Report:

In this project, the goal was to implement and evaluate two machine learning algorithms—Naive Bayes and Perceptron—for the task of digit classification. This included analyzing their performance across different dataset sizes, understanding their strengths and weaknesses, and addressing challenges encountered during debugging and refinement. Needless to say, this entire process was a rollercoaster of trial, error, perseverance, and patience.

Instructions to run the project:

The naiveBayes.py and perceptron.py files both have implementations for digitdata and facedata. You can just press the run python file button to see a table of results.

Initial Implementation Challenges

As recommended by the writeup, I started out by looking at the Berkeley code online. However, I ran into multiple issues with the Berkeley code as it was written in Python 2, while I had Python 3 installed. Fixing the import errors and compatibility issues took a significant amount of time, so I decided to pivot and start from scratch.

To begin with, the Naive Bayes classifier is based on the assumption that features are independent given the label, which simplifies calculations for high-dimensional datasets. The initial implementation utilized binary features, where each feature was either 0 or 1, representing the presence or absence of specific characteristics in the data. However, several challenges arose. For instance, low variances led to numerical instability, and the assumption of feature independence was unrealistic, given the strong correlations between pixels in image data. Moreover, using raw pixel intensity values (0, 1, 2) caused the Gaussian likelihood calculation to behave poorly. Debugging involved inserting print statements to monitor variances, experimenting with thresholds, and recalculating log-likelihoods. Despite these efforts, the initial learning curve plateaued at an accuracy of around 60%.

In contrast, I initially had an easier time with the Perceptron algorithm. The initial implementation leveraged raw pixel intensity values as features. Unlike Naive Bayes, the Perceptron quickly achieved an accuracy of 80–82%, even on smaller training sets. However, challenges arose in ensuring that weights were initialized properly and determining the optimal number of iterations required experimentation to balance convergence and overfitting. The initial learning curve for the Perceptron exhibited consistent performance, achieving high accuracy early in the training process. Below is a table depicting my best initial results:

Train Size (%)	**Naive Bayes Accuracy**	**Perceptron Accuracy**
10	0.66	0.80
20	0.72	0.81
30	0.76	0.82
40	0.75	0.80
50	0.75	0.79
60	0.74	0.80
70	0.75	0.81
80	0.76	0.81
90	0.75	0.82
100	0.76	0.81

Transition to Improved Implementations

Despite these initial successes, performance on the validation dataset fell short, with accuracies stagnating in the 50s and 60s range. I used my resources and found a link in the class GroupMe to a more compatible version of the Berkeley code written in Python 3. (https://people.cs.rutgers.edu/~lirong.xia/Teaching/2024SAI/projects/classification/classification.html) It turns out that initially, I had been looking at an outdated version of the code/website. With this, I started over with a much better template.

Naive Bayes: Enhancements and Smoothing

One critical improvement to the Naive Bayes classifier was the introduction of Laplace smoothing, as suggested by the website. This technique mitigated the issue of zero probabilities in the conditional likelihoods, ensuring stability in log-likelihood calculations. By applying a smoothing parameter of +3, the model avoided overfitting while maintaining robust estimates for unseen features. Additionally, feature engineering—such as symmetry detection, stroke density calculations, and gradient magnitudes—enhanced the classifier's ability to discern meaningful patterns in the data.

Additionally, recognizing that certain pixels contribute more to classification, particularly those in central regions, a Gaussian weighting scheme was introduced to prioritize central pixels. Additionally, symmetry features, gradient magnitudes, and stroke density measurements were added to capture structural characteristics of digits. For instance, symmetry helped identify digits like '0', '8', and '3' that were structured somewhat similar, while gradient magnitudes detected edges, and stroke density highlighted areas of concentrated pixel activity.

Finally, hole detection using depth-first search (DFS) was implemented to identify loops in digits such as '8' or '0'. Aspect ratios of bounding boxes and pixel intensity normalization also helped account for variations among digits. Despite these enhancements, I found that no matter how much I tried, the digit classification accuracy with Naive Bayes plateaued at around 71%. This was particularly frustrating, as I experimented with various combinations of features, smoothing parameters, and preprocessing techniques. While the classifier excelled in face classification, consistently achieving over 88% accuracy, it struggled to push beyond the 70% threshold for digits, but I did reach the accuracy required by the writeup.

Perceptron: Tuning for an already better performing algorithm

From every result table that I had generated with my initial implementation, I observed that the Perceptron classifier always performed significantly better. Consequently, I had an easier time with Perceptron than Naive Bayes. The implementation iteratively adjusted weights using raw pixel intensity values and enhanced features such as symmetry detection and edge gradients. These features helped capture structural details and edge information crucial for distinguishing digits and facial patterns.

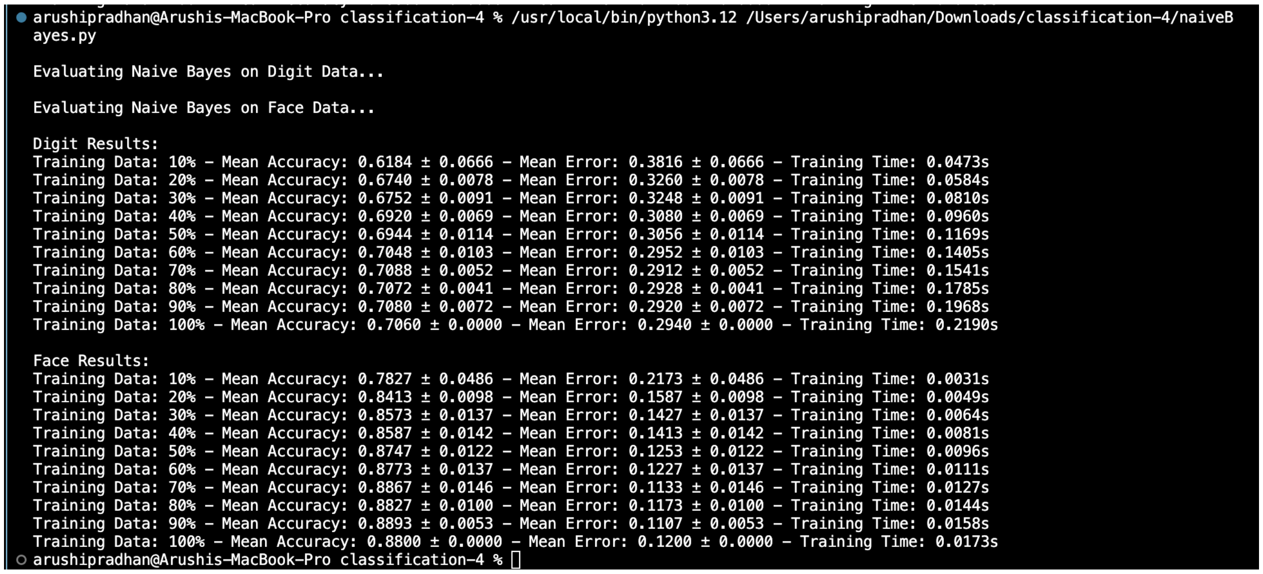
Furthermore, training involved multiple iterations over the training data, and early stopping was incorporated to terminate training when no weight updates were required, improving efficiency. Symmetry, gradient magnitudes, stroke density, and aspect ratio features were incorporated, leveraging the success seen in Naive Bayes. These features allowed the model to better capture the structural nuances of digits. While the classifier consistently achieved strong accuracy for face classification (~89%) and performed well on digits (~73%), training times were longer for larger datasets due to the iterative nature of weight updates.

Conclusion:

In conclusion, both Naive Bayes and Perceptron classifiers demonstrated strengths and challenges when applied to digit and face classification tasks. The Naive Bayes classifier's probabilistic framework and reliance on conditional independence made it efficient and interpretable, particularly for face data. However, it faced limitations with digit classification, plateauing at around 71% accuracy despite extensive tuning. The Perceptron classifier, while computationally more demanding as depicted by its training time, showcased superior performance with enhanced features such as symmetry and edge gradients. Ultimately, both classifiers met the requirements of the write-up, providing valuable insights into the trade-offs between simplicity, accuracy, and computational efficiency in AI.

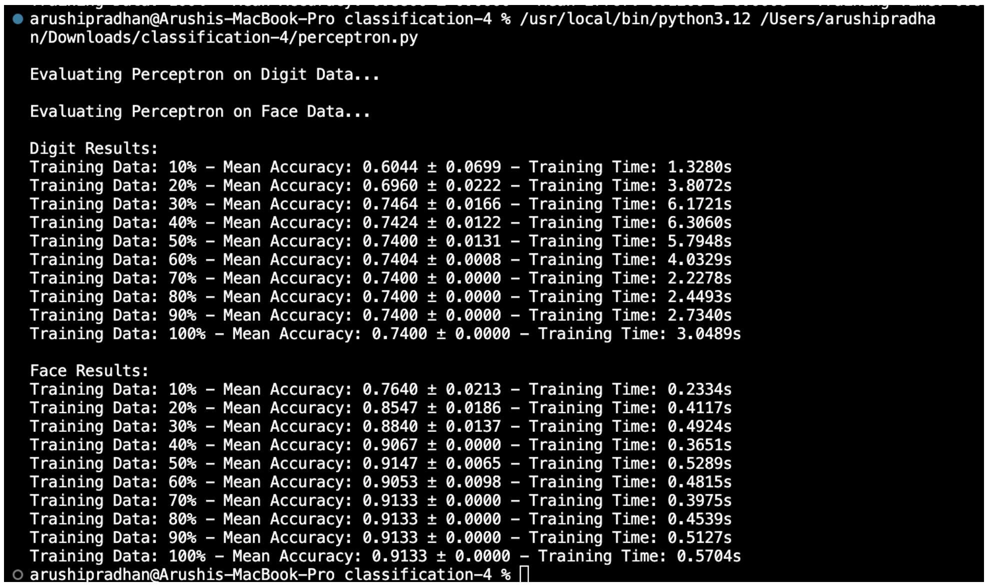
Here are my results:

Naive Bayes:



```
arushipradhan@Arushis-MacBook-Pro classification-4 % /usr/local/bin/python3.12 /Users/arushipradhan/Downloads/classification-4/naiveB
ayes.py

Evaluating Naive Bayes on Digit Data...

Evaluating Naive Bayes on Face Data...

Digit Results:
Training Data: 10% - Mean Accuracy: 0.6184 ± 0.0666 - Mean Error: 0.3816 ± 0.0666 - Training Time: 0.0473s
Training Data: 20% - Mean Accuracy: 0.6740 ± 0.0078 - Mean Error: 0.3260 ± 0.0078 - Training Time: 0.0584s
Training Data: 30% - Mean Accuracy: 0.6752 ± 0.0091 - Mean Error: 0.3248 ± 0.0091 - Training Time: 0.0810s
Training Data: 40% - Mean Accuracy: 0.6920 ± 0.0069 - Mean Error: 0.3080 ± 0.0069 - Training Time: 0.0960s
Training Data: 50% - Mean Accuracy: 0.6944 ± 0.0114 - Mean Error: 0.3056 ± 0.0114 - Training Time: 0.1169s
Training Data: 60% - Mean Accuracy: 0.7048 ± 0.0103 - Mean Error: 0.2952 ± 0.0103 - Training Time: 0.1405s
Training Data: 70% - Mean Accuracy: 0.7088 ± 0.0052 - Mean Error: 0.2912 ± 0.0052 - Training Time: 0.1541s
Training Data: 80% - Mean Accuracy: 0.7072 ± 0.0041 - Mean Error: 0.2928 ± 0.0041 - Training Time: 0.1785s
Training Data: 90% - Mean Accuracy: 0.7080 ± 0.0072 - Mean Error: 0.2920 ± 0.0072 - Training Time: 0.1968s
Training Data: 100% - Mean Accuracy: 0.7060 ± 0.0000 - Mean Error: 0.2940 ± 0.0000 - Training Time: 0.2190s

Face Results:
Training Data: 10% - Mean Accuracy: 0.7827 ± 0.0486 - Mean Error: 0.2173 ± 0.0486 - Training Time: 0.0031s
Training Data: 20% - Mean Accuracy: 0.8413 ± 0.0098 - Mean Error: 0.1587 ± 0.0098 - Training Time: 0.0049s
Training Data: 30% - Mean Accuracy: 0.8573 ± 0.0137 - Mean Error: 0.1427 ± 0.0137 - Training Time: 0.0064s
Training Data: 40% - Mean Accuracy: 0.8587 ± 0.0142 - Mean Error: 0.1413 ± 0.0142 - Training Time: 0.0081s
Training Data: 50% - Mean Accuracy: 0.8747 ± 0.0122 - Mean Error: 0.1253 ± 0.0122 - Training Time: 0.0096s
Training Data: 60% - Mean Accuracy: 0.8773 ± 0.0137 - Mean Error: 0.1227 ± 0.0137 - Training Time: 0.0111s
Training Data: 70% - Mean Accuracy: 0.8867 ± 0.0146 - Mean Error: 0.1133 ± 0.0146 - Training Time: 0.0127s
Training Data: 80% - Mean Accuracy: 0.8827 ± 0.0100 - Mean Error: 0.1173 ± 0.0100 - Training Time: 0.0144s
Training Data: 90% - Mean Accuracy: 0.8893 ± 0.0053 - Mean Error: 0.1107 ± 0.0053 - Training Time: 0.0158s
Training Data: 100% - Mean Accuracy: 0.8800 ± 0.0000 - Mean Error: 0.1200 ± 0.0000 - Training Time: 0.0173s
arushipradhan@Arushis-MacBook-Pro classification-4 %
```

Perceptron:



```
arushipradhan@Arushis-MacBook-Pro classification-4 % /usr/local/bin/python3.12 /Users/arushipradha
n/Downloads/classification-4/perceptron.py

Evaluating Perceptron on Digit Data...

Evaluating Perceptron on Face Data...

Digit Results:
Training Data: 10% - Mean Accuracy: 0.6044 ± 0.0699 - Training Time: 1.3280s
Training Data: 20% - Mean Accuracy: 0.6960 ± 0.0222 - Training Time: 3.8072s
Training Data: 30% - Mean Accuracy: 0.7464 ± 0.0166 - Training Time: 6.1721s
Training Data: 40% - Mean Accuracy: 0.7424 ± 0.0122 - Training Time: 6.3060s
Training Data: 50% - Mean Accuracy: 0.7400 ± 0.0131 - Training Time: 5.7948s
Training Data: 60% - Mean Accuracy: 0.7404 ± 0.0008 - Training Time: 4.0329s
Training Data: 70% - Mean Accuracy: 0.7400 ± 0.0000 - Training Time: 2.2278s
Training Data: 80% - Mean Accuracy: 0.7400 ± 0.0000 - Training Time: 2.4493s
Training Data: 90% - Mean Accuracy: 0.7400 ± 0.0000 - Training Time: 2.7340s
Training Data: 100% - Mean Accuracy: 0.7400 ± 0.0000 - Training Time: 3.0489s

Face Results:
Training Data: 10% - Mean Accuracy: 0.7640 ± 0.0213 - Training Time: 0.2334s
Training Data: 20% - Mean Accuracy: 0.8547 ± 0.0186 - Training Time: 0.4117s
Training Data: 30% - Mean Accuracy: 0.8840 ± 0.0137 - Training Time: 0.4924s
Training Data: 40% - Mean Accuracy: 0.9067 ± 0.0000 - Training Time: 0.3651s
Training Data: 50% - Mean Accuracy: 0.9147 ± 0.0065 - Training Time: 0.5289s
Training Data: 60% - Mean Accuracy: 0.9053 ± 0.0098 - Training Time: 0.4815s
Training Data: 70% - Mean Accuracy: 0.9133 ± 0.0000 - Training Time: 0.3975s
Training Data: 80% - Mean Accuracy: 0.9133 ± 0.0000 - Training Time: 0.4539s
Training Data: 90% - Mean Accuracy: 0.9133 ± 0.0000 - Training Time: 0.5127s
Training Data: 100% - Mean Accuracy: 0.9133 ± 0.0000 - Training Time: 0.5704s
arushipradhan@Arushis-MacBook-Pro classification-4 %
```