David Jose Fernandez

NLP HW #2

1)

a) "This film was <u>hilarios</u>! I didn't <u>yawn</u> once. Not a single <u>bland</u> moment. Every minute was a <u>laugh</u>!"

$\log(x) = \ln(x)$

$$P(+|s) = P(+) \sum_{s \in S} \log P(s|+)$$

$$P(-|s) = P(-) \sum_{s \in S} \log P(s|-)$$

$$P(+|s) = .5[\log(\text{hilarious}|+) + \log(\text{yawn}|+) + \log(\text{bland}|+) + \log(\text{laugh}|+)]$$

$$= .5[\log(.143) + \log(.071) + \log(.071) + \log(.357)]$$

$$= -4.133$$

$$P(-|s) = .5[\log(\text{hilarious}|-) + \log(\text{yawn}|-) + \log(\text{bland}|-) + \log(\text{laugh}|-)]$$

$$= .5[\log(.091) + \log(.136) + \log(.091) + \log(.091)]$$

$$= -4.593$$

The model will choose funny (+) as the class.

b) P(+|S) & P(-|S) after add-1 smoothing

$P(+|S) = .5[\log(.143) + \log(.095) + \log(.095) + \log(.286)]$

$= -3.952$

$P(-|S) = .5[\log(.103) + \log(.138) + \log(.103) + \log(.103)]$

$=$ [illegible] -4.400

the labels didn't change, the decision was more notacibly one sided w/ smoothing applied.

c) An additional feature that could be extracted from S would be to use bigram counts instead of only using unigrams this would allow the model to capture more of the nuances in the text by taking into consideration longer sequences of text. You could also consider expanding the vocabulary to capture a better understanding of what other words are associated w/ positive and negative reviews.

2) show that $\|\theta^*\|_2^2 \le \|\hat{\theta}\|_2^2$

negative log likelihood

NLLH $\ell(\theta) = -\sum_{i=1}^{m} y^{(i)} \log h(x^{i}) + (1-y^{(i)}) \log(1-h(x^{(i)}))$

where m is the # of i.i.d. training examples and $h(x^{(i)}) = g(\theta^T x^{(i)}) = \frac{1}{1+e^{-\theta^T x}}$

$\ell(\theta)$ w/ L^2 Regularization $= \ell(\theta) + \frac{\lambda}{2}\|\theta\|_2^2$ call it $RL(\theta)$.

Since we are trying to minimize $\ell(\theta)$ we will use gradient descent to find the values of θ that minimize $\ell(\theta)$. In order to do this, we must find the gradient of $\ell(\theta)$ w/ respect to θ in order to apply the following update rule: $\theta_{t+1} := \theta_t - \alpha \nabla_\theta \ell(\theta)$.

For 1 datapoint (x,y)

$$\frac{\partial \ell}{\partial \theta_j} = \left(y \frac{1}{g(\theta^T x)} - (1-y)\frac{1}{1-g(\theta^T x)}\right) \frac{\partial}{\partial \theta_j} g(\theta^T x)$$

$$= \left(y\frac{1}{g(\theta^T x)} - (1-y)\frac{1}{1-g(\theta^T x)}\right) g(\theta^T x)(1-g(\theta^T x)) \frac{\partial}{\partial \theta_j}\theta^T x$$

$$= (y(1-g(\theta^T x)) - (1-y)\, g(\theta^T x))\, x_j$$

$$= (y - g(\theta^T x))\, x_j$$

☆ IMPORTANT this assumes that the initial weights for both models are the same

$$\frac{\partial RL}{\partial \theta_j} = \frac{\partial \ell}{\partial \theta_j} + \lambda \|\theta\|_2$$

☆ Given the update rule above and the partial derivatives of $RL(\theta)$ + $\ell(\theta)$ w/ respect to θ we can see why $\|\theta^*\|_2^2 \le \|\hat{\theta}\|_2^2$. When not using regularization ($\ell(\theta)$) the update rule becomes $\theta_{t+1} := \theta_t - \alpha(\sum_{i=1}^{m}(y - g(\theta^T x))x_i$ but when regularization is used ($RL(\theta)$) the rule becomes $\theta_{t+1} := \theta_t - \alpha(\sum_{i=1}^{m}(y-g(\theta^T x))x_i) - \lambda\|\theta\|_2$ which will always produce smaller weights except when $\lambda = 0$. $\therefore \|\theta^*\|_2^2 \le \|\hat{\theta}\|_2^2$.

3a)

For n-gram

$$P(W) = P(w_1, w_2, \ldots, w_k)$$
$$= \prod_{i=1}^{k} P(w_k \mid w_{k-1}, \ldots, w_{k-n+1})$$

For bi-gram

$$P(W) = P(w_1, w_2, \ldots, w_k)$$
$$= \prod_{i=1}^{k} P(w_k \mid w_{k-1})$$

3b) $P(w_i \mid w_{i-1}) = \dfrac{\text{count}(w_{i-1}, w_i)}{\text{count}(w_{i-1})}$

P([BOS] I like cheese made at home [EOS]) =
P(I | [BOS]) · P(like | I) · P(cheese | like) · P(made | cheese)
· P(at | made) · P(home | at) · P([EOS] | home)
$= .75 * .67 * .5 * .25 * .33 * 1 * .33 \approx \boxed{.00684}$

3c) Perplexity is the multiplicative inverse of the probability assigned to the test set by the language model, normalized by the # of words in the test set.

$$PP(W) = P(w_1 w_2 \cdots w_n)^{-\frac{1}{n}}$$
$$= \sqrt[n]{\frac{1}{P(w_1, w_2 \cdots)}}$$

For bi-grams this results in: $PP(w) = \sqrt[n]{\prod_{i=1}^{n} \frac{1}{P(w_i \mid w_{i-1})}}$

Perplexity of sequence in b:

$$PP(\text{sequence in b}) = \sqrt[8]{\frac{1}{.75} * \frac{1}{.67} * \frac{1}{.5} * \frac{1}{.25} * \frac{1}{.33} * 1 * \frac{1}{.33}}$$
$$= 1.8647$$

fix dinominator

3d) P(sequence in b after add-k smoothing) = .596 * .5 * .344 * .212 * .262 * .656 * .262

For k = .1 ≈ .00098

3e) P([BOS]I like pepperjack cheese [EOS]) using threshold of count >1 + add-.1 smoothing w/ bigrams.

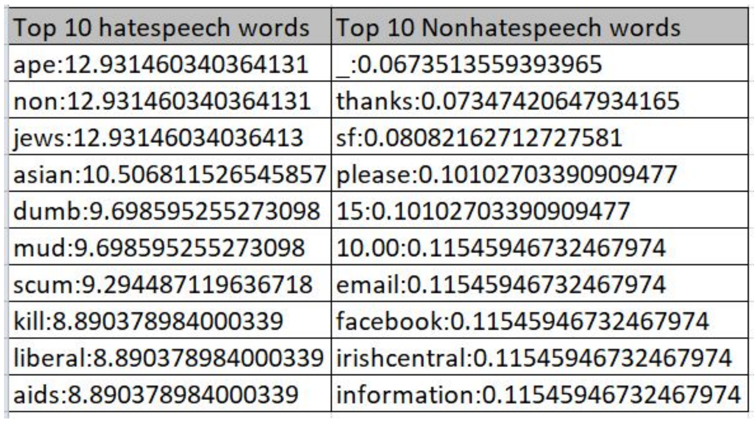
$P(s) = .0196 * .5122 * .0323 * .0323 * .2157$

$= 2.259 * 10^{-6}$

David Jose Fernandez
903206927

Hate Speech Classifier Analysis

4a)



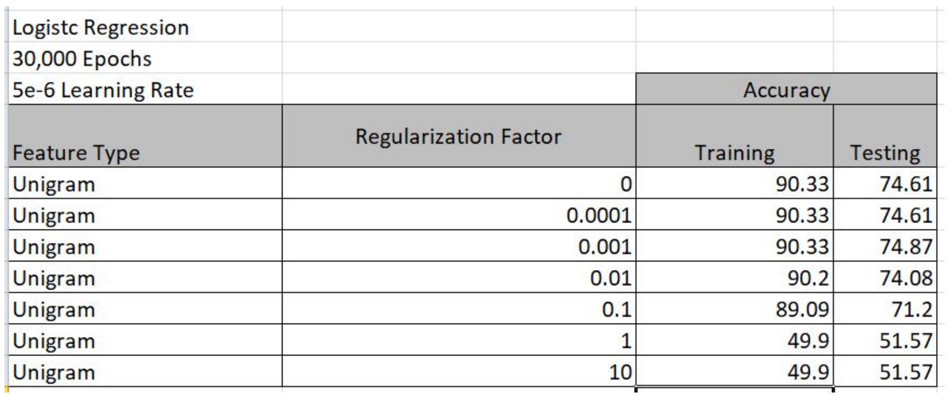
Top 10 hatespeech words	Top 10 Nonhatespeech words
ape:12.931460340364131	_:0.0673513559393965
non:12.931460340364131	thanks:0.07347420647934165
jews:12.93146034036413	sf:0.08082162712727581
asian:10.506811526545857	please:0.10102703390909477
dumb:9.698595255273098	15:0.10102703390909477
mud:9.698595255273098	10.00:0.11545946732467974
scum:9.294487119636718	email:0.11545946732467974
kill:8.890378984000339	facebook:0.11545946732467974
liberal:8.890378984000339	irishcentral:0.11545946732467974
aids:8.890378984000339	information:0.11545946732467974

We can see from the table above that the words with the lowest ratios seem to be those that are not crucial to the meaning of a sentence like numbers, acronyms, and filler words. Also passive words like “please” and “thanks” can be found in this list since those words are usually related to kind sentences. On the other hand, the top 10 hate speech words are those that usually refer to the subject of the sentence or describe the subject of the sentence in a negative way. It is also the case that these types of words usually refer to a specific race or religion.

4b) After running both the logistic regression model and the naive bayes model using unigram features the following accuracies were obtained:

- Naive Bayes
  - Training Accuracy: 92.1%
  - Testing Accuracy: 70.42%
- Logistic Regression
  - Training Accuracy: 90.33%
  - Testing Accuracy: 74.61%

From this we can see that both models performed very similarly with logistic regression getting an extra 4% accuracy during testing. This can be partially due to the fact that the logistic regression algorithm doesn’t make as many assumptions about the data set compared to the naive bayes model and can therefore have better generalization.



Logistc Regression			
30,000 Epochs			
5e-6 Learning Rate		Accuracy	
Feature Type	Regularization Factor	Training	Testing
Unigram	0	90.33	74.61
Unigram	0.0001	90.33	74.61
Unigram	0.001	90.33	74.87
Unigram	0.01	90.2	74.08
Unigram	0.1	89.09	71.2
Unigram	1	49.9	51.57
Unigram	10	49.9	51.57

In the image above we can see how the training and testing accuracy of the logistic regression model is affected based on how we tune the regularization factor lambda. A couple of interesting things can be observed from the gathered data. For one, it is interesting to note that when lambda is equal to .0001 the testing data is not affected at all, this is probably due to the fact that such a small lambda barely gives any importance to the regularization term when computing the gradient and therefore the change in accuracy is too small to be perceived when using 4 significant figures. Another interesting thing to note is that when lambda is set to too large of a value (i.e. 1 or 10) both the training and testing accuracy drop down to ~50%. Out of curiosity I decided to debug my code and see what was happening that was causing the accuracy to drop so low and I found that when a large lambda is chosen the regularization term grows extremely fast and quickly causes a numerical overflow in the program. This numerical overflow breaks all subsequent calculations and stops the learning process as a whole so the optimizer, in this case gradient descent, is not able to find the best weights for the model.

4c) For this part I decided to take a shot at implementing Term Frequency-Inverse Document Frequency (TF-IDF) as a feature extractor. I first implemented a TF-IDF feature extractor that used unigrams as terms and found that the accuracy of the model for both the naive bayes and the logistic regression failed to improve and in fact dropped by ~1-2%. As a way of trying to improve the accuracy I decided to use a combination of TF-IDF and bigrams in which the feature extractor consisted of calculating the TF-IDF value using bigrams as the terms in this case. To my initial surprise this caused the testing accuracy to drop really low ~55%, after doing some research I found that using bigram features doesn't always produce a model that is able to better generalize. This is due to the fact that using bigrams greatly increases the feature dimension of the model, in this case from ~5000 with unigrams to ~20000 with bigrams. This increase in the feature dimension can lead to overfitting due to the common issue of "the curse of dimensionality" which states that an exponential amount of training data is required relative to the feature dimensions. Once again, in order to improve the accuracy of the model I tried maxing out the number of bigrams used as terms by only keeping 5000, 7000, and 10000 most common bigrams. This approach also turned out to not improve the testing data beyond what was initially achieved using only unigram features. As a last attempt of improving the performance of the model I attempted to remove all special characters and punctuation from the text, as well as removing any URLS found. This once again failed to improve the accuracy of the model, in specific removing URLS from the text really seemed to hurt the accuracy since some of the tweets in the data set consisted of only URL preventing the model from extracting any information from those specific cases. In the end I was very happy about the feature extractor that I was able to engineer but was disappointed at the fact that it was not able to improve the accuracy of the model beyond what was initially achieved. I believe that given a larger data set in which a large feature space can be leveraged, my custom feature extractor would improve the performance of the overall model but given the simple dataset we were provided a simple approach that doesn't cause overfitting, like using unigrams, can give you the best performance.