Predicting European Call Option Values on S&P 500: A Comparative Analysis of Statistical and Machine Learning Models

Saier Hu

Email: saier.hu@alumni.usc.edu

University of Southern California

Executive Summary

The project aims to predict European call option values using machine learning models. A few variables in Black-Scholoes (BS) formula are used as predictors in our models, including current option value (Value), current asset value (S), strike price of the option (K), annual interest rate (r), time to maturity (tau), and the prediction made by the Black-Scholes equation (BS) denoted as “under” or “over".

We build regression models to predict option values, and classification models to classify whether Black-Scholes equation prediction overestimated (“over”) or underestimated (“under”) the European call option value.

To select the best regression model, we compare various models based on the mean R-squared of 5-fold Cross-Validation, including linear regression, Decision Tree, Random Forest, KNN regression, Boosting (XGBoost), and SVM. For the classification model selection, we evaluate different models -including KNN, Logistic, LDA, Naive Bayes, Decision Tree, Random Forest, Boosting, and SVM- based on the mean classification error of Stratified 5-fold CV. We found out in terms of predicting European options values and classifying BS’s prediction (using 0 for “under” and 1 for “over”), Random Forest regressor (mean R-squared of 0.9959) and Boosting (mean classification error of 6.57%) classifier performed the best.

In conclusion, machine learning models can enhance investment decision-making by helping us detect whether traditional statistical methods (such as the Black-Scholes equation) overestimate or underestimate option prices. More importantly, machine learning models might offer more accurate predictions when traditional models don’t work well, given their high mean R-squared of 0.9959 in our prediction of European call option prices.

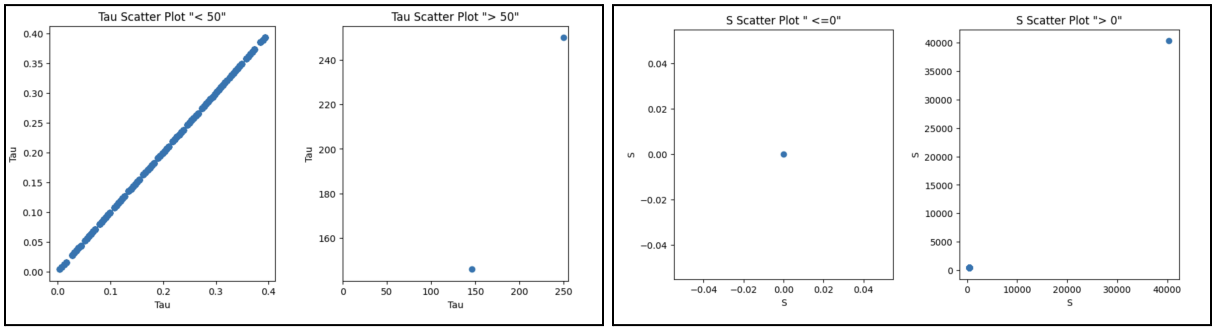
Introduction

This project aims to predict European option pricing using a training dataset of 1,680 options and a test dataset of 1,120 options. A regression model was built to predict option values, and R-Squared will be used to evaluate its performance. The project will also convert the problem into a binary classification task to categorize options as overestimated or underestimated by the Black-Scholes equation (using 0 for "under" and 1 for "over"). The findings can be applied to real-world options trading and risk management.

Approach

(1) Data Cleaning & Pre-processing

Data cleaning is a crucial step in the process. Our first step is to check and remove null values in the training dataset: we identify any missing or incomplete data points and remove them from the dataset to ensure that the data is accurate and ready for analysis (see *Appendix 1*). Another important step in data cleaning is to check and remove outliers in the predictors. In this particular dataset, the predictors where tau (time to maturity in years) > 50 years, S (strike price of option) <= 0, and S (strike price of option) > 40000 will be considered as outliers because they are implausible in real life and were removed from the dataset (see *Appendix 2*). As shown in the graphs below, these outliers are indeed singular existences in the training dataset.



After data cleaning, our team moves on to data pre-processing. In this dataset, we will change the 'BS' variable that indicates option price momentum from "over" and "under" to 1

and 0 respectively so that it can be used as the response variable in the classification problem (see *Appendix* 3). The predictor variables also need to be standardized and normalized so that they can be used to test if they will help certain models perform better (see *Appendix* 4). Additionally, we will create K-fold Cross-Validation to test the robustness of different models. For the regression problem, we will create a K-fold of K=5, while for the classification problem, we will create a Stratified K-fold of K=5 (see *Appendix* 5).

(2) Regression Model Selection

In the regression problem, we seek to predict the current option value (Value) based on 4 predictors: current asset value (S), strike price of the option (K), annual interest rate (r), and time to maturity (tau).

We then compared the robustness among different regression models based on the mean R-squared of 5-fold Cross-Validation (see *Appendix* 7), including K-Nearest Neighbors (KNN) regression, Decision Tree regression, Random Forest regression, Boosting regression, and Support Vector Machine (SVM) regression. The results of different regression models were then compared with each other, which showed that the Random Forest regression model's prediction yielded the highest mean R^2 (see *Appendix* 6). Notice that, (1) we tuned the parameters based on mean R-squared of 5-fold CV; (2) we used the normalized predictors to fit a linear regression model, which theoretically would boost its performance, but yielded the same mean R-squared of 5-fold CV as the linear regression model fitted on predictors that are not normalized.

The Random Forest regression model is determined to be the best (mean R-squared of 0.9959) then fitted on the entire training set and used to predict European call option prices in the test set, where the current option value (Value) is not available to us.

(3) Classification Model Selection

In the classification problem, we classify the predictions made by the Black-Scholes equation (using 0 for underestimate and 1 for overestimate) based on 4 predictors: current asset value (S), strike price of the option (K), annual interest rate (r), and time to maturity (tau).

Next, based on the Stratified 5-fold Cross-Validation (see *Appendix* 8), we compared the mean classification errors produced by different models, including K-Nearest Neighbor (KNN) classifier, logistic, linear discriminant analysis (LDA) classifier, Naive Bayes classifier, Random Forest classifier, Decision Tree classifier, Boosting classifier, and Support Vector Machine (SVM) classifier. Notice that, (1) we tuned the parameters based on mean classification errors of 5-fold CV; (2) we used the standardized predictors to fit a KNN, a Logistic, and a SVM classification model, which theoretically would boost their performance, but yielded the same or higher mean classification errors as the models fitted on predictors that are not normalized.

The Boosting classification model is determined to be the best (mean classification error of 6.57%) and fitted on the entire training set and used to BS's prediction (using 0 for "under" and 1 for "over") in the test set.

Conclusion

Machine learning models can enhance investment decision-making and risk management by helping us gauge whether traditional statistical methods (such as the Black-Scholes equation) overestimate or underestimate option prices. More importantly, machine learning models might offer more accurate predictions when traditional models don't work well, given their high mean R-squared of 0.9959 in our prediction of European call option prices.

Business Understandings

Valuing call options accurately is critical to making investment decisions as even slight errors can lead to significant financial losses or missed profits. In this project, *prediction accuracy is more important than interpretation.* The given predictors do not provide us with enough information to understand the underlying reasons behind the price of an option, especially so in this case where we are unable to identify the underlying security and its volatility.

Machine learning models may outperform the Black-Scholes model in predicting option values because they can discover complex relationships between predictors and option prices, and capture patterns that would not have been shown in a traditional financial model if certain variables were not selected. Additionally, ML-models can adapt to changing market conditions, whereas a formula like the black-scholes can stay only within its defined assumptions.

In this use case no variable selection is necessary. Given that there are only 4 variables, and each option has an unknown for its underlying security, omitting any of the four predictor variables used in the prediction model would result in a less accurate model.

Ultimately, *it would be irresponsible to apply our trained model to predict option values for Tesla stock* with the intention of making investment decisions. Firstly, our model may have a high accuracy performance, but the option data is not specific to Tesla's option; thus, we are missing key time-series data about Tesla's stock. Additionally, Tesla's most popular strike prices, and even their stock price is not represented well in our dataset (see *Appendix* 2).

Appendix

Appendix 1: Check and Remove Null Values

```
# need to drop NaNs
df_train.isnull().sum()
```

```
Value    2
S        1
K        2
tau      1
r        0
BS       0
dtype: int64
```

```
# drop NaNs
df_options= df_train.dropna()

df_options.isnull().sum()
```

```
Value    0
S        0
K        0
tau      0
r        0
BS       0
dtype: int64
```

Appendix 2: Check and Remove Outliers

```
df_train.describe()
```

	Value	S	K	tau	r
count	1678.000000	1679.000000	1678.000000	1679.000000	1680.000000
mean	15.068709	464.402535	438.241955	0.437519	0.030235
std	14.040023	973.652179	23.408989	7.057555	0.000557
min	0.125000	0.000000	375.000000	0.003968	0.029510
25%	2.255001	433.863864	420.000000	0.119048	0.029820
50%	11.190967	442.634081	440.000000	0.202381	0.030130
75%	25.747434	447.320414	455.000000	0.285714	0.030540
max	60.149367	40333.000000	500.000000	250.000000	0.031880

```
# drop observations where t > 50 years, S = 0, and S > 40000
df_options = df_options[df_options['tau'] <= 50]
df_options = df_options[df_options['S'] <= 40000]
df_options = df_options[df_options['S'] > 0]
```

Appendix 3:

```
# Replacing 'Over' and 'Under' in the training data BS variable with 1 and 0

BS_mapping = {'Under': 0, 'Over': 1}

df_train['BS'] = df_train['BS'].map(BS_mapping)

df_train.head()
```

	Value	S	K	tau	r	BS
0	21.670404	431.623898	420.0	0.341270	0.03013	0
1	0.125000	427.015526	465.0	0.166667	0.03126	1
2	20.691244	427.762336	415.0	0.265873	0.03116	0
3	1.035002	451.711658	460.0	0.063492	0.02972	1
4	39.553020	446.718974	410.0	0.166667	0.02962	0

Appendix 4: Normalization & Standardization

```
from sklearn.preprocessing import MinMaxScaler
mms = MinMaxScaler()

X_norm = mms.fit_transform(X_reshape)

X_correct_dimensions = X_norm.reshape(-1,4)
```

```
from sklearn.preprocessing import StandardScaler
stdsc = StandardScaler()
X_standardized = stdsc.fit_transform(X_reshape)

# restore X's original dimension so we can use cross_val_score() function
X_correct_dimensions = X_standardized.reshape(-1,4)
```

Note: The normalized predictors are used in the regression model selection problem, while the standardized predictors are used in the classification model selection problem. Thus, the 2 sets of predictors are used in different files and can use the same variable name (X_correct_dimensions)

Appendix 5: Creating K-fold and Stratified K-fold

```
# create k-fold where k =5
kfolds = KFold(n_splits = 5, random_state = 1, shuffle = True)
```

```
kfold = StratifiedKFold(n_splits = 5, random_state = 42, shuffle = True)
```

Note: These two variables are used in different files

Appendix 6: Regression Model Comparison

Model	5-fold Mean R2
Linear Regression	0.910376
KNN Regression	0.971128
Decision Tree Regression	0.990445
Random Forest Regression	0.995957
Boosting Regression	0.995827
SVM Regression	0.894469

Appendix 7: Regression Model Fitting

```
# Fit linear regression model: R-squared of 5-fold CV is 0.91
linear_regresssion_model = LinearRegression()

#linear_regresssion_model.fit(X, y)

r2_linear_regresssion_model_cv = cross_val_score(linear_regresssion_model, X, y, cv=kfolds, scoring = 'r2')

print("Linear Regression Model R-squared of 5-folds:", r2_linear_regresssion_model_cv ,"(mean r squared:", np. mean(r2_linear_regresssion_model_cv),

regression_result_df.loc[len(regression_result_df.index)] = ['Linear Regression', np. mean(r2_linear_regresssion_model_cv)]
```

```
Linear Regression Model R-squared of 5-folds: [0.91970852 0.91645431 0.90703987 0.89906318 0.9096128 ] (mean r squared: 0.9103757373884772 )
```

```
KNN_model = KNeighborsRegressor(n_neighbors=5)

#KNN_model.fit(X, y)

r2_KNN_regresssion_model_cv = cross_val_score(KNN_model, X, y, cv=kfolds, scoring = 'r2')

print("KNN Regression Model R-squared of 5-folds:", r2_KNN_regresssion_model_cv ,"(mean r squared:", np. mean(r2_KNN_regresssion_model_cv),")")

regression_result_df.loc[len(regression_result_df.index)] = ['KNN Regression', np. mean(r2_KNN_regresssion_model_cv)]
```

```
KNN Regression Model R-squared of 5-folds: [0.96758867 0.97089989 0.97249587 0.97199414 0.97266035] (mean r squared: 0.9711277864853083 )
```

```
Decision_Tree_Model = DecisionTreeRegressor(max_depth=25)

r2_Decision_Tree_Model_cv = cross_val_score(Decision_Tree_Model, X, y, cv=kfolds, scoring = 'r2')

print("Decision Tree Model R-squared of 5-folds:", r2_Decision_Tree_Model_cv ,"(mean r squared:", np. mean(r2_Decision_Tree_Model_cv),")")

regression_result_df.loc[len(regression_result_df.index)] = ['Decision Tree Regression', np. mean(r2_Decision_Tree_Model_cv)]
```

```
Decision Tree Model R-squared of 5-folds: [0.98928062 0.99029455 0.99069626 0.99064689 0.99130713] (mean r squared: 0.990445088763704 )
```

```
# The parameters are optimal according to mean R^2 of 5-fold CV
Random_Forest_Model = RandomForestRegressor(n_estimators=100, max_depth=30)

r2_Random_Forest_Model_cv = cross_val_score(Random_Forest_Model, X, y, cv=kfolds, scoring = 'r2')

print("Random Forest Model R-squared of 5-folds:", r2_Random_Forest_Model_cv ,"(mean r squared:", np. mean(r2_Random_Forest_Model_cv),")")

regression_result_df.loc[len(regression_result_df.index)] = ['Random Forest Regression', np. mean(r2_Random_Forest_Model_cv)]
```

```
Random Forest Model R-squared of 5-folds: [0.99436317 0.9958412  0.99723412 0.99602224 0.99632298] (mean r squared: 0.9959567423805149 )
```

```
# The parameters are optimal according to mean R^2 of 5-fold CV
Boosting_model = XGBRegressor(n_estimators=100, max_depth=10, eta=0.2, subsample=0.7, colsample_bytree=0.8)

r2_Boosting_Model_cv = cross_val_score(Boosting_model, X, y, cv=kfolds, scoring = 'r2')

print("Boosting Model R-squared of 5-folds:", r2_Boosting_Model_cv ,"(mean r squared:", np. mean(r2_Boosting_Model_cv),")")

regression_result_df.loc[len(regression_result_df.index)] = ['Boosting Regression', np. mean(r2_Boosting_Model_cv)]
```

```
Boosting Model R-squared of 5-folds: [0.99490032 0.9961635  0.99446725 0.99666447 0.99694179] (mean r squared: 0.9958274656810977 )
```

```
SVM_Model = SVR(kernel='linear', C=1.0, epsilon=0.1)

r2_SVM_Model_cv = cross_val_score(SVM_Model, X, y, cv=kfolds, scoring = 'r2')

print("SVM Model R-squared of 5-folds:", r2_SVM_Model_cv ,"(mean r squared:", np. mean(r2_SVM_Model_cv),")")

regression_result_df.loc[len(regression_result_df.index)] = ['SVM Regression', np. mean(r2_SVM_Model_cv)]
```

```
SVM Model R-squared of 5-folds: [0.90913961 0.90697327 0.8910591  0.87030715 0.89486515] (mean r squared: 0.8944688556306705 )
```

Appendix 8: Classification Model Comparison

	Classifier	Error %
0	KNN	9.62
1	Logistic	9.21
2	LDA	8.73
3	Naive Bayes	11.96
4	Random Forest	6.63
5	Decision Tree	9.20
6	SVM	9.15
7	XGB	6.57

Appendix 9: Classification Model Fitting

```
#KNN

knn = KNeighborsClassifier()

accuracy = cross_val_score(knn, X, y, cv=kfold, scoring = 'accuracy')

error_list = [1 - x for x in accuracy]

print('KNN Classifier Error List for KFolds : ', error_list,
      '\n\nMean Classification Error : ', round(np.mean(error_list),4)
)

result_df.loc[len(result_df.index)] = ['KNN', ((round(np.mean(error_list),4))) * 100]
```

```
KNN Classifier Error List for KFolds :  [0.09850746268656718, 0.10149253731343288, 0.11044776119402988, 0.08383233532934131, 0.08682634730538918]

Mean Classification Error :  0.0962
```

```
#Logistic

logreg = LogisticRegression()


accuracy = cross_val_score(logreg, X, y, cv=kfold)


error_list = [1 - x for x in accuracy]

print('Logistic Classifier Error List for KFolds : ', error_list,
      '\n\nMean Classification Error : ', round(np.mean(error_list),4)
)

result_df.loc[len(result_df.index)] = ['Logistic', ((round(np.mean(error_list),4))) * 100]
```

```
Logistic Classifier Error List for KFolds :  [0.09850746268656718, 0.08656716417910448, 0.08955223880597019, 0.09281437125748504, 0.09281437125748504]

Mean Classification Error :  0.0921
```

```
#LDA

lda = LinearDiscriminantAnalysis()

accuracy = cross_val_score(lda, X, y, cv=kfold)


error_list = [1 - x for x in accuracy]

print('LDA Classifier Error List for KFolds : ', error_list,
      '\n\nMean Classification Error : ', round(np.mean(error_list),4)
)

result_df.loc[len(result_df.index)] = ['LDA', ((round(np.mean(error_list),4))) * 100]
```

```
LDA Classifier Error List for KFolds :  [0.10149253731343288, 0.07164179104477608, 0.09253731343283578, 0.09580838323353291, 0.07485029940119758]

Mean Classification Error :  0.0873
```

```
#Naive Bayes

nb = GaussianNB()

accuracy = cross_val_score(nb, X, y, cv=kfold)


error_list = [1 - x for x in accuracy]

print('Naive Bayes Classifier Error List for KFolds : ', error_list,
      '\n\nMean Classification Error : ', round(np.mean(error_list),4)
)

result_df.loc[len(result_df.index)] = ['Naive Bayes', ((round(np.mean(error_list),4))) * 100]
```

```
Naive Bayes Classifier Error List for KFolds :  [0.09850746268656718, 0.11641791044776117, 0.12238805970149258, 0.12574850299401197, 0.1347305389221557]

Mean Classification Error :  0.1196
```

```
#Random Forest                          (parameter) max_depth: Any | None
rf = RandomForestClassifier(n_estimators=100, max_depth=20)

accuracy = cross_val_score(rf, X, y, cv=kfold)


error_list = [1 - x for x in accuracy]

print('Random Forest Classifier Error List for KFolds : ', error_list,
      '\n\nMean Classification Error : ', round(np.mean(error_list),4)
)

result_df.loc[len(result_df.index)] = ['Random Forest', ((round(np.mean(error_list),4))) * 100]
```

```
Random Forest Classifier Error List for KFolds :  [0.06567164179104479, 0.047761194029850795, 0.11044776119402988, 0.06586826347305386, 0.05688622754491013]

Mean Classification Error :  0.0693
```

```
#Decison Tree

dt = DecisionTreeClassifier(max_depth = 25)

accuracy = cross_val_score(dt, X, y, cv=kfold)


error_list = [1 - x for x in accuracy]

print('Decision Tree Classifier Error List for KFolds : ', error_list,
      '\n\nMean Classification Error : ', round(np.mean(error_list),4)
)

result_df.loc[len(result_df.index)] = ['Decision Tree', ((round(np.mean(error_list),4))) * 100]
```

```
Decision Tree Classifier Error List for KFolds :  [0.08358208955223878, 0.10447761194029848, 0.11044776119402988, 0.09281437125748504, 0.08083832335329344]

Mean Classification Error :  0.0944
```

```
#SVM
clf = svm.SVC(kernel='linear', C=1, random_state=42)


accuracy = cross_val_score(clf, X, y, cv=kfold)


error_list = [1 - x for x in accuracy]

print('SVM Classifier Error List for KFolds : ', error_list,
      '\n\nMean Classification Error : ', round(np.mean(error_list),4)
)

result_df.loc[len(result_df.index)] = ['SVM', ((round(np.mean(error_list),4))) * 100]
```

```
SVM Classifier Error List for KFolds :  [0.09850746268656718, 0.08656716417910448, 0.08358208955223878, 0.09880239520958078, 0.08982035928143717]

Mean Classification Error :  0.0915
```

```
#XGBoost

xgb_clf = xgb.XGBClassifier(objective='binary:logistic', random_state=42)

accuracy = cross_val_score(xgb_clf, X, y, cv=kfold)


error_list = [1 - x for x in accuracy]

print('XGBoost Classifier Error List for KFolds : ', error_list,
      '\n\nMean Classification Error : ', round(np.mean(error_list),4)
)

result_df.loc[len(result_df.index)] = ['XGB', ((round(np.mean(error_list),4))) * 100]
```

```
XGBoost Classifier Error List for KFolds :  [0.07164179104477608, 0.05671641791044779, 0.08358208955223878, 0.06586826347305386, 0.050898203592814384]

Mean Classification Error :  0.0657
```