EECS 445 PROJECT 1 REPORT Red represents data from code.

PART 2 Feature Extraction

- d = 2850
- Average rating = 15.624

PART 3 Hyperparameter and Model Selection

3.1 Hyperparameter Selection For a Linear-Kernel SVM

(a) Because for either training dataset or test dataset, maintain a sustainable proportion among folders is very important for us to get a great classification and effective test case.

If we have all positive class in the training data, we would absolutely conclude with a really bad hyperparameter that only depends on these positive data. Same reason, if we have all positive class in the test data, our test result would be unrealistic. Same for the negative class.

What we want for our test data and train data is to minimize all the errors. If we do not maintain class proportion across field, we would possibly face huge amount of estimation loss. Because our train/test data only depends on data of one label and we are impossible to give a proper estimate.

(b) Code Only

(c)

Performance Measures	**C**	**Performance**
Accuracy	0.1	0.8390000000000001
F1-Score	0.1	0.8377282080627986
AUROC	0.1	0.92036
Precision	10	0.8412795192518695
Sensitivity	0.001	0.8640000000000001
Specificity	10	0.844

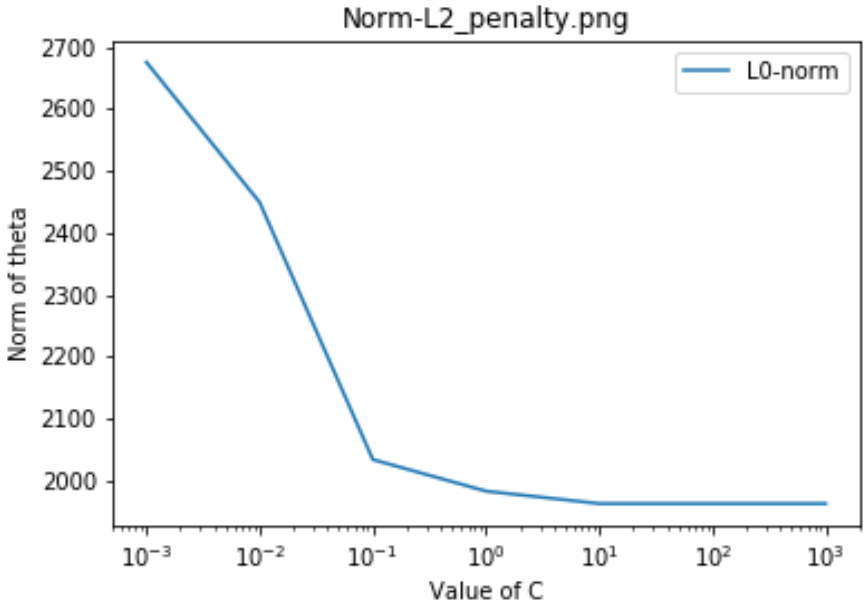
I would choose “AUROC” as my performance measure and hence c = 0.1 for my following classification. First of all, the most intuitive reason, “AUROC” gives me the highest performance among all the measures. More importantly, “AUROC” is a good way to measure how our positive points are estimated more positive than the negative points. Thus, I think this is a good way for us to measure final performance. We cannot have neither too many false negative nor too many false positive. A high “AUROC” score would be really save for us to do the following prediction on this parameter.

Let’s check the performance of c = 0.1 for the other three measure. Precision = 0.8396602879906849; Sensitivity = 0.8380000000000001; Specificity = 0.8400000000000001, which is not really bad, even close to their optimal hyperparameter. So, I would choose “AUROC” and c = 0.1 for my hyperparameter for my future estimation.

(d)

Performance Measures	**Performance**
Accuracy	0.8325
F1-Score	0.8295165394402036
AUROC	0.92055
Precision	0.844559585492228
Sensitivity	0.815
Specificity	0.85

(e)



I find that with all the other parameters remain the same, the 0-norm of theta would decrease with the increase of C. When C is less than 10, the 0-norm of theta is strictly decreasing with increase of C. However, when it comes to be greater than 10, the 0-norm of theta would nearly keep the same no matter how large C becomes.

Because theta represents how strong this piece of feature(word) would affect the final prediction and is related to the overfitting problem, the l0-norm, which represents the number of non-zero terms in our theta. So, the decrease in l0-norm indicates that with the increase of C, the number of "effective" features would be decreasing. Our model is simpler, and we are facing less overfitting in a monotone trend.

(f)

Positive Coefficient	Word
0.969453053931356	thanks
0.901084078877419	thank
0.7654231353985255	great
0.5959079712992326	good

Negative Coefficient	Word
-0.6157880744020268	hours
-0.5495052637207417	delayed
-0.5208214853103377	due
-0.5074326295539983	worst

3.2 Hyperparameter Selection For a Quatratic-Kernel SVM

(a) Code only

(b)

Tuning Scheme	C	R	AUROC
Grid search	1000	0.1	0.91776
Random Search	605.6043276989294	0.18318442222293044	0.91798

The 5-fold CV performance would increase with decrease of C and increase of R. With the decrease of C, our soft margin SVM would be looser so the result would perform worse. However, we have larger R, our kernel, method of calculating dot product in higher dimension, will be higher so that when we do the SVM, we would be doing more precise than lower R. I think this is R that offset our decrease in performance on C.

Random Search is better because we are not just stick ourselves into a range of values, we have more flexible values for us to find the optimal one. Furthermore, random search would perform better than the fixed one in expectation. For different data the optimal value may be quite different,

therefore we cannot only fix our hyperparameter to a set of number. We need some randomness in our algorithm to help us find the optimal one.

3.3 Learning Non-linear Classifiers With a Linear-Kernel SVM

(a) $Kernel(\bar{x}, \bar{y}) = (\bar{x}\bar{y} + r)^2 = (x_1y_1 + x_2y_2 + \cdots + x_dy_d + r)^2$

$$= \sum_{i=1}^{d} x_i{}^2 y_i{}^2 + r^2 + 2\mathrm{r}\sum_{i=1}^{d} x_i y_i + \sum_{i=1}^{d}\sum_{j=1, j\neq i}^{d} x_i y_i x_j y_j$$

$\phi(\bar{x}) = [\, r\,, \sqrt{2r}x_1 \ldots\,, \sqrt{2r}x_d\,, x_1{}^2\,, x_1x_2\,, \ldots, x_ix_j\,, \ldots\,, x_{d-1}x_d\,, x_d{}^2]$

Where for the last several terms, $i \leq j$

This $\phi(\bar{x})$ is the feature mapping formula we want.

(b) While using feature mapping formula, we can get the exactly mapping vector we want and have a good idea about what we are doing and maybe can visualize the mapping. More importantly, we can do some feature selection if we do feature mapping because we know exactly vector of our features. However, kernels do not.

But it may cost tons of calculations when we are doing these mapping transformations and then do dot product. With kernel method, we do not need to know the exactly mapping vector, while we can get the dot product result by simply calculating the kernel in our original dimension, never step into the higher dimension, which save a lot of time and effort.

3.4 Learning-Kernel SVM with L1 Penalty and Squared Hinge Loss

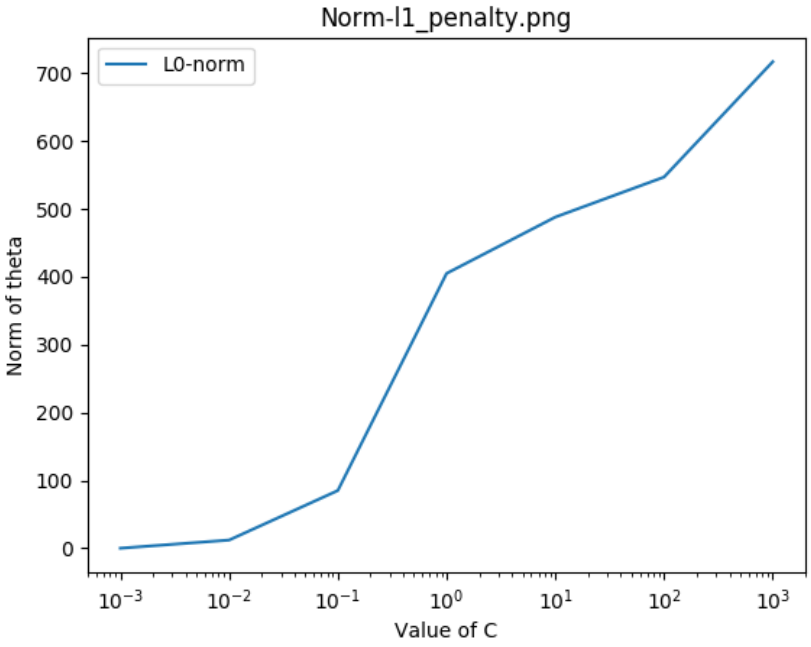
(a)

C	Performance
0.001	0.5
0.01	0.79738
0.1	0.9016299999999999
1	0.9049000000000001
10	0.9030600000000002
100	0.9054399999999999
1000	0.92218

The optimal point happens hat C = 1000 where has AUROC score of 0.92218.

There's an interesting finding that each time I run this L1-penalty Linear SVM, even with the same C, the function would always give me different performance each time I run. But the general trend does not change, the performance goes up with our C increasing. Which make sense to me because if we have higher C, the coefficient of theta would be lower, and we are facing higher loss penalty and the penalty on a feature coefficient would be lower. We are admitted deriving a theta with more "diversified" features. Therefore, while we have a higher C, the model would try to minimize the loss in order to minimize our whole function, thus our final performance will absolutely go better.

(b)



(c) While we have very small c, we will have a norm of theta equals to almost 0! Which indicates that with a higher penalty on θ, the "L1-penalty" will give us a sparse optimal solution, all the data that are not related will have a weight of zero. In this case, while we have c = 10^(-3), our optimal solution becomes an almost 0 theta. And we can conclude that with L1-pernalty, because our gradient does not change by the time, we can always get to the optimal solution, thus, we can do feature selection with "L1-penalty" but not with "L2-penalty".

(d) I believe now the optimal solution would yield a lower loss on the points for the outliers, but losses do not change a lot for the wrong-predicted points near the decision boundary. Thus, being more reasonable. Because when we square loss, for the points that have loss less than 1, the square of the number that is less than one would even diminish the loss; for the points that have loss higher that 1, the square of that number would greatly increase that loss. In conclusion, the loss that higher than 1 would have been penalized more due to the square but less than 1 would go oppositely, therefore, the final parameter would have less loss in general I believe.

PART 4 Asymmetric Cost Functions and Class Imbalance

4.1 Arbitrary class weights

(a) This may result in that situation that we put different emphasis on positive and negative points. With different weight, our solution will perform differently on prediting positive and negative points. The ratio between W_p and W_n may affects our SVM a lot. Because what we want is to minimize that equation, so if we have W_n really large, we need to let ξ to be very small for negative points, which means our misclassification cost for negative points would be really large. Thus, our final decision boundary would be trying its best to correctly classify negative points but put less emphasis on classifying positive points.

If Wn is much greater than Wp, which indicates that the classification criterion for negative points would be much stricter than that of positive points. Therefore, it means that in our solution, the prediction performance of positive points may be very bad, but for negative points, the performance may be much better.

(b)

Performance Measures	**Performance**
Accuracy	0.5625
F1-Score	0.2222222222222222
AUROC	0.90495
Precision	1.0
Sensitivity	0.125
Specificity	1.0

(c) "F1-Score" and "Sensitivity" are the two measures that affected the most by the new class weight. It seems pretty reasonable to me because they are the two measures that would consider the prediction on FALSE POSITIVE. And we put heavy weight on negative points slack so our final prediction would be really bad on positive points, which lead to low performance on "F1-Score" and "sensitivity" measure.

4.2 Imbalanced Data

(a)

Class Weights	Performance Measures	Performance
$W_n = 1, W_p = 1$	Accuracy	0.384
$W_n = 1, W_p = 1$	F1-Score	0.3739837398373984
$W_n = 1, W_p = 1$	AUROC	0.9113500000000001
$W_n = 1, W_p = 1$	Precision	1.0
$W_n = 1, W_p = 1$	Sensitivity	0.23
$W_n = 1, W_p = 1$	Specificity	1.0

(b) It would greatly affect our performance of prediction because of the unbalanced number of two type of data points. In this case we have more negative points than positive points, and our resulting classification boundary perform really bad on positive points with a really low Sensitivity score (high FN) but pretty good on negative points (low FP). So, the unbalanced data will result in unbalance in our final classification boundary, which would lean to the minority side.

4.3 Choosing appropriate class weights

(a) I would use F1-Score because according to the previous problems, F1-Score is the most significant performance score that would be sensitive to the imbalance problem of data. Although AUROC measures how positive points preform more positively than negative points, based on our previous measurement, AUROC does not preform so well on distinguish the imbalance between positive and negative points.

And more importantly, I have done this part with AUROC and F1-Score. With AUROC, I got the optimal ratio as 7:9, which is not reliable and realistic because it's too close to 1:1. With F1-Score, I got the optimal ratio as 5:9, which more reliable than AUROC. So I believe F1-Score does a better job that AUROC.

After deciding which matrix I'll apply for measuring, I decide to loop through all the possible ratio to find the one that perform best, which is:

```
for neg_ratio in range(1, 10):
    for pos_ratio in range (neg_ratio, 10):
```

and then base on the ratio to do the prediction and find the ratio that give us the best AUROC score.

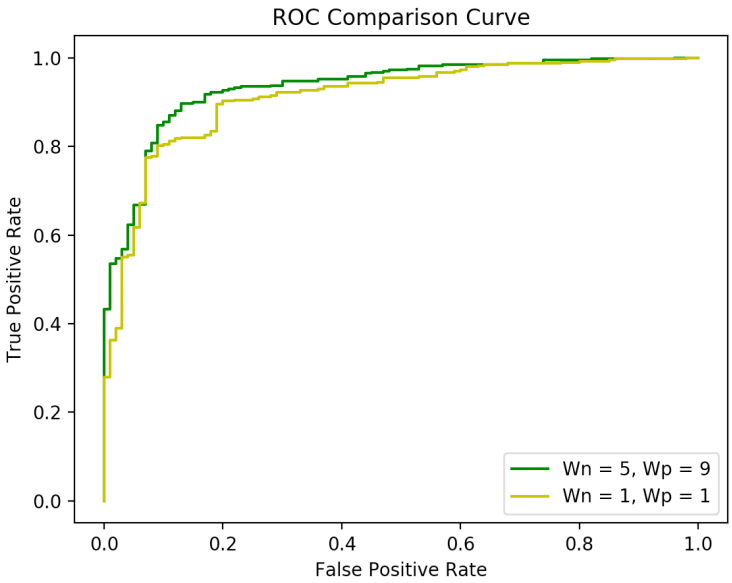
The one I find that optimize is: NEG : POS = 5 : 9

(b)

Class Weights	**Performance Measures**	**Performance**
$W_n = 5, W_p = 9$	Accuracy	0.836
$W_n = 5, W_p = 9$	F1-Score	0.8885869565217391
$W_n = 5, W_p = 9$	AUROC	0.934625
$W_n = 5, W_p = 9$	Precision	0.9732142857142857
$W_n = 5, W_p = 9$	Sensitivity	0.8175
$W_n = 5, W_p = 9$	Specificity	0.91

4.4 The Roc Curve

(a)



PART 5 Challenge

First, let me talk about the aspect that list on the spec and then talk about something more that I would have to get a better classification boundary. I try to use 80% of the dataset as train data and 20% of the dataset as test data. Because there are totally 3000 reviews and 1000 of each labels, I select same portion among all labels to keep balance among three labels.

- Feature engineering

  After reading in all words, there are around 5000 words in total in our dictionary which is really a high number. This high dimension would lead us to tons of calculation while doing SVM so we need to do feature engineering.

  However, before I get rid of all the unimportant features, I would first add an important parameter that would also accounts in our classification – TIMEZONE. While building the word dictionary, I'll also read in the time zone string to build a similar dictionary as the

words. I create a single feature for each time zone and label them the same way as the words and attach them after my feature matrix after words.

```
str = df.loc[i]["user_timezone"]
if str in time_zone: feature_matrix[i][number_of_words + time_zone.get(str)] = 1
```

(Attach time zone in the feature matrix)

```
{nan: 0, 'London': 1, 'Islamabad': 2, 'Berlin': 3, 'Solomon Is.': 4, 'Alaska': 5, 'America/Chicago': 6, 'Hong Kong': 7, 'Bei
jing': 8, 'EST': 9, 'Amsterdam': 10, 'America/Los_Angeles': 11, 'Central Time (US & Canada)': 12, 'Atlantic Time (Canada)':
13, 'Brisbane': 14, 'Mountain Time (US & Canada)': 15, 'Tehran': 16, 'Santiago': 17, 'Pacific Time (US & Canada)': 18, 'Ariz
ona': 19, 'Singapore': 20, 'Buenos Aires': 21, 'Sydney': 22, 'Casablanca': 23, 'Tokyo': 24, 'Helsinki': 25, 'Adelaide': 26,
'Brussels': 27, 'Greenland': 28, 'Abu Dhabi': 29, 'Hawaii': 30, 'America/Boise': 31, 'Quito': 32, 'Eastern Time (US & Canad
a)': 33, 'Edinburgh': 34, 'Caracas': 35, 'Central America': 36, 'America/New_York': 37, 'Monterrey': 38, 'New Caledonia': 3
9, 'New Delhi': 40, 'Athens': 41, 'Dublin': 42, 'Rome': 43, 'Madrid': 44, 'Seoul': 45, 'Tijuana': 46, 'Indiana (East)': 47,
'Mid-Atlantic': 48, 'Vienna': 49, 'Paris': 50}
```

(Time zone Dictionary)

Besides reading in one more feature – time zone, I also do one more step while we read in data. I no longer only read in binary data for the words, I read in FREQUENCY of words. Read in once and increment once in the feature matrix.

```
for part in str.split():
    if part in word_dict: feature_matrix[i][word_dict.get(part)] += 1
```

(reading frequency of words)

Then, for the reason that I now have my feature matrix not only in binary form, I try to NORMALIZE the feature matrix within [0,1] range based on the max value (because the min value is usually zero so we can ignore that) in order to treat all the feature in a fair way.

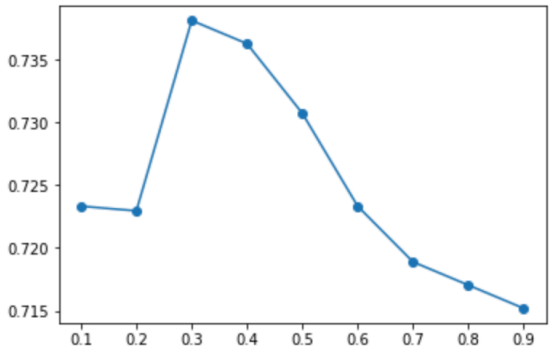
The next step is to do feature selection. I tried two strategies to do this part, using RFE or SELECT_FROM_MODEL. Both testing around 50 times for different parameter, but finally come up with one optimal that is select from model. But ideally, I think both of them would work well on our model because RFE is running my model multiple times and eliminating the most redundant feature and SFM is typically selecting based on importance weights. I have no idea about which way is better, so I run both of them plenty of times to choose the best. For the reason that linearSVC with Lasso regression would help us on feature selection, I pass class linearSVC to my selection model to do testing.

```
# model = LinearSVC(C=0.4, penalty="l1", dual=False, max_iter = 100000, multi_class ='ovr')
# print(X_train.shape)
# selector = RFE(model, n_features_to_select = 600, step = 100)
# selector = selector.fit(X_train, Y_train)
# X_train_new = selector.transform(X_train)
# X_test_new = selector.transform(X_test)
# X_train = normalize(X_train, norm = 'max')
# X_test = normalize(X_test, norm = 'max')
# print(X_train_new.shape)
```

('wasted' RFE training tries)

- Hyperparameter Selection

  First important parameter for me to select is the parameter C while I'm doing feature selection. I try different C, doing the selection, and then measure the performance by running linear SVC cross-validation performance with C = 1 to choose the best C that would fit. The following diagram is the general pattern that I have while running this part, specific value may change, but in general, trend is same.



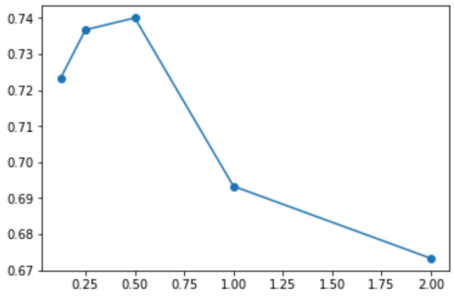
X-axis is C and Y-axis is cv_performance score

```
c_range = [0.1 * x for x in range(1,10)]
x_range = []
for x in c_range:
    model = LinearSVC(C = x, penalty="l1", dual=False, max_iter = 100000, multi_class ='ovr').fit(X_train, Y_train)
    model = SelectFromModel(model, prefit=True, max_features = 2000)
    X_train_new = model.transform(X_train)
    X_test_new = model.transform(X_test)
    clf = LinearSVC(C = 1, penalty="l1", dual=False, max_iter = 1000000, multi_class ='ovr')
    x_range.append(cv_performance(clf, X_train_new, Y_train, 5, "Accuracy"))
plt.plot(c_range,x_range,'-o')
plt.show()
```

(choosing and visualizing the parameter and trend)

The optimal points usually happen at c = 0.2 – 0.4 and my following classification would base on c = 0.4.

Next parameter selecting is the parameter c selection while I'm doing the prediction, that is, c for us SVM model. Same as before, I would use C in 2^x where x in range -3 to 3 and to test and visualize to choose the best one.



X-axis is C and Y-axis is cv_performance score

```
c_range = [2 ** x for x in range(-3,2)]
p_range = [0.1 * x for x in range(1,10)]
n = 0.25
ans = []
for x in c_range:
    clf = select_classifier(c = x, penalty = 'l2')
    clf.fit(X_train_new, Y_train)
    Y_pred = clf.predict(X_test_new)
    ans.append(performance(Y_test, Y_pred,'Accuracy'))
plt.plot(c_range,ans,'-o')
plt.show()
```

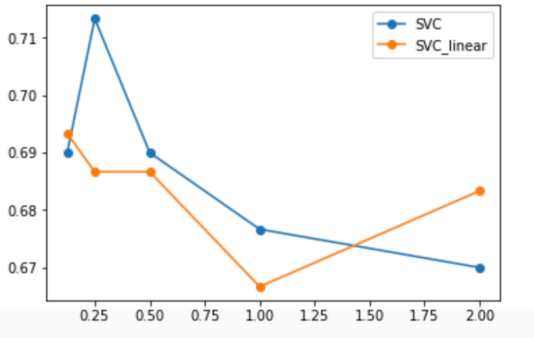
(choosing and visualizing the parameter and trend)

The optimal points usually happen at c = 0.25 – 0.5 and my following classification would base on c = 0.25.

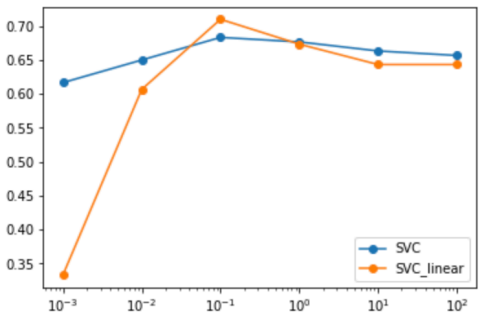
This two C's are the most important parameters that I meet in my classification. The other underlying parameter could be the max number of features that I need to set when I'm doing feature selection. However, I find that even though I do not set an upper bound, with low C, the resulting number of would still be lower than 1500 which is already much better than our original 5000+ features. And there after no more hyperparameter need to be chosen.

- Algorithm Selection

  This part would be the most time-consuming part for me. I have several model to choose from, SVC w/ degree 1, SVC w/ degree 2 and linearSVC. I have run tons of variables on these three models and now I will focus on SVC w/ degree 1 and linear SVC. I will not test on SVC w/ degree 2 because even with our reduced model, the runtime of that algorithm is still pretty long, and the performance is almost same as linear one. SVC with linear and linear SVC are actually doing the same things and SVC have more paramters we can modify. I do a little comparation on linearSVC and SVC.



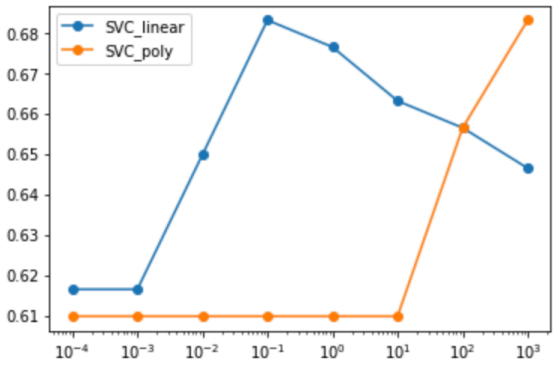
*c = [2 ** x for x in range(-3,3)]*



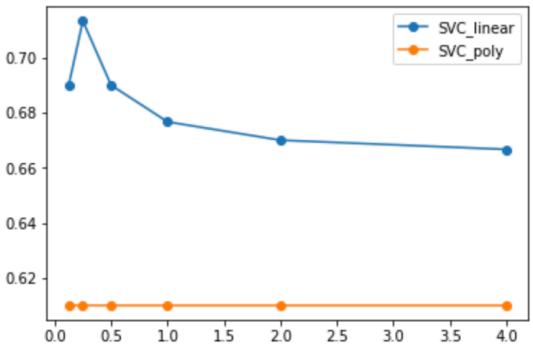
*c = [10 ** x for x in range (-4,4)]*

Based on the graph, these two ways really differ a little in performance. But SVC may give us more parameter to choose from so I choose SVC for my following classification.

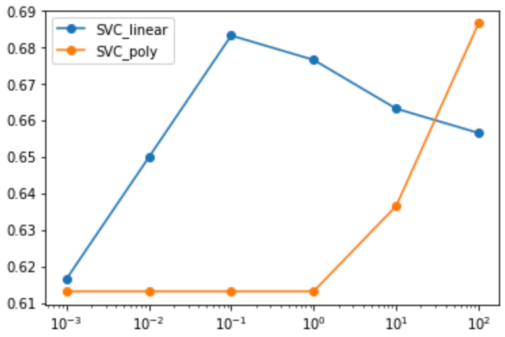
The next step is choosing the parameter of our algorithm – kernel. Kernel has three different type we can choose from, “linear”, “poly”, “rbf”. I’m not sure which kernel would be better while doing our classification. So, I would run one by one to test on these methods. First, the candidates are “linear” and “poly”:



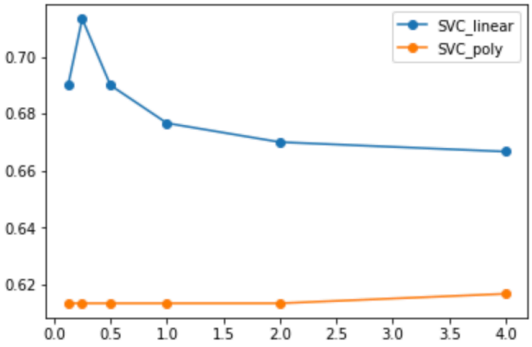
*c = [10 ** x for x in range (-4,4)]; r = 0.1*



*c = [2 ** x for x in range(-3,3)]; r = 0.1*

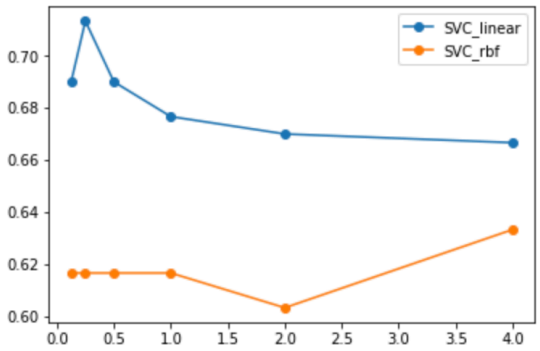


*c = [10 ** x for x in range (-4,4)]; r = 0.5*

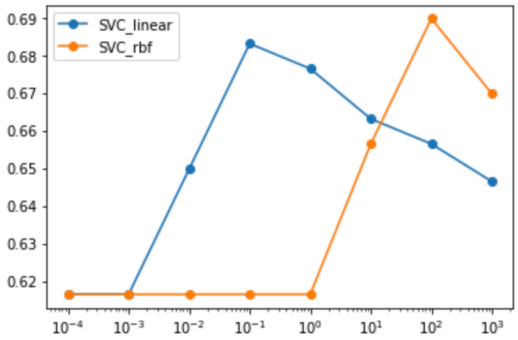


*c = [2 ** x for x in range(-3,3)]; r = 0.5*

I use r = 0.1 for our poly SVC but it performs really bad on our data. I believe it may has been some overfitting happening underlying my classification so I would suppose “rbf” would also perform not so well because overfitting may be even worse. And then we start doing “linear” and “rbf”, I set gamma as “auto” and run the two classification on the same model:



*c = [2 ** x for x in range(-3,3)]; gamma = "auto"*



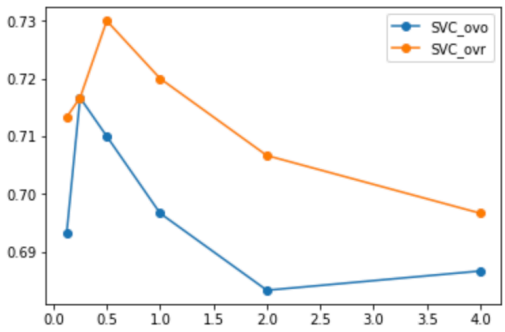
*c = [10 ** x for x in range (-4,4)] ; gamma = "auto"*

Same, after comparation between performance of "linear" and "rbf" model, we can easily find that "linear" is the one that work best for us.

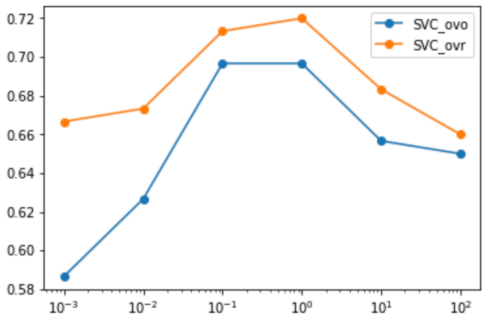
- Multiclass Method

  For the choice between one vs one and one vs all. For my personal understanding, I prefer one vs all because the logic really make sense to me. We run regression on specific one against all the rest and treat the classifier as the probability to choose that parameter. One vs one has the similar idea but run regression on labels one to one. I think run one to rest would have a better idea on how one specific label perform against the other as a whole but not just run regression again and again to different labels. "ovr" may have longer runtime but may have better performance, I think.

  However, when I run the actual model, the performance of ovr is better. Therefore, I would use "ovr" for my following regression.



*c = [2 ** x for x in range(-3,3)]*



*c = [10 ** x for x in range (-4,4)]*

- Beyond
    - SelectFromModel and RFE

I've already talked about them in the previous feature selection part. To my own understanding, we should choose our features by their weight/importance in our final prediction. But how could we know that? REF is actually something that would help us to do that even though I do not use this model finally. REF is essentially running the model and them eliminate the least "important" data from our model. It sounds like would run very slow, but we can set a step size that enable us to eliminate multiple data at a time. There is also a parameter that let us set a lower bound for the feature cancellation. So, I believe it's a great way to feature selection but maybe not for our project this time.

SelectFromModel is the one that I choose to do feature selection in this project. This is a method based on our estimator class (e.g. SVM). We can declare the class while passing our estimator as a parameter. Then after we do prediction on this class, it would rank our features and allow us to transfer our feature matrix to a more "essential" matrix. We can also set an upper bound for our transforming matrix. Internally, this function would weight our features and choose the several max feature cores in order to give us better feature matrix.

    - Train_test_split and Normalize

Train_test_split is a really good way for us to test our model only applying our own model. We can customize our own test ratio in order to control the number of test data

Normalize is a way to normalize our matrix. We can change the parameter norm between "l1", "l2" and "max", which are different ways of normalization.

APPENDIX

Code for Regular Part (NOT Challenge)

```
# EECS 445 - Winter 2018
# Project 1 - project1.py

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

from sklearn.svm import SVC, LinearSVC
from sklearn.model_selection import StratifiedKFold, GridSearchCV
from sklearn import metrics
from matplotlib import pyplot as plt

from helper import *


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 penalty == 'l1': return LinearSVC(penalty = 'l1', dual = False, C = c, class_weight = class_weight, max_iter
= 10000)
    if degree == 1: return SVC(kernel='linear', C=c, class_weight=class_weight, degree = degree)
    if degree == 2: return SVC(gamma = 'auto', kernel='poly', C=c, class_weight=class_weight, degree = degree,
coef0 = r)
    # TODO: Optionally implement this helper function if you would like to
    # instantiate your SVM classifiers in a single function. You will need
    # to use the above parameters throughout the assignment.


def extract_dictionary(df):
    """
    Reads a panda dataframe, and returns a dictionary of distinct words
    mapping from each distinct word to its index (ordered by when it was found).
    Input:
        df: dataframe/output of load_data()
    Returns:
        a dictionary of distinct words that maps each distinct word
        to a unique index corresponding to when it was first found while
        iterating over all words in each review in the dataframe df
    """
    word_dict = {}
    # TODO: Implement this function
    set_d = set()
    for i in range(df.index.size):
        str = df.loc[i]["text"].lower()
```

```
        for c in string.punctuation:
            str = str.replace(c, ' ')
        set_d = set_d.union(set(str.split()))
    set_d = list(set_d)
    for i in range(len(set_d)):
        word_dict[set_d[i]] = i
    return word_dict


def generate_feature_matrix(df, word_dict):
    """
    Reads a dataframe and the dictionary of unique words
    to generate a matrix of {1, 0} feature vectors for each review.
    Use the word_dict to find the correct index to set to 1 for each place
    in the feature vector. The resulting feature matrix should be of
    dimension (number of reviews, number of words).
    Input:
        df: dataframe that has the ratings and labels
        word_list: dictionary of words mapping to indices
    Returns:
        a feature matrix of dimension (number of reviews, number of words)
    """
    number_of_reviews = df.shape[0]
    number_of_words = len(word_dict)
    feature_matrix = np.zeros((number_of_reviews, number_of_words))
    # TODO: Implement this function
    for i in range(df.index.size):
        str = df.loc[i]["text"].lower()
        for c in string.punctuation:
            str = str.replace(c, ' ')
        for part in str.split():
            if part in word_dict: feature_matrix[i][word_dict.get(part)] = 1
    return feature_matrix


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
    """
    # TODO: Implement this function
```

```
    #HINT: You may find the StratifiedKFold from sklearn.model_selection
    #to be useful

    #Put the performance of the model on each fold in the scores array
    scores = []
    skf = StratifiedKFold(k)
    skf.get_n_splits(X, y)

    for train_ind, test_ind in skf.split(X, y):
        X_train = X[train_ind]
        y_train = y[train_ind]
        clf.fit(X_train,y_train)
        X_test = X[test_ind]
        if metric == 'AUROC':
            y_pred = clf.decision_function(X_test)
        else:
            y_pred = clf.predict(X_test)
        y_true = y[test_ind]
        scores.append(performance(y_true, y_pred, metric))

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


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.
    """
    # TODO: Implement this function
    #HINT: You should be using your cv_performance function here
    #to evaluate the performance of each SVM


    max, max_val = 0, 0
    for potential in C_range:
        clf = select_classifier(c = potential, penalty = penalty)
        cur = cv_performance(clf,X,y,k,metric)
        if cur > max_val:
            max = potential
            max_val = cur
        print(potential, ":", cur)
    print(metric, ":", max, max_val)
```

```
    return max


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

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

    # TODO: Implement this part of the function
    #Here, for each value of c in C_range, you should
    #append to norm0 the L0-norm of the theta vector that is learned
    #when fitting an L2- or L1-penalty, degree=1 SVM to the data (X, y)
    for potential in C_range:
        clf = select_classifier(c=potential, penalty = penalty)
        clf.fit(X,y)
        norm_c0 = 0
        for num in clf.coef_[0]:
            if num != 0: norm_c0 += 1
        norm0.append(norm_c0)



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


def select_param_quadratic(X, y, k=5, 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')
            parameter_values: 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 value(s) for a quadratic-kernel SVM that maximize
            the average 5-fold CV performance
    """
    # 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...

    max_c,max_r, max_val = 0, 0, 0
    for potent_c, potent_r in param_range:
        clf = select_classifier(c = potent_c, r = potent_r, degree = 2)
        cur = cv_performance(clf,X,y,k,metric)
        if cur > max_val:
            max_c = potent_c
            max_r = potent_r
            max_val = cur
    print(metric, ":", max_c, max_r, max_val)
    return [max_c, max_r]


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
    """
    # 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.
        # Accuracy = (FP + FN) / N
    if (metric == 'Accuracy'):
        return metrics.accuracy_score(y_true, y_pred)

    # Recall/Sensitivity = TP / (TP + FN)
    elif (metric == 'Sensitivity'):
        return metrics.recall_score(y_true, y_pred)

    # Precision = TP / (TP + FP)
    elif (metric == 'Precision'):
        return metrics.precision_score(y_true, y_pred)

    # F1-Score = 2 * Precision * Sensitivity / (Precision + Sensitivity)
    elif (metric == "F1-Score"):
        return metrics.f1_score(y_true, y_pred)

    # AUROC
    elif (metric == "AUROC"):
```

```
        return metrics.roc_auc_score(y_true, y_pred)

    #Specificity = TN / (TN + FP)
    elif (metric == "Specificity"):
        TN, FP, FN, TP = metrics.confusion_matrix(y_true, y_pred).ravel()
        return TN / (TN + FP)




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()

    # 2
    # print("Second Part:")
    # print("d =",X_train.shape[1])
    # count = 0
    # for i in range(X_train.shape[0]):
    #     count += np.count_nonzero(X_train[i])
    # print("AVE =", count/X_train.shape[0])

    # print("Third Part:")
    # 3.1 (c)
    # print('(c)')
    # C_range = [10 ** x for x in range(-3, 4)]
    # select_param_linear(X_train,Y_train,5,"Accuracy", C_range)
    # select_param_linear(X_train,Y_train,5,"F1-Score", C_range)
    # select_param_linear(X_train,Y_train,5,"AUROC", C_range)
    # select_param_linear(X_train,Y_train,5,"Precision", C_range)
    # select_param_linear(X_train,Y_train,5,"Sensitivity", C_range)
    # select_param_linear(X_train,Y_train,5,"Specificity", C_range)

    # 3.1 (d)
    # clf = select_classifier(c = 0.1)
    # clf.fit(X_train, Y_train)
    # Y_pred = clf.predict(X_test)
    # Y_pred_AU = clf.decision_function(X_test)
    # print('(d)')
    # print("Accuray =", performance(Y_test, Y_pred,'Accuracy'))
    # print("F1-Score =", performance(Y_test, Y_pred,'F1-Score'))
    # print("AUROC =", performance(Y_test, Y_pred_AU, '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'))

    # print('(e)')
    # plot_weight(X_train,Y_train,'L2','Accuracy', C_range)

    # print('(f)')
```

```
# clf = select_classifier(c = 0.1)
# clf.fit(X_train,Y_train)
# coef = np.array(clf.coef_[0])
# large_4 = np.argpartition(coef, -4)[-4:]
# least_4 = np.argpartition(coef, 4)[:4]
# for key, value in dictionary_binary.items():
#     for item in large_4:
#         if value == item:
#             print(key, coef[value])
#     for item in least_4:
#         if value == item:
#             print(key, coef[value])

# 3.2 (b)
# Grid Search
# print('3.2 (b)')
# param_range = []
# for i in range(-3,4):
#     for j in range (-3,4): param_range.append([10**i, 10**j])
# select_param_quadratic(X_train, Y_train, 5, "AUROC", param_range)

# Random Search
# param_range = []
# c = np.random.uniform(-3,3,25)
# r = np.random.uniform(-3,3,25)
# for i in range(25): param_range.append([10 ** c[i],10 ** r[i]])
# select_param_quadratic(X_train, Y_train, 5, "AUROC", param_range)

# 3.4 (a)
# select_param_linear(X_train, Y_train, 5, 'AUROC', C_range, 'l1')

# 3.4 (b)
# plot_weight(X_train, Y_train, 'l1', 'AUROC', C_range)

# 4.1 (b)
# clf = select_classifier(c = 0.01, class_weight = {-1:10, 1:1})
# clf.fit(X_train, Y_train)
# Y_pred = clf.predict(X_test)
# Y_pred_AU = clf.decision_function(X_test)
# print("Accuray =", performance(Y_test, Y_pred,'Accuracy'))
# print("F1-Score =", performance(Y_test, Y_pred,'F1-Score'))
# print("AUROC =", performance(Y_test, Y_pred_AU, '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'))



IMB_features, IMB_labels = get_imbalanced_data(dictionary_binary)
IMB_test_features, IMB_test_labels = get_imbalanced_test(dictionary_binary)

# 4.2
# clf = select_classifier(c = 0.01, class_weight = {-1:7, 1:3})
# clf.fit(IMB_features, IMB_labels)
# Y_pred = clf.predict(IMB_test_features)
```

```
# Y_pred_AU = clf.decision_function(IMB_test_features)
# print("Accuray =", performance(IMB_test_labels, Y_pred,'Accuracy'))
# print("F1-Score =", performance(IMB_test_labels, Y_pred,'F1-Score'))
# print("AUROC =", performance(IMB_test_labels, Y_pred_AU, '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'))

# 4.3 (a)
# max_val = 0
# max_i = 0;
# max_j = 0;
# for i in range(1, 10):
#     for j in range (i, 10):
#         clf = select_classifier(c = 0.01, class_weight = {-1:i, 1:j})
#         perform = cv_performance(clf, IMB_features, IMB_labels, 5, 'F1-Score')
#         if (max_val < perform) :
#             max_val = perform
#             max_i = i
#             max_j = j
# print("Max Neg =", max_i)
# print("Max Pos =", max_j)
# print("Max AUROC =", max_val)


# 4.3 (b)
# clf = select_classifier(c = 0.01, class_weight = {-1:5, 1:9})
# clf.fit(IMB_features, IMB_labels)
# Y_pred = clf.predict(IMB_test_features)
# Y_pred_AU = clf.decision_function(IMB_test_features)
# print("Accuray =", performance(IMB_test_labels, Y_pred,'Accuracy'))
# print("F1-Score =", performance(IMB_test_labels, Y_pred,'F1-Score'))
# print("AUROC =", performance(IMB_test_labels, Y_pred_AU, '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'))

# 4.4
# clf_1 = select_classifier(c = 0.01, class_weight = {-1:1, 1:1})
# clf_1.fit(IMB_features, IMB_labels)
# Y_pred_1 = clf_1.predict(IMB_test_features)
# Y_pred_AU_1 = clf_1.decision_function(IMB_test_features)
# fp_1, tp_1, thresholds = metrics.roc_curve(IMB_test_labels, Y_pred_AU_1)
# fp, tp, thresholds = metrics.roc_curve(IMB_test_labels, Y_pred_AU)
# plt.title('ROC Comparison Curve')
# plt.plot(fp, tp, '-g', label = 'Wn = 5, Wp = 9')
# plt.plot(fp_1, tp_1, '-y', label = 'Wn = 1, Wp = 1')
# plt.ylabel('True Positive Rate')
# plt.xlabel('False Positive Rate')
# plt.legend(loc = 'lower right')
# plt.show()



# TODO: Questions 2, 3, 4
```

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


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

Code for challenge part

```
#!/usr/bin/env python
# coding: utf-8

# In[22]:


import pandas as pd
import numpy as np

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

from sklearn.svm import SVC, LinearSVC
from sklearn.model_selection import StratifiedKFold, GridSearchCV
from sklearn.feature_selection import SelectFromModel
from sklearn.preprocessing import normalize
from sklearn.multiclass import OneVsRestClassifier
from sklearn.feature_selection import RFE
from sklearn.feature_selection import VarianceThreshold
from sklearn.model_selection import train_test_split
from sklearn.multiclass import OneVsOneClassifier
from sklearn import metrics
from matplotlib import pyplot as plt


# In[3]:


def load_data(fname):
    """
    Reads in a csv file and return a dataframe. A dataframe df is similar to dictionary.
    You can access the label by calling df['label'], the content by df['content']
    the rating by df['rating']
    """
    return pd.read_csv(fname)


# In[4]:


def extract_dictionary(df):
    """
    Reads a panda dataframe, and returns a dictionary of distinct words
    mapping from each distinct word to its index (ordered by when it was found).
    Input:
        df: dataframe/output of load_data()
    Returns:
```

```
        a dictionary of distinct words that maps each distinct word
        to a unique index corresponding to when it was first found while
        iterating over all words in each review in the dataframe df
    """
    word_dict = {}
    # TODO: Implement this function
    set_d = set()
    for i in range(df.index.size):
        str = df.loc[i]["text"].lower()
        for c in string.punctuation:
            str = str.replace(c, ' ')
        set_d = set_d.union(set(str.split()))


    # ignore the unnecessary words
    ignore = ['a', 'the', 'i', 'is', 'are', 'an']
    set_d = [a for a in set_d if a not in ignore]
    for i in range(len(set_d)):
        word_dict[set_d[i]] = i
    return word_dict


# In[5]:


def generate_feature_matrix(df, word_dict, time_zone):
    """
    Reads a dataframe and the dictionary of unique words
    to generate a matrix of {1, 0} feature vectors for each review.
    Use the word_dict to find the correct index to set to 1 for each place
    in the feature vector. The resulting feature matrix should be of
    dimension (number of reviews, number of words).
    Input:
        df: dataframe that has the ratings and labels
        word_list: dictionary of words mapping to indices
    Returns:
        a feature matrix of dimension (number of reviews, number of words)
    """
    number_of_reviews = df.shape[0]
    number_of_words = len(word_dict)
    number_of_timezones = len(time_zone)
    feature_matrix = np.zeros((number_of_reviews, number_of_words + number_of_timezones))
    # TODO: Implement this function
    for i in range(df.index.size):
        str = df.loc[i]["text"].lower()
        for c in string.punctuation:
            str = str.replace(c, ' ')
        for part in str.split():
            if part in word_dict: feature_matrix[i][word_dict.get(part)] += 1
        str = df.loc[i]["user_timezone"]
        if str in time_zone: feature_matrix[i][number_of_words + time_zone.get(str)] = 1
    feature_matrix = normalize(feature_matrix, norm = 'max')
    return feature_matrix
```

```
# In[121]:


def get_multiclass_training_data():
    """
    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.
    """
    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()
#     n_x_train, n_x_test = train_test_split(neutralDF, test_size = 0.1)
#     p_x_train, p_x_test = train_test_split(positiveDF, test_size = 0.1)
#     g_x_train, g_x_test = train_test_split(negativeDF, test_size = 0.1)
#     X_train = pd.concat([n_x_train, p_x_train, g_x_train]).reset_index(drop=True).copy()
    X_train = pd.concat([neutralDF, positiveDF, negativeDF]).reset_index(drop=True).copy()
    dictionary = extract_dictionary(X_train)
    time_zone = extract_time_zone(X_train)
#     X_test = pd.concat([n_x_test, p_x_test, g_x_test]).reset_index(drop=True).copy()
    X_test = pd.concat([neutralDF, positiveDF, negativeDF]).reset_index(drop=True).copy()
    Y_train = X_train['label'].values.copy()
    Y_test = X_test['label'].values.copy()
    X_train = generate_feature_matrix(X_train, dictionary, time_zone)
    X_test = generate_feature_matrix(X_test, dictionary, time_zone)



    return (X_train, Y_train, X_test, Y_test, dictionary, time_zone)


# In[7]:


def extract_time_zone(df):
    time_zone = {}
    # TODO: Implement this function
    set_t = set()
    for i in range(df.index.size):
        str = df.loc[i]["user_timezone"]
        set_t.add(str)
    set_t = list(set_t)
    for i in range(len(set_t)):
        time_zone[set_t[i]] = i
    return time_zone


# In[8]:
```

```
def generate_challenge_labels(y, uniqname):
    """
    Takes in a numpy array that stores the prediction of your multiclass
    classifier and output the prediction to held_out_result.csv. Please make sure that
    you do not change the order of the ratings in the heldout dataset since we will
    this file to evaluate your classifier.
    """
    pd.Series(np.array(y)).to_csv(uniqname+'.csv', header=['label'], index=False)
    return


# In[9]:


def get_heldout_reviews(dictionary, time_zone):
    """
    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 = generate_feature_matrix(dataframe, dictionary, time_zone)
    return X


# In[10]:


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 penalty == 'l1': return LinearSVC(penalty = 'l1', dual = False, C = c, class_weight = class_weight, max_iter
= 1000000, multi_class ='ovr')
    if degree == 1: return SVC(kernel='linear', C=c, class_weight=class_weight, degree =
degree,decision_function_shape = 'ovr')
    if degree == 2: return SVC(gamma = 'auto', kernel='poly', C=c, class_weight=class_weight, degree = degree,
coef0 = r, decision_function_shape = 'ovo')


# In[11]:


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

    #Put the performance of the model on each fold in the scores array
    scores = []
    skf = StratifiedKFold(k)
    skf.get_n_splits(X, y)

    for train_ind, test_ind in skf.split(X, y):
        X_train = X[train_ind]
        y_train = y[train_ind]
        clf = clf.fit(X_train,y_train)
        X_test = X[test_ind]
        if metric == 'AUROC':
            y_pred = clf.decision_function(X_test)
        else:
            y_pred = clf.predict(X_test)
        y_true = y[test_ind]
        scores.append(performance(y_true, y_pred, metric))

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


# In[12]:


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.
    """
    # TODO: Implement this function
    #HINT: You should be using your cv_performance function here
    #to evaluate the performance of each SVM


    max, max_val = 0, 0
    for potential in C_range:
        clf = select_classifier(c = potential, penalty = penalty)
        cur = cv_performance(clf,X,y,k,metric)
        if cur > max_val:
            max = potential
            max_val = cur
    return max, max_val


# In[13]:


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')
            parameter_values: 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 value(s) for a quadratic-kernel SVM that maximize
            the average 5-fold CV performance
    """
    # 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...

    max_c,max_r, max_val = 0, 0, 0
    for potent_c, potent_r in param_range:
        clf = select_classifier(c = potent_c, r = potent_r, degree = 2)
        cur = cv_performance(clf,X,y,k,metric)
        print(potent_c, potent_r, cur)
        if cur > max_val:
            max_c = potent_c
            max_r = potent_r
            max_val = cur
    print(metric, ":", max_c, max_r, max_val)
```

```
    return [max_c, max_r]


# In[14]:


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
    """
    # 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.
        # Accuracy = (FP + FN) / N
    if (metric == 'Accuracy'):
        return metrics.accuracy_score(y_true, y_pred)

    # Recall/Sensitivity = TP / (TP + FN)
    elif (metric == 'Sensitivity'):
        return metrics.recall_score(y_true, y_pred)

    # Precision = TP / (TP + FP)
    elif (metric == 'Precision'):
        return metrics.precision_score(y_true, y_pred)

    # F1-Score = 2 * Precision * Sensitivity / (Precision + Sensitivity)
    elif (metric == "F1-Score"):
        return metrics.f1_score(y_true, y_pred)

    # AUROC
    elif (metric == "AUROC"):
        return metrics.roc_auc_score(y_true, y_pred)

    #Specificity = TN / (TN + FP)
    elif (metric == "Specificity"):
        TN, FP, FN, TP = metrics.confusion_matrix(y_true, y_pred).ravel()
        return TN / (TN + FP)


# In[126]:


X_train, Y_train, X_test, Y_test, dictionary, time_zone = get_multiclass_training_data()
print(X_train.shape)
```

```
# In[119]:


# model = LinearSVC(C = 0.4, penalty="l1", dual=False, max_iter = 100000, multi_class ='ovr').fit(X_train, Y_train)
# model = SelectFromModel(model, prefit=True, max_features = 2000)
# X_train_new = model.transform(X_train)
# X_test_new = model.transform(X_test)

# clf = SVC(kernel='linear', C=0.25 ,degree = 1,decision_function_shape = 'ovr')
# clf.fit(X_train_new, Y_train)
# Y_pred = clf.predict(X_test_new)
# print(performance(Y_test, Y_pred,'Accuracy'))


# In[127]:


model = LinearSVC(C = 0.4, penalty="l1", dual=False, max_iter = 100000, multi_class ='ovr').fit(X_train, Y_train)
model = SelectFromModel(model, prefit=True, max_features = 2000)
X_train_new = model.transform(X_train)
X_test_new = model.transform(X_test)

clf = OneVsRestClassifier(SVC(kernel='linear', C=0.25 ,degree = 1))
clf.fit(X_train_new, Y_train)
# Y_pred = clf.predict(X_test_new)
# print(performance(Y_test, Y_pred,'Accuracy'))


# In[128]:


hw_x_train = get_heldout_reviews(dictionary, time_zone)
hw_x_train_new = model.transform(hw_x_train)
Y_pred = clf.predict(hw_x_train_new)
generate_challenge_labels(Y_pred, 'jamiean')


# In[125]:


# c_range = [0.1 * x for x in range(1,10)]
# x_range = []
# for x in c_range:
#     model = LinearSVC(C = x, penalty="l1", dual=False, max_iter = 100000, multi_class ='ovr').fit(X_train, Y_train)
#     model = SelectFromModel(model, prefit=True, max_features = 2000)
#     X_train_new = model.transform(X_train)
#     X_test_new = model.transform(X_test)
#     clf = LinearSVC(C = 1, penalty="l1", dual=False, max_iter = 1000000, multi_class ='ovr')
#     x_range.append(cv_performance(clf, X_train_new, Y_train, 5, "Accuracy"))
# plt.plot(c_range,x_range,'-o')
# plt.show()
```

In[ ]:

```
# model = LinearSVC(C=i * 0.1, penalty="l1", dual=False, max_iter = 100000, multi_class ='ovr').fit(X_train,
Y_train)
# print(X_train.shape)
# model = SelectFromModel(model, prefit=True, max_features = 2000)
# X_train_new = model.transform(X_train)
```

```
# select_param_linear(X_train_new,Y_train,5,"Accuracy",c_range, penalty = 'l1')
# X_test_new = model.transform(X_test)
# print(X_train_new.shape)
# print(X_test_new.shape)
```

In[28]:

```
c_range = [2 ** x for x in range(-3,3)]
p_range = [0.1 * x for x in range(1,10)]
n = 0.25
ans = []
ans_1 = []
for x in c_range:
    clf = OneVsOneClassifier(SVC(kernel='linear', C=x,degree = 1))
    clf.fit(X_train_new, Y_train)
    Y_pred = clf.predict(X_test_new)
    ans.append(performance(Y_test, Y_pred,'Accuracy'))
    clf = OneVsRestClassifier(SVC(kernel='linear', C=x,degree = 1))
    clf.fit(X_train_new, Y_train)
    Y_pred = clf.predict(X_test_new)
    ans_1.append(performance(Y_test, Y_pred,'Accuracy'))

plt.plot(c_range,ans,'-o', label = 'SVC_ovo')
plt.plot(c_range,ans_1,'-o', label = 'SVC_ovr')
plt.legend()
plt.show()
```

In[126]:

```
# model = LinearSVC(C=0.4, penalty="l1", dual=False, max_iter = 100000, multi_class ='ovr')
# print(X_train.shape)
# selector = RFE(model, n_features_to_select = 600, step = 100)
# selector = selector.fit(X_train, Y_train)
# X_train_new = selector.transform(X_train)
# X_test_new = selector.transform(X_test)
# X_train = normalize(X_train, norm = 'max')
# X_test = normalize(X_test, norm = 'max')
# print(X_train_new.shape)
```

In[127]:

```
# X_train = normalize(X_train, norm = 'max')
# X_test = normalize(X_test, norm = 'max')

# for j in range(X_train.shape[1]):
#     std = np.std(X_train[:,j])
#     mean = np.mean(X_train[:,j])
#     for i in range(X_train.shape[0]):
#         X_train[i,j] = st.norm.cdf((X_train[i,j] - mean) / std)
```

In[ ]:

```
# param_range = [[40,4], [39,4], [41,4], [40,5], [40,3], [39, 5], [39, 3], [41,3]]
# # c = np.random.uniform(1,2,10)
# # r = np.random.uniform(1,2,10)
# # for i in range(10): param_range.append([10 ** c[i],10 ** r[i]])
# select_param_quadratic(X_train,Y_train,5,"Accuracy", param_range)

model = LinearSVC(C=0.5, penalty="l1", dual=False, max_iter = 100000, multi_class ='ovr').fit(X_train, Y_train)
print(X_train.shape)
model = SelectFromModel(model, prefit=True, max_features = 1600)
X_train_new = model.transform(X_train)
X_test_new = model.transform(X_test)
```

In[19]:

```
# After finding best parameter is among 0.1 ~ 1
# Run the regression selection
c_range = [ 10 ** x for x in range(-2,3)]
# c_range = [ 0.1 * x for x in range(1,10)]
select_param_linear(X_train_new,Y_train,5,"Accuracy",c_range, penalty = 'l1')
```

In[24]:

```
plt.plot(c_range,ans,'-o', label = 'SVC_ovo')
plt.plot(c_range,ans_1,'-o', label = 'SVC_ovr')
plt.legend()
plt.xscale('log')
plt.show()
```

In[ ]: