Machine Learning HWS21
Assignment 1: Naives Bayes

Timur Michael Carstensen - 1722194

17.10.2021

Contents

1 Training

To train a Naive Bayes classifier for categorical data with a symmetric Dirichlet prior, a fully Bayesian approach is used. First, the prior class distribution and the class conditional densities (i.e. the likelihood for each class) must be computed.

When using a symmetric Dirichlet prior, the alphas for each class are identical (i.e. $\alpha_c = \alpha$). Hence, the equations given in slide set three of the Machine Learning class for the prior and class conditional densities can be slightly adapted to more accurately reflect the implementation used here:

$$\bar{\pi}_c = \frac{n_c + (\alpha - 1)}{N + C \times (\alpha - 1)} \tag{1}$$

$$\left[\bar{\theta}_{cj}\right]_k = \frac{n_{cjk} + (\alpha - 1)}{n_c + K \times (\alpha - 1)} \tag{2}$$

The exact approach for both Equation 1 and Equation 2 was to first compute the absolute values of the occurrences.

In the case of the prior class distribution, this was done by first counting the number of training samples from each class. Then Equation 1 was applied where n_c corresponded to the number of occurrences of each class in the training set, N to the number of samples (i.e. 60.000) and C to the number of classes (i.e. 10).

For the class conditional densities a similar approach was followed. Here, it is quite helpful to visualize the training set as a cube as shown in Figure 1.

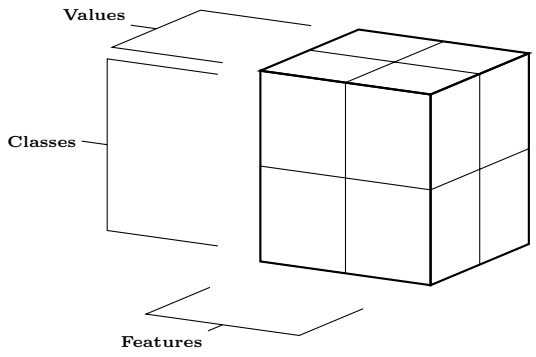

Figure 1: Training set visualised as a cube.

The first dimension representing the individual classes (0 through 9), the second representing the features (0 through 783) and the third representing the number of possible values of each feature (0 through 255). Here, the first

step was to, again, count the number of occurences of each value a feature can take, for each feature and for each class . Then, Equation 2 was applied to this pre-computed "cube" of absolute values. Here, n_{cjk} represents the previously computed quantity, K the number of possible values of each feature and n_c is defined as above.

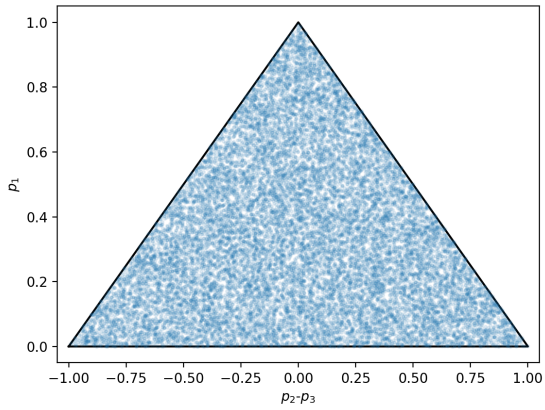

Figure 2: Symmetric Dirichlet distribution with three classes and $\alpha = 1$

When using a (symmetric) Dirichlet prior, setting $\alpha = 1$ is equivalent to having no prior knowledge over the distribution of training samples. In other words, it is assumed that all probability vectors are equally likely as can be seen for the three class case in Figure 2.

Hence, for an $\alpha = 1$, one then obtains the maximum likelihood estimate (MLE). For $\alpha \geq 2$ one obtains add-one smoothing (or rather for an $\alpha = 2$). For increasing values of α, one becomes more certain about the distribution of probabilities. For a symmetric prior, increasing the value of α will increase the certainty that the probabilities are uniformly distributed as shown in Figure 3.

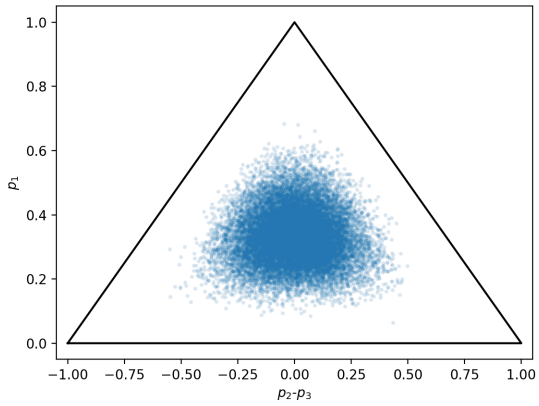

Figure 3: Symmetric Dirichlet distribution with three classes and $\alpha \geq 2$.

Finally, the function returns the natural log of the prior distribution and class conditional densities to be used in section 2.

2 Prediction

To predict the most likely class label for each new example, it is first necessary to compute the (log) joint probabilities of each new example and each class. When following the naive bayes assumption of conditional independce among features given their class, this would mean that one would need to compute the product of probabilities of each individual feature value seen in a new example, given a particular class (cf. Equation 3).

$$p(y = c \mid \mathbf{x}, \mathcal{D}) \propto \bar{\pi}_c \prod_{j=1}^{D} [\bar{\theta}_{cj}]_{x_j} \tag{3}$$

However, as the MNIST dataset contains 784 features for each sample, computing the product of those for each new example would most likely result in numerical underflow as discussed in class previously. Therefore, log probabilities are used in the implementation. The log posterior predictive therefore is as shown in Equation 4:

$$p(y = c \mid \mathbf{x}, \mathcal{D}) \propto \log \bar{\pi}_c + \sum_{j=1}^{D} \log[\bar{\theta}_{cj}]_{x_j} \tag{4}$$

The implemented algorithm used to calculate the joint probability matrix corresponds to the multi-class version of algorithm 3.2 in [Mur12]. That is, the LogSumExp-trick, which was also discussed in Exercise 1 of the Machine Learning class, was implemented:

$$p(y = c \mid \mathbf{x}, \mathcal{D}) = \log \bar{\pi}_c + \sum_{j=1}^{D} \log[\bar{\theta}_{cj}]_{x_j} - logsumexp$$
$$\text{where } logsumexp(\mathbf{x}) = \max_x + \log \sum_{i=1}^{n} \exp(x_i - \max_x) \tag{5}$$

In other words, the elements of the joint probability matrix are first calculated as the conditional probability of each class given a particular example and then shifted using the LogSumExp-trick. The result of the above is then a $n \times C$ matrix where n is the number of new examples and C the number of classes.

Finally, for each row vector (i.e. the joint probabilites of a new example with classes 0 through 9), the most likely class is predicted as the one with the highest probability and returned along with the log probability of that class for that example.

3 Experiments on MNIST digits data

3.1 a)

The accuracy of the model with add-one smoothing ($\alpha = 2$) is 0.8363.

3.2 b)

To further inspect model performance apart from accuracy mentioned in subsection 3.1, the provided code for precision, recall and F1-score is quite helpful. One can see that the model performs quite well at correctly classifying ones with recall = 0.97 while also being reasonably good at preventing false positives with precision = 0.86. The model also performs well for zeros, sixes, nines and sevens. Model performance is the worst for fives with recall = 0.67 and precision = 0.78. Model performance is also quite bad for twos, threes and fours and eights.

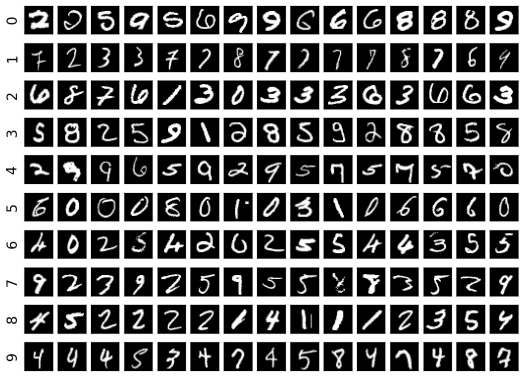

Figure 4: Errors grouped by predicted label.

An interesting observation that one can make is that the model tends to misclassify digits that have an overall (or just partly) similar shape to others. This becomes quite apparent in Figure 4. Here, the wrong predictions are arranged by the predicted class. It is visible that the model often confuses twos, fives, sixes, eights and nines for zeros. That is, digits with round lines are often misclassified for another with similar characteristics. This is also the case for digits seven and one, as both have a similar shape (straight lines with sharp bends).

Intuitively, this makes sense as digits with similar shapes would have similar likelihoods for the same values of the same features. The finding above is corroborated by the confusion matrix (Figure 5): fives are the most misclassified class and are often confused with threes. Fives, sixes and eights are particularly often misclassified as zeros.

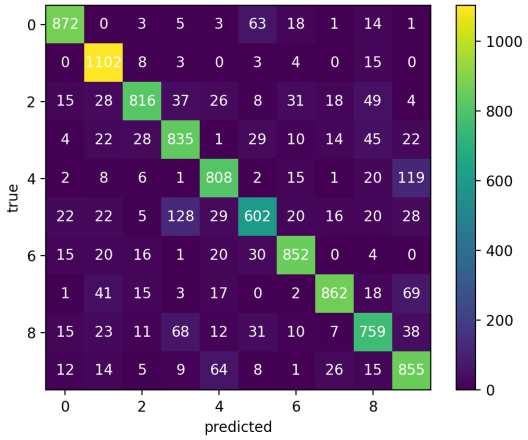

Figure 5: Confusion matrix.

Another way to visualise model performance is to order the predictions by their confidence (Figure 6). Here, confidence is equivalent to the probability that the model assigns to the class of a given example in the test set. Up until 7.000 samples in the test set, the model is 100% confident that the predicted label is the correct label and the accuracy of the predictions is also more or less constant after a sharp initial decline. After those first 7.000 samples, the confidence falls below the 100% mark and the accuracy steadily declines from then on.

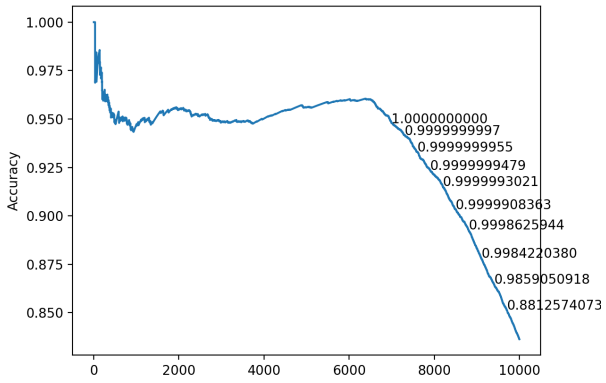

Figure 6: Predictions ordered by confidence.

4 Model selection (optional)

I ran out of time to implement k-fold cross-validation to find the "optimal" value of α. However, intuitively, I would expect the predictions to get worse with increases of $\alpha > 2$. To confirm this, I implemented a loop for α from 0 to 50 (code in the notebook is only for values up 3 as the runtime was too high). The accuracy of the model dropped, as expected, for values above two. Therefore, I would expect the "best" α to close or equal to two.

5 Generating data

5.1 a)

Here, the function nb_generate() uses the class conditional densities that were computed in section 1 to generate examples for a given class that follow the distribution of the training data.

5.2 b)

First off, it is important to understand what is learnt when nb_train() is called. The model learns the likelihood of all brightness intensities for all pixels, given their class. When nb_generate() is called, one samples from this likelihood distribution to generate new examples. This also means that for pixels that always take a certain value in a given class, the generated examples will also take that value for those pixels.

The previous is only applicable for $\alpha = 1$. For $\alpha = 2$ one obtains the likelihood distribution with add-one smoothing which remediates the zero-count problem. However, this has adverse effects on the "noisiness" of the generated examples as can be seen in Figure 7 and Figure 8.

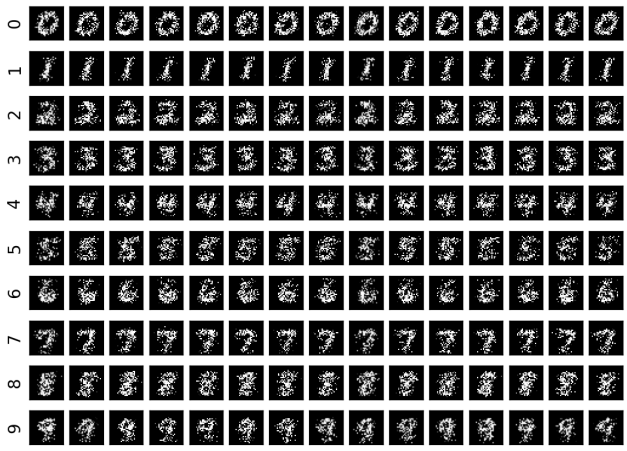

Figure 7: Generated examples with $\alpha = 1$.

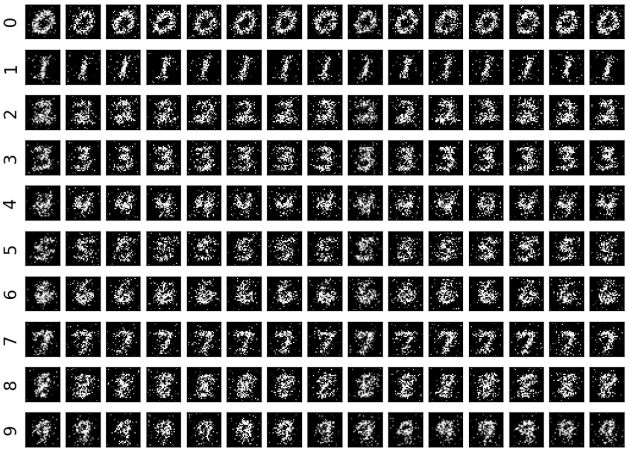

Figure 8: Generated examples with $\alpha = 2$.

Essentially, by increasing α, one increases the weights/probabilities placed on very unlikely feature values in each class. For example, it is reasonable to assume that the topmost left pixel is always black (i.e. has an intensity of zero). However, with add-one smoothing even for this pixel, the other intensities (1 to 255) become more likely (or rather: likely at all), so that, when drawing from the likelihood distribution, some of the generated examples might have nonzero intensities for that topmost left pixel. This "noisiness" will increase proportionately to the increase in α.

References

[Mur12] Kevin P Murphy. *Machine learning: a probabilistic perspective*. MIT press, 2012.