Eugene_Tang_p1

October 12, 2018

1 Project 1: Digit Classification with KNN and Naive Bayes

In this project, you'll implement your own image recognition system for classifying digits. Read through the code and the instructions carefully and add your own code where indicated. Each problem can be addressed succinctly with the included packages – please don't add any more. Grading will be based on writing clean, commented code, along with a few short answers.

As always, you're welcome to work on the project in groups and discuss ideas on the course wall, but please prepare your own write-up (with your own code).

If you're interested, check out these links related to digit recognition:
Yann Lecun's MNIST benchmarks: http://yann.lecun.com/exdb/mnist/
Stanford Streetview research and data: http://ufldl.stanford.edu/housenumbers/

In [1]:
```
# This tells matplotlib not to try opening a new window for each plot.
%matplotlib inline

# Import a bunch of libraries.
import time
import numpy as np
import matplotlib.pyplot as plt
from matplotlib.ticker import MultipleLocator
from sklearn.pipeline import Pipeline
from sklearn.datasets import fetch_mldata
from sklearn.neighbors import KNeighborsClassifier
from sklearn.metrics import confusion_matrix
from sklearn.linear_model import LinearRegression
from sklearn.naive_bayes import BernoulliNB
from sklearn.naive_bayes import MultinomialNB
from sklearn.naive_bayes import GaussianNB
from sklearn.grid_search import GridSearchCV
from sklearn.metrics import classification_report

# Set the randomizer seed so results are the same each time.
np.random.seed(0)
```

```
/Users/eugenetang/miniconda3/envs/W207/lib/python3.7/site-packages/sklearn/utils/__init__.py:4
  from collections import Sequence
/Users/eugenetang/miniconda3/envs/W207/lib/python3.7/site-packages/sklearn/cross_validation.py
```

```
  "This module will be removed in 0.20.", DeprecationWarning)
/Users/eugenetang/miniconda3/envs/W207/lib/python3.7/site-packages/sklearn/grid_search.py:42: D
  DeprecationWarning)
```

Load the data. Notice that we are splitting the data into training, development, and test. We also have a small subset of the training data called mini_train_data and mini_train_labels that you should use in all the experiments below, unless otherwise noted.

In [2]:
```
# Load the digit data either from mldata.org, or once downloaded to data_home, from di
# should take a while the first time your run it.
mnist = fetch_mldata('MNIST original', data_home='~/Desktop/EugeneTang/Grad School/Berk

X, Y = mnist.data, mnist.target

# Rescale grayscale values to [0,1].
X = X / 255.0

# Shuffle the input: create a random permutation of the integers between 0 and the numb
# permutation to X and Y.
# NOTE: Each time you run this cell, you'll re-shuffle the data, resulting in a differ
shuffle = np.random.permutation(np.arange(X.shape[0]))
X, Y = X[shuffle], Y[shuffle]

print('data shape: ', X.shape)
print('label shape:', Y.shape)

# Set some variables to hold test, dev, and training data.
test_data, test_labels = X[61000:], Y[61000:]
dev_data, dev_labels = X[60000:61000], Y[60000:61000]
train_data, train_labels = X[:60000], Y[:60000]
mini_train_data, mini_train_labels = X[:1000], Y[:1000]

# My variables (possible labels):
POSSIBLE_LABELS = list(range(0,10)) # digits 0-9
```

```
data shape:  (70000, 784)
label shape: (70000,)
```

(1) Create a 10x10 grid to visualize 10 examples of each digit. Python hints:

- plt.rc() for setting the colormap, for example to black and white
- plt.subplot() for creating subplots
- plt.imshow() for rendering a matrix
- np.array.reshape() for reshaping a 1D feature vector into a 2D matrix (for rendering)

In [3]:
```
def P1(num_examples=10):

    # setup plot
```

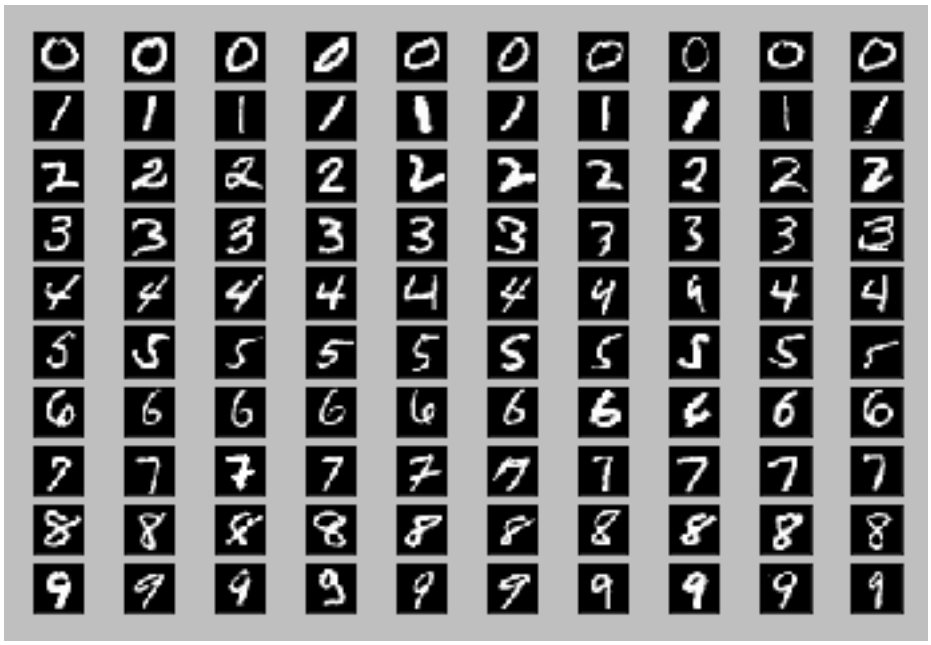
```
    plt.style.use('grayscale')
    plot_index = 1

    # plot ten samples of each label from the mini training data
    for label in POSSIBLE_LABELS:
        label_examples = [d for d, l in zip(mini_train_data, mini_train_labels) if l ==
        ten_random_examples = label_examples[0:10]
        for example in ten_random_examples:
            # update place to plot numbers
            plt.subplot(len(POSSIBLE_LABELS), num_examples, plot_index)
            plot_index += 1

            # plot
            fig = plt.imshow(example.reshape(28,28))

            # remove axes from plots to clean it up
            fig.axes.get_xaxis().set_visible(False)
            fig.axes.get_yaxis().set_visible(False)

P1(10)
```



Each row holds the num_examples examples for each digit

(2) Evaluate a K-Nearest-Neighbors model with k = [1,3,5,7,9] using the mini training set. Report accuracy on the dev set. For k=1, show precision, recall, and F1 for each label. Which is the most difficult digit?

- KNeighborsClassifier() for fitting and predicting

- classification_report() for producing precision, recall, F1 results

```
In [4]: def P2(k_values):
            for k in k_values:
                classifier = KNeighborsClassifier(n_neighbors=k, n_jobs=-1)
                classifier.fit(train_data, train_labels)
                print('Accuracy on dev set when k={}: {}'.format(k, classifier.score(dev_data,

                # print a classification report for k=1
                if k == 1:
                    print('--Printing Classification Report for k=1--')
                    dev_pred = classifier.predict(dev_data)
                    print(classification_report(dev_labels, dev_pred, labels=POSSIBLE_LABELS))

        k_values = [1, 3, 5, 7, 9]
        P2(k_values)
```

```
Accuracy on dev set when k=1: 0.977
--Printing Classification Report for k=1--
             precision    recall  f1-score   support

          0       0.96      1.00      0.98        99
          1       1.00      1.00      1.00       105
          2       0.98      0.96      0.97       102
          3       0.95      0.97      0.96        86
          4       0.99      0.98      0.99       104
          5       0.97      0.97      0.97        91
          6       0.99      0.98      0.98        98
          7       0.99      0.98      0.99       113
          8       0.98      0.93      0.95        96
          9       0.95      1.00      0.98       106

avg / total       0.98      0.98      0.98      1000

Accuracy on dev set when k=3: 0.974
Accuracy on dev set when k=5: 0.973
Accuracy on dev set when k=7: 0.972
Accuracy on dev set when k=9: 0.973
```

ANSWER: If we use the F1-score as the measure of "difficulty", 8 is the most difficult digit. 8 is also the digit with the lowest recall. On the other hand, 9 and 3 are the digits with the lowest precisions.

(3) Using k=1, report dev set accuracy for the training set sizes below. Also, measure the amount of time needed for prediction with each training size.

- time.time() gives a wall clock value you can use for timing operations

```
In [5]: def P3(train_sizes, accuracies):
            print('Training Size | Accuracy | Time Taken (seconds)')
            print('-----------------------------------------------')
            for size in train_sizes:

                # train
                classifier = KNeighborsClassifier(n_neighbors=1, n_jobs=-1)
                train_begin = time.time()
                classifier.fit(train_data[0:size], train_labels[0:size])
                train_end = time.time()

                # evaluate
                accuracy = classifier.score(dev_data, dev_labels)
                print('{:13d} | {:8.2f} | {:20.5f}'.format(size, accuracy, train_end - train_be
                accuracies.append(accuracy)

        train_sizes = [100, 200, 400, 800, 1600, 3200, 6400, 12800, 25000]
        accuracies = []
        P3(train_sizes, accuracies)
```

```
Training Size | Accuracy | Time Taken (seconds)
-----------------------------------------------
          100 |     0.72 |              0.00158
          200 |     0.79 |              0.00197
          400 |     0.84 |              0.00295
          800 |     0.88 |              0.00989
         1600 |     0.90 |              0.03399
         3200 |     0.93 |              0.12993
         6400 |     0.94 |              0.43133
        12800 |     0.96 |              1.51560
        25000 |     0.97 |              5.43309
```

(4) Fit a regression model that predicts accuracy from training size. What does it predict for n=60000? What's wrong with using regression here? Can you apply a transformation that makes the predictions more reasonable?

- Remember that the sklearn fit() functions take an input matrix X and output vector Y. So each input example in X is a vector, even if it contains only a single value.

```
In [6]: def P4():

            lr = LinearRegression()
            lr.fit(np.asarray(train_sizes).reshape(-1, 1), accuracies)
            print('Prediction for n=60000: {:.2f}'.format(lr.predict(np.asarray([60000]).resha
            print('This prediction is > 1, which is strange.')
            lr = LinearRegression()
            lr.fit(np.log(np.asarray(train_sizes)).reshape(-1, 1), accuracies)
            print('New prediction for n=60000: {:.2f}'.format(lr.predict(np.log(np.asarray([60
```

```
P4()
```

```
Prediction for n=60000: 1.24
This prediction is > 1, which is strange.
New prediction for n=60000: 1.03
```

ANSWER: The regression model predicts an accuracy of 1.23 when n=60000. This is unreasonable because accuracies cannot be larger than 1. A regression is not the best choice here because the accuracy can go arbitrarily high depending on n, which we know in reality cannot happen. To try to make the predictions more reasonable, we can try to take the logarithm of the counts, since we might expect that going from 1000->1001 samples has much less of an impact than going from 10->11 samples; a plot of the points (not shown) also shows that it seems to curve logarithmically). While the resulting prediction is still > 1, it is much more reasonable.

Fit a 1-NN and output a confusion matrix for the dev data. Use the confusion matrix to identify the most confused pair of digits, and display a few example mistakes.

- confusion_matrix() produces a confusion matrix

```
In [8]: def P5():
            # train
            classifier = KNeighborsClassifier(n_neighbors=1, n_jobs=-1)
            classifier.fit(train_data, train_labels)

            # evaluate
            dev_predictions = classifier.predict(dev_data)
            cm = confusion_matrix(dev_labels, dev_predictions)
            print('Confusion Matrix')
            print(cm)

            # find most-commonly confused pair of digits
            digits_to_num_mistakes = {}
            for i in range(10):
                for j in range(i+1, 10):
                    digits_to_num_mistakes[(i,j)] = cm[i][j] + cm[j][i]
            hardest_pair = max(digits_to_num_mistakes, key=digits_to_num_mistakes.get)

            # find all mistakes
            print('The pair of digits with the most mistakes are {} and {}'.format(hardest_pai
            mistakes = dev_data[np.intersect1d(np.where(dev_labels == hardest_pair[0]), np.whe
            mistakes = np.append(mistakes, dev_data[np.intersect1d(np.where(dev_labels == hard

            # plot a few example mistakes - in this case, there are only 5, so we print them a
            plt.style.use('grayscale')
            plot_index = 1
            for example in mistakes:
                plt.subplot(1, len(mistakes), plot_index)
                plot_index += 1
```

```
        # plot
        fig = plt.imshow(example.reshape(28,28))

        # remove axes from plots to clean it up
        fig.axes.get_xaxis().set_visible(False)
        fig.axes.get_yaxis().set_visible(False)
```

```
P5()
```

```
Confusion Matrix
[[ 99   0   0   0   0   0   0   0   0   0]
 [  0 105   0   0   0   0   0   0   0   0]
 [  1   0  98   2   0   0   0   1   0   0]
 [  0   0   0  83   0   1   0   0   1   1]
 [  0   0   0   0 102   0   0   0   0   2]
 [  1   0   0   0   0  88   0   0   1   1]
 [  1   0   0   0   1   0  96   0   0   0]
 [  0   0   1   0   0   0   0 111   0   1]
 [  1   0   1   2   0   2   1   0  89   0]
 [  0   0   0   0   0   0   0   0   0 106]]
The pair of digits with the most mistakes are 3 and 8
```

(6) A common image processing technique is to smooth an image by blurring. The idea is that the value of a particular pixel is estimated as the weighted combination of the original value and the values around it. Typically, the blurring is Gaussian – that is, the weight of a pixel's influence is determined by a Gaussian function over the distance to the relevant pixel.

Implement a simplified Gaussian blur by just using the 8 neighboring pixels: the smoothed value of a pixel is a weighted combination of the original value and the 8 neighboring values. Try applying your blur filter in 3 ways: - preprocess the training data but not the dev data - preprocess the dev data but not the training data - preprocess both training and dev data

Note that there are Guassian blur filters available, for example in scipy.ndimage.filters. You're welcome to experiment with those, but you are likely to get the best results with the simplified version I described above.

In [9]:
```
# weights based on a standard Gaussian function (in two dimensions)
weight_d0 = 1/(2*np.pi)                    # points distance 0 away
weight_d1 = 1/(2*np.pi) * np.exp(-(1/2.)) # points distance 1 away
weight_d2 = 1/(2*np.pi) * np.exp(-(2/2.)) # points distance 1,1 away

# normalize weights so they sum to 1 for each pixel
total_weight = weight_d0 + weight_d1 * 4 + weight_d2 * 2
weight_d0_norm = weight_d0 / total_weight
weight_d1_norm = weight_d1 / total_weight
weight_d2_norm = weight_d2 / total_weight

# blur the image by taking a weighted sum of the adjacent pixels
def blur_image(matrix):
    # shift the matrix in the 8 different directions
    top = np.roll(matrix, -1, axis=0)
    top[27,:] = 0 # set border row to 0 to avoid overflow
    bottom = np.roll(matrix, 1, axis=0)
    bottom[0,:] = 0
    left = np.roll(matrix, -1, axis=1)
    left[:,27] = 0
    right = np.roll(matrix, 1, axis=1)
    right[:,0] = 0
    top_left = np.roll(matrix, (-1,-1), axis=(0,1))
    top_left[27,:] = 0
    top_left[:,27] = 0
    top_right = np.roll(matrix, (-1,1), axis=(0,1))
    top_right[27,:] = 0
    top_right[:,0] = 0
    bottom_left = np.roll(matrix, (1,-1), axis=(0,1))
    bottom_left[0,:] = 0
    bottom_left[:,27] = 0
    bottom_right = np.roll(matrix, (1,1), axis=(0,1))
    bottom_right[0,:] = 0
    bottom_right[:,0] = 0

    # create the blurred matrix by taking a weighted sum of the surrounding pixels
    blurred_matrix = matrix * weight_d0_norm + top * weight_d1_norm + bottom * weight_
    right * weight_d1_norm + top_left * weight_d2_norm + top_right * weight_d2_norm + 
    bottom_left * weight_d2_norm + bottom_right * weight_d2_norm
    return blurred_matrix

def preprocess_input(data):
    return [blur_image(m.reshape(28,28)).flatten() for m in data]

def P6():
    classifier = KNeighborsClassifier(n_neighbors=1, n_jobs=-1)
    classifier.fit(preprocess_input(train_data), train_labels)
    accuracy = classifier.score(dev_data, dev_labels)
```

```
    print('Accuracy with preprocessed training data only: {}'.format(accuracy))

    classifier = KNeighborsClassifier(n_neighbors=1, n_jobs=-1)
    classifier.fit(train_data, train_labels)
    accuracy = classifier.score(preprocess_input(dev_data), dev_labels)
    print('Accuracy with preprocessed dev data only: {}'.format(accuracy))

    classifier = KNeighborsClassifier(n_neighbors=1, n_jobs=-1)
    classifier.fit(preprocess_input(train_data), train_labels)
    accuracy = classifier.score(preprocess_input(dev_data), dev_labels)
    print('Accuracy with preprocessed training and dev data: {}'.format(accuracy))


P6()
```

```
Accuracy with preprocessed training data only: 0.979
Accuracy with preprocessed dev data only: 0.973
Accuracy with preprocessed training and dev data: 0.98
```

ANSWER: When we blurred only the training or only the dev data, our results did not look as good as when we blurred both the training and dev data. That is because in the first two cases, our training and dev data were processed in different ways. When only the dev data was preprocessed, it actually hurt the results (0.977 to 0.973). Overall, blurring the training and dev dataset helped increase the overall accuracy of the classifier (from 0.977 to 0.980).

(7) Fit a Naive Bayes classifier and report accuracy on the dev data. Remember that Naive Bayes estimates P(feature | label). While sklearn can handle real-valued features, let's start by mapping the pixel values to either 0 or 1. You can do this as a preprocessing step, or with the binarize argument. With binary-valued features, you can use BernoulliNB. Next try mapping the pixel values to 0, 1, or 2, representing white, grey, or black. This mapping requires MultinomialNB. Does the multi-class version improve the results? Why or why not?

```
In [10]: # map pixel values to 0, 1, or 2 (white, grey, black)
         def preprocess_to_wgb(data):
             n_data = data.copy()
             n_data[data < 1/3.] = 0
             n_data[(data >= 1/3.) & (data < 2/3.)] = 1
             n_data[data >= 2/3.] = 2
             return n_data

         def P7():
             # black/white binarization
             nb = BernoulliNB(binarize=0.5)
             nb.fit(train_data, train_labels)
             accuracy = nb.score(dev_data, dev_labels)
             print('Accuracy of Bernoulli NB with binary arguments: {}'.format(accuracy))
```

```
            # black, white, gray featurization
            nb = MultinomialNB()
            nb.fit(preprocess_to_wgb(train_data), train_labels)
            accuracy = nb.score(preprocess_to_wgb(dev_data), dev_labels)
            print('Accuracy of Multinomial NB with white, grey, black arguments: {}'.format(a


        P7()
```

```
Accuracy of Bernoulli NB with binary arguments: 0.845
Accuracy of Multinomial NB with white, grey, black arguments: 0.826
```

ANSWER: The multi-class does not improve the results in this case. This is likely because the multinomial Naive Bayes is overfit to the training data. While having white, grey, black allows for more granularity, they don't always occur in the same location (people's handwritings are different), so it is likely that as we increased the granularity, the multinomial Naive Bayes overfit.

(8) Use GridSearchCV to perform a search over values of alpha (the Laplace smoothing parameter) in a Bernoulli NB model. What is the best value for alpha? What is the accuracy when alpha=0? Is this what you'd expect?

- Note that GridSearchCV partitions the training data so the results will be a bit different than if you used the dev data for evaluation.

```
In [13]: def P8(alphas):
            gs = GridSearchCV(BernoulliNB(binarize=0.5), alphas)
            gs.fit(train_data, train_labels)
            print('  Alpha | Mean CV score')
            print('-----------------------')
            for score in gs.grid_scores_:
                print('{:8.5f}| {:13.5}'.format(score.parameters["alpha"], score.mean_validati
            return gs

        alphas = {'alpha': [0.0, 0.0001, 0.001, 0.01, 0.1, 0.5, 1.0, 2.0, 10.0]}
        nb = P8(alphas)
```

```
/Users/eugenetang/miniconda3/envs/W207/lib/python3.7/site-packages/sklearn/naive_bayes.py:472:
  'setting alpha = %.1e' % _ALPHA_MIN)
/Users/eugenetang/miniconda3/envs/W207/lib/python3.7/site-packages/sklearn/naive_bayes.py:472:
  'setting alpha = %.1e' % _ALPHA_MIN)
/Users/eugenetang/miniconda3/envs/W207/lib/python3.7/site-packages/sklearn/naive_bayes.py:472:
  'setting alpha = %.1e' % _ALPHA_MIN)


  Alpha | Mean CV score
-----------------------
 0.00000|       0.83712
```

```
 0.00010|        0.8369
 0.00100|        0.8367
 0.01000|       0.83652
 0.10000|       0.83615
 0.50000|       0.83543
 1.00000|       0.83505
 2.00000|       0.83452
10.00000|       0.83245


/Users/eugenetang/miniconda3/envs/W207/lib/python3.7/site-packages/sklearn/naive_bayes.py:472:
  'setting alpha = %.1e' % _ALPHA_MIN)


In [12]: print(nb.best_params_)

{'alpha': 0.0}
```

ANSWER: The best value for alpha is 0. The accuracy when alpha = 0 is 0.83712. This is slightly surprising, because usually smoothing helps, but perhaps in this case this helps because the main benefit of smoothing is when we may have not enough examples of a certain class or feature/class pair (to better approximate the probability). In this case, we have plenty of examples, so smoothing in this case hurt us rather than help.

(9) Try training a model using GuassianNB, which is intended for real-valued features, and evaluate on the dev data. You'll notice that it doesn't work so well. Try to diagnose the problem. You should be able to find a simple fix that returns the accuracy to around the same rate as BernoulliNB. Explain your solution.

Hint: examine the parameters estimated by the fit() method, theta_ and sigma_.

```
In [14]: def P9():
             # Gaussian NB
             nb = GaussianNB()
             nb.fit(train_data, train_labels)
             accuracy = nb.score(dev_data, dev_labels)
             print('Accuracy of Gaussian NB: {}'.format(accuracy))

             # plot thetas and sigmas
             for i in range(len(nb.theta_)):
                 plt.subplot(2,len(nb.theta_),i+1)
                 fig = plt.imshow(nb.theta_[i].reshape(28,28))
                 fig.axes.get_xaxis().set_visible(False)
                 fig.axes.get_yaxis().set_visible(False)

                 plt.subplot(2,len(nb.sigma_),i+1+len(nb.sigma_))
                 fig = plt.imshow(nb.sigma_[i].reshape(28,28))
                 fig.axes.get_xaxis().set_visible(False)
```

```
            fig.axes.get_yaxis().set_visible(False)

        # Fix Gaussian NB by adding random noise to the data
        nb = GaussianNB()
        nb.fit(train_data+np.random.normal(0,0.1,(train_data.shape[0], train_data.shape[1]
        accuracy = nb.score(dev_data+np.random.normal(0,0.1,(dev_data.shape[0], dev_data.s
        print('Accuracy of Updated Gaussian NB: {}'.format(accuracy))
    gnb = P9()
```

```
Accuracy of Gaussian NB: 0.571
Accuracy of Updated Gaussian NB: 0.838
```

ANSWER: Looking at the plots of the mean and standard deviations, we find that because people write digits in different orientations/places, the mean looks very blurry in certain areas, and we can see that some areas have very high standard deviation. The problem, however, is if someone writes the digit, for example 1, in an area that no one in the training data has written. Then since the standard deviation of the "black" regions is very low, the algorithm will automatically reject the idea that the digit is one. To try to accomodate for this, we can try adding random noise so that the standard-deviations are higher, thus being more flexible in allowing for variants in how digits are written.

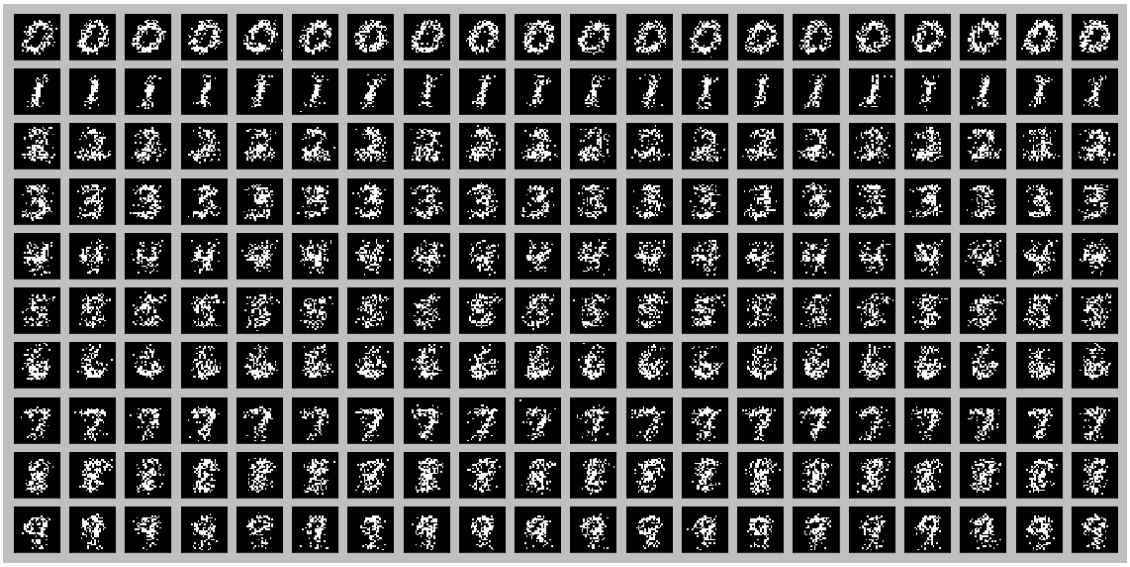
(10) Because Naive Bayes is a generative model, we can use the trained model to generate digits. Train a BernoulliNB model and then generate a 10x20 grid with 20 examples of each digit. Because you're using a Bernoulli model, each pixel output will be either 0 or 1. How do the generated digits compare to the training digits?

- You can use np.random.rand() to generate random numbers from a uniform distribution
- The estimated probability of each pixel is stored in feature_log_prob_. You'll need to use np.exp() to convert a log probability back to a probability.

```
In [15]: def P10(num_examples):
            nb = BernoulliNB(binarize=0.5)
```

```
    nb.fit(train_data, train_labels)
    probabilities = np.exp(nb.feature_log_prob_)
    plt.figure(figsize=(20,10))
    for i in POSSIBLE_LABELS:
        for j in range(num_examples):
            plt.subplot(len(POSSIBLE_LABELS), num_examples, i*num_examples + j + 1)
            fig = plt.imshow(np.array([np.random.rand() < p for p in probabilities[i]]
            # remove axes
            fig.axes.get_xaxis().set_visible(False)
            fig.axes.get_yaxis().set_visible(False)

P10(20)
```



ANSWER: The generated digits generally have the same shape as the training digits, but they appear a lot more grainy because they are binary. Some digits, such as 4,6, and 8 pretty blurry in some photos. This is likely because each pixel is chosen to be 0 or 1 independently, where as in reality, each pixel is not independent (and "8" still has a predetermined shape).

(11) Remember that a strongly calibrated classifier is rougly 90% accurate when the posterior probability of the predicted class is 0.9. A weakly calibrated classifier is more accurate when the posterior is 90% than when it is 80%. A poorly calibrated classifier has no positive correlation between posterior and accuracy.

Train a BernoulliNB model with a reasonable alpha value. For each posterior bucket (think of a bin in a histogram), you want to estimate the classifier's accuracy. So for each prediction, find the bucket the maximum posterior belongs to and update the "correct" and "total" counters.

How would you characterize the calibration for the Naive Bayes model?

```
In [16]: def P11(buckets, correct, total):
             log_buckets = np.log(buckets) # take logarithm to avoid floating-point errors
```

```
            log_buckets = np.insert(log_buckets, 0, -np.infty) # insert infinity for the digi

            # fit model
            nb = BernoulliNB(binarize=0.5, alpha=0.0)
            nb.fit(train_data, train_labels)

            # get relevant data to compute statistics
            dev_log_probs = nb.predict_log_proba(dev_data)
            top_indices = np.argmax(dev_log_probs, axis=1)
            top_log_probs = np.choose(top_indices, dev_log_probs.T)
            correct_predictions = top_indices == dev_labels
            log_prob_bins = np.digitize(top_log_probs, bins=log_buckets, right=True)

            # compute statistics
            correct_bins, correct_counts = np.unique(log_prob_bins[correct_predictions], retu
            total_bins, total_counts = np.unique(log_prob_bins, return_counts=True) # do a co
            bin_to_correct_count = dict(zip(correct_bins, correct_counts))
            bin_to_total_count = dict(zip(total_bins, total_counts))

            # transfer values to output array
            for i in range(len(buckets)):
                correct[i] = bin_to_correct_count.get(i+1, 0) # the bins are one-indexed
                total[i] = bin_to_total_count.get(i+1, 0)

        buckets = [0.5, 0.9, 0.999, 0.99999, 0.9999999, 0.999999999, 0.99999999999, 0.9999999
        correct = [0 for i in buckets]
        total = [0 for i in buckets]

        P11(buckets, correct, total)

        for i in range(len(buckets)):
            accuracy = 0.0
            if (total[i] > 0): accuracy = correct[i] / total[i]
            print('p(pred) <= %.13f    total = %3d    accuracy = %.3f' % (buckets[i], total[i]
```

```
p(pred) <= 0.5000000000000    total =   3    accuracy = 0.333
p(pred) <= 0.9000000000000    total =  39    accuracy = 0.590
p(pred) <= 0.9990000000000    total =  97    accuracy = 0.495
p(pred) <= 0.9999900000000    total =  74    accuracy = 0.635
p(pred) <= 0.9999999000000    total =  64    accuracy = 0.719
p(pred) <= 0.9999999990000    total =  69    accuracy = 0.855
p(pred) <= 0.9999999999900    total =  76    accuracy = 0.895
p(pred) <= 0.9999999999999    total =  81    accuracy = 0.914
p(pred) <= 1.0000000000000    total = 497    accuracy = 0.974


/Users/eugenetang/miniconda3/envs/W207/lib/python3.7/site-packages/sklearn/naive_bayes.py:472:
  'setting alpha = %.1e' % _ALPHA_MIN)
```

ANSWER: The Bernoulli Naive Bayes model has a decent calibration in the sense that as the probability of the prediction goes up, its accuracy also goes up. I would consider it weakly calibrated. It is not strongly calibrated because, e.g. when the predicted probability is 0.9999999999999, the accuracy is still 0.855.

(12) EXTRA CREDIT

Try designing extra features to see if you can improve the performance of Naive Bayes on the dev set. Here are a few ideas to get you started: - Try summing the pixel values in each row and each column. - Try counting the number of enclosed regions; 8 usually has 2 enclosed regions, 9 usually has 1, and 7 usually has 0.

Make sure you comment your code well!

In [20]:
```
##################################################
# append number of enclosed regions to the matrix
##################################################
# find all unseen pixels
def get_unseen_pixel(pixel_seen):
    for i in range(pixel_seen.shape[0]):
        for j in range(pixel_seen.shape[1]):
            if not pixel_seen[i,j]:
                return (i,j)
    return None # all pixels have been seen

# "flood" the given region
def fill_in_region(i,j,pixel_seen):
    if i < 0 or j < 0 or i >= pixel_seen.shape[0] or j >= pixel_seen.shape[1] or pixel
        return
    pixel_seen[i,j] = True

    # recursively look in the four directions
    fill_in_region(i+1,j,pixel_seen)
    fill_in_region(i,j+1,pixel_seen)
    fill_in_region(i-1,j,pixel_seen)
    fill_in_region(i,j-1,pixel_seen)

# find the number of enclosed regions in an image using 0 as a threshold for a bounda
# (anything greater than 0 is seen as a boundary while anything > 0 is seen as empty
def get_num_enclosed_regions(matrix):
    n_matrix_flat = matrix.copy()
    n_matrix = n_matrix_flat.reshape(28,28)
    pixel_seen = n_matrix > 0

    num_regions = 0
    unseen_pixel = get_unseen_pixel(pixel_seen)
    while unseen_pixel is not None:
        fill_in_region(unseen_pixel[0], unseen_pixel[1], pixel_seen)
        unseen_pixel = get_unseen_pixel(pixel_seen)
```

```
                num_regions += 1

            # one of the regions is the border (unenclosed)
            return num_regions-1

        ########################################################
        # append the sum of the rows / columns to the matrix
        ########################################################
        def append_sums(matrix):
            n_matrix_flat = matrix.copy()
            n_matrix = n_matrix_flat.reshape(28,28)

            n_matrix_flat = np.append(n_matrix_flat, np.sum(n_matrix, axis=1)) # add row sums
            n_matrix_flat = np.append(n_matrix_flat, np.sum(n_matrix, axis=0)) # add column s
            return n_matrix_flat

        ########################################################
        # main method
        ########################################################
        # transform the raw data
        def transform_input(data):
            # make a copy to avoid modifying the original
            n_data = data.copy()

            # add the sum of the rows/columns
            n_data = [append_sums(matrix) for matrix in n_data]

            # add the number of enclosed regions (run only on original data (without sums))
            n_data = np.array([np.append(matrix, get_num_enclosed_regions(matrix[0:784])) for

            # add random noise to the Gaussian classifier
            n_data = n_data + np.random.normal(0,0.1,(n_data.shape[0], n_data.shape[1]))

            return n_data

        def P12():
            nb = GaussianNB()
            nb.fit(transform_input(train_data), train_labels)
            accuracy = nb.score(transform_input(dev_data), dev_labels)
            print('Accuracy of NB Model: {}'.format(accuracy))

        P12()
```

```
Accuracy of NB Model: 0.85
```

ANSWER: Summing the pixel values in each row and each column and counting the number of enclosed regions increased the accuracy of the Naive Bayes Gaussian classifier from 0.838 to 0.850.