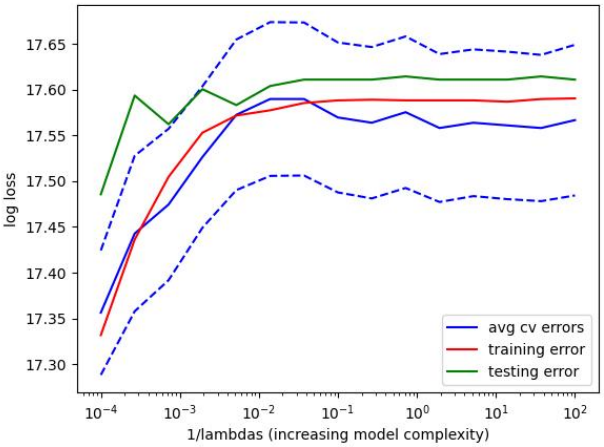
K-fold visualization & Analysis

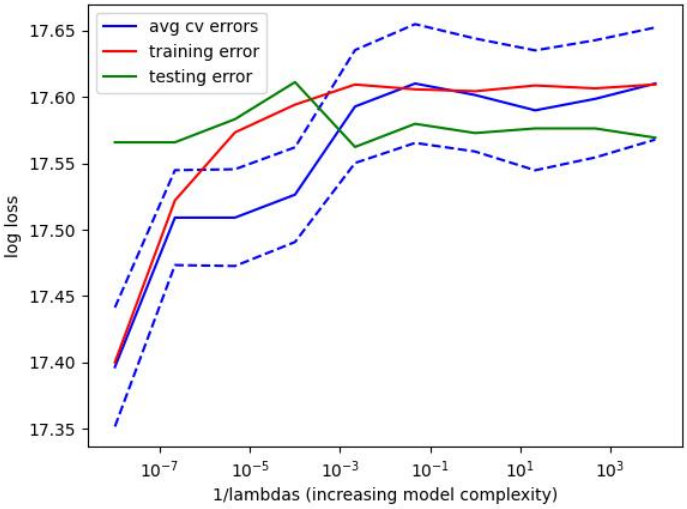
For both models, I performed 5-fold cross validation to tune parameters using 10 parameter options. For each parameter value, I saved all the CV errors and plotted the average in dark blue, and the lines representing that average +/- the standard deviation in a dotted blue line.

Sparse Logistic
model complexity vs log loss



Sparse Logistic Regression:
The correct implementation chose C = 0.00193, and my implementation chose C = 0.000268

Linear SVMs
model complexity vs log loss



Linear SVMs:
The correct implementation chose C = 1000.0, and my implementation chose C = 16.681

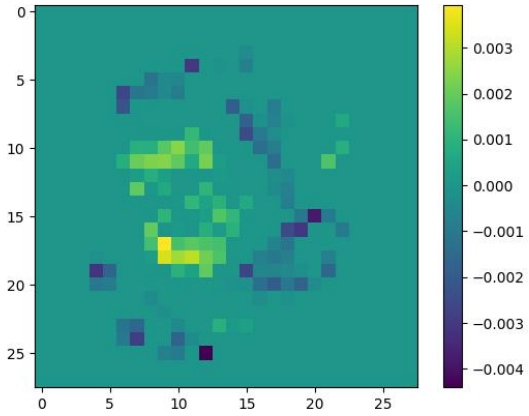
Observations

To start, note that the parameter values C are inversely proportional to lambda (the strength of regularization); i.e. larger values of C correspond to less regularization and higher model complexity wheras smaller values of C correspond to greater regularization and lower model complexity.

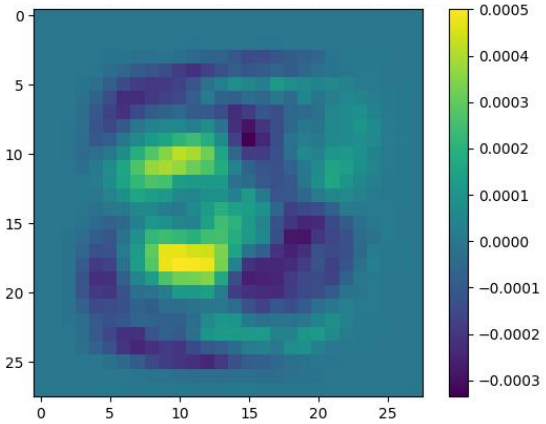
For the Sparse Logistic Regression Model, the correct implementation chose a slightly bigger C value (less regularization), than I chose, but overall, the values were relatively similar.
With linear SVM's my implementation chose a smaller C (more regularization) that the correct implementation.

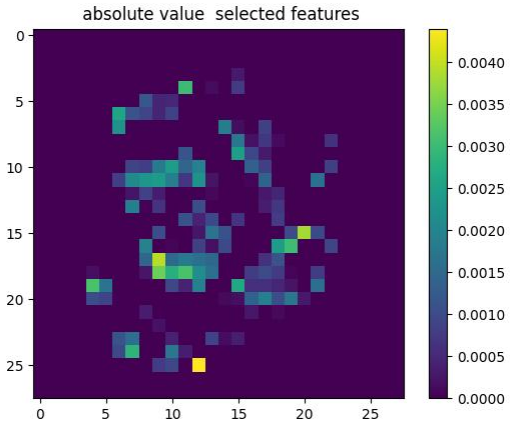
Feature Selection: Comparing my implementation to a correct implementation

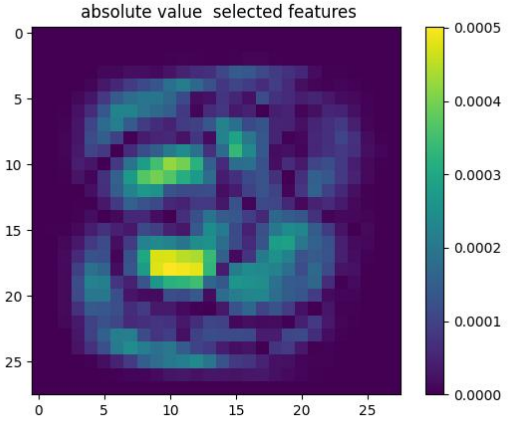
Sparse Logistic Regression Feature Plot



LinearSVM's features plot







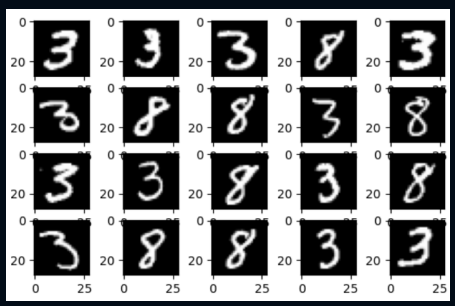
Comparing / Contrasting the features selected by both models:

We can visualize the selected features in the heat map above. We can see that the pixels with the largest absolute value coefficients appeared in the shape of almost a 3 or an 8, surrounding the area that the numbers usually showed up; these pixels were consistently selected across the models which makes sense as they are the most indicative of which number it is. Areas around the border that didn't have information about the numbers displayed in the image, and thus had coefficients in the chosen models closer to 0. The absolute value plots (the second row of plots) really highlight the selected features; features with zero weight are the darkest, and lighter features had larger coefficients in absolute value.

These visualizations are very good at showing sparsity in sparse logistic regression in comparison to the linear SVM's. We see in the spare logistic regression plots, that most features are sent to 0 except for the ones that outline borders of a 3/8. The Linear SVM considered more features as seen in the figure as far more pixels had non zero values. We also see a bigger range in parameters (the largest weights on features occured closer to the centers of the images which makes sense). Moreover, our testing error in sparse logistic regression ended up being higher than with linear SVM's as illustrated in the error plots on the previous page. From this we can infer that images where the 3/8 isn't written in the center of the image/doesn't match the scale of these images would be hard to classify based on the features selected in the plots.

loading, processing, and visualizing the data

```
# LOAD IN THE DATASET
(full_X_train, full_y_train), (full_X_test, full_y_test) = mnist.load_data()

#use a smaller data set first (use half of the training data)
# full_X_train, dummyX, full_y_train, dummyY = train_test_split(full_X_train, full_y_train, test_size = 0.1)

X_train, y_train = [], []
X_test, y_test = [], []

# MODIFY THE DATA TO ONLY INCLUDE 3's and 8'set
for i in range(len(full_X_train)):
    if full_y_train[i] == 3 or full_y_train[i] == 8:
        X_train.append(full_X_train[i])
        y_train.append(full_y_train[i])

for i in range(len(full_X_test)):
    if full_y_test[i] == 3 or full_y_test[i] == 8:
        X_test.append(full_X_test[i])
        y_test.append(full_y_test[i])

X_train, y_train, X_test, y_test = np.array(X_train), np.array(y_train), np.array(X_test), np.array(y_test)

# PLOT THE TRAINING DATA

fig = plt.figure()
num_images = 20 # the number of images to display
grid_dim = math.ceil(num_images**0.5) # the number of images per row

for i in range(num_images):
    plt.subplot(grid_dim,grid_dim,i+1)
    plt.imshow(X_train[i], cmap=plt.get_cmap('gray'))
plt.show()
```



```
# RESHAPE THE DATA
def make2D(X):
    nsamples, nx, ny = X.shape
    d2_X = X.reshape((nsamples,nx*ny))
    return d2_X

X_train = make2D(X_train)
X_test = make2D(X_test)
```

creating the folds

```
def makeKFolds(K, X, y):
    """
    Inputs:
        • K, the number of folds for this kfoldCV
        • X, nxp data matrix
        • y, nx1 response variable matrix

    Splits the data randomly into k folds.

    Returns:
        • folds, a list of the form:
            [(X_fold1, y_fold2), (X_fold2, y_fold2), .. , (X_foldk, y_foldk)]
    """

    folds = []
    for k in range(K, 1, -1):
        # randomly take 1/k of the data for this fold
        X, X_fold, y, y_fold = train_test_split(X, y, test_size = float(1)/float(k))
        folds.append((X_fold, y_fold))
    folds.append((X,y))
    return folds
```

performing kfold cross validation for parameter tuning

```python
def kFoldCV(K, X, y, X_test, y_test, model, param_vals):
    """
    Inputs:
        • K, the number of folds for this kfoldCV
        • X, nxp data matrix
        • y, nx1 response variable matrix
        • model, the model to use kFold CV on (1 parameter)

    Performs k-fold cross validation
    """
    # Keep track of the CV error's of each best lambda
    cv_errors = {}

    # Keep track of all lambdas and the corresponding errors in each fold
    all_cv_errors = defaultdict(list) # a mapping from each param_value to all the error values that it yields during CV
    all_training_errors = defaultdict(list)
    all_testing_errors = defaultdict(list)

     #1. Randomly split data into K-folds
    folds = makeKFolds(K, X, y)

    #progress bar for my sanity
    total = K*len(param_vals)
    current = 0

    #2. Parameter tuning
    for k, (X_val, y_val) in enumerate(folds):
        # the training set is the rest of the folds, merge them into one matrix
        remaining_folds = [folds[i] for i in range(len(folds)) if i!= k]
        X_train = np.concatenate([data[0] for data in remaining_folds], axis = 0)
        y_train = np.concatenate([data[1] for data in remaining_folds], axis = 0)

        # # standardize the data
        # scaler = StandardScaler()
        # X_train = scaler.fit_transform(X_train)
        # X_train = scaler.transform(X_train)
        # X_val = scaler.transform(X_val)
        # X_test = scaler.transform(X_test)

         # fit the model for each parameter value, test error on validation set
        param_star, min_error = 0, float('inf') #keep track of the min error and corresponding param val
        for param_val in param_vals:
            #print progress
            current +=1
            print(str(current/total*100)+"%")

            #fit the model for the parameter value
            model.set_params(C=param_val)
            model.fit(X_train, y_train)
            cv_error = log_loss(model.predict(X_val), y_val, labels = [3,8])

            if cv_error < min_error:
                param_star, min_error = param_val, cv_error
            all_cv_errors[param_val].append(cv_error) #store the errors for each param val to plot

            #compute train error
            train_error = log_loss(model.predict(X_train), y_train,labels = [3,8])
            all_training_errors[param_val].append(train_error)

            #compute test error
            test_error = log_loss(model.predict(X_test), y_test,labels = [3,8])
            all_testing_errors[param_val].append(test_error)

        # store the error of the chosen parameter for this fold
        cv_errors[param_star] = min_error

    #determine the optimal parameter
    final_param_star = min(cv_errors, key=cv_errors.get)

    avg_training_errors = [np.sum(errors)/len(errors) for errors in all_training_errors.values()]
    avg_testing_errors = [np.sum(errors)/len(errors) for errors in all_testing_errors.values()]
    # return the error data
    return all_cv_errors, avg_training_errors, avg_testing_errors, final_param_star
```

creating the error plots

```python
def plot_errors(K, param_vals, all_errors, training_errors, testing_errors):
    # error_dataframe = pd.DataFrame.from_dict(all_errors)
    # print(error_dataframe.head())

    # df.boxplot(by ='day', column =['total_bill'], grid = False)
    fig = plt.figure()

    #plot the average of all cv errors
    average_errors = [np.sum(errors)/len(errors) for errors in all_errors.values()]
    standard_devs = [np.std(errors)/len(errors) for errors in all_errors.values()]
    plt.plot(param_vals, average_errors, label = "avg cv errors", color='blue')

    #plot std lines
    plt.plot(param_vals, [average_errors[i]+standard_devs[i] for i in range(len(average_errors))], color='blue', linestyle='dashed')
    plt.plot(param_vals, [average_errors[i]-standard_devs[i] for i in range(len(average_errors))], color='blue', linestyle='dashed')

    #plot the testing and training errors
    plt.plot(param_vals, training_errors, label = "training error", color='red')
    plt.plot(param_vals, testing_errors, label = "testing error", color='green')


    #add labels
    plt.legend()
    plt.xlabel("1/lambdas (increasing model complexity)")
    plt.xscale("log")
    plt.ylabel("log loss")
    fig.savefig("errorplot.jpg")
```

```python
# ------------MY CV IMPLEMENTATION------------
all_cv_errors, training_errors, testing_errors, my_param_star = kFoldCV(K, X_train, y_train, X_test, y_test, clf, param_vals)
avg_cv_errors = [np.sum(errors)/len(errors) for errors in all_cv_errors.values()]

#plot the errors for my implementation
plot_errors(K, param_vals, all_cv_errors, training_errors, testing_errors)
```

visualizing selected fetures

```python
#====================PRINT RESULTS====================
print("----------------MY IMPLEMENTATION RESULTES--------------------")
print("param star: ", my_param_star)

#determine the features selected by my implementation
clf.set_params(C = my_param_star)
clf.fit(X_train, y_train)

coefficients = np.array((clf.coef_).tolist()[0])
twoD_coefficients = np.reshape(coefficients, (28,28))
fig = plt.figure()
plt.imshow(twoD_coefficients, cmap='viridis')
plt.colorbar()

plt.title(str(clf)[:-2] + " selected features")
plt.savefig(str(clf)[:-2] + " selectedFeatures.jpg")

twoD_coefficients = np.abs(twoD_coefficients)
fig = plt.figure()
plt.imshow(twoD_coefficients, cmap='viridis')
plt.colorbar()

plt.title(" absolute value  selected features")
plt.savefig( "absolute_value_selectedFeatures.jpg")
```

3. *Naive Bayes.* Prove that under certain assumptions, the Naive Bayes Classifier is equivalent to the Nearest Centroid Classifier.

Claim If $n_k = n_{k'} \quad \forall k, k' \in \{1, 2, \dots K\}$ and the distance metric is Euclidean distance,
Then NBC $\iff$ NCC.

1. First note how $p(y=k) = \frac{n_k}{n} = \frac{n_{k'}}{n} = p(y=k') \quad \forall k, k' \in \{1 \dots K\}$
2. Now WTS: $\hat{y}^{NBC} = \hat{y}^{NCC} = \underset{k}{\arg\min} \|x - \mu\|_2$

$$\underset{k}{\operatorname{argmax}}\ p(y=k \mid X) = \underset{k}{\operatorname{argmax}}\ \frac{P(X \mid y=k)\,P(y=k)}{\sum_{k'=1}^{K} P(X \mid y=k')\,P(y=k')}$$

since $P(y=k) = P(y=k')$, we can cancel these

$$= \underset{k}{\operatorname{argmax}}\ \frac{P(X \mid y=k)}{\sum_{k'=1}^{K} P(X \mid y=k')}$$

This value will be the same $\forall k$, so minimizing $P(X \mid y=k)$ is equivalent

$$= \underset{k}{\operatorname{argmax}}\ P(X \mid y=k)$$

$$= \underset{k}{\operatorname{argmax}}\ \frac{\exp\left[-\frac{1}{2}(x-\mu_k)^T \Sigma^{-1} (x-\mu_k)\right]}{\sqrt{(2\pi)^p\ |\Sigma^{-1}|}}$$

Take the log

$$= \underset{k}{\operatorname{argmax}}\ -\frac{1}{2}(x-\mu_k)^T \Sigma^{-1} (x-\mu_k) \underbrace{-\frac{1}{2}\left(p \log(2\pi) + \log|\Sigma^{-1}|\right)}_{\text{same for all } k}$$

$$= \underset{k}{\arg\min}\ \frac{1}{2}(x-\mu_k)^T \Sigma^{-1} (x-\mu_k)$$

minimizing is the same as maximizing it's negative.

$$= \underset{k}{\arg\min}\ \frac{1}{2}(x-\mu_k)^T (x-\mu_k)$$

Because Σ is the same for all k

$$\underset{k}{\operatorname{argmin}}\ p(y=k \mid X) = \underset{k}{\arg\min}\ \|x-\mu_k\|_2^2 = \underset{k}{\arg\min}\ \|x-\mu_k\|_2$$

Therefore we see that our $\boxed{\hat{y}^{NBC} = \underset{k}{\arg\min}\ \|x-\mu_k\|_2 = \hat{y}^{NCC}}$ because the distance metric is Euclidean distance!

$\therefore$ NBC $\iff$ NCC ■

4. *Computations for Logistic Regression.* Derive an algorithm to fit logistic regression using Newton's method. Show that this approach can be written as Iteratively Re-Weighted Least Squares (IRWLS). For extra credit, code up this approach and demonstrate it's convergence.

MLE for Logistic: $\ell(\beta) = y^T \tilde{X}\beta - \log(1+e^{-\tilde{X}\beta})$ ← we should use an iterative method as this has no closed form solution.

- $\nabla(\ell(\beta^t)) = \sum_i (\mu_i - y_i) x = X^T(\mu - y)$
- $\nabla^2(\ell(\beta^t)) = \frac{\partial}{\partial \beta}(X^T(y-\mu))^T = \frac{\partial}{\partial \beta}\sum_i (\nabla_\beta \mu_i) x_i^T = \sum_i \mu_i(1-\mu_i) x_i x_i^T = X^T S X$ where $S = \text{diag}(\mu_i(1-\mu_i))$

If we apply Newton's Method to the logistic regression MLE,

$$\begin{aligned}
\beta^{t+1} &= \beta^t - \nabla^2(\ell(\beta^t))^{-1} \nabla \ell(\beta^t) \\
&= \beta^t - (X^T S_k X)^{-1} X^T(y-\mu) \\
&= (X^T S_k X)^{-1}\left((X^T S_k X)\beta^t + X^T(y-\mu_k)\right) \quad \text{factor out } (X^T S_k X)^{-1} \\
&= (X^T S_k X)^{-1} X^T (S_k X \beta^t + y - \mu_k) \quad \text{factor out } X^T \\
&= (X^T S_k X)^{-1} X^T S_k (X\beta^t + S_k^{-1}(y-\mu_k)) \qquad z_k
\end{aligned}$$

$$\boxed{\beta^{t+1} = (X^T S_k X)^{-1} X^T S_k z_k} \quad \text{where } z_k = X\beta^t + S_k^{-1}(y-\mu_k)$$

Thus we see we can formulate this as an IRLS problem by realizing $\beta^{t+1} = (X^T S_k X)^{-1} X^T S_k z_k$ is a minimizer of $\sum_{i=1}^n S_{ki}(z_{ki} - \beta^T x_i)^2$

For $i = 1 \dots n$, we get $z_{ki} = w_k^T x_i + \frac{y_i - \mu_{ki}}{\mu_{ki}(1-\mu_{ki})}$.

$$z_i = w_0 + w^T x_i + \frac{y_i - \mu_i}{\mu_i(1-\mu_i)}$$

IRLS for Logistic Regression Fitting

1. $w = 0_D$
2. $w_0 = \log\left(\frac{\bar{y}}{1-\bar{y}}\right)$
3. until converged do:
4. $\quad \eta_i = w_0 + w^T x_i$
5. $\quad \mu_i = \text{sigmoid}(\eta_i)$
6. $\quad s_i = \mu_i(1-\mu_i)$
7. $\quad z_i = \eta_i + \frac{y_i - \mu_i}{s_i}$
8. $\quad S = \text{diag}(s_1, s_2, \dots, s_N)$
9. $\quad w = (X^T S X)^{-1} X^T S z$.

5. *SVMs.* Prove that the linear SVM is equivalent to empirical risk minimization with the hinge loss plus ridge penalty.

Empirical Risk minimization:

$$\min_{\beta,\beta_0} \sum_{i=1}^{n} \underbrace{\max\left(0,\, 1-y_i\left(x_i^T\beta+\beta_0\right)\right)}_{\text{Hinge loss}} + \underbrace{\lambda\|\beta\|_2^2}_{\ell_2\text{-regularization}}$$

$$\Leftrightarrow \min_{\beta,\beta_0} \sum_{i=1}^{n} \eta_i + \lambda\|\beta\|_2^2 \qquad \text{where } \eta_i = \max\left(0,\, 1-y_i\left(x_i^T\beta+\beta_0\right)\right)$$

$$\Leftrightarrow \min \eta_i \;:\; \eta_i \geq 0 \text{ and } \eta_i \geq 1-y_i\left(x_i^T\beta+\beta_0\right)$$

$$\Leftrightarrow \min_{\beta,\beta_0} C\sum_{i=1}^{n} \eta_i + \tfrac{1}{2}\|\beta\|_2^2 \qquad \text{since } C,\lambda \text{ are positive constants}$$

Linear SVM problem: