2.a.i)
It is possible to identify individuals, either directly or indirectly from the dataset. First, we would align each datapoint using the timestamp so that the datapoint will be in the order in which the timestamp increases. Then, we could identify instances of a conversation from our datapoint. The most usual conversation I was able to identify in the dataset was person a stating a problem, then person b answering it, and then person a thanking person b. In this case, we would be able to identify person a.

2.a.ii)
The task that the dataset could be used for is to measure politeness on the internet. In the dataset, there were many instances of “thank you” and “you’re welcome”. Therefore, I think we could use this dataset to detect politeness on the web.

2.a.iii)
A task for which the dataset should not be used is to collect opinion on a certain topic. In this dataset, there are a variety of topics that are brought up. Therefore, there wouldn’t be enough data when trying to collect opinions of people on a specific topic.

3.b) `['best', 'book', 'ever', 'it', 's', 'great']`

3.b) 4855

3.c.i) `12.241140215716486`

3.c.ii) `i`

4.1.a)
It might be beneficial to maintain class proportions across fold so that a training set represents the test set. The training data set must represent the test set for stratified splits to be run successfully.

4.1.b)

```
Metric: accuracy
Best c: 1
CV Score: 0.9210263820957463

Metric: f1-score
Best c: 1
CV Score: 0.9202149609972474

Metric: auroc
Best c: 0.1
CV Score: 0.9675533001302232

Metric: precision
Best c: 0.01
CV Score: 0.9300177176503188

Metric: sensitivity
Best c: 1
CV Score: 0.9106326106326106

Metric: specificity
Best c: 0.01
CV Score: 0.9314166914166913
```

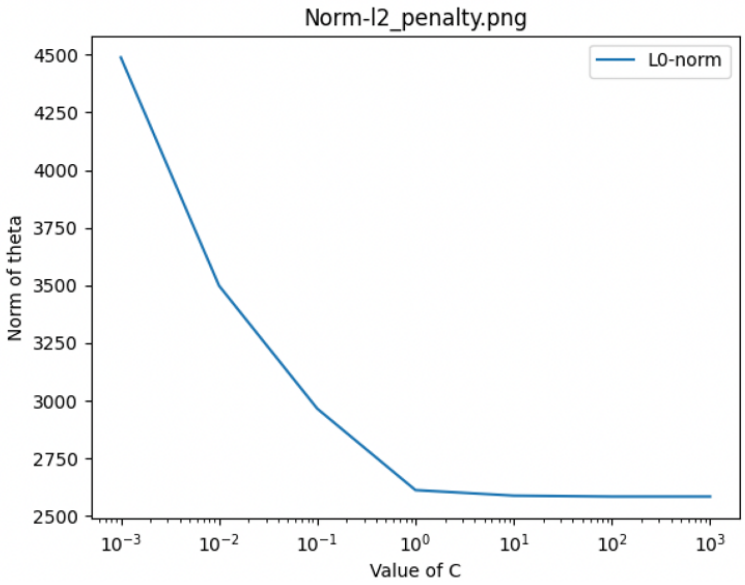
For accuracy, f1-score, auroc, precision and specificity, performance increases as C increases until C reaches the optimal value. From then on, the performance decreases.
For sensitivity, performance decreases as C increases. The possible optimal value for C for sensitivity would be a smaller value of C than the range that we searched for.

The performance measure that I think is most valid is accuracy. Accuracy is usually used when the datasets are balanced. Because it mentions in the spec (4.1.a) that class proportions should be roughly equal across the folds since the original training data has equal class proportions, we know the proportion of positive and non-positive points in our dataset are balanced.

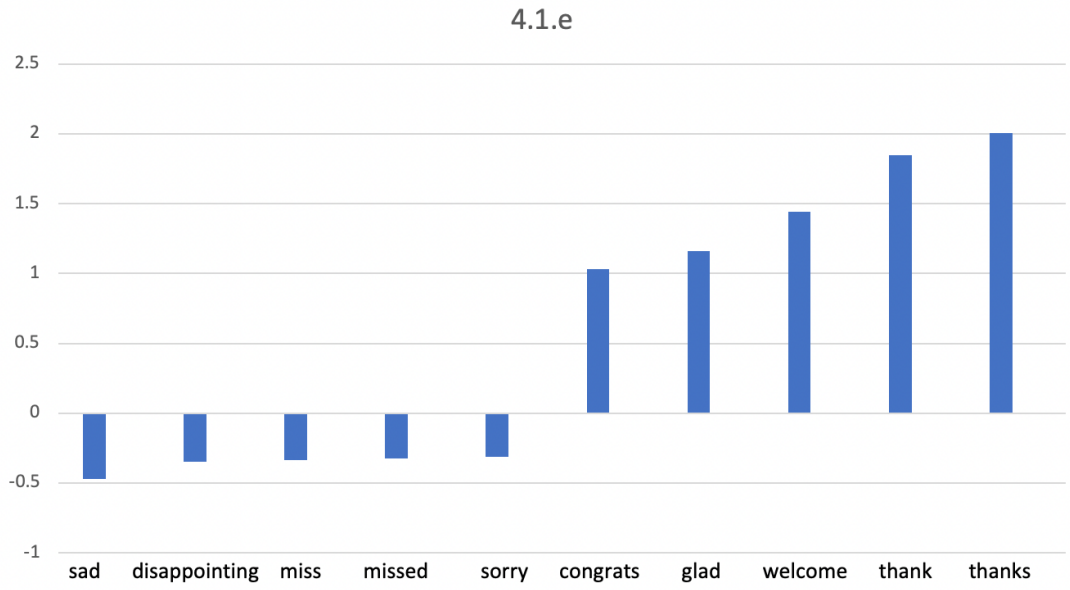
4.1.c)
From 4.1.b, “accuracy”, was maximized by the value of C = 1.

```
Accuracy: 0.9199384141647421
F1-score: 0.9204892966360856
Auroc: 0.9734289439374185
Precision: 0.9148936170212766
Sensitivity: 0.9261538461538461
Specificity: 0.9137134052388289
```

4.1.d)



4.1.e)

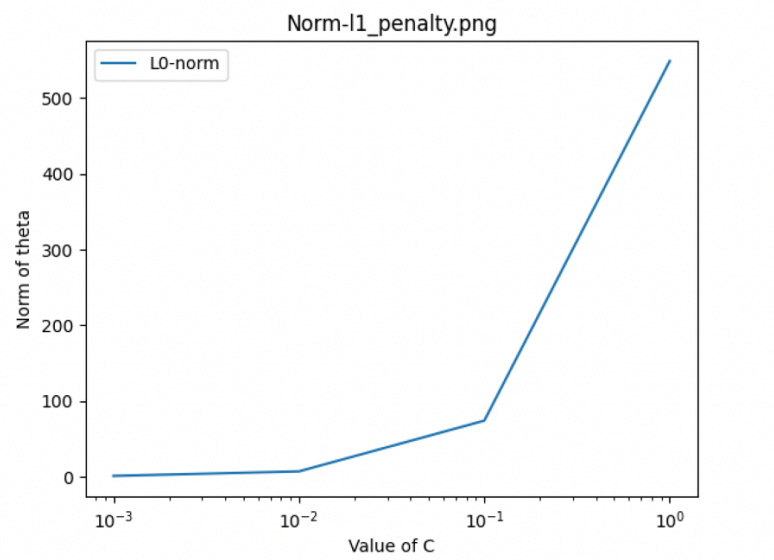


4.1.f)
Despite the welcome messages from my new neighbors, I am not glad to be far away from home and family thanks to the Covid 19 pandemic.

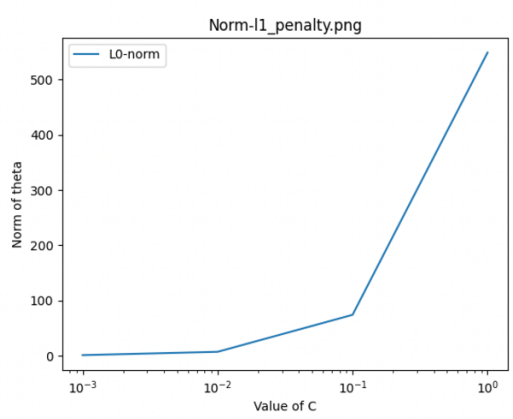
4.2.a)

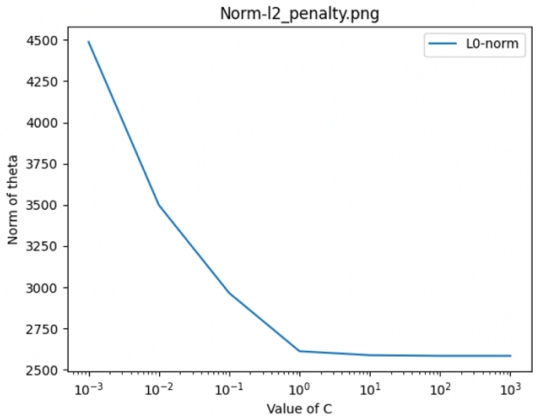
```
C value: 0.1
mean CV AUROC score: 0.9167963539350822
AUROC score on test set: 0.9214701514701515
```

4.2.b)



4.2.c)





We know that the gradient of the L1 norm is constant. This means that it will push the coefficients of the less important features to exactly zero. This will produce a sparse graph as we can see above. On the other hand, we know that the gradient of the L2 norm is dependent on theta. This results in balancing the weights of all features so that it won't have expensive large values.

4.2.d)

Compared to hinge loss, squared hinge loss allows data points that are correctly classified but is placed within the margin while severely punishing misclassified data points. Therefore, when we use squared hinge loss, we see a wider margin and more support vectors than when using hinge loss.

4.3.a)

```
Quadratic SVM with grid search and auroc metric:
Best c: 1
Best coeff: 100
Test performance: 0.9700957754231456

Quadratic SVM with random search and auroc metric:
Best c: 28.351807224575275
Best coeff: 4.316816720594751
Test performance: 0.9709827989541765
```

4.3.b)

For grid search, performance tends to increase as c and r increase. For random search, the performance tends to be inconsistent from a range performance value for different c and r values. Grid search works better than random search for comparing algorithms and models. On the other hand, random search is more computationally efficient than grid search.

4.4.a)

$$\phi(x) = \left[r,\ \sqrt{2r}x_1,\ \cdots\ \sqrt{2r}x_n,\ \sqrt{2}x_1x_2\ \ \sqrt{2}x_1x_3\ \cdots\ \sqrt{2}x_1x_n\ \sqrt{2}x_2x_3\ \cdots\ \sqrt{2}x_{n-1}x_n,\ x_1^2 \cdots x_n^2 \right]$$

4.4.b)

The pros of using explicit feature mapping over a kernel is that we can access feature weights for each feature which provides a better interpretation of the parameters.

The cons of using explicit feature mapping over a kernel is that it is computationally expensive and less efficient in most cases.

5.1.a)
If Wn is much greater than Wp, this means that in terms of classifying positive and negative points, we allow more misclassified positive points while we care more about correctly classifying negative points.

5.1.b)
When we set Wn = 0.25 and Wp = 1, Wn decreases the C value for negative data points which allows more misclassified negative points while Wp doesn't affect C so classification of positive data points remain the same. When we set Wn = 1 and Wp = 4, on the other hand, while the classification of negative data points remain the same as Wn doesn't affect C, Wp = 4 cares more about correctly classifying positive points.

5.1.c)
```
Accuracy: 0.541966141966142
F1-score: 0.6857594619749185
Auroc: 0.9600983441126555
Precision: 0.5222573267064308
Sensitivity: 0.9984615384615385
Specificity: 0.08478234943351222
```

5.1.d)
The performance measures that were affected the most by the new class weights are sensitivity and specificity. When we set Wn = 1 and Wp = 10, our classifier penalizes misclassifications of positive data points more than negative data points.

5.2.a)
```
Accuracy: 0.7995152616829508
F1-score: 0.8885875917527702
Auroc: 0.9125830419580419
Precision: 0.7995152616829508
Sensitivity: 1.0
Specificity: 0.0
```

5.2.b)
By training on imbalanced data, the performance metrics that were affected the most are sensitivity and specificity. Sensitivity increased significantly while specificity decreased significantly. Because we changed the data set to consist more positive data points than negative data points, the classifier trained on this dataset tend to classify positive samples correctly. Sensitivity is 1.0 which clearly shows that almost all positive data points were classified correctly. Specificity is 0.0 which means that almost none of the negative data points were correctly classified.

5.2.c)
The precision score matches my intuition. The precision score is calculated using the ratio of true positives over total predicted positives (TP / TP + FP). The precision score being 0.7995 means while all positive data points were correctly classified, almost all negative data points were misclassified.

In addition, the F1-score matches my intuition as well because the F1-score considers both precision and sensitivity. When considering both precision and sensitivity scores from above, the F1-score seems appropriate.

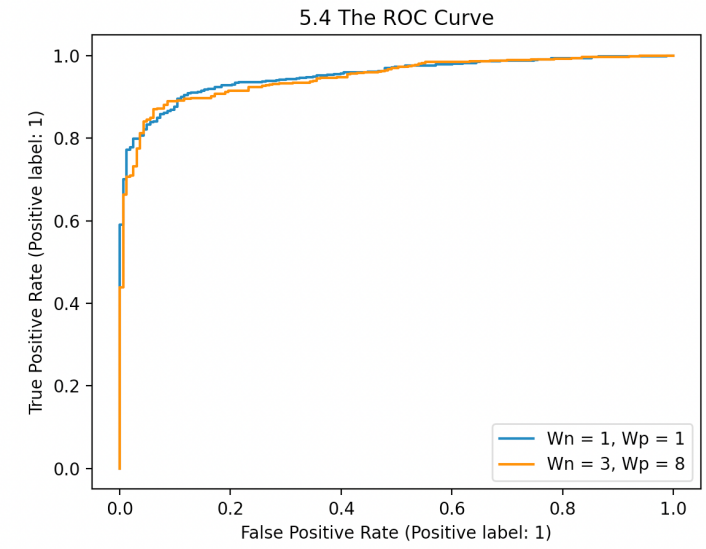
5.3.a)
An appropriate performance metric to use would be the F1-score. With an imbalanced data set, F1-score considers both precision and sensitivity, which means that it takes into account correctly classified positive data points and misclassified negative data points.

After using cross validation across a range of Wn = 2 to 8, Wp = 2 to 8, I found weights Wn = 3, Wp = 8 yielded the highest score of `0.9215497903430023`

5.3.b)

```
Class_weight={-1: 3, 1: 8}
Test Performance on metric on accuracy: 0.8241082410824109
Test Performance on metric on f1-score: 0.9006254343293955
Test Performance on metric on auroc: 0.9465219443133552
Test Performance on metric on precision: 0.8212927756653993
Test Performance on metric on sensitivity: 0.9969230769230769
Test Performance on metric on specificity: 0.13496932515337423
```

5.4)



6.1)
After reading about one-vs-one and one-vs-all multiclass method, I chose to use the one-vs-rest multiclass method. Because we are given only 3 classes in the data set, “Gratitude”, “Neutral”, “Sadness”, using one-vs-all multiclass method will not be computationally expensive. Because one-vs-all method creates one model for each class, it will create only 3 models in our case.

Then I used the select_param_linear function to retain the optimal value of c. As for the parameters of select_param_linear, I chose “accuracy” as the metric because the spec mentioned that our classifier will be evaluated on accuracy. For the value of k, after trying different values of k from 2 to 10, k = 9 yielded the highest cv performance score of `0.7426666666666667`. Lastly, I used the same C_range of

```
C_range = [ 10**(-3), 10**(-2), 10**(-1), 10**(0), 10**(1), 10**(2), 10**(3)]
```

To extract features, my first intuition was to include punctuation such as “!”, “?”, or “…”. When looking through the dataset, I thought punctuation could be useful in determining labels. However, after I tried including some punctuations listed above, the cv performance on the training dataset did not change or even decreased. When not replacing “!”, with a space, I got `0.7311111111111112` cv performance score and when not replacing “!” and “?” with a space, I got `0.7293333333333334` cv performance score. However, when I used the original extract_word function, I got `0.7426666666666667` cv performance score. Therefore, I didn’t make any adjustments to the original extract_word function.

After using these methods, I got `0.7426666666666667` cv performance score.

```
"""EECS 445 - Winter 2022.

Project 1
"""

# from msilib.schema import Binary
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 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.
    """
    #convert to lowercase
    input = input_string.lower()
```

```
    #replace punctuation with space
    for i in input:
        if i in string.punctuation:
            input = input.replace(i, " ")

    return input.split()

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 = {}
    index = 0

    for i in df['text']:
        for j in extract_word(i):
            if (j not in word_dict):
                word_dict[j] = index
                index = index + 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))

    index = 0
    for i in df['text']:
        for j in extract_word(i):
            if (j in word_dict):
                feature_matrix[index][word_dict[j]] = 1
        index += 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 == "accuracy"):
        return metrics.accuracy_score(y_true, y_pred)
    elif (metric == "f1-score"):
        return metrics.f1_score(y_true, y_pred)
    elif (metric == "auroc"):
        return metrics.roc_auc_score(y_true, y_pred)
    elif (metric == "precision"):
        return metrics.precision_score(y_true, y_pred)
    else:
        tn, fp, fn, tp = metrics.confusion_matrix(y_true, y_pred).ravel()
        if (metric == "sensitivity"):
            return (tp / (tp+fn))
        else:
            return (tn / (tn+fp))

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

    scores = []

    skf = StratifiedKFold(n_splits=k, shuffle=False)
    for train_index, test_index in skf.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)
        if (metric == "auroc"):
```

```
            y_pred = clf.decision_function(X_test)
        else:
            y_pred = clf.predict(X_test)
        # Put the performance of the model on each fold in the scores array
        scores.append(performance(y_test, y_pred, metric))
    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 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
    maxPerformance = 0
    index = 0
    for i in C_range:
        clf = LinearSVC(C = i, random_state = 445, loss='hinge', penalty='l2')
        temp = cv_performance(clf, X, y, k, metric)
        if (maxPerformance < temp):
            index = i
            maxPerformance = temp
    return index

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: penalty to be forwarded to the LinearSVC constructor
        C_range: list of C values to train a classifier on
        loss: 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(C = c, random_state = 445, loss=loss, penalty=penalty, dual =
dual)
        clf.fit(X, y)
        norm0.append(np.linalg.norm(clf.coef_[0], ord = 0, axis = 0))

    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 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
    maxPerformance = 0
    for c, r in param_range:
        clf = SVC(kernel='poly', random_state = 445, degree=2, C=c, coef0=r,
gamma='auto')
        temp = cv_performance(clf, X, y, k, metric)
        if (maxPerformance < temp):
            best_C_val = c
            best_r_val = r
            maxPerformance = temp

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

    # TODO: Questions 3, 4, 5

    # 3.a
    print(extract_word("'BEST book ever! It\'s great'"))

    # 3.b
    print(X_train.shape)

    # 3.c.i
    count_list = []
    for i in range(X_train.shape[0]):
        count = 0
        for j in range(X_train.shape[1]):
```

```
            if X_train[i][j] != 0:
                count += 1
        count_list.append(count)
    print(sum(count_list) / len(count_list))

    # # 3.c.ii
    word_list = np.zeros((1, X_train.shape[1]))
    for i in range(X_train.shape[0]):
        for j in range(X_train.shape[1]):
            word_list[0][j] += X_train[i][j]

    for key, val in dictionary_binary.items():
        if val == np.argmax(word_list):
            print(key)

    # 4.1.b CORRECT
    C_range = [ 10**(-3), 10**(-2), 10**(-1), 10**(0), 10**(1), 10**(2), 10**(3)]
    c = select_param_linear(X_train, Y_train, 5, "accuracy", C_range)
    clf = LinearSVC(C = c, random_state = 445, loss='hinge', penalty='l2')
    performance = cv_performance(clf, X_train, Y_train, k=5, metric="accuracy")
    print("\nMetric: accuracy", "\nBest c:", c, "\nCV Score:", performance)

    c = select_param_linear(X_train, Y_train, 5, "f1-score", C_range)
    performance = cv_performance(clf, X_train, Y_train, k=5, metric="f1-score")
    print("\nMetric: f1-score", "\nBest c:", c, "\nCV Score:", performance)

    c = select_param_linear(X_train, Y_train, 5, "auroc", C_range)
    performance = cv_performance(clf, X_train, Y_train, k=5, metric="auroc")
    print("\nMetric: auroc", "\nBest c:", c, "\nCV Score:", performance)

    c = select_param_linear(X_train, Y_train, 5, "precision", C_range)
    performance = cv_performance(clf, X_train, Y_train, k=5, metric="precision")
    print("\nMetric: precision", "\nBest c:", c, "\nCV Score:", performance)

    c = select_param_linear(X_train, Y_train, 5, "sensitivity", C_range)
    performance = cv_performance(clf, X_train, Y_train, k=5, metric="sensitivity")
    print("\nMetric: sensitivity", "\nBest c:", c, "\nCV Score:", performance)

    c = select_param_linear(X_train, Y_train, 5, "specificity", C_range)
    performance = cv_performance(clf, X_train, Y_train, k=5, metric="specificity")
    print("\nMetric: specificity", "\nBest c:", c, "\nCV Score:", performance)

    # 4.1.c performance
    clf = LinearSVC(C = 1, random_state = 445, loss='hinge', penalty='l2')
    clf.fit(X_train, Y_train)
    y_pred = clf.predict(X_test)
    print("Accuracy:", performance(Y_test, y_pred, "accuracy"))
    print("F1-score:", performance(Y_test, y_pred, "f1-score"))
```

```
    print("Auroc:", performance(Y_test, clf.decision_function(X_test), "auroc"))
    print("Precision:", performance(Y_test, y_pred, "precision"))
    print("Sensitivity:", performance(Y_test, y_pred, "sensitivity"))
    print("Specificity:", performance(Y_test, y_pred, "specificity"))

    # 4.1.d CORRECT
    plot_weight(X_train, Y_train, "l2", C_range, "hinge", True)

    # 4.1.e CORRECT
    clf = LinearSVC(C = 0.1, random_state = 445, loss='hinge', penalty='l2')
    clf.fit(X_train, Y_train)

    max_ind = clf.coef_[0].argsort()[-5:]
    for key, val in dictionary_binary.items():
        if val in max_ind:
            print(key, clf.coef_[0][val])

    min_ind = clf.coef_[0].argsort()[:5]
    for key, val in dictionary_binary.items():
        if val in min_ind:
            print(key, clf.coef_[0][val])

    # 4.2.a CORRECT
    C_range = [ 10**(-3), 10**(-2), 10**(-1), 10**(0)]
    c = select_param_linear(X_train, Y_train, 5, "auroc", C_range, loss =
'squared_hinge', penalty = 'l1', dual = False)
    clf = LinearSVC(C = c, random_state = 445, penalty = 'l1', loss = 'squared_hinge',
dual = False)
    mean_score = cv_performance(clf, X_train, Y_train, k=5, metric="accuracy")
    best_score = cv_performance(clf, X_test, Y_test, k=5, metric="accuracy")
    print("\nC value:", c, "\nmean CV AUROC score:", mean_score, "\nAUROC score on
test set:", best_score)

    plot_weight(X_train, Y_train, 'l1', C_range, 'squared_hinge', False)

    # 4.3.a CORRECT
    c_range = [10**(-2), 10**(-1), 10**(0), 10**(1), 10**(2), 10**(3)]
    r_range = [10**(-2), 10**(-1), 10**(0), 10**(1), 10**(2), 10**(3)]
    param = []
    for i in c_range:
        for j in r_range:
            param.append([i, j])

    c, r = select_param_quadratic(X_train, Y_train, 5, "auroc", param)
    clf = SVC(kernel='poly', random_state = 445, degree=2, C=c, coef0=r, gamma='auto')
    performance = cv_performance(clf, X_test, Y_test, k=5, metric="auroc")
    print("\nQuadratic SVM with grid search and auroc metric:", "\nBest c:", c,
"\nBest coeff:", r, "\nTest performance:", performance)
```

```
    # 4.3.b CORRECT
    c_random = np.random.uniform(-2,3,25)
    c_range = 10**c_random
    r_random = np.random.uniform(-2,3,25)
    r_range = 10**r_random
    param = np.vstack((c_range, r_range)).T

    c, r = select_param_quadratic(X_train, Y_train, 5, "auroc", param)
    clf = SVC(kernel='poly', random_state = 445, degree=2, C=c, coef0=r, gamma='auto')
    performance = cv_performance(clf, X_test, Y_test, k=5, metric="auroc")
    print("\nQuadratic SVM with random search and auroc metric:", "\nBest c:", c,
"\nBest coeff:", r, "\nTest performance:", performance)

    # 5.1.c CORRECT
    clf = LinearSVC(C = 0.01, random_state = 445, loss = 'hinge', penalty = 'l2',
class_weight = {-1: 1, 1: 10})
    clf.fit(X_train, Y_train)
    y_pred = clf.predict(X_test)
    print("Accuracy:", performance(Y_test, y_pred, "accuracy"))
    print("F1-score:", performance(Y_test, y_pred, "f1-score"))
    print("Auroc:", performance(Y_test, clf.decision_function(X_test), "auroc"))
    print("Precision:", performance(Y_test, y_pred, "precision"))
    print("Sensitivity:", performance(Y_test, y_pred, "sensitivity"))
    print("Specificity:", performance(Y_test, y_pred, "specificity"))

    # 5.2.a CORRECT
    clf = LinearSVC(C = 0.01, random_state = 445, loss = 'hinge', penalty = 'l2',
class_weight = {-1: 1, 1: 1})
    clf.fit(IMB_features, IMB_labels)
    y_pred = clf.predict(IMB_test_features)
    print("Accuracy:", performance(IMB_test_labels, y_pred, "accuracy"))
    print("F1-score:", performance(IMB_test_labels, y_pred, "f1-score"))
    print("Auroc:", performance(IMB_test_labels,
clf.decision_function(IMB_test_features), "auroc"))
    print("Precision:", performance(IMB_test_labels, y_pred, "precision"))
    print("Sensitivity:", performance(IMB_test_labels, y_pred, "sensitivity"))
    print("Specificity:", performance(IMB_test_labels, y_pred, "specificity"))

    # 5.3.a
    clf = LinearSVC(C = 0.01, random_state = 445, loss = 'hinge', penalty = 'l2',
class_weight = {-1: 3, 1: 8})
    clf.fit(IMB_features, IMB_labels)
    y_pred = clf.predict(IMB_test_features)
    print("Class_weight={-1: 3, 1: 8}")
    print("Test Performance on metric on accuracy:", performance(IMB_test_labels,
y_pred, "accuracy"))
```

```
    print("Test Performance on metric on f1-score:", performance(IMB_test_labels,
y_pred, "f1-score"))
    print("Test Performance on metric on auroc:", performance(IMB_test_labels,
clf.decision_function(IMB_test_features), "auroc"))
    print("Test Performance on metric on precision:", performance(IMB_test_labels,
y_pred, "precision"))
    print("Test Performance on metric on sensitivity:", performance(IMB_test_labels,
y_pred, "sensitivity"))
    print("Test Performance on metric on specificity:", performance(IMB_test_labels,
y_pred, "specificity"))

    # 5.4
    clf_11 = LinearSVC(C = 0.01, random_state = 445, loss = 'hinge', penalty = 'l2',
class_weight = {-1: 1, 1: 1})
    clf_11.fit(IMB_features, IMB_labels)

    clf_38 = LinearSVC(C = 0.01, random_state = 445, loss = 'hinge', penalty = 'l2',
class_weight = {-1: 3, 1: 8})
    clf_38.fit(IMB_features, IMB_labels)

    fig = metrics.plot_roc_curve( clf_11, IMB_test_features, IMB_test_labels, label =
"Wn = 1, Wp = 1")
    fig = metrics.plot_roc_curve( clf_38, IMB_test_features, IMB_test_labels, ax =
fig.ax_, label = "Wn = 3, Wp = 8")

    plt.title('5.4 The ROC Curve')
    plt.show()

    # 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)

    C_range = [10**(-3), 10**(-2), 10**(-1), 10**(0), 10**(1), 10**(2), 10**(3)]
    c = select_param_linear(multiclass_features, multiclass_labels, 9, "accuracy",
C_range)
    clf = LinearSVC(C = c, random_state = 445, multi_class = 'ovr')
    clf.fit(multiclass_features, multiclass_labels)
    y_pred = clf.predict(heldout_features)
    generate_challenge_labels(y_pred, "stae")
    print(cv_performance(clf, multiclass_features, multiclass_labels, k=9,
metric="accuracy"))
```

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