# HW3_Jacob_Sayono

May 25, 2024

Jacob Sayono

505368811

CS M146 HW 3

## 1 Problem 1

**(a)** TRUE

In Support Vector Machines (SVM), only a subset of the training data affects the decision boundary. These data points are known as support vectors. Data points that are not support vectors do not influence the hyperplane parameters (i.e., they are not on the margin or do not violate the margin). Therefore, removing these non-support vector points from the training dataset would not change the solution of the SVM, implying that we can indeed eliminate some training points and still get the same solution.

**(b)** TRUE

In a soft-margin SVM formulation, the objective is to find the hyperplane parameters $(w, b)$ and the slack variables $\xi_i$ for each data sample. Here, $w$ is a $d$-dimensional vector (one variable for each feature), $b$ is the bias term (one variable), and $\xi_i$ are the slack variables (one for each of the $n$ samples). Thus, the total number of variables optimized is $d + 1 + n$.

**(c)** TRUE

AdaBoost adjusts the weights of training samples based on their classification in the previous rounds. Samples that are consistently correctly classified receive exponentially decreasing weights, as the algorithm focuses more on the harder-to-classify samples. Therefore, samples that have been correctly classified in all previous rounds end up with the lowest weights, as their correct classification suggests that they are 'easy' cases for the classifiers.

**(d)**

- **(i)** FALSE. Samples with $\xi_i = 0$ are correctly classified and outside the margin. They do not affect the positioning of the hyperplane and thus are not support vectors.
- **(ii-iv)** TRUE. Samples with $0 < \xi_i$ are on the wrong side of the margin but still contribute to the loss term in the SVM objective. These include samples within the margin $(0 < \xi_i \leq 1)$ and those misclassified $(\xi_i > 1)$. Since they influence the loss term, they are support vectors.

**(e)**

- **(i)** FALSE. AdaBoost assigns a lower weight to samples that were correctly classified, as it focuses on correcting misclassifications.
- **(ii)** FALSE. AdaBoost assigns a higher weight to weak classifiers with higher error rates during classifier combination, not lower.
- **(iii)** FALSE. Samples that were incorrectly classified in the previous round are assigned higher weights, but not necessarily equal weights, as their weights depend on their individual errors and the performance of the classifier.
- **(iv)** TRUE. Since all the other options are false, this is the correct answer.

## 2 Problem 2

**(a)** YES

The optimization problem P1, defined as $\min_w \|w\|^2 + \lambda \sum_{i=1}^m \xi_i^2$, subject to $y_i w^T x_i \geq 1 - \xi_i$ and $\xi_i \geq 0$ for all $i \in [m]$, is convex. The objective function $\|w\|^2$ is convex as it is a quadratic form. The term $\lambda \sum_{i=1}^m \xi_i^2$ is also convex since the sum of convex functions (squared terms) remains convex. The constraints are linear, thus preserving convexity. Therefore, both the objective and constraints are convex, making the entire optimization problem convex.

**(b)**

$$L(w, \xi, \alpha, \beta) = \|w\|^2 + \lambda \sum_{i=1}^m \xi_i^2 + \sum_{i=1}^m \alpha_i (1 - \xi_i - y_i w^T x_i) + \sum_{i=1}^m \beta_i (-\xi_i)$$

Where $\alpha$ and $\beta$ are the vectors of Lagrange multipliers for the inequality constraints $y_i w^T x_i \geq 1 - \xi_i$ and $\xi_i \geq 0$, respectively.

**(c)**

$$\frac{\partial L}{\partial w} = 2w - \sum_{i=1}^m \alpha_i y_i x_i$$

Set this derivative to zero to solve for $w$ in terms of $\alpha$:

$$2w = \sum_{i=1}^m \alpha_i y_i x_i \implies w = \frac{1}{2} \sum_{i=1}^m \alpha_i y_i x_i$$

**(d)** To find the derivative of the Lagrangian with respect to $\xi_i$, compute:

$$\frac{\partial L}{\partial \xi_i} = 2\lambda \xi_i - \alpha_i - \beta_i$$

Set this derivative to zero to solve for $\xi_i$ in terms of $\alpha_i$ and $\beta_i$:

$$2\lambda \xi_i = \alpha_i + \beta_i \implies \xi_i = \frac{\alpha_i + \beta_i}{2\lambda}$$

**(e)** Using the solutions for $w$ and $\xi_i$ found in parts (c) and (d), substitute these back into the Lagrangian:

$$L\left(\frac{1}{2}\sum_{i=1}^{m}\alpha_i y_i x_i, \frac{\alpha_i+\beta_i}{2\lambda}, \alpha, \beta\right) = \left\|\frac{1}{2}\sum_{i=1}^{m}\alpha_i y_i x_i\right\|^2 + \lambda\sum_{i=1}^{m}\left(\frac{\alpha_i+\beta_i}{2\lambda}\right)^2 + \text{substituting constraints}$$

Evaluating this function gives us the dual function $g(\alpha, \beta)$ which needs to be maximized with respect to $\alpha$ and $\beta$ under the conditions $\alpha_i, \beta_i \geq 0$.

**(f)** The primal variable $w$ can be expressed as a linear combination of the training examples $x_i$, weighted by the corresponding $\alpha_i y_i$. The dual problem does not depend explicitly on the feature vectors $x_i$ but only through dot products between pairs of $x_i$. Thus, we can replace these dot products by a kernel function $K(x_i, x_j) = \phi(x_i)^T\phi(x_j)$, which computes dot products in a potentially higher-dimensional space without explicitly mapping the vectors to this space. This makes the solution kernelizable, allowing the use of the kernel trick for non-linear classification boundaries.

## 3 Problem 3

**Part (a) and (b)** Table 1.

| i | $x_1$ | $x_2$ | Label | $w_0$ | $h_1 \equiv \epsilon_1$ | $\beta_1$ | $w_1$ | $h_2 \equiv \epsilon_2$ | $\beta_2$ |
|---|---|---|---|---|---|---|---|---|---|
| 1 | 2 | -1 | + | 1/5 | 0.4 | 0.203 | 0.25 | 0.5 | 0.0 |
| 2 | 3 | 1 | + | 1/5 | 0.4 | 0.203 | 0.25 | 0.5 | 0.0 |
| 3 | 2 | 4 | - | 1/5 | 0.4 | 0.203 | 0.167 | 0.5 | 0.0 |
| 4 | 1 | 4 | - | 1/5 | 0.4 | 0.203 | 0.167 | 0.5 | 0.0 |
| 5 | -1 | 3 | - | 1/5 | 0.4 | 0.203 | 0.167 | 0.5 | 0.0 |

[ ]:
```python
# part (a) and (b)

import numpy as np

x1 = np.array([2, 3, 2, 1, -1])
x2 = np.array([-1, 1, 4, 4, 3])
labels = np.array([1, 1, -1, -1, -1])
w0 = np.array([0.2, 0.2, 0.2, 0.2, 0.2])

def hypothesis(x1, x2, theta):
    return np.sign(-2 * x1 + x2 - theta)

def weighted_error(theta, x1, x2, labels, weights):
    predictions = hypothesis(x1, x2, theta)
    errors = (predictions != labels).astype(float)
    weighted_errors = weights @ errors
```

```
    return weighted_errors

# first iteration calculations
theta_values = np.linspace(-10, 10, 400)
errors = [weighted_error(theta, x1, x2, labels, w0) for theta in theta_values]
min_error_idx = np.argmin(errors)
theta_optimal1 = theta_values[min_error_idx]
epsilon1 = errors[min_error_idx]
beta1 = 0.5 * np.log((1 - epsilon1) / epsilon1)
w1 = w0 * np.exp(-beta1 * labels * hypothesis(x1, x2, theta_optimal1))
w1 /= np.sum(w1)

print(f"Firt Iteration - Optimal Theta: {theta_optimal1}")
print(f"Firt Iteration - Weighted Error (epsilon1): {epsilon1}")
print(f"Firt Iteration - Coefficient (beta1): {beta1}")
print(f"Firt Iteration - Updated Weights (w1): {w1}")

# second iteration calculations
errors = [weighted_error(theta, x1, x2, labels, w1) for theta in theta_values]
min_error_idx = np.argmin(errors)
theta_optimal2 = theta_values[min_error_idx]
epsilon2 = errors[min_error_idx]
beta2 = 0.5 * np.log((1 - epsilon2) / epsilon2)
w2 = w1 * np.exp(-beta2 * labels * hypothesis(x1, x2, theta_optimal2))
w2 /= np.sum(w2)

print(f"Second Iteration - Optimal Theta: {theta_optimal2}")
print(f"Second Iteration - Weighted Error (epsilon2): {epsilon2}")
print(f"Second Iteration - Coefficient (beta2): {beta2}")
print(f"Second Iteration - Updated Weights (w2): {w2}")
```

```
Firt Iteration - Optimal Theta: 5.037593984962406
Firt Iteration - Weighted Error (epsilon1): 0.4
Firt Iteration - Coefficient (beta1): 0.2027325540540821
Firt Iteration - Updated Weights (w1): [0.25       0.25       0.16666667
0.16666667 0.16666667]
Second Iteration - Optimal Theta: -10.0
Second Iteration - Weighted Error (epsilon2): 0.5
Second Iteration - Coefficient (beta2): 0.0
Second Iteration - Updated Weights (w2): [0.25       0.25       0.16666667
0.16666667 0.16666667]
```

**Part (c) and (d)** Table 2.

Due to the lack of changes in the weight distribution from the first iteration and the practical limits of updating in case of a perfect classification, the second iteration could not effectively proceed.

| i | $x_1$ | $x_2$ | Label | $w_0$ | $h_1 \equiv \epsilon_1$ | $\beta_1$ | $w_1$ | $h_2 \equiv \epsilon_2$ | $\beta_2$ |
|---|---|---|---|---|---|---|---|---|---|
| 1 | 2 | -1 | + | 1/5 | 0.0 | $\infty$ | 1/5 | | |
| 2 | 3 | 1 | + | 1/5 | 0.0 | $\infty$ | 1/5 | | |
| 3 | 2 | 4 | - | 1/5 | 0.0 | $\infty$ | 1/5 | | |
| 4 | 1 | 4 | - | 1/5 | 0.0 | $\infty$ | 1/5 | | |
| 5 | -1 | 3 | - | 1/5 | 0.0 | $\infty$ | 1/5 | | |

[ ]:
```python
# part (c) and (d)

import numpy as np

x1 = np.array([2, 3, 2, 1, -1])
x2 = np.array([-1, 1, 4, 4, 3])
labels = np.array([1, 1, -1, -1, -1])
w0 = np.array([0.2, 0.2, 0.2, 0.2, 0.2])

def hypothesis_h2(x1, x2, theta):
    return np.sign(2 * x1 - x2 - theta)

def weighted_error_h2(theta, x1, x2, labels, weights):
    predictions = hypothesis_h2(x1, x2, theta)
    errors = (predictions != labels).astype(float)
    weighted_errors = weights @ errors
    return weighted_errors

def adaboost_iteration(x1, x2, labels, w, theta_values):
    errors = [weighted_error_h2(theta, x1, x2, labels, w) for theta in␣
 ↪theta_values]
    min_error_idx = np.argmin(errors)
    theta_optimal = theta_values[min_error_idx]
    epsilon = errors[min_error_idx]
    beta = 0.5 * np.log((1 - epsilon) / epsilon)
    w_new = w * np.exp(-beta * labels * hypothesis_h2(x1, x2, theta_optimal))
    w_new /= np.sum(w_new)
    return theta_optimal, epsilon, beta, w_new

# first iteration
theta_values = np.linspace(-10, 10, 400)
theta1, epsilon1, beta1, w1 = adaboost_iteration(x1, x2, labels, w0,␣
 ↪theta_values)

# second iteration
theta2, epsilon2, beta2, w2 = adaboost_iteration(x1, x2, labels, w1,␣
 ↪theta_values)

print(f"First Iteration - Optimal Theta: {theta1}")
```

```
print(f"First Iteration - Weighted Error (epsilon1): {epsilon1}")
print(f"First Iteration - Coefficient (beta1): {beta1}")
print(f"First Iteration - Updated Weights (w1): {w1}")

print(f"Second Iteration - Optimal Theta: {theta2}")
print(f"Second Iteration - Weighted Error (epsilon2): {epsilon2}")
print(f"Second Iteration - Coefficient (beta2): {beta2}")
print(f"Second Iteration - Updated Weights (w2): {w2}")
```

```
First Iteration - Optimal Theta: 0.025062656641603454
First Iteration - Weighted Error (epsilon1): 0.0
First Iteration - Coefficient (beta1): inf
First Iteration - Updated Weights (w1): [nan nan nan nan nan]
Second Iteration - Optimal Theta: -10.0
Second Iteration - Weighted Error (epsilon2): nan
Second Iteration - Coefficient (beta2): nan
Second Iteration - Updated Weights (w2): [nan nan nan nan nan]
```

```
/tmp/ipykernel_7536/1025930726.py:24: RuntimeWarning: divide by zero encountered
in double_scalars
  beta = 0.5 * np.log((1 - epsilon) / epsilon)
/tmp/ipykernel_7536/1025930726.py:26: RuntimeWarning: invalid value encountered
in divide
  w_new /= np.sum(w_new)
```

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

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

[ ]:
```
# Please append your own directory after '/content/drive/My Drive/'
### ========== TODO : START ========== ###
sys.path += ['/content/drive/My Drive/cm146-spring23/hw3/HW3-code']
### ========== TODO : END ========== ###
```

[ ]:
```
"""
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.predict(X)
↪!!!
```

```
# (this makes ''continuous-valued'' predictions)
from sklearn.svm import LinearSVC
from sklearn.model_selection import StratifiedKFold
from sklearn import metrics
```

## 4 Problem 4: Twitter Analysis Using SVM

[ ]:
```
##########################################################################
# 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)
```

[ ]:
```
import string
##################################################################
# functions -- feature extraction
##################################################################

def extract_words(input_string):
    """
    Processes the input_string, separating it into "words" based on the presence
    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 string.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 unique
    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 the
    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 a␣
 ↪string
                            n is the number of non-blank lines in the text file
                            d is the number of unique words in the text file
    """

    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
```

[ ]:
```
from sklearn.metrics import accuracy_score, f1_score, roc_auc_score,␣
 ↪precision_score, recall_score, confusion_matrix
from sklearn.model_selection import cross_val_score
from sklearn.svm import LinearSVC
import numpy as np

#####################################################################
# 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":
        return accuracy_score(y_true, y_label)
    elif metric == "f1-score":
        return f1_score(y_true, y_label)
    elif metric == "auroc":
        return roc_auc_score(y_true, y_pred)  # AUROC expects score, not labels
    elif metric == "precision":
        return precision_score(y_true, y_label)
    elif metric == "sensitivity":
        return recall_score(y_true, y_label)
    elif metric == "specificity":
        tn, fp, _, _ = confusion_matrix(y_true, y_label).ravel()
        return tn / (tn + fp)
    else:
        raise ValueError("Unknown metric.")
    ### ========== TODO : END ========== ###

from sklearn.metrics import make_scorer
def specificity_score(y_true, y_pred):
    tn, fp, _, _ = confusion_matrix(y_true, y_pred).ravel()
    return tn / (tn + fp)

specificity_scorer = make_scorer(specificity_score, greater_is_better=True)
```

```
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
    if metric == "specificity":
        score = cross_val_score(clf, X, y, scoring=specificity_scorer, cv=kf)
    else:
        score = cross_val_score(clf, X, y, scoring=metric, cv=kf)
    return score.mean()
    ### ========== 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 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 = -1
    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=metric)
        if score > best_score:
            best_score = score
            best_C = C

    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 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 in ["accuracy", "f1", "precision", "recall", "specificity"]:
        y_pred = np.sign(y_scores)
        y_pred[y_pred == 0] = 1  # Treat zero as positive

        if metric == "accuracy":
            return accuracy_score(y, y_pred)
        elif metric == "f1":
            return f1_score(y, y_pred)
        elif metric == "precision":
            return precision_score(y, y_pred)
        elif metric == "recall":
            return recall_score(y, y_pred)
        elif metric == "specificity":
            tn, fp, _, _ = confusion_matrix(y, y_pred).ravel()
            return tn / (tn + fp)
    elif metric == "roc_auc":
        return roc_auc_score(y, y_scores)  # AUROC expects scores, not binary␣
 ↪labels
    else:
        raise ValueError("Unknown metric: {}".format(metric))
    ### ========== TODO : END ========== ###
```

[ ]:
```
from sklearn.model_selection import StratifiedKFold
#####################################################################
# 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", "roc_auc", "precision", "recall",␣
↪"specificity"]

    ### ========== TODO : START ========== ###
    # part 1b: create stratified folds (5-fold CV)
    kf = StratifiedKFold(n_splits=5, shuffle=True, random_state=1234)
    best_C = {}

    # part 1c: for each metric, select optimal hyperparameter for linear SVM␣
↪using CV
    for metric in metric_list:
        optimal_C = select_param_linear(X_train, y_train, kf, metric=metric)
        best_C[metric] = optimal_C
        print(f"Best C for {metric}: {optimal_C}")

    # part 2a: train linear SVMs with selected hyperparameters
    classifiers = {}
    for metric, C in best_C.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.
    performances = {}
    for metric, clf in classifiers.items():
        score = performance_test(clf, X_test, y_test, metric=metric)
        performances[metric] = score
        print(f"Performance on test data for {metric}: {score}")

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


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

```
1811
Linear SVM Hyperparameter Selection based on accuracy:
Best C for accuracy: 10.0
Linear SVM Hyperparameter Selection based on f1:
Best C for f1: 10.0
Linear SVM Hyperparameter Selection based on roc_auc:
Best C for roc_auc: 10.0
Linear SVM Hyperparameter Selection based on precision:
Best C for precision: 10.0
Linear SVM Hyperparameter Selection based on recall:
Best C for recall: 0.001
Linear SVM Hyperparameter Selection based on specificity:
```

```
Best C for specificity: 10.0
Performance on test data for accuracy: 0.7428571428571429
Performance on test data for f1: 0.43749999999999994
Performance on test data for roc_auc: 0.7453838678328474
Performance on test data for precision: 0.6363636363636364
Performance on test data for recall: 1.0
Performance on test data for specificity: 0.9183673469387755
```

## 5 Problem 5: Boosting vs. Decision Tree

[ ]:
```
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
```

[ ]:
```
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
```

[ ]:
```
# 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
```

[ ]:
```
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 =␣
 ↪test_size, random_state = i)
        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
```

[ ]:
```
### ========== TODO : START ========== ###
# Part 4(a): Implement the decision tree classifier and report the training␣
 ↪error.
print('Classifying using Decision Tree...')
dt_clf = DecisionTreeClassifier(criterion='entropy', random_state=0)
dt_clf.fit(X, y)

y_pred_train = dt_clf.predict(X)
train_error = 1 - metrics.accuracy_score(y, y_pred_train)
print(f"Training Error (DT): {train_error}")
```

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

```
Classifying using Decision Tree…
Training Error (DT): 0.014044943820224698
```

[ ]:
```
### ========== 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...')
best_test_error = float('inf')
best_max_samples = None

for max_samples_percent in range(10, 90, 10):  # From 10% to 80%
    max_samples = int(n * (max_samples_percent / 100))
    rf_clf = RandomForestClassifier(criterion='entropy', random_state=0,␣
 ↪max_samples=max_samples)
    train_error, test_error = error(rf_clf, X, y)
    print(f"Max Samples {max_samples_percent}%: Train Error = {train_error},␣
 ↪Test Error = {test_error}")

    if test_error < best_test_error:
        best_test_error = test_error
        best_max_samples = max_samples_percent

print(f"Best Test Error: {best_test_error} at {best_max_samples}% max_samples")
### ========== TODO : END ========== ###
```

```
Classifying using Random Forest…
Max Samples 10%: Train Error = 0.1357293497363796, Test Error =
0.19587412587412587
Max Samples 20%: Train Error = 0.10314586994727591, Test Error =
0.18797202797202794
Max Samples 30%: Train Error = 0.0818629173989455, Test Error =
0.18888111888111891
Max Samples 40%: Train Error = 0.05869947275922671, Test Error =
0.19216783216783218
Max Samples 50%: Train Error = 0.03388400702987697, Test Error =
0.19888111888111892
Max Samples 60%: Train Error = 0.017785588752196824, Test Error =
0.20111888111888113
Max Samples 70%: Train Error = 0.012390158172232001, Test Error =
0.20475524475524473
Max Samples 80%: Train Error = 0.011528998242530775, Test Error =
0.20671328671328676
Best Test Error: 0.18797202797202794 at 20% max_samples
```

[ ]:
```
### ========== 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_test_error = float('inf')
best_max_features = None

best_max_samples = int(n * (best_max_samples / 100))
for max_features in range(1, d + 1):  # d is the number of features
    rf_clf = RandomForestClassifier(criterion='entropy', random_state=0,␣
 ↪max_samples=best_max_samples, max_features=max_features)
    train_error, test_error = error(rf_clf, X, y)
    print(f"Max Features {max_features}: Train Error = {train_error}, Test␣
 ↪Error = {test_error}")

    if test_error < best_test_error:
        best_test_error = test_error
        best_max_features = max_features

print(f"Best Test Error: {best_test_error} at {best_max_features} max_features")
### ========== TODO : END ========== ###
```

```
Classifying using Random Forest…
Max Features 1: Train Error = 0.10121265377855888, Test Error =
0.18776223776223777
Max Features 2: Train Error = 0.10314586994727591, Test Error =
0.18797202797202794
Max Features 3: Train Error = 0.10244288224956065, Test Error =
0.1872727272727273
Max Features 4: Train Error = 0.10430579964850617, Test Error =
0.1874125874125874
Max Features 5: Train Error = 0.10544815465729351, Test Error =
0.1886013986013986
Max Features 6: Train Error = 0.10581722319859402, Test Error =
0.189020979020979
Max Features 7: Train Error = 0.10776801405975397, Test Error =
0.18895104895104897
Best Test Error: 0.1872727272727273 at 3 max_features
```