EECS 445 – Project 1: Support Vector Machines

Author: Dino Mastropietro
February 12, 2021

Abstract

In this report, I discuss the Support Vector Machine (SVM) models I have implemented help our fictional character Naitian choose the rights books to take with him on his trip to the Antarctic. Using data from Amazon's large catalogue of book reviews, I have trained various types of SVMs to determine the books worth reading. There are four main sections to this report: Feature Extraction, Hyperparameter and Model Selection, Asymmetric Cost Functions and Class Imbalance and the Challenge. I use Python 3.9.1, scikit-learn 0.24, numpy 1.19, pandas 1.2.1 and matplotlib 3.3.3. All code can be found in appendix A.

Feature Extraction

To extract the features which would determine the dimensions of our set of training data, I used a dictionary to store the key-value pair of unique words and that word's index (ordered by when it was found). These values were read in from only the reviews themselves and I removed any punctuation within them. I found there to be 7508 unique words in the data set; the word "the" appeared the most, 3894 times.

To then test my model, I created feature matrices for each test review. They had a length equal to the number of words in the dictionary and had a value of 1 or 0 in to denote whether the word in that index appeared in this review.

Hyperparameter and Model Selection

Using the data I trained on in the previous section, I separate the reviews with binary labels, positive and negative (1 and -1, respectively). I use two different SVMs, linear and quadratic, to fit this data. I further consider 5 different performance metrics outlined below in Table 1 and, using a 5-fold cross-validation (CV) choose a value of 'C' that gives the best performance score. It is important to keep classification proportions across all subsets of the training data while splitting into k-folds so that each group is as generalized to the full group of data well as possible; it allows more consistency among post-processing statistical tests.

Performance Measures	C	Performance Score
Accuracy	0.0100	0.817000000
F1-Score	0.0100	0.816003714
AUROC	0.1000	0.904740000
Precision	0.1000	0.825260644
Sensitivity	0.0001	0.890000000
Specificity	0.1000	0.828000000

Table 1: 3.1.d The Best Setting for C for Each Performance Measure

It is of note that for higher values of C, the 5-fold variation will take longer to run on non-linearly separable data. This is precisely because a high value of C indicates that there is more penalty for misclassification, and this results in the soft-margin SVM becoming increasingly like a hard-margin SVM. Based on the data, I want to optimize for our AUROC metric. AUROC stands for Area Under the Receiver Operating Characteristic, this is a metric for discrimination. For unbalanced data, AUROC is more informative but can be over-optimistic about data sets with a larger number of negative examples. Our data do not happen to be that case. Here, our data are balanced. Table 2, below, shows the performance of each SVM given we use C = .1, the optimized value for the AUROC model.

Performance Measures	Performance Score
Accuracy	0.816000000
F1-Score	0.813309153
AUROC	0.904740000
Precision	0.825260644
Sensitivity	0.803999999
Specificity	0.828000000

Table 2: Performance of SVM given AUROC optimized C = 0.1

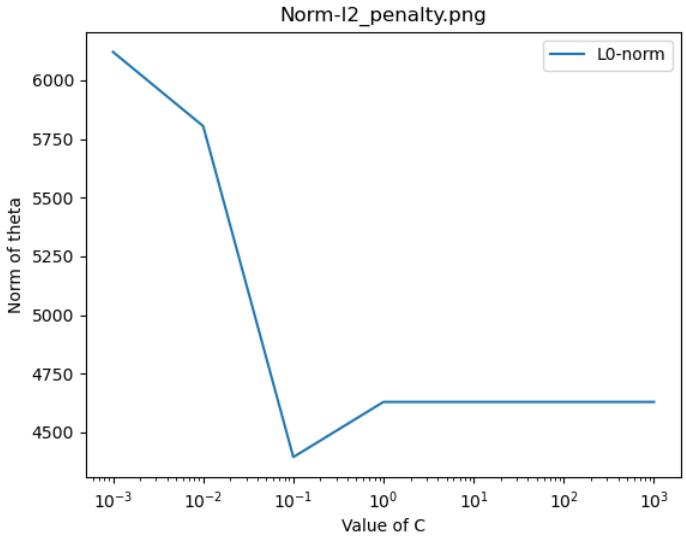

Figure 1: L0-norm of our parameter vector (theta) plotted against C

Another example of greater values of C turning our soft-margin SVM into a hard-margin SVM is on display in Figure 1 above, mainly since we note that the norm of the parameter vector converges to a single value for C values above C = 1. This means our model complexity converges. It is also of interest to know the ten most positive and negative words that our SVM has learned. This plot can be seen in Figure 2 on the next page. From this data, we can see how reviews can sometimes be extremely difficult for a classifier to judge correctly. For example, in this sentence, "I *highly* doubt that anyone has *highly enjoyed* reading this *hot* mess of a book," I use for words which the SVM categorizes as positive words yet the sentence is obviously a negative one.

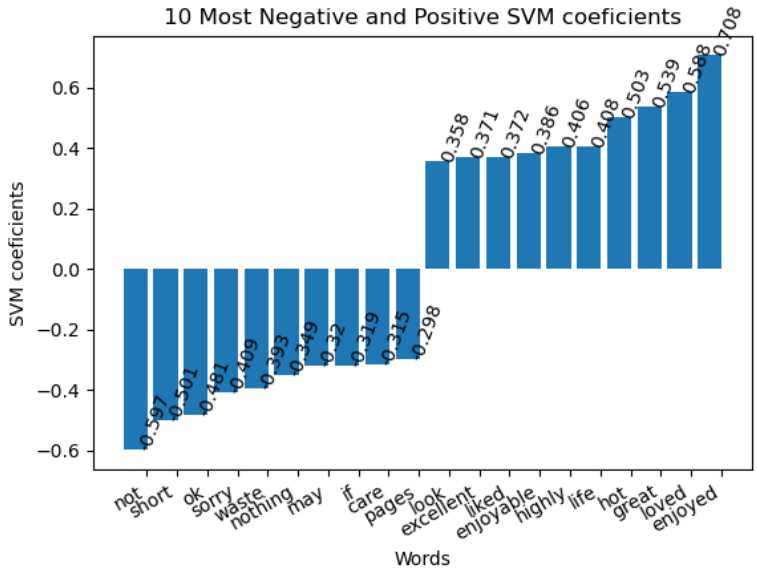

Figure 2: Ten Most Negative and Positive Words according to trained SVM coefficients

For the hyperparameter selection of a quadratic kernel I used both a grid-search and a log of a random variable uniformly distributed over .001 and 1000. Again, we notice that the 5-fold CV performance is hurt with higher values of C since our soft-margin SVM is becoming stricter with the penalties it incurs. The 'r' value does not affect the performance of 5-fold CV since it is only a scalar by which the hyper-plane is shifted. We also note that grid search does a very good job of getting the breath of values in this range. It is exhaustive to the granularity of our choosing, which albeit is not very fine here. Nonetheless it out-performs the random search in this case since random search does a very good job a working more "fine grained" values it does not necessarily test any specific range of values and indiscriminately jumps around.

Tuning Scheme	**C**	**R**	**AUROC**
Grid Search	10.0	10.0	0.9060799
Random Search	2.298928	1.134497	0.9011200

Table 3: Best C & r values for AUROC after Grid & Random Search

Imagine we had a feature mapping, $\phi(\bar{x})$, that maps from the data to the same feature space as the one implied here and has a kernel of the form:$(\bar{x} \cdot \bar{x}' + r)^2$. Then doing the reverse math $\phi(\bar{x}) = [x_1^2\ ,\ x_2^2\ ,\sqrt{2}x_1x_2\ ,\ \sqrt{2r}x_1\ ,\ \sqrt{2r}x_2\ ,r]^T$. A kernel allows us, in one simple operation, to produce the result of the application of the feature vector without having to go into that higher dimensional space, thus saving computational resources.

Further, there is an idea that instead of a quadratic SVM, we could simply map the data to this higher dimensional space via this mapping, $\phi(\bar{x})$, and then learn a linear classifier in this higher-dimensional space. This is an undesirable solution since the higher-dimensional space could be intractably large. In instances, it could even be infinitely large; this means that the space and computational overhead to map to this space to fit our data is not worth the trouble and is even sometimes infeasible. Thus, the kernel method, of using some $\phi(\bar{x})$ to map directly to our desired space is much preferable.

Next, I looked at the L1 penalty function and the Squared Hing Loss to see how a different penalty (i.e. regularization term) and a loss function would affect the optimization problem. Table 4, below, reports my findings. Below that, in Figure 3, I again plot the L0-norm of the parameter vector against C. This graph has notably the opposite shape as Figure 1. This is explained by the gradients of each function. The norm of theta here is increasing because as C increases our model complexity increases. If we were to use the squared Hinge Loss Functions, as opposed to the Hing Loss Function, there would be a greater penalty for more incorrectly classified points. This means the program would likely take longer to run as it is trying to optimize more. This has the effect of bringing the final SVM closer to the optimal solution.

C	AUROC
0.001	0.50000
0.010	0.75582
0.100	0.8865
1.000	0.8901

Table 4: Best C that Maximizes AUROC (grid search for given rang of C)

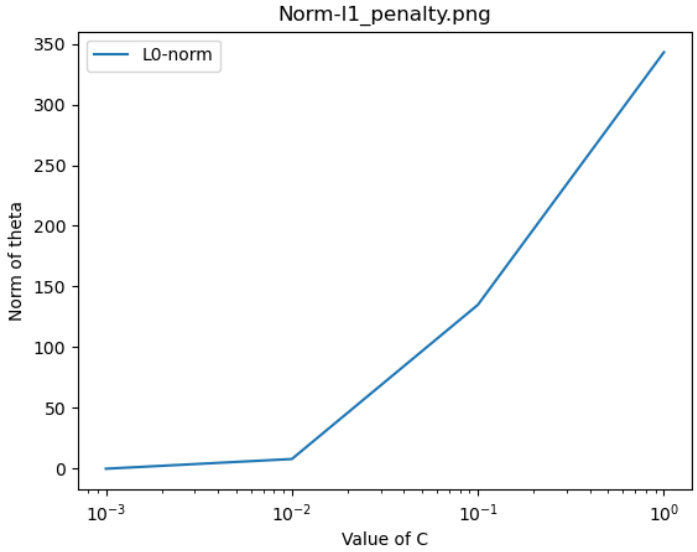

Figure 3: L0-norm of our parameter vector (theta) plotted against C

Finally, I compared the SVM classifier with one of the earliest classification algorithms – the Perceptron. I used a class size of 500 so that the data would be linearly separable and evaluated the decision boundary given by the perceptron algorithm by checking the sign of the dot product of $\bar{\theta}$ and the feature matrices given by our testing data. The perceptron performs worse than the linear SVM since the SVM is trying to find the maximum margin between data points which makes the margin more generalizable. Meanwhile, the perceptron algorithm is just trying to find any boundary that satisfies the linearly separability of the data and not trying to generalize.

Classifier	Accuracy
Perceptron	0.822
Linear SVM	0.836

Table 5: Perceptron performance vs. SVM performance

Asymmetric Cost Functions and Class Imbalance

Here I began to work with unbalanced data. To cope with the imbalance of positive and negative reviews in a data set we can weight each slack variable base on whether or not that feature is positive or negative. This formulation is described below in Equation 1:

$$\underset{\bar{\theta},b,\xi_i}{\text{minimize}} \frac{||\bar{\theta}||^2}{2} + W_p * C \sum_{i|y^{(i)}=1} \xi_i + W_n * C \sum_{i|y^{(i)}=-1} \xi_i$$

$$\text{subject to } y^{(i)}\left(\bar{\theta} \cdot \phi(\bar{x}^{(i)}) + b\right) \geq 1 - \xi_i$$

$$\xi_i \geq 0, \forall i = 1, 2, 3, ..., n$$

Equation 1: Asymmetric Cost Function Formulation

Here, if W_n is much greater than W_p, we will see more penalty given to the negative features. This means our algorithm will be trying to optimize more for classifying the negative values correctly over the positive values. This would in turn decrease our false negatives from our training set and thus increase our sensitivity.

Performance Measures	Performance Score
Accuracy	0.824000
F1-Score	0.828125
AUROC	0.904176
Precision	0.809160
Sensitivity	0.848000
Specificity	0.800000

Table 6: Performance of SVM using arbitrary class weights

Like predicted, the sensitivity of this SVM with a weighted cost formulation was the most dramatically affected. Noticeably, AUROC was changed very little; this was also to be expected since the weights applied gave made the SVM optimize more for the negatively classified points which are still all under the curve.

Class Weights	Performance Measures	Performance Score
$W_n = 1, W_p = 1$	Accuracy	0.840000
$W_n = 1, W_p = 1$	F1-Score	0.902913
$W_n = 1, W_p = 1$	AUROC	0.880200
$W_n = 1, W_p = 1$	Precision	0.877358
$W_n = 1, W_p = 1$	Sensitivity	0.930000
$W_n = 1, W_p = 1$	Specificity	0.480000

Table 7: Performance of SVM with Imbalanced Data

From table 7, we note the sensitivity has drastically increased which means the number of false negatives has decreased. The F1-score also increased drastically which means the general accuracy of the SVM increased. These changes are attributed to the fact that there are not more true positives in the data than there were. Of further interest is the AUROC score which has decreased slightly and the specificity which has been absolutely slashed. This is due to the fact that there are now more true positives in the set than there were before.

Class Weights	Performance Measures	Performance Score
$W_n = 2.5, W_p = 0.1$	Accuracy	0.7660000
$W_n = 2.5, W_p = 0.1$	F1-Score	0.8389875
$W_n = 2.5, W_p = 0.1$	AUROC	0.8593437
$W_n = 2.5, W_p = 0.1$	Precision	0.9330113
$W_n = 2.5, W_p = 0.1$	Sensitivity	0.7625000
$W_n = 2.5, W_p = 0.1$	Specificity	0.7800000

Noting here that we have mitigated the specificity problem by putting a greater weight on the negative classifications. Thus, the SVM places a higher price on their misclassification and now performs better. This has the unfortunate consequence of hurting other scores however. This is why, in the AUROC graph below, we notice that the $W_p = W_n = 1$ case outperforms my custom case. This can be attributed to the unbalanced dataset; in my trying to optimize the classifier for getting more true negatives the true positives went down considerably and thus hurt the overall accuracy and the AUROC score.

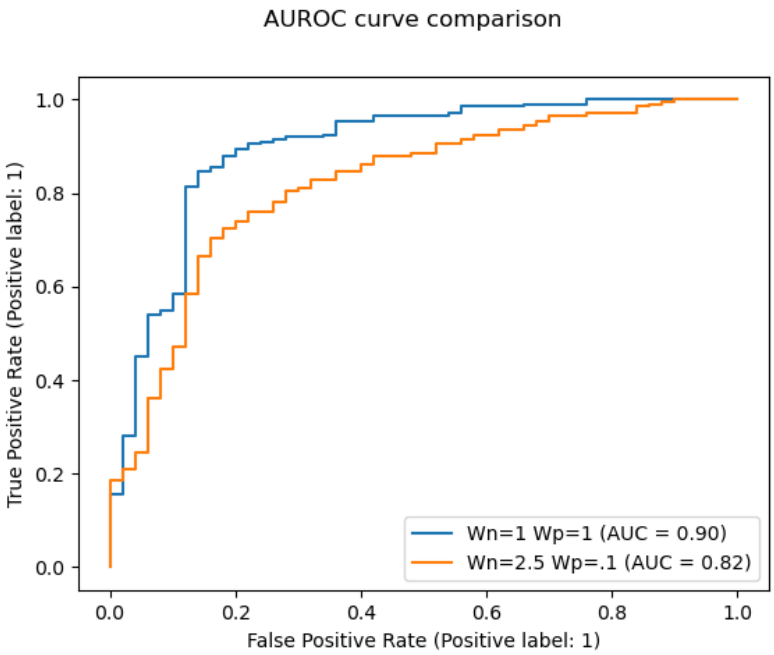

Figure 4: AUROC Curve Comparison

Challenge

For this part I categorized reviews as good, neutral, or negative, (1, 0, -1 respectively). I re-implemented, get_multiclass_training_data() as get_multiclass_training_data_custom() and get_heldout_reviews() as get_heldout_reviews_custom() so as to be able to break up the sentences into n-grams. This allows me to parse the sentences in by not only the individual words but also phrases within them. The user is able to specify how many words they want in a phrase. The purpose of this is to gain a greater understanding of how phrases of words effect the classification of reviews. For example, the words “best” and “worst” mean one thing on their

own (positive and negative, respectively) but together in an example of a 3-gram, “best of the worst,” has a neutral meaning. Thus, I hoped to glean more information about the review in this way.

I further implemented a functionality that lets the user choose to discard words in the dictionary that are used over a defined percentage and under a certain percentage. For example, a word like “the” does not really tell us all that much even though it is the most used word from the Feature Extraction section of this report. This feature and other like it could obscure our data and throw off the classifier, thus the user may want to discard it. Further a word that is only used once may be considered an outlier since contexts are hard to glean from a single usage. Thus, selected a quadratic kernel to give the classifier another degree of freedom.

I selected my parameters based on the data used in dataset.csv. I trained my classifier on a subset of that data then tested on the other and optimized for accuracy. This is obviously not the most accurate for our dataset of heldout.csv but it is a decent guess-timate. I used the one-versus-rest classifier. This splits my multi-class classificatier into one binary classification problem per class. I did this because, a binary classifier is trained on each binary classification problem and predictions are made using the model that is the most confident.

I tried to implement scaling of the features based on their usage (how many times the word/phrase is used) instead of just a binary 0 or 1, but I had difficulty integrating this new weight method into the existing infrastructure given the time constraints.

Conclusion

I leave this report with a discussion of the greater implications of ML and NLP. If I were to implement this model on a news article comment section bias could arise in a lot of different forms. To name a few, if the sight is a predominantly left-leaning political organization the overwhelming majority of comments could vilify people with conservative views. Worse even, the comment section could be filled with internet trolls who spew hate-speech. In both of the situations the learned model will be biased to classify the contrarian views as negative. Thus, the classifier would perpetuate and reinforce the hate-speech and/or vilification that happens in these comment sections.

This is why it is of the utmost importance for us computer (science) engineers to understand the dataset we are training on. The naïve way to mitigate this problem would be to train on as much data as possible so as to have the extreme views “drowned out” by the centrist ones. However, this is not fool-proof. More systematically, if a system were in place to classify the reviews as extreme, then these reviews could be excluded from training. This meaning the classifier could be shipped already having trained on hate-speech/vilifying comments and be able to toss out comments that are too similar to this category. But then this of course brings to light the issue of how to initially train.

APPENDIX: A

DinoMastropietroProject1

February 12, 2021

[ ]:
```
# EECS 445 - Winter 2021
# Project 1 - project1.py
# Due Friday Feb 12, 2021
# Dino Mastropietro

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

from sklearn.model_selection import cross_val_score

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 *

## for challenge
from sklearn.feature_extraction.text import CountVectorizer


def extract_dictionary(df):
    """
    Reads a pandas dataframe, and returns a dictionary of distinct words
    mapping from each distinct word to its index (ordered by when it was found).
    Input:
        df: dataframe/output of load_data()
    Returns:
        a dictionary of distinct words that maps each distinct word
        to a unique index corresponding to when it was first found while
        iterating over all words in each review in the dataframe df
    """
    ##our dictionary
    word_dict = {}
```

```
    ##used to find the number of occurances of each word
    ##word_occr = {}

    exclist = string.punctuation
    ## remove punctuations and digits from old review
    table_ = str.maketrans(exclist, ' '*len(exclist))

    i = 0;
    for sentence in df['reviewText']:
        S = sentence.translate(table_)
        s = S.lower()
        for w in s.split():
            if (w not in word_dict):
                word_dict[w] = i
                ##word_occr[w] = word_occr.get(w, 0) + 1
                ## "the" occurs 3894 times, the most of any word in the␣
↪training set
                i += 1

    return word_dict

def extract_dictionary_advanced(df, n_gram=2, max_occr=1.0, min_occr=0.0):
    """
    Reads a pandas dataframe, and returns a dictionary of distinct words &␣
↪phrases
    mapping from each distinct word(s) a pair of values:
    [its index (ordered by when it was found), number of occurances of the␣
↪phrase/word]
    This is essntially a paired back version of sklearn.feature_extraction.text.
↪CountVectorizer
    Input:
        df: dataframe/output of load_data()
        n:  number of tokens to define N-grams
        max_occr: % above which to ignore words, eg: "the" tells us nothing but␣
↪occrs alot
        min_occr: % below which to ignore words, eg: if "rug" only appears once␣
↪it won't tell us too much
    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
    """
    ##our dictionary
    word_dict = {}
```

```
    ##used to find the number of occurances of each word
    word_occr = {}

    ## remove punctuations and digits from old review, keep apostrophes
    exclist = '''!()-[]{};:"\, <>./?@#$%^&*_~''' + string.digits
    table_ = str.maketrans(exclist, ' '*len(exclist))

    i = 0;
    for sentence in df['reviewText']:
        S = sentence.translate(table_)
        s = S.lower()
        for w in s.split():
            if (w not in word_occr):
                word_occr[w] = 1
                i += 1
            else :
                word_occr[w] += 1

        if(n_gram > 1) : ## phrases can be powerful
            words = s.split()
            for g in range(0, len(words)-(n_gram-1), 1):
                phrase = " ".join(words[g:g+n_gram])
                if (phrase not in word_occr):
                    word_occr[phrase] = 1
                    #i += 1
                else :
                    word_occr[phrase] += 1

    print ("total words: ", i)
    j = 0;
    for w in word_occr :
        if  not (((word_occr[w] / i ) > max_occr) or ((word_occr[w] / i ) <␣
↪min_occr)):
            word_dict[w] = [j, word_occr[w]]
            j+=1

    print("kept words & phrases:  ", j)
    return word_dict


def generate_feature_matrix_custom(df, word_dict, n_gram = 1, n = 0):
    """
    Reads a dataframe and the dictionary of unique words
    to generate a matrix of {1, 0} feature vectors for each review.
    *****ADDED**************
    if n is specified to be non zero, the user chooses to generate a matrix␣
↪that uses the # of
```

```
    occurances instated of just 1 or 0
    ************************
    Use the word_dict to find the correct index to set to 1 for each place
    in the feature vector. The resulting feature matrix should be of
    dimension (# of reviews, # of words in dictionary).
    Input:
        df: dataframe that has the ratings and labels
        word_dict: dictionary of words mapping to indices
    Returns:
        a feature matrix of dimension (# of reviews, # of words in dictionary)
    """
    number_of_reviews = df.shape[0]
    number_of_words = len(word_dict)
    feature_matrix = np.zeros((number_of_reviews, number_of_words))

    ## remove punctuations and digits from old review, keep apostrophes
    exclist = '''!()-[]{};:"\, <>./?@#$%^&*_~''' + string.digits
    table_ = str.maketrans(exclist, ' '*len(exclist))


    i = 0
    for sentence in df['reviewText']:
        S = sentence.translate(table_)
        s = S.lower()
        for w in s.split():
            if w in word_dict:
                ## add a 1 to the right spot in feature matrix [i][j]
                feature_matrix[i][word_dict[w][0]] = word_dict[w][1] if not (n␣
↪== 0) else 1

        if(n_gram > 1) : ## phrases can be powerful
            words = s.split()
            for g in range(0, len(words)-(n_gram-1), 1):
                phrase = " ".join(words[g:g+n_gram])
                if (phrase in word_dict):
                    feature_matrix[i][word_dict[phrase][0]] =␣
↪word_dict[phrase][1] if not (n == 0) else 1

        i+=1

    return feature_matrix


def generate_feature_matrix(df, word_dict):
    """
    Reads a dataframe and the dictionary of unique words
    to generate a matrix of {1, 0} feature vectors for each review.
```

```
    Use the word_dict to find the correct index to set to 1 for each place
    in the feature vector. The resulting feature matrix should be of
    dimension (# of reviews, # of words in dictionary).
    Input:
        df: dataframe that has the ratings and labels
        word_dict: dictionary of words mapping to indices
    Returns:
        a feature matrix of dimension (# of reviews, # of words in dictionary)
    """
    number_of_reviews = df.shape[0]
    number_of_words = len(word_dict)
    feature_matrix = np.zeros((number_of_reviews, number_of_words))

    ## remove punctuations from old review
    exclist = string.punctuation
    table_ = str.maketrans(exclist, ' '*len(exclist))

    i = 0
    for sentence in df['reviewText']:
        S = sentence.translate(table_)
        s = S.lower()
        for w in s.split():
            if w in word_dict:
                ## add a 1 to the right spot in feature matrix [i][j]
                feature_matrix[i][word_dict[w]] = 1
        i+=1

    return feature_matrix

def performance(y_true, y_pred, metric="accuracy"):
    """
    Calculates the performance metric as 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
    """
    if metric == "auroc":
        return metrics.roc_auc_score(y_true, y_pred)

    tn = metrics.confusion_matrix(y_true, y_pred)[0][0]
    fn = metrics.confusion_matrix(y_true, y_pred)[1][0]
```

```
    fp = metrics.confusion_matrix(y_true, y_pred)[0][1]
    tp = metrics.confusion_matrix(y_true, y_pred)[1][1]

    if metric == "accuracy":
        return ((tp + tn) / (tp + tn + fp + fn))

    elif metric == "f1-score":
        return  0 if ((2*tp + fp + fn) == 0) else  ((2 * tp) / (2*tp + fp + fn))
        ## maybe use built in fuction if errors occur

    elif metric == "precision":
         return (tp / (tp + fp))

    elif metric == "sensitivity":
        return (tp / (tp+fn))

    elif metric == "specificity":
        return (tn / (tn+fp))

    ## This is a very useful function
    ## See the sklearn.metrics documentation

def cv_performance(clf, X, y, k=5, metric="accuracy"):
    """
    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 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
    """
    ##StratifiedKFold makes subsets of roughly equal size for us
    scores = []
    skf = StratifiedKFold(n_splits=k)

    for train_index, test_index in skf.split(X,y):
```

```
        X_train, X_test = X[train_index], X[test_index] ## populate arrays with␣
↪inicies of training/
        y_train, y_test = y[train_index], y[test_index] ## testing data for␣
↪both data and labels

        clf.fit(X_train, y_train) ## fit the svm model to this data
        if (metric == "auroc"):
            y_pred = clf.decision_function(X_test)
        else :
            y_pred = clf.predict(X_test)

        ##Put the performance score of the model of each fold in the scores␣
↪array
        scores.append(performance(y_test, y_pred, metric))

    ## this is the real CV implementation
    ## used in testing to verify my implementation
    ## only checks for 'accuracy' metric
    ## check = cross_val_score(clf, X, y, cv=k)

    return np.array(scores).mean()

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.
    """
    if degree == 1:
        if penalty == 'l2':
            ## will not use LinearSVC function here to match with classes␣
↪implementation
            clf = SVC(kernel='linear', C=c, degree=degree, coef0=r,␣
↪class_weight=class_weight)
        else :
            clf = LinearSVC(penalty='l1', dual=False, C=c,␣
↪class_weight='balanced', max_iter=10000)

    elif degree == 2 :
        clf = SVC(kernel='poly', C=c, degree=degree, coef0=r,␣
↪class_weight=class_weight, gamma='auto')

    return clf

def select_param_linear(X, y, k=5, metric="accuracy", C_range = [],␣
↪penalty='l2'):
```

```
    """
    Sweeps different settings for the hyperparameter of a linear-kernel SVM,
    calculating the k-fold CV performance for each setting on X, y.
    Input:
        X: (n,d) array of feature vectors, where n is the number of examples
        and d is the number of features
        y: (n,) array of binary labels {1,-1}
        k: int specifying the number of folds (default=5)
        metric: string specifying the performance metric (default='accuracy',
             other options: 'f1-score', 'auroc', 'precision', 'sensitivity',
             and 'specificity')
        C_range: an array with C values to be searched over
    Returns:
        The parameter value for a linear-kernel SVM that maximizes the
        average 5-fold CV performance.
    """

    best_C_val=0.0
    best_performance=0.0

    for c in C_range :
        clf = select_classifier(penalty, c)
        perf = cv_performance(clf, X, y, k, metric)
        print (" - C:", '{0:5}'.format(c), "  perf: ", '{0:5}'.format(perf)) ##␣
↪debug output
        if perf > best_performance : ## only record stricly better performances
            best_C_val = c
            best_performance = perf

    print (metric, "C: ", best_C_val, " Perf: ", best_performance) ## debug␣
↪output
    return best_C_val

def test_SVM(X_train, Y_train, X_test, Y_test, C, penalty = 'l2',␣
↪metric="accuracy", class_weight='balanced'):

    clf = select_classifier(penalty=penalty, c=C, class_weight=class_weight)
    clf.fit(X_train, Y_train)
    y_pred = clf.predict(X_test) if (metric!="auroc") else clf.
↪decision_function(X_test)

    return performance(Y_test, y_pred, metric)


def eval_decision_boundary(THETA, X_test, Y_test, metric="accuracy"):

    y_pred = []
```

```
    for feat_vec in X_test:
        pred = np.dot(THETA, feat_vec)
        if pred > 0 :
            y_pred.append(1)
        else :
            y_pred.append(-1)

    return performance(Y_test, y_pred, metric)


def train_SVM(X_train, Y_train, C, dict_bin, penalty = 'l2', metric="accuracy"):

    clf = select_classifier(penalty, C)
    clf.fit(X_train, Y_train)
    word_list = list(dict_bin.keys())

    cpys = clf.coef_[0] ## need to cpy because sorted() will perform operation␣
↪in place
    idxs = range(len(cpys))
    manm = sorted(cpys)
    maxAndmin = [idxs for _,idxs in sorted(zip(cpys,idxs))]

    words = ['']*20
    vals = []

    maxidxs = maxAndmin[-10:]
    minidxs = maxAndmin[:10]

    print("METRIC: ", metric)
    for i in range(0,20):
        if (i < 10) :
            words[i] = word_list[minidxs[i]]
            vals.append(manm[i])
        else :
            words[i] = word_list[maxidxs[i-10]]
            vals.append(manm[len(manm)-20+i])

    print("Plotting the number of 10 Most Negative and Positive SVM␣
↪coeficients")
    vals = np.round(vals,decimals=3)
    fig, ax = plt.subplots()

    width = 0.75 # the width of the bars
    ind = np.arange(len(vals))  # the x locations for the groups
    ax.bar(ind, vals, width, color="blue")
    ax.set_xticks(ind+width/2)
```

```
    ax.set_xticklabels(words, minor=False)

    for i, v in enumerate(vals):
        ax.text(i, v, str(v),rotation=70)

    plt.xlabel('Words')
    plt.ylabel('SVM coeficients')
    plt.title('10 Most Negative and Positive SVM coeficients')
    plt.bar(range(len(vals)), vals)
    fig.autofmt_xdate()
    plt.savefig('Extremes-'+penalty+'_penalty.png')
    plt.close()

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

    print("Plotting the number of nonzero entries of the parameter vector as a␣
↪function of C")
    norm0 = np.empty(len(C_range), dtype=np.float64)

    ## Here, for each value of c in C_range, I 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)

    i = 0
    for c in C_range:
        clf = select_classifier(penalty, c)
        cv_performance(clf, X, y, 5, "auroc")
        norm0[i] = np.linalg.norm(clf.coef_[0], ord=0)
        i+=1

    #Plotting
    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 train_perceptron(X_train, Y_train):
    """
    Takes in an input training data X and labels y and
    returns a valid decision boundary theta, b found through
    the Perceptron algorithm. If a valid decision boundary
    can't be found, this function fails to terminate.

    # NOTE: use the first 500 points of the dataset
    # provided & this functions should converge
    """

    k = 0
    theta = np.zeros(X_train.shape[1])
    b = 0
    mclf = True
    while mclf:
        mclf = False
        for i in range(len(X_train)):
            if Y_train[i] * (np.dot(theta, X_train[i]) + b) <= 0:
                theta = theta + 0.1 * (Y_train[i] -  np.dot(theta, X_train[i]))␣
↪* X_train[i]
                b += Y_train[i]
                mclf = True
                k += 1
    return theta, b



def select_param_quadratic(X, y, k=5, metric="accuracy", param_range=[]):
    """
        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)
    """
    best_C_val,best_r_val = 0.0, 0.0
    best_performance = 0.0

    ## Random Search :
    ## Log-Uniform Distribution
    ## p2 = np.random.default_rng().uniform(0,1000,(25,2))
    ## param_range = np.log10(p2)

    for p_r in param_range :
        ## Grid Search :
        clf = select_classifier('l2', p_r[0], 2, p_r[1],'balanced')
        perf = cv_performance(clf, X, y, k, metric)
        #print (" - C:", '{0:5}'.format(p_r[0]), "  r: ", '{0:5}'.
↪format(p_r[1]), "  perf: ", '{0:5}'.format(perf))
        if perf > best_performance :
            best_C_val = p_r[0]
            best_r_val = p_r[1]
            best_performance = perf

    # print (metric, "C: ", best_C_val, "r: ", best_r_val, " Perf: ",␣
↪best_performance)

    # This is very similar to select_param_linear, except
    # the type of SVM model
    return best_C_val,best_r_val

def get_multiclass_training_data_custom(class_size=750, n_gram = 1 , max_occr =␣
↪1.0, min_occr = 0.0, n = 0):
    """
    Reads in the data from data/dataset.csv and returns it using
    extract_dictionary and generate_feature_matrix as a tuple
    (X_train, Y_train) where the labels are multiclass as follows
        -1: poor
         0: average
         1: good
    Also returns the dictionary used to create X_train.
    Input:
        class_size: Size of each class (pos/neg/neu) of training dataset.
    """
    fname = "data/dataset.csv"
    dataframe = load_data(fname)
    neutralDF = dataframe[dataframe['label'] == 0].copy()
    positiveDF = dataframe[dataframe['label'] == 1].copy()
```

```
    negativeDF = dataframe[dataframe['label'] == -1].copy()
    X_train = pd.concat([positiveDF[:class_size], negativeDF[:class_size],␣
↪neutralDF[:class_size]]).reset_index(drop=True).copy()
    dictionary = project1.extract_dictionary_advanced(X_train, n_gram = n_gram␣
↪, max_occr = max_occr, min_occr = min_occr)
    Y_train = X_train['label'].values.copy()
    X_train = project1.generate_feature_matrix_custom(X_train, dictionary,␣
↪n_gram = n_gram, n = n)

    return (X_train, Y_train, dictionary)

def get_heldout_reviews_custom(dictionary, n_gram = 1, n = 0):
    """
    Reads in the data from data/heldout.csv and returns it as a feature
    matrix based on the functions extract_dictionary and generate_feature_matrix
    Input:
        dictionary: the dictionary created by get_multiclass_training_data
    """
    fname = "data/heldout.csv"
    dataframe = load_data(fname)
    X = project1.generate_feature_matrix_custom(dataframe, dictionary, n_gram =␣
↪4, n = 0)
    return X


def main():

    ##############################
    ####                      ####
    #### Feature Extraction: ####
    ####                      ####
    ##############################

    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)
    print(len(X_train[0])) # 7508
    print(((X_train.sum())/ 1000)) #53.794 = avg number of non-zero features

    ############################################
    ####                                    ####
    #### Hyperparameter and Model Selection ####
    ####                                    ####
    ############################################

    ####### 3.1 Hyperparameter Selection for a Linear-Kernel SVM ########
```

```
   C_range = np.logspace(-3, 3, 7)
   penalty = 'l2'
   mets = ["accuracy", "f1-score", "auroc", "precision", "sensitivity",␣
↪"specificity"]

   for m in metrics:
       C = select_param_linear(X_train, Y_train, 5, m, [.1], penalty)
       print(m, ": ", C)

   plot_weight(X_train, Y_train, 'l2', C_range)

   #train_SVM(X_train, Y_train, .1, dictionary_binary, penalty, "accuracy")

   ######## 3.2 Hyperparameter Selection for a Quadratic-Kernel SVM ########
   p_r = [[C, r] for C in np.logspace(-3, 3, 7) for r in np.logspace(-3, 3, 7)]

   auroc_QC  = select_param_quadratic(X_train, Y_train, 5, "auroc", p_r)
   #auroc C:  2.2989280947804698 r:  1.1344975765620917  Perf:  0.90112

   ######## 3.4 Linear-Kernel SVM with L1 Penalty and Squared Hinge Loss␣
↪########
   C2 = np.logspace(-3, 0, 4)

   C = select_param_linear(X_train, Y_train, 5, "auroc", C2, 'l1')
   print("auroc l1: ", C)

   plot_weight(X_train, Y_train, 'l1', C2)


   ######## 3.5 Perceptron Classifier ########
   X_trainP, Y_trainP, X_testP, Y_testP, dictionary_binaryP =␣
↪get_split_binary_data(class_size=500)

   THETA, B = train_perceptron(X_trainP, Y_trainP)

   perceptron_perf = eval_decision_boundary(THETA, X_testP, Y_testP,␣
↪metric="accuracy")
   SVM_perf        = test_SVM(X_trainP, Y_trainP, X_testP, Y_testP, .1,␣
↪penalty, metric="accuracy")
   print("perceptron_perf : ", perceptron_perf)
   print("SVM_perf        : ", SVM_perf       )

   # Dot product of Theta and X_Test points
   # check if it's +/- then evaluate against Y_test and compute accuracy␣
↪performance
```

```
    ##########################################################
    ####                                                  ####
    #### Asymmetric Cost Functions and Class Imbalance ####
    ####                                                  ####
    ##########################################################

    ######## 4.1 Arbitrary class weights ########
    c_w = {-1:1,1:10}

    for m in mets :
        perf_score = test_SVM(X_train, Y_train, X_test, Y_test, 0.1,␣
↪penalty='l2', metric=m, class_weight=c_w)
        print(m, " : ", perf_score)

    ######## 4.2 Imbalanced Data ########
    c_w2 = {-1:50,1:1}

    for m in mets :
        perf_score = test_SVM(IMB_features, IMB_labels, IMB_test_features,␣
↪IMB_test_labels,
                              0.1, penalty='l2', metric=m, class_weight=c_w2)
        print(m, " : ", perf_score)


    ######## 4.3 Choosing appropriate class weights ########

    best_scores = {}

    for m in mets :
        best_scores[m] = np.array([0, 0, 0])

    mets = ["auroc", "specificity"]

    for wn in range(1,10, +2) :
        for wp in range(1,10, +2)

        print(" Wp:", '{0:3}'.format(wp/10), "  Wn: ", '{0:3}'.format(wn/10))
        c_w3 = {-1:wn/10,1:wp/10}
        for m in mets :
            clf = select_classifier(penalty='l2', c=.1, class_weight=c_w3)
            score = cv_performance(clf, IMB_features, IMB_labels, k=5, metric=m)
            print(" - score:", '{0:7}'.format(score), "  metric: ", m)
            if score > best_scores[m]
                best_scores[m] = score
    print(best_scores)

    ######## 4.4 The ROC Curve ########
```

```
   c_w = {-1:1,1:1}
   clf1 = select_classifier(penalty='l2', c=.1, class_weight=c_w)
   score1 = cv_performance(clf1, IMB_features, IMB_labels, k=5, metric='auroc')

   c_w_custom = {-1:2.5,1:.1}
   clf2 = select_classifier(penalty='l2', c=.1, class_weight=c_w_custom)
   score2 = cv_performance(clf2, IMB_features, IMB_labels, k=5, metric='auroc')
   print("score1: ", score1, " score2: ", score2)


   clf1_disp = metrics.plot_roc_curve(clf1, X=IMB_test_features,␣
↪y=IMB_test_labels,
                           name='Wn=1 Wp=1')
   clf2_cisp = metrics.plot_roc_curve(clf2, X=IMB_test_features,␣
↪y=IMB_test_labels,
                           name='Wn=2.5 Wp=.1', ax=clf1_disp.ax_)
   clf1_disp.figure_.suptitle("AUROC curve comparison")

   plt.savefig('AUROC_Curve.png')
   plt.close()


   ##########################################################
   ####                                                  ####
   ####                   Challenge                      ####
   ####                                                  ####
   ##########################################################
   n_gram = 2
   max_occr = .8
   min_occr = 0.0003
   weight = 0

   multiclass_features, multiclass_labels, multiclass_dictionary =␣
↪get_multiclass_training_data_custom(750, n_gram, max_occr, min_occr, weight)
   print("********** got test feats & labels, making train feats *********")

   heldout_features = get_heldout_reviews_custom(multiclass_dictionary,␣
↪n_gram, weight)
   print("********** got train heldout features, making model... *********")

   model = SVC(kernel='poly', C=.75, degree=2, coef0=2,␣
↪decision_function_shape='ovr', gamma='scale')
   print("**********          made the model, fitting...         *********")

   model.fit(multiclass_features, multiclass_labels)
   print("**********         fit the model, predicting...        *********")
```

```
    predictions = model.predict(heldout_features)
    print("**********             predicted, saving...            *********")

    generate_challenge_labels(predictions, 'donatom')
    print("****************              DONE              ****************")


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