Machine Learning for Option Trading

Mengran Wang

August 2017

1 Introduction

The goal of the project is to predict price of the option on the settlement date so we can buy or sell options based on the result. Options are a type of derivative securities which means its value depends on some underlying asset. In our case, we choose SP500 index options- SPX. This is an index option and its underlying asset is SP500 index. Therefore, at the settlement date, the options will be cash-settled based on the SP500 index price at that time. The options can either be call options or put options. There is a strike price of each option and it is the price that the underlying asset will be traded at the settlement date. If it is a call option, when the price of the underlying asset goes above the strike price, the buyer of this call option can make money because the underlying asset is worth more than the strike price, and vice versa for a put option. So in our case, if we buy a call option of SPX and the strike price is 1600, the price of SP500 is 1650 at the settlement date, we can make 50 and the money we make will only be cash settled. Moreover, the option buyer has the right to choose whether to execute the option or not. If the market is against the buyer, he will choose not to execute the option and the option will expire worthless. On the other hand, the seller of the option has only the obligation to execute the option.

2 Methodology

2.0 Goal
What we want is to predict SP500 price at the settlement date based on the data we have. And then we are going to make different strategy about option trading according to our prediction. Finally, we will backtest each strategy and find the most promising one.

2.1 Data
The data we use is historical SPX data from 2013.07.09 to 2017.03.01. Each row corresponds to an option at a specific day. Each observation has 12 columns: today's date, settlement date, remaining days to settle, today's SP500 price, settlement date's SP500 price, option type, option strike price, option closing

price, option high price, option low price, option volume and option open interest. We can use option type, option strike price and settlement date to identify different options.

2.2 Data Manipulation
First, we want to remove all duplicate oberservations.
Second, we add yesterday's volume, open interest and closing price for each option
Third, we add last week's volume, open interest and closing price for each option; if the options does not exist one week before, we will use its yesterday's data instead.
Forth, add yesterday's SP500 price and last weeks' price.
Fifth, add corresponding ratios of today's price over yesterday's price and today's price over last week's price
Sixth, we filter all the options and only use those whose today's SP500 price is within 30 of settlement day's SP500 price.
Seventh, group all the options based on their today's date and settlement date.

Code:

Listing 1: Data Manipulation

```
## Data Manipulation

# Data manipulation:
#     1. remove duplicate rows
#     2. add last day's volumn, open interest rate,
    closing price for each option
#     3. add last week's volumn, open interest rate,
    closing price for each option
#     4. add last day's sp500 price and last week's price
#     5. select options whose today's sp500 price is
    within 30 of strike price
#     5. remove rows with null values
#     6. group all the options corresponding to their (
    settlement date, today's date) and pick all the
    interested features

# In[1]:

get_ipython().magic(u'matplotlib inline')
import pandas as pd
import numpy as np
import matplotlib as mpl
import matplotlib.pyplot as plt
import datetime
```

```
# In[4]:

def readToDataFrame(input):
    """ read data from input path
        drop duplicates
        and return dataframe where settle date and
            todaydate is transfferd to datetime format
    """
    optiondata = pd.read_table( input, delimiter = ",")
    todaydate_tmp = [pd.datetime.strptime(str(x), '%Y%m%d
        ') for x in optiondata.todaydate]
    optiondata.todaydate = todaydate_tmp
    settledate_tmp = [pd.datetime.strptime(str(x), '%Y%m%
       d') for x in optiondata.settledate]
    optiondata.settledate = settledate_tmp
    return optiondata.drop_duplicates()
optiondata = readToDataFrame("relevantoptions2")


# In[5]:

# create dictionary to store everyday's sp500 price

def createPriceDict(row, pricedict):
    if not pricedict.has_key(row['todaydate'].strftime("%
       B␣%d,␣%Y")):
        pricedict[row['todaydate'].strftime("%B␣%d,␣%Y")]
            = row['today_sp_price']

def getPrice(df):
    """get vol and sp500 today price for each day
    """
    price = {}
    df.apply(lambda row: createPriceDict(row, price),
        axis=1)
    return price

priceDict = getPrice(optiondata)


# In[6]:

# create dictionary to store every option's volumn, open
    interest rate, closing price for each day
# option can be identified by settledate, optiontype,
```

```
    option strike
def createOptionDict(row, optdict):
    if not optdict.has_key((row['todaydate'],row['
        settledate'],row['optiontype'],row['optionstrike'
        ])):
        optdict[(row['todaydate'],row['settledate'],row['
            optiontype'],row['optionstrike'])] = (row['
            optionvol'],
                    row['optionopenint'], row['
                        optionclosingprice'])

def getOption(df):
    """get vol and sp500 today price for each day
    """
    optionDic = {}
    df.apply(lambda row: createOptionDict(row, optionDic)
        , axis=1)

    return optionDic

optionDic = getOption(optiondata)


# In[7]:

#add sp500 yesterday's and last week's price
def prevPrice1d(row, priceDic):
    i = 1;
    while (i < 5):
        if priceDic.has_key((row['todaydate']-datetime.
            timedelta(i)).strftime("%B %d, %Y")):
            return priceDic[(row['todaydate']-datetime.
                timedelta(i)).strftime("%B %d, %Y")]
        else:
            i+=1
    return 0

def prevPrice1week(row, priceDic):
    i = 7;
    if priceDic.has_key((row['todaydate']-datetime.
        timedelta(i)).strftime("%B %d, %Y")):
        return priceDic[(row['todaydate']-datetime.
            timedelta(i)).strftime("%B %d, %Y")]
    return 0
```

```
def addPrevPrice(df, priceDic):
    """ add s&p price of previous n days to dataframe
    """
    prvPrice =  df.apply(lambda row: prevPrice1d(row,
        priceDic), axis=1)
    df['sp1d'] = prvPrice
    prvweekPrice =  df.apply(lambda row: prevPrice1week(
        row, priceDic), axis=1)
    df['sp1w'] = prvweekPrice

addPrevPrice(optiondata,priceDict)


# In[8]:

# add option's yesterday and last week's vol, openint and
    closing price

def prevOption1d(row, optionDic):
    i = 1;
    while (i < 5):
        if optionDic.has_key((row['todaydate'] - datetime
            .timedelta(i),row['settledate'],row['
            optiontype'],row['optionstrike'])):
            return optionDic[((row['todaydate'] -
                datetime.timedelta(i)),row['settledate'],
                row['optiontype'],row['optionstrike'])]
        else:
            i += 1
    return -1

def prevOption1w(row, optionDic):
    i = 7;
    if optionDic.has_key((row['todaydate'] - datetime.
        timedelta(i),row['settledate'],row['optiontype'],
        row['optionstrike'])):
        return optionDic[((row['todaydate'] - datetime.
            timedelta(i)),row['settledate'],row['
            optiontype'],row['optionstrike'])]
    return -1


def addOptionDay(df, optionDic):
    # add option volume of previous n days to dataframe
    prevoptionday = df.apply(lambda row: prevOption1d(row
        , optionDic), axis=1)
```

```
    #print prevoptionday.head(20)
    df['volopenint%closing1d'] = prevoptionday
    prevoptionweek = df.apply(lambda row: prevOption1w(
        row, optionDic), axis=1)
    df['volopenint%closing1w'] = prevoptionweek


addOptionDay(optiondata, optionDic)
optiondata.to_csv("optiondata1", index=False)


### some analysis

# In[9]:

# select options that whose today's 500 price is within
    30 of strike price
within30data = optiondata[abs(optiondata.today_sp_price -
    optiondata.optionstrike)<=30]
within100data = optiondata[abs(optiondata.today_sp_price
   - optiondata.optionstrike)<=100]
print within100data.describe()
print within30data.describe()
print optiondata.describe()
haslastweek = within30data[within30data['volopenint%
    closing1w'] == -1]
haslastweek.describe()


# Summary
#
# At first, we have 55139 rows;
#
# There are 49326 options whose price is within 100 of
    strikeprice;
#
# 18479 options whose price is within 30 of strikeprice;
#
# Out of 18479 options, 10367 of them don't exist exactly
    one week before

# In[10]:

# add closing price, vol and open interest of  yesterday
def closing1day(row):
    if row['volopenint%closing1d'] == -1:
```

```
        return row['optionclosingprice']
    else:
        return row['volopenint%closing1d'][2]

def addClosing1d(df):
    """ yesterday's closing price
    """

    prvPrice =  df.apply(lambda row: closing1day(row),
        axis=1)
    df['closing1day'] = prvPrice

def vol1day(row):
    if row['volopenint%closing1d'] == -1:
        return row['optionvol']
    else:
        return row['volopenint%closing1d'][0]

def addvol1d(df):
    """ yesterday's volume
    """
    prvVol =  df.apply(lambda row: vol1day(row), axis=1)
    df['vol1day'] = prvVol
    #df['prevPrice1d'].hist()

def opint1day(row):
    if row['volopenint%closing1d'] == -1:
        return row['optionopenint']
    else:
        return row['volopenint%closing1d'][1]

def addopint1d(df):
    """ yesterday's volume
    """

    prvopen =  df.apply(lambda row: opint1day(row), axis
        =1)
    df['opint1day'] = prvopen
    #df['prevPrice1d'].hist()

def addYesterdayData(df):
    addClosing1d(df)
    addvol1d(df)
    addopint1d(df)

addYesterdayData(within30data)
```

```
# In[11]:

# add closing price, vol and open interest of  lastweek
def closing1w(row):
    if row['volopenint%closing1w'] == -1:
        return row['closing1day']
    else:
        return row['volopenint%closing1w'][2]

def addClosing1w(df):
    """ last week's closing price
    """
    prvPrice =  df.apply(lambda row: closing1w(row), axis
        =1)
    df['closing1w'] = prvPrice

def vol1w(row):
    if row['volopenint%closing1w'] == -1:
        return row['vol1day']
    else:
        return row['volopenint%closing1w'][0]

def addvol1w(df):
    """ last week's volume
    """
    prvVol =  df.apply(lambda row: vol1w(row), axis=1)
    df['vol1w'] = prvVol

def opint1w(row):
    if row['volopenint%closing1w'] == -1:
        return row['opint1day']
    else:
        return row['volopenint%closing1w'][1]

def addopint1w(df):
    """ yesterday's open interest
    """
    prvopen =  df.apply(lambda row: opint1w(row), axis=1)
    df['opint1w'] = prvopen

def addLastWeekData(df):
    addClosing1w(df)
    addvol1w(df)
    addopint1w(df)
```

```
addLastWeekData(within30data)


# In[12]:

print within30data.describe()


# we found that distribution of top 25% vol are 0 which
    means they may not be stable. So we ignore features
    related to vol

# In[13]:

#add ratios for closing price and openinterest: today
    price/ yesterday price and today price/ last week's
    price
# if the option doesn't exist one week before, the ratio
    will be the same as today price/ yesterday's price

def closing1dayratio(row):
    return row['optionclosingprice']/row['closing1day']

def addClosing1dratio(df):
    prvPrice =  df.apply(lambda row: closing1dayratio(row
        ), axis=1)
    df['closing1dayratio'] = prvPrice


def opint1dayratio(row):
    return float(row['optionopenint'])/row['opint1day']

def addopint1dratio(df):
    prvopen =  df.apply(lambda row: opint1dayratio(row),
        axis=1)
    df['opint1dayratio'] = prvopen
    #df['prevPrice1d'].hist()

def closing1wratio(row):
    return row['optionclosingprice']/row['closing1w']

def addClosing1wratio(df):
    """ last week's closing price
    """
```

```
    prvPrice =  df.apply(lambda row: closing1wratio(row),
        axis=1)
    df['closing1wratio'] = prvPrice

def opint1wratio(row):
     return float(row['optionopenint'])/row['opint1w']

def addopint1w(df):
    prvopen =  df.apply(lambda row: opint1wratio(row),
       axis=1)
    df['opint1wratio'] = prvopen


def addRatio(df):
    addopint1w(df)
    addClosing1wratio(df)
    addopint1dratio(df)
    addClosing1dratio(df)

addRatio(within30data)


# In[14]:

##check whether result is right
print within30data.count()
print within30data[within30data['closing1wratio']==
    within30data['closing1dayratio']].count()
# this is consistent with  " And there are 10367 of them
    that doesn't exist exactly one week before"


# In[15]:

#remove unrelated columns
cleandata = within30data.drop(['optionhighprice','
    optionlowprice','optionvol','volopenint%closing1d',
                                'volopenint%closing1w','
                                    closing1day','vol1day',
                                    'opint1day','closing1w'
                                    ,
                                'vol1w','opint1w'], axis
                                    =1)
cleandata = cleandata[cleandata['sp1d']!=0]
cleandata = cleandata[cleandata['sp1w']!=0]
cleandata.describe()
```

```
# ### group options belong to same (today's date and
    settlement date)
# If we use 20 options in a row, there will be only 109
    observations and 203 features, so I  combine top 15
    options for each pair
# 1. Next, we are going to put the first 15 options that
    has strike price closest to today's sp500
    corresponding to the same (today date, settlement date
    ) in one row according to following steps:
# 2. sort data according to their absolute value of (
    strike price - today's sp500 price)
# 3. create a dictionary, whose key is (today date,
    settlement date) pair and value is a list of our
    interested feature of each option.(suppose m vecor for
     each element)
# 4. add each option in the dictionary.
# 5. adjust each dict value to include 15 options
# 6. sort data accoridng to settledate and if settledate
    are the same, sort the data based on today's date.
#
#

# In[16]:

def getOptionType(row):
    if row['optiontype']=='put':
        return 0
    else:
        return 1
def changeType(df):
    df['type'] = df.apply(lambda row: getOptionType(row),
        axis = 1 )
    df['optiontype']=df['type']



def diffStrikeSp500(row):
    return abs(row['optionstrike']-row['today_sp_price'])
def sortAbs(df):
    df['abs'] = df.apply(lambda row: diffStrikeSp500(row)
        , axis = 1 )
    return df.sort_values(by = 'abs')
```

```
changeType(cleandata)
cleandata = sortAbs(cleandata)


# In[18]:

def addToDict(row, optiondict):
    if optiondict.has_key((row['todaydate'],row['
        settledate'])):
        if len(optiondict[(row['todaydate'],row['
            settledate'])]) != 15:
            optiondict[(row['todaydate'],row['settledate'
                ])].extend(
            [row['optiontype'],row['optionstrike'],
            row['optionclosingprice'], row['optionopenint
                '],row['sp1d'],row['sp1w'],row['
                opint1wratio'],
            row['closing1wratio'],row['opint1dayratio'],
                row['closing1dayratio']])
    else:
        optiondict[(row['todaydate'],row['settledate'])]
            = [row['daystosettle'],row['settle_sp_price'],
            row['today_sp_price'],row['optiontype'],row['
            optionstrike'],
        row['optionclosingprice'], row['optionopenint'],
            row['sp1d'] ,row['sp1w'],row['opint1wratio'],
        row['closing1wratio'],row['opint1dayratio'],row['
            closing1dayratio']]

def createDic(df):
    myDic ={}
    df.apply(lambda row: addToDict(row, myDic), axis=1)
    return myDic


myDic = createDic(cleandata)


# In[19]:

## adjust dictionary to include only 15 options
def adjustDic(myDict):
    for k, v in myDict.items():
        if len(v) < 153:
            #print len(myDict[k])
            del myDict[k]
```

```
        else:
            myDict[k] = myDict[k][:153]

adjustDic(myDic)
df3=pd.DataFrame.from_dict(myDic)
df4 = df3.T
df4.reset_index(inplace=True)
df5 = df4.rename(columns={'level_1': 'settledate', '
    level_0': 'todaydate'})
df5.to_csv("combinedoption2", index=False)


# In[ ]:

def adjustDic(myDict):
    for k, v in myDict.items():
        if len(v) < 153:
            #print len(myDict[k])
            del myDict[k]
        else:
            myDict[k] = myDict[k][:153]

adjustDic(myDic)
df3=pd.DataFrame.from_dict(myDic)
df4 = df3.T
df4.reset_index(inplace=True)
df5 = df4.rename(columns={'level_1': 'settledate', '
    level_0': 'todaydate'})
df5.to_csv("combinedoption2", index=False)


# Summary:
# 1. df5 is after we combine options belong to same (
    today's date, settle date)
# 2. clean data is optiondata before combining
# 3. After combining options belong to the same pair, we
    got 276 rows and 153 features for each pair(column)
#
```

2.3 Training Phase

We train a random forest regressor on the clean data.
First, divide training set and test set based on settlement date. If a option's settlement date is before June 30, 2015 , we use it as training data; the rest will be used as test data.
Second, our target will be settlement day's SP500 price - today's SP500 price
Third, features will be option type, option strike, option closing price, option

high price, option low price, option open interest, SP500 price of yesterday , SP500 price of last week, open interest one week ratio, closing price one week ratio, open interest 1day ratio , closing price 1day ratio for each option and there are 15 options for each observation. Finally, we add days to settle and today's SP500 price to each observation.
Forth, training a random forest regressor against the data.
The performance of the training is: mean squared error for training set is 57.19 and the mean squared error for test set is 1742.
Code:

Listing 2: Training

```
## Training

# Random Forest Regressor
#     1. we divide trainig set and test set based on
    settlement date. If a option's settledate is after the
     half of           2015, we use it as test data;
    otherwise, training data.
#     2. Target: settlement day's sp500 price - today's
    sp500 price
#           X: ['optiontype','optionstrike',
#           'optionclosingprice','optionhighprice', '
    optionlowprice', 'optionopenint',
#           'sp1d','sp1w','opint1wratio',
#         'closing1wratio','opint1dayratio','
    closing1dayratio'] for each option and ['daystosettle
    ', 'today_sp_price']                  for all options
    belong to the same (settledate, toadaydate) pair.
#     3. Train a random forest regressor

# In[22]:

from sklearn.ensemble import RandomForestClassifier
from sklearn import metrics
from sklearn.metrics import confusion_matrix
from sklearn.ensemble import RandomForestRegressor
from datetime import datetime


# In[23]:

def target(row):
    return row[1] - row[2]

def addTarget(df):
    """ target = settle price - today's price
```

```
    """
    t = df.apply(lambda row: target(row), axis=1)
    df['target'] = t

addTarget(df5)


# In[24]:

mask1 = (df5['settledate'] < datetime.strptime('Jun 30
    2015  1:33PM', '%b %d %Y %I:%M%p'))
training= df5.loc[mask1]
mask2 = (df5['settledate'] > datetime.strptime('Jun 30
    2015  1:33PM', '%b %d %Y %I:%M%p'))
test= df5.loc[mask2]

col_list = list(df5)
col_list[4:155]
x_list=[0]
x_list.extend(col_list[4:155])

X_test = test.as_matrix(x_list)
y_test = test.as_matrix(['target'])[:,0]

X_train = training.as_matrix(x_list)
y_train = training.as_matrix(['target'])[:,0]



regres = RandomForestRegressor(n_estimators=1000,
    max_features="sqrt", n_jobs=-1, oob_score = True)
regres.fit(X_train, y_train)
pred = regres.predict(X_test)
print("Mean squared error - Test Set: %.6f"
  % np.mean((pred - y_test) ** 2))
# Explained variance score: 1 is perfect prediction
print('Variance score: %.2f' % regres.score(X_test,
    y_test))

pred_train = regres.predict(X_train)
print("Mean squared error - Training Set: %.6f"
  % np.mean((pred_train - y_train) ** 2))
# Explained variance score: 1 is perfect prediction
print('Variance score: %.2f' % regres.score(X_train,
    y_train))
```

```
### performance analysis

In[26]:

# scatter plot of real difference over prediction on test
    set
import numpy
plt.plot(pred, y_test, ".")
plt.ylabel('real diff')
plt.xlabel('test prediction')
print numpy.corrcoef(pred,y_test)


In[27]:

# scatter plot of real difference over prediction on
    training set
plt.plot(pred_train, y_train, ".")
plt.ylabel('real diff')
plt.xlabel('training prediction')
print numpy.corrcoef(pred_train,y_train)


In[28]:

regres.oob_prediction_


In[29]:

pred_train
plt.plot(pred_train, regres.oob_prediction_, ".")
print numpy.corrcoef(pred_train,regres.oob_prediction_)
```

2.4 Strategy

There are three strategies that we implemented in the project.
First, normal strategy.

Suppose we predict the difference between settlement date's price and today's SP500 price is larger than 0, we will sell all the put options whose strike price is lower or equal to today's SP500 price. If our prediction turns out to be right, we will make money by selling this option because this option will expire worthless at the end. On the other hand, if we predict the difference will be negative, we will sell call options whose strike price is equal or larger than

today's SP500 price.
Second, a more restrictive strategy.

Suppose we predict the difference between settlement date's price and today's SP500 price is larger than 0, and we name it d. We will sell all the options whose strike price is lower or equal to $(today'sSP500price - d)$. And vice versa if the prediction is negative. In this way, we are more guaranteed that the profit of trading options should be positive because it gives us some error space for the prediction.
Third, trade as much as possible strategy

In this strategy, we will also consider the price of an option. Therefore, if we are selling an option, as long as the price of the option can cover the trading loss, we will sell the option. And if we are buying an option, we need to guarantee that the profit we make at the settlement date must be more that the option price. So suppose we predict the difference between settlement date's price and today's SP500 price is larger than 0, we will sell a put option if its strike price is lower of equal to $(todaySp500price + optionclosingprice)$. And if we predict the difference will be negative, we will sell a call option if its strike price is above $(todaySp500price - optionclosingprice)$.

Code:

Listing 3: Strategies

```
## Strategies

### Add prediction to test set

# In[31]:

# get dictionary for ((today's date, settlement's date):
    prediction)
def createPredDict(row, pred_dict):
    if not pred_dict.has_key((row['todaydate'],row['
        settledate'])):
         pred_dict[(row['todaydate'],row['settledate'])] =
             row['pred']

def getPredict(df):
    """get predition for each today's date and settledate
    """
    prediction = {}
    df.apply(lambda row: createPredDict(row, prediction),
         axis=1)
    return prediction

predAll = numpy.append(pred_train,pred,axis=0)
df5['pred'] = predAll.tolist()
```

```
predDict = getPredict(df5)


# In[32]:

## get dataframe for testing
mask3 = (cleandata['settledate'] > datetime.strptime('Jun
    30 2015  1:33PM', '%b %d %Y %I:%M%p'))
testdf= cleandata.loc[mask3]


# In[33]:

## add sp500 difference predction to each option

def getPred(row,pred_dict):
    if pred_dict.has_key((row['todaydate'],row['
        settledate'])):
        return pred_dict[(row['todaydate'],row['
            settledate'])]
    else:
        return 0

def changePred(df, pred_dict):
    pred0 = df.apply(lambda row: getPred(row, pred_dict),
        axis=1)
    df['pred'] = pred0

changePred(testdf, predDict)


### First Strategy

# In[34]:

testdf['pl']= 0

def compPL(row, pred_dict, multiplier):
    if row.pred == 0:
        return 0
    else:
        if row.pred > 0:
            if row.optionstrike <= row.today_sp_price and
                row.optiontype == 0: #put
                if row.settle_sp_price >= row.
                    optionstrike:
```

```
                    #print row.optionclosingprice
                    return row.optionclosingprice*
                        multiplier
                else:
                    #print row.optionclosingprice +
                        multiplier * (row.settle_sp_price
                        - row.optionstrike)
                    return row.optionclosingprice*
                        multiplier + multiplier * (row.
                        settle_sp_price - row.optionstrike
                        )
        elif row.pred < 0:
            if row.optionstrike >= row.today_sp_price and
                row.optiontype == 1: #call
                if row.settle_sp_price <= row.
                    optionstrike:
                    #print row.optionclosingprice
                    return row.optionclosingprice*
                        multiplier
                else:
                    #print row.optionclosingprice -
                        multiplier * (row.settle_sp_price
                        - row.optionstrike)
                    return row.optionclosingprice*
                        multiplier - multiplier * (row.
                        settle_sp_price - row.optionstrike
                        )
        else:
            return 0
def getPl(df, pred_dict):
    pl0 = df.apply(lambda row: compPL(row, pred_dict, 1),
        axis=1)
    df['pl'] = pl0


getPl(testdf, predDict)
testdf['pl'].fillna(0, inplace=True)


# In[35]:

## result of first strategy
positivedf = testdf
neagtivedf = testdf
positivedf = testdf[testdf.pl > 0]
negativedf = testdf[testdf.pl < 0]
```

```
# In[37]:

print negativedf.describe()
print positivedf.describe()
print "profit is "+ str(positivedf['pl'].sum())
print "loss is " + str(negativedf['pl'].sum())


# testdf is all the options that we are going to trade.
#
# putdf are put options and calldf are call options
#
# generate scatter points for put and call option where x
    axis is prediction and y is profit

# In[39]:

interestedDf = testdf
interestedDf = testdf[testdf.pl!=0]
putDf = interestedDf[interestedDf.optiontype == 0]
callDf = interestedDf[interestedDf.optiontype == 1]


# In[40]:

print putDf.describe()
print callDf.describe()


# In[41]:

plt.scatter(putDf['pred'], putDf['pl'])


# In[42]:

plt.scatter(callDf['pred'], callDf['pl'])


## Second Strategy

# In[43]:

testdf2 = testdf
```

```
testdf2['pl']= 0

def compPL(row, pred_dict, multiplier):
    if row.pred == 0:
        return 0
    else:
        if row.pred > 0:
            if row.optionstrike <= row.today_sp_price -
               row.pred and row.optiontype == 0:
                if row.settle_sp_price >= row.
                   optionstrike:
                    #print row.optionclosingprice
                    return row.optionclosingprice*
                       multiplier
                else:
                    #print row.optionclosingprice +
                       multiplier * (row.settle_sp_price
                       - row.optionstrike)
                    return row.optionclosingprice*
                       multiplier + multiplier * (row.
                       settle_sp_price - row.optionstrike
                       )
        elif row.pred < 0:
            if row.optionstrike >= row.today_sp_price  -
               row.pred and row.optiontype == 1:
                if row.settle_sp_price <= row.
                   optionstrike:
                    #print row.optionclosingprice
                    return row.optionclosingprice*
                       multiplier
                else:
                    #print row.optionclosingprice -
                       multiplier * (row.settle_sp_price
                       - row.optionstrike)
                    return row.optionclosingprice*
                       multiplier - multiplier * (row.
                       settle_sp_price - row.optionstrike
                       )
        else:
            return 0
def getPl(df, pred_dict):
    pl0 = df.apply(lambda row: compPL(row, pred_dict, 1),
        axis=1)
    df['pl'] = pl0
```

```
getPl(testdf2, predDict)
testdf2['pl'].fillna(0, inplace=True)



### result of second strategy

# In[44]:

positivedf2 = testdf2
neagtivedf2 = testdf2
positivedf2 = testdf2[testdf2.pl > 0]
negativedf2 = testdf2[testdf2.pl < 0]
print negativedf2.describe()
print positivedf2.describe()
print "profit is "+ str(positivedf2['pl'].sum())
print "loss is " + str(negativedf2['pl'].sum())


### seperate analysis of call and put option

# In[46]:

interestedDf2 = testdf2
interestedDf2 = testdf2[testdf2.pl!=0]
putDf2 = interestedDf2[interestedDf2.optiontype == 0]
callDf2 = interestedDf2[interestedDf2.optiontype == 1]


# In[47]:

## scatter plot of put options' prediction over its P&l
plt.scatter(putDf2['pred'], putDf2['pl'])


# In[48]:

## scatter plot of call options' prediction over its P&l
plt.scatter(callDf['pred'], callDf['pl'])


### analysis based on the days to settle date

# In[50]:

X_test = test.as_matrix(x_list)
```

```
y_test = test.as_matrix(['target'])[:,0]

X_train = training.as_matrix(x_list)
y_train = training.as_matrix(['target'])[:,0]
pred = regres.predict(X_test)
pred_train = regres.predict(X_train)



# In[51]:

# convert X_test, y_test, pred to df
df6 = pd.DataFrame(X_test.tolist())
df6['real'] = y_test.tolist()
df6['pred'] = pred.tolist()


# In[52]:

#get day_difference list
def createDayDiffDict(row, dayList):
    if row[0] not in dayList:
        dayList.append(row[0])

def getDayDiff(df):
    """get vol and sp500 today price for each day
    """
    dayDiff = []
    df.apply(lambda row: createDayDiffDict(row, dayDiff),
        axis=1)
    dayDiff.sort()
    return dayDiff

dayDiffList = getDayDiff(df6)


# In[53]:

groups = df6.groupby(0)
fig, ax = plt.subplots()
fig.suptitle('Scatter plot group by day of difference',
    fontsize=15)
ax.margins(0.05)
for name, group in groups:
    ax.plot(group.real, group.pred, marker='.', linestyle
        ='', ms=10, label=name)
```

```
ax.legend()


# In[54]:

## generate correlation group by day of difference one by
    one, calculate correlation coefficient
corrList = []
def oneDayDiff(key, df):
    df0 = df
    df0 = df[df[0]==key]
    fig = plt.figure()
    plt.xlabel('real value')
    plt.ylabel('prediction')
    plt.title('plot for day of difference: ' + str(key),
        fontsize=10)
    plt.suptitle('correlation coefficient is '+ str(numpy
        .corrcoef(df0['real'],df0['pred'])[0][1]),
        fontsize=8)
    plt.plot(df0['real'], df0['pred'], ".")
    return numpy.corrcoef(df0['real'],df0['pred'])[0][1]


def allDayDiff(df, dayList, corr):
    for d in dayList:
        res = oneDayDiff(d, df)
        corr.append(res)
    return corr

corrList = allDayDiff(df6, dayDiffList, corrList)


### Third Startegy

# In[56]:

testdf['pl']= 0

def compPL(row, pred_dict, multiplier):
    if row.pred == 0:
        return 0
    else:
        if row.pred > 0:
            if row.optionstrike < row.today_sp_price +
                row.optionclosingprice and row.optiontype
                == 0:
```

```
                if row.settle_sp_price >= row.
                    optionstrike:
                    #print row.optionclosingprice
                    return row.optionclosingprice*
                        multiplier
                else:
                    #print row.optionclosingprice +
                        multiplier * (row.settle_sp_price
                        - row.optionstrike)
                    return row.optionclosingprice*
                        multiplier + multiplier * (row.
                        settle_sp_price - row.optionstrike
                        )
        elif row.pred < 0:
            if row.optionstrike >= row.today_sp_price -
                row.optionclosingprice  and row.optiontype
                == 1:
                if row.settle_sp_price <= row.
                    optionstrike:
                    #print row.optionclosingprice
                    return row.optionclosingprice*
                        multiplier
                else:
                    #print row.optionclosingprice -
                        multiplier * (row.settle_sp_price
                        - row.optionstrike)
                    return row.optionclosingprice*
                        multiplier - multiplier * (row.
                        settle_sp_price - row.optionstrike
                        )
        else:
            return 0
def getPl(df, pred_dict):
    pl0 = df.apply(lambda row: compPL(row, pred_dict, 1),
        axis=1)
    df['pl'] = pl0


# In[58]:

### result of third strategy


# In[59]:

getPl(testdf, predDict)
```

```
testdf['pl'].fillna(0, inplace=True)
interestedDf3 = testdf
interestedDf3 = testdf[testdf.pl!=0]


# In[60]:

positivedf = interestedDf3
positivedf = interestedDf3[interestedDf3.pl > 0]
negadf = interestedDf3
negadf = interestedDf3[interestedDf3.pl < 0]


# In[61]:

negadf.describe()


# In[62]:

positivedf.describe()


# In[63]:

def plSummary(df):
    positivedf = df
    neagtivedf = df
    positivedf = df[df.pl > 0]
    negativedf = df[df.pl < 0]
    print "profit is "+ str(positivedf['pl'].sum())
    print "loss is " + str(negativedf['pl'].sum())


interestedDf3.describe()
plSummary(interestedDf3)


### calculate sharpe ratio based on days to settle date
    and plot P&l distribution based on days to settle
    date

# In[64]:

#get day_difference list
def createDayDiffDict(row, dayList):
```

```
    if row.daystosettle not in dayList:
        dayList.append(row.daystosettle)

def getDayDiff(df):
    dayDiff = []
    df.apply(lambda row: createDayDiffDict(row, dayDiff),
        axis=1)
    dayDiff.sort()
    return dayDiff

dayDiffList = getDayDiff(interestedDf3)


# In[65]:

## draw pl distribution for each number of days
def plotDistribution(df):
    groups = df.groupby('daystosettle')
    fig, ax = plt.subplots()
    fig.suptitle('Distribution plot group by day of
       difference', fontsize=15)
    ax.margins(0.05)

    for name, group in groups:
        group.pl.plot(kind='kde', label=name)
        ##ax.plot(group.real, group.pred, marker='.',
            linestyle='', ms=10, label=name)
    # Put a legend to the right of the current axis
    ax.legend(loc='center left', bbox_to_anchor=(1, 0.5))


plotDistribution(interestedDf3)


# In[66]:

def computeSharpeRatio(df):
    return df['pl'].mean() / df['pl'].std()

def getSharpeRatio(df):
    groups = df.groupby('daystosettle')
    res = []
    for name, group in groups:
        tmp = computeSharpeRatio(group)
        res.append((name, tmp))
    return res
```

```
SharpeList = getSharpeRatio(interestedDf3)
print SharpeList
```

3 Experimental Evaluation

In this part, we are going to talk about the performance of these strategies and calculate the Sharpe Ratio of the winner grouped by the number of days to settle. Also, we will implement random strategies which means we random choose what we want to do with the option, either buy or sell or do nothing. Finally, we are going to prove the winning strategy is statistically different from a break-even strategy.

3.1 The performance of three strategies

The test data we use to back-test these strategies is from June 30, 2015 to March 1, 2017. By implement the first strategy, we are trading 1184 out of 10389 options of different time. We are making money by trading 963 of them and we profit 10461. On the other hand, we are losing money by trading the remaining 221 of them, and we lose 7311.

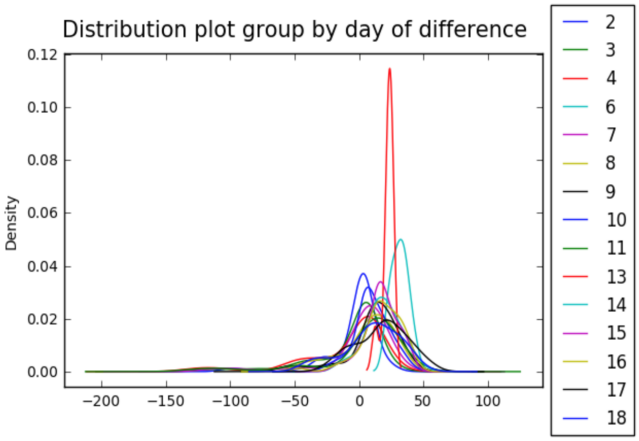
Because of the second strategy is more restrictive, we are only trading 650 of them. And we are making 3592 by trading 529 of them and lose 4124 for trading the rest.

For the third strategy, we are expected to trade more of them. The total trading number is 1931, and the profit of 1477 of them 21016.59 and the loss of the rest is 14487.965.

From the above result, we have a reasonable guess that it might be advantages to take a short position for the option becasue most of the option will expire worthless which leads to the result that more trade makes more money.

3.2 Analysis of the winning strategy

To further analyze the winning strategy, we group the options we are trading by number of days to the settlement date draw their profit and loss distribution as follows.



And then we calculate their Sharpe Ratio. The result is as follows:(it is ordered according to the number of days to settle date)

(2, -0.16476283326599236), (3, -0.26149427421593485), (4, -0.3081070564303187), (6, 4.874166118930129), (7, -0.0017944492300422522), (8, 0.3207765376211285), (9, 0.5124990392622327), (10, 0.7570763248230924), (11, 0.06857511455022872), (13, 5.611222538009364), (14, 1.3293559450765022), (15, 1.3659055129805246), (16, 1.0628867689638228), (17, 0.891318428060952), (18, 0.42899266398168934)

From the distribution graph and Sharpe Ratio, we can see that when the number of days to settle date is 6 and 13, it has pretty good result.

3.3 In comparison to random strategy

For the random strategy, we use the same test data from June 30, 2015 to March 1, 2017. For each different option, we randomly choose what we want to do with it, either buy or sell or nothing. Therefore the total number of trading would be 10461. We ran it ten times and record their total profit and loss. The result is : [-1194.1300000000128, -1595.6650000000084, 867.8349999999972, -1407.7449999999967, 4137.859999999979, -389.21499999999946, -269.7950000000052, -299.6099999999954, 1946.4049999999875, 844.9649999999891]. From the result, we can see that random strategy is a break-even strategy.

Further, we ran one sample t test on random strategy and the winning strategy. The null hypothesis is that they are from the same distribution. The p value of the test is 1.4311431494958151e-09. Therefore, we can reject our null hypothesis. And we can conclude that our winning strategy is statistically different from a break-even strategy.

Code:

Listing 4: Random Strategy

```
## Comparison with Random Startegies

# In[68]:
```

```
## get a sample of random profit

import random
def getSamples(num, testdf):
    res = []
    i = 1
    while i <= num:
        i += 1
        experimentalDf = testdf
        experimentalDf['pl'] = 0
        experimentalDf['buyorsell'] = 0
        getBuyOrSell(experimentalDf, options)
        getPl(experimentalDf)
        experimentalDf['pl'].fillna(0, inplace=True)
        res.append(experimentalDf['pl'].sum())
        print "profit is " + str(experimentalDf['pl'].sum
            ())
    return res

# random choice either buy or sell or do nothing
options = [-1,1, 0]

def getBuyOrSell(df, options):
    option = df.apply(lambda row: buyOrSell(row, options)
        , axis=1)
    df['buyorsell'] = option

def buyOrSell(row, options):
    return random.choice(options)


# computer p&l based on the random choice
def compPL(row):
    if row.buyorsell == 0:
        return 0
    elif row.buyorsell == 1:
        pl = -row.optionclosingprice
        if row.optiontype == 1:
            if row.optionstrike < row.settle_sp_price:
                pl = pl + row.settle_sp_price - row.
                    optionstrike
        else:
            if row.optionstrike > row.settle_sp_price:
                pl = pl + row.optionstrike - row.
                    settle_sp_price
```

```
        return pl
    else:
        pl = row.optionclosingprice
        if row.optiontype == 1:
            if row.optionstrike < row.settle_sp_price:
                pl = pl -(row.settle_sp_price - row.
                    optionstrike)
        else:
            if row.optionstrike > row.settle_sp_price:
                pl = pl -(row.optionstrike - row.
                    settle_sp_price)
        return pl

def getPl(df):
    pl0 = df.apply(lambda row: compPL(row), axis=1)
    df['pl'] = pl0

sample = getSamples(10, testdf)


# In[69]:

## test weather third strategy and random starategy are
    from the same distribution
from scipy.stats import ttest_ind

## gen a list of PL from third strategy
def genStrategyList(res, num):
    i = 1
    reslist = []
    while i <= num:
        i = i+1
        reslist.append(res)
    return reslist
stra3 = genStrategyList(6529,10)

## null hypothesis: third strategy and random starategy
    are from the same distribution; get p value
ttest_ind(sample,stra3)
```

4 Other trials

We tried to add some new signals to improve model performance. Therefore, we include Black Scholes price for each option and add current BS price over yesterday BS price and current BS price over last week's BS price for each option.

And then use the same method as described above. The new model performance: mean squared error for training set is : 55.892807 and mean squared error for test set is 1739.970828.

5 Code Repository

To get code, see Machine Learning for Option Trading and Prediction with Black-Scholes Price