CS 578 Preliminary Report

Group Member:

Mian Yang(yang1324@purdue.edu)

Shupei Wang(wang3107@purdue.edu)

Xinru Wang(xinruw@purdue.edu)

Raw Dataset Description

Dataset use: http://www.cs.jhu.edu/~mdredze/datasets/sentiment/ We use the electronic products reviews subset.

Dataset Description: The Multi-Domain Sentiment Dataset contains product reviews taken from Amazon.com from electronic products.

Dataset Format: feature: .... feature: #label#:.

Each line is an user review. Each `feature` is a single word and `count` indicates the frequency of the word in this review. Positive and negative reviews are represented by `label == positive` or `label == negative`.

Preprocessing

Process: First we find all unique words(feature) in raw dataset. Then we loop through each observation of the original dataset and record the frequency of each word for the observation.

Output: Two matrix: `X` and `y`.

- `X` contains 552 rows(# of observations) and 12157 columns(# of unique words). So, `X(i,j)` is the frequency of `j`th feature in `i`th observation.
- `y` contains 552 rows which are labels for each observation.

Cross-validation

> *For this preliminary report, we only use bootstrapping*

We first split the dataset to training and test sets:

```
X_train_val, X_test, y_train_val, y_test = train_test_split(X, y, test_size=0.2)
```

Then we perform cross-validation on training set.

Bootstrapping

Our bootstrapping algorithm use 5 bootstraps:

```
def bootstrap_score(X_train_val, y_train_val, clf, B=100, metric='accuracy'):
    n = len(y_train_val) # number of training samples
    bs_scores = [] # bootstrap scores
    for round in range(B):
        indices = choices(range(n), k=n) # pick n samples with replacement
        #indices = random.sample(range(n), n)
        X_train = X_train_val[indices]
        y_train = y_train_val[indices]
        X_val = np.delete(X_train_val, list(set(indices)), axis=0)
        y_val = np.delete(y_train_val, list(set(indices)), axis=0)
        train_classifier(X_train, y_train, clf)
        bs_scores.append(test_classifier(X_val, y_val, clf, metric))
    return sum(bs_scores) / B # return the average score
```

k-fold

We use 5-fold for this project:

```
def kfold_score(X_train_val, y_train_val, clf, k=5, metric='accuracy'):
    n = len(y_train_val) # number of training samples
    val_size = math.floor(n / k)
    cv_scores = []
    for round in range(k):
        if round == k - 1:
            indices = range(round * val_size, n)
        else:
            indices = range(round * val_size, (round + 1) * val_size)
        X_val = X_train_val[indices]
        y_val = y_train_val[indices]
        X_train = np.delete(X_train_val, indices, axis=0)
        y_train = np.delete(y_train_val, indices, axis=0)
        train_classifier(X_train, y_train, clf)
        cv_scores.append(test_classifier(X_val, y_val, clf, metric))
    return sum(cv_scores) / k
```

Hyperparameter Tuning

> *For this preliminary report, we only use SVM*

We first construct a parameter grid that contains hyperparameters for SVM: five different offsets and 2 kernel functions:

```
Cs = [0.001, 0.01, 0.1, 1, 10]
kernels = ['linear', 'rbf']
parameter_grid = {'C': Cs, 'kernel': kernels}
```

Then by computing Cartesian product of parameter grid, we can have ten combinations of hyperparameters:

```
items = sorted(parameter_grid.items())
    keys, values = zip(*items)
    grid = []
    for v in product(*values):
        grid.append(dict(zip(keys, v)))
```

Then we iterate all combination in the grid and use k-fold/bootstrapping to get the score for each pair of parameters:

```
for idx, param in enumerate(grid):
    print('running {}/{} in parameter grid ...'.format(idx + 1, len(grid)))
    print('C = {}, kernel = {}'.format(param.get('C'), param.get('kernel')))
    clf.set_params(**param) # set the classifier's hyperparameter to the current
parameter
    # if technique='k-fold', then use k-fold cross validation
    if (cv_technique == 'k-fold'):
        score = kfold_score(X_train_val, y_train_val, clf, k=k, metric=metric)
    elif (cv_technique == 'bootstrap'):
        score = bootstrap_score(X_train_val, y_train_val, clf, B=B,
metric=metric)
    print('cv score: {}'.format(score))
    params.append(param)
    scores.append(score)
    if (score > best_score):
        best_score = score
        best_param = param
```

Finally we use the best hyperparameter we get from above procedure to get the final training and test scores. Below is our results for running SVM with bootstrapping:

```
------ import dataset ------
------ split into training set and test set ------
------ train classifier ------
------ training ------
running 1/10 in parameter grid ...
C = 0.001, kernel = linear
cv score: 0.6197964352587502
running 2/10 in parameter grid ...
C = 0.001, kernel = rbf
cv score: 0.597245307470641
running 3/10 in parameter grid ...
C = 0.01, kernel = linear
cv score: 0.8101372843735206
running 4/10 in parameter grid ...
C = 0.01, kernel = rbf
cv score: 0.6135795858887281
running 5/10 in parameter grid ...
C = 0.1, kernel = linear
```

```
cv score: 0.8245137913981123
running 6/10 in parameter grid ...
C = 0.1, kernel = rbf
cv score: 0.6277628624822267
running 7/10 in parameter grid ...
C = 1, kernel = linear
cv score: 0.8154548304227317
running 8/10 in parameter grid ...
C = 1, kernel = rbf
cv score: 0.7660374710223443
running 9/10 in parameter grid ...
C = 10, kernel = linear
cv score: 0.8365819657133307
running 10/10 in parameter grid ...
C = 10, kernel = rbf
cv score: 0.7946789302506009
Best parameter:  {'C': 10, 'kernel': 'linear'}
Best score:  0.8365819657133307
test score:  0.8468468468468469
```