Chaojie Feng

CM146, Fall 2018
Problem Set 3: Computational Learning Theory, Kernel, SVM

1 Problem 1

Solution: The VC dimension is 3. First of all, any cases of three points in the space can be shattered by this hypothesis space. This can be shown in the following figure 1 through figure 4. Note that there are $2^3 = 8$ possible configurations in total and I only select 4 of them. Just by switching the labels from original 4 examples will generate the other 4 examples.

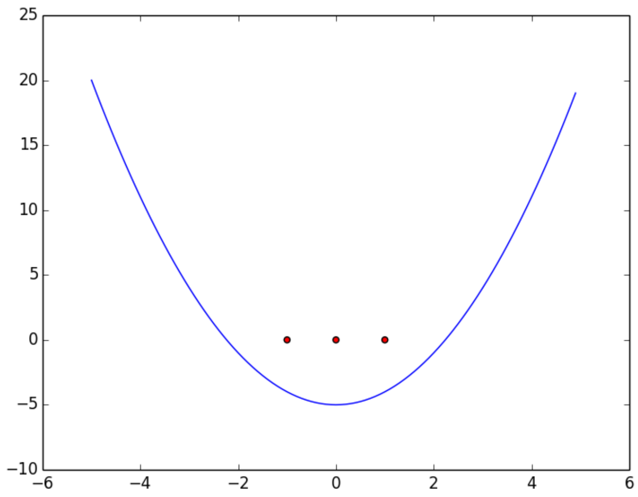


Figure 1: 1st case of 3-point shattering

However, this is not the case when there are four points in the space, 4-points cannot be shattered using this hypothesis space. As a result, $3 \leq VC(H) < 4$, $VC(H) = 3$

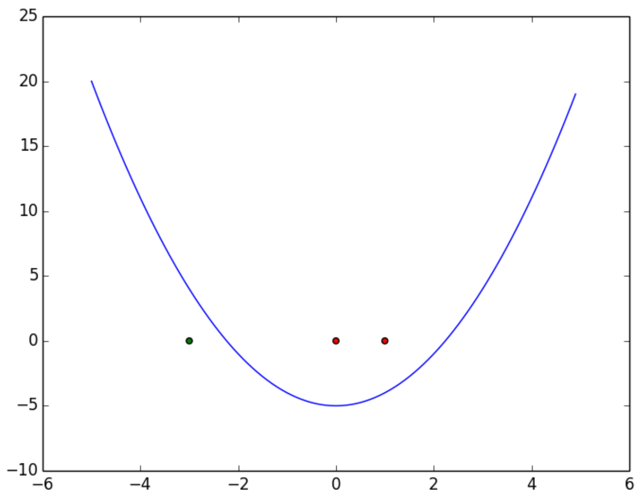


Figure 2: 2nd case of 3-point shattering

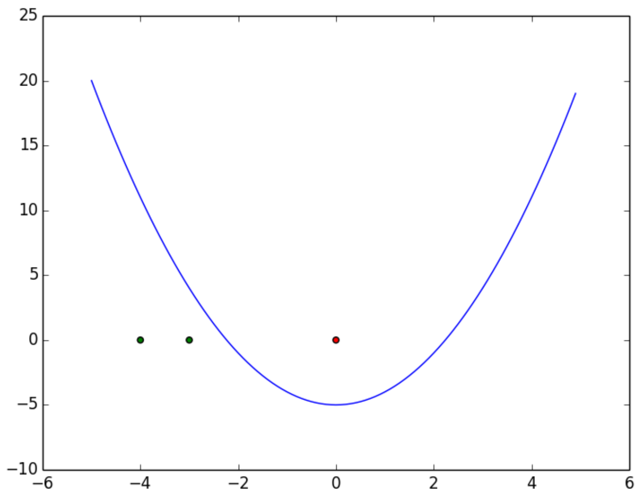


Figure 3: 3rd case of 3-point shattering

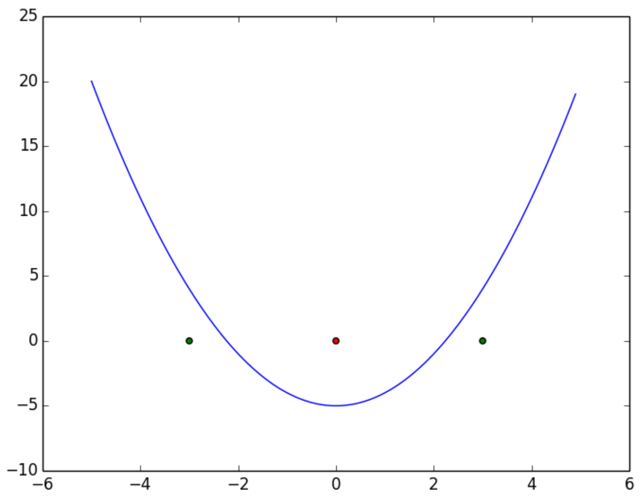


Figure 4: 4th case of 3-point shattering

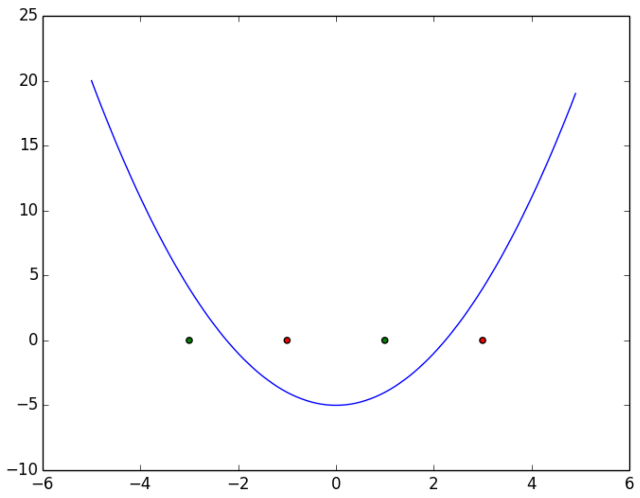


Figure 5: 4-point shattering

2 Problem 2

Solution: From $(1+\beta x\cdot z)^3$, we can expand it as:

$$1+3\beta\sum_d^D x_dz_d+3\beta^2\sum_d^D\sum_e^D x_dz_ex_ez_d+\beta^3\sum_d^D\sum_e^D\sum_f^D x_dz_dx_ez_ex_fz_f$$

It can be rather expanded and reformulated as:

$$1+3\beta\sum_d^D x_dz_d+3\beta^2(\sum_d^D x_d^2z_d^2+2\sum_d^D\sum_{e\neq d}^D x_dz_dx_ez_e)+$$

$$\beta^3(\sum_d^D x_d^3z_d^3+3\sum_d^D\sum_{e\neq d}^D+6\sum_d^D\sum_{e\neq d}^D\sum_{f\neq d,e}^D x_dx_ex_fz_dx_ez_f)$$

We want to represent the above term as an inner product of two expanded feature vector $\phi_\beta(x)$

$$\phi_\beta(x)=[1,\sqrt{3\beta}x_1,\sqrt{3\beta}x_2,...\beta\sqrt{3}x_1^2,\beta\sqrt{3}x_2^2.......$$

$$\beta\sqrt{6}x_1x_2,...\sqrt{\beta^3}x_1^3,\sqrt{\beta^3}x_2^3...\sqrt{3\beta^3}x_1x_2^2...\sqrt{6\beta^3}x_1x_2x_3...]^T$$

In the same format, we can derive $\phi_\beta(z)$:

$$\phi_\beta(z)=[1,\sqrt{3\beta}z_1,\sqrt{3\beta}z_2,...\beta\sqrt{3}z_1^2,\beta\sqrt{3}z_2^2.......$$

$$\beta\sqrt{6}z_1z_2,...\sqrt{\beta^3}z_1^3,\sqrt{\beta^3}z_2^3...\sqrt{3\beta^3}z_1z_2^2...\sqrt{6\beta^3}z_1z_2z_3...]^T$$

Compared to $K(x,z)=(1+x\cdot z)^3$ (i.e. $\beta=1$), an additional β can be tuned in the training process in order to fit the model better. So it provides more flexibility.

3 Problem 3

(a) Problem 3a

Solution: Since we only have two points x_1 and x_2 in 2D space, they must be points where SVM lies on. As a result, all of the inequalities must satisfy with equality in order to minimize the objective function:

$$1w(1,1)^T = 1$$

$$-1w(1,0)^T = 1$$

Which suggests:

$$w_1 + w_2 = 1$$

$$-w_1 = 1$$

As a result:

$$w = (-1,2)^T$$

(b) Problem 3b

Solution: The procedure looks similar to problem 3a. In order to maximize margin, all of the inequalities must satisfy with equality:

$$1w(1,1)^T + b = 1$$

$$-1w(1,0)^T + b = 1$$

$$w_1 + w_2 + b = 1$$

$$-w_1 - b = 1$$

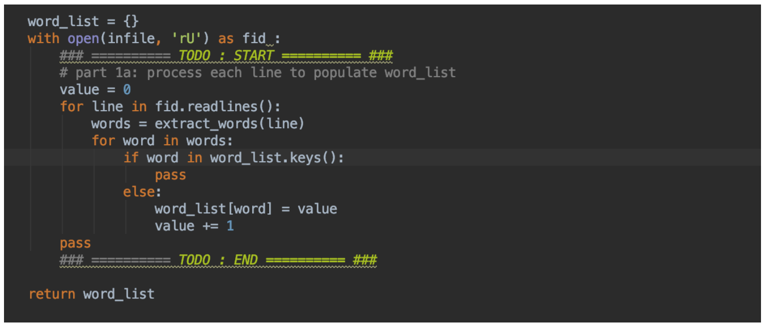
$$w_2 = 2$$

In order to minimize objective function, $w_1 = 0$. As a result, $b = -1$. The solution is different from without offset term.

4 Problem 4.1

(a) Problem 4.1(a)

Solution:

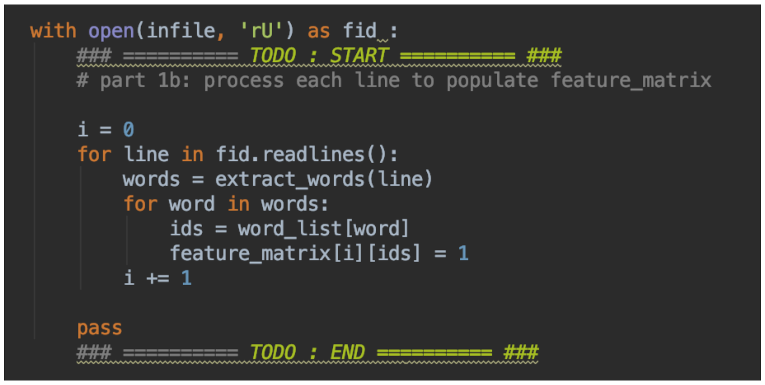


```
word_list = {}
with open(infile, 'rU') as fid :
    ### ========== TODO : START ========== ###
    # part 1a: process each line to populate word_list
    value = 0
    for line in fid.readlines():
        words = extract_words(line)
        for word in words:
            if word in word_list.keys():
                pass
            else:
                word_list[word] = value
                value += 1
    pass
    ### ========== TODO : END ========== ###

return word_list
```

Figure 6: Extract word dictionary from tweets

(b) Problem 4.1(b)

Solution:

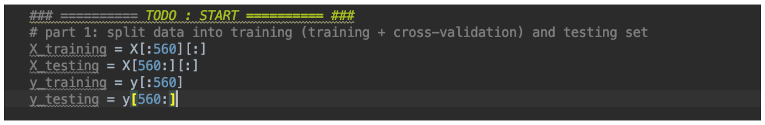


```
with open(infile, 'rU') as fid :
    ### ========== TODO : START ========== ###
    # part 1b: process each line to populate feature_matrix

    i = 0
    for line in fid.readlines():
        words = extract_words(line)
        for word in words:
            ids = word_list[word]
            feature_matrix[i][ids] = 1
        i += 1

    pass
    ### ========== TODO : END ========== ###
```

Figure 7: Building up feature matrix

(c) Problem 4.1(c)

Solution:



```
### ========== TODO : START ========== ###
# part 1: split data into training (training + cross-validation) and testing set
X_training = X[:560][:]
X_testing = X[560:][:]
y_training = y[:560]
y_testing = y[560:]
```

Figure 8: Train/test split

(d) Problem 4.1(d)

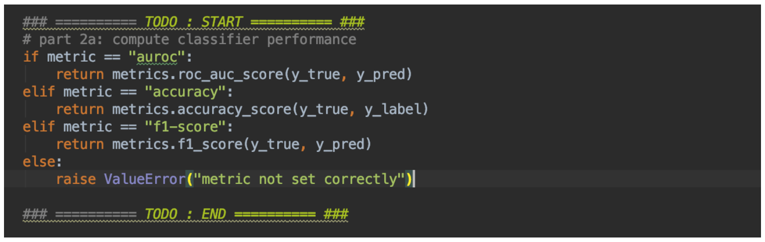
Solution:

Finished feature extraction and generated train/test split

5 Problem 4.2

(a) Problem 4.2(a)

Solution:

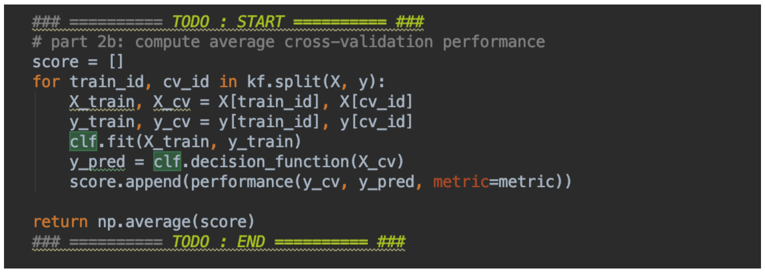


```
### ========== TODO : START ========== ###
# part 2a: compute classifier performance
if metric == "auroc":
    return metrics.roc_auc_score(y_true, y_pred)
elif metric == "accuracy":
    return metrics.accuracy_score(y_true, y_label)
elif metric == "f1-score":
    return metrics.f1_score(y_true, y_pred)
else:
    raise ValueError("metric not set correctly")

### ========== TODO : END ========== ###
```

Figure 9: Implementation of performance function

(b) Problem 4.2(b)

Solution:



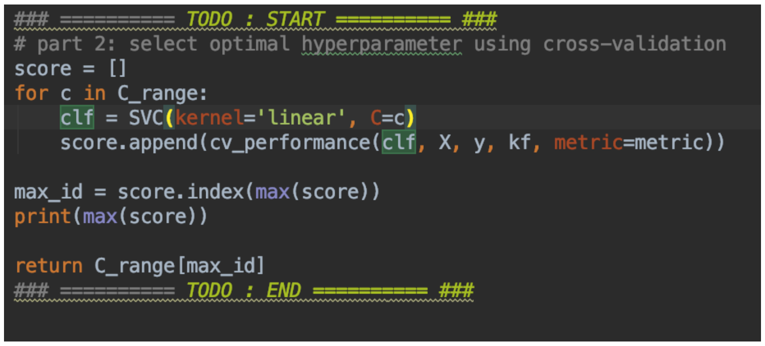
```
### ========== TODO : START ========== ###
# part 2b: compute average cross-validation performance
score = []
for train_id, cv_id in kf.split(X, y):
    X_train, X_cv = X[train_id], X[cv_id]
    y_train, y_cv = y[train_id], y[cv_id]
    clf.fit(X_train, y_train)
    y_pred = clf.decision_function(X_cv)
    score.append(performance(y_cv, y_pred, metric=metric))

return np.average(score)
### ========== TODO : END ========== ###
```

Figure 10: Implementation of cross validation performance function

During implementation of K-Fold validation, I used $StratifiedKFold$ to split the train and dev data in order to maintain class proportions across folds. This is important because due to the unbalance response distribution across dataset, it is possible that every train/dev split will yield a dataset from only one class, thus the model will lose generality. By using $StratifiedKFold$, we manually input some generality of the dataset as well as keep the randomness of sampling, which helps reach a balance distribution of class across folds.

(c) Problem 4.2(c)

Solution:



```
### ========== TODO : START ========== ###
# part 2: select optimal hyperparameter using cross-validation
score = []
for c in C_range:
    clf = SVC(kernel='linear', C=c)
    score.append(cv_performance(clf, X, y, kf, metric=metric))

max_id = score.index(max(score))
print(max(score))

return C_range[max_id]
### ========== TODO : END ========== ###
```

Figure 11: Implementation of selecting hyper parameter C

(d) Problem 4.2(d)

Solution:

Based on "select_param_ linear" function, I found out the best setting for C for each performance measure, tabulated below:

Table 1: Hyper parameter C for different performance measure

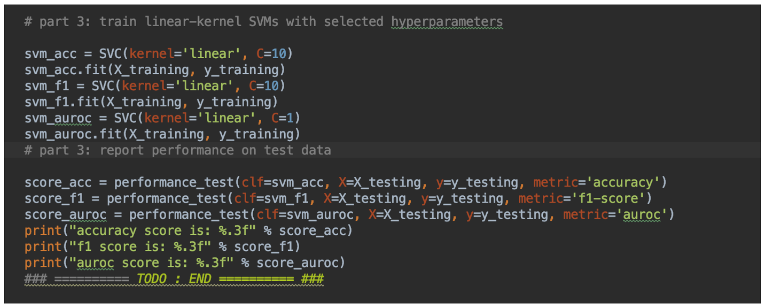
C	Accuracy	F1-score	AUROC
10^{-3}	0.7089	0.8297	0.8106
10^{-2}	0.7107	0.8306	0.8111
10^{-1}	0.8060	0.8755	0.8576
10^{0}	0.8143	0.8749	0.8712
10^{1}	0 .8182	0.8766	0.8696
10^{2}	0.8182	0.8766	0.8696
Best C	10^{1}	10^{1}	10^{0}

As we can see from the table, performances of model increases as penalty term C increases from a very small value (i.e. 10^{-2}, 10^{-1}), but it will remain the same if we continue to increase C after C reaches 10. In general, a large value of C suggests that model will care more about penalties and it will try to make less mistakes in training examples, but may cause overfitting while a smaller value of C suggests model cares less about making mistakes. In this example, we can see that even if we increase C from 10 to 100, none of the performance metrics increase. This suggests the training data is non-separable by using linear kernel.

6 Problem 4.3

(a) Problem 4.3(a)

Solution:

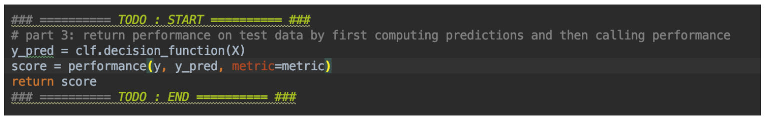


```
# part 3: train linear-kernel SVMs with selected hyperparameters

svm_acc = SVC(kernel='linear', C=10)
svm_acc.fit(X_training, y_training)
svm_f1 = SVC(kernel='linear', C=10)
svm_f1.fit(X_training, y_training)
svm_auroc = SVC(kernel='linear', C=1)
svm_auroc.fit(X_training, y_training)
# part 3: report performance on test data

score_acc = performance_test(clf=svm_acc, X=X_testing, y=y_testing, metric='accuracy')
score_f1 = performance_test(clf=svm_f1, X=X_testing, y=y_testing, metric='f1-score')
score_auroc = performance_test(clf=svm_auroc, X=X_testing, y=y_testing, metric='auroc')
print("accuracy score is: %.3f" % score_acc)
print("f1 score is: %.3f" % score_f1)
print("auroc score is: %.3f" % score_auroc)
### ========== TODO : END ========== ###
```

Figure 12: Evaluation on testing data using different metrics

(b) Problem 4.3(b)

Solution:



```
### ========== TODO : START ========== ###
# part 3: return performance on test data by first computing predictions and then calling performance
y_pred = clf.decision_function(X)
score = performance(y, y_pred, metric=metric)
return score
### ========== TODO : END ========== ###
```

Figure 13: Implementation of performance test function

(c) Problem 4.3(c)

Solution:
For "accuracy", performance score is 0.743, C = 10
For "f1-score", performance score is 0.437, C = 10
For "AUROC", performance score is 0.741, C = 1