In [1]:
```
"""
    Project 1 for CS135 Summer 2025

    We need to build a classifier for handwritten digits and for
    discerning trouser from dresses using logistic regression.

    Okay, I need to remember what logistic regression is.
    It's a classifier, even though it has regression in the name.
    This is because of historical naming reasons (the statisticians
    got to the naming of it first).


"""
from pathlib import Path

import matplotlib.pyplot as plt
import numpy as np
import pandas as pd
from sklearn.linear_model import LogisticRegression
from sklearn.metrics import log_loss, confusion_matrix
```

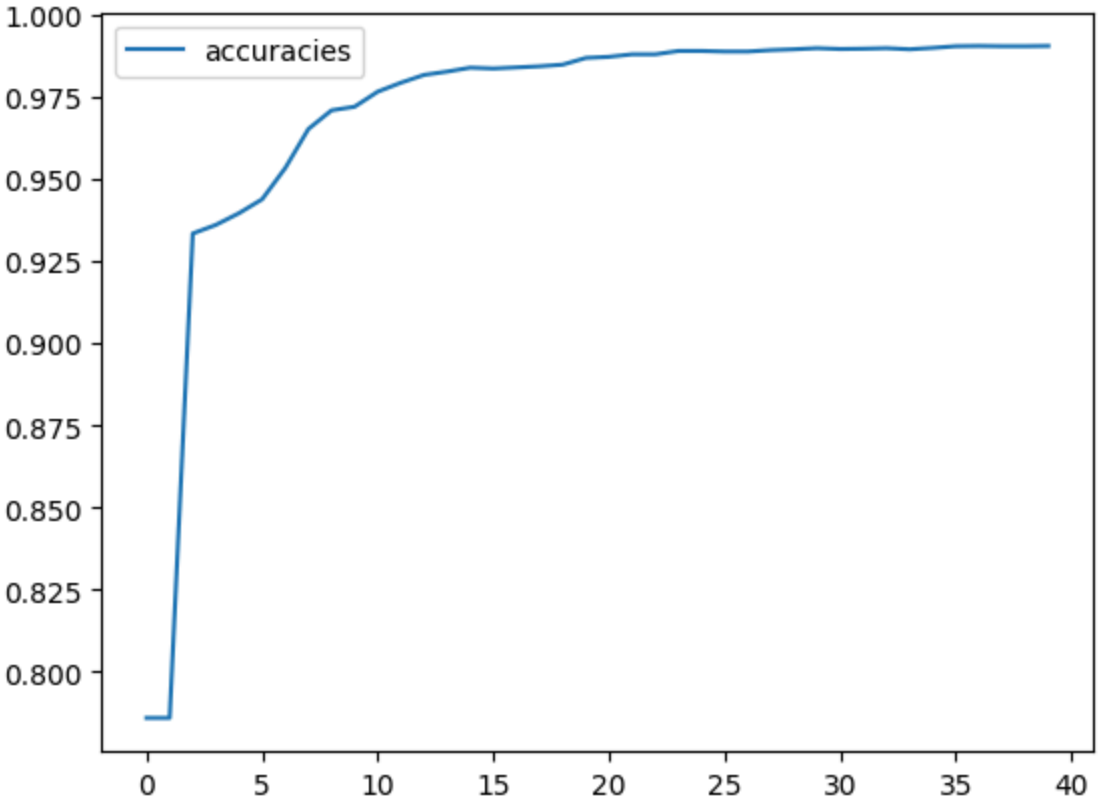
Part 1: 8s vs 9s from MNIST dataset

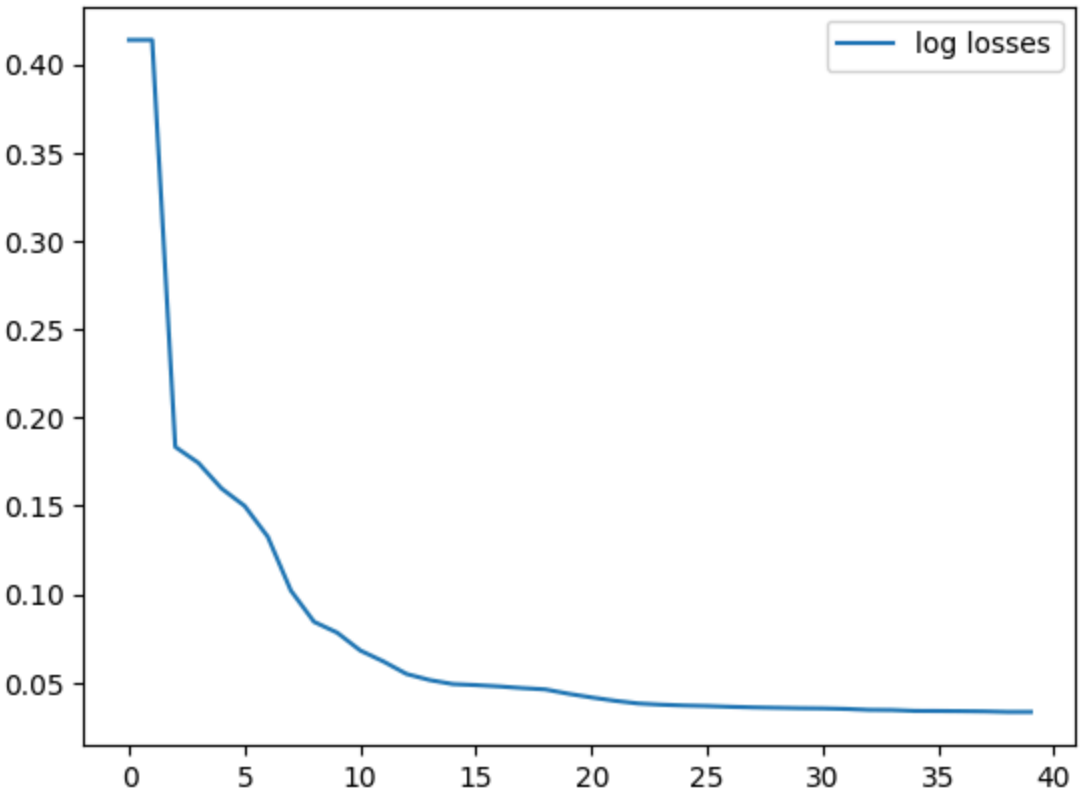
In [2]:
```
# read data
folder = "data_digits_8_vs_9_noisy"
x_test = pd.read_csv(Path(folder) / Path("x_test.csv"))
x_train = pd.read_csv(Path(folder) / Path("x_train.csv"))
y_test = pd.read_csv(Path(folder) / Path("y_test.csv"))
y_train = pd.read_csv(Path(folder) / Path("y_train.csv"))
```

In [3]:
```
import warnings

# to ignore the warnings we're aware will come up
with warnings.catch_warnings():
    warnings.simplefilter("ignore")

    max_iters = list(range(40))
    models     = []
    accuracies = []
    loglosses  = []

    for max_iter in max_iters:

        # set up LogReg classifier object
        logreg = LogisticRegression(max_iter=max_iter)

        # fit data
        model = logreg.fit(x_train, y_train)
        models.append(model)

        # record important stats
        accuracies.append(model.score(x_train, y_train))
```

```
        y_train_pred = model.predict_proba(x_train)
        loglosses.append(log_loss(y_train, y_train_pred))
```

In [4]:
```
plt.figure()
plt.plot(max_iters, accuracies, label='accuracies')
plt.legend()

plt.figure()
plt.plot(max_iters, loglosses, label='log losses')
plt.legend()
plt.show()
```

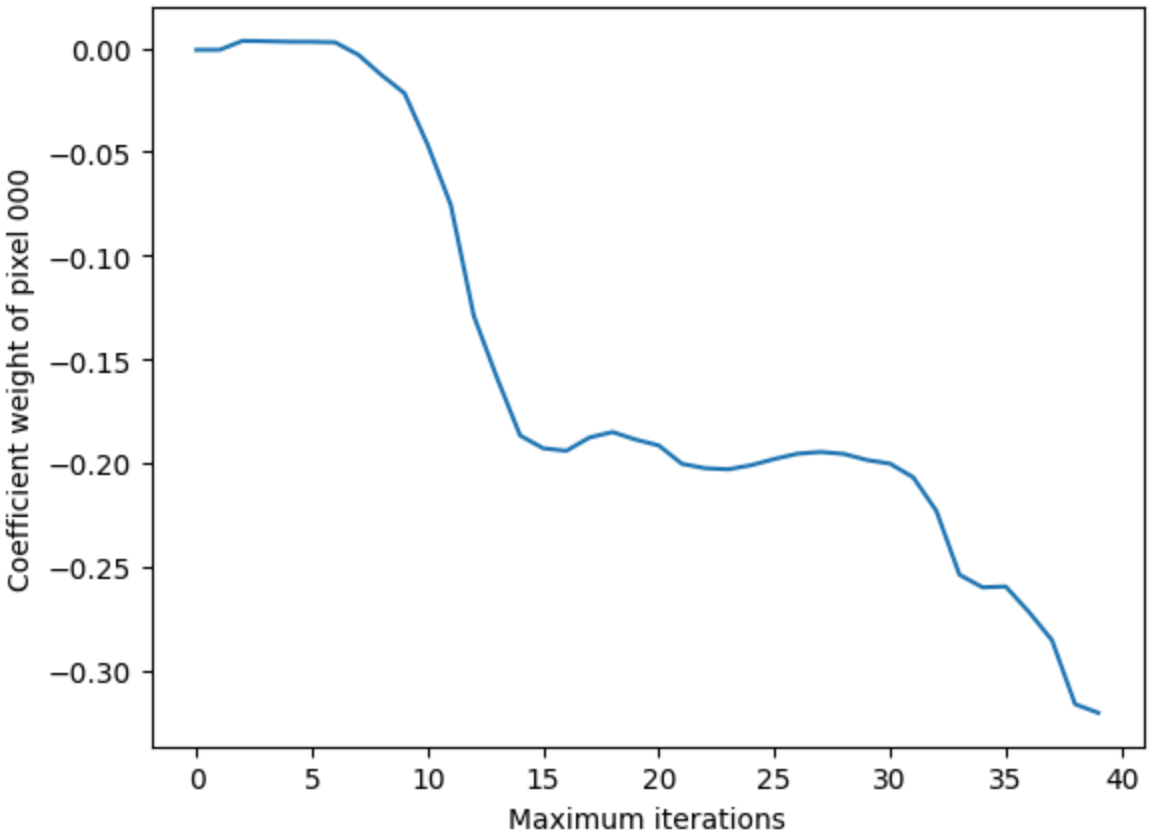




The graphs show that, as the accuracy of the model goes up, the log loss decreases, which behaves exactly as loss should behave if the problem is constructed correctly. The other thing that the graphs show is that more iterations leads to lower loss, which generally makes sense as well. More iterations means the solver can take more steps towards a better solution.

In [5]:
```
pixel000_weights = [model.coef_[0, 0] for model in models]
plt.figure()
plt.plot(max_iters, pixel000_weights)
plt.xlabel("Maximum iterations")
plt.ylabel("Coefficient weight of pixel 000")
```

Out[5]:
```
Text(0, 0.5, 'Coefficient weight of pixel 000')
```



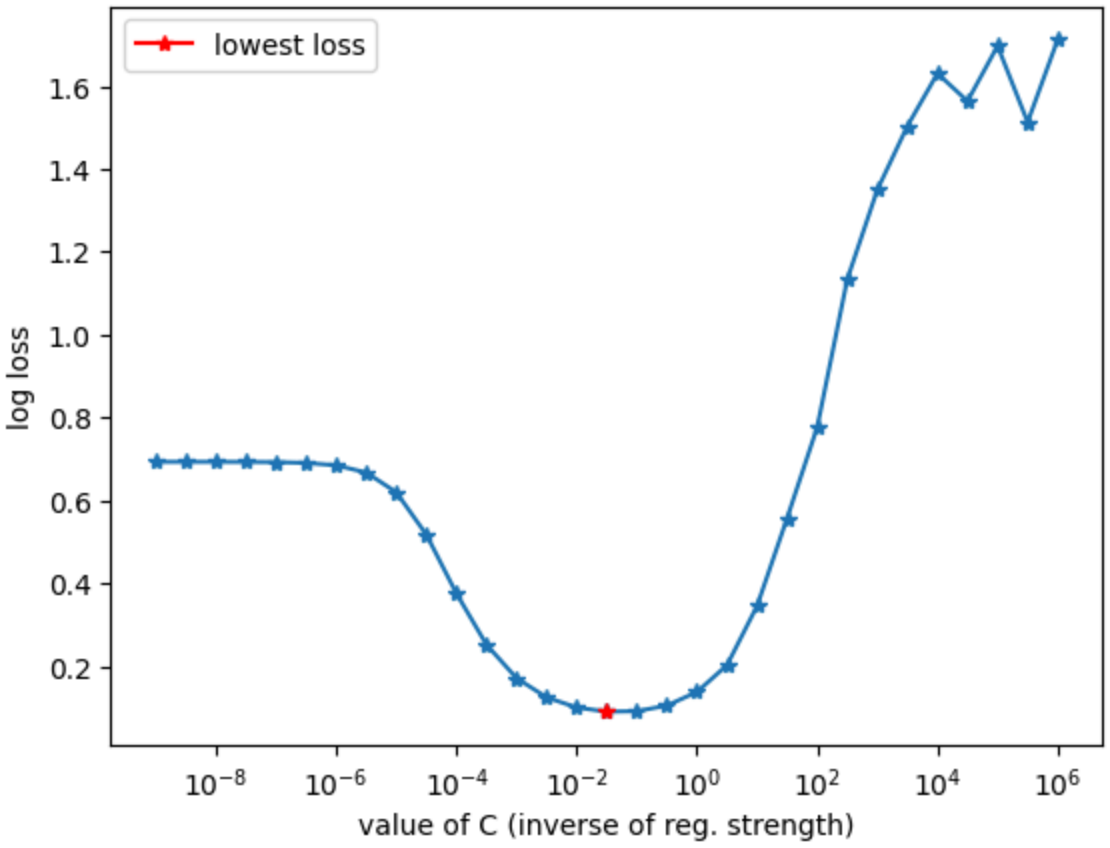
The graph shows that importance of pixel 000 grows as the solver is allowed to run for more iterations (importance being judged as the absolute value of the coefficient assigned to pixel 000).

In [6]:
```
# this time, let max_iters be large enough so we dont't converge, and change the re
models_C = []
log_losses_C = []
C_grid = np.logspace(-9, 6, 31)
for C in C_grid:

    # set up LogReg classifier object
    model_C = LogisticRegression(C=C, max_iter=200)

    # fit data
    model_C.fit(x_train, y_train.to_numpy().ravel())
    models_C.append(model_C)

    y_test_proba = model_C.predict_proba(x_test)
    log_losses_C.append(log_loss(y_test, y_test_proba))
```

In [7]:
```
plt.figure()
plt.semilogx(C_grid, log_losses_C, marker='*')
plt.xlabel('value of C (inverse of reg. strength)')
plt.ylabel('log loss')
best_idx = np.argmin(log_losses_C)
best_loss = log_losses_C[best_idx]
best_C = C_grid[best_idx]
best_model_C = models_C[best_idx]
```

```
plt.plot(best_C, best_loss, marker='*', color='r', label='lowest loss')
plt.legend()

# regularization and loss
best_C, best_loss,

# metrics
y_hat_test = best_model_C.predict(x_test)
accuracy = best_model_C.score(x_test, y_test)
conf_mat = confusion_matrix(y_test, y_hat_test)
print(f"accuracy: {accuracy}")
print(f"--confusion matrix-- \n{conf_mat}")
```

```
accuracy: 0.9662128088754413
--confusion matrix--
[[942  32]
 [ 35 974]]
```



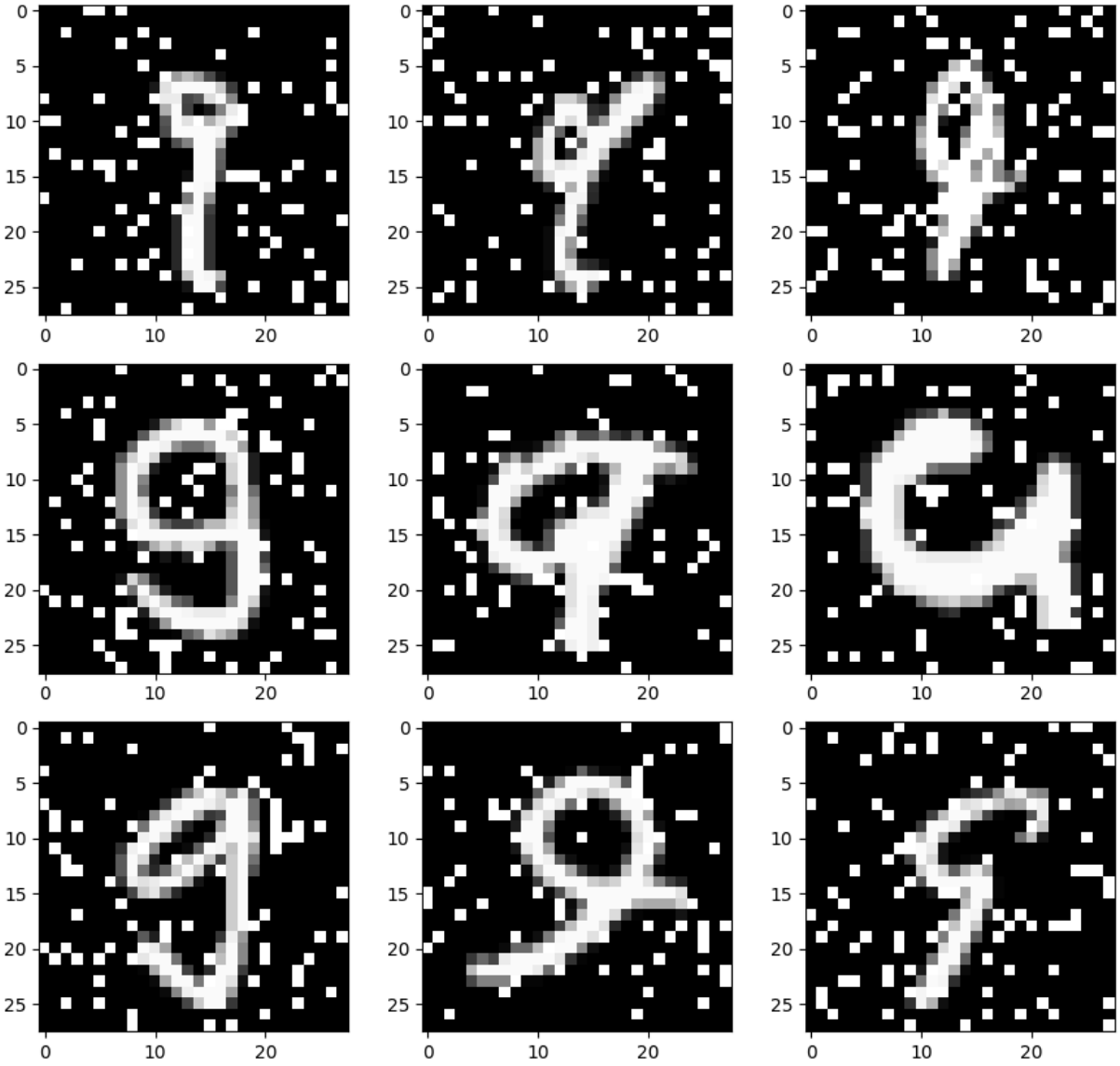
best regularization value is C=0.0316 granting a loss of 0.0900 with an accuracy of 0.966. confusion matrix supports the returned number for accuracy.

In [8]:
```
### 1.4 - plotting misclassified data
rng = np.random.default_rng()
num_images = 9

y_test_np = y_test.to_numpy().ravel()


def get_subset_of_particular_pred_actual(y_hat_test, y_test_np, pred, actual, size)
    idxes = [
```

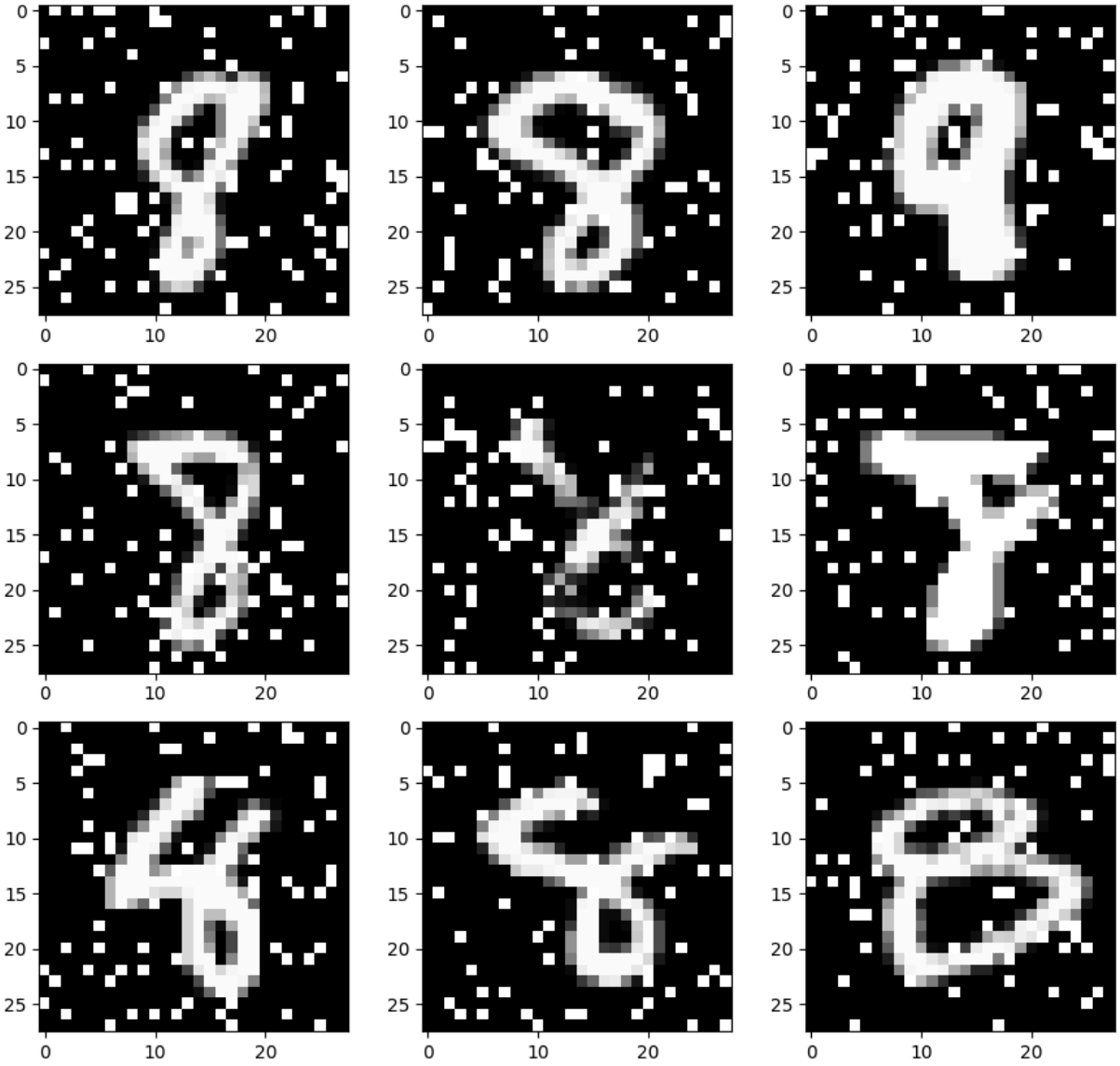
```
        ii for ii, (_pred, _actual)
        in enumerate(zip(y_hat_test, y_test_np))
        if _pred == pred and _actual == actual
    ]
    select_idxes = rng.choice(idxes, size=size, replace=False)
    return select_idxes

# step 1 - grab 9 indices of false negatives
select_fn_idxes = get_subset_of_particular_pred_actual(y_hat_test, y_test_np, pred=
print(select_fn_idxes)

# step 2 - grab 9 indices of false positives
select_fp_idxes = get_subset_of_particular_pred_actual(y_hat_test, y_test_np, pred=
print(select_fp_idxes)


# also personally curious about tp and tn
select_tp_idxes = get_subset_of_particular_pred_actual(y_hat_test, y_test_np, pred=
print(select_tp_idxes)
select_tn_idxes = get_subset_of_particular_pred_actual(y_hat_test, y_test_np, pred=
print(select_tn_idxes)


# step 4 - graph
def plot_9_images(x_test, select_idxes, title=''):

    # step 3 - convert pixel data into images
    fig, axes = plt.subplots(3, 3, figsize=(9, 9))
    for ii, idx in enumerate(select_idxes):
        data = x_test.loc[idx]
        image = data.to_numpy().reshape((28, 28))
        j = ii // 3
        k = ii % 3
        axes[j, k].imshow(image, cmap='grey', vmin=0, vmax=1)
        # plt.imshow(image)


    plt.tight_layout()
    fig.suptitle(title)
    plt.subplots_adjust(top=0.92)  # leave room for suptitle (adjust value as neede



plot_9_images(x_test, select_fn_idxes, title='plots of some false negatives')
plot_9_images(x_test, select_fp_idxes, title='plots of some false positives')
plot_9_images(x_test, select_tp_idxes, title='plots of some true positives')
plot_9_images(x_test, select_tn_idxes, title='plots of some true negatives')
```

```
[  56  958  466   39  905  444  335  407 1490]
[1427  580  174  355 1327  290  998  401  440]
[ 137 1398   55  628  232 1827 1752  153 1431]
[1121 1979 1213 1297  272 1371 1175  953 1344]
```

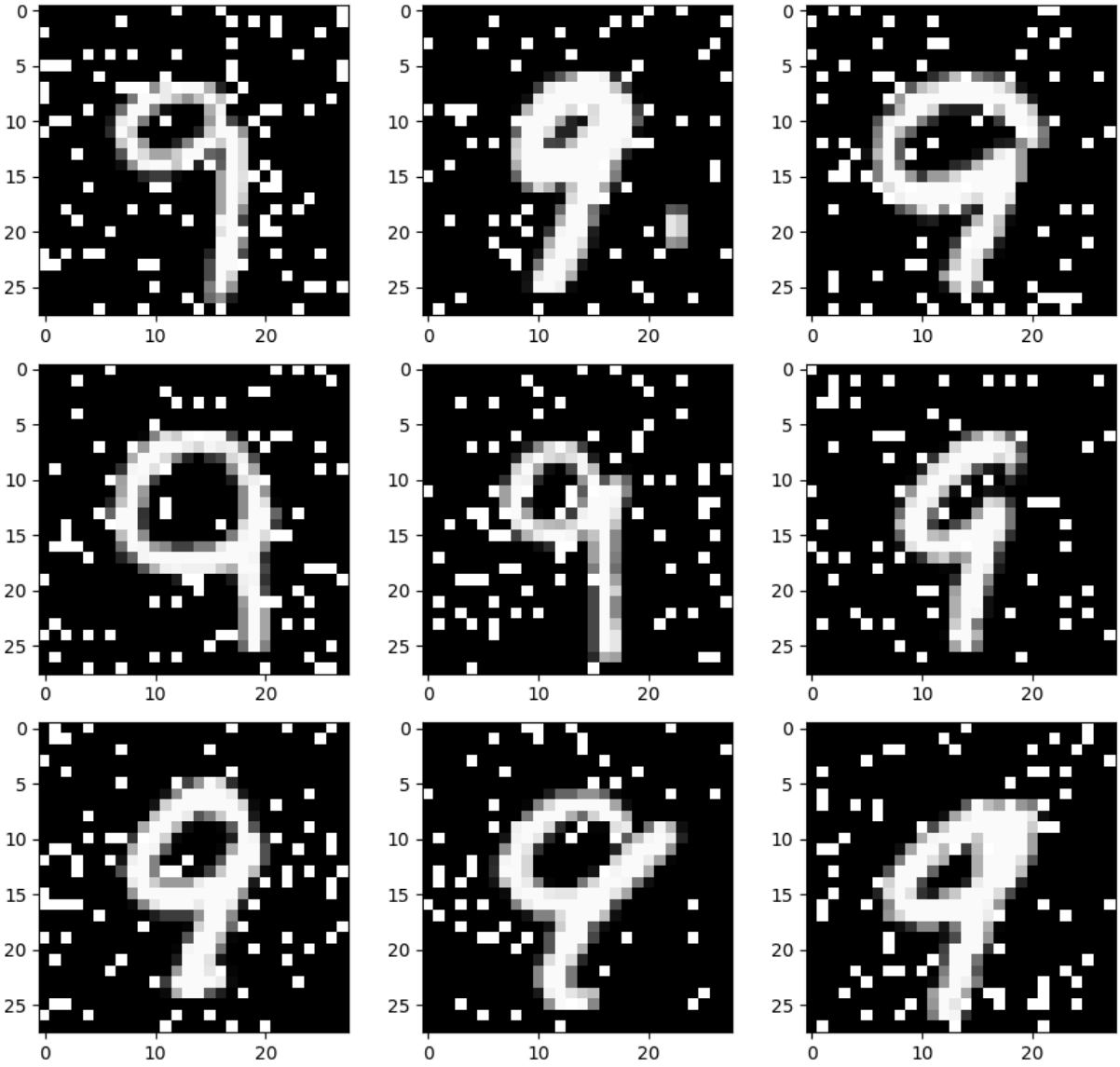
plots of some false negatives

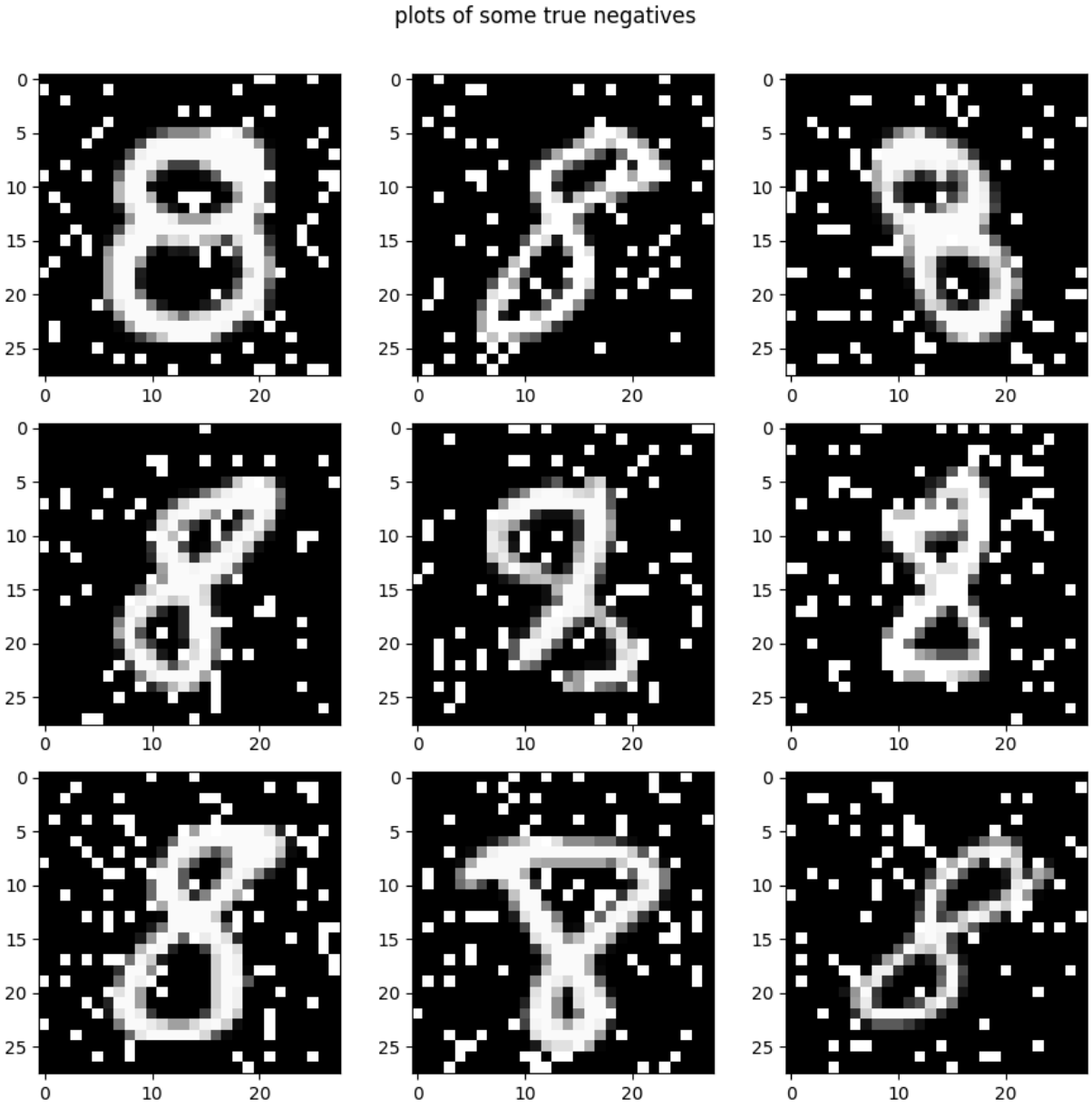


plots of some false positives



plots of some true positives





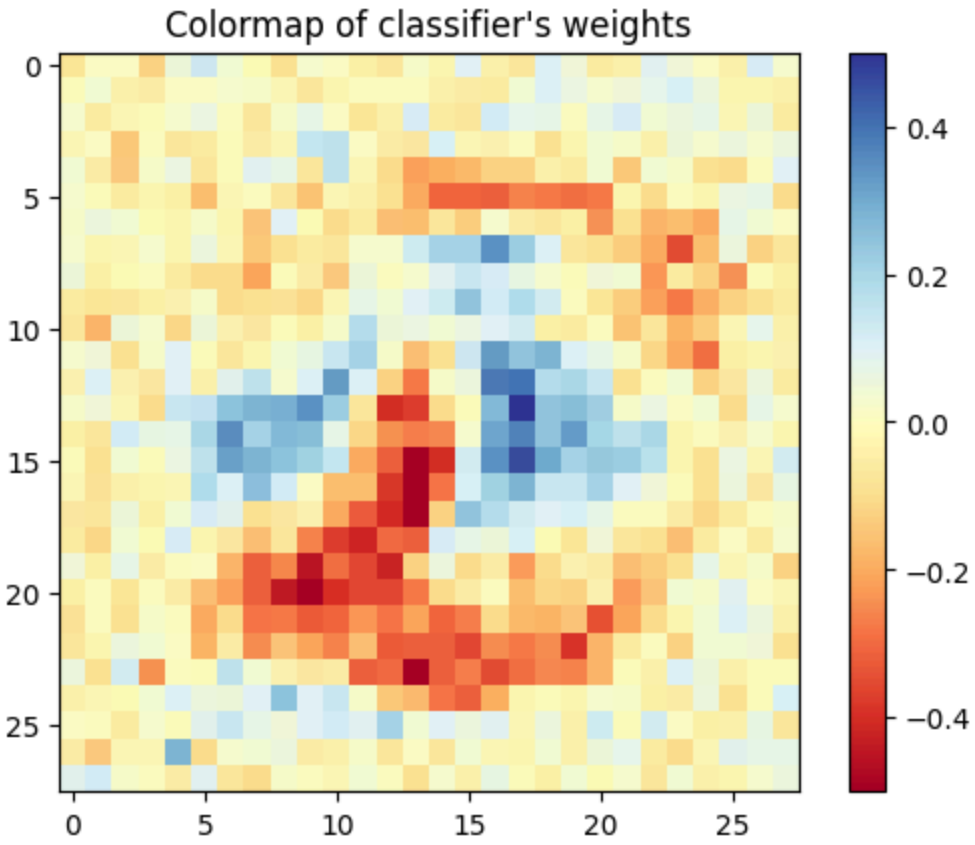
generally, i think the algorithm is more likely to classify a 9 as a 9 when the 9 is written with a straight stem as opposed to a curled one. This makes sense, as curling the stem could confuse the classifier with the curlyness of an 8.

8s get misclassified as a 9 when the the top lobe and bottom lobe balance resembles a 9. generally, this means that the top lobe is larger than the bottom lobe and that the top lobe extend further out to the left than the bottom lobe does. in some cases, the bottom lobe is written so tightly that it almost looks like a straight line

In [9]:
```
best_model_C2 = LogisticRegression(C=best_C, max_iter=200)
best_model_C2.fit(x_train.to_numpy(), y_train.to_numpy().ravel())

coef_image = best_model_C2.coef_.reshape((28, 28))
plt.figure()
plt.imshow(coef_image, cmap='RdYlBu', vmin=-0.5, vmax=0.5)
```

```
plt.colorbar()
plt.title("Colormap of classifier\'s weights")
```

Out[9]: Text(0.5, 1.0, "Colormap of classifier's weights")



there are a large swath of negative coefficients towards the bottom left and top right of the image. The bottom left section makes a lot of sense because generally, a 9 will leave that area empty, whereas an 8 will fill that area. the top right having a decent amount of red pixels is interesting because this might reflect a fact about how people generally initiate the writing of the number 8 vs the number 9. I can imagine the dataset including mostly 9s which were written with the top side of the lobe curving up, whereas 8s might be written with the top side of the top lobe slanting downward from northeast to southwest.

the pixels that the classifier decided on blue may reflect general hotspots where 9 will take room but 8 won't. notice in the plots of accurately classified numbers that the thinnest part of the 8 is often on the same latitude as the thickest part of the 9. the graph suggests that checking for thinness in this area is core to how the classifier decides between an 8 and a 9.

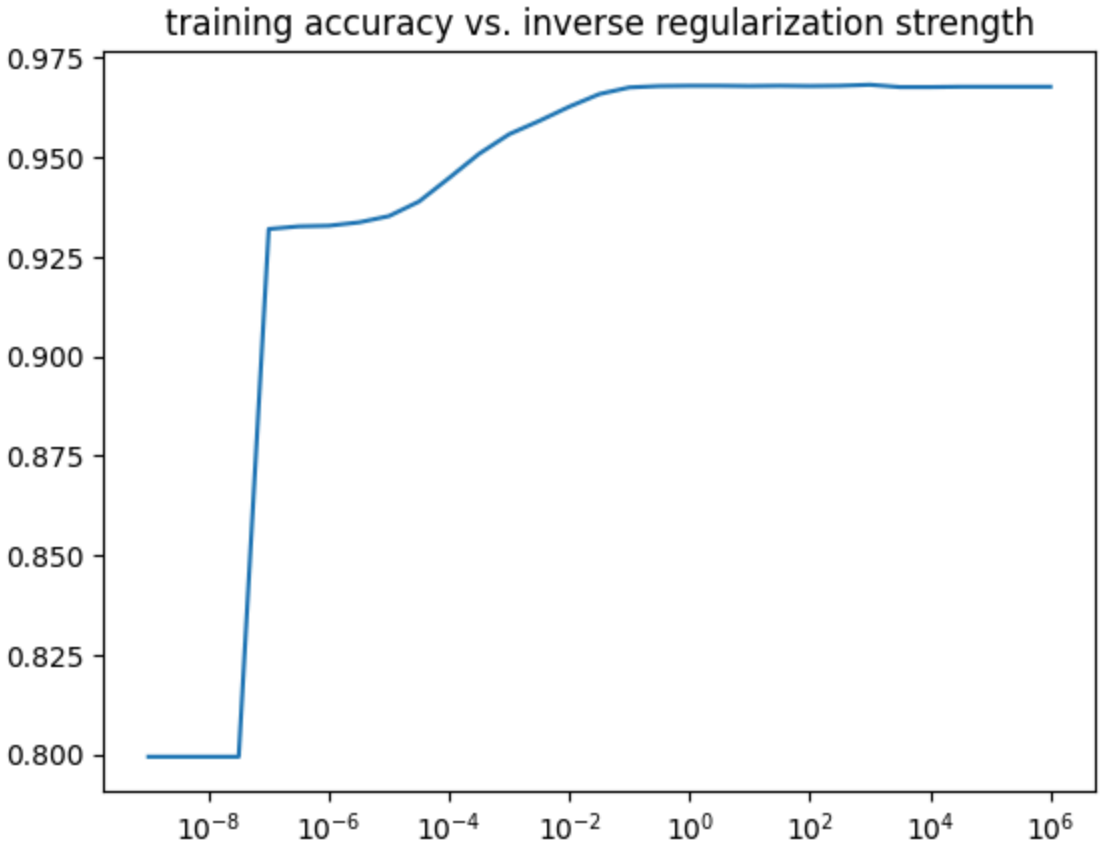
the coefficient map explains some of the inaccurate classifications. based on this classifier, a 9 where the stem is curled up will resemble an 8 because the curled stem takes up space in the area that 9s usually leave empty. a 9 that is tilted or not completely centered is also at risk of not passing the thickness check in the center of the map. for false positive cases, we also see some instances of 8s which pass as 9s due to lopsidedness making the 8 fail the thickness check or the empty bottom left space check.

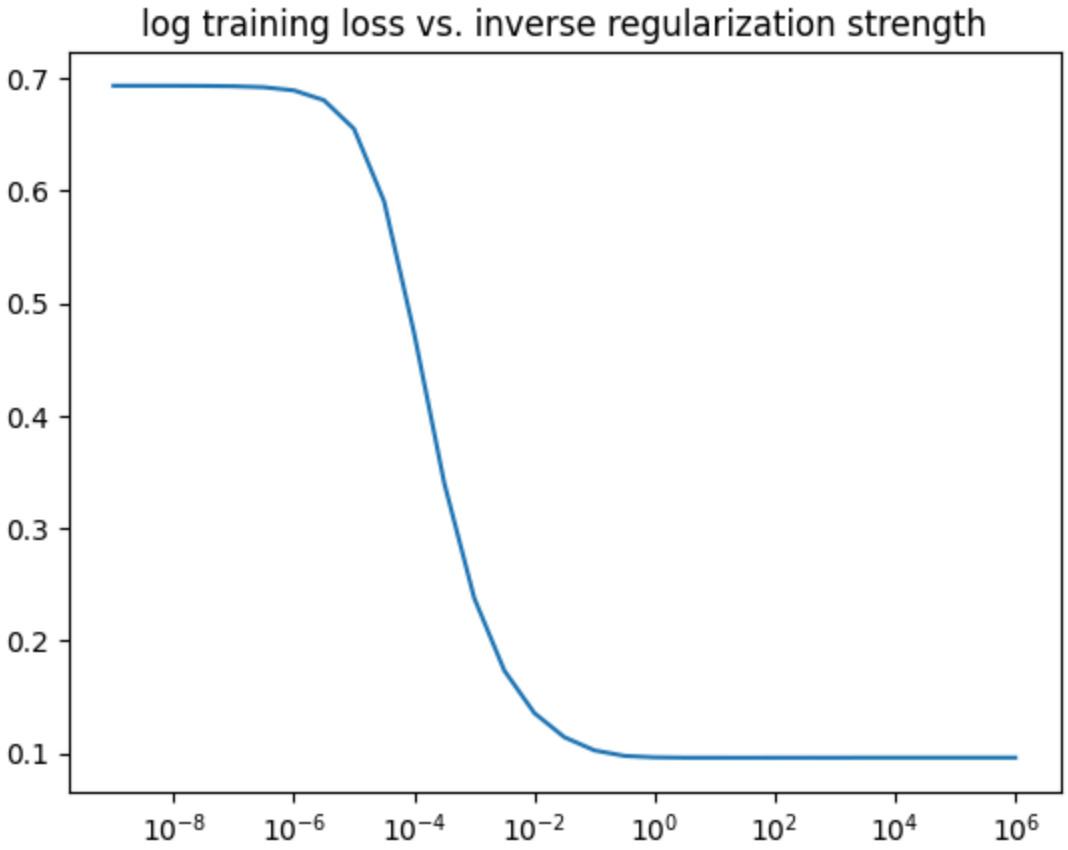
Part 2: Trousers vs. Dresses in Fashion MNIST

In [10]:
```
# read data
folder = "data_trouser_dress"
troudress_test_x = pd.read_csv(Path(folder) / Path("troudress_test_x.csv"))
troudress_train_x = pd.read_csv(Path(folder) / Path("troudress_train_x.csv"))
troudress_train_y = pd.read_csv(Path(folder) / Path("troudress_train_y.csv"))
```

In [11]:
```
def basic_logreg_varying_C(train_x, train_y):
    # logistic regression on trousers vs dressers, adding nonzero pixel count featu
    tr_models = []
    training_accuracies = []
    training_log_losses = []
    C_grid = np.logspace(-9, 6, 31)

    for C in C_grid:
        # set up LogReg classifier object
        tr_model = LogisticRegression(C=C, max_iter=1000)

        # fit data
        tr_model.fit(train_x, train_y.to_numpy().ravel())
        tr_models.append(tr_model)

        training_accuracies.append(tr_model.score(train_x, train_y))

        train_y_proba = tr_model.predict_proba(train_x)
        training_log_losses.append(log_loss(train_y, train_y_proba))

    plt.figure()
    plt.semilogx(C_grid, training_accuracies)
    plt.title("training accuracy vs. inverse regularization strength")
    plt.figure()
    plt.semilogx(C_grid, training_log_losses)
    plt.title("log training loss vs. inverse regularization strength")

    # using best C according to training loss to start
    best_tr_C_idx = np.argmin(training_log_losses)
    best_tr_C = C_grid[best_tr_C_idx]
    best_tr_model = tr_models[best_tr_C_idx]

    return best_tr_model, best_tr_C

best_model, best_C = basic_logreg_varying_C(troudress_train_x, troudress_train_y)
print(f'best C: {best_C}')
```

```
best C: 31.622776601683793
```





Training loss goes down as C increases (regularization strength decreases). This suggests that every pixel is necessary for making a determination between trouser and dress. This makes

sense, as clothing takes up a lot of image area, whereas handwritten digits do not.

Some ideas

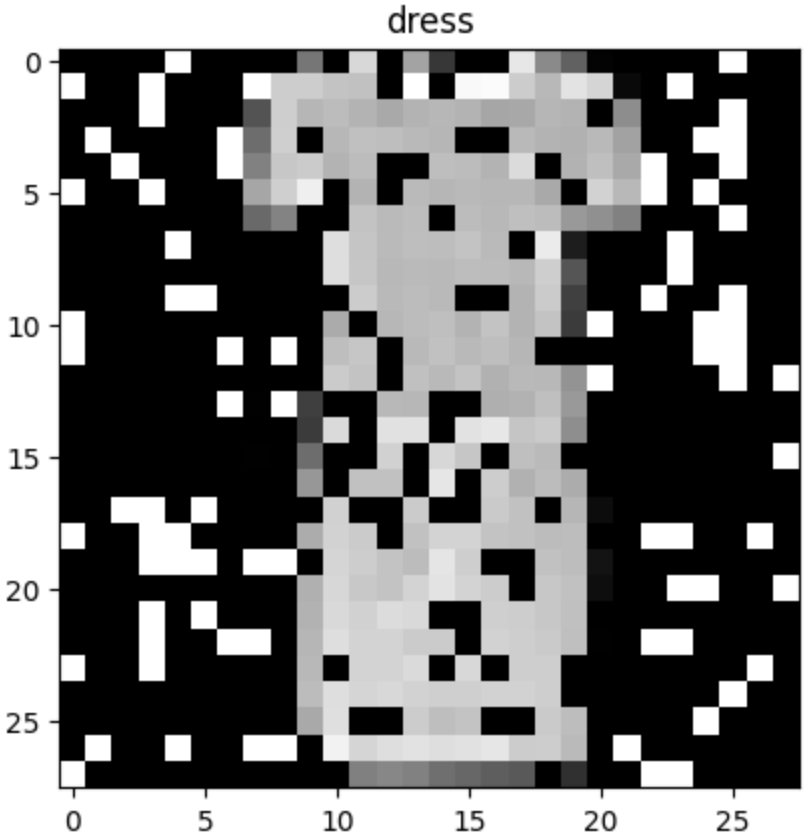
Would be super interesting to perform cross-validation to make the most out of the training set

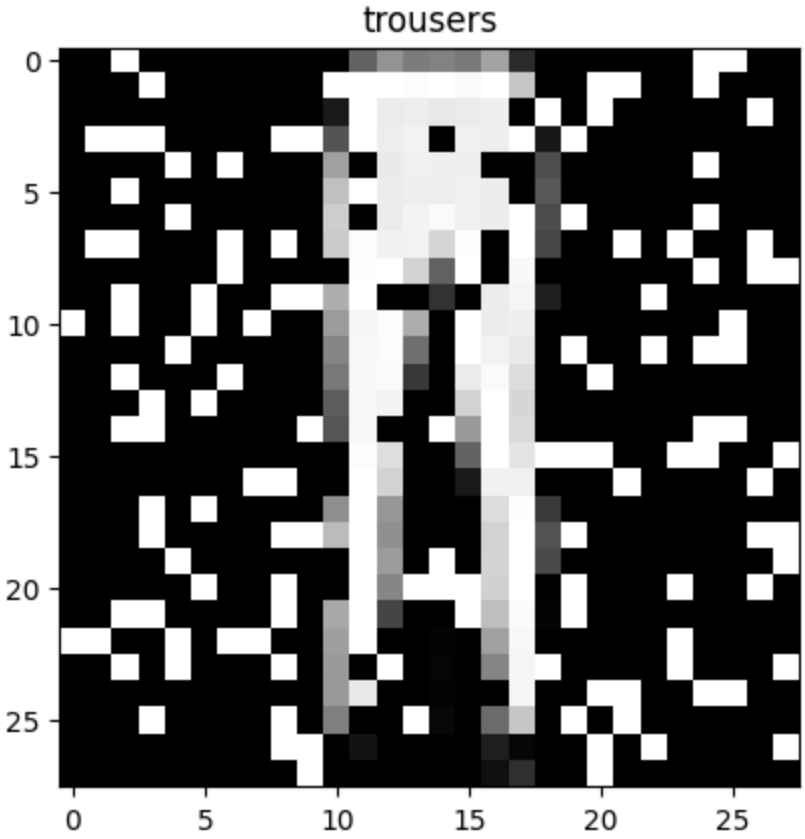
what features transformations to try?

I should plot the images to see what's going on

In [12]:
```
# plotting some of the data to see what's going on

for ii in range(troudress_train_x.shape[0]):
    label = troudress_train_y.loc[ii].to_numpy()
    data = troudress_train_x.loc[ii].to_numpy()
    num_nonzero = np.sum(data > 0)

    if not label and ii > 1000:
        print(ii, label)
        break
image = data.reshape((28, 28))
plt.figure()
plt.imshow(image, cmap='grey')
plt.title('dress')
print(f'num_nonzero: {num_nonzero}')


for ii in range(troudress_train_x.shape[0]):
    label = troudress_train_y.loc[ii].to_numpy()
    data = troudress_train_x.loc[ii].to_numpy()
    num_nonzero = np.sum(data > 0)

    if label and ii > 1000:
        print(ii, label)
        break
image = data.reshape((28, 28))
plt.figure()
plt.imshow(image, cmap='grey')
plt.title('trousers')
print(f'num_nonzero: {num_nonzero}')
```

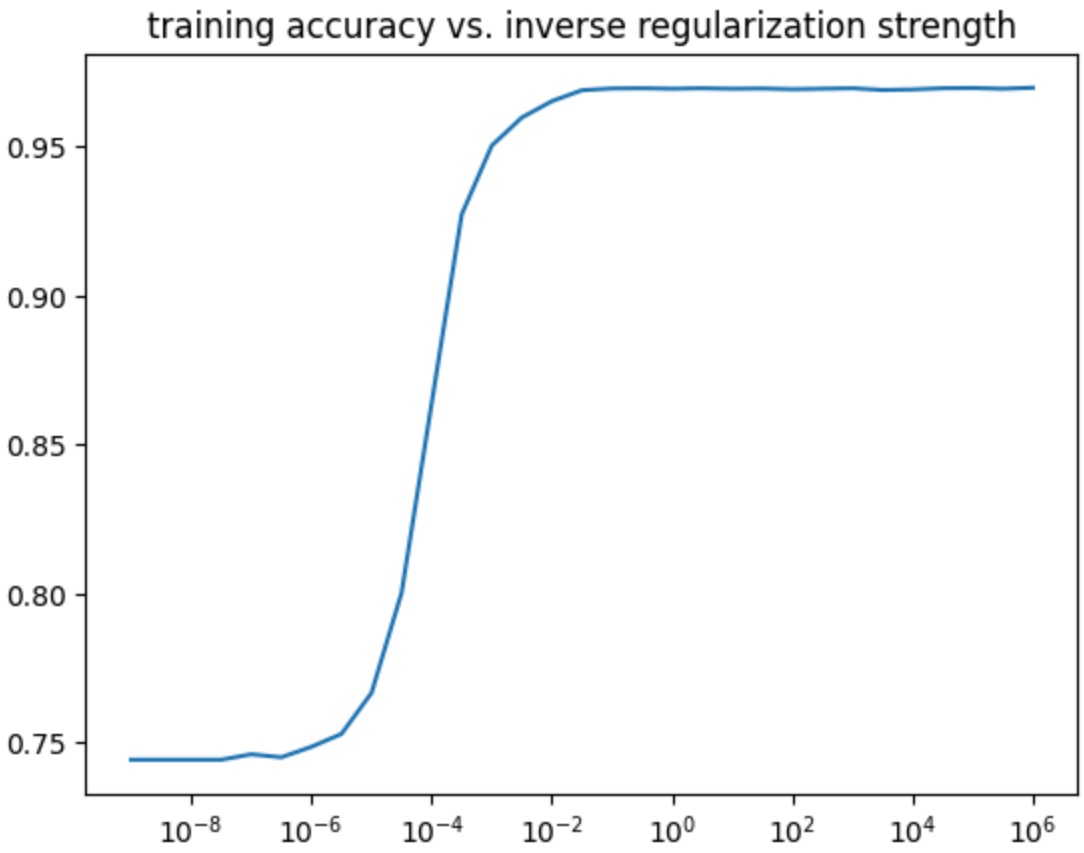
```
1001 [0]
num_nonzero: 346
1003 [1]
num_nonzero: 289
```

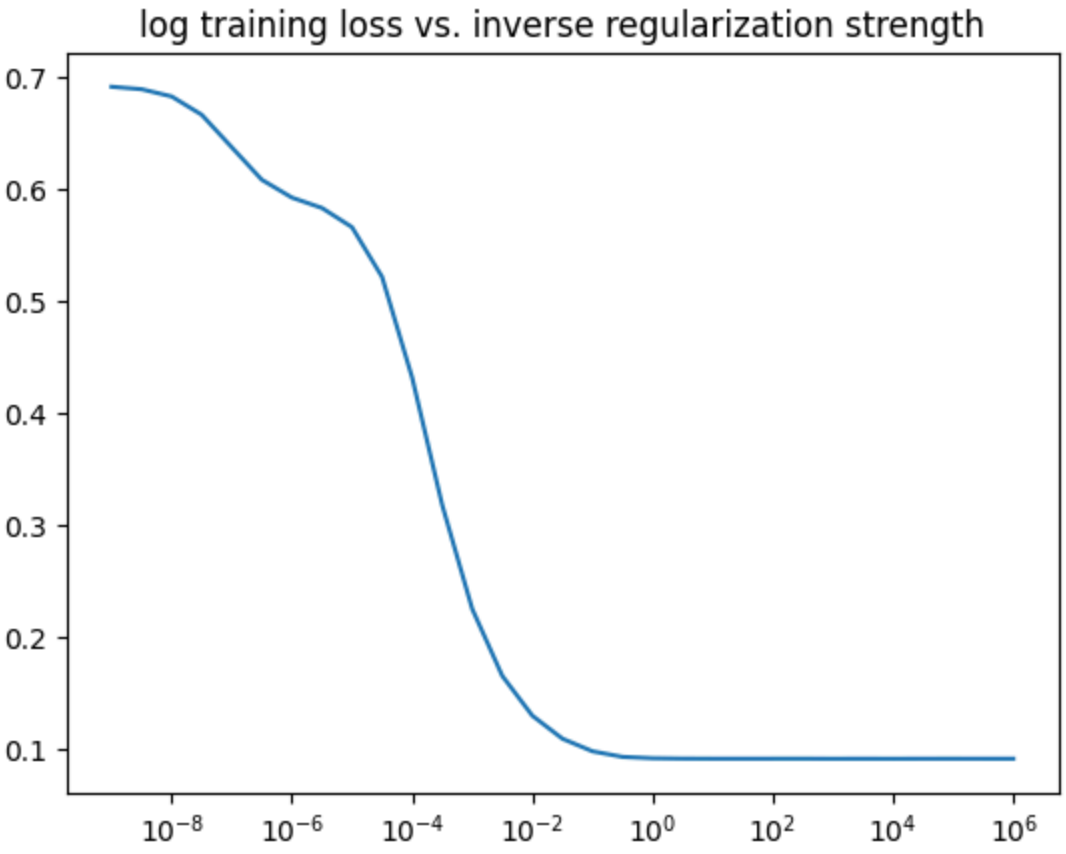




hmm. from a quick glance, a dress might take up more pixels than a pair of trousers. a feature to add could be the number of non-zero pixel values. a higher nonzero pixel count would denote a dress

In [13]:
```
# augment the feature matrix with a nonzero pixel count feature
nonzero_pixel_count = np.sum(troudress_train_x.to_numpy() > 0, axis=1)
aug_train_x = troudress_train_x.copy()
aug_train_x["nonzero_pixel_count"] = nonzero_pixel_count
best_model, best_C = basic_logreg_varying_C(aug_train_x, troudress_train_y)
```





Adding just the one feature of nonzero pixel count helps with training accuracy for highly regularized classifiers. That benefit goes away as regularization strength dies down (it would appear that the change it brings is very slightly negative).

Another pixel intensity-counting feature I could try could be counting the pixels in a center column with a particular width (motivated by the fact that trousers seem to take up less width than dresses).

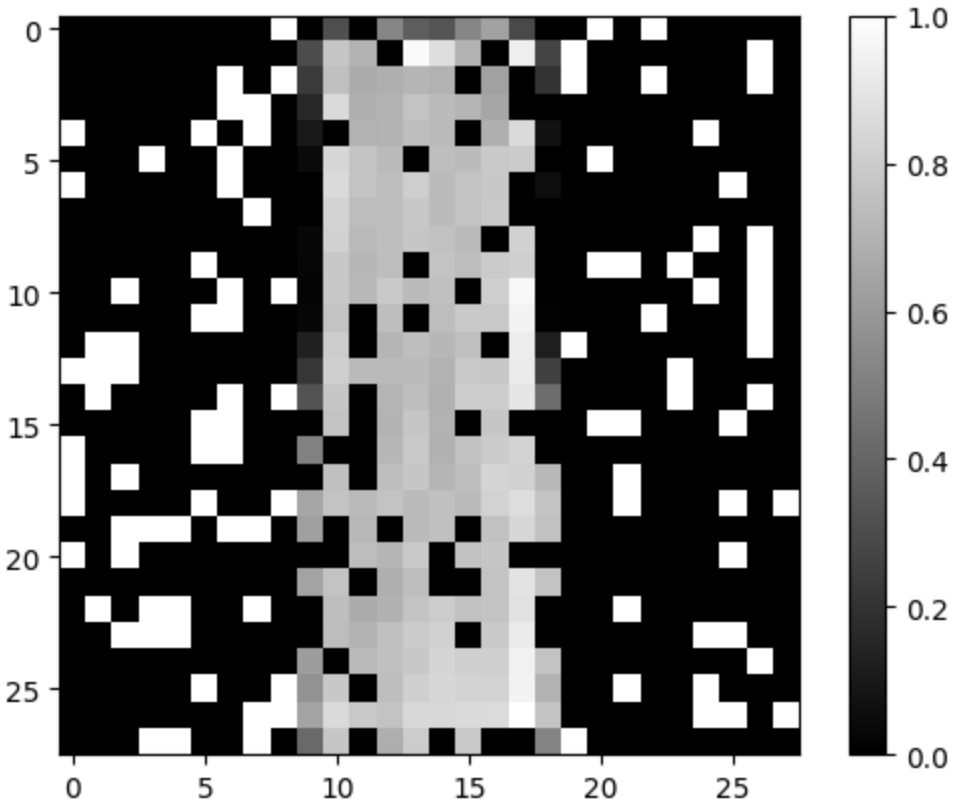
In lieu of an ad hoc decision on an area of the image to count non-zero pixels, I am also curious about creating a feature based on the outline of the article of clothing.

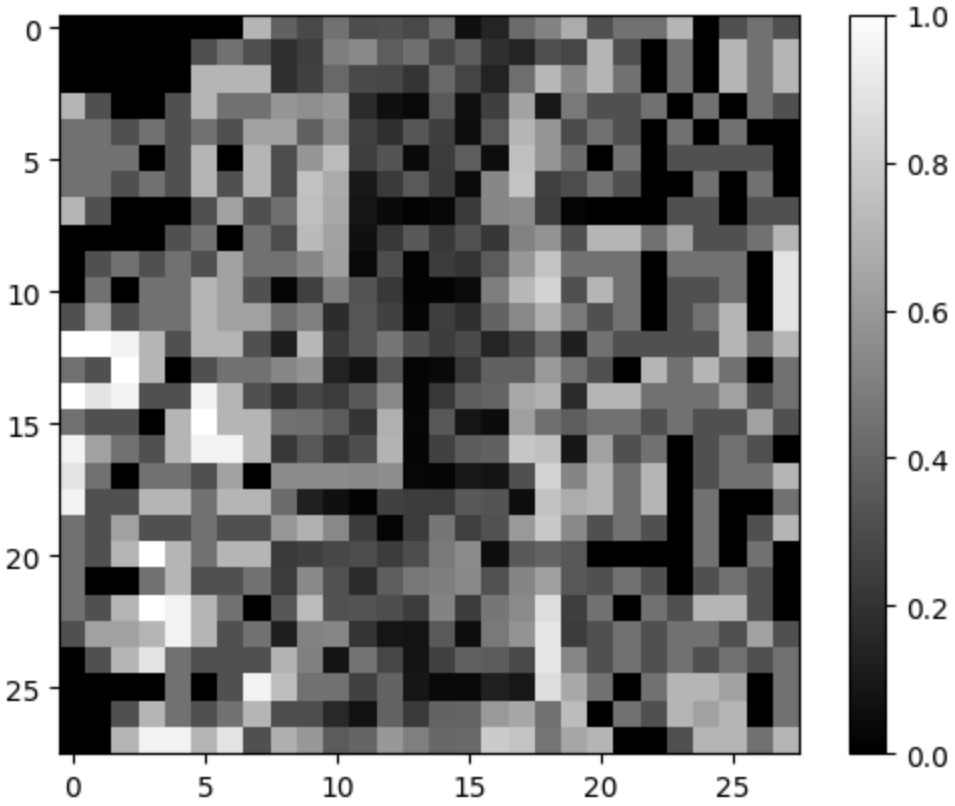
The Sobel operator is a simple way to find the gradient at every point in an image. It works by convolving the original image with a kernel to produce an alternate image. The kernel is a 3x3 matrix with values chosen to detect rapid transitions in pixel intensity, which is akin to a derivative, in a 3x3 area. The operator has variants for the x and y direction, and the x and y variants can even be combined via the Pythagorean theorem. The latter is what I will try to attempt here.

To clarify: I would like to use the combined Sobel operator to get image containing the outline of the clothing. I'll add this entire image as features to the original feature matrix. This doubles the number of features I have.

In [14]:
```
# scipy makes filtering with a sobel kernel easier
from scipy.ndimage import sobel
```

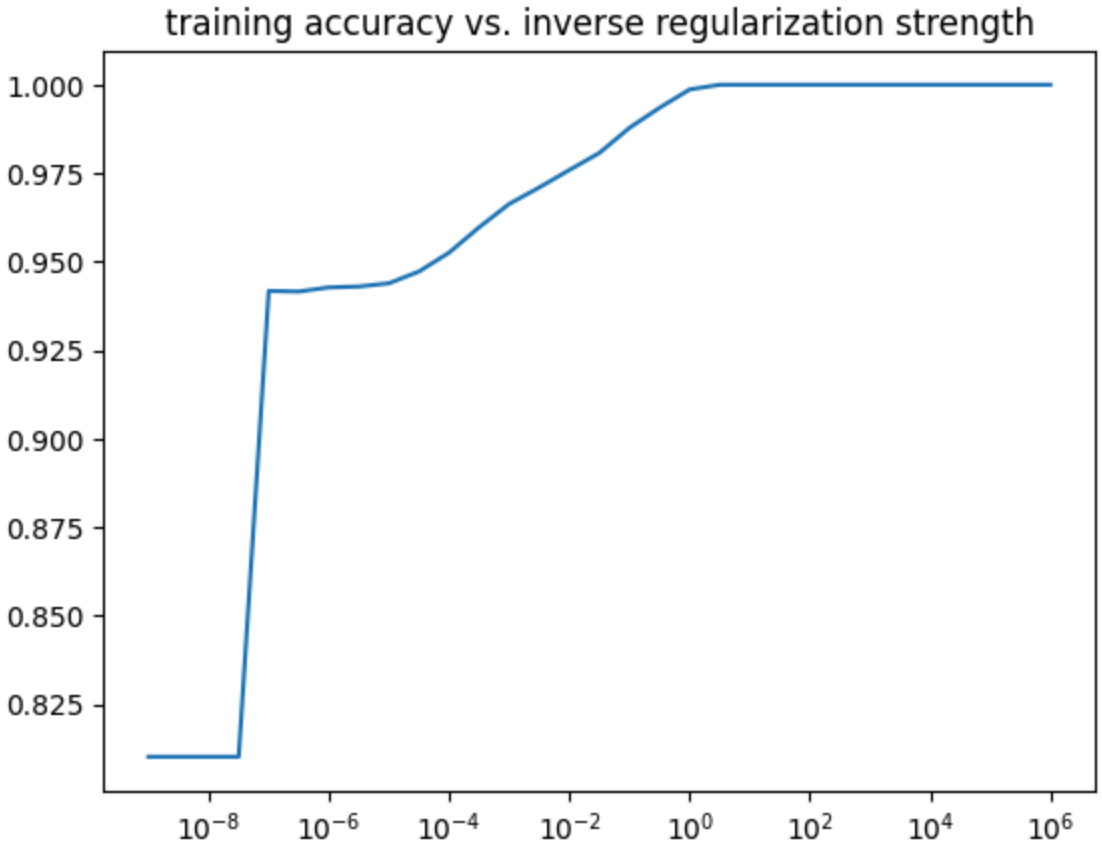
```
def create_sobelized_feature(train_x, col_prefix='s'):
    altered_train_x = train_x.copy()
    num_images = altered_train_x.shape[0]

    for ii in range(num_images):
        data = train_x.loc[ii].to_numpy()
        image = data.reshape((28, 28))
        sobeled_h = sobel(image, 0)
        sobeled_v = sobel(image, 1)
        sobeled = np.sqrt(sobeled_h**2 + sobeled_v**2)
        sobeled_norm = (sobeled - np.min(sobeled)) / (np.max(sobeled) - np.min(sobe
        sobeled_data = sobeled_norm.reshape(-1)
        altered_train_x.loc[ii] = sobeled_data

    plt.figure()
    plt.imshow(image, cmap='grey')
    plt.colorbar()
    plt.figure()
    plt.imshow(sobeled_norm, cmap='grey')
    plt.colorbar()

    columns  = altered_train_x.columns
    scolumns = [col_prefix+col for col in columns]
    scolumns

    rename_mapping = {col:scol for col, scol in zip(columns, scolumns)}
    altered_train_x = altered_train_x.rename(columns=rename_mapping)
    return altered_train_x

outline_train_x = create_sobelized_feature(troudress_train_x, col_prefix='s')
```

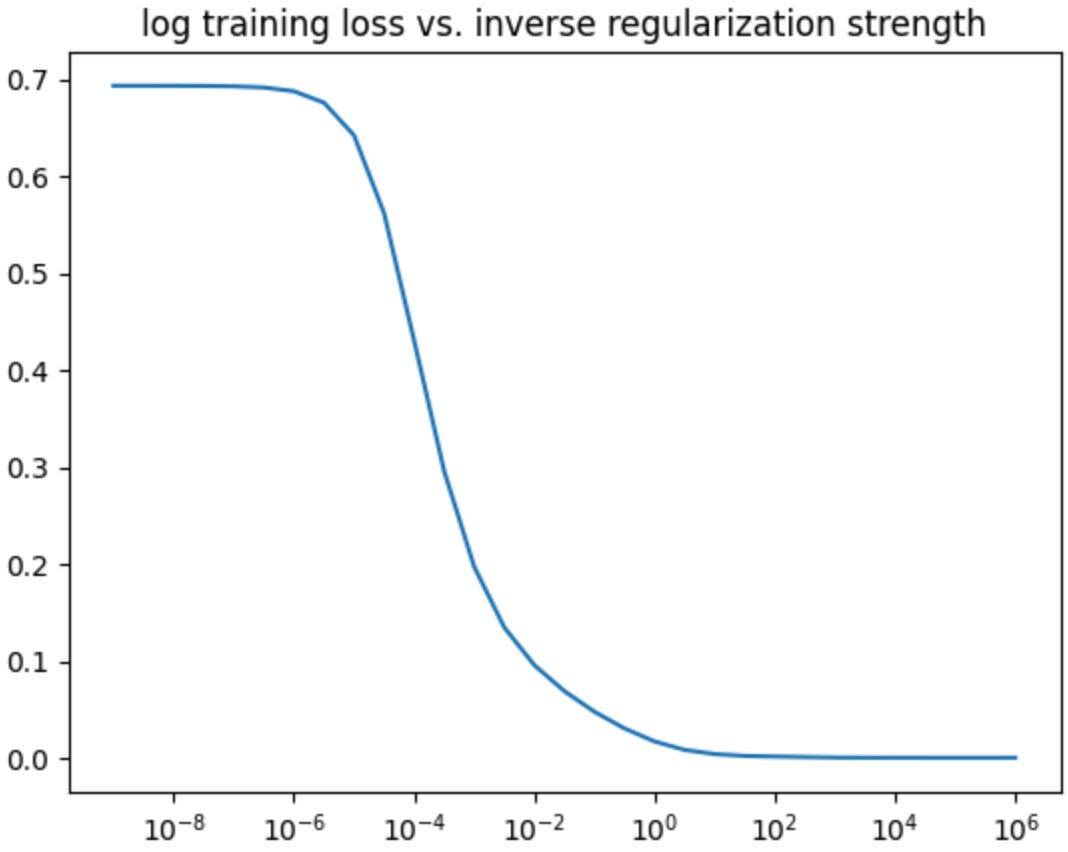




honestly, the "outline" map looks terrible. I'm curious to see how it functions as a transformed feature though

In [15]:
```
aug_train_x2 = troudress_train_x.join(outline_train_x)
best_model, best_C = basic_logreg_varying_C(aug_train_x2, troudress_train_y)
```





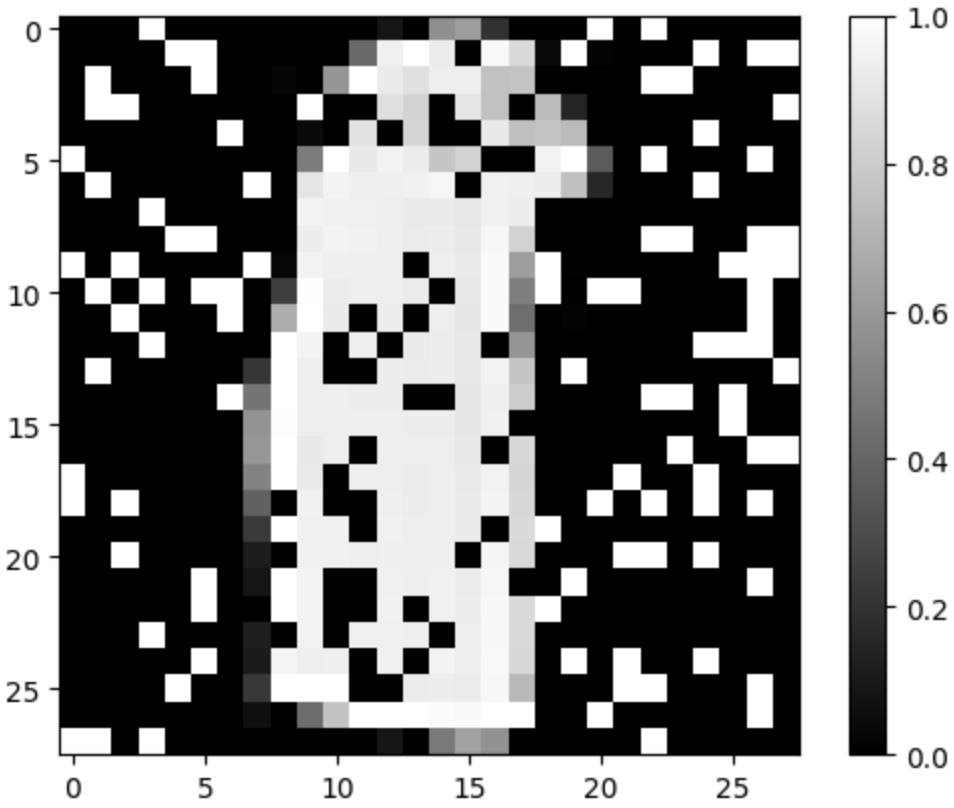
looks pretty effective at C=1 or greater. smells a little of overfitting tho... we'll have to see how it works on the test set

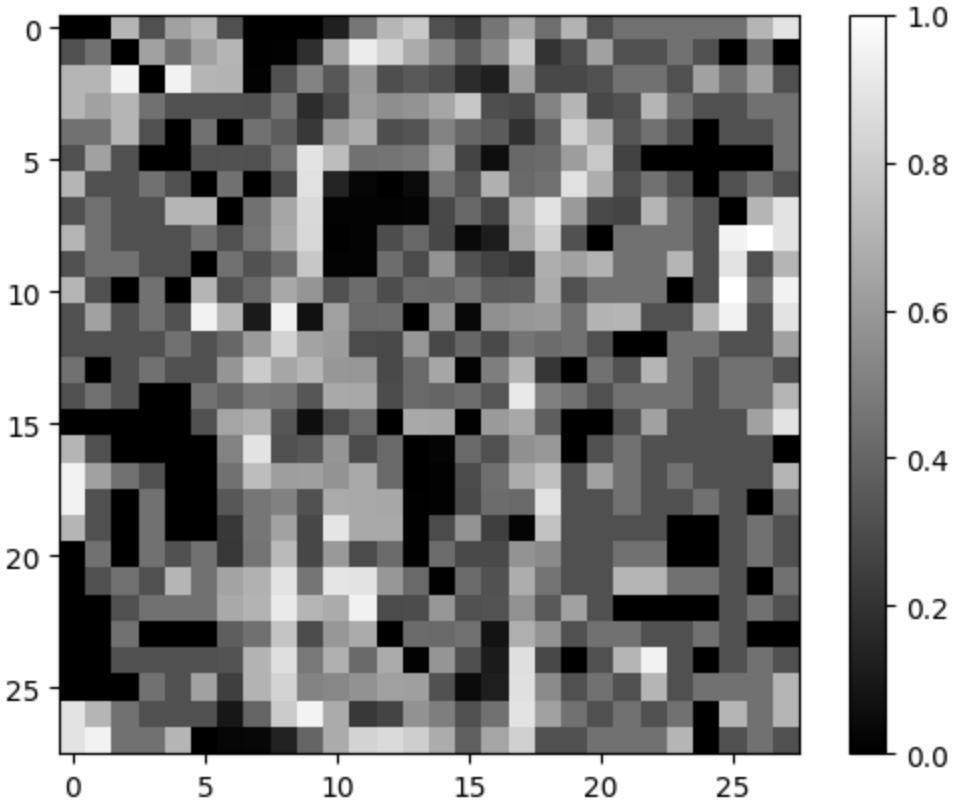
In [16]:
```
best_C
```

Out[16]:
```
np.float64(1000000.0)
```

best C value is the one that corresponds to the minimum regularization strength? so maybe 0 regularization is best

In [17]:
```
outline_test_x = create_sobelized_feature(troudress_test_x)
aug_test_x = troudress_test_x.join(outline_test_x)
predictions = best_model.predict(aug_test_x)
predictions_proba = best_model.predict_proba(aug_test_x)[:, 1]
predictions
predictions_proba

savename = 'yproba1_test.txt'
np.savetxt(savename, predictions_proba)
```

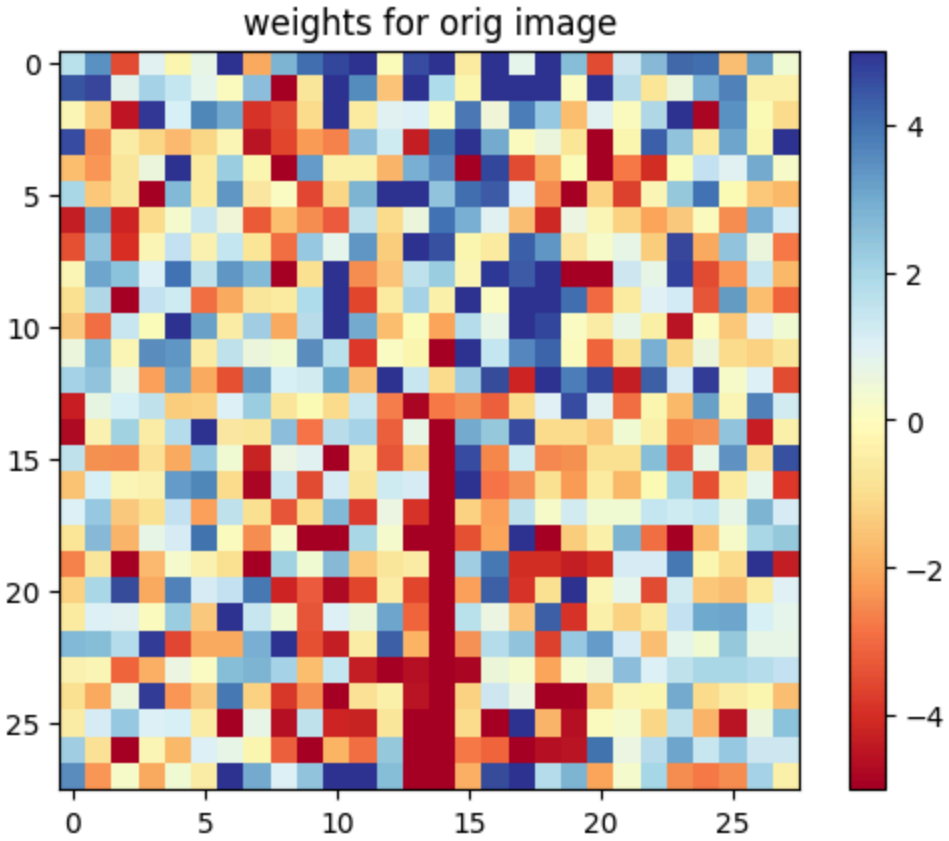


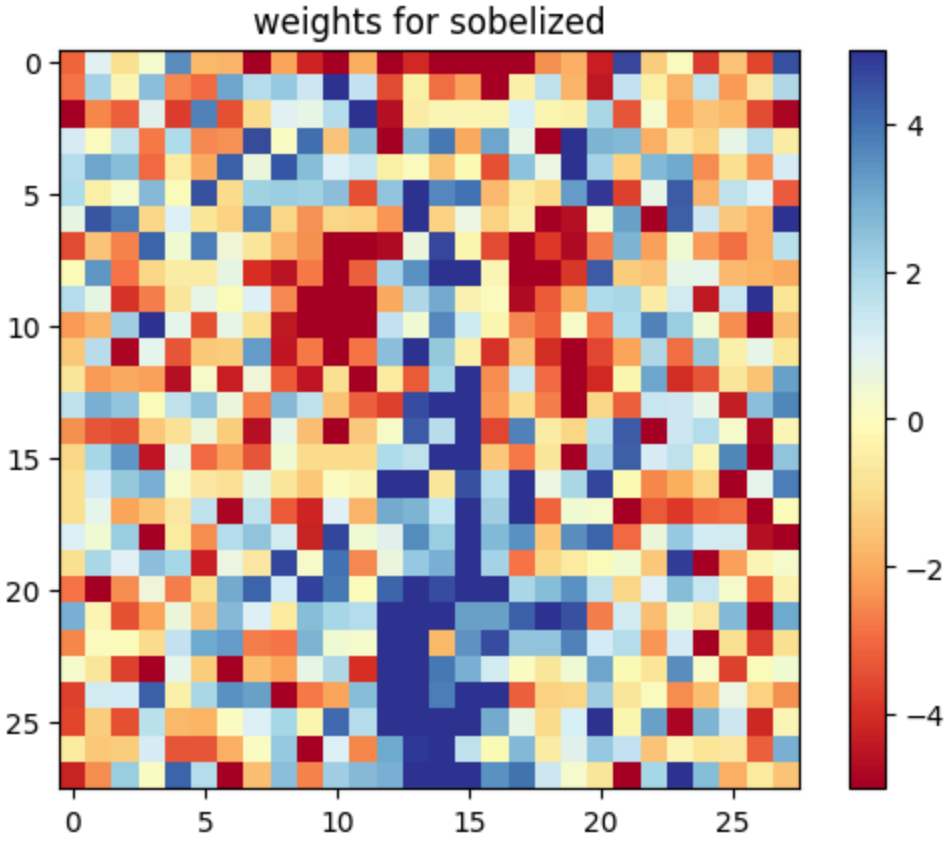


well, the test error of the model trained on sobelized images has an error rate of 6%, which is better than the first error rate that i got (which was 6.8%). let's take a look at the color map of the coefficients

In [18]:
```
def coef_to_image(coef, title='', vmin=-5, vmax=5):
    image = coef.reshape((28, 28))
    plt.figure()
    plt.imshow(image, cmap='RdYlBu', vmin=vmin, vmax=vmax)
    plt.colorbar()
    plt.title(title)
```

In [19]:
```
# reconstruct the two images that the coefs correspond to
coef1 = best_model.coef_[:, :784]
coef2 = best_model.coef_[:, 784:]

coef_to_image(coef1, title="weights for orig image")
coef_to_image(coef2, title="weights for sobelized")
```



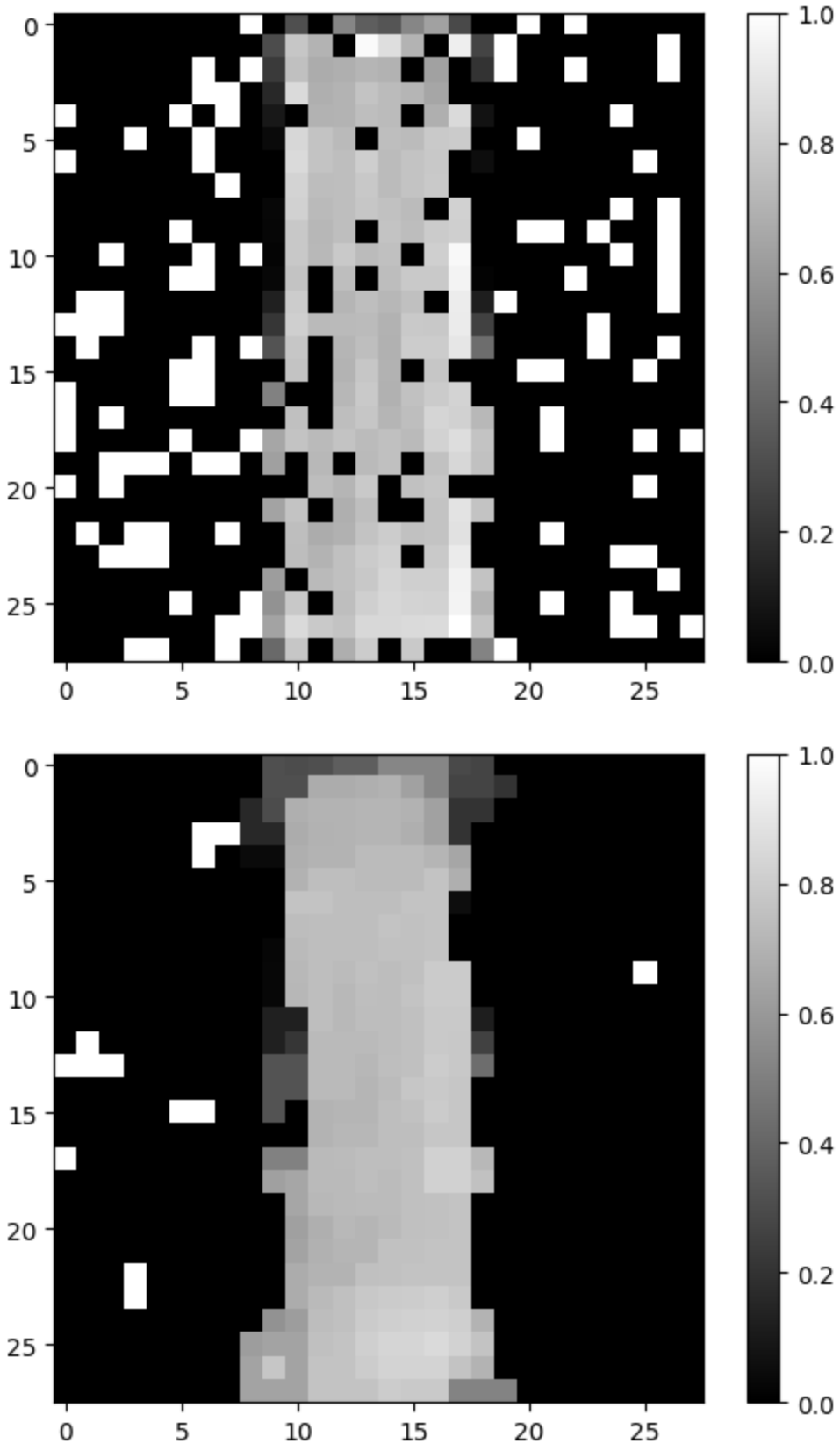


it's kinda cool: both sets of coefficient weights seem to place high importance on the bottom side of the columns near the middle. likely this is checking the split in the fabric due to trousers having two columns. other areas with large importance seem to be the armpit

area for the dress and the top of the piece of clothing. all of these generally concern the silhouette of the article of clothing.

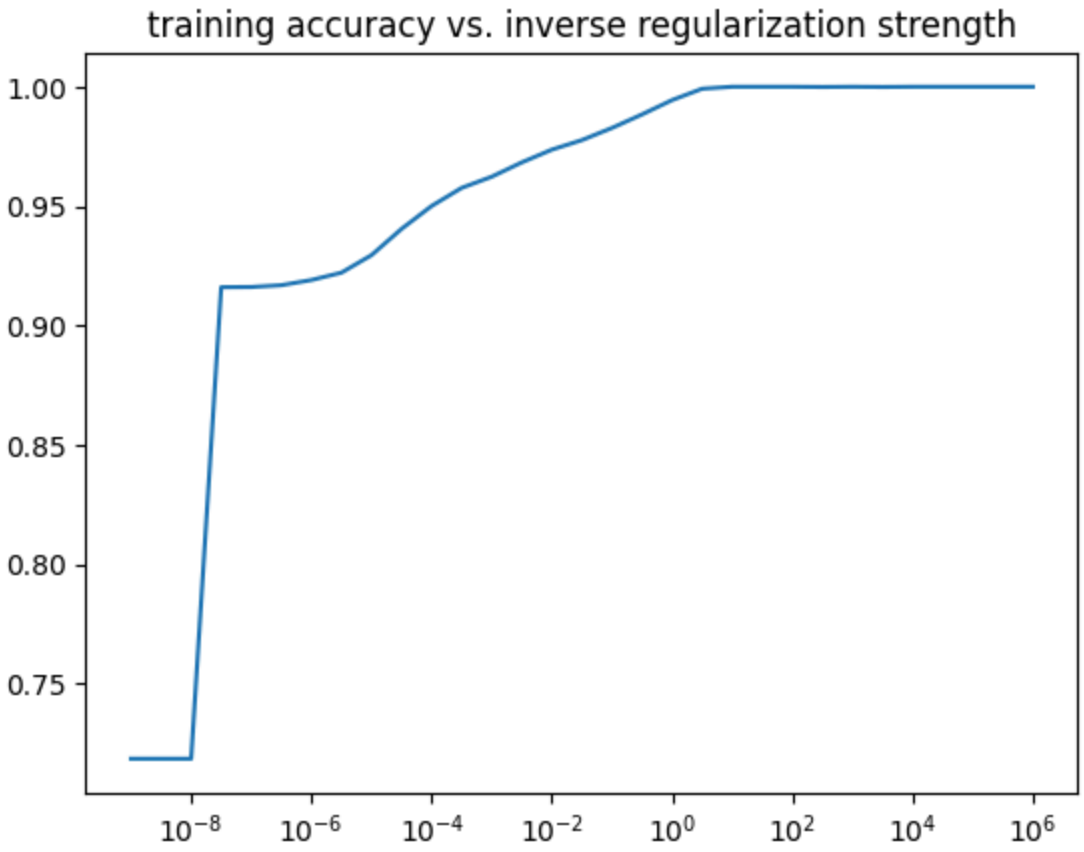
due to the salt and pepper noise in the training data, i'm very curious to see how a median filter would work to remove that noise. and i wonder if a median filtered image would be easier for the logistic regression to classify trousers and dresses

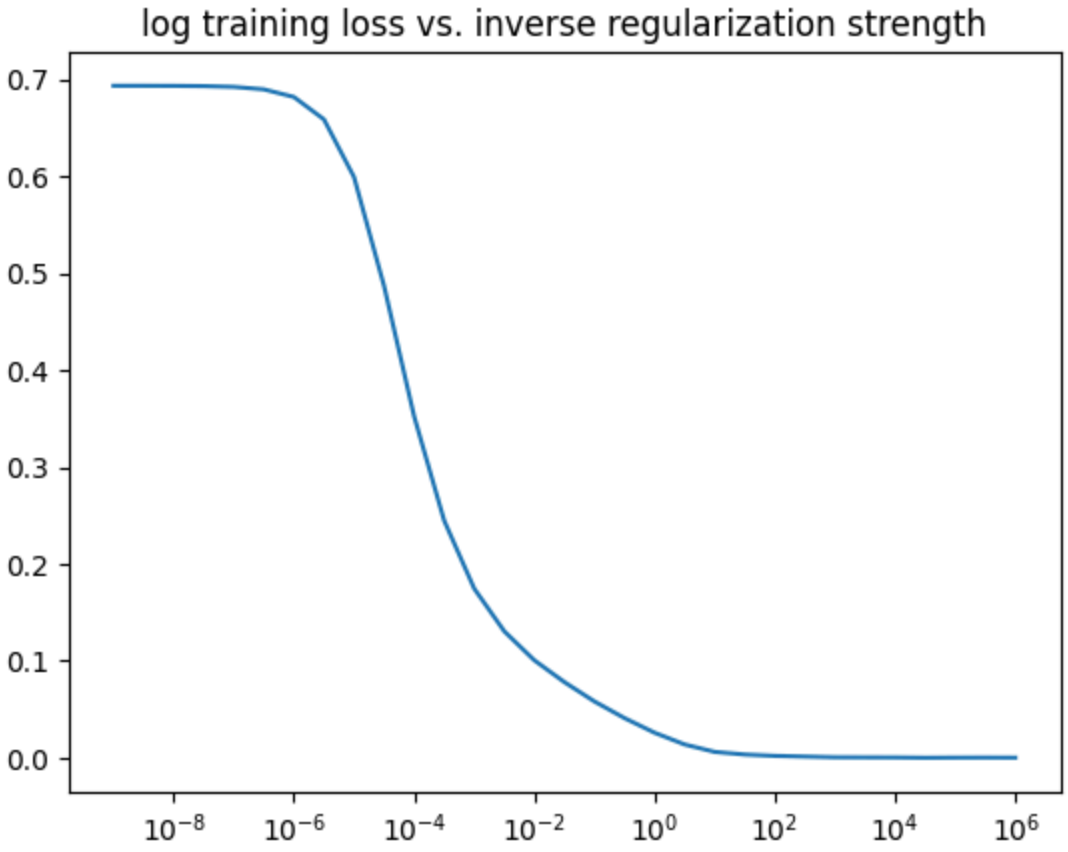
In [20]:
```
from scipy.ndimage import median_filter

def create_medfilt_feature(troudress_train_x, col_prefix='m'):
    medfilt_train_x = troudress_train_x.copy()
    num_images = medfilt_train_x.shape[0]

    for ii in range(num_images):
        data = troudress_train_x.loc[ii].to_numpy()
        image = data.reshape((28, 28))
        medfilt = median_filter(image, size=(3,3))
        medfilt_data = medfilt.reshape(-1)
        medfilt_train_x.loc[ii] = medfilt_data

    plt.figure()
    plt.imshow(image, cmap='grey')
    plt.colorbar()
    plt.figure()
    plt.imshow(medfilt, cmap='grey')
    plt.colorbar()

    columns  = medfilt_train_x.columns
    scolumns = [col_prefix+col for col in columns]
    scolumns

    rename_mapping = {col:scol for col, scol in zip(columns, scolumns)}
    medfilt_train_x = medfilt_train_x.rename(columns=rename_mapping)
    return medfilt_train_x

medfilt_train_x = create_medfilt_feature(troudress_train_x, col_prefix='m')
```



ok that worked kind of really well. the only thing i'm worried about is if it removes the black pixels from the trouser pants split, or the black pixels from the dress underarms. oh well, we should just try it now

In [21]:
```
aug_train_x_medfilt = troudress_train_x.join(medfilt_train_x)
best_model, best_C = basic_logreg_varying_C(aug_train_x_medfilt, troudress_train_y)
```



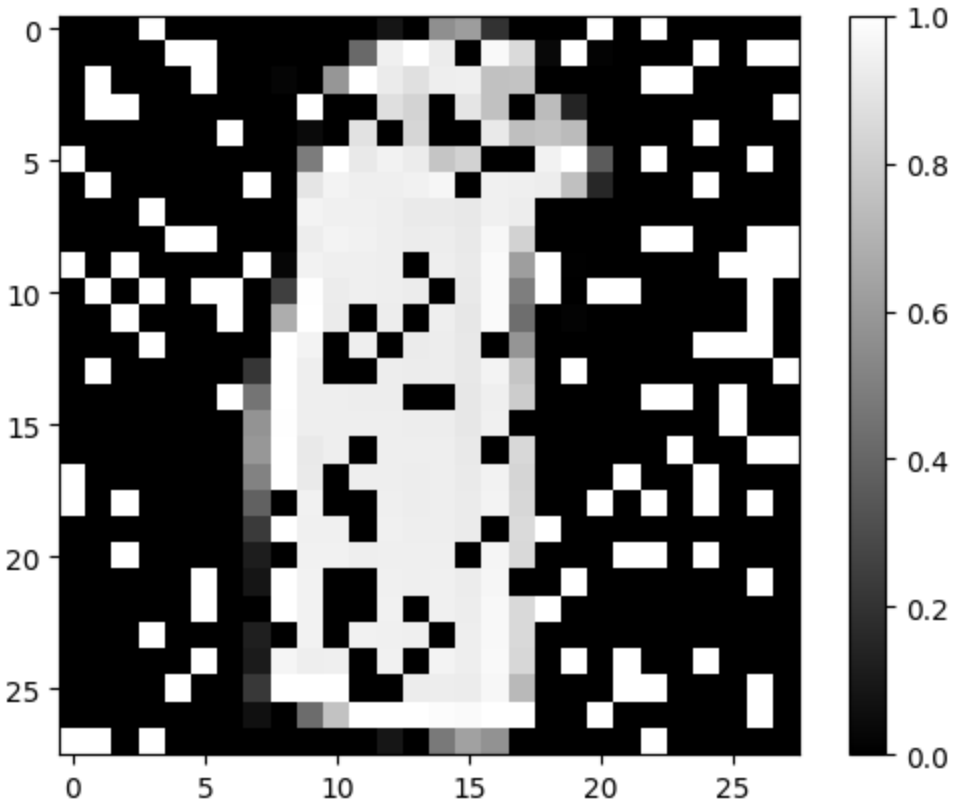


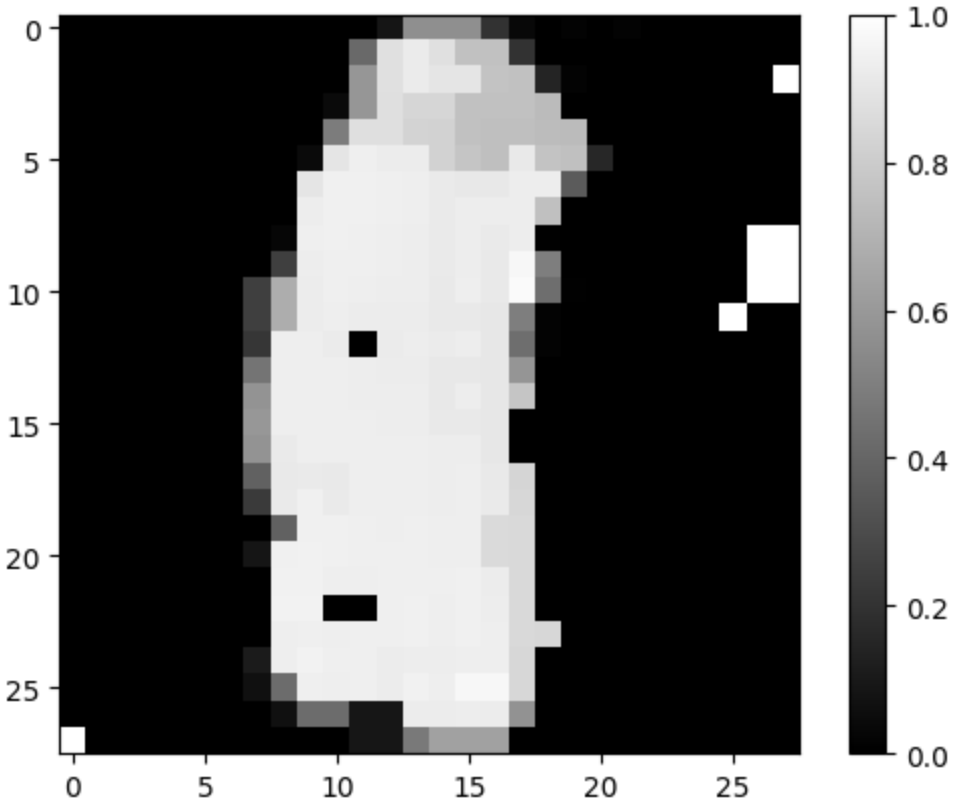
similar behavior to the sobelized images, tho the training loss peaks at a lower regularization strength

In [22]:
```
best_C
```

Out[22]:
```
np.float64(31622.776601683792)
```

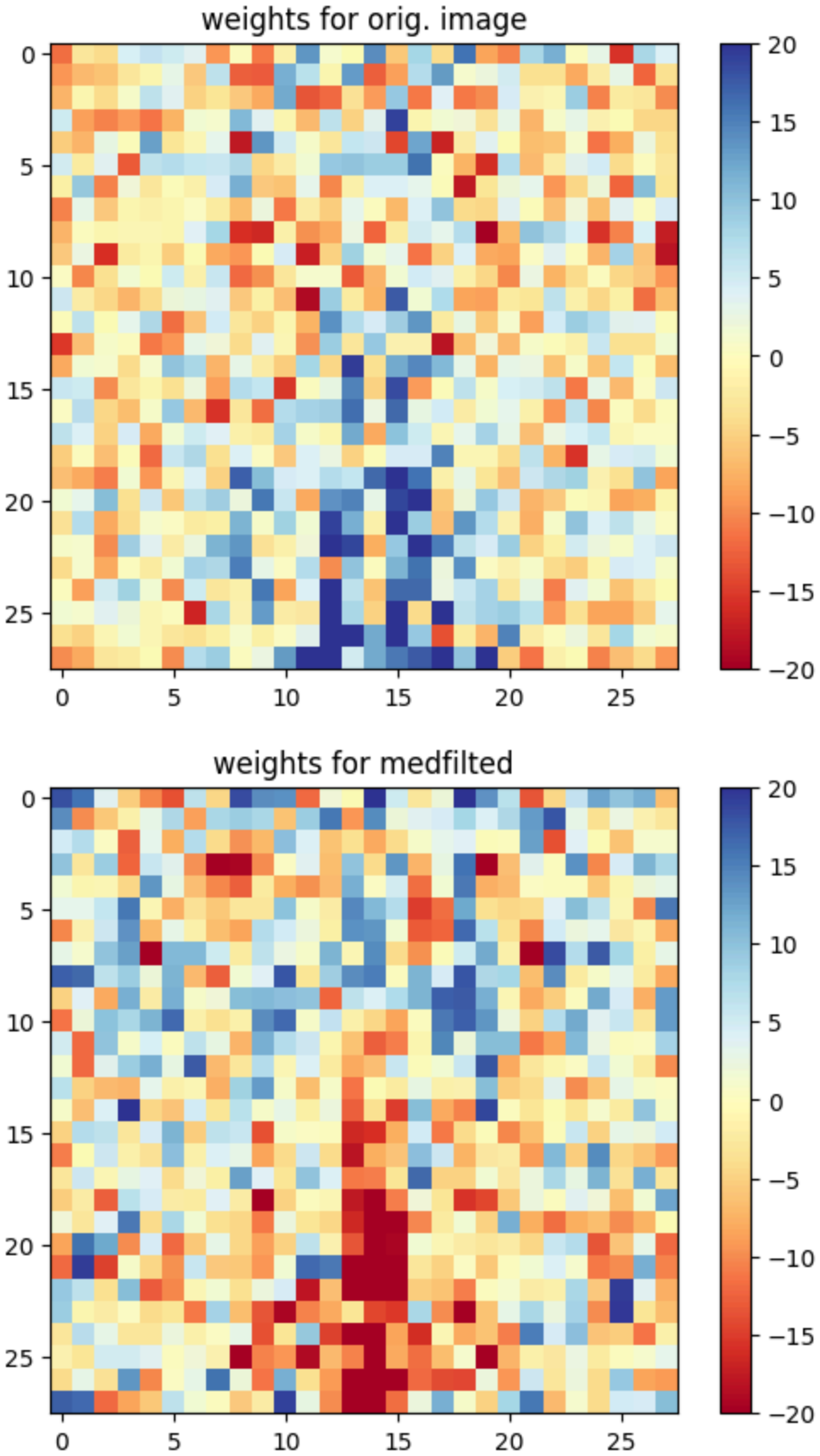
In [23]:
```
medfilt_test_x = create_medfilt_feature(troudress_test_x)
aug_test_x_medfilt = troudress_test_x.join(medfilt_test_x)
predictions = best_model.predict(aug_test_x_medfilt)
predictions_proba = best_model.predict_proba(aug_test_x_medfilt)[:, 1]
predictions
predictions_proba

savename = 'yproba1_test.txt'
np.savetxt(savename, predictions_proba)
```





median filter gives me an improvement of 0.05% over sobel. not much gains. but still better than the baseline performance

In [24]:
```
coef1 = best_model.coef_[:, :784]
coef2 = best_model.coef_[:, 784:]
coef_to_image(coef1, title="weights for orig. image", vmin=-20, vmax=20)
coef_to_image(coef2, title="weights for medfilted", vmin=-20, vmax=20)
```



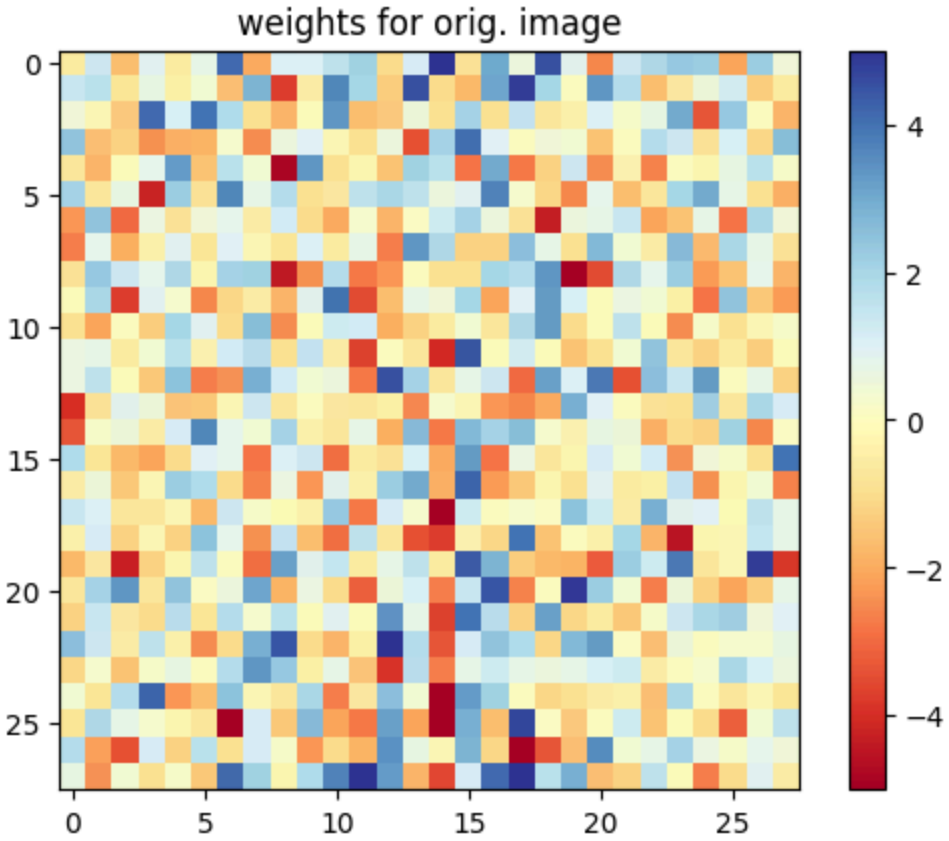
from the coefficient weiights, the classifier is generally still looking in the same areas

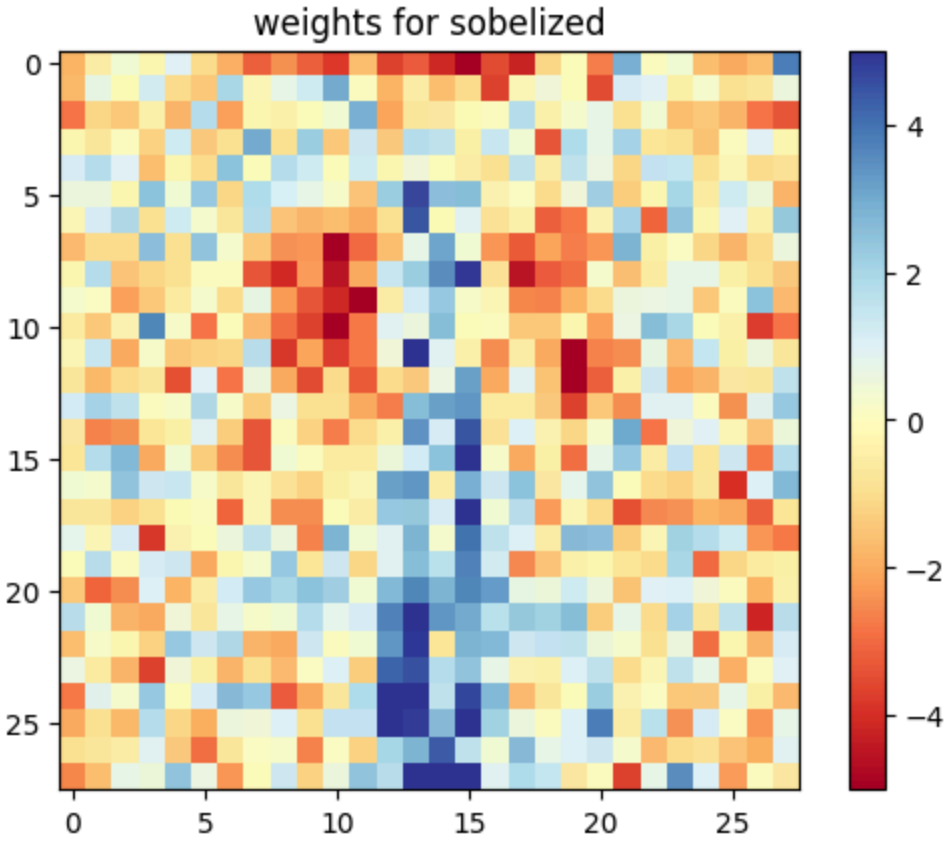
now i'll try to put the two kernel-filtered feature ideas together

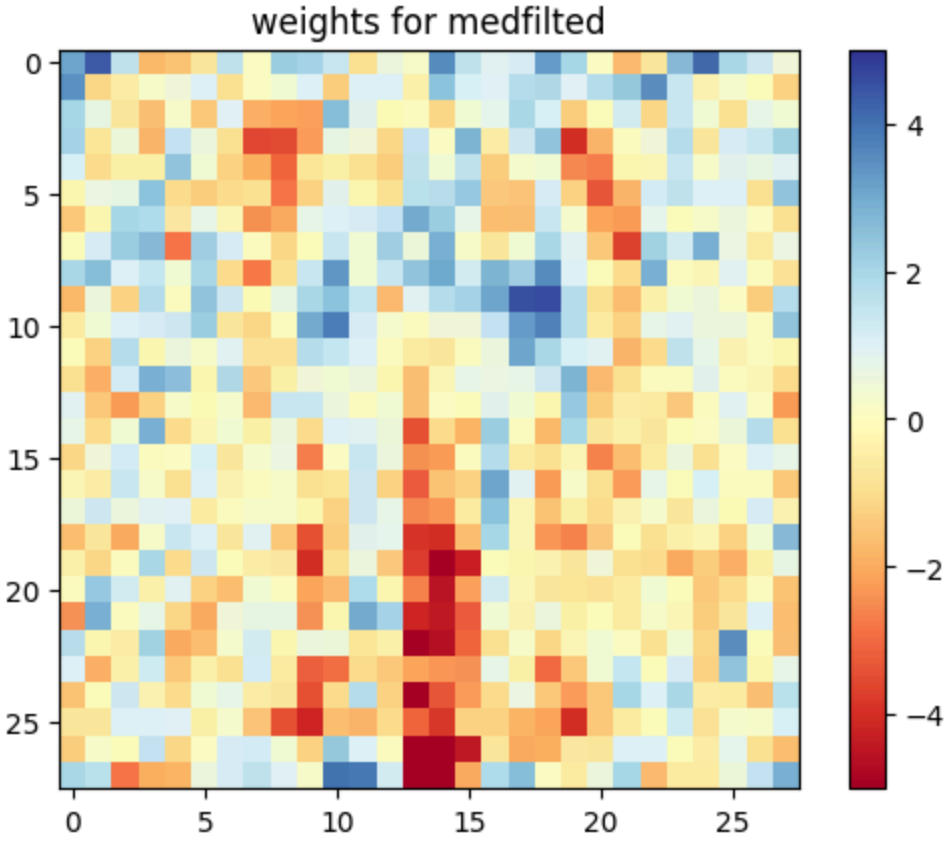
In [29]:
```
# train
aug_train_x_full = troudress_train_x.join([outline_train_x, medfilt_train_x])
tr_model = LogisticRegression(C=best_C, max_iter=1000)      # best_C was around 31k
tr_model.fit(aug_train_x_full, troudress_train_y.to_numpy().ravel())

# infer
aug_test_x_full = troudress_test_x.join([outline_test_x, medfilt_test_x])
predictions = tr_model.predict(aug_test_x_full)
predictions_proba = tr_model.predict_proba(aug_test_x_full)[:, 1]
predictions
predictions_proba

savename = 'yproba1_test.txt'
np.savetxt(savename, predictions_proba)
```

i get an error rate of 4.65%. significant improvement from either of the two filters by themselves! it's honestly rather exciting.

In [30]:
```
coef1 = tr_model.coef_[:, :784]
coef2 = tr_model.coef_[:, 784:1568]
coef3 = tr_model.coef_[:, 1568:]
coef_to_image(coef1, title="weights for orig. image")#, vmin=-1, vmax=1)
coef_to_image(coef2, title="weights for sobelized")#, vmin=-1, vmax=1)
coef_to_image(coef3, title="weights for medfilted")#, vmin=-1, vmax=1)
```







it looks like the features that the classifier used to look for in a single image are now split up into two images. this might have meaningful benefit. since the sobelized and median-filtered images each offer an alternate view of the same object, it's possible that one feature

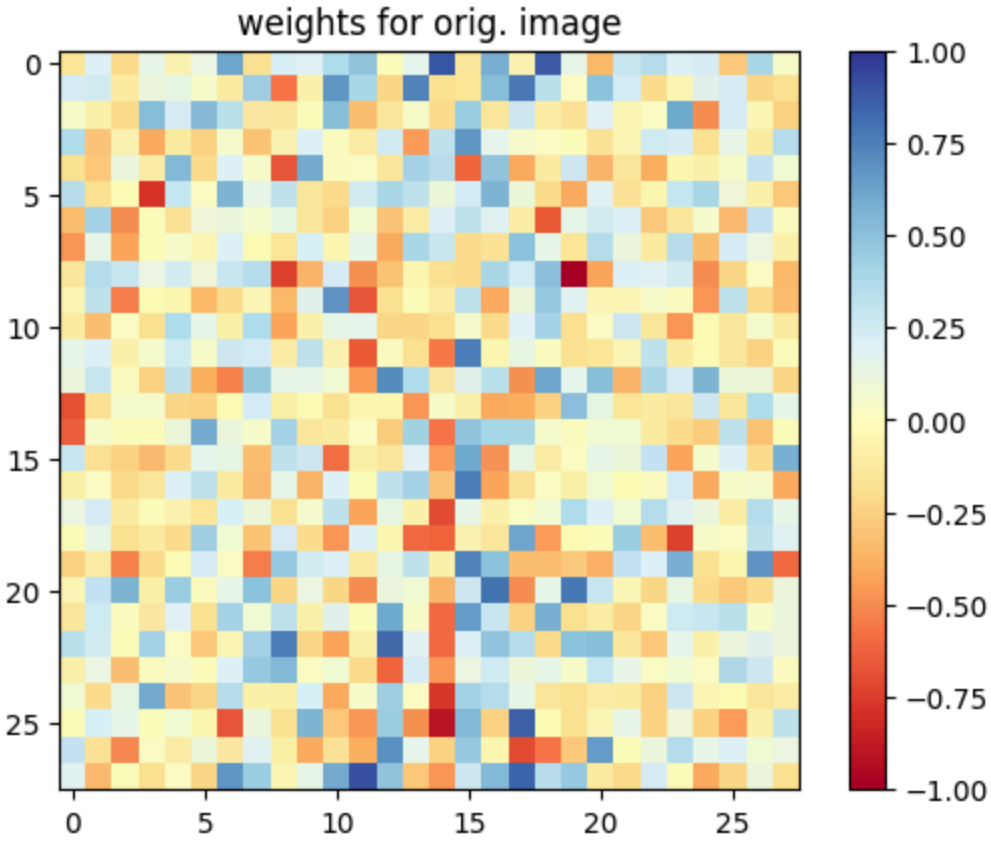
presents itself better in one view vs. the other. it's as if one features has a higher SNR in one view than the other. in any case, it's cool that throwing more data augmentations at the classifier is helpful.

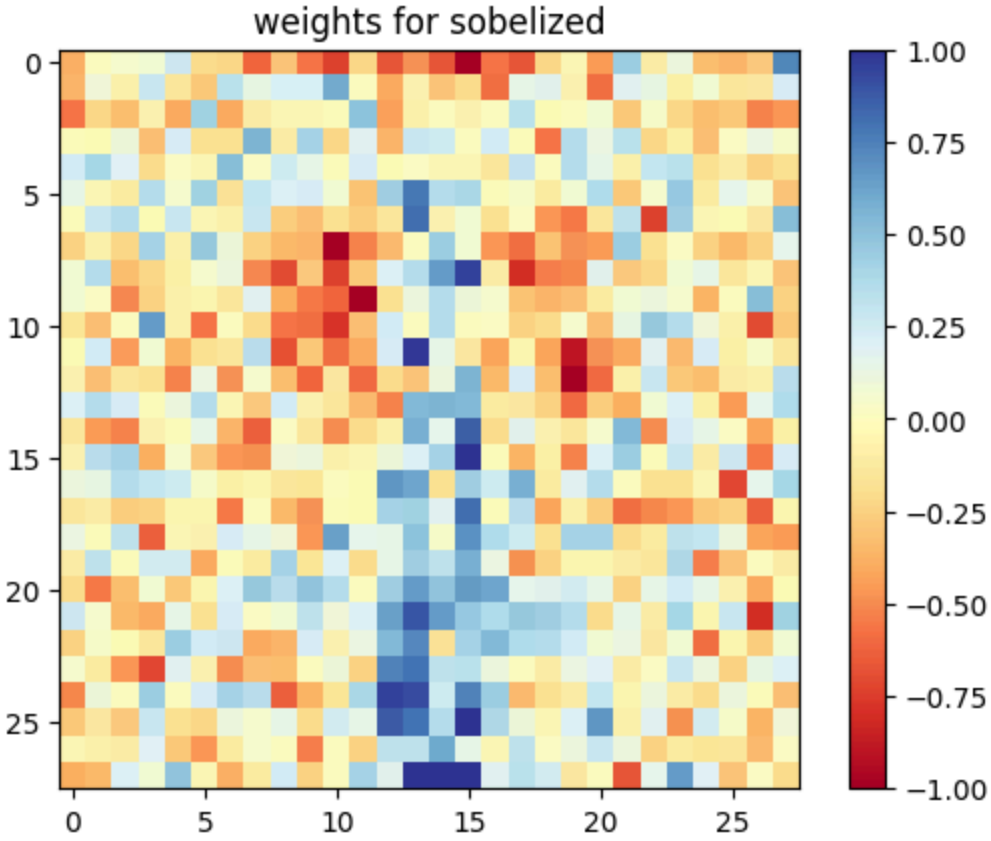
It's bothering me that the "best C" is one that is basically no regularization. I think this will lead to overfitting. Since the training loss curve showed that C=1 still yielded basically 0 training loss, I would like to try that value for C instead of 31k so that I enforce at least some kind of regularization.

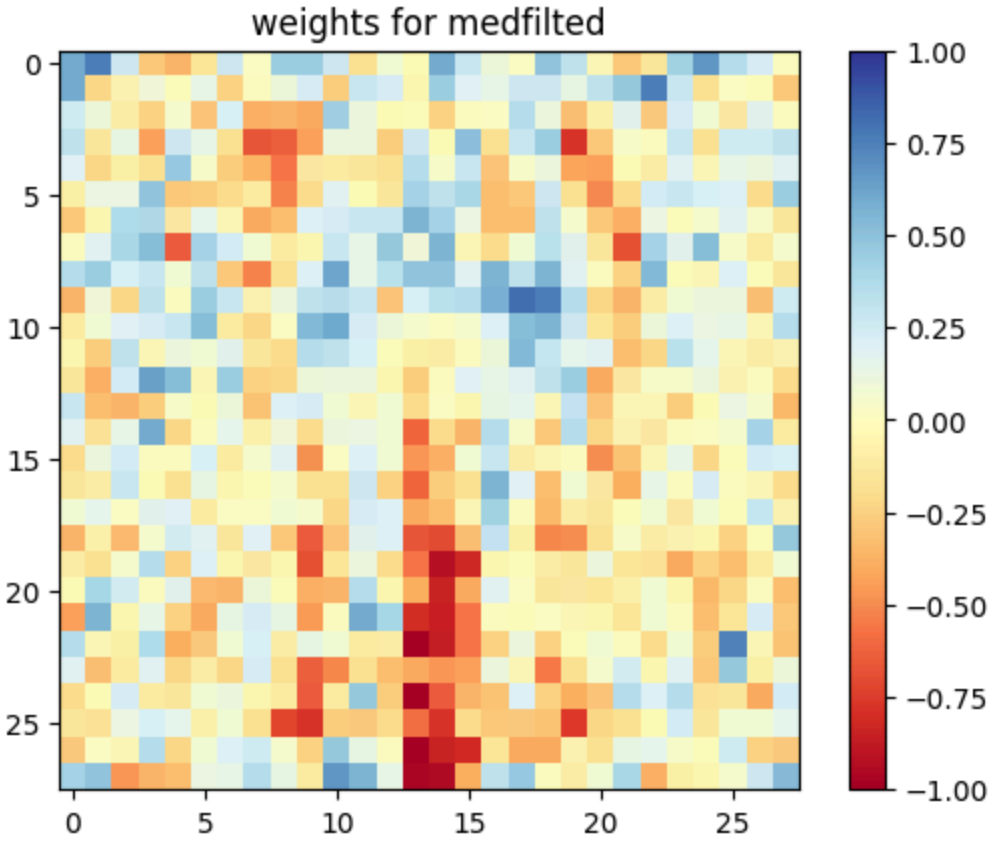
In [27]:
```
# train
aug_train_x_full = troudress_train_x.join([outline_train_x, medfilt_train_x])
tr_model = LogisticRegression(C=1, max_iter=1000)
tr_model.fit(aug_train_x_full, troudress_train_y.to_numpy().ravel())

# infer
aug_test_x_full = troudress_test_x.join([outline_test_x, medfilt_test_x])
predictions = tr_model.predict(aug_test_x_full)
predictions_proba = tr_model.predict_proba(aug_test_x_full)[:, 1]
predictions
predictions_proba

savename = 'yproba1_test.txt'
np.savetxt(savename, predictions_proba)
```

I get an error rate of 4.15%. Marked improvement from C=31k! I believe this is proof for my hunch that a little regularization helped with overfitting. I do not have time to test other values of C. But I believe I made a good choice for C based on the training loss curve I saw. If I had time to test more values of C, I would try those values for which the training loss is a little worse

In [28]:
```
coef1 = tr_model.coef_[:, :784]
coef2 = tr_model.coef_[:, 784:1568]
coef3 = tr_model.coef_[:, 1568:]
coef_to_image(coef1, title="weights for orig. image", vmin=-1, vmax=1)
coef_to_image(coef2, title="weights for sobelized", vmin=-1, vmax=1)
coef_to_image(coef3, title="weights for medfilted", vmin=-1, vmax=1)
```


weights for orig. image
0
5
10
15
20
25
1.00
0.75
0.50
0.25
0.00
−0.25
−0.50
−0.75
−1.00


weights for sobelized
0
5
10
15
20
25
1.00
0.75
0.50
0.25
0.00
−0.25
−0.50
−0.75
−1.00



Visually hard to tell the difference between coefficients when C=31k and C=1k. biggest difference is the dynamic range of the coefficients. that appears to be the work of the L2 penalty in decreasing the overall magnitude of the coefficient vector. the test error rate speaks for itself though. I do believe the regularization to avoid overfitting explains the improvement in performance