Report for Classification

Kongling Zhang, Li Zhang, Zihao Huang

Feature

We use a basic feature that we get from Berkeley code, we get a great result on that so we decide not building our own feature. The feature is pretty straight forward that is introduced by the professor too. One binary feature per pixel in each image, if the pixel is black it becomes 1, otherwise, it is 0, it is simple but it works.

Shuffle

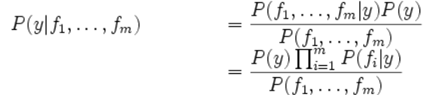
In order to get the correct result, we control the input of the raw training data. We create an int list(0-5000)or(0-450) and shuffle the element, then selects the percentage of the first part of the list, to get the desired input of the training data. We write two methods to achieve this goal. Because the raw training data uses Stack to store all the images and labels, we reverse the list and pop HEIGHT of image times to the element such that we combine the exact image we want.

Naive Bayes

|  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| **Naive Bayes** |  |  |  |  |  |  |  |  |  |  |
| digit | 10 | 20 | 30 | 40 | 50 | 60 | 70 | 80 | 90 | 100 |
| 1 | 79% | 78% | 79% | 82% | 79% | 81% | 81% | 80% | 80% | 79% |
| 2 | 81% | 79% | 78% | 82% | 77% | 80% | 78% | 78% | 79% | 79% |
| 3 | 76% | 78% | 79% | 83% | 78% | 78% | 81% | 81% | 79% | 79% |
| 4 | 76% | 75% | 79% | 80% | 77% | 78% | 78% | 80% | 79% | 79% |
| 5 | 80% | 77% | 82% | 78% | 80% | 78% | 78% | 79% | 78% | 79% |
| **mean** | 78% | 77% | 79% | 81% | 78% | 79% | 79% | 80% | 79% | 79% |
| **prediction error** | 0.02302172887 | 0.01516575089 | 0.01516575089 | 0.02 | 0.01303840481 | 0.01414213562 | 0.01643167673 | 0.01140175425 | 0.007071067812 | 0 |
| **time** |  |  |  |  |  |  |  |  |  |  |
|  | 4.788565 | 6.436478 | 7.914455 | 9.519329 | 10.645044 | 12.209288 | 13.305055 | 14.735117 | 16.55576 | 18.018166 |
|  | 4.831689 | 6.296281 | 8.007225 | 9.288276 | 10.632205 | 12.545375 | 13.350633 | 14.767381 | 16.287926 | 17.299333 |
|  | 4.799725 | 6.294343 | 7.833214 | 9.250372 | 10.714151 | 12.080998 | 13.088387 | 14.690863 | 15.976715 | 17.392675 |
|  | 4.788887 | 6.369601 | 7.867794 | 9.144591 | 10.626377 | 12.196081 | 13.342918 | 14.838288 | 16.449312 | 17.396675 |
|  | 4.833903 | 6.341132 | 7.839669 | 9.420652 | 10.959814 | 12.113873 | 13.337065 | 14.841566 | 16.234698 | 17.369275 |
| **avg time** | 4.8085538 | 6.347567 | 7.8924714 | 9.324644 | 10.7155182 | 12.229123 | 13.2848116 | 14.774643 | 16.3008822 | 17.4952248 |

|  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| face | 10 | 20 | 30 | 40 | 50 | 60 | 70 | 80 | 90 | 100 |
| 1 | 77% | 73% | 86% | 80% | 87% | 87% | 89% | 88% | 87% | 88% |
| 2 | 68% | 84% | 84% | 86% | 87% | 90% | 85% | 90% | 88% | 88% |
| 3 | 61% | 80% | 83% | 83% | 88% | 87% | 88% | 88% | 91% | 88% |
| 4 | 73% | 77% | 86% | 85% | 86% | 89% | 88% | 86% | 89% | 88% |
| 5 | 57% | 85% | 86% | 84% | 85% | 88% | 85% | 88% | 88% | 88% |
| **mean** | 67% | 80% | 85% | 84% | 87% | 88% | 87% | 88% | 89% | 88% |
| **prediction error** | 0.08258329129 | 0.04969909456 | 0.01414213562 | 0.02302172887 | 0.01140175425 | 0.01303840481 | 0.01870828693 | 0.01414213562 | 0.01516575089 | 0 |
|  |  |  |  |  |  |  |  |  |  |  |
| time | 5.721518 | 6.128481 | 6.801041 | 7.390044 | 8.139946 | 8.967353 | 9.818223 | 10.652903 | 11.09436 | 11.708318 |
|  | 5.517006 | 6.20218 | 6.862251 | 7.721896 | 8.580283 | 9.011106 | 9.716571 | 10.635415 | 11.314124 | 12.030398 |
|  | 5.617626 | 6.107013 | 6.804207 | 7.465301 | 8.419021 | 9.217901 | 9.861079 | 10.496543 | 10.945595 | 11.686664 |
|  | 5.669901 | 6.304082 | 6.84585 | 7.474989 | 8.528886 | 8.915747 | 9.781228 | 10.5175 | 11.361329 | 11.965086 |
|  | 5.629097 | 6.099743 | 7.073979 | 7.453384 | 8.083352 | 8.824168 | 9.870398 | 10.507614 | 11.240693 | 11.799134 |
| Avg time | 5.6310296 | 6.1682998 | 6.8774656 | 7.5011228 | 8.3502976 | 8.987255 | 9.8094998 | 10.561995 | 11.1912202 | 11.83792 |

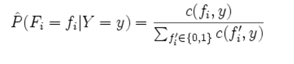
In naive Bayes, we compare the value of P(Y|F1,F2...Fn) to classify data. i.e. Given features F1,F2,...Fn, we choose the label Y with the largest probability. The formula is



And because multiplying these probabilities together may result in underflow, we compute the follow equations instead.



Here, the prior probability P(y) is the number of training instance divided by the total number of training instances, and the posterior probability P(fi|y) is computed by:



We also use k=0.05 to smooth the posterior probability.

The average for Naive Bayes algorithm is around 80 and stable while we testing on the digit, and the error becomes smaller and smaller while we input more and more data. It is a great result to show this algorithm is reliable. While we testing on faces, the accuracy increases smoothly when we training more data until 100%, it shows that this algorithm becomes more reliable when we have more data. The running time for the Naive Bayes algorithm smaller than the other two algorithms, and it becomes higher when we training more data as usual.

Perceptron

|  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| **Perceptron** |  |  |  |  |  |  |  |  |  |  |
| digit | 10 | 20 | 30 | 40 | 50 | 60 | 70 | 80 | 90 | 100 |
| 1 | 80% | 76% | 83% | 80% | 75% | 86% | 83% | 85% | 82% | 81% |
| 2 | 79% | 77% | 84% | 83% | 73% | 81% | 78% | 83% | 80% | 81% |
| 3 | 69% | 76% | 80% | 82% | 78% | 81% | 85% | 84% | 82% | 81% |
| 4 | 72% | 67% | 71% | 77% | 77% | 80% | 86% | 81% | 83% | 81% |
| 5 | 73% | 82% | 81% | 79% | 83% | 82% | 78% | 84% | 83% | 81% |
| **mean** | 75% | 76% | 80% | 80% | 77% | 82% | 82% | 83% | 82% | 81% |
| **prediction error** | 0.04722287581 | 0.05412947441 | 0.05167204273 | 0.02387467277 | 0.03768288736 | 0.0234520788 | 0.03807886553 | 0.01516575089 | 0.01224744871 | 0 |
| time |  |  |  |  |  |  |  |  |  |  |
|  | 22.32031 | 42.371732 | 62.839176 | 84.329747 | 102.757415 | 122.082859 | 141.194051 | 160.972595 | 176.895017 | 198.297749 |
|  | 22.37172 | 42.304936 | 61.179379 | 80.284839 | 101.831453 | 122.645547 | 142.907181 | 161.461047 | 178.718809 | 197.401426 |
|  | 23.39982 | 42.869667 | 62.793989 | 80.943647 | 102.006808 | 121.069211 | 140.82105 | 161.366505 | 177.422246 | 198.949737 |
|  | 22.441127 | 42.232555 | 60.900052 | 81.456945 | 103.899355 | 122.789718 | 141.62758 | 161.560054 | 177.766525 | 197.142687 |
|  | 22.803307 | 41.768724 | 61.075674 | 83.250552 | 102.79847 | 122.326896 | 140.520182 | 163.411991 | 177.559422 | 197.438896 |
| avg time | 22.6672568 | 42.3095228 | 61.757654 | 82.053146 | 102.6587002 | 122.1828462 | 141.4140088 | 161.7544384 | 177.6724038 | 197.846099 |

|  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| face | 10 | 20 | 30 | 40 | 50 | 60 | 70 | 80 | 90 | 100 |
| 1 | 70% | 74% | 85% | 85% | 88% | 79% | 84% | 77% | 88% | 85% |
| 2 | 70% | 72% | 82% | 75% | 91% | 75% | 88% | 89% | 82% | 85% |
| 3 | 78% | 83% | 85% | 80% | 81% | 79% | 87% | 88% | 82% | 85% |
| 4 | 77% | 83% | 82% | 75% | 85% | 79% | 77% | 86% | 79% | 85% |
| 5 | 74% | 78% | 72% | 83% | 80% | 88% | 78% | 78% | 85% | 85% |
| **mean** | 74% | 78% | 81% | 80% | 85% | 80% | 83% | 84% | 83% | 85% |
| **prediction error** | 0.03768288736 | 0.05049752469 | 0.05357238094 | 0.045607017 | 0.04636809248 | 0.04795831523 | 0.05069516742 | 0.05683308895 | 0.03420526275 | 0 |
|  |  |  |  |  |  |  |  |  |  |  |
|  | 6.778475 | 9.012816 | 12.785484 | 14.070378 | 16.854974 | 18.319977 | 21.06039 | 23.779757 | 25.889173 | 31.398791 |
|  | 6.839453 | 9.408371 | 12.741107 | 14.273379 | 15.930971 | 18.723673 | 20.956952 | 23.761009 | 26.060313 | 30.919479 |
|  | 6.969436 | 9.381661 | 12.785484 | 14.411409 | 16.545642 | 18.823696 | 21.240467 | 23.635318 | 25.405026 | 27.902934 |
|  | 6.963568 | 9.370099 | 11.971553 | 14.640157 | 16.59889 | 18.594034 | 21.873103 | 23.608975 | 25.299802 | 28.757357 |
|  | 6.758804 | 9.490223 | 11.53484 | 14.492393 | 16.462569 | 18.595016 | 21.433062 | 23.791011 | 25.963141 | 29.07474 |
| avg time | 6.8619472 | 9.332634 | 12.3636936 | 14.3775432 | 16.4786092 | 18.6112792 | 21.3127948 | 23.715214 | 25.723491 | 29.6106602 |

For perceptron, the key idea of this algorithm is to update the weight. The perceptron algorithm is not calculating the probability but computes the class whose weight vector is most similar to the input feature. We set up two different Y, one is called BigY which get from the label, and the other is get from the training data. If they are same, then we are right, we do nothing, but if they are different, we update the two weights vector. The vector BigY should become higher and the other should become lower. Moreover, we implement a function call findHighWeightFeature, which allow us to show a list of 100 feature with the highest weight for a specific label.

The learning curve for perceptron algorithm is not straight forward, the accuracy does not have obvious change as we training more data, however, while we training more data, the standard deviation becomes smaller which means it becomes more stable. In addition, the running time gets higher while we training more data, as usual, it increases smoothly.

K-Nearest Neighbors

|  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| KNN |  |  |  |  |  |  |  |  |  |  |
| digit | 10 | 20 | 30 | 40 | 50 | 60 | 70 | 80 | 90 | 100 |
| 1 | 85% | 87% | 87% | 91% | 94% | 94% | 92% | 91% | 92% | 92% |
| 2 | 85% | 90% | 85% | 93% | 94% | 92% | 90% | 93% | 94% | 92% |
| 3 | 86% | 93% | 86% | 91% | 90% | 91% | 91% | 92% | 93% | 92% |
| 4 | 87% | 87% | 89% | 89% | 92% | 91% | 92% | 94% | 93% | 92% |
| 5 | 87% | 87% | 90% | 89% | 90% | 93% | 93% | 92% | 92% | 92% |
| **mean** | 86% | 89% | 87% | 91% | 92% | 92% | 92% | 92% | 93% | 92% |
| **prediction error** | 0.01 | 0.02683281573 | 0.02073644135 | 0.01673320053 | 0.02 | 0.01303840481 | 0.01140175425 | 0.01140175425 | 0.008366600265 | 0 |
| time | 139.619623 | 354.91956 | 462.2018 | 557.92628 | 695.27018 | 874.406 | 1018.562 | 1289.1848 | 1469.134 | 1650.27 |
|  | 118.09 | 337.8312 | 455.966 | 667.416 | 714.74 | 867.79 | 1016.11 | 1288.312 | 1454.25 | 1612.35 |
|  | 133.5294 | 354.3053 | 467.0015 | 528.244 | 675.472 | 877.439 | 1043.26 | 1292.45 | 1476.85 | 1699.54 |
|  | 156.6069 | 394.3662 | 462.6426 | 473.0189 | 579.7199 | 882.241 | 1008.17 | 1290.27 | 1477.23 | 1632.45 |
|  | 159.7045 | 346.0058 | 462.2784 | 541.1229 | 732.477 | 863.892 | 1012.9 | 1278.48 | 1455.41 | 1608.34 |
|  | 130.167315 | 342.0893 | 463.1205 | 579.8296 | 773.942 | 880.668 | 1012.37 | 1296.412 | 1481.93 | 1698.67 |

|  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| face | 10 | 20 | 30 | 40 | 50 | 60 | 70 | 80 | 90 | 100 |
| 1 | 65% | 53% | 57% | 75% | 69% | 56% | 56% | 57% | 53% | 53% |
| 2 | 56% | 61% | 59% | 72% | 72% | 53% | 58% | 53% | 54% | 69% |
| 3 | 54% | 54% | 60% | 69% | 70% | 60% | 53% | 59% | 58% | 61% |
| 4 | 50% | 61% | 54% | 76% | 67% | 62% | 60% | 63% | 60% | 57% |
| 5 | 54% | 53% | 56% | 70% | 68% | 59% | 61% | 55% | 55% | 53% |
| mean | 56% | 56% | 57% | 72% | 69% | 58% | 58% | 57% | 56% | 59% |
| **prediction error** | 0.05585696018 | 0.04219004622 | 0.02387467277 | 0.03049590136 | 0.01923538406 | 0.03535533906 | 0.03209361307 | 0.03847076812 | 0.02915475947 | 0.06693280212 |
|  | 63.1650916 | 125.6275492 | 186.661961 | 248.1150382 | 313.9728546 | 377.9182448 | 440.20896 | 496.5696944 | 561.881978 | 615.8146072 |
|  | 63.721356 | 125.007146 | 187.754405 | 246.693191 | 308.372173 | 373.935024 | 439.1394 | 495.759572 | 558.33129 | 615.175636 |
|  | 63.434602 | 122.7749 | 185.4423 | 240.2538 | 317.1439 | 384.9412 | 435.2928 | 502.8428 | 570.7273 | 620.3482 |
|  | 60.4396 | 120.9384 | 186.9321 | 252.5278 | 330.1428 | 370.8428 | 430.4929 | 500.6512 | 550.3892 | 624.2842 |
|  | 65.2487 | 130.4252 | 192.4382 | 258.2482 | 310.8752 | 392.9429 | 445.2959 | 490.6537 | 562.8328 | 610.8283 |
|  | 62.9812 | 128.9921 | 180.7428 | 242.8522 | 303.3302 | 366.9293 | 450.8238 | 492.9412 | 567.1293 | 608.4367 |

In k nearest neighbors, we use the “distance” between the datum in test data and all of the datum in training data to classify the data. Here by “distance” we mean:



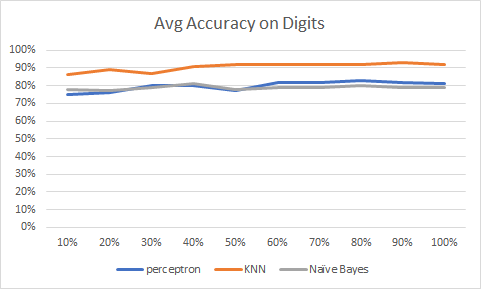
Where pi and qi are the value of each feature in the given two data p,q.

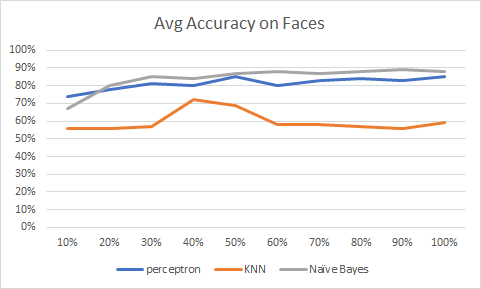
Then we choose the k nearest data in training data, and compute the frequency of the label of each datum in these data. Finally, the most frequent label will be the result of classification.

The KNN algorithm seems to need much more time than Naive Bayes and perceptron, one of the biggest reason that is we use a different computer to test it, the other reason may because it just not as efficient as other two algorithms. However, the difference between each time for training is small from 10% data till 100% of data, it is a great result because it shows that this algorithm is stable. On the other hand, the KNN algorithm seems not good on testing on faces, it only has around 60% accuracy for total, maybe it is not a good algorithm for testing faces.

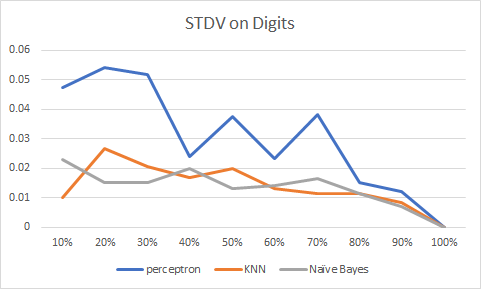
We combine the table and show it in graph.

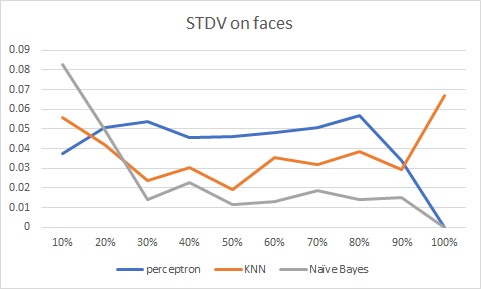
Average Accuracy





Standard derivation





As the graph shows above, we can notice that the standard derivation coverage to zero. Thus, while we using more training data, we get much more correct solution.

Average Time

