AI Classification Coursework 4 Readme File

April 30, 2021

Part I
SPAM FILTERING

1 DESCRIPTION OF PROBLEM - BACKGROUND

A spam email is defined as an email where:

1. The personal identity of the recipient of the email is not important.
2. The recipient did not give permission for the email to be sent.

Most spam emails either advertise something, or contain malware; and thus are unwanted.
Hence, we want to create a way to sort spam emails from non-spam (which we will call ham) emails automatically.
The aim of part 1 is to create an agent that differentiates between ham emails and spam emails.

2 HOW THE PROBLEM IS ABSTRACTED

Many spam emails:

1. Contain certain words (such as "money", "free", "receive", "bank", "transfer", "buy") and special characters (such as ":", "!", "$").
2. Lack certain words (such as "paper", "conference", "proposal", "experiment") and special characters (such as ",", ";").

This is how our agent will try to differentiate between ham email and spam email.
For each email, we will note each of these certain words / special characters as a feature, where:

- 0 represents the feature not appearing in the email
- 1 represents the feature appearing in the email

Then, we will let the agent learn which features appear in ham emails and which features appear in spam emails, and thus let the agent classify the emails as ham or spam.
We are given access to data extracted from 1500 emails (1000 in the "training_spam.csv" file and 500 in the "testing_spam.csv" file), where each email (in addition to being classified as 0 for ham or 1 for spam) has 54 features.

3 MEASURING HOW WELL THE AIM IS ACHIEVED

The aim of part 1 (as stated above) is to create an agent that differentiates between ham emails and spam emails.
How successfully this is done is quantified by the accuracy our agent achieves when classifying the emails.
The higher the accuracy is, the better the agent is at classifying the emails.
The accuracy is defined as:

$$\text{accuracy} = \frac{\text{number of emails correctly classified}}{\text{total number of emails classified}}$$

and is a number between 0 and 1 where:

- 0 represents no emails were correctly classified
- 1 represents all emails where correctly classified

To avoid over-fitting the data in order to gain a higher accuracy, our agent will be tested against unseen data.

4 CHOICE OF ALGORITHM

As stated above, the main priority is making the agent classify the emails correctly, and thus it must have a high accuracy.
A good implementation of Naïve Bayes ensures a high enough accuracy, and thus is a suitable choice of algorithm.
Hence, the algorithm I chose to implement in the notebook for part 1 is Naïve Bayes.
The event model chosen is multinomial and the algorithm modifications are Laplace smoothing and loging everything to increase numerical stability.
It is worth noting that there are other good algorithms for the purpose of classifying the emails correctly (such as Artificial Neural Networks, Support-Vector Machines, Logistic Regression, K-Nearest Neighbour).

4.1 NAÏVE BAYES EXPLAINED

Naïve Bayes is an algorithm used for classification problems that assumes conditional independence (i.e. that features are independent from each other).
More technically, if A and B are events representing features, then

$$P(A \cap B) = P(A) \cdot P(B)$$

The algorithm relies on the following formula:

$$P(C = c|x) = \frac{P(x|C = c) \cdot P(C = c)}{P(x)}$$

Where:

1. $P(C = c)$ represents the prior probability, which is the probability a given email is of class c before considering anything.
2. $P(x) = \sum_{i \in C} P(x|C = i) \cdot P(C = i)$ represents the prior probability of the predictor. This is constant over the classes.
3. $P(x|C = c)$ represents the likelihood, which is the probability of the predictor given it is class c.
4. $P(C = c|x)$ represents the posterior probability, which is the probability a given email is of class c given the predictor.

For a given email representation x, we are only interested in finding which class ($\widehat{c} \in C = \{\text{ham}, \text{spam}\}$) maximises $P(C = \widehat{c}|x)$, and hence classifying the email represented by x as class $\widehat{c}$.
Thus:

$$\widehat{c} = \underset{c \in C}{\text{argmax}}(P(C = c|x)) = \underset{c \in C}{\text{argmax}}(\frac{P(x|C = c) \cdot P(C = c)}{P(x)})$$

$$\overset{\text{As } P(x) \text{ is constant throught the classes}}{=} \underset{c \in C}{\text{argmax}}(P(x|C = c) \cdot P(C = c))$$

4.2 CHOICE OF EVENT MODEL

There are multiple models which can be used (being Multinomial, Gaussian and Bernoulli) for our Naïve Bayes algorithm, and each make assumptions of the distribution of data to estimate the likelihood $P(x|C = c)$.
I chose to use the Multinomial model, which means that an email *email* is represented as a subset (hence the order in which words appear in the email does not matter) of 54 keywords $email \subseteq \{k_1, \ldots, k_{54}\}$, and is abstracted as a vector $x = \begin{pmatrix} w_1 & w_2 & \ldots & w_{54} \end{pmatrix}$ where $w_i = \begin{cases} 1 & k_i \in email \\ 0 & k_i \notin email \end{cases}$.
As we are using Naïve Bayes, we may assume that conditional independence takes place, and hence
$P(w_i \cap w_j|C = c) = P(k_i, k_j \in email|C = c)$
$\overset{\text{By conditional independence}}{=} P(k_i \in email|C = c) \cdot P(k_j \in email|C = c)$
$= P(w_i|C = c) \cdot P(w_j|C = c)$
More generally,

$$P(w_1 \cap \ldots \cap w_{54}|C = c) = \prod_{i=1}^{54} P(w_i|C = c) = \prod_{i=1}^{54} \theta_{c,w_i}$$

where $\theta_{c,w_i} = \frac{\text{(number of times feature } w_i \text{ occurs for all emails in class } c)}{\text{(total number of features for all emails in class } c)} = \frac{n_{c,w_i}}{\sum_{i=1}^{54} n_{c,w_i}} = \frac{n_{c,w_i}}{n_c}$ (and is modified next section).
Hence, we get $P(email|C = c) = \prod_{i=1}^{54} \theta_{c,w_i} = (\prod_{i=1}^{54} (\theta_{c,w_i})^{w_i}) \cdot (\prod_{i=1}^{54} (\theta_{c,w_j})^{1-w_i})$
Which implies that $P(email|C = c) \propto \prod_{i=1}^{54} (\theta_{c,w_i})^{w_i}$
Hence, $\widehat{c} = \underset{c \in C}{\text{argmax}}(P(x|C = c) \cdot P(C = c))$
$\overset{\text{As (positive) proportionality is a strictly increasing function}}{=} \underset{c \in C}{\text{argmax}}(\prod_{i=1}^{54} (\theta_{c,w_i})^{w_i} \cdot P(C = c))$

4.3 ALGORITHM MODIFICATIONS

We derived above that $\widehat{c} = \text{argmax}(\prod_{i=1}^{54} (\theta_{c,w_i})^{w_i} \cdot P(C = c))$ where $\theta_{c,w_i} = \frac{n_{c,w_i}}{n_c}$.
However, there are a couple of problems with our current algorithm, and hence modifications that can be made to the algorithm to make it more accurate.

1. **Laplace Smoothing**

   Problem - Keyword not appearing in training set

   Consider the scenario when one of the keywords that appears in the email (i.e. keyword k_j) does not appear in the training data for a spam email ($c = 1$).

   This will imply that $P(x|C = 1) = \prod_{i=1}^{54} \theta_{c,w_i} = (\prod_{i=1}^{j-1} \frac{n_{c,w_i}}{n_c}) \cdot (\frac{n_{c,w_j}}{n_c}) \cdot (\prod_{i=j+1}^{54} \frac{n_{c,w_i}}{n_c}) = (\prod_{i=1}^{j-1} \frac{n_{c,w_i}}{n_c}) \cdot (\frac{0}{n_c}) \cdot (\prod_{i=j+1}^{54} \frac{n_{c,w_i}}{n_c}) = 0$

   $\implies 0 = P(x|C = 1) \propto \prod_{i=1}^{54} (\theta_{c,w_i})^{w_i}$

   $\implies \prod_{i=1}^{54} (\theta_{c,w_i})^{w_i} = 0$

   Hence, no matter how many other features suggest this is a spam email (if the email included many spam words such as “money”, “free”, “receive”, “bank”, “transfer”, “buy”), the classifier will still classify the email as ham as the k_j does not appear in the training data for spam email.

Solution - Laplace Smoothing

In order to deal with this, we can add a constant α to the numerator of every θ_{c,w_i} to ensure we never get 0 for any n_{c,w_i}.

To ensure that $\sum_{i=1}^{54} \frac{n_{c,w_i}}{n_c}$ still equals 1, we add $54 \cdot \alpha$ to the denominator to get $\theta_{c,w_i} = \frac{n_{c,w_i}+\alpha}{n_c+54\cdot\alpha}$

Note that what α does is decrease the variance between the data.

2. **Increasing Numerical Stability**

   Problem - Rounding / truncation errors

   In our algorithm, we end up multiplying many probabilities (which are numbers between 0 and 1), and hence the product may be a number very close to 0.

   A computer allocates a certain amount of memory for storing numbers, and hence, when a number requiring more memory than allocated needs to be stored, the computer may round / truncate the number, leading to rounding / truncation errors.

   Solution - Increasing Numerical Stability

   As *log* is a strictly increasing function, then

$$\widehat{c} = \underset{c\in C}{\operatorname{argmax}}(\prod_{i=1}^{54}(\theta_{c,w_i})^{w_i} \cdot P(C=c)) \overset{\text{As } log \text{ is a strictly increasing function}}{=} \underset{c\in C}{\operatorname{argmax}}(log(\prod_{i=1}^{54}(\theta_{c,w_i})^{w_i} \cdot P(C=c)))$$

   Due to the rules of *log*s (i.e. $log(AB) = log(A) + log(B)$ and $log(A^n) = n \cdot log(A)$), we can express this as the following:

$$\widehat{c} = \underset{c\in C}{\operatorname{argmax}}(log(\prod_{i=1}^{54}(\theta_{c,w_i})^{w_i} \cdot P(C=c))) = \underset{c\in C}{\operatorname{argmax}}(\sum_{i=1}^{54} w_i \cdot log(\theta_{c,w_i}) + log(P(C=c)))$$

   The reason this is better is since:

   (a) $log : [0,1] \to (-\infty, 0]$, and hence most numbers we will be dealing with will not be near 0.

   (b) Due to the rules of *log*s, we will sum the numbers rather than multiply them and hence massively reducing rounding / truncation errors.

Thus, the final formula for $\widehat{c}$ (after implementing the 2 modifications above) is:

$$\widehat{c} = \underset{c\in C}{\operatorname{argmax}}(log(P(C=c) + \sum_{i=1}^{54} w_i \cdot log(\theta_{c,w_i})))$$

where $\theta_{c,w_i} = \frac{n_{c,w_i}+\alpha}{n_c+54\cdot\alpha}$

5 ESTIMATING ACCURACY

In order to estimate the accuracy correctly (without overfitting) we must estimate our accuracy using similar conditions to the way it will be tested - unseen.

Thus, we must divide our data into training data and test data where the test data will just be used at the end to estimate the accuracy.

As stated above, we have access to 1500 emails.

I will divide the data into 1200 data points for training and 300 data points for testing.

5.1 TRAINING THE DATA

5.1.1 K-FOLD AND LEAVE-ONE-OUT CROSS VALIDATION

In k-fold cross validation, we partition the training data to k equal batches and assign $k-1$ batches to training the model and 1 batch to validation (i.e. estimating the accuracy by the formula explained above).
The cross validation process is repeated k times, where each time a different batch is considered as the validation batch.
Leave-one-out cross validation is a special case of k-fold cross validation where k is the number of data points (in our case it is $k = 1200$).
It is computationally expensive, but gives an accurate estimation (assuming the training data is distributed similarly to the test data).

5.1.2 CHOICE OF α

Notice that for different values of α, the agent might classify emails differently (as $\widehat{c}$ depends on α), hence affecting the accuracy of the algorithm.
Thus, we wish to find the value of α which maximises the accuracy.
Using the 1200 data points we have for training purposes, we will be using leave-one-out cross validation (i.e. k-fold cross validation where $k = 1200$) on different values of α to check which value yields the maximum accuracy.
This means using 1199 data points for training and 1 data point for validation (each time changing the data point used for validation) and then averaging the accuracies.
We will start by considering α values on a logarithmic scale to see which magnitude α should be in, then we will focus on a linear scale more deeply to find a good α value.

α value	10^{-4}	10^{-3}	10^{-2}	10^{-1}	10^{0}	10^{1}	10^{2}	10^{3}	10^{4}
Average accuracy to 4 d.p	0.8833	0.8833	0.8833	0.8833	0.8892	0.8858	0.8925	0.8658	0.6192

Hence we will be focusing on values between 10^{-1} and 10^{3} (to make sure we did not miss any peak points)
$0.1 - 0.9$

α value	0.1	0.2	0.3	0.4	0.5	0.6	0.7	0.8	0.9
Average accuracy to 4 d.p	0.8833	0.8833	0.8833	0.8833	0.8833	0.8833	0.8833	0.8833	0.8892

$1 - 9$

α value	1	2	3	4	5	6	7	8	9
Average accuracy to 4 d.p	0.8892	0.8875	0.8867	0.8850	0.8850	0.8850	0.8850	0.8850	0.8850

$10 - 90$

α value	10	20	30	40	50	60	70	80	90
Average accuracy to 4 d.p	0.8858	0.8892	0.8875	0.8883	0.8900	0.8883	0.8883	0.8917	0.8917

$100 - 900$

α value	100	200	300	400	500	600	700	800	900
Average accuracy to 4 d.p	0.8925	0.9017	0.9108	0.9067	0.9075	0.8992	0.8925	0.8808	0.8750

Hence we can deduce that a good α is between $200 - 400$
$200 - 290$

α value	210	220	230	240	250	260	270	280	290
Average accuracy to 4 d.p	0.9033	0.9050	0.9050	0.9050	0.9067	0.9067	0.9042	0.905	0.9075

$300 - 390$

α value	310	320	330	340	350	360	370	380	390
Average accuracy to 4 d.p	0.9083	0.9083	0.9075	0.9075	0.9075	0.9092	0.9092	0.9083	0.9083

Hence we will use the value $\alpha = 300$

5.2 TESTING THE DATA

Once training the data (using the 1200 training data points) on $\alpha = 300$ as chosen, we will use the 300 data points we have for testing purposes to see how accurate our algorithm is.
Running this, we get a 0.91 accuracy.
Thus, we can estimate that our agent's accuracy is 0.91.

Part II
FEATURE ENGINEERING FOR DIGIT CLASSIFICATION

6 DESCRIPTION OF PROBLEM - BACKGROUND

In this day and age, information is being highly digitalised.
This is due to many factors, including:

1. Data can be accessed quicker, as it is searchable.
2. The risk of losing data is highly reduced (compared to data on a piece of paper).
3. Data is accessable from multiple places.
4. Data is easily modifiable.

As there is a lot of data, we wish to automate the digitalisation process. This is where hand-written digit classification comes in.
In order to automate this process, the agent must be able to differentiate between different digits.
In addition, we wish to help the agent recognise the patterns in digits better by using feature engineering.
The aim of part 2 is to improve the agent's accuracy using feature engineering.

7 HOW THE PROBLEM IS ABSTRACTED

An image of a hand-written digit can be represented as a matrix of 28×28 where each entry represents a pixel in the image.
A pixel can take values between 0 (representing black) to 255 (representing white).
We must supply the agent with binary features which help differentiate between the images (for example, a binary feature might be "is there a line on the bottom of the digit" which helps differentiate the digits $0, 2, 3, 5, 6, 8$, from $1, 4, 7, 9$. This can be abstracted by considering how white, on average, the region at the bottom of the image is).
This is how our agent will try to differentiate between the images of the hand-written digits.
For each digit, we will note whether each of the features we defined exists or not, where 0 represents the feature not appearing in the image and 1 represents the feature appearing in the image.
Then, we will let the agent learn which features appear in which digits, and thus let the agent classify the images of the hand-written digits.
We are given access to the images of 600 hand-written digits (300 in the "training_digit_input.npy" file and 300 in the "test_digit_input.npy" file), where each image is classified from 0 to 9 in the respective label files "training_digit_label.npy" and "testing_digit_label.npy".

8 MEASURING HOW WELL THE AIM IS ACHIEVED

As stated above, the main priority of part 2 is to **improve** the agent's accuracy using feature engineering.
This can be quantified by considering the following formula:

Change in accuracy = accuracy achieved with new features − accuracy achieved with basic features

This is a number between -1 and 1, where:

- -1 implies the accuracy with basic features is 1 and the accuracy with new features is 0.
- 0 implies the accuracy with new features is the same as the accuracy with basic features.
- 1 implies the accuracy with basic features is 0 and the accuracy with new features is 1.

To avoid over-fitting the data in order to gain a higher accuracy, our agent will be tested against unseen data.

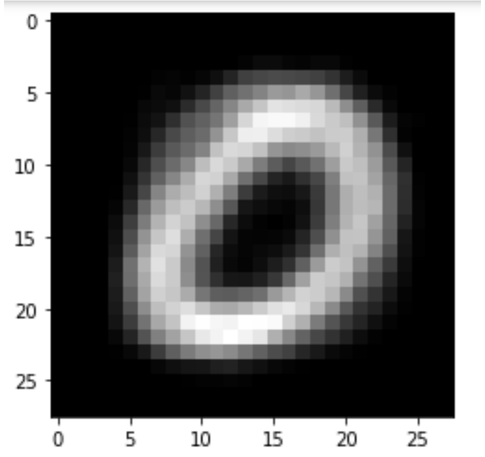
9 CHOICE OF ALGORITHM

As the aim of part 2 is to **improve** the classification accuracy via feature engineering, some algorithms which were good for part 1 may not be good for part 2 as the aim is different.
For example, Neural Networks can be thought of as "feature extractors", which means they extract their own features from features given by adjusting the importance of certain segments by adjusting the weights and biases.
Hence neural networks may get a high accuracy for the basic feature extractors and hence it will be hard to improve on the accuracy using feature engineering.
However, using Naïve Bayes ensures there is place to improve on the basic features, and hence is a suitable choice of algorithm.
Hence, my algorithm of choice for part 2 is Naïve Bayes.
Similar to part 1, the event model chosen is multinomial and the algorithm modifications are Laplace smoothing and *log*ing everything to increase numerical stability.

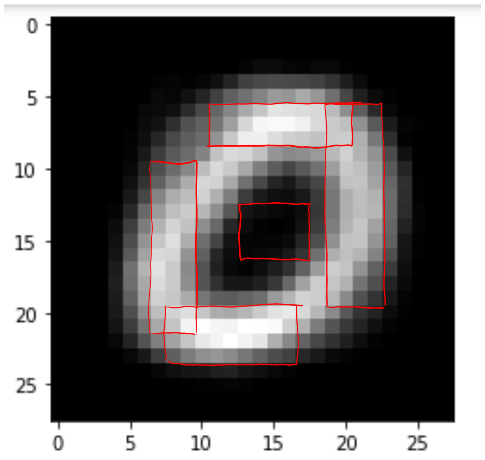
10 FEATURE ENGINEERING

10.1 MANUAL FEATURES

Originally, I attempted to manually find features.
This involved taking the mean of all the training data for each digit and looking for specific regions within the mean digit that are white (activated) and black (not activated).
For example, below is the average 0-digit that is created by taking the mean of each pixel of digits that were classified as 0.



A human could easily classify this as a 0, but expressing this to a computer is harder.
I originally gave features, being how white (activated) certain regions of the 0 are.



Consider the regions highlighted:

- The region in the centre has very little activation, hence this implies that nearly no training data which is classified as 0 has activation in the middle region (compared to a 3 or 8 which will have a high activation in the middle region).
- The regions on the 0 elipse have a very high activation, hence this implies that most / nearly all of the training data which is classified as 0 has activation in those regions (compared to a 7 which will have a low activation at the bottom and left regions and a 5 which will have a low activation at the right region).

However, these features are not binary (they represent **how** activated the region is as a number between 0 and 255).
To make the features binary, I introduced a bias, and:

- For regions with an activation lower than (or equal to) the bias, we give the feature a value of 0.
- For regions with activation higher than the bias, we give the feature a value of 1.

I picked values for the biases manually by considering what the mean of the training data yields, but this is problematic as the data has high variance (for example if the digit was shifted a bit on the picture).
This method (combined with the basic feature extractor) managed to get a $1.\dot{3}\%$ increase (from $0.80\dot{6}$ to 0.82). (I have commented out the 120 lines of code that went into this.)
Hence, I was dissuade from using manual feature engineering and turned to algorithms.

10.2 AUTOMATIC FEATURES

Instead of picking regions manually, I decided to loop over all the regions of certain size (arbitrarily being 4×4) and check the mean activation in them.
As we need binary features, I introduced a bias (b), as described above, and arbitrarily set its value to 130. This combined with the arbitrarily set $\alpha = 1$ got an increase of $3.\dot{6}\%$ (from $0.80\dot{6}$ to $0.84\dot{3}$).
After experimenting with the values manually, I managed to get a 4% increase (from $0.80\dot{6}$ to $0.84\dot{6}$) using 5×5, $b = 97, \alpha = 1$.
I then also noticed that the regions do not need to be square, and I managed to get a $4.\dot{3}\%$ increase (from $0.80\dot{6}$ to 0.85) using 2×15, $b = 97, \alpha = 1$.
However, this is using the testing that was given to us and not the leave-one-out cross validation, and in addition, this is not a systematic way of finding the best hyperparameter values.

10.2.1 CHOICES OF HYPERPARAMETER VALUES

As we did in part 1, we will be using leave-one-out cross validation, this time splitting the data to 480 points for training and 120 for testing.

However, as we have multiple hyperparameters, each hyperparameter choice will be a nested loop, hence requiring a lot more computation.
I will start with covering the entire scale by jumping large intervals for each hyperparameter to look at which areas we should focus our testing on.
The gaps for each variable are jumps of 5 for the height and width, jumps of 25 for the bias and logarithmic scale as above for α.
As there are 4 hyperparameters, this would ideally be given as a 4D table/graph, but as such things are hard to visualise, so instead I will sort the data for each variable and attempt to find a pattern from that.
After doing a grid search for the values explained above, an odd phenomenon appeared. In order to increase the change in accuracy (which is our aim), we can also try to make the accuracy of the basic feature extractor smaller by increasing α substantially. Hence, I managed to get a 30.8% increase (from 0.246 to 0.554 to 3d.p.) using $\alpha = 10000, 1 \times 6, b = 76$.
However, discussing this with the lead lecturer, I was told we should not consider such a high α (α should be within the same order of magnitude as the training data), and that $\alpha = 300$ is acceptable.
Thus, ignoring the $\alpha = 1000$ and 10000 results, the higher accuracies cluster around (height, width, bias) as follows:

1. $(1, 11 - 21, 76 - 126)$ for $\alpha = 300$ with $5.42\% - 7.5\%$ increase
2. $(1, 16 - 21, 76 - 126)$ for $\alpha = 100$ with $6.25\% - 7.5\%$ increase
3. $(1, 16 - 21, 76 - 126)$ for $\alpha = 10$ with $3.96\% - 4.79\%$ increase
4. $(1, 16 - 21, 76 - 126)$ for $\alpha = 1$ with $3.54\% - 4.38\%$ increase

Hence it would seem as though we should focus our search to $\text{height} = 1 - 5, \text{width} = 12 - 22, \text{bias} = 60 - 130, \alpha = 300$ (for the reason explained above).
After doing this second grid search, the top 5 values were (in format of (height, width, bias)):

1. $(3, 14, 93)$
2. $(3, 14, 92)$
3. $(2, 15, 92)$ (Notice this is very close to what I managed to find manually for $\alpha = 1$)
4. $(1, 18, 80)$
5. $(1, 17, 83)$

All of which have 10% increase.
Hence, we will consider $\text{height} = 3, \text{width} = 14, \text{bias} = 93, \alpha = 300$ on our test data.

10.2.2 ESTIMATING THE ACCURACY INCREASE

Once we have our hyperparameter values, we can use them and test the test data (the 120 testing data points).
We get 7.5% increase (0.692 to 0.767).
Thus, we estimate a 7.5% increase in accuracy.
It is worth noting there is some of fluctuation compared to the 10% estimated increase from the validation data as there is not a lot of data to use for the training and testing.