PROJECT 4 REPORT

SAM LEVINE

1. TASK 4A: Naïve Bayes (30pt)

1.1. Task 4A-I: Calculate Prior Probabilities.

Our first task was implementing the `calculate_class_priors(y_train)` function so it calculates and returns the prior probabilities $P(Y)$ of each class in the training dataset. This is the approach used:

1. Iterate through y_train to count the occurrences of each class.
2. Then divide the count of each class by the total number of samples to get the probability.
3. Store these probabilities in a dictionary, where each key is a class label and each value is the corresponding prior probability.

We've taken advatange of `np.unique` and `np.sum` for efficiency's sake.

```
def calculate_class_priors(y_train):
    """
    Calculates the prior probabilities P(Y) for each class in the training
set.

    For example, if we have a vector y_train = [0, 1, 0, 0, 1, 0]
    composed of two classes 0 and 1, the prior probabilities of each class
    that you would return are:
    {
        0: 0.666,
        1: 0.333,
    }

    Parameters:
    - y_train: Array of training labels

    Returns:
    - Dictionary of prior probabilities for each class
    """
    priors = {}
    ### Your code starts here
    total_samples = len(y_train)

    for label in np.unique(y_train):
        priors[label] = np.sum(y_train == label) / total_samples
    ### End of your code
    return priors
```

1.2. TASK 4A-II: Calculate class-conditional density parameters.

For this task we implemented `calculate_gaussian_density_params(features, labels)` to calculate the parameters of the class-conditional data density $P(X|Y)$ of observing each feature given each class. This is the approach used:

1. Identify the unique classes in the labels.

2. For each class, filter the features that correspond to that class.
3. Calculate the mean and variance for each feature within this subset.
4. Store these parameters in a dictionary, structured such that each class key maps to a list of tuples, with each tuple containing the mean and variance of a feature given the class.

```
def calculate_gaussian_density_params(features, labels):
    """
    Calculates the likelihood P(X|Y) for each feature given a class.

    For example, if we had one feature X_1 and a target Y \in {0, 1},
    we would return:
    {
        0: [(mean(X_1 | Y=0), var(X_1 | Y=0))],
        1: [(mean(X_1 | Y=1), var(X_1 | Y=1))],
    }

    Parameters:
    - features: Array of features in the training set
    - labels: Array of labels corresponding to the features

    Returns:
    - Dictionary of Gaussian density parameters for each feature given each
class
    """
    likelihood = {}
    ### Your code starts here
    classes = np.unique(labels)

    for cls in classes:
        cls_features = features[labels == cls]
            params = [(np.mean(feature), np.var(feature)) for feature in
zip(*cls_features)]
        likelihood[cls] = params

    ### End of your code
    return likelihood
```

1.3. **TASK 4A-III: Implement the Classifier.** For this task we implemented the `naive_bayes_classifier(X_train, y_train, X_test)` function. It takes the **priors** and **likelihoods** calculated in the previous tasks to classify each sample in the test set. The implementation follows these steps:

1. Calculate class prior probabilities using the training set labels (`y_train`).
2. Calculate class-conditional density parameters (mean and variance for each feature given each class) from the training data.
3. Classify each sample in the test set (X__test) by computing the posterior probability of each class given the sample and then predicting the class with the highest posterior probability. This is done by applying Bayes' theorem.

```python
def naive_bayes_classifier(X_train, y_train, X_test):
    """
   Classifies each sample in the test set based on the Naive Bayes algorithm.

    Steps:
    - Compute class prior probabilities using the training set labels.
    - Compute class-conditional density parameters from training data.
    - For each sample x in the test set:
         - Use the density params (mu_i, var_i) learned from training data
to compute
          the log-likelihood of observing feature x_i.
        - Apply Bayes theorem to compute posterior class probabilities P(y|
x)
          from the feature log-likelihood and prior log-likelihood.
          You will use log-likelihood here to avoid underflow.

    Parameters:
    - X_train: Training set features
    - y_train: Training set labels
    - X_test: Test set features

    Returns:
    - Predicted classes for the test set
    """
    predictions = []

    ### Your code starts here
    priors = calculate_class_priors(y_train)
    likelihood = calculate_gaussian_density_params(X_train, y_train)
    for x in X_test:
        log_posterior = {}
        for cls, params in likelihood.items():
            log_likelihood = np.log(priors[cls])
            for i, feature in enumerate(x):
                mean, var = params[i]
                log_likelihood += np.log(gaussian_pdf(feature, mean, var))
            log_posterior[cls] = log_likelihood

        predicted_class = max(log_posterior, key=log_posterior.get)
        predictions.append(predicted_class)

    ### End of your code
    return predictions
```

1.3.1. *TASK 4A-III Results.*

We had a test accuracy of 78.0% after running the classifier.

```python
predictions = naive_bayes_classifier(X_train, y_train, X_test)
accuracy = evaluate_accuracy(predictions, y_test)
print(f"The test accuracy is {accuracy * 100}%.")
```

```
The test accuracy is 78.0%.
```

2. TASK 4B: Image Completion with Mixture of Bernoullis and EM (70pt + 20 bonus pt)

2.1. TASK 4B-I: Parameter Learning via EM (35pt).

In this task we derived and implemented the M-step update rules for Θ amd π.

$$\sum_{i=1}^{N}\sum_{k=1}^{K} r_k^{(i)}\left[\log p\big(z^{\{(i)\}}=k\big)+\log p\big(x^{(i)}|z^{(i)}=k\big)\right]+\log p(\pi)+\log p(\Theta)$$
$$=\sum_{i=1}^{N}\sum_{k=1}^{K} r_k^{(i)}\left[\log \pi_k+\sum_{j=1}^{D} x_j^{(i)}\log\theta_{k,j}+\left(1-x_j^{(i)}\right)\log\big(1-\theta_{k,j}\big)\right]+$$
$$\sum_{k=1}^{K}(a_k-1)\log\pi_k+\sum_{k=1}^{K}\sum_{j=1}^{D}\left[(a-1)\log\theta_{k,j}+(b-1)\log\big(1-\theta_{k,j}\big)\right]+C$$

Note: In order to make things easier to write, I'm using θ has shorthand for $\theta_{k,j}$.

2.1.1. *Derivation & Implementation for Θ.*

Take the derivative wrt θ

$$\frac{\partial}{\partial\theta}=\sum_{i=1}^{N} r_k^{(i)}\left[x_j^{(i)}\frac{1}{\theta}+\left(1-x_j^{(i)}\right)\left(\frac{1}{\theta-1}\right)\right]+(a-1)\frac{1}{\theta}+(b-1)\frac{1}{\theta-1}$$
$$=\frac{1}{\theta}\left(\sum_{i=1}^{N}\left[r_k^{(i)}x_j^{(i)}\right]+(a-1)\right)+\frac{1}{\theta-1}\left(\sum_{i=1}^{N}\left[r_k^{(i)}\right]-\sum_{i=1}^{N}\left[r_k^{(i)}x_j^{(i)}\right]+(b-1)\right)$$

Setting this to zero (as instructed) and multiplying both sides by $\theta(\theta-1)$ gives us:

$$0=(\theta-1)\left(\sum_{i=1}^{N}\left[r_k^{(i)}x_j^{(i)}\right]+(a-1)\right)+\theta\left(\sum_{i=1}^{N}\left[r_k^{(i)}\right]-\sum_{i=1}^{N}\left[r_k^{(i)}x_j^{(i)}\right]+(b-1)\right)$$

This gives us:

$$\theta_{k,j}=\frac{\sum_{i=1}^{N}\left[r_k^{(i)}x_j^{(i)}\right]+(a-1)}{\sum_{i=1}^{N}\left[r_k^{(i)}x_j^{(i)}\right]+(a-1)+\sum_{i=1}^{N}\left[r_k^{(i)}\right]-\sum_{i=1}^{N}\left[r_k^{(i)}x_j^{(i)}\right]+(b-1)}$$
$$=\frac{\sum_{i=1}^{N}\left[r_k^{(i)}x_j^{(i)}\right]+a-1}{\sum_{i=1}^{N}\left[r_k^{(i)}\right]+a+b-2}$$

```
def update_theta(self, X, R):
    """Compute the update for the Bernoulli parameters in the M-step of the
E-M algorithm.
```

```
    You should derive the optimal value of theta (the one which maximizes
the expected log
     probability) by setting the partial derivatives to zero. You should
implement this in
    terms of NumPy matrix and vector operations, rather than a for loop."""

    ######################## Your code here ########################
    N, D = X.shape
    K = R.shape[1]
    a_pixels = self.prior.a_pixels
    b_pixels = self.prior.b_pixels

    numerator = np.dot(R.T, X) + (a_pixels - 1)
    denominator = np.sum(R, axis=0)[:, None] + (a_pixels + b_pixels - 2)
    # self.params.theta = numerator / denominator
    theta = numerator / denominator
    return theta
    #################################################################
```

2.1.2. *Derivation & Implementation for π.*

Take the derivative wrt π. After some reading, we found that we could use a Lagrange multiplier here since we need to account for the condition $\sum_{k=1}^{K} \pi_k = 1$.

Let $F_\lambda = F + \lambda\left(\sum_{k=1}^{K} [\pi_k] - 1\right)$ where F is the original equation above.

$$\frac{\partial F_\lambda}{\partial \pi_k} = \sum_{i=1}^{N} r_k^{(i)} \frac{1}{\pi_k} + (a_k - 1)\frac{1}{\pi_k} + \lambda$$

Setting this to zero gives us:

$$\pi_k = \frac{(a_k - 1) + \sum_{i=1}^{N}\left[r_k^{(i)}\right]}{\lambda}$$

We know π_k sums to 1, giving us:

$$\pi_k = \frac{(a_k - 1) + \sum_{i=1}^{N}\left[r_k^{(i)}\right]}{\sum_{k=1}^{K}\left[(a_k - 1) + \sum_{i=1}^{N}\left[r_k^{(i)}\right]\right]} = \frac{(a_k - 1) + \sum_{i=1}^{N}\left[r_k^{(i)}\right]}{N + \sum_{k=1}^{K}(a_k - 1)}$$

```
def update_pi(self, R):
    """Compute the update for the mixing proportions in the M-step of the
E-M algorithm.
    You should derive the optimal value of pi (the one which maximizes the
expected log
     probability) by setting the partial derivatives of the Lagrangian to
zero. You should
    implement this in terms of NumPy matrix and vector operations, rather
than a for loop."""
```

```
    ######################## Your code here ########################
    N, K = R.shape
    a_mix = self.prior.a_mix
    self.params.pi = (np.sum(R, axis=0) + a_mix - 1) / (N + K * (a_mix - 1))
    pi = (np.sum(R, axis=0) + a_mix - 1) / (N + K * (a_mix - 1))
    return pi
    #################################################################
```

Running `check_m_step()` yielded:

```
The theta update seems OK.
The pi update seems OK.
```

2.2. TASK 4B-II: Posterior inference (30pt).

In this task we derived the posterior probabilitiy distribution $p(z|x)$, and then implemented `Model.compute_posterior`.

In our following notation, we represent partial observations in terms of $m_j^{(i)}$ where $m_j^{(i)} = 1$ if the jth pixel of the ith image is observed, and 0 otherwise.

$$p(z=k|x) = \frac{p(x|z=k)p(z=k)}{p(x)} = \frac{\pi_k \prod_{j=1}^{D} \theta_{k,j}^{m_j^{(i)} x_j} \left(1 - \theta_{k,j}^{m_j^{(i)}(1-x_j)}\right)}{\sum_{l=1}^{K} \pi_l \prod_{j=1}^{D} \theta_{l,j}^{m_j^{(i)} x_j} \left(1 - \theta_{l,j}^{m_j^{(i)}(1-x_j)}\right)}$$

We implemented this in terms of log probabilities for stability:

```
def compute_posterior(self, X, M=None):
    """Compute the posterior probabilities of the cluster assignments given
the observations.
    This is used to compute the E-step of the E-M algorithm. It's also used
in computing the
   posterior predictive distribution when making inferences about the hidden
part of the image.
   It takes an optional parameter M, which is a binary matrix the same size
as X, and determines
    which pixels are observed. (1 means observed, and 0 means unobserved.)
    Your job is to compute the variable log_p_z_x, which is a matrix whose
(i, k) entry is the
    log of the joint proability, i.e.
         log p(z^(i) = k, x^(i)) = log p(z^(i) = k) + log p(x^(i) | z^(i) =
k)
   Hint: the solution is a small modification of the computation of log_p_z_x
in
    Model.log_likelihood.
    """

    if M is None:
        M = np.ones(X.shape, dtype=int)

    ######################## Your code here ########################
    # log P(x^(i) | z^(i) = k)
    log_p_x_given_z = np.dot(X * M, np.log(self.params.theta).T) + \
                     np.dot((1 - X) * M, np.log(1 - self.params.theta).T)
```

```
    # log of the joint probability of z and X
    log_p_z_x = log_p_x_given_z + np.log(self.params.pi)


    #################################################################

    # subtract the max of each row to avoid numerical instability
    log_p_z_x_shifted = log_p_z_x - log_p_z_x.max(1).reshape((-1, 1))

    # convert the log probabilities to probabilities and renormalize
    R = np.exp(log_p_z_x_shifted)
    R /= R.sum(1).reshape((-1, 1))
    return R
```

Running `check_e_step()` yielded:

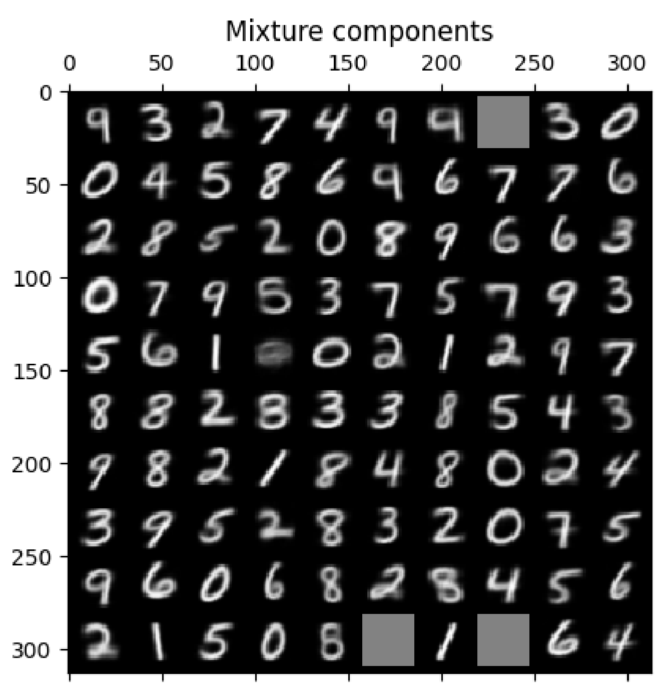
```
The E-step seems OK.
```

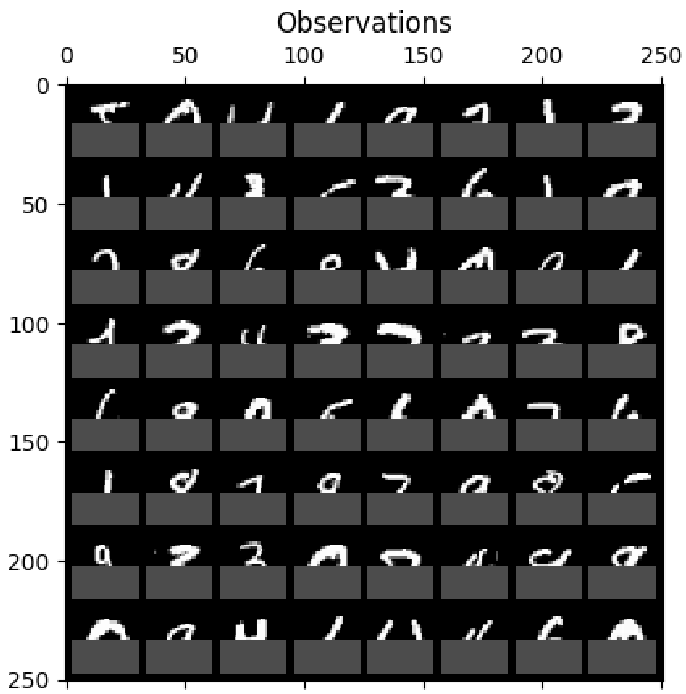
2.3. TASK 4B-III: Report the results (5pt).

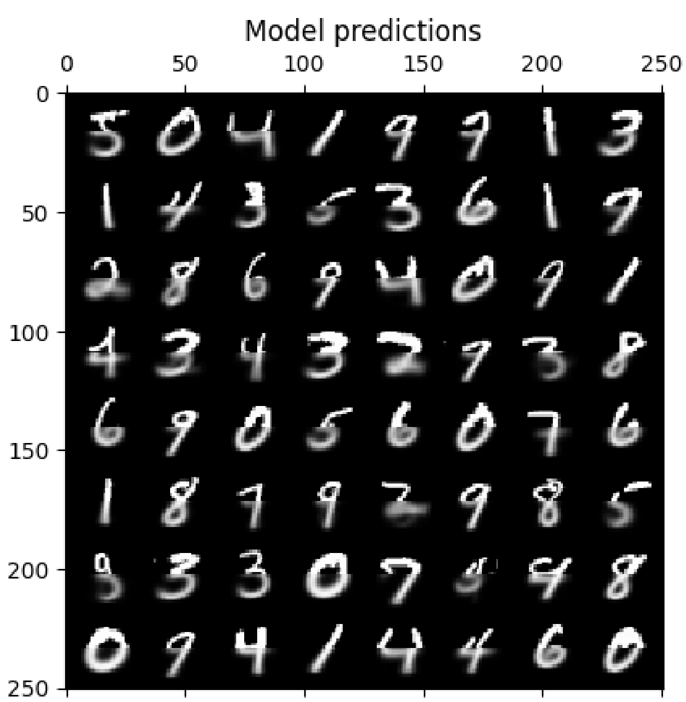
Running the model yielded a final training log-likelihood of -137.79 and a final test log-likelihood of -138.27.

```
model = train_with_em(num_components=100, num_steps=50)


Final training log-likelihood: -137.7888926038781
Final test log-likelihood: -138.26583341994683
```

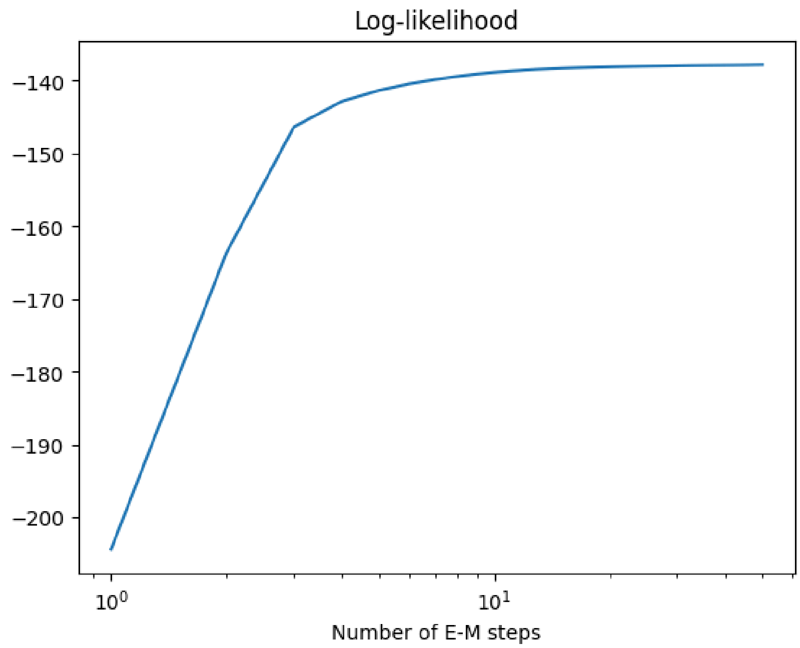
Having this values close to each other is a good sign as it indicates that our model works well with unseen data!







We can see the predictions we've made based on the observed top-halfs of the digits! For the most part they're pretty good predictions :)



This plot shows the convergence of our E-M algorithm. The Increased E-M steps helps improve our log-likelihood, eventually plateauing out.

2.4. TASK 4B-IV: Anomaly detection.

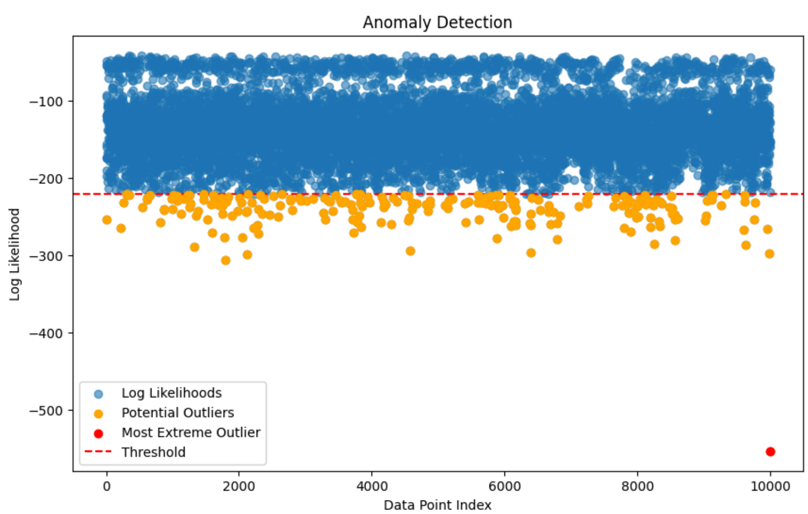
For the bonus task we implemented `detect_and_visualize_outlier(X_test, model)` to find and show the outlier. Note, the outlier's log-likelihood was behaving strangely until we normalized it. Here is how we normalized the outlier:

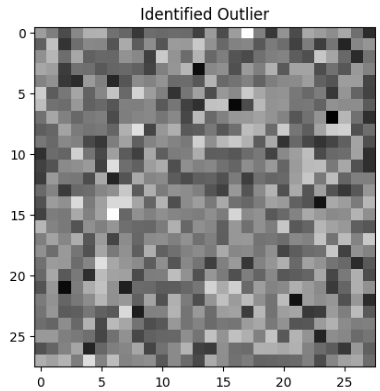
```
X_test = util.read_mnist_images(TEST_IMAGES_FILE)
outlier = np.random.randn(1, 784)

# NORMALIZE THE OUTLIER!!!
outlier_min = outlier.min()
outlier_max = outlier.max()
outlier_normalized = (outlier - outlier_min) / (outlier_max - outlier_min)

# X_test = np.vstack([X_test, outlier])
X_test = np.vstack([X_test, outlier_normalized])
```

Our algorithm calculates the threshold for potential outliers as being more than two standard deviations below the mean log-likelihood. Then we find also find the image with the minimum log-likelihood. As expected, this was the outlier we created, which was just random noise! See the next page:

```
def detect_and_visualize_outlier(X_test, model):
    # compute log-likelihood for each image in the test dataset
    log_likelihoods = np.array([model.log_likelihood(X_test[i:i+1, :]) for
i in range(X_test.shape[0])])

    # find potential outliers as those with log-likelihood significantly
```

```
lower than the mean
    threshold = np.mean(log_likelihoods) - 2 * np.std(log_likelihoods)
    potential_outliers = log_likelihoods < threshold

    plt.figure(figsize=(10, 6))
    plt.scatter(range(len(log_likelihoods)), log_likelihoods, label='Log
Likelihoods', alpha=0.6)
    plt.scatter(np.where(potential_outliers),
log_likelihoods[potential_outliers], color='orange', label='Potential
Outliers', zorder=5)

    # highlight most extreme outlier (minimum log-likelihood)
    outlier_index = np.argmin(log_likelihoods)
    plt.scatter(outlier_index, log_likelihoods[outlier_index], color='red',
label='Most Extreme Outlier', zorder=5)

    plt.axhline(y=threshold, color='r', linestyle='--', label='Threshold')
    plt.xlabel('Data Point Index')
    plt.ylabel('Log Likelihood')
    plt.title('Anomaly Detection')
    plt.legend()
    plt.show()

    outlier_image = X_test[outlier_index, :].reshape(28, 28)
    plt.imshow(outlier_image, cmap='gray')
    plt.title("Identified Most Extreme Outlier")
    plt.show()

    return outlier_index, threshold

outlier_index, threshold = detect_and_visualize_outlier(X_test, model)
```




Identified Outlier
0
5
10
15
20
25
0
5
10
15
20
25