Reference & Citation:

https://zhuanlan.zhihu.com/p/37357981 (https://zhuanlan.zhihu.com/p/37357981)
https://zhuanlan.zhihu.com/p/38329631 (https://zhuanlan.zhihu.com/p/38329631)
https://github.com/topics/gradient-boosting-machine?l=python (https://github.com/topics/gradient-boosting-machine?l=python) https://scikit-learn.org/stable/modules/generated/sklearn.ensemble.GradientBoostingClassifier.html (https://scikit-learn.org/stable/modules/generated/sklearn.ensemble.GradientBoostingClassifier.html)

Q2(a)

In [2]:

```
import pandas as pd

heart = pd.read_csv("Heart.csv")

heart.shape
```

Out[2]:

```
(303, 15)
```

In [1]:

```
class AdaBoost(object):

    def __init__(self, M, clf, learning_rate=1.0, method="discrete", tol=None, weig
ht_trimming=None):
        self.M = M
        self.clf = clf
        self.learning_rate = learning_rate
        self.method = method
        self.tol = tol
        self.weight_trimming = weight_trimming

    def fit(self, X, y):

        # tol is the threadhold for early_stopping; if we use early_stopping, we ne
ed to split the training set and validation set
        if self.tol is not None:
            X, X_val, y, y_val = train_test_split(X, y, random_state=2)
            former_loss = 1
            count = 0
            tol_init = self.tol

        w = np.array([1 / len(X)] * len(X))   # Initiate the original weight to be
 1/n
        self.clf_total = []
        self.alpha_total = []

        for m in range(self.M):
            classifier = clone(self.clf)
            if self.method == "discrete":
                if m >= 1 and self.weight_trimming is not None:
                    # implementation of weight_trimming: firstly, sort the weight,
 calculate the accumulation, then remove the weights that are too small
                    sort_w = np.sort(w)[::-1]
                    cum_sum = np.cumsum(sort_w)
                    percent_w = sort_w[np.where(cum_sum >= self.weight_trimming)][0
]
                    w_fit, X_fit, y_fit = w[w >= percent_w], X[w >= percent_w], y[w
>= percent_w]
                    y_pred = classifier.fit(X_fit, y_fit, sample_weight=w_fit).pred
ict(X)

                else:
                    y_pred = classifier.fit(X, y, sample_weight=w).predict(X)
                loss = np.zeros(len(X))
                loss[y_pred != y] = 1
                err = np.sum(w * loss)    # calculate the error rate with weight
                alpha = 0.5 * np.log((1 - err) / err) * self.learning_rate  # machi
ne learner's coefficience | alpha
                w = (w * np.exp(-y * alpha * y_pred)) / np.sum(w * np.exp(-y * alph
a * y_pred))  # Update the weight distribution.

                self.alpha_total.append(alpha)
                self.clf_total.append(classifier)

            elif self.method == "real":
```

```
                if m >= 1 and self.weight_trimming is not None:
                    sort_w = np.sort(w)[::-1]
                    cum_sum = np.cumsum(sort_w)
                    percent_w = sort_w[np.where(cum_sum >= self.weight_trimming)][0
]
                    w_fit, X_fit, y_fit = w[w >= percent_w], X[w >= percent_w], y[w
>= percent_w]
                    y_pred = classifier.fit(X_fit, y_fit, sample_weight=w_fit).pred
ict_proba(X)[:, 1]

                else:
                    y_pred = classifier.fit(X, y, sample_weight=w).predict_proba(X)
[:, 1]
                y_pred = np.clip(y_pred, 1e-15, 1 - 1e-15)
                clf = 0.5 * np.log(y_pred / (1 - y_pred)) * self.learning_rate
                w = (w * np.exp(-y * clf)) / np.sum(w * np.exp(-y * clf))

                self.clf_total.append(classifier)

            '''early stopping'''
            if m % 10 == 0 and m > 300 and self.tol is not None:
                if self.method == "discrete":
                    p = np.array([self.alpha_total[m] * self.clf_total[m].predict(X
_val) for m in range(m)])
                elif self.method == "real":
                    p = []
                    for m in range(m):
                        ppp = self.clf_total[m].predict_proba(X_val)[:, 1]
                        ppp = np.clip(ppp, 1e-15, 1 - 1e-15)
                        p.append(self.learning_rate * 0.5 * np.log(ppp / (1 - ppp
)))
                    p = np.array(p)

                stage_pred = np.sign(p.sum(axis=0))
                later_loss = zero_one_loss(stage_pred, y_val)

                if later_loss > (former_loss + self.tol):
                    count += 1
                    self.tol = self.tol / 2
                else:
                    count = 0
                    self.tol = tol_init
                if count == 2:
                    self.M = m - 20
                    print("early stopping in round {}, best round is {}, M = {}".fo
rmat(m, m - 20, self.M))
                    break
                former_loss = later_loss

        return self

    def predict(self, X):
        if self.method == "discrete":
            pred = np.array([self.alpha_total[m] * self.clf_total[m].predict(X) for
m in range(self.M)])

        elif self.method == "real":
```

```
            pred = []
            for m in range(self.M):
                p = self.clf_total[m].predict_proba(X)[:, 1]
                p = np.clip(p, 1e-15, 1 - 1e-15)
                pred.append(0.5 * np.log(p / (1 - p)))

        return np.sign(np.sum(pred, axis=0))


if __name__ == "__main__":
    #Test each model's accuracy and time consumed.
    X, y = datasets.make_hastie_10_2(n_samples=20000, random_state=1)   # data
    X_train, X_test, y_train, y_test = train_test_split(X, y, random_state=1)

    start_time = time.time()
    model_discrete = AdaBoost(M=2000, clf=DecisionTreeClassifier(max_depth=1, rando
m_state=1), learning_rate=1.0,
                              method="discrete", weight_trimming=None)
    model_discrete.fit(X_train, y_train)
    pred_discrete = model_discrete.predict(X_test)
    acc = np.zeros(pred_discrete.shape)
    acc[np.where(pred_discrete == y_test)] = 1
    accuracy = np.sum(acc) / len(pred_discrete)
    print('Discrete Adaboost accuracy: ', accuracy)
    print('Discrete Adaboost time: ', '{:.2f}'.format(time.time() - start_time),'\n
')


    start_time = time.time()
    model_real = AdaBoost(M=2000, clf=DecisionTreeClassifier(max_depth=1, random_st
ate=1), learning_rate=1.0,
                          method="real", weight_trimming=None)
    model_real.fit(X_train, y_train)
    pred_real = model_real.predict(X_test)
    acc = np.zeros(pred_real.shape)
    acc[np.where(pred_real == y_test)] = 1
    accuracy = np.sum(acc) / len(pred_real)
    print('Real Adaboost accuracy: ', accuracy)
    print("Real Adaboost time: ", '{:.2f}'.format(time.time() - start_time),'\n')

    start_time = time.time()
    model_discrete_weight = AdaBoost(M=2000, clf=DecisionTreeClassifier(max_depth=1
, random_state=1), learning_rate=1.0,
                                     method="discrete", weight_trimming=0.995)
    model_discrete_weight.fit(X_train, y_train)
    pred_discrete_weight = model_discrete_weight.predict(X_test)
    acc = np.zeros(pred_discrete_weight.shape)
    acc[np.where(pred_discrete_weight == y_test)] = 1
    accuracy = np.sum(acc) / len(pred_discrete_weight)
    print('Discrete Adaboost(weight_trimming 0.995) accuracy: ', accuracy)
    print('Discrete Adaboost(weight_trimming 0.995) time: ', '{:.2f}'.format(time.t
ime() - start_time),'\n')

    start_time = time.time()
    mdoel_real_weight = AdaBoost(M=2000, clf=DecisionTreeClassifier(max_depth=1, ra
ndom_state=1), learning_rate=1.0,
                                     method="real", weight_trimming=0.999)
```

```
    mdoel_real_weight.fit(X_train, y_train)
    pred_real_weight = mdoel_real_weight.predict(X_test)
    acc = np.zeros(pred_real_weight.shape)
    acc[np.where(pred_real_weight == y_test)] = 1
    accuracy = np.sum(acc) / len(pred_real_weight)
    print('Real Adaboost(weight_trimming 0.999) accuracy: ', accuracy)
    print('Real Adaboost(weight_trimming 0.999) time: ', '{:.2f}'.format(time.time
() - start_time),'\n')

    start_time = time.time()
    model_discrete = AdaBoost(M=2000, clf=DecisionTreeClassifier(max_depth=1, rando
m_state=1), learning_rate=1.0,
                              method="discrete", weight_trimming=None, tol=0.0001)
    model_discrete.fit(X_train, y_train)
    pred_discrete = model_discrete.predict(X_test)
    acc = np.zeros(pred_discrete.shape)
    acc[np.where(pred_discrete == y_test)] = 1
    accuracy = np.sum(acc) / len(pred_discrete)
    print('Discrete Adaboost accuracy (early_stopping): ', accuracy)
    print('Discrete Adaboost time (early_stopping): ', '{:.2f}'.format(time.time()
- start_time),'\n')

    start_time = time.time()
    model_real = AdaBoost(M=2000, clf=DecisionTreeClassifier(max_depth=1, random_st
ate=1), learning_rate=1.0,
                          method="real", weight_trimming=None, tol=0.0001)
    model_real.fit(X_train, y_train)
    pred_real = model_real.predict(X_test)
    acc = np.zeros(pred_real.shape)
    acc[np.where(pred_real == y_test)] = 1
    accuracy = np.sum(acc) / len(pred_real)
    print('Real Adaboost accuracy (early_stopping): ', accuracy)
    print('Discrete Adaboost time (early_stopping): ', '{:.2f}'.format(time.time()
- start_time),'\n')
```

```
---------------------------------------------------------------------------
----
NameError                                 Traceback (most recent call l
ast)
<ipython-input-1-c1d3c27934b7> in <module>
    108 if __name__ == "__main__":
    109     #Test each model's accuracy and time consumed.
--> 110     X, y = datasets.make_hastie_10_2(n_samples=20000, random_st
ate=1)   # data
    111     X_train, X_test, y_train, y_test = train_test_split(X, y, r
andom_state=1)
    112

NameError: name 'datasets' is not defined
```

Gradient Boosting

In [ ]:

```
# Firstly, we have to define the various loss function; logistic loss，modified hub
er loss
def SquaredLoss_NegGradient(y_pred, y):
    return y - y_pred

def Huberloss_NegGradient(y_pred, y, alpha):
    diff = y - y_pred
    delta = stats.scoreatpercentile(np.abs(diff), alpha * 100)
    g = np.where(np.abs(diff) > delta, delta * np.sign(diff), diff)
    return g

def logistic(p):
    return 1 / (1 + np.exp(-2 * p))

def LogisticLoss_NegGradient(y_pred, y):
    g = 2 * y / (1 + np.exp(1 + 2 * y * y_pred))  # logistic_loss = log(1+exp(-2*y*
y_pred))
    return g

def modified_huber(p):
    return (np.clip(p, -1, 1) + 1) / 2

def Modified_Huber_NegGradient(y_pred, y):
    margin = y * y_pred
    g = np.where(margin >= 1, 0, np.where(margin >= -1, y * 2 * (1-margin), 4 * y))
    # modified_huber_loss = np.where(margin >= -1, max(0, (1-margin)^2), -4 * margi
n)
    return g


class GradientBoosting(object):
    def __init__(self, M, base_learner, learning_rate=1.0, method="regression", tol
=None, subsample=None,
                 loss="square", alpha=0.9):
        self.M = M
        self.base_learner = base_learner
        self.learning_rate = learning_rate
        self.method = method
        self.tol = tol
        self.subsample = subsample
        self.loss = loss
        self.alpha = alpha

    def fit(self, X, y):

        if self.tol is not None:
            X, X_val, y, y_val = train_test_split(X, y, random_state=2)
            former_loss = float("inf")
            count = 0
            tol_init = self.tol

        init_learner = self.base_learner
        y_pred = init_learner.fit(X, y).predict(X)   # initial value
        self.base_learner_total = [init_learner]
        for m in range(self.M):
```

```
            if self.subsample is not None:  # subsample
                sample = [np.random.choice(len(X), int(self.subsample * len(X)), re
place=False)]
                X_s, y_s, y_pred_s = X[sample], y[sample], y_pred[sample]
            else:
                X_s, y_s, y_pred_s = X, y, y_pred

            # calculate the negative gradient
            if self.method == "regression":
                if self.loss == "square":
                    response = SquaredLoss_NegGradient(y_pred_s, y_s)
                elif self.loss == "huber":
                    response = Huberloss_NegGradient(y_pred_s, y_s, self.alpha)
            elif self.method == "classification":
                if self.loss == "logistic":
                    response = LogisticLoss_NegGradient(y_pred_s, y_s)
                elif self.loss == "modified_huber":
                    response = Modified_Huber_NegGradient(y_pred_s, y_s)

            base_learner = clone(self.base_learner)
            y_pred += base_learner.fit(X_s, response).predict(X) * self.learning_ra
te
            self.base_learner_total.append(base_learner)

            '''early stopping'''
            if m % 10 == 0 and m > 300 and self.tol is not None:
                p = np.array([self.base_learner_total[m].predict(X_val) for m in ra
nge(1, m+1)])
                p = np.vstack((self.base_learner_total[0].predict(X_val), p))
                stage_pred = np.sum(p, axis=0)
                if self.method == "regression":
                    later_loss = np.sqrt(mean_squared_error(stage_pred, y_val))
                if self.method == "classification":
                    stage_pred = np.where(logistic(stage_pred) >= 0.5, 1, -1)
                    later_loss = zero_one_loss(stage_pred, y_val)

                if later_loss > (former_loss + self.tol):
                    count += 1
                    self.tol = self.tol / 2
                    print(self.tol)
                else:
                    count = 0
                    self.tol = tol_init

                if count == 2:
                    self.M = m - 20
                    print("early stopping in round {}, best round is {}, M = {}".fo
rmat(m, m - 20, self.M))
                    break
                former_loss = later_loss

        return self

    def predict(self, X):
        pred = np.array([self.base_learner_total[m].predict(X) * self.learning_rate
for m in range(1, self.M + 1)])
```

```
        pred = np.vstack((self.base_learner_total[0].predict(X), pred))
        if self.method == "regression":
            pred_final = np.sum(pred, axis=0)
        elif self.method == "classification":
            if self.loss == "modified_huber":
                p = np.sum(pred, axis=0)
                pred_final = np.where(modified_huber(p) >= 0.5, 1, -1)
            elif self.loss == "logistic":
                p = np.sum(pred, axis=0)
                pred_final = np.where(logistic(p) >= 0.5, 1, -1)
        return pred_final


class GBRegression(GradientBoosting):
    def __init__(self, M, base_learner, learning_rate, method="regression", loss="s
quare",tol=None, subsample=None, alpha=0.9):
        super(GBRegression, self).__init__(M=M, base_learner=base_learner, learning
_rate=learning_rate, method=method,
                                           loss=loss, tol=tol, subsample=subsample
, alpha=alpha)

class GBClassification(GradientBoosting):
    def __init__(self, M, base_learner, learning_rate, method="classification", los
s="logistic", tol=None, subsample=None):
        super(GBClassification, self).__init__(M=M, base_learner=base_learner, lear
ning_rate=learning_rate, method=method,
                                               loss=loss, tol=tol, subsample=subsa
mple)

if __name__ == "__main__":
    # creat the dataset and start training
    X, y = datasets.make_regression(n_samples=20000, n_features=10, n_informative=4
, noise=1.1, random_state=1)
    X_train, X_test, y_train, y_test = train_test_split(X, y, random_state=42)
    model = GBRegression(M=1000, base_learner=DecisionTreeRegressor(max_depth=2, ra
ndom_state=1), learning_rate=0.1,
                         loss="huber")
    model.fit(X_train, y_train)
    pred = model.predict(X_test)
    rmse = np.sqrt(mean_squared_error(y_test, pred))
    print('RMSE: ', rmse)

    X, y = datasets.make_classification(n_samples=20000, n_features=10, n_informati
ve=4, flip_y=0.1,
                                        n_clusters_per_class=1, n_classes=2, random_sta
te=1)
    y[y==0] = -1
    X_train, X_test, y_train, y_test = train_test_split(X, y)
    model = GBClassification(M=1000, base_learner=DecisionTreeRegressor(max_depth=1
, random_state=1), learning_rate=1.0,
                             method="classification", loss="logistic")
    model.fit(X_train, y_train)
    pred = model.predict(X_test)
    acc = np.zeros(pred.shape)
    acc[np.where(pred == y_test)] = 1
    accuracy = np.sum(acc) / len(pred)
    print('accuracy logistic score: ', accuracy)
```

```
    model = GBClassification(M=1000, base_learner=DecisionTreeRegressor(max_depth=1
, random_state=1), learning_rate=1.0,
                             method="classification", loss="modified_huber")
    model.fit(X_train, y_train)
    pred = model.predict(X_test)
    acc = np.zeros(pred.shape)
    acc[np.where(pred == y_test)] = 1
    accuracy = np.sum(acc) / len(pred)
    print('accuracy modified_huber score: ', accuracy)
```

Q2(b)

In [30]:

```
from sklearn.model_selection import train_test_split
# read in data
heart = pd.read_csv("Heart.csv")
heart.shape
```

Out[30]:

```
(303, 15)
```

In [44]:

```
X = heart[heart.columns[:-2]]
X
```

Out[44]:

	Unnamed: 0	Age	Sex	ChestPain	RestBP	Chol	Fbs	RestECG	MaxHR	ExAng	Oldpeak
0	1	63	1	typical	145	233	1	2	150	0	2.3
1	2	67	1	asymptomatic	160	286	0	2	108	1	1.5
2	3	67	1	asymptomatic	120	229	0	2	129	1	2.6
3	4	37	1	nonanginal	130	250	0	0	187	0	3.5
4	5	41	0	nontypical	130	204	0	2	172	0	1.4
...	...	...	...	...	...	...	...	...	...	...	..
298	299	45	1	typical	110	264	0	0	132	0	1.2
299	300	68	1	asymptomatic	144	193	1	0	141	0	3.4
300	301	57	1	asymptomatic	130	131	0	0	115	1	1.2
301	302	57	0	nontypical	130	236	0	2	174	0	0.0
302	303	38	1	nonanginal	138	175	0	0	173	0	0.0

303 rows × 13 columns

In [43]:

```
y = heart[heart.columns[-2:]]
y
```

Out[43]:

	Thal	AHD
0	fixed	No
1	normal	Yes
2	reversable	Yes
3	normal	No
4	normal	No
...	...	...
298	reversable	Yes
299	reversable	Yes
300	reversable	Yes
301	normal	Yes
302	normal	No

303 rows × 2 columns

In [45]:

```
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.33, random_st
ate=42)
```

XGBoost:

In [ ]:

```
# specify parameters via map
param = {'max_depth':2, 'eta':1, 'silent':1, 'objective':'binary:logistic' }
num_round = 2

bst = xgb.train(param, X_train, num_round)
# make prediction
preds = bst.predict(X_test)
```

GBM:

In [ ]:

```
import lightgbm as lgb
import pandas as pd
from sklearn.metrics import mean_squared_error
from sklearn.model_selection import GridSearchCV
from sklearn.datasets import load_iris
from sklearn.model_selection import train_test_split
from sklearn.datasets import  make_classification

print('Load data...')

iris = load_iris()
data=iris.data
target = iris.target
X_train,X_test,y_train,y_test =train_test_split(data,target,test_size=0.2)

# df_train = pd.read_csv('../regression/regression.train', header=None, sep='\t')
# df_test = pd.read_csv('../regression/regression.test', header=None, sep='\t')
# y_train = df_train[0].values
# y_test = df_test[0].values
# X_train = df_train.drop(0, axis=1).values
# X_test = df_test.drop(0, axis=1).values

print('Start training...')
# Creat model
gbm = lgb.LGBMRegressor(objective='regression',num_leaves=31,learning_rate=0.05,n_e
stimators=20)
gbm.fit(X_train, y_train,eval_set=[(X_test, y_test)],eval_metric='l1',early_stoppin
g_rounds=5)

print('Start predicting...')
# give prediction
y_pred = gbm.predict(X_test, num_iteration=gbm.best_iteration_)

# test the model
print('The rmse of prediction is:', mean_squared_error(y_test, y_pred) ** 0.5)

# feature importances
print('Feature importances:', list(gbm.feature_importances_))

# opitmization
estimator = lgb.LGBMRegressor(num_leaves=31)

param_grid = {
    'learning_rate': [0.01, 0.1, 1],
    'n_estimators': [20, 40]
}

gbm = GridSearchCV(estimator, param_grid)

gbm.fit(X_train, y_train)

print('Best parameters found by grid search are:', gbm.best_params_)
```