Project 1 - Sachchit's Social SVMs

2 DataSet Considerations

a) **(i)** Most messages merely convey an emotion that can be classified as sad or happy, so it's hard to say whether we can identify an individual. Some messages do you share personal anecdotes from which it may be possible to roughly guess the identity of a person like in the Reddit message "Noooooooooo I was a student staff member at the WCC and this makes me so sad and angryyyyy." However, most messages are far more vague like "Honestly im a pretty clumsy belligerent and i lasted 9 months and no case is great life i regret nothing," in which case nothing is deducible. In general, no, it is not possible to deduce the identity of an individual easily based on their messages from this dataset or using other data unless they give easily identifiable hints.

(ii) Since the goal of this dataset is to analyze the emotion of a given online forum through reading its messages, You could try testing it on random reddit forums whose comments follow a somewhat similar distribution to our dataset. For example, testing it out on a reddit forum about a swimming tournament and seeing the general emotion of the crowd after a certain swimmer wins. From that you could get a rough idea of how much the crowd likes the swimmer. If most comments are gratitude, you know the crowd was cheering for the swimmer from the beginning. If most comments are neutral or sad despite the swimmer winning, you can tell that the swimmer is not liked very much.

(iii) The model trained from our data set shouldn't be used to generalize for online forums where there can be a larger range of emotions including anger or several people making sarcastic comments where it is difficult for even a human to identify the exact emotion of the writer. Take for example, a recent chess scandal, where the world champion accused someone of cheating. There will definitely be several people joking or writing angry comments and it is hard to classify each one into gratitude, sadness, and even neutral. Hence, we cannot carelessly generalize the areas where our model can be used and we need to keep in mind the kind of dataset it has been trained on.

b) **(i)** Since the data is given to several people to label because of crowdsourcing, the training data is bound to be error-prone because no one is supervising the humans who label the data and they may label it wrongly.

(ii) Wrong labels tend to add 'noise' to our training data and our model, especially if the wrong labels have a specific pattern to them which the model will likely try to incorporate. Oftentimes, random wrong labels are better than having structures labeling errors. In any case, wrong labels can cause the model to misclassify new input data.

3 Feature Extraction

a) Result = ['it', 's', 'a', 'test', 'sentence', 'does', 'it', 'look', 'correct']

b) d = 4920

c) Number of non-zero features = 12.2678
Word appearing in the greatest number of comments = judging

4 Hyperparameter and Model Selection

4.1 Hyperparameter Selection for a Linear-Kernel SVM

a) It is useful to maintain class proportions across folds because doing this with the target variable ensures that the cross-validation error is consistent/ a close approximation of the generalization error.

b)

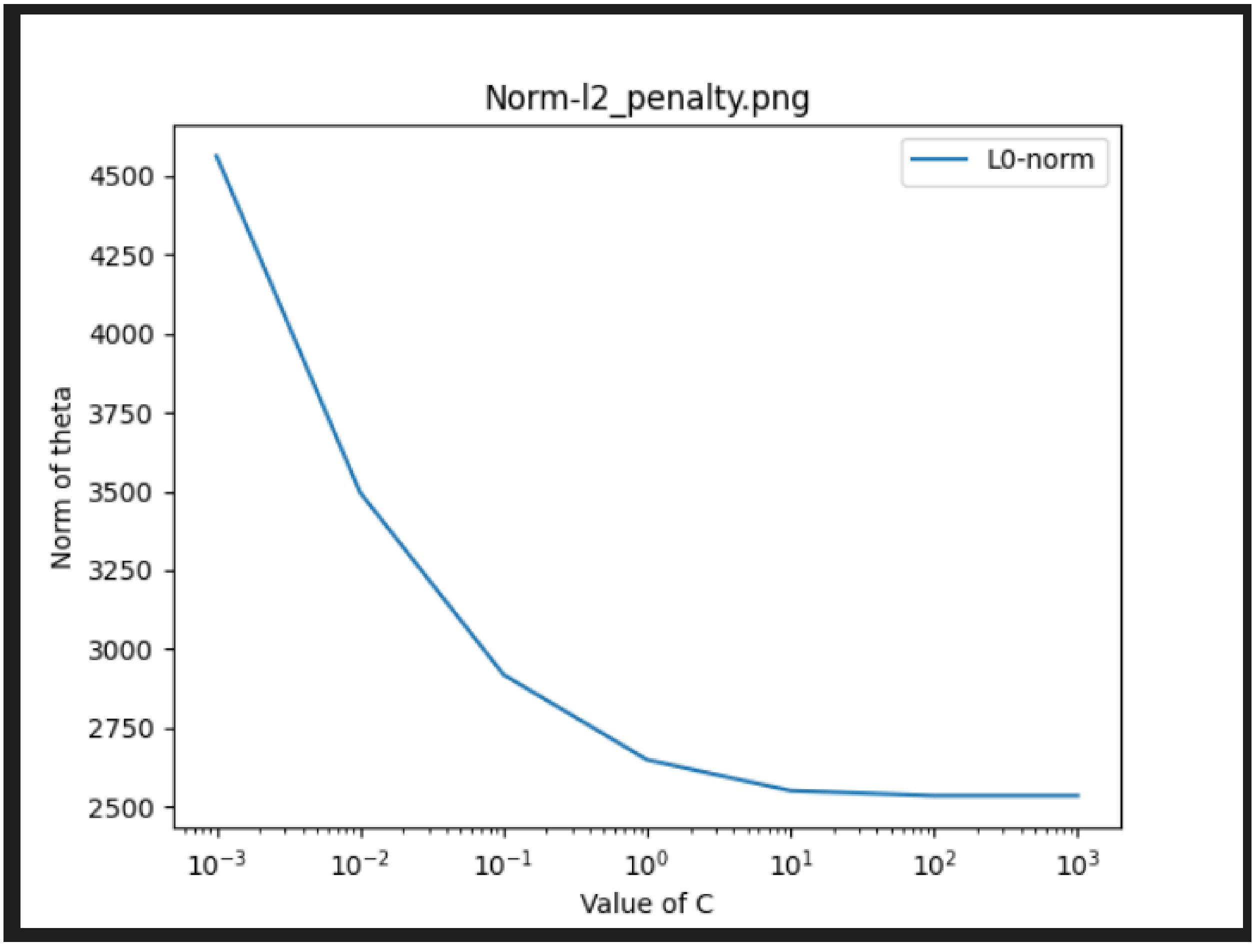
Performance Measures	C	CV Performance
Accuracy	0.1	0.9229
F1-Score	1	0.9211
AUROC	0.1	0.9730
Precision	0.01	0.9941
Sensitivity	10	0.9099

Specificity	10	0.9099

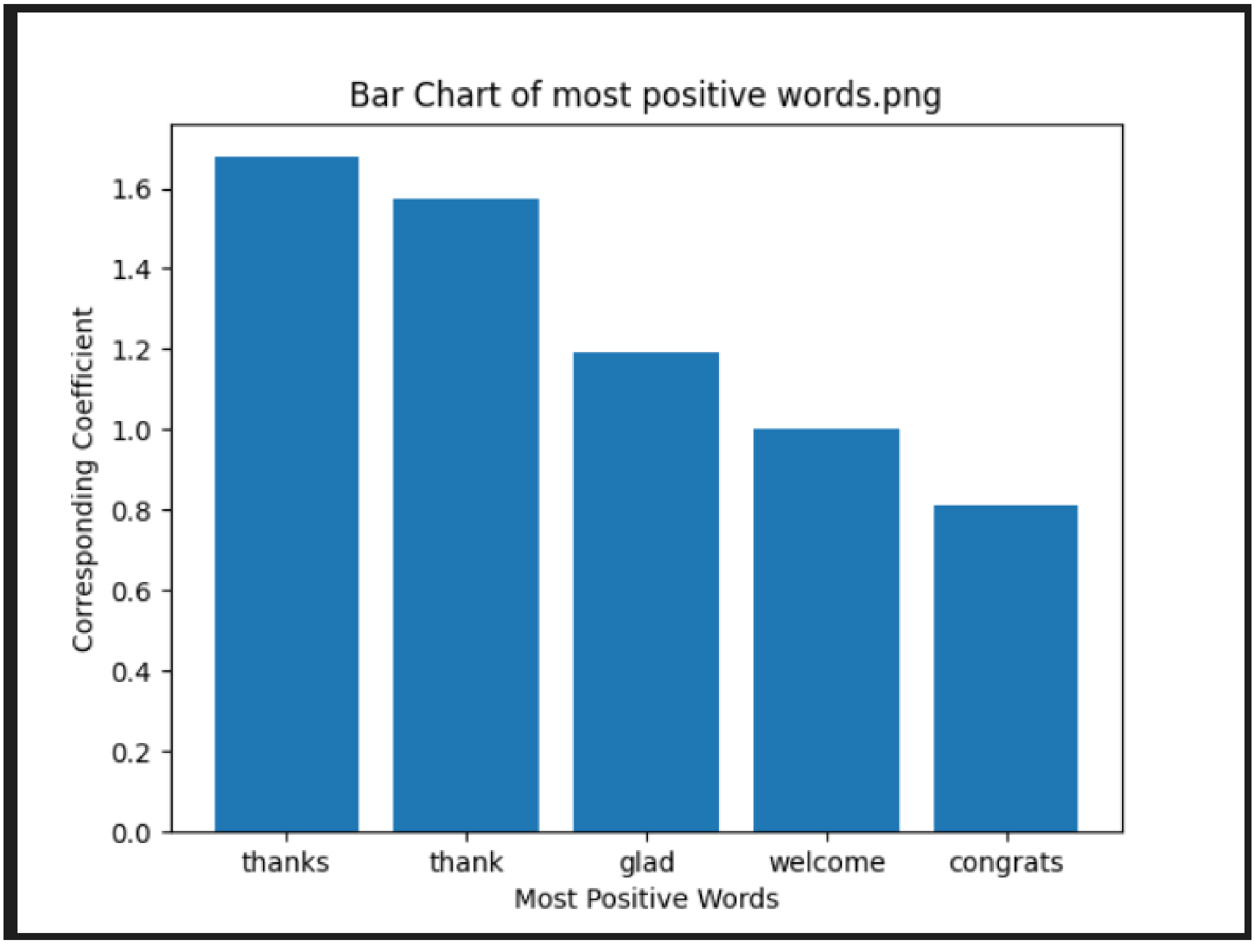
c) Value of C that maximizes chosen performance = 0.1

Performance Measures	Performance
Accuracy	0.9332
F1-Score	0.9291
AUROC	0.9771
Precision	0.9898
Sensitivity	0.8754
Specificity	0.8754

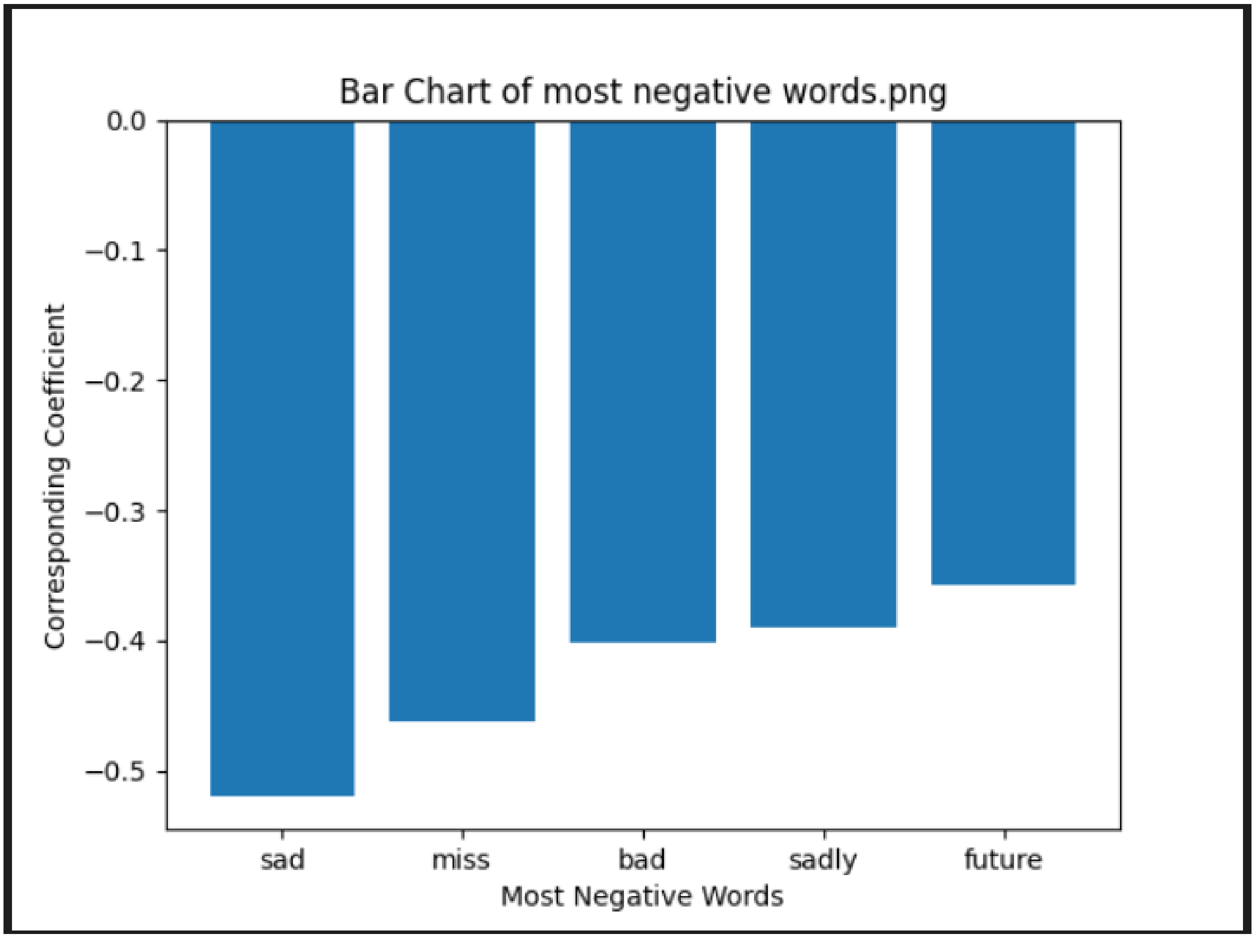
d) Plot produced using penalty = L2



e) Positive Words Bar Chart:



Negative Words Bar Chart:

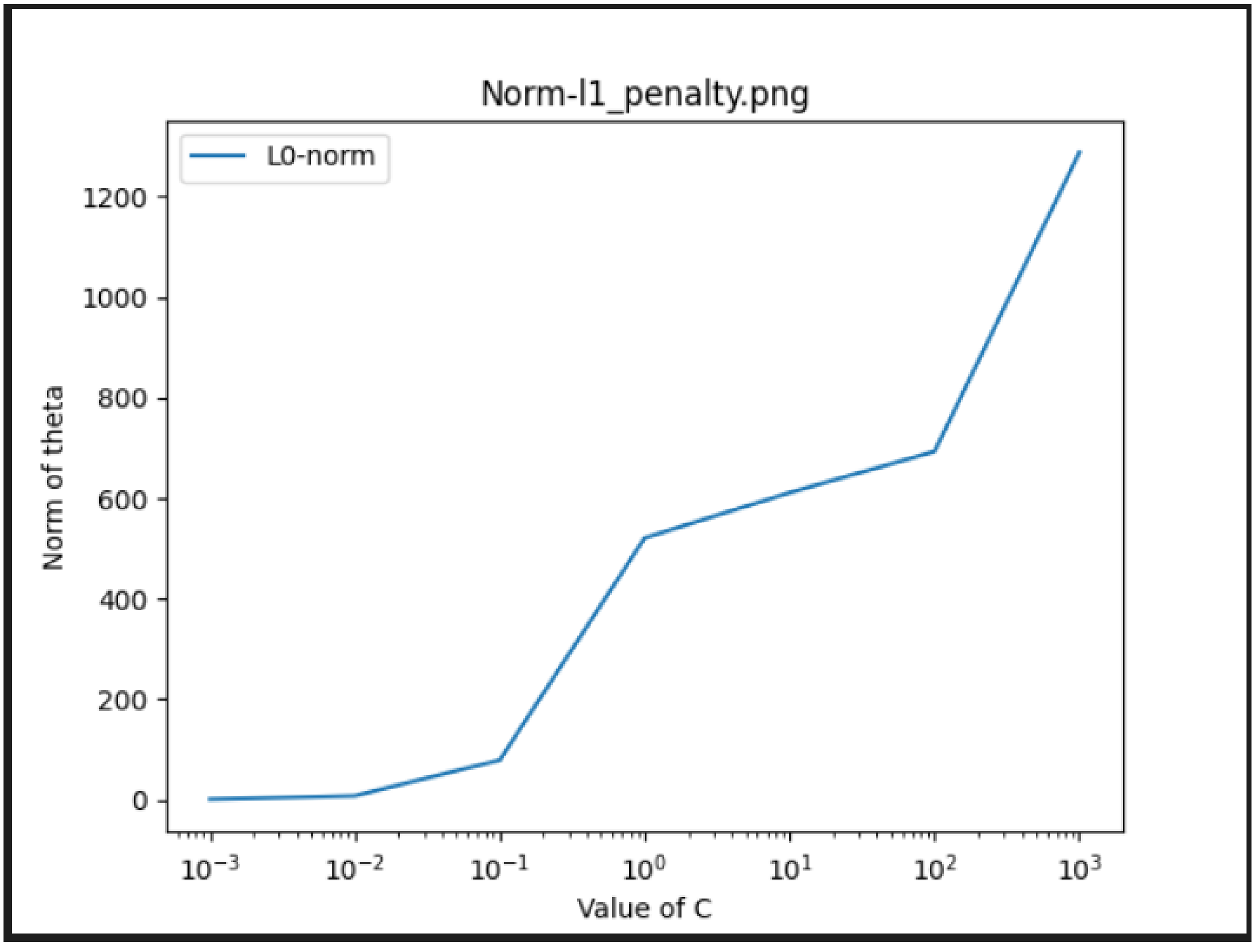


f) Sentence: HAHAHA! Thanks so much! Thank you so much for everything! I am SO merry! Ahahahaha! Haaaa! Haa… Yes, I'm being sarcastic!

4.2 Linear Kernel SVM with L1 Penalty and Squared Hinge Loss

a) C = 0.001 is the optimal solution with an auroc score = 0.8645 and a CV auroc score = 0.9204

b)



c) The graph for the l1 penalty observes a monotonically increasing value of the L0-norm of theta as the value of C increases, whereas the l2 penalty observes the opposite where the L0 norm of theta is monotonically decreasing as the value of C increases. The gradient descent is used to decrease the loss and regularization is used to avoid over-fitting. For the gradient of an L2 regularizer, the answer is a closed form solution, so it is understandable that the norm of theta decreases because in a L2 regularizer,

d) The squared hinge loss's optimal solution is affected more by outliers than the hinge loss. Also, if there are several points closer to the margin boundary, hinge loss' optimal solution will be affected more than the square hinge loss function since squaring a number less than 1 decreases it and consequently, the objective function's value will be decreased.

4.3 Hyperparameter Selection for a Quadratic-Kernel SVM

a)

Tuning Scheme	C	r	AUROC
Grid Search	100	10	0.93996
Random Search	0.8798	667.102	0.93996

b) The AUROC values for the two are the same at 0.93996, but that's because I used the same code for my AUROC variable in both. That could be wrong, and I think I need to set the feature vectors (n,d) as the number of hyperparameter combinations used in each, where the "d" for Grid Search will be greater than the "d" for Random search since Grid Search does a more extensive search. Ignoring that, the C value for Grid Search, C = 100, is much higher than for Random Search's C = 0.8798, whereas the r value for Random Search, r = 667.102, is much higher than that of Grid search's r = 10.

Grid search looks at every possible combination of hyperparameters, whereas random search tests a random combination of hyperparameters. Random search is more efficient on hyperparameter optimization since it tests a smaller number of hyperparameter combinations, so if you're looking for a relatively faster optimization performance, random search is better. On the other hand, grid search tests all possible hyperparameter combinations and returns the set of parameters with the top accuracy. If you want a highly accurate model and don't care about efficiency, grid search is better.

4.4 Learning Non-linear Classifiers with a Linear-Kernel SVM

a) Feature mapping:

$$(\bar{x} \cdot \bar{x}' + r)^2$$

$$\bar{x} \cdot \bar{x}' = \begin{bmatrix} x_1 \\ x_2 \end{bmatrix} \cdot \begin{bmatrix} x_1' \\ x_2' \end{bmatrix} = x_1x_1' + x_2x_2'$$

$$(\bar{x} \cdot \bar{x}' + r)(\bar{x} \cdot \bar{x}' + r)$$

$$= (x_1x_1' + x_2x_2' + r)(x_1x_1' + x_2x_2' + r)$$

$$= x_1^2x_1^{2'} + x_1x_1'x_2x_2' + rx_1x_1' + x_1x_1'x_2x_2' + x_2^2x_2^{2'} + rx_2x_2' + rx_1x_1' + rx_2x_2' + r^2$$

$$= r^2 + 2rx_1x_1' + 2rx_2x_2' + 2x_1x_1'x_2x_2' + x_1^2x_1^{2'} + x_2^2x_2^{2'}$$

$$\begin{bmatrix} r \\ \sqrt{2r}\,x_1 \\ \sqrt{2r}\,x_2 \\ \sqrt{2}\,x_1x_2 \\ x_1^2 \\ x_2^2 \end{bmatrix} \cdot \begin{bmatrix} r \\ \sqrt{2r}\,x_1' \\ \sqrt{2r}\,x_2' \\ \sqrt{2}\,x_1'x_2' \\ x_1^{2'} \\ x_2^{2'} \end{bmatrix}$$

$$\phi(z) = [r, \sqrt{2r}\,z_1, \sqrt{2r}\,z_2, \sqrt{2}\,z_1z_2, z_1^2, z_2^2]$$

b) The quadratic-kernel SVM can be inexpensive to calculate whereas an explicit feature mapping could map to a very high dimension resulting in a very long runtime. However, for a large n number of training examples, calculating the kernel for every pair of input can similarly be very expensive and it is better to use feature maps for very large datasets because they can be quite efficient if we can transform and store the data input efficiently.

5 Asymmetric Cost Functions

5.1 Arbitrary Class Weights

a) If Wn is much larger than Wp, the negative points will come with a heavier penalty since the SVM formulation multiplies Wn to C * summation. This is necessary in the case that there are less negative labels than positive labels, it will help the algorithm to correct the imbalance to a certain degree because the associated penalties are greater with the negative class.

b) Multiplying by Wn = 0.25 and Wp = 1.0 means there is a smaller penalty associated with the negative class whereas Wn = 1 and Wp = 4 means there is a greater penalty associated with the positive class. This is an important distinction because although both weight classes differ by a factor of 4, one influences the algorithm to be more insensitive to the negative class while the other influences the algorithm to be more sensitive to the positive class.

c)

Performance Measures	Performance
Accuracy	0.6381
Precision	0.5808
Sensitivity	0.9925
Specificity	0.9925
F1-score	0.7328
AUROC	0.9571

d) The accuracy, precision, and f1-score values changed the most from earlier. Since the f1-score is directly proportional to precision, it is no surprise that a decrease in precision brought a decrease in the f1-score value. The only reason it didn't decrease as much is because the sensitivity value didn't fluctuate much despite adding class weights. The only variable different in precision and sensitivity is False Positive and Negative. Since Precision's value decreased, it indicates that FP's value increased significantly and FN didn't change much. This also explains why there was a significant decrease in the accuracy as well. This makes sense since the weight of every positive label was amplified by 10 times, so FP values saw an increase whereas negative label values were weighted just like before hence not making much of a difference to the accuracy value.

5.2 Imbalanced Data

a)

Class Weights	**Performance Measures**	**Performance**
Wn = ?, Wp = ?	Accuracy	0.8055
Wn = ?, Wp = ?	Precision	0.8043
Wn = ?, Wp = ?	Sensitivity	1.0
Wn = ?, Wp = ?	Specificity	1.0
Wn = ?, Wp = ?	F1-score	0.8915
Wn = ?, Wp = ?	AUROC	0.9589

b) The sensitivity and specificity values rose to 1, likely indicating that the False negative and False positive values decreased to a negligible amount. This indicates that the weights for both the positive and negative labels decreased greatly.

c) The f1-score didn't change too much and only decreased slightly, and this is understandable considering the value of precision fell by a pretty big margin whereas sensitivity increased but not as much as precision decreased, hence decreasing the overall value of the f1-score.

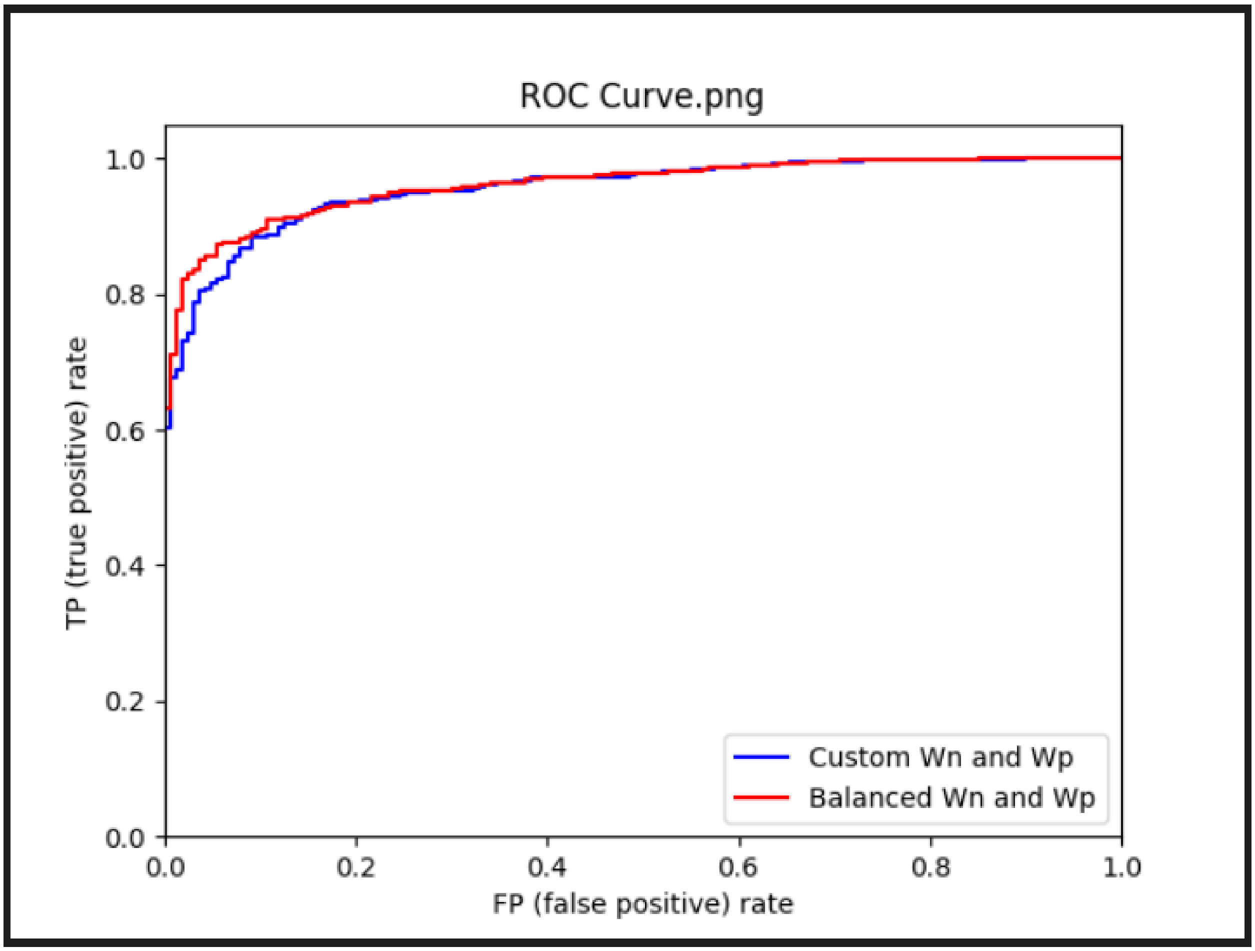
5.3 Choosing Appropriate Class Weights

a) The ROC AUC is sensitive to class imbalance in the sense that when there is a minority class, you typically define this as the positive class and it will have a strong impact on the AUC value. This is very much desirable behavior. Accuracy is for example not sensitive in that way. It can be very high even if the minority class is not well predicted at all. Hence , I chose my metric as "auroc." Also, considering the suggestion that I should use 1 to 2 orders of magnitude, I set the weights from a range of 10e-2 to 10e2. I also found the optimal weight from the array of weights I entered and it was Wn = 100, Wp = 100, which means the function is trying to optimize by a factor of 100.

b)

Class Weights	Performance Measures	Performance
Wn = 100, Wp = 100	Accuracy	0.9159
Wn = 100, Wp = 100	Precision	0.9282
Wn = 100, Wp = 100	Sensitivity	0.9699
Wn = 100, Wp = 100	Specificity	0.7006
Wn = 100, Wp = 100	F1-score	0.9486
Wn = 100, Wp = 100	AUROC	0.9528

5.4 The ROC Curve



6 Challenge

Firstly, I set c = 0.1 because although I didn't want the penalty to be too high, I still wanted the model to be fairly cautious of how many misclassified data points it is ignoring. For the model being trained on multiclass features, I used an SVC rather than a Linear SVC because the documentation says "multiclass support is handled according to a one-vs-the-rest scheme." for Linear SVC, whereas for SVC, it has two options 'ovr' and 'ovo' and the documentation mentions how "one-vs-one ('ovo') is always used as a multi-class strategy to train models; an ovr matrix is only constructed from the ovo matrix," so I realized there wasn't much point using a Linear SVC because constructing an 'ovr' matrix from an 'ovo' matrix just reduces efficiency and the documentation recommends using a one vs one approach anyways which is added under the decision function shape parameter of my SVC function. Furthermore, I read some info on how SVMs in their most simple type don't support multiclass features "natively," and how RBF kernels are the best predictors out of the different kernel types for multiclass classification, so I chose an RBF kernel. I also set the class weight to "balanced" since I think that automatically adjusts weights based on the data it reads through the multiclass and I don't have to worry about one class being significantly more numerical than another.

Code Appendix:

```
"""EECS 445 - Fall 2022.

Project 1
"""

import pandas as pd
import numpy as np
import itertools
import string

from sklearn.svm import SVC, LinearSVC
from sklearn.model_selection import StratifiedKFold
from sklearn import metrics
from matplotlib import pyplot as plt
from stack_data import RangeInLine
```

```python
from helper import *

import warnings
from sklearn.exceptions import ConvergenceWarning

warnings.simplefilter(action="ignore", category=FutureWarning)
warnings.simplefilter(action="ignore", category=ConvergenceWarning)

np.random.seed(445)



def extract_word(input_string):
    """Preprocess review into list of tokens.

    Convert input string to lowercase, replace punctuation with spaces,
and split along whitespace.
    Return the resulting array.

    E.g.
    > extract_word("I love EECS 445. It's my favorite course!")
    > ["i", "love", "eecs", "445", "it", "s", "my", "favorite", "course"]

    Input:
        input_string: text for a single review
    Returns:
        a list of words, extracted and preprocessed according to the
directions
        above.
    """
    # TODO: Implement this function
    output_list = []
    word = ""
    input_string = input_string.lower()
    for i in string.punctuation:
        input_string = input_string.replace(i,' ')
    return input_string.split()
```

```python
def extract_dictionary(df):
    """Map words to index.

    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).

    E.g., with input:
        | text                          | label | ... |
        | It was the best of times.     |  1    | ... |
        | It was the blurst of times.   | -1    | ... |

    The output should be a dictionary of indices ordered by first
occurence in
    the entire dataset:
        {
           it: 0,
           was: 1,
           the: 2,
           best: 3,
           of: 4,
           times: 5,
           blurst: 6
        }
    The index should be autoincrementing, starting at 0.

    Input:
        df: dataframe/output of load_data()
    Returns:
        a dictionary mapping words to an index
    """
    word_dict = {}
    # TODO: Implement this function
    num = 0
    for text in df['text']:
        output_list = extract_word(text)
        for word in output_list:
            if word not in word_dict:
                word_dict[word] = num
                num += 1
```

```
    return word_dict


def generate_feature_matrix(df, word_dict):
    """Create matrix of feature vectors for dataset.

    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 text and labels
        word_dict: dictionary of words mapping to indices
    Returns:
        a numpy 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))
    # TODO: Implement this function
    row = 0
    for text in df['text']:
        output_list = extract_word(text)
        for word in output_list:
            if word in word_dict:
                feature_matrix[row, word_dict[word]] = 1
        row += 1
    return feature_matrix


def performance(y_true, y_pred, metric="accuracy"):
    """Calculate performance metrics.

    Performance metrics are evaluated on the true labels y_true versus the
```

```
    predicted labels y_pred.

    Input:
        y_true: (n,) array containing known labels
        y_pred: (n,) array containing predicted scores
        metric: string specifying the performance metric
(default='accuracy'
                  other options: 'f1-score', 'auroc', 'precision',
'sensitivity',
                  and 'specificity')
    Returns:
        the performance as an np.float64
    """
    # TODO: Implement this function
    # This is an optional but very useful function to implement.
    # See the sklearn.metrics documentation for pointers on how to
implement
    # the requested metrics.
    if metric == "auroc":
        return metrics.roc_auc_score(y_true, y_pred)

    m = metrics.confusion_matrix(y_true, y_pred)
    tn, fp, fn, tp = m.ravel()
    if metric == "accuracy":
        return np.float64((tp+tn)/(tp+fn+fp+tn))
    if metric == "precision":
        return np.float64(tp/(tp+fp))
    if metric == "specificity":
        return np.float64(tn/(tn+fp))
    if metric == "f1-score": # 2*prec*sens/prec+sens
        prec = tp/(tp+fp)
        sens = tp/(tp+fn)
        return np.float64(2*prec*sens/(prec+sens))
    else: #metric == "sensitivity":
        return (np.float64(tp)/(tp+fn))


def cv_performance(clf, X, y, k=5, metric="accuracy"):
    """Split data into k folds and run cross-validation.
```

```
    Splits the data X and the labels y into k-folds and runs k-fold
    cross-validation: for each fold i in 1...k, trains a classifier on
    all the data except the ith fold, and tests on the ith fold.
    Calculates and returns the k-fold cross-validation performance metric
for
    classifier clf by averaging the performance across folds.
    Input:
        clf: an instance of SVC()
        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: an 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')
    Returns:
        average 'test' performance across the k folds as np.float64
    """
    # 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 = []
    strat = StratifiedKFold(n_splits=k)
    #strat.get_n_splits(X,y)
    for train_index, test_index in strat.split(X,y):
        X_train, X_test = X[train_index], X[test_index]
        y_train, y_test = y[train_index], y[test_index]
        clf.fit(X_train, y_train)
        y_pred = clf.predict(X_test)
        if metric == "auroc": #use decision_function in auroc, not predict
            y_pred = clf.decision_function(X_test)
        score = performance(y_test, y_pred, metric)
        #print("SCORE:", score)
        if (score is not None):
            scores.append(score)
```

```
    #print("scores =", scores)
    return np.array(scores).mean()


def select_param_linear(
    X, y, k=5, metric="accuracy", C_range=[], loss="hinge", penalty="l2",
dual=True
):
    """Search for hyperparameters from the given candidates of linear SVM
with
    best k-fold CV performance.

    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
        loss: string specifying the loss function used (default="hinge",
             other option of "squared_hinge")
        penalty: string specifying the penalty type used (default="l2",
             other option of "l1")
        dual: boolean specifying whether to use the dual formulation of
the
             linear SVM (set True for penalty "l2" and False for penalty
"l1"ß)
    Returns:
        the parameter value for a linear-kernel SVM that maximizes the
        average 5-fold CV performance.
    """
    # TODO: Implement this function
```

```
    # HINT: You should be using your cv_performance function here
    # to evaluate the performance of each SVM
    maxc = 0
    maxperf = 0
    for c in C_range:
        clf = LinearSVC(penalty = penalty, loss = loss, dual = True, C =
c, random_state = 445)
        perf = cv_performance(clf, X, y, k = 5, metric = metric)
        print(c, perf)
        if perf > maxperf:
            maxperf = perf
            maxc = c
    return maxc


def plot_weight(X, y, penalty, C_range, loss, dual):
    """Create a plot of the L0 norm learned by a classifier for each C in
C_range.

    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}
        penalty: string for penalty type to be forwarded to the LinearSVC
constructor
        C_range: list of C values to train a classifier on
        loss: string for loss function to be forwarded to the LinearSVC
constructor
        dual: whether to solve the dual or primal optimization problem, to
be
            forwarded to the LinearSVC constructor
    Returns: None
        Saves a plot of the L0 norms to the filesystem.
    """
    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)
```

```
    for c in C_range:
        # clf = LinearSVC(penalty = penalty, loss = loss, dual = True, c =
c, random_state = 445)
        clf = LinearSVC(penalty = penalty, loss = loss, C = c, dual =
dual, random_state = 445)
        clf.fit(X, y)
        L0_norm = 0
        for theta in clf.coef_:
            for c in theta:
                if c != 0:
                    L0_norm += 1
        norm0.append(L0_norm)


    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, metric="accuracy", param_range=[]):
    """Search for hyperparameters from the given candidates of quadratic
SVM
    with best k-fold CV performance.

    Sweeps different settings for the hyperparameters of an
quadratic-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: an 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')
        param_range: a (num_param, 2)-sized array containing the
            parameter values to search over. The first column should
            represent the values for C, and the second column should
            represent the values for r. Each row of this array thus
            represents a pair of parameters to be tried together.
    Returns:
        The parameter values for a quadratic-kernel SVM that maximize
        the average 5-fold CV performance as a pair (C,r)
    """
    # 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...
    best_C_val, best_r_val = 0.0, 0.0
    maxperf = 0
    for c, r in param_range:
        clf = SVC(kernel= "poly", degree=2, C=c, coef0=r, gamma="auto")
        #clf = SVC(kernel = 'poly', degree = 2, C = c,  coef0 = r, gamma =
'auto')
        perf = cv_performance(clf, X, y, k = k, metric = metric)
        print(c, r, perf)
        if perf > maxperf:
            best_C_val = c
            best_r_val = r
            maxperf = perf
    return best_C_val, best_r_val


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(
        fname="data/dataset.csv"
    )
```

```python
    IMB_features, IMB_labels, IMB_test_features, IMB_test_labels =
get_imbalanced_data(
        dictionary_binary, fname="data/dataset.csv"
    )

    #print(extract_word("It's a test sentence. Does it look correct?"))

    # TODO: Questions 3, 4, 5

    #3.b
    print("number of unique words, d, = ", len(X_train[0]))

    #3.c
    print('Avg number of non-zero features = ',
np.sum(X_train)/len(X_train))
    #word appearing in greatest number of comments
    most_common_word = max(dictionary_binary, key = dictionary_binary.get)
    print("Most common word =", most_common_word)
    #4.1b
    print("4.1b
---------------------------------------------------------")
    metrics = ["accuracy", "precision", "sensitivity", "specificty",
"f1-score", "auroc"]
    selected_C = 0
    C_range = [0.001, 0.01, 0.1, 1, 10, 100, 1000]
    for m in metrics:
        maxc = select_param_linear(X_train, Y_train, penalty = "l2",
                                loss = "hinge", metric = m, C_range = C_range)
        clf = LinearSVC(penalty = "l2", loss = "hinge", dual = True, C =
maxc, random_state = 445)
        score = cv_performance(clf, X_train, Y_train, metric = m)
        print("C = ", maxc, "is optimal under", m, "metric, cv_performance
=", score)
        if m == "auroc":
            selected_C = maxc

    #4.1c
    #train performance on X_test and Y_Test
    clf = LinearSVC(penalty = "l2", loss = "hinge", dual = True, C =
selected_C)
```

```python
    clf.fit(X_train, Y_train)
    Y_pred = clf.decision_function(X_test)
    auroc_score = performance(Y_test, Y_pred, metric = 'auroc')
    print("The C that maximizes AUROC is", selected_C)
    print("AUROC score: ", auroc_score)
    Y_pred = clf.predict(X_test)
    for m in metrics:
        if m != "auroc":
            score = performance(Y_test, Y_pred, metric = m)
            print("The", m, "score is", score)

    #4.1d
    plot_weight(X_train, Y_train, penalty = "l2", loss = "hinge", dual =
True, C_range = C_range)

    #4.1e
    #bar coefficient vs each word, most pos and most neg, C = 0.1

print("4.1e------------------------------------------------------------------
---------")
    clf = LinearSVC(C = 0.1)
    clf.fit(X_train, Y_train)
    arg = clf.coef_[0].argsort()
    min_ind5 = arg[:5]
    max_ind5 = arg[:-6:-1]
    minwords = []
    maxwords = []

    for ind in min_ind5:
        for word, index in dictionary_binary.items():
            if index == ind:
                minwords.append(word)
    print("Most negative words")
    for i in range(5): #Return 5 most negative words
        print(clf.coef_[0][min_ind5[i]], minwords[i])


    plt.bar(minwords, clf.coef_[0][min_ind5])
    plt.xlabel("Most Negative Words")
    plt.ylabel("Corresponding Coefficient")
```

```
    plt.title("Bar Chart of most negative words.png")
    plt.savefig("bar Chart of most negative words.png")
    plt.close()


    for ind in max_ind5:
        for word, index in dictionary_binary.items():
            if index == ind:
                maxwords.append(word)
    print("Most positive words")
    for i in range(5): #Return 5 most positive words
        print(clf.coef_[0][max_ind5[i]], maxwords[i])

    #plt.bar(maxwords, max_ind5, color = "blue")
    #plt.xlabel("Most Positive Words")
    #plt.ylabel("Corresponding Coefficient")
    #plt.show()

    plt.bar(maxwords, clf.coef_[0][max_ind5])
    #plt.xscale("Words")
    #plt.legend(["L0-norm"])
    plt.xlabel("Most Positive Words")
    plt.ylabel("Corresponding Coefficient")
    plt.title("Bar Chart of most positive words.png")
    plt.savefig("bar Chart of most positive words.png")
    plt.close()

    #4.2
    #Use squared hinge and l1, dual = false

    #4.2a, reset maxc and maxperf
    maxc = 0
    maxperf = 0
    for c in C_range:
        clf = LinearSVC(penalty = "l1", loss = "squared_hinge", C = c,
dual = False)
        clf.fit(X_train, Y_train)
        y_pred = clf.decision_function(X_test)
        perf = performance(Y_test, Y_pred, "auroc")
        CV_auroc = cv_performance(clf, X_train, Y_train, metric = "auroc")
```

```
        if perf > maxperf:
            maxc = c
            maxperf = perf

print("4.2a--------------------------------------------------------------------------
------------------")
    print("C = ", maxc, "is the optimal solution with an auroc score of",
             perf, "and a CV auroc score of", CV_auroc)

    #4.2b plot weight
    plot_weight(X_train, Y_train, penalty = "l1", loss = "squared_hinge",
                C_range = C_range, dual = False)

    #4.3a
    #(i) Grid Search
    r_range = [0.001, 0.01, 0.1, 1, 10, 100, 1000]
    cr_range = []
    for c in C_range:
        for r in r_range:
            cr_range.append([c,r])
    [maxc, maxr] = select_param_quadratic(X_train, Y_train, param_range =
cr_range)

    clf1 = SVC(kernel="poly", degree=2, C=c, coef0=r, gamma="auto")
    auroc_score_grid = cv_performance(clf1, X_train, Y_train, metric =
"auroc")

    print("4.3a (i) Grid
Search--------------------------------------------------")
    print("C = ", maxc, "r = ", maxr,
         "is part of the optimal solution with an auroc score of",
auroc_score_grid)

    #(ii) Random Search
    import random
    cr_range = []
    for i in range(25):
        #range of values (-10e2, 10e3) --> 36 pairs
        lgc = random.uniform(-2, 3)
        lgr = random.uniform(-2, 3)
```

```
        cr_range.append([10**lgc, 10**lgr])
    [maxc, maxr] = select_param_quadratic(X_train, Y_train, param_range =
cr_range)

    clf2 = SVC(kernel="poly", degree=2, C=c, coef0=r, gamma="auto")
    auroc_score_rand = cv_performance(clf2, X_train, Y_train, metric =
"auroc")

    print("4.3a (ii) Random
Search--------------------------------------------------")
    print("C = ", maxc, "r = ", maxr, "is the optimal solution with an
auroc score of",
                auroc_score_rand)


    #5.1c

print("5.1c------------------------------------------------------------")
    clf = LinearSVC(penalty = "l2", loss = "hinge", C = 0.01, class_weight
= {-1: 1, 1: 10})
    clf.fit(X_train, Y_train)
    Y_pred = clf.decision_function(X_test)
    perf = performance(Y_test, Y_pred, metric = "auroc")
    print("AUROC score: ", perf)
    Y_pred = clf.predict(X_test)
    for m in metrics:
        if m != "auroc":
            perf = performance(Y_test, Y_pred, metric = m)
            print("Score: ", perf)

    #5.2a
    print("5.2a--------------------------------------------------------")
    clf = LinearSVC(penalty = "l2", loss = "hinge", C = 0.01, class_weight
= {-1: 1, 1 : 1})
    clf.fit(IMB_features, IMB_labels)
    Y_pred = clf.decision_function(IMB_test_features)
    perf = performance(IMB_test_labels, Y_pred, metric = "auroc")
    print("AUROC score: ", perf)
    Y_pred = clf.predict(IMB_test_features)
    #print("Y_pred:", np.shape(Y_pred))
```

```
    #print("IMB:", np.shape(IMB_test_labels))
    for m in metrics:
        if m != "auroc":
            perf = performance(IMB_test_labels, Y_pred, metric = m)
        print("Score: ", perf)


    #5.3a
    W_range = [-2, -1, 0, 1, 2]
    W_range = [10**w for w in W_range]
    maxWn = 0
    maxWp = 0
    maxperf = 0
    for Wn in W_range:
        for Wp in W_range:
            clf = SVC(C = 1, class_weight = {-1:Wn, 1:Wp})
            perf = cv_performance(clf, IMB_features, IMB_labels, metric =
"auroc")
            if perf > maxperf:
                maxperf = perf
                maxWn = Wn
                maxWp = Wp

print("5.3a--------------------------------------------------------------------------
-------")
    print("Wn =", Wn, "is optimal and Wp =", Wp, "is optimal and
performance = ", maxperf)

    #based on values from 5.3a, the most optimal Wn value = 100,
    # the most optimal Wp value = 100 and performance = 0.9528
    clf = SVC(C=1, class_weight = {-1:100, 1:100})
    perf = cv_performance(clf, IMB_features, IMB_labels, metric = "auroc")

    #5.3b
    metrics = ["accuracy", "precision", "sensitivity", "specificity",
"f1-score", "auroc"]

print("5.3b--------------------------------------------------------------------------
")
    print("the auroc score:", maxperf) #maxperf comes from 5.3a
```

```
    clf = SVC(C = 1, class_weight = {-1:maxWn, 1:maxWp})
    clf.fit(IMB_features, IMB_labels)
    y_pred = clf.predict(IMB_test_features)
    for m in metrics:
        if m != "auroc":
            perf = performance(IMB_test_labels, y_pred, metric = m)
            print("Metric", m, " =", perf)



    #5.4
    print("5.4---------------------------------------")
    clf = SVC(C=1, class_weight = {-1:100, 1:100})
    clf.fit(IMB_features, IMB_labels)
    y_pred = clf.decision_function(IMB_test_features)
    fpr, tpr, threshhold1 = metrics.roc_curve(IMB_test_labels, y_pred)
    perf = cv_performance(clf, IMB_features, IMB_labels, metric = "auroc")

    ROCclf = SVC(C = 0.01, class_weight = {-1:1, 1:1})
    ROCclf.fit(IMB_features, IMB_labels)

    y_pred2 = ROCclf.decision_function(IMB_test_features)
    fpr2, tpr2, threshold2 = metrics.roc_curve(IMB_test_labels, y_pred2)
    perf2 = performance(IMB_test_labels, y_pred2, metric = "auroc")

    plt.figure()
    plt.plot(fpr, tpr, color = "blue", label = "Custom Wn and Wp" % perf)
    plt.plot(fpr2, tpr2, color = "red", label = "Balanced Wn and Wp" %
perf2)
    plt.xlim([0.0, 1.0])
    plt.ylim([0.0, 1.05])
    plt.legend(loc = "lower right")
    plt.xlabel("FP (false positive) rate")
    plt.ylabel("TP (true positive) rate")
    plt.title("ROC Curve.png")
    plt.savefig("ROC Curve.png")
    plt.close()



    # Read multiclass data
```

```
    # TODO: Question 6: 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)

    #decision_function_shape = "ovo"
    clf = SVC(C = 0.1, kernel = "rbf", gamma = "auto", class_weight =
"balanced",
                decision_function_shape = 'ovo')
    clf.fit(multiclass_features, multiclass_labels)
    y_pred = clf.predict(heldout_features)
    generate_challenge_labels(y_pred, "cnrusimh")


if __name__ == "__main__":
    main()
```