

# Empirical Comparison of Option Pricing Models

Andrea Donazzan, Amadeus Lars Linge

*IEDA 4520*

# Content

- Monte Carlo Pricing Models
- Pricing with Machine Learning
- Data Collection
- Calibration
- Results

# Pricing Models

European Options:

- **Black Scholes** Model (GMB)
- Merton's **Jump Diffusion** (GMB with Poisson)

American Options:

- **Cox-Ross-Rubinstein** Model (Binomial Tree)
- **Longstaff-Schwartz** Model (Least Squares Monte Carlo)

## Black Scholes

$$S(t_{i+1}) = S(t_i) \exp \left( \left( \mu - \frac{\sigma}{2} \right) \Delta t + \sigma \sqrt{\Delta t} Z_{i+1} \right)$$

## Jump Diffusion

$$S(t_{i+1}) = S(t_i) \exp \left( \left( \mu - \frac{\sigma}{2} \right) \Delta t + \sigma \sqrt{\Delta t} Z_{i+1} + a N_{i+1} + b \sqrt{N_{i+1}} Z'_{i+1} \right)$$

with  $N_i \sim \text{Pois}(\lambda \Delta t)$

# American Options

Allow for **early exercise**

Value at each time step: maximum between early exercise value and continuation value.

For an **American call** option:

$$O_t^i = \max \left( \underbrace{(S_t^i - K)^+}_{\text{early exercise}}, \underbrace{e^{-(T-t)r} (E[S_T | S_t^i] - K)^+}_{\text{continuation value}} \right)$$

# Binomial Trees 1: Simulate Stock Value

Stock price can:

- Move up to  $uS_0$  with probability  $q$
- Move down to  $dS_0$  with probability  $(1 - q)$

$$u = e^{\sigma\sqrt{\Delta t}} \quad d = e^{-\sigma\sqrt{\Delta t}} = \frac{1}{u} \quad q = \frac{e^{r\Delta t} - d}{u - d}$$

Fill the best outcome nodes with  $S_{i,0} = uS_{i-1,0}$  and the others with  $S_{i,j} = dS_{i-1,j-1}$

	0	1	2	3	4
0	1.000	0.000	0.000	0.000	0.000
1	1.221	0.819	0.000	0.000	0.000
2	1.492	1.000	0.670	0.000	0.000
3	1.822	1.221	0.819	0.549	0.000
4	2.226	1.492	1.000	0.670	0.449

# Binomial Trees 2: Compute Option Price

Start from end of tree and move upwards:

At maturity, compute value as  $\max(S_{T,j} - K, 0)$

For each previous node compute

$$O_{i,j} = \max \left( e^{-r\Delta t} (qO_{(i+1),j} + (1-q)O_{(i+1),(j+1)}), S_{i,j} - K \right)$$

	0	1	2	3	4
0	0.302	0.000	0.0	0.0	0.0
1	0.427	0.104	0.0	0.0	0.0
2	0.598	0.162	0.0	0.0	0.0
3	0.827	0.252	0.0	0.0	0.0
4	1.126	0.392	0.0	0.0	0.0

## Why do we need the Longstaff-Schwartz algorithm?

- How can we evaluate the **continuation value**  $E[S_T | S_t^i]$ ?

Nested Monte Carlo simulations → **unfeasible** for large numbers

- *Binomial Tree*: **discretization** error if used with long time steps.

Will **underestimate** the number of **early exercise opportunities** as it only provides two outcomes for the value of the underlying.

*Time complexity:  $O(2^n)$*



## LSMC explanation: an example

We take as example an **American call** option with 1 year maturity, exercisable at times 1,2,3:

$$S_0 = 1, K = 1.1, r = 0.1, \sigma = 0.2$$

	<b>t = 0</b>	<b>t = 1</b>	<b>t = 2</b>	<b>t = 3</b>
<b>0</b>	1.0	1.15	0.92	1.05
<b>1</b>	1.0	1.20	1.19	1.33
<b>2</b>	1.0	0.96	0.97	1.04
<b>3</b>	1.0	0.96	1.19	1.46
<b>4</b>	1.0	1.04	1.02	0.99
<b>5</b>	1.0	1.07	1.02	1.08
<b>6</b>	1.0	1.13	1.18	1.36
<b>7</b>	1.0	1.08	1.18	1.19

## Step 1

Setup **cash flow matrix**:

Determine expected payoff at **maturity**.

Since continuation value is zero = payoff of a vanilla European option

	0	1	2	3
0	0.0	0.0	0.0	0.00
1	0.0	0.0	0.0	0.23
2	0.0	0.0	0.0	0.00
3	0.0	0.0	0.0	0.36
4	0.0	0.0	0.0	0.00
5	0.0	0.0	0.0	0.00
6	0.0	0.0	0.0	0.26
7	0.0	0.0	0.0	0.09

## Step 2

- One time step back: consider the paths where the option is **in the money** at  $t = T - 1$ .
- Discount the future **cash flow** of holding the option:

$$y_{t=2,i} = e^{-r} \pi_{t=3,i}$$

- Get value of underlying at time T-1

	Disc val	S_2
<b>0</b>	0.23	1.19
<b>1</b>	0.35	1.19
<b>2</b>	0.25	1.18
<b>3</b>	0.09	1.18

## Step 3

- Regress  $y_{t=2}$  on a set of basis functions of  $S_{t=2}$  to obtain the **continuation value**

$$\hat{C}_{t,i} = \sum_{j=0}^n a_{t,j} B_j(S_{t,i})$$

The parameters  $a_t$  are obtained minimizing

$$\frac{1}{I} \sum_{i=0}^I \left( y_{t,i} - \hat{C}_{t,i} \right)^2$$

```
X = np.column_stack([np.ones(M), S[:,i], S[:,i]**2, S[:,i]**3, S[:,i]**4, S[:,i]**5])
beta = np.linalg.lstsq(cond_x, Y, rcond=None)[0]
continue_val = np.dot(X, beta)
```

## Step 4

If  $S_{t,i} > \hat{C}_{t,i}$ , fill cash flow matrix with resulting cash flow from this path.

Repeat until  $t = 0$ .

	Continue val	P_2
<b>0</b>	0.227	0.086
<b>1</b>	0.348	0.087
<b>2</b>	0.251	0.075
<b>3</b>	0.090	0.076

# Pricing with Machine Learning

# Splitting the data

4 sets of data in total for training:

- European Options: calls/puts
- American Options: calls/puts

```
from sklearn.model_selection import train_test_split
```

```
X_train_calls, X_test_calls, y_train_calls, y_test_calls = train_test_split(X_calls, y_calls, test_size=0.25)  
X_train_puts, X_test_puts, y_train_puts, y_test_puts = train_test_split(X_puts, y_puts, test_size=0.25)
```

```
Train calls: (27187, 5)
```

```
Test calls: (9063, 5)
```

```
Train puts: (18194, 5)
```

```
Test puts: (6065, 5)
```

# Models

DTR – Decision Tree Regressor  
XGBr – Xtreme Gradient Booster

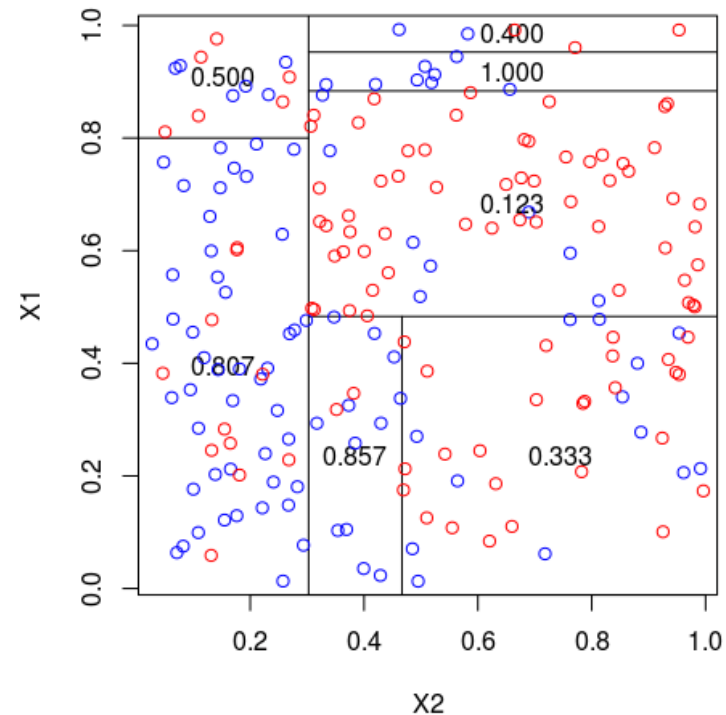
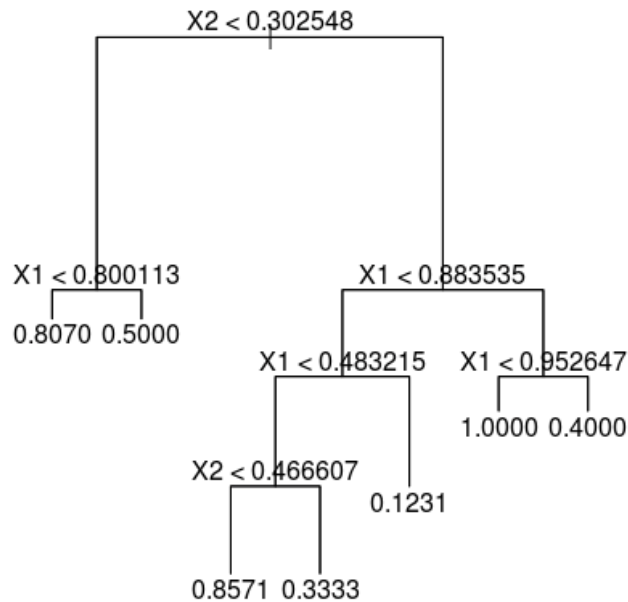
## Supervised Learning Models

$(S_0, K, T, \sigma, r) \rightarrow \text{option price}$



# DecisionTreeRegressor

```
DTR_calls = DecisionTreeRegressor(max_depth=24, min_samples_leaf=1)
DTR_calls.fit(X_train_calls, y_train_calls)
DTR1_pred = DTR_calls.predict(X_test_calls)
DTR1_rmse = np.sqrt(mean_squared_error(y_test_calls, DTR1_pred))
```

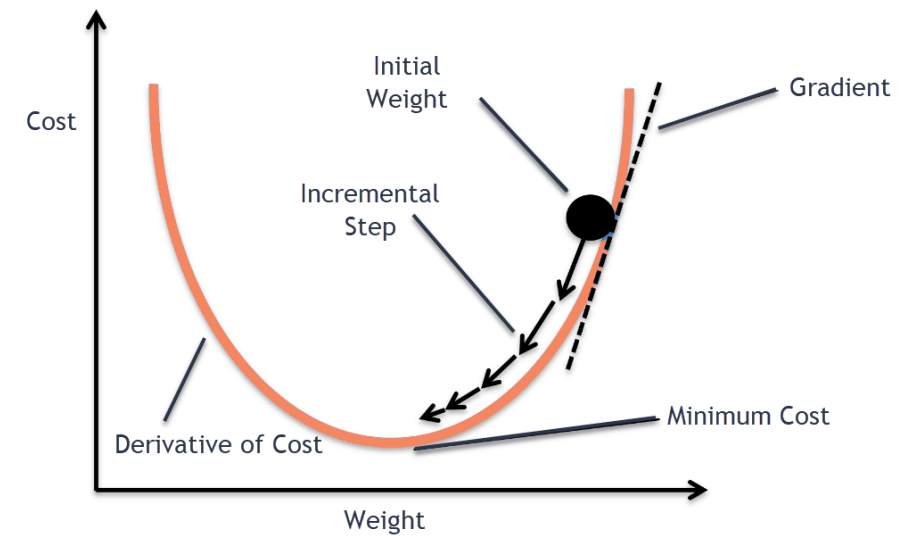


# XGBr

```
XGBr = xg.XGBRegressor(learning_rate=0.1, gamma= 0.001,  
max_depth= 5, min_child_weight= 6,  
subsample= 1, n_estimators=900)
```

```
XGBr.fit(X_train_calls, y_train_calls)  
XGBr_pred = XGBr.predict(X_test_calls)
```

- Advanced version of GBM
- Ensemble of decision trees

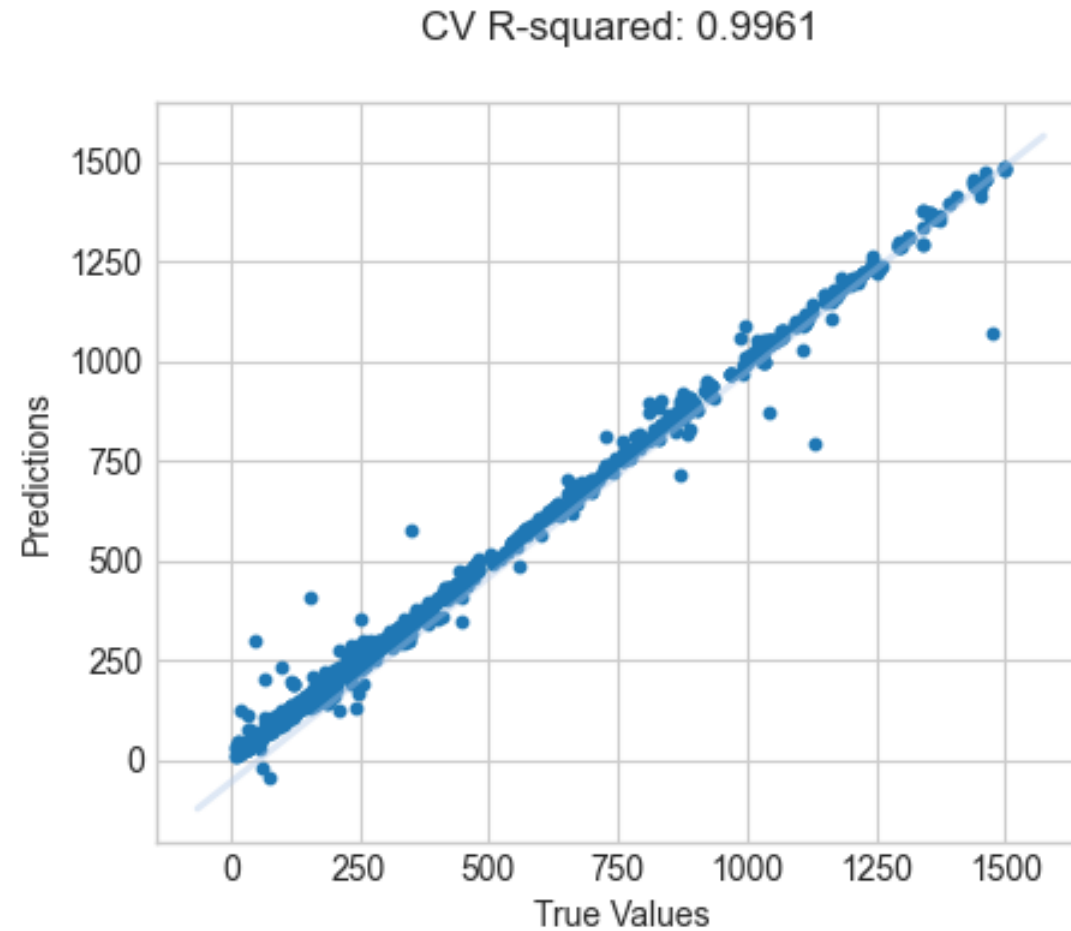


- Handles missing data
- Uses L1 and L2 regularization (Lasso- and Ridge-regularization)
- Tree pruning
- Parallelized

# Pipeline

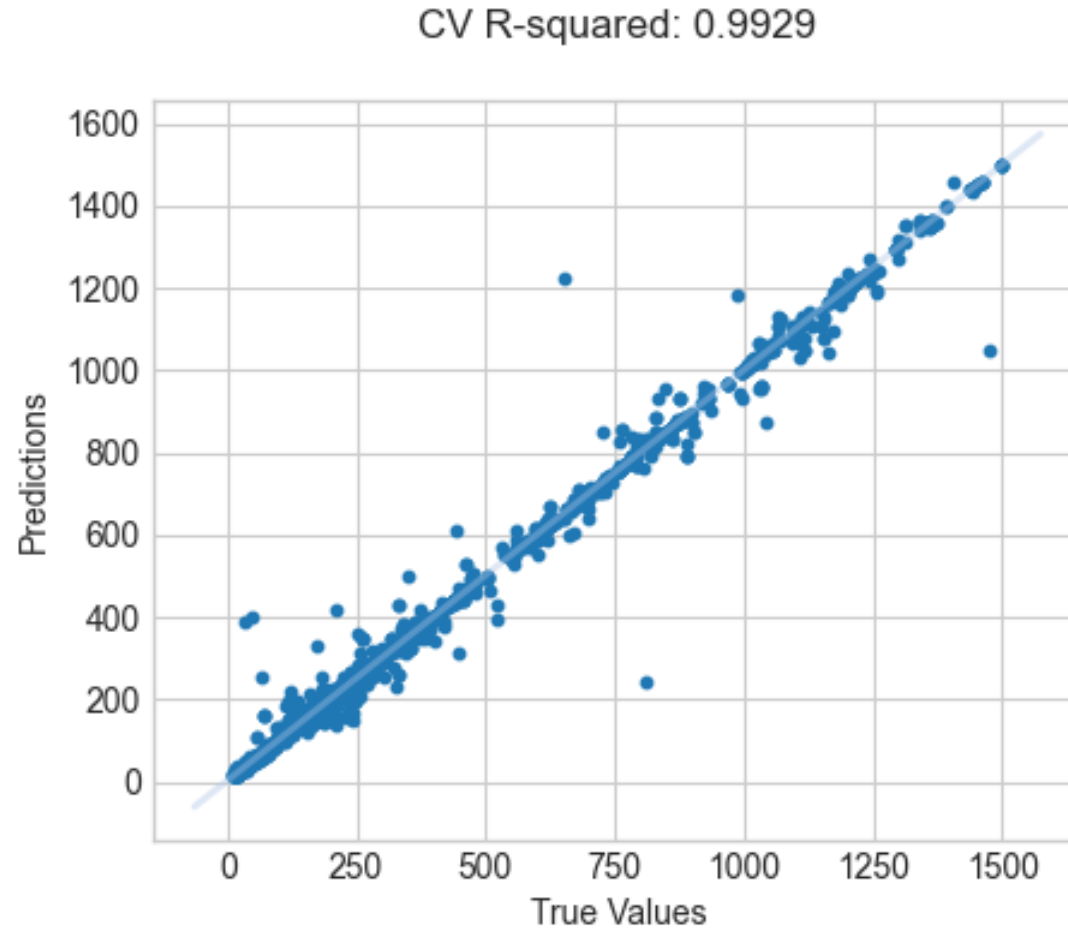
```
def model(pipeline, parameters, X_train, y_train, X, y, figname):  
  
    grid_obj = GridSearchCV(estimator = pipeline, param_grid = parameters, cv = 5,  
                           scoring = 'r2', verbose = 0, n_jobs = 1, refit = True)  
  
    grid_obj.fit(X_train, y_train)  
  
    print("Best Param:", grid_obj.best_params_)  
    estimator = grid_obj.best_estimator_  
    shuffle = KFold(n_splits = 5, shuffle = True, random_state = 0)  
    cv_scores = cross_val_score(estimator, X, y.ravel(), cv=shuffle, scoring='r2')  
  
    y_pred = cross_val_predict(estimator, X, y, cv = shuffle)
```

# XGBr optimal parameters



Best Param: {'xgb\_\_colsample\_bytree': 1, 'xgb\_\_gamma': 0.01, 'xgb\_\_max\_depth': 5, 'xgb\_\_min\_child\_weight': 3, 'xgb\_\_subsample': 0.6}

# DecisionTreeRegressor optimal parameters



Best Param: `{'dt__max_depth': 16, 'dt__min_samples_leaf': 1}`

# Data Collection

Goal: build database with market price of options and information about underlying

Stock data: `yfinance`      Option data: `yahooquery`

```
def BS(S0, K, T, sigma, r, type):
```

## Option Chain:

			contractSymbol	strike	currency	lastPrice	change	percentChange	volume	openInterest	bid	ask	contractSize	lastTradeDate	impliedVolatility
symbol	expiration	optionType													
AAPL	2023-11-24	calls	AAPL231124C00050000	50.0	USD	141.35	0.0	0.0	6.0	0	0.0	0.0	REGULAR	2023-11-20 20:54:50	0.000010
		calls	AAPL231124C00075000	75.0	USD	100.21	0.0	0.0	1.0	0	0.0	0.0	REGULAR	2023-11-03 18:07:22	0.000010
		calls	AAPL231124C00090000	90.0	USD	81.35	0.0	0.0	1.0	0	0.0	0.0	REGULAR	2023-11-01 14:00:00	0.000010
		calls	AAPL231124C00095000	95.0	USD	77.10	0.0	0.0	0.0	0	0.0	0.0	REGULAR	2023-10-25 14:10:07	0.000010
		calls	AAPL231124C00100000	100.0	USD	85.83	0.0	0.0	7.0	0	0.0	0.0	REGULAR	2023-11-10 19:30:52	0.000010
	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...
	2026-01-16	puts	AAPL260116P00250000	250.0	USD	75.00	0.0	0.0	0.0	0	69.3	73.5	REGULAR	2023-09-13 18:50:54	0.286048

# Modifying Option Chain

Add price of underlying, maturity, variability, mean returns.

```
import Data
Data.GetData('2018,11,22', '2022,11,22', 252, False)
```

	symbol	optionType	expiration	strike	lastPrice	lastTradeDate	inTheMoney	maturity	S0	sigma	returns	method
0	AAPL	puts	2024-06-21	280.0	143.10	2022-11-09	True	590	134.120331	0.346374	0.297289	A
1	AMZN	calls	2024-01-19	1040.0	1499.75	2022-06-03	False	595	122.349998	0.343390	0.138612	A
2	AMZN	calls	2024-01-19	1800.0	882.51	2022-06-03	False	595	122.349998	0.343390	0.138612	A
3	AMZN	calls	2024-01-19	2000.0	757.85	2022-06-03	False	595	122.349998	0.343390	0.138612	A
4	AMZN	calls	2024-01-19	2150.0	653.86	2022-06-03	False	595	122.349998	0.343390	0.138612	A
...	...	...	...	...	...	...	...	...	...	...	...	...
1448	^SPX	puts	2026-12-18	5900.0	1579.20	2022-07-25	True	1607	3966.840088	0.230712	0.112161	E
1449	^SPX	puts	2026-12-18	7000.0	2527.00	2022-10-13	True	1527	3669.909912	0.231177	0.085586	E
1450	^SPX	puts	2026-12-18	7200.0	2605.90	2022-09-29	True	1541	3640.469971	0.230415	0.084374	E
1451	^SPX	puts	2026-12-18	8800.0	4088.31	2022-06-23	True	1639	3795.729980	0.231362	0.102437	E
1452	^SPX	puts	2026-12-18	9000.0	4261.30	2022-06-23	True	1639	3795.729980	0.231362	0.102437	E



# Jump Diffusion Calibration

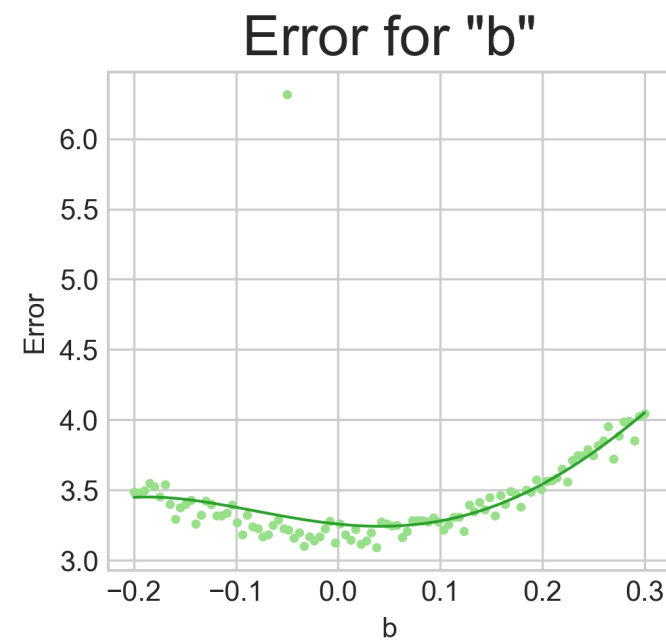
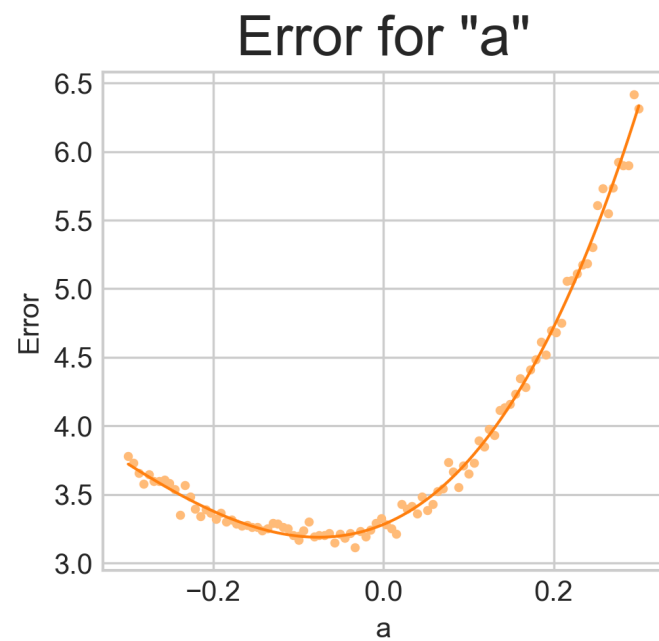
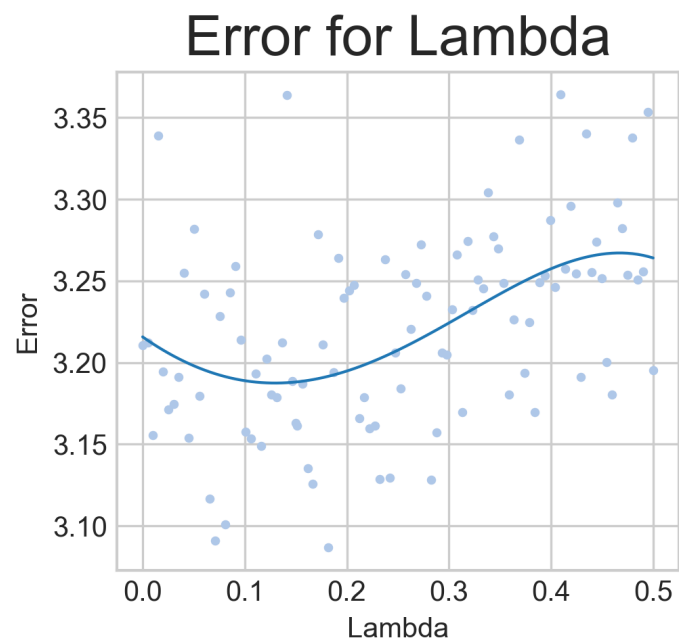
Parameters:

- $\lambda \rightarrow$  rate of Poisson process
- $a$  and  $b \rightarrow$  size of the jump

Test different values for each parameter holding the other two constant

Compute mean squared error of estimated prices

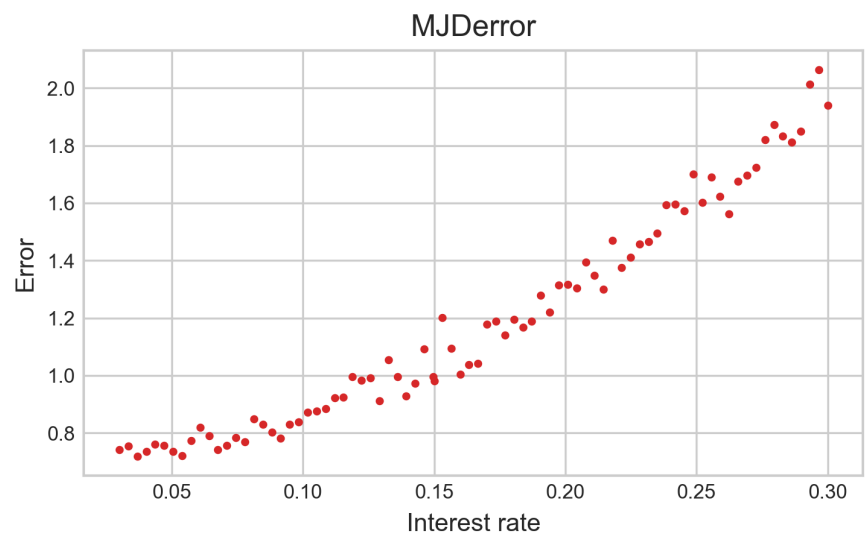
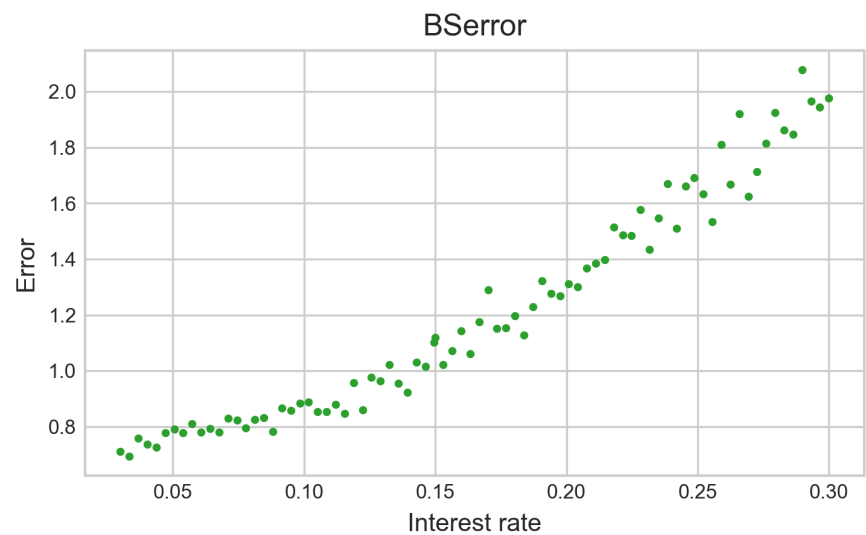
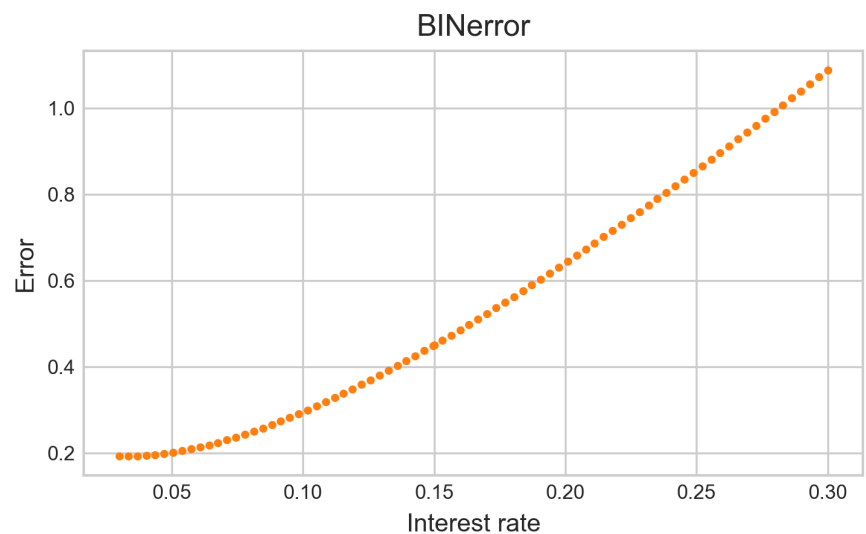
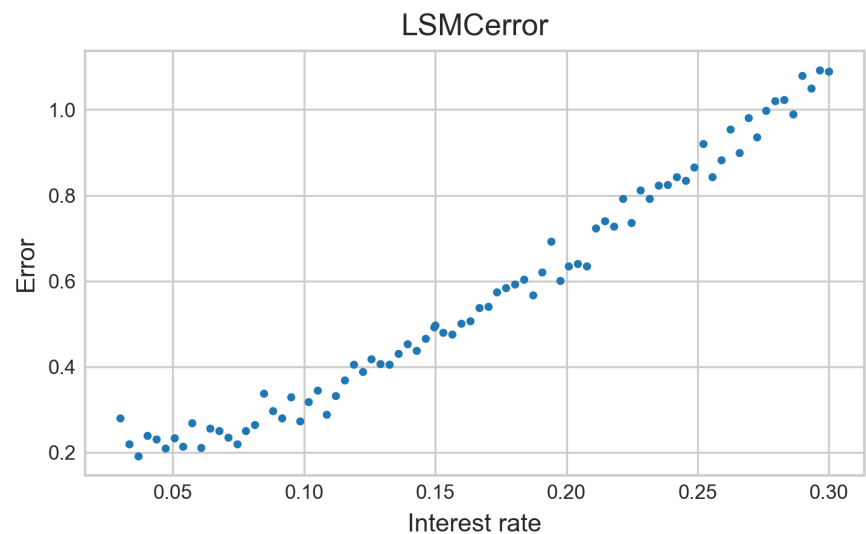
```
df = df.sample(n)
lamb_values = np.linspace(0,0.4, iterations)
for i in range(iterations):
    lamb = lamb_values[i]
    errors.append(compute_errors(lamb, a, b))
```



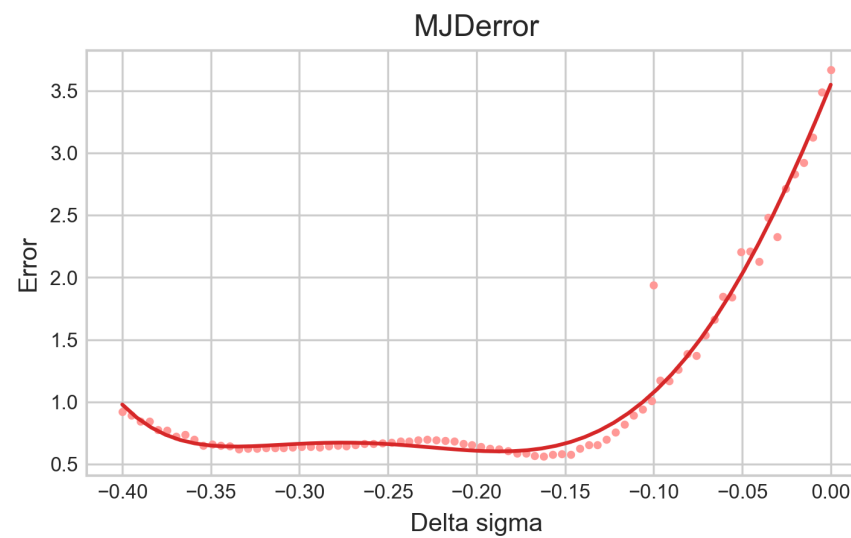
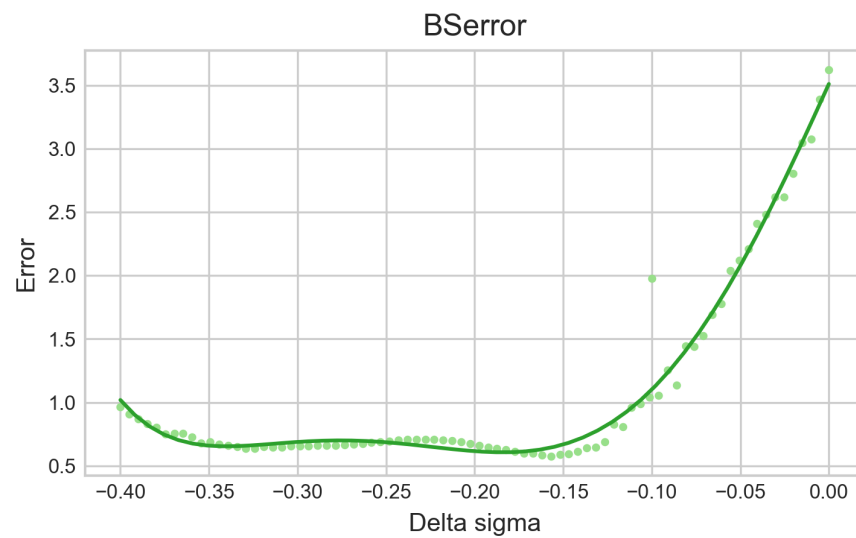
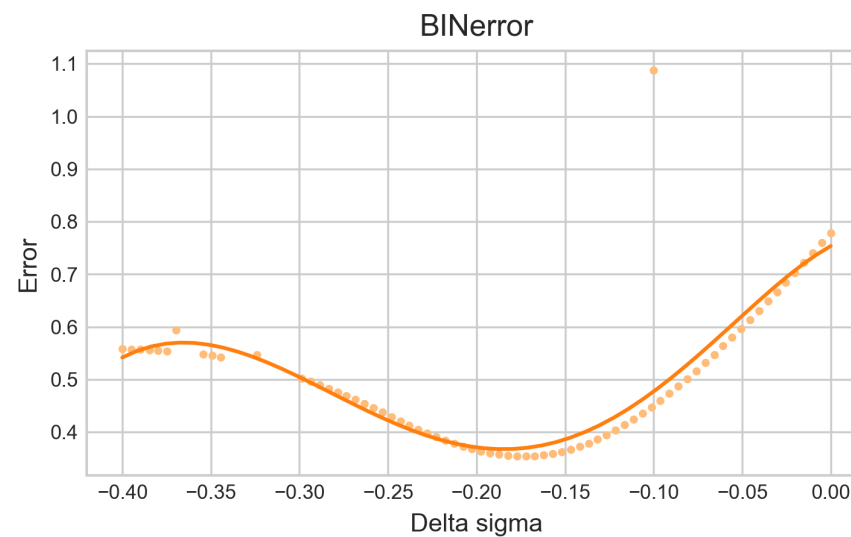
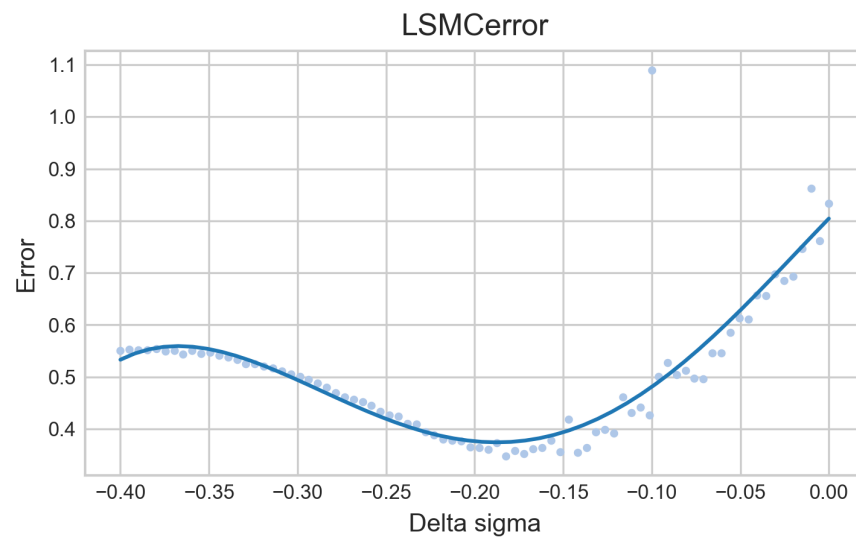
$$\lambda = 0.15, a = -0.1, b = 0.05$$

## **Interest rate and Standard Deviation calibration**

# Error given interest rate



# Error given volatility

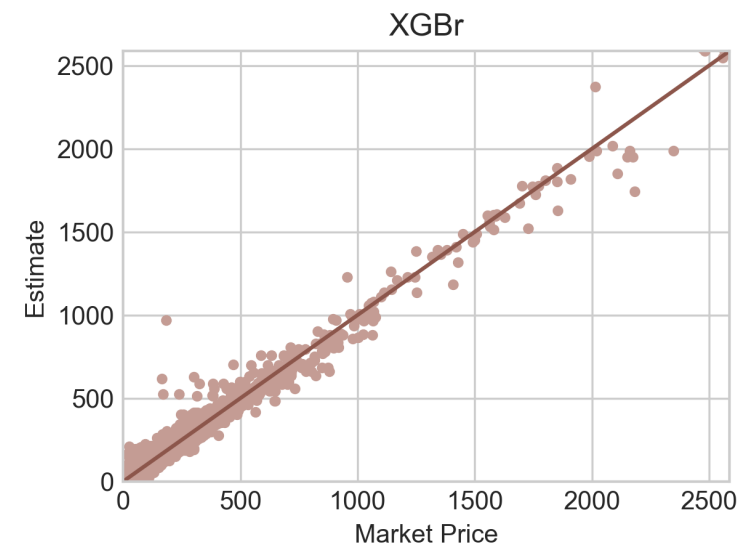
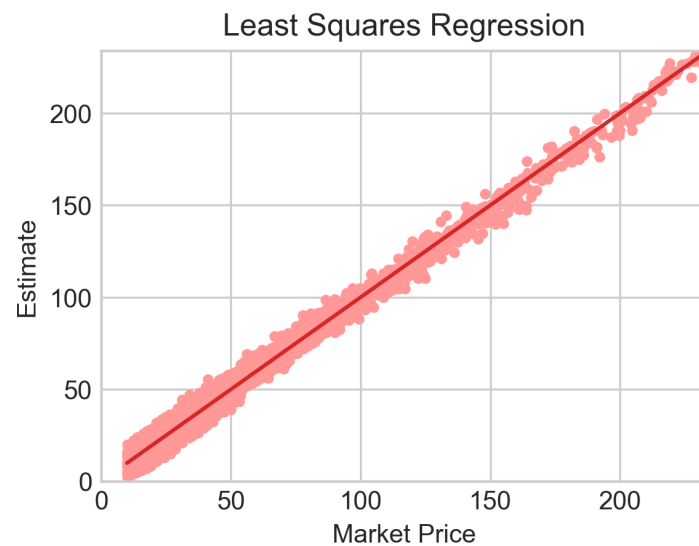
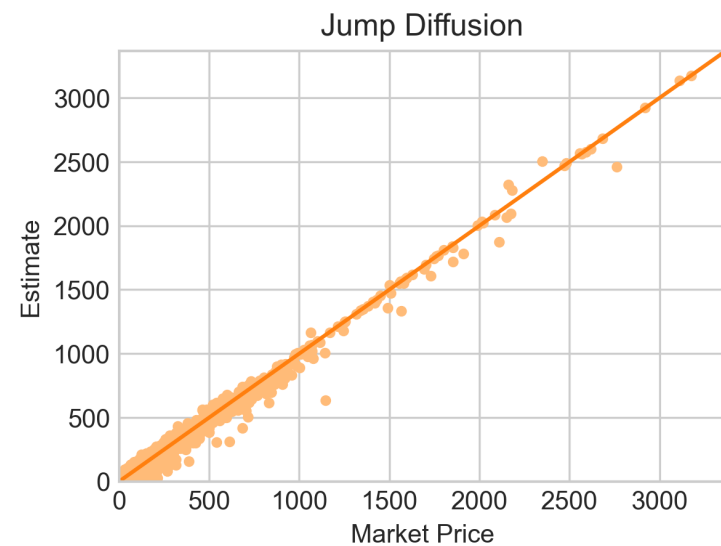
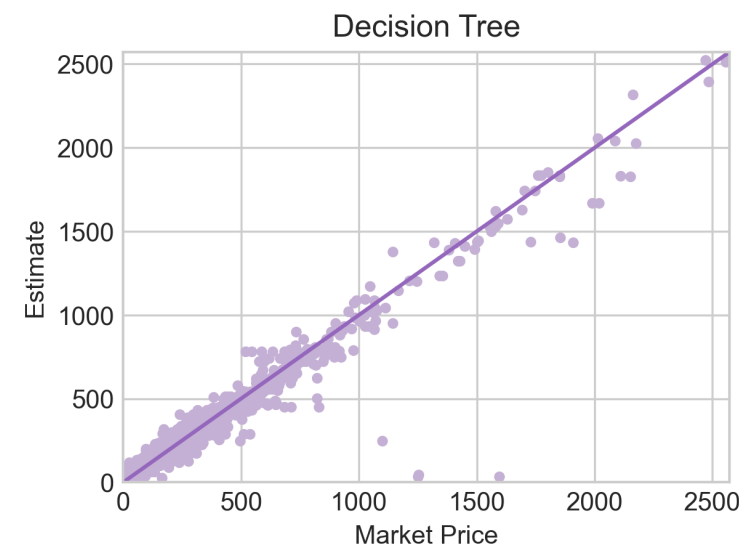
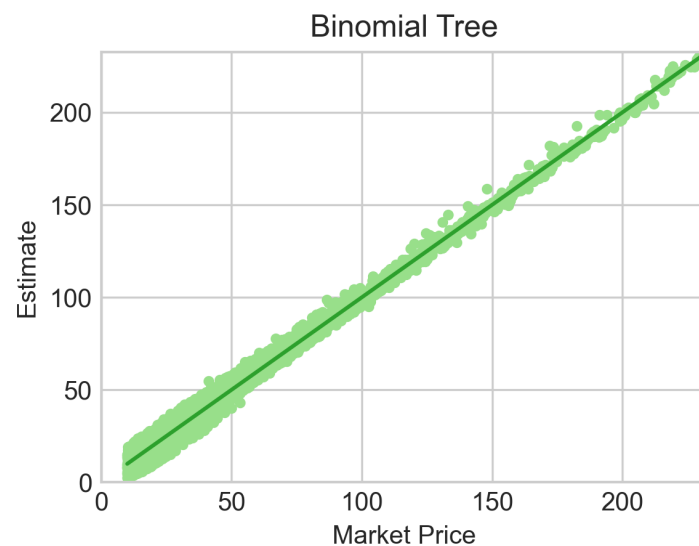
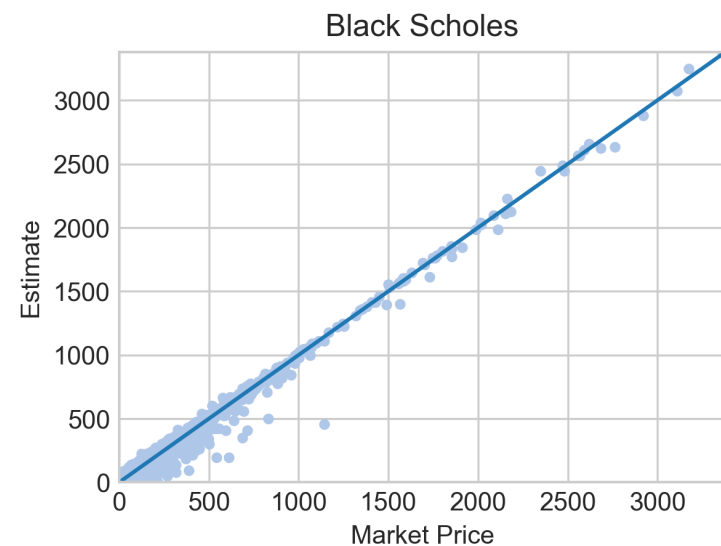


# Results

# Algorithm runtime

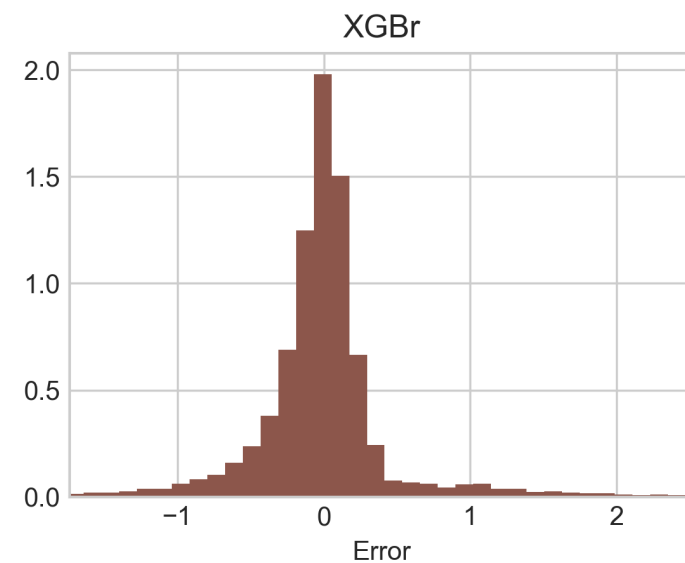
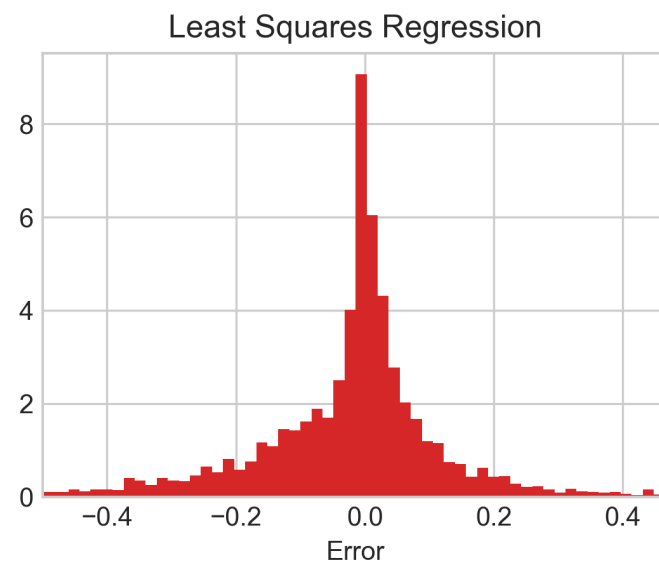
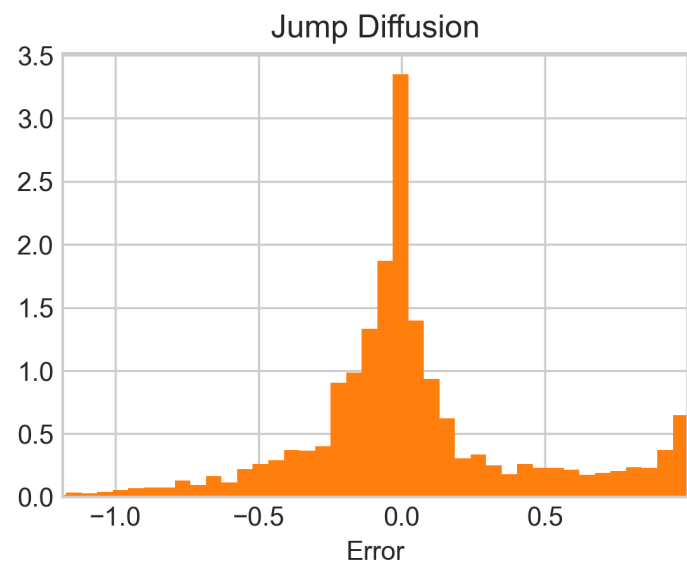
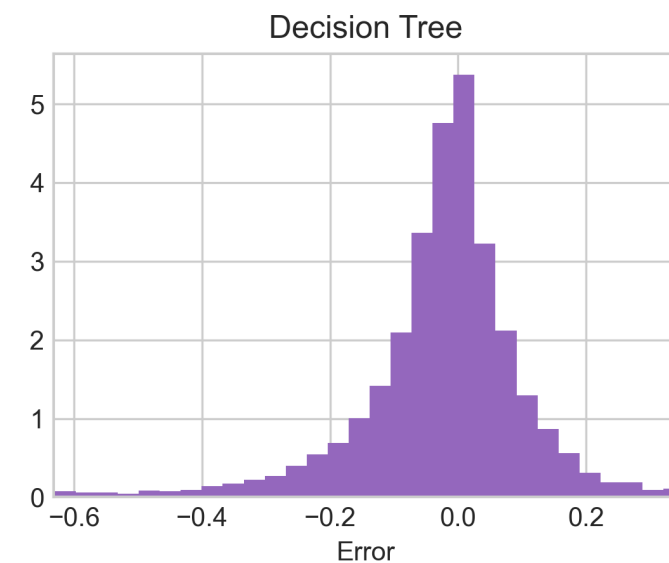
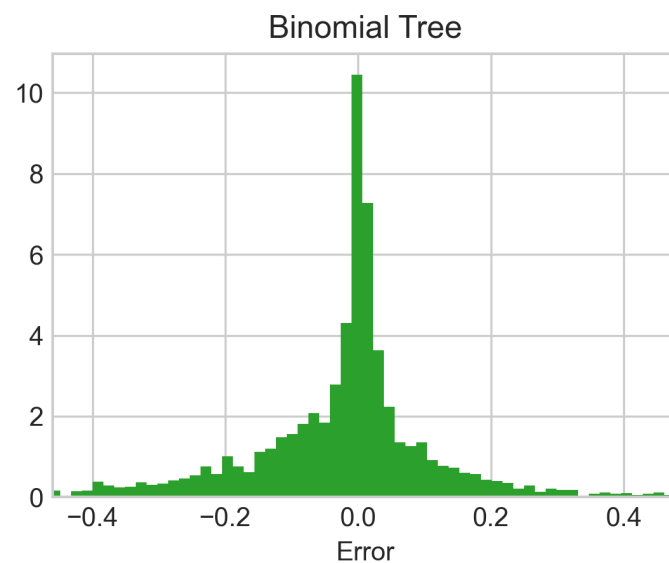
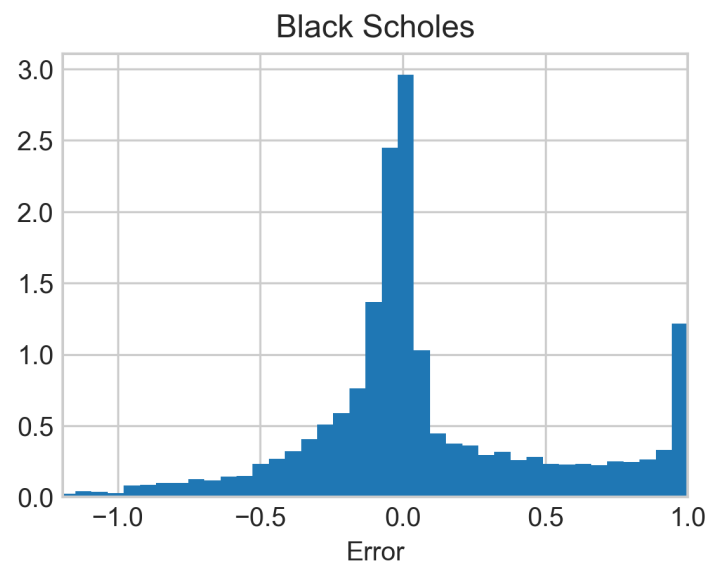


# Option prices: Predicted against Market





# Distribution of errors



	Average	MSE	Std Dev
<b>BS</b>	0.0664	0.2289	0.4738
<b>MJD</b>	0.0211	0.1910	0.4365
<b>LSMC</b>	-0.0215	0.0254	0.1580
<b>BIN</b>	-0.0196	0.0242	0.1545
<b>DTR</b>	-0.0307	0.0334	0.1802
<b>XGBr</b>	-0.0009	0.4305	0.6561

