Volatility Surface on SPX option

October 4, 2023

1 Volatility Surface of SPX options

1.1 a) Data Downloading

1.1.1 a.1 SP500 index data

[ ]:
```
import yfinance as yf
import pandas as pd

# Define the S&P 500 index symbol
spx_symbol = '^GSPC'

# Get S&P 500 historical data for 1 year
spx_data = yf.download(spx_symbol, period="1y").iloc[:-1, ]

spx_data.head()
```

```
[**********100%%**********]  1 of 1 completed
```

[ ]:
```
                  Open         High          Low        Close    Adj Close  \
Date
2022-10-03  3609.780029  3698.350098  3604.929932  3678.429932  3678.429932
2022-10-04  3726.459961  3791.919922  3726.459961  3790.929932  3790.929932
2022-10-05  3753.250000  3806.909912  3722.659912  3783.280029  3783.280029
2022-10-06  3771.969971  3797.929932  3739.219971  3744.520020  3744.520020
2022-10-07  3706.739990  3706.739990  3620.729980  3639.659912  3639.659912

                Volume
Date
2022-10-03  4806680000
2022-10-04  5146580000
2022-10-05  4293180000
2022-10-06  4252100000
2022-10-07  4449660000
```

[ ]:
```
spx_data.tail()
```

[ ]:
```
                  Open         High          Low        Close    Adj Close  \
Date
2023-09-25  4310.620117  4338.509766  4302.700195  4337.439941  4337.439941
2023-09-26  4312.879883  4313.009766  4265.979980  4273.529785  4273.529785
2023-09-27  4282.629883  4292.069824  4238.629883  4274.509766  4274.509766
2023-09-28  4269.649902  4317.270020  4264.379883  4299.700195  4299.700195
2023-09-29  4328.180176  4333.149902  4274.859863  4288.049805  4288.049805

                Volume
Date
2023-09-25  3195650000
2023-09-26  3472340000
2023-09-27  3875880000
2023-09-28  3846230000
2023-09-29  3865960000
```

[ ]:
```
spx_data.info()
```

```
<class 'pandas.core.frame.DataFrame'>

DatetimeIndex: 250 entries, 2022-10-03 to 2023-09-29

Data columns (total 6 columns):

 #   Column     Non-Null Count  Dtype

---  ------     --------------  -----

 0   Open       250 non-null    float64

 1   High       250 non-null    float64

 2   Low        250 non-null    float64

 3   Close      250 non-null    float64

 4   Adj Close  250 non-null    float64

 5   Volume     250 non-null    int64

dtypes: float64(5), int64(1)

memory usage: 13.7 KB
```

1.1.2 a.2 get risk-free int rate

from https://www.federalreserve.gov/releases/h15/

we know fed fund rate is 5.33%

here we use fed fund rate as risk-free rate

[ ]:
```
r = 5.33 /100
```

1.1.3 a.3 get SPX option data

source: https://www.cboe.com/delayed_quotes/spx/quote_table

a.3.1 One month SPX option data

[ ]:
```
data = pd.read_csv('spx_quotedata_1month.csv', skiprows = 3)
data['Expiration Date'] = pd.to_datetime(data['Expiration Date'])
data[['Calls','Puts']] = data[['Calls','Puts']].astype(str)
data.columns = [col.replace(' ', '_') for col in [col.replace('.1', '_put') for␣
 ↪col in data.columns]]
SPX_option_1month = data.loc[data["Expiration_Date"] == '2023-10-30',␣
 ↪['Expiration_Date', 'Calls', 'Bid', 'Ask', 'IV', 'Strike', 'Puts',␣
 ↪'Bid_put', 'Ask_put', 'IV_put']].copy()
SPX_option_1month.head()
```

[ ]:

	Expiration_Date	Calls	Bid	Ask	IV	Strike	\
2770	2023-10-30	SPXW231030C02800000	1497.6	1505.7	0.5903	2800.0	
2771	2023-10-30	SPXW231030C03000000	1298.8	1306.8	0.5132	3000.0	
2772	2023-10-30	SPXW231030C03200000	1100.0	1107.8	0.4393	3200.0	
2773	2023-10-30	SPXW231030C03300000	1001.7	1007.5	0.4040	3300.0	
2774	2023-10-30	SPXW231030C03400000	901.5	909.2	0.3710	3400.0	

	Puts	Bid_put	Ask_put	IV_put
2770	SPXW231030P02800000	0.35	0.55	0.5340
2771	SPXW231030P03000000	0.55	0.75	0.4728
2772	SPXW231030P03200000	0.95	1.10	0.4166
2773	SPXW231030P03300000	1.20	1.45	0.3898
2774	SPXW231030P03400000	1.60	1.75	0.3602

a.3.1 Three month SPX option data

[ ]:
```
data = pd.read_csv('spx_quotedata_3month.csv', skiprows = 3)
data['Expiration Date'] = pd.to_datetime(data['Expiration Date'])
data[['Calls','Puts']] = data[['Calls','Puts']].astype(str)
data.columns = [col.replace(' ', '_') for col in [col.replace('.1', '_put') for␣
 ↪col in data.columns]]
SPX_option_3month = data.loc[data["Expiration_Date"] == '2023-12-29',␣
 ↪['Expiration_Date', 'Calls', 'Bid', 'Ask', 'IV', 'Strike', 'Puts',␣
 ↪'Bid_put', 'Ask_put', 'IV_put']].copy()
SPX_option_3month.head()
```

[ ]:

	Expiration_Date	Calls	Bid	Ask	IV	Strike	\
462	2023-12-29	SPXW231229C00800000	3480.8	3493.3	0.0	800.0	
463	2023-12-29	SPXW231229C01000000	3287.8	3297.8	0.0	1000.0	
464	2023-12-29	SPXW231229C01200000	3090.6	3100.5	0.0	1200.0	
465	2023-12-29	SPXW231229C01400000	2890.2	2901.9	0.0	1400.0	
466	2023-12-29	SPXW231229C01800000	2496.6	2507.9	0.0	1800.0	

	Puts	Bid_put	Ask_put	IV_put
462	SPXW231229P00800000	0.00	0.10	0.9747
463	SPXW231229P01000000	0.00	0.15	0.8710
464	SPXW231229P01200000	0.05	0.20	0.7805
465	SPXW231229P01400000	0.10	0.25	0.7265
466	SPXW231229P01800000	0.35	0.55	0.6082

a.3.2 Six month SPX option data

[ ]:
```
data = pd.read_csv('spx_quotedata_6month.csv', skiprows = 3)
data['Expiration Date'] = pd.to_datetime(data['Expiration Date'])
data[['Calls','Puts']] = data[['Calls','Puts']].astype(str)
data.columns = [col.replace(' ', '_') for col in [col.replace('.1', '_put') for␣
 ↪col in data.columns]]
SPX_option_6month = data.loc[data["Expiration_Date"] == '2024-3-28',␣
 ↪['Expiration_Date', 'Calls', 'Bid', 'Ask', 'IV', 'Strike', 'Puts',␣
 ↪'Bid_put', 'Ask_put', 'IV_put']].copy()
SPX_option_6month.head()
```

[ ]:

	Expiration_Date	Calls	Bid	Ask	IV	Strike	\
121	2024-03-28	SPXW240328C02000000	2315.1	2329.7	0.4766	2000.0	
122	2024-03-28	SPXW240328C02600000	1739.0	1748.5	0.3751	2600.0	
123	2024-03-28	SPXW240328C02850000	1499.7	1508.7	0.3375	2850.0	
124	2024-03-28	SPXW240328C03100000	1262.4	1270.8	0.3028	3100.0	
125	2024-03-28	SPXW240328C03150000	1215.3	1223.6	0.2962	3150.0	

	Puts	Bid_put	Ask_put	IV_put
121	SPXW240328P02000000	3.1	3.5	0.4812
122	SPXW240328P02600000	7.7	8.1	0.3770
123	SPXW240328P02850000	11.1	11.7	0.3388
124	SPXW240328P03100000	16.4	17.1	0.3038
125	SPXW240328P03150000	17.8	18.5	0.2969

a.3.3 One year SPX option data

[ ]:
```
data = pd.read_csv('spx_quotedata_1year.csv', skiprows = 3)
data['Expiration Date'] = pd.to_datetime(data['Expiration Date'])
data[['Calls','Puts']] = data[['Calls','Puts']].astype(str)
data.columns = [col.replace(' ', '_') for col in [col.replace('.1', '_put') for␣
 ↪col in data.columns]]
```

```
SPX_option_1year = data.loc[data["Expiration_Date"] == '2024-9-30',␣
 ↪['Expiration_Date', 'Calls', 'Bid', 'Ask', 'IV', 'Strike', 'Puts',␣
 ↪'Bid_put', 'Ask_put', 'IV_put']].copy()
SPX_option_1year.head()
```

[ ]:

	Expiration_Date	Calls	Bid	Ask	IV	Strike	\
82	2024-09-30	SPXW240930C01000000	3279.6	3303.5	0.6135	1000.0	
83	2024-09-30	SPXW240930C03825000	729.2	738.5	0.2163	3825.0	
84	2024-09-30	SPXW240930C03850000	709.8	719.1	0.2145	3850.0	
85	2024-09-30	SPXW240930C03900000	671.6	680.6	0.2108	3900.0	
86	2024-09-30	SPXW240930C04000000	596.9	605.5	0.2037	4000.0	

	Puts	Bid_put	Ask_put	IV_put
82	SPXW240930P01000000	1.8	2.25	0.5908
83	SPXW240930P03825000	112.9	117.80	0.2163
84	SPXW240930P03850000	117.1	122.10	0.2145
85	SPXW240930P03900000	126.5	131.00	0.2109
86	SPXW240930P04000000	146.0	150.90	0.2037

1.2 b.1 get option price based on BSM model

https://www.macroption.com/black-scholes-formula/

[ ]:
```
import numpy as np
from scipy.stats import norm

def get_bsm_option_price(S, K, T, r, sigma, q, option_type='call'):

    d1 = (np.log(S / K) + (r - q + (sigma ** 2) / 2) * T) / (sigma * np.sqrt(T))
    d2 = d1 - sigma * np.sqrt(T)

    if option_type == 'call':
        option_price = S * np.exp(-q * T) * norm.cdf(d1) - K * np.exp(-r * T) *␣
 ↪norm.cdf(d2)
    elif option_type == 'put':
        option_price = K * np.exp(-r * T) * norm.cdf(-d2) - S * np.exp(-q * T)␣
 ↪* norm.cdf(-d1)
    else:
        raise ValueError("Option type must be 'call' or 'put'")

    return option_price
```

1.3 b2. get IV with Newton Method

[ ]:
```
def get_implied_volatility(option_price, S, K, T, r, q, option_type='call',␣
↪max_iter=1000, tol=1e-6):
    # Initial guess for implied volatility (e.g., use historical volatility)
    sigma_guess = 0.2

    for i in range(max_iter):
        option_price_calculated = get_bsm_option_price(S, K, T, r, sigma_guess,␣
↪q, option_type)
        vega = S * np.exp(-q * T) * np.sqrt(T) * norm.pdf((np.log(S / K) + (r -␣
↪q + (sigma_guess ** 2) / 2) * T) / (sigma_guess * np.sqrt(T)))

        # Newton's method update formula
        sigma_guess -= (option_price_calculated - option_price) / vega

    return sigma_guess
```

1.4 c) generate IV smiles

dividend yield source: https://www.gurufocus.com/economic_indicators/150/sp-500-dividend-yield

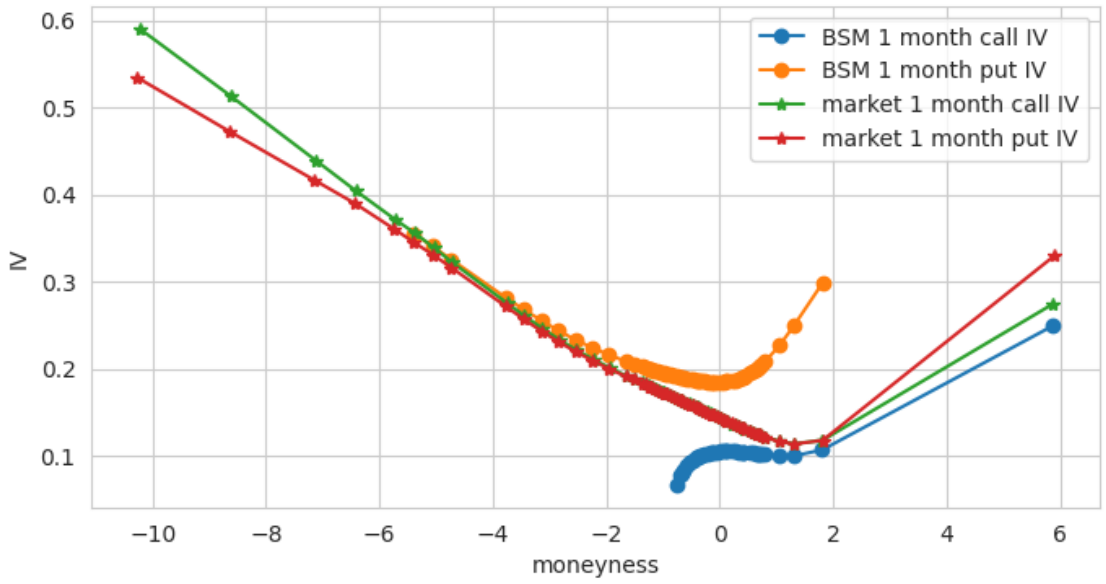
[ ]:
```
maturities = {
    '1 Month': [SPX_option_1month, pd.DataFrame([])],
    '3 Months': [SPX_option_3month, pd.DataFrame([])],
    '6 Months': [SPX_option_6month, pd.DataFrame([])],
    '12 Months': [SPX_option_1year, pd.DataFrame([])]
}
```

[ ]:
```
for k, v in maturities.items():
  v[1]['price_call'] = (v[0].iloc[:, 2] + v[0].iloc[:, 3])/2
  v[1]['price_put'] = (v[0].iloc[:, 7] + v[0].iloc[:, 8])/2
  v[1]['time_to_maturity'] = ((v[0].iloc[:, 0] - pd.to_datetime('2023-09-29')).
↪dt.days)/360
  v[1]['volatility'] = np.log(spx_data.iloc[-(v[0].iloc[0, 0] - pd.
↪to_datetime('2023-09-29')).days:, 4]/ spx_data.iloc[-(v[0].iloc[0, 0] - pd.
↪to_datetime('2023-09-29')).days:, 4].shift(1)).std() * np.sqrt(252)
  v[1]['strike'] = v[0].iloc[:, 5].copy()
  v[1]['market_IV_call'] = v[0].iloc[:, 4].copy()
  v[1]['market_IV_put'] = v[0].iloc[:, 9].copy()
  v[1]['int_rate'] = r
  # v[1]['dividend_yield'] = 1.59/100
  v[1]['dividend_yield'] = 0
  # v[1]['index_price'] = spx_data.iloc[-1, 4]
  v[1]['index_price'] = spx_data.iloc[-1, 0]
```

```
    v[1]['bsm_call_price'] = get_bsm_option_price(v[1]['index_price'],␣
    ↪v[1]['strike'], v[1]['time_to_maturity'], v[1]['int_rate'],␣
    ↪v[1]['volatility'], v[1]['dividend_yield'], option_type='call')
    v[1]['bsm_put_price'] = get_bsm_option_price(v[1]['index_price'],␣
    ↪v[1]['strike'], v[1]['time_to_maturity'], v[1]['int_rate'],␣
    ↪v[1]['volatility'], v[1]['dividend_yield'], option_type='put')
    v[1]['bsm_call_IV'] = get_implied_volatility(v[1]['price_call'],␣
    ↪v[1]['index_price'], v[1]['strike'], v[1]['time_to_maturity'],␣
    ↪v[1]['int_rate'], v[1]['dividend_yield'], option_type='call')
    v[1]['bsm_put_IV'] = get_implied_volatility(v[1]['price_put'],␣
    ↪v[1]['index_price'], v[1]['strike'], v[1]['time_to_maturity'],␣
    ↪v[1]['int_rate'], v[1]['dividend_yield'], option_type='put')
    v[1]['moneyness_call'] = np.log(v[1]['strike']/(v[1]['index_price'] * np.
    ↪exp(v[1]['int_rate'] * v[1]['time_to_maturity'])))/v[0].iloc[np.
    ↪argmin(abs(v[0]['Strike'] - spx_data.iloc[-1, 0])), 4]/np.
    ↪sqrt(v[1]['time_to_maturity'])
    v[1]['moneyness_put'] = np.log(v[1]['strike']/(v[1]['index_price'] * np.
    ↪exp(v[1]['int_rate'] * v[1]['time_to_maturity'])))/v[0].iloc[np.
    ↪argmin(abs(v[0]['Strike'] - spx_data.iloc[-1, 0])), 9]/np.
    ↪sqrt(v[1]['time_to_maturity'])
```

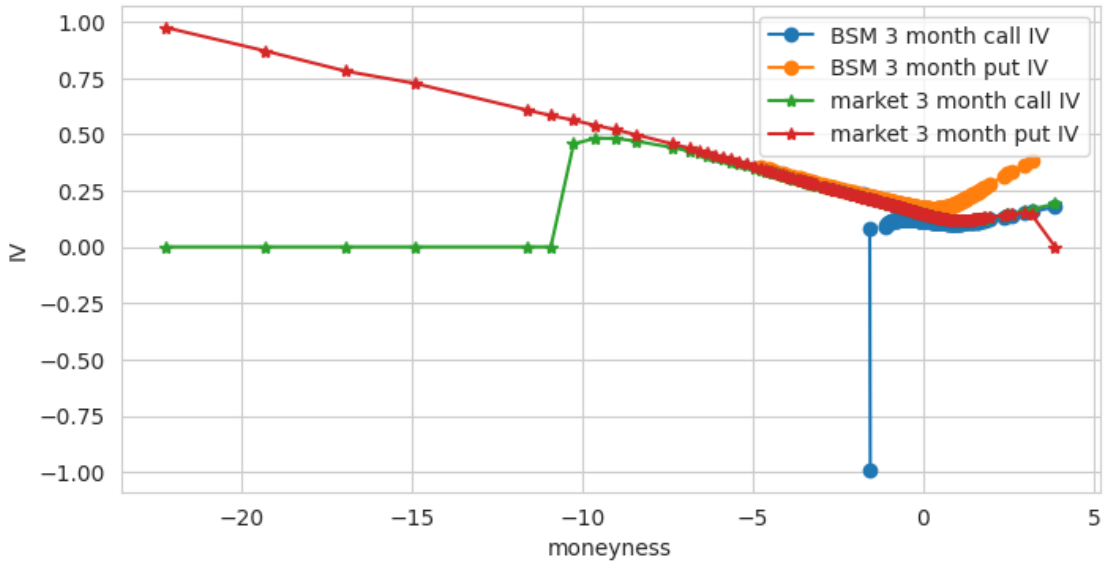
[ ]:
```
import matplotlib.pyplot as plt
plt.figure(figsize = (8, 4))
plt.plot(maturities['1 Month'][1]['moneyness_call'], maturities['1␣
  ↪Month'][1]['bsm_call_IV'], label = 'BSM 1 month call IV', marker = 'o')
plt.plot(maturities['1 Month'][1]['moneyness_put'], maturities['1␣
  ↪Month'][1]['bsm_put_IV'], label = 'BSM 1 month put IV',  marker = 'o')
plt.plot(maturities['1 Month'][1]['moneyness_call'], maturities['1␣
  ↪Month'][1]['market_IV_call'], label = 'market 1 month call IV', marker = '*')
plt.plot(maturities['1 Month'][1]['moneyness_put'], maturities['1␣
  ↪Month'][1]['market_IV_put'], label = 'market 1 month put IV',  marker = '*')

plt.xlabel('moneyness')
plt.ylabel('IV')

plt.legend()
```

[ ]:
```
<matplotlib.legend.Legend at 0x7b02bd04d0f0>
```



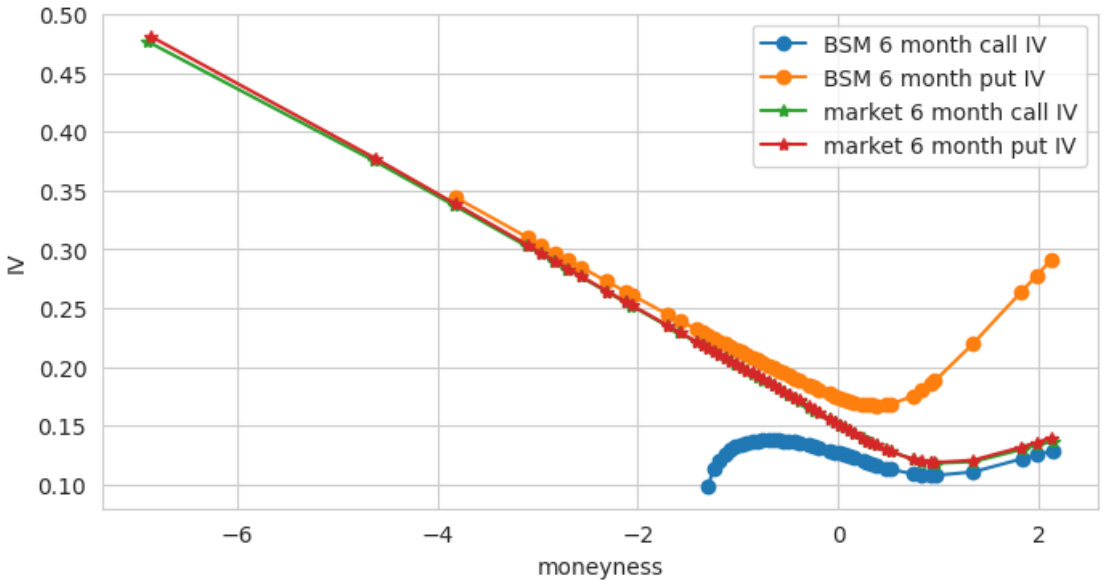
[ ]:
```
plt.figure(figsize = (8, 4))
plt.plot(maturities['3 Months'][1]['moneyness_call'], maturities['3␣
 ↪Months'][1]['bsm_call_IV'], label = 'BSM 3 month call IV', marker = 'o')
plt.plot(maturities['3 Months'][1]['moneyness_put'], maturities['3␣
 ↪Months'][1]['bsm_put_IV'], label = 'BSM 3 month put IV',  marker = 'o')
plt.plot(maturities['3 Months'][1]['moneyness_call'], maturities['3␣
 ↪Months'][1]['market_IV_call'], label = 'market 3 month call IV', marker =␣
 ↪'*')
plt.plot(maturities['3 Months'][1]['moneyness_put'], maturities['3␣
 ↪Months'][1]['market_IV_put'], label = 'market 3 month put IV',  marker = '*')

plt.xlabel('moneyness')
plt.ylabel('IV')

plt.legend()
```

[ ]:
```
<matplotlib.legend.Legend at 0x7b02bc726740>
```



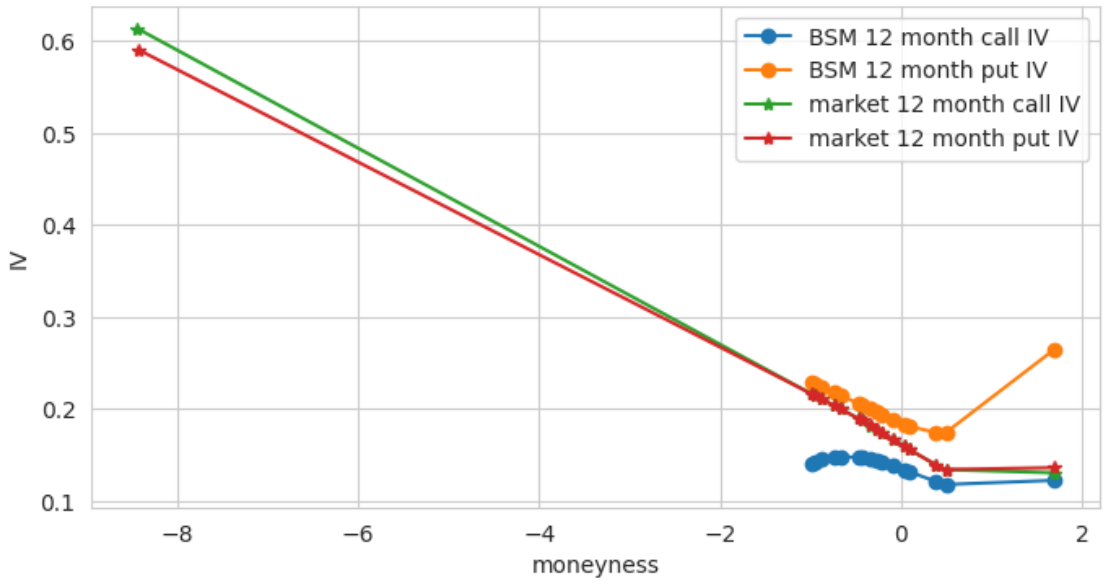
[ ]:
```
plt.figure(figsize = (8, 4))
plt.plot(maturities['6 Months'][1]['moneyness_call'], maturities['6␣
 ↪Months'][1]['bsm_call_IV'], label = 'BSM 6 month call IV', marker = 'o')
plt.plot(maturities['6 Months'][1]['moneyness_put'], maturities['6␣
 ↪Months'][1]['bsm_put_IV'], label = 'BSM 6 month put IV',  marker = 'o')
plt.plot(maturities['6 Months'][1]['moneyness_call'], maturities['6␣
 ↪Months'][1]['market_IV_call'], label = 'market 6 month call IV', marker =␣
 ↪'*')
plt.plot(maturities['6 Months'][1]['moneyness_put'], maturities['6␣
 ↪Months'][1]['market_IV_put'], label = 'market 6 month put IV',  marker = '*')
plt.xlabel('moneyness')
plt.ylabel('IV')

plt.legend()
```

[ ]:
```
<matplotlib.legend.Legend at 0x7b02bc5c5570>
```



[ ]:
```
plt.figure(figsize = (8, 4))
plt.plot(maturities['12 Months'][1]['moneyness_call'], maturities['12␣
 ↪Months'][1]['bsm_call_IV'], label = 'BSM 12 month call IV', marker = 'o')
plt.plot(maturities['12 Months'][1]['moneyness_put'], maturities['12␣
 ↪Months'][1]['bsm_put_IV'], label = 'BSM 12 month put IV',  marker = 'o')
plt.plot(maturities['12 Months'][1]['moneyness_call'], maturities['12␣
 ↪Months'][1]['market_IV_call'], label = 'market 12 month call IV', marker =␣
 ↪'*')
plt.plot(maturities['12 Months'][1]['moneyness_put'], maturities['12␣
 ↪Months'][1]['market_IV_put'], label = 'market 12 month put IV',  marker =␣
 ↪'*')

plt.xlabel('moneyness')
plt.ylabel('IV')

plt.legend()
```

[ ]:
```
<matplotlib.legend.Legend at 0x7b02bc643b50>
```



1.5 d) generate implied volatility surface

[ ]:
```
dfs = [maturities['1 Month'][1][['time_to_maturity', 'moneyness_call',␣
 ↪'market_IV_call', 'strike']], \
          maturities['1 Month'][1][['time_to_maturity', 'moneyness_put',␣
 ↪'market_IV_put', 'strike']], \
          maturities['3 Months'][1][['time_to_maturity', 'moneyness_call',␣
 ↪'market_IV_call', 'strike']], \
          maturities['3 Months'][1][['time_to_maturity', 'moneyness_put',␣
 ↪'market_IV_put', 'strike']], \
          maturities['6 Months'][1][['time_to_maturity', 'moneyness_call',␣
 ↪'market_IV_call', 'strike']], \
          maturities['6 Months'][1][['time_to_maturity', 'moneyness_put',␣
 ↪'market_IV_put', 'strike']], \
          maturities['12 Months'][1][['time_to_maturity', 'moneyness_call',␣
 ↪'market_IV_call', 'strike']], \
          maturities['12 Months'][1][['time_to_maturity', 'moneyness_put',␣
 ↪'market_IV_put', 'strike']], \
          ]

for i in range(len(dfs)):
  option_type = 'call' if 'call' in dfs[i].columns[1] else 'put'
  dfs[i] = dfs[i].reset_index(drop=True)
  dfs[i].columns = ['time_to_maturity', 'moneyness', 'market_IV', 'strike']
  dfs[i]['option_type'] = option_type
```

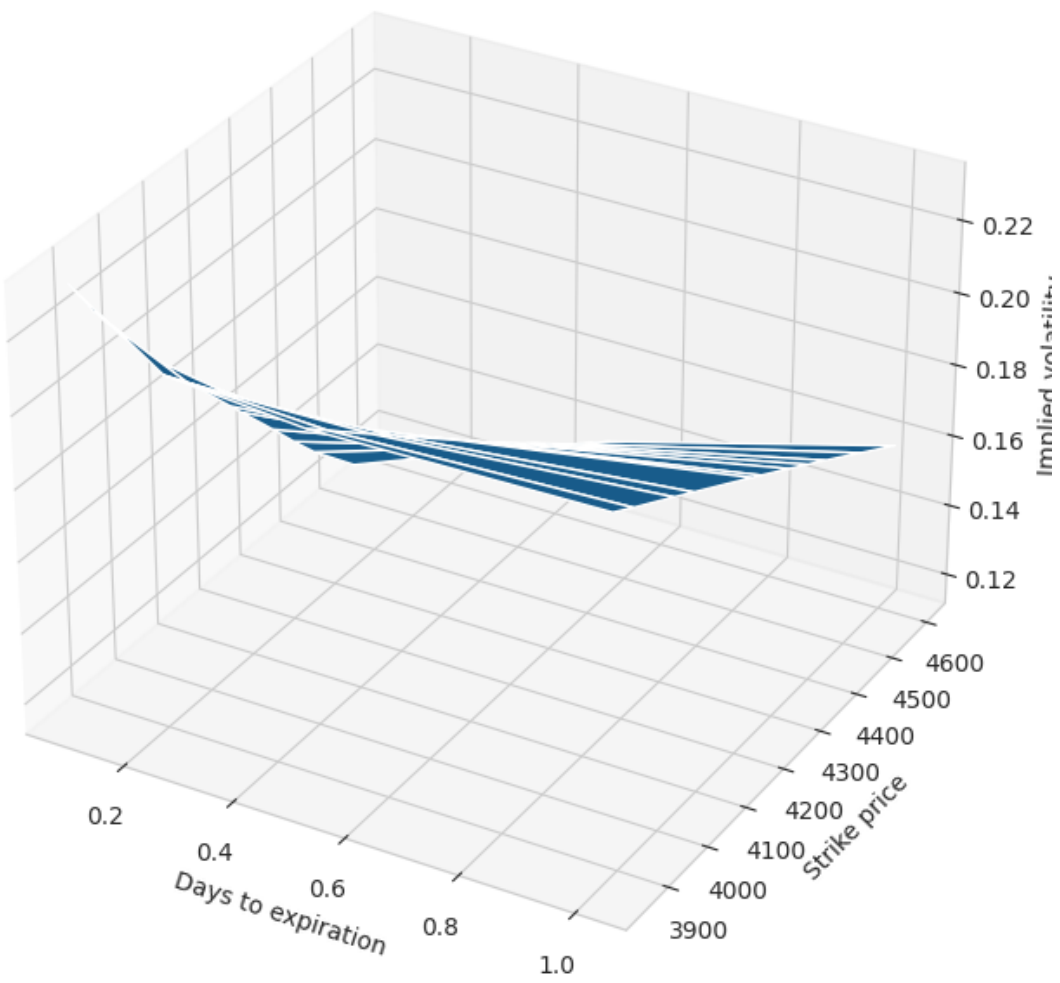
[ ]:
```
data = pd.concat(dfs)
```

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

[ ]:
```
   time_to_maturity  moneyness  market_IV  strike option_type
0          0.086111 -10.230693     0.5903  2800.0        call
1          0.086111  -8.626928     0.5132  3000.0        call
2          0.086111  -7.126706     0.4393  3200.0        call
3          0.086111  -6.411408     0.4040  3300.0        call
4          0.086111  -5.717465     0.3710  3400.0        call
```

[ ]:
```
# pivot the dataframe
surface_call = (
    data.loc[data['option_type'] == 'call', ['time_to_maturity', 'strike',␣
 ↪'market_IV']]
    .pivot_table(values='market_IV', index='strike', columns='time_to_maturity')
    .dropna()
)

# create the figure object
fig = plt.figure(figsize=(10, 8))

# add the subplot with projection argument
ax = fig.add_subplot(111, projection='3d')

# get the 1d values from the pivoted dataframe
x, y, z = surface_call.columns.values, surface_call.index.values, surface_call.
 ↪values

# return coordinate matrices from coordinate vectors
X, Y = np.meshgrid(x, y)

# set labels
ax.set_xlabel('Days to expiration')
ax.set_ylabel('Strike price')
ax.set_zlabel('Implied volatility')
ax.set_title('Call implied volatility surface')

# plot
ax.plot_surface(X, Y, z)
```

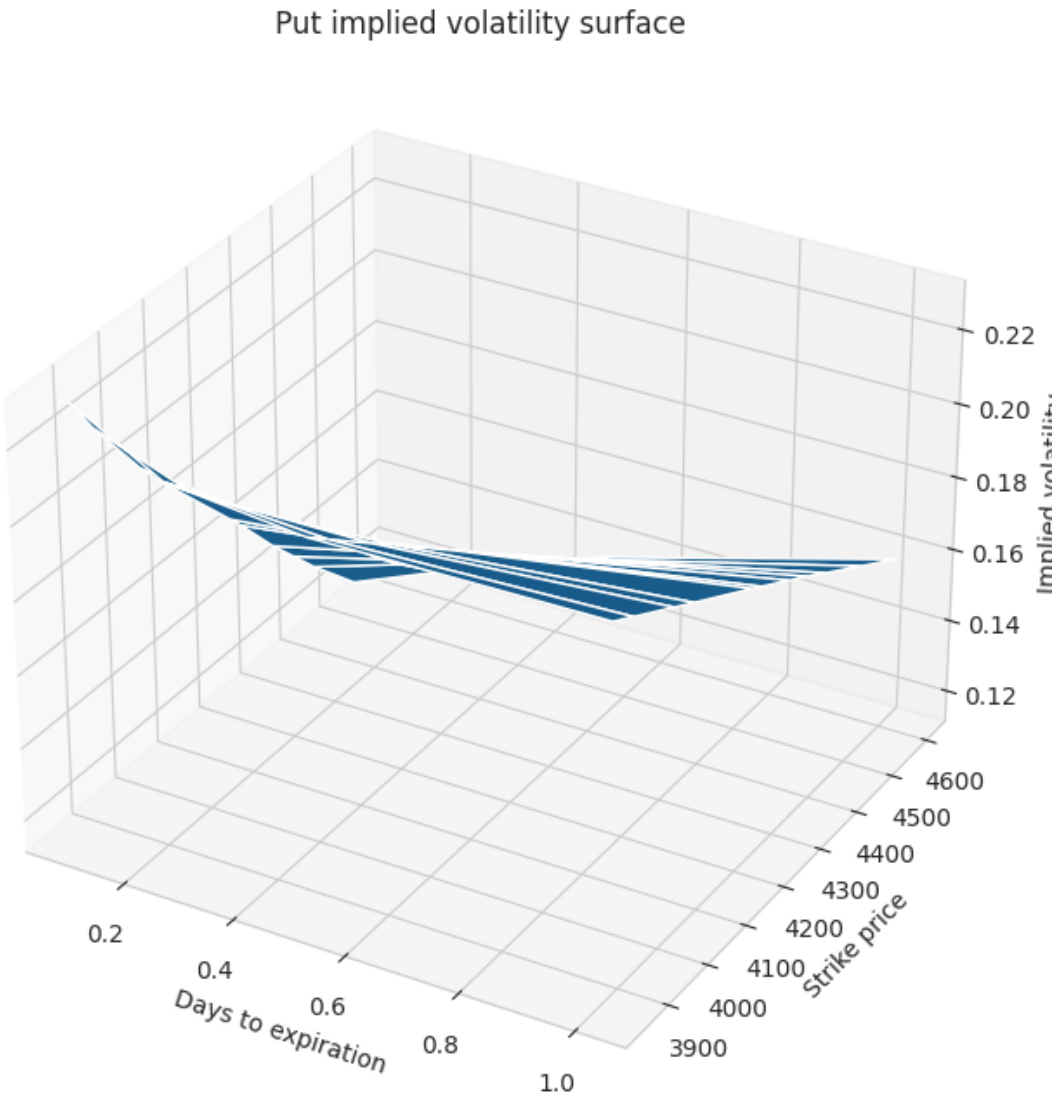
[ ]:
```
<mpl_toolkits.mplot3d.art3d.Poly3DCollection at 0x7b02bc4cb2e0>
```



[ ]:
```
# pivot the dataframe
surface_put = (
    data.loc[data['option_type'] == 'put', ['time_to_maturity', 'strike',␣
 ↪'market_IV']]
    .pivot_table(values='market_IV', index='strike', columns='time_to_maturity')
    .dropna()
)

# create the figure object
fig = plt.figure(figsize=(10, 8))

# add the subplot with projection argument
```

```
ax = fig.add_subplot(111, projection='3d')

# get the 1d values from the pivoted dataframe
x, y, z = surface_put.columns.values, surface_put.index.values, surface_put.
 ↪values

# return coordinate matrices from coordinate vectors
X, Y = np.meshgrid(x, y)

# set labels
ax.set_xlabel('Days to expiration')
ax.set_ylabel('Strike price')
ax.set_zlabel('Implied volatility')
ax.set_title('Put implied volatility surface')

# plot
ax.plot_surface(X, Y, z)
```

[ ]: <mpl_toolkits.mplot3d.art3d.Poly3DCollection at 0x7b02bc563d60>

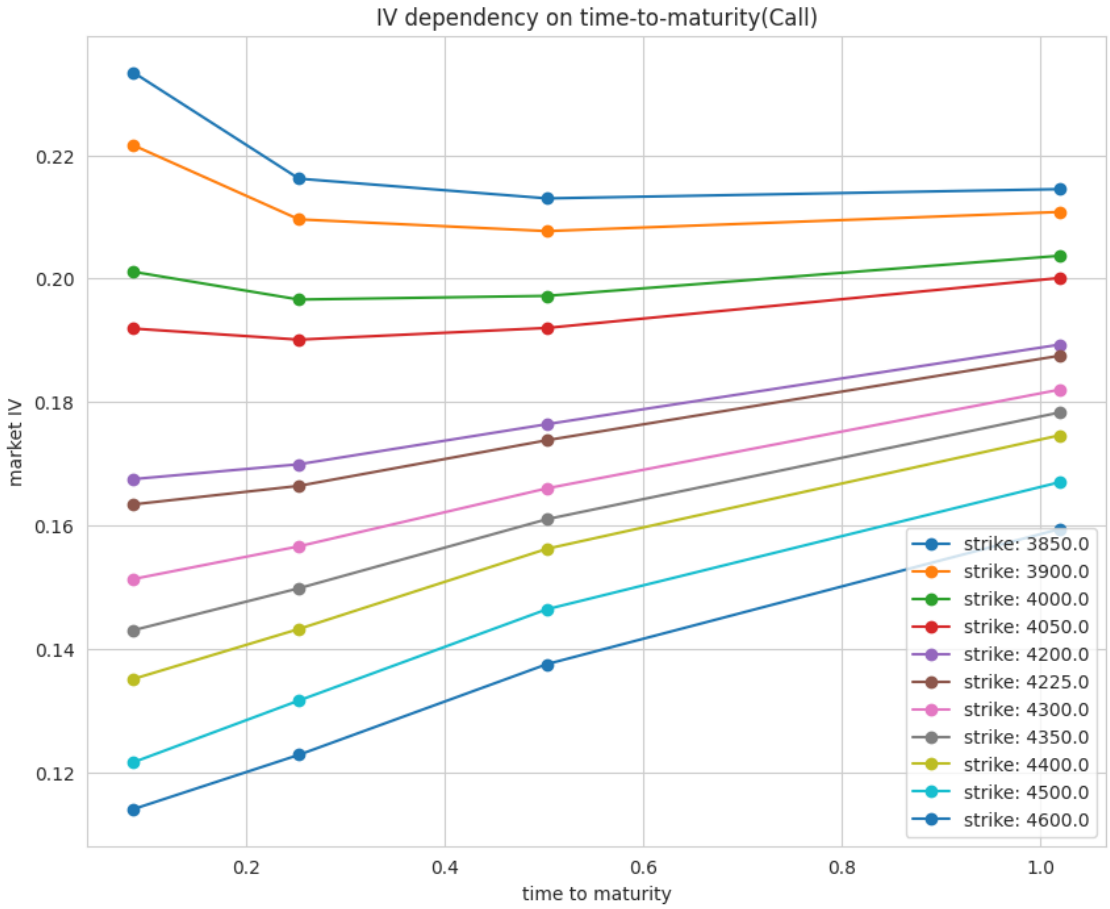


[ ]:
```
from collections import Counter
call_strike = [k for k, v in Counter(data.loc[data['option_type'] == 'call',␣
 ↪'strike']).items() if v >= 4]
put_strike = [k for k, v in Counter(data.loc[data['option_type'] == 'put',␣
 ↪'strike']).items() if v >= 4]
```

[ ]:
```
plt.figure(figsize = (10, 8))
for k in call_strike:
  temp = data.loc[(data['option_type'] == 'call')&(data['strike'] == k),␣
 ↪['time_to_maturity','market_IV']]
  plt.plot(temp['time_to_maturity'], temp['market_IV'], label = f'strike: {k}',␣
 ↪marker = 'o')
```

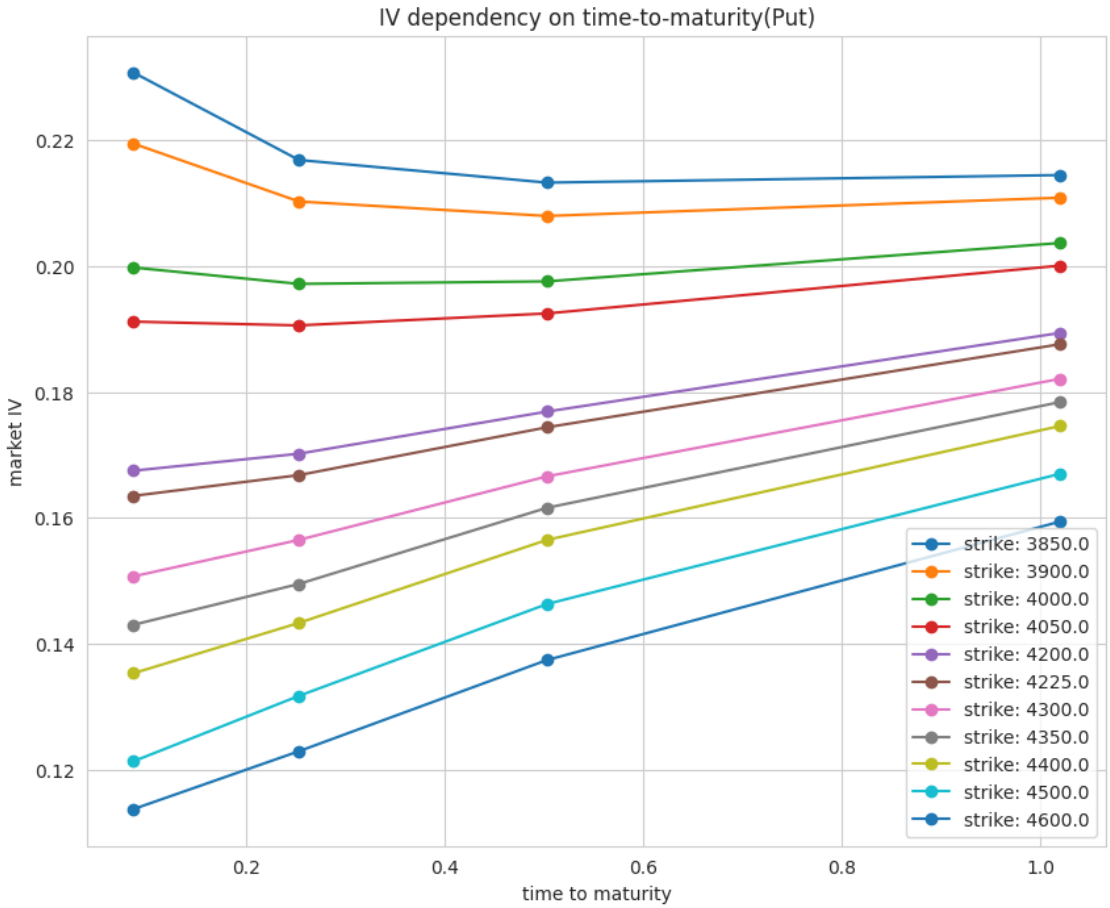
```
plt.xlabel('time to maturity')
plt.ylabel('market IV')
plt.title('IV dependency on time-to-maturity(Call)')
plt.legend()
```

[ ]: <matplotlib.legend.Legend at 0x7b02bc3bc220>



[ ]:
```
plt.figure(figsize = (10, 8))
for k in put_strike:
  temp = data.loc[(data['option_type'] == 'put')&(data['strike'] == k),␣
 ↪['time_to_maturity','market_IV']]
  plt.plot(temp['time_to_maturity'], temp['market_IV'], label = f'strike: {k}',␣
 ↪marker = 'o')
plt.xlabel('time to maturity')
plt.ylabel('market IV')
plt.title('IV dependency on time-to-maturity(Put)')
plt.legend()
```

[ ]: <matplotlib.legend.Legend at 0x7b02bd0549d0>



IV increases along with time to maturity, given a strike price.

That makes sense, because time value is increasing

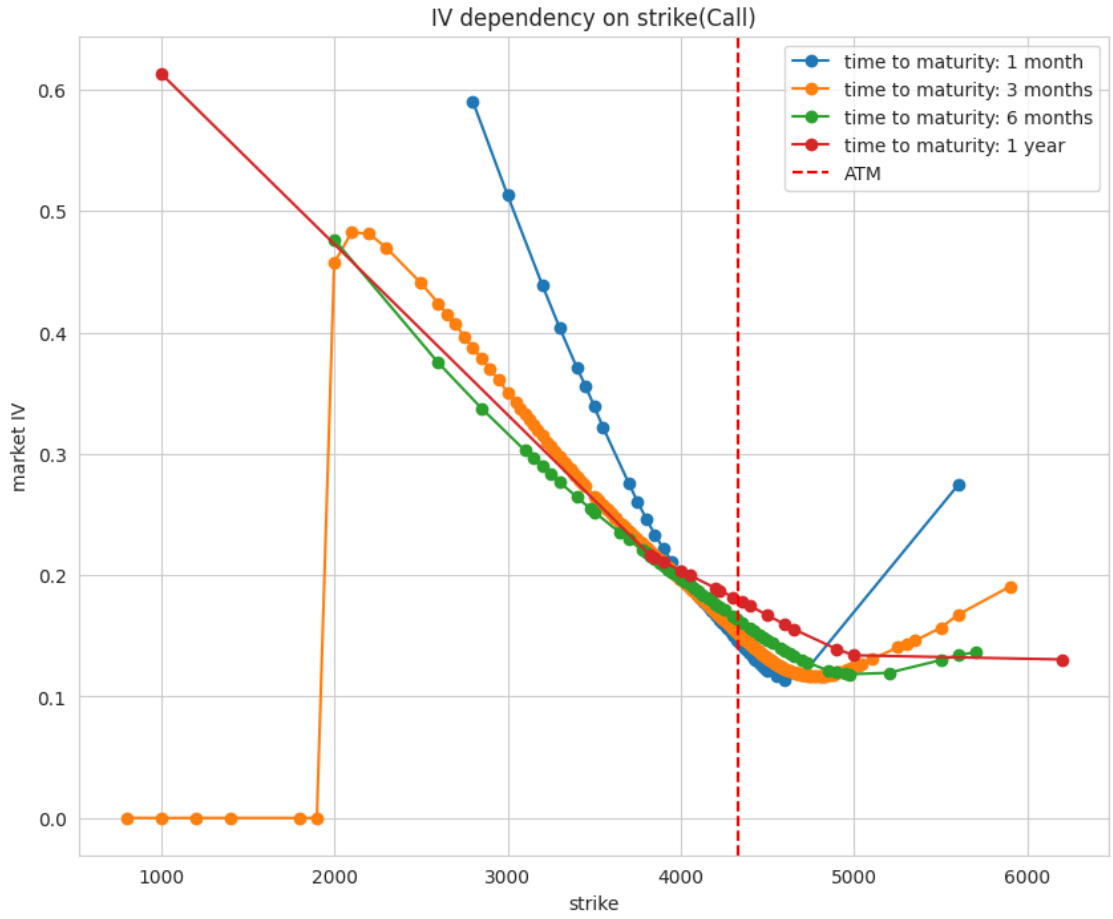
[ ]:
```
call_ttm = [k for k in Counter(data.loc[data['option_type'] == 'call',␣
 ↪'time_to_maturity']).keys()]
call_ttm
```

[ ]:
```
[0.08611111111111111,
 0.25277777777777777,
 0.5027777777777778,
 1.0194444444444444]
```

[ ]:
```
ttm_dic= {0.08611111111111111: '1 month',
          0.25277777777777777: '3 months',
          0.5027777777777778: '6 months',
          1.0194444444444444: '1 year'}
```

[ ]:
```
plt.figure(figsize = (10, 8))
for t in call_ttm:
  temp = data.loc[(data['option_type'] == 'call')&(data['time_to_maturity'] ==␣
 ↪t), ['strike','market_IV']]
  plt.plot(temp['strike'], temp['market_IV'], label = f'time to maturity:␣
 ↪{ttm_dic[t]}', marker = 'o')
plt.xlabel('strike')
plt.ylabel('market IV')
plt.title('IV dependency on strike(Call)')
plt.axvline(x=spx_data.iloc[-1, 0], color='red', linestyle='--', label='ATM')
plt.legend()
```

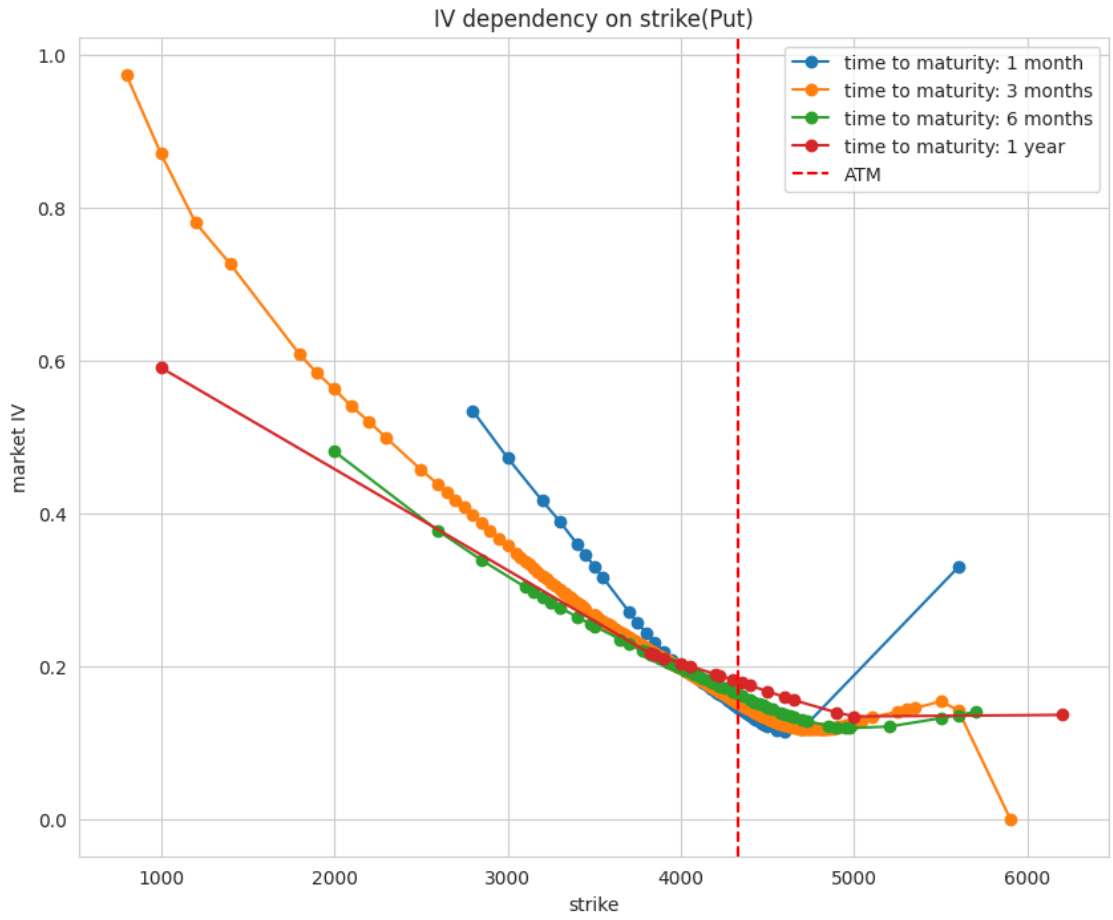
[ ]:
```
<matplotlib.legend.Legend at 0x7b02bc3336d0>
```



[ ]:
```
plt.figure(figsize = (10, 8))
for t in call_ttm:
  temp = data.loc[(data['option_type'] == 'put')&(data['time_to_maturity'] ==␣
 ↪t), ['strike','market_IV']]
```

```
  plt.plot(temp['strike'], temp['market_IV'], label = f'time to maturity:␣
 ↪{ttm_dic[t]}', marker = 'o')
plt.xlabel('strike')
plt.ylabel('market IV')
plt.axvline(x=spx_data.iloc[-1, 0], color='red', linestyle='--', label='ATM')
plt.title('IV dependency on strike(Put)')
plt.legend()
```

[ ]: <matplotlib.legend.Legend at 0x7b02bc1c8940>



ATM options tend to have a lower IV

OTM and ITM options have a higher IV

1.6 e) testing put call parity

1.6.1 e.1 show that put call parity under no arbitrage must have the same implied volatility

put call parity: $P + S = C + Ke^{-rt}$

BSM pricing formula for call: $c = S * N(d1_{call}) - Ke^{-rt}N(d2_{call})$

BSM pricing formula for put: $p = Ke^{-rt}N(-d2_{put}) - SN(-d1_{put})$

plug BSM pricing formula into put call parity: $Ke^{-rt} * (N(-d2_{put}) + Ke^{-rt} N(d2_{call})) - S(N(-d1_{put}) + N(d1_{call}))) = Ke^{-rt} - S$

we have $N(-d2_{put}) + N(d2_{call}) = 1$ and $N(-d1_{put}) + N(d1_{call}) = 1$

we know $N(x) + N(-x) = 1$ and $d1 = d1 - \sigma * \sqrt{t}$

so $d1_{call} = d1_{put}$

from d1 formula: $d1 = \frac{ln(\frac{S}{K})+(r+\frac{\sigma^2}{2})t}{\sigma\sqrt{t}} = d1(\sigma)$ we know it is a mono-increment function of σ

Therefore, $\sigma_{call} = \sigma_{put}$

1.6.2 e.2 test put call parity with market data

[ ]:
```
put_call = pd.concat([maturities['1 Month'][1], \
           maturities['3 Months'][1], \
           maturities['6 Months'][1], \
           maturities['12 Months'][1]]).dropna()
```

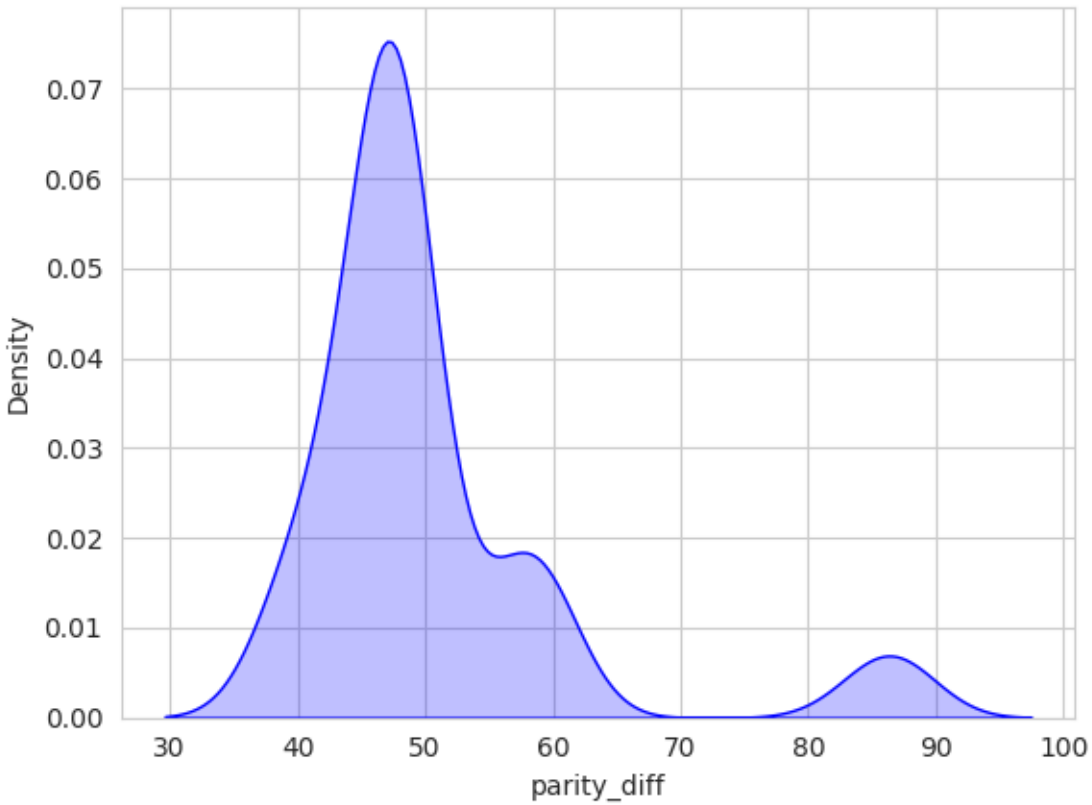
[ ]:
```
def get_N(S, K, T, r, sigma, q):

    d1 = (np.log(S / K) + (r - q + (sigma ** 2) / 2) * T) / (sigma * np.sqrt(T))
    d2 = d1 - sigma * np.sqrt(T)

    return norm.cdf(d1), norm.cdf(d2), norm.cdf(-d1), norm.cdf(-d2)
```

[ ]:
```
put_call['Nd1'], put_call['Nd2'], put_call['N_d1'], put_call['N_d2'] =␣
 ↪get_N(put_call['index_price'],\
                                                                    ␣
 ↪put_call['strike'], \
                                                                    ␣
 ↪put_call['time_to_maturity'], \
                                                                    ␣
 ↪put_call['int_rate'], \
                                                                    ␣
 ↪put_call['volatility'], \
                                                                    ␣
 ↪put_call['dividend_yield']
                                                                     )
```

[ ]:
```
import math
put_call['parity_diff'] = put_call['price_put'] - put_call['price_call'] -␣
 ↪put_call['strike'] * math.e**(-put_call['int_rate'] *␣
 ↪put_call['time_to_maturity']) + put_call['index_price']
```

[ ]:
```
import seaborn as sns

data = put_call['parity_diff']

# Create a KDE plot
sns.set_style('whitegrid')
sns.kdeplot(data, shade=True, color='blue')

# Show the plot
plt.show()
```

```
<ipython-input-233-c3dda9540212>:7: FutureWarning:


`shade` is now deprecated in favor of `fill`; setting `fill=True`.

This will become an error in seaborn v0.14.0; please update your code.


  sns.kdeplot(data, shade=True, color='blue')
```



Market data does not coincide with put call parity and I do not think there is mispricing.

Here are my reasons:

- model bias: assumed int rate and volatility constant during life of expiration; assumed SPX return is normally distributed
- data bias: option contracts for some term and strike are illquid and their prices are stale
- behavior bias: people are risk-averse and tend to buy put to protect downside market movement, causing OTM put is more expensive

2 Bisection Method vs Newton Method

As I used newton method to get implied volatility, I am curious about newton method vs bisection method to find root for equations

2.1 a.1) bisection method

psuedo code

[ ]:
```
# get_bisection_root(equation, x1, x2, interval_threhold, max_iteration):

#   if equation(x1) * equation(x2) > 0:
#     print('change interval please')
#     break
#   while max_iteration and abs(x2 - x1) > interval_threhold:
#     if equation((x1 + x2)/2) * equation(x1) > 0:
#       x1 = (x1 + x2)/2
#   return x1
```

initial search interval is $[a, \frac{1}{a}]$ or $[\frac{1}{a}, a]$

2.2 a.2) newton method

psuedo code

[ ]:
```
# get_newton_root(x0, interval_threhold, max_iteration)
#   while max_iteration:
#     x1 = x0/2 + a/2/x0
#     while abs(x1 - x0) <= interval_threhold:
#       return x1
#   print('failed to converge!')
```

start searching for root from mid point of interval $[a, \frac{1}{a}]$: $\frac{a^2+1}{2a}$

2.3 b.1) implement bisection method to find root

[22]:
```
import numpy as np
a = 0.5
iteration_ls = []
error_ls = []
def equation(x):
  return x ** 2 - a

def get_bisection_root(equation, x1, x2, interval_threhold, max_iteration):
  i = 0
  if equation(x1) * equation(x2) > 0:
    print('change interval please')
    return
  while i < max_iteration and abs(x2 - x1) > interval_threhold:
    i += 1
    if equation((x1 + x2)/2) * equation(x1) > 0:
      x1 = (x1 + x2)/2
    else:
      x2 = (x1 + x2)/2
    iteration_ls.append(i)
    error_ls.append(abs(np.sqrt(a) - (x1 + x2)/2))
  return x1, i, abs(np.sqrt(a) - (x1 + x2)/2)
```

[23]:
```
get_bisection_root(equation, a, 1/a, 10**(-6), 100)
```

[23]:
```
(0.7071065902709961, 21, 1.6671231717335644e-07)
```

[24]:
```
df_bisection_half = pd.DataFrame({'iteration': iteration_ls,  'log_error': np.
 ↪log(error_ls)})
```

[25]:
```
a = 2
iteration_ls = []
error_ls = []
def equation(x):
  return x ** 2 - a

def get_bisection_root(equation, x1, x2, interval_threhold, max_iteration):
  i = 0
  if equation(x1) * equation(x2) > 0:
    print('change interval please')
    return
  while i < max_iteration and abs(x2 - x1) > interval_threhold:
    i += 1
    if equation((x1 + x2)/2) * equation(x1) > 0:
      x1 = (x1 + x2)/2
    else:
```

```
      x2 = (x1 + x2)/2
    iteration_ls.append(i)
    error_ls.append(abs(np.sqrt(a) - (x1 + x2)/2))
  return x1, i, abs(np.sqrt(a) - (x1 + x2)/2)
```

[26]:
```
get_bisection_root(equation, 1/a, a, 10**(-6), 100)
```

[26]:
```
(1.4142134189605713, 21, 2.1421534479593163e-07)
```

[27]:
```
df_bisection_two = pd.DataFrame({'iteration': iteration_ls,  'log_error': np.
 ↪log(error_ls)})
```

2.4 b.2) implement newton method to find root

[54]:
```
def get_newton_root(x0, interval_threhold, max_iteration):
  i = 0
  iteration_ls = []
  error_ls = []
  while i < max_iteration:
    i += 1
    temp = x0
    x0 = x0/2 + a/2/x0
    iteration_ls.append(i)
    error_ls.append(abs(np.sqrt(a) -  x0))
    while abs(temp - x0) <= interval_threhold:
      return x0, i, iteration_ls, error_ls
  print('failed to converge!')
```

[60]:
```
a = 2
x, i, iteration_ls, error_ls = get_newton_root((a ** 2 + 1)/2/a, 10**(-6), 100)
x, i
```

[60]:
```
(1.414213562373095, 4)
```

[61]:
```
df_newton_two = pd.DataFrame({'iteration': iteration_ls,  'log_error': np.
 ↪log(error_ls)})
```

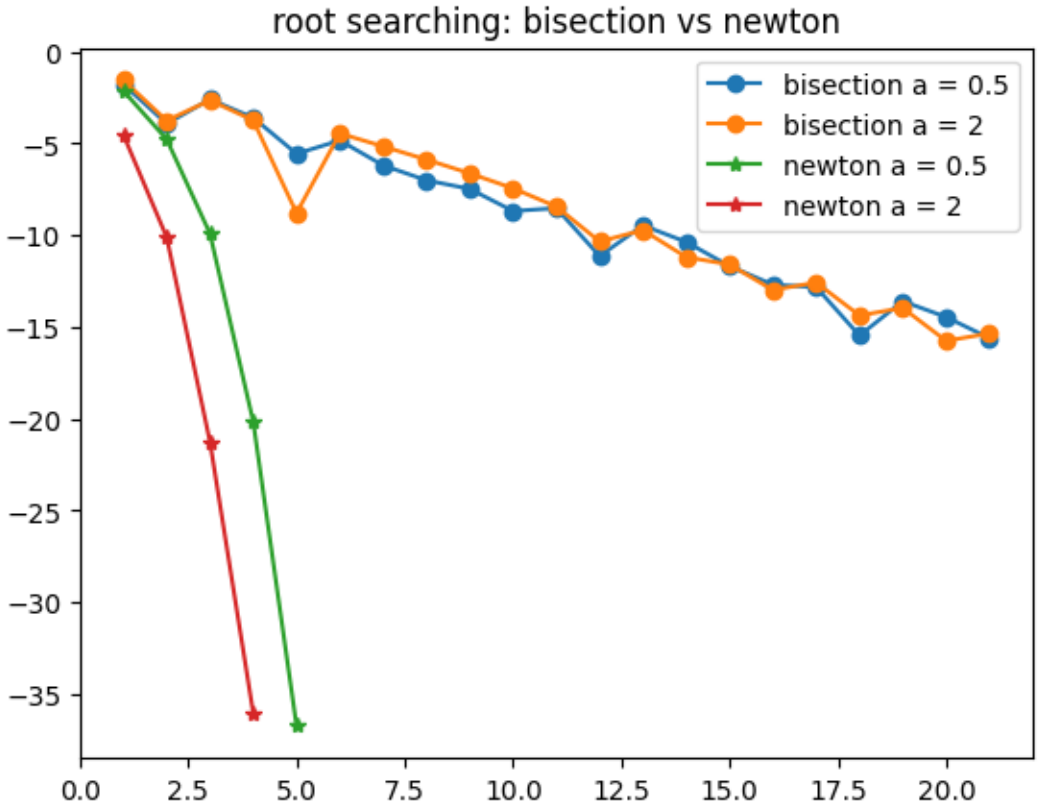
[63]:
```
a = 1/2
x, i, iteration_ls, error_ls = get_newton_root((a ** 2 + 1)/2/a, 10**(-6), 100)
x, i
```

[63]:
```
(0.7071067811865475, 5)
```

[64]:
```
df_newton_half = pd.DataFrame({'iteration': iteration_ls,  'log_error': np.
 ↪log(error_ls)})
```

[70]:
```
plt.plot(df_bisection_half['iteration'], df_bisection_half['log_error'], label␣
↪= 'bisection a = 0.5', marker = 'o')
plt.plot(df_bisection_two['iteration'], df_bisection_two['log_error'], label =␣
↪'bisection a = 2', marker = 'o')
plt.plot(df_newton_half['iteration'], df_newton_half['log_error'], label =␣
↪'newton a = 0.5', marker = '*')
plt.plot(df_newton_two['iteration'], df_newton_two['log_error'], label =␣
↪'newton a = 2', marker = '*')
plt.title('root searching: bisection vs newton')
plt.legend()
```

[70]:
```
<matplotlib.legend.Legend at 0x7be3ceb076a0>
```



Newton method converges faster

Newton method converges faster but it does not guarantee convergence

Bisection method converges slower but it could gurantee convergence