Technical Appendix: Group A

ECON 611/588

By: Jack R, Ammaar M, Colton L, Diego J, Prince K, and Andy H

Technical Appendix Tartigrade Group A Planning

Python Code:

Function: The Prototype will consist of individual codes that will be put together. Each of these individual codes will determine a relationship between Tartigrade's net/gross profits, number of trades and market variables assigned to the individual. Once together, the prototype will take all these findings into account and show what happens to net/gross profits and contracts when any of the market variables shifts. This will not just be cumulative effects of the market variables as they impact each other which requires for the prototype to account for indirect effects on these target variables. For example, interest rates, while not correlated with the target variables directly, affect VIX and SPY which have been found to have correlation with the targets. If there is time, this code will be modified to include visualization in the form of a dashboard to ease interpretation for those not familiar with the data.

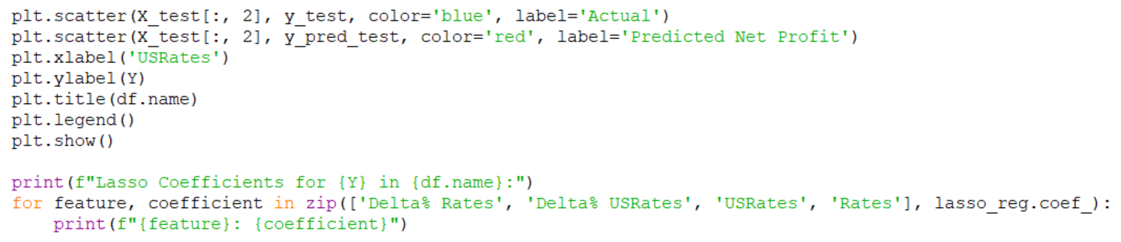
A1: Sample of Python Code which conducts OLS Regression:

```
#OLS Regression
if Regression == 'OLS':
    for df in dataframes:
        for responding in YVars:
            for independent in XVars:
                print('Dataframe: '+df.name)
                print('Responding Variable: '+responding)
                print('Independent Variable: '+independent, end='')
                y=df[responding]
                y.name=responding
                x=df[independent]
                x.name=independent
                x=sm.add_constant(x)
                results=sm.OLS(y,x).fit()
                print(results.summary())
                graph=input('press \'y\' to see graph')
                if graph=='y':

                    predictions=results.predict(x)
                    plt.plot(df[independent],predictions)
                    plt.scatter(df[independent],y)
                    plt.title(df.name)
                    plt.xlabel(independent)
                    plt.ylabel(responding)
                    plt.show()
        NEXTDF=input('hit enter to go to next DF')
```

A2: Sample of Python Code which sets up LASSO Regression

```
elif Regression=='LASSO':
    alpha_values = []
    for i in range(1,201):
        alpha_values.append(float(i)/20)

    for df in dataframes:
        print(df.name)
        df['Date'] = pd.to_datetime(df['Date'], format='%Y-%m-%d')
        for Y in YVars:

            X = df[['Delta% Rates', 'Delta% USRates', 'USRates', 'Rates']]
            y = df[Y]
            #Scaling the data
            scaler = StandardScaler()
            X_scaled = scaler.fit_transform(X)
            # Splitting data into training and test sets
            X_train, X_test, y_train, y_test = train_test_split(X_scaled, y, test_size=0.75, random_state=42
            #Optimizing for alpha parameter for LASSO model
            mse_scores = []

            for alpha in alpha_values:
                lasso_reg = Lasso(alpha=alpha)

                scores = cross_val_score(lasso_reg, X_train, y_train, cv=5, scoring='neg_mean_squared_error')
                mse_scores.append(np.mean(-scores))

            optimal_alpha = alpha_values[np.argmin(mse_scores)]
            print("Optimal alpha:", optimal_alpha)

            lasso_reg = Lasso(alpha=optimal_alpha)
            lasso_reg.fit(X_train, y_train)
```

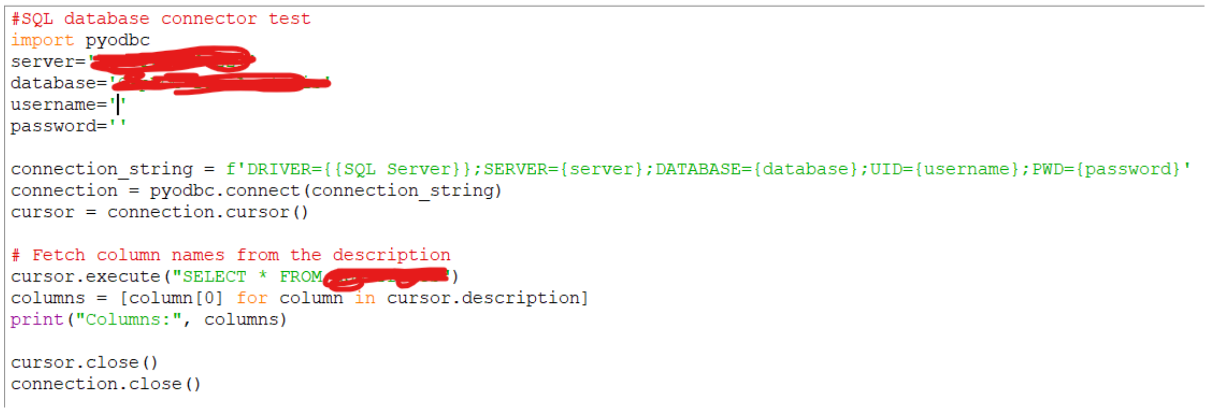
A3: Sample of Python Code which visualizes and prints coefficients from LASSO Regression:



```
plt.scatter(X_test[:, 2], y_test, color='blue', label='Actual')
plt.scatter(X_test[:, 2], y_pred_test, color='red', label='Predicted Net Profit')
plt.xlabel('USRates')
plt.ylabel(Y)
plt.title(df.name)
plt.legend()
plt.show()

print(f"Lasso Coefficients for {Y} in {df.name}:")
for feature, coefficient in zip(['Delta% Rates', 'Delta% USRates', 'USRates', 'Rates'], lasso_reg.coef_):
    print(f"{feature}: {coefficient}")
```

Data: Early versions of this prototype will accept .csv files of data which will be merged on the date/time of the trades. Because the data needs to be merged, significant parts of the code will be dedicated to cleaning it. Later versions will connect directly to Tartigrade's SQL server and to our personalized database which will contain all market variables we will use. This will remove the need for cleaning and merging.

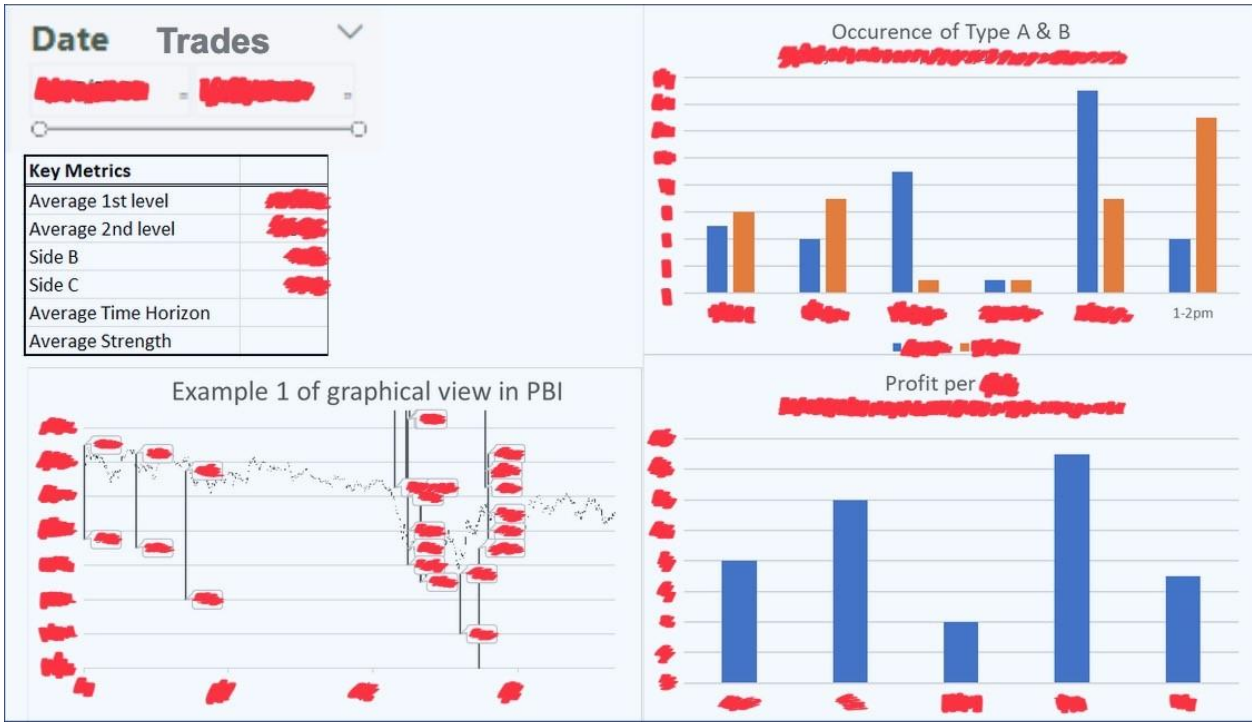
A4: Sample of Python Code which connects to the SQL server and conduct queries in Python



```
#SQL database connector test
import pyodbc
server=[redacted]
database=[redacted]
username=''
password=''

connection_string = f'DRIVER={{SQL Server}};SERVER={server};DATABASE={database};UID={username};PWD={password}'
connection = pyodbc.connect(connection_string)
cursor = connection.cursor()

# Fetch column names from the description
cursor.execute("SELECT * FROM[redacted])
columns = [column[0] for column in cursor.description]
print("Columns:", columns)

cursor.close()
connection.close()
```

Dashboard: The primary deliverable of this project as previously stated consists of a PowerBI or Seaborn and Matplotlib dashboard honed to deliver unknown insights into the behavior of their current trading algorithm/model. Tartigrade has requested said dashboard to be easily configurable so future trades can be added at their convenience.

A5: Sample of Requested Dashboard (PowerBI), (Values hidden per IP protection)



Key Metrics	
Average 1st level	[redacted]
Average 2nd level	[redacted]
Side B	[redacted]
Side C	[redacted]
Average Time Horizon	
Average Strength	

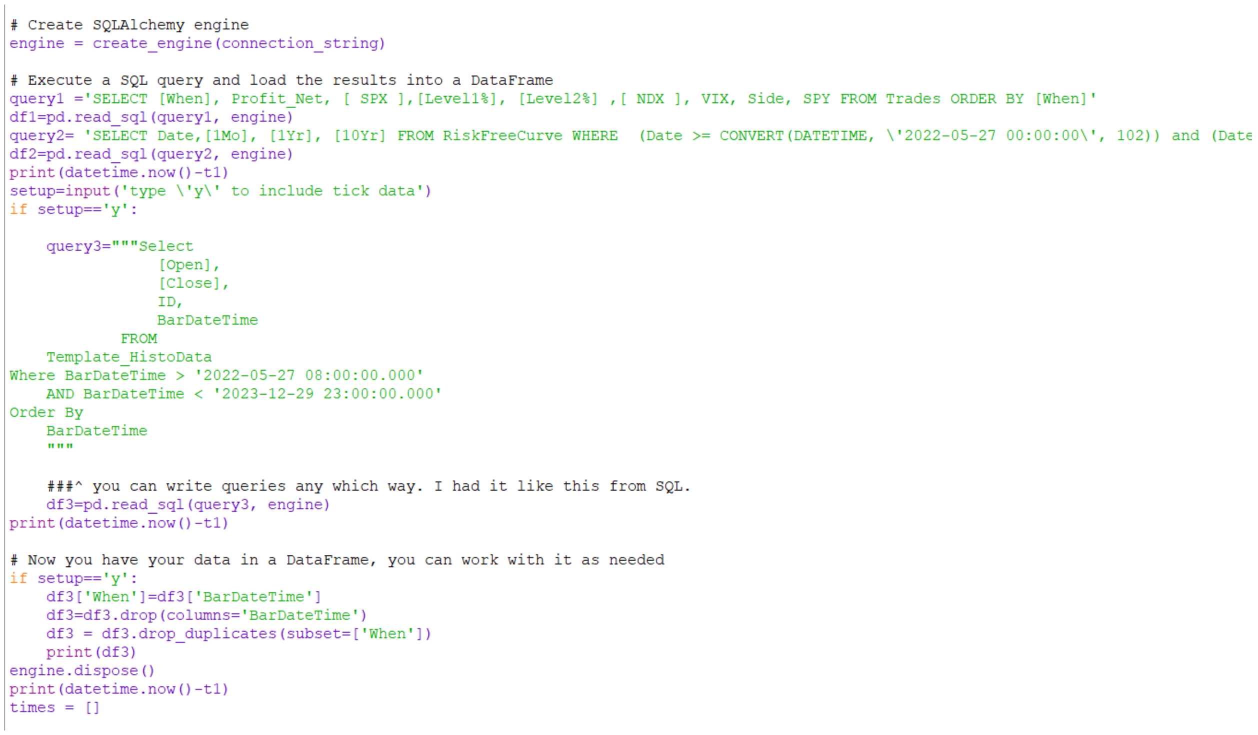
Progress Report 1 Technical Appendix

Group A (Tartigrade)

Colton Code:

New mechanism for connecting to SQL for Python: For the ability to efficiently pull and upload data to SQL, we had to switch methods and install ‘SQLalchemy’ instead of ‘pyodbc’ as seen in the previous appendix.

A1:



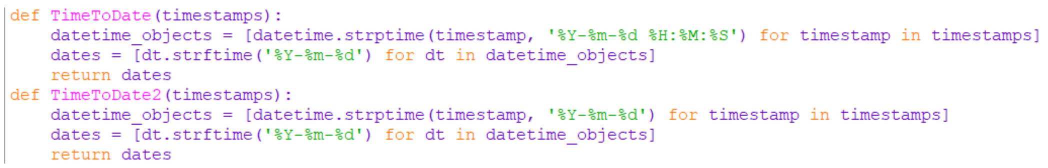
```
# Create SQLAlchemy engine
engine = create_engine(connection_string)

# Execute a SQL query and load the results into a DataFrame
query1 ='SELECT [When], Profit_Net, [ SPX ],[Level1%], [Level2%] ,[ NDX ], VIX, Side, SPY FROM Trades ORDER BY [When]'
df1=pd.read_sql(query1, engine)
query2= 'SELECT Date,[1Mo], [1Yr], [10Yr] FROM RiskFreeCurve WHERE  (Date >= CONVERT(DATETIME, \'2022-05-27 00:00:00\', 102)) and (Date
df2=pd.read_sql(query2, engine)
print(datetime.now()-t1)
setup=input('type \'y\' to include tick data')
if setup=='y':

    query3="""Select
                [Open],
                [Close],
                ID,
                BarDateTime
            FROM
    Template_HistoData
Where BarDateTime > '2022-05-27 08:00:00.000'
    AND BarDateTime < '2023-12-29 23:00:00.000'
Order By
    BarDateTime
    """

    ###^ you can write queries any which way. I had it like this from SQL.
    df3=pd.read_sql(query3, engine)
print(datetime.now()-t1)

# Now you have your data in a DataFrame, you can work with it as needed
if setup=='y':
    df3['When']=df3['BarDateTime']
    df3=df3.drop(columns='BarDateTime')
    df3 = df3.drop_duplicates(subset=['When'])
    print(df3)
engine.dispose()
print(datetime.now()-t1)
times = []
```

Pseudocode:

```
1. Create connection string (server, database, username, password)
2. Create engine using the connection string
3. Create your queries written in SQL and made into a string specifying what data you want
   from which data base
3a. Select if you wish to import data in the 5s form (this takes a while and using the smaller data
is still helpful for finding relationships. Additionally, analysis is skipped for this data. Selecting
this option changes how the code functions)
4. Put queries into pandas data frames using the engine
```

Various Cleaning Techniques: The data coming from SQL often had missing values. However, these were not always in the form of NaN. Sometimes the values were filled but wrongly so. Many different approaches to cleaning were needed to make the data ready for analysis. Dropping rows with a missing value was to be avoided if possible. All cleaning techniques were on the smaller data frame. When merging the smaller data frame with the 5s data, all data was forward filled as that data frame had NaN values which are easily fixed with this method.

A2:



```
def TimeToDate(timestamps):
    datetime_objects = [datetime.strptime(timestamp, '%Y-%m-%d %H:%M:%S') for timestamp in timestamps]
    dates = [dt.strftime('%Y-%m-%d') for dt in datetime_objects]
    return dates
def TimeToDate2(timestamps):
    datetime_objects = [datetime.strptime(timestamp, '%Y-%m-%d') for timestamp in timestamps]
    dates = [dt.strftime('%Y-%m-%d') for dt in datetime_objects]
    return dates
```

Pseudocode Date Cleaning (Dates from each data base on the server has dates in different formats):

```
1. Create a list of all times in the first data frame
2. Feed list to TimeToDate function
3. Function converts times to desired format
4. Function returns a list of times/dates
5. Repeat steps with the 2nd data frame instead feeding it in TimeToDate2
```

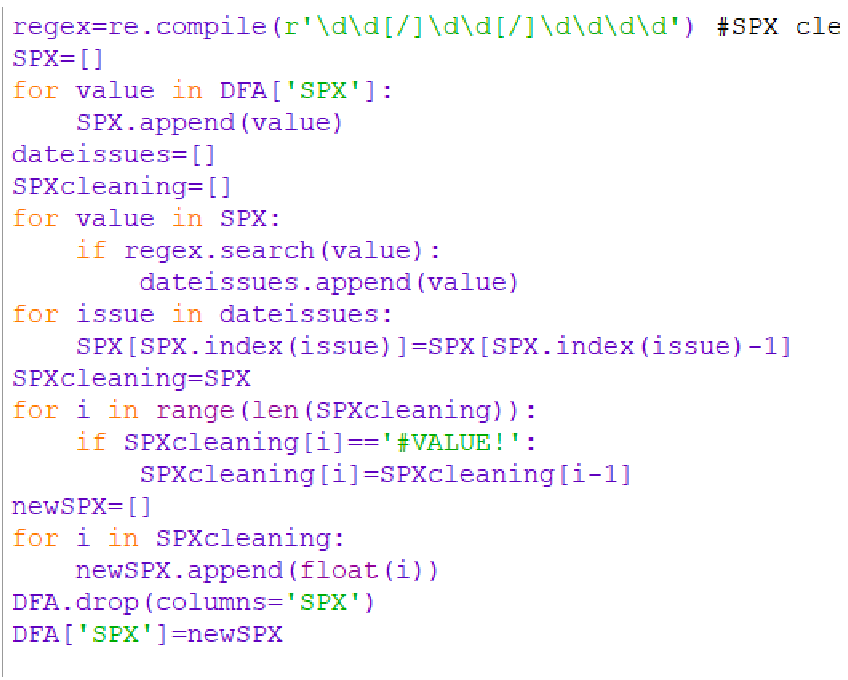
A3:

```
sides=[]
for i in DFA.Side:
    sides.append(i)
adjsides=[]
for i in sides:
    if i =='A  ':
        adjsides.append('A')
    elif i =='B  ':
        adjsides.append('B')
DFA=DFA.drop(columns='Side')
DFA['Side']=adjsides


columns=['1Mo','1Yr','10Yr']
for column in columns:
    DFA[column].ffill(inplace=True)
    #DFA[column].bfill(inplace=True)

mktvars=['VIX','NDX'] #Easy stuff to work with
for var in mktvars:
    mktvarcleaning = []
    for value in DFA[var]:
        mktvarcleaning.append(value)

    for i in range(len(mktvarcleaning)):
        if mktvarcleaning[i] == '#VALUE!':
            mktvarcleaning[i] = mktvarcleaning[i - 1]

    newmktvar = []
    for i in mktvarcleaning:
        newmktvar.append(float(i.rstrip('%')))
    DFA=DFA.drop(columns=var)
    DFA[var]=newmktvar
```

Pseudocode: Side and VIX/NDX Cleaning (two spaces were attached to each side value. #VALUE! appeared in a few rows. These are not recognized as NaN values.)

1. Put merged data frame columns into list
2. Create new list for new data frame column
3. Cleans by creating new ‘A’/’B’ values for side or replaces the ‘#VALUE!’ with the previous valid value (this works even with the 5s data where often ‘Side’ takes no value as no trade happened.)
4. Previous data frame column is deleted
5. Re-create the data frame column with list of corrected values while changing their type to ‘float’
6. (interest rates) forward fill the data to remove null values (bfill was initially included afterwards as null values were initially at the start of the dataset (this is done later for the larger data set so all variables can be included in a ‘for’ loop)

A4:



```
regex=re.compile(r'\d\d[/]\d\d[/]\d\d\d\d') #SPX cle
SPX=[]
for value in DFA['SPX']:
    SPX.append(value)
dateissues=[]
SPXcleaning=[]
for value in SPX:
    if regex.search(value):
        dateissues.append(value)
for issue in dateissues:
    SPX[SPX.index(issue)]=SPX[SPX.index(issue)-1]
SPXcleaning=SPX
for i in range(len(SPXcleaning)):
    if SPXcleaning[i]=='#VALUE!':
        SPXcleaning[i]=SPXcleaning[i-1]
newSPX=[]
for i in SPXcleaning:
    newSPX.append(float(i))
DFA.drop(columns='SPX')
DFA['SPX']=newSPX
```

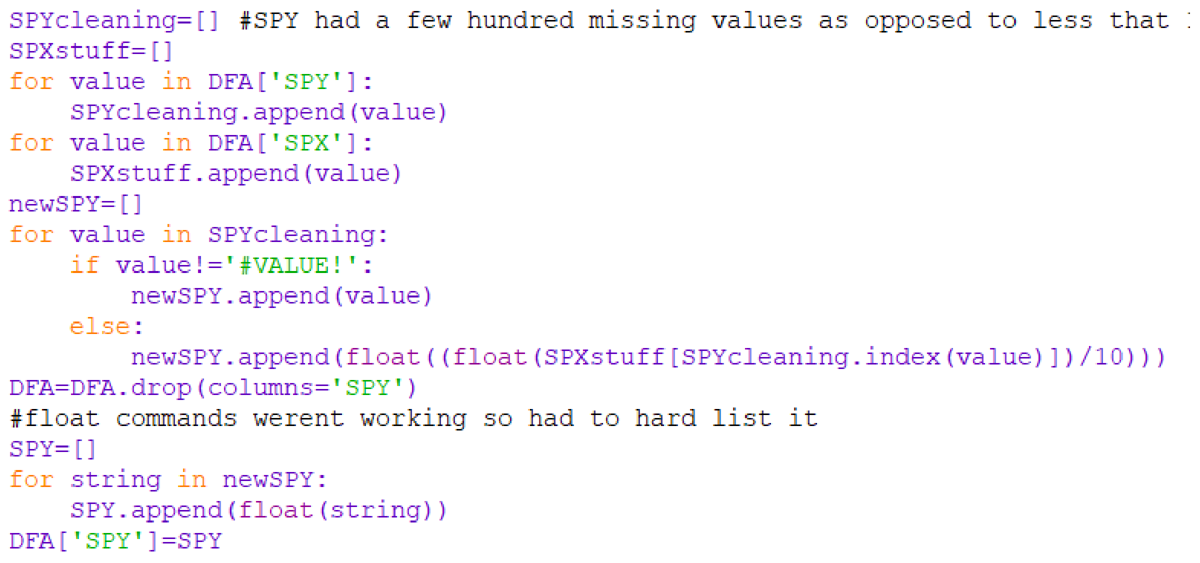
Pseudocode: SPX cleaning (There were dates mixed in with the values along with #VALUE!. These values accounted for only a few observations)

1. Define regular expression (all erroneous entries were in the date format written)
2. Create list and append all SPX data to it
3. Establish a list for numerical and date values
4. Iterate over the SPX list

4a. If a date is detected, it is put into the date list

5. Iterate over all dates in the date list and replace them with the previous valid entry
6. Drop old column
7. Convert all types to float and recreate the column in the data frame

A5:



```
SPYcleaning=[] #SPY had a few hundred missing values as opposed to less that
SPXstuff=[]
for value in DFA['SPY']:
    SPYcleaning.append(value)
for value in DFA['SPX']:
    SPXstuff.append(value)
newSPY=[]
for value in SPYcleaning:
    if value!='#VALUE!':
        newSPY.append(value)
    else:
        newSPY.append(float((float(SPXstuff[SPYcleaning.index(value)])/10)))
DFA=DFA.drop(columns='SPY')
#float commands werent working so had to hard list it
SPY=[]
for string in newSPY:
    SPY.append(float(string))
DFA['SPY']=SPY
```

Pseudocode: SPY Cleaning (SPY had hundreds of missing values amounting to around 10% of what we were given. Instead of filling, had to find another way to make it work)

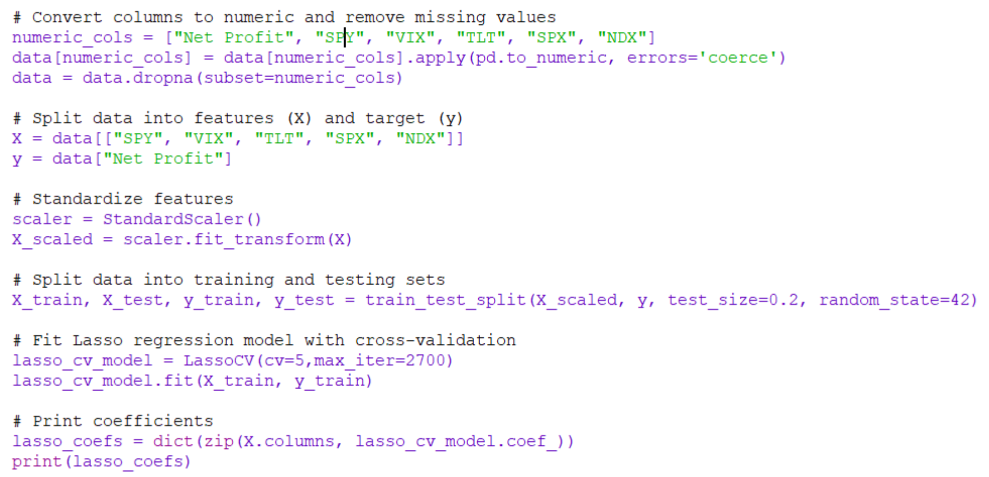
1. Put SPY and SPX values into separate lists
2. Make a new list for the cleaned SPY values
3. Iterate over SPY looking for #VALUE!. Appending to new SPY list.

3a. If issue is found, convert SPX value at the same time to SPY which is usually 10x smaller than SPX

4. Get rid of old SPY column and establish new one

LASSO coefficient usage: LASSO regression was used to find out which variables were most influential on Net Profit and Trading. As we are not creating a predictive model, data was split only into training and test sets. Though the number of variables appears manageable, every additional variable added increases the time it takes to import and export data and increases the likelihood of server errors. We aim to find a balance between accuracy and efficiency.

A6:



```
# Convert columns to numeric and remove missing values
numeric_cols = ["Net Profit", "SPY", "VIX", "TLT", "SPX", "NDX"]
data[numeric_cols] = data[numeric_cols].apply(pd.to_numeric, errors='coerce')
data = data.dropna(subset=numeric_cols)

# Split data into features (X) and target (y)
X = data[["SPY", "VIX", "TLT", "SPX", "NDX"]]
y = data["Net Profit"]

# Standardize features
scaler = StandardScaler()
X_scaled = scaler.fit_transform(X)

# Split data into training and testing sets
X_train, X_test, y_train, y_test = train_test_split(X_scaled, y, test_size=0.2, random_state=42)

# Fit Lasso regression model with cross-validation
lasso_cv_model = LassoCV(cv=5,max_iter=2700)
lasso_cv_model.fit(X_train, y_train)

# Print coefficients
lasso_coefs = dict(zip(X.columns, lasso_cv_model.coef_))
print(lasso_coefs)
```

Pseudocode: Market Variables on Net Profit (Originally done in R before being brought over to Python. Then Modified using LASSOCV from sklearn to produce similar results)

```
1. Split data into X and Y
2. Scale the date
3. Split data into X and Y training/test sets
4. Cross validates with 5 folds (data is large so this is better than 10)
5. Perform LASSO regression
6. Store coefficients in a dictionary and display
```

Interest Rate Selection: If using the smaller data frame, an option will be given to look at the interest rate volatility for a given interest rate. Afterwards, a new table is uploaded to SQL all in one chunk as it is relatively small (~10000 rows)

A7:

```
if setup!='y':
    Restart='yes'
    while Restart=='yes':
        valid=['1Mo','1Yr','10Yr']
        irate=''
        while irate not in valid:
            print(valid)
            irate=input('Please select rate you wish to see always change. Choose from printed list')


        ratelist=[]
        for i in DFA[irate]:
            ratelist.append(float(i))
        intratelist=[]
        for i in range(len(ratelist)):
            i=(ratelist[i]-ratelist[i-1])/ratelist[i-1]
            i*=100
            intratelist.append(i)
        finalratelist=[]
        for i in intratelist:
            if i <= 0:
                i*=-1
            finalratelist.append(i)
        DFA['%Change '+irate]=finalratelist
        print(DFA)

        engine = create_engine(connection_string)
        DFA.to_sql('With Rate Volatility', engine, if_exists='replace', index=False)
        engine.dispose()
```

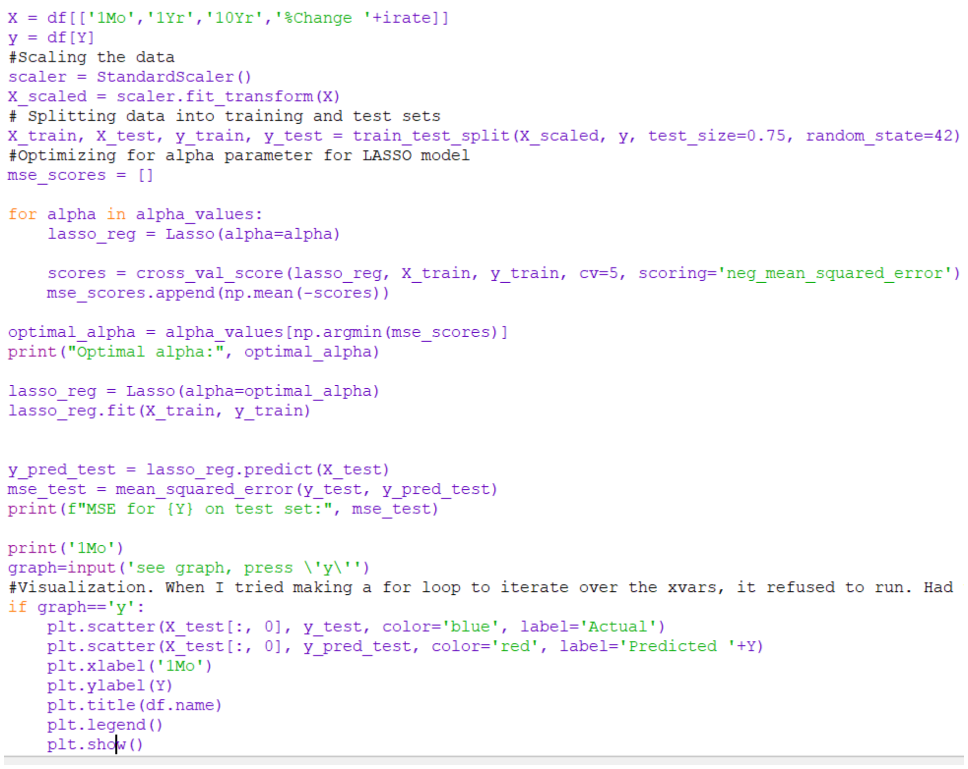
Pseudocode: The Restart variable is there as it is here where a loop starts so that after finishing OLS/LASSO regression on the smaller data frame, the user may restart and select a different interest rate/ regression type.

1. Select which interest rate to show volatility for (short term is better)
2. List created for rate volatility
3. Calculates the change in the interest rate for each observation.

   3a. If negative change occurs, multiplied by –1 (looking for absolute changes)

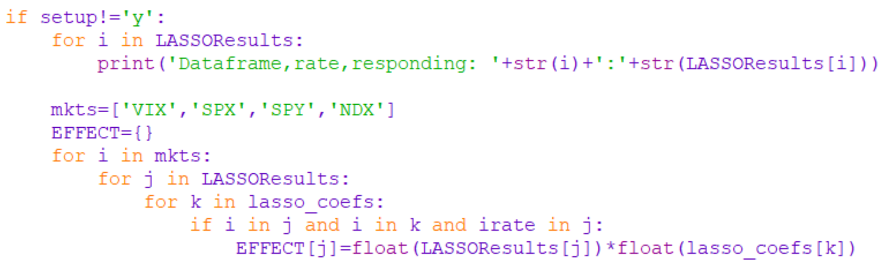
4. Creates new column in data frame using the list
5. Uploads data frame to SQL for visualization

A8:



```
X = df[['1Mo','1Yr','10Yr','%Change '+irate]]
y = df[Y]
#Scaling the data
scaler = StandardScaler()
X_scaled = scaler.fit_transform(X)
# Splitting data into training and test sets
X_train, X_test, y_train, y_test = train_test_split(X_scaled, y, test_size=0.75, random_state=42)
#Optimizing for alpha parameter for LASSO model
mse_scores = []

for alpha in alpha_values:
    lasso_reg = Lasso(alpha=alpha)

    scores = cross_val_score(lasso_reg, X_train, y_train, cv=5, scoring='neg_mean_squared_error')
    mse_scores.append(np.mean(-scores))

optimal_alpha = alpha_values[np.argmin(mse_scores)]
print("Optimal alpha:", optimal_alpha)

lasso_reg = Lasso(alpha=optimal_alpha)
lasso_reg.fit(X_train, y_train)


y_pred_test = lasso_reg.predict(X_test)
mse_test = mean_squared_error(y_test, y_pred_test)
print(f"MSE for {Y} on test set:", mse_test)

print('1Mo')
graph=input('see graph, press \'y\'')
#Visualization. When I tried making a for loop to iterate over the xvars, it refused to run. Had
if graph=='y':
    plt.scatter(X_test[:, 0], y_test, color='blue', label='Actual')
    plt.scatter(X_test[:, 0], y_pred_test, color='red', label='Predicted '+Y)
    plt.xlabel('1Mo')
    plt.ylabel(Y)
    plt.title(df.name)
    plt.legend()
    plt.show()
```

A9:



```
if setup!='y':
    for i in LASSOResults:
        print('Dataframe,rate,responding: '+str(i)+':'+str(LASSOResults[i]))

    mkts=['VIX','SPX','SPY','NDX']
    EFFECT={}
    for i in mkts:
        for j in LASSOResults:
            for k in lasso_coefs:
                if i in j and i in k and irate in j:
                    EFFECT[j]=float(LASSOResults[j])*float(lasso_coefs[k])
```

LASSO for interest rates/rate volatility (Lots is the same except it uses LASSO from sklearn, not LASSOCV.) This iterates over a trading data frame and one that only has times when rates change (to look at rate volatility). It also iterates the X variables. Instead of using only cross validation, it optimizes alpha to minimize MSE in addition to the cross validation. Afterwards, an optional graph may be displayed showing the actual values against the LASSO predicted ones (in slide deck). All coefficients are stored in a list. Then, the coefficients that are not related to volatility are dropped from the coefficients done on the data frame where only rate changes are happening. Afterwards, they are multiplied by the respective market variable for an indirect effect which is then added to a dictionary (in slide deck).

Data frame upload size management: The large data frame with 5s data is extremely large. Uploading cleaned data was extremely troublesome due to the size. A code had to be made to ensure no errors occurred. This was done by splitting the large data frame into chunks and uploading them one at a time. It takes roughly 90 minutes.

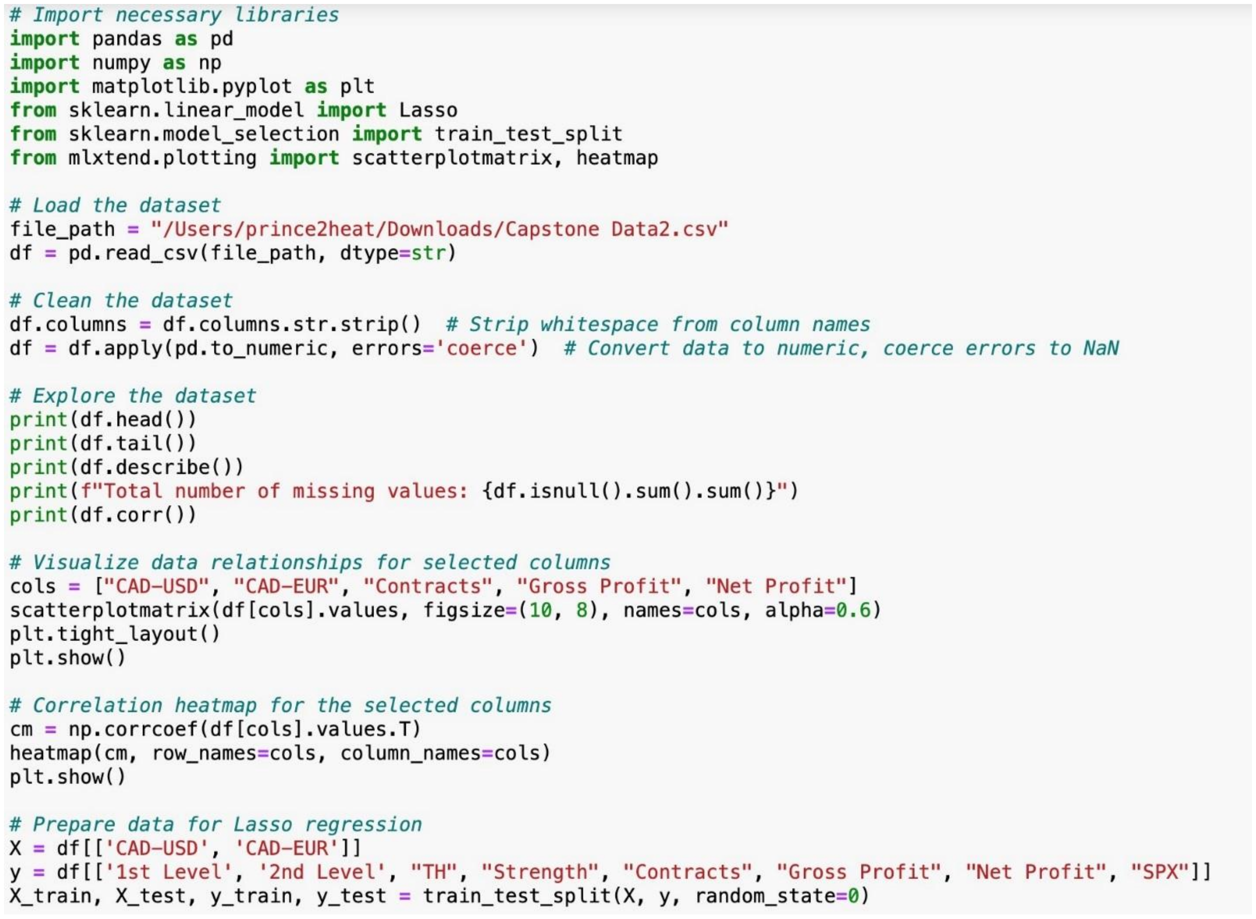
A10:

```
DFB=DFB.drop(columns='ID')
newsql=''
newsql=input('Want to update SQL table?')
if newsql=='y':
    engine = create_engine(connection_string)
    chunks = np.array_split(DFB, 100)
    chunks[0].to_sql('5sDataMerged', engine, if_exists='replace', index=False) #roughly 48,000 rows per chunk


    for i in range(1, 100):
        chunks[i].to_sql('5sDataMerged', engine, if_exists='append', index=False)
        print(datetime.now()-t1)

engine.dispose()
```

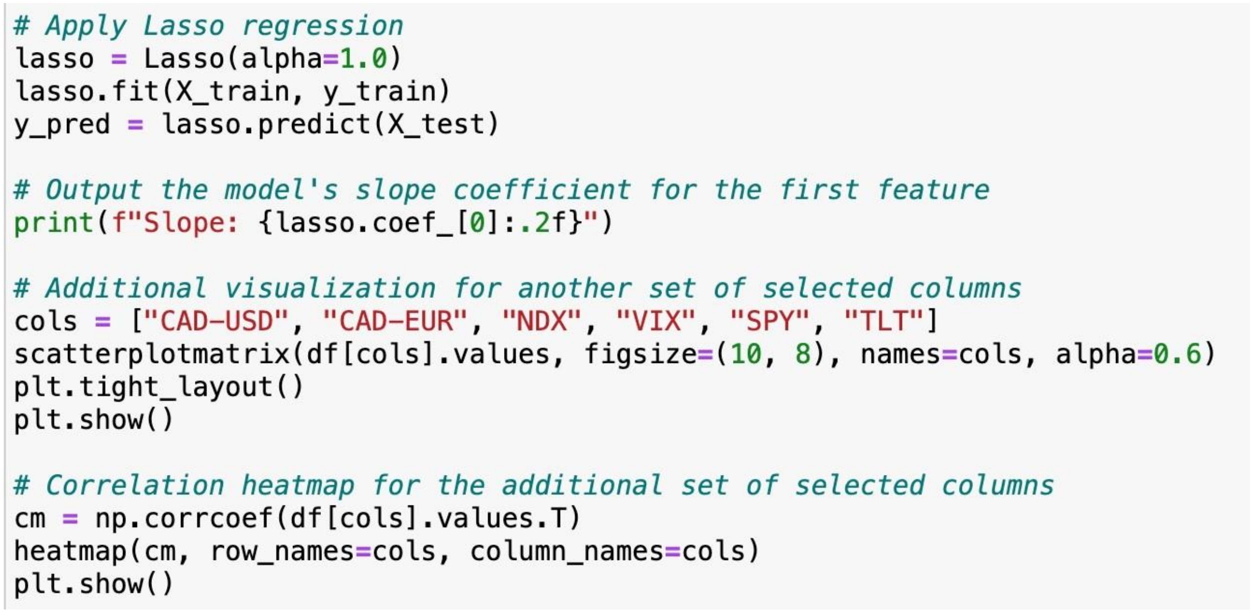
Pseudocode:

1. Drop as many useless columns as possible. Each column dropped saves 3-5 seconds per chunk. (Around 5-8 minutes)
2. Re-connect to SQL server using the same steps as above
3. Split the large data frame into 100 chunks
4. Create/Replace table in SQL with first chunk
5. Iterate over remaining chunks to complete table
6. Close connection to the server to avoid issues when running the code again

Prince:



```python
# Import necessary libraries
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
from sklearn.linear_model import Lasso
from sklearn.model_selection import train_test_split
from mlxtend.plotting import scatterplotmatrix, heatmap

# Load the dataset
file_path = "/Users/prince2heat/Downloads/Capstone Data2.csv"
df = pd.read_csv(file_path, dtype=str)

# Clean the dataset
df.columns = df.columns.str.strip()  # Strip whitespace from column names
df = df.apply(pd.to_numeric, errors='coerce')  # Convert data to numeric, coerce errors to NaN

# Explore the dataset
print(df.head())
print(df.tail())
print(df.describe())
print(f"Total number of missing values: {df.isnull().sum().sum()}")
print(df.corr())

# Visualize data relationships for selected columns
cols = ["CAD-USD", "CAD-EUR", "Contracts", "Gross Profit", "Net Profit"]
scatterplotmatrix(df[cols].values, figsize=(10, 8), names=cols, alpha=0.6)
plt.tight_layout()
plt.show()

# Correlation heatmap for the selected columns
cm = np.corrcoef(df[cols].values.T)
heatmap(cm, row_names=cols, column_names=cols)
plt.show()

# Prepare data for Lasso regression
X = df[['CAD-USD', 'CAD-EUR']]
y = df[['1st Level', '2nd Level', "TH", "Strength", "Contracts", "Gross Profit", "Net Profit", "SPX"]]
X_train, X_test, y_train, y_test = train_test_split(X, y, random_state=0)
```



```python
# Apply Lasso regression
lasso = Lasso(alpha=1.0)
lasso.fit(X_train, y_train)
y_pred = lasso.predict(X_test)

# Output the model's slope coefficient for the first feature
print(f"Slope: {lasso.coef_[0]:.2f}")

# Additional visualization for another set of selected columns
cols = ["CAD-USD", "CAD-EUR", "NDX", "VIX", "SPY", "TLT"]
scatterplotmatrix(df[cols].values, figsize=(10, 8), names=cols, alpha=0.6)
plt.tight_layout()
plt.show()

# Correlation heatmap for the additional set of selected columns
cm = np.corrcoef(df[cols].values.T)
heatmap(cm, row_names=cols, column_names=cols)
plt.show()
```

1. Import necessary libraries:

   - pandas for data manipulation

   - sklearn.linear_model for the Lasso regression model

   - sklearn.model_selection for splitting the dataset

   - sklearn.metrics for calculating the r2 score

   - numpy for numerical operations

   - matplotlib.pyplot and mlxtend.plotting for plotting

2. Load the dataset from a CSV file located at a specific path.

3. Clean the dataset:

   - Strip whitespace from the column names.

   - Convert all data to numeric, coercing errors to NaN (missing values).

4. Explore the dataset:

   - Display the first few rows.

   - Display the last few rows.

   - Generate descriptive statistics.

   - Calculate the total number of missing values.

   - Compute pairwise correlation of columns.

5. Visualize data relationships:

   - Use scatterplot matrix to visualize relationships between selected columns: ["CAD-USD", "CAD-EUR", "Contracts", "Gross Profit", "Net Profit"].

   - Plot a heatmap of the correlation coefficients between these selected columns.

6. Prepare data for Lasso regression:

   - Define the features (X) as ["CAD-USD", "CAD-EUR"] and the target variables (y) as ["1st Level", "2nd Level", "TH", "Strength", "Contracts", "Gross Profit", "Net Profit", "SPX"].

   - Split the data into training and test sets.

7. Apply Lasso regression:

   - Train the Lasso model on the training set.

   - Predict the target variables for the test set.

   - Output the model's slope coefficient for the first feature.

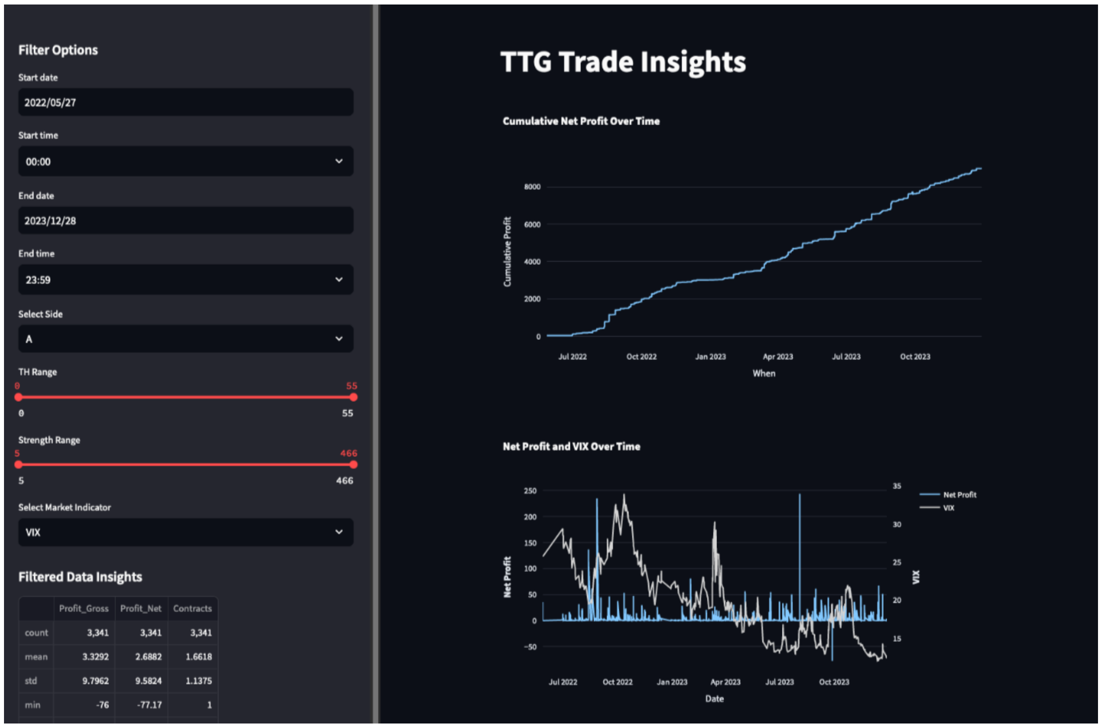
8. Further visualization:

   - Use scatterplot matrix to visualize relationships between another set of selected columns: ["CAD-USD", "CAD-EUR", "NDX", "VIX", "SPY", "TLT"].

   - Plot a heatmap of the correlation coefficients between these selected columns.

Ammaar

Trading Insights GUI:



Filter Options

Start date: 2022/05/27

Start time: 00:00

End date: 2023/12/28

End time: 23:59

Select Side: A

TH Range: 0 – 55

Strength Range: 5 – 466

Select Market Indicator: VIX

Filtered Data Insights

	Profit_Gross	Profit_Net	Contracts
count	3,341	3,341	3,341
mean	3.3292	2.6882	1.6618
std	9.7962	9.5824	1.1375
min	-76	-77.17	1

Visualization Code Samples (Interface done using python STREAMLIT package):

```
# 2. Visualization: Net Profit and Market Indicator Over Time
fig = make_subplots(specs=[[{"secondary_y": True}]])
# Adding Profit_Net trace
fig.add_trace(go.Scatter(x=filtered_data['When'], y=filtered_data['Profit_Net'], name='Net Profit'), secondary_y=False)
# Adding dynamic market data trace based on user selection
fig.add_trace(go.Scatter(x=filtered_data['When'], y=filtered_data[market_indicator], name=market_indicator,
marker_color='lightgrey'), secondary_y=True)
fig.update_layout(title_text=f"Net Profit and {market_indicator} Over Time")
fig.update_xaxes(title_text="Date")
fig.update_yaxes(title_text="<b>Net Profit</b>", secondary_y=False)
fig.update_yaxes(title_text=f"<b>{market_indicator}</b>", secondary_y=True, showgrid=False)
st.plotly_chart(fig)


# 3. Visualization: Profit Efficiency per Contract
filtered_data['Profit Per Contract'] = filtered_data['Profit_Net'] / filtered_data['Contracts']
fig = px.bar(filtered_data.groupby('Contracts')['Profit Per Contract'].mean().reset_index(),
         x='Contracts', y='Profit Per Contract', text='Profit Per Contract')
fig.update_traces(texttemplate='%{text:.2f}', textposition='outside')
fig.update_layout(uniformtext_minsize=8, uniformtext_mode='hide',
              title_text='Average Net Profit per Contract', xaxis_title="Contracts", yaxis_title="Average Profit Per Contract")
st.plotly_chart(fig)
```

Data Pipeline for GUI:

1. Establish connection to SQL Server and run pre-defined queries to select relevant columns from desired table in database
2. Pull data from database into a pandas data frame
3. Clean data frame by handling #VALUEs and #DIV/0, and omitted NaN values where possible
4. Import the pickle library
5. Open a file named 'my_dataframe.pkl' in write-binary ('wb') mode as 'file':
   a. Use the pickle library to dump (write) the Trade_df dataframe into 'file'. This saves the pulled data frame for transferring into the GUI

Building the GUI:

1. Import necessary libraries (streamlit, pickle, plotly, pandas)

2. Load DataFrame 'df' from a pickle file named 'my_dataframe.pkl'

Streamlit App

3. Set up the Streamlit app with a title 'TTG Trade Insights'

4. Create sidebar for filter options:

   a. Add date and time input widgets for start and end datetime selection based on 'When' column in 'df'

   b. Add a select box for choosing a trading side from unique values in 'Side' column

   c. Add sliders for 'TH' and 'Strength' ranges based on their min and max values

   d. Combine date and time inputs to create datetime range filters

5. Filter 'df' based on the sidebar inputs

6. Display statistics of filtered data in the sidebar

7. Visualization 1: Cumulative Net Profit Over Time

   a. Sort filtered data by 'When'

   b. Calculate cumulative profit and create a line plot with Plotly Express

8. Visualization 2: Net Profit and Market Indicator Over Time

   a. Create a subplot with primary and secondary y-axes using Plotly's make_subplots

   b. Add traces for 'Net Profit' and the selected market indicator

   c. Customize layout and axis titles

9. Visualization 3: Profit Efficiency per Contract

   a. Calculate profit per contract

   b. Group by 'Contracts', calculate average profit per contract, and create a bar plot with Plotly Express

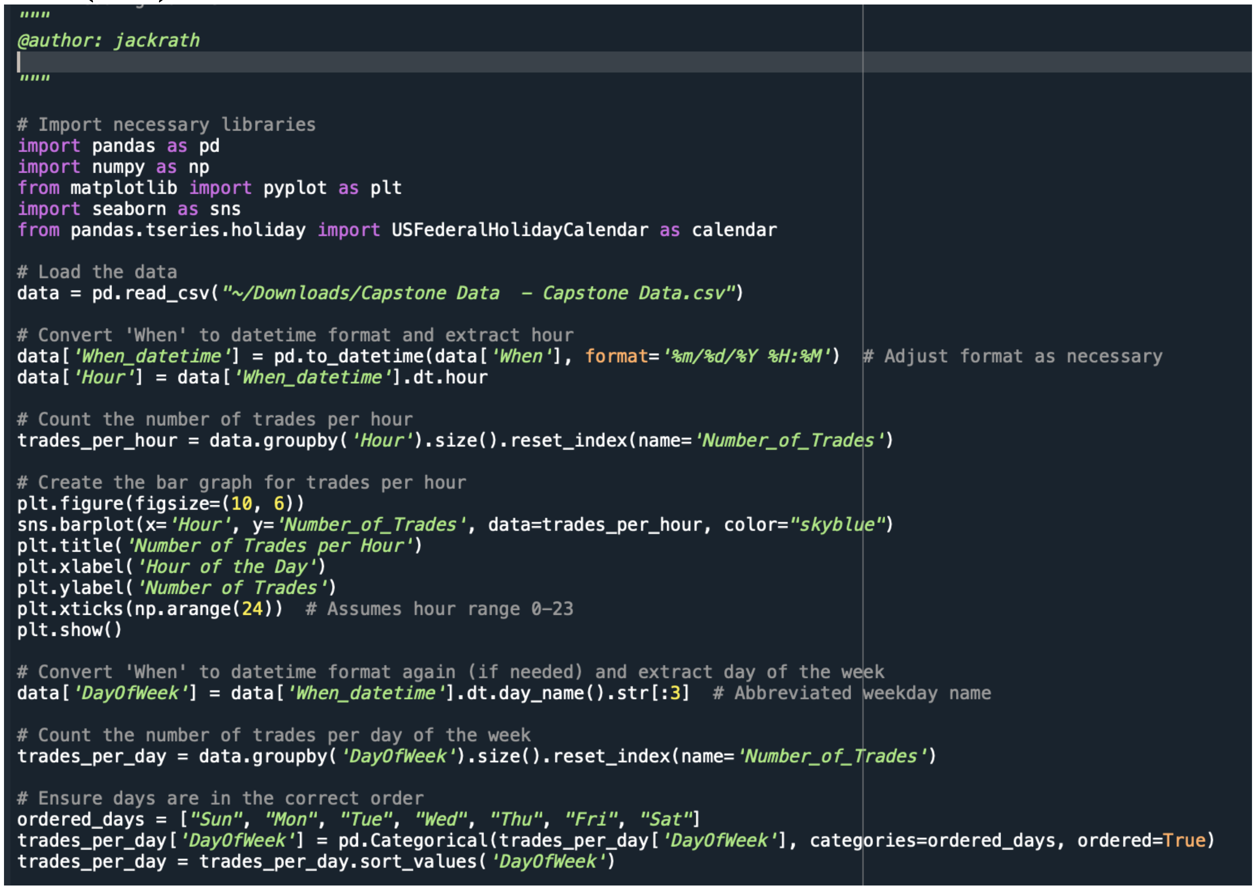
10. Visualization 4: Hourly Profit Trends

   a. Extract hour from 'When', group by 'Hour', calculate average 'Profit_Net', and create a line plot with Plotly Express

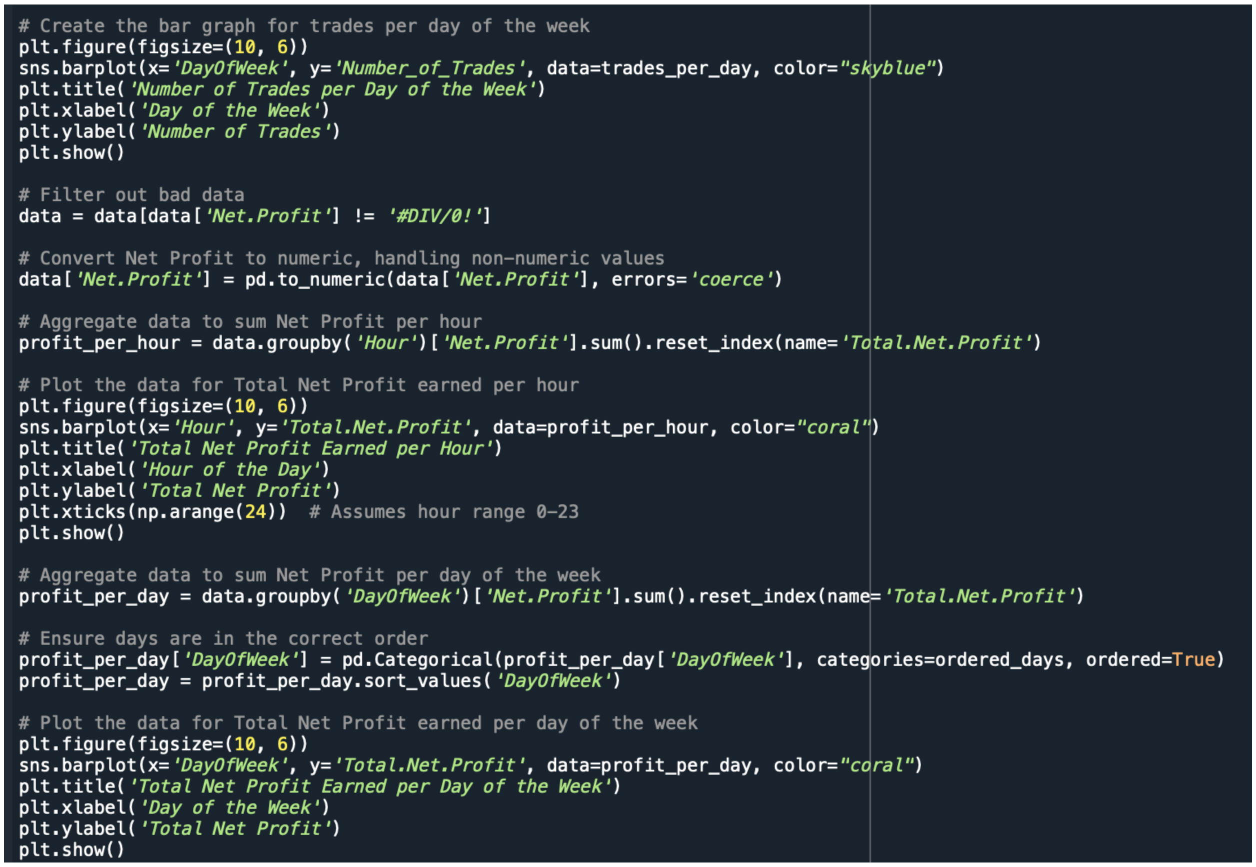
Extension of GUI:

The graphs from the output of the following code will be added into the above GUI. Note that the pseudocode has been added as comments in the code itself.

Code (Jack):

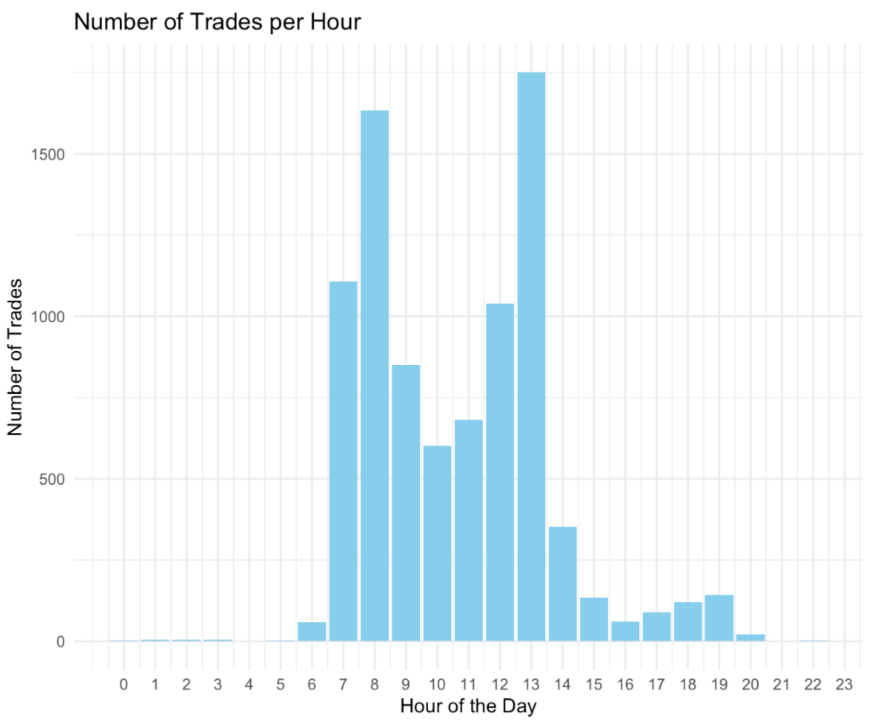


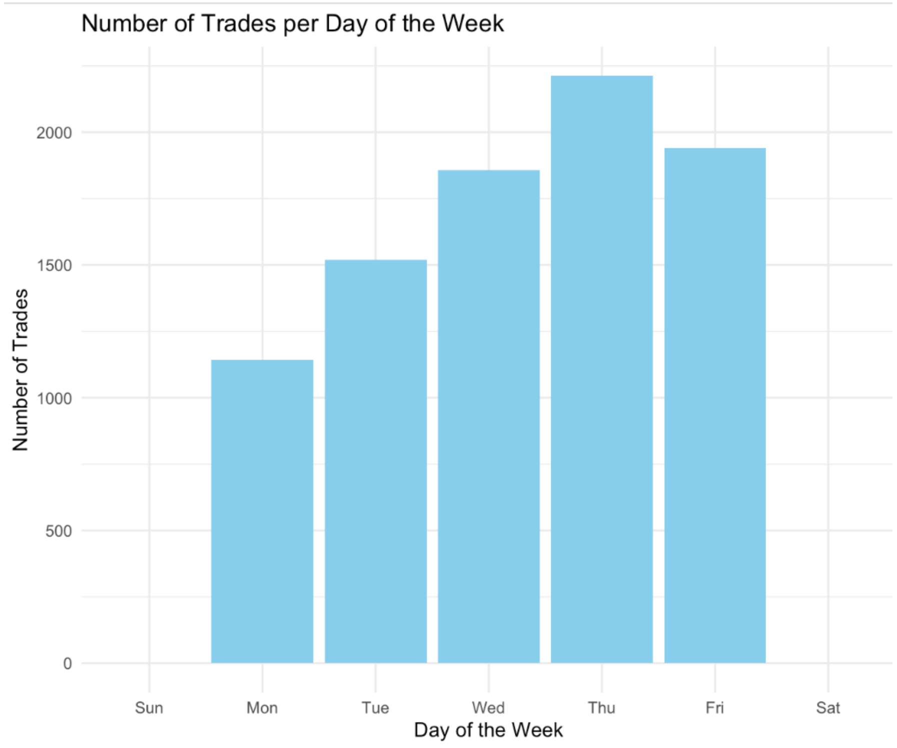
```python
"""
@author: jackrath

"""

# Import necessary libraries
import pandas as pd
import numpy as np
from matplotlib import pyplot as plt
import seaborn as sns
from pandas.tseries.holiday import USFederalHolidayCalendar as calendar

# Load the data
data = pd.read_csv("~/Downloads/Capstone Data  - Capstone Data.csv")

# Convert 'When' to datetime format and extract hour
data['When_datetime'] = pd.to_datetime(data['When'], format='%m/%d/%Y %H:%M')  # Adjust format as necessary
data['Hour'] = data['When_datetime'].dt.hour

# Count the number of trades per hour
trades_per_hour = data.groupby('Hour').size().reset_index(name='Number_of_Trades')

# Create the bar graph for trades per hour
plt.figure(figsize=(10, 6))
sns.barplot(x='Hour', y='Number_of_Trades', data=trades_per_hour, color="skyblue")
plt.title('Number of Trades per Hour')
plt.xlabel('Hour of the Day')
plt.ylabel('Number of Trades')
plt.xticks(np.arange(24))  # Assumes hour range 0-23
plt.show()

# Convert 'When' to datetime format again (if needed) and extract day of the week
data['DayOfWeek'] = data['When_datetime'].dt.day_name().str[:3]  # Abbreviated weekday name

# Count the number of trades per day of the week
trades_per_day = data.groupby('DayOfWeek').size().reset_index(name='Number_of_Trades')

# Ensure days are in the correct order
ordered_days = ["Sun", "Mon", "Tue", "Wed", "Thu", "Fri", "Sat"]
trades_per_day['DayOfWeek'] = pd.Categorical(trades_per_day['DayOfWeek'], categories=ordered_days, ordered=True)
trades_per_day = trades_per_day.sort_values('DayOfWeek')
```

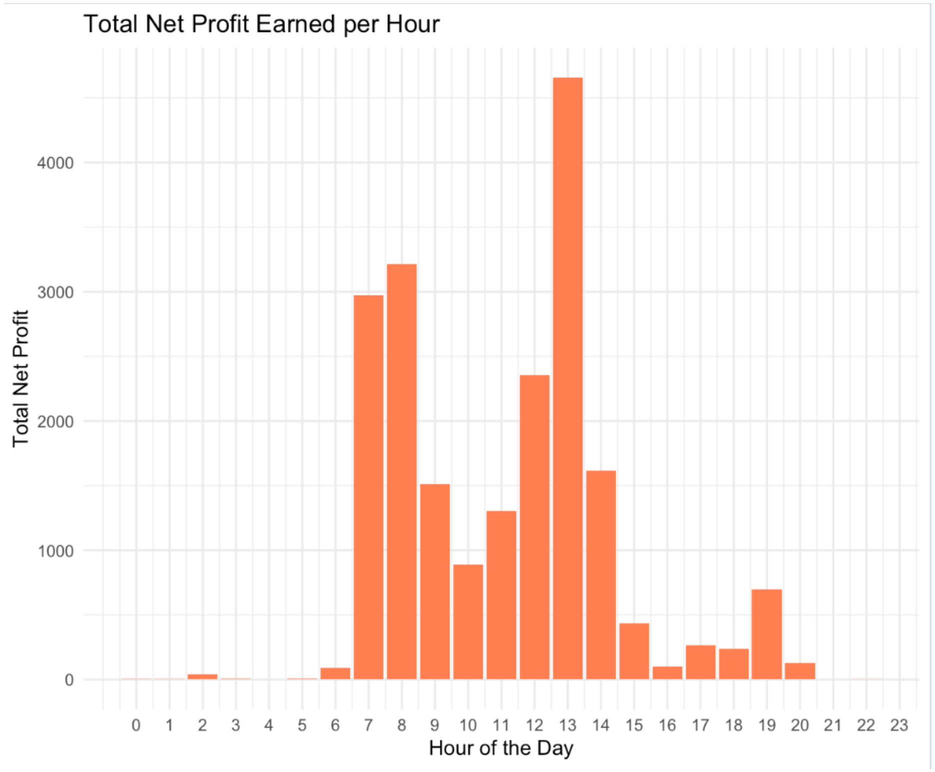


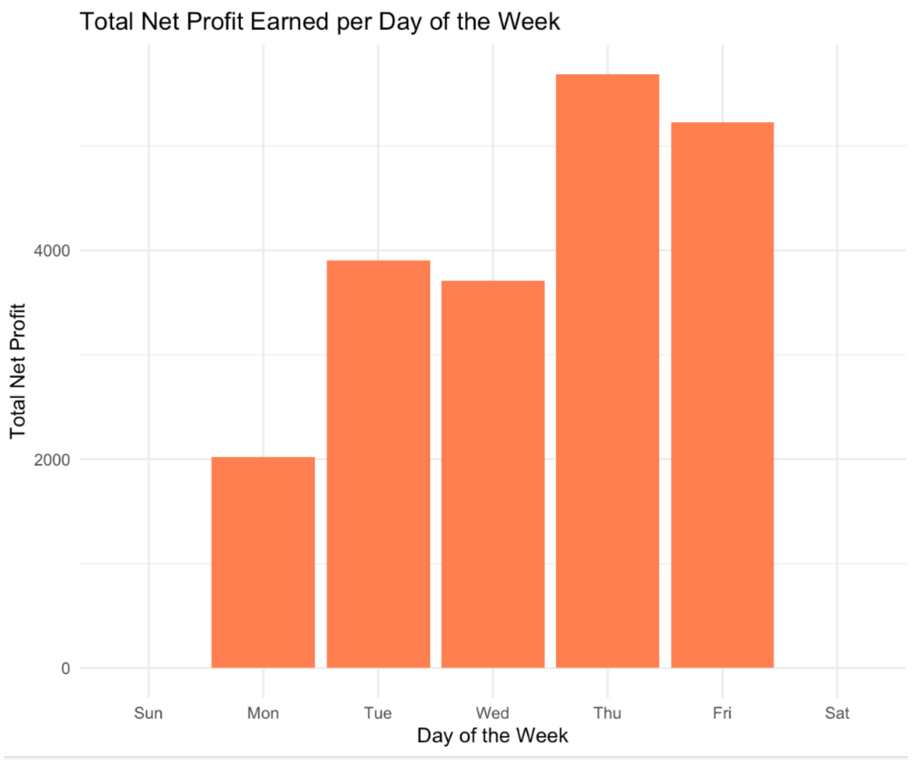
```python
# Create the bar graph for trades per day of the week
plt.figure(figsize=(10, 6))
sns.barplot(x='DayOfWeek', y='Number_of_Trades', data=trades_per_day, color="skyblue")
plt.title('Number of Trades per Day of the Week')
plt.xlabel('Day of the Week')
plt.ylabel('Number of Trades')
plt.show()

# Filter out bad data
data = data[data['Net.Profit'] != '#DIV/0!']

# Convert Net Profit to numeric, handling non-numeric values
data['Net.Profit'] = pd.to_numeric(data['Net.Profit'], errors='coerce')

# Aggregate data to sum Net Profit per hour
profit_per_hour = data.groupby('Hour')['Net.Profit'].sum().reset_index(name='Total.Net.Profit')

# Plot the data for Total Net Profit earned per hour
plt.figure(figsize=(10, 6))
sns.barplot(x='Hour', y='Total.Net.Profit', data=profit_per_hour, color="coral")
plt.title('Total Net Profit Earned per Hour')
plt.xlabel('Hour of the Day')
plt.ylabel('Total Net Profit')
plt.xticks(np.arange(24))  # Assumes hour range 0-23
plt.show()

# Aggregate data to sum Net Profit per day of the week
profit_per_day = data.groupby('DayOfWeek')['Net.Profit'].sum().reset_index(name='Total.Net.Profit')

# Ensure days are in the correct order
profit_per_day['DayOfWeek'] = pd.Categorical(profit_per_day['DayOfWeek'], categories=ordered_days, ordered=True)
profit_per_day = profit_per_day.sort_values('DayOfWeek')

# Plot the data for Total Net Profit earned per day of the week
plt.figure(figsize=(10, 6))
sns.barplot(x='DayOfWeek', y='Total.Net.Profit', data=profit_per_day, color="coral")
plt.title('Total Net Profit Earned per Day of the Week')
plt.xlabel('Day of the Week')
plt.ylabel('Total Net Profit')
plt.show()
```

Output:

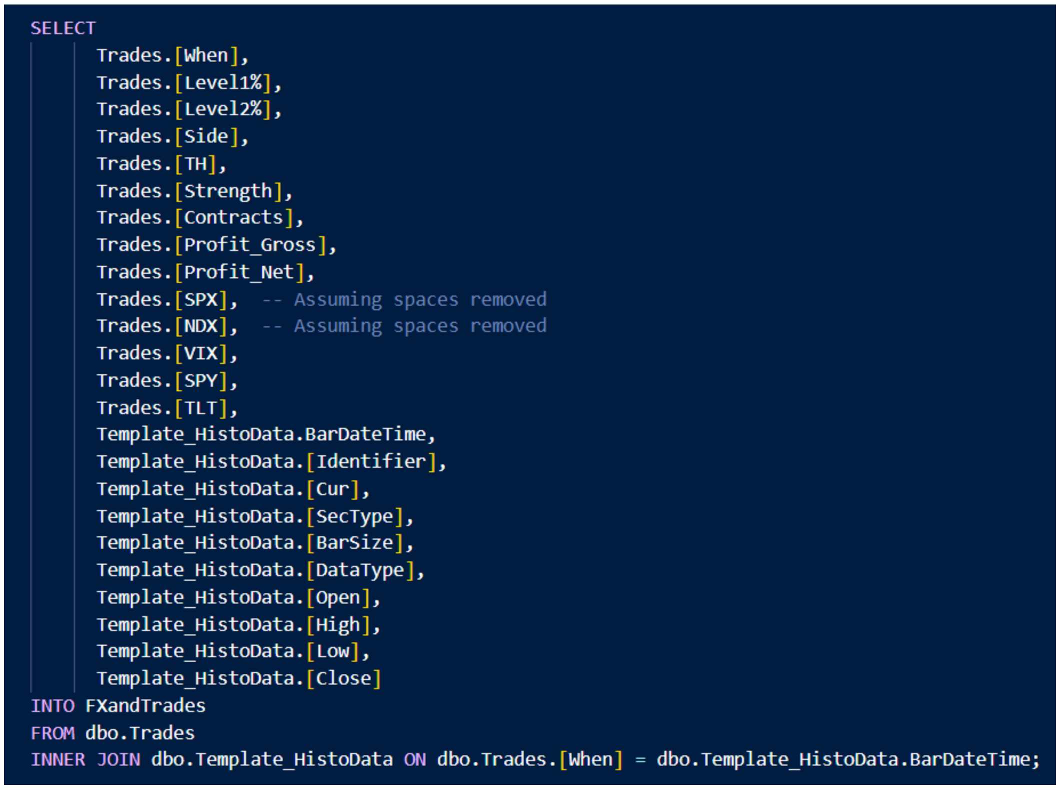



Number of Trades per Day of the Week
Number of Trades
2000
1500
1000
500
0
Sun
Mon
Tue
Wed
Thu
Fri
Sat
Day of the Week


Total Net Profit Earned per Hour
Total Net Profit
4000
3000
2000
1000
0
0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23
Hour of the Day



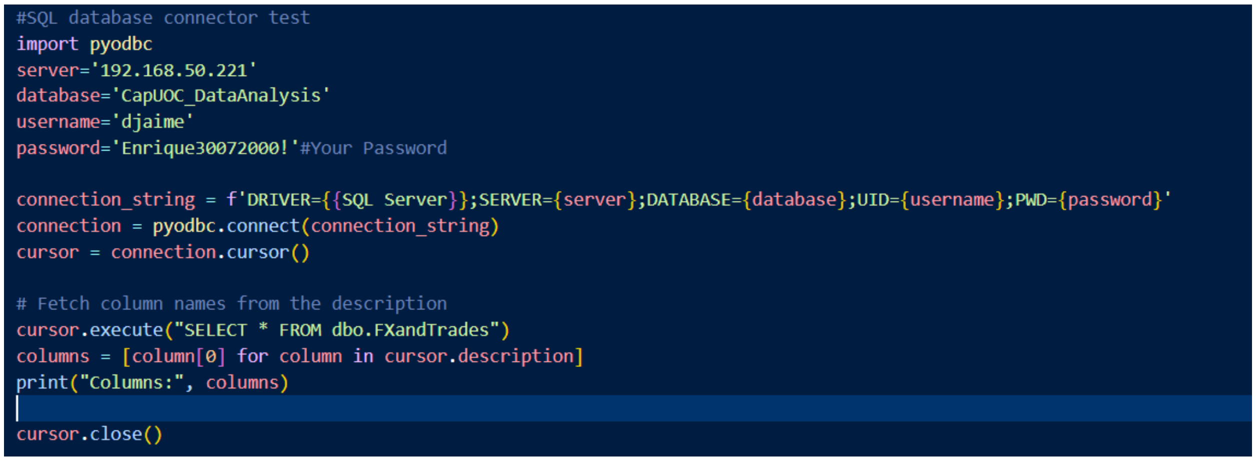
Diego: Join Partner provided Tables in SQL, run Lasso on net profit and other market variables including international market trading sessions:



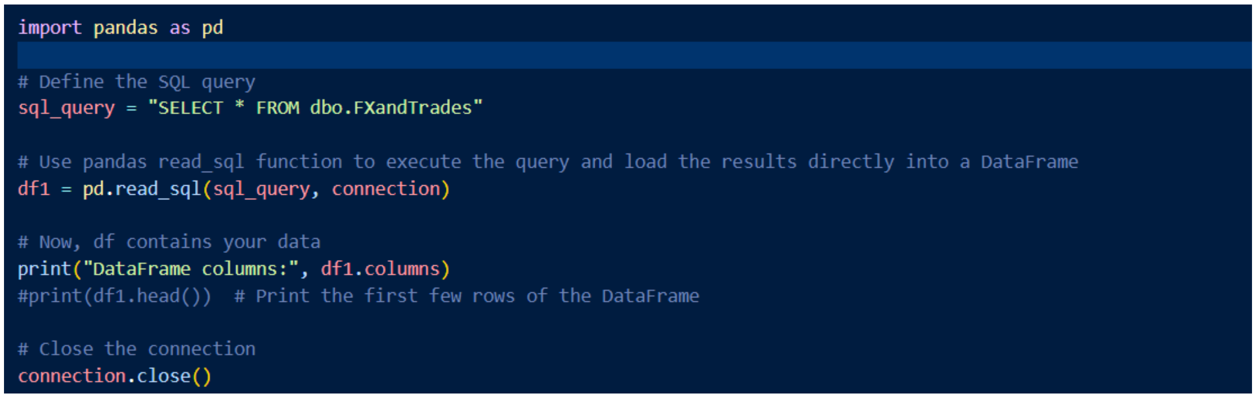
```
SELECT
      Trades.[When],
      Trades.[Level1%],
      Trades.[Level2%],
      Trades.[Side],
      Trades.[TH],
      Trades.[Strength],
      Trades.[Contracts],
      Trades.[Profit_Gross],
      Trades.[Profit_Net],
      Trades.[SPX],  -- Assuming spaces removed
      Trades.[NDX],  -- Assuming spaces removed
      Trades.[VIX],
      Trades.[SPY],
      Trades.[TLT],
      Template_HistoData.BarDateTime,
      Template_HistoData.[Identifier],
      Template_HistoData.[Cur],
      Template_HistoData.[SecType],
      Template_HistoData.[BarSize],
      Template_HistoData.[DataType],
      Template_HistoData.[Open],
      Template_HistoData.[High],
      Template_HistoData.[Low],
      Template_HistoData.[Close]
INTO FXandTrades
FROM dbo.Trades
INNER JOIN dbo.Template_HistoData ON dbo.Trades.[When] = dbo.Template_HistoData.BarDateTime;
```

The code above simply does an inner merge (or "join" in SQL terminology) by matching the column "When" with the column "BarDateTime".

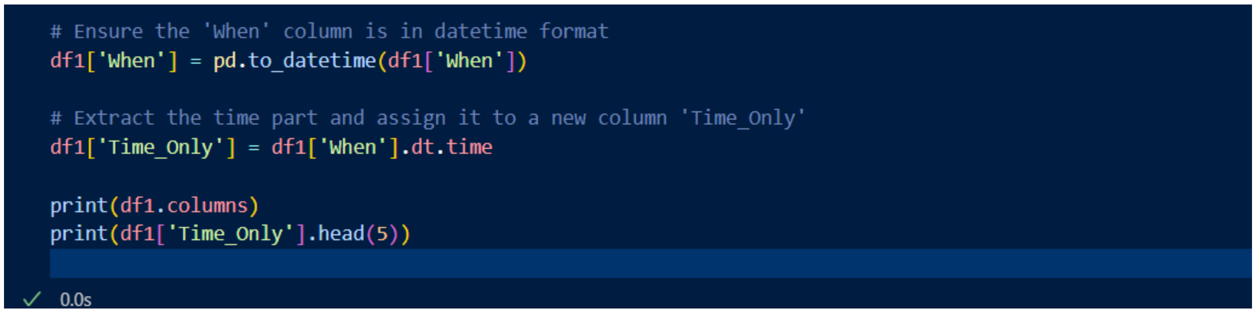
Python Code:



```python
#SQL database connector test
import pyodbc
server='192.168.50.221'
database='CapUOC_DataAnalysis'
username='djaime'
password='Enrique30072000!'#Your Password

connection_string = f'DRIVER={{SQL Server}};SERVER={server};DATABASE={database};UID={username};PWD={password}'
connection = pyodbc.connect(connection_string)
cursor = connection.cursor()

# Fetch column names from the description
cursor.execute("SELECT * FROM dbo.FXandTrades")
columns = [column[0] for column in cursor.description]
print("Columns:", columns)

cursor.close()
```

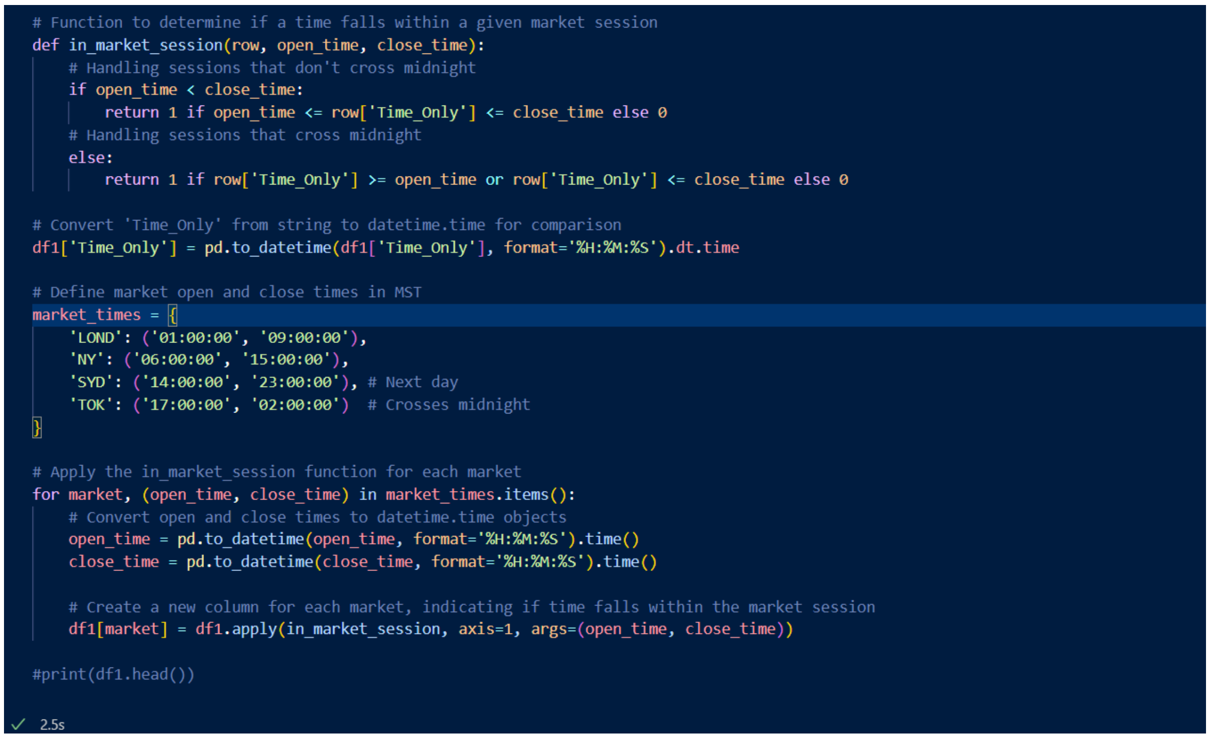


```python
import pandas as pd

# Define the SQL query
sql_query = "SELECT * FROM dbo.FXandTrades"

# Use pandas read_sql function to execute the query and load the results directly into a DataFrame
df1 = pd.read_sql(sql_query, connection)

# Now, df contains your data
print("DataFrame columns:", df1.columns)
#print(df1.head())  # Print the first few rows of the DataFrame

# Close the connection
connection.close()
```



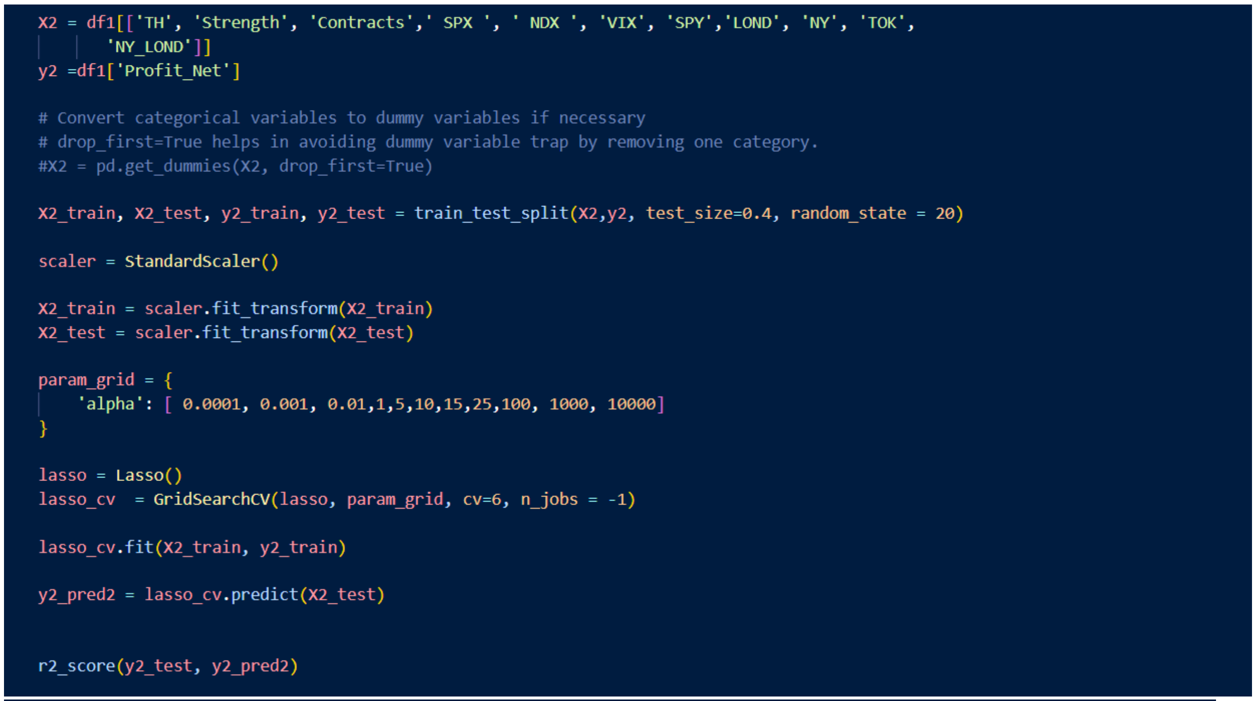
```python
# Ensure the 'When' column is in datetime format
df1['When'] = pd.to_datetime(df1['When'])

# Extract the time part and assign it to a new column 'Time_Only'
df1['Time_Only'] = df1['When'].dt.time

print(df1.columns)
print(df1['Time_Only'].head(5))
```

✓ 0.0s

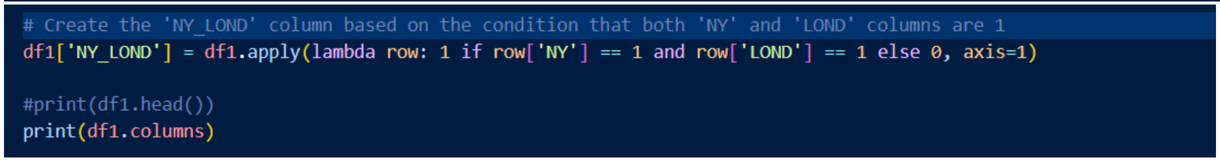


```python
# Function to determine if a time falls within a given market session
def in_market_session(row, open_time, close_time):
    # Handling sessions that don't cross midnight
    if open_time < close_time:
        return 1 if open_time <= row['Time_Only'] <= close_time else 0
    # Handling sessions that cross midnight
    else:
        return 1 if row['Time_Only'] >= open_time or row['Time_Only'] <= close_time else 0

# Convert 'Time_Only' from string to datetime.time for comparison
df1['Time_Only'] = pd.to_datetime(df1['Time_Only'], format='%H:%M:%S').dt.time

# Define market open and close times in MST
market_times = {
    'LOND': ('01:00:00', '09:00:00'),
    'NY': ('06:00:00', '15:00:00'),
    'SYD': ('14:00:00', '23:00:00'), # Next day
    'TOK': ('17:00:00', '02:00:00')  # Crosses midnight
}

# Apply the in_market_session function for each market
for market, (open_time, close_time) in market_times.items():
    # Convert open and close times to datetime.time objects
    open_time = pd.to_datetime(open_time, format='%H:%M:%S').time()
    close_time = pd.to_datetime(close_time, format='%H:%M:%S').time()

    # Create a new column for each market, indicating if time falls within the market session
    df1[market] = df1.apply(in_market_session, axis=1, args=(open_time, close_time))

#print(df1.head())
```

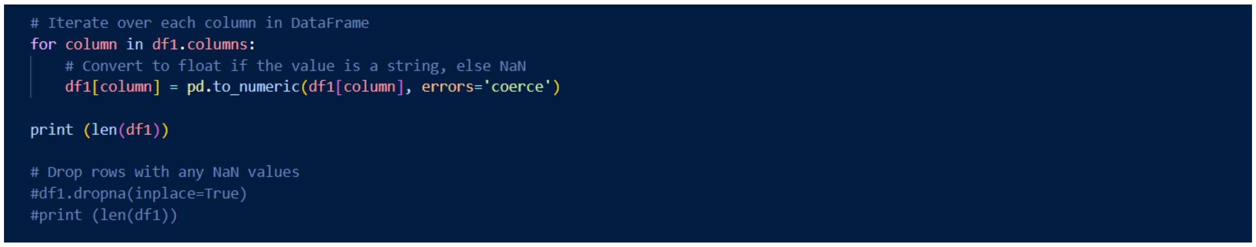
✓ 2.5s



```python
X2 = df1[['TH', 'Strength', 'Contracts',' SPX ', ' NDX ', 'VIX', 'SPY','LOND', 'NY', 'TOK',
        'NY_LOND']]
y2 =df1['Profit_Net']

# Convert categorical variables to dummy variables if necessary
# drop_first=True helps in avoiding dummy variable trap by removing one category.
#X2 = pd.get_dummies(X2, drop_first=True)

X2_train, X2_test, y2_train, y2_test = train_test_split(X2,y2, test_size=0.4, random_state = 20)

scaler = StandardScaler()

X2_train = scaler.fit_transform(X2_train)
X2_test = scaler.fit_transform(X2_test)

param_grid = {
    'alpha': [ 0.0001, 0.001, 0.01,1,5,10,15,25,100, 1000, 10000]
}

lasso = Lasso()
lasso_cv  = GridSearchCV(lasso, param_grid, cv=6, n_jobs = -1)

lasso_cv.fit(X2_train, y2_train)

y2_pred2 = lasso_cv.predict(X2_test)


r2_score(y2_test, y2_pred2)
```



```python
# Create the 'NY_LOND' column based on the condition that both 'NY' and 'LOND' columns are 1
df1['NY_LOND'] = df1.apply(lambda row: 1 if row['NY'] == 1 and row['LOND'] == 1 else 0, axis=1)

#print(df1.head())
print(df1.columns)
```

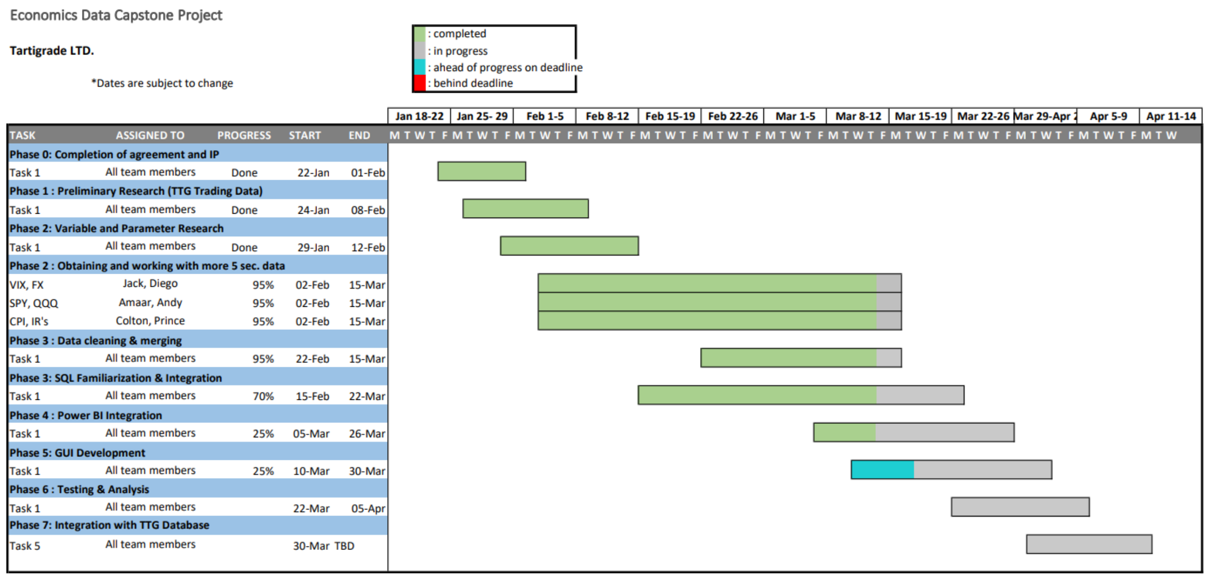


```python
# Iterate over each column in DataFrame
for column in df1.columns:
    # Convert to float if the value is a string, else NaN
    df1[column] = pd.to_numeric(df1[column], errors='coerce')

print (len(df1))

# Drop rows with any NaN values
#df1.dropna(inplace=True)
#print (len(df1))
```

Pseudocode:

```
1. Import necessary libraries (pyodbc, pandas, sklearn)
2. Define connection parameters for SQL database (server, database, username, password)
3. Create a connection string and establish a connection to the database
4. Initialize a cursor for the connection
5. Execute an SQL query to fetch column names from the "dbo.FXandTrades" table
   - Store and print the column names
6. Close the cursor
7. Use pandas to execute an SQL query to select all data from "dbo.FXandTrades" table
   - Load the results directly into a DataFrame named df1
8. Print the columns of the DataFrame
9. Convert the 'When' column in df1 to datetime format
10. Extract the time part from the 'When' column and assign it to a new column 'Time_Only'
11. Define a function to determine if a time falls within a given market session
12. Convert 'Time_Only' from string to datetime.time for comparison
13. Define market open and close times for various markets
14. Apply the market session function for each market to create new columns in df1 indicating if the time falls within the market session
15. Create the 'NY_LOND' column based on specific conditions relating to 'NY' and 'LOND' columns
16. Iterate over each column in DataFrame to convert values to float if they are strings, else convert to NaN
17. Drop rows with any NaN values
18. Select specific columns from df1 to form the features (X2) and target (y2) for modeling
19. Split the dataset into training and test sets
```

20. Initialize and fit a StandardScaler to normalize features
21. Define a parameter grid for Lasso regularization strength
22. Initialize a Lasso model and a GridSearchCV to find the optimal regularization strength
23. Fit the GridSearchCV to the training data
24. Predict the target variable for the test set
25. Calculate and print the R-squared score for the test set predictions



Economics Data Capstone Project

Tartigrade LTD.

*Dates are subject to change

🟩 : completed
▨ : in progress
🟦 : ahead of progress on deadline
🟥 : behind deadline

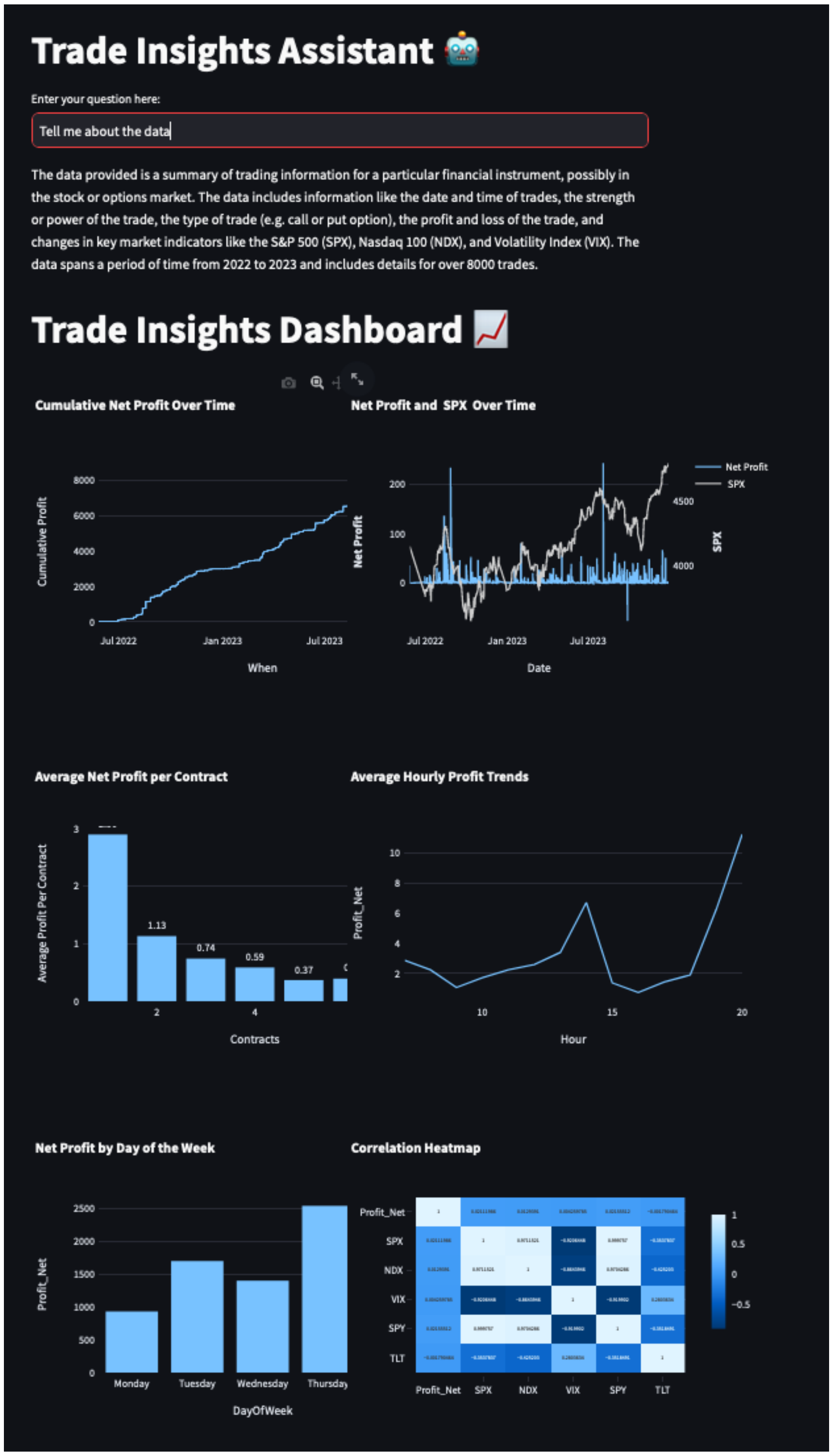
TASK	ASSIGNED TO	PROGRESS	START	END
Phase 0: Completion of agreement and IP				
Task 1	All team members	Done	22-Jan	01-Feb
Phase 1 : Preliminary Research (TTG Trading Data)				
Task 1	All team members	Done	24-Jan	08-Feb
Phase 2: Variable and Parameter Research				
Task 1	All team members	Done	29-Jan	12-Feb
Phase 2 : Obtaining and working with more 5 sec. data				
VIX, FX	Jack, Diego	95%	02-Feb	15-Mar
SPY, QQQ	Amaar, Andy	95%	02-Feb	15-Mar
CPI, IR's	Colton, Prince	95%	02-Feb	15-Mar
Phase 3 : Data cleaning & merging				
Task 1	All team members	95%	22-Feb	15-Mar
Phase 3: SQL Familiarization & Integration				
Task 1	All team members	70%	15-Feb	22-Mar
Phase 4 : Power BI Integration				
Task 1	All team members	25%	05-Mar	26-Mar
Phase 5: GUI Development				
Task 1	All team members	25%	10-Mar	30-Mar
Phase 6 : Testing & Analysis				
Task 1	All team members		22-Mar	05-Apr
Phase 7: Integration with TTG Database				
Task 5	All team members		30-Mar	TBD

Updated Gantt Chart :

Technical Appendix Progress Report 2: Group A

Additions to GUI:

1. OpenAI powered chatbot with access to selected trade data.
2. Minor layout changes to optimize space usage.


Trade Insights Assistant 🤖

Enter your question here:

Tell me about the data

The data provided is a summary of trading information for a particular financial instrument, possibly in the stock or options market. The data includes information like the date and time of trades, the strength or power of the trade, the type of trade (e.g. call or put option), the profit and loss of the trade, and changes in key market indicators like the S&P 500 (SPX), Nasdaq 100 (NDX), and Volatility Index (VIX). The data spans a period of time from 2022 to 2023 and includes details for over 8000 trades.

Trade Insights Dashboard 📈

Pseudocode (AI Chatbot):

```
// Initialize OpenAI API key (ensure to comment this out before deploying to view the GUI without an OpenAI API key)
Set openai.api_key to an empty string
```

```
// Display the title of the application as 'Trade Insights Assistant 🤖'
Display title "Trade Insights Assistant 🤖"

// Prompt the user to enter their query into a text input field
user_query = Display text input field with prompt "Enter your question here:"

// Define a function to call the OpenAI API with a question and their trade dataframe
Function ask_openai(question, dataframe)
Try
// Create a prompt combining the user's question with a data summary
prompt = "Question: " + question + "\n\nData Summary:\n" + dataframe + "\n\nAnswer:"

// Call the OpenAI API with the specified parameters
response = Call OpenAI completions API with:
    model set to "gpt-3.5-turbo-instruct"
    prompt set to the created prompt
    max_tokens set to 150
    n set to 1

// Extract and return the text from the API's response
answer = Extract text from response and trim whitespace
Return answer
Catch any exception as e
// Return the exception message if an error occurs
Return the exception message as a string

// Display the answer to the user's query if a query was submitted
If user_query is not empty
Display spinner with text "Getting insights from Data..."
answer = Call ask_openai with user_query and dataframe
    Display answer
```

Additional GUI Visualizations:

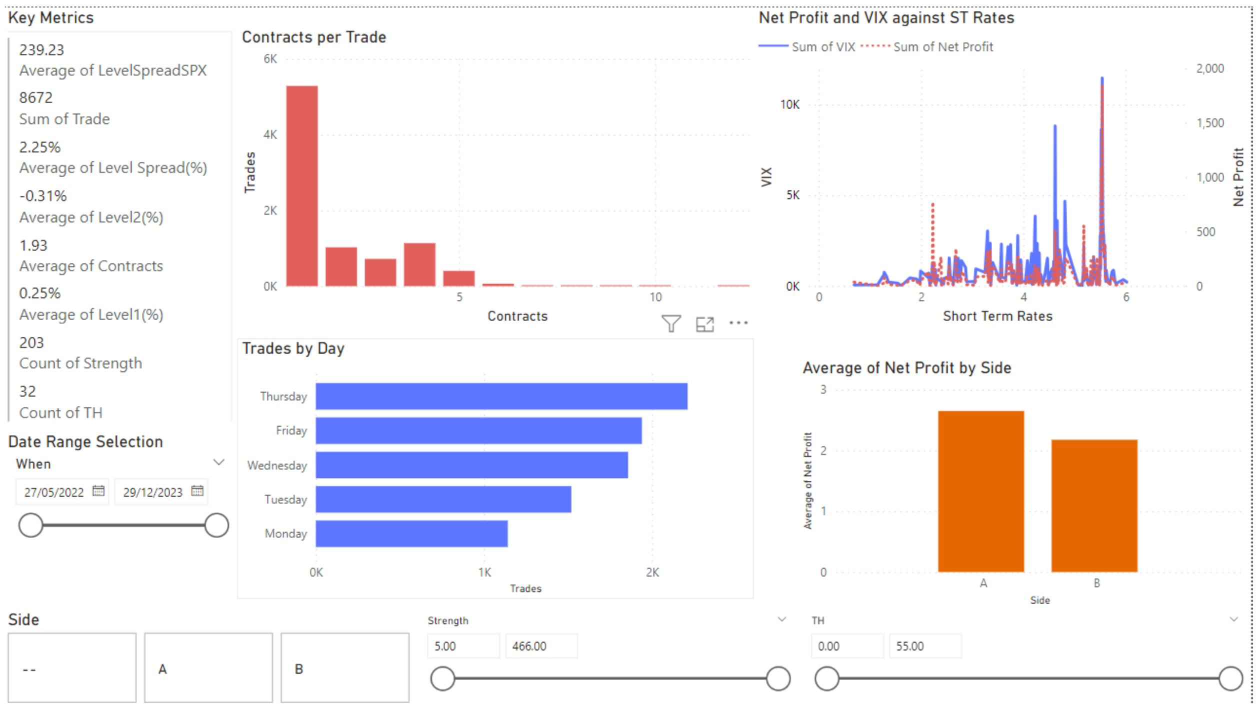
Code for Additional Visualizations:

```
# 6. Visualization: Average Profit by Day of the Week
# Extract the day of the week from 'When' (0 = Monday, 6 = Sunday)
filtered_data['Day of the Week'] = filtered_data['When'].dt.dayofweek
# Calculate the average profit for each day of the week
average_profit_weekday = filtered_data.groupby('Day of the Week')['Profit_Net'].mean().reset_index()
```

```
# Map the integers to weekday names for clearer visualization
weekday_mapping = {0: 'Monday', 1: 'Tuesday', 2: 'Wednesday', 3: 'Thursday', 4: 'Friday', 5: 'Saturday', 6: 'Sunday'}
average_profit_weekday['Day of the Week'] = average_profit_weekday['Day of the Week'].map(weekday_mapping)
# Plot
fig_weekday = px.bar(average_profit_weekday, x='Day of the Week', y='Profit_Net', text='Profit_Net',
                 title='Average Net Profit by Day of the Week')
fig_weekday.update_traces(texttemplate='%{text:.2s}', textposition='outside')
fig_weekday.update_layout(xaxis_title="Day of the Week", yaxis_title="Average Net Profit")
st.plotly_chart(fig_weekday)

# 7. Visualization: Average Profit by Day of the Month
filtered_data['Day of the Month'] = filtered_data['When'].dt.day
# Calculate the average profit for each day of the month
average_profit_day = filtered_data.groupby('Day of the Month')['Profit_Net'].mean().reset_index()
# Plot
fig_day = px.bar(average_profit_day, x='Day of the Month', y='Profit_Net', text='Profit_Net',
             title='Average Net Profit by Day of the Month')
fig_day.update_traces(texttemplate='%{text:.2s}', textposition='outside')
fig_day.update_layout(xaxis_title="Day of the Month", yaxis_title="Average Net Profit")
st.plotly_chart(fig_day)
```

Power BI (PBI) updates:

**Page 1*



Key Metrics

239.23
Average of LevelSpreadSPX

8672
Sum of Trade

2.25%
Average of Level Spread(%)

-0.31%
Average of Level2(%)

1.93
Average of Contracts

0.25%
Average of Level1(%)

203
Count of Strength

32
Count of TH

Date Range Selection
When
27/05/2022 29/12/2023

Side
-- A B

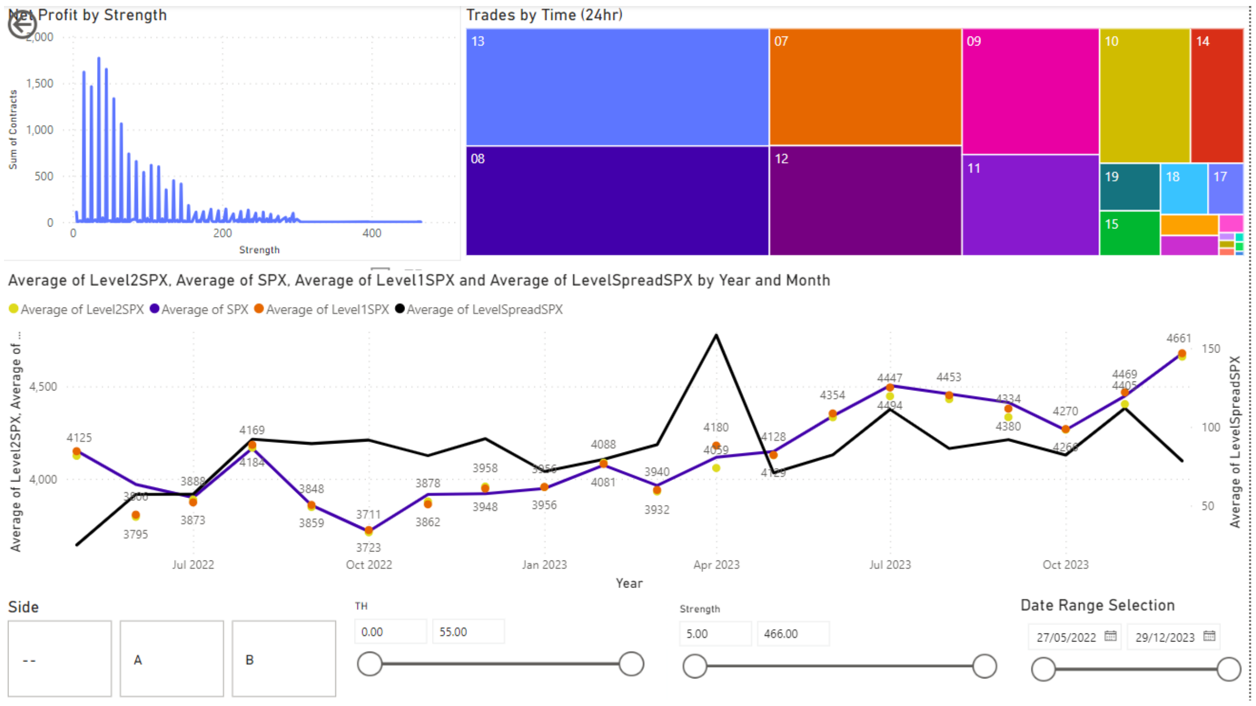
Strength
5.00 466.00

TH
0.00 55.00

PBI Graphs/Charts:

1. (Top Left) Shows the contracts per trade in a clustered column format. As the firm is still quite new, it shows that often, fewer contracts per trade occur. As the firm has expanded, it has been trading more contracts per trade which is slowly bringing the average up
2. (Top Right) Shows relationship between short term interest rates (US) and the response from the VIX. Net profit is also on there to see its relationship with the VIX as it had no direct relationship with the rates. As found, the VIX becomes more volatile as rates increase. Being a firm, whose strategy is to capitalize on market volatility, their net profit spikes and fluctuates with the VIX
3. (Bottom Left) Shows trades by day of the week. Thursday being when the trades are happening the most (therefore profit). Possible explanations for this obvious advantage this day brings are that earning reports and some key economic reports are released on Thursdays which has a short impact on volatility which could be when their system chooses to trade
4. (Bottom Right) Shows profit per side of trade. Side A trades are more profitable for the firm overall. However there seems to be increased risk for taking this side as occasionally the firm loses money while taking this side. Side B, while its profits are less, seldom loses money and when it does, the losses are a fraction of what they are when taking a poor performing side A trade.
5. (Left side) Key metrics table has some key metrics about the data the partner may find useful
6. (Bottom) Slicers allow for user to customize variables that the partner said were in their control (Most of these slicers repeat on subsequent pages)

**Page 2*



Side

--	A	B

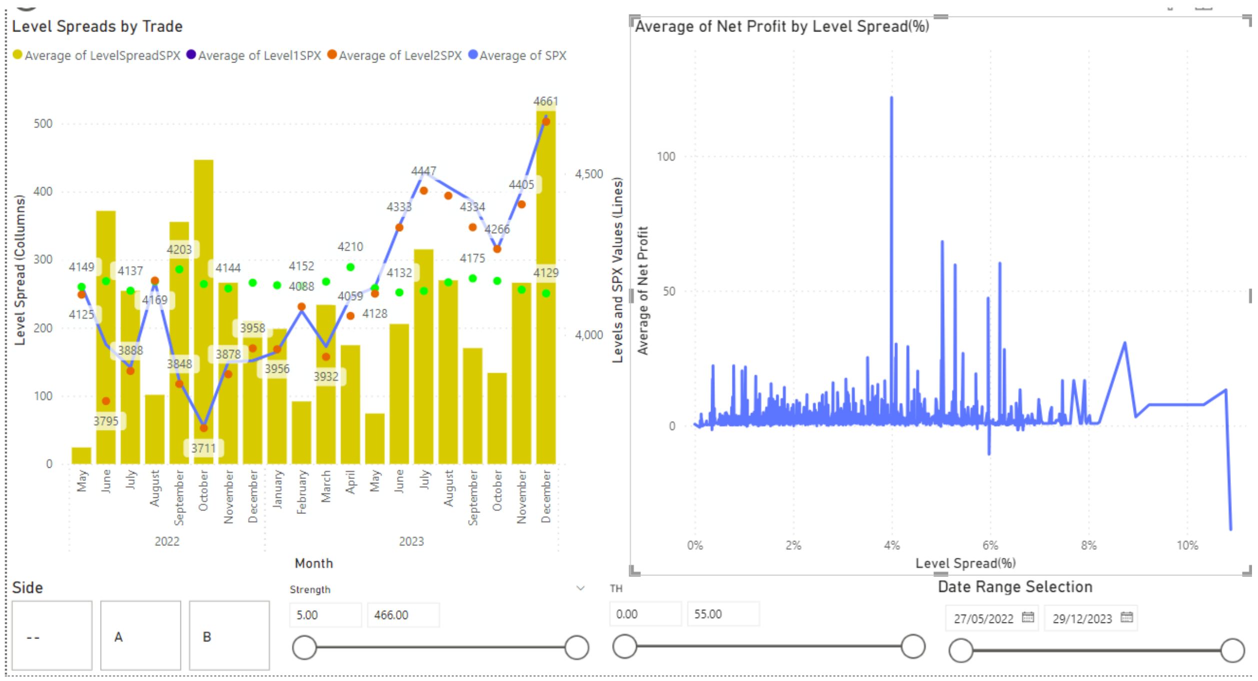
TH: 0.00 – 55.00

Strength: 5.00 – 466.00

Date Range Selection: 27/05/2022 – 29/12/2023

7. (Top Left) Strength of trades against the firm's net profit. Every 5 units of strength yields profit spikes/troughs. The number of contracts traded at the trough values is likewise low, which impacts this graph. When filtering for sides, A has much more variation in the strength values of trade than side B.
8. (Top Right) Tree map of the times (24hr) at which trades are occurring. The most frequent times are 8AM and 1PM. The hours surrounding these times take up most of the trades being done. The early trading times (7,8,9) correlate heavily with the NYSE open time. In MST that exchange opens at 7:30. This could explain why trades at 6 are much less common. It is much the same for after 2PM MST where a sharp trading falloff occurs. More than 90% of trades happen during the times when the NYSE is open.
9. (Bottom) Tracking of the SPX along with the spread between levels (aiming to fix the SPX at a value for easier viewing and more insight)

Page 3



Side: -- | A | B

Strength: 5.00 – 466.00

TH: 0.00 – 55.00

Date Range Selection: 27/05/2022 – 29/12/2023

10. (Left) This combination chart attempts to do much of what #9 did. Showing the levels along with the SPX. The difference between the levels is in the form of columns. Levels, since they are a function of SPX, are expected to be correlated with it. This is not the case for level 1, which is currently mislabeled in bright green.
11. (Right) The line chart shows the profit against the level spread. While always fluctuating, the profit is nearly always positive at spreads less than 5% in difference. After that point, observations are limited yet does appear to become less predictable and potentially costly to the firm

SQL / Code Updates:

TTG expressed to us that if possible, they would like to be able to visualize what the relationship is between how much the market (denominated by an important index like the SPX, or a currency pair, like USD.CAD) moved in absolute terms (i.e. the absolute sum of how much the price fluctuated) and net profit.

In response, we created new tables in SQL that did performed a left join between their trades table and tables with 5 second price data for a given security. The result was multiple very large tables (about 1.9 million rows), with trades occurring for approximately 7,000 of those rows, all matched up by time. We then downloaded the new table(s) into python as dataframes, created a function to calculate the absolute sum of price movements between trades (where the running sum restarts after every trade is made),and then create a smaller dataframe where each row had a trade occurrence and also showed how much the price moved before the trade was made.

The pseudocode for the python operations above is as follows:

```
1. Initialize database connection parameters:
   1.1. Set server address.
   1.2. Set database name.
   1.3. Set username.
   1.4. Set password.

2. Establish connection to the SQL database using provided parameters.

3. Fetch column names from a specified table:
   3.1. Execute SQL query to select all from a designated table.
   3.2. Retrieve column names from the query result.
   3.3. Close database cursor.

4. Define SQL queries to extract necessary data from database.

5. Load data into DataFrame using pandas for further analysis:
   5.1. Execute defined SQL queries.
   5.2. Close database connection.

6. Data preprocessing:
   6.1. Convert specific columns to datetime or numeric formats as required.
   6.2. Sort DataFrame by datetime column in ascending order.
   6.3. Identify and count NaN values in selected columns.

7. Feature engineering:
   7.1. Add a column to indicate presence of trades based on conditions.
```

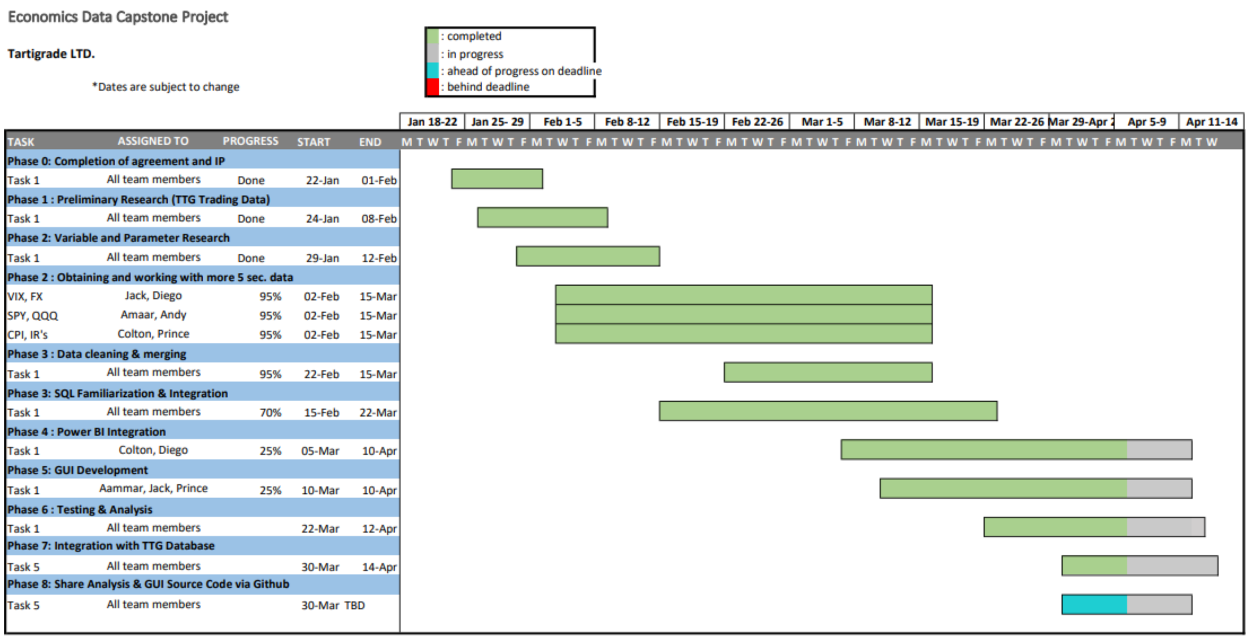
7.2. Calculate absolute price differences and store in new column.
   7.3. Initialize columns for movement metrics and calculate running totals.

8. Create a filtered DataFrame including only rows where trades occurred:
   8.1. Filter original DataFrame based on 'Trades' column.
   8.2. Optionally, reset index of the new DataFrame.

9. Prepare data for database insertion:
   9.1. Establish parameters for database connection.
   9.2. Define target table name for data insertion.

10. Write DataFrame back to the SQL database using SQLAlchemy:
    10.1. Create database engine with connection parameters.
    10.2. Execute data write operation to specified table, replacing if exists.

Gantt Chart:


Economics Data Capstone Project

Tartigrade LTD.

*Dates are subject to change

🟩	: completed
▨	: in progress
🟦	: ahead of progress on deadline
🟥	: behind deadline

TASK	ASSIGNED TO	PROGRESS	START	END
Phase 0: Completion of agreement and IP				
Task 1	All team members	Done	22-Jan	01-Feb
Phase 1 : Preliminary Research (TTG Trading Data)				
Task 1	All team members	Done	24-Jan	08-Feb
Phase 2: Variable and Parameter Research				
Task 1	All team members	Done	29-Jan	12-Feb
Phase 2 : Obtaining and working with more 5 sec. data				
VIX, FX	Jack, Diego	95%	02-Feb	15-Mar
SPY, QQQ	Amaar, Andy	95%	02-Feb	15-Mar
CPI, IR's	Colton, Prince	95%	02-Feb	15-Mar
Phase 3 : Data cleaning & merging				
Task 1	All team members	95%	22-Feb	15-Mar
Phase 3: SQL Familiarization & Integration				
Task 1	All team members	70%	15-Feb	22-Mar
Phase 4 : Power BI Integration				
Task 1	Colton, Diego	25%	05-Mar	10-Apr
Phase 5: GUI Development				
Task 1	Aammar, Jack, Prince	25%	10-Mar	10-Apr
Phase 6 : Testing & Analysis				
Task 1	All team members		22-Mar	12-Apr
Phase 7: Integration with TTG Database				
Task 5	All team members		30-Mar	14-Apr
Phase 8: Share Analysis & GUI Source Code via Github				
Task 5	All team members		30-Mar	TBD

How to Update PBI Dashboard

For the Code to Run:

1. Ensure you are connected to the TTG VPN
2. Install Python
3. Using a PC to run the code is best
4. Be in a place with strong connection to the network as any break in the connection will corrupt the running process and will have to be run again.
5. Ensure all packages are installed via your command prompt (These packages are at the top of the codes and are surrounded by red highlighted language such as 'import')
6. Adjust computer sleep settings. This code takes a few hours to run, and a sleeping computer will sever the connection resulting in the same issue identified in 3.
7. When updating SQL, updating the data should not yield any negative effects so long as the same number of observations are being pulled. If one were to add millions more rows of data, the run time will lengthen.
8. It is assumed that you are updating the tables you made for us (RiskFreeCurve, Trades, TemplateHistoData)

For Most Visuals (‘5sDataMerged’ Table Update) ~2.5 hours

1. In the GitHub repository, enter the ‘Analysis’ folder and select ‘Colton Analysis
2. Scroll down to line 73 and replace the username and password sections of the connection string with your own credentials
3. When prompted, select ‘y’ (This is a response to the question on screen)
4. Wait a while (<15 minutes) times will be appearing on the screen to show that the code is in fact running
5. When prompted, select ‘y’
6. Leave your computer running, every time a chunk is uploaded, a time stamp will appear on screen. Periodically check back in on the code. This process can take (2.5 hours)
7. If upon adding many new rows of data (millions more), a rollback error may occur which requires a restart. To fix, increase the higher of the two numbers in lines 493 and 497. This breaks the data into smaller chunks to more easily upload back to SQL
8. Open the Dashboard and refresh the data for the table

For VIX and Interest Rates Visual ('With Rate Volatility' Table Update) ~2 minutes

1. In the GitHub repository, enter the ‘Analysis’ folder and select ‘Colton Analysis
2. Scroll down to line 73 and replace the username and password sections of the connection string with your own credentials
3. When prompted to select ‘y’, simply hit enter
4. Analysis section is included. When the analysis selection appears on screen (OLS/LASSO prompt), you may kill the code as the table will have been updated by then. Feel free to carry on with the code but the table update in SQL for the dashboard is complete.
5. Open the Dashboard and refresh the data for the table.

SPX Volatility and Trades (‘PBI’ Table Update) ~2.5 hours

1. In the GitHub repository, in the SQLDB connection methods, open the PBI code.
2. Go to line 43 and replace the username and password sections of the connection string with your own credentials
3. When prompted, select ‘y’
4. Wait a while (15-20 minutes) times will appear on the screen to show that the code is in fact running.
5. When prompted, select ‘y’
6. Leave your computer running, every time a chunk is uploaded, a time stamp will appear on screen. Periodically check back in on the code. This process can take (2.5 hours)

7. If upon adding many new rows of data (millions more), a rollback error may occur which requires a restart. To fix, increase the higher of the two numbers in lines 197 and 201. This breaks the data into smaller chunks to more easily upload back to SQL
8. Open the Dashboard and refresh the data for the table

Market Sessions and News Sentiments PBI Pages ~ 20 minutes

1. In the GitHub repository in the “Required SQL Queries Guide”, open the document and use run the respective queries for the operation you wish to do.
    a. The first query works to create a new table of merged data to analyze market sessions and is required for the news sentiment web scraper – **Use this Query**.
    b. The second queries work to create the new tables required for the code that calculates how much the market moves (either for the SPX or USD.CAD) between trades .
2. In the GitHub Repository in the “Analysis” folder, scroll to the bottom and select the Python file called “Mkt Sessions + News Sentiment Analysis.py”.
3. Run the code, ensuring that the TTG VPN is connected beforehand, a valid SQL username and password are input in the required fields, and the SQL database is open in the background.
4. Open the Dashboard and refresh the data for the table.

Market Movement PBI Page ~ 20 minutes

1. In the GitHub repository in the “Required SQL Queries Guide”, open the document and use run the respective queries for the operation you wish to do.
    a. The first query works to create a new table of merged data to analyze market sessions and is required for the news sentiment web scraper.
    b. The second queries work to create the new tables required for the code that calculates how much the market moves (either for the SPX or USD.CAD) between trades – **Use this Query**.
2. In the GitHub Repository in the “Analysis” folder, scroll to the bottom and select the Python file called “Market Movement Analysis.py”.
3. Run the code, ensuring that the TTG VPN is connected beforehand, a valid SQL username and password are input in the required fields, and the SQL database is open in the background.

    a. Alter the code as is explained in the comment in the first lines to determine if you want to update USD.CAD or SPX movement calculations.

4. Open the Dashboard and refresh the data for the table.