CS 334: Homework #4 Solutions

1. **Feature Extraction + Model Selection**: The full code can be viewed in `q1.py`

   (a) We use train-test split and 5-fold validation to assess perceptron and logistic regression models.

```python
def model_assessment(filename):
    """
    Given the entire data, decide how
    you want to assess your different models
    to compare perceptron, logistic regression,
    and naive bayes, the different parameters,
    and the different datasets.
    """
    Y = []
    X = []
    with open(filename) as fp:
       line = fp.readline()
       while line:
           label = [int(i) for i in line.split() if i.isdigit()]
           text = [i for i in line.split() if i.isdigit()==False]
           Y.append(label)
           X.append(text)
           line = fp.readline()
    data = {'y':np.ravel(Y), 'text':X}
    df = pd.DataFrame(data)
    msk = np.random.rand(len(df)) < 0.7
    train = df[msk]
    test = df[~msk]
    return train, test
```

   (b) To extract frequent words, we should only use words appearing in the *training data* to avoid data leakage. The code is as follows:

```python
def build_vocab_map(train_set):
    word_counter = Counter()
    for s in range(train_set.shape[0]):
        word_counter.update(set(train.text.iloc[s]))
    fre_words = [k for k, v in word_counter.items() if v >= 30]
    return word_counter, fre_words
```

   (c) Binary Dataset:

```python
def construct_binary(train, frequent_words):
    """
    Construct the email datasets based on
    the binary representation of the email.
    For each e-mail, transform it into a
    feature vector where the ith entry,
    £x_i£, is 1 if the ith word in the
    vocabulary occurs in the email,
    or 0 otherwise
    """
    binary_train = np.zeros((len(train), len(frequent_words)))
    for i in range(len(train)):
        words = set(train.text.iloc[i])
        for j in range(len(frequent_words)):
            if frequent_words[j] in words:
                binary_train[i, j] = 1
    return binary_train
```

   To create training and test datasets, just call above function twice:

```
    count_train = construct_count(train, frequent_words)
    count_test = construct_count(test, frequent_words)
```

.

(d) Count Dataset: Similarly, count dataset is obtained by calling construct_count twice.

```
def construct_count(train,frequent_words):
    """
    Construct the email datasets based on
    the count representation of the email.
    For each e-mail, transform it into a
    feature vector where the ith entry,
    £x_i£, is the number of times the ith word in the
    vocabulary occurs in the email,
    or 0 otherwise
    """
    count_train = np.zeros((len(train), len(frequent_words)))
    for i in range(len(train)):
        words = list(train.text.iloc[i])
        for j in range(len(frequent_words)):
                count_train[i, j] = words.count(frequent_words[j])
    return count_train
```

.

(e) (Extra dataset) TFIDF Dataset: This is not part of the homework, but this is a common way to represent text document. For each email, we can construct the term frequency-inverse document frequency (TF-IDF) to capture how important a word is to the corpus. The term frequency of a word is the number of times the word appears in the document. The inverse document frequency refers to the logarithmically scaled inverse fraction of the documents that contain the word. We can use the `TfidfTransformer` or `TfidfVectorizer` in `sklearn.feature_extraction.text` to calculate the TF-IDF representation using the features in (c). We added a function `construct_tfidf`.

```
def construct_tfidf(train, test, frequent_words):
    """
    Construct the email datasets based on
    the TF-IDF representation of the email.
    """
    train_corpus = []
    for i in range(len(train)):
        set1 = set(train.text.iloc[i])
        set2 = list(set1.intersection(frequent_words))
        my_lst_str =' '.join(map(str, set2))
        train_corpus.append(my_lst_str)
    test_corpus = []
    for i in range(len(test)):
        set1 = set(test.text.iloc[i])
        set2 = list(set1.intersection(frequent_words))
        my_lst_str =' '.join(map(str, set2))
        test_corpus.append(my_lst_str)
    TfidfVect = TfidfVectorizer(vocabulary=frequent_words)
    TfidfVect.fit(train_corpus)
    tf_train = TfidfVect.transform(train_corpus)
    tf_test = TfidfVect.transform(test_corpus)
    return tf_train.toarray(), tf_test.toarray()
```

.

2. **Spam Detection via Perceptron**: The full code can be found in `perceptron.py` and `q2bc.py`.

(a) Based on the update formula of perceptron in the slides, we randomly initialize the weights, where the dimension should be the *number of features*. The stats records the mistakes for each epoch.

```
class Perceptron(object):
    mEpoch = 1000  # maximum epoch size
    w = None       # weights of the perceptron

    def __init__(self, epoch):
        self.mEpoch = epoch
        self.lrate = 1
        self.w = None

    def train(self, xFeat, y):
        """
        Train the perceptron using the data

        Parameters
        ----------
        xFeat : nd-array with shape n x d
            Training data
        y : 1d array with shape n
            Array of responses associated with training data.

        Returns
        -------
        stats : object
            Keys represent the epochs and values the number of mistakes
        """
        stats = {}
        # TODO implement this
        self.w = np.zeros(xFeat.shape[1]+1)
        for i in range(self.mEpoch):
                errs = 0
                for datx, label in zip(xFeat, y):
                        prediction = self.predicts(datx)
                        self.w[1:]+= self.lrate * (label - prediction) *datx
                        self.w[0]+= self.lrate * (label - prediction)
                        errs  += np.abs(label[0]-prediction)
                if errs==0:
                        stats[i+1]={'Mistakes': 0}
                        return stats
                stats[i+1] ={'Mistakes': errs}
        return stats
```

```
    def predict(self, xFeat):
            """
        Given the feature set xFeat, predict
        what class the values will have.

        Parameters
        ----------
        xFeat : nd-array with shape m x d
            The data to predict.

        Returns
        -------
        yHat : 1d array or list with shape m
            Predicted response per sample
        """
            yHat = []
            res = self.w[0]+np.dot(xFeat, self.w[1:])
            res = np.sign(res)
            res[res<=0] = 0
            yHat = res
```

```
        return yHat


def calc_mistakes(yHat, yTrue):
    """
    Calculate the number of mistakes
    that the algorithm makes based on the prediction.

    Parameters
    ----------
    yHat : 1-d array or list with shape n
        The predicted label.
    yTrue : 1-d array or list with shape n
        The true label.

    Returns
    -------
    err : int
        The number of mistakes that are made
    """
    mistake = 0
    for i in range(len(yHat)):
            if yHat[i] != yTrue[i]:
                    mistake +=1
    return mistake
```

(b) We use the three datasets from Q1. For finding the optimal number of epochs, we implement the 5-fold cross validation method for a range of epoch numbers. Notice that the cross validation will be utilized on the training set, and the test set will never be touched when we find the optimal number of epochs.

```
def kfold_validation(xtrain, ytrain,foldnum,epochnum):
    kfold = KFold(n_splits=foldnum)
    mistake = []
    for epoch in epochnum:
        total = 0
        for trainIndex, testIndex in kfold.split(xtrain):
            xTrain_k, xTest_k = xtrain[trainIndex], xtrain[testIndex]
            yTrain_k, yTest_k = ytrain[trainIndex], ytrain[testIndex]
            model = Perceptron(epoch)
            trainStats = model.train(xTrain_k, yTrain_k)
            yHat = model.predict(xTest_k)
            total += calc_mistakes(yHat, yTest_k)
        average_mistakes = total / foldnum
        mistake.append(average_mistakes)
    return mistake
```

After obtaining the optimal number of epochs, we will run the model on whole training set with the optimal epoch number, and then test it on test set.

```
#Train new model based on the best parameters
#For Binary datasets
recval_bin = kfold_validation(xTrain_bin, yTrain_bin, 5, iterns)
perc1 = Perceptron(int(iterns[recval_bin.index(min(recval_bin))]))
perc1.train(xTrain_bin, yTrain_bin)
Error1 = calc_mistakes(perc1.predict(xTest_bin), yTest_bin)
pos1,neg1 = obtain_words(perc1, xTrain_bin)
print('Average mistake for each fold in Binary dataset', recval_bin)
print("The mistake before algorithm terminates is", perc1.error[-2])
print("The best optimal eopch number is", iterns[recval_bin.index(min(recval_bin))])
print("The error for Binary test dataset is", Error1)
print("Positive words are", pos1)
print("Negtive words are", neg1)
```

```python
#For Count datasets
recval_co = kfold_validation(xTrain_co, yTrain_co, 5, iterns)
perc2 = Perceptron(int(iterns[recval_co.index(min(recval_co))]))
perc2.train(xTrain_co, yTrain_co)
Error2 = calc_mistakes(perc2.predict(xTest_co), yTest_co)
pos2,neg2 = obtain_words(perc2, xTrain_co)
print('Average mistake for each fold in Count dataset', recval_co)
print("The mistake before algorithm terminates is", perc2.error[-2])
print("The best optimal eopch number is", iterns[recval_co.index(min(recval_co))])
print("The error for Count test dataset is", Error2)
print("Positive words are", pos2)
print("Negtive words are", neg2)

#For Tf-idf datasets
recval_tf = kfold_validation(xTrain_tf, yTrain_tf, 5, iterns)
perc3 = Perceptron(int(iterns[recval_tf.index(min(recval_tf))]))
perc3.train(xTrain_tf, yTrain_tf)
Error3 = calc_mistakes(perc3.predict(xTest_tf), yTest_tf)
pos3,neg3 = obtain_words(perc3, xTrain_tf)
print('Average mistake for each fold in Tf-idf dataset', recval_tf)
print("The mistake before algorithm terminates is", perc3.error[-2])
print("The best optimal eopch number is", iterns[recval_tf.index(min(recval_tf))])
print("The error for Tf-idf test dataset is", Error3)
print("Positive words are", pos3)
print("Negtive words are", neg3)
```

Finally, we will obtain the results,

Dataset	Average mistake in each fold of CV	Mistake before algorithm terminates	Optimal number of epoch	Mistake count for test set
Binary	29.4, 19.8, 19.8, 19.8, 19.8, 19.8	2	30	42
Count	89.6, 50.6, 29.2, 48.6, 30.4, 30.4	31	50	63
Tf-idf	24.2, 17.8, 15.8, 16.8, 16.6, 16.6	4	50	39

(c) To find the 15 words for each case, we simply sort the weights.

```python
def obtain_words(model, xTrain)
    indices = np.argsort(model.w)
    xFeat = pd.read_csv(args.xTrain)
    words_list = list(xFeat.columns.values)
    indices_pos = indices[-15:]
    words_pos = []
    for i in reversed(range(15)):
        index = indices_pos[i]
        words_pos.append(words_list[index])
    # get 15 words with most negative weights
    indices_neg = indices[:15]
    words_neg = []
    for i in range(15):
        index = indices_neg[i]
        words_neg.append(words_list[index])
    return words_pos, words_neg
```

The results are shown below,
For Binary dataset:
Positive words are *'renam', 'client', 'monitor', 'ourselv', 'sign', 'debat', 'william', 'martin', 'without', 'v', 'programm', 'guess', 'offic', 'yourself', 'each'*;
Negtive words are *'www', 'ian', 'onc', 'dave', 'plan', 'startup', 'settl', 'premium', 'british', 'nextpart', 'seen', 'beyond', 'quit', 'never', 'upon'*.
For Count dataset:

Positive words are *'numbercnumb', 'domain', 'renam', 'came', 'ugli', 'martin', 'isn', 'pleasur', 'nation', 'client', 'nobodi', 'yesterdai', 'recent', 'compar', 'numberam'*;
Negtive words are *'numberpm', 'dave', 'button', 'reach', 'cnumber', 'filenam', 'www', 'spamassassin', 'startup', 'us', 'wipe', 'short', 'substanti', 'da', 'sa'*.
For Tf-idf dataset:
Positive words are *'teledynam', 'renam', 'ourselv', 'yourself', 'monitor', 'guess', 'william', 'young', 'blame', 'client', 'debat', 'numbercnumb', 'sign', 'freedom', 'visual'*;
Negtive words are *'www', 'hand', 'auto', 'button', 'cnumber', 'dave', 's', 'us', 'reach', 'must', 'short', 'onc', 'newspap', 'rss', 'kept'*.

3. **Spam Detection using Naive Bayes and Logistic Regression**

(a) There are several kinds of Naive Bayes algorithms including Gauss Naive Bayes, Multinomial Naive Bayes, and Bernoulli Naive Bayes. For each dataset, we need to select a appropriate model based on assumption of likelihood of the features. Obviously, Multinomial Naive Bayes is a more appropriate algorithm for Count dataset.

```
def NB_gauss(train, y, test, Y):
    clf = GaussianNB()
    clf.fit(train, y)
    y_hat = clf.predict(test)
    count =0
    for i in range(len(y_hat)):
        if(y_hat[i] != Y[i]):
            count +=1
    return count

def Ber_gauss(train, y, test, Y):
    clf = BernoulliNB()
    clf.fit(train, y)
    y_hat = clf.predict(test)
    count =0
    for i in range(len(y_hat)):
        if(y_hat[i] != Y[i]):
            count +=1
    return count

def multi_gauss(train, y, test, Y):
    clf = MultinomialNB()
    clf.fit(train, y)
    y_hat = clf.predict(test)
    count =0
    for i in range(len(y_hat)):
        if(y_hat[i] != Y[i]):
            count +=1
    return count
```

The performance against each of the 3 datasets is summarized as:

	Gauss	Bernoulli	Multinomial
Binary	329	106	54
Count	302	106	67
TFIDF	118	115	56

(b)
```
def logistic(train, y, test, Y):
    clf = LogisticRegressionCV(cv=5, random_state=0).fit(train, y)
    y_hat = clf.predict(test)
    count =0
    for i in range(len(y_hat)):
        if(y_hat[i] != Y[i]):
```

```
        count +=1
    return count
```

Note that this dataset is unbalanced, so we can pass a class_weight parameter to the logistic regression model. The weight dictionary is estimated by using training set only.

```
        Logistic
Binary  40
Count   49
TFIDF   27
```