

Project 1 Write-Up

2. a)

```
def extract_dictionary(df):
    """
    Reads a panda 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 = {}

    # TODO: Implement this function

    for i in range(len(list(df["text"]))):
        for punc in string.punctuation:
            df.iat[i, 1] = df.iat[i, 1].replace(punc, " ")
        df.iat[i, 1] = df.iat[i, 1].lower()
        for word in df.iat[i, 1].split():
            if word not in word_dict:
                word_dict[word] = 0
            word_dict[word] += 1

    return word_dict
```

2. b)

```
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 (number of reviews, number of words).
    Input:
        df: dataframe that has the ratings and labels
        word_list: dictionary of words mapping to indices
    Returns:
        a feature matrix of dimension (number of reviews, number of words)
    """

    number_of_reviews = df.shape[0]
    number_of_words = len(word_dict)
    feature_matrix = np.zeros((number_of_reviews, number_of_words))
    # TODO: Implement this function

    dict = list(word_dict.keys())
    col = list(df["text"])

    for i in range(number_of_reviews):
        review = col[i].split()
        for j in range(number_of_words):
            if dict[j] in review:
                feature_matrix[i][j] = 1

    return feature_matrix
```

2. c) The number of unique words is $d = 2850$ and the average number of non-zero features per rating is 15.624.

3.1 a)

```
def cv_performance(clf, X, y, k=5, metric="accuracy"):
    """
    # 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 = []

    skf = StratifiedKFold(n_splits=k, shuffle=False)
    skf.get_n_splits(X, y)

    for train_index, test_index in skf.split(X, y):
        clf.fit(X[train_index], y[train_index])
        if metric == "auROC":
            pred = clf.decision_function(X[test_index])
        else:
            pred = clf.predict(X[test_index])
        actual = y[test_index]
        scores.append(compute_score(pred, actual, metric))

    #And return the average performance across all fold splits.
    return np.array(scores).mean()
```

```
def compute_score(y_pred, y_true, metric):
    """
    :param y_pred: Predicted y values from svm model
    :param y_true: True y values from test data
    :param metric: Metric to be evaluated and returned
    :return: Returns the metric specified, computed from y_pred and y_true
    """

    if metric == 'auROC':
        return metrics.roc_auc_score(y_true, y_pred)

    tp, fn, fp, tn = metrics.confusion_matrix(y_true, y_pred, [1, -1]).ravel()

    if metric == 'accuracy':
        return metrics.accuracy_score(y_true, y_pred)
    if metric == 'precision':
        return metrics.precision_score(y_true, y_pred)
    if metric == 'sensitivity':
        return tp/(tp+fn)
    if metric == 'specificity':
        return tn/(tn+fp)
    if metric == 'f1-score':
        return metrics.f1_score(y_true, y_pred)
```

It is important to maintain class balances across folds because we want our training data to be a good representation of the data as a whole. Therefore, by keeping the proportions of positive and negative ratings fairly even across folds, it is possible to simulate the entire data set better.

3.1 b)

```
def select_param_linear(X, y, k=5, metric="accuracy", C_range = [], penalty='l2'):
    # TODO: Implement this function
    #HINT: You should be using your cv_performance function here
    #to evaluate the performance of each SVM

    print(metric)

    C_final = C_range[0]
    score = 0
    for c in C_range:
        clf = SVC(C=c, kernel="linear")
        print(c)
        print(cv_performance(clf, X, y, k, metric))
        print()
        if cv_performance(clf, X, y, k, metric) > score:
            score = cv_performance(clf, X, y, k, metric)
            C_final = c

    print(C_final)
    print(score)
    return C_final
```

3.1 c)

Performance Measure	C	Performance
Accuracy	0.1	0.83900000000000001
F1-score	0.1	0.8377282080627986
AUROC	0.1	0.92036
Precision	10	0.8412795192518695
Sensitivity	0.001	0.86400000000000001
Specificity	10	0.844

Specificity starts out at its optimal values, then drops and remains constant as C increases. Sensitivity increases as C does, until hitting its optimal value, and remaining constant as C continues to increase. Accuracy, f1-score, auroc, and precision all increase until they reach their optimal value, then decrease and remain constant as C continues to increase.

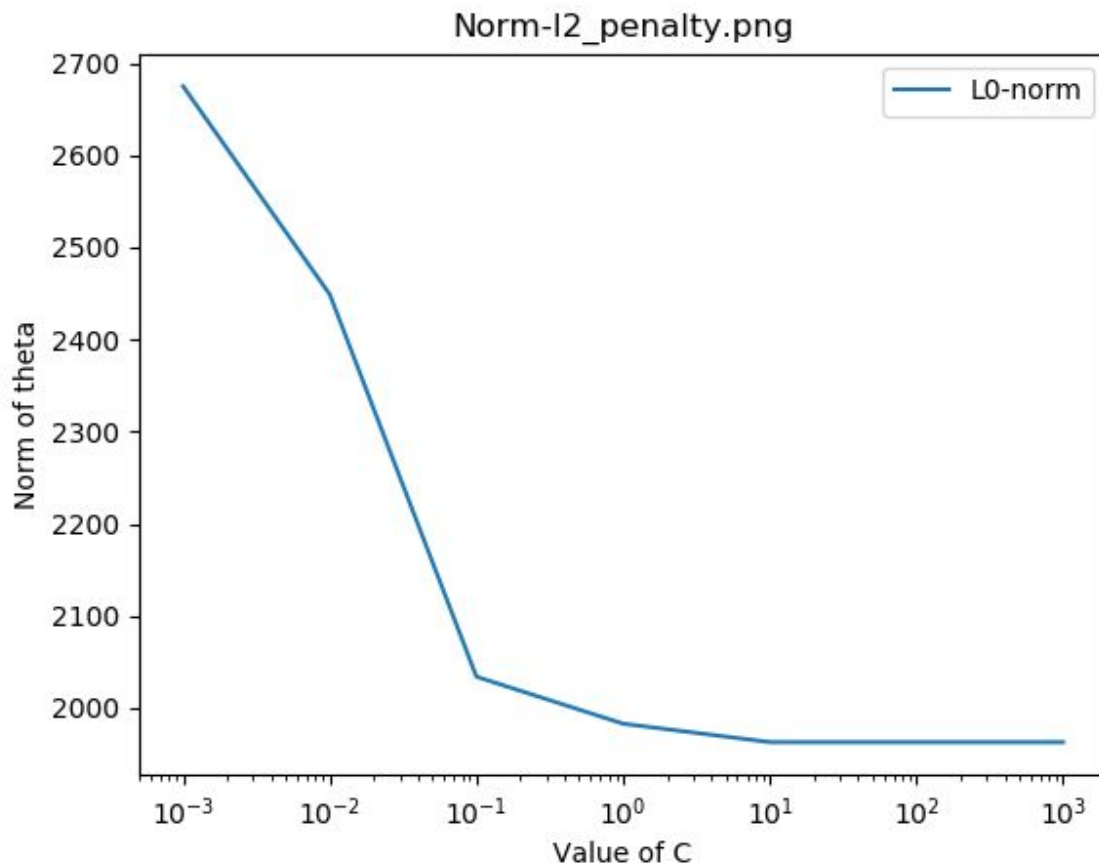
Accuracy seems like a reasonable performance measure to optimize for. It is straightforward, since it is simply the ratio of the correct predictions to the total number of observations. Also, accuracy is best when the data is symmetric. Our data is split evenly between positive and negative reviews, therefore we would expect our false positive and false negative rates to be

fairly similar (i.e. not biased one way or the other as would more likely be the case when one class of data is more prevalent than another). With accuracy as our chosen performance measure, we would choose a value $C=0.1$.

3.1 d)

Performance Measure	Performance
Accuracy	0.7975
F1-score	0.8019559902200488
AUROC	0.8933000000000001
Precision	0.784688995215311
Sensitivity	0.82
Specificity	0.775

3.1 e)



As the C value increases, the norm of theta decreases until C is about 10. This could be because of the increased penalty against the slack variable as C increases. This would cause there to be fewer non-zero coefficients, yet these coefficients would have greater magnitudes.

3.1 f)

Positive Coefficient	Word
0.96945305	thanks
0.90108408	thank
0.76542314	great
0.59590797	good

Negative Coefficient	Word
-0.61578807	hours
-0.54950526	delayed
-0.52082149	due
-0.50743263	worst

3.2 a) i.

```
def select_param_quadratic(X, y, k=5, metric="accuracy", param_range=[]):
    # 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...

    #
    # Grid Search
    #
    print("-----Part 3.2.b-----")
    print("-----Grid Search-----")
    final_c = 0
    final_r = 0
    final_score = 0
    for c_ind in range(len(param_range)):
        for r_ind in range(len(param_range)):
            c = param_range[c_ind][0]
            r = param_range[r_ind][1]
            clf = SVC(kernel="poly", degree=2, C=c, coef0=r, class_weight="balanced")
            score = cv_performance(clf, X, y, k, metric)
            print("Score: ")
            print(score)
            print(c)
            print(r)
            print()
            if score > final_score:
                final_score = score
                final_c = c
                final_r = r

    print(final_score)
    print(final_c)
    print(final_r)
```


3.2 a) ii.

```
def select_param_quadratic(X, y, k=5, metric="accuracy", param_range=[]):
    """
    # 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...

    #
    # Random Search
    #
    print("-----Part 3.2.b-----")
    print("-----Random Search-----")
    final_c = 0
    final_r = 0
    final_score = 0

    for i in range(25):
        c = 10 ** uniform(-3, 3)
        r = 10 ** uniform(-3, 3)
        clf = SVC(kernel="poly", degree=2, C=c, coef0=r, class_weight="balanced")
        score = cv_performance(clf, X, y, k, metric)
        print("Score: ")
        print(score)
        print(c)
        print(r)
        print()
        if score > final_score:
            final_score = score
            final_c = c
            final_r = r

    print(final_score)
    print(final_c)
    print(final_r)
```

3.2 b)

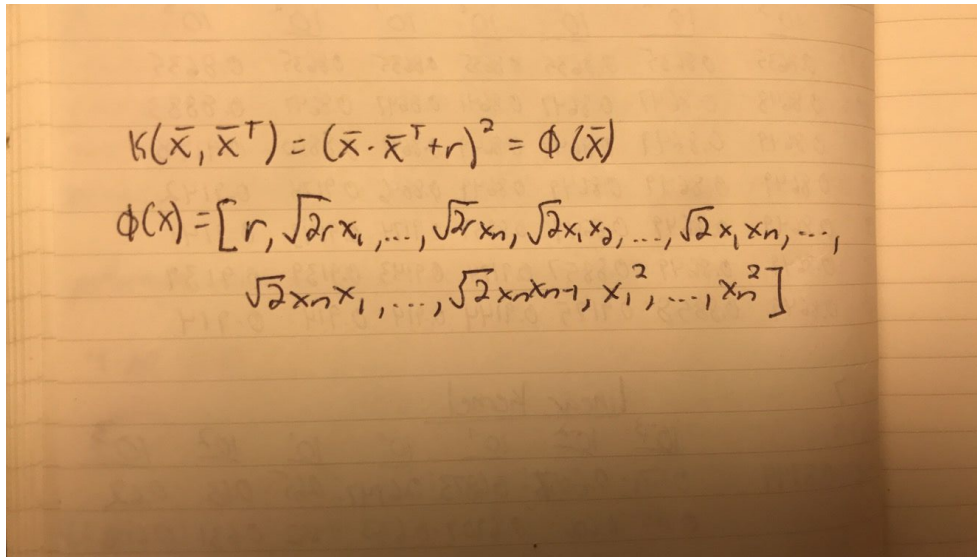
Tuning Scheme	C	r	AUROC
Grid Search	1000	0.1	0.91776
Random Search	35.40065235104957	7.135576847564638	0.9239

Using grid search: Holding C constant, as r increased, there tended to be an increase in AUROC score. As we held r constant and increased C, AUROC score tended to remain fairly consistent. At higher r values, AUROC increased more as C increased.

Using random search: AUROC scores varied from about 0.864 to 0.92, which is very close to the same range that we got in grid search.

Random search can be better than grid search. We only ran 25 random searches and ended up with hyperparameters that better optimized AUROC. This is in contrast to running 49 grid searches, only to end up with an AUROC score that was not as high. In our case, random search was better and more computationally efficient. However, random search does not guarantee a better result as it is not deterministic.

3.3 a)



$$k(\bar{x}, \bar{x}^T) = (\bar{x} \cdot \bar{x}^T + r)^2 = \Phi(\bar{x})$$

$$\Phi(x) = [r, \sqrt{2r}x_1, \dots, \sqrt{2r}x_n, \sqrt{x_1x_2}, \dots, \sqrt{x_1x_n}, \dots, \sqrt{x_nx_1}, \dots, \sqrt{x_nx_{n-1}}, x_1^2, \dots, x_n^2]$$

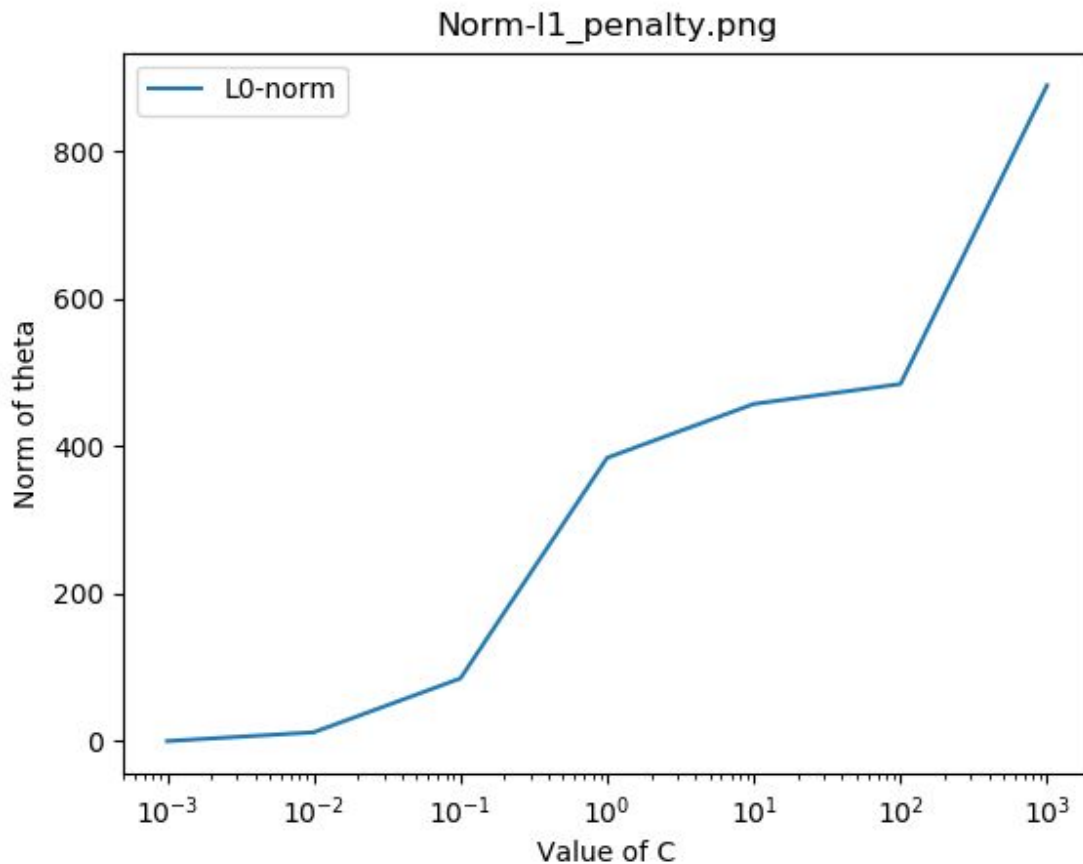
3.3 b)

Pros: We can access the individual feature weights when using explicit feature mapping.

Cons: There are many more features in the transformed data so explicit feature mapping can be much more computationally expensive.

3.4 a) The setting that maximizes AUROC is $C = 100$, with an AUROC score of 0.9166. We can see that as C increases, so does AUROC, until it reaches its optimal score at 100 and decreases.

3.4 b)



3.4 c) Unlike the L2-penalty, as the value of C increases, the norm of θ increases, meaning the number of non-zero coefficients increases. When C is small, there are very few non-zero coefficients.

3.4 d) Squared hinge loss is more lenient on points that are within the margin, yet more punishing to points that are misclassified. Because of this, the optimal solution will likely have a wider margin and more support vectors when compared to the hinge loss.

4.1 a) If W_n is much greater than W_p , it means that negative points are in a sense “more important” than positive points. A higher weight means that the point is punished more when misclassified, and therefore we can expect more of the negative points to be correctly classified.

4.1 b)

Performance Measure	Performance
Accuracy	0.515

F1-score	0.058252427184466014
AUROC	0.869175
Precision	1.0
Sensitivity	0.03
Specificity	1.0

4.1 c) F1-score, precision, sensitivity, and specificity were all greatly affected. Accuracy also took a hit, just not quite as extreme. These changes are as we would expect. Since the negative weight is much greater than the positive weight, most of the negative points are correctly classified. A precision and specificity of 1 indicate that there were 0 false positive values, meaning every negative point was classified correctly. However, the drawback of this is having very low sensitivity and f1-score, which is caused by having lots of false negative values.

4.2 a)

Class Weights	Performance Measure	Performance
$W_n = 1, W_p = 1$	Accuracy	0.206
$W_n = 1, W_p = 1$	F1-score	0.014888337468982629
$W_n = 1, W_p = 1$	AUROC	0.87365
$W_n = 1, W_p = 1$	Precision	1.0
$W_n = 1, W_p = 1$	Sensitivity	0.0075
$W_n = 1, W_p = 1$	Specificity	1.0

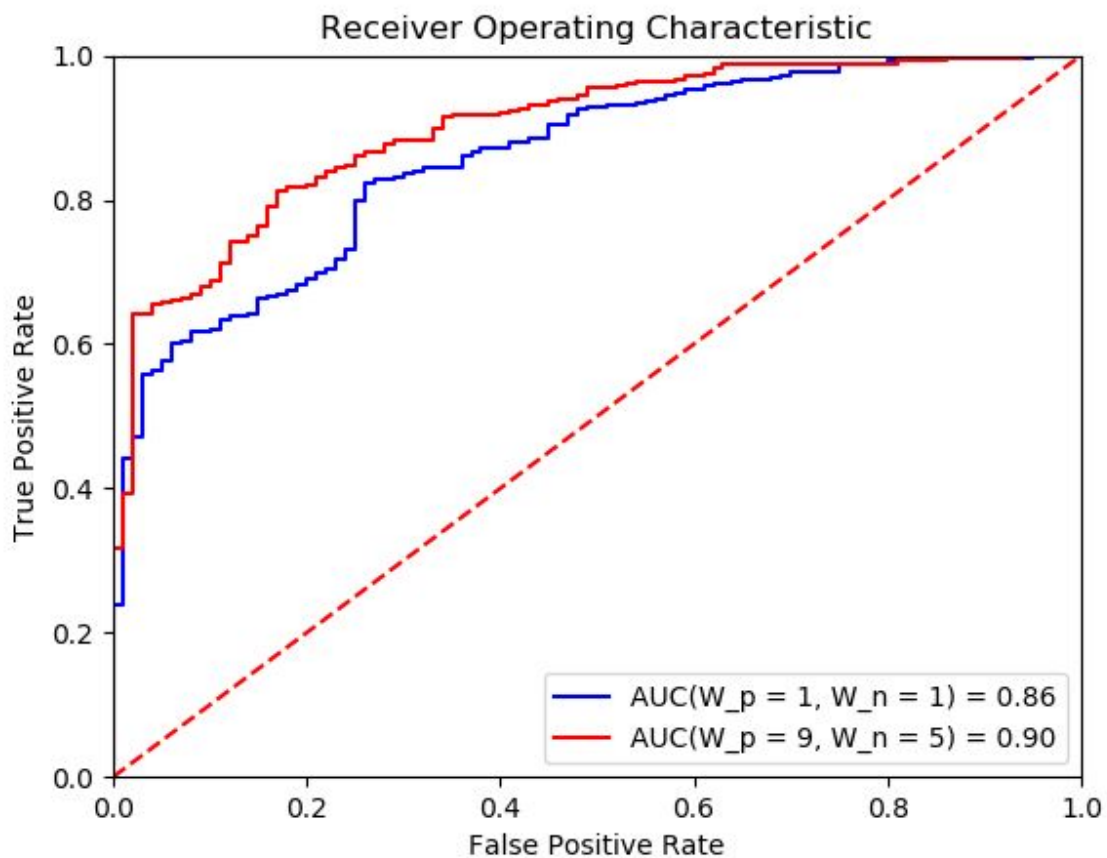
4.2 b) Training on an imbalanced dataset has hurt performance in a very similar way to what we saw in 4.1. Accuracy, f1-score, and sensitivity all decreased, while precision and specificity both increased. Since their value is again 1, it seems like we have zero false positive values, yet lots of false negatives.

4.3 a) Knowing that the training data has 700 negative points and 300 positive points, we can make an educated prediction about the class weights we should choose. We would expect the ratio of W_n/W_p to be inversely proportional to the ratio of negative to positive points, so 3/7. Knowing this we can set the test range as 1-5 for W_n and 6-10 for W_p . Given that the data is imbalanced, f1-score would be a better performance measure to use. F1 is the weighted average of sensitivity and precision, and therefore more useful than accuracy if there is an uneven class distribution. A grid search through the values for W_n and W_p , optimizing the f1-score results in $W_n=5$ and $W_p=9$.

4.3 b)

Class Weights	Performance Measure	Performance
$W_n = 5, W_p = 9$	Accuracy	0.792
$W_n = 5, W_p = 9$	F1-score	0.8571428571428571
$W_n = 5, W_p = 9$	AUROC	0.898225
$W_n = 5, W_p = 9$	Precision	0.9512195121951219
$W_n = 5, W_p = 9$	Sensitivity	0.78
$W_n = 5, W_p = 9$	Specificity	0.84

4.4 a)



5. Challenge

To train a multiclass SVM classifier, we first decided to apply some feature engineering. To start, I created a dictionary from the reviews in the csv file. I saw that it contained over 5000

words. Given that we only had 3000 samples, I figured it would be a good idea to get rid of some features. After reading about some strategies, I decided to remove all of the stopwords using the nltk library. Stopwords are very common words that should have very little effect on the sentiment of a sentence (such as a, an, in). I then decided to take out all of the one letter words that were left. Actual one letter words like 'a' and 'i' were already removed, so the remaining one letter words were essentially nonsense like 'g', 'p', and one digit numbers. I then decided to do the same thing with the 2-letter words. I decided to keep quite a few of them in, as many more of the 2-letter words seemed relevant, including the 2-digit numbers. This greatly reduced the number of features. Finally, I ended up adding a feature for the number of retweets. The idea being that if a tweet gets retweets, its sentiment is felt more strongly. After all of this I had a feature matrix with 2178 features.

I was then ready to start tuning hyperparameters. The clear performance metric to optimize for was accuracy. It is the most straightforward to calculate for a multiclass problem, and it is what we will be tested on. Accuracy was measured as the mean accuracy from 5-fold cross validation. I started with trying a quadratic kernel, but found it was not very computationally efficient, and had 2 hyperparameters to tune. I went to a linear kernel and found that it ran faster, and gave similar performance compared to quadratic. The C value I found was 0.1. I decided to try a radial kernel, and was impressed with its performance despite its computational inefficiency. It resulted in a C value around 200 however. After seeing this I became concerned with overfitting and decided to go back to a linear kernel, this time using LinearSVC rather than SVC. Due to the underlying LinearSVC implementation, it runs much faster than SVC with a linear kernel. Not only did it run faster, it also performed better than even the radial kernel. I then found that a hinge loss gave even better performance than the default squared hinge loss. SVC uses a one-vs-one multiclass implementation, whereas LinearSVC uses a one-vs-rest strategy. Based on the performance of LinearSVC, I decided to go with the one-vs-rest implementation it uses. The optimum mean accuracy was found using LinearSVC with C=0.1 and loss='hinge' and had a value of 0.7136.

Appendix

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

    part_2(X_train)

    # Part 3.1.c
    part_3_c(X_train, Y_train, X_test, Y_test)

    # Part 3.1.d
    m_s = ['accuracy', 'f1-score', 'auROC', 'precision', 'sensitivity', 'specificity']
    for m in m_s:
        part_3_d(X_train, Y_train, X_test, Y_test, m)

    # Part 3.1.e
    c_range = [10 ** -3, 10 ** -2, 10 ** -1, 10 ** 0, 10 ** 1, 10 ** 2, 10 ** 3]
    plot_weight(X_train, Y_train, "l2", "accuracy", c_range)

    # Part 3.1.f
    part_3_f(X_train, Y_train, dictionary_binary)

    # Part 3.2.b
    pr = np.array([[10 ** -3, 10 ** -3],
                  [10 ** -2, 10 ** -2],
                  [10 ** -1, 10 ** -1],
                  [10 ** 0, 10 ** 0],
                  [10 ** 1, 10 ** 1],
                  [10 ** 2, 10 ** 2],
                  [10 ** 3, 10 ** 3]])
    select_param_quadratic(X_train, Y_train, 5, metric="auROC", param_range=pr)

```

```

# Part 3.4.a
part_3_4_a(X_train, Y_train, 5, metric='auROC')

# Part 3.4.b
part_3_4_b(X_train, Y_train)

# Part 4.1.b
part_4_1_b(X_train, Y_train, X_test, Y_test, m_s)

# Part 4.2
part_4_2(IMB_features, IMB_labels, IMB_test_features, IMB_test_labels, m_s)

# Part 4.3
part_4_3(IMB_features, IMB_labels, IMB_test_features, IMB_test_labels, m_s)
clf = SVC(C=0.01, kernel='linear', class_weight={-1: 5, 1: 9})
clf.fit(IMB_features, IMB_labels)
for m in m_s:
    if m == 'auROC':
        pred = clf.decision_function(IMB_test_features)
    else:
        pred = clf.predict(IMB_test_features)
    print(m)
    print(compute_score(pred, IMB_test_labels, m))

# Part 4.4
part_4_4(IMB_features, IMB_labels, IMB_test_features, IMB_test_labels)

```

```

# 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)

print('====Challenge====')
clf = LinearSVC(C=0.1, loss='hinge')
print(cv_performance(clf, multiclass_features, multiclass_labels, 5, 'accuracy'))
clf = LinearSVC(C=0.1, loss='hinge')
clf.fit(multiclass_features, multiclass_labels)
pred = clf.predict(heldout_features)
generate_challenge_labels(pred, 'codelau')

```

2.

```
def part_2(x_train):  
    print("-----Part 2-----")  
    print(x_train.shape[1])  
    tot = 0  
    for i in range(x_train.shape[0]):  
        tot += sum(x_train[i])  
    print(tot/x_train.shape[0])
```

3. c)

```
def part_3_c(X_train, Y_train, X_test, Y_test):  
    print("-----Part 3_c-----")  
    c_range = [10 ** -3, 10 ** -2, 10 ** -1, 10 ** 0, 10 ** 1, 10 ** 2, 10 ** 3]  
    m_s = ['accuracy', 'f1-score', 'auROC', 'precision', 'sensitivity', 'specificity']  
    for m in m_s:  
        select_param_linear(X_train, Y_train, k=5, metric=m, C_range=c_range)
```

3. d)

```
def part_3_d(X_train, Y_train, X_test, Y_test, metric="accuracy"):  
    print("-----Part 3_d-----")  
    clf = SVC(C=0.1, kernel="linear", class_weight="balanced")  
    clf.fit(X_train, Y_train)  
    if metric == "auROC":  
        pred = clf.decision_function(X_test)  
    else:  
        pred = clf.predict(X_test)  
  
    print(metric)  
    print(compute_score(pred, Y_test, metric))
```

3. f)


```
def part_3_f(X_train, Y_train, dict):
    print("-----Part 3_f-----")
    clf = SVC(C=0.1, kernel="linear")
    clf.fit(X_train, Y_train)
    a = np.array(clf.coef_[0])
    top_ind = np.argpartition(a, -4)[-4:]
    bot_ind = np.argpartition(a, 4)[:4]

    print(top_ind)
    print(a[top_ind])
    print(bot_ind)
    print(a[bot_ind])
    for i in top_ind:
        print(list(dict.keys())[i])
    for ix in bot_ind:
        print(list(dict.keys())[ix])
```

3.4 a)

```
def part_3_4_a(X_train, Y_train, k=5, metric='accuracy'):
    print("-----Part 3_4_a-----")
    final_c = 0
    final_score = 0
    c_range = [10 ** -3, 10 ** -2, 10 ** -1, 10 ** 0, 10 ** 1, 10 ** 2, 10 ** 3]
    for c in c_range:
        clf = LinearSVC(penalty='l1', dual=False, C=c, class_weight='balanced')
        score = cv_performance(clf, X_train, Y_train, k, metric)
        print(c)
        print(score)
        print()
        if score > final_score:
            final_score = score
            final_c = c

    print(final_score)
    print(final_c)
```

3.4 b)

```
def part_3_4_b(X_train, Y_train):
    print("-----Part 3_4_b-----")
    c_range = [10 ** -3, 10 ** -2, 10 ** -1, 10 ** 0, 10 ** 1, 10 ** 2, 10 ** 3]
    plot_weight(X_train, Y_train, penalty='l1', metric='auROC', C_range=c_range)
```

4.1 b)

```

def part_4_1_b(X_train, Y_train, X_test, Y_test, mets):
    print("-----Part 4_1_b-----")
    for m in mets:
        #clf = SVC(C=0.01, kernel='linear', class_weight={-1: 10, 1: 1})
        clf = SVC(kernel='linear', C=0.01, class_weight={-1: 10, 1: 1})
        clf.fit(X_train, Y_train)
        if m == 'auROC':
            pred = clf.decision_function(X_test)
        else:
            pred = clf.predict(X_test)
        print(m)
        print(compute_score(pred, Y_test, m))

```

4.2

```

def part_4_2(X_train, Y_train, X_test, Y_test, mets):
    print("-----Part 4_2-----")
    for m in mets:
        clf = SVC(C=0.01, kernel='linear', class_weight={-1: 1, 1: 1})
        clf.fit(X_train, Y_train)
        if m == 'auROC':
            pred = clf.decision_function(X_test)
        else:
            pred = clf.predict(X_test)
        print(m)
        print(compute_score(pred, Y_test, m))

```

4.3

```

def part_4_3(X_train, Y_train, X_test, Y_test, mets):
    print("-----Part 4_3-----")
    w_p = [6, 7, 8, 9, 10]
    w_n = [1, 2, 3, 4, 5]

    final_pweight = 0
    final_nweight = 0
    final_score = 0
    for p in w_p:
        for n in w_n:
            clf = SVC(C=0.01, kernel='linear', class_weight={-1: n, 1: p})
            score = cv_performance(clf, X_train, Y_train, 5, 'f1-score')
            print(score)
            print(p)
            print(n)
            print()
            if score > final_score:
                final_score = score
                final_pweight = p
                final_nweight = n

    print(final_score)
    print(final_pweight)
    print(final_nweight)

```

4.4

```

def part_4_4(X_train, Y_train, X_test, Y_test):
    print("-----Part 4_4-----")
    clf1 = SVC(C=0.01, kernel='linear', class_weight='balanced')
    clf2 = SVC(C=0.01, kernel='linear', class_weight={-1: 5, 1: 9})
    clf1.fit(X_train, Y_train)
    clf2.fit(X_train, Y_train)
    pred1 = clf1.decision_function(X_test)
    pred2 = clf2.decision_function(X_test)
    fpr1, tpr1, threshold1 = metrics.roc_curve(Y_test, pred1)
    fpr2, tpr2, threshold2 = metrics.roc_curve(Y_test, pred2)
    roc_auc1 = metrics.auc(fpr1, tpr1)
    roc_auc2 = metrics.auc(fpr2, tpr2)
    plt.title('Receiver Operating Characteristic')
    plt.plot(fpr1, tpr1, 'b', label='AUC(W_p = 1, W_n = 1) = %0.2f' % roc_auc1)
    plt.plot(fpr2, tpr2, 'r', label='AUC(W_p = 9, W_n = 5) = %0.2f' % roc_auc2)
    plt.legend(loc='lower right')
    plt.plot([0, 1], [0, 1], 'r--')
    plt.xlim([0, 1])
    plt.ylim([0, 1])
    plt.ylabel('True Positive Rate')
    plt.xlabel('False Positive Rate')
    plt.savefig('4_4_AUROC.png')
    plt.close()

```

Text_process:

```

def text_process(text):
    strip_punc = [c for c in text if c not in string.punctuation]
    strip_punc = ''.join(strip_punc)
    strip_punc = strip_punc.lower()
    stopwords = nltk.corpus.stopwords.words('english')
    newstopwords = ['4', '2', '7', '3', '5', '1', '8', '0', '9', 'f', 'n', 'g', 'u', 'w', 'b', 'p', 'r', '6', 'k', 'x',
                    'cs', 'kp', 'kn', 'fa', 'ua', 'fo', 'st', 'jt', 'rr', 'pr', 'ey', 'gt', 'ff',
                    'lk', 'yo', 'um', 'jj', 'jh', 'ya', 'cr', 'th', 'lh', 'http']
    stopwords.extend(newstopwords)
    return [word for word in strip_punc.split() if word.lower() not in stopwords]

```

Select_classifier:

```

def select_classifier(penalty='l2', c=1.0, degree=1, r=0.0, class_weight='balanced'):
    """
    Return a linear svm classifier based on the given
    penalty function and regularization parameter c.
    """
    # TODO: Optionally implement this helper function if you would like to
    # instantiate your SVM classifiers in a single function. You will need
    # to use the above parameters throughout the assignment.

    if penalty == 'l1':
        clf = LinearSVC(penalty=penalty, C=c, dual=False, class_weight='balanced')
    else:
        clf = LinearSVC(C=c, kernel='linear', class_weight='balanced')

    return clf

```