John Corio
EECS 445
P1 Writeup

Part 2

Q (c)

The value of d, i.e. the number of unique words recorded from our training examples in *get_split_binary_data()* using class size 500, is **8483**. This was calculated using *len(dictionary_binary)*.

The average number of nonzero features per review was calculated using *sum(sum(X_train))/X_train.shape[0]*, and this yielded a value of **74.415.**

Part 3

Q 3.1 (b)

It is a good idea to maintain class proportions across folds so as not to create a fold with a data distribution noticeably different from the overall population. If we feed in one fold that has a large difference in class proportions, the performance levels, model fitting, and overall results of the classifier will have some bias introduced, creating a model that could generalize poorly.

Q 3.1 (d)

Performance measure	C	Performance
Accuracy	0.1	0.871999999
F1-Score	0.1	0.872151394
AUROC	0.01	0.939920000
Precision	0.1	0.873027713
Sensitivity	0.1	0.871999999
Specificity	0.01	0.884

As C varies, most examples peaked at 0.01 or 0.1, as we can see in the chart**.** After each measure peaks, based on terminal output it is seen that the performance measure

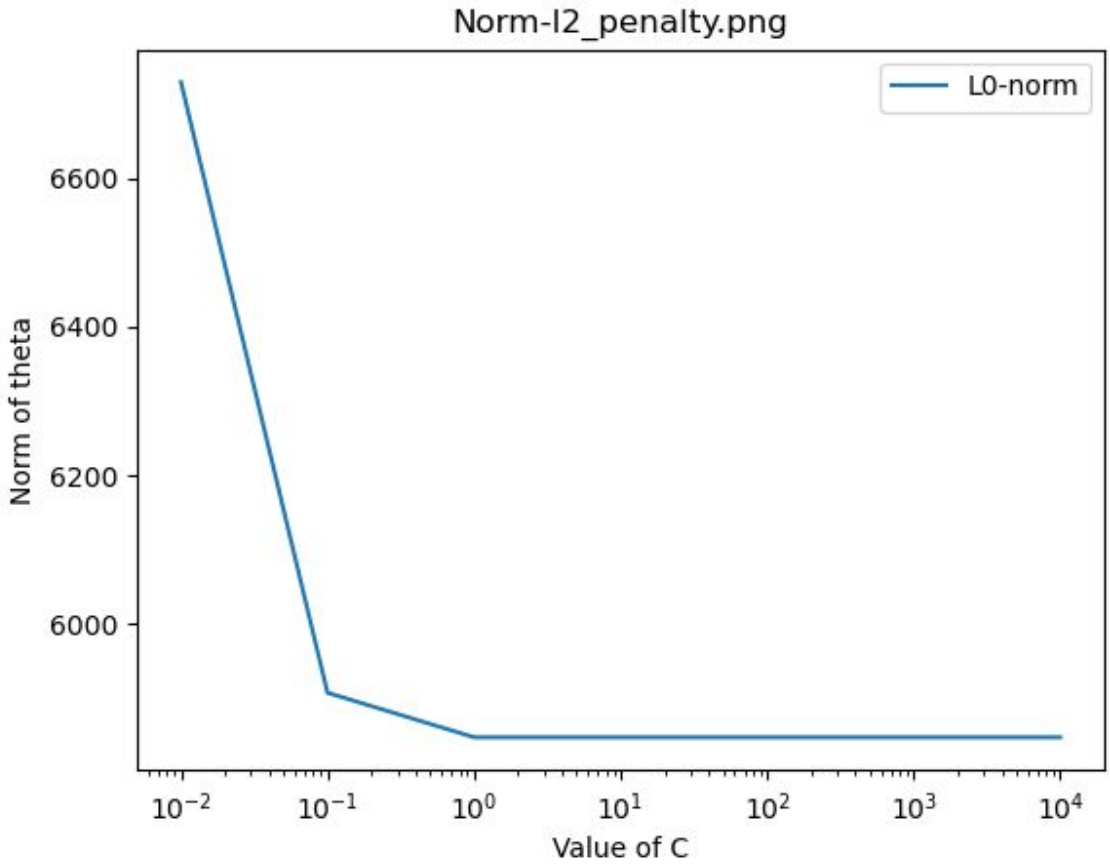
either levels out at this value, or starts a trajectory downwards towards slightly worse performance. If I had to choose a measure to optimize upon, it would most likely be accuracy, since this isn't a medical data scenario where false negatives are emphasized or something akin to this. In this case, raw accuracy is a good measure. Because of this, Q3.1 E will be using the C value of 0.1. However, it is also important to note that AUROC is, overall, a pretty good measure for performance in this set up as well. The closer to 1.00 you are, the closer to a perfect classifier you have produced. It displays the classifier's relative ability to discriminate between positive and negative reviews. This measure will be more useful, however, in question 4 where classes are imbalanced and a simple measure of accuracy falls apart simply because of the overwhelming number of examples in one class.

Q 3.1 (e)

Performance measure	Performance
Accuracy	0.806
F1-Score	0.7974947807933
AUROC	0.89433599999
Precision	0.8340611353711
Sensitivity	0.848
Specificity	0.764

['accuracy', 0.806]
['f1-score', 0.7974947807933194]
['auroc', 0.8943359999999999]
['precision', 0.834061135371179]
['sensitivity', 0.848]
['specificity', 0.764]

Q 3.1 (g)



Q 3.1 (h)

Positive Coefficient	Word
0.41740152559776594	definitely
0.4264157945565001	delicious
0.500723332804897	amazing
0.5273428235727176	excellent

Negative Coefficient	Word
-0.4511754287602237	not
-0.3224441819723553	worst
-0.27569270875905316	rude
-0.2538363977269925	cold

Positive words:
[['definitely'], 0.41740152559776594]
[['delicious'], 0.4264157945565001]
[['amazing'], 0.500723332804897]
[['excellent'], 0.5273428235727176]
Negative words:
[['not'], -0.4511754287602237]
[['worst'], -0.3224441819723553]
[['rude'], -0.27569270875905316]
[['cold'], -0.2538363977269925]

Q 3.1 (i)

NEGATIVE REVIEW WITH ¾ POSITIVE WORDS:

Don’t see what all the hype is about, this place is definitely overrated. It was amazing how bad the service was… Will not be coming back, despite (only!) a delicious appetizer.

Q 3.2 (b)

Grid Search:

C value: 0.01 ,r value: 0.01, score: 0.7266
C value: 0.01 ,r value: 0.1, score: 0.73636
C value: 0.01 ,r value: 1.0, score: 0.75816
C value: 0.01 ,r value: 10.0, score: 0.8916599999999999
C value: 0.01 ,r value: 100.0, score: 0.94442
C value: 0.01 ,r value: 1000.0, score: 0.93964
C value: 0.01 ,r value: 10000.0, score: 0.9397800000000001
C value: 0.1 ,r value: 0.01, score: 0.7635400000000001
C value: 0.1 ,r value: 0.1, score: 0.7912
C value: 0.1 ,r value: 1.0, score: 0.8994199999999999
C value: 0.1 ,r value: 10.0, score: 0.9439
C value: 0.1 ,r value: 100.0, score: 0.93964
C value: 0.1 ,r value: 1000.0, score: 0.93964
C value: 0.1 ,r value: 10000.0, score: 0.9397800000000001
C value: 1.0 ,r value: 0.01, score: 0.8604999999999998
C value: 1.0 ,r value: 0.1, score: 0.90412
C value: 1.0 ,r value: 1.0, score: 0.9399599999999999
C value: 1.0 ,r value: 10.0, score: 0.9398799999999999
C value: 1.0 ,r value: 100.0, score: 0.93964
C value: 1.0 ,r value: 1000.0, score: 0.93964
C value: 1.0 ,r value: 10000.0, score: 0.9397800000000001
C value: 10.0 ,r value: 0.01, score: 0.9182599999999999
C value: 10.0 ,r value: 0.1, score: 0.93216
C value: 10.0 ,r value: 1.0, score: 0.93986
C value: 10.0 ,r value: 10.0, score: 0.9398799999999999
C value: 10.0 ,r value: 100.0, score: 0.93964
C value: 10.0 ,r value: 1000.0, score: 0.93964
C value: 10.0 ,r value: 10000.0, score: 0.9397800000000001
C value: 100.0 ,r value: 0.01, score: 0.9229799999999999
C value: 100.0 ,r value: 0.1, score: 0.9326599999999999
C value: 100.0 ,r value: 1.0, score: 0.93986
C value: 100.0 ,r value: 10.0, score: 0.9398799999999999
C value: 100.0 ,r value: 100.0, score: 0.93964
C value: 100.0 ,r value: 1000.0, score: 0.93964
C value: 100.0 ,r value: 10000.0, score: 0.9397800000000001
C value: 1000.0 ,r value: 0.01, score: 0.9229799999999999
C value: 1000.0 ,r value: 0.1, score: 0.9326599999999999
C value: 1000.0 ,r value: 1.0, score: 0.93986
C value: 1000.0 ,r value: 10.0, score: 0.9398799999999999
C value: 1000.0 ,r value: 100.0, score: 0.93964
C value: 1000.0 ,r value: 1000.0, score: 0.93964
C value: 1000.0 ,r value: 10000.0, score: 0.9397800000000001
C value: 10000.0 ,r value: 0.01, score: 0.9229799999999999
C value: 10000.0 ,r value: 0.1, score: 0.9326599999999999
C value: 10000.0 ,r value: 1.0, score: 0.93986
C value: 10000.0 ,r value: 10.0, score: 0.9398799999999999
C value: 10000.0 ,r value: 100.0, score: 0.93964
C value: 10000.0 ,r value: 1000.0, score: 0.93964
C value: 10000.0 ,r value: 10000.0, score: 0.9397800000000001

Random Search

random search
C value: 2.1116451553729765 ,r value: 10.141888120327184, score: 0.9399000000000001
C value: 2.1116451553729765 ,r value: 0.11242726955349497, score: 0.9248000000000001
C value: 2.1116451553729765 ,r value: 45.21338526310731, score: 0.9397
C value: 2.1116451553729765 ,r value: 128.0129987266177, score: 0.9396599999999999
C value: 2.1116451553729765 ,r value: 146.14763280446485, score: 0.9396800000000001
C value: 0.26672327822763603 ,r value: 10.141888120327184, score: 0.9398200000000001
C value: 0.26672327822763603 ,r value: 0.11242726955349497, score: 0.8519400000000001
C value: 0.26672327822763603 ,r value: 45.21338526310731, score: 0.9397
C value: 0.26672327822763603 ,r value: 128.0129987266177, score: 0.9396599999999999
C value: 0.26672327822763603 ,r value: 146.14763280446485, score: 0.9396800000000001
C value: 8.194228899851808 ,r value: 10.141888120327184, score: 0.9399000000000001
C value: 8.194228899851808 ,r value: 0.11242726955349497, score: 0.9323
C value: 8.194228899851808 ,r value: 45.21338526310731, score: 0.9397
C value: 8.194228899851808 ,r value: 128.0129987266177, score: 0.9396599999999999
C value: 8.194228899851808 ,r value: 146.14763280446485, score: 0.9396800000000001
C value: 1.9640177884340162 ,r value: 10.141888120327184, score: 0.9399000000000001
C value: 1.9640177884340162 ,r value: 0.11242726955349497, score: 0.9239199999999999
C value: 1.9640177884340162 ,r value: 45.21338526310731, score: 0.9397
C value: 1.9640177884340162 ,r value: 128.0129987266177, score: 0.9396599999999999
C value: 1.9640177884340162 ,r value: 146.14763280446485, score: 0.9396800000000001
C value: 0.005782486017402186 ,r value: 10.141888120327184, score: 0.84548
C value: 0.005782486017402186 ,r value: 0.11242726955349497, score: 0.7373799999999999
C value: 0.005782486017402186 ,r value: 45.21338526310731, score: 0.9351799999999999
C value: 0.005782486017402186 ,r value: 128.0129987266177, score: 0.94558
C value: 0.005782486017402186 ,r value: 146.14763280446485, score: 0.94496
Grid C, Grid R, Grid performance:
0.01 100.0 0.94442
Random C, Random R, Random performance:
0.005782486017402186 128.0129987266177 0.94558

Tuning Scheme	C	r	AUROC
Grid Search	0.01	100	0.94442
Random Search	0.0057824860174	128.012998726617	0.94558

From terminal output, we can see that at first, unsurprisingly, the AUROC seems to grow as C and r increase overall from the smallest values provided. However, what is surprising is that as C grows in powers of ten, even just having r factored in to parametrize the model seems to increase its performance drastically. It doesn't seem to exhibit a drop in performance either as C and r increase as well, only leveling off and losing only the slightest bit of performance quality as C and r reach into their upper reaches of the grid search values. However, for each value of C, iterating through the choices for r shows that they *do* reach a maximum somewhere in the middle of the r grid, typically at a value of 1, 10 or 100.

One of the pros of grid search is that if you wish to use it as a sort of guideline, you can come up with a series of values that you can tune in between to narrow your search for optimal parameters. It also allows you to predefine and hone in your search area if you already have a good idea of where a hyperparameter's suitable value is. The cons, however, is that it can miss optimal values simply by being predefined. One of the pros of random search is exactly this, in fact. Random search allows you to produce values you wouldn't normally try otherwise, producing unexpected combinations of hyperparameters that could perform quite well compared to predefined search values. It also can be an effective way to try a large range of values using a shorter search time, which allows us to get a feel for approximately which order of magnitude our hyperparameters perform well or poorly at. A con of random search, however, is precisely that it is random. You have no guarantee if the 30 minutes you just dropped to train many models and measure their performance will yield you any results that could be deemed adequate or even be useful for determining more about your effective hyperparameter values.

Q 3.3 (a)

Expand the X vector into a vector containing both X and X^2, as well as terms involving each X component multiplied with another component in X.

Q 3.3 (b)

One of the pros of using an explicit feature mapping is that we have some form of interpretability for our feature coefficients and their relative importances that is not retained when utilizing kernels. The theta vector is easily interpreted in the original feature space as well as the higher-dimensional mapping. The obvious drawback of explicit feature maps is that it is incredibly inefficient in comparison, since we'd have to translate every single data point into a higher dimension and then learn in this larger feature space, which incurs the curse of dimensionality and adds orders of time to any sort of learning algorithm.

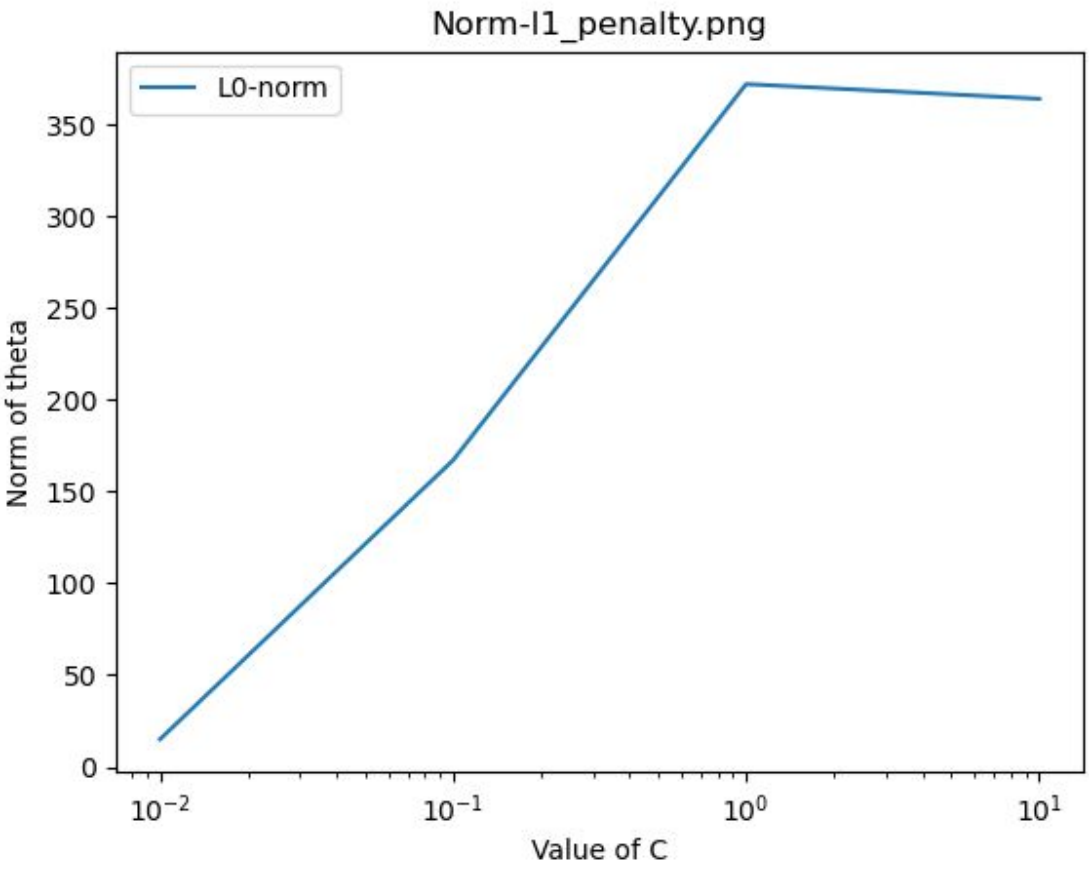
Q 3.4 (a)

Terminal Output:

L1 Penalty + Hinge Loss Parameter Selection (C, AUROC)
0.1, 0.92454

From this output, we get an optimal C value of 0.1, which produces a cross-validated AUROC score of 0.92454.

Q 3.4 (b)



Q 3.4 (c)

When we compare the two graphs, we most noticeably see the norm of theta is an order of magnitude smaller in the L1 penalty. This makes sense since optimizing the L1 penalty involves a gradient which doesn't have theta in it, allowing feature values to actually shrink to 0. However, the gradient of the L2 penalty retains an instance of theta, meaning that it can never truly shrink any of the components to 0. Another odd contrast is that it seems like the value of C has a more unpredictable effect on the norm of theta in the L1 case. It shows both an increase and a decrease as C increases, while the L2 penalty shows theta's norm only decreasing.

Using squared Hinge Loss also affects the solution by producing a more curved loss function. This should produce a more aggressive training speed towards the optimal solution, but in the end the optimal solution should stay the same as regular Hinge Loss.

Part 4

Q 4.1 (a)

Based on the weighted SVM formulation, we can expect each mistake in the negative class to be more heavily penalized. This is because it multiplies the sum of the slack vector for the negative class by 10, creating harsher penalties for misclassifications of the negative class and adjusting our decision boundary in turn to more aggressively minimize the slack vector of our negative classes. This should imply an emphasis on more examples classified correctly for the negative class, sacrificing some performance in the positive class.

Q 4.1 (c)

Terminal Output:

4.1 B
['accuracy', 0.722]
['f1-score', 0.6312997347480105]
['auroc', 0.8950400000000001]
['precision', 0.937007874015748]
['sensitivity', 0.968]
['specificity', 0.476]

Performance Measure	Performance
Accuracy	0.722
F1-Score	0.6312997347
AUROC	0.89504
Precision	0.93700787
Sensitivity	0.968
Specificity	0.476

Terminal Output from 3.1 (e):

['accuracy', 0.806]
['f1-score', 0.7974947807933194]
['auroc', 0.8943359999999999]

['precision', 0.834061135371179]
['sensitivity', 0.848]
['specificity', 0.764]

Q 4.1 (d)

It appears that accuracy, F1-score, and specificity were most heavily affected negatively. Accuracy decreased because of the adjustment to the negative points' individual weighting, simply because it emphasizes correctly classifying more negatives than overall accuracy across both classes. F1-score is a mean of sorts between precision and recall, which both measure a rate of positive identification, albeit with some slight differences. It's unsurprising that this is affected negatively by weight adjustments, as more positive examples are labelled as negative due to the weight shift. Specificity was the most negatively affected, which is quite bizarre as it measures the number of true negatives over the number of total negatives. However, this may be saying that this weight selection is an overadjustment, classifying a great deal of true positive points as negatives and thus massively affecting our specificity.

However, some measures exhibit signs of a more optimized performance, like precision and most obviously sensitivity. Sensitivity is a measure of the number of true positives divided by the number of all positive examples. This increase is probably implying that the number of false negatives is going down. This is unsurprising, as this class weight adjustment heavily punishes mistakes for the negative class, pushing it to classify more points as negative. Precision is akin to this, as it measures the number of true positives over all positive examples. Since we're more heavily penalizing wrongly classified negatives, logically it follows that this measure should increase as less points overall are classified negatively.

AUROC seems to have been relatively unaffected by this change in the class_weight parameter. Since it is a measure of the false positive rate versus the true positive rate, this is unsurprising, since you'd expect the false positive rate to go down due to the emphasis on penalizing mistakes in the negative class.

Q 4.2 (a)

Terminal Output:

4.2 A
['accuracy', 0.816]
['f1-score', 0.8963963963963965]
['auroc', 0.8488]
['precision', 0.8155737704918032]
['sensitivity', 0.1]
['specificity', 0.995]

Class Weights	Performance Measure	Performance
$W_n = 1, W_p = 1$	Accuracy	0.816
$W_n = 1, W_p = 1$	F1-Score	0.89639639
$W_n = 1, W_p = 1$	AUROC	0.8488
$W_n = 1, W_p = 1$	Precision	0.81557377
$W_n = 1, W_p = 1$	Sensitivity	0.1
$W_n = 1, W_p = 1$	Specificity	0.995

Q 4.2 (b)

Performance, as we can see, has taken a pretty significant hit compared to the CV results with balanced class data. Almost every single score has decreased by a noticeable score, most obviously sensitivity. This is implying that we are producing horrible rates for classifying true positives, producing a lot of false negative classifications. Specificity, however, seemed to jump noticeably upwards, which is implying that our rate of identifying true negatives has improved dramatically. Of course, this can easily be explained by the sudden shift in class proportions. By default, this makes it easy to just predict the majority class and still retain great performance within that class. The classifier's accuracy, AUROC, and precision have dropped a little bit, but still remain at a solid level of performance.

Q 4.3 (a)

Since our dataset now has imbalanced class proportions between negative and positive reviews, simply using accuracy won't produce a good classifier. It will just pick up on the overwhelming majority of one side and simply just introduce bias towards that class, producing a seemingly great test accuracy, but overall poor generalization. This is because it doesn't actually learn anything about the less common class. For this reason, looking at some measure involving true positives or negatives in comparison with false positives or negatives would yield a more effective classifier that generalizes well to this class imbalance. Adjusting weight parameters, however, can help overcome this imbalance by heavily penalizing mistakes in classifying the class proportion with less

examples. This artificially adjusts the class proportions and makes accuracy slightly better.

I'd create weight parameters based on investigation of the proportion of negative to positive classes (which revealed positive_class_size = 800, and ratio = 0.25), as well as looking over a base level of performance of the examples produced beforehand on imbalanced class data. Since positive is the majority class from this look into helper.py, I will leave positive as is in order to create a control value of sorts.

As for the evaluation metric to optimize the CV performance for, I decided on the f1-score. Since we see there is a large imbalance towards the positive class, and we have no real idea whether false positives or false negatives matter more than the other, the f1-score will function as a balancing point between the two and indicate how well we're doing across the two measures.

Training Results
[0.25, 0.8]
[2, 0.8467340103983734]
[4, 0.9019431943000267]
[6, 0.9237046775017417]
[8, 0.932029532967033]
[10, 0.9396263207305886]

Q 4.3 (b)

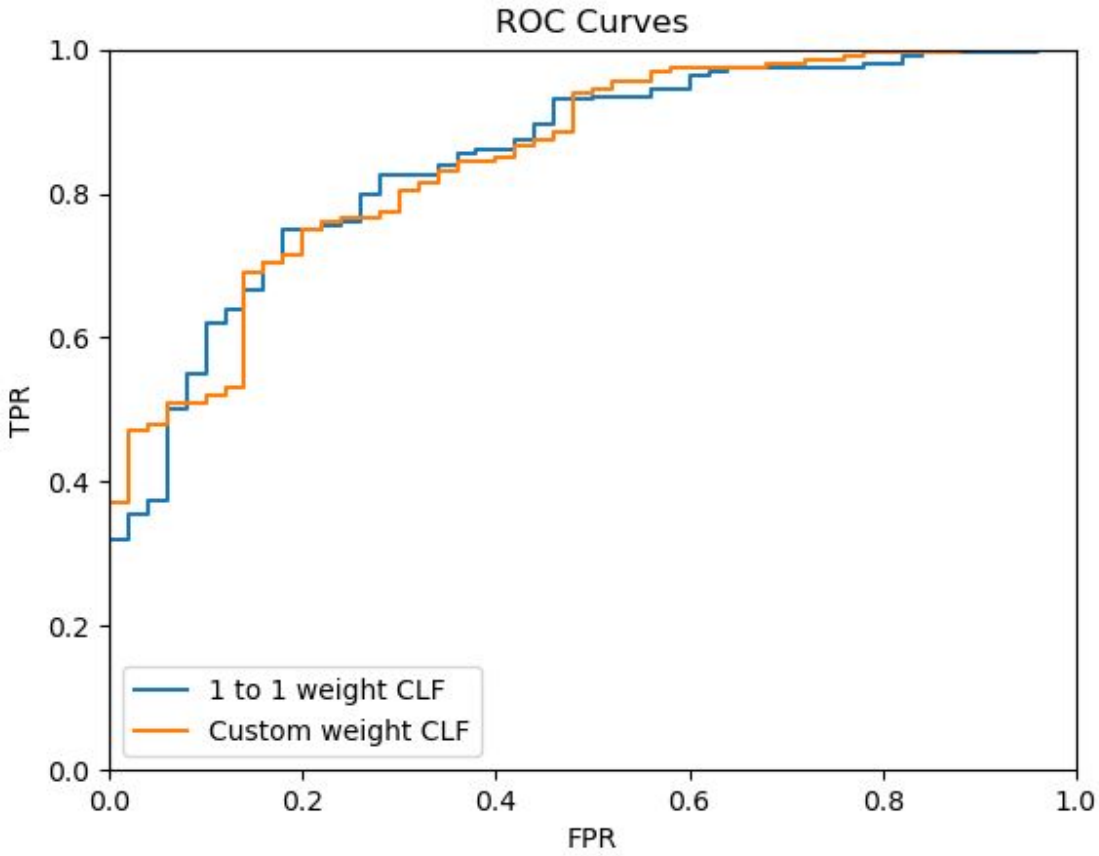
Terminal Output

F1-SCORE CV
[0.25, 0.888888888888889]
[2, 0.908449142495303]
[4, 0.9012619847186137]
[6, 0.8633165776302224]
[8, 0.8414938017044971]
[10, 0.8343162169841671]

WN = 2 TEST RESULTS
['accuracy', 0.86]
['f1-score', 0.9172576832151299]
['auroc', 0.8493]
['precision', 0.8699551569506726]
['sensitivity', 0.42]
['specificity', 0.97]

Weight Parameters	Perf. Measures	Performance
$W_n = 2, W_p = 1$	Accuracy	0.86
$W_n = 2, W_p = 1$	F1-Score	0.91725768
$W_n = 2, W_p = 1$	AUROC	0.8493
$W_n = 2, W_p = 1$	Precision	0.86995515
$W_n = 2, W_p = 1$	Sensitivity	0.42
$W_n = 2, W_p = 1$	Specificity	0.97

Q 4.4



PART 5

Q1

Feature Engineering

My first thought was to utilize a “fluff” remover for the dataset, i.e. remove words that offer little to no information on the overall tone and valence of the message left in the review. I utilized the NLTK stopwords attribute inside of my redefined *extract_dictionary* function.

My second idea was to track word frequency. This, in conjunction with removing stop words, makes a lot of sense to me since most stopwords are some of the most frequent words used. With the most frequent and uninformative words removed, tracking word counts should theoretically have more impact than simply indicating their presence.

Of course, this idea necessitated edits of *generate_feature_matrix*. As such, both functions were redefined with an ending tag *part_5* to denote that they handle this. Furthermore, any functions in the helper.py file pertaining to the datasets, particularly

get_multiclass_training_data and *get_heldout_reviews,* were edited to reflect this change in data representation.

Related to this idea of tooling around with the feature vectors, I decided against normalizing the data, despite it usually having a large effect on SVM classification. When I considered the nature of the data, normalizing it seems a bit counterintuitive. If we have a really positive review, we'd expect a ton of instances of words like "great," "excellent," "delicious," etc., so normalizing data and shifting probability mass into words that wouldn't appear otherwise such as "horrible," "disgusting," or "nasty" makes no sense. Plus, frequencies in general are measured in integer counts, and there is no logical meaning to a word appearing 1.4 times in a review.

However, after seeing some (possibly fishy) performance values, I decided to try a run with normalization to measure its effects on cross-validation scores on the best classifier I produced so far. For this, I imported a scaler library from scikit-learn. After a single run of normalization on the best classifier produced thus far, a 20% drop in accuracy told me all that I needed to know about utilizing normalization for this idea.

Hyperparameter Selection

Our only hyperparameter of interest for this section was C, as in previous runs in this project the value of r did not seem to have a huge effect. Hyperparameter selection was done in a similar manner to the rest of this project. A 5-fold cross-validation model was set up using random selection of data points. Since class imbalance was not present, accuracy was used as the metric of choice to select the values of our hyperparameter.

Specifically, values of C found in the project earlier were used with an L2 penalty. The values of C were chosen because they represent a wide range of possible values that I thought could all reveal something interesting about the nature of our features. It made no sense to me to use an L1 penalty because, theoretically, all of our columns should be relevant with the removal of stop words. An L1-penalty shrinks columns to 0 while an L2-penalty doesn't actually set them to 0, so that's why L2 remained appealing for me.

The optimal hyperparameter variable was chosen by measuring accuracy on a holdout set created within our training data. This was done in order to produce an approximation to the testing error we'd produce if the model's actual test set was given to us.
After selecting an algorithm (via the methods described below), more values of C were tried out.

Algorithm Selection

To test the performance of various kernels, a framework similar to the selection of our hyperparameter C was created using linear, polynomial, and RBF kernels.

I decided to select C *while* selecting a kernel to create a grid search type of approach, since if the kernel is going to vary along with C, it makes no logical sense to arbitrarily fit kernels to a singular C value then suddenly vary the C value right afterwards as well.

Based on the terminal output below, the quadratic kernel seemed like it'd be quite a poor choice to try and delve further into, and the linear kernel did quite well off an initial C-value of 0.01, so I decided to try and optimize based on that. The RBF kernel seemed to do well, but the linear kernel clearly did the best by a few percent, which indicated to me the extra model flexibility didn't actually help much. As such, the linear kernel was chosen to examine further. I didn't get to test much further, however, simply because of the sheer amount of time this cross-validation approach took to train.

Multiclass Method

For this option, I decided to use the SVC default setting, which is one-vs-one. Obviously, this fits a lot more individual models than one-vs-rest, but since I don't have an idea of when this time complexity becomes a bit unreasonable as a function of data size, I saw no reason to change it. I was also hoping that since it's only three classes it'd help to have direct, probabilistic comparisons between individual classes instead of one versus all the rest.

Extra Techniques

I debated coming up with some sort of different way to represent our feature counts, but in the end the training time for the initial test run of various C values and kernels took much longer than I had anticipated, so I had to give it up.

I also noted the numbers and commas at the end of each review noting columns for funny, cool, and helpful. I didn't end up using these, but if I had to consider it, I'd only use the helpful indicator.

INITIAL RUN TERMINAL OUTPUT:

```
PART 5
LINEAR CLASSIFIER CV PERFORMANCE
linear, 0.687111111111111, 0.01
linear, 0.6537777777777778, 0.1
linear, 0.6346666666666666, 1.0
linear, 0.632, 10.0
linear, 0.632, 100.0
linear, 0.632, 1000.0
```

linear, 0.632, 10000.0
QUADRATIC CLASSIFIER CV PERFORMANCE
degree 2 polynomial, 0.3342222222222225, 0.01
degree 2 polynomial, 0.3577777777777775, 0.1
degree 2 polynomial, 0.4564444444444445, 1.0
degree 2 polynomial, 0.5542222222222222, 10.0
degree 2 polynomial, 0.6146666666666667, 100.0
degree 2 polynomial, 0.5973333333333334, 1000.0
degree 2 polynomial, 0.5542222222222223, 10000.0
RBF CLASSIFIER CV PERFORMANCE
rbf, 0.4488888888888889, 0.01
rbf, 0.47733333333333333, 0.1
rbf, 0.6648888888888889, 1.0
rbf, 0.6608888888888889, 10.0
rbf, 0.6488888888888888, 100.0
rbf, 0.6488888888888888, 1000.0
rbf, 0.6488888888888888, 10000.0

In the end, this first training run took an obscene amount of time and taught me that maybe I should look into other ways of evaluating the model and using cross validation for our scores. Perhaps involve less folds or bring in other measures to get more information from the time spent simply training these models. The way I chose to present the training data probably had an effect as well. However, with the information I did get from the models, it seemed like producing a multi-class problem creates a lot more error, almost a 30% difference in the best-performing cases in the binary problem. Whether something is wrong with my model fundamentally or something else is wrong is a matter I could not investigate properly simply due to time constraints and school being school, however this number seems quite a bit fishy to me.

NORMALIZATION RUN TERMINAL OUTPUT:

Linear Test: 0.01, 0.48133333333333334

As I suspected, normalization didn't help all that much. In the end, it drastically hindered the performance of our best model. After this, I decided with the time I had left to try a few more values close to 0.01 to determine if the performance could be pushed upwards somehow.

FINAL CV RUNS TERMINAL OUTPUT:

LINEAR CLASSIFIER CV PERFORMANCE
Linear Test: 0.015, 0.6888888888888889
Linear Test: 0.001, 0.4728888888888889
Linear Test: 0.005, 0.6551111111111111
Linear Test: 0.02, 0.6831111111111111
Linear Test: 0.025, 0.6768888888888889

After some more test runs highlighted above, it appeared that the best I could get was a linear kernel with a C value of 0.015. As such, I ran this model to make my predictions for number 2.

Q3

Problem Statement

My problem would be to try and create an in-game chat log detector for harassment, spam, or otherwise poor player behavior to help try and ban toxic players. Toxic players make in-game experiences for other players less fun by verbally harassing other players, intentionally giving the other team advantages, or generally just making it harder for their teammates to enjoy the game.

Dataset

I'd love to use chat logs from popular online team-based games, such as League of Legends and DotA. I'd also love to have some access to players' performance levels during the specific games their chat logs come from, as this could help tell whether players online are being sarcastically friendly or sweet or if they are genuinely positive.

I'd maybe extend out into chat logs for even single-player online games such as Runescape or World of Warcraft simply to have more data for the ways players online convey emotion through text.

NLP Techniques

One technique that I'd make use of is certainly this principle or removing stop words and punctuation. This allows us to hone in on words that convey information about the emotional valence behind a message.

Another technique I'd make use of is one that I didn't get to make use of in this project: multi-word phrases. In online games, there are certain words or phrases that have meaning in game but have no real usage outside of that sphere. For example, *running it down* is a common term in League of Legends for throwing away your advantage, and thus the game, to the enemy team. Another example is ___ *diff*, which essentially means one team's player in position ___ played noticeably better than the other player in position ___.

Additional Considerations

Of course, with the concept of tightening up on player bans come some very real risks. On the one hand, online games are hard to police and anonymous. When something

competitive is on the line, it makes it much more likely that players will get heated and say something that crosses a line. The more this happens, the less fun the typical player will have, so properly policing player conduct is imperative to an enjoyable experience.

However, one cannot simply ban a player off of the decision of an ML algorithm that reads chat messages, excluding some words that very obviously have no place being said. Banning players who simply say one negatively-charged remark would undoubtedly ruin their experience in-game as well. You can counteract this by optimizing your model to stray away from false positives or by making it track repeat offenses by user ID or IP address, but in the end this only can do so much. Even tracking repeat offenses can introduce some algorithmic bias against players who possibly are trying to play more positively.

APPENDIX (CODE):

```
def extract_dictionary(df):
    """
    Reads a pandas dataframe, and returns a dictionary of distinct words
    mapping from each distinct word to its index (ordered by when it was found).
    Input:
        df: dataframe/output of load_data()
    Returns:
        a dictionary of distinct words that maps each distinct word
        to a unique index corresponding to when it was first found while
        iterating over all words in each review in the dataframe df
    """
    word_dict = {}
    # punctuation constant
    for char in string.punctuation:
        df['reviewText'] = df['reviewText'].apply(lambda x: x.replace(char, " "))

    # index counter
    i = 0

    # apply vectorized lowercase
    df['reviewText'] = df['reviewText'].str.lower()

    # for each msg in content
    for msg in df['reviewText']:
        # split into separate words
        contents = msg.split()
        # iterate thru split msg
        for word in contents:
            # check if current word is already found
            if word not in word_dict:
                # if not, insert it w/ key = word, val = index
                word_dict[word] = i
                i = i + 1
    # return dict
    return word_dict
```

```python
def generate_feature_matrix(df, word_dict):
    """
    Reads a dataframe and the dictionary of unique words
    to generate a matrix of {1, 0} feature vectors for each review.
    Use the word_dict to find the correct index to set to 1 for each place
    in the feature vector. The resulting feature matrix should be of
    dimension (# of reviews, # of words in dictionary).
    Input:
        df: dataframe that has the ratings and labels
        word_dict: dictionary of words mapping to indices
    Returns:
        a feature matrix of dimension (# of reviews, # of words in dictionary)
    """
    number_of_reviews = df.shape[0]
    number_of_words = len(word_dict)
    feature_matrix = np.zeros((number_of_reviews, number_of_words))
    # go thru each review
    for i in range(0, number_of_reviews):
        # store text in a list
        msg = (df.iloc[i]['reviewText']).split()
        # go thru words in msg and mark their respective indices w a 1
        for word in msg:
            if word in word_dict:
                # word_dict[word] gives index to place 1
                feature_matrix[i][word_dict[word]] = 1
    return feature_matrix
```

def cv_performance(clf, X, y, k = 5, metric = ‘accuracy’)

```python
    # TODO: Implement this function
    #HINT: You may find the StratifiedKFold from sklearn.model_selection
    #to be useful

    #Put the performance of the model on each fold in the scores array
    scores = []
    # get scoring method
    method = metric
    # do StratifiedKFold stuff
    skf = StratifiedKFold(n_splits = k)

    # fit an SVC on each training fold, then get performance across testing folds and put into scores vector
    for trainindices, testindices in skf.split(X, y):
        clf.fit(X = X[trainindices], y = y[trainindices])
        pred = clf.predict(X[testindices])
        score = 0
        if method == 'auroc':
            # decision boundary
            test = clf.decision_function(X[testindices])
            score = metrics.roc_auc_score(y_true = y[testindices], y_score = test)
        elif method == 'accuracy':
            score = clf.score(X = X[testindices], y = y[testindices])
        elif method == 'f1-score':
            score = metrics.f1_score(y_true = y[testindices], y_pred = pred)
        elif method == 'precision':
            score = metrics.precision_score(y_true = y[testindices], y_pred = pred)
        elif method == 'sensitivity':
            # Sensitivity = (True Positive)/(True Positive + False Negative)
            tn, fp, fn, tp = metrics.confusion_matrix(y[testindices], pred, labels = [1,-1]).ravel()
            score = tp/(tp + fn)
        elif method == 'specificity':
            # Specificity = (True Negative)/(True Negative + False Positive)
            tn, fp, fn, tp = metrics.confusion_matrix(y[testindices], pred, labels = [1,-1]).ravel()
            score = tn/(tn + fp)
        scores.append(score)
    #And return the average performance across all fold splits.
    return np.array(scores).mean()
```

```python
def select_param_linear(X, y, k=5, metric="accuracy", C_range = [], penalty='l2'):
    """
    Sweeps different settings for the hyperparameter of a linear-kernel SVM,
    calculating the k-fold CV performance for each setting on X, y.
    Input:
        X: (n,d) array of feature vectors, where n is the number of examples
        and d is the number of features
        y: (n,) array of binary labels {1,-1}
        k: int specifying the number of folds (default=5)
        metric: string specifying the performance metric (default='accuracy',
             other options: 'f1-score', 'auroc', 'precision', 'sensitivity',
             and 'specificity')
        C_range: an array with C values to be searched over
    Returns:
        The parameter value for a linear-kernel SVM that maximizes the
        average 5-fold CV performance.
    """
    best_C_val = C_range[0]
    best_metric = 0
    # TODO: Implement this function
    #HINT: You should be using your cv_performance function here
    #to evaluate the performance of each SVM
    # loop over C_range, fit a clf, pass into cv_performance, get best_C_val from results
    for i in C_range:
        clf = SVC(kernel = 'linear', C = i, class_weight = "balanced")
        score = cv_performance(clf = clf, X = X, y = y, k=k, metric=metric)
        print("C value: " + str(i) + ", score: " + str(score))
        if score > best_metric:
            best_C_val = i
            best_metric = score
    # SHOULD I BE RETURNING THE PERFORMANCE HERE AS WELL?
    return best_C_val, best_metric
```

```
def plot_weight(X,y,penalty,C_range):
    """
    Takes as input the training data X and labels y and plots the L0-norm
    (number of nonzero elements) of the coefficients learned by a classifier
    as a function of the C-values of the classifier.
    """

    print("Plotting the number of nonzero entries of the parameter vector as a function of C")
    norm0 = []

    # TODO: Implement this part of the function
    #Here, for each value of c in C_range, you should
    #append to norm0 the L0-norm of the theta vector that is learned
    #when fitting an L2- or L1-penalty, degree=1 SVM to the data (X, y)
    if penalty == 'l2':
        for c in C_range:
            clf = SVC(kernel = 'linear', C = c, class_weight = "balanced").fit(X,y)
            theta = clf.coef_
            norm = np.linalg.norm(theta[0], 0)
            norm0.append(norm)
    elif penalty == 'l1':
        for c in C_range:
            clf = LinearSVC(penalty = 'l1', dual = False, C = c, class_weight='balanced').fit(X,y)
            theta = clf.coef_
            norm = np.linalg.norm(theta[0], 0)
            norm0.append(norm)

    #This code will plot your L0-norm as a function of c
    plt.plot(C_range, norm0)
    plt.xscale('log')
    plt.legend(['L0-norm'])
    plt.xlabel("Value of C")
    plt.ylabel("Norm of theta")
    plt.title('Norm-'+penalty+'_penalty.png')
    plt.savefig('Norm-'+penalty+'_penalty.png')
    plt.close()
```

def select_param_quadratic(X, y, k = 5, measure = ‘accuracy’, param_range = [])

```
        Returns:
            The parameter values for a quadratic-kernel SVM that maximize
            the average 5-fold CV performance as a pair (C,r)
    """
    best_C_val,best_r_val = 0.0, 0.0
    # TODO: Implement this function
    # Hint: This will be very similar to select_param_linear, except
    # the type of SVM model you are using will be different...
    # loop over C_range, fit a clf, pass into cv_performance, get best_C_val from results

    best_metric = 0
    # loop over C_range
    for i in param_range[0]:
        # loop over r_range
        for j in param_range[1]:
            # fit polynomial degree 2 SVM and perform CV score
            clf = SVC(kernel = 'poly', degree = 2, C = i, coef0 = j, class_weight = "balanced")
            score = cv_performance(clf = clf, X = X, y = y, k=k, metric=metric)
            # score check
            print("C value: " + str(i) + ", r value: " + str(j) + ", score: " + str(score))
            if score > best_metric:
                best_C_val = i
                best_r_val = j
                best_metric = score
    # return our tuple
    return best_C_val,best_r_val,best_metric
```

ALL OF THE FOLLOWING SCREENSHOTS ARE OF MY MAIN FUNCTION!!!!

```
def main():
    # Read binary data
    # NOTE: READING IN THE DATA WILL NOT WORK UNTIL YOU HAVE FINISHED
    #       IMPLEMENTING generate_feature_matrix AND extract_dictionary
    X_train, Y_train, X_test, Y_test, dictionary_binary = get_split_binary_data()
    IMB_features, IMB_labels = get_imbalanced_data(dictionary_binary)
    IMB_test_features, IMB_test_labels = get_imbalanced_test(dictionary_binary)

    # TODO: Questions 2, 3, 4

    ########## 2 C, i and ii ##########
    print("The length of the dictionary is: {} \n".format(len(dictionary_binary)))
    print("The average # of non-zero features in the training set is: {} \n".format(sum(sum(X_train))/X_train.shape[0]))

    ########## 3.1 ##########
    #'''
    # 3.1 D
    # constants
    print("3.1 D")
    c_vals = [10e-3, 10e-2, 10e-1, 10e0, 10e1, 10e2, 10e3]#(i.e.10-3,10-2,...,102,103).
    measures = ["accuracy", "f1-score", "auroc", "precision", "sensitivity", "specificity"]
    chosendata = []

    # iterate thru choices
    for meas in measures:
        chosen_c, performance = select_param_linear(X = X_train, y = Y_train, k=5, metric=meas, C_range = c_vals, penalty='l2')
        group = [meas, chosen_c, performance]
        chosendata.append(group)

    # output for the writeup 3.1 D
    for tup in chosendata:
        print(tup)
```

```
    # 3.1 E
    # from our table we choose accuracy to optimize
    print("3.1 E")
    chosenC = 0.1
    performance_measures = []
    # fit classifier
    part3E = SVC(kernel = 'linear', C = chosenC, class_weight = "balanced").fit(X = X_train, y = Y_train)

    # get prediction vector
    pred = part3E.predict(X_test)
    # go thru each measure
    for meas in measures:
        performance = 0
        if meas == 'auroc':
            # decision boundary
            test = part3E.decision_function(X_test)
            performance = metrics.roc_auc_score(y_true = Y_test, y_score = test)
        elif meas == 'accuracy':
            performance = part3E.score(X = X_test, y = Y_test)
        elif meas == 'f1-score':
            performance = metrics.f1_score(y_true = Y_test, y_pred = pred)
        elif meas == 'precision':
            performance = metrics.precision_score(y_true = Y_test, y_pred = pred)
        elif meas == 'sensitivity':
            # Sensitivity = (True Positive)/(True Positive + False Negative)
            tn, fp, fn, tp = metrics.confusion_matrix(Y_test, pred, labels = [1,-1]).ravel()
            performance = tp/(tp + fn)
        elif meas == 'specificity':
            # Specificity = (True Negative)/(True Negative + False Positive)
            tn, fp, fn, tp = metrics.confusion_matrix(Y_test, pred, labels = [1,-1]).ravel()
            performance = tn/(tn + fp)
        group = [meas, performance]
        performance_measures.append(group)

    # for writeup
    for tup in performance_measures:
        print(tup)
```

```
# 3.1 F and G
plot_weight(X = X_train, y = Y_train, penalty = "l2", C_range = c_vals)

# 3.1 H & I
print("3.1 H & I")
# fit and get coefficients
theta = part3E.coef_

# positive words
positive_ind = np.argsort(a = theta[0])[-5:-1]
keys, values = zip(*dictionary_binary.items())
positive_words = []
for i in positive_ind:
    word = [k for k,v in dictionary_binary.items() if v == i]
    positive_words.append([word,theta[0,i]])

# negative words
negative_ind = np.argsort(a = theta[0])[0:4]
negative_words = []
for i in negative_ind:
    word = [k for k,v in dictionary_binary.items() if v == i]
    negative_words.append([word,theta[0,i]])

# output
print("Positive words: ")
for i in positive_words:
    print(i)

print("Negative words: ")
for i in negative_words:
    print(i)
```

```
    ########## 3.2 ##########
    # 3.2 A & B

    print("3.2 B")
    # grid search
    print("grid search")
    range_r = c_vals
    params = np.array([c_vals, range_r])
    grid_c, grid_r, performancegrid = select_param_quadratic(X = X_train, y = Y_train, k=5, metric = 'auroc', param_range =
params)

    # random search
    print("random search")
    rand_r = [pow(10,r) for r in np.random.uniform(low = -3, high = 3, size = 5)]
    rand_c = [pow(10,c) for c in np.random.uniform(low = -3, high = 3, size = 5)]
    params = np.array([rand_c, rand_r])
    rand_c, rand_r, performancerand = select_param_quadratic(X = X_train, y = Y_train, k=5, metric = 'auroc', param_range =
params)

    # output
    print("Grid C, Grid R, Grid performance: ")
    print(str(grid_c) + " " + str(grid_r) + " " + str(performancegrid))
    print("Random C, Random R, Random performance: ")
    print(str(rand_c) + " " + str(rand_r) + " " + str(performancerand))

    ########## 3.4 ##########
    # 3.4 A
    print("3.4 A")
    c_vals = [10e-3, 10e-2, 10e-1, 10e0]
    best_c = 0
    best_auroc = 0
    for c in c_vals:
        clf = LinearSVC(penalty = 'l1', dual = False, C = c, class_weight='balanced')
        score = cv_performance(clf, X = X_train, y = Y_train, k=5, metric="auroc")
        if score > best_auroc:
            best_auroc = score
            best_c = c

    # output
    print("L1 Penalty + Hinge Loss Parameter Selection (C, AUROC)")
    print(str(best_c) + ", " +  str(best_auroc))
```

```
# 3.4 B
print("3.4 B")
# need to pass this a linear SVC
plot_weight(X = X_train, y = Y_train, penalty = 'l1', C_range = c_vals)

########## 4.1 ##########
# 4.1 B #
# fit SVC
print("4.1 B")
# SVC minimizes the regular hinge loss w/ L2 penalty, so all we need is to set class weights and C value
unbclf = SVC(C = 0.01, kernel = "linear", class_weight = {-1: 10, 1: 1}).fit(X_train, Y_train)
# list of performances
B41performance = []
# get prediction vector
pred = unbclf.predict(X_test)
# go thru each measure
for meas in measures:
    performance = 0
    if meas == 'auroc':
        # decision boundary
        test = unbclf.decision_function(X_test)
        performance = metrics.roc_auc_score(y_true = Y_test, y_score = test)
    elif meas == 'accuracy':
        performance = unbclf.score(X = X_test, y = Y_test)
    elif meas == 'f1-score':
        performance = metrics.f1_score(y_true = Y_test, y_pred = pred)
    elif meas == 'precision':
        performance = metrics.precision_score(y_true = Y_test, y_pred = pred)
    elif meas == 'sensitivity':
        # Sensitivity = (True Positive)/(True Positive + False Negative)
        tn, fp, fn, tp = metrics.confusion_matrix(Y_test, pred, labels = [1,-1]).ravel()
        performance = tp/(tp + fn)
    elif meas == 'specificity':
        # Specificity = (True Negative)/(True Negative + False Positive)
        tn, fp, fn, tp = metrics.confusion_matrix(Y_test, pred, labels = [1,-1]).ravel()
        performance = tn/(tn + fp)
    group = [meas, performance]
    B41performance.append(group)
# output
for group in B41performance:
    print(group)
```

```
# 4.2 A #
print("4.2 A")
# fit classifier
clfA42 = SVC(C = 0.01, kernel = "linear", class_weight = {-1: 1, 1: 1}).fit(IMB_features, IMB_labels)
# list of performances
B42performance = []
# get prediction vector
pred = clfA42.predict(IMB_test_features)
# go thru each measure
for meas in measures:
    performance = 0
    if meas == 'auroc':
        # decision boundary
        test = clfA42.decision_function(IMB_test_features)
        performance = metrics.roc_auc_score(y_true = IMB_test_labels, y_score = test)
    elif meas == 'accuracy':
        performance = clfA42.score(X = IMB_test_features, y = IMB_test_labels)
    elif meas == 'f1-score':
        performance = metrics.f1_score(y_true = IMB_test_labels, y_pred = pred)
    elif meas == 'precision':
        performance = metrics.precision_score(y_true = IMB_test_labels, y_pred = pred)
    elif meas == 'sensitivity':
        # Sensitivity = (True Positive)/(True Positive + False Negative)
        tn, fp, fn, tp = metrics.confusion_matrix(IMB_test_labels, pred, labels = [1,-1]).ravel()
        performance = tp/(tp + fn)
    elif meas == 'specificity':
        # Specificity = (True Negative)/(True Negative + False Positive)
        tn, fp, fn, tp = metrics.confusion_matrix(IMB_test_labels, pred, labels = [1,-1]).ravel()
        performance = tn/(tn + fp)
    group = [meas, performance]
    B42performance.append(group)
# output
for group in B42performance:
    print(group)
```

```
# 4.3 #
print("4.3 B")
# investigating the helper.py reveals positive_class_size = 800, ratio = 0.25
# based on this, I will try various Wn starting with the inverse of this ratio
negative_weight_list = [0.25, 2, 4, 6, 8, 10]
# fit classifiers based on this weight and choose best cross_validation value
scores = []
imbscore = 0
imbweight = 0
for weight in negative_weight_list:
    clf = SVC(C = 0.01, kernel = "linear", class_weight = {-1: weight, 1: 1})
    score = cv_performance(clf, X = IMB_features, y = IMB_labels, k=5, metric="f1-score")
    scores.append([weight, score])
    if score > imbscore:
        imbscore = score
        imbweight = weight

# output
for group in scores:
    print(group)

# then we produce our testing data performance
clf43 = SVC(C = 0.01, kernel = "linear", class_weight = {-1: 2, 1: 1}).fit(IMB_features, IMB_labels)
# list of performances
B43performance = []
# get prediction vector
pred = clf43.predict(IMB_test_features)
# go thru each measure
for meas in measures:
    performance = 0
    if meas == 'auroc':
        # decision boundary
        test = clf43.decision_function(IMB_test_features)
        performance = metrics.roc_auc_score(y_true = IMB_test_labels, y_score = test)
    elif meas == 'accuracy':
        performance = clf43.score(X = IMB_test_features, y = IMB_test_labels)
    elif meas == 'f1-score':
        performance = metrics.f1_score(y_true = IMB_test_labels, y_pred = pred)
    elif meas == 'precision':
        performance = metrics.precision_score(y_true = IMB_test_labels, y_pred = pred)
    elif meas == 'sensitivity':
        [illegible]
    elif meas == 'sensitivity':
        # Sensitivity = (True Positive)/(True Positive + False Negative)
        tn, fp, fn, tp = metrics.confusion_matrix(IMB_test_labels, pred, labels = [1,-1]).ravel()
        performance = tp/(tp + fn)
    elif meas == 'specificity':
        # Specificity = (True Negative)/(True Negative + False Positive)
        tn, fp, fn, tp = metrics.confusion_matrix(IMB_test_labels, pred, labels = [1,-1]).ravel()
        performance = tn/(tn + fp)
    group = [meas, performance]
    B43performance.append(group)
# output
for group in B43performance:
    print(group)
```

```
# 4.4 #
print("4.4")
# decision score w clfA42 (Wp = 1, Wn = 1)
IMB_score42 = clfA42.decision_function(IMB_test_features)
fpr42, tpr42, _ = metrics.roc_curve(IMB_test_labels, IMB_score42)

# predict w custom weights (Wp = 1, Wn = 2)
IMB_score43 = clf43.decision_function(IMB_test_features)
fpr43, tpr43, _ = metrics.roc_curve(IMB_test_labels, IMB_score43)

# plot on same graph & save image
plt.plot(fpr42, tpr42)
plt.plot(fpr43, tpr43)
plt.title(label = "ROC Curves")
plt.xlim([0, 1])
plt.ylim([0, 1])
plt.legend(['1 to 1 weight CLF', 'Custom weight CLF'])
plt.xlabel("FPR")
plt.ylabel("TPR")
plt.savefig('Q4 Weight-Based ROC Curves.png')
plt.close()
#'''
# Read multiclass data
# TODO: Question 5: Apply a classifier to heldout features, and then use
#       generate_challenge_labels to print the predicted labels
multiclass_features, multiclass_labels, multiclass_dictionary = get_multiclass_training_data()
heldout_features = get_heldout_reviews(multiclass_dictionary)
```

CHALLENGE CODE HERE INCLUDING EDITED FUNCTIONS AND MAIN PORTION

Added functions

```
################################## PART 5 FUNCTIONS ##########################################
def extract_dictionary_part5(df):
    """
    Reads a pandas dataframe, and returns a dictionary of distinct words
    mapping from each distinct word to its index (ordered by when it was found).
    Input:
        df: dataframe/output of load_data()
    Returns:
        a dictionary of distinct words that maps each distinct word
        to a unique index corresponding to when it was first found while
        iterating over all words in each review in the dataframe df
    """
    word_dict = {}
    # part 5 edit
    stop_words = set(stopwords.words('english'))
    # punctuation constant
    for char in string.punctuation:
        df['reviewText'] = df['reviewText'].apply(lambda x: x.replace(char, " "))

    # index counter
    i = 0

    # apply vectorized lowercase
    df['reviewText'] = df['reviewText'].str.lower()

    # for each msg in content
    for msg in df['reviewText']:
        # split into separate words
        contents = [w for w in msg.split() if w not in stop_words]
        # iterate thru split msg
        for word in contents:
            # check if current word is already found
            if word not in word_dict:
                # if not, insert it w/ key = word, val = index
                word_dict[word] = i
                i = i + 1
    # return dict
    return word_dict


################################## PART 5 FUNCTIONS ##########################################
################################## PART 5 FUNCTIONS ##########################################
def generate_feature_matrix_part5(df, word_dict):
    """
    Reads a dataframe and the dictionary of unique words
    to generate a matrix of {1, 0} feature vectors for each review.
    Use the word_dict to find the correct index to set to 1 for each place
    in the feature vector. The resulting feature matrix should be of
    dimension (# of reviews, # of words in dictionary).
    Input:
        df: dataframe that has the ratings and labels
        word_dict: dictionary of words mapping to indices
    Returns:
        a feature matrix of dimension (# of reviews, # of words in dictionary)
    """
    number_of_reviews = df.shape[0]
    number_of_words = len(word_dict)
    # create a feature data frame instead of a matrix filled w/ 0's
    feature_df = pd.DataFrame(np.zeros((number_of_reviews, number_of_words)), index=np.arange(number_of_reviews), columns =
list(word_dict.keys()))
    # go thru each review
    for i in range(0, number_of_reviews):
        # store review text in a list
        msg = (df.iloc[i]['reviewText']).split()
        # go thru words in review
        for word in msg:
            # if it exists, increment amount by 1
            if word in word_dict:
                feature_df.iloc[i][word] += 1
    return feature_df
```

Helper.py edited

```
def get_multiclass_training_data(class_size=750):
    """
    Reads in the data from data/dataset.csv and returns it using
    extract_dictionary and generate_feature_matrix as a tuple
    (X_train, Y_train) where the labels are multiclass as follows
        -1: poor
         0: average
         1: good
    Also returns the dictionary used to create X_train.
    Input:
        class_size: Size of each class (pos/neg/neu) of training dataset.
    """
    fname = "data/dataset.csv"
    dataframe = load_data(fname)
    neutralDF = dataframe[dataframe['label'] == 0].copy()
    positiveDF = dataframe[dataframe['label'] == 1].copy()
    negativeDF = dataframe[dataframe['label'] == -1].copy()
    X_train = pd.concat([positiveDF[:class_size], negativeDF[:class_size], neutralDF[:class_size]]).reset_index(drop=True).copy()
    dictionary = project1.extract_dictionary_part5(X_train)
    Y_train = X_train['label'].values.copy()
    X_train = project1.generate_feature_matrix_part5(X_train, dictionary)

    return (X_train, Y_train, dictionary)


def get_heldout_reviews(dictionary):
    """
    Reads in the data from data/heldout.csv and returns it as a feature
    matrix based on the functions extract_dictionary and generate_feature_matrix
    Input:
        dictionary: the dictionary created by get_multiclass_training_data
    """
    fname = "data/heldout.csv"
    dataframe = load_data(fname)
    X = project1.generate_feature_matrix_part5(dataframe, dictionary)
    return X
```

Main illustrating part of looping CV structure

```
    # Read multiclass data
    # TODO: Question 5: Apply a classifier to heldout features, and then use
    #       generate_challenge_labels to print the predicted labels
    print("producing feature matrices")
    multiclass_features, multiclass_labels, multiclass_dictionary = get_multiclass_training_data()
    heldout_features = get_heldout_reviews(multiclass_dictionary)

    ########## 5 ##########
    print("PART 5")
    # do one-vs-one, so scikitlearn default for SVC
    c_values = [0.001, 0.005, 0.02, 0.025, 0.05]

    ### PRINT OUTPUT FOR EACH CLASSIFIER TO TRACK OVERFITTING
    print("LINEAR CLASSIFIER CV PERFORMANCE")
    for c in c_values:
        clf = SVC(C = c, kernel = 'linear')
        linear_acc = cv_performance(clf, X = multiclass_features, y = multiclass_labels, k=5, metric="accuracy")
        print("Linear Test: " + str(c) + ", " + str(linear_acc))
```

Final Main

```
# Read multiclass data
# TODO: Question 5: Apply a classifier to heldout features, and then use
#       generate_challenge_labels to print the predicted labels
print("producing feature matrices")
multiclass_features, multiclass_labels, multiclass_dictionary = get_multiclass_training_data()
heldout_features = get_heldout_reviews(multiclass_dictionary)

########## 5 ##########
print("PART 5")

### PRINT OUTPUT FOR EACH CLASSIFIER TO TRACK OVERFITTING
print("Fit classifier")
clf = SVC(C = 0.015, kernel = 'linear').fit(X = multiclass_features, y = multiclass_labels)

print("Make predictions on held out set")
predVec = clf.predict(heldout_features)

print("Generate Challenge Labels")
generate_challenge_labels(predVec, "corioj")
```