Instructions

1. This homework has two parts: Q1, Q2 and Q3 are theory questions and Q4 is a programming assignment with some parts requiring a written answer. Each part needs to be submitted as follows:

   • Submit the answers to the theory questions as a pdf file on Canvas for the assignment corresponding to Homework 2 Theory. This should consist answers to Q1, Q2, Q3 and the descriptive answers from Q4. Name the pdf file as- `LastName_FirstName.pdf`. We recommend students type answers with LaTeX or word processors for this part. A scanned handwritten copy would also be accepted, try to be clear as much as possible. No credit may be given to unreadable handwriting.

   • The programming assignment requires you to work on boilerplate code. Submit the answers to the programming assignment in a zip that contain all the code files. This submission is to be made on Canvas for the assignment corresponding to Homework 2 Programming. Name the zip file as- `LastName_FirstName.zip`.

2. For the theory questions, write out all steps required to find the solutions so that partial credit may be awarded.

3. The second question is meant for graduate students only. Undergraduate students do not need to attempt Q2. Each of the other three questions is mandatory for all students. There is no extra credit for answering additional questions than what is required.

4. We generally encourage collaboration with other students. You may discuss the questions and potential directions for solving them with another student. However, you need to write your own solutions and code separately, and not as a group activity. Please list the students you collaborated with.

5. The code files needed to complete the homework are included in a zip file on Canvas.

1. A collection of reviews about comedy movies (data $\mathcal{D}$) contains the following keywords and binary labels for whether each movie was funny (+) or not funny (-). The data are shown below: for example, the cell at the intersection of "Review 1" and "laugh" indicates that the text of Review 1 contains 2 tokens of the word "laugh."

Review	laugh	hilarious	awesome	dull	yawn	bland	Y
1	2	1	1	1	1	0	1
2	0	1	2	0	0	0	1
3	3	0	0	0	0	1	1
4	0	1	0	2	1	0	0
5	1	1	1	2	0	2	0
6	1	0	0	2	2	0	0

You may find it easier to complete this problem if you copy the data into a spreadsheet and use formulas for calculations, rather than doing calculations by hand. Please report all scores as **log-probabilities**, with 3 significant figures. [10 pts]

(a) Assume that you have trained a Naive Bayes model on data $\mathcal{D}$ to detect funny vs. not funny movie reviews. Compute the model's predicted score for funny and not-funny to the following sentence S (i.e. $P(+|S)$ and $P(-|S)$), and determine which label the model will apply to S. [4 pts]
S: "This film was hilarious! I didn't yawn once. Not a single bland moment. Every minute was a laugh."

(b) The counts in the original data are sparse and may lead to overfitting, e.g. a strong prior on assigning the "not funny" label to reviews that contain "yawn." What would happen if you applied *smoothing*? Apply add-1 smoothing and recompute the Naive Bayes model's predicted scores for S. Did the label change? [4 pts]

(c) What is an additional feature that you could extract from text to improve the classification of sentences like S, and how would it help improve the classification? [2 pt]

Solution.
NOTE: All logarithms computed in this question use base e

a) In the provided sentence, given the bag of words in the table, we can gather sentence S has the following vector representation: [1, 1, 0, 0, 1, 1]. This represents S has 1 laugh token, 1 hilarious token, 1 yawn token, and 1 bland token, with 0 awesome and dull tokens. The following probabilities of each token in each class can be represented with the following equation: $P(\text{w}|\text{class}) = \frac{\text{count(w, class)}}{\sum_{w \in V} \text{count(w, class)}}$. The probabilities of each token in each class can be seen below:

Class	laugh	hilarious	awesome	dull	yawn	bland
1	0.385	0.154	0.231	0.077	0.077	0.077
0	0.125	0.125	0.063	0.375	0.188	0.125

Now, with the independence assumption of Naive Bayes, we can calculate $P(\text{class}|\text{S}) = \frac{P(\text{S}|\text{class})P(\text{class})}{P(S)} = P(\text{S}|\text{class})P(\text{class}) = P(\text{S}|\text{class})$. We can remove the $P(S)$ because it's the same value for both classes. Additionally, we can remove the $P(\text{class})$ for the same reason. Including either probability would be pointless when it comes to classification. So, when we go to calculate the probability of the sentence belonging to each class, we can calculate the log probability $\log \prod_{w \in V} P(\text{S}|\text{class})I(w \in S) = \sum_{w \in V} \log(P(\text{S}|\text{class})I(w \in S))$. As such, we can now calculate the scores for each class as follows:

$P(\text{S}|+) = \log P(+) + \log(P(\text{laugh}|+)I(w \in S)) + \log(P(\text{hilarious}|+)I(w \in S)) + \log(P(\text{awesome}|+)I(w \in S)) + \log(P(\text{dull}|+)I(w \in S)) + \log(P(\text{yawn}|+)I(w \in S)) + \log(P(\text{bland}|+)I(w \in S)) = \log P(0.5) + \log(P(\text{laugh}|+) \cdot 1) + \log(P(\text{hilarious}|+) \cdot 1) + \log(P(\text{awesome}|+) \cdot 0) + \log(P(\text{dull}|+) \cdot 0) + \log(P(\text{yawn}|+) \cdot 1) + \log(P(\text{bland}|+) \cdot 1) = -0.6931 - 0.95551 - 1.8718 - 2.56495 - 2.56495 = -8.65031$

$P(\text{S}|\text{-}) = \log P(-) + \log(P(\text{laugh}|\text{-})I(w \in S)) + \log(P(\text{hilarious}|\text{-})I(w \in S)) + \log(P(\text{awesome}|\text{-})I(w \in S)) + \log(P(\text{dull}|\text{-})I(w \in S)) + \log(P(\text{yawn}|\text{-})I(w \in S)) + \log(P(\text{bland}|\text{-})I(w \in S)) = \log P(0.5) + \log(P(\text{laugh}|\text{-}) \cdot 1) + \log(P(\text{hilarious}|\text{-}) \cdot 1) + \log(P(\text{awesome}|\text{-}) \cdot 0) + \log(P(\text{dull}|\text{-}) \cdot 0) + \log(P(\text{yawn}|\text{-}) \cdot 1) + \log(P(\text{bland}|\text{-}) \cdot 1) = -0.6931 - 2.07944 - 2.07944 - 1.67398 - 2.07944 = -8.6054$

We can see $P(\text{S}|\text{-}) > P(\text{S}|+)$, so we assign the label of not funny to this sentence.

b) After applying add-1 smoothing, the probabilities of each word for each class has changed to the following:

Class	laugh	hilarious	awesome	dull	yawn	bland
1	0.316	0.158	0.211	0.105	0.105	0.105
0	0.136	0.136	0.091	0.318	0.182	0.136

Also, the scores (log probabilities) for each class has changed to the following:

$P(\text{S}|+) = -0.6931 - 1.15268 - 1.84583 - 2.25129 - 2.25129 = -8.1942$

$P(\text{S}|\text{-}) = -0.6931 - 1.99243 - 1.99243 - 1.70475 - 1.99243 = -8.3751$

So, now we see $P(\text{S}|+) > P(\text{S}|\text{-})$, therefore we label this sentence as funny. So, yes, with add-1 smoothing, the label did change.

c) The most important to add to this algorithm, in my opinion, is the bigram feature. Currently, we use "yawn" and "bland" independently, as well as "didn't" and "yawn" independently. However, with a bigram feature, those words would be clumped together into "(single, bland)" or "(didn't, yawn)", which provide more context than naively only paying attention to the word "yawn". As such, this feature, as well as higher

Markov orders (e.g. trigrams) would improve accuracy by taking into account the neighbor words of each word, providing more context for classification later in the process.

Another fun feature (that I thought I would mention) I would extract from the text is its sentiment analysis. This has an immediate and obvious effect on classifying the sentences. For example, if a sentence has a generally negative sentiment, there should be less chance of the review being funny; negativity and funny are somewhat opposites, most of the time. Meanwhile, if somebody found the comedy movie funny, they most likely wrote a review with a positive sentiment. As such, this feature has clear advantages to improving classification of sentences like S.

■

2. **[CS 7650 Only]**

Assume that you are training several logistic regression models. After training on the same data, $\hat{\theta}$ is the optimal weight for an unregularized logistic regression model and θ^* is the optimal weight for a logistic regression model with L2 regularization. Prove that $||\theta^*||_2^2 \leq ||\hat{\theta}||_2^2$.
Note: you may find it useful to look at the likelihood equations for regularized and unregularized logistic regression. [5 pts]

Solution.
Let $L(\theta) = -\log(\theta)$ be the negative log-likelihood loss function for logistic regression, while $L_{\text{reg}}(\theta) = L(\theta) + R(\theta) = -\log(\theta) + ||\theta||_2^2$ is the L2 regularized version of the previous loss function ($R(\theta)$ indicates the regularization term).

Let $\hat{\theta} = \text{argmin}_\theta\ L(\theta)$.

Let $\theta^* = \text{argmin}_\theta\ L_{\text{reg}}(\theta)$.

We know $L(\hat{\theta}) \leq L(\theta^*)$ since $\hat{\theta}$ is a global minimum for the non-regularized loss function. Conversely, we know $L_{\text{reg}}(\theta^*) \leq L_{\text{reg}}(\hat{\theta})$ since θ^* is a global minimum for the regularized loss function. The only time either equality holds is if $\theta^* = \hat{\theta}$.

Since $L_{\text{reg}}(\theta^*) \leq L_{\text{reg}}(\hat{\theta})$, we get the following:

$L_{\text{reg}}(\theta^*) \leq L_{\text{reg}}(\hat{\theta})$

$L(\theta^*) + R(\theta^*) \leq L(\hat{\theta}) + R(\hat{\theta})$

$L(\theta^*) + ||\theta^*||_2^2 \leq L(\hat{\theta}) + ||\hat{\theta}||_2^2$

We have already shown above $L(\hat{\theta}) \leq L(\theta^*)$, so we can subtract both loss function values from the inequality and it will still hold:

$||\theta^*||_2^2 \leq ||\hat{\theta}||_2^2$

$\therefore ||\theta^*||_2^2 \leq ||\hat{\theta}||_2^2$

■

3. Language Modeling is the technique that allows us to compute the probabilities of word sequences. The probability of a sequence $\mathbf{W} = w_1^n = \{w_1, w_2...w_n\}$, with the use of chain rule, can be estimated as the product of probabilities of each word given the history, as shown-

$$\begin{aligned} P(\mathbf{W}) &= P(w_1, w_2...w_n) \\ &= P(w_1)\ P(w_2|w_1)\ P(w_3|w_1, w_2)...P(w_n|w_1, w_2...w_{n-1}) \\ &= \prod_{i=1}^{n} P(w_i|w_1^{i-1}) \end{aligned}$$

   (a) Using an n-gram model allows us to approximate the above probability using only a subset of of $n-1$ words from the history at each step. Simplify the above expression for the general n-gram case, and the bi-gram case. [3 pts]

   (b) A common way to have markers for the start and the end of sentence is to add the [BOS] (beginning of sentence) and [EOS] (end of sentence) tokens at the start and end of every sentence. Consider the following text snippet-

   [BOS] i made cheese at home [EOS]
   [BOS] i like home made cheese [EOS]
   [BOS] cheese made at home is tasty [EOS]
   [BOS] i like cheese that is salty [EOS]

   Using the expression derived in (a), find the probability of the following sequence as per the bi-gram model- P([BOS] I like cheese made at home [EOS]). [5 pts]

   (c) In practice, instead of raw probability, perplexity is used as the metric for evaluating a language model. Define perplexity and find the value of perplexity for the sequence in (b) for the bi-gram case. [2 pts]

   (d) One way to deal with unseen word arrangements in the test set is to use Laplace smoothing, which adds 1 to all bi-gram counts, before we normalize them into probabilities. An alternative to Laplace smoothing (add-1 smoothing) is add-k smoothing, where k is a fraction that allows assigning a lesser probability mass to unseen word arrangements. Find the probability of the sequence in (b) with add-k smoothing for $k = 0.1$. [5 pts]

   (e) To deal with unseen words in the test set, a common way is to fix a vocabulary by thresholding on the frequency of words, and assigning an [UNK] token to represent all out-of-vocabulary words. In the example from (a), use a threshold of $count > 1$ to fix the vocabulary. Find the probability for the following sequence for an add-0.1 smoothed bi-gram model- P([BOS] i like pepperjack cheese [EOS]). [5 pts]

Solution.
a) n-gram case:
$P(\mathbf{W}) = \prod_{i=1}^{N} P(w_i|w_{i-n}^{i-1})$

bi-gram case:

$P(\mathbf{W}) = \prod_{i=1}^{N} P(w_i|w_{i-1})$

Note: in the bigram model, when $i = 1$, the token at w_{i-1} is the start token, which in this question is denoted as [BOS]. Similarly, in the n-gram case, the token located at w_0 is [BOS] as well. Honestly, this also kind of just depends on how you implement both models (i.e. instead of putting [BOS] at w_0, you can put it at w_1, which is the opposite of what I have described in the previous sentence, which means whenever you calculate the probability of [BOS], you get $P(w_1) = P([\text{BOS}]) = 1$), but it should not make a difference in the calculations.

b) $P(S) = P(\text{I}|[\text{BOS}])P(\text{like}|\text{I})P(\text{cheese}|\text{like})P(\text{made}|\text{cheese})P(\text{at}|\text{made})P(\text{home}|\text{at})P([\text{EOS}]|\text{home})$
$= 0.75 \cdot \frac{2}{3} \cdot 0.5 \cdot 0.25 \cdot 0.33 \cdot 1 \cdot 0.33$
$= 0.00694$

c) Perplexity is the inverse probability of a test sentence, which is then normalized by the sentence's length. The normalization is used to control for the nature of long sentences, where many more probabilities are multiplied together than a short sentence (so, the model would be biased toward shorter sentences without the normalization). The equation is as follows:

$\text{PP}(S) = \sqrt[N]{\frac{1}{P(S)}} = \sqrt[N]{\frac{1}{P(w_1, w_2, ..., w_n)}}$
$= \sqrt[N]{\prod_{i=1}^{N} \frac{1}{P(w_i|w_{i-1})}}$ (for the bi-gram case)

From the above equation, it is easy to see why low perplexity is good and high perplexity is bad (we are using $\frac{1}{P(w_i|w_{i-1})}$ instead of $P(w_i|w_{i-1})$ directly). For the sentence given in part (b), the perplexity is as follows for the bi-gram model:

$$\text{PP}(S) = \sqrt[N]{\frac{1}{P(\text{I}|[\text{BOS}])} \frac{1}{P(\text{like}|\text{I})} \frac{1}{P(\text{cheese}|\text{like})} \frac{1}{P(\text{made}|\text{cheese})} \frac{1}{P(\text{at}|\text{made})} \frac{1}{P(\text{home}|\text{at})} \frac{1}{P([\text{EOS}]|\text{home})}}$$
$$= \sqrt[8]{\frac{1}{P(\text{I}|[\text{BOS}])P(\text{like}|\text{I})P(\text{cheese}|\text{like})P(\text{made}|\text{cheese})P(\text{at}|\text{made})P(\text{home}|\text{at})P([\text{EOS}]|\text{home})}}$$
$$= \sqrt[8]{\frac{1}{0.00694}}$$
$$= \sqrt[8]{144.0922}$$
$$= 1.8613$$

d) $P(S) = P(\text{I}|[\text{BOS}])P(\text{like}|\text{I})P(\text{cheese}|\text{like})P(\text{made}|\text{cheese})P(\text{at}|\text{made})P(\text{home}|\text{at})P([\text{EOS}]|\text{home})$
$= \frac{3+0.1}{4+0.1 \cdot 12} \frac{2+0.1}{3+0.1 \cdot 12} \frac{1+0.1}{2+0.1 \cdot 12} \frac{1+0.1}{4+0.1 \cdot 12} \frac{1+0.1}{0.3+0.1 \cdot 12} \frac{2+0.1}{2+0.1 \cdot 12} \frac{1+0.1}{3+0.1 \cdot 12}$
$= (0.5962)(0.5)(0.3438)(0.2115)(0.2619)(0.6563)(0.2619)$
$= 0.000978$

e) Using the examples from (b), we can construct a training set of vocabulary. As such, all the words with count > 1 are "[BOS]", "[EOS]", "i", "made", "cheese", "at", "home", "like", and "is". All other tokens should be considered out-of-vocabulary ([UNK]). As such, the sentence provided to us, "[BOS] i like pepperjack cheese [EOS]", is transformed into "[BOS] i like [UNK] cheese [EOS]". Therefore, to calculate P([BOS] i like [UNK] cheese [EOS]) with add-0.1 smoothing in a bi-gram model, we can combine

what we have learned from previous parts of this question to create the following formula:

$$
\begin{aligned}
&P(S) \\
&= P(\text{I}|\text{[BOS]})P(\text{like}|\text{I})P(\text{[UNK]}|\text{like})P(\text{cheese}|\text{[UNK]})P(\text{[EOS]}|\text{cheese}) \\
&= \frac{3+0.1}{4+0.1\cdot 10}\frac{2+0.1}{3+0.1\cdot 10}\frac{0+0.1}{2+0.1\cdot 10}\frac{0+0.1}{3+0.1\cdot 10}\frac{1+0.1}{4+0.1\cdot 10} \\
&= (0.62)(0.525)(0.0333)(0.025)(0.22) \\
&= 0.0000597
\end{aligned}
$$

■

4. In this problem, you will do text classifications for Hate Speech. You need both answer the questions and submit your codes.

   Hate speech is a

   (a) **deliberate attack,**

   (b) **directed towards a specific group of people,**

   (c) **motivated by aspects of the group's identity.**

   The three premises must be true for a sentence to be categorized as HATE. Here are two examples:

   (a) "Poor white kids being forced to treat apes and parasites as their equals."

   (b) "Islam is a false religion however unlike some other false religions it is crude and appeals to crude people such as arabs."

   In (a), the speaker uses "apes" and "parasites" to refer to children of dark skin and implies they are not equal to "white kids". That is, it is an attack to the group composed of children of dark skin based on an identifying characteristic, namely, their skin colour. Thus, all the premises are true and (a) is a valid example of HATE. Example (b) brands all people of Arab origin as crude. That is, it attacks the group composed of Arab people based on their origin. Thus, all the premises are true and (b) is a valid example of HATE.

   This problem will require programming in **Python 3**. The goal is to build a **Naive Bayes model** and a **logistic regression model** that you learnt from the class on a real-world hate speech classification dataset. Finally, you will explore how to design better features and improve the accuracy of your models for this task.

   The dataset you will be using is collected from Twitter online. Each example is labeled as 1 (hatespeech) or 0 (Non-hatespeech). To get started, you should first download the data and starter code from https://www.cc.gatech.edu/classes/AY2020/cs7650_spring/programming/h2_text_classification.zip. Try to run:

   ```
   python main.py -- model AlwaysPredictZero
   ```

   This will load the data and run a default classifier `AlwaysPredictZero` which always predicts label 0 (non-hatespeech). You should be able to see the reported train accuracy = 0.4997. That says, always predicting non-hatespeech isn't that good. Let's try to build better classifiers!

Note that you need to implement models without using any machine learning packages such as `sklearn`. We will only provide train set, and we will evaluate your code based on our test set.

To have a quick check with your implementations, you can randomly split the dataset we give you into train and test set at a ration 8:2, compare the accuracy between the models you have implemented and related models in `sklearn` packages. You would expect an accuracy at around 0.65 (or above) on your test set.

(a) **(Naive Bayes)** In this part, you should implement a Naive Bayes model with add-1 smoothing, as we taught in the class. You are required to implement the `NaiveBayesClassifier` class in `classifiers.py`. You would probably want to take a look at the `UnigramFeature` class in `utils.py` that we have implemented for you already. After you finish your codes, run `python main.py --model NaiveBayes` to check the performance. List the 10 words that, under your model, have the higest ratio of $\frac{P(w|1)}{P(w|0)}$ (the most distinctly hatespeech words). List the 10 words with the lowest ratio. What trends do you see? [25 pts]

(b) **(Logistic Regression)** In this part, you should implement a Logistic Regression model. You are required to implement the `LogisticRegressionClassifier` class in `classifiers.py`. First, implement a logistic regression model without regularization and run `python main.py --model LogisticRegression`, compare the performance with your Naive Bayes approach. Next, we would like to experiment with L2 regularization, add L2 regularization with different weight such as $\alpha = \{0.0001, 0.001, 0.01, 0.1, 1, 10\}$, describe what you observed. (You may want to split the train set we give you into your own train and test set to observe the performance) [25 pts]

(c) **(Features)** In the last part, you'll explore and implement a more sophisicated set of features. You need to implement the class `BigramFeature` or modify the class `CustomFeature` in`utils.py`. Here are some common strategies (you are welcome to implement some of them but try to come up with more!):

i. Remove stopwords (e.g. a, the, in),
ii. Use a mixture of unigrams, bigrams or trigrams,
iii. Use TF-IDF (refer to http://www.tfidf.com/) features.

Use your creativity for this problem and try to obtain an accuracy as high as possible on your test set! After you implement `CustomFeature` , run:

```
python main.py --model NaiveBayes -- feature customized
python main.py --model LogisticRegression -- feature customized
```

Describe the features that you have implemented. We'll evaluate your two models on the test set. [Bonus: 10 points]

You will receive up to 10 bonus points: up to 5 points based on the novel features you try and the rest based on how well your models perform compared to other submissions:

$$Bonus = 5 + 5 * \frac{1}{rank}$$

e.g. if you rank first in the class, you will receive the full bonus point! We will share the winners' codes as well.

Solution.
a) The most distinct hate speech words that were discovered are:

(a) non
(b) asian
(c) ape
(d) mud
(e) liberal
(f) dumb
(g) filth
(h) kill
(i) non-white
(j) aids

Clearly, there are some slurs here (e.g. ape), as well as racial characteristics which are common occurrences in hate speech (e.g. asian, non-white, mud), violent verbs and adjectives (e.g. filth, kill, dumb), stereotypical traits (e.g. AIDs for blacks), and political attacks (e.g. liberal because the model might have found most hate groups are non-liberal). Finally, I believe "non" is in there because hate groups are associated with attacking people outside of their group, so they are "non-white", "non-American", or other characteristics.

On the other side, the least distinct hate speech words are as follows:

(a) thanks
(b) sf
(c) html
(d) sports
(e) information
(f) irishcentral
(g) spirit
(h) email
(i) report
(j) facebook

Please note I left out tokens such as "15", "[", and "]" because those are not words. In this list, we clearly observe "thanks" is the most distinct non-hate speech word, which makes sense because it is associated with being a caring person. Additionally, there are more neutral or happy words, such as "spirit", "sports", and "email". I am unsure whether "sf" is associated with a specific acronym, or if it is short for San Francisco. If it stands for San Francisco, that makes sense because it is a typically left-leaning political area, which contrasts with the list we found above that included "liberal" as a hate speech word.

b) **NOTE: All simulations were done with an 80/20 train/test split.**

Naive Bayes (unigram, add-1 smoothing): 76.34%

Logistic Regression (no regularization, $\eta = 0.01$, 200 epochs): 68.82%

Logistic Regression (regularization w/ $\alpha = 0.0001$, $\eta = 0.05$, 100 epochs): 71.51%

Logistic Regression (regularization w/ $\alpha = 0.001$, $\eta = 0.02$, 100 epochs): 72.58%

Logistic Regression (regularization w/ $\alpha = 0.01$, $\eta = 0.005$, 160 epochs): 69.35%

Logistic Regression (regularization w/ $\alpha = 0.1$, $\eta = 0.0005$, 100 epochs): 66.13%

Logistic Regression (regularization w/ $\alpha = 1$, $\eta = 0.0005$, 100 epochs): 63.17%

Logistic Regression (regularization w/ $\alpha = 10$, $\eta = 0.00005$, 50 epochs): 58.87%

From the above results, we can conclude two things. One, we see the logistic regression classifier performs worse than the Naive Bayes with unigram modeling and add-1 smoothing (76.34% vs. 68.82% accuracy).

Second, logistic regression with a moderate amount of regularization increases performance by a decent amount. We can see with the lowest amount of regularization ($\alpha = 0.0001$), the performance increased about 3%, but with a little bit more regularization ($\alpha = 0.001$), the performance increases a bit more until extra regularization begins to decrease the accuracy. With a large amount of regularization, the regularization term in the loss overpowers the actual log likelihood loss (based on the error of predicted values vs. ground-truth values), leading to the loss function basically prioritizing the weights being close to 0, leading to a massive drop in performance. As such, what we can conclude is a little bit of regularization can help, but a lot of regularization can hurt a lot.

c) **NOTE: All simulations were done with an 80/20 train/test split.**

(Naive Bayes) Replacing words in bottom 90% of counts (count ≤ 7) with "UNK": 68.55%

(Naive Bayes) Replacing words in bottom 75% of counts (count ≤ 3) with "UNK": 73.39%

(Naive Bayes) Replacing words in bottom 60% of counts (count ≤ 2) with "UNK": 74.19%

(Naive Bayes) Replacing words in bottom 25% of counts (count ≤ 1) with "UNK": 75.81%

(Naive Bayes) Bigram model: 70.16%

(Naive Bayes) TF-IDF: 48.66%

(Naive Bayes) Removing NLTK stopwords: 74.46%

(Naive Bayes) Unigrams + trigrams model: 73.12%

As seen from above, none of these models or custom features can beat the Naive Bayes unigram model with add-1 smoothing. All of these implementations are left in utils.py to ensure the TA's can see I actually implemented them. I did not perform any simulations with the logistic regression model because the best regularized model ($\alpha = 0.001$) performed 4% worse than the unigram NB model with add-1 smoothing. As such, with the above features, since none of them increased accuracy on the NB model, I believe there is reasonable doubt a logistic regression with this features would fail to surpass the 76.34% accuracy benchmark set by the NB model discussed above. I truly did try to be creative with all the features possible. I tried to replace infrequent words (infrequent defined by the percentile of their frequency) with the "UNK" token, with varying percentile numbers, to remove what I deem useless or unimportant words, but that did not work. Then, I tried the vanilla bigram model, and that failed to work as well. I implemented TF-IDF to weight the counts of each word in the unigram model, but that failed as well. Then, I did the naive approach of removing all the stopwords in NLTK, but that decreased accuracy by about 2%. Furthermore, if you look at the 'preprocess_word' function in my CustomFeature class, you will see I tried experimenting with a custom stopword list (I went with this approach because the NLTK stopword list is a bit aggressive, and I felt like it removed too many words), based primarily around gender pronouns, but that failed even more than the NLTK stopwords, so I omitted that accuracy from my results above. Then, finally, I decided to combine the unigram and trigrams features, since unigrams are more prone to introducing high variance into the model, while bigrams are more prone to high bias, but that produced a worse accuracy as well.

I genuinely tried to be creative, but as you can see from the above, none of the features I implemented actually improved accuracy. The current implementation of my CustomFeature class is for the unigram + bigram model. Therefore, in the competition across students to determine who has the best accuracy, since my unigram NB model performed the best out of all my models, please use my unigram Naive Bayes model: "main.py –model NaiveBayes –feature unigram". Thanks.

■