= Udacity P7 =

Random Forest???

Create 35 X and 5 Y

—-

“too good to be true”, and therefore a sign of overfitting, is if the sharpe ratio is greater than 4. Based on this, we recommend using min\_sampes\_leaf of 10 \* 500, or 5,000.

—-

=================================================

### **Train/Valid/Test Splits**

def train\_valid\_test\_split(all\_x, all\_y, train\_size, valid\_size, test\_size):

"""

Generate the train, validation, and test dataset.

Parameters

----------

all\_x : DataFrame

All the input samples

all\_y : Pandas Series

All the target values

train\_size : float

The proportion of the data used for the training dataset

valid\_size : float

The proportion of the data used for the validation dataset

test\_size : float

The proportion of the data used for the test dataset

Returns

-------

x\_train : DataFrame

The train input samples

x\_valid : DataFrame

The validation input samples

x\_test : DataFrame

The test input samples

y\_train : Pandas Series

The train target values

y\_valid : Pandas Series

The validation target values

y\_test : Pandas Series

The test target values

"""

assert train\_size >= 0 and train\_size <= 1.0

assert valid\_size >= 0 and valid\_size <= 1.0

assert test\_size >= 0 and test\_size <= 1.0

assert train\_size + valid\_size + test\_size == 1.0

# TODO: Implement

#(..) ==========================

train\_cutoff = int(train\_size\*len(all\_x))

valid\_cutoff = int((train\_size+valid\_size)\*len(all\_x))

x\_train, x\_valid, x\_test = all\_x[:train\_cutoff], all\_x[train\_cutoff:valid\_cutoff], all\_x[valid\_cutoff:]

y\_train, y\_valid, y\_test = all\_y[:train\_cutoff], all\_y[train\_cutoff:valid\_cutoff], all\_y[valid\_cutoff:]

return x\_train, x\_valid, x\_test, y\_train, y\_valid, y\_test

project\_tests.test\_train\_valid\_test\_split(train\_valid\_test\_split)

## **!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!**

def train\_valid\_test\_split(all\_x, all\_y, train\_size, valid\_size, test\_size):

"""

Generate the train, validation, and test dataset.

Parameters

----------

all\_x : DataFrame

All the input samples

all\_y : Pandas Series

All the target values

train\_size : float

The proportion of the data used for the training dataset

valid\_size : float

The proportion of the data used for the validation dataset

test\_size : float

The proportion of the data used for the test dataset

Returns

-------

x\_train : DataFrame

The train input samples

x\_valid : DataFrame

The validation input samples

x\_test : DataFrame

The test input samples

y\_train : Pandas Series

The train target values

y\_valid : Pandas Series

The validation target values

y\_test : Pandas Series

The test target values

"""

assert train\_size >= 0 and train\_size <= 1.0

assert valid\_size >= 0 and valid\_size <= 1.0

assert test\_size >= 0 and test\_size <= 1.0

assert train\_size + valid\_size + test\_size == 1.0

# TODO: Implement

#(..) ==========================

idx = all\_x.index.levels[0]

train\_dates, valid\_dates, test\_dates = np.split(idx, [int(len(idx) \* train\_size), int(len(idx) \* (train\_size + valid\_size))])

x\_train=all\_x.loc[train\_dates[0]:train\_dates[-1]]

y\_train=all\_y.loc[train\_dates[0]:train\_dates[-1]]

x\_valid=all\_x.loc[valid\_dates[0]:valid\_dates[-1]]

y\_valid=all\_y.loc[valid\_dates[0]:valid\_dates[-1]]

x\_test=all\_x.loc[test\_dates[0]:test\_dates[-1]]

y\_test=all\_y.loc[test\_dates[0]:test\_dates[-1]]

return x\_train, x\_valid, x\_test, y\_train, y\_valid, y\_test

# print(X\_train.iloc[-10:])

# print(X\_valid.iloc[:10])

# print()

# print(all\_x.index.levels[0])

# DatetimeIndex(['2009-07-04', '2009-07-05', '2009-07-06', '2009-07-07',

# '2009-07-08', '2009-07-09', '2009-07-10', '2009-07-11',

# '2009-07-12', '2009-07-13'],

# dtype='datetime64[ns]', freq=None)

# idx = all\_x.index.levels[0]

# print("idx",len(idx)) #10

# print("all\_x",len(all\_x)) #30

# print("all\_y",len(all\_y)) #30

# train\_cutoff = int(train\_size\*len(all\_x))

# valid\_cutoff = int((train\_size+valid\_size)\*len(all\_x))

# print(train\_size) #0.6

# print(valid\_size) #0.2

# print(test\_size) #0.2

# print(train\_cutoff) #18 = 0.6 \* 30

# print(valid\_cutoff) #24 = (0.6+0.2) \* 30

# x\_train, x\_valid, x\_test = all\_x[:train\_cutoff], all\_x[train\_cutoff:valid\_cutoff], all\_x[valid\_cutoff:]

# y\_train, y\_valid, y\_test = all\_y[:train\_cutoff], all\_y[train\_cutoff:valid\_cutoff], all\_y[valid\_cutoff:]

# print(x\_train.iloc[-10:])

# print("----------------")

# print(x\_valid.iloc[:10] )

# test column 1 test column 2 test column 3

# 2015-03-11 Equity(2 [AAP]) 8 38 68

# 2015-03-12 Equity(0 [A]) 9 39 69

# Equity(1 [AAL]) 10 40 70

# Equity(2 [AAP]) 11 41 71

# 2015-03-13 Equity(0 [A]) 12 42 72

# Equity(1 [AAL]) 13 43 73

# Equity(2 [AAP]) 14 44 74

# 2015-03-14 Equity(0 [A]) 15 45 75

# Equity(1 [AAL]) 16 46 76

# Equity(2 [AAP]) 17 47 77

# ------------------

# test column 1 test column 2 test column 3

# 2015-03-15 Equity(0 [A]) 18 48 78

# Equity(1 [AAL]) 19 49 79

# Equity(2 [AAP]) 20 50 80

# 2015-03-16 Equity(0 [A]) 21 51 81

# Equity(1 [AAL]) 22 52 82

# Equity(2 [AAP]) 23 53 83

##############################################################

idx = all\_x.index.levels[0]

train\_cutoff\_date = idx[int(train\_size\*len(idx))]

valid\_cutoff\_date = idx[int((train\_size+valid\_size)\*len(idx))]

print( int(train\_size\*len(idx)) )

print( int((train\_size+valid\_size)\*len(idx)) )

print()

print(train\_cutoff\_date) #2015-06-25 00:00:00

print(valid\_cutoff\_date) #2015-06-27 00:00:00

train\_dates, valid\_dates, test\_dates = np.split(idx, [int(len(idx) \* train\_size), int(len(idx) \* (train\_size + valid\_size))])

print(train\_dates)

print(valid\_dates)

print(test\_dates)

# print(train\_cutoff) #6 = 0.6 \* 30

# print(valid\_cutoff) #8 = (0.6+0.2) \* 30

x\_train, x\_valid, x\_test = all\_x[:train\_cutoff\_date], all\_x[train\_cutoff\_date:valid\_cutoff\_date], all\_x[valid\_cutoff\_date:]

y\_train, y\_valid, y\_test = all\_y[:train\_cutoff\_date], all\_y[train\_cutoff\_date:valid\_cutoff\_date], all\_y[valid\_cutoff\_date:]

print(x\_train.iloc[-10:])

print(x\_valid.iloc[:10] )

# test column 1 test column 2 test column 3

# 2012-07-23 Equity(2 [AAP]) 11 41 71

# 2012-07-24 Equity(0 [A]) 12 42 72

# Equity(1 [AAL]) 13 43 73

# Equity(2 [AAP]) 14 44 74

# 2012-07-25 Equity(0 [A]) 15 45 75

# Equity(1 [AAL]) 16 46 76

# Equity(2 [AAP]) 17 47 77

# 2012-07-26 Equity(0 [A]) 18 48 78

# Equity(1 [AAL]) 19 49 79

# Equity(2 [AAP]) 20 50 80

# test column 1 test column 2 test column 3

# 2012-07-26 Equity(0 [A]) 18 48 78

# Equity(1 [AAL]) 19 49 79

# Equity(2 [AAP]) 20 50 80

# 2012-07-27 Equity(0 [A]) 21 51 81

# Equity(1 [AAL]) 22 52 82

# Equity(2 [AAP]) 23 53 83

# 2012-07-28 Equity(0 [A]) 24 54 84

# Equity(1 [AAL]) 25 55 85

# Equity(2 [AAP]) 26 56 86

#############################################################

# idx = all\_x.index.levels[0]

# print(idx)

# train\_dates, valid\_dates, test\_dates = np.split(idx, [int(len(idx) \* train\_size), int(len(idx) \* (train\_size + valid\_size))])

# print()

# print(train\_dates)

# # DatetimeIndex(['2013-08-12', '2013-08-13', '2013-08-14', '2013-08-15',

# # '2013-08-16', '2013-08-17'],

# # dtype='datetime64[ns]', freq=None)

# print()

# print(valid\_dates)

# # DatetimeIndex(['2013-08-18', '2013-08-19'], dtype='datetime64[ns]', freq=None)

# print()

# print(test\_dates)

# # DatetimeIndex(['2013-08-20', '2013-08-21'], dtype='datetime64[ns]', freq=None)

# x\_train=all\_x.loc[train\_dates[0]:train\_dates[-1]]

# y\_train=all\_y.loc[train\_dates[0]:train\_dates[-1]]

# x\_valid=all\_x.loc[valid\_dates[0]:valid\_dates[-1]]

# y\_valid=all\_y.loc[valid\_dates[0]:valid\_dates[-1]]

# x\_test=all\_x.loc[test\_dates[0]:test\_dates[-1]]

# y\_test=all\_y.loc[test\_dates[0]:test\_dates[-1]]

# print(train\_dates[0] ) #2013-04-01 00:00:00

# print(train\_dates[-1]) #2013-04-06 00:00:00

# print(x\_train.iloc[-10:])

# print(x\_valid.iloc[:10] )

# test column 1 test column 2 test column 3

# 2009-07-15 Equity(2 [AAP]) 8 38 68

# 2009-07-16 Equity(0 [A]) 9 39 69

# Equity(1 [AAL]) 10 40 70

# Equity(2 [AAP]) 11 41 71

# 2009-07-17 Equity(0 [A]) 12 42 72

# Equity(1 [AAL]) 13 43 73

# Equity(2 [AAP]) 14 44 74

# 2009-07-18 Equity(0 [A]) 15 45 75

# Equity(1 [AAL]) 16 46 76

# Equity(2 [AAP]) 17 47 77

# test column 1 test column 2 test column 3

# 2009-07-19 Equity(0 [A]) 18 48 78

# Equity(1 [AAL]) 19 49 79

# Equity(2 [AAP]) 20 50 80

# 2009-07-20 Equity(0 [A]) 21 51 81

# Equity(1 [AAL]) 22 52 82

# Equity(2 [AAP]) 23 53 83

return x\_train, x\_valid, x\_test, y\_train, y\_valid, y\_test

# Here you are splitting the data by row

# but it has to be splitted by day index.

# You can check this by running this

# X\_train.iloc[-10:] and X\_valid.iloc[:10]

# after the cell that's below the function definition.

# Tip: You can use all\_x.index.levels[0] to obtain the day indexes,

# then split those indexes and use all\_x.ix[train\_idx] to obtain the

# training input samples.

# You can do this for valid and testing as well.

project\_tests.test\_train\_valid\_test\_split(train\_valid\_test\_split)

## **Overlapping Samples**

def non\_overlapping\_samples(x, y, n\_skip\_samples, start\_i=0):

"""

Get the non overlapping samples.

Parameters

----------

x : DataFrame

The input samples

y : Pandas Series

The target values

n\_skip\_samples : int

The number of samples to skip

start\_i : int

The starting index to use for the data

Returns

-------

non\_overlapping\_x : 2 dimensional Ndarray

The non overlapping input samples

non\_overlapping\_y : 1 dimensional Ndarray

The non overlapping target values

"""

assert len(x.shape) == 2

assert len(y.shape) == 1

# TODO: Implement

# print("x",x)

# print("y",y)

# print("n\_skip\_samples",n\_skip\_samples) #2

#(..)======================================

id\_x = x.index.levels[0].tolist()[start\_i::n\_skip\_samples+1]

id\_y = y.index.levels[0].tolist()[start\_i::n\_skip\_samples+1]

return x.loc[id\_x,:], y.loc[id\_y,:]

project\_tests.test\_non\_overlapping\_samples(non\_overlapping\_samples)

—————————————————————————————————————————

### **Use BaggingClassifier's max\_samples**

--------------------------------------------------------------------

from sklearn.ensemble import BaggingClassifier

from sklearn.tree import DecisionTreeClassifier

def bagging\_classifier(n\_estimators, max\_samples, max\_features, parameters):

"""

Build the bagging classifier.

Parameters

----------

n\_estimators : int

The number of base estimators in the ensemble

max\_samples : float

The proportion of input samples drawn from when training each base estimator

max\_features : float

The proportion of input sample features drawn from when training each base estimator

parameters : dict

Parameters to use in building the bagging classifier

It should contain the following parameters:

criterion

min\_samples\_leaf

oob\_score

n\_jobs

random\_state

Returns

-------

bagging\_clf : Scikit-Learn BaggingClassifier

The bagging classifier

"""

required\_parameters = {'criterion', 'min\_samples\_leaf', 'oob\_score', 'n\_jobs', 'random\_state'}

assert not required\_parameters - set(parameters.keys())

# TODO: Implement

clf = BaggingClassifier(base\_estimator = DecisionTreeClassifier(criterion = parameters['criterion']

, min\_samples\_leaf = parameters['min\_samples\_leaf'])

, max\_samples = max\_samples

, n\_estimators = n\_estimators

, max\_features = max\_features

, oob\_score = parameters['oob\_score']

, n\_jobs = parameters['n\_jobs']

, random\_state = parameters['random\_state'])

return clf

project\_tests.test\_bagging\_classifier(bagging\_classifier)

————————————————————————————————--———————

OOB Score

——————————----------------------------------------------------------———————

def calculate\_oob\_score(classifiers):

"""

Calculate the mean out-of-bag score from the classifiers.

Parameters

----------

classifiers : list of Scikit-Learn Classifiers

The classifiers used to calculate the mean out-of-bag score

Returns

-------

oob\_score : float

The mean out-of-bag score

"""

# TODO: Implement

oob\_score = []

for clf in classifiers:

oob\_score.append(clf.oob\_score\_)

return np.mean(oob\_score)

project\_tests.test\_calculate\_oob\_score(calculate\_oob\_score)

————————————————————————————————————————

Non Overlapping Estimators

————————————————————————————————————————

--