Preston Khiev
EE 518 MLTA
MP3: JPX Tokyo Stock Exchange Prediction

```
# This Python 3 environment comes with many helpful analytics libraries
installed
# It is defined by the kaggle/python Docker image:
https://github.com/kaggle/docker-python
# For example, here's several helpful packages to load

import numpy as np # linear algebra
import pandas as pd # data processing, CSV file I/O (e.g. pd.read_csv)

# Input data files are available in the read-only "../input/" directory
# For example, running this (by clicking run or pressing Shift+Enter) will
list all files under the input directory

import os
for dirname, _, filenames in os.walk('/kaggle/input'):
    for filename in filenames:
        print(os.path.join(dirname, filename))

# You can write up to 20GB to the current directory (/kaggle/working/) that
gets preserved as output when you create a version using "Save & Run All"
# You can also write temporary files to /kaggle/temp/, but they won't be
saved outside of the current session

import warnings, gc
import numpy as np
import pandas as pd
import matplotlib.colors
import seaborn as sns
import plotly.express as px
import plotly.graph_objects as go
from plotly.subplots import make_subplots
from plotly.offline import init_notebook_mode
from datetime import datetime, timedelta
from sklearn.model_selection import TimeSeriesSplit
from sklearn.metrics import mean_squared_error,mean_absolute_error
from lightgbm import LGBMRegressor
from decimal import ROUND_HALF_UP, Decimal
warnings.filterwarnings("ignore")
import plotly.figure_factory as ff

init_notebook_mode(connected=True)
temp = dict(layout=go.Layout(font=dict(family="Franklin Gothic", size=12),
width=800))
colors=px.colors.qualitative.Plotly

train=pd.read_csv("../input/jpx-tokyo-stock-exchange-
prediction/train_files/stock_prices.csv", parse_dates=['Date'])
stock_list=pd.read_csv("../input/jpx-tokyo-stock-exchange-
prediction/stock_list.csv")

print("The training data begins on {} and ends on
{}.\n".format(train.Date.min(),train.Date.max()))
```

```
display(train.describe().style.format('{:,.2f}'))
```

```
The training data begins on 2017-01-04 00:00:00 and ends on 2021-12-03 00:00:00.
```

	SecuritiesCode	Open	High	Low	Close	Volume	AdjustmentFactor	ExpectedDividend	Target
count	2,332,531.00	2,324,923.00	2,324,923.00	2,324,923.00	2,324,923.00	2,332,531.00	2,332,531.00	18,865.00	2,332,293.00
mean	5,894.84	2,594.51	2,626.54	2,561.23	2,594.02	691,936.56	1.00	22.02	0.00
std	2,404.16	3,577.19	3,619.36	3,533.49	3,576.54	3,911,255.94	0.07	29.88	0.02
min	1,301.00	14.00	15.00	13.00	14.00	0.00	0.10	0.00	-0.58
25%	3,891.00	1,022.00	1,035.00	1,009.00	1,022.00	30,300.00	1.00	5.00	-0.01
50%	6,238.00	1,812.00	1,834.00	1,790.00	1,811.00	107,100.00	1.00	15.00	0.00
75%	7,965.00	3,030.00	3,070.00	2,995.00	3,030.00	402,100.00	1.00	30.00	0.01
max	9,997.00	109,950.00	110,500.00	107,200.00	109,550.00	643,654,000.00	20.00	1,070.00	1.12

1) Problem Description: In contrast to traditional stock trading, quantitative stock trading relies on trained machine learning models to predict the future performance of stocks. These predictions are used to formulate and execute trading strategies to maximize returns. As such, it is important to predict stock performance as accurately as possible. The objective of this project is to build a model that will predict future returns of Japanese stocks, and rank stocks from the Tokyo Stock Exchange (TSE) in order of predicted performance. Predictions will be generated using the LightGBM regressor.

The data is from the Kaggle JPX Tokyo Stock Exchange Prediction competition [1]. The dataset is historical data for a variety of Japanese stocks and options.

1. The file "stock_prices.csv" includes the daily closing price for each stock and a target column.

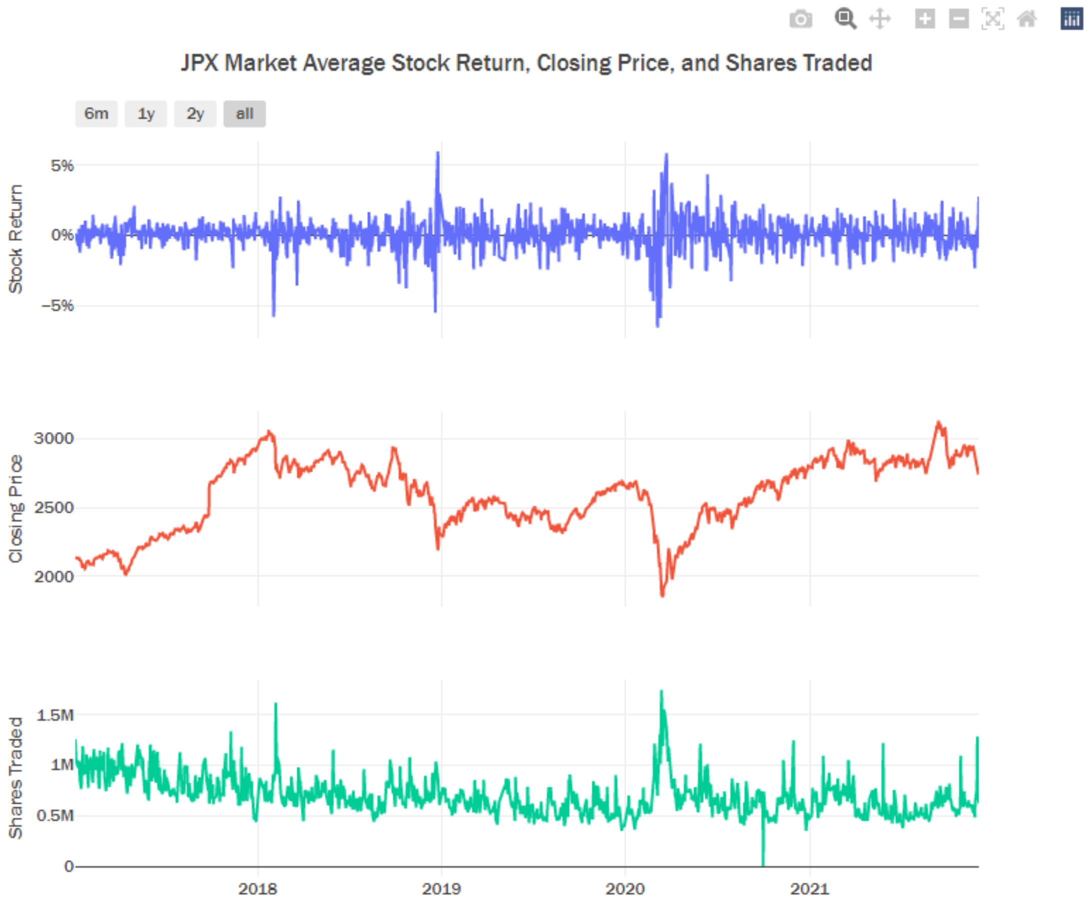
- RowId is the unique ID of price records
- Date is the trade date
- SecuritiesCode is the local securities code
- Open is the opening price of the security for the day
- Close is the closing price of the security for the day
- Volume is the number of stocks traded on a day
- AdjustmentFactor isused to calculate the theoretical price and volume when a split or reverse splitt happens. This does NOT include dividend or alotment of shares.
- ExpectedDividend is the expected dividend value for the ex-right date. It is recorded 2 days before the ex-dividend date.
- SupervisionFlag is a flag for securities under supervision and decurities to be delisted.
- Target is the change ratio of adjusted closing price between t+2 and t+1 where t+0 is the date of the trade.

1. "options.csv" contains data on the status of stock options based on the broader market.
2. "secondary_stock_prices.csv" is the core dataset and contains data for 2000 of the most commonly traded equities in the Tokyo market. Data for other, less liquid securities are included as well, however these are not scored.

3. "trades.csv" contains the aggregated summary of trading volumes from the previous business week. "financials.csv" contains results from quarterly earnings reports.
4. Finally, "stock_list.csv" has mappings between the SecuritiesCode variable and company names as well as general information about each company's industry.

2) EDA To begin EDA, The average return, closing price, and trading volume vs time was plotted for all stocks. It was noted that trading volume tends to spike with large fluctuations in closing price which also tends to coincide with fluctions in return. This makes sense because people are typically trading their shares in response to trading price. It is also interesting to note that average trading volume appears to decrease over the training period.

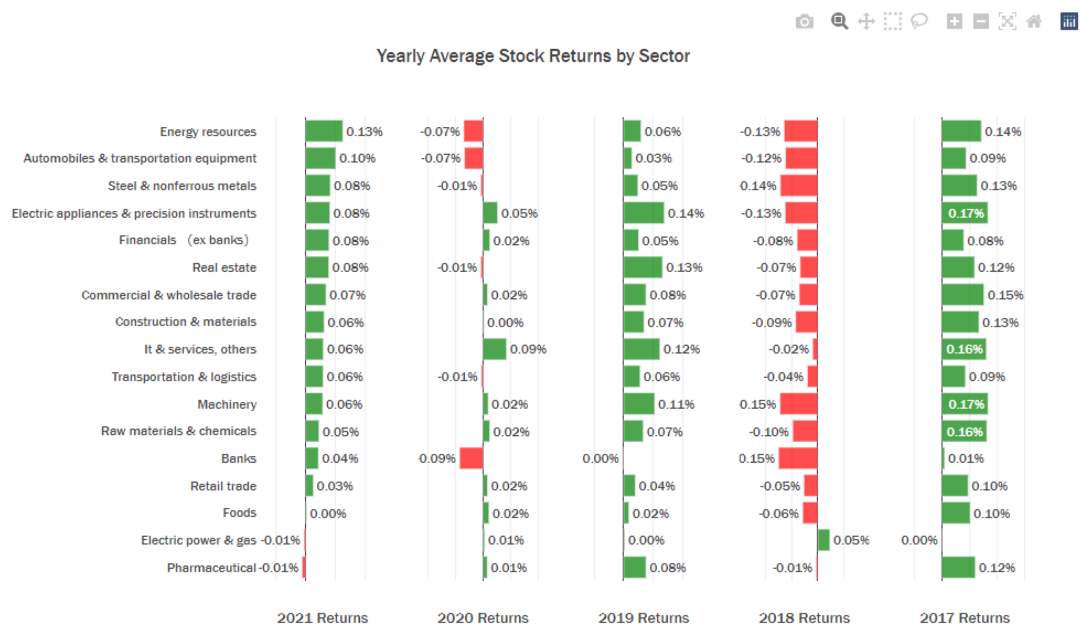
```
train_date=train.Date.unique()
returns=train.groupby('Date')['Target'].mean().mul(100).rename('Average
Return')
close_avg=train.groupby('Date')['Close'].mean().rename('Closing Price')
vol_avg=train.groupby('Date')['Volume'].mean().rename('Volume')

fig = make_subplots(rows=3, cols=1,
                    shared_xaxes=True)
for i, j in enumerate([returns, close_avg, vol_avg]):
    fig.add_trace(go.Scatter(x=train_date, y=j, mode='lines',
                             name=j.name, marker_color=colors[i]), row=i+1,
col=1)
fig.update_xaxes(rangeslider_visible=False,
                 rangeselector=dict(
                     buttons=list([
                         dict(count=6, label="6m", step="month",
stepmode="backward"),
                         dict(count=1, label="1y", step="year",
stepmode="backward"),
                         dict(count=2, label="2y", step="year",
stepmode="backward"),
                         dict(step="all")])),
                 row=1,col=1)
fig.update_layout(template=temp,title='JPX Market Average Stock Return,
Closing Price, and Shares Traded',
                  hovermode='x unified', height=700,
                  yaxis1=dict(title='Stock Return', ticksuffix='%'),
                  yaxis2_title='Closing Price', yaxis3_title='Shares Traded',
                  showlegend=False)
fig.show()
```



2) EDA Next, the average annual returns by sector were plotted for the training period. It was noted that (nearly) all sectors saw positive returns in 2021, 2019, and 2017. Meanwhile, all sectors besides Electic Power and Gas saw negative returns in 2018.

```
stock_list['SectorName']=[i.rstrip().lower().capitalize() for i in
stock_list['17SectorName']]
stock_list['Name']=[i.rstrip().lower().capitalize() for i in
stock_list['Name']]
train_df = train.merge(stock_list[['SecuritiesCode','Name','SectorName']],
on='SecuritiesCode', how='left')
train_df['Year'] = train_df['Date'].dt.year
years = {year: pd.DataFrame() for year in train_df.Year.unique()[::-1]}
for key in years.keys():
    df=train_df[train_df.Year == key]
    years[key] =
df.groupby('SectorName')['Target'].mean().mul(100).rename("Avg_return_{}".for
mat(key))
df=pd.concat((years[i].to_frame() for i in years.keys()), axis=1)
df=df.sort_values(by="Avg_return_2021")

fig = make_subplots(rows=1, cols=5, shared_yaxes=True)
for i, col in enumerate(df.columns):
```

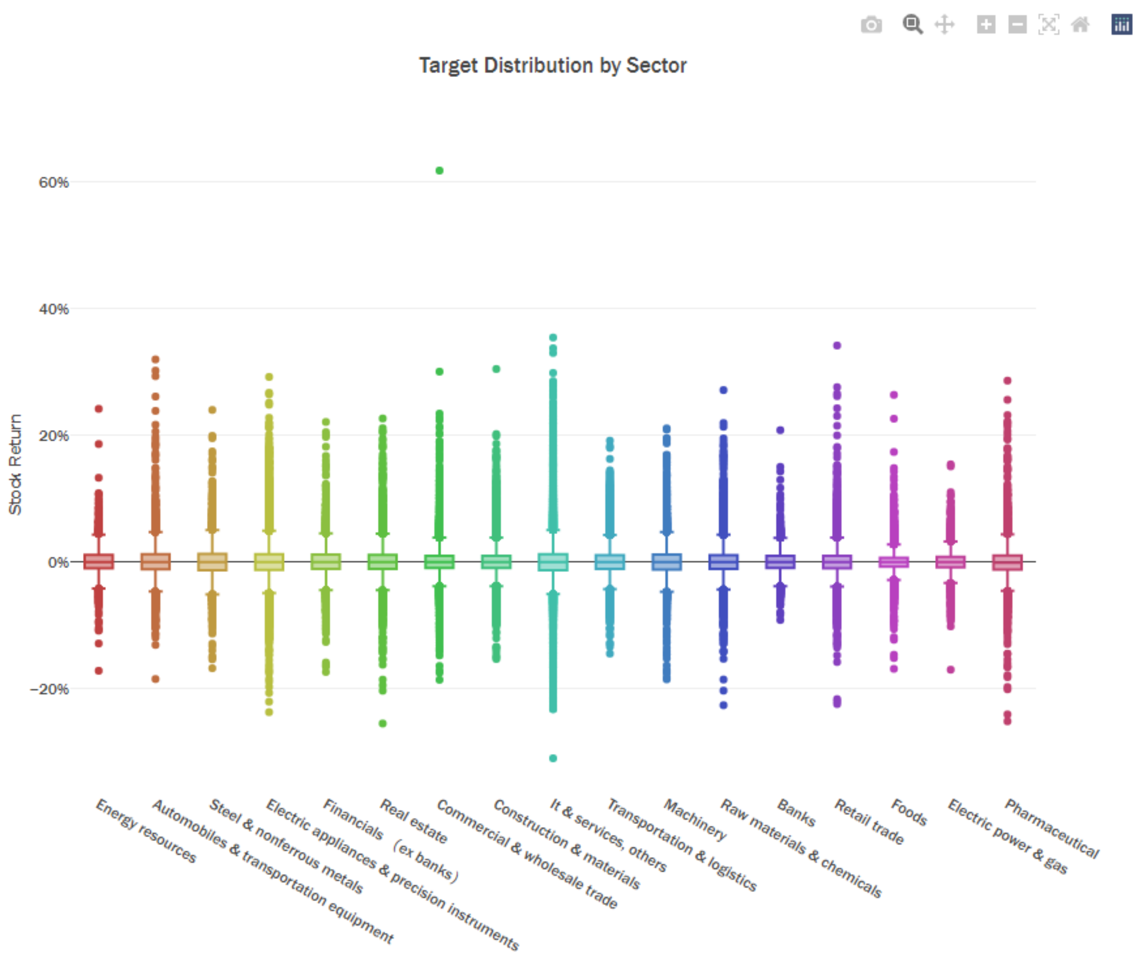
```
    x = df[col]
    mask = x<=0
    fig.add_trace(go.Bar(x=x[mask], y=df.index[mask],orientation='h',
                         text=x[mask],
texttemplate='%{text:.2f}%',textposition='auto',
                         hovertemplate='Average Return in %{y} Stocks =
%{x:.4f}%',
                         marker=dict(color='red', opacity=0.7),name=col[-
4:]),
                  row=1, col=i+1)
    fig.add_trace(go.Bar(x=x[~mask], y=df.index[~mask],orientation='h',
                         text=x[~mask], texttemplate='%{text:.2f}%',
textposition='auto',
                         hovertemplate='Average Return in %{y} Stocks =
%{x:.4f}%',
                         marker=dict(color='green', opacity=0.7),name=col[-
4:]),
                  row=1, col=i+1)
    fig.update_xaxes(range=(x.min()-.15,x.max()+.15), title='{}
Returns'.format(col[-4:]),
                     showticklabels=False, row=1, col=i+1)
fig.update_layout(template=temp,title='Yearly Average Stock Returns by
Sector',
                  hovermode='closest',margin=dict(l=250,r=50),
                  height=600, width=1000, showlegend=False)
fig.show()
```



2) EDA: Since data for some stocks were added in December 2020, the data was filtered after this date. The Target distribution was plotted by sector to see if certain sectors had better performance than others. It was observed that all sectors had similar returns, spanning from -1% to 1%. However, it was noted that there were numerous outliers.

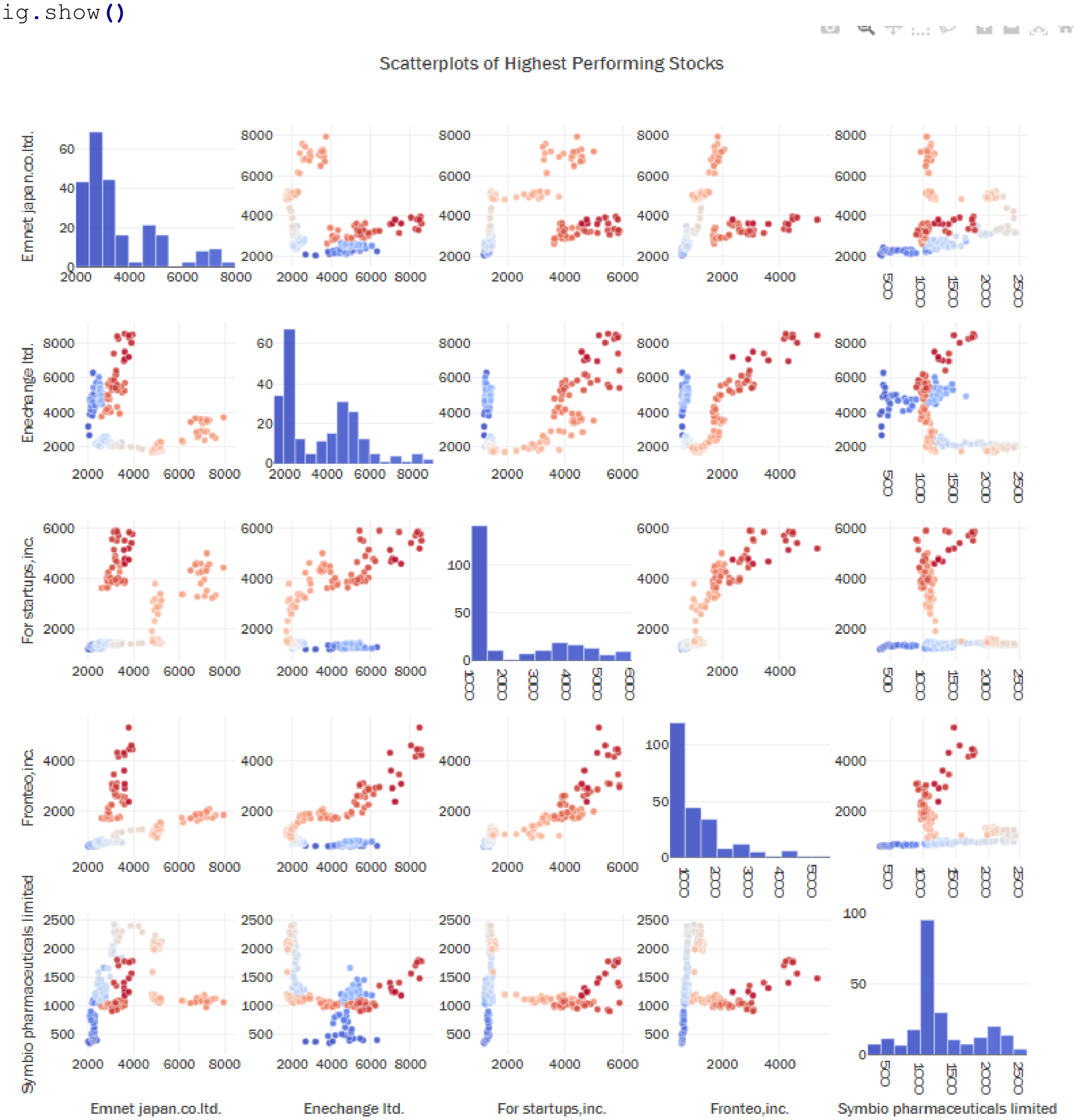
```
train_df=train_df[train_df.Date>'2020-12-23']
print("New Train Shape {}.\nMissing values in Target =
{}".format(train_df.shape,train_df['Target'].isna().sum()))
```

```
pal = ['hsl('+str(h)+',50%'+',50%)' for h in np.linspace(0, 360, 18)]
fig = go.Figure()
for i, sector in enumerate(df.index[::-1]):
    y_data=train_df[train_df['SectorName']==sector]['Target']
    fig.add_trace(go.Box(y=y_data*100, name=sector,
                         marker_color=pal[i], showlegend=False))
fig.update_layout(template=temp, title='Target Distribution by Sector',
                  yaxis=dict(title='Stock Return',ticksuffix='%'),
                  margin=dict(b=150), height=750, width=900)
fig.show()
```




```
stocks=train_df[train_df.SecuritiesCode.isin([4169,7089,4582,2158,7036])]
df_pivot=stocks.pivot_table(index='Date', columns='Name',
values='Close').reset_index()
pal=['rgb'+str(i) for i in sns.color_palette("coolwarm", len(df_pivot))]

fig = ff.create_scatterplotmatrix(df_pivot.iloc[:,1:], diag='histogram',
name='')
```

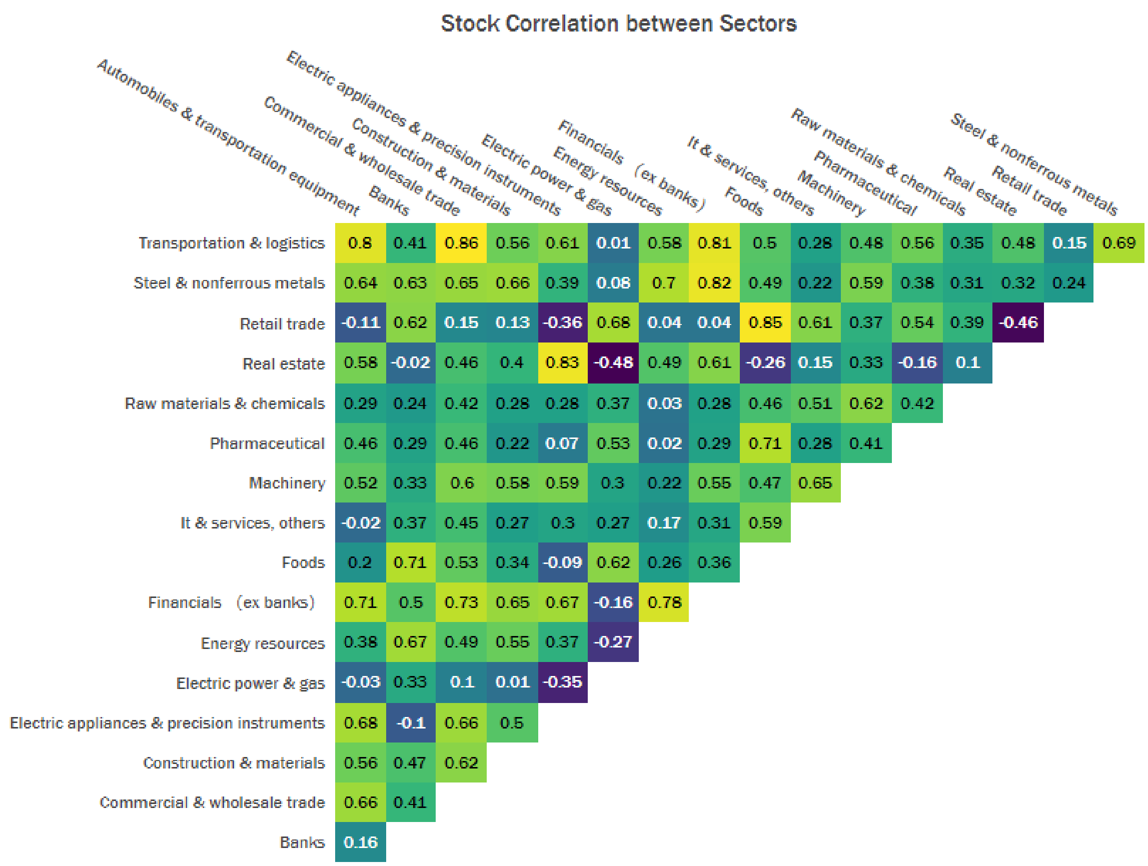
```
fig.update_traces(marker=dict(color=pal, opacity=0.9, line_color='white',
line_width=.5))
fig.update_layout(template=temp, title='Scatterplots of Highest Performing
Stocks',
                  height=1000, width=1000, showlegend=False)
fig.show()
```



2)EDA Next, the correlation of stocks were visualized by sector. It is interesting to note that there are a lot of pairs of sectors that are strongly correlated to each other. For example, Commercial & wholesale trade is strongly correlated to transportation & logistics with a correlation of 0.86.

```
df_pivot=train_df.pivot_table(index='Date', columns='SectorName',
values='Close').reset_index()
corr=df_pivot.corr().round(2)
mask=np.triu(np.ones_like(corr, dtype=bool))
```

```
c_mask = np.where(~mask, corr, 100)
c=[]
for i in c_mask.tolist()[1:]:
    c.append([x for x in i if x != 100])

cor=c[::-1]
x=corr.index.tolist()[:-1]
y=corr.columns.tolist()[1:][::-1]
fig=ff.create_annotated_heatmap(z=cor, x=x, y=y,
                                hovertemplate='Correlation between %{x} and
%{y} stocks = %{z}',
                                colorscale='viridis', name='')
fig.update_layout(template=temp, title='Stock Correlation between Sectors',
                  margin=dict(l=250,t=270),height=800,width=900,
                  yaxis=dict(showgrid=False, autorange='reversed'),
                  xaxis=dict(showgrid=False))
fig.show()
```



3) Challenges The main challenge for this dataset was definitely associated with the understanding/manipulating the data. It wasn't immediately to me how to go about selecting/engineering/eliminating features for machine learning. For example, as mentioned earlier, options include implicit predictions of the future prices of the stock market. How are those predictions made, and can they provide any insight to our model? How about the included data for the smaller securities? They aren't scored, but they do influence the entire market still. Does that mean they should be taken into account when training the model? Ultimately, I had to learn pretty heavily on examples from other kaggle notebooks to complete this assigment.

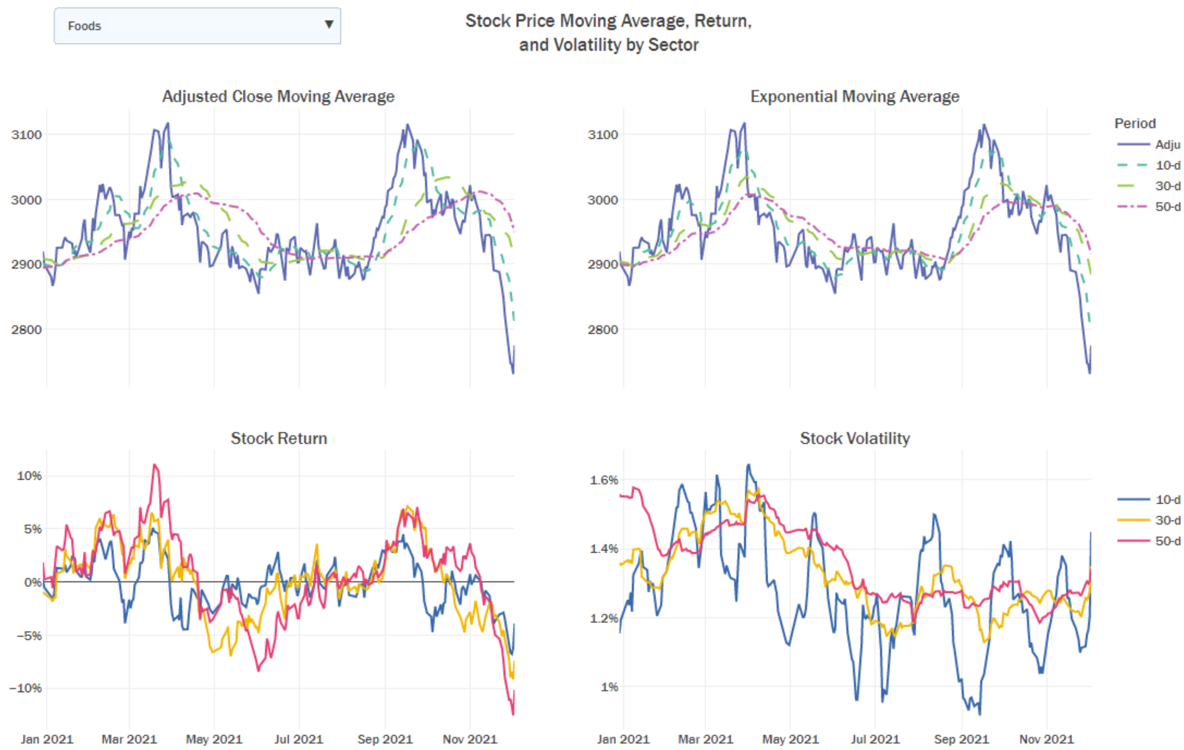
Lastly, I had difficulty understanding how to use kaggle notebooks and API for competition scoring.

4) Approach: Feature Engineering First, it should be noted that the Date and SecuritiesCode columns were only useful for facillitating data analysis and manipulation and likely do not contain any useful information for predictions. Therefore, they were later dropped when training the model. Next, its noted closing prices for some of the stocks were affected by splits or reverse splits which can cause problems for the model. Instead, adjusted closing prices were generated using a function from [2]. Then, a new set of features were engineered using a function from [3]. This function generates price moving average, exponential moving average, return, and volatility over periods of 5, 10, 20, 50 days. These features were plotted by sector. As pointed out in [3], for both moving average and exponential moving average, when the 10 day average crosses the 50 day from below, the closing price increases. This is commonly regarded by investors as a buy signal [4].

```
def adjust_price(price):
    """
    Args:
        price (pd.DataFrame)  : pd.DataFrame include stock_price
    Returns:
        price DataFrame (pd.DataFrame): stock_price with generated
AdjustedClose
    """
    # transform Date column into datetime
    price.loc[: ,"Date"] = pd.to_datetime(price.loc[: ,"Date"], format="%Y-
%m-%d")

    def generate_adjusted_close(df):
        """
        Args:
            df (pd.DataFrame)  : stock_price for a single SecuritiesCode
        Returns:
            df (pd.DataFrame): stock_price with AdjustedClose for a single
SecuritiesCode
        """
        # sort data to generate CumulativeAdjustmentFactor
        df = df.sort_values("Date", ascending=False)
        # generate CumulativeAdjustmentFactor
        df.loc[:, "CumulativeAdjustmentFactor"] =
df["AdjustmentFactor"].cumprod()
        # generate AdjustedClose
        df.loc[:, "AdjustedClose"] = (
            df["CumulativeAdjustmentFactor"] * df["Close"]
        ).map(lambda x: float(
            Decimal(str(x)).quantize(Decimal('0.1'), rounding=ROUND_HALF_UP)
        ))
        # reverse order
        df = df.sort_values("Date")
        # to fill AdjustedClose, replace 0 into np.nan
        df.loc[df["AdjustedClose"] == 0, "AdjustedClose"] = np.nan
        # forward fill AdjustedClose
        df.loc[:, "AdjustedClose"] = df.loc[:, "AdjustedClose"].ffill()
        return df

    # generate AdjustedClose
```

```
    price = price.sort_values(["SecuritiesCode", "Date"])
    price =
price.groupby("SecuritiesCode").apply(generate_adjusted_close).reset_index(dr
op=True)
    return price

train=train.drop('ExpectedDividend',axis=1).fillna(0)
prices=adjust_price(train)

def create_features(df):
    df=df.copy()
    col='AdjustedClose'
    periods=[5,10,20,30,50]
    for period in periods:
        df.loc[:,"Return_{}Day".format(period)] =
df.groupby("SecuritiesCode")[col].pct_change(period)
        df.loc[:,"MovingAvg_{}Day".format(period)] =
df.groupby("SecuritiesCode")[col].rolling(window=period).mean().values
        df.loc[:,"ExpMovingAvg_{}Day".format(period)] =
df.groupby("SecuritiesCode")[col].ewm(span=period,adjust=False).mean().values
        df.loc[:,"Volatility_{}Day".format(period)] =
np.log(df[col]).groupby(df["SecuritiesCode"]).diff().rolling(period).std()
    return df

price_features=create_features(df=prices)
price_features.drop(['RowId','SupervisionFlag','AdjustmentFactor','Cumulative
AdjustmentFactor','Close'],axis=1,inplace=True)

price_names=price_features.merge(stock_list[['SecuritiesCode','Name','SectorN
ame']], on='SecuritiesCode').set_index('Date')
price_names=price_names[price_names.index>='2020-12-29']
price_names.fillna(0, inplace=True)

features=['MovingAvg','ExpMovingAvg','Return', 'Volatility']
names=['Average', 'Exp. Moving Average', 'Period', 'Volatility']
buttons=[]

fig = make_subplots(rows=2, cols=2,
                    shared_xaxes=True,
                    vertical_spacing=0.1,
                    subplot_titles=('Adjusted Close Moving Average',
                                    'Exponential Moving Average',
                                    'Stock Return', 'Stock Volatility'))

for i, sector in enumerate(price_names.SectorName.unique()):

    sector_df=price_names[price_names.SectorName==sector]
    periods=[0,10,30,50]
    colors=px.colors.qualitative.Vivid
    dash=['solid','dash', 'longdash', 'dashdot', 'longdashdot']
    row,col=1,1

    for j, (feature, name) in enumerate(zip(features, names)):
        if j>=2:
            row,periods=2,[10,30,50]
            colors=px.colors.qualitative.Bold[1:]
        if j%2==0:
```

```
            col=1
        else:
            col=2

        for k, period in enumerate(periods):
            if (k==0)&(j<2):

plot_data=sector_df.groupby(sector_df.index)['AdjustedClose'].mean().rename('
Adjusted Close')
            elif j>=2:

plot_data=sector_df.groupby(sector_df.index)['{}_{}Day'.format(feature,period
)].mean().mul(100).rename('{}-day {}'.format(period,name))
            else:

plot_data=sector_df.groupby(sector_df.index)['{}_{}Day'.format(feature,period
)].mean().rename('{}-day {}'.format(period,name))
            fig.add_trace(go.Scatter(x=plot_data.index, y=plot_data,
mode='lines',
                                     name=plot_data.name,
marker_color=colors[k+1],
                                     line=dict(width=2,dash=(dash[k] if j<2
else 'solid')),
                                     showlegend=(True if (j==0) or (j==2)
else False), legendgroup=row,
                                     visible=(False if i != 0 else True)),
row=row, col=col)

    visibility=[False]*14*len(price_names.SectorName.unique())
    for l in range(i*14, i*14+14):
        visibility[l]=True
    button = dict(label = sector,
                  method = "update",
                  args=[{"visible": visibility}])
    buttons.append(button)

fig.update_layout(title='Stock Price Moving Average, Return,<br>and
Volatility by Sector',
                  template=temp, yaxis3_ticksuffix='%',
yaxis4_ticksuffix='%',
                  legend_title_text='Period', legend_tracegroupgap=250,
                  updatemenus=[dict(active=0, type="dropdown",
                                    buttons=buttons, xanchor='left',
                                    yanchor='bottom', y=1.105, x=.01)],
                  hovermode='x unified', height=800,width=1200,
margin=dict(t=150))
fig.show()
```



4)Approach: Price Prediction As noted in [6], the competition is scored by Sharpe Ratio where the score is average returns divided by standard devation. This means the model needs to account for investment risk over the course of the competition rahter than trying to optimize for massive returns on specific days. To evaluate model performance, an implementation from [6] for the sharpe ratio was used. The LightGBM regressor was used to generate target predictions. A method adapted from [3] was used to perform cross validation. The model was cross validated by k-Fold cross validation. k = 10 folds were used.

```
def calc_spread_return_sharpe(df: pd.DataFrame, portfolio_size: int = 200,
toprank_weight_ratio: float = 2) -> float:
    """
    Args:
        df (pd.DataFrame): predicted results
        portfolio_size (int): # of equities to buy/sell
        toprank_weight_ratio (float): the relative weight of the most highly
ranked stock compared to the least.
    Returns:
        (float): sharpe ratio
    """
    def _calc_spread_return_per_day(df, portfolio_size,
toprank_weight_ratio):
        """
        Args:
            df (pd.DataFrame): predicted results
            portfolio_size (int): # of equities to buy/sell
            toprank_weight_ratio (float): the relative weight of the most
highly ranked stock compared to the least.
        Returns:
```

```
            (float): spread return
        """
        assert df['Rank'].min() == 0
        assert df['Rank'].max() == len(df['Rank']) - 1
        weights = np.linspace(start=toprank_weight_ratio, stop=1,
num=portfolio_size)
        purchase = (df.sort_values(by='Rank')['Target'][:portfolio_size] *
weights).sum() / weights.mean()
        short = (df.sort_values(by='Rank',
ascending=False)['Target'][:portfolio_size] * weights).sum() / weights.mean()
        return purchase - short

    buf = df.groupby('Date').apply(_calc_spread_return_per_day,
portfolio_size, toprank_weight_ratio)
    sharpe_ratio = buf.mean() / buf.std()
    return sharpe_ratio

    ts_fold = TimeSeriesSplit(n_splits=10, gap=10000)
prices=price_features.dropna().sort_values(['Date','SecuritiesCode'])
y=prices['Target'].to_numpy()
X=prices.drop(['Target'],axis=1)

feat_importance=pd.DataFrame()
sharpe_ratio=[]

for fold, (train_idx, val_idx) in enumerate(ts_fold.split(X, y)):

    print("\n========================== Fold {}
==========================".format(fold+1))
    X_train, y_train = X.iloc[train_idx,:], y[train_idx]
    X_valid, y_val = X.iloc[val_idx,:], y[val_idx]

    print("Train Date range: {} to
{}".format(X_train.Date.min(),X_train.Date.max()))
    print("Valid Date range: {} to
{}".format(X_valid.Date.min(),X_valid.Date.max()))

    X_train.drop(['Date','SecuritiesCode'], axis=1, inplace=True)

X_val=X_valid[X_valid.columns[~X_valid.columns.isin(['Date','SecuritiesCode']
)]]
    val_dates=X_valid.Date.unique()[1:-1]
    print("\nTrain Shape: {} {}, Valid Shape: {} {}".format(X_train.shape,
y_train.shape, X_val.shape, y_val.shape))

    params = {'n_estimators': 500,
              'num_leaves' : 100,
              'learning_rate': 0.1,
              'colsample_bytree': 0.9,
              'subsample': 0.8,
              'reg_alpha': 0.4,
              'metric': 'mae',
              'random_state': 21}

    gbm = LGBMRegressor(**params).fit(X_train, y_train,
                                      eval_set=[(X_train, y_train), (X_val,
y_val)],
```

```
                                              verbose=300,
                                              eval_metric=['mae','mse'])
    y_pred = gbm.predict(X_val)
    rmse = np.sqrt(mean_squared_error(y_val, y_pred))
    mae = mean_absolute_error(y_val, y_pred)
    feat_importance["Importance_Fold"+str(fold)]=gbm.feature_importances_
    feat_importance.set_index(X_train.columns, inplace=True)

    rank=[]
    X_val_df=X_valid[X_valid.Date.isin(val_dates)]
    for i in X_val_df.Date.unique():
        temp_df = X_val_df[X_val_df.Date ==
i].drop(['Date','SecuritiesCode'],axis=1)
        temp_df["pred"] = gbm.predict(temp_df)
        temp_df["Rank"] = (temp_df["pred"].rank(method="first",
ascending=False)-1).astype(int)
        rank.append(temp_df["Rank"].values)

    stock_rank=pd.Series([x for y in rank for x in y], name="Rank")
    df=pd.concat([X_val_df.reset_index(drop=True),stock_rank,

prices[prices.Date.isin(val_dates)]['Target'].reset_index(drop=True)],
axis=1)
    sharpe=calc_spread_return_sharpe(df)
    sharpe_ratio.append(sharpe)
    print("Valid Sharpe: {}, RMSE: {}, MAE: {}".format(sharpe,rmse,mae))

    del X_train, y_train,  X_val, y_val
    gc.collect()

print("\nAverage cross-validation Sharpe Ratio: {:.4f}, standard deviation =
{:.2f}.".format(np.mean(sharpe_ratio),np.std(sharpe_ratio)))
```

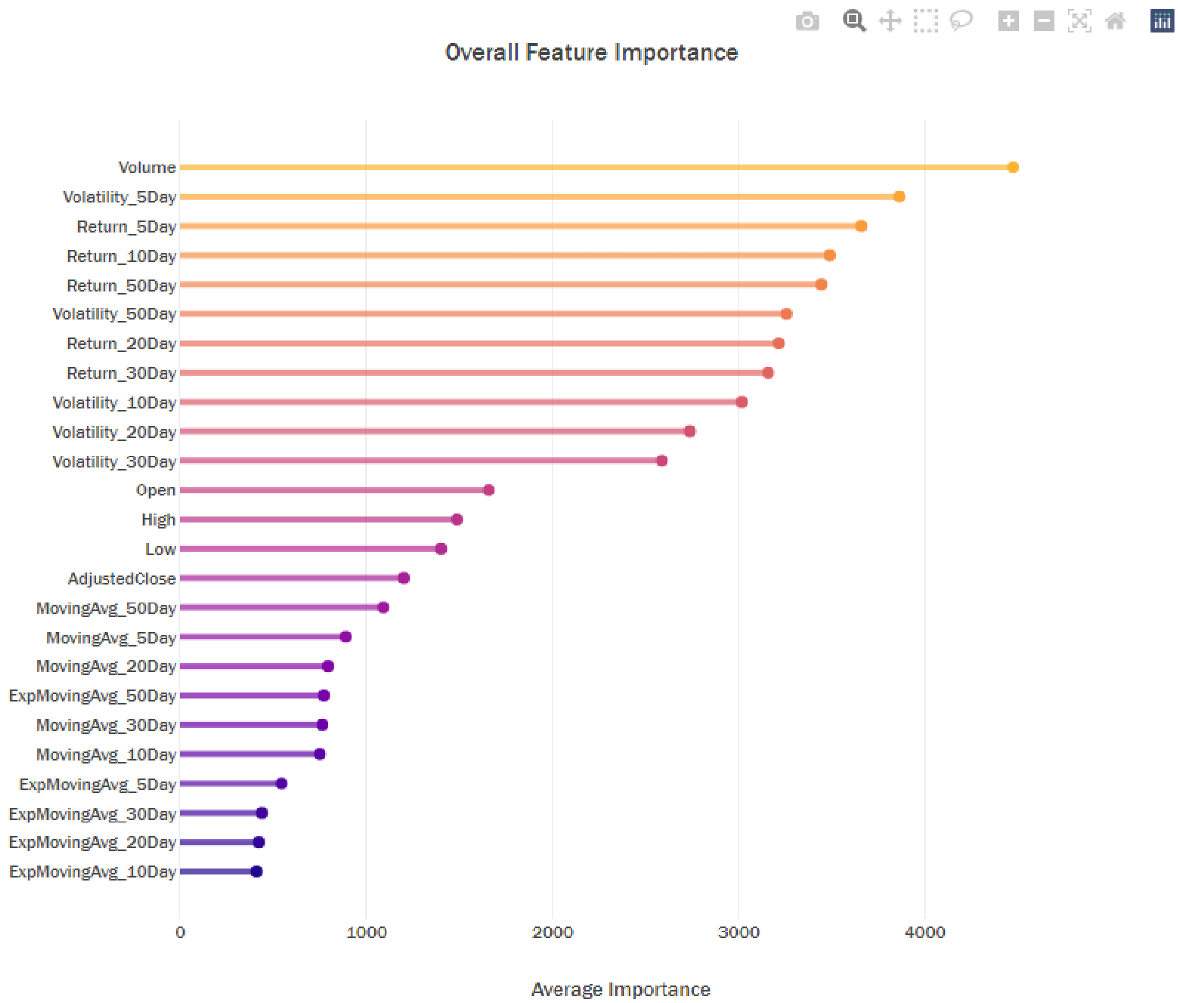
Output:

```
========================== Fold 1 ==========================
Train Date range: 2017-03-16 00:00:00 to 2017-08-16 00:00:00
Valid Date range: 2017-08-23 00:00:00 to 2018-02-01 00:00:00

Train Shape: (192937, 25) (192937,), Valid Shape: (202933, 25) (202933,)
[300]   training's l2: 0.000284428      training's l1: 0.011252 valid_1's l2: 0.0
00397147        valid_1's l1: 0.0127853
Valid Sharpe: 0.2783291308355168, RMSE: 0.02000764254268327, MAE: 0.0128614872306
26002

========================== Fold 2 ==========================
Train Date range: 2017-03-16 00:00:00 to 2018-01-25 00:00:00
Valid Date range: 2018-02-01 00:00:00 to 2018-07-09 00:00:00

Train Shape: (395870, 25) (395870,), Valid Shape: (202933, 25) (202933,)
[300]   training's l2: 0.000312683      training's l1: 0.0116643        valid_1's
l2: 0.000528399 valid_1's l1: 0.015333
Valid Sharpe: 0.09079768727593772, RMSE: 0.02304624892972341, MAE: 0.015388213618
88036
```

```
========================== Fold 3 ==========================
Train Date range: 2017-03-16 00:00:00 to 2018-07-02 00:00:00
Valid Date range: 2018-07-09 00:00:00 to 2018-12-11 00:00:00

Train Shape: (598803, 25) (598803,), Valid Shape: (202933, 25) (202933,)
[300]   training's l2: 0.00037429      training's l1: 0.0127419       valid_1's
l2: 0.000585011 valid_1's l1: 0.0163691
Valid Sharpe: 0.17050060566706488, RMSE: 0.02424558589860348, MAE: 0.016417873592
566138

========================== Fold 4 ==========================
Train Date range: 2017-03-16 00:00:00 to 2018-12-04 00:00:00
Valid Date range: 2018-12-11 00:00:00 to 2019-05-28 00:00:00

Train Shape: (801736, 25) (801736,), Valid Shape: (202933, 25) (202933,)
[300]   training's l2: 0.000417895     training's l1: 0.0135345       valid_1's
l2: 0.000638743 valid_1's l1: 0.0169478
Valid Sharpe: 0.06401325529221698, RMSE: 0.02532078754333139, MAE: 0.016989655270
28628

========================== Fold 5 ==========================
Train Date range: 2017-03-16 00:00:00 to 2019-05-21 00:00:00
Valid Date range: 2019-05-28 00:00:00 to 2019-10-29 00:00:00

Train Shape: (1004669, 25) (1004669,), Valid Shape: (202933, 25) (202933,)
[300]   training's l2: 0.000455933     training's l1: 0.0141845       valid_1's
l2: 0.000415519 valid_1's l1: 0.0140295
Valid Sharpe: 0.1467098027948083, RMSE: 0.02041449038353124, MAE: 0.0140558741306
10506

========================== Fold 6 ==========================
Train Date range: 2017-03-16 00:00:00 to 2019-10-21 00:00:00
Valid Date range: 2019-10-29 00:00:00 to 2020-04-03 00:00:00

Train Shape: (1207602, 25) (1207602,), Valid Shape: (202933, 25) (202933,)
[300]   training's l2: 0.000447895     training's l1: 0.0141511       valid_1's
l2: 0.000952124 valid_1's l1: 0.0202303
Valid Sharpe: 0.042973110086945106, RMSE: 0.030936045010883293, MAE: 0.0202777592
47045463

========================== Fold 7 ==========================
Train Date range: 2017-03-16 00:00:00 to 2020-03-27 00:00:00
Valid Date range: 2020-04-03 00:00:00 to 2020-09-04 00:00:00

Train Shape: (1410535, 25) (1410535,), Valid Shape: (202933, 25) (202933,)
[300]   training's l2: 0.000490876     training's l1: 0.0147284       valid_1's
l2: 0.000750551 valid_1's l1: 0.0190318
Valid Sharpe: -0.007587986114749248, RMSE: 0.02746336119533849, MAE: 0.0190771041
4387534

========================== Fold 8 ==========================
Train Date range: 2017-03-16 00:00:00 to 2020-08-28 00:00:00
```

```
Valid Date range: 2020-09-04 00:00:00 to 2021-02-04 00:00:00

Train Shape: (1613468, 25) (1613468,), Valid Shape: (202933, 25) (202933,)
[300]   training's l2: 0.000524328      training's l1: 0.0153064       valid_1's
l2: 0.000558809 valid_1's l1: 0.0161323
Valid Sharpe: 0.13456058197181694, RMSE: 0.023665441317334886, MAE: 0.01614935809
6265113

========================== Fold 9 ==========================
Train Date range: 2017-03-16 00:00:00 to 2021-01-28 00:00:00
Valid Date range: 2021-02-04 00:00:00 to 2021-07-06 00:00:00

Train Shape: (1816401, 25) (1816401,), Valid Shape: (202933, 25) (202933,)
[300]   training's l2: 0.000526032      training's l1: 0.0153737       valid_1's
l2: 0.000462331 valid_1's l1: 0.0149358
Valid Sharpe: 0.4234131130065347, RMSE: 0.021519001386825262, MAE: 0.014948005962
151227

========================== Fold 10 ==========================
Train Date range: 2017-03-16 00:00:00 to 2021-06-29 00:00:00
Valid Date range: 2021-07-06 00:00:00 to 2021-12-03 00:00:00

Train Shape: (2019334, 25) (2019334,), Valid Shape: (202933, 25) (202933,)
[300]   training's l2: 0.000520921      training's l1: 0.0153538       valid_1's
l2: 0.000494375 valid_1's l1: 0.01515
Valid Sharpe: -0.053196914301438754, RMSE: 0.022253027141850953, MAE: 0.015160728
372451873

Average cross-validation Sharpe Ratio: 0.1291, standard deviation = 0.13.
```

```
feat_importance['avg'] = feat_importance.mean(axis=1)
feat_importance = feat_importance.sort_values(by='avg',ascending=True)
pal=sns.color_palette("plasma_r", 29).as_hex()[2:]

fig=go.Figure()
for i in range(len(feat_importance.index)):
    fig.add_shape(dict(type="line", y0=i, y1=i, x0=0,
x1=feat_importance['avg'][i],
                       line_color=pal[::-1][i],opacity=0.7,line_width=4))
fig.add_trace(go.Scatter(x=feat_importance['avg'], y=feat_importance.index,
mode='markers',
                         marker_color=pal[::-1], marker_size=8,
                         hovertemplate='%{y} Importance =
%{x:.0f}<extra></extra>'))
fig.update_layout(template=temp,title='Overall Feature Importance',
                  xaxis=dict(title='Average Importance',zeroline=False),
                  yaxis_showgrid=False, margin=dict(l=120,t=80),
                  height=700, width=800)
fig.show()
```



5) Evaluation and Summary Submissions were submitted to the Kaggle competition and were evaluated by the Sharpe Ratio [5] which essentially measures the risk adjusted performance of a security compared to a risk free asset. It is the difference between the returns of an investment and a risk free return dividied by the standard deviation of investment returns. On a daily basis, the top 200 and bottom 200 performing stocks are to be predicted in order. For the top 200 ranked stocks, returns are calculated as if these stocks were purchased at opening price. For the bottom 200, returns are calculated as if these stocks were shorted at opening price. Then, these returns are weighted based on their ranking and the total portfolio return is calculated as if the stocks were purchased the next day and sold on the third day. A more detailed description of the evaluation metric is detailed at [6].

Kaggle Submission Score: 0.246

Due to time contraints, I was unable to tune hyperparameters for the LightGBM regressor which is likely a large contributor to approximation error.

As seen in the plots below, volume was the most important feature followed by the engineered 5 day volatility and 5 day return features. Interestingly, the moving average and exponential moving average features were by far the least important features.

Overall this approach seemed to work well. The engineered volatility and return features were given very high importance weights compared to the provided features for open, close, and high prices. By comparison, the engineered moving average and exponential moving average features were given low importance so they did not help the model much. I think the main limitations with the approach taken here is due to the quantity/quality of engineered features. For example, Relative Strength Index (RSI) is a commonly used momentum indicator that looks at the magnitute of price changes to evaluate if a security is overbought or oversold [7]. Bolinger Bands are a commonly used indicator of volatility by setting a threshold based on standard deviation and movin average [8]. Volume-Weighted Average Price (VWAP) represents the average trading price of a security based on both volume and price [9]. All of these metrics could potentially be feature engineered to improve this model, however I did not have time for implementation. Lastly, as mentioned above, this approach likely suffers from a lack of tuning of hyperparameters and model selection. I only tested the LightGBM regressor, but it's possible that a different regressor could yield netter performance here.

6) What I learned I learned a lot more about EDA and how to generate effective plots using plotly from this project. Unfortunately,I had to learn pretty heavily on examples from [3] and [10]. However, I now know how to use these skills for the future!

```
cols_fin=feat_importance.avg.nlargest(3).index.tolist()
cols_fin.extend(('Open','High','Low'))
X_train=prices[cols_fin]
y_train=prices['Target']
gbm = LGBMRegressor(**params).fit(X_train, y_train)

import jpx_tokyo_market_prediction
env = jpx_tokyo_market_prediction.make_env()
iter_test = env.iter_test()

cols=['Date','SecuritiesCode','Open','High','Low','Close','Volume','Adjustmen
tFactor']
train=train[train.Date>='2021-08-01'][cols]

counter = 0
for (prices, options, financials, trades, secondary_prices,
sample_prediction) in iter_test:

    current_date = prices["Date"].iloc[0]
    if counter == 0:
        df_price_raw = train.loc[train["Date"] < current_date]
    df_price_raw = pd.concat([df_price_raw,
prices[cols]]).reset_index(drop=True)
    df_price = adjust_price(df_price_raw)
    features = create_features(df=df_price)
    feat = features[features.Date == current_date][cols_fin]
    feat["pred"] = gbm.predict(feat)
    feat["Rank"] = (feat["pred"].rank(method="first", ascending=False)-
1).astype(int)
    sample_prediction["Rank"] = feat["Rank"].values
    display(sample_prediction.head())

    assert sample_prediction["Rank"].notna().all()
    assert sample_prediction["Rank"].min() == 0
```

```
    assert sample_prediction["Rank"].max() == len(sample_prediction["Rank"])
- 1

    env.predict(sample_prediction)
    counter += 1
```

	Date	SecuritiesCode	Rank
0	2021-12-06	1301	590
1	2021-12-06	1332	1257
2	2021-12-06	1333	852
3	2021-12-06	1375	1658
4	2021-12-06	1376	1313

	Date	SecuritiesCode	Rank
0	2021-12-07	1301	1306
1	2021-12-07	1332	112
2	2021-12-07	1333	298
3	2021-12-07	1375	1186
4	2021-12-07	1376	1092

Citations:

[1] https://www.kaggle.com/competitions/jpx-tokyo-stock-exchange-prediction/overview

[2] https://www.kaggle.com/code/smeitoma/train-demo#Generating-AdjustedClose-price

[3] https://www.kaggle.com/code/kellibelcher/jpx-stock-market-analysis-prediction-with-lgbm/notebook

[4] https://www.investopedia.com/articles/active-trading/052014/how-use-moving-average-buy-stocks.asp

[5] https://en.wikipedia.org/wiki/Sharpe_ratio

[6] https://www.kaggle.com/code/smeitoma/jpx-competition-metric-definition

[7] https://www.investopedia.com/terms/r/rsi.asp

[8] https://www.fidelity.com/learning-center/trading-investing/technical-analysis/technical-indicator-guide/bollinger-bands#:~:text=Bollinger%20Bands%20are%20envelopes%20plotted,Period%20and%20Standard%20Deviations%2C%20StdDev.

[9] https://www.investopedia.com/terms/v/vwap.asp#:~:text=The%20volume%2Dweighted%20average%20price%20(VWAP)%20is%20a%20technical,on%20both%20volume%20and%20price.

[10] https://www.kaggle.com/code/abaojiang/jpx-detailed-eda/notebook