**Python for Finance**

|  |  |
| --- | --- |
| **Finance** | |
| Financial Statements | #Gross Profit (Direct sales and costs) = Sales – Cost of goods sold (Material + Labour + Factory)  #Sales = Income = Revenue = Turnover  #Sales = Sales price per unit \* Number of units sold – (Discounts + Credit Sales [Income not received upfront] + Sales Mix [different pricing])  #COGS = Total fixed costs + Variable costs per unit + (Inventory Opening Balance – Inventory Closing Balance)  #Gross Profit Margin =  #Net Profit (Indirect income and expenses) = Gross Profit – Total Operating Expenses (Admin + Marketing + R&D + Training + Insurance + Office)  #Break Even Point = |
| Balance Sheet | #Asset: Economic resource used to make money (e.g. House, Cash in banks, Inventory)  #Liability: Economic obligation to pay for something (e.g. Mortgage, Loans)  #Equity = Assets – Liabilities (i.e. Owner’s interest / capital)  #Sales on credit under accounts receivable (debtors)  #Purchase on credit (liability to pay at later date) under accounts payable (creditor: gross profit or net profit)  #Debtor days ratio = (Lower is better)  #Days payable outstanding ratio = (Higher is better)  #Days in inventory ratio = (Higher may be better)  #Asset turnover ratio = (Higher may be better but depends on industry) |
| Investment Risks | #Risk: Measure of uncertainty of future returns (Variance of financial returns: Standard deviation, Kurtosis, Skewness)  #Discrete returns (Simple returns): Periodic price movements, aggregate across assets  #Log returns (Continuous compounding): Academic research and modelling, aggregate across time (Smaller than discrete returns) )  #Volatility (Standard deviation)  #Kurtosis for probability of outliers (>0 non-normality): Financial returns are leptokurtic (positive excess kurtosis of above 3) that is high risk  # Import numpy as np  import numpy as np  # Calculate the average daily return of the stock  mean\_return\_daily = np.mean(StockPrices['Returns'])  print(mean\_return\_daily)  # Calculate the implied annualized average return  mean\_return\_annualized = ((1 + mean\_return\_daily)\*\*252) - 1  print(mean\_return\_annualized)  # Calculate the standard deviation of daily return of the stock  sigma\_daily = np.std(StockPrices['Returns'])  print(sigma\_daily)  # Calculate the daily variance  variance\_daily = sigma\_daily\*\*2  print(variance\_daily)  # Annualize the standard deviation  sigma\_annualized = sigma\_daily\*np.sqrt(252)  print(sigma\_annualized)  # Calculate the annualized variance  variance\_annualized = sigma\_annualized\*\*2  print(variance\_annualized)  # Import skew from scipy.stats  from scipy.stats import skew  # Drop the missing values  clean\_returns = StockPrices['Returns'].dropna()  # Calculate the third moment (skewness) of the returns distribution  returns\_skewness = skew(clean\_returns)  print(returns\_skewness)  # Import kurtosis from scipy.stats  from scipy.stats import kurtosis  # Calculate the excess kurtosis of the returns distribution  excess\_kurtosis = kurtosis(clean\_returns)  print(excess\_kurtosis)  # Derive the true fourth moment of the returns distribution  fourth\_moment = excess\_kurtosis + 3  print(fourth\_moment)  # Import shapiro from scipy.stats  from scipy.stats import shapiro  # Run the Shapiro-Wilk test on the stock returns  shapiro\_results = shapiro(clean\_returns)  print("Shapiro results:", shapiro\_results)  # Extract the p-value from the shapiro\_results  p\_value = shapiro\_results[1]  print("P-value: ", p\_value)  # Finish defining the portfolio weights as a numpy array  portfolio\_weights = np.array([0.12, 0.15, 0.08, 0.05, 0.09, 0.10, 0.11, 0.14, 0.16])  # Calculate the weighted stock returns  WeightedReturns = StockReturns.mul(portfolio\_weights, axis=1)  # Calculate the portfolio returns  StockReturns['Portfolio'] = WeightedReturns.sum(axis=1)  # Plot the cumulative portfolio returns over time  CumulativeReturns = ((1+StockReturns["Portfolio"]).cumprod()-1)  CumulativeReturns.plot()  plt.show()  # How many stocks are in your portfolio?  numstocks = 9  # Create an array of equal weights across all assets  portfolio\_weights\_ew = np.repeat(1 / numstocks, numstocks)  # Calculate the equally-weighted portfolio returns  StockReturns['Portfolio\_EW'] = StockReturns.iloc[:,0:numstocks].mul(portfolio\_weights\_ew, axis=1).sum(axis=1)  cumulative\_returns\_plot(['Portfolio', 'Portfolio\_EW'])  # Create an array of market capitalizations (in billions)  market\_capitalizations = np.array([601.51, 469.25, 349.5, 310.48, 299.77, 356.94, 268.88, 331.57, 246.09])  # Calculate the market cap weights  mcap\_weights = market\_capitalizations / sum(market\_capitalizations)  # Calculate the market cap weighted portfolio returns  StockReturns['Portfolio\_MCap'] = StockReturns.iloc[:, 0:9].mul(mcap\_weights, axis=1).sum(axis=1)  cumulative\_returns\_plot(['Portfolio', 'Portfolio\_EW', 'Portfolio\_MCap'])  #Portfolio Standard Deviation  #Correlation: Normalized measure of covariance  #Covariance: Joint variability of 2 random variables, used for portfolio optimization and risk management |
| **Exploratory Data Analysis** | |
| EDA | import pandas as pd  print(lng\_df.head()) # examine the DataFrames  print(spy\_df.head())  # Plot the Adj\_Close columns for SPY and LNG  spy\_df['Adj\_Close'].plot(label='SPY', legend=True)  lng\_df['Adj\_Close'].plot(label='LNG', legend=True, secondary\_y=True)  plt.show() # show the plot  plt.clf() # clear the plot space  # Histogram of the daily price change percent of Adj\_Close for LNG  lng\_df['Adj\_Close'].pct\_change().plot.hist(bins=50)  plt.xlabel('Adjusted close 1-day percent change')  plt.show() |
| **Correlation, Moving Averages, RSI** | |
| Correlation | # Create 5-day % changes of Adj\_Close for the current day, and 5 days in the future  lng\_df['5d\_future\_close'] = lng\_df['Adj\_Close'].shift(-5)  lng\_df['5d\_close\_future\_pct'] = lng\_df['5d\_future\_close'].pct\_change(5)  lng\_df['5d\_close\_pct'] = lng\_df['Adj\_Close'].pct\_change(5)  # Calculate the correlation matrix between the 5d close pecentage changes (current and future)  corr = lng\_df[['5d\_close\_pct', '5d\_close\_future\_pct']].corr()  print(corr)  # Scatter the current 5-day percent change vs the future 5-day percent change  plt.scatter(lng\_df['5d\_close\_pct'], lng\_df['5d\_close\_future\_pct'])  plt.show() |
| Covariance | # Calculate the covariance matrix  cov\_mat = StockReturns.cov()  # Annualize the co-variance matrix  cov\_mat\_annual = cov\_mat \* 252  # Print the annualized co-variance matrix  print(cov\_mat\_annual)  # Import numpy as np  import numpy as np  # Calculate the portfolio standard deviation  portfolio\_volatility = np.sqrt(np.dot(portfolio\_weights.T, np.dot(cov\_mat\_annual, portfolio\_weights)))  print(portfolio\_volatility) |
| Moving Averages  (14, 50, 200 days for stocks) | import TAlib  feature\_names = ['5d\_close\_pct'] # a list of the feature names for later  # Create moving averages and rsi for timeperiods of 14, 30, 50, and 200  for n in [14, 30, 50, 200]:  # Create the moving average indicator and divide by Adj\_Close  lng\_df['ma' + str(n)] = talib.SMA(lng\_df['Adj\_Close'].values,  timeperiod=n) / lng\_df['Adj\_Close']  # Create the RSI indicator  lng\_df['rsi' + str(n)] = talib.RSI(lng\_df['Adj\_Close'].values, timeperiod=n)    # Add rsi and moving average to the feature name list  feature\_names = feature\_names + ['ma' + str(n), 'rsi' + str(n)]  print(feature\_names)  # Drop all na values  lng\_df = lng\_df.dropna()  # Create features and targets  # use feature\_names for features; '5d\_close\_future\_pct' for targets  features = lng\_df[feature\_names]  targets = lng\_df['5d\_close\_future\_pct']  # Create DataFrame from target column and feature columns  feature\_and\_target\_cols = ['5d\_close\_future\_pct'] + feature\_names  feat\_targ\_df = lng\_df[feature\_and\_target\_cols]  # Calculate correlation matrix  corr = feat\_targ\_df.corr()  print(corr)  #Import seaborn package  import seaborn as sns  # Plot heatmap of correlation matrix  sns.heatmap(corr,  annot=True,  cmap="YlGnBu",  linewidths=0.3,  annot\_kws={"size": 8})  plt.yticks(rotation=0); plt.xticks(rotation=90) # fix ticklabel directions  plt.tight\_layout() # fits plot area to the plot, "tightly"  plt.show() # show the plot  plt.clf() # clear the plot area  # Create a scatter plot of the most highly correlated variable with the target  plt.scatter(lng\_df['ma200'], lng\_df['5d\_close\_future\_pct'])  plt.show() |
| Relative Strength Index (RSI) | #RSI =  #[0, 100]  #0: price due to rebound from recent lows  #100: price due to decline |
| **Weighted Probability** | |
| Weighted Probability | # Create the combined list for sales and probability  sales\_probability = ['0|0.05', '200|0.1', '300|0.4', '500|0.2', '800|0.25']  weighted\_probability = 0  # Create a for loop to calculate the weighted probability  for pair in sales\_probability:  parts = pair.split('|')  weighted\_probability += float(parts[0]) \* float(parts[1])  # Print the weighted probability result  print("The weighted probability is {}.".format(weighted\_probability)) |
| **Linear Regression** | |
| Linear model | # Import the statsmodels.api library with the alias sm  import statsmodels.api as sm  # Add a constant to the features  linear\_features = sm.add\_constant(features)  # Create a size for the training set that is 85% of the total number of samples  train\_size = int(0.85 \* targets.shape[0])  train\_features = linear\_features[:train\_size]  train\_targets = targets[:train\_size]  test\_features = linear\_features[train\_size:]  test\_targets = targets[train\_size:]  print(linear\_features.shape, train\_features.shape, test\_features.shape)  # Create the linear model and complete the least squares fit  model = sm.OLS(train\_targets, train\_features)  results = model.fit() # fit the model  print(results.summary())  # examine pvalues  # Features with p <= 0.05 are typically considered significantly different from 0  print(results.pvalues)  # Make predictions from our model for train and test sets  train\_predictions = results.predict(train\_features)  test\_predictions = results.predict(test\_features)  #Import library  import matplotlib as plt  # Scatter the predictions vs the targets with 80% transparency  plt.scatter(train\_predictions, train\_targets, alpha=0.2, color='b', label='train')  plt.scatter(test\_predictions, test\_targets, alpha=0.2, color='r', label='test')  # Plot the perfect prediction line  xmin, xmax = plt.xlim()  plt.plot(np.arange(xmin, xmax, 0.01), np.arange(xmin, xmax, 0.01), c='k')  # Set the axis labels and show the plot  plt.xlabel('predictions')  plt.ylabel('actual')  plt.legend() # show the legend  plt.show() |
| Feature Engineering | # Create 2 new volume features, 1-day % change and 5-day SMA of the % change  new\_features = ['Adj\_Volume\_1d\_change', 'Adj\_Volume\_1d\_change\_SMA']  feature\_names.extend(new\_features)  lng\_df['Adj\_Volume\_1d\_change'] = lng\_df['Adj\_Volume'].pct\_change(1)  lng\_df['Adj\_Volume\_1d\_change\_SMA'] = talib.SMA(lng\_df['Adj\_Volume\_1d\_change'].values,  timeperiod=5)  # Plot histogram of volume % change data  lng\_df[new\_features].plot(kind='hist', sharex=False, bins=50)  plt.show()  # Use pandas' get\_dummies function to get dummies for day of the week  days\_of\_week = pd.get\_dummies(lng\_df.index.dayofweek,  prefix='weekday',  drop\_first=True)  # Set the index as the original dataframe index for merging  days\_of\_week.index = lng\_df.index  # Join the dataframe with the days of week dataframe  lng\_df = pd.concat([lng\_df, days\_of\_week], axis=1)  # Add days of week to feature names  feature\_names.extend(['weekday\_' + str(i) for i in range(1, 5)])  lng\_df.dropna(inplace=True) # drop missing values in-place  print(lng\_df.head())  # Add the weekday labels to the new\_features list  new\_features.extend(['weekday\_' + str(i) for i in range(1, 5)])  # Plot the correlations between the new features and the targets  sns.heatmap(lng\_df[new\_features + ['5d\_close\_future\_pct']].corr(), annot=True)  plt.yticks(rotation=0) # ensure y-axis ticklabels are horizontal  plt.xticks(rotation=90) # ensure x-axis ticklabels are vertical  plt.tight\_layout()  plt.show() |
| **Decision Tree** | |
| Decision Tree | from sklearn.tree import DecisionTreeRegressor  # Create a decision tree regression model with default arguments  decision\_tree = DecisionTreeRegressor()  # Fit the model to the training features and targets  decision\_tree.fit(train\_features, train\_targets)  # Check the score on train and test  print(decision\_tree.score(train\_features, train\_targets))  print(decision\_tree.score(test\_features, test\_targets))  # Loop through a few different max depths and check the performance  for d in [3, 5, 10]:  # Create the tree and fit it  decision\_tree = DecisionTreeRegressor(max\_depth = d)  decision\_tree.fit(train\_features, train\_targets)  # Print out the scores on train and test  print('max\_depth=', str(d))  print(decision\_tree.score(train\_features, train\_targets))  print(decision\_tree.score(test\_features, test\_targets), '\n')  # Use the best max\_depth of 3 from last exercise to fit a decision tree  decision\_tree = DecisionTreeRegressor(max\_depth = 3)  decision\_tree.fit(train\_features, train\_targets)  # Predict values for train and test  train\_predictions = decision\_tree.predict(train\_features)  test\_predictions = decision\_tree.predict(test\_features)  # Scatter the predictions vs actual values  plt.scatter(train\_predictions, train\_targets, label='train')  plt.scatter(test\_predictions, test\_targets, label='test')  plt.show() |
| Grid Search | from sklearn.model\_selection import ParameterGrid  # Create a dictionary of hyperparameters to search  grid = {'n\_estimators': [200], 'max\_depth': [3], 'max\_features': [4, 8], 'random\_state': [42]}  test\_scores = []  from pprint import pprint  pprint(list(ParameterGrid(grid)))  # Loop through the parameter grid, set the hyperparameters, and save the scores  for g in ParameterGrid(grid):  rfr.set\_params(\*\*g) # \*\* is "unpacking" the dictionary  rfr.fit(train\_features, train\_targets)  test\_scores.append(rfr.score(test\_features, test\_targets))  # Find best hyperparameters from the test score and print  best\_idx = np.argmax(test\_scores)  print(test\_scores[best\_idx], ParameterGrid(grid)[best\_idx])  # Use the best hyperparameters from before to fit a random forest model  rfr = RandomForestRegressor(n\_estimators=200, max\_depth=3, max\_features=4, random\_state=42)  rfr.fit(train\_features, train\_targets)  # Make predictions with our model  train\_predictions = rfr.predict(train\_features)  test\_predictions = rfr.predict(test\_features)  # Create a scatter plot with train and test actual vs predictions  plt.scatter(train\_targets, train\_predictions, label='train')  plt.scatter(test\_targets, test\_predictions, label='test')  plt.legend()  plt.show() |
| Random Forest | from sklearn.ensemble import RandomForestRegressor  # Create the random forest model and fit to the training data  rfr = RandomForestRegressor(n\_estimators=200)  rfr.fit(train\_features, train\_targets)  # Look at the R^2 scores on train and test  print(rfr.score(train\_features, train\_targets))  print(rfr.score(test\_features, test\_targets)) |
| Gradient Boosting | from sklearn.ensemble import GradientBoostingRegressor  # Create GB model -- hyperparameters have already been searched for you  gbr = GradientBoostingRegressor(max\_features=4,  learning\_rate=0.01,  n\_estimators=200,  subsample=0.6,  random\_state=42)  gbr.fit(train\_features, train\_targets)  print(gbr.score(train\_features, train\_targets))  print(gbr.score(test\_features, test\_targets)) |
| Feature Importance | # Get feature importances from our random forest model  importances = rfr.feature\_importances\_  # Get the index of importances from greatest importance to least  sorted\_index = np.argsort(importances)[::-1]  x = range(len(importances))  # Create tick labels  labels = np.array(feature\_names)[sorted\_index]  plt.bar(x, importances[sorted\_index], tick\_label=labels)  # Rotate tick labels to vertical  plt.xticks(rotation=90)  plt.show()  # Extract feature importances from the fitted gradient boosting model  feature\_importances = gbr.feature\_importances\_  # Get the indices of the largest to smallest feature importances  sorted\_index = np.argsort(feature\_importances)[::-1]  x = range(features.shape[1])  # Create tick labels  labels = np.array(feature\_names)[sorted\_index]  plt.bar(x, feature\_importances[sorted\_index], tick\_label=labels)  # Set the tick lables to be the feature names, according to the sorted feature\_idx  plt.xticks(rotation=90)  plt.show() |
| **KNN** | |
| Scaling | #Scaling options   * Min-Max * Standardization * Median – MAD * Sigmoid, tanh   from sklearn.preprocessing import scale  # Remove unimportant features (weekdays)  train\_features = train\_features.iloc[:, :-4]  test\_features = test\_features.iloc[:, :-4]  # Standardize the train and test features  scaled\_train\_features = scale(train\_features)  scaled\_test\_features = scale(test\_features)  # Plot histograms of the 14-day SMA RSI before and after scaling  f, ax = plt.subplots(nrows=2, ncols=1)  train\_features.iloc[:, 2].hist(ax=ax[0])  ax[1].hist(scaled\_train\_features[:, 2])  plt.show() |
| KNN | from sklearn.neighbors import KNeighborsRegressor  for n in range(2,13):  # Create and fit the KNN model  knn = KNeighborsRegressor(n\_neighbors=n)    # Fit the model to the training data  knn.fit(scaled\_train\_features, train\_targets)    # Print number of neighbors and the score to find the best value of n  print("n\_neighbors =", n)  print('train, test scores')  print(knn.score(scaled\_train\_features, train\_targets))  print(knn.score(scaled\_test\_features, test\_targets))  print() # prints a blank line  # Create the model with the best-performing n\_neighbors of 5  knn = KNeighborsRegressor(5)  # Fit the model  knn.fit(scaled\_train\_features, train\_targets)  # Get predictions for train and test sets  train\_predictions = knn.predict(scaled\_train\_features)  test\_predictions = knn.predict(scaled\_test\_features)  # Plot the actual vs predicted values  plt.scatter(train\_predictions, train\_targets, label='train')  plt.scatter(test\_predictions, test\_targets, label='test')  plt.legend()  plt.show() |
| **Neural Network** | |
| Neural Network | from keras.models import Sequential  from keras.layers import Dense  # Create the model  model\_1 = Sequential()  model\_1.add(Dense(100, input\_dim=scaled\_train\_features.shape[1], activation='relu'))  model\_1.add(Dense(20, activation='relu'))  model\_1.add(Dense(1, activation='linear'))  # Fit the model  model\_1.compile(optimizer='adam', loss='mse')  history = model\_1.fit(scaled\_train\_features, train\_targets, epochs=25)  # Plot the losses from the fit  plt.plot(history.history['loss'])  # Use the last loss as the title  plt.title('loss:' + str(round(history.history['loss'][-1], 6)))  plt.show()  from sklearn.metrics import r2\_score  # Calculate R^2 score  train\_preds = model\_1.predict(scaled\_train\_features)  test\_preds = model\_1.predict(scaled\_test\_features)  print(r2\_score(train\_targets, train\_preds))  print(r2\_score(test\_targets, test\_preds))  # Plot predictions vs actual  plt.scatter(train\_preds, train\_targets, label='train')  plt.scatter(test\_preds, test\_targets, label='test')  plt.legend()  plt.show() |
| Dropout | from keras.layers import Dropout  # Create model with dropout  model\_3 = Sequential()  model\_3.add(Dense(100, input\_dim=scaled\_train\_features.shape[1], activation='relu'))  model\_3.add(Dropout(0.2))  model\_3.add(Dense(20, activation='relu'))  model\_3.add(Dense(1, activation='linear'))  # Fit model with mean squared error loss function  model\_3.compile(optimizer='adam', loss='mse')  history = model\_3.fit(scaled\_train\_features, train\_targets, epochs=25)  plt.plot(history.history['loss'])  plt.title('loss:' + str(round(history.history['loss'][-1], 6)))  plt.show() |
| Ensemble | # Make predictions from the 3 neural net models  train\_pred1 = model\_1.predict(scaled\_train\_features)  test\_pred1 = model\_1.predict(scaled\_test\_features)  train\_pred2 = model\_2.predict(scaled\_train\_features)  test\_pred2 = model\_2.predict(scaled\_test\_features)  train\_pred3 = model\_3.predict(scaled\_train\_features)  test\_pred3 = model\_3.predict(scaled\_test\_features)  # Horizontally stack predictions and take the average across rows  train\_preds = np.mean(np.hstack((train\_pred1, train\_pred2, train\_pred3)), axis=1)  test\_preds = np.mean(np.hstack((test\_pred1, test\_pred2, test\_pred3)), axis=1)  print(test\_preds[-5:])  from sklearn.metrics import r2\_score  # Evaluate the R^2 scores  print(r2\_score(train\_targets, train\_preds))  print(r2\_score(test\_targets, test\_preds))  # Scatter the predictions vs actual -- this one is interesting!  plt.scatter(train\_preds, train\_targets, label='train')  plt.scatter(test\_preds, test\_targets, label='test')  plt.legend(); plt.show() |
| **Custom loss functions** | |
| Custom loss function | import keras.losses  import tensorflow as tf  # Create loss function to penalize wrong direction  def sign\_penalty(y\_true, y\_predicted):  penalty = 100.  loss = tf.where(tf.less(y\_true \* y\_pred, 0), \  penalty \* tf.square(y\_true - y\_pred), \  tf.square(y\_true - y\_pred))  return tf.reduce\_mean(loss, axis=-1)  keras.losses.sign\_penalty = sign\_penalty # enable use of loss with keras  print(keras.losses.sign\_penalty)  # Create the model  model\_2 = Sequential()  model\_2.add(Dense(100, input\_dim=scaled\_train\_features.shape[1], activation='relu'))  model\_2.add(Dense(20, activation='relu'))  model\_2.add(Dense(1, activation='linear'))  # Fit the model with our custom 'sign\_penalty' loss function  model\_2.compile(optimizer='adam', loss=sign\_penalty)  history = model\_2.fit(scaled\_train\_features, train\_targets, epochs=25)  plt.plot(history.history['loss'])  plt.title('loss:' + str(round(history.history['loss'][-1], 6)))  plt.show()  # Evaluate R^2 scores  train\_preds = model\_2.predict(scaled\_train\_features)  test\_preds = model\_2.predict(scaled\_test\_features)  print(r2\_score(train\_targets, train\_preds))  print(r2\_score(test\_targets, test\_preds))  # Scatter the predictions vs actual -- this one is interesting!  plt.scatter(train\_preds, train\_targets, label='train')  plt.scatter(test\_preds, test\_targets, label='test') # plot test set  plt.legend(); plt.show() |
| **Modern Portfolio Theory** | |
| Modern Portfolio Theory | #Y-axis: % change in returns of stock  #X-axis: Standard deviation of risks |
| Modern Portfolio Theory | # Join 3 stock dataframes together  full\_df = pd.concat([lng\_df, spy\_df, smlv\_df], axis=1).dropna()  # Resample the full dataframe to monthly timeframe  monthly\_df = full\_df.resample('BMS').first() #Business Month Start  # Calculate daily returns of stocks  returns\_daily = full\_df.pct\_change()  # Calculate monthly returns of the stocks  returns\_monthly = monthly\_df.pct\_change().dropna()  print(returns\_monthly.tail())  # Daily covariance of stocks (for each monthly period)  covariances = {}  rtd\_idx = returns\_daily.index  for i in returns\_monthly.index:  # Mask daily returns for each month and year, and calculate covariance  mask = (rtd\_idx.month == i.month) & (rtd\_idx.year == i.year)    # Use the mask to get daily returns for the current month and year of monthy returns index  covariances[i] = returns\_daily[mask].cov()  print(covariances[i])  portfolio\_returns, portfolio\_volatility, portfolio\_weights = {}, {}, {}  # Get portfolio performances at each month  for date in sorted(covariances.keys()):  cov = covariances[date]  for portfolio in range(10):  weights = np.random.random(3)  weights /= np.sum(weights) # /= divides weights by their sum to normalize  returns = np.dot(weights, returns\_monthly.loc[date])  volatility = np.sqrt(np.dot(weights.T, np.dot(cov, weights)))  portfolio\_returns.setdefault(date, []).append(returns)  portfolio\_volatility.setdefault(date, []).append(volatility)  portfolio\_weights.setdefault(date, []).append(weights)    print(portfolio\_weights[date][0])  # Get latest date of available data  date = sorted(covariances.keys())[-1]  # Plot efficient frontier  # warning: this can take at least 10s for the plot to execute...  plt.scatter(x=portfolio\_volatility[date], y=portfolio\_returns[date], alpha=0.1)  plt.xlabel('Volatility')  plt.ylabel('Returns')  plt.show() |
| Sharpe Ratio | #Sharpe Ratio:  # Empty dictionaries for Sharpe ratios and best Sharpe indexes by date  sharpe\_ratio, max\_sharpe\_idxs = {}, {}  # Loop through dates and get sharpe ratio for each portfolio  for date in portfolio\_returns.keys():  for i, ret in enumerate(portfolio\_returns[date]):    # Divide returns by the volatility for the date and index, i  sharpe\_ratio.setdefault(date, []).append(ret / portfolio\_volatility[date][i])  # Get the index of the best sharpe ratio for each date  max\_sharpe\_idxs[date] = np.argmax(sharpe\_ratio[date])  print(portfolio\_returns[date][max\_sharpe\_idxs[date]])    # Calculate exponentially-weighted moving average of daily returns  ewma\_daily = returns\_daily.ewm(span=30).mean()  # Resample daily returns to first business day of the month with the first day for that month  ewma\_monthly = ewma\_daily.resample('BMS').first()  # Shift ewma for the month by 1 month forward so we can use it as a feature for future predictions  ewma\_monthly = ewma\_monthly.shift(1).dropna()  print(ewma\_monthly.iloc[-1])  targets, features = [], []  # Create features from price history and targets as ideal portfolio  for date, ewma in ewma\_monthly.iterrows():  # Get the index of the best sharpe ratio  best\_idx = max\_sharpe\_idxs[date]  targets.append(portfolio\_weights[date][best\_idx])  features.append(ewma) # add ewma to features  targets = np.array(targets)  features = np.array(features)  print(targets[-5:])  # Get most recent (current) returns and volatility  date = sorted(covariances.keys())[-1]  cur\_returns = portfolio\_returns[date]  cur\_volatility = portfolio\_volatility[date]  # Plot efficient frontier with sharpe as point  plt.scatter(x=cur\_volatility, y=cur\_returns, alpha=0.1, color='blue')  best\_idx = max\_sharpe\_idxs[date]  # Place an orange "X" on the point with the best Sharpe ratio  plt.scatter(x=cur\_volatility[best\_idx], y=cur\_returns[best\_idx], marker='x', color='orange')  plt.xlabel('Volatility')  plt.ylabel('Returns')  plt.show()  # Make train and test features  train\_size = int(0.85 \* features.shape[0])  train\_features = features[:train\_size]  test\_features = features[train\_size:]  train\_targets = targets[:train\_size]  test\_targets = targets[train\_size:]  # Fit the model and check scores on train and test  rfr = RandomForestRegressor(n\_estimators=300, random\_state=42)  rfr.fit(train\_features, train\_targets)  print(rfr.score(train\_features, train\_targets))  print(rfr.score(test\_features, test\_targets))  # Get predictions from model on train and test  train\_predictions = rfr.predict(train\_features)  test\_predictions = rfr.predict(test\_features)  # Calculate and plot returns from our RF predictions and the SPY returns  test\_returns = np.sum(returns\_monthly.iloc[train\_size:] \* test\_predictions, axis=1)  plt.plot(test\_returns, label='algo')  plt.plot(returns\_monthly['SPY'].iloc[train\_size:], label='SPY')  plt.legend()  plt.show()  # Calculate the effect of our portfolio selection on a hypothetical $1k investment  cash = 1000  algo\_cash, spy\_cash = [cash], [cash] # set equal starting cash amounts  for r in test\_returns:  cash \*= 1 + r  algo\_cash.append(cash)  # Calculate performance for SPY  cash = 1000 # reset cash amount  for r in returns\_monthly['SPY'].iloc[train\_size:]:  cash \*= 1 + r  spy\_cash.append(cash)  print('algo returns:', (algo\_cash[-1] - algo\_cash[0]) / algo\_cash[0])  print('SPY returns:', (spy\_cash[-1] - spy\_cash[0]) / spy\_cash[0])  # Plot the algo\_cash and spy\_cash to compare overall returns  plt.plot(algo\_cash, label='algo')  plt.plot(spy\_cash, label='SPY')  plt.legend() # show the legend  plt.show() |
| Sharpe Ratio | #Efficient Frontier (Top left edge)  #Past performance is not a guarantee of future returns  #Markowitz Portfolio consists of the following 2:  #Tangency portfolio with highest Sharpe ratio (Max Sharpe Ratio) and crosses capital allocation line according to Capital Asset Pricing Model theory (MSR portfolio tends to be rather inconsistent as compared to GMV)  #Global Minimum Volatility (Lowest volatility)    # Risk free rate  risk\_free = 0  # Calculate the Sharpe Ratio for each asset  RandomPortfolios['Sharpe'] = (RandomPortfolios['Returns'] - risk\_free) / RandomPortfolios['Volatility']  # Print the range of Sharpe ratios  print(RandomPortfolios['Sharpe'].describe()[['min', 'max']])  # Sort the portfolios by Sharpe ratio  sorted\_portfolios = RandomPortfolios.sort\_values(by=['Sharpe'], ascending=False)  # Extract the corresponding weights  MSR\_weights = sorted\_portfolios.iloc[0, 0:numstocks]  # Cast the MSR weights as a numpy array  MSR\_weights\_array = np.array(MSR\_weights)  # Calculate the MSR portfolio returns  StockReturns['Portfolio\_MSR'] = StockReturns.iloc[:, 0:numstocks].mul(MSR\_weights\_array, axis=1).sum(axis=1)  # Plot the cumulative returns  cumulative\_returns\_plot(['Portfolio\_EW', 'Portfolio\_MCap', 'Portfolio\_MSR'])  # Sort the portfolios by volatility  sorted\_portfolios = RandomPortfolios.sort\_values(by=['Volatility'], ascending=True)  # Extract the corresponding weights  GMV\_weights = sorted\_portfolios.iloc[0, 0:numstocks]  # Cast the GMV weights as a numpy array  GMV\_weights\_array = np.array(GMV\_weights)  # Calculate the GMV portfolio returns  StockReturns['Portfolio\_GMV'] = StockReturns.iloc[:, 0:numstocks].mul(GMV\_weights\_array, axis=1).sum(axis=1)  # Plot the cumulative returns  cumulative\_returns\_plot(['Portfolio\_EW', 'Portfolio\_MCap', 'Portfolio\_MSR', 'Portfolio\_GMV']) |
| Capital Asset Pricing Model | #Excess returns: Return – Risk-free return  #Capital Asset Pricing Model (~70% variability explained)  Exposure β: Measure of exposure to broad stock market. For every 1% increase in market, portfolio will increase by β%  # Calculate excess portfolio returns  FamaFrenchData['Portfolio\_Excess'] = FamaFrenchData['Portfolio'] - FamaFrenchData['RF']  # Plot returns vs excess returns  CumulativeReturns = ((1+FamaFrenchData[['Portfolio','Portfolio\_Excess']]).cumprod()-1)  CumulativeReturns.plot()  plt.show()  # Calculate the co-variance matrix between Portfolio\_Excess and Market\_Excess  covariance\_matrix = FamaFrenchData[['Portfolio\_Excess', 'Market\_Excess']].cov()  # Extract the co-variance co-efficient  covariance\_coefficient = covariance\_matrix.iloc[0, 1]  print(covariance\_coefficient)  # Calculate the benchmark variance  benchmark\_variance = FamaFrenchData['Market\_Excess'].var()  print(benchmark\_variance)  # Calculating the portfolio market beta  portfolio\_beta = covariance\_coefficient / benchmark\_variance  print(portfolio\_beta)  # Import statsmodels.formula.api  import statsmodels.formula.api as smf  # Define the regression formula  CAPM\_model = smf.ols(formula = 'Portfolio\_Excess ~ Market\_Excess', data=FamaFrenchData)  # Print adjusted r-squared of the fitted regression  CAPM\_fit = CAPM\_model.fit()  print(CAPM\_fit.rsquared\_adj)  # Extract the beta  regression\_beta = CAPM\_fit.params['Market\_Excess']  print(regression\_beta) |
| Fama-French 3 Factor Model | #Fama-French 3 Factor Model (~90% variability explained): Extension of CAPM  #SMB: Small stocks outperforms big stocks (Small size returns premium)  #HML: Value versus growth is more cyclical (Value stocks outperform during crisis; Growth stocks outperform during bull runs)  #Negative SMB, Positive HML: When small stocks increase, portfolio decreases. When momentum stocks increase, portfolio increases.  #Positive error: Outperformance due to timing, skill, luck  #Weighted sum of all alpha in market must be 0. Therefore, some funds will be positive, some will be negative.  #Weighted sum of returns of all investors is market portfolio.  #Efficient Market Hypothesis: Market prices all information, hence alpha is missing factor in a complex economic pricing model  # Import statsmodels.formula.api  import statsmodels.formula.api as smf  # Define the regression formula  FamaFrench\_model = smf.ols(formula='Portfolio\_Excess ~ Market\_Excess + SMB + HML', data=FamaFrenchData)  # Fit the regression  FamaFrench\_fit = FamaFrench\_model.fit()  # Extract the adjusted r-squared  regression\_adj\_rsq = FamaFrench\_fit.rsquared\_adj  print(regression\_adj\_rsq)  # Extract the p-value of the SMB factor  smb\_pval = FamaFrench\_fit.pvalues['SMB']  # If the p-value is significant, print significant  if smb\_pval < 0.05:  significant\_msg = 'significant'  else:  significant\_msg = 'not significant'  # Print the SMB coefficient  smb\_coeff = FamaFrench\_fit.params['SMB']  print("The SMB coefficient is ", smb\_coeff, " and is ", significant\_msg)  # Calculate your portfolio alpha  portfolio\_alpha = FamaFrench\_fit.params['Intercept']  print(portfolio\_alpha)  # Annualize your portfolio alpha  portfolio\_alpha\_annualized = ((1 + portfolio\_alpha)\*\*252) - 1  print(portfolio\_alpha\_annualized) |
| 5 Factor Model | #5 Factor Model: Extension of 3 Factor Model to include RMW Profitability and CMA Investment  # Import statsmodels.formula.api  import statsmodels.formula.api as smf  # Define the regression formula  FamaFrench5\_model = smf.ols(formula='Portfolio\_Excess ~ Market\_Excess + SMB + HML + RMW + CMA', data=FamaFrenchData)  # Fit the regression  FamaFrench5\_fit = FamaFrench5\_model.fit()  # Extract the adjusted r-squared  regression\_adj\_rsq = FamaFrench5\_fit.rsquared\_adj  print(regression\_adj\_rsq) |
| Tail Risk | #Tail Risk: Risk of extreme investment outcomes, especially on negative side (left tail) of distribution  #Historical drawdown: Percentage loss from highest cumulative historical point (Lower is better: Growing consistently over time)  # Calculate the running maximum  running\_max = np.maximum.accumulate(cum\_rets)  # Ensure the value never drops below 1  running\_max[running\_max < 1] = 1  # Calculate the percentage drawdown  drawdown = (cum\_rets)/running\_max - 1  # Plot the results  drawdown.plot()  plt.show() |
| Historical value at risk | #Value at Risk: Estimate (single) day price movement; Threshold with given confidence level that losses will not historically exceed certain level (X% for VaR(95) at 95% 🡪 Loss will not exceed X% for 95% of scenarios)  #Need to try out different quantiles (90, 95, 99) to evaluate risks  # Calculate historical VaR(95)  var\_95 = np.percentile(StockReturns\_perc, 5)  print(var\_95)  # Sort the returns for plotting  sorted\_rets = StockReturns\_perc.sort\_values()  # Plot the probability of each sorted return quantile  plt.hist(sorted\_rets, normed=True)  # Denote the VaR 95 quantile  plt.axvline(x=var\_95, color='r', linestyle='-', label="VaR 95: {0:.2f}%".format(var\_95))  plt.show()  # Aggregate forecasted VaR  forecasted\_values = np.empty([100, 2])  # Loop through each forecast period  for i in range(0, 100):  # Save the time horizon i  forecasted\_values[i, 0] = i  # Save the forecasted VaR 95  forecasted\_values[i, 1] = var\_95 \* np.sqrt(i + 1)    # Plot the results  plot\_var\_scale() |
| Conditional value at risk (Historical Expected Shortfall) | #Conditional value at risk: Estimate of expected losses sustained in worst (1 – X)% of scenarios (Expected shortfall) (X% for CVaR(95) at 95% 🡪 Loss will be average of X% for worst 5% of scenarios)  #Subset mean of all returns <= Var(95) (CVaR is always worse than VaR of same quantile)  # Historical CVaR 95  cvar\_95 = StockReturns\_perc[StockReturns\_perc <= var\_95].mean()  print(cvar\_95)  # Sort the returns for plotting  sorted\_rets = sorted(StockReturns\_perc)  # Plot the probability of each return quantile  plt.hist(sorted\_rets, normed=True)  # Denote the VaR 95 and CVaR 95 quantiles  plt.axvline(x=var\_95, color="r", linestyle="-", label='VaR 95: {0:.2f}%'.format(var\_95))  plt.axvline(x=cvar\_95, color='b', linestyle='-', label='CVaR 95: {0:.2f}%'.format(cvar\_95))  plt.show() |
| Parametric VaR | # Import norm from scipy.stats  from scipy.stats import norm  # Estimate the average daily return  mu = np.mean(StockReturns)  # Estimate the daily volatility  vol = np.std(StockReturns)  # Set the VaR confidence level  confidence\_level = 0.05  # Calculate Parametric VaR  var\_95 = norm.ppf(confidence\_level, mu, vol)  print('Mean: ', str(mu), '\nVolatility: ', str(vol), '\nVaR(95): ', str(var\_95)) |
| Random walks / Monte Carlo Simulation | # Set the simulation parameters  mu = np.mean(StockReturns)  vol = np.std(StockReturns)  T = 252  S0 = 10  # Add one to the random returns  rand\_rets = np.random.normal(mu, vol, T) + 1  # Forecasted random walk  forecasted\_values = rand\_rets.cumprod() \* S0  # Plot the random walk  plt.plot(range(0, T), forecasted\_values)  plt.show()  # Loop through 100 simulations  for i in range(0, 100):  # Generate the random returns  rand\_rets = np.random.normal(mu, vol, T) + 1    # Create the Monte carlo path  forecasted\_values = S0\*(rand\_rets).cumprod()    # Plot the Monte Carlo path  plt.plot(range(T), forecasted\_values)  # Show the simulations  plt.show()  # Aggregate the returns  sim\_returns = []  # Loop through 100 simulations  for i in range(100):  # Generate the Random Walk  rand\_rets = np.random.normal(mu, vol, T)    # Save the results  sim\_returns.append(rand\_rets)  # Calculate the VaR(99)  var\_99 = np.percentile(sim\_returns, 1)  print("Parametric VaR(99): ", round(100\*var\_99, 2),"%") |