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/

Finally, if you'd like to get started with Tensorflow, you can read through this tutorial: https://www.tensorflow.org/tutorials/keras/basic_classification. It uses a dataset called "fashion_mnist", which is identical in structure to the original digit mnist, but uses images of clothing rather than images of digits. The number of training examples and number of labels is the same. In fact, you can simply replace the code that loads "fashion_mnist" with "mnist" and everything should work fine.

```
# 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_openml
from sklearn.neighbors import KNeighborsClassifier
from sklearn.metrics import confusion_matrix, r2_score
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.model_selection import GridSearchCV
from sklearn.metrics import classification_report

from typing import * # Used so I can add PEP 484 type annotations

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

```
import sklearn
sklearn.__version__
```

```
'0.22.2.post1'
```

Load the data. Notice that the data gets partitioned into training, development, and test sets. Also, a small subset of the training data called mini_train_data and mini_train_labels gets defined, which you should use in all the experiments below, unless otherwise noted.

```
# Load the digit data from https://www.openml.org/d/554 or from default local locatio
X, Y = fetch_openml(name='mnist_784', return_X_y=True, cache=False)


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

# Shuffle the input: create a random permutation of the integers between 0 and the nu
# permutation to X and Y.
# NOTE: Each time you run this cell, you'll re-shuffle the data, resulting in a diffe
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]
```

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

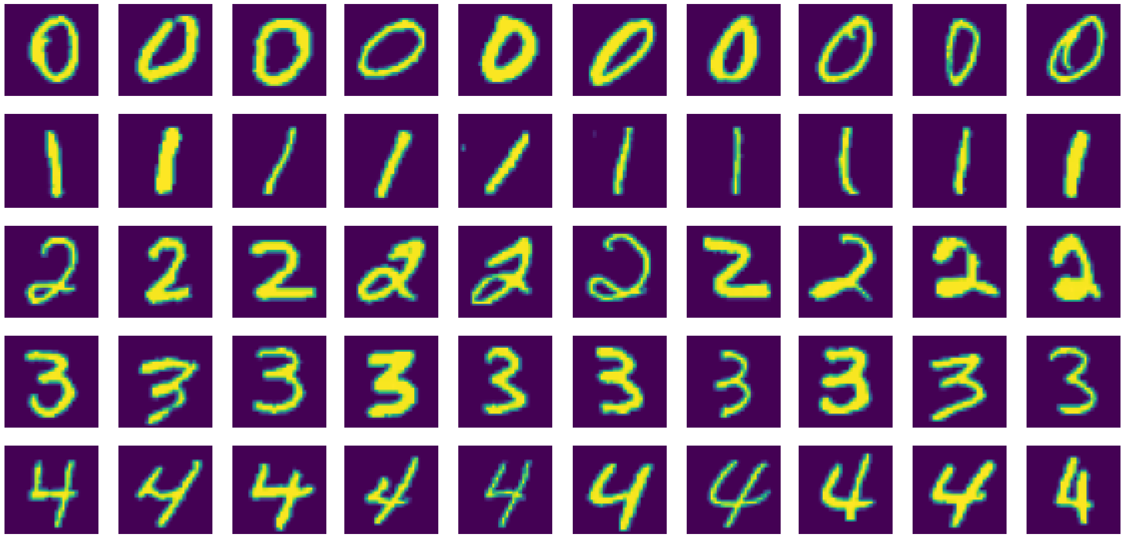
Part 1:

Show a 10x10 grid that visualizes 10 examples of each digit.

Notes:

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

```
#def P1(num_examples=10):

### STUDENT START ###
def P1(num_examples=10):
  digits = {str(i): [] for i in range(10)}
  remaining = [str(i) for i in range(10)]

  # Separating digits into 0-9 labeled examples
  for x, y in zip(X, Y):
    if y in remaining:
      digits[y].append(x)

      if len(digits[y]) == num_examples:
        remaining.remove(y)

    if len(remaining) == 0:
      break

  # Drawing the samples
  f = plt.figure()
  f.set_figwidth(10)
  f.set_figheight(10)
  for d in range(10):
    digit_samples = digits[str(d)]

    # For each sample in the digit, draw it to a subplot
    for i, sample in enumerate(digit_samples):
      plt.subplot(10, num_examples, 10 * d + i + 1)
      plt.imshow(sample.reshape(28, 28))
      plt.axis('off')

### STUDENT END ###

P1(10)
```



Part 2:

Produce k-Nearest Neighbors models with k ∈ [1,3,5,7,9]. Evaluate and show the accuracy of each model. For the 1-Nearest Neighbor model, additionally show the precision, recall, and F1 for each label. Which digit is the most difficult for the 1-Nearest Neighbor model to recognize?

Notes:

- Train on the mini train set.
- Evaluate performance on the dev set.
- You can use `KNeighborsClassifier` to produce a k-nearest neighbor model.
- You can use `classification_report` to get precision, recall, and F1 results.

```
#def P2(k_values):

### STUDENT START ###

def P2(k_values: List[int]):
  for k in k_values:
    # First, train a classifier with the given K
    knn_classifier = KNeighborsClassifier(k)
    knn_classifier.fit(mini_train_data, mini_train_labels)

    # If k == 1, show more detailed performance characteristics
    if k == 1:
      print('k: 1')
      true_labels = dev_labels
      pred_labels = knn_classifier.predict(dev_data)

      print(classification_report(true_labels, pred_labels))

    # Otherwise, just find and print the accuracy
    else:
      print(f'k: {k}\taccuracy: {knn_classifier.score(dev_data, dev_labels)}')
```

```
### STUDENT END ###

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

```
k: 1
              precision    recall  f1-score   support

           0       0.95      0.95      0.95       106
           1       0.89      0.98      0.93       118
           2       0.90      0.79      0.84       106
           3       0.93      0.87      0.90        97
           4       0.91      0.85      0.88        92
           5       0.86      0.88      0.87        88
           6       0.92      0.92      0.92       102
           7       0.85      0.94      0.89       102
           8       0.83      0.77      0.80        94
           9       0.80      0.86      0.83        95

    accuracy                           0.88      1000
   macro avg       0.88      0.88      0.88      1000
weighted avg       0.89      0.88      0.88      1000

k: 3    accuracy: 0.876
k: 5    accuracy: 0.882
k: 7    accuracy: 0.877
k: 9    accuracy: 0.875
```

ANSWER:

Part 3:

Produce 1-Nearest Neighbor models using training data of various sizes. Evaluate and show the performance of each model. Additionally, show the time needed to measure the performance of each model.

Notes:

- Train on subsets of the train set. For each subset, take just the first part of the train set without re-ordering.
- Evaluate on the dev set.
- You can use `KNeighborsClassifier` to produce a k-nearest neighbor model.
- You can use `time.time()` to measure elapsed time of operations.

```
#def P3(train_sizes, accuracies):

### STUDENT START ###

def P3(train_sizes: List[int], accuracies: List[float]):
```

```
    # For each training size, it trains a classifier and prints
    # its performance
    for train_size in train_sizes:
      nn_classifier = KNeighborsClassifier(1)
      nn_classifier.fit(train_data[:train_size], train_labels[:train_size])

      print(f'Trained 1NN Classifier with {train_size} data points')
      start = time.time()
      predictions = nn_classifier.predict(dev_data)
      print(f'Time needed to evaluate model: {time.time() - start} seconds')
      print(classification_report(dev_labels, predictions))

      correct = len([None for l, r in zip(predictions, dev_labels) if l == r])
      accuracies.append(correct / len(dev_labels))

      print('\n====\n')

  ### STUDENT END ###

  train_sizes = [100, 200, 400, 800, 1600, 3200, 6400, 12800, 25600]
  accuracies = []
  P3(train_sizes, accuracies)
```

```
           3       0.92      0.91      0.91        97
           4       0.93      0.96      0.94        92
           5       0.93      0.92      0.93        88
           6       0.93      0.97      0.95       102
           7       0.92      0.96      0.94       102
           8       0.94      0.83      0.88        94
           9       0.92      0.93      0.92        95

    accuracy                           0.94      1000
   macro avg       0.94      0.94      0.94      1000
weighted avg       0.94      0.94      0.94      1000


====

Trained 1NN Classifier with 12800 data points
Time needed to evaluate model: 21.599946975708008 seconds
              precision    recall  f1-score   support

           0       0.98      0.99      0.99       106
           1       0.96      0.99      0.97       118
           2       0.98      0.95      0.97       106
           3       0.95      0.90      0.92        97
           4       0.95      0.95      0.95        92
           5       0.94      0.93      0.94        88
           6       0.95      0.97      0.96       102
           7       0.94      0.98      0.96       102
           8       0.93      0.88      0.91        94
           9       0.93      0.96      0.94        95

    accuracy                           0.95      1000
   macro avg       0.95      0.95      0.95      1000
```

```
weighted avg       0.95      0.95      0.95      1000


====

Trained 1NN Classifier with 25600 data points
Time needed to evaluate model: 43.23791527748108 seconds
              precision    recall  f1-score   support

           0       0.98      0.99      0.99       106
           1       0.96      0.98      0.97       118
           2       0.98      0.94      0.96       106
           3       0.96      0.95      0.95        97
           4       0.97      0.96      0.96        92
           5       0.97      0.95      0.96        88
           6       0.97      0.97      0.97       102
           7       0.94      1.00      0.97       102
           8       0.97      0.90      0.93        94
           9       0.95      0.97      0.96        95

    accuracy                           0.96      1000
   macro avg       0.96      0.96      0.96      1000
weighted avg       0.96      0.96      0.96      1000


====
```

Part 4:

Produce a linear regression model that predicts accuracy of a 1-Nearest Neighbor model given training set size. Show R^2 of the linear regression model. Show the accuracies predicted for training set sizes 60000, 120000, and 1000000. Show a lineplot of actual accuracies and predicted accuracies vs. training set size over the range of training set sizes in the training data. What's wrong with using linear regression here?

Apply a transformation to the predictor features and a transformation to the outcome that make the predictions more reasonable. Show R^2 of the improved linear regression model. Show the accuracies predicted for training set sizes 60000, 120000, and 1000000. Show a lineplot of actual accuracies and predicted accuracies vs. training set size over the range of training set sizes in the training data - be sure to display accuracies and training set sizes in appropriate units.

Notes:

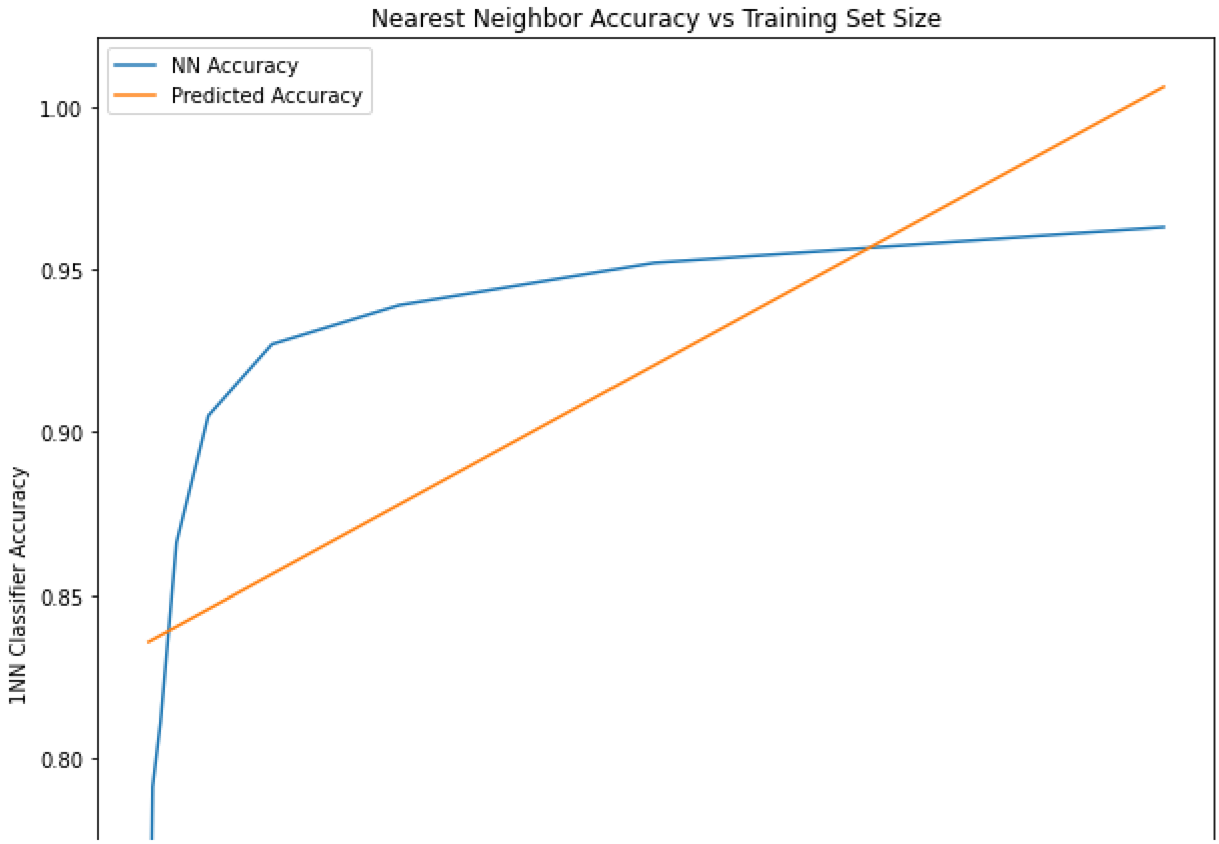
- Train the linear regression models on all of the (transformed) accuracies estimated in Problem 3.
- Evaluate the linear regression models on all of the (transformed) accuracies estimated in Problem 3.
- You can use `LinearRegression` to produce a linear regression model.

- 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.
- Hint re: predictor feature transform: Accuracy increases with training set size logarithmically.
- Hint re: outcome transform: When y is a number in range 0 to 1, then odds(y)=y/(1-y) is a number in range 0 to infinity.

```
# def P4():
### STUDENT START ###

def P4():
  import math
  # First, make a model with no input transformations.
  accuracy_predictor = LinearRegression()
  accuracy_predictor.fit([[size] for size in train_sizes], accuracies)

  predicted_accuracies = accuracy_predictor.predict([[size] for size in train_sizes])

  # Then, show its R² value
  r_sq = r2_score(accuracies, predicted_accuracies)
  print(f'R² score is {r_sq}')

  for train_size in [60000, 120000, 1000000]:
    print(f'Predicted accuracy for training size {train_size}: {accuracy_predictor.pr

  # Drawing the plot...
  f = plt.figure()
  f.set_figwidth(10)
  f.set_figheight(10)
  plt.title('Nearest Neighbor Accuracy vs Training Set Size')
  plt.ylabel('1NN Classifier Accuracy')
  plt.xlabel('Training Size')

  plt.plot(train_sizes, accuracies, label='NN Accuracy')
  plt.plot(train_sizes, predicted_accuracies, label='Predicted Accuracy')
  plt.legend()

  # Next, train a model with log input, which has better performance
  f = plt.figure()
  f.set_figwidth(10)
  f.set_figheight(10)
  plt.title('Nearest Neighbor Accuracy vs Log Training Set Size')
  plt.ylabel('1NN Classifier Accuracy')
  plt.xlabel('Log of Training Size')

  plt.plot(train_sizes, accuracies, label='NN Accuracy')
  accuracy_predictor.fit([[math.log(size)] for size in train_sizes], accuracies)
  predicted_accuracies = accuracy_predictor.predict([[math.log(size)] for size in tra
  plt.plot(train_sizes, predicted_accuracies, label='Predicted Accuracy')
  plt.legend()
```

```
    plt.legend()

    # Showing the improved R² score
    r_sq = r2_score(accuracies, predicted_accuracies)
    print(f'Improved R² score is {r_sq}')

  ### STUDENT END ###

  P4()
```

```
R² score is 0.4177006634161019
Predicted accuracy for training size 60000: 1.2361731707874237
Predicted accuracy for training size 120000: 1.637428053637104
Predicted accuracy for training size 1000000: 7.522499668765751
Improved R² score is 0.9068304252436641
```



ANSWER:

Part 5:

Produce a 1-Nearest Neighbor model and show the confusion matrix. Which pair of digits does the model confuse most often? Show the images of these most often confused digits.

Notes:

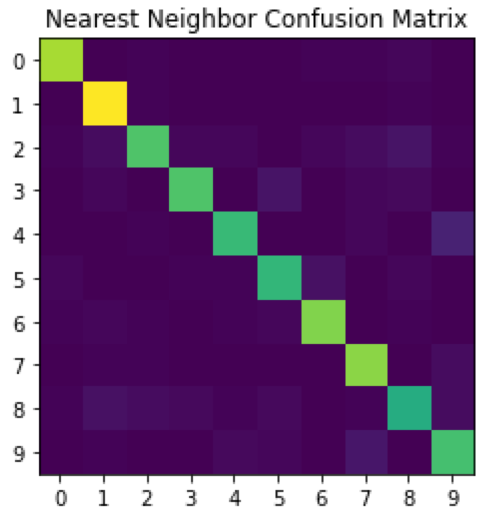
- Train on the mini train set.
- Evaluate performance on the dev set.
- You can use `confusion_matrix()` to produce a confusion matrix.

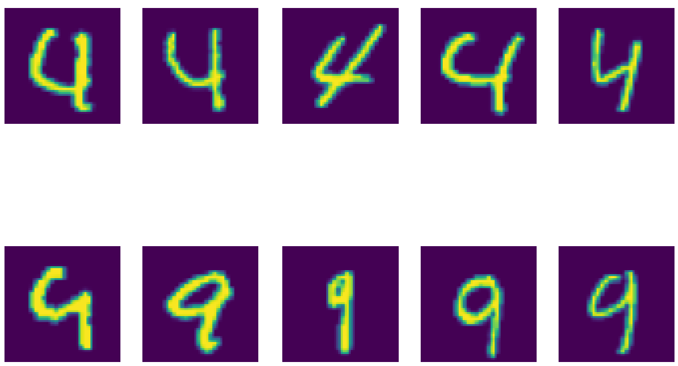
```
#def P5():

### STUDENT START ###

def P5():
  # Train a nearest neighbor classifier
  nn_classifier = KNeighborsClassifier(1)
  nn_classifier.fit(mini_train_data, mini_train_labels)
```

```
# Get its confusion matrix
cm = confusion_matrix(dev_labels, nn_classifier.predict(dev_data))

# Draw the confusion matrix to show its performance visually
plt.imshow(cm.reshape(10, 10))
plt.xticks(list(range(10)))
plt.yticks(list(range(10)))
plt.title('Nearest Neighbor Confusion Matrix')

# This next loop finds the most commonly confused digits
max_confusion = 0
mc_i = None
mc_j = None
for i in range(10):
  for j in range(10):
    if i == j:
      continue

    confusion = cm[i][j]

    if confusion > max_confusion:
      max_confusion = confusion
      mc_i = i
      mc_j = j

print(f'The most confused digits are {mc_i} and {mc_j}, at {max_confusion} confusio

# This is just a helper function to find some sample digits with a given label
def find_examples(X, Y, label, num_examples):
  examples = []
  for x, y in zip(X, Y):
    if y == label:
      examples.append(x)

      if len(examples) >= num_examples:
        return examples

plt.figure()

# Show 5 samples of the first confused digit
num_examples = 5
for i, sample in enumerate(find_examples(dev_data, dev_labels, str(mc_i), num_examp
  plt.subplot(2, num_examples, i + 1)
  plt.imshow(sample.reshape(28, 28))
  plt.axis('off')
# Show 5 samples of the other confused digit
for i, sample in enumerate(find_examples(dev_data, dev_labels, str(mc_j), num_examp
  plt.subplot(2, num_examples, num_examples + i + 1)
  plt.imshow(sample.reshape(28, 28))
  plt.axis('off')
```

```
### STUDENT END ###

P5()
```

```
The most confused digits are 4 and 9, at 11 confusions
```





ANSWER: The most confused digits are 4 and 9

Part 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, i.e., 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 filter by just using the 8 neighboring pixels like this: the smoothed value of a pixel is a weighted combination of the original value and the 8 neighboring values.

Pick a weight, then produce and evaluate four 1-Nearest Neighbor models by applying your blur filter in these ways:

- Do not use the filter
- Filter the training data but not the dev data

- Filter the dev data but not the training data
- Filter both training data and dev data

Show the accuracies of the four models evaluated as described. Try to pick a weight that makes one model's accuracy at least 0.9.

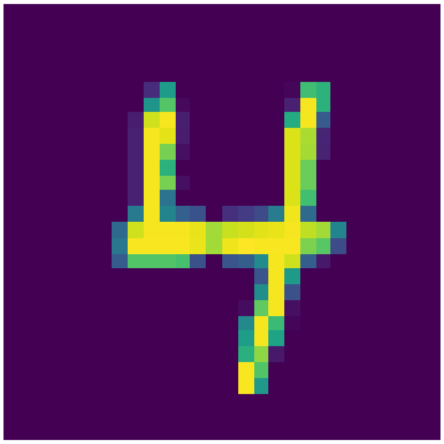
Notes:

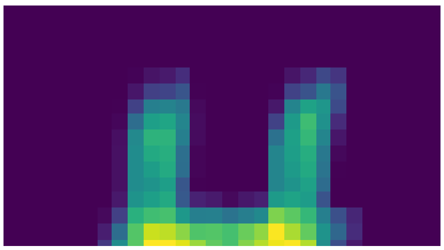
- Train on the (filtered) mini train set.
- Evaluate performance on the (filtered) dev set.
- There are other Guassian blur filters available, for example in `scipy.ndimage.filters`. You are welcome to experiment with those, but you are likely to get the best results with the simplified version described above.

```
#def P6():

### STUDENT START ###

def P6():
  from numpy import ndarray
  from scipy.ndimage.filters import gaussian_filter

  # I've found that the harder the training set is, the better the classifier
  # performs. I've designed this blur matrix to make '4' and '9' as hard to
  # differentiate as possible, and that seemed to make the model perform best.
  blur_matrix = ndarray((3, 3), buffer=np.array([
    1.0, 1.0, 3.0,
    2.0, 0.0, 2.0,
    3.0, 1.0, 1.0
  ]))

  # Helper function to blur a single image
  def blur(image, blur_matrix: ndarray):
    # return gaussian_filter(image, sigma=1.3)

    assert image.shape in [(784, ), (28, 28)] # Only supporting these shapes
    assert blur_matrix.shape == (3, 3)
    image = image.reshape(28, 28)
    # This is simpler than having 2 delta i/j loops
    deltas = [(-1, -1), (-1, 0), (-1, 1), (0, -1), (0, 0), (0, 1), (1, -1), (1, 0), (
    # Turns the gaussian matrix into a simple array of weights
    weights = blur_matrix.reshape((9,))

    result = ndarray((28, 28), buffer=np.zeros((28, 28)))

    # For each pixel in the result, generate it with a weighted average of its neighb
    for i in range(1, 27):
      for j in range(1, 27):
        inputs = [image[i + di, j + dj] for di, dj in deltas]
```

```
          result[i, j] = np.average(inputs, weights=weights)

      return result.reshape((784,))

    # Helper function to blur an entire dataset
    def blur_dataset(data):
      return [blur(d, blur_matrix) for d in data]

    # This just demonstrates one normal and blurred image
    plt.imshow(X[1].reshape(28, 28))
    plt.axis('off')
    plt.figure()
    plt.imshow(blur(X[1].reshape(28, 28), blur_matrix).reshape(28, 28))
    plt.axis('off')

    # Blurs the mini training / test sets
    start = time.time()
    print('Blurring dev...')
    blurred_dev = blur_dataset(dev_data)
    print(f'Done blurring dev in {time.time() - start} sec')

    start = time.time()
    print('Blurring training...')
    blurred_train = blur_dataset(mini_train_data)
    print(f'Done blurring training in {time.time() - start} sec')

    print(time.time() - start)

    # Trains both unblurred and blurred classifiers
    unblurred_cls = KNeighborsClassifier(1)
    unblurred_cls.fit(mini_train_data, mini_train_labels)

    blurred_cls = KNeighborsClassifier(1)
    blurred_cls.fit(blurred_train, mini_train_labels)

    # Shows performance for [classifiers] x [datasets]
    print(f'Unblurred classifier, Unblurred dataset: {unblurred_cls.score(dev_data, dev
    print(f'Unblurred classifier,   Blurred dataset: {unblurred_cls.score(blurred_dev,
    print(f'  Blurred classifier, Unblurred dataset: {blurred_cls.score(dev_data, dev_l
    print(f'  Blurred classifier,   Blurred dataset: {blurred_cls.score(blurred_dev, de

  ### STUDENT END ###

  P6()
```

```
Blurring dev...
Done blurring dev in 17.066384315490723 sec
Blurring training...
Done blurring training in 16.9928617477417 sec
16.993298768997192
Unblurred classifier, Unblurred dataset: 0.884
Unblurred classifier,   Blurred dataset: 0.87
  Blurred classifier, Unblurred dataset: 0.914
  Blurred classifier,   Blurred dataset: 0.907
```





Part 7:

Produce two Naive Bayes models and evaluate their performances. Recall that Naive Bayes estimates P(feature|label), where each label is a categorical, not a real number.

For the first model, map pixel values to either 0 or 1, representing white or black - you should pre-process the data or use `BernoulliNB`'s `binarize` parameter to set the white/black separation threshold to 0.1. Use `BernoulliNB` to produce the model.

For the second model, map pixel values to either 0, 1, or 2, representing white, gray, or black - you should pre-process the data, seting the white/gray/black separation thresholds to 0.1 and 0.9. Use `MultinomialNB` to produce the model.

Show the Bernoulli model accuracy and the Multinomial model accuracy.

Notes:

- Train on the mini train set.
- Evaluate performance on the dev set.
- `sklearn`'s Naive Bayes methods can handle real numbers, but for this exercise explicitly do the mapping to categoricals.

Does the multinomial version improve the results? Why or why not?

```
#def P7():

### STUDENT START ###

def P7():
  # Helper function to quantize given images
  def discretize(vector):
    nda = np.ndarray((784,))

    for i, v in enumerate(vector):
      nda[i] = 0 if v < 0.1 else 1 if v < 0.9 else 2

    return nda

  # Helper function to quantize an entire dataset
  def discretize_dataset(vectors):
    return [discretize(v) for v in vectors]

  # Trains and tests a Bernoulli classifier
  nbn = BernoulliNB(binarize=0.1)
  nbn.fit(mini_train_data, mini_train_labels)

  print(f'Binary NB score: {nbn.score(dev_data, dev_labels)}')

  # Trains and tests a Multinomial classifier with quantized input
  mbn = MultinomialNB()
  discretized_train = discretize_dataset(mini_train_data)
  mbn.fit(discretized_train, mini_train_labels)

  print(f'Multinomial NB score: {mbn.score(discretize_dataset(dev_data), dev_labels)}

### STUDENT END ###

P7()
```

```
Binary NB score: 0.814
Multinomial NB score: 0.807
```

ANSWER:

Part 8:

Search across several values of the LaPlace smoothing parameter (alpha) to find its effect on a Bernoulli Naive Bayes model's performance. Show the accuracy at each alpha value.

Notes:

- Set binarization threshold to 0.
- Train on the mini train set.
- Evaluate performance by 5-fold cross-validation.
- Use `GridSearchCV(..., ..., cv=..., scoring='accuracy', iid=False)` to vary alpha and evaluate performance by cross-validation.
- Cross-validation is based on partitions of the training data, so results will be a bit different than if you had used the dev set to evaluate performance.

What is the best value for alpha? What is the accuracy when alpha is near 0? Is this what you'd expect?

```
#def P8(alphas):

### STUDENT START ###

def P8(alphas):
  bnb = BernoulliNB(binarize=0.0)

  # This for loop isn't required but shows the effect of alpha on performance
  for alpha in [1.0e-10, 0.0001, 0.001, 0.01, 0.1, 0.5, 1.0, 2.0, 10.0]:
    nbnb = BernoulliNB(binarize=0.0, alpha=alpha)
    nbnb.fit(mini_train_data, mini_train_labels)
    print(f'alpha: {alpha}\tscore: {nbnb.score(dev_data, dev_labels)}')

  # Create the required validator and fit it
  x_validator = GridSearchCV(bnb, alphas, cv=5, scoring='accuracy', iid=False)
  x_validator.fit(mini_train_data, mini_train_labels)

  return x_validator

### STUDENT END ###

alphas = {'alpha': [1.0e-10, 0.0001, 0.001, 0.01, 0.1, 0.5, 1.0, 2.0, 10.0]}
nb = P8(alphas)
print()
print("Best alpha = ", nb.best_params_)
```

```
alpha: 1e-10    score: 0.816
alpha: 0.0001   score: 0.823
alpha: 0.001    score: 0.823
alpha: 0.01     score: 0.824
alpha: 0.1      score: 0.822
alpha: 0.5      score: 0.816
alpha: 1.0      score: 0.809
alpha: 2.0      score: 0.811
alpha: 10.0     score: 0.779

Best alpha =  {'alpha': 0.001}
/usr/local/lib/python3.7/dist-packages/sklearn/model_selection/_search.py:823: F
  "removed in 0.24.", FutureWarning
```

ANSWER:

Part 9:

Produce a model using Guassian Naive Bayes, which is intended for real-valued features, and evaluate performance. You will notice that it does not work so well. Diagnose the problem and apply a simple fix so that the model accuracy is around the same as for a Bernoulli Naive Bayes model. Show the model accuracy before your fix and the model accuracy after your fix. Explain your solution.

Notes:

- Train on the mini train set.
- Evaluate performance on the dev set.
- Consider the effects of theta and sigma. These are stored in the model's `theta_` and `sigma_` attributes.

```
#def P9():

### STUDENT END ###

def P9():
  # Poorly performing Gaussian NB classifier
  gnb = GaussianNB()
  gnb.fit(mini_train_data, mini_train_labels)

  print(f'With no tuning, Gaussian NB scores {gnb.score(dev_data, dev_labels)}')

  # Improved Gaussian NB classifier, with a variance smoothing determined by trial-an
  gnb = GaussianNB(var_smoothing=0.06765)
  gnb.fit(mini_train_data, mini_train_labels)

  print(f'After tuning var_smoothing, Gaussian NB scores {gnb.score(dev_data, dev_lab

### STUDENT END ###

P9()
```

```
With no tuning, Gaussian NB scores 0.593
After tuning var_smoothing, Gaussian NB scores 0.82
```

ANSWER:

Part 10:

Because Naive Bayes produces a generative model, you can use it to generate digit images.

Produce a Bernoulli Naive Bayes model and then use it to generate a 10x20 grid with 20 example images of each digit. Each pixel output should be either 0 or 1, based on comparing some randomly generated number to the estimated probability of the pixel being either 0 or 1. Show the grid.

Notes:

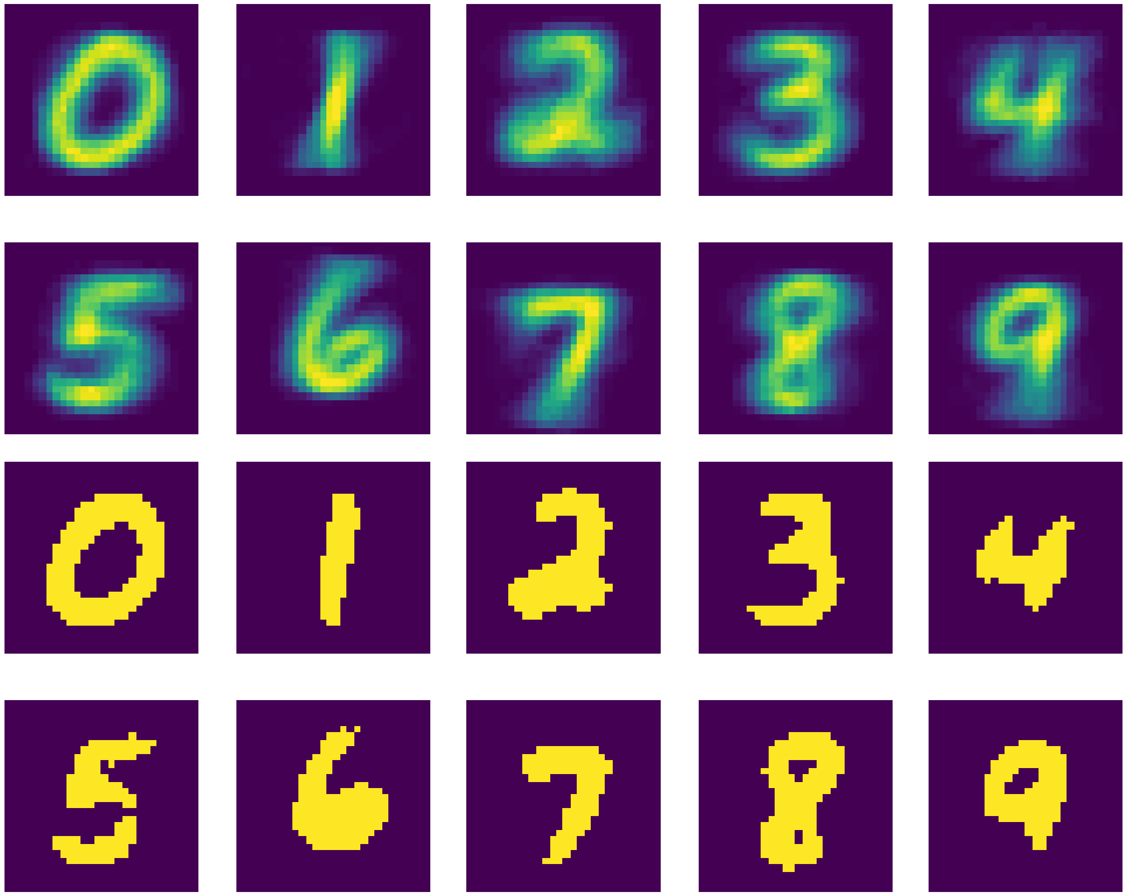
- You can use np.random.rand() to generate random numbers from a uniform distribution.
- The estimated probability of each pixel being 0 or 1 is stored in the model's `feature_log_prob_` attribute. You can use `np.exp()` to convert a log probability back to a probability.

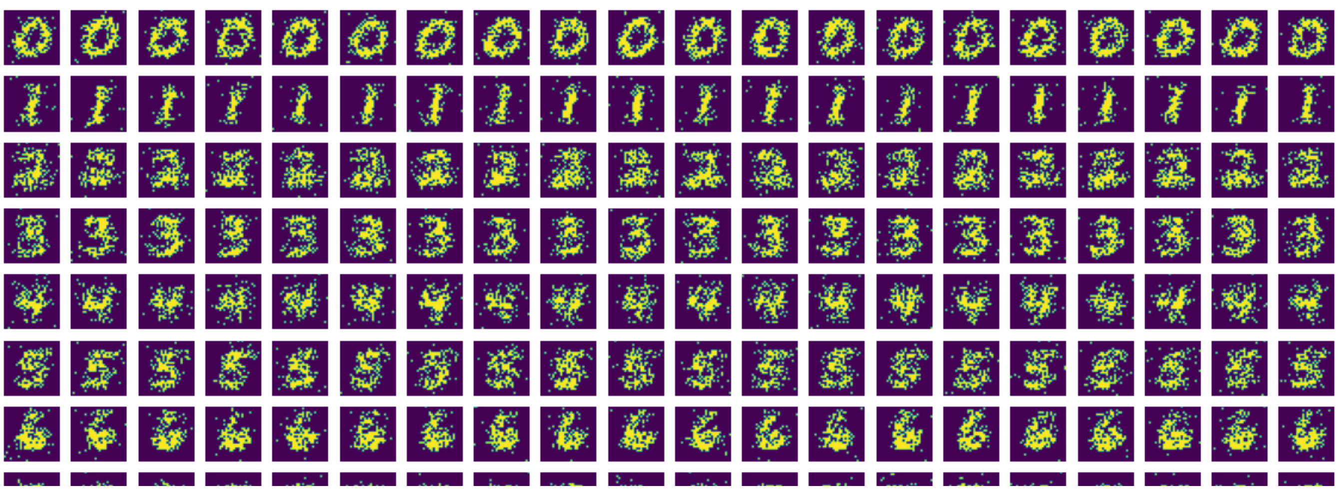
How do the generated digit images compare to the training digit images?

```
#def P10(num_examples):

### STUDENT START ###

def P10(num_examples: int):
  bnb = BernoulliNB(binarize=0.1)
  bnb.fit(mini_train_data, mini_train_labels)

  # Because I wanted to show various generated images, I made this
  # helper function take a section argument, which is a function that
  # maps probabilities to pixel values.
  def generate_image(feature_log_prob, pix_mapper):
    nda = np.ndarray((784,))

    for i, log_prob in enumerate(feature_log_prob):
      p = np.exp(log_prob)
      # nda[i] = 1 if np.random.rand() < p else 0
      nda[i] = pix_mapper(p)

    return nda

  # The weighted random pixel map for the problem
  def weighted_random(p):
    return 1 if np.random.rand() < p else 0

  # Showing a quantized "ground truth"
  def quantize(p):
    return 1 if p > 0.5 else 0

  # Showing the raw "ground truth"
  def noop(p):
    return p
```

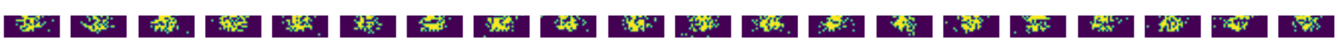
```
    log_features = bnb.feature_log_prob_

    print('Showing the Bernoulli "ground truth", both as the raw probability and quanti

    # First demonstrating the pixel stats the classifier found
    f = plt.figure()
    f.set_figwidth(10)
    f.set_figheight(4)
    for i in range(10):
      plt.subplot(2, 5, 1 + i)
      plt.imshow(generate_image(log_features[i], noop).reshape(28, 28))
      plt.axis('off')
    f = plt.figure()
    f.set_figwidth(10)
    f.set_figheight(4)
    for i in range(10):
      plt.subplot(2, 5, 1 + i)
      plt.imshow(generate_image(log_features[i], quantize).reshape(28, 28))
      plt.axis('off')

    f = plt.figure()
    f.set_figwidth(20)
    f.set_figheight(10)

    # Now, drawing 20 randomly generated, noisy samples of each digit
    for i in range(10):
      for j in range(num_examples):
        plt.subplot(10, 20, 1 + 20 * i + j)
        plt.imshow(generate_image(log_features[i], weighted_random).reshape(28, 28))
        plt.axis('off')

  ### STUDENT END ###

  P10(20)
```

```
Showing the Bernoulli "ground truth", both as the raw probability and quantized
```





ANSWER:



Part 11:

Recall 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 probability of the predicted class is 90% than when it is 80%. A poorly calibrated classifier has no positive correlation between posterior probability and accuracy.

Produce a Bernoulli Naive Bayes model. Evaluate performance: partition the dev set into several buckets based on the posterior probabilities of the predicted classes - think of a bin in a histogram- and then estimate the accuracy for each bucket. So, for each prediction, find the bucket to which the maximum posterior probability belongs, and update "correct" and "total" counters accordingly. Show the accuracy for each bucket.

Notes:

- Set LaPlace smoothing (alpha) to the optimal value (from part 8).
- Set binarization threshold to 0.
- Train on the mini train set.
- Evaluate perfromance on the dev set.

```
#def P11(buckets, correct, total):

### STUDENT START ###

def P11(buckets, correct, total):
  # First, pick the best alpha found earlier
  alpha = nb.best_params_['alpha']
  print(f'Using alpha = {alpha}')

  # Now, train a classifier with that hyperparameter
  bnb = BernoulliNB(alpha=alpha, binarize=0)
  bnb.fit(mini_train_data, mini_train_labels)

  # Helper function to sort max probabilities into buckets
  def find_bucket(prob):
    for i, upper in enumerate(buckets):
      if upper >= prob:
        return i

  # For each item in the test set, find its confidence bucket and count it
  for probs, y in zip(bnb.predict_proba(dev_data), dev_labels):
    max_prob = max(probs)
    bucket = find_bucket(max_prob)
    pred_y = str(list(probs).index(max_prob))

    total[bucket] += 1

    if pred_y == y:
      correct[bucket] += 1

### STUDENT END ###

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)
```

```
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) is %.13f to %.13f    total = %3d    accuracy = %.3f' % (0 if i==0
```

```
 Using alpha = 0.001
 p(pred) is 0.0000000000000 to 0.5000000000000    total =   0    accuracy = 0.000
 p(pred) is 0.5000000000000 to 0.9000000000000    total =  31    accuracy = 0.355
 p(pred) is 0.9000000000000 to 0.9990000000000    total =  67    accuracy = 0.433
 p(pred) is 0.9990000000000 to 0.9999900000000    total =  59    accuracy = 0.458
 p(pred) is 0.9999900000000 to 0.9999999000000    total =  46    accuracy = 0.652
 p(pred) is 0.9999999000000 to 0.9999999990000    total =  62    accuracy = 0.774
 p(pred) is 0.9999999990000 to 0.9999999999900    total =  33    accuracy = 0.788
 p(pred) is 0.9999999999900 to 0.9999999999999    total =  43    accuracy = 0.791
 p(pred) is 0.9999999999999 to 1.0000000000000    total = 659    accuracy = 0.938
```

ANSWER:

Part 12 EXTRA CREDIT:

Design new features to see if you can produce a Bernoulli Naive Bayes model with better performance. Show the accuracy of a model based on the original features and the accuracy of the model based on the new features.

Here are a few ideas to get you started:

- Try summing or averaging the pixel values in each row.
- Try summing or averaging the pixel values in each column.
- Try summing or averaging the pixel values in each square block. (pick various block sizes)
- Try counting the number of enclosed regions. (8 usually has 2 enclosed regions, 9 usually has 1, and 7 usually has 0)

Notes:

- Train on the mini train set (enhanced to comprise the new features).
- Evaulate performance on the dev set.
- Ensure that your code is well commented.

```
#def P12():

### STUDENT START ###


### STUDENT END ###

#P12()
```

```
#P12()
```

check 1s completed at 5:34 PM