Assignment 3: Support Vector Machines and Ensemble Methods

Part 1: For the Breast Cancer Wisconsin (Diagnostic) Dataset, implement and train a SVM classifier from scratch for 10,000 iterations with hyper-parameter setting C = 1, lr = 0.002(C is the regularization parameter, lr is the learning rate), and use a sub-gradient. Report the F1 score, precision, and recall of the model on the training set and testing set. Next, use scikit-learns SVM package to train a SVM with linear, polynomial, and RBF kernels. Report F1 score, precision, and recall for all kernels.

Python script name on github: support_vector_machine.py

```
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt

from sklearn import metrics
from sklearn import svm


class SVM(object):
    """
    SVM Support Vector Machine
    with sub-gradient descent training.
    """
    def __init__(self, C=1):
        """
        Params
        ------
        n_features  : number of features (D)
        C           : regularization parameter (default: 1)
        """
        self.C = C

    def fit(self, X, y, lr=0.002, iterations=10000):
        """
        Fit the model using the training data.

        Params
        ------
        X           :   (ndarray, shape = (n_samples, n_features)):
                        Training input matrix where each row is a feature vector.
        y           :   (ndarray, shape = (n_samples,)):
                        Training target. Each entry is either -1 or 1.
        lr          :   learning rate
        iterations  :   number of iterations
        """
        n_samples, n_features = X.shape

        # Initialize the parameters wb
        self.wb =np.zeros((n_features+1))
        # initialize any container needed for save results during training
        obj = 0
        #initialize an objective list to see if model is converging
        obj_list = []
        for i in range(iterations):
            # calculate learning rate with iteration number i
            lr_t = lr/(np.sqrt(i+1))
            # calculate the subgradients
            subgrad = self.subgradient(self.wb, X, y)
            # update the parameter wb with the gradients
            self.wb = self.wb - (lr_t*subgrad)
            # calculate the new objective function value
            current_obj = self.objective(self.wb, X, y)
            # compare the current objective function value with the saved best value
            # update the best value if the current one is better
            if current_obj < obj:
                obj = current_obj
            # Logging
            #if (i+1)%1000 == 0:
            #    print(f"Training step {i+1:6d}: LearningRate[{lr_t:.7f}], Objective[{f:.7f}]")
            #append the objective list with the current_obj value
```

```
            obj_list.append(current_obj)
        # Save the best parameter found during training
        #I'm not really sure why we have to save the parameters again if the most previous update of the
parameters
        #was out last version of them
        self.wb = self.wb
        # optinally return recorded objective function values (for debugging purpose) to see
        # if the model is converging
        return obj_list

    @staticmethod
    def unpack_wb(wb, n_features):
        """
        Unpack wb into w and b
        """
        w = wb[:n_features]
        b = wb[-1]

        return (w,b)

    def g(self, X, wb):
        """
        Helper function for g(x) = WX+b
        """
        n_samples, n_features = X.shape

        w,b = self.unpack_wb(wb, n_features)
        gx = np.dot(w, X.T) + b

        return gx

    def hinge_loss(self, X, y, wb):
        """
        Hinge loss for max(0, 1 - y(Wx+b))

        Params
        ------

        Return
        ------
        hinge_loss
        hinge_loss_mask
        """
        hinge = 1 - y*(self.g(X, wb))
        hinge_mask = (hinge > 0).astype(np.int)
        hinge = hinge * hinge_mask

        return hinge, hinge_mask

    def objective(self, wb, X, y):
        """
        Compute the objective function for the SVM.

        Params
        ------
        X   :   (ndarray, shape = (n_samples, n_features)):
                Training input matrix where each row is a feature vector.
        y   :   (ndarray, shape = (n_samples,)):
                Training target. Each entry is either -1 or 1.
        Return
        ------
        obj (float): value of the objective function evaluated on X and y.
        """
        n_samples, n_features = X.shape
        #Hyperparameter C can be multiplied by objective function at different step

        # Calculate the objective function value
        # Be careful with the hinge loss function
        hinge_loss, hinge_mask = self.hinge_loss(X, y, wb)
        #calcualte w
        w = wb[:n_features]
        #calculate the objective function
        obj = np.sum(hinge_loss) + np.dot(w,w)

        return obj

    def subgradient(self, wb, X, y):
        """
        Compute the subgradient of the objective function.

        Params
```

```
        ------
        X   :   (ndarray, shape = (n_samples, n_features)):
                Training input matrix where each row is a feature vector.
        y   :   (ndarray, shape = (n_samples,)):
                Training target. Each entry is either -1 or 1.
        Return
        ------
        subgrad (ndarray, shape = (n_features+1,)):
                subgradient of the objective function with respect to
                the coefficients wb=[w,b] of the linear model
        """
        n_samples, n_features = X.shape
        w, b = self.unpack_wb(wb, n_features)

        # Retrieve the hinge mask
        hinge, hinge_mask = self.hinge_loss(X, y, wb)

        ## Cast hinge_mask on y to make y to be 0 where hinge loss is 0
        cast_y = - hinge_mask * y

        # Cast the X with an addtional feature with 1s for b gradients: -y
        cast_X = np.concatenate([X, np.ones((n_samples, 1))], axis=1)

        # Calculate the gradients for w and b in hinge loss term
        grad = self.C * np.dot(cast_y, cast_X)

        # Calculate the gradients for regularization term
        grad_add = np.append(2*w,0)

        # Add the two terms together
        subgrad = grad+grad_add

        return subgrad

    def predict(self, X):
        """
        Predict class labels for samples in X.

        Params
        ------
        X   :    (ndarray, shape = (n_samples, n_features)): test data

        Return
        ------
        y   :   (ndarray, shape = (n_samples,):
                Predictions with values of -1 or 1.
        """
        n_samples, n_features = X.shape

        #initialize y_pred list
        y_pred = np.array([])
        # retrieve the parameters wb
        params_wb = self.unpack_wb(self.wb, n_features)
        w = params_wb[0]
        b = params_wb[1]
        # calculate the predictions
        for i in range(n_samples):
            current_y = np.sign(np.dot(w,X[i]))
            y_pred = np.append(y_pred, current_y)
        # return the predictions
        return y_pred

    def get_params(self):
        """
        Get the model parameters.

        Params
        ------
        None

        Return
        ------
        w       (ndarray, shape = (n_features,)):
                coefficient of the linear model.
        b       (float): bias term.
        """
        return (self.w, self.b)

    def set_params(self, w, b):
        """
        Set the model parameters.
```

```
        Params
        ------
        w       (ndarray, shape = (n_features,)):
                coefficient of the linear model.
        b       (float): bias term.
        """
        self.w = w
        self.b = b



def load_data():
    """
    Helper function for loading in the data

    ------
    # of training samples: 419
    # of testing samples: 150
    ------
    """
    train_X = np.load(f"../../Data/breast_cancer_data/train_x.npy")
    train_y = np.load(f"../../Data/breast_cancer_data/train_y.npy")
    test_X = np.load(f"../../Data/breast_cancer_data/test_x.npy")
    test_y = np.load(f"../../Data/breast_cancer_data/test_y.npy")

    return train_X, train_y, test_X, test_y


def plot_decision_boundary(clf, X, y, title):
    """
    Helper function for plotting the decision boundary

    Params
    ------
    clf     :   The trained SVM classifier
    X       :   (ndarray, shape = (n_samples, n_features)):
                Training input matrix where each row is a feature vector.
    y       :   (ndarray, shape = (n_samples,)):
                Training target. Each entry is either -1 or 1.
    title   :   The title of the plot

    Return
    ------
    """
    x_min, x_max = X[:, 0].min() - 1, X[:, 0].max() + 1
    y_min, y_max = X[:, 1].min() - 1, X[:, 1].max() + 1
    h = (x_max / x_min)/100

    xx, yy = np.meshgrid(np.arange(x_min, x_max, h), np.arange(y_min, y_max, h))
    plt.subplot(1, 1, 1)

    meshed_data = np.c_[xx.ravel(), yy.ravel()]
    Z = clf.predict(meshed_data)
    Z = Z.reshape(xx.shape)

    plt.contourf(xx, yy, Z, cmap=plt.cm.Paired, alpha=0.5)
    plt.scatter(X[:, 0], X[:, 1], c=y, cmap=plt.cm.Paired, linewidth=1, edgecolor='black')
    plt.xlabel('First dimension')
    plt.ylabel('Second dimension')
    plt.xlim(xx.min(), xx.max())
    plt.title(title)
    plt.savefig("../Figures/"+title)
    plt.show()



def main():
    # Set the seed for numpy random number generator
    # so we'll have consistent results at each run
    np.random.seed(0)

    ###QUESTION 1.3###
    #For Question 1.3 and 1.4
    #create a dictionary to store the f1 score, precision, and recall
    metrics_keys = ['Train_F1_Score', 'Train_Precision', 'Train_Recall', 'Test_F1_Score', 'Test_Precision',
'Test_Recall']
    results = dict([(key, []) for key in metrics_keys])

    train_X, train_y, test_X, test_y = load_data()
```

```
    #For Question 1.3
    clf = SVM(C=1)
    objs = clf.fit(train_X, train_y, lr=0.002, iterations=10000)

    train_preds  = clf.predict(train_X)
    test_preds  = clf.predict(test_X)

    #For Question 1.3
    print('Question 1:')
    print('Results for Question 1.3:\n')
    print('1st column is testing set scores & 2nd column is training set scores')
    print(f"F1_score: [{metrics.f1_score(test_y, test_preds)}, {metrics.f1_score(train_y, train_preds)}]")
    print(f"Precision: [{metrics.precision_score(test_y, test_preds)}, {metrics.precision_score(train_y,
train_preds)}]")
    print(f"Recall: [{metrics.recall_score(test_y, test_preds)}, {metrics.recall_score(train_y, train_preds)}]")
    print('')

    ###QUESTION 1.4###
    #Linear Kernel
    #generate SVM class for training data
    clf = svm.SVC(kernel = 'linear')
    #fit with training data
    clf.fit(train_X, train_y)
    #predict with training data
    train_preds = clf.predict(train_X)
    #predict with test data
    test_preds = clf.predict(test_X)
    print('Results for Question 1.4\n')
    print('Linear Kernel')
    print('1st column is testing set scores & 2nd column is training set scores')
    print(f"F1_score: [{metrics.f1_score(test_y, test_preds)}, {metrics.f1_score(train_y, train_preds)}]")
    print(f"Precision: [{metrics.precision_score(test_y, test_preds)}, {metrics.precision_score(train_y,
train_preds)}]")
    print(f"Recall: [{metrics.recall_score(test_y, test_preds)}, {metrics.recall_score(train_y, train_preds)}]")
    print('')

    #3rd Degree Polynomial Kernel
    #generate SVM class for training data
    clf = svm.SVC(kernel = 'poly', degree=3)
    #fit with training data
    clf.fit(train_X, train_y)
    #predict with training data
    train_preds = clf.predict(train_X)
    #predict with test data
    test_preds = clf.predict(test_X)
    print('Polynomial Kernel with Degree=3')
    print('1st column is testing set scores & 2nd column is training set scores')
    print(f"F1_score: [{metrics.f1_score(test_y, test_preds)}, {metrics.f1_score(train_y, train_preds)}]")
    print(f"Precision: [{metrics.precision_score(test_y, test_preds)}, {metrics.precision_score(train_y,
train_preds)}]")
    print(f"Recall: [{metrics.recall_score(test_y, test_preds)}, {metrics.recall_score(train_y, train_preds)}]")
    print('')

    #RBF Kernel
    #generate SVM class for training data
    clf = svm.SVC(kernel = 'rbf')
    #fit with training data
    clf.fit(train_X, train_y)
    #predict with training data
    train_preds = clf.predict(train_X)
    #predict with test data
    test_preds = clf.predict(test_X)
    print('RBF Kernel')
    print('1st column is testing set scores & 2nd column is training set scores')
    print(f"F1_score: [{metrics.f1_score(test_y, test_preds)}, {metrics.f1_score(train_y, train_preds)}]")
    print(f"Precision: [{metrics.precision_score(test_y, test_preds)}, {metrics.precision_score(train_y,
train_preds)}]")
    print(f"Recall: [{metrics.recall_score(test_y, test_preds)}, {metrics.recall_score(train_y, train_preds)}]")
    print('')

    #For Question 1.3 and 1.4
    #calculate metrics and store in dictionary
    results['Train_F1_Score'].append(metrics.f1_score(train_y, train_preds))
    results['Train_Precision'].append(metrics.precision_score(train_y, train_preds))
    results['Train_Recall'].append(metrics.recall_score(train_y, train_preds))
    results['Test_F1_Score'].append(metrics.f1_score(test_y, test_preds))
    results['Test_Precision'].append(metrics.precision_score(test_y, test_preds))
    results['Test_Recall'].append(metrics.recall_score(test_y, test_preds))

    #For Question 1.5
    #For using the first two dimensions of the data - Question 1.5
```

```
    train_X = train_X[:,:2]
    test_X = test_X[:,:2]

    print('Question 1.5:')
    print('For question 1.5, see report or Submissions/Figures for decision boundaries\n')

    #Plotting RBF Kernel Decision Boundary
    clf = svm.SVC(kernel = 'linear')
    clf.fit(train_X, train_y)
    plot_decision_boundary(clf, train_X, train_y, title='SVM Decision Boundary for Linear Kernel')

    #Plotting RBF Kernel Decision Boundary
    clf = svm.SVC(kernel = 'poly', degree=3)
    clf.fit(train_X, train_y)
    plot_decision_boundary(clf, train_X, train_y, title='SVM Decision Boundary for 3rd Degree Polynomial
Kernel')

    #Plotting RBF Kernel Decision Boundary
    clf = svm.SVC(kernel = 'rbf')
    clf.fit(train_X, train_y)
    plot_decision_boundary(clf, train_X, train_y, title='SVM Decision Boundary for RBF Kernel')

    print('Template_SVM script completed')
    print('----------')

    return

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

Part 2: Train a random forest classifier on the provided gene expression data using sklearn.ensemble.RandomForestClassifier. Vary the number of trees from 1 to 150, using parameter n estimators. Provide one combined plot containing three graphs showing test classification error (1accuracy) vs number of trees for three different values of max features [p/3, p, √ (p)], where p is the total number of features in gene expression data.

Python file name on github: ensemble_models.py

```
import numpy as np
from sklearn.ensemble import AdaBoostClassifier
from sklearn.ensemble import RandomForestClassifier
from sklearn.tree import DecisionTreeClassifier
import matplotlib.pyplot as plt

def load_data():
    """
    Helper function for loading in the data

    ------
    # of training samples: 63
    # of testing samples: 20
    ------
    """
    train_X = np.genfromtxt("../../Data/gene_data/gene_train_x.csv", delimiter= ",")
    train_y = np.genfromtxt("../../Data/gene_data/gene_train_y.csv", delimiter= ",")
    test_X = np.genfromtxt("../../Data/gene_data/gene_test_x.csv", delimiter= ",")
    test_y = np.genfromtxt("../../Data/gene_data/gene_test_y.csv", delimiter= ",")

    return train_X, train_y, test_X, test_y



def main():
    np.random.seed(0)
    train_X, train_y, test_X, test_y = load_data()

    #get the number of features and samples in training data
    train_samples, train_features = train_X.shape

    ###Question 2.4###
    #Create empty arrays to store the error
    sqrt_error = np.array([])
    n_features_error = np.array([])
```

```
n_features_d10_error = np.array([])
N = 150 # Each part will be tried with 1 to 150 estimators
#loop over the number of estimators
x_axis = np.array(range(N))
for i in range(N):

    #Train RF with m = sqrt(n_features) recording the errors (errors will be of size 150)
    clf_sqrt = RandomForestClassifier(n_estimators=i+1, max_features='sqrt')
    #Fit the model with training data
    clf_sqrt.fit(train_X, train_y)
    #Use built in method to get the accuracy (or score)
    sqrt_accuracy = clf_sqrt.score(test_X, test_y)
    #append the error vector with the error for the current model
    sqrt_error = np.append(sqrt_error, 1-sqrt_accuracy)

    #Train RF with m = n_features recording the errors (errors will be of size 150)
    clf_n_features = RandomForestClassifier(n_estimators=i+1, max_features=int(train_features))
    #Fit the model with training data
    clf_n_features.fit(train_X, train_y)
    #Use built in method to get the accuracy (or score)
    n_features_accuracy = clf_n_features.score(test_X, test_y)
    #append the error vector with the error for the current model
    n_features_error = np.append(n_features_error, 1-n_features_accuracy)

    #Train RF with m = n_features/10 recording the errors (errors will be of size 150)
    clf_n_features_d10 = RandomForestClassifier(n_estimators=i+1, max_features=int(train_features/10))
    #Fit the model with training data
    clf_n_features_d10.fit(train_X, train_y)
    #Use built in method to get the accuracy (or score)
    n_features_d10_accuracy = clf_n_features_d10.score(test_X, test_y)
    #append the error vector with the error for the current model
    n_features_d10_error = np.append(n_features_d10_error, 1-n_features_d10_accuracy)

#plot the Random Forest results
#plot error for n_estimars = sqrt(n_features)
plt.plot(x_axis, sqrt_error, label = 'Square Root of N_Features')
#plot error for n_estimators = n_features
plt.plot(x_axis, n_features_error, label = 'N_Features')
#plot error for n_estimators = n_features/10
plt.plot(x_axis, n_features_d10_error, label = 'N_Features/10')
plt.title('Random Forest Classifier\nError for Various Numbers of Trees Ranging from 1 to 150')
plt.xlabel('Number of Trees (count)')
plt.ylabel('Error')
plt.legend()
plt.savefig("../Figures/random_forest_error.png")
plt.show()
plt.clf()

###Question 2.6###
#create empty vectors to store error in
clf_1_error = np.array([])
clf_3_error = np.array([])
clf_5_error = np.array([])
N = 150 # Each part will be tried with 1 to 150 estimators
#loop over the number of estimators
x_axis = np.array(range(N))
for i in range(N):

    # Train AdaBoost with max_depth = 1 recording the errors (errors will be of size 150)
    clf_1 = AdaBoostClassifier(DecisionTreeClassifier(max_depth=1), n_estimators=i+1, learning_rate=0.1)
    #Fit the model with training data
    clf_1.fit(train_X, train_y)
    #Use built in method to calculate accuracy (or score)
    clf_1_accuracy = clf_1.score(test_X, test_y)
    #append the error vector with the error for the current model
    clf_1_error = np.append(clf_1_error, 1-clf_1_accuracy)

    # Train AdaBoost with max_depth = 3 recording the errors (errors will be of size 150)
    clf_3 = AdaBoostClassifier(DecisionTreeClassifier(max_depth=3), n_estimators=i+1, learning_rate=0.1)
    #Fit the model with training data
    clf_3.fit(train_X, train_y)
    #Use built in method to calculate accuracy (or score)
    clf_3_accuracy = clf_3.score(test_X, test_y)
    #append the error vector with the error for the current model
    clf_3_error = np.append(clf_3_error, 1-clf_3_accuracy)

    # Train AdaBoost with max_depth = 5 recording the errors (errors will be of size 150)
    clf_5 = AdaBoostClassifier(DecisionTreeClassifier(max_depth=5), n_estimators=i+1, learning_rate=0.1)
    #Fit the model with training data
    clf_5.fit(train_X, train_y)
    #Use built in method to calculate accuracy (or score)
```

```
        clf_5_accuracy = clf_5.score(test_X, test_y)
        #append the error vector with the error for the current model
        clf_5_error = np.append(clf_5_error, 1-clf_5_accuracy)

    # plot the adaboost results
    #plot error for DecisionTreeClassifier max_depth = 1
    plt.plot(x_axis, clf_1_error, label = 'max_depth = 1')
    #plot error for DecisionTreeClassifier max_depth = 3
    plt.plot(x_axis, clf_3_error, label = 'max_depth = 3')
    #plot error for DecisionTreeClassifier max_depth = 5
    plt.plot(x_axis, clf_5_error, label = 'max_depth = 5')
    plt.title('AdaBoost Classifier\nError for Various Numbers of Trees Ranging from 1 to 150')
    plt.xlabel('Number of Trees (count)')
    plt.ylabel('Error')
    plt.legend()
    plt.savefig("../Figures/ada_boost_error.png")
    plt.show()

    print('Template_Ensembles script for Q2 completed.\nSee Report or Submission/Figures for error plots')
    print('----------')
    print('')

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