Problem 1:

(a). $y^{(i)} w^T x^{(i)} \geq 1$, $y^{(i)} = y = -1$, $x^{(i)} = x = \begin{pmatrix} 1 \\ 1 \end{pmatrix}$

$\Rightarrow y(w_1^* x_1 + w_2^* x_2) \geq 1$

$\Rightarrow -1(w_1^* \cdot 1 + w_2^* \cdot 1) \geq 1$

$\Rightarrow w_1^* + w_2^* \leq -1$

Also want $\min \frac{1}{2}\|w^*\|^2 = \min\left(\frac{1}{2}\sqrt{w_1^{*2} + w_2^{*2}}\right)$

So we have $w_1^* = w_2^* = -\frac{1}{2}$

Hence $\vec{w}^* = \begin{pmatrix} -1/2 \\ -1/2 \end{pmatrix}$

(b). $x^{(1)} = \begin{pmatrix} 1 \\ 1 \end{pmatrix}$, $x^{(2)} = \begin{pmatrix} 1 \\ 0 \end{pmatrix}$. $y^{(1)} = 1$. $y^{(2)} = -1$

When $i=1$:

$y^{(1)} w^{*T} x^{(1)} = 1 \cdot (w_1^* \cdot 1 + w_2^* \cdot 1) = w_1^* + w_2^* \geq 1$

when $i=2$:

$y^{(1)} w^{*T} x^{(2)} = -1 \cdot (w_1^* \cdot 1 + w_2^* \cdot 0) = -w_1^* \geq 1 \Rightarrow w_1^* \leq -1$

Also want to minimize $w_1^{*2} + w_2^{*2} = |w_1^*|^2 + |w_2^*|^2$.

Thus, we have $w_1^* = -1$, $w_2^* = 2$

So $\vec{w}^* = \begin{pmatrix} -1 \\ 2 \end{pmatrix}$

(c). since $b \neq 0$. we have $y^{(i)} \theta^T x^{(i)} = y^{(i)}(w^{*T} x^{(i)} + b^*) \geq 1$

And $x^{(1)} = \begin{pmatrix} 1 \\ 1 \end{pmatrix}$. $x^{(2)} = \begin{pmatrix} 1 \\ 0 \end{pmatrix}$, $y^{(1)} = 1$, $y^{(2)} = -1$.

When $i=1$: $1 \cdot (w_1^* + w_2^* + b^*) = w_1^* + w_2^* + b^* \geq 1 \Rightarrow w_1^* + w_2^* \geq 1 - b^*$

when $i=2$: $-1 \cdot (w_1^* + 0 + b^*) = -w_1^* - b^* \geq 1 \Rightarrow w_1^* \leq -1 - b^*$

When $b^* \geq 1$: $1 - b^* \leq 0$ and $-1 - b^* \leq 0 \Rightarrow w_1^* = -1 - b^*$, $w_2^* = 2$. So $(\vec{w}^*, b^*) = \left(\begin{pmatrix} -1-b^* \\ 2 \end{pmatrix}, b^*\right)$

when $1 \geq b^* \geq -1$: $1 - b^* \geq 0$ and $-1 - b^* \leq 0 \Rightarrow w_1^* = -1 - b^*$, $w_2^* = 2$. So $(\vec{w}^*, b^*) = \left(\begin{pmatrix} -1-b^* \\ 2 \end{pmatrix}, b^*\right)$

when $-1 \geq b^*$: $1 - b^* \geq 0$ and $-1 - b^* \geq 0 \Rightarrow w_1^* = -1 - b^*$, $w_2^* = 2$. So $(\vec{w}^*, b^*) = \left(\begin{pmatrix} -1-b^* \\ 2 \end{pmatrix}, b^*\right)$

To minimize $\|w^*\|^2$, we namely want to minimize $|w_1^*| = |-1 - b^*| \Rightarrow b^* = -1$

Hence, $(\vec{w}^*, b^*) = \left(\begin{pmatrix} 0 \\ 2 \end{pmatrix}, -1\right)$

Problem 2:

i	Label	Hypothesis 1 (1st iteration)				Hypothesis 2 (2nd iteration)			
		$\mathbf{w}_0$	$f_1 \equiv$ sign$(x_1 - 2)$	$f_2 \equiv$ sign$(x_2 - 6)$	$h_1 \equiv f_2$	$\mathbf{w}_1$	$f'_1 \equiv$ sign$(x_1 - 2)$	$f'_2 \equiv$ sign$(x_2 - 0)$	$h_2 \equiv f'_1$
(1)	(2)	(3)	(4)	(5)	(6)	(7)	(8)	(9)	(10)
1	−	0.1	−1	−1	−1	0.0625	−1	+1	−1
2	−	0.1	−1	−1	−1	0.0625	−1	+1	−1
3	+	0.1	+1	+1	+1	0.0625	+1	+1	+1
4	+	0.1	−1	−1	−1	0.25	−1	+1	−1
5	−	0.1	−1	+1	+1	0.25	−1	+1	−1
6	−	0.1	+1	−1	−1	0.0625	+1	+1	+1
7	+	0.1	+1	+1	+1	0.0625	+1	+1	+1
8	−	0.1	−1	−1	−1	0.0625	−1	−1	−1
9	+	0.1	−1	+1	+1	0.0625	−1	+1	−1
10	+	0.1	+1	+1	+1	0.0625	+1	+1	+1
			$\varepsilon_1 = 0.3$	$\varepsilon_1 = 0.2$			$\varepsilon_2 = 0.375$	$\varepsilon_2 = 0.4375$	

(c).

$$\varepsilon_1 = 0.1 + 0.1 = 0.2$$

$$\beta_1 = \frac{1}{2} \ln\left(\frac{1-0.2}{0.2}\right) = \frac{1}{2} \cdot \ln(4) = 0.6931$$

If correct: $w_{1,i} = 0.1 \cdot \exp(-0.6931) = 0.05$

If incorrect: $w_{1,i} = 0.1 \cdot \exp(0.6931) = 0.2$

Normalize;

If correct: $w_{1,i} = \dfrac{0.05}{0.05 \cdot 8 + 0.2 \cdot 2} = 0.0625$

If incorrect: $w_{1,i} = \dfrac{0.2}{0.05 \cdot 8 + 0.2 \cdot 2} = 0.25$

(d). $\varepsilon_2 = 0.375$

$$\beta_2 = \frac{1}{2} \ln\left(\frac{1-0.375}{0.375}\right) = \frac{1}{2} \cdot \ln\left(\frac{5}{3}\right) = 0.2554$$

Since $\beta_2 < \beta_1$, the final hypothesis solely depends on h_1.

Which is: $H(x) = h_1 = (-1, -1, +1, -1, +1, -1, +1, -1, +1, +1)^T$

Problem 3.

1.(a).

```python
######################################################################
# functions -- evaluation
######################################################################

def performance(y_true, y_pred, metric="accuracy"):
    """
    Calculates the performance metric based on the agreement between the
    true labels and the predicted labels.

    Parameters
    --------------------
        y_true -- numpy array of shape (n,), known labels
        y_pred -- numpy array of shape (n,), (continuous-valued) predictions
        metric -- string, option used to select the performance measure
                  options: 'accuracy', 'f1-score', 'auroc', 'precision',
                           'sensitivity', 'specificity'

    Returns
    --------------------
        score  -- float, performance score
    """
    # map continuous-valued predictions to binary labels
    y_label = np.sign(y_pred)
    y_label[y_label==0] = 1

    ### ========== TODO : START ========== ###
    # part 1a: compute classifier performance

    if metric == "accuracy":
        score = metrics.accuracy_score(y_true, y_pred)
    elif metric == "f1-score":
        score = metrics.f1_score(y_true, y_pred)
    elif metric == "auroc":
        score = metrics.roc_auc_score(y_true, y_pred)
    elif metric == "precision":
        score = metrics.precision_score(y_true, y_pred)
    elif metric == "sensitivity": #same as recall
        score = metrics.recall_score(y_true, y_pred)
    elif metric == "specificity":
        tn, fp, _, _ = metrics.confusion_matrix(y_true, y_label).ravel()
        return tn / (tn + fp)

    return score
    ### ========== TODO : END ========== ###
```

(b)

```python
def cv_performance(clf, X, y, kf, metric="accuracy"):
    """
    Splits the data, X and y, into k-folds and runs k-fold cross-validation.
    Trains classifier on k-1 folds and tests on the remaining fold.
    Calculates the k-fold cross-validation performance metric for classifier
    by averaging the performance across folds.

    Parameters
    --------------------
        clf    -- classifier (instance of LinearSVC)
        X      -- numpy array of shape (n,d), feature vectors
                    n = number of examples
                    d = number of features
        y      -- numpy array of shape (n,), binary labels {1,-1}
        kf     -- model_selection.StratifiedKFold
        metric -- string, option used to select performance measure

    Returns
    --------------------
        score   -- float, average cross-validation performance across k folds
    """

    ### ========== TODO : START ========== ###
    # part 1b: compute average cross-validation performance
    scores = []

    for train_index, test_index in kf.split(X, y):
        X_train, X_test = X[train_index], X[test_index]
        y_train, y_test = y[train_index], y[test_index]

        clf.fit(X_train, y_train)

        y_scores = clf.decision_function(X_test)

        if metric == "auroc":
            score = performance(y_test, y_scores, metric)
        else:
            y_pred = np.sign(y_scores)
            y_pred[y_pred == 0] = 1
            score = performance(y_test, y_pred, metric)

        scores.append(score)

    return np.mean(scores)
    ### ========== TODO : END ========== ###
```

(c)

```python
def select_param_linear(X, y, kf, metric="accuracy"):
    """
    Sweeps different settings for the hyperparameter of a linear SVM,
    calculating the k-fold CV performance for each setting, then selecting the
    hyperparameter that 'maximize' the average k-fold CV performance.

    Parameters
    --------------------
        X      -- numpy array of shape (n,d), feature vectors
                    n = number of examples
                    d = number of features
        y      -- numpy array of shape (n,), binary labels {1,-1}
        kf     -- model_selection.StratifiedKFold
        metric -- string, option used to select performance measure

    Returns
    --------------------
        C -- float, optimal parameter value for linear SVM
    """

    print('Linear SVM Hyperparameter Selection based on ' + str(metric) + ':')
    C_range = 10.0 ** np.arange(-3, 3)

    ### ========== TODO : START ========== ###
    # part 1c: select optimal hyperparameter using cross-validation

    best_score = 0
    best_C = None

    for C in C_range:
        clf = LinearSVC(loss = 'hinge', random_state=0, C=C)
        score = cv_performance(clf, X, y, kf, metric)
        if score > best_score:
            best_score = score
            best_C = C

    print("For {}: Best C = {:.3f}. Best Score = {:.3f}".format(metric, best_C, best_score))

    return best_C
    ### ========== TODO : END ========== ###
```

3.2.(b).

```python
def performance_test(clf, X, y, metric="accuracy"):
    """
    Estimates the performance of the classifier.

    Parameters
    --------------------
        clf          -- classifier (instance of LinearSVC)
                          [already fit to data]
        X            -- numpy array of shape (n,d), feature vectors of test set
                          n = number of examples
                          d = number of features
        y            -- numpy array of shape (n,), binary labels {1,-1} of test set
        metric       -- string, option used to select performance measure

    Returns
    --------------------
        score        -- float, classifier performance
    """


    ### ========== TODO : START ========== ###
    # part 2b: return performance on test data under a metric.
    y_scores = clf.decision_function(X)

    if metric != "auroc":
        y_pred = np.sign(y_scores)
        y_pred[y_pred == 0] = 1
    else:
        y_pred = y_scores

    # Use the performance function to calculate the score
    score = performance(y, y_pred, metric)
    return score

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

3.1.(b), (c), 3.2.(a), (b). main function:

```
###########################################################################
# main
###########################################################################

def main() :
    np.random.seed(1234)

    # read the tweets and its labels, change the following two lines to your own path.
    ### ========== TODO : START ========== ###
    file_path = '../data/tweets.txt'
    label_path = '../data/labels.txt'
    ### ========== TODO : END ========== ###
    dictionary = extract_dictionary(file_path)
    print(len(dictionary))
    X = extract_feature_vectors(file_path, dictionary)
    y = read_vector_file(label_path)
    # split data into training (training + cross-validation) and testing set
    X_train, X_test = X[:560], X[560:]
    y_train, y_test = y[:560], y[560:]

    metric_list = ["accuracy", "f1-score", "auroc", "precision", "sensitivity", "specificity"]

    ### ========== TODO : START ========== ###
    # part 1b: create stratified folds (5-fold CV)
    kf = StratifiedKFold(n_splits = 5, shuffle = True, random_state = 0)

    # part 1c: for each metric, select optimal hyperparameter for linear SVM using CV
    best_C_values = {}
    for metric in metric_list:
        best_C_values[metric] = select_param_linear(X_train, y_train, kf, metric)

    # part 2a: train linear SVMs with selected hyperparameters
    classifiers = {}
    for metric, C in best_C_values.items():
        clf = LinearSVC(loss='hinge', random_state=0, C=C)
        clf.fit(X_train, y_train)
        classifiers[metric] = clf

    # part 2b: test the performance of your classifiers.
    for metric, clf in classifiers.items():
        score = performance_test(clf, X_test, y_test, metric)
        print(f"Performance for {metric} with C={C}: {score}")
    ### ========== TODO : END ========== ###


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

```
Linear SVM Hyperparameter Selection based on accuracy:
For accuracy: Best C = 10.000. Best Score = 0.829
Linear SVM Hyperparameter Selection based on f1-score:
For f1-score: Best C = 10.000. Best Score = 0.883
Linear SVM Hyperparameter Selection based on auroc:
For auroc: Best C = 10.000. Best Score = 0.895
Linear SVM Hyperparameter Selection based on precision:
For precision: Best C = 10.000. Best Score = 0.856
Linear SVM Hyperparameter Selection based on sensitivity:
For sensitivity: Best C = 0.001. Best Score = 1.000
Linear SVM Hyperparameter Selection based on specificity:
For specificity: Best C = 10.000. Best Score = 0.625
Performance for accuracy with C=10.0: 0.7428571428571429
Performance for f1-score with C=10.0: 0.43749999999999994
Performance for auroc with C=10.0: 0.7453838678328474
Performance for precision with C=10.0: 0.6363636363636364
Performance for sensitivity with C=10.0: 1.0
Performance for specificity with C=10.0: 0.9183673469387755
```

Problem 4:

(a).

```
### ========== TODO : START ========== ###
# Part 4(a): Implement the decision tree classifier and report the training error.
print('Classifying using Decision Tree...')
clf = DecisionTreeClassifier(criterion='entropy', random_state=0)
clf.fit(X, y)
ypred_train = clf.predict(X)
train_error = 1 - metrics.accuracy_score(y, ypred_train)
print(f"Training Error: {train_error}")
train_error, test_error = error(clf, X, y)
print(f"Average Training Error after 100 trials: {train_error}")
print(f"Average Test Error after 100 trials: {test_error}")
### ========== TODO : END ========== ###
```

✓ 0.3s

```
Classifying using Decision Tree...
Training Error: 0.014044943820224698
Average Training Error after 100 trials: 0.011528998242530775
Average Test Error after 100 trials: 0.24104895104895108
```

(b)

```
### ========== TODO : START ========== ###
# Part 4(b): Implement the random forest classifier and adjust the number of samples used in bootstrap sampling.
print('Classifying using Random Forest...')
n_samples = len(X)  # Total number of samples in your data
max_samples_options = [0.1, 0.2, 0.3, 0.4, 0.5, 0.6, 0.7, 0.8]

best_test_error = float('inf')
best_max_samples = None
best_train_error = None

for max_samples_ratio in max_samples_options:
    max_samples = int(n_samples * max_samples_ratio)
    clf = RandomForestClassifier(criterion='entropy', random_state=0, max_samples=max_samples)

    # Use the provided error function to get training and test error
    train_error, test_error = error(clf, X, y)

    if test_error < best_test_error:
        best_test_error = test_error
        best_train_error = train_error
        best_max_samples = max_samples

print(f"Best setting for max_samples: {best_max_samples} samples")
print(f"Training Error for this setting: {best_train_error}")
print(f"Test Error for this setting: {best_test_error}")
### ========== TODO : END ========== ###
```

✓ 1m 52.0s

```
Classifying using Random Forest...
Best setting for max_samples: 142 samples
Training Error for this setting: 0.10314586994727591
Test Error for this setting: 0.18797202797202794
```

(c)

```
### ========== TODO : START ========== ###
# Part 4(c): Implement the random forest classifier and adjust the number of features for each decision tree.
print('Classifying using Random Forest...')
best_max_samples = 142

best_test_error = float('inf')
best_max_features = None
best_train_error = None

for max_features in range(1, 8):
    clf = RandomForestClassifier(criterion='entropy', random_state=0, max_samples=best_max_samples, max_features=max_features)

    train_error, test_error = error(clf, X, y)

    if test_error < best_test_error:
        best_test_error = test_error
        best_train_error = train_error
        best_max_features = max_features

print(f"Best setting for max_features: {best_max_features}")
print(f"Training Error for this setting: {best_train_error}")
print(f"Test Error for this setting: {best_test_error}")
### ========== TODO : END ========== ###
```

✓ 1m 29.9s

```
Classifying using Random Forest...
Best setting for max_features: 3
Training Error for this setting: 0.10244288224956065
Test Error for this setting: 0.18727272727272727
```

In [ ]:
```
import os
import sys
```

In [ ]:
```
# To add your own Drive Run this cell.
from google.colab import drive
drive.mount('/content/drive')
```

In [ ]:
```
# Please append your own directory after '/content/drive/My Drive/'
### ========== TODO : START ========== ###
sys.path += ['/content/drive/My Drive/path_to_your_code']
### ========== TODO : END ========== ###
```

In [ ]:
```
"""
Author      : Yi-Chieh Wu, Sriram Sankararman
Description : Twitter
"""

from string import punctuation

import numpy as np
import matplotlib.pyplot as plt
# !!! MAKE SURE TO USE LinearSVC.decision_function(X), NOT LinearSVC.pred
# (this makes ''continuous-valued'' predictions)
from sklearn.svm import LinearSVC
from sklearn.model_selection import StratifiedKFold
from sklearn import metrics
```

Problem 3: Twitter Analysis Using SVM

In [ ]:
```
######################################################################
# functions -- input/output
######################################################################

def read_vector_file(fname):
    """
    Reads and returns a vector from a file.

    Parameters
    --------------------
        fname  -- string, filename

    Returns
    --------------------
        labels -- numpy array of shape (n,)
                    n is the number of non-blank lines in the text file
    """
    return np.genfromtxt(fname)


def write_label_answer(vec, outfile):
    """
    Writes your label vector to the given file.

    Parameters
    --------------------
        vec     -- numpy array of shape (n,) or (n,1), predicted scores
        outfile -- string, output filename
    """

    # for this project, you should predict 70 labels
    if(vec.shape[0] != 70):
        print("Error - output vector should have 70 rows.")
        print("Aborting write.")
        return

    np.savetxt(outfile, vec)
```

In [ ]:
```
######################################################################
# functions -- feature extraction
######################################################################

def extract_words(input_string):
    """
    Processes the input_string, separating it into "words" based on the p
    of spaces, and separating punctuation marks into their own words.

    Parameters
    --------------------
        input_string -- string of characters

    Returns
    --------------------
        words        -- list of lowercase "words"
    """
```

```
    for c in punctuation :
        input_string = input_string.replace(c, ' ' + c + ' ')
    return input_string.lower().split()


def extract_dictionary(infile):
    """
    Given a filename, reads the text file and builds a dictionary of uniq
    words/punctuations.

    Parameters
    --------------------
        infile    -- string, filename

    Returns
    --------------------
        word_list -- dictionary, (key, value) pairs are (word, index)
    """

    word_list = {}
    idx = 0
    with open(infile, 'r') as fid :
        # process each line to populate word_list
        for input_string in fid:
            words = extract_words(input_string)
            for word in words:
                if word not in word_list:
                    word_list[word] = idx
                    idx += 1
    return word_list


def extract_feature_vectors(infile, word_list):
    """
    Produces a bag-of-words representation of a text file specified by th
    filename infile based on the dictionary word_list.

    Parameters
    --------------------
        infile         -- string, filename
        word_list      -- dictionary, (key, value) pairs are (word, index

    Returns
    --------------------
        feature_matrix -- numpy array of shape (n,d)
                          boolean (0,1) array indicating word presence in
                            n is the number of non-blank lines in the tex
                            d is the number of unique words in the text f
    """

    num_lines = sum(1 for line in open(infile,'r'))
    num_words = len(word_list)
    feature_matrix = np.zeros((num_lines, num_words))

    with open(infile, 'r') as fid :
        # process each line to populate feature_matrix
```

```
        for i, input_string in enumerate(fid):
            words = extract_words(input_string)
            for word in words:
                feature_matrix[i, word_list[word]] = 1.0

    return feature_matrix
```

In [ ]:

```
######################################################################
# functions -- evaluation
######################################################################

def performance(y_true, y_pred, metric="accuracy"):
    """
    Calculates the performance metric based on the agreement between the
    true labels and the predicted labels.

    Parameters
    --------------------
        y_true -- numpy array of shape (n,), known labels
        y_pred -- numpy array of shape (n,), (continuous-valued) predicti
        metric -- string, option used to select the performance measure
                  options: 'accuracy', 'f1-score', 'auroc', 'precision',
                           'sensitivity', 'specificity'

    Returns
    --------------------
        score  -- float, performance score
    """
    # map continuous-valued predictions to binary labels
    y_label = np.sign(y_pred)
    y_label[y_label==0] = 1

    ### ========== TODO : START ========== ###
    # part 1a: compute classifier performance

    if metric == "accuracy":
        score = metrics.accuracy_score(y_true, y_pred)
    elif metric == "f1-score":
        score = metrics.f1_score(y_true, y_pred)
    elif metric == "auroc":
        score = metrics.roc_auc_score(y_true, y_pred)
    elif metric == "precision":
        score = metrics.precision_score(y_true, y_pred)
    elif metric == "sensitivity": #same as recall
        score = metrics.recall_score(y_true, y_pred)
    elif metric == "specificity":
        tn, fp, _, _ = metrics.confusion_matrix(y_true, y_label).ravel()
        return tn / (tn + fp)

    return score
    ### ========== TODO : END ========== ###


def cv_performance(clf, X, y, kf, metric="accuracy"):
    """
    Splits the data, X and y, into k-folds and runs k-fold cross-validati
```

```
    Trains classifier on k-1 folds and tests on the remaining fold.
    Calculates the k-fold cross-validation performance metric for classif
    by averaging the performance across folds.

    Parameters
    --------------------
        clf    -- classifier (instance of LinearSVC)
        X      -- numpy array of shape (n,d), feature vectors
                    n = number of examples
                    d = number of features
        y      -- numpy array of shape (n,), binary labels {1,-1}
        kf     -- model_selection.StratifiedKFold
        metric -- string, option used to select performance measure

    Returns
    --------------------
        score   -- float, average cross-validation performance across k f
    """

    ### ========== TODO : START ========== ###
    # part 1b: compute average cross-validation performance
    scores = []

    for train_index, test_index in kf.split(X, y):
        X_train, X_test = X[train_index], X[test_index]
        y_train, y_test = y[train_index], y[test_index]

        clf.fit(X_train, y_train)

        y_scores = clf.decision_function(X_test)

        if metric == "auroc":
            score = performance(y_test, y_scores, metric)
        else:
            y_pred = np.sign(y_scores)
            y_pred[y_pred == 0] = 1
            score = performance(y_test, y_pred, metric)

        scores.append(score)

    return np.mean(scores)
    ### ========== TODO : END ========== ###


def select_param_linear(X, y, kf, metric="accuracy"):
    """
    Sweeps different settings for the hyperparameter of a linear SVM,
    calculating the k-fold CV performance for each setting, then selectin
    hyperparameter that 'maximize' the average k-fold CV performance.

    Parameters
    --------------------
        X      -- numpy array of shape (n,d), feature vectors
                    n = number of examples
                    d = number of features
        y      -- numpy array of shape (n,), binary labels {1,-1}
        kf     -- model_selection.StratifiedKFold
```

```
        metric -- string, option used to select performance measure

    Returns
    --------------------
        C -- float, optimal parameter value for linear SVM
    """

    print('Linear SVM Hyperparameter Selection based on ' + str(metric) +
    C_range = 10.0 ** np.arange(-3, 3)

    ### ========== TODO : START ========== ###
    # part 1c: select optimal hyperparameter using cross-validation

    best_score = 0
    best_C = None

    for C in C_range:
        clf = LinearSVC(loss = 'hinge', random_state=0, C=C)
        score = cv_performance(clf, X, y, kf, metric)
        if score > best_score:
            best_score = score
            best_C = C

    print("For {}: Best C = {:.3f}. Best Score = {:.3f}".format(metric, b

    return best_C
    ### ========== TODO : END ========== ###


def performance_test(clf, X, y, metric="accuracy"):
    """
    Estimates the performance of the classifier.

    Parameters
    --------------------
        clf          -- classifier (instance of LinearSVC)
                          [already fit to data]
        X            -- numpy array of shape (n,d), feature vectors of te
                          n = number of examples
                          d = number of features
        y            -- numpy array of shape (n,), binary labels {1,-1} o
        metric       -- string, option used to select performance measure

    Returns
    --------------------
        score        -- float, classifier performance
    """


    ### ========== TODO : START ========== ###
    # part 2b: return performance on test data under a metric.
    y_scores = clf.decision_function(X)

    if metric != "auroc":
        y_pred = np.sign(y_scores)
        y_pred[y_pred == 0] = 1
    else:
```

```
        y_pred = y_scores

    # Use the performance function to calculate the score
    score = performance(y, y_pred, metric)
    return score

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

In [ ]:

```
######################################################################
# main
######################################################################

def main() :
    np.random.seed(1234)

    # read the tweets and its labels, change the following two lines to y
    ### ========== TODO : START ========== ###
    file_path = '../data/tweets.txt'
    label_path = '../data/labels.txt'
    ### ========== TODO : END ========== ###
    dictionary = extract_dictionary(file_path)
    print(len(dictionary))
    X = extract_feature_vectors(file_path, dictionary)
    y = read_vector_file(label_path)
    # split data into training (training + cross-validation) and testing
    X_train, X_test = X[:560], X[560:]
    y_train, y_test = y[:560], y[560:]

    metric_list = ["accuracy", "f1-score", "auroc", "precision", "sensiti

    ### ========== TODO : START ========== ###
    # part 1b: create stratified folds (5-fold CV)
    kf = StratifiedKFold(n_splits = 5, shuffle = True, random_state = 0)

    # part 1c: for each metric, select optimal hyperparameter for linear
    best_C_values = {}
    for metric in metric_list:
        best_C_values[metric] = select_param_linear(X_train, y_train, kf,

    # part 2a: train linear SVMs with selected hyperparameters
    classifiers = {}
    for metric, C in best_C_values.items():
        clf = LinearSVC(loss='hinge', random_state=0, C=C)
        clf.fit(X_train, y_train)
        classifiers[metric] = clf

    # part 2b: test the performance of your classifiers.
    for metric, clf in classifiers.items():
        score = performance_test(clf, X_test, y_test, metric)
        print(f"Performance for {metric} with C={C}: {score}")
    ### ========== TODO : END ========== ###


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

```
1811
Linear SVM Hyperparameter Selection based on accuracy:
For accuracy: Best C = 10.000. Best Score = 0.829
Linear SVM Hyperparameter Selection based on f1-score:
For f1-score: Best C = 10.000. Best Score = 0.883
Linear SVM Hyperparameter Selection based on auroc:
For auroc: Best C = 10.000. Best Score = 0.895
Linear SVM Hyperparameter Selection based on precision:
For precision: Best C = 10.000. Best Score = 0.856
Linear SVM Hyperparameter Selection based on sensitivity:
For sensitivity: Best C = 0.001. Best Score = 1.000
Linear SVM Hyperparameter Selection based on specificity:
For specificity: Best C = 10.000. Best Score = 0.625
Performance for accuracy with C=10.0: 0.7428571428571429
Performance for f1-score with C=10.0: 0.43749999999999994
Performance for auroc with C=10.0: 0.7453838678328474
Performance for precision with C=10.0: 0.6363636363636364
Performance for sensitivity with C=10.0: 1.0
Performance for specificity with C=10.0: 0.9183673469387755
```

Problem 4: Boosting vs. Decision Tree

In [ ]:

```
from sklearn.tree import DecisionTreeClassifier
from sklearn.ensemble import RandomForestClassifier
from sklearn import metrics
from sklearn.model_selection import cross_val_score, train_test_split
```

In [ ]:

```
class Data :

    def __init__(self) :
        """
        Data class.

        Attributes
        --------------------
            X -- numpy array of shape (n,d), features
            y -- numpy array of shape (n,), targets
        """

        # n = number of examples, d = dimensionality
        self.X = None
        self.y = None

        self.Xnames = None
        self.yname = None

    def load(self, filename, header=0, predict_col=-1) :
        """Load csv file into X array of features and y array of labels."

        # determine filename
        f = filename

        # load data
        with open(f, 'r') as fid :
```

```
            data = np.loadtxt(fid, delimiter=",", skiprows=header)

        # separate features and labels
        if predict_col is None :
            self.X = data[:,:]
            self.y = None
        else :
            if data.ndim > 1 :
                self.X = np.delete(data, predict_col, axis=1)
                self.y = data[:,predict_col]
            else :
                self.X = None
                self.y = data[:]

        # load feature and label names
        if header != 0:
            with open(f, 'r') as fid :
                header = fid.readline().rstrip().split(",")

            if predict_col is None :
                self.Xnames = header[:]
                self.yname = None
            else :
                if len(header) > 1 :
                    self.Xnames = np.delete(header, predict_col)
                    self.yname = header[predict_col]
                else :
                    self.Xnames = None
                    self.yname = header[0]
        else:
            self.Xnames = None
            self.yname = None


# helper functions
def load_data(filename, header=0, predict_col=-1) :
    """Load csv file into Data class."""
    data = Data()
    data.load(filename, header=header, predict_col=predict_col)
    return data
```

In [ ]:

```
# Change the path to your own data directory
### ========== TODO : START ========== ###
titanic = load_data("../data/titanic_train.csv", header=1, predict_col=0)
### ========== TODO : END ========== ###
X = titanic.X; Xnames = titanic.Xnames
y = titanic.y; yname = titanic.yname
n,d = X.shape  # n = number of examples, d =  number of features
```

In [ ]:
```
def error(clf, X, y, ntrials=100, test_size=0.2) :
    """
    Computes the classifier error over a random split of the data,
    averaged over ntrials runs.

    Parameters
    --------------------
        clf         -- classifier
        X           -- numpy array of shape (n,d), features values
        y           -- numpy array of shape (n,), target classes
        ntrials     -- integer, number of trials
        test_size   -- proportion of data used for evaluation

    Returns
    --------------------
        train_error -- float, training error
        test_error  -- float, test error
    """

    train_error = 0
    test_error = 0

    train_scores = []; test_scores = [];
    for i in range(ntrials):
        xtrain, xtest, ytrain, ytest = train_test_split (X,y, test_size =
        clf.fit (xtrain, ytrain)

        ypred = clf.predict (xtrain)
        err = 1 - metrics.accuracy_score (ytrain, ypred, normalize = True
        train_scores.append (err)

        ypred = clf.predict (xtest)
        err = 1 - metrics.accuracy_score (ytest, ypred, normalize = True)
        test_scores.append (err)

    train_error =  np.mean (train_scores)
    test_error = np.mean (test_scores)
    return train_error, test_error
```

In [ ]:
```
### ========== TODO : START ========== ###
# Part 4(a): Implement the decision tree classifier and report the traini
print('Classifying using Decision Tree...')
clf = DecisionTreeClassifier(criterion='entropy', random_state=0)
clf.fit(X, y)
ypred_train = clf.predict(X)
train_error = 1 - metrics.accuracy_score(y, ypred_train)
print(f"Training Error: {train_error}")
train_error, test_error = error(clf, X, y)
print(f"Average Training Error after 100 trials: {train_error}")
print(f"Average Test Error after 100 trials: {test_error}")
### ========== TODO : END ========== ###
```

```
Classifying using Decision Tree...
Training Error: 0.014044943820224698
Average Training Error after 100 trials: 0.011528998242530775
Average Test Error after 100 trials: 0.24104895104895108
```

In [ ]:
```
### ========== TODO : START ========== ###
# Part 4(b): Implement the random forest classifier and adjust the number
print('Classifying using Random Forest...')
n_samples = len(X)  # Total number of samples in your data
max_samples_options = [0.1, 0.2, 0.3, 0.4, 0.5, 0.6, 0.7, 0.8]

best_test_error = float('inf')
best_max_samples = None
best_train_error = None

for max_samples_ratio in max_samples_options:
    max_samples = int(n_samples * max_samples_ratio)
    clf = RandomForestClassifier(criterion='entropy', random_state=0, max

    # Use the provided error function to get training and test error
    train_error, test_error = error(clf, X, y)

    if test_error < best_test_error:
        best_test_error = test_error
        best_train_error = train_error
        best_max_samples = max_samples

print(f"Best setting for max_samples: {best_max_samples} samples")
print(f"Training Error for this setting: {best_train_error}")
print(f"Test Error for this setting: {best_test_error}")
### ========== TODO : END ========== ###
```

```
Classifying using Random Forest...
Best setting for max_samples: 142 samples
Training Error for this setting: 0.10314586994727591
Test Error for this setting: 0.18797202797202794
```

In [ ]:
```
### ========== TODO : START ========== ###
# Part 4(c): Implement the random forest classifier and adjust the number
print('Classifying using Random Forest...')
best_max_samples = 142

best_test_error = float('inf')
best_max_features = None
best_train_error = None

for max_features in range(1, 8):
    clf = RandomForestClassifier(criterion='entropy', random_state=0, max

    train_error, test_error = error(clf, X, y)

    if test_error < best_test_error:
        best_test_error = test_error
        best_train_error = train_error
        best_max_features = max_features

print(f"Best setting for max_features: {best_max_features}")
print(f"Training Error for this setting: {best_train_error}")
print(f"Test Error for this setting: {best_test_error}")
### ========== TODO : END ========== ###
```

```
Classifying using Random Forest...
Best setting for max_features: 3
Training Error for this setting: 0.10244288224956065
Test Error for this setting: 0.1872727272727273
```