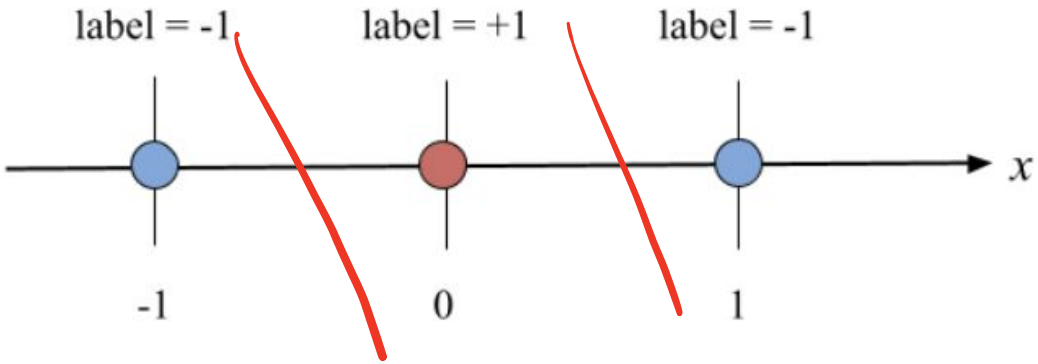
567 HW2 Sols.

Feb 23

Problem 1: SVMs



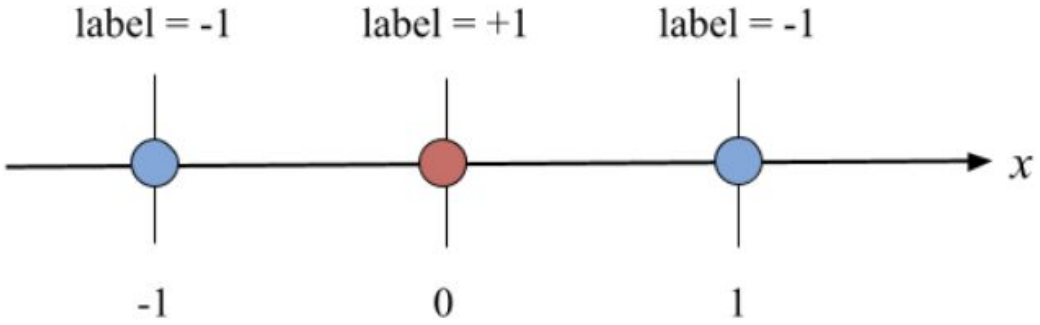
1.1 (2pts) Can these three points in their current one-dimensional feature space be perfectly separated with a linear classifier? Why or why not?

No. A one-dimensional linear model $\text{sgn}(wx + b)$ is equivalent to a simple threshold function of the form

$$f(x) = \begin{cases} +1 & \text{if } x \geq \theta \\ -1 & \text{else} \end{cases} \quad \text{or} \quad f(x) = \begin{cases} -1 & \text{if } x \geq \theta \\ +1 & \text{else} \end{cases}$$

for some threshold $\theta \in \mathbb{R}$. It is thus clear that if we want points x_1 and x_2 to be correctly classified, then x_3 must be incorrectly classified. (Other correct reasoning gets full points as well.) (2 points)

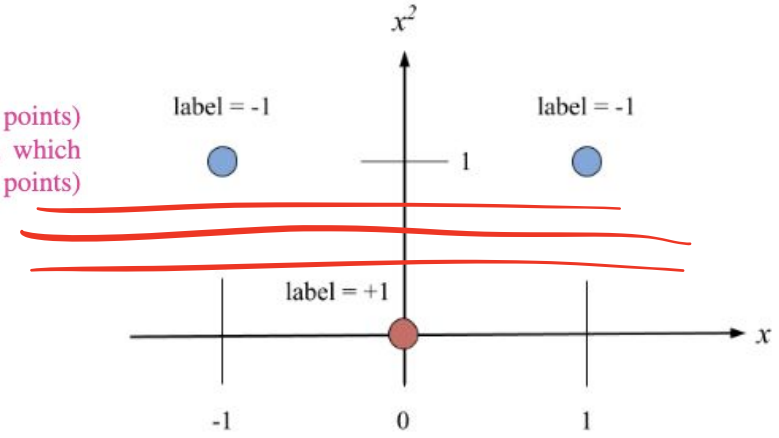
Problem 1: SVMs



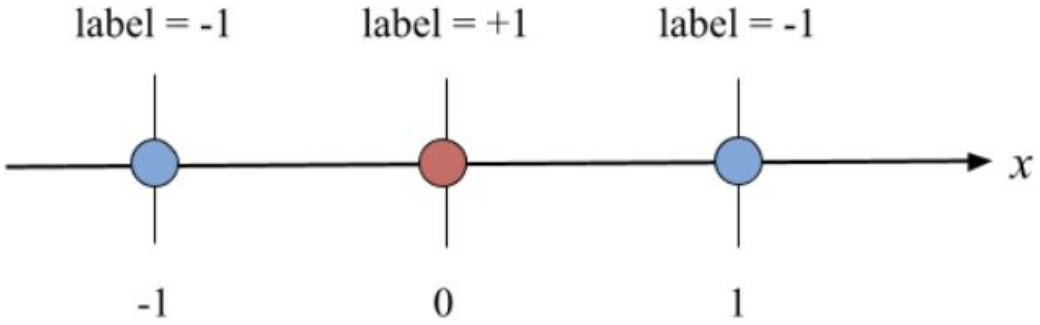
1.2 (3pts) Now we define a simple feature mapping $\phi(x) = [x, x^2]^T$ to transform the three points from one-dimensional to two-dimensional feature space. Plot the transformed points in the new two-dimensional feature space. Is there a linear model $\mathbf{w}^T\mathbf{x} + b$ for some $\mathbf{w} \in \mathbb{R}^2$ and $b \in \mathbb{R}$ that can correctly separate the three points in this new feature space? Why or why not?

See Figure 2 for the plot. (1 points)

Yes. For example, any horizontal line with an intercept between 0 and 1 can correctly separate the data, which corresponds to $\mathbf{w} = (0, 1)$ and any $b \in (-1, 0)$. (It is enough to give one correct example.) (2 points)



Problem 1: SVMs

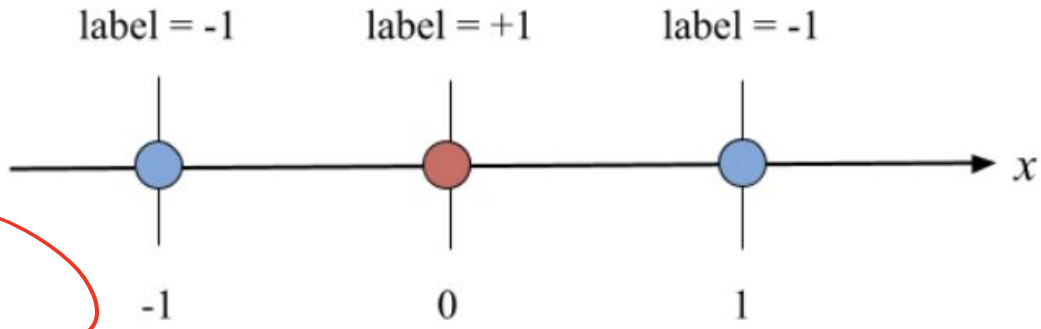


; $(x_1, y_1) = (-1, -1)$, $(x_2, y_2) = (1, -1)$, and $(x_3, y_3) = (0, 1)$

1.3 (2pts) Given the feature mapping $\phi(x) = [x, x^2]^T$, write down the 3×3 kernel/Gram matrix $\mathbf{K}$ for this dataset.

The kernel function is $k(x, x') = \phi(x)^T \phi(x') = xx' + (xx')^2$, so the Gram matrix is $\mathbf{K} = \begin{bmatrix} 2 & 0 & 0 \\ 0 & 2 & 0 \\ 0 & 0 & 0 \end{bmatrix}$.

Problem 1: SVMs



; $(x_1, y_1) = (-1, -1)$, $(x_2, y_2) = (1, -1)$, and $(x_3, y_3) = (0, 1)$

1.4 (4pts) Now write down the primal and dual formulations of SVM for this dataset in the two-dimensional feature space. Note that when the data is separable, we set the hyperparameter C to be $+\infty$ which makes sure that all slack variables (ξ) in the primal formulation have to be 0 (and thus can be removed from the optimization).

Solution:

$$\min_{w,b} \frac{1}{2}\|w\|_2^2$$

$$\text{s.t.} \quad y_n[w^T\phi(x_n) + b] \geq 1 \quad \forall n$$

$$\min \ \frac{1}{2}(w_1^2 + w_2^2)$$

$$\text{s.t.} \quad w_1 - w_2 - b \geq 1$$

$$w_1 + w_2 + b \leq -1$$

$$b \geq 1$$

$$\max_{\alpha} \sum_n \alpha_n - \frac{1}{2} \sum_{nn} y_n y_n \alpha_n \alpha_n k(x_n, x_n)$$

$$\text{s.t} \quad \alpha_n \geq 0 \ \forall n$$

$$\sum_n \alpha_n y_n = 0$$

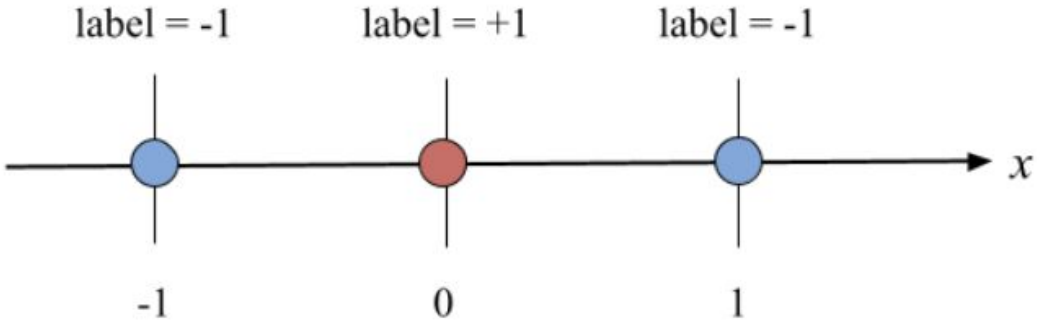
(intentionally left blank)

$$\max_{\alpha_1 \alpha_2 \alpha_3 \geq 0} \alpha_1 + \alpha_2 + \alpha_3 - \alpha_1^2 - \alpha_2^2$$

$$\text{s.t.} \quad \alpha_1 + \alpha_2 = \alpha_3$$

(intentionally left blank)

Problem 1: SVMs



; $(x_1, y_1) = (-1, -1)$, $(x_2, y_2) = (1, -1)$, and $(x_3, y_3) = (0, 1)$

1.5 (5pts) Next, solve the dual formulation exactly (note: while this is not generally feasible, the simple form of this dataset makes it possible). Based on that, calculate the primal solution.

Solution:

$\alpha_1 + \alpha_2 = \alpha_3$

max $2\alpha_1 - \alpha_1^2 + 2\alpha_2 - \alpha_2^2$

$\alpha_1, \alpha_2 \geq 0$

$(x_1, y_1) = (-1, -1)$, $(x_2, y_2) = (1, -1)$, and $(x_3, y_3) = (0, 1)$

$$\alpha_1^* = \alpha_2^* = 1$$

$$\alpha_3^* = 2$$

$$w = \sum_{n=1}^{3} y_n \alpha_n^* \phi(x_n) = (0, -2)$$

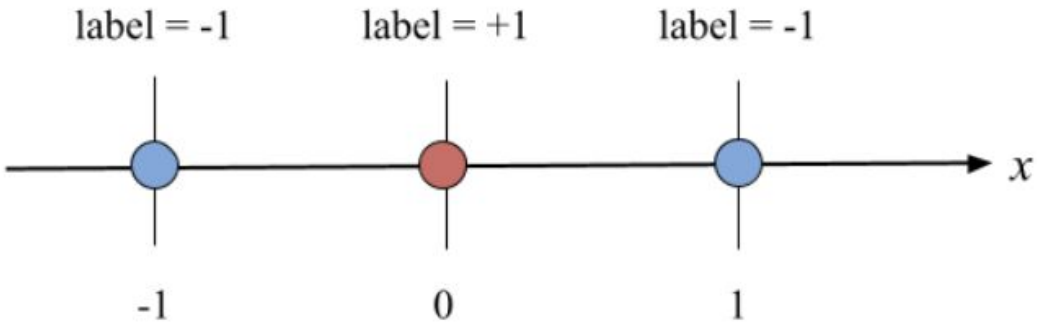
$$b^* = y_1 - w^{*T} \phi(x_1) = 1$$

(intentionally left blank)

$(x_1, y_1) = (-1, -1)$, $(x_2, y_2) = (1, -1)$, and $(x_3, y_3) = (0, 1)$

(intentionally left blank)

Problem 1: SVMs



1.6 (3pts) Plot the decision boundary (which is a line) of the linear model $\mathbf{w}^{*T}\mathbf{x} + b^*$ in the two-dimensional feature space, where $\mathbf{w}^*$ and b^* are the primal solution you got from the previous question. Then circle all support vectors. Finally, plot the corresponding decision boundary in the original one-dimensional space (recall that the decision boundary is just the set of all points x such that $\mathbf{w}^{*T}\phi(x) + b^* = 0$).

The decision boundary for the two-dimensional space is a horizontal line with intercept $1/2$. All three training points are support vectors. (2 points)

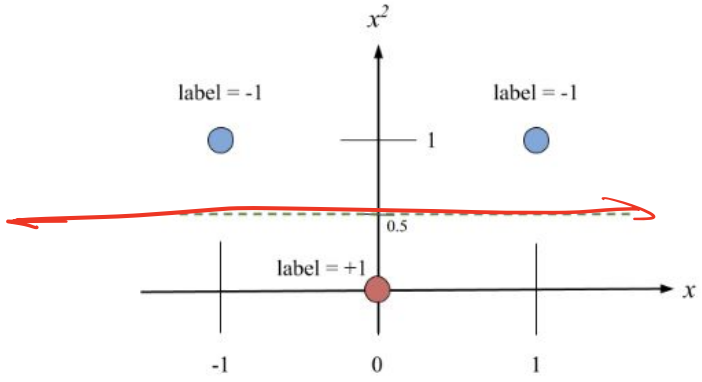


Figure 3: Plot for Q1.6: decision boundary in 2D space

The decision boundary for the one-dimensional space consists of two points $\frac{\sqrt{2}}{2}$ and $-\frac{\sqrt{2}}{2}$ (obtained by solving $\mathbf{w}^{*T}\phi(\mathbf{x}) + b^* = -2x^2 + 1 = 0$). (1 points)

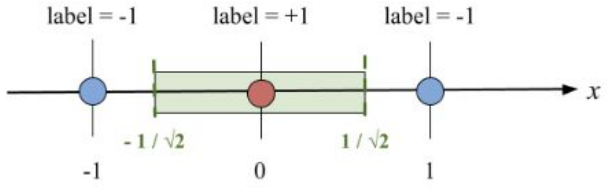


Figure 4: Plot for Q1.6: decision boundary in 1D space

Problem 2: Kernel Composition

Prove that if $k_1, k_2 : \mathbb{R}^d \times \mathbb{R}^d \to \mathbb{R}$ are both kernel functions, then $k(\mathbf{x}, \mathbf{x}') = k_1(\mathbf{x}, \mathbf{x}')k_2(\mathbf{x}, \mathbf{x}')$ is a kernel function too. Specifically, suppose that ϕ_1 and ϕ_2 are the corresponding mappings for k_1 and k_2 respectively. Construct the mapping ϕ that certifies k being a kernel function.

Solution:

$$k(x, x') = \phi(x)^T \phi(x')$$

ϕ_1 M_1

ϕ_2 M_2

$$k(x, x') = k_1(x, x')\, k_2(x, x')$$

$$= \left(\sum_{i=1}^{M_1} \phi_1(x)_i \, \phi_1(x')_i \right) \left(\sum_{j=1}^{M_2} \phi_2(x)_j \, \phi_2(x')_j \right)$$

$$= \sum_i^{M_1} \sum_j^{M_2} \phi_1(x)_i \phi_1(x')_i \phi_2(x)_j \phi_2(x')_j$$

$$= \sum_i^{M_1} \sum_j^{M_2} \phi_1(x)_i \phi_2(x)_i \phi_1(x')_j \phi_2(x')_j$$

$$\left[\phi(x)_{ij} = \phi_1(x)_i \, \phi_2(x)_j \right.$$

(intentionally left blank)

k $\quad\quad$ $k \to (i, j)$

$$K(x, x') = \sum_k \phi(x)_k \phi(x')_k$$

$$K(x, x') = \phi(x)^T \phi(x')$$

□

(intentionally left blank)

Debugging ML for dummies (i.e. *humans*)

Debugging ML is *hard*

- We don’t always know what outputs to expect from our model
- If we get unexpected outputs, there are many possible contributing factors

Some tips:

1. Log values at each step; see if values make sense
    - E.g. maybe your gradients are exploding; or maybe you forgot to normalize your data
    - Visualize *everything*---you never know what you’ll find!
2. Sanity check---make your problem simpler
    - Train your model on the most basic cases---e.g. simpler/smaller dataset, easier environment (for RL), etc
3. Tuning hyperparameters
    - Start with most basic---e.g. size of model, learning rate, regularization coefficients, etc; run simple grid search
4. Ask ChatGPT 🙏

A note on vectorization in Python

- Libraries like NumPy and PyTorch provide vectorized computational support for you
- “for” loops in Python have high overhead, compared to optimized C/Fortran code

$$a \cdot b = \underset{(1 \times n)}{\begin{bmatrix} a_1 & a_2 & a_3 & a_4 & a_5 \end{bmatrix}} \underset{(n \times 1)}{\begin{bmatrix} b_1 \\ b_2 \\ b_3 \\ b_4 \\ b_5 \end{bmatrix}} = \left\{ a_1b_1 + a_2b_2 + a_3b_3 + a_4b_4 + a_5b_5 \right\}$$

Dot Product

example credit: https://www.geeksforgeeks.org/vectorization-in-python/

```
# 8 bytes size int
a = array.array('q')
for i in range(100000):
    a.append(i);

b = array.array('q')
for i in range(100000, 200000):
    b.append(i)

# classic dot product of vectors implementation
tic = time.process_time()
dot = 0.0;

for i in range(len(a)):
      dot += a[i] * b[i]

toc = time.process_time()

print("dot_product = "+ str(dot));
print("Computation time = " + str(1000*(toc - tic )) + "ms")


n_tic = time.process_time()
n_dot_product = numpy.dot(a, b)
n_toc = time.process_time()

print("\nn_dot_product = "+str(n_dot_product))
print("Computation time = "+str(1000*(n_toc - n_tic ))+"ms")
```

Output:

```
dot_product = 833323333350000.0
Computation time = 35.59449199999999ms

n_dot_product = 833323333350000
Computation time = 0.1559900000000225ms
```

Problem 3: Regularization

This problem is a continuation of the linear regression problem from the previous homework (HW1 Problem 5). Let us recall the setup. Given d-dimensional input data $\mathbf{x}_1, \ldots, \mathbf{x}_n \in \mathbb{R}^d$ with real-valued labels $y_1, \ldots, y_n \in \mathbb{R}$, the goal is to find the coefficient vector $\mathbf{w}$ that minimizes the sum of the squared errors. The total squared error of $\mathbf{w}$ can be written as $f(\mathbf{w}) = \sum_{i=1}^{n} f_i(\mathbf{w})$, where $f_i(\mathbf{w}) = (\mathbf{w}^{\mathbf{T}}\mathbf{x_i} - y_i)^2$ denotes the squared error of the ith data point. We will refer to $f(\mathbf{w})$ as the *objective function* for the problem.

3.1 (2pts) We will first setup a baseline, by finding the test error of the linear regression solution $\mathbf{w} = \mathbf{X}^{-1}\mathbf{y}$ without any regularization. This is the closed-form solution for the minimizer of the objective function $f(\mathbf{w})$. (Note the formula is simpler than what we saw in the last homework because now $\mathbf{X}$ is square as $d = n$). Report the training error and test error of this approach, *averaged over 10 trials*. For better interpretability, report the normalized error $\hat{f}(\mathbf{w})$ rather than the value of the objective function $f(\mathbf{w})$, where we define $\hat{f}(\mathbf{w})$ as

$$\hat{f}(\mathbf{w}) = \frac{\| \mathbf{X}\mathbf{w} - \mathbf{y} \|_2}{\| \mathbf{y} \|_2}.$$

```python
def normalized_error(X, y, w):
    return np.linalg.norm(np.matmul(X, w) - y) / np.linalg.norm(y)
```

```python
f_hat_train, f_hat_test = [], []
for t in range(10):
    w_true, X_train, y_train, X_test, y_test = generate_data()

    w_est = np.matmul(np.linalg.inv(X_train), y_train)

    f_hat_train.append(normalized_error(X_train, y_train, w_est))
    f_hat_test.append(normalized_error(X_test, y_test, w_est))

print(f"Normalized train error (linalg soln): {np.mean(f_hat_train):.5e}")
print(f"Normalized test error (linalg soln): {np.mean(f_hat_test):.5e}")
```

Normalized train error (linalg soln): 4.66310e-14 (more generally in range magnitude 1e-13 to 1e-14) (1 point)
Normalized test error (linalg soln): 3.30159e+00 (more generally in range of 7 to 0.5) (1 point)

Problem 3: Regularization

3.2 (7pts) We will now examine ℓ_2 regularization as a means to prevent overfitting. The ℓ_2 regularized objective function is given by the following expression:

$$\sum_{i=1}^{m}(\mathbf{w}^{\mathbf{T}}\mathbf{x_i} - y_i)^2 + \lambda\|\mathbf{w}\|_2^2.$$

As discussed in class, this has a closed-form solution $\mathbf{w} = (\mathbf{X}^{\mathbf{T}}\mathbf{X} + \lambda\mathbf{I})^{-1}\mathbf{X}^{\mathbf{T}}\mathbf{y}$. Using this closed-form solution, present a plot of the normalized training error and normalized test error $\hat{f}(\mathbf{w})$ for $\lambda = \{0.0005, 0.005, 0.05, 0.5, 5, 50, 500\}$. As before, you should average over 10 trials. Discuss the characteristics of your plot, and also compare it to your answer to (**3.1**).

```python
for t in trange(n_trials):
    w_true, X_train, y_train, X_test, y_test = generate_data()

    for l_idx, l_reg in enumerate(l_reg_list):
        w_est = np.matmul(np.linalg.inv(np.matmul(X_train.T, X_train) + l_reg * np.eye(d)),
                          np.matmul(X_train.T, y_train))

        avg_f_hat_train_per_l[l_idx] += normalized_error(X_train, y_train, w_est) / n_trials
        avg_f_hat_test_per_l[l_idx] += normalized_error(X_test, y_test, w_est) / n_trials
```

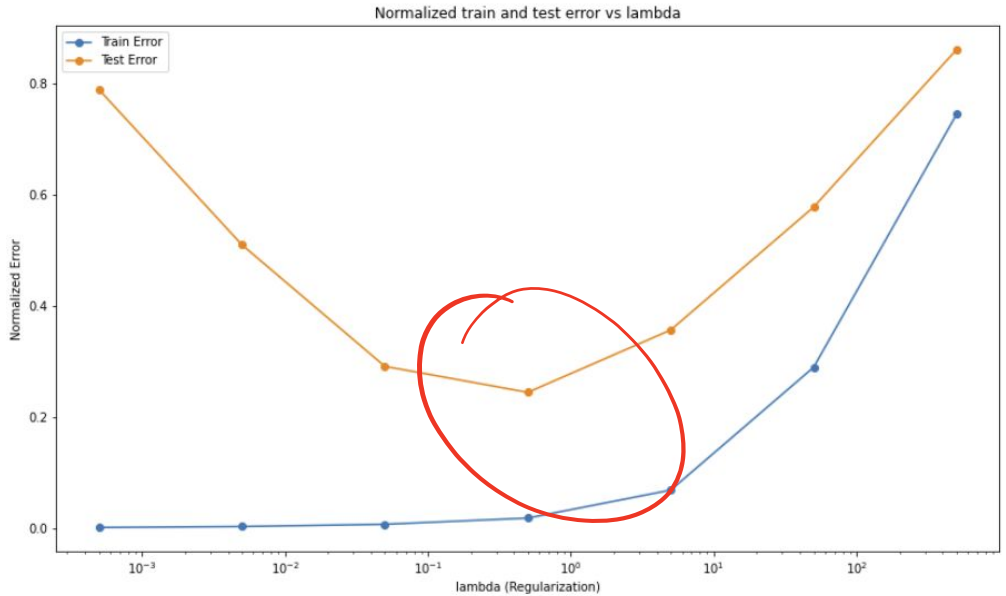


Figure 5: Plot of the normalized training error and normalized test error $\hat{f}(\mathbf{w})$ for varying values of regularization coefficient λ

Rubric:

- Increasing train error plot (2 points)
- U-shaped test error plot (2 points)

λ	0.0005	0.005	0.05	0.5	5	50	500
Normalized Train Error	9.39e-4	2.65e-3	6.65e-3	1.78e-2	6.82e-2	2.89e-1	7.45e-1
Normalized Test Error	0.79	0.51	0.29	0.24	0.36	0.58	0.86

We see a U-shaped curve where small values of lambda lead to over-fitting (train error is disproportionately lower than test error) and large values lead to under-fitting (both train and test errors are high). There is an optimal value where both errors are minimized. (2 points)

Train errors are all higher than the values reported in part (3.1). (0.5 points)

The best test error (for $\lambda = 0.5$) is lower than the test error in part (3.1). (0.5 points)

Problem 3: Regularization

3.3 (7pts) Run stochastic gradient descent (SGD) on the original objective function $f(\mathbf{w})$, with the initial guess of $\mathbf{w}$ set to be the all 0's vector. Run SGD for 1,000,000 iterations for each different choice of the step size, $\{0.00005, 0.0005, 0.005\}$. Report the normalized training error and the normalized test error for each of these three settings, averaged over 10 trials. How does the SGD solution compare with the solutions obtained using ℓ_2 regularization? Note that SGD is minimizing the original objective function, which does *not* have any regularization. In Part (**3.1**) of this problem, we found the *optimal* solution to the original objective function with respect to the training data. How does the training and test error of the SGD solutions compare with those of the solution in (**3.1**)? Can you explain your observations? (It may be helpful to also compute the normalized training and test error corresponding to the true coefficient vector $\mathbf{w}^*$, for comparison.)

```
def sgd(X, y, w, step_size, n_steps):
    for _ in trange(n_steps):
        rand_idx = np.random.randint(0, len(y))
        rand_x, rand_y = X[rand_idx], y[rand_idx]
        residual = rand_x.dot(w) - rand_y
        w -= step_size * 2 * residual * rand_x.reshape(-1, 1)
    return w
```

```python
for t in trange(n_trials, desc="Trail #"):
    w_true, X_train, y_train, X_test, y_test = generate_data()

    for ss_idx, step_size in enumerate(step_size_list):
        w_est = sgd(X_train, y_train, np.zeros((d, 1)), step_size, n_steps= 1000000)

        avg_f_hat_train_per_ss[ss_idx] += normalized_error(X_train, y_train, w_est) / n_trials
        avg_f_hat_test_per_ss[ss_idx] += normalized_error(X_test, y_test, w_est) / n_trials
```

Step Size	5e-5	5e-4	5e-3
Normalized Train Error	1.53e-2	7.02e-3	5.48e-3
Normalized Test Error	0.238	0.237	0.446

Rubric for table: 3 points for all values in the correct range.

The best test errors from the SGD solutions are comparable to the best test errors seen in part (3.2) [this was for the exact (linear algebra) solution to the regularized objective with lambda = 0.5]. Train errors are slightly lower than the train errors seen with the linear algebra solution. (1.5 points)

The best test errors from the SGD solutions are better than the best test errors seen in part (3.1). The train errors are significantly higher than those observed in part (3.1). (1.5 points)

It appears as though SGD (for appropriate step sizes) is implicitly optimizing the regularized objective. [Deduct 0.5 points if the connection to regularization is not made]

Problem 3: Regularization

3.4 (10pts) We will now examine the behavior of SGD in more detail. For step sizes $\{0.00005, 0.005\}$ and 1,000,000 iterations of SGD,

(i) Plot the normalized training error vs. the iteration number. On the plot of training error, draw a line parallel to the x-axis indicating the error $\hat{f}(\mathbf{w}^*)$ of the true model $\mathbf{w}^*$.

(ii) Plot the normalized test error vs. the iteration number. Your code might take a long time to run if you compute the test error after every SGD step—feel free to compute the test error every 100 iterations of SGD to make the plots.

(iii) Plot the ℓ_2 norm of the SGD solution vs. the iteration number.

Comment on the plots. What can you say about the generalization ability of SGD with different step sizes? Does the plot correspond to the intuition that a learning algorithm starts to overfit when the training error becomes too small, i.e. smaller than the noise level of the true model so that the model is fitting the noise in the data? How does the generalization ability of the final solution depend on the ℓ_2 norm of the final solution?

```python
def sgd_w_tracking(X, y, X_te, y_te, w, step_size, n_steps, eval_every=100):
    train_errs, test_errs, w_norms = [], [], []
    print(f"Step size: {step_size}")
    for step_ctr in trange(n_steps, desc="SGD iteration"):
        rand_idx = np.random.randint(0, len(y))
        rand_x, rand_y = X[rand_idx], y[rand_idx]
        residual = rand_x.dot(w) - rand_y
        w -= step_size * 2 * residual * rand_x.reshape(-1, 1)
        train_errs.append(normalized_error(X, y, w))
        if step_ctr % eval_every == 0 or step_ctr == n_steps - 1:
            test_errs.append(normalized_error(X_te, y_te, w))
        w_norms.append(np.linalg.norm(w))

    return w, train_errs, test_errs, w_norms
```

```python
for ss_idx, step_size in enumerate(step_size_list):
    w_est, train_errors, test_errors, w_est_norms = sgd_w_tracking(
        X_train, y_train, X_test, y_test, np.zeros((d, 1)), step_size, n_steps= 1000000, eval_every=100)
```

Plot 3.4(i) Fig 6 The training error quickly goes lower than the train error of the true model w_true. This means that the model starts fitting the noise in the data.

Plot 3.4(ii) Fig 7 The test error keeps decreasing with step size 5e-5. However, with step size 5e-3 the test error first decreases and then starts increasing showing that the model over-fits.

Plot 3.4(iii) Fig 8 The norm of the model weights increases quickly but starts to saturate with step size 5e-5. However, with step size 5e-3, the weight norm keeps increasing.

SGD takes longer to converge but generalizes better with smaller learning rates. With larger learning rates it is prone to over-fitting.

Over-fitting can be seen from the trend of training error when the training error drops below the noise level in the data (the train error with the true weight values). However, this does not seem to be true for all step sizes. For step size 5e-5, both train and test error keep decreasing throughout training.

The larger L2 norm of weight vectors is correlated with the tendency to overfit. Models generalize better when the weight norms are lower.

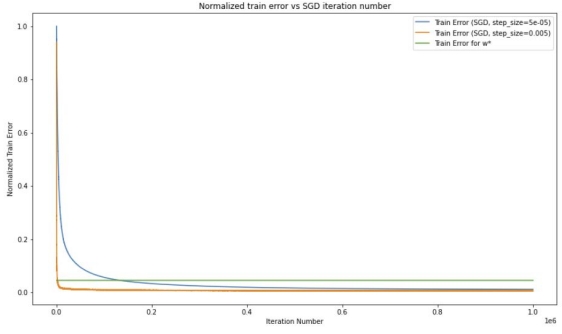


Figure 6: 3.4(i) Plot of the normalized training error of the SGD iterate $\mathbf{w}^{(t)}$ vs. the iteration number t

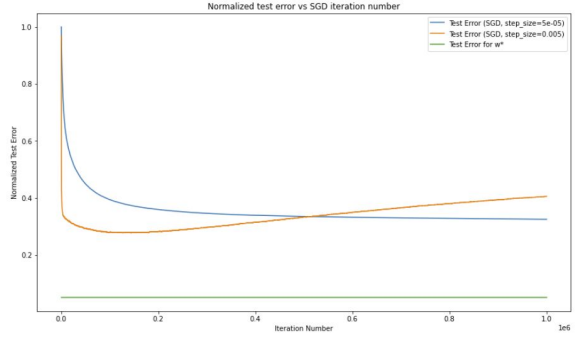


Figure 7: 3.4(ii) Plot of the normalized test error of the SGD iterate $\mathbf{w}^{(t)}$ vs. the iteration number t

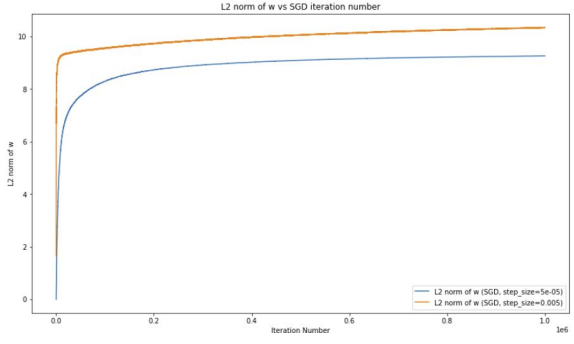


Figure 8: 3.4(iii) Plot of the ℓ_2 norm of the SGD iterate $\mathbf{w}^{(t)}$ vs. the iteration number t

Problem 3: Regularization

(Bonus) 3.5 (5pts) We will now examine the effect of the starting point on the SGD solution. Fixing the step size at 0.00005 and the maximum number of iterations at 1,000,000, choose the initial point randomly from the d-dimensional sphere with radius $r = \{0, 0.1, 0.5, 1, 10, 20, 30\}$ (to do this random initialization, you can sample from the standard Gaussian $N(0, \mathbf{I})$, and then renormalize the sampled point to have ℓ_2 norm r). Plot the average normalized training error and the average normalized test error over 10 trials vs r. Comment on the results, in relation to the results from part (**3.2**) where you explored different ℓ_2 regularization coefficients. Can you provide an explanation for the behavior seen in this plot?

```
for t in trange(n_trials, desc="Trail #"):
    w_true, X_train, y_train, X_test, y_test = generate_data()

    for rad_idx, radius in enumerate(r_list):
        if radius == 0:
            w_start = np.zeros((d, 1))
        else:
            w_start = np.random.randn(d)
            w_start = radius * w_start.reshape((d, 1)) / np.linalg.norm(w_start)
        w_est = sgd(X_train, y_train, w_start, step_size=5e-5, n_steps= 1000000)

        avg_f_hat_train_per_r[rad_idx] += normalized_error(X_train, y_train, w_est) / n_trials
        avg_f_hat_test_per_r[rad_idx] += normalized_error(X_test, y_test, w_est) / n_trials
        w_norm_per_r[rad_idx] += np.linalg.norm(w_est) / n_trials
```

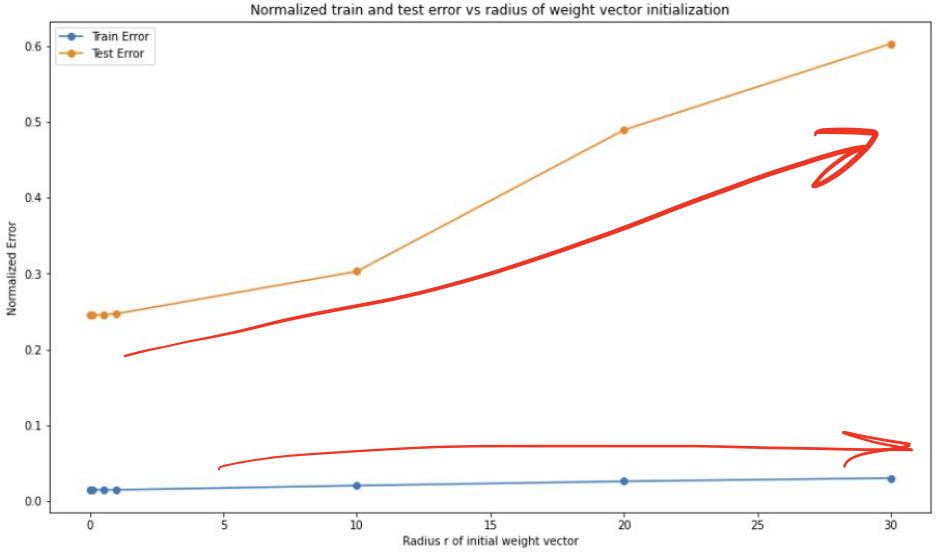


Figure 9: 3.5 Plot of the average normalized training error and the average normalized test error over 10 trials vs r

Rubrics for figure:

- Flat trend line for train error (1 points)
- Matching (increasing) trend line for test error (1 points)

We see that the train error is comparable for small initial radii and increases slightly with higher radius. However, test error for smaller radii 0, 0.1, 0.5, 1 are much lower than test errors for larger radii. (1 point)

The larger initial radius means that the final solutions will also have larger norms. Whereas similar to observations in part 3.4, smaller initial radius means that final solutions also have a smaller norm and the model generalizes better. (1 point)

The best SGD test errors for small initial weight norm are comparable to performing L2 regularization with the optimal value of λ. This is an indicator that SGD on the original objective with a small initialization norm implicitly performs L2 regularization. [Deduct 1 point for any answer that does not link low initial radius to implicit regularization]

Problem 4: Logistic Regression

In this problem we will consider a simple binary classification task. We are given d-dimensional input data $\mathbf{x}_1, \cdots, \mathbf{x}_n \in \mathbb{R}^d$ along with labels $y_1, \cdots, y_n \in \{-1, +1\}$. Our goal is to learn a linear classifier $\text{sign}(\mathbf{w}^T\mathbf{x})$ to classify the datapoints $\mathbf{x}$. We will find $\mathbf{w}$ by minimizing the logistic loss. The total logistic loss for any $\mathbf{w}$ can be written as $f(\mathbf{w}) = \frac{1}{n}\sum_{i=1}^{n} f_i(\mathbf{w})$, where $f_i(\mathbf{w}) = \log\left(1 + \exp(-y_i\mathbf{w}^\mathbf{T}\mathbf{x_i})\right)$ denotes the logistic loss of the ith data point $(\mathbf{x}_i, y_i)$. We will refer to $f(\mathbf{w})$ as the *objective function* for the problem.

4.1 (7pts) As the algorithm proceeds, compute the value of the objective function on the train and test data at each iteration. For this part, you will need to fill in `sigmoid` and `sgd` functions in the skeleton code. For now, you only need to compute and return the `tr_obj_vals` (training objective values) and `obj_vals` (test objective values). Run the following lines to plot the objective function value on the training data vs. the iteration number for all 3 step sizes. On the same graph, you will plot the objective function value on the test data vs. the iteration number for all 3 step sizes. (The deliverable is a single graph with 6 lines and a suitable legend). How do the objective function values on the train and test data relate with each other for different step sizes? Comment in 3-4 sentences.

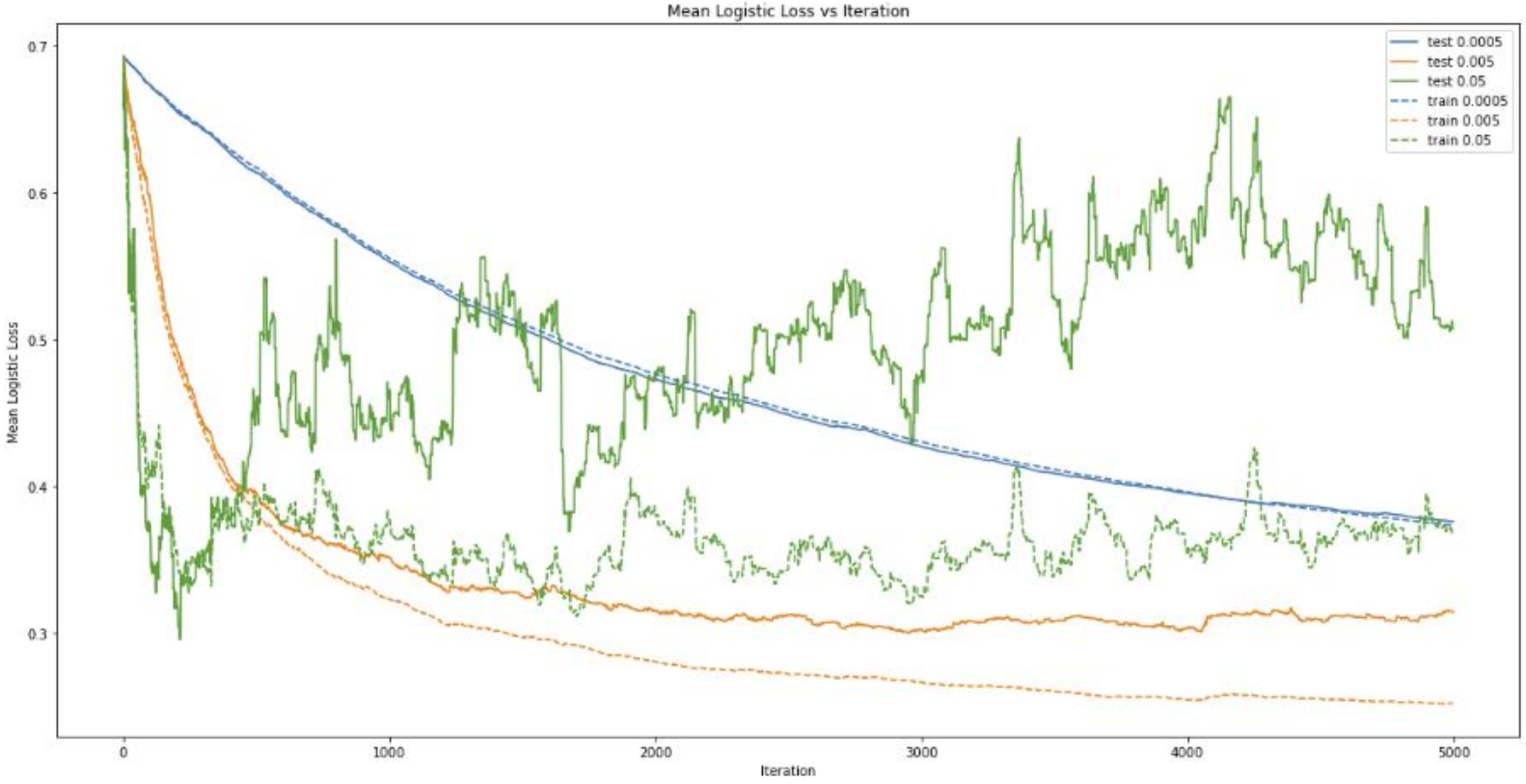
```python
def sigmoid(z):
    return 1. / (1. + np.exp(-z))
```

```python
def sgd(w, X, y, X_te, y_te, step_sz, steps):
    train_objective_vals = []
    objective_vals = []
    loss01_vals = []
    tr_loss01_vals = []
    for i in range(steps):
        ridx = np.random.randint(0, len(y), 1)
        # ridx = i % len(y)
        x_i = X[ridx].ravel()
        y_i = y[ridx]
        w += step_sz * sigmoid(-y_i * w.dot(x_i)) * y_i * x_i
        train_objective_vals.append(np.mean(-np.log(sigmoid(y * X.dot(w)))))
        objective_vals.append(np.mean(-np.log(sigmoid(y_te * X_te.dot(w)))))
        loss01_vals.append(1 - np.mean(sigmoid(y_te * X_te.dot(w)) > 0.5))
        tr_loss01_vals.append(1 - np.mean(sigmoid(y * X.dot(w)) > 0.5))
    return train_objective_vals, objective_vals, loss01_vals, tr_loss01_vals
```

With step size 0.0005 the test error initially decreases and then increases rapidly. The train error also stops decreasing and then has a slight upward trend i.e. the model does not converge.

With step size 0.005, the model has the lowest train logistic error but we observe over-fitting i.e. test error stops decreasing while train error continues to decrease.

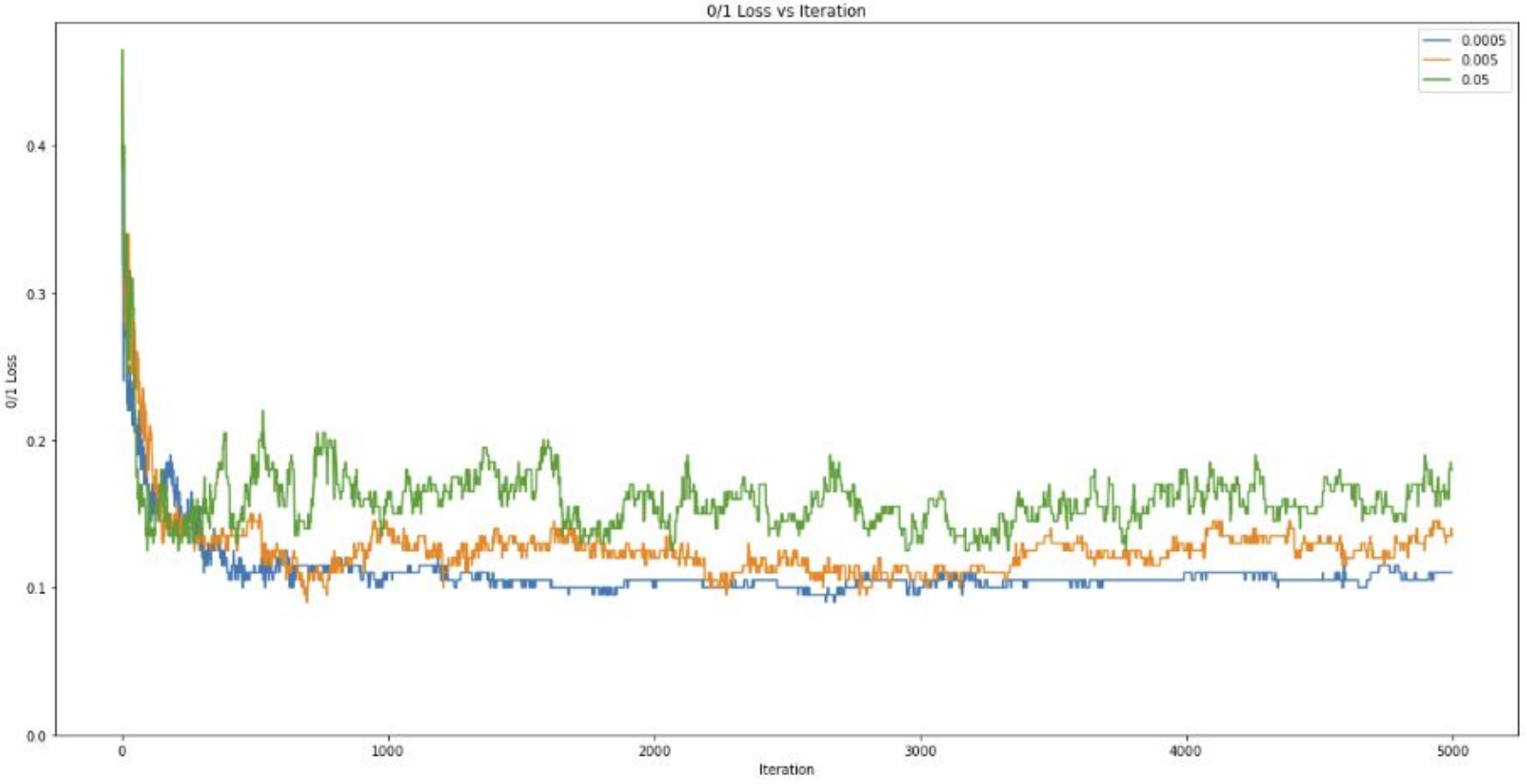
With step size 0.05 both train and test errors continue to decreases as the model converges in a stable manner.



Problem 4: Logistic Regression

In this problem we will consider a simple binary classification task. We are given d-dimensional input data $\mathbf{x}_1, \cdots, \mathbf{x}_n \in \mathbb{R}^d$ along with labels $y_1, \cdots, y_n \in \{-1, +1\}$. Our goal is to learn a linear classifier $\text{sign}(\mathbf{w}^T\mathbf{x})$ to classify the datapoints $\mathbf{x}$. We will find $\mathbf{w}$ by minimizing the logistic loss. The total logistic loss for any $\mathbf{w}$ can be written as $f(\mathbf{w}) = \frac{1}{n}\sum_{i=1}^{n} f_i(\mathbf{w})$, where $f_i(\mathbf{w}) = \log\left(1 + \exp(-y_i\mathbf{w}^\mathbf{T}\mathbf{x_i})\right)$ denotes the logistic loss of the ith data point $(\mathbf{x}_i, y_i)$. We will refer to $f(\mathbf{w})$ as the *objective function* for the problem.

4.2 (2pts) So far in the problem, we've minimized the logistic loss on the data. However, remember from class that our actual goal with binary classification is to minimize the classification error given by the 0/1 loss, and logistic loss is just a surrogate loss function we work with. We will examine the average 0-1 loss on the test data in this part (note that the average 0-1 loss on the test data is just the fraction of test datapoints that are classified incorrectly). In `sgd`, compute and return the average 0-1 loss on the test data (`loss01_vals`), and use the following lines of code to plot these values vs. the iteration number for all 3 step sizes on the same graph. Also report the step size that had the lowest final 0-1 loss on the test set and the corresponding value of the 0-1 loss.



Step size 0.0005 has the lowest 0/1 loss of 0.11.

Problem 4: Logistic Regression

In this problem we will consider a simple binary classification task. We are given d-dimensional input data $\mathbf{x}_1, \cdots, \mathbf{x}_n \in \mathbb{R}^d$ along with labels $y_1, \cdots, y_n \in \{-1, +1\}$. Our goal is to learn a linear classifier $\text{sign}(\mathbf{w}^T\mathbf{x})$ to classify the datapoints $\mathbf{x}$. We will find $\mathbf{w}$ by minimizing the logistic loss. The total logistic loss for any $\mathbf{w}$ can be written as $f(\mathbf{w}) = \frac{1}{n}\sum_{i=1}^{n} f_i(\mathbf{w})$, where $f_i(\mathbf{w}) = \log\left(1 + \exp(-y_i\mathbf{w}^\mathbf{T}\mathbf{x_i})\right)$ denotes the logistic loss of the ith data point $(\mathbf{x}_i, y_i)$. We will refer to $f(\mathbf{w})$ as the *objective function* for the problem.

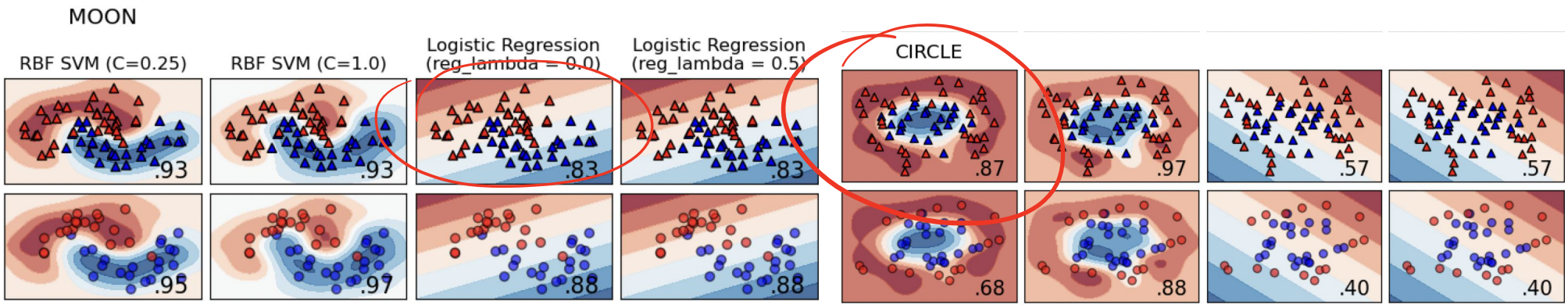
4.3 (2pts) Comment on how well the logistic loss act as a surrogate for the 0-1 loss.

There is a discrepancy when using the surrogate loss to choose the best learning rate vs using the 0/1 loss. The surrogate loss is necessary for better optimization. However, we shouldn't perform model selection based on the surrogate loss values; the task loss (in this case classification error) is the better choice for selecting the best model.

Problem 5: Classifier Comparison

5.1 (2pts) Notice that MOON and CIRCLES are not linearly separable. Do linear classifiers do well on these? How does SVM with RBF kernel do on these? Comment on the difference.

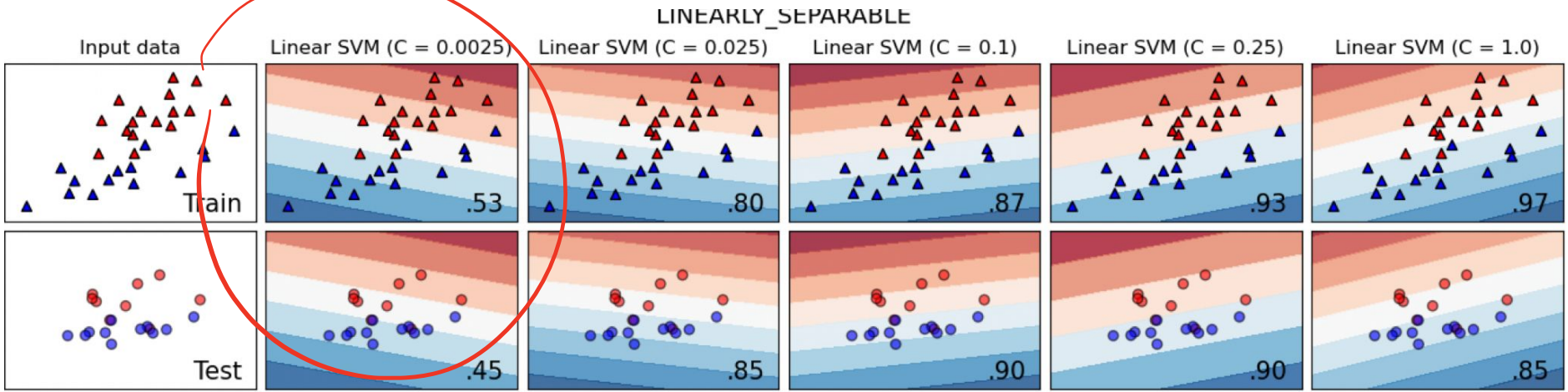
We can see that the linear classifiers struggle to fit the first two datasets MOON and CIRCLES (which are not linearly separable). The RBF kernel allows SVMs to fit all the 3 datasets. Observe that the linear models perform better than an RBF-kernel SVM on the last LINEARLY_SEPARABLE dataset.



Problem 5: Classifier Comparison

5.2 (2pts) Try various values of the penalty term C for the SVM with linear kernel. On the LINEARLY_SEPARABLE dataset, how does the train and test set accuracy relate to C? On the LINEARLY_SEPARABLE dataset, how does the decision boundary change?

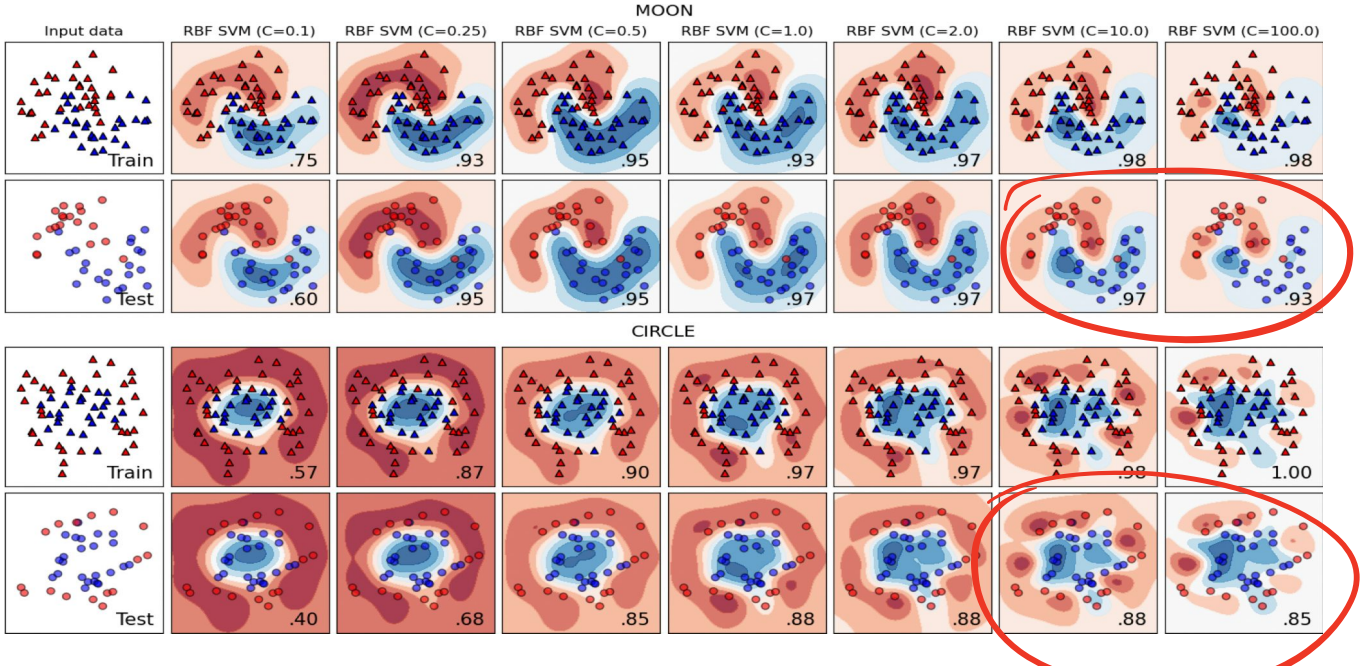
The penalty term describes how many training errors are acceptable when fitting the data. For the linearly separable dataset, very small penalty (stronger regularization) learns a worse decision boundary.



Problem 5: Classifier Comparison

5.3 (2pts) Try various values of the penalty term C for the SVM with RBF kernel. How does the train and test set accuracy relate to C? How does the decision boundary change?

The strength of the penalty term determines how well the decision boundary fits the datasets. The fit improves on the MOON and CIRCLE dataset as the penalty term is increased.



Problem 5: Classifier Comparison

5.4 (2pts) Try various values of C for Logistic Regression (Note: C is the inverse regularization strength). Do you see any effect of regularization strength on Logistic Regression? Hint: Under what circumstances do you expect regularization to affect the behavior of a Logistic Regression classifier?

Due to the low dimensionality and since both dimensions are crucial to classification, regularization does not play a big role and effects are negligible.