Options Pricing Prediction

Executive Summary

Project Overview

Our project aims to utilize machine-learning techniques to predict the option value and the accuracy of its estimates using the European call option pricing data on the S&P 500. We divided our data into training and testing sets (80/20 split), employing cross-validation for optimal hyperparameter selection. Preprocessing techniques included standardization and outlier handling, enhancing our models' relevance to financial datasets. Following this, we trained various regression and classification models and selected the best-performing techniques based on cross-validation R-squared and MSE to make our final predictions.

Regression and Classification Analysis

Our regression analysis began with Linear Regression and progressed to more complex models like K-nearest neighbors (KNN), Support Vector Machine (SVM), and XGBoost, focusing on minimizing the mean squared error (MSE). The XGBoost model demonstrated superior predictive accuracy with a cross-validated R-squared of 0.998. For classification predictions, we used three models and four packages, the models are Random Forest, SVM, and gradient boosting machine. Among these, LightGBM achieved the lowest classification error of 0.056 on the dataset with unstandardized data.

Our Model vs. Black-Scholes

Our top-performing classification model - LightGBM - shows that machine learning can outperform traditional methods like Black-Scholes in financial predictions. These models adapt well to market complexities, providing accurate predictions crucial for strategic decision-making.

Conclusion

Our models demonstrate the potential to enhance competitive business outcomes through improved predictive accuracy. The accuracy of predictions, vital for strategic decision-making and influencing business outcomes, emphasizes the need for complex models that incorporate a wide range of features to reflect real-world complexities. However, it is crucial to apply these models with caution, as overconfidence in their predictions can lead to significant risks in business operations. Machine learning models offer superior options value prediction compared to Black-Scholes due to their adaptability to dynamic markets, emphasizing the significance of prediction accuracy while urging caution in their application. For our scenario, we are confident that it makes business sense to keep all four predictors in our model as they are important in explaining our dependent variable and our models achieved high R-squared scores and low classification errors. Finally, despite the great performance of our models, it is important to be aware of the limitations of models and use caution when applying them to different datasets. We would advise to use our experience and knowledge gained from this project to train new models on new dataset to ensure accuracy and relevancy of our predictions.

Final Report

Data Preprocessing

To begin our analysis, we conducted some basic EDA to gain an understanding of the data we are working with. Our data did not have any missing values. According to the Black-Scholes variable (BS_Binary), most options are classified as underestimated. Current asset value (S), strike price (K), and annual interest rate (r) follow a normal distribution while time to maturity in years (tau) and option value (Value) are slightly left-skewed (Appendix 1.1). The correlation matrix showed that K exhibits a strong negative correlation (-0.88) with Value, while S shows a remarkably strong negative correlation (-0.98) with r. Additionally, K moderately correlates (0.54) with Black-Scholes model predictions (Appendix 1.2). The box plot showed that there are some outliers present in the Value and K variables (Appendix 1.3).

To ensure optimal comparison across different models and to prevent overfitting, we divided our data into an 80/20 split for the training and testing sets, respectively. We trained our models on the training subset and used the testing subset to obtain a performance estimate of our models before fitting them to the designated test data.

For our regression models, we standardized the data using the StandardScaler. This approach ensures that all features within our dataset contribute equally to the model's performance, which should prevent any single feature with a larger scale from disproportionately influencing the model. In contrast, for our classification models, we chose not to scale the data. This decision was based on observations that standardization did not improve and sometimes even worsened the performance of our classification models. Tree-based classification algorithms inherently handle features with varying scales effectively because they split nodes based on feature order rather than feature magnitude. Thus, preserving the original scale of the data helps these models utilize the inherent distance and ordering of the data more effectively.

For our classification analysis, further refinements include feature engineering, where we introduce a new feature called "S/K," calculated by dividing S by K. We also apply log transformations to the highly skewed predictors such as tau and “S/K” to normalize their distribution.

To address the outliers, we considered winsorization - removing outliers based on interquartile range, and trimming outliers based on standard deviation. We fit our models on different versions of our dataset in terms of outliers and concluded that keeping the outliers in the dataset contributes to creating better models that are more applicable to real-life scenarios. It is very common for financial variables like S and K to have extreme values which implies that removing outliers would introduce an issue of setting unrealistic assumptions for our models.

In our classification analysis, we ended up working with six different versions of our data set: unstandardized data, standardized data, unstandardized data with engineered features, standardized data with engineered features, unstandardized data with log-transformed and engineered features, and standardized and log-transformed and engineered features. In regression analysis, we only considered standardized data with and without outliers.

Hyperparameter Tuning

Regression

For the regression analysis part of the project, we employed the GridSearchCV algorithm, a systematic method of tuning hyperparameters to identify the optimal values for our models (Appendix 2). Essentially, GridSearchCV searches through a specified subset of hyperparameters and evaluates each combination based on a predefined score, such as negative means squared error. This method helps identify the most effective hyperparameters that contribute to the best predictive performance of the models.

For the Linear Regression model, no specific hyperparameters were tuned since the model complexity depends largely on the data itself. However, for more complex models, such as KNN Regression, Support Vector Machine (SVM), and Gradient Boosting Models, we utilized GridSearchCV to find their optimal settings.

For the KNN, the optimal number of neighbors (*n_neighbors*) found was specific for each dataset, influencing how the model generalizes to new data (Appendix 4.2.1). The parameter '*n_neighbors*' controls the count of nearest points considered when making predictions, where a lower number can capture finer details but may lead to overfitting. After performing the GridSearchCV, we have found that the most optimal number of k-neighbors is four.

For our SVM model, parameters *'C'*, *'epsilon'*, and *'kernel'* were tuned. *'C'* is the penalty of the error term that controls the trade-off between achieving a low error on the training data and minimizing the model complexity for better generalization (Appendix 4.3.1). *'Epsilon'* specifies the epsilon-tube within which no penalty is associated with predictions deviating from the actual observations, thus controlling the width of the margin of tolerance where predictions are considered acceptable. The '*kernel*' parameter determines the type of hyperplane used to separate the data. The best parameters for our SVM appeared to be the following: *'C'*: 1000, *'epsilon'*: 1, *'kernel'*: 'rbf'.

Finally, for our XGBoost model, parameters such as *n_estimators*, *learning_rate*, *max_depth*, *subsample*, and *colsample_bytree* were optimized (Appendix 4.4.1). *N_estimators* controls the number of weak learners to be built, affecting both accuracy and overfitting risk. *Learning_rate* shrinks the contribution of each tree, where a lower rate requires more trees but can lead to better model performance. *Max_depth* sets the maximum depth of the trees, controlling the complexity of the model. The *subsample* parameter specifies the fraction of the training data to be used for each tree, helping to prevent overfitting by adding more randomness into the model building. *Colsample_bytree* affects how many features are used for each tree, introducing feature randomness and further control overfitting. The optimal parameters we found to be the following: *n_estimators*=300, *max_depth*=5, *learning_rate*=0.2, *subsample*=0.8, and *colsample_bytree*=0.8.

Classification

For classification in analysis, we implemented StratifiedKFold with five folds to maintain representative proportions of each class across all folds (Appendix 3). During grid search, we evaluated model performance using both AUC-ROC and accuracy as scoring metrics. Ultimately, we focused on accuracy for interpretative purposes after experimental results showed comparable performance across both metrics.

For the Random Forest, we adjusted the number of trees - *n_estimators* (Appendix 5.1.1). While a larger number of trees generally improves performance on test data, it can also lead to overfitting.

In the case of SVM, we tuned several parameters: the regularization parameter *(C)*, which helps control overfitting; the kernel type *(kernel)* , which determines the transformation of the data into a higher dimensional space for linear separation; and gamma *(gamma)*, which defines the influence of individual training samples on the decision boundary (Appendix 5.2.1).

For XGBoost, we fine-tuned the maximum depth of each tree *(max_depth)*, the step size of each learning iteration *(learning_rate)*, and the number of boosted trees *(n_estimators)* to optimize performance (Appendix 5.3.1).

Similarly, for LightGBM, we adjusted the maximum depth of the trees *(max_depth)*, the learning rate *(learning_rate)*, and the number of boosted trees *(n_estimators)*. Additionally, we tuned *min_split_gain*, which is the minimum loss reduction required to make an additional partition on a leaf node of the tree, helping to control over-complex models (Appendix 5.4.1).

Review of approaches

Regression Results

For regression analysis, we start from linear regression as the bassline, then move on to more sophisticated models like K-nearest neighbors (KNN) regression, Support Vector Machine (SVM) for regression, and Gradient Boosting with XGBoost. We are comparing our models based on the cross-validation R-squared. The Linear Regression model achieved an R-squared value of 0.974 (Appendix 4.1.2). For the KNN and SVM Regressions, we achieved an R-squared of 0.997 (Appendix 4.2.3 & 4.3.3). Finally, for the XGBoost, we got a cross-validated R-squared of 0.998, demonstrating excellent predictive accuracy (Appendix 4.4.3).

Classification Results

For our classification tasks, we utilized three modeling techniques: Random Forest, Support Vector Machine (SVM), and Gradient Boosting. Here, we compare our models based on the classification error. In our analysis, Random Forest recorded an estimated error rate of 0.066 (Appendix 5.1.2), while SVM had an error rate of 0.086 (Appendix 5.2.2). XGBoost demonstrated a slightly better performance with an estimated error of 0.061 (Appendix 5.3.2). LightGBM, our top-performing model, achieved the lowest error rate of 0.056 (Appendix 5.4.2).

Summary of Final Approaches

Regression Analysis

Our top-performing model is the XGBoost model with an R-squared of 0.998. Prior to making predictions on the target sample, we fit the model on the entire dataset to ensure robustness and accuracy. We chose this model as our winning model and used it for our final predictions (Appendix 6.1) because it has the highest cross-validation R-squared and MSE. In addition, XGBoost is a widely used model among practitioners who work with finance data.

Classification Analysis

Our best-performing model is the Gradient boosting with LightGBM package using unstandardized data. We also tested another LightGBM model which demonstrated almost identical performance on unscaled data that included engineered parameters and log-transformed features. This suggests that for Gradient Boosting Machines, engineered features and log transformations may be redundant and should be used cautiously. This model has been selected for our final classification predictions (Appendix 6.2).

Conclusion

In conclusion, the models we propose may outperform Black-Schole in options value prediction. While the Black-Scholes model is praised for its simplicity, it is constrained by its assumption of log-normal stock price distributions. On the other hand, machine learning models can adapt and learn from the intricacies within data and non-linear relationships, making them more robust and reliable under dynamic market conditions.

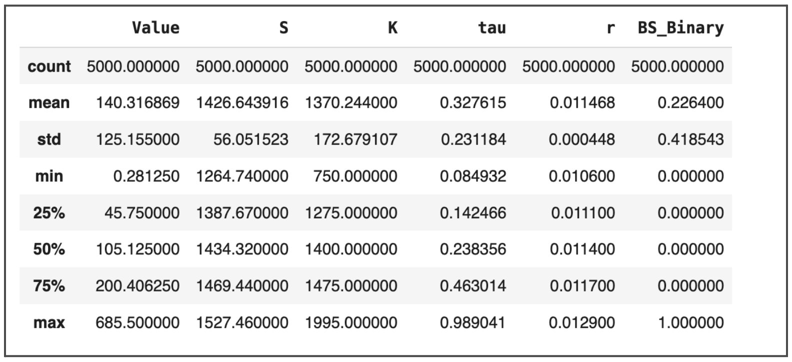
The primacy of prediction accuracy in a business environment cannot be overstated. Despite the inherent value of model interpretability, the ultimate measure of a model's worth resides in the accuracy of its predictions. Because of the importance of accuracy it is often counterproductive to diminish the complexity of models by removing variables.

In terms of variable selection, it is always important to consider a method of best model selection to ensure that the final model includes only useful predictors, especially when the number of predictors is high. In our case, we only have four predictors that are not highly correlated and explain different important parts of option pricing. Additionally, all of our models achieved high R-square scores. Therefore, we can argue that all four predictor variables should be included in our prediction.

Finally, the application of any model must be tempered with prudence. Overconfidence in model predictions can lead to significant miscalculations and introduce substantial risks to business operations. It is imperative that models are used with an awareness of their limitations. This becomes especially important when considering using a model that is trained and tested on one type of data on a different type of data. Considering this, we would not be comfortable to directly use our model to predict option values for Tesla stocks.

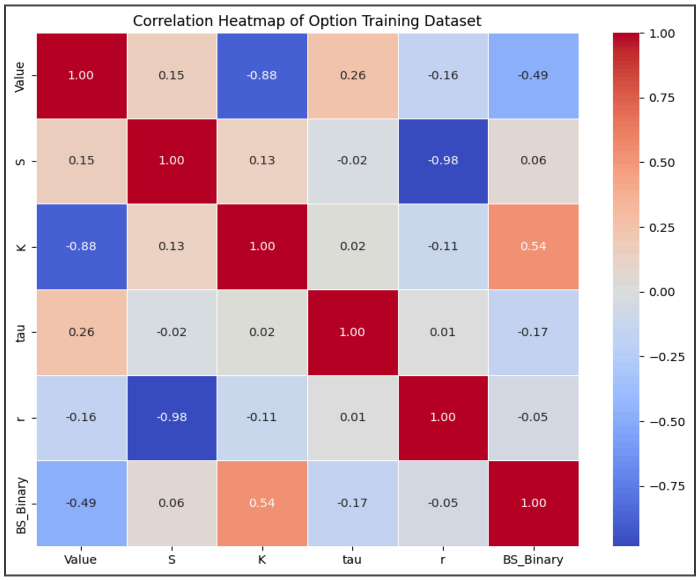
Appendix

1. Descriptive Statistics

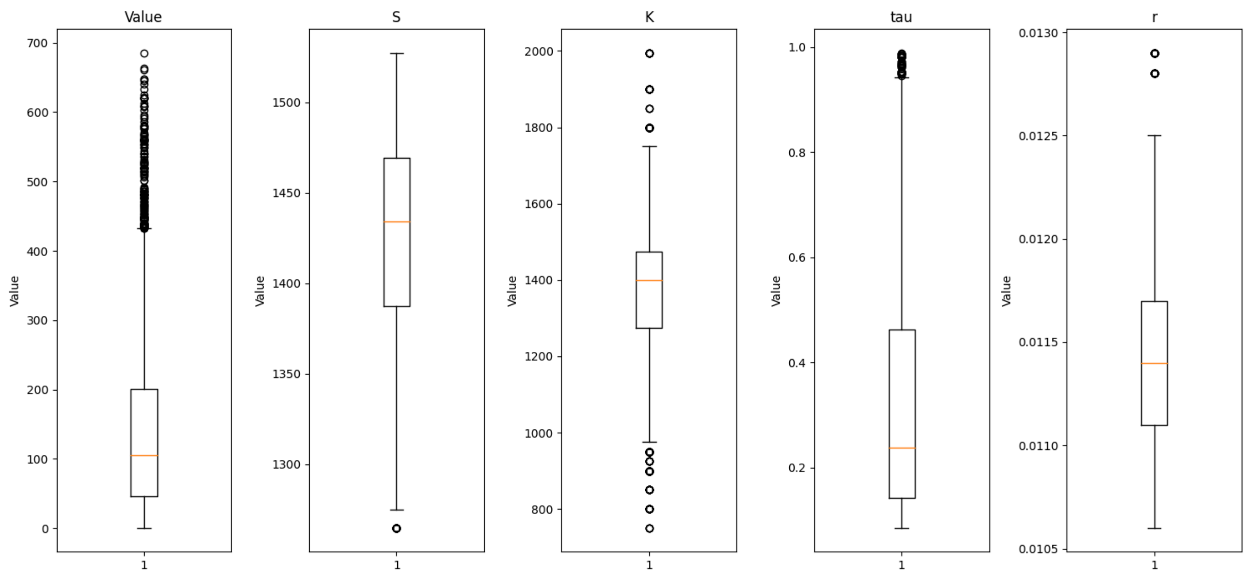


	Value	S	K	tau	r	BS_Binary
count	5000.000000	5000.000000	5000.000000	5000.000000	5000.000000	5000.000000
mean	140.316869	1426.643916	1370.244000	0.327615	0.011468	0.226400
std	125.155000	56.051523	172.679107	0.231184	0.000448	0.418543
min	0.281250	1264.740000	750.000000	0.084932	0.010600	0.000000
25%	45.750000	1387.670000	1275.000000	0.142466	0.011100	0.000000
50%	105.125000	1434.320000	1400.000000	0.238356	0.011400	0.000000
75%	200.406250	1469.440000	1475.000000	0.463014	0.011700	0.000000
max	685.500000	1527.460000	1995.000000	0.989041	0.012900	1.000000

1.1 Descriptive Statistics



1.2 Correlation Matrix



1.3 Boxplots

2. Hyperparameter tuning for regression

Model	**Parameter grid**
Linear Regression	NA
K Nearest Neighbors	'n_neighbors' : range(1,21)
Support Vector Machine	'C': [1, 10, 100, 1000], 'epsilon': [0.001, 0.01, 0.1, 1], 'kernel': ['linear', 'poly', 'rbf']
Gradient Boosting Machine (XBGoost)	'n_estimators': [100, 200, 300], 'learning_rate': [0.01, 0.1, 0.2], 'max_depth': [3, 4, 5], 'min_samples_split': [2, 4] 'min_samples_leaf': [1, 2]

3. Hyperparameter tuning for classification

Model	**Parameter grid**
Random Forest	'n_estimators': [10, 25, 50, 75, 100, 150, 200]
Support Vector Machine	'C': [0.1, 1, 10, 100], 'gamma': [1, 0.1, 0.01, 0.001], 'kernel': ['rbf', 'linear', 'poly']
Gradient Boosting Machine (XBGoost)	'max_depth': [3, 4, 5], 'learning_rate': [0.1, 0.01, 0.05], 'n_estimators': [100, 200, 300]
Gradient Boosting Machine (LightGBM)	'num_leaves': [31, 50, 100], 'learning_rate': [0.1, 0.01, 0.001], 'n_estimators': [100, 200, 300], 'min_split_gain': [0.0, 0.1, 0.01]

4. Code for Regression Models

```
# Linear Regression Analysis for the original data
from sklearn.linear_model import LinearRegression
from sklearn.metrics import mean_squared_error

lr_model_original = LinearRegression()

#fitting the model
lr_model_original.fit(X_train_original_scaled, y_train_original)

#predicting on the test set
y_pred_original = lr_model_original.predict(X_test_original_scaled)

#mse
mse_original = mean_squared_error(y_test_original,y_pred_original)
mse_original
```

4.1.1 Fitting the Linear Regression

```
from sklearn.model_selection import cross_val_score

# Perform 10-fold cross-validation to evaluate MSE
cv_mse_scores_orig = cross_val_score(lr_model_original,
X_train_original_scaled, y_train_original, cv=10,
scoring='neg_mean_squared_error')
mean_cv_mse_orig = -cv_mse_scores_orig.mean()
std_cv_mse_orig = cv_mse_scores_orig.std()

# Perform 10-fold cross-validation to evaluate R^2
cv_r2_scores_orig = cross_val_score(lr_model_original,
X_train_original_scaled, y_train_original, cv=10, scoring='r2')
mean_cv_r2_orig = cv_r2_scores_orig.mean()
std_cv_r2_orig = cv_r2_scores_orig.std()

# Print the cross-validation results
print("Original Dataset - Linear Regression - Cross-Validated MSE:",
mean_cv_mse_orig, "±", std_cv_mse_orig)
print("Original Dataset - Linear Regression - Cross-Validated R^2:",
mean_cv_r2_orig, "±", std_cv_r2_orig)
```

4.1.2 Cross-validation for Linear Regression

```
from sklearn.neighbors import KNeighborsRegressor
from sklearn.metrics import mean_squared_error
from sklearn.model_selection import GridSearchCV

#using the GridSearch to find the optimal K (potential range 1
through 20)
param_grid = {'n_neighbors' : range(1,21)}

#using 5 fold CV to find the best K (best K turned out to be 4)
knn = KNeighborsRegressor()
grid_search_orig = GridSearchCV(knn, param_grid, cv = 5, scoring
='neg_mean_squared_error')
grid_search_orig.fit(X_train_original_scaled, y_train_original)

best_k_orig = grid_search_orig.best_params_['n_neighbors']
print(f"Best K for Original Dataset: {best_k_orig}")
```

4.2.1 Hyperparameter tuning for KNN regression

```
from sklearn.metrics import r2_score

#fitting and evaluating the KNN Regression
knn_best_orig = KNeighborsRegressor(n_neighbors=best_k_orig)
knn_best_orig.fit(X_train_original_scaled, y_train_original)

predictions_orig = knn_best_orig.predict(X_test_original_scaled)
mse_orig = mean_squared_error(y_test_original, predictions_orig)
r2_orig = r2_score(y_test_original, predictions_orig)

print(f"MSE for Original Dataset: {mse_orig}")
print(f"R^2 for Original Dataset: {r2_orig}")
```

4.2.2 Fitting the KNN Regression

```
from sklearn.model_selection import cross_val_score


knn_best_orig = KNeighborsRegressor(n_neighbors=best_k_orig)


cv_mse_scores_orig = cross_val_score(knn_best_orig,
X_train_original_scaled, y_train_original, cv=10,
scoring='neg_mean_squared_error')
mean_cv_mse_orig = -cv_mse_scores_orig.mean()
std_cv_mse_orig = cv_mse_scores_orig.std()

# Perform 10-fold cross-validation to evaluate R^2
cv_r2_scores_orig = cross_val_score(knn_best_orig,
X_train_original_scaled, y_train_original, cv=10, scoring='r2')
mean_cv_r2_orig = cv_r2_scores_orig.mean()
std_cv_r2_orig = cv_r2_scores_orig.std()

# Print the results
print("Original Dataset - KNN Regression - Cross-Validated MSE:",
mean_cv_mse_orig, "±", std_cv_mse_orig)
print("Original Dataset - KNN Regression - Cross-Validated R^2:",
mean_cv_r2_orig, "±", std_cv_r2_orig)
```

4.2.3 Cross-validation for KNN Regression

```
from sklearn.svm import SVR
from sklearn.model_selection import GridSearchCV
from sklearn.metrics import mean_squared_error, r2_score

param_grid = {'C': [1, 10, 100, 1000],
   'epsilon': [0.001, 0.01, 0.1, 1],
   'kernel': ['linear', 'poly', 'rbf']}


#original dataset grid search
grid_search_orig =  GridSearchCV(SVR(), param_grid, cv = 5,
scoring='neg_mean_squared_error', verbose=2)

grid_search_orig.fit(X_train_original_scaled, y_train_original)

best_params_orig = grid_search_orig.best_params_
print(f"Best parameters for Original Dataset: {best_params_orig}")
```

4.3.1 Hyperparameter tuning for SVM Regression

```
#fitting the SVM regression

svr_best_orig = SVR(**best_params_orig)
svr_best_orig.fit(X_train_original_scaled, y_train_original)
predictions_orig_svr = svr_best_orig.predict(X_test_original_scaled)

#evaluating
mse_orig_svr = mean_squared_error(y_test_original,
predictions_orig_svr)
r2_orig_svr = r2_score(y_test_original, predictions_orig_svr)

print(f"Original Dataset SVR - MSE: {mse_orig_svr}, R^2:
{r2_orig_svr}")
```

4.3.2 Fitting the SVM Regression

```
from sklearn.model_selection import cross_val_score

# Cross-validate the best SVR model on the original dataset
cv_mse_scores_orig = cross_val_score(svr_best_orig,
X_train_original_scaled, y_train_original, cv=10,
scoring='neg_mean_squared_error')
mean_cv_mse_orig = -cv_mse_scores_orig.mean()
std_cv_mse_orig = cv_mse_scores_orig.std()
cv_r2_scores_orig = cross_val_score(svr_best_orig,
X_train_original_scaled, y_train_original, cv=10, scoring='r2')
mean_cv_r2_orig = cv_r2_scores_orig.mean()
std_cv_r2_orig = cv_r2_scores_orig.std()

print("Original Dataset - Cross-Validated MSE:", mean_cv_mse_orig,
"±", std_cv_mse_orig)
print("Original Dataset - Cross-Validated R^2:", mean_cv_r2_orig,
"±", std_cv_r2_orig)
```

4.3.3 Cross-validation for SVM Regression

```
import xgboost as xgb
from sklearn.model_selection import GridSearchCV
from sklearn.metrics import mean_squared_error, r2_score

param_grid = {
   'n_estimators': [100, 200, 300],
   'max_depth': [3, 4, 5, 6],
   'learning_rate': [0.01, 0.1, 0.2],
   'subsample': [0.7, 0.8, 0.9],
   'colsample_bytree': [0.7, 0.8, 0.9]}
xgb_reg = xgb.XGBRegressor(objective ='reg:squarederror')
grid_search = GridSearchCV(estimator=xgb_reg, param_grid=param_grid,
                          scoring='neg_mean_squared_error', cv=5,
verbose=1)

grid_search.fit(X_train_original_scaled, y_train_original)
best_params = grid_search.best_params_
print("Best parameters found: ", best_params)
```

4.4.1 Hyperparameter tuning for XGBoost

```
# fitting the XGBoost model
xgb_best = xgb.XGBRegressor(**best_params, objective
='reg:squarederror')

xgb_best.fit(X_train_original_scaled, y_train_original)
y_pred_xgb = xgb_best.predict(X_test_original_scaled)

mse_xgb = mean_squared_error(y_test_original, y_pred_xgb)
r2_xgb = r2_score(y_test_original, y_pred_xgb)

print(f"Test MSE: {mse_xgb}")
print(f"Test R^2: {r2_xgb}")
```

4.4.2 Fitting the XGBoost Model

```
# Performin 10-fold cross-validation for MSE
cv_mse_scores = cross_val_score(xgb_best, X_train_original_scaled,
y_train_original,
                               cv=10,
scoring='neg_mean_squared_error')
mean_cv_mse = -cv_mse_scores.mean()
std_cv_mse = cv_mse_scores.std()

# Perform 10-fold cross-validation for R^2
cv_r2_scores = cross_val_score(xgb_best, X_train_original_scaled,
y_train_original,
                              cv=10, scoring='r2')
mean_cv_r2 = cv_r2_scores.mean()
std_cv_r2 = cv_r2_scores.std()

print(f"Cross-Validated MSE: {mean_cv_mse:.2f} ± {std_cv_mse:.2f}")
print(f"Cross-Validated R^2: {mean_cv_r2:.2f} ± {std_cv_r2:.2f}")
```

4.4.3 Cross-validation for XGBoost Model

5. Code for Classification Models

```
#find best n_estimator for new features data
param_grid = {
   'n_estimators': [10, 25, 50, 75, 100, 150, 200]
}

# Initialize the grid search
grid_search_3 =
GridSearchCV(RandomForestClassifier(random_state=42), param_grid,
cv=stratified_kfold, scoring='accuracy')

# Perform the grid search
grid_search_3.fit(X_cat_train, Y_cat_train)

# Best number of trees
best_n_estimators = grid_search_3.best_params_['n_estimators']
print(f"Best number of estimators: {best_n_estimators}")
```

5.1.1 Hyperparameter tuning for Random Forest

```
random_forest = RandomForestClassifier(n_estimators=150,
random_state=42)

# Train the model on the training data
random_forest.fit(X_cat_train, Y_cat_train)

# Make predictions on the test data
predictions = random_forest.predict(X_cat_test)

# Evaluate the model
Prediction_error = 1-accuracy_score(Y_cat_test, predictions)

print(f"Model error rate: {Prediction_error * 100:.4f}%")
```

5.1.2 Fitting Model and Estimation Error for Random Forest

```
param_grid = {
   'C': [0.1, 1, 10, 100],
   'gamma': [1, 0.1, 0.01, 0.001],
   'kernel': ['rbf', 'linear', 'poly']
}

# Create GridSearchCV object
grid = GridSearchCV(SVC(), param_grid, refit=True, verbose=2,
cv=stratified_kfold)

# Fit the model
grid.fit(X_cat_train_scaled, Y_cat_train)

# Print best parameters and scores
print(f"Best parameters: {grid.best_params_}")
print(f"Best cross-validation score: {grid.best_score_}")
```

5.2.1 Hyperparameter tuning for SVM

```
svm_classifier = SVC(kernel='rbf', C=100, gamma=1)

# Train the model using the training sets
svm_classifier.fit(X_cat_train_scaled, Y_cat_train)

# Predict the response for the test dataset
y_pred = svm_classifier.predict(X_cat_test_scaled)

# Evaluate the model
print("Error:", 1-accuracy_score(Y_cat_test, y_pred))
print("\nClassification Report:")
print(classification_report(Y_cat_test, y_pred))
```

5.2.2 Model fitting for SVM

```
param_grid = {
   'max_depth': [3, 4, 5],
   'learning_rate': [0.1, 0.01, 0.05],
   'n_estimators': [100, 200, 300]
}

# Initialize the XGBClassifier
xgb_clf = xgb.XGBClassifier(use_label_encoder=False,
eval_metric='mlogloss')

# Initialize GridSearchCV
grid_search = GridSearchCV(estimator=xgb_clf, param_grid=param_grid,
cv=stratified_kfold, verbose=2, scoring='accuracy')

# Fit GridSearchCV
grid_search.fit(X_cat_train, Y_cat_train)

# Print best parameters and scores
print(f"Best parameters: {grid_search.best_params_}")
```

5.3.1 Hyperparameter Tuning for XGBoost

```
best_estimator = xgb.XGBClassifier(learning_rate=0.1, max_depth=5,
n_estimators=300)
best_estimator.fit(X_cat_train, Y_cat_train)

# Predictions on the test set
y_pred = best_estimator.predict(X_cat_test)

# Compute the accuracy of the model
accuracy = accuracy_score(Y_cat_test, y_pred)
print(f"Error: {1-(accuracy):.6f}")
```

5.3.2 Model Fitting for XGBoost

```
param_grid = {
   'num_leaves': [31, 50, 100],
   'learning_rate': [0.1, 0.01, 0.001],
   'n_estimators': [100, 200, 300],
   'min_split_gain': [0.0, 0.1, 0.01]
}

# Set up the grid search
grid_search = GridSearchCV(estimator=model, param_grid=param_grid,
cv=stratified_kfold, scoring='accuracy', verbose=1)

# Perform the grid search on the training data
grid_search.fit(X_cat_train_scaled, Y_cat_train)
print(f"Best parameters: {grid_search.best_params_}")
```

5.4.1 Hyperparameter Tuning for LightGBM

```
model = lgb.LGBMClassifier(**best_params)

model.fit(X_cat_train_scaled, Y_cat_train)

# Make predictions on the test set
y_pred = model.predict(X_cat_test_scaled)

# Calculate the accuracy
accuracy = accuracy_score(Y_cat_test, y_pred)

# Print the best parameters and test set accuracy
print(f"Error: {1- accuracy:.8f}")
```

5.4.2 Model Fitting for LightGBM

6. Final Predictions

```
# Initialize the XGBoost regressor with the best parameters
best_params = {'colsample_bytree': 0.8, 'learning_rate': 0.2,
'max_depth': 5, 'n_estimators': 300, 'subsample': 0.8}
xgb_best = xgb.XGBRegressor(**best_params,
objective='reg:squarederror')

xgb_best.fit(X_train_original_scaled, y_train_original)

X_test_scaled = scaler.transform(option_test)

y_pred_test = xgb_best.predict(X_test_scaled)
submission = pd.DataFrame({
   'Value': y_pred_test
})

# Print and save the submission
print(submission.head())
```

6.1 Final Prediction For Regression

```
best_params = {
   'learning_rate': 0.1,
   'min_split_gain': 0.01,
   'n_estimators': 300,
   'num_leaves': 50
}
model = lgb.LGBMClassifier(**best_params)

model.fit(X_full, Y_full)

# Make predictions on the test set
y_pred = model.predict(option_test[['S','K','tau','r']])

print(y_pred)
```

6.2 Final Prediction for Classification