Option Pricing Prediction Model

Master of Science in Business Analytics, University of Southern California
DSO 530: Applied Modern Statistical Learning Methods
Dr. Xin Tong

Hsuan-Ting Wu (3400005886)
Liang-Chi Liu (3944052733)
Nattawut Kananusorn (5053473152)
Suwara Thianrungrot (8623977364)
Yi-Ching Lin (7613130093)
contact email: lliu4737@usc.edu

May 5, 2022

Executive summary

The Nobel Prize winning Black-Scholes formula is a valuable method for evaluating European call options, but could a machine learning model deliver results that are just as good? This project aims to build a supervised learning model that can predict the value of an option and whether the option is worth purchasing. A dataset from S&P 500 contains 1,680 records of stock options and 6 fields, which includes C (current option value), S (current asset value), K (strike price), tau (time to maturity in years), r (annual interest rate) and BS (Over/Under based on whether (result of Black-Scholes formula - C) > 0). To go further in our analysis, we also cleaned the data to ensure the data quality is beyond the standard.

Before building the models, we decided to keep S, K, tau and r as variables since these four variables are all necessary for predicting value and BS in the practical business world. We also standardized these values as another variable group to compare with the non-standardized group to see which sets will have better performance when putting in the models.

Then, we came up with two potential approaches for both regression and classification problems. As for the first approach, we split the dataset into training and testing sets. Next, GridSearchCV was performed with 5-fold cross validation to find the best hyperparameters on 7 regression models and 9 classification models utilizing the training set. After all models were tuned, we compared the out-of-sample R squared in regression models as well as the accuracy on the testing set in classification models as one of the decision factors for choosing the final model. In addition, we also took its stability of cross validation scores in training set into consideration. To sum up, we find out that the performance of standardized variables are more stable than non-standardized variables. Therefore, for the second approach, we decided to use the entire dataset with standardized variables in our models. We conducted GridSearchCV with 10-fold cross validation to find the best hyperparameters on all models and check the out-of-sample R squared in regression models and accuracy scores in classification models. After running these two approaches, we decided to select the second approach to select our final model due to underfitting issues in the first approach.

For our final models, a Gradient Boosting Tree model was selected for the regression problem for its high 99.922% out-of-sample R-squared value. A XGBoost model was selected for the classification problem due to a high accuracy of 94.328%.

In conclusion, the final model can be used to predict the S$P500 call options value and make a decision whether the option is worth purchasing. Currently, we can find out that our machine learning models might outperform the Black-scholes formula, when predicting values of European call options. This is because the machine learning model is more flexible with the capability of adding more variables and tuning hyperparameters. We can keep track of the model performance and periodically validate the model. However, one thing we need to remind the user is that we cannot use our training models to predict any other stock price because different assets might have different nature shown on the data.

Project Background

A European call option is a type of contract that allows the holder to exercise their option to buy shares at the strike price on the day of its expiration date. For example, an investor purchases a call option of stock X on June 2 with a strike price of $100 and an expiration date on June 30. These are some possible scenarios on June 30:

- Stock X's price is higher than $100, the investor can then exercise their option and purchase it with only $100.
- Stock X's price is lower than $100, the investor can choose to not exercise their option.

Project Objectives

The objective of this project is to build a supervised learning model that can predict the value of an option and whether the option is worth purchasing.

Our Data Set

The data was provided to us from the S&P 500. It contains 6 variables and 1,680 rows, with each row being an individual option. There are 5 numeric variables, which include C (current option value), S (current asset value), K (strike price), Tau (time to maturity in years) and r (annual interest rate). There is only one categorical variable which is BS (Over/Under based on whether (result of Black-Scholes formula - C) > 0). Detailed fields description can be found in technical appendix Table 1 and 2.

Data Cleaning

After doing some data exploration, we discovered 2 rows with null values (Table 3 in technical appendix). We decided to drop them as the empty fields could have an effect on the model.

We observed a record with a value of 0 in the S field. This is considered an outlier as it is unlikely that a stock will be worth $0 plus the all other values here are between 375 and 490 (Figure 1 in technical appendix). We decided to drop this row.

Two other outliers were detected in the tau field with values of 146 and 250. Since stock options should not have a maturity time of over 100 years, and all other values here are between 0.0039 and 0.3929 (Figure 2 in technical appendix), we decided to drop these 2 rows.

After dealing with outliers, we convert BS to a binary column:

- If the BS is Under, then it converts to 0.
- If the BS is Over, then it converts 1.

Lastly, we created 5 additional columns by standardizing the variables S, K, r, tau, and value using the StandardScaler library in Python to make sure all variables weigh the same.

In total, we dropped 5 rows and created 5 new variables. The end result was a dataset of 1,675 rows and 11 columns, which will be used for building our models.

Variable Selection

Currently, we have 8 independent variables, which are S, K, r, Tau and their standardized values. We split them into a non-standardized group and a standardized group. We will not perform subset selection within the group since these four variables are all necessary for predicting value and BS. In practice, options traders use the price of the underlying security, time, and volatility to estimate an option's fair value. In terms of the price of the underlying security, it is related to S, K and r. Tau stands for time, which approximates the risk and the uncertainty. In conclusion, we can find that these four variables are actually being used in a real life setting. Therefore, we will use either the non-standardized group or the standardized group with these four variables in our regression and classification models.

Model Selection

We came up with two potential approaches to conduct model selection for both regression and classification problems:

1st Approach (The results are shown in technical appendix Table 6 and 7)

1. Set up the random seed as 20, separate the dataset with 80:20 ratio and stratify by y.
2. Create a 5-fold CV with KFold/StratifiedKFold and set shuffle = True.
3. Use non-standardized variables in the training set and conduct cross validation with 5-fold CV created in step two to find the best hyperparameters for each model utilizing GridSearchCV. Models and corresponding hyperparameters used in GridSearchCV can be found in technical appendix Table 4&5.
4. Validate tuned models with testing set and compare
   (1) Out of sample R squared on the testing set for regression models.
   (2) Accuracy scores on testing set and the variation of accuracy scores in training set from cross validation for classification models.
5. Repeat steps 1 to 4 again with standardized variables and compare the results.

2nd Approach

1. Create a 10-fold CV with KFold/StratifiedKFold and set shuffle = True.
2. Since we found that standardized variables perform better in terms of stability in the first approach, we only use standardized variables in the whole dataset and conduct cross validation with 10-fold CV created in step two to find the best hyperparameters for each model utilizing GridSearchCV. The models and corresponding hyperparameters used in GridSearchCV are the same as the first approach.
3. Compare different models with average out of sample R squared for regression models and average accuracy score for classification models.

Final Approach and Key Result

Final Approach

Eventually, we have selected the second approach to solve these problems. After we trained the model with the first approach, we found that the 5-fold average out-of-sample R-squared and accuracy score on the training set (80% split data) were lower than those on the testing set (20%

split data). This is the underfitting problem when the model is unable to capture the relationship between the input and output variables accurately. In addition, our data set is fixed and extremely small, having only 1,680 rows which might not be large enough to achieve the goal of training and validation data set. For a training data set, the more data we have, the better performance we achieve on the model. For a validation data set, it should be large to be fair and avoid any random errors. Therefore, we have decided to use the whole data set to train the model and achieve these two conditions by using 10-fold cross validation to find the best parameter and the best performance model. However, we kept with the standardized data because we found that it gives the accuracy score on each run (5-fold cross validation) less volatile.

Final Model for Regression

We determined that the final model for regression problem is Gradient Boosting Tree with the following hyperparameters:

- Criterion: friedman_mse
- max_depth: 4
- min_leaf: 1
- min_split: 5
- n_estinators: 800
- learning_rate: 0.1

This model performs the best with 99.922% average out-of-sample R-squared. Table 1 below shows the hyperparameters and results of each model for Regression.

Table 1: Hyperparameters and results of each model for Regression

Model	Parameters						R2
Linear Regression							99.242%
Decision Tree	Criterion	max_depth	min_leaf	min_split	splitter		99.264%
	squared_error	None	1	2	best		
Random Forest	Criterion	max_depth	min_leaf	min_split	n_estimators		99.651%
	mae	None	1	2	200		
Gradient Boosting Tree	Criterion	max_depth	min_leaf	min_split	n_estimators	learning_rate	99.922%
	friedman_mse	4	1	5	800	0.1	
XGBoost	learning_rate	max_depth	n_estimators				99.916%
	0.1	4	800				
LightGBM	learning_rate	max_depth	n_estimators				99.570%
	0.1	-1	200				
Neural Network	activation	# of nodes	# of layer	learning_rate			99.242%
	relu	10	2	adaptive			

Final Model for Classification

We determined that the final model for classification problem is XGBoost with the following hyperparameters:

- colsample_bytree: 0.8
- gamma: 1.5

- max_depth: 4
- min_child_weight: 1
- subsample: 0.6

This model performs the best with 94.328% mean accuracy score or 5.672 classification error. Table 2 below shows the hyperparameters and results of each model for Classification.

Table 2: Hyperparameters and results of each model for Classification

Model	Parameters					Accuracy
Logistic Regression	penalty	solver	C			91.465%
	l2	newton-cg	0.1			
Decision Tree	max_depth	min_leaf	min_split			91.637%
	9	1	2			
Random Forest	max_depth	min_leaf	min_split	n_estimators		93.669%
	9	1	5	150		
Gradient Boosting Tree	max_depth	min_leaf	min_split	n_estimators	learning_rate	93.614%
	5	4	5	100	0.1	
XGBoost	colsample_bytree	gamma	max_depth	min_child_weight	subsample	94.328%
	0.8	1.5	4	1	0.6	
LightGBM	lambda_l1	lambda_l2	max_depth	n_estimators		93.970%
	0	1	7	100		
Neural Network	activation	hidden_layer_sizes	learning_rate			93.489%
	relu	(100, 2)	invscaling			
SVM	kernel	gamma	C			93.312%
	rbf	0.1	10			
KNN	n_neighbors					93.072%
	9					

Conclusion and Business Understanding

In conclusion, the predictive model has been built in which we are able to predict a Value and BS of S&P 500 call options for a new data set. Firstly, the data set was explored and cleaned of the missing value and any entries errors. We also converted BS from 'Under' and 'Over' to number '0' and '1', respectively. All variables are standardized before being used to train the model to reduce the difference of their measures. All four variables, S, K, r, Tau, are included in the model due to the nature of call options related to those variables and usually being used in a real life setting. Then, we have tried several approaches and eventually applied the 10-fold cross validation to find the best hyperparameters in several algorithms we used on both regression and classification problems as our final approach. The best model for regression problem is Gradient Boosting Tree, resulting in 99.922% average out-of-sample R-squared while the best model for classification problem is XGBoost, resulting in 94.328% average accuracy score.

There are two issues to be considered after we have built the model to ensure the future use of the model. Firstly, the machine learning model might perform better than the Black-scholes, a concept currently used in the industry, because it is more flexible and adaptable. While the Black-scholes is fixed with some assumptions which are not realistic, the machine learning model can add more dimensions of variables and can tune the hyperparameter to have the model that most reflects reality. Another interesting point is that machine learning models can predict what is the same underlying security on training data. Like our model, it can predict new S&P 500 call options, but cannot predict anything else because different assets might have different nature shown on the data.

Technical Appendix

Table 1: Numeric fields description

Variable	Description	Data type	% Populated	Min	Max
Value (C)	Current option value	Float	99%	0.125	60.15
S	Current asset value	Float	99%	0	455.89
K	Strike price of option	Float	99%	375	500
tau	Time to maturity in years	Float	99%	0.0039	250
r	Annual interest rate	Float	100%	0.0295	0.0319

Table 2: Categorical fields description

Variable	Description	Data type	% Populated	Most common value	Unique values
BS	Over/Under based on whether (result of Black-Scholes formula - C) > 0	String	100%	Over	2

Table 3: Data with Missing Values

Index	Value	S	K	tau	r	BS
292	8.625	NaN	NaN	NaN	0.03003	Over
818	NaN	431.2846	NaN	0.230159	0.02972	Over

Figure 1: Distribution of S and K

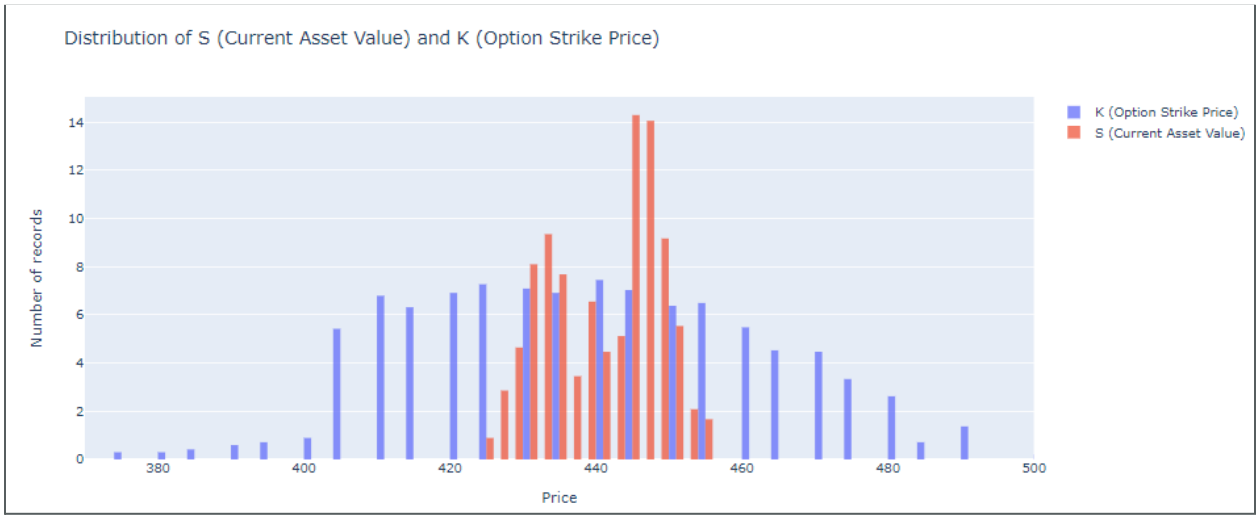

Figure 2: Box plot of tau

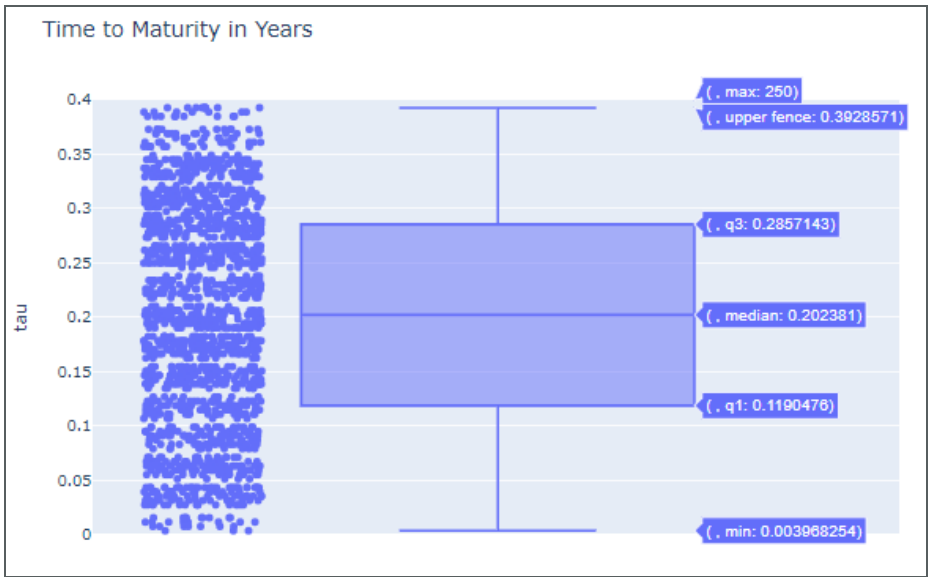

Table 4: Models and hyperparameters applied in GridSearchCV (Regression)

Model	Parameter	Values
Linear regression	NA	NA
Random Forest	max_depth	[2, 4, 5, 10, none]
	min_samples_leaf	[1, 5, 10]
	min_samples_split	[2, 5, 10, 20]
	n_estimators	[100, 150, 200]
	criterion	[mse, mae, poisson]
Gradient Boosting Classifier	learning_rate	[0.01, 0.1]
	max_depth	[2, 4, 5, 10, none]
	min_samples_leaf	[1, 5, 10]
	min_samples_split	[2, 5, 10]
	n_estimators	[100, 150, 200, 500, 800]
	criterion	[friedman_mse, mse, mae]
XGBoost	n_estimators	[100, 150, 200, 500, 800]
	learning_rate	[0.1, 0.01]
	max_depth	[2, 4, 5, 10, none]
Neural Network	learning_rate	[constant, invscaling, adaptive]
	hidden_layer_sizes	[(3,), (5,), (10,), (3,3), (5,5), (10,10)]
	activation	[logistic, relu, Tanh]
Decision Tree	max_depth	[2, 4, 5, 10, none]
	min_samples_leaf	[1, 5, 10]
	min_samples_split	[2, 5, 10, 20, 50]
	criterion	[squared_error, friedman_mse, absolute_error, poisson]
	splitter	[best, random]
LightGBM	max_depth	[2, 4, 5, 10, -1]
	n_estimators	[100, 150, 200, 500]
	learning_rate	[0.1, 0.01]

Table 5: Models and hyperparameters applied in GridSearchCV (Classification)

Model	Parameter	Values
Logistic regression	penalty	[11, 12]
	C	[0.001, 0.01, 0.1, 1, 10, 100, 1000]
	solver	[newton-cg, lbfgs, liblinear]
Random Forest	max_depth	[2, 3, 4, 5, 6, 7, 8, 9, 10]
	min_samples_leaf	[1, 2, 4]
	min_samples_split	[2, 5, 10]
	n_estimators	[50, 100, 150, 200]
Gradient Boosting Classifier	learning_rate	[0.001, 0.01, 0.1]
	max_depth	[2, 5, 10]
	min_samples_leaf	[1, 2, 4]
	min_samples_split	[2, 5, 10]
	n_estimators	[10, 100]
SVM	C	[0.1, 1, 10, 100, 1000]
	gamma	[1, 0.1, 0.01, 0.001, 0.0001]
	kernel	[rbf]
XGBoost	min_child_weight	[1, 5, 10]
	gamma	[0.5, 1, 1.5, 2, 5]
	subsample	[0.6, 0.8, 1.0]
	colsample_bytree	[0.6, 0.8, 1.0]
	max_depth	[3, 4, 5]
KNN	n_neighbor	[3, 4, 5, 6, 7, 8, 9, 10, 11, 12]
Neural Network	learning_rate	[constant, invscaling, adaptive]
	hidden_layer_sizes	[(100,1), (100,2), (100,3)]
	activation	[logistic, relu, Tanh]
Decision Tree	max_depth	[2, 3, 4, 5, 6, 7, 8, 9, 10]

	min_samples_leaf	[1, 2, 4]
	min_samples_split	[2, 5, 10]
LightGBM	max_depth	[2, 3, 4, 5, 6, 7, 8, 9, 10]
	n_estimators	[50, 100, 150, 200]
	lambda_l1	[0, 1, 1.5]
	lambda_l2	[0, 1]

Table 6: Hyperparameters and results of each model for Regression in the first Approach

Model	Parameters						R2	
Linear Regression							Train	Test
							91.083%	91.140%
Decision Tree	Criterion	max_depth	min_leaf	min_split	splitter		Train	Test
	absolute_error	18	1	2	random		98.632%	98.859%
Random Forest	Criterion	max_depth	min_leaf	min_split	n_estimators		Train	Test
	mse	None	1	2	100		99.460%	99.552%
Gradient Boosting Tree	Criterion	max_depth	min_leaf	min_split	n_estimators	learning_rate	Train	Test
	mse	4	1	10	800	0.1	99.874%	99.902%
XGBoost	learning_rate	max_depth	n_estimators				Train	Test
	0.1	4	800				99.858%	99.898%
LightGBM	learning_rate	max_depth	n_estimators				Train	Test
	0.1	-1	150				99.346%	99.713%
Neural Network	activation	# of nodes	# of layer	learning_rate			Train	Test
	relu	10	2	adaptive			99.102%	98.998%

Table 7: Hyperparameters and results of each model for Classification in the first Approach

Model	Parameters					Accuracy	
Logistic Regression	penalty	solver	C			Train	Test
	l2	newton-cg	1			91.493%	91.642%
Decision Tree	max_depth	min_leaf	min_split			Train	Test
	9	2	2			90.746%	92.239%
Random Forest	max_depth	min_leaf	min_split	n_estimators		Train	Test
	10	2	5	50		92.239%	92.537%
Gradient Boosting Tree	max_depth	min_leaf	min_split	n_estimators	learning_rate	Train	Test
	5	4	2	100	0.1	92.239%	92.537%
XGBoost	colsample_bytree	gamma	max_depth	min_child_weight	subsample	Train	Test
	0.8	1	4	1	0.6	93.284%	93.134%
LightGBM	lambda_l1	lambda_l2	max_depth	n_estimators		Train	Test
	1	1	4	150		92.836%	93.731%
Neural Network	activation	hidden_layer_sizes	learning_rate			Train	Test
	relu	(100, 3)	constant			93.209%	93.134%
SVM	kernel	gamma	C			Train	Test
	rbf	0.1	100			92.687%	93.433%
KNN	n_neighbors					Train	Test
	9					92.015%	92.537%

Regression

May 5, 2022

[ ]:
```
import pandas as pd
import numpy as np
import pandas as pd
from sklearn.preprocessing import StandardScaler
from sklearn.model_selection import train_test_split
from sklearn.linear_model import LinearRegression
from sklearn.tree import DecisionTreeRegressor
from sklearn.ensemble import RandomForestClassifier
from sklearn.ensemble import RandomForestRegressor
from sklearn.ensemble import GradientBoostingRegressor
from xgboost import XGBRegressor
from sklearn.neural_network import MLPRegressor
from sklearn.neighbors import KNeighborsClassifier
from sklearn.model_selection import cross_val_score
from lightgbm import LGBMRegressor
import matplotlib.pyplot as plt
from sklearn.model_selection import KFold, StratifiedKFold
from sklearn.model_selection import GridSearchCV
import lightgbm as lgb
import seaborn as sns
from sklearn import tree
import lightgbm as lgb
from xgboost import XGBClassifier
from sklearn.linear_model import LogisticRegression
from sklearn.naive_bayes import GaussianNB, MultinomialNB, BernoulliNB
from sklearn.tree import DecisionTreeClassifier
from sklearn.ensemble import GradientBoostingClassifier
from sklearn.svm import SVC
from sklearn.ensemble import VotingClassifier
from sklearn.metrics import accuracy_score, precision_score, recall_score,␣
 ↪f1_score, confusion_matrix, classification_report, mean_squared_error,␣
 ↪r2_score
from sklearn.utils import class_weight
from sklearn.neural_network import MLPClassifier

import warnings
warnings.filterwarnings("ignore")
```

1 Import data

[ ]:
```
data = pd.read_csv('option_train.csv')
```

[ ]:
```
data.head()
```

Encode BS column - 1 is Over, 0 is Under

[ ]:
```
data['BS_encode'] = [1 if i == 'Over' else 0 for i in data['BS']]
```

[ ]:
```
data.shape
```

2 Data filtering

2.1 Drop 2 records that has null values

[ ]:
```
data.dropna(axis=0, inplace = True)
```

2.2 Drop 3 outliers

[ ]:
```
data.describe()
```

[ ]:
```
data.sort_values(by = 'tau',ascending = False)
```

[ ]:
```
data.sort_values(by = 'S',ascending = True)
```

[ ]:
```
data.drop([12,33,879], inplace = True)
```

3 Z-Scaling before building the models

[ ]:
```
stdx = StandardScaler()
stdy = StandardScaler()
X = data[['S','K','tau','r']]
y_value = data['Value']
y_BS = data['BS_encode']
X_z = stdx.fit_transform(data[['S','K','tau','r']])
y_z = (stdy.fit_transform(data[['Value']])).reshape(-1)
```

[ ]:
```
print(X_z.shape)
print(y_z.shape)
print(y_BS.shape)
```

4 Method 1

4.1 Regression Models

[ ]:
```
X_train, X_test, y_train, y_test = train_test_split(X_z, y_z, test_size = .2,␣
↪random_state = 20)
```

4.1.1 Linear Regression

There is no hyperparameter that can be tuned to improve the r2

[ ]:
```
kf5 = KFold(n_splits = 5, shuffle = True)

LR = LinearRegression()
cv = cross_val_score(LR, X_train, y_train, cv = kf5)
print(cv, 'mean: ', cv.mean())

LR.fit(X_train, y_train)
y_pred = LR.predict(X_test)
print(LR.score(X_test, y_test))
print(mean_squared_error(y_test, y_pred))
```

4.1.2 Decision Tree

[ ]:
```
kf5 = KFold(n_splits = 5, shuffle = True)
DT_param = {'criterion' :␣
↪['squared_error','friedman_mse','absolute_error','poisson'],
            'splitter' : ['best','random'],
             'max_depth': list(range(2,21,2))+[None],
             'min_samples_split'  : [2,5,10,20,50],
             'min_samples_leaf'  : [1,5,10]}

DT = DecisionTreeRegressor()
DT_cv = GridSearchCV(DT, DT_param, cv = kf5, refit=True, verbose=3)
DT_cv.fit(X_train, y_train)
print(DT_cv.best_score_)
print(DT_cv.best_params_)
```

[ ]:
```
DT = DecisionTreeRegressor(criterion = 'absolute_error', max_depth = 18,␣
↪min_samples_leaf = 1, min_samples_split = 2, splitter = 'random')
cv = cross_val_score(DT, X_train, y_train, cv = kf5)
print(cv, 'mean: ', cv.mean())

DT.fit(X_train, y_train)
y_pred = DT.predict(X_test)
print(DT.score(X_test, y_test))
print(mean_squared_error(y_test, y_pred))
```

4.1.3 Random Forest

```
[ ]: kf5 = KFold(n_splits = 5, shuffle = True)
RF_param = {'n_estimators' : [10, 50, 100, 150, 200],
            'criterion': ['mse', 'mae', 'poisson'],
             'max_depth': list(range(2,11,2))+[None],
             'min_samples_split'  : [2,5,10,20],
             'min_samples_leaf'  : [1,5,10]}

RF = RandomForestRegressor()
RF_cv = GridSearchCV(RF, RF_param, cv = kf5, refit=True, verbose=3)
RF_cv.fit(X_train, y_train)
print(RF_cv.best_score_)
print(RF_cv.best_params_)
```

```
[ ]: RF = RandomForestRegressor(criterion = 'mse', max_depth = None,␣
 ↪min_samples_leaf = 1, min_samples_split = 2, n_estimators = 100)
cv = cross_val_score(RF, X_train, y_train, cv = kf5)
print(cv, 'mean: ', cv.mean())

RF.fit(X_train, y_train)
y_pred = RF.predict(X_test)
print(RF.score(X_test, y_test))
print(mean_squared_error(y_test, y_pred))
```

4.1.4 Gradient Boosting Tree

```
[ ]: kf5 = KFold(n_splits = 5, shuffle = True)
GB_param = {'n_estimators' : [10, 20, 50, 100, 150, 200, 500, 800],
            'learning_rate' : [0.1, 0.01],
            'criterion' : ['friedman_mse', 'mse', 'mae'],
             'max_depth': [3, 4, 5, 10, None],
             'min_samples_split'  : [2,5,10],
             'min_samples_leaf'  : [1,5,10]}

GB = GradientBoostingRegressor()
GB_cv = GridSearchCV(GB, GB_param, cv = kf5, refit=True, verbose=3)
GB_cv.fit(X_train, y_train)
print(GB_cv.best_score_)
print(GB_cv.best_params_)
```

```
[ ]: GB = GradientBoostingRegressor(criterion = 'mse', learning_rate = 0.1,␣
 ↪max_depth = 4, min_samples_leaf = 1, min_samples_split = 10, n_estimators =␣
 ↪800)
cv = cross_val_score(GB, X_train, y_train, cv = kf5)
print(cv, 'mean: ', cv.mean())
```

```
GB.fit(X_train, y_train)
y_pred = GB.predict(X_test)
print(GB.score(X_test, y_test))
print(mean_squared_error(y_test, y_pred))
```

4.1.5 Xgboost

[ ]:
```
kf5 = KFold(n_splits = 5, shuffle = True)
XG_param = {'n_estimators' : [10, 50, 100, 150, 200, 500, 800],
            'learning_rate' : [0.1, 0.01],
             'max_depth': [3, 4, 5, 10, None]}

XG = XGBRegressor()
XG_cv = GridSearchCV(XG, XG_param, cv = kf5, refit=True, verbose=3)
XG_cv.fit(X_train, y_train)
print(XG_cv.best_score_)
print(XG_cv.best_params_)
```

[ ]:
```
XG = XGBRegressor(learning_rate = 0.1, max_depth = 4, n_estimators = 800)
cv = cross_val_score(XG, X_train, y_train, cv = kf5)
print(cv, 'mean: ', cv.mean())

XG.fit(X_train, y_train)
y_pred = XG.predict(X_test)
print(XG.score(X_test, y_test))
print(mean_squared_error(y_test, y_pred))
```

4.1.6 LGB

[ ]:
```
kf5 = KFold(n_splits = 5, shuffle = True)
LG_param = {'n_estimators' : [10, 50, 100, 150, 200, 500, 800],
            'learning_rate' : [0.1, 0.01],
             'max_depth': [3, 4, 5, 10, -1]}

LG = LGBMRegressor()
LG_cv = GridSearchCV(LG, LG_param, cv = kf5, refit=True, verbose=3)
LG_cv.fit(X_train, y_train)
print(LG_cv.best_score_)
print(LG_cv.best_params_)
```

[ ]:
```
LG = LGBMRegressor(learning_rate = 0.1, max_depth = -1, n_estimators = 150)
cv = cross_val_score(LG, X_train, y_train, cv = kf5)
print(cv, 'mean: ', cv.mean())

LG.fit(X_train, y_train)
y_pred = LG.predict(X_test)
```

```
print(LG.score(X_test, y_test))
print(mean_squared_error(y_test, y_pred))
```

4.1.7 Neural Network

[ ]:
```
kf5 = KFold(n_splits = 5, shuffle = True)
NN_param = {'learning_rate': ['constant', 'invscaling', 'adaptive'],
            'hidden_layer_sizes': [(3,), (5,), (10,), (3,3), (5,5), (10,10)],
            'activation': ["logistic", "relu", "Tanh"]}

NN = MLPRegressor()
NN_cv = GridSearchCV(NN, NN_param, cv = kf5, refit=True, verbose=3)
NN_cv.fit(X_train, y_train)
print(NN_cv.best_score_)
print(NN_cv.best_params_)
```

[ ]:
```
NN = MLPRegressor(activation = 'relu', hidden_layer_sizes = (10,10),␣
↪learning_rate = 'adaptive')
cv = cross_val_score(NN, X_train, y_train, cv = kf5)
print(cv, 'mean: ', cv.mean())

NN.fit(X_train, y_train)
y_pred = NN.predict(X_test)
print(NN.score(X_test, y_test))
print(mean_squared_error(y_test, y_pred))
```

4.1.8 Summary Result

[ ]:
```
regressions = [LinearRegression(),
               DecisionTreeRegressor(criterion = 'absolute_error', max_depth =␣
↪18, min_samples_leaf = 1, min_samples_split = 2, splitter = 'random'),
               RandomForestRegressor(criterion = 'mse', max_depth = None,␣
↪min_samples_leaf = 1, min_samples_split = 2, n_estimators = 100),
               GradientBoostingRegressor(criterion = 'mse', learning_rate = 0.
↪1, max_depth = 4, min_samples_leaf = 1, min_samples_split = 10, n_estimators␣
↪= 800),
               XGBRegressor(learning_rate = 0.1, max_depth = 4, n_estimators =␣
↪800),
               LGBMRegressor(learning_rate = 0.1, max_depth = -1, n_estimators␣
↪= 150),
               MLPRegressor(activation = 'relu', hidden_layer_sizes = (10,10),␣
↪learning_rate = 'adaptive')
               ]
```

[ ]:
```
r2 = []
mse = []
```

```
for regression in regressions:
    regression.fit(X_train, y_train)
    y_pred = regression.predict(X_test)
    r2.append(r2_score(y_test, y_pred))
    mse.append(mean_squared_error(y_test, y_pred))
```

[ ]:
```
models = pd.DataFrame({'Model': ['Linear Regression','Decision Tree', 'Random␣
↪Forest',
                                 'Gradient Boosting Tree', 'Xgboost', 'LGB',␣
↪'Neural Network'],
                       'r2':r2,
                       'mse':mse})
models.sort_values(by='r2', ascending=False)
```

5 Method 2

5.1 Regression Models

5.1.1 Linear Model

[ ]:
```
kf10 = KFold(n_splits = 10, shuffle = True)

LR = LinearRegression()
cv = cross_val_score(LR, X_z, y_z, cv = kf10)
LR_r2 = cv.mean()
print(cv, 'mean: ', cv.mean())
```

5.1.2 Decision Tree

[ ]:
```
kf10 = KFold(n_splits = 10, shuffle = True)
DT_param = {'criterion' :␣
↪['squared_error','friedman_mse','absolute_error','poisson'],
            'splitter' : ['best','random'],
             'max_depth': [2, 4, 5, 10, None],
             'min_samples_split'  : [2,5,10,20,50],
             'min_samples_leaf'  : [1,5,10]}

DT = DecisionTreeRegressor()
DT_cv = GridSearchCV(DT, DT_param, cv = kf10, refit=True, verbose=3)
DT_cv.fit(X_z, y_z)
DT_r2 = DT_cv.best_score_
DT_best_parm = DT_cv.best_params_
print(DT_cv.best_score_)
print(DT_cv.best_params_)
```

5.1.3 Random Forest

[ ]:
```
kf10 = KFold(n_splits = 10, shuffle = True)
RF_param = {'n_estimators' : [100, 150, 200],
            'criterion': ['mse', 'mae', 'poisson'],
             'max_depth': [2, 4, 5, 10, None],
             'min_samples_split'  : [2,5,10,20],
             'min_samples_leaf'  : [1,5,10]}

RF = RandomForestRegressor()
RF_cv = GridSearchCV(RF, RF_param, cv = kf10, refit=True, verbose=3)
RF_cv.fit(X_z, y_z)
RF_r2 = RF_cv.best_score_
RF_best_parm = RF_cv.best_params_
print(RF_cv.best_score_)
print(RF_cv.best_params_)
```

5.1.4 Gradient Boosting Tree

[ ]:
```
kf10 = KFold(n_splits = 10, shuffle = True)
GB_param = {'n_estimators' : [100, 150, 200, 500, 800],
            'learning_rate' : [0.1, 0.01],
            'criterion' : ['friedman_mse', 'mse', 'mae'],
             'max_depth': [2, 4, 5, 10, None],
             'min_samples_split'  : [2,5,10],
             'min_samples_leaf'  : [1,5,10]}

GB = GradientBoostingRegressor()
GB_cv = GridSearchCV(GB, GB_param, cv = kf10, refit=True, verbose=3)
GB_cv.fit(X_z, y_z)
GB_r2 = GB_cv.best_score_
GB_best_parm = GB_cv.best_params_
print(GB_cv.best_score_)
print(GB_cv.best_params_)
```

5.1.5 XGBoost

[ ]:
```
kf10 = KFold(n_splits = 10, shuffle = True)
XG_param = {'n_estimators' : [100, 150, 200, 500, 800],
            'learning_rate' : [0.1, 0.01],
             'max_depth': [2, 4, 5, 10, None]}

XG = XGBRegressor()
XG_cv = GridSearchCV(XG, XG_param, cv = kf10, refit=True, verbose=3)
XG_cv.fit(X_z, y_z)
XG_r2 = XG_cv.best_score_
XG_best_parm = XG_cv.best_params_
```

```
print(XG_cv.best_score_)
print(XG_cv.best_params_)
```

5.1.6 LGB

[ ]:
```
kf10 = KFold(n_splits = 10, shuffle = True)
LG_param = {'n_estimators' : [100, 150, 200, 500],
            'learning_rate' : [0.1, 0.01],
             'max_depth': [2, 4, 5, 10, -1]}

LG = LGBMRegressor()
LG_cv = GridSearchCV(LG, LG_param, cv = kf10, refit=True, verbose=3)
LG_cv.fit(X_z, y_z)
LG_r2 = LG_cv.best_score_
LG_best_parm = LG_cv.best_params_
print(LG_cv.best_score_)
print(LG_cv.best_params_)
```

5.1.7 Neural Network

[ ]:
```
kf10 = KFold(n_splits = 10, shuffle = True)
NN_param = {'learning_rate': ['constant', 'invscaling', 'adaptive'],
            'hidden_layer_sizes': [(3,), (5,), (10,), (3,3), (5,5), (10,10)],
            'activation': ["logistic", "relu", "Tanh"]}

NN = MLPRegressor()
NN_cv = GridSearchCV(NN, NN_param, cv = kf10, refit=True, verbose=3)
NN_cv.fit(X_z, y_z)
NN_r2 = NN_cv.best_score_
NN_best_parm = NN_cv.best_params_
print(NN_cv.best_score_)
print(NN_cv.best_params_)
```

5.1.8 Summary Result

[ ]:
```
models = pd.DataFrame({'Model': ['Linear Regression','Decision Tree', 'Random␣
 ↪Forest',
                                  'Gradient Boosting Tree', 'Xgboost', 'LGB',␣
 ↪'Neural Network'],
                        'parm': ['', DT_best_parm, RF_best_parm, GB_best_parm,␣
 ↪XG_best_parm, LG_best_parm, NN_best_parm],
                        'r2':[LR_r2, DT_r2, RF_r2, GB_r2, XG_r2, LG_r2, NN_r2]})
models.sort_values(by='r2', ascending=False).reset_index(drop = True)
```

6 Predict Test data

[ ]:
```
GB_best_parm
```

[ ]:
```
test_data = pd.read_csv('option_test_wolabel.csv')
```

[ ]:
```
test_data.head()
```

Scale test data

[ ]:
```
X_test_scale = stdx.transform(test_data)
```

Train the model with the best parameter and all training data set

[ ]:
```
GB_use = GradientBoostingRegressor(
    criterion = 'friedman_mse',
    learning_rate = 0.1,
    max_depth = 4,
    min_samples_leaf = 1,
    min_samples_split = 5,
    n_estimators = 800)
GB_use.fit(X_z,y_z)
y_pred_z = GB_use.predict(X_test_scale)
```

[ ]:
```
y_pred_z
```

Transform value back to unscaled data

[ ]:
```
y_pred = stdy.inverse_transform(y_pred_z.reshape(-1,1))
```

[ ]:
```
test_data['value'] = y_pred
```

[ ]:
```
test_data
```

[ ]:
```
test_data.to_csv('test_data_pred.csv')
```

Classification

May 5, 2022

[ ]:
```
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
import seaborn as sns
```

[ ]:
```
# read data
df = pd.read_csv('option_train.csv')
df.head()
```

[ ]:
```
df.groupby('BS')['S'].count()
```

1 Data Cleaning

[ ]:
```
df.isnull().sum()
```

[ ]:
```
# change BS to dummy
df['BS'] = [1 if i == 'Over' else 0 for i in df['BS']]
df.head()
```

[ ]:
```
df.shape
```

[ ]:
```
# drop the row with null values
df = df.dropna(axis=0)
df.shape
```

[ ]:
```
# remove outliers
df = df[df['tau'] != 250]
df = df[df['tau'] != 146]
df = df[df['S'] != 0]
df.shape
```

2 Version 1

3 Preprocessing

[ ]:
```
from sklearn.model_selection import train_test_split

X = df.loc[:, ['S', 'K', 'tau', 'r']]
y = df.loc[:, 'BS']
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2,
                                                    stratify = y, ␣
↪random_state=20)
```

[ ]:
```
from sklearn.preprocessing import StandardScaler

scaler = StandardScaler()
scaled_X_train = scaler.fit_transform(X_train)
scaled_X_test = scaler.fit_transform(X_test)
```

[ ]:
```
print(scaled_X_train.shape)
print(y_train.shape)
```

4 Test Models

[ ]:
```
from sklearn.model_selection import cross_val_score,KFold ## for regression
from sklearn.model_selection import StratifiedKFold ## recommended for␣
↪classification
```

[ ]:
```
# importing the modules
import lightgbm as lgb
from xgboost import XGBClassifier
from sklearn.model_selection import cross_val_score, cross_validate,␣
↪GridSearchCV
from sklearn.linear_model import LogisticRegression
from sklearn.neighbors import KNeighborsClassifier
from sklearn.naive_bayes import GaussianNB, MultinomialNB, BernoulliNB
from sklearn.tree import DecisionTreeClassifier
from sklearn.svm import SVC
from sklearn.ensemble import RandomForestClassifier, GradientBoostingClassifier
from sklearn.metrics import accuracy_score, precision_score, recall_score,
f1_score, confusion_matrix, classification_report
from sklearn.utils import class_weight
from sklearn.neural_network import MLPClassifier
```

4.0.1 Logistic Regression

[ ]:
```
kf5 = StratifiedKFold(n_splits = 5, shuffle = True)
log_param = {'penalty' : ['l1','l2'],
             'C': np.logspace(-3,3,7),
             'solver'  : ['newton-cg', 'lbfgs', 'liblinear']}

log = LogisticRegression()
log_cv = GridSearchCV(log, log_param, cv = kf5, refit=True, verbose=3)
log_cv.fit(X_train, y_train)
print(log_cv.best_score_)
print(log_cv.best_params_)
```

[ ]:
```
log = LogisticRegression(C = 10, penalty = 'l1', solver= 'liblinear')
cv = cross_val_score(log, X_train, y_train, cv = kf5)
print(cv, 'mean: ', cv.mean())

log.fit(X_train, y_train)
y_pred = log.predict(X_test)
print(confusion_matrix(y_test, y_pred))
print(classification_report(y_test, y_pred))
print(log.score(X_test, y_test))
```

4.0.2 Random Forest

[ ]:
```
kf5 = StratifiedKFold(n_splits = 5, shuffle = True)
rf_param = {'max_depth': np.arange(2, 11),
            'min_samples_leaf': [1, 2, 4],
            'min_samples_split': [2, 5, 10],
            'n_estimators': np.arange(50, 201, 50)}

param = {'alpha':np.arange(0,1,0.1)}

rf = RandomForestClassifier()
rf_cv = GridSearchCV(rf, rf_param, cv=kf5, refit=True, verbose=3)
rf_cv.fit(X_train, y_train)
print(rf_cv.best_score_)
print(rf_cv.best_params_)
```

[ ]:
```
rf = RandomForestClassifier(max_depth= 9, min_samples_leaf= 2,
                            min_samples_split= 5, n_estimators= 200)
cv = cross_val_score(rf, X_train, y_train, cv = kf5)
print(cv, 'mean: ', cv.mean())

rf.fit(X_train, y_train)
y_pred = rf.predict(X_test)
print(confusion_matrix(y_test, y_pred))
```

```
print(classification_report(y_test, y_pred))
print(rf.score(X_test, y_test))
```

4.0.3 Gradient Boosting

[ ]:
```
kf5 = StratifiedKFold(n_splits = 5, shuffle = True)
gb_param = {
    "learning_rate": [0.001, 0.01, 0.1],
    'max_depth': [2, 5, 10],
    'min_samples_leaf': [1, 2, 4],
    'min_samples_split': [2, 5, 10],
    "n_estimators":[10, 100]
    }

gb = GradientBoostingClassifier()
gb_cv = GridSearchCV(gb, gb_param, cv=kf5, refit=True, verbose=3)
gb_cv.fit(X_train, y_train)
print(gb_cv.best_score_)
print(gb_cv.best_params_)
```

[ ]:
```
gb = GradientBoostingClassifier(learning_rate=0.1, max_depth= 5,
                                min_samples_leaf= 2, min_samples_split= 2,
                                n_estimators= 100)
cv = cross_val_score(gb, X_train, y_train, cv = kf5)
print(cv, 'mean: ', cv.mean())

gb.fit(X_train, y_train)
y_pred = gb.predict(X_test)
print(confusion_matrix(y_test, y_pred))
print(classification_report(y_test, y_pred))
print(gb.score(X_test, y_test))
```

4.0.4 SVM

[ ]:
```
kf5 = StratifiedKFold(n_splits = 5, shuffle = True)
svm_param = {'C': [0.1, 1, 10, 100, 1000],
             'gamma': [1, 0.1, 0.01, 0.001, 0.0001],
             'kernel': ['rbf']}

X = df.loc[:, ['S', 'K', 'tau', 'r']]
y = df.loc[:, 'BS']
svm = SVC()
svm_cv = GridSearchCV(svm, svm_param, cv=kf5, refit=True, verbose=3)
svm_cv.fit(X_train, y_train)
print(svm_cv.best_score_)
print(svm_cv.best_params_)
```

[ ]:
```
svm = SVC(C=1000, gamma=0.001, kernel='rbf')
cv = cross_val_score(svm, X_train, y_train, cv = kf5)
print(cv, 'mean: ', cv.mean())

svm.fit(X_train, y_train)
y_pred = svm.predict(X_test)
print(confusion_matrix(y_test, y_pred))
print(classification_report(y_test, y_pred))
print(svm.score(X_test, y_test))
```

4.0.5 Xgboost

[ ]:
```
kf5 = StratifiedKFold(n_splits = 5, shuffle = True)
kb_param = {'min_child_weight': [1, 5, 10],
          'gamma': [0.5, 1, 1.5, 2, 5],
          'subsample': [0.6, 0.8, 1.0],
          'colsample_bytree': [0.6, 0.8, 1.0],
          'max_depth': [3, 4, 5]}


X = df.loc[:, ['S', 'K', 'tau', 'r']]
y = df.loc[:, 'BS']
kb = XGBClassifier()
kb_cv = GridSearchCV(kb, kb_param, cv=kf5, refit=True, verbose=3)
kb_cv.fit(X_train, y_train)
print(kb_cv.best_score_)
print(kb_cv.best_params_)
```

[ ]:
```
kb = XGBClassifier(colsample_bytree = 1.0, gamma = 2, max_depth = 3,
                   min_child_weight = 1, subsample = 0.8)
cv = cross_val_score(kb, X_train, y_train, cv = kf5)
print(cv, 'mean: ', cv.mean())

kb.fit(X_train, y_train)
y_pred = kb.predict(X_test)
print(confusion_matrix(y_test, y_pred))
print(classification_report(y_test, y_pred))
print(kb.score(X_test, y_test))
```

4.0.6 KNN

[ ]:
```
kf5 = StratifiedKFold(n_splits = 5, shuffle = True)
knn_param = {'n_neighbors': [3, 4, 5, 6, 7, 8, 9, 10, 11, 12]}

knn = KNeighborsClassifier()
knn_cv = GridSearchCV(knn, knn_param, cv=kf5, refit=True, verbose=3)
```

```
knn_cv.fit(X_train, y_train)
print(knn_cv.best_score_)
print(knn_cv.best_params_)
```

[ ]:
```
knn = KNeighborsClassifier(n_neighbors = 7)
cv = cross_val_score(knn, X_train, y_train, cv = kf5)
print(cv, 'mean: ', cv.mean())

knn.fit(X_train, y_train)
y_pred = knn.predict(X_test)
print(confusion_matrix(y_test, y_pred))
print(classification_report(y_test, y_pred))
print(knn.score(X_test, y_test))
```

4.0.7 Neural Network

[ ]:
```
kf5 = StratifiedKFold(n_splits = 5, shuffle = True)
nn_param = {'learning_rate': ["constant", "invscaling", "adaptive"],
            'hidden_layer_sizes': [(100,1), (100,2), (100,3)],
            'activation': ["logistic", "relu", "Tanh"]}

nn = MLPClassifier()
nn_cv = GridSearchCV(nn, nn_param, cv=kf5, refit=True, verbose=3)
nn_cv.fit(X_train, y_train)
print(nn_cv.best_score_)
print(nn_cv.best_params_)
```

[ ]:
```
nn = MLPClassifier(activation = 'logistic', hidden_layer_sizes = (100, 2),␣
↪learning_rate = 'invscaling')
cv = cross_val_score(nn, X_train, y_train, cv = kf5)
print(cv, 'mean: ', cv.mean())

nn.fit(X_train, y_train)
y_pred = nn.predict(X_test)
print(confusion_matrix(y_test, y_pred))
print(classification_report(y_test, y_pred))
print(nn.score(X_test, y_test))
```

4.0.8 Decision Tree

[ ]:
```
kf5 = StratifiedKFold(n_splits = 5, shuffle = True)
dt_param = {'max_depth': np.arange(2, 11),
            'min_samples_leaf': [1, 2, 4],
            'min_samples_split': [2, 5, 10]}

dt = DecisionTreeClassifier()
dt_cv = GridSearchCV(dt, dt_param, cv=kf5, refit=True, verbose=3)
```

```
dt_cv.fit(X_train, y_train)
print(dt_cv.best_score_)
print(dt_cv.best_params_)
```

[ ]:
```
dt = DecisionTreeClassifier(max_depth = 9, min_samples_leaf = 1,␣
↪min_samples_split = 2)
cv = cross_val_score(dt, X_train, y_train, cv = kf5)
print(cv, 'mean: ', cv.mean())

dt.fit(X_train, y_train)
y_pred = dt.predict(X_test)
print(confusion_matrix(y_test, y_pred))
print(classification_report(y_test, y_pred))
print(dt.score(X_test, y_test))
```

4.0.9 LGB

[ ]:
```
kf5 = StratifiedKFold(n_splits = 5, shuffle = True)
lgb_param = {'max_depth': np.arange(2, 11),
             'n_estimators': np.arange(50, 201, 50),
             'lambda_l1': [0, 1, 1.5],
             'lambda_l2': [0, 1]}

LGB = lgb.LGBMClassifier()
lgb_cv = GridSearchCV(LGB, lgb_param, cv=kf5, refit=True, verbose=3)
lgb_cv.fit(X_train, y_train)
print(lgb_cv.best_score_)
print(lgb_cv.best_params_)
```

[ ]:
```
LGB = lgb.LGBMClassifier(lambda_l1=0, lambda_l2=1, max_depth=9,␣
↪n_estimators=200)
cv = cross_val_score(LGB, X_train, y_train, cv = kf5)
print(cv, 'mean: ', cv.mean())

LGB.fit(X_train, y_train)
y_pred = LGB.predict(X_test)
print(confusion_matrix(y_test, y_pred))
print(classification_report(y_test, y_pred))
print(LGB.score(X_test, y_test))
```

[ ]:
```
classifiers = [LogisticRegression(C = 10, penalty = 'l1', solver= 'liblinear'),
               KNeighborsClassifier(n_neighbors=7),
               SVC(C=1000, gamma=0.001, kernel='rbf'),
               DecisionTreeClassifier(max_depth=9, min_samples_leaf=1,␣
↪min_samples_split=2),
```

```
                RandomForestClassifier(max_depth=9, min_samples_leaf=2,␣
↪min_samples_split=5, n_estimators=200),
                GradientBoostingClassifier(learning_rate=0.1, max_depth=5,␣
↪min_samples_leaf=2,
                                           min_samples_split=2,␣
↪n_estimators=100),
                MLPClassifier(activation='logistic', hidden_layer_sizes=(100,␣
↪2), learning_rate='invscaling'),
                XGBClassifier(colsample_bytree=1.0, gamma=2, max_depth=3,␣
↪min_child_weight=1, subsample=0.8),
                lgb.LGBMClassifier(lambda_l1=0, lambda_l2=1, max_depth=9,␣
↪n_estimators=200)]
```

[ ]:
```
acc = []
pre = []
rec = []
f1 = []

for classifier in classifiers:
    classifier.fit(X_train, y_train)
    y_pred = classifier.predict(X_test)
    acc.append(accuracy_score(y_test, y_pred))
    pre.append(precision_score(y_test, y_pred, average='weighted'))
    rec.append(recall_score(y_test, y_pred, average='weighted'))
    f1.append(f1_score(y_test, y_pred, average='weighted'))
```

[ ]:
```
models = pd.DataFrame({'Model': ['Logistic Regression','KNN', 'SVM', 'Decision␣
↪Tree', 'Random Forest',
                                 'Gradient Boosting','Neural␣
↪Network','Xgboost', 'LGB'],
                       'Accuracy':acc,
                       'Precision':pre,
                       'Recall':rec,
                       'F1':f1})
models.sort_values(by='Accuracy', ascending=False)
```

5 Standardize (Use scaled_X_train)

5.0.1 Final

[ ]:
```
classifiers = [LogisticRegression(C=1, penalty='l2', solver='newton-cg'),
                KNeighborsClassifier(n_neighbors=9),
                SVC(C=100, gamma=0.1, kernel='rbf'),
                DecisionTreeClassifier(max_depth=9, min_samples_leaf=2,␣
↪min_samples_split=2),
```

```
                RandomForestClassifier(max_depth=10, min_samples_leaf=2,␣
↪min_samples_split=5, n_estimators=50),
                GradientBoostingClassifier(learning_rate=0.1, max_depth=5,␣
↪min_samples_leaf=4,
                                           min_samples_split=2,␣
↪n_estimators=100),
                MLPClassifier(activation='relu', hidden_layer_sizes=(100, 3),␣
↪learning_rate='constant'),
                XGBClassifier(colsample_bytree=0.8, gamma=1, max_depth=4,␣
↪min_child_weight=1, subsample=0.6),
                lgb.LGBMClassifier(lambda_l1=1, lambda_l2=1, max_depth=4,␣
↪n_estimators=150)]
```

[ ]:
```
acc = []
pre = []
rec = []
f1 = []

for classifier in classifiers:
    classifier.fit(scaled_X_train, y_train)
    y_pred = classifier.predict(scaled_X_test)
    acc.append(accuracy_score(y_test, y_pred))
    pre.append(precision_score(y_test, y_pred, average='weighted'))
    rec.append(recall_score(y_test, y_pred, average='weighted'))
    f1.append(f1_score(y_test, y_pred, average='weighted'))
```

[ ]:
```
models = pd.DataFrame({'Model': ['Logistic Regression','KNN', 'SVM', 'Decision␣
↪Tree', 'Random Forest',
                                 'Gradient Boosting','Neural␣
↪Network','Xgboost', 'LGB'],
                       'Accuracy':acc,
                       'Precision':pre,
                       'Recall':rec,
                       'F1':f1})
models.sort_values(by='Accuracy', ascending=False)
```

6 Version 2

[ ]:
```
from sklearn.preprocessing import StandardScaler

X = df.loc[:, ['S', 'K', 'tau', 'r']]
y = df.loc[:, 'BS']

scaler = StandardScaler()
scaled_X = scaler.fit_transform(X)
print(scaled_X.shape)
```

[ ]:
```
from sklearn.model_selection import cross_val_score,KFold ## for regression
from sklearn.model_selection import StratifiedKFold ## recommended for␣
 ↪classification

# importing the modules
import lightgbm as lgb
from xgboost import XGBClassifier
from sklearn.model_selection import cross_val_score, cross_validate,␣
 ↪GridSearchCV
from sklearn.linear_model import LogisticRegression
from sklearn.neighbors import KNeighborsClassifier
from sklearn.naive_bayes import GaussianNB, MultinomialNB, BernoulliNB
from sklearn.tree import DecisionTreeClassifier
from sklearn.svm import SVC
from sklearn.ensemble import RandomForestClassifier, GradientBoostingClassifier
from sklearn.metrics import accuracy_score, precision_score, recall_score,␣
 ↪f1_score, confusion_matrix, classification_report
from sklearn.utils import class_weight
from sklearn.neural_network import MLPClassifier
```

6.0.1 Logistic Regression

[ ]:
```
kf10 = StratifiedKFold(n_splits = 10, shuffle = True)
log_param = {'penalty' : ['l1','l2'],
             'C': np.logspace(-3,3,7),
             'solver'  : ['newton-cg', 'lbfgs', 'liblinear']}

log = LogisticRegression()
log_cv = GridSearchCV(log, log_param, cv = kf10, refit=True, verbose=3)
log_cv.fit(scaled_X, y)
print(log_cv.best_score_)
print(log_cv.best_params_)
```

6.0.2 Random Forest

[ ]:
```
kf10 = StratifiedKFold(n_splits = 10, shuffle = True)
rf_param = {'max_depth': np.arange(2, 11),
            'min_samples_leaf': [1, 2, 4],
            'min_samples_split': [2, 5, 10],
            'n_estimators': np.arange(50, 201, 50)}

param = {'alpha':np.arange(0,1,0.1)}

rf = RandomForestClassifier()
rf_cv = GridSearchCV(rf, rf_param, cv=kf10, refit=True, verbose=3)
rf_cv.fit(scaled_X, y)
print(rf_cv.best_score_)
print(rf_cv.best_params_)
```

6.0.3 Gradient Boosting

[ ]:
```
kf10 = StratifiedKFold(n_splits = 10, shuffle = True)
gb_param = {
    "learning_rate": [0.001, 0.01, 0.1],
    'max_depth': [2, 5, 10],
    'min_samples_leaf': [1, 2, 4],
    'min_samples_split': [2, 5, 10],
    "n_estimators":[10, 100]
    }

gb = GradientBoostingClassifier()
gb_cv = GridSearchCV(gb, gb_param, cv=kf10, refit=True, verbose=3)
gb_cv.fit(scaled_X, y)
print(gb_cv.best_score_)
print(gb_cv.best_params_)
```

6.0.4 SVM

[ ]:
```
kf10 = StratifiedKFold(n_splits = 10, shuffle = True)
svm_param = {'C': [0.1, 1, 10, 100, 1000],
             'gamma': [1, 0.1, 0.01, 0.001, 0.0001],
             'kernel': ['rbf']}

X = df.loc[:, ['S', 'K', 'tau', 'r']]
y = df.loc[:, 'BS']
svm = SVC()
svm_cv = GridSearchCV(svm, svm_param, cv=kf10, refit=True, verbose=3)
svm_cv.fit(scaled_X, y)
print(svm_cv.best_score_)
print(svm_cv.best_params_)
```

6.0.5 Xgboost

[ ]:
```
kf10 = StratifiedKFold(n_splits = 10, shuffle = True)
kb_param = {'min_child_weight': [1, 5, 10],
          'gamma': [0.5, 1, 1.5, 2, 5],
          'subsample': [0.6, 0.8, 1.0],
          'colsample_bytree': [0.6, 0.8, 1.0],
          'max_depth': [3, 4, 5]}


X = df.loc[:, ['S', 'K', 'tau', 'r']]
y = df.loc[:, 'BS']
kb = XGBClassifier()
kb_cv = GridSearchCV(kb, kb_param, cv=kf10, refit=True, verbose=3)
kb_cv.fit(scaled_X, y)
print(kb_cv.best_score_)
print(kb_cv.best_params_)
```

6.0.6 KNN

[ ]:
```
kf10 = StratifiedKFold(n_splits = 10, shuffle = True)
knn_param = {'n_neighbors': [3, 4, 5, 6, 7, 8, 9, 10, 11, 12]}

knn = KNeighborsClassifier()
knn_cv = GridSearchCV(knn, knn_param, cv=kf10, refit=True, verbose=3)
knn_cv.fit(scaled_X, y)
print(knn_cv.best_score_)
print(knn_cv.best_params_)
```

6.0.7 Neural Network

[ ]:
```
kf10 = StratifiedKFold(n_splits = 10, shuffle = True)
nn_param = {'learning_rate': ["constant", "invscaling", "adaptive"],
            'hidden_layer_sizes': [(100,1), (100,2), (100,3)],
            'activation': ["logistic", "relu", "Tanh"]}

nn = MLPClassifier()
nn_cv = GridSearchCV(nn, nn_param, cv=kf10, refit=True, verbose=3)
nn_cv.fit(scaled_X, y)
print(nn_cv.best_score_)
print(nn_cv.best_params_)
```

6.0.8 Decision Tree

[ ]:
```
kf10 = StratifiedKFold(n_splits = 10, shuffle = True)
dt_param = {'max_depth': np.arange(2, 11),
            'min_samples_leaf': [1, 2, 4],
            'min_samples_split': [2, 5, 10]}

dt = DecisionTreeClassifier()
dt_cv = GridSearchCV(dt, dt_param, cv=kf10, refit=True, verbose=3)
dt_cv.fit(scaled_X, y)
print(dt_cv.best_score_)
print(dt_cv.best_params_)
```

6.0.9 LGB

[ ]:
```
kf10 = StratifiedKFold(n_splits = 10, shuffle = True)
lgb_param = {'max_depth': np.arange(2, 11),
            'n_estimators': np.arange(50, 201, 50),
            'lambda_l1': [0, 1, 1.5],
            'lambda_l2': [0, 1]}

LGB = lgb.LGBMClassifier()
lgb_cv = GridSearchCV(LGB, lgb_param, cv=kf10, refit=True, verbose=3)
lgb_cv.fit(scaled_X, y)
print(lgb_cv.best_score_)
print(lgb_cv.best_params_)
```

7 Final model: XGBoost in version 2

[ ]:
```
test = pd.read_csv('option_test_wolabel.csv')
test.head()
```

[ ]:
```
X_test = scaler.transform(test)
X_test
```

[ ]:
```
kb = XGBClassifier(colsample_bytree = 0.8, gamma = 1.5, max_depth = 4,
                   min_child_weight = 1, subsample = 0.6)
kb.fit(scaled_X, y)
y_pred = kb.predict(X_test)
```

[ ]:
```
BS = pd.DataFrame(y_pred, columns = ['BS'])
BS.head()
```

[ ]:
```
BS.to_csv('BS_result.csv')
```