# Homework 4

David Lee
dlee449@emory.edu

1.
(a)
Using train_test_split from sklearn, the data is split into training and test data with size of 0.7 and 0.3 of the original data respectively.

```python
def model_assessment(filename):

    # open the data file
    dat = open(filename)

    # initialize the labels and features lists
    labels = []
    features = []

    # for every line of the data file
    for line in dat:

        # append the first character as label (label is the first character)
        labels.append(line[0])

        # append the text to features list (the text starts from the third character)
        features.append(line[2:])

    # split the data into training and test
    xTrain, xTest, yTrain, yTest = train_test_split(features, labels, test_size = 0.3, random_state = 1)

    # create dataframes for training and test
    train_dat = {'y': yTrain, 'text': xTrain}
    test_dat = {'y': yTest, 'text': xTest}
    train_df = pd.DataFrame(train_dat)
    test_df = pd.DataFrame(test_dat)

    # return train and test dataframes
    return train_df, test_df
```

(b)
A vocabulary map is created by getting the number of emails that each word appears in and filtering the words that appear in at least 30 emails.

```python
def build_vocab_map(train_df):

    # create a default dict for counting unique vocabs in each email
    vocab_counts = defaultdict(int)

    # for every email
    for i in range(train_df.shape[0]):
        # create a list of vocabs of the email
        vocabs = train_df.text[i].split(" ")
        # make list into a list of unique vocabs
        vocabs = set(vocabs)
        # for each unique vocabs
        for vocab in vocabs:
            # count unique vocabs in each email
            vocab_counts[vocab] += 1

    # create a dictionary for the vocabulary map
    vocab_map = {}

    # for every vocab and its counts
    for word, count in vocab_counts.items():
        # select the words that appear in at least 30 emails
        if count >= 30:
            vocab_map[word] = count

    return vocab_map
```

(c)
The Binary dataset is created by transforming each email into a vector of 1 if the vocabulary in the vocabulary map occurs in the email or 0 otherwise.

```
def construct_binary(train_df, vocab_map):
    """
    Construct 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
    """
    # create a list of words for the vocab map
    frequent_words = list(vocab_map.keys())

    # initialize the binary dataset
    binary_train = np.zeros((train_df.shape[0], len(frequent_words)))

    # for each email
    for i in range(train_df.shape[0]):
        # create a list of unique vocabs in an email
        vocabs = train_df.text[i].split(" ")
        vocabs = set(vocabs)

        # for each words in the vocabulary map
        for j in range(len(frequent_words)):
            # if the words in the vocabulary map is in the email
            if frequent_words[j] in vocabs:
                # set vector as 1
                binary_train[i, j] = 1

    return binary_train
```

(d)
The Count dataset is created by transforming each email into a vector by the number of times the vocabulary in the vocabulary map occurs in the email.

```
def construct_count(train_df, vocab_map):
    """
    Construct 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
    """
    # create a list of words for the vocab map
    frequent_words = list(vocab_map.keys())

    # initialize the count dataset
    count_train = np.zeros((train_df.shape[0], len(frequent_words)))

    # for each email
    for i in range(train_df.shape[0]):
        # create a list of vocabs in an email
        vocabs = train_df.text[i].split(" ")

        # for each words in the vocabulary map
        for j in range(len(frequent_words)):
            # count the number of times the jth word appears in the email
            count_train[i,j] = vocabs.count(frequent_words[j])

    return count_train
```

2.

(a)

Train function of perceptron

```python
def train(self, xFeat, y):

    stats = {}
    # TODO implement this

    # initialize the weights
    self.w = np.zeros(xFeat.shape[1]+1)

    # for each epoch
    for i in range(self.mEpoch):
        # initialize the number of mistakes as 0
        mistakes = 0
        # for each email
        for i in range(xFeat.shape[0]):
            text_vector = xFeat[i]
            label = y[i][0]
            # calculate w * xi
            prediction = self.w[0] + np.dot(text_vector, self.w[1:])
            # if w*xi is >= 0
            if prediction >= 0:
                # predict as +1 class
                y_pred = 1
                # if mistake on negative
                if label != y_pred:
                    # update weight
                    self.w[1:] -= text_vector
                    self.w[0] -= 1
                    # add 1 to number of mistakes
                    mistakes += 1
```

```python
            else:
                # predict as 0 class
                y_pred = 0
                # if mistake on positive
                if label != y_pred:
                    # update weight
                    self.w[1:] += text_vector
                    self.w[0] += 1
                    # add 1 to number of mistakes
                    mistakes += 1

    # if there were no mistakes
    if mistakes == 0:
        # update stats as 0 mistakes and break
        stats[i] = {'mistakes': 0}
        return stats
    # if there were mistakes
    else:
        # update stats by the number of mistakes and continue
        stats[i] = {'mistakes': mistakes}

return stats
```

Predict function of perceptron

```python
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 = []

    # calculate w * x
    predictions = self.w[0] + np.dot(xFeat, self.w[1:])

    # for each prediction
    for i in range(len(predictions)):
        # if the prediction is greater than or equal to 0
        if predictions[i] >= 0:
            # predict label as +1
            yHat.append(1)
        # if the prediction is less than 0
        else:
            # predict label as 0
            yHat.append(0)

    return yHat
```

Cal_mistakes function of perceptron

```python
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
    """

    # initialize the number of mistakes
    mistakes = 0

    # for each label
    for i in range(len(yHat)):
        # if the predicted label is not equal to the true label
        if yHat[i] != yTrue[i][0]:
            # add 1 to mistakes
            mistakes += 1

    return mistakes
```

(b)
Implemented in q2.py

Function for getting optimal epoch value

```python
# set the epoch range to test for optimal performance
epoch_range = [1, 50, 100, 150, 200]

# function for getting the optimal epoch value
def optimal_epoch(xTrain, yTrain, folds, epoch_range):
    # get k folds
    kfold = KFold(n_splits = folds, shuffle = True, random_state = 1)
    # Initialize the mistakes dictionary
    mistakes = {}
    # for each epoch value
    for epoch in epoch_range:
        # Initialize the total number of mistakes
        total_mistakes = 0
        # for each fold
        for trainIndex, testIndex in kfold.split(xTrain):
            # Get the training and testing data of the fold
            xTrain_k, xTest_k = xTrain[trainIndex], xTrain[testIndex]
            yTrain_k, yTest_k = yTrain[trainIndex], yTrain[testIndex]
            # Create a perceptron model of the current epoch value
            model = Perceptron(epoch)
            # Train the model using the current fold's training data
            train_stats = model.train(xTrain_k, yTrain_k)
            # Get the predicted values on the testing data
            yHat = model.predict(xTest_k)
            # Get the total mistakes
            total_mistakes += calc_mistakes(yHat, yTest_k)
        # Calculate the average mistakes of the folds on the current epoch value
        average_mistakes = total_mistakes / folds
        # key of epoch value with average mistakes as value
        mistakes[epoch] = average_mistakes

    return mistakes
```

```
--------------------------------------------------------
Average number of mistakes based on size of epoch
            1      50     100    150    200
binary    83.4   20.0   20.0   20.0   20.0
count    479.8   26.8   25.4   24.6   24.6
--------------------------------------------------------
```

| | Epoch Size | Mistakes on Training | Mistakes on Test |
|---|---|---|---|
| Binary | 50 | 0 | 29 |
| Count | 150 | 2 | 45 |

```
------------------------------------------------------------
Training Binary Dataset
Using epoch:  50
Number of mistakes on training set:  0
Number of mistakes on test set:  29
------------------------------------------------------------

------------------------------------------------------------
Training Count Dataset
Using epoch:  150
Number of mistakes on training set:  2
Number of mistakes on test set:  45
------------------------------------------------------------
```

(c)

Function for getting 15 words with the most positive weights and 15 words with the most negative weights

```python
# get the columns of the dataframe
binary_train_df = pd.read_csv("binary_train.csv")
words_list = binary_train_df.columns

def pos_neg_words(model, words_list):
    pos_indices = np.argsort(model.w)[::-1]
    words_pos = []
    for i in range(15):
        index = pos_indices[i]
        words_pos.append(words_list[index])

    neg_indices = np.argsort(model.w)
    words_neg = []
    for i in range(15):
        index = neg_indices[i]
        words_neg.append(words_list[index])

    return words_pos, words_neg
```

```
------------------------------------------------------------
Binary Dataset
15 most positive words:
['manufactur', 'ftp', 'dnumber', 'click', 'inform', 'hundr', 'presid', 'buyer', 'california', 'fals', 'compens', 'target', 'spent', 'end', 'financi']
15 most negative words:
['talk', 'invest', 'see', 'fight', 'check', 'copyright', 'sent', 'have', 'tend', 'for', 'ms', 'better', 'type', 'user', 'origin']
------------------------------------------------------------

------------------------------------------------------------
Count Dataset
15 most positive words:
['ratio', 'd', 'if', 'switch', 'definit', 'click', 'cnumber', 'noth', 'inform', 'execut', 'stick', 'su', 'googl', 'earn', 'idea']
15 most negative words:
['user', 'oct', 'header', 'wa', 'up', 'talk', 'occur', 'storag', 'environ', 'boost', 'for', 'an', 'welcom', 'hewlett', 'w']
------------------------------------------------------------
```

-----------------------------------------------------------
Binary Dataset
15 most positive words:
['manufactur', 'ftp', 'dnumber', 'click', 'inform', 'hundr', 'presid', 'buyer', 'california', 'fals', 'compens', 'target', 'spent', 'end', 'financi']
15 most negative words:
['talk', 'invest', 'see', 'fight', 'check', 'copyright', 'sent', 'have', 'tend', 'for', 'ms', 'better', 'type', 'user', 'origin']
-----------------------------------------------------------

-----------------------------------------------------------
Count Dataset
15 most positive words:
['ratio', 'd', 'if', 'switch', 'definit', 'click', 'cnumber', 'noth', 'inform', 'execut', 'stick', 'su', 'googl', 'earn', 'idea']
15 most negative words:
['user', 'oct', 'header', 'wa', 'up', 'talk', 'occur', 'storag', 'environ', 'boost', 'for', 'an', 'welcom', 'hewlett', 'w']
-----------------------------------------------------------

3.
Implemented in q3.py
(a)
Functions for MultinomialNB and BernoulliNB

```python
def multinomial(xTrain, yTrain, xTest, yTest):
    mistakes = 0
    mult_classifier = MultinomialNB()
    mult_classifier.fit(xTrain, yTrain)
    yHat = mult_classifier.predict(xTest)
    for i in range(len(yHat)):
        if(yHat[i] != yTest[i]):
            mistakes += 1
    return mistakes

def bernoulli(xTrain, yTrain, xTest, yTest):
    mistakes = 0
    ber_classifier = BernoulliNB()
    ber_classifier.fit(xTrain, yTrain)
    yHat = ber_classifier.predict(xTest)
    for i in range(len(yHat)):
        if yHat[i] != yTest[i]:
            mistakes += 1
    return mistakes
```

(b)
Function for Logistic Regression

```python
def logistic(xTrain, yTrain, xTest, yTest):
    mistakes = 0
    log_classifier = LogisticRegressionCV(cv = 5, random_state = 1, max_iter = 1000)
    log_classifier.fit(xTrain, yTrain)
    yHat = log_classifier.predict(xTest)
    for i in range(len(yHat)):
        if(yHat[i] != yTest[i]):
            mistakes += 1
    return mistakes
```

Output for (a) and (b)

```
------------------------------------------------------------
Binary Dataset
BernoulliNB mistakes:  88
MultinomialNB mistakes:  58
Logistic Regression mistakes:  29
------------------------------------------------------------

------------------------------------------------------------
Count Dataset
BernoulliNB mistakes:  88
MultinomialNB mistakes:  64
Logistic Regression mistakes:  39
------------------------------------------------------------
```