Report: Hidden Markov Models and Text Classification

Kobee Raveendran

September 24, 2020

1 HMM for DNA Sequencing

Below is the V-matrix and backtracking matrix, as well as the relevant Viterbi algorithm computations done from the start to end of the sequence. A more complete, handwritten version that shows *all* work is available in `misc/`.

1.1 Matrices and Decoded States

	T = 1	T = 2	T = 3	T = 4	T = 5
S_1	0.1	0.024	0.00384	0.0006144	0.000147456
S_2	0.2	0.016	0.00512	0.0016384	0.000131072

Table 1: V-matrix

	T = 1	T = 2	T = 3	T = 4	T = 5
S_1	<s>	S_1	S_1	S_1	S_1
S_2	<s>	S_2	S_2	S_2	S_2

Table 2: Backtracking matrix

Decoded sequence from backtracking matrix: $<s> \rightarrow S_1 \rightarrow S_1 \rightarrow S_1 \rightarrow S_1 \rightarrow S_1$

1.2 Viterbi Algorithm Walkthrough

$V_1(S_1) = P(S_1| < s >) \cdot P(C|S_1) = 0.1$
$V_1(S_2) = P(S_2| < s >) \cdot P(C|S_2) = 0.2$

$V_2(S_1|S_1) = V_1(S_1) \cdot P(S_1|S_1) \cdot P(G|S_1) = 0.024$
$V_2(S_1|S_2) = V_1(S_2) \cdot P(S_1|S_2) \cdot P(G|S_1) = 0.012$

$V_2(S_2|S_1) = V_1(S_1) \cdot P(S_2|S_1) \cdot P(G|S_2) = 0.002$
$V_2(S_2|S_2) = V_1(S_2) \cdot P(S_2|S_2) \cdot P(G|S_2) = 0.016$

$V_3(S_1|S_1) = V_2(S_1) \cdot P(S_1|S_1) \cdot P(T|S_1) = 0.00384$
$V_3(S_1|S_2) = V_2(S_2) \cdot P(S_1|S_2) \cdot P(T|S_1) = 0.00064$

$V_3(S_2|S_1) = V_2(S_1) \cdot P(S_2|S_1) \cdot P(T|S_2) = 0.00192$
$V_3(S_2|S_2) = V_2(S_2) \cdot P(S_2|S_2) \cdot P(T|S_2) = 0.00512$

$V_4(S_1|S_1) = V_3(S_1) \cdot P(S_1|S_1) \cdot P(C|S_1) = 0.0006144$
$V_4(S_1|S_2) = V_3(S_2) \cdot P(S_1|S_2) \cdot P(C|S_1) = 0.0002048$

$V_4(S_2|S_1) = V_3(S_1) \cdot P(S_2|S_1) \cdot P(C|S_2) = 0.0003072$
$V_4(S_2|S_2) = V_3(S_2) \cdot P(S_2|S_2) \cdot P(C|S_2) = 0.0016384$

$V_5(S_1|S_1) = V_4(S_1) \cdot P(S_1|S_1) \cdot P(A|S_1) = 0.000147456$
$V_5(S_1|S_2) = V_4(S_2) \cdot P(S_1|S_2) \cdot P(A|S_1) = 0.000098304$

$V_5(S_2|S_1) = V_4(S_1) \cdot P(S_2|S_1) \cdot P(A|S_2) = 0.000012288$
$V_5(S_2|S_2) = V_4(S_2) \cdot P(S_2|S_2) \cdot P(A|S_2) = 0.000131072$

Though not shown above, for each group calculated per timestep-state pair, the max of the two is used to populate the corresponding cell in the V-matrix and the max argument is used to populate the corresponding cell in the backtracking matrix, leading to the results above. The most likely hidden state sequence is then generated by starting from the final time step and tracing to the start, writing the content of the relevant backtracking matrix cell at each step. It is interesting to see that all hidden states are the same, but this is likely due to the high probability of getting to a state from itself (0.8), which can dominate the other terms of the multiplication as the algorithm progresses.

2 Binary Text Classification with Naive Bayes

2.1 Questions

The pruned dataset consists of 1,194 training examples (documents) throughout the entire set (600 of which are hockey and 594 of which are auto-related). The feature set for the training set consists of 18,748 unique lowercase words. The trained model is evaluated on the test set, which has a size of 795 (the training feature set is used, and any words unseen during training are ignored if encountered in the test set).

After training, the multinomial NB model achieves an F1 score of 98.35% on the positive class (hockey). To ensure that the model also performs decently on the autos class (by setting autos as the positive class), I calculated its F1 score as well, which was 98.37%.

2.2 Experimental Setup

2.2.1 Technologies Used

The program is written entirely in Python (3.7), and makes use of several helpful libraries: `spaCy` for better token extraction (i.e. separating symbols like '>' from email text) and part-of-speech tagging, `numpy` for vectorization, and `scikit-learn` for utility functions. Specifically, regarding utility, `sklearn` allowed for convenient index-consistent shuffling of the train/test sets and efficient model evaluation and scoring (calculating F1 scores, accuracy, etc.). I also used `sklearn`'s built-in multinomial Naive Bayes class to construct and fit data to the NB model and make predictions.

2.2.2 Preprocessing Steps

My preprocessing pipeline first excludes the header (begins recording relevant text after the number of lines is specified, excluding most of the superfluous text). In some rare cases, the number of lines is not specified, so recording is done after line 4. Then, all lines are read into memory and concatenated. This is done since the placement of words in a sentence can change their meaning, and accurate parts of speech from spaCy may not be guaranteed if that meaning is lost due to split sentences. This concatenated text is then run through spaCy's English language NLP model (which automatically splits the input by whitespace and other factors to retrieve tokens) to extract part-of-speech tags and check for stopwords.

In my implementation, I only consider purely-alphabetical tokens, and ignore stopwords and words that have a non-standard UPOS tag (any token that spaCy determines has a POS of 'X'). This ensures that all words are both entirely alphabetical and that they belong in the English language (in addition to proper nouns). Finally, the remaining valid words are lowercased and added to the feature set, so duplicates of words that appear in mixed cases are considered the same. The counts of each word are preserved on a per-sentence basis, and these counts are used with the feature mapping to form the feature vector for that document, where each feature vector is of length $|N|$, where N is size of the feature set (number of unique words). These are then added to the $M \times N$ training set matrix, where M is the number of documents in the training set.

2.3 Evaluation and Runtime

As mentioned in 2.1, I initially tested with F1 scores on the positive class. This led to issues later on when I discovered bugs where only the hockey class was loaded, resulting in high F1 scores for the positive class, but very low accuracy. So, for the sake of safety, I also computed F1 scores on the negative class by treating it as positive, as well as overall classification accuracy. The total accuracy over the test set is 98.36%.

Since the dataset is generated from scratch each time, the running time of the entire program is about 50-60 seconds on my machine. This slow execution time is likely due to the many file I/O operations being done on a large number of files. Once the dataset is preprocessed and loaded into arrays, the actual model fitting and predictions take a second or less.