# CS520 Class Project - Face and Digit Classification

Shang-Hao Huang (netid: sh1384)

Guoyao Hao (netid: gh343)

April 2020

## 1 Classification Methods

### 1.1 Features Extraction

For each image, the feature we extract is simply all the pixels of that image. For digits recognition task, each image is $28 \times 28 = 784$ pixels in dimension, so each feature is stored as a 2-dimension array of size $28 \times 28$, where each entry indicates whether a specific pixel is black (1) or white (0), the corresponding codes is the function "basicFeatureExtractorDigit" in the file "dataClassifier.py". Similarly, for face detection task, each image is $60 \times 70 = 4200$ pixels in dimension, and thus each feature is represented as a 2d array of size $60 \times 70$ with each entry indicating if a specific pixel is an edge (1) or not (0). Please refer to the function "basicFeatureExtractorFace" in the file "dataClassifier.py" for the implementation.

### 1.2 Perceptron

- Training phase:
  1. First, we randomly initialize (using "random()" function from python library "random") a weight vector $\mathbf{w^y}$ for each prediction class $\mathbf{y}$, where $\mathbf{w}$ is of the size of a feature. There are 10 classes for digits recognition task and 2 for face detection task.
  2. For each instance of training data, represented by a feature vector $\mathbf{f}$, for each class we compute a score $score(f, y) = \sum_j \mathbf{f_j} \cdot \mathbf{w_j^y}$, the dot product of $\mathbf{f}$ and $\mathbf{w^y}$.
  3. For each training instance, select the class with highest score to be the guessed label $\mathbf{y'}$.
  4. Compare the guessed label $\mathbf{y'}$ with the true label $\mathbf{y}$ in the training data:
     (a) If it's the same, then update the weight vector for each label: $\mathbf{w^y} \leftarrow \mathbf{w^y} + \mathbf{f}$
     (b) If not, update the weight vector for each label: $\mathbf{w^y} \leftarrow \mathbf{w^y} - \mathbf{f}$
  5. Repeat the above procedure for the user-specified number of times, the default is 3.
- Testing phase:
  1. For each testing instance, represented by a feature vector $\mathbf{f}$. Similarly, for each class we compute a score $score(f, y) = \sum_j \mathbf{f_j} \cdot \mathbf{w_j^y}$, the dot product of $\mathbf{f}$ and $\mathbf{w^y}$.
  2. Select the class with highest score to be the predicted label $\mathbf{y'}$.

Please refer to the file "perceptron.py" for the implementation of Perceptron algorithm.

## 1.3 Naive Bayes Classifier

- Training phase: Let a training instance be represented by a feature vector $\mathbf{f} = (f_1, f_2, ..., f_n)$.
  1. Estimate the prior distribution over each label $\mathbf{y}$ from the whole training set:

$$\hat{P}(y) = \frac{c(y)}{N}, \tag{1}$$

     where $c(y)$ is the number of training instances with label $\mathbf{y}$ and $N$ is the training set size.
  2. Calculate smoothed estimate of conditional probability of each pixel in $f$ given each label $\mathbf{y}$ from the whole training set using *Laplace smoothing*:

$$\hat{P}(f_j|y) = \frac{c(f_j, y) + k}{\sum_{f'_j \in \{0,1\}} (c(f'_j, y) + k)}, \tag{2}$$

     where $k$ is specified by user and $c(f_j, y)$ is the number of times of $j - th$ pixel of f took value $f_j$ in the training instances of label $\mathbf{y}$.

- Testing phase: For each testing instance, which is represented by a feature vector $\mathbf{f} = (f_1, f_2, ..., f_n)$,
  1. For each class $\mathbf{y}$, we calculate ***log joint posterior probabilities*** $P_y$:

$$\begin{aligned} P_y &= \log P(y|f_1, f_2, ..., f_n) = \log \frac{P(f_1, f_2, ..., f_n|y)P(y)}{P(f_1, f_2, ..., f_n)} \\ &= \log \frac{\prod_{j=1}^{n} P(f_j|y)P(y)}{P(f_1, f_2, ..., f_n)}, \end{aligned}$$

     as $f_1, f_2, ..., f_n$ are assumed mutually independent, conditional on the class $\mathbf{y}$
  2. Then, select the label $\mathbf{y}'$ with the highest $P_y$ as the predicted label for this instance. That is,

$$\begin{aligned} \arg\max_y P_y &= \arg\max_y \log \frac{\prod_{j=1}^{n} P(f_j|y)P(y)}{P(f_1, f_2, ..., f_n)} \\ &= \arg\max_y \left\{ \log P(y) + \sum_{j=1}^{n} \log P(f_j|y) - \log P(f_1, f_2, ..., f_n) \right\} \end{aligned}$$

Please refer to the file "naiveBayes.py" for the implementation of Naive Bayes algorithm.

## 1.4 K-Nearest Neighbors

- Training phase: as there′s not really a training phase for K-Nearest Neighbors algorithms, what we do in "train" function of the class "kNNClassifier" is simply storing the feature of each image in the training set as 2d "numpy" array of size $28 \times 28$ for a digit image, or $60 \times 70$ for a face image.
- Testing phase:

1. For each image in the testing set, we compute its Euclidean distance between each image in the training set.
2. We sort the these distances in an increasing order and select "K" images from the training set with the shortest distance possible.
3. Last, we count the times the labels occur in this "K" images set, then we predict the label of this testing image with the label that occurs the most frequently in this "K" images set.

The full implementation for K-Nearest Neighbors is in the file "kNN.py"

### 1.5 Implementation

Our programs is built upon the codes and data hosted on the Berkeley′s CS188 course page [1], which can accessed via the same website. The program is developed under a python2 environment, please install python2 and switch to python2 prior to the execution. All the core functions of Perceptron, Naive Bayes, and K-Nearest Neighbors were implemented by ourselves. The four main programs are "perceptron.py", "naiveBayes.py", "kNN", and "dataClassifier.py".

## 2 Experiments & Results

### 2.1 Parameters

The sets of hyper-parameter we′ve tested for each of these three methods are listed in the following table.

| Methods | | |
|---|---|---|
| Perceptron | Naive Bayes | K-Nearest Neighbors |
| Number of Iterations = [3, 5] | k = [0.001, 0.01, 0.05, 0.1, 0.5, 1, 5, 10, 20, 50] (using grid search) | K = [5, 10] |

### 2.2 Command

- For digit recognition task, please run the following command via Terminal: python dataClassfier -c [model] -t [training set size]
- For face detection task, please run the following command via Terminal: python dataClassfier -c [model] -t [training set size] -d faces
- Options:
  1. The model option includes: perceptron, naiveBayes, kNN.
  2. The default training set size is 100, the maximum is 5000 for digits data set and 451 for faces data set.

3. Options for Perceptron: -i [the number of training iterations to run, default is 3]
4. Options for Naive Bayes: -k [smoothing parameter, default is 2.0, ignored when using –autotune] –autotune [whether to automatically tune hyper-parameters]
5. Options for K-Nearest Neighbors: -n [the number of nearest neighbors, default is 5]

## 2.3 Experiment

We run each method and the set of hyper-parameters for 10 iterations. We first train on 10% of the whole training set, and increase the number of data points used for training by 10% of the whole training set each time. For each iteration, we calculate the training time needed and the prediction accuracy. Then, we repeat this procedure 5 times, and compute the average training time and prediction accuracy of these 5 executions as well as the standard deviation to evaluate the consistency of prediction accuracy.

### 2.3.1 Random Module

We simply use the "sample" function the library "random" in python to randomly select specified amount of data points in the training set for the training.

## 2.4 Training Time Comparison

To compare how much time needed for training for each method tested, we plot the training time for each method and set of hyper-parameters against the number of data points (in percentage) used for training for digit recognition and face detection tasks, as shown in Fig.1 and Fig.2. Notice that, as K-Nearest Neighbors algorithm does not really have a training phase, the training time is zero. The training time needed by Perceptron increases linearly as we use more training data because the model will update its parameter (weight vector) during the training stage for each training item. The training speed of Naive Bayes is much faster. Because unlikely to the Perception, it doesn't need prediction and calculate the loss.

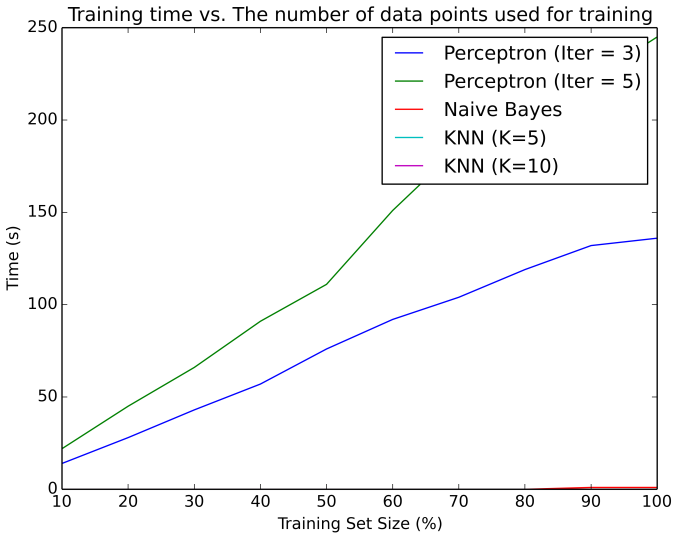


Figure 1: Train time (s) comparison for digit recognition task

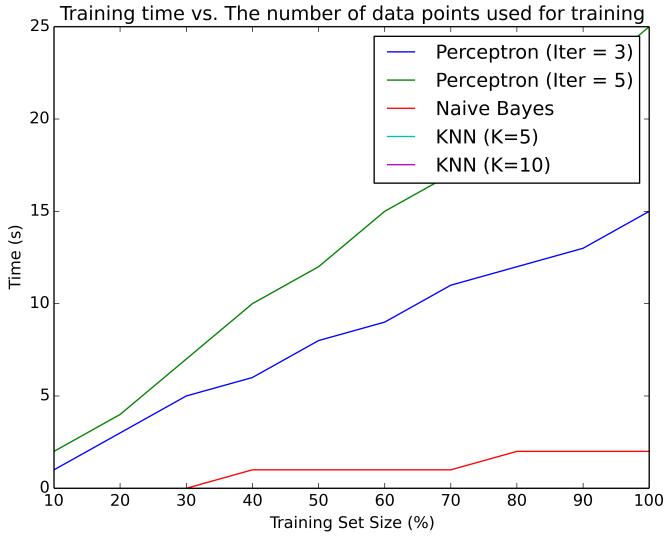


Figure 2: Train time (s) comparison for face detection task

## 2.5 Prediction Accuracy

### 2.5.1 Mean

In order to evaluate how well different methods perform, we plot the mean accuracy for each method and set of hyper-parameters against the number of data points (in percentage) used for training for digit recognition and face detection tasks, and the results are shown in Fig.3 and Fig.4. In general, KNN with $k = 5$ outperforms KNN with $k = 10$. The Perceptron with $iteration = 5$ is slightly better than Perceptron with

*iteration* = 3. For digit recognition task, shown in Fig.3, we can see that the accuracy of KNN gradually increases when more training data are used while the accuracy of Perceptron and Native Bayes are relatively stable. For face detection task, Fig.4, the accuracy of Perceptron and Native Bayes continuously increase, while the accuracy of KNN is relatively stable. By comparing Fig.3 and Fig.4, we can observe that KNN outperforms Naive Bayes and Perceptron in digit recognition task. On the other hand, in face detection task, the performance of KNN is better. This is because the characteristic of our two data sets are different. For the digits, because the shapes of the digits are solid, two similar number have many digits in common. So their distance is close to each other, and thus KNN have better performance.
However, in the face data set, we only have the outline of faces instead of a solid shape. This means two face can have very large distance because their position are different even though the outline of a face and other object could be similar. For some specific point, the probability of having "1" in that location could be vastly different. For example, faces have less "1" on the four corners. As a result, Perceptron and Naive Bayes could get better result.

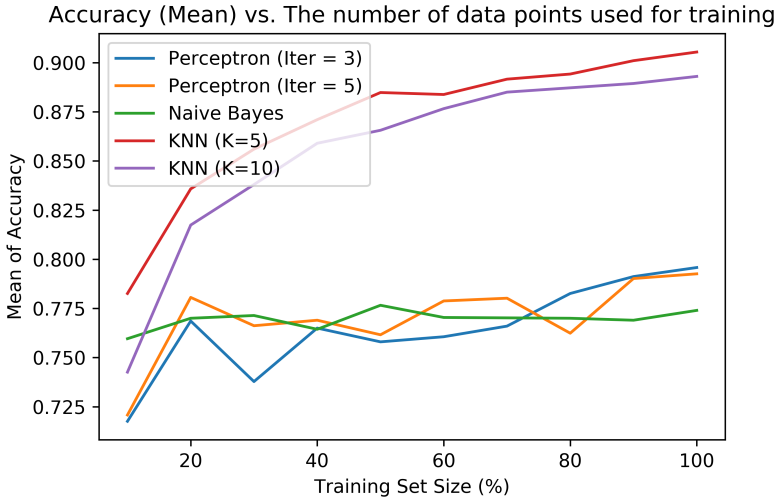


Figure 3: Mean Accuracy comparison for digit recognition task

### 2.5.2 Standard Deviation of Accuracy

The results are shown in Fig.5 and Fig.6. In general, the deviation goes down as we using more training data. We can also find that the standard deviation of accuracy is related to the average of accuracy. Accuracy with higher average will result in lower standard deviation, which means if a model has a better result on a data set, it is more consistent.

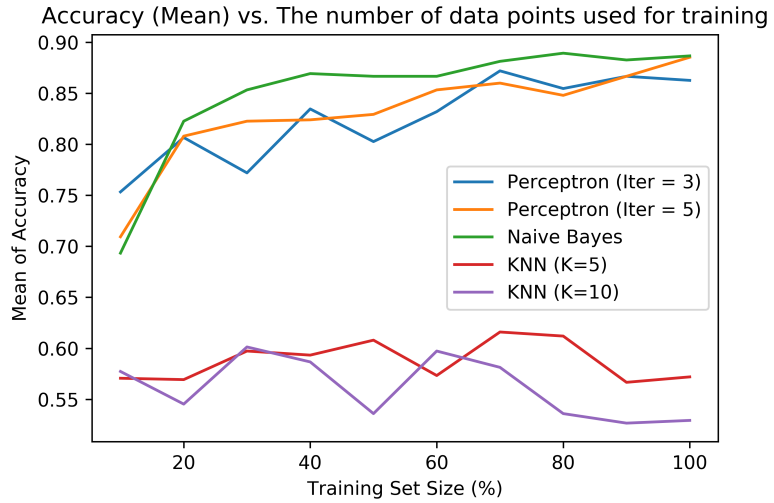


Figure 4: Mean Accuracy comparison for face detection task

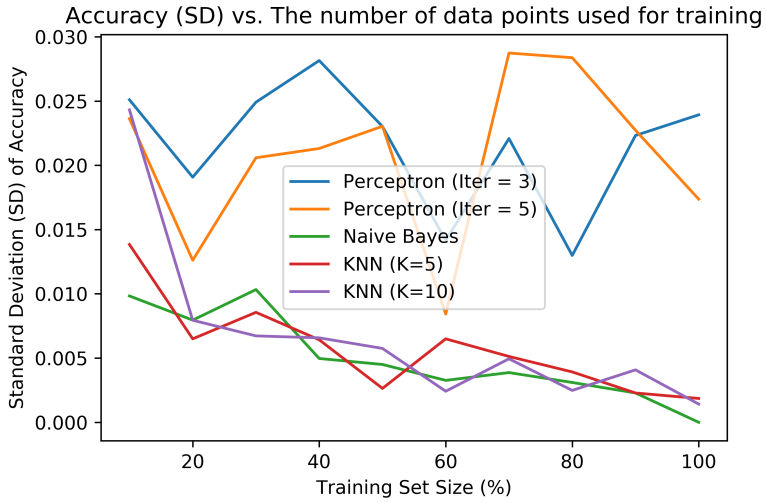


Figure 5: Standard Deviation of Accuracy comparison digit recognition task

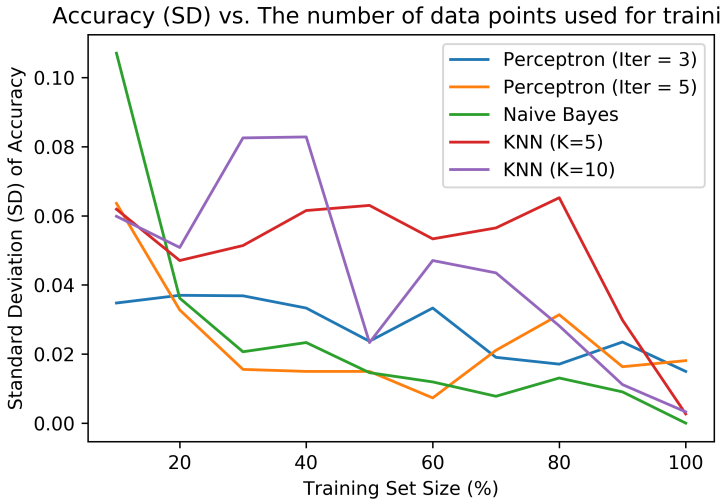


Figure 6: Standard Deviation of Accuracy comparison for face detection task

# References

[1] D. Klein and J. DeNero. Project 5: Classification. [Online]. Available: http://inst.eecs.berkeley.edu/ cs188/sp11/projects/classification/classification.html