Image Classification

Jaehyun Kim, Alex Vobornov

August 2019

1 Perceptron

1.1 Extracting

In classifier.py, the raw data from training data returns a set of pixel features indicating whether each pixel in the provided datum is '0' if it is empty ,or '1' if it is '+' or '#'. This occurs in basicFeatureExtractorDigit and basicFeatureExtractorFace functions. For example, the data will be $\{(0,0) : 0, (0,1) : 0, (0,2) : 1...(28,28) : .0\}$.

1.2 Perceptron Classifier

The Perceptron classifier is called by classifier.py. Arguments of 'legalLabels' and 'option.iterations' are used. 'LegalLabels' is a list that uses 0 to 9 to compare labels for digits, and 0 to 1 for faces. 'option.iterations' is the option with -i that indicates how many to repeat the training. The default iteration is 3. That means hidden layers are 3.
For each layer, we scan one instance at a time and find the label with the highest score. To get a high score, we set y as real label and y' is guess label. The guess label can get from this formula y' = arg max score(feature, y") .
To get y', the guessed label, we use classify function in perceptron.py. We Multiply weight[label] and current data, datum. This means multiplying each features of value and then return the sum of the values to vector which is type of list. For example,

vectors[l] = weights[l] * datum
$= sum+ = weight[label] * datum[label]$
$= sum+ = (a,b) : z * (a',b') : z' = z * z'$
$= ((0,0) : 0 * (0',0') : 0) + ... + ((28,28) : 0 * (28,28) : 0)$
vector[l] = (0*0)+...+(0*0)
vector = {0:0, 1:144, 2:4, ... , 9:0}

After calculating all of the vectors, we choose the maximum number in the vector list and using argMax function, which is in util.py, we return the highest key of the highest value.

And then, We compare y and y'. If y and y' are same then skip because we guessed correctly. If y and y' are not matching. It means that we guessed y' but we should have guessed y. The weight of y should have scored f higher and weight of y' should have scored f lower.
In our code, for example, we used training data to see how y and y' work.

```
('Starting iteration ', 0, '...')
We have guessed 0 but should have guessed 5
We have guessed 5 but should have guessed 0
We have guessed 0 but should have guessed 4
We have guessed 5 but should have guessed 1
We have guessed 0 but should have guessed 9
We have guessed 0 but should have guessed 2
We have guessed 9 but should have guessed 1
We have guessed 1 but should have guessed 3
We have guessed 1 but should have guessed 4
We have guessed 1 but should have guessed 3
We have guessed 3 but should have guessed 5
We have guessed 3 but should have guessed 6
We have guessed 3 but should have guessed 1
We have guessed 3 but should have guessed 7
We have guessed 3 but should have guessed 2
We have guessed 3 but should have guessed 8
We have guessed 3 but should have guessed 6
We have guessed 1 but should have guessed 9
We have guessed 3 but should have guessed 4
We have guessed 3 but should have guessed 0
We have guessed 4 but should have guessed 9
We have guessed 9 but should have guessed 1
We have guessed 3 but should have guessed 2
We have guessed 2 but should have guessed 4
We have guessed 2 but should have guessed 3
We have guessed 3 but should have guessed 2
We have guessed 2 but should have guessed 7
We have guessed 2 but should have guessed 3
We have guessed 3 but should have guessed 8
We have guessed 3 but should have guessed 6
We have guessed 3 but should have guessed 9
We have guessed 3 but should have guessed 0
We have guessed 3 but should have guessed 5
We have guessed 3 but should have guessed 6
We have guessed 2 but should have guessed 0
We have guessed 2 but should have guessed 7
We have guessed 3 but should have guessed 1
We have guessed 0 but should have guessed 8
```

We have guessed 3 but should have guessed 7
We have guessed 0 but should have guessed 9
We have guessed 3 but should have guessed 8
We have guessed 0 but should have guessed 5
We have guessed 3 but should have guessed 9
We have guessed 0 but should have guessed 7
We have guessed 9 but should have guessed 4
We have guessed 9 but should have guessed 8
We have guessed 9 but should have guessed 4
We have guessed 8 but should have guessed 1
We have guessed 8 but should have guessed 4
We have guessed 4 but should have guessed 6
We have guessed 6 but should have guessed 4
We have guessed 4 but should have guessed 5
We have guessed 8 but should have guessed 1
We have guessed 2 but should have guessed 0
We have guessed 4 but should have guessed 0
We have guessed 0 but should have guessed 1
We have guessed 4 but should have guessed 7
We have guessed 4 but should have guessed 6
We have guessed 0 but should have guessed 3
We have guessed 1 but should have guessed 2
We have guessed 4 but should have guessed 7
We have guessed 0 but should have guessed 9
We have guessed 0 but should have guessed 2
We have guessed 2 but should have guessed 7
We have guessed 7 but should have guessed 9
We have guessed 9 but should have guessed 4
We have guessed 9 but should have guessed 8
We have guessed 9 but should have guessed 7
('Starting iteration ', 1, '...')
We have guessed 8 but should have guessed 5
We have guessed 4 but should have guessed 3
We have guessed 4 but should have guessed 5
We have guessed 8 but should have guessed 1
We have guessed 7 but should have guessed 1
We have guessed 7 but should have guessed 4
We have guessed 1 but should have guessed 8
We have guessed 7 but should have guessed 9
We have guessed 8 but should have guessed 0
We have guessed 1 but should have guessed 5
We have guessed 0 but should have guessed 6
We have guessed 8 but should have guessed 5
We have guessed 0 but should have guessed 3
We have guessed 9 but should have guessed 4
We have guessed 4 but should have guessed 9

We have guessed 9 but should have guessed 8
We have guessed 8 but should have guessed 1
We have guessed 9 but should have guessed 7
We have guessed 1 but should have guessed 2
We have guessed 0 but should have guessed 9
We have guessed 9 but should have guessed 6
We have guessed 9 but should have guessed 3
We have guessed 9 but should have guessed 7
('Starting iteration ', 2, '...')
We have guessed 3 but should have guessed 5
We have guessed 9 but should have guessed 2
We have guessed 9 but should have guessed 1
We have guessed 2 but should have guessed 0

In this case, we updated the weights as $w^y = w^y + f$ and $w^{y\prime} = w^{y\prime} - f$. In the code, we uses util.py methods which are __radd__ and __sub__ to increment/decrement counters.

$self.weights[y].\_\_radd\_\_(trainingData[i])$
$self.weights[yPrime].\_\_sub\_\_(trainingData[i])$

1.3 Training data & Analysis

1.3.1 Digit

Training Data Used	Labels out of 5000	Validation Accuracy	Test Accuracy
10%	500	81%	76%
20%	1000	77%	79%
30%	1500	80%	78%
40%	2000	85%	77%
50%	2500	84%	82%
60%	3000	80%	82%
70%	3500	81%	84%
80%	4000	83%	86%
90%	4500	84%	87%
100%	5000	82%	86%

Table 1: Percentage of training data with iteration 3

1.3.2 Face

Training Data Used	Labels out of 450	Validation Accuracy	Test Accuracy
10%	45	74%	62%
20%	90	91%	76%
30%	135	91%	72%
40%	180	94%	81%
50%	225	95%	79%
60%	270	99%	87%
70%	315	95%	80%
80%	360	93%	80%
90%	405	98%	82%
100%	450	100%	85%

Table 2: Percentage of face training data with iteration 3

2 Naive Bayes

2.1 Prior Probability in Naive classifier

Our naive Bayes model has several parameters to estimate. One parameter is the prior probability over labels (digits, or face/not-face), $\Pr[Y]$.
To estimate $\Pr[Y]$, we extract data from training data labels.

$$\Pr[y] = \frac{c(y)}{n}$$

To implemet this formula, we use incrementAll, and normalize function from Counter() class in util.py. First initialize dictionary using Counter class and using incrementAll function, all the label keys to set 0. And then, count all of the label in the training data and save to dictionary, which is total_cnt_label in our code. For example, using 100 of labels, it looks {0: 13, 1: 14, 2: 6, 3: 11, 4: 11, 5: 5, 6: 11, 7: 10, 8: 8, 9: 11}. Next step is that using normalize function, we divide each value by the sum of all values. For example, it looks {0: 0.13, 1: 0.14, 2: 0.06, 3: 0.11, 4: 0.11, 5: 0.05, 6: 0.11, 7: 0.1, 8: 0.08, 9: 0.11}. Final step is that declare self.prior_probabilty variable and put it in this value so that we can use the other function to calculate based on Naive Bayes.

2.2 Conditional Probabilities in Naive classifier

The next estimation is the conditional probabilities of our features given each label y: $\Pr[F_i|Y = y]$.

$$\Pr[F_i = f_i|Y = y] \quad = \quad \frac{c(f_i, y)}{\sum_{f_i' \in \{0,1\}} c(f_i, y)}$$

First, getting a cost of feature and label set, we extract data from training data. While extracting data, to calculate the formula in the denominator ($\sum_{f_i' \in \{0,1\}} c(f_i, y)$), we declare conditional_feature_label with util.Counter() for counting value 1 which means that during searching pixels and if pixel is 1 then increment 1. We also declare non_conditional_feature_label for counting value 0. For example, the format of these two variables look like {(feature, label):value ... } and this is for real example from our code $\{((4, 23), 5) : 1, ((4, 24), 5) : 1, ((5, 23), 5) : 1, ...\}$. Following the formula which denominator, we declare total_features_labels and add conditional_feature_label with non_conditional_feature_label.

Second, to calculate the formula in the numerator ($c(f_i, y)$), we used already declared variable which is conditional_feature_label with util.Counter(). In this case, we already added all of information to the variable.

Third step is that we need to smooth our cost of feature and label sets. It helps to improve more accuracy and reduce overfitting to recognize digit and face. In our project, we fixed smooth value as 0.05 instead default value because 0.05 show us the highest accuracy and reduce running time.

Final step is to get conditional probability. Following the formula, we already have all information to get conditional probability, so we extract value from conditional_feature_label and total_features_labels. And then divide using that two information and save to variable.

To get a posterior probability, we called classify function. This function is also called calculateLogJointProbabilities fucntion to calculate log joint probability. As description of Berkeley project mention, during using Bayes theorem, multiplying many probabilities together often results in underflow, so instead we use log probability. We use characteristic of log that logMN = logM + logN which means log(prior) + log(conditional) which is likelihood.

$$\arg\max_y log(P(y|f_1, \ldots, f_m) \quad = \quad argmax_y(log(P(y)) + \sum_{i=1}^{m} log(P(f_i|y)))$$

More specifically, to calculate the logJoint probability for each label, we loop through all the labels and initialize the probability to log of the prior probability of each label, given to us by the data in TrainAndTune. We then loop through the features and check to see if there is a feature of value in datum in order to get the conditional probability of our features given each label. The conditional probability of the features occurring given the label can be calculated by checking if the value for those features in datum is 1, which indicates there is data present. If the value is 0 there are no features of value, and the event 'does NOT occur' so we need to calculate 1-P(event occurs) to adjust our probability. We then sum our prior probability (already initialized) with the log of the likelihood of the features occurring given the label, giving us the posterior distribution for that label.

For implementing this code, we encountered an issue that 'ValueError: math domain error' was occurred. After debugging the code, we found the solution that log must not include less than or equal to 0, so we check if value is less than 0 then we just set 1 to avoid error because log 1 is 0.

2.3 Training data & Analysis

2.3.1 digit

Training Data Used	Labels out of 5000	Validation Accuracy	Test Accuracy
10%	500	83%	74%
20%	1000	80%	78%
30%	1500	82%	79%
40%	2000	85%	78%
50%	2500	84%	80%
60%	3000	83%	82%
70%	3500	84%	81%
80%	4000	82%	81%
90%	4500	83%	79%
100%	5000	84%	79%

Table 3: Percentage of digit training data

2.3.2 Face

Training Data Used	Labels out of 450	Validation Accuracy	Test Accuracy
10%	45	68%	53%
20%	90	95%	82%
30%	135	98%	85%
40%	180	98%	85%
50%	225	97%	83%
60%	270	97%	85%
70%	315	97%	86%
80%	360	97%	86%
90%	405	95%	88%
100%	450	95%	88%

Table 4: Percentage of face training data

3 Conclusion

In the conclusion, the different thing between Perceptron and Naive Bayes is that Perceptron uses weight instead of probability to learning data. On the other hands, Naive Bayes is based on Bayes rule, so it depends on probability to learning data.
Both Perceptron and Naive Bayes classifier show similar accuracy. However Naive Bayes accuracy is a little bit higher than Perceptron that include face and digit. The result graphs show in 3.1 Comparing data & Analysis.

3.1 Comparing data & Analysis

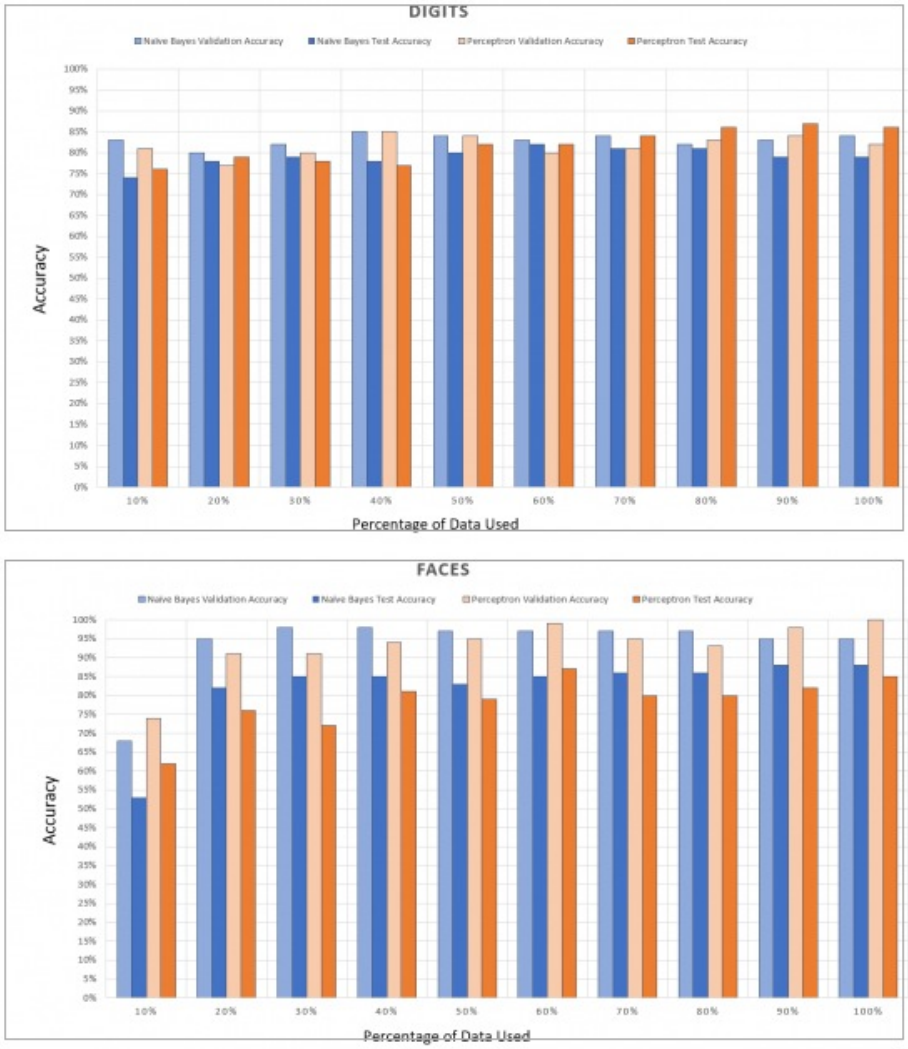

Figure 1: Comparing between Perceptron and Navie Bayes

3.2 Command

-t: number of training data to use, default is 100
-c: perceptron, default is nb
-d: digits or faces, default is digits
-k: smoothing, default is 0.05
-i: iteration, default is 3
Caution: Training Faces data only have 450. Digit data has 5000.

3.2.1 Perceptron

Digit: python3 classifier.py -t 500 -c perceptron
Face: python3 classifier.py -t 100 -d faces -c perceptron

3.2.2 Naive Bayes

Digit: python3 classifier.py -t 500 -c naiveBayes
Face: python3 classifier.py -t 100 -d faces -c naiveBayes