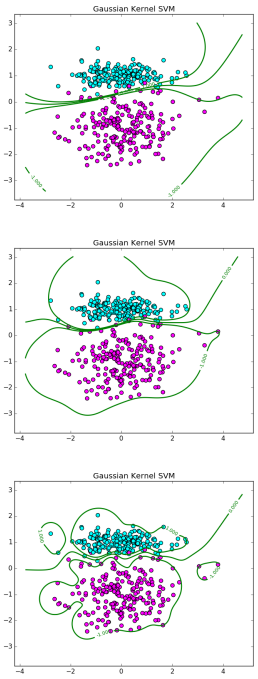


Figure 9: $\gamma = 0.25, 1, 4$

4 Handwriting recognition

In this section we study classifiers that differentiate between the following sets of digits: $\{1\}$ and $\{7\}$, $\{3\}$ and $\{5\}$, $\{4\}$, and $\{9\}$, and $\{0, 2, 4, 6, 8\}$ and $\{1, 3, 5, 7, 9\}$. First, we use the Logistic Regression tool that was used in Section 1. For the first three datasets, the algorithm easily classifies the training and validation datasets with zero error, for example for L_2 regularization and $\lambda = 1$. However, for the last dataset, the best thing that this method gives is a 7.03% error on the validation dataset, for L_2 regularization with $\lambda = 10^{-7}$. For the test datasets, we get error percentages of $1.3\%, 5.3\%, 5.7\%, 9.28\%$.

Looking at images of digits that were not correctly classified, we see, as suspected, that most of them are outliers and so inherently contribute to the error. It is easy to notice that normalization doesn't matter here. The reason is that we are not penalizing the bias, so for scaling or shifting of the data points we can accordingly scale or shift the bias, so that the predictor doesn't change.

Now let's consider a Gaussian RBF SVM classifier. Here, normalization matters, since our kernel is not linear any more. For the first three datasets one gets zero validation error for $\gamma \in \{0.1, 1, 10, 100\}$, and for any $C \in \{0.001, 0.01, 0.1, 1, 10, 100\}$. For the respective test datasets for these three datasets, the error percentages are 0. So in this case it makes sense to use this kind of classifier instead of a linear classifier, because it has better accuracy. However, it is much slower, something that should be taken into account when dealing with big data.

Finally, comparing Pegasos to the Quadratic Programming solution, we note that the accuracies of the two methods are pretty close. Moreover, a serious advantage of Pegasos is that it is more efficient than QP, so it makes much more sense to use it when we have to deal with large amounts of data. Specifically, as we increase the number of samples, we observe a significant slowdown of QP in contrast to Pegasos.