5.3 Lesson Plan - Track to the Future!

Overview

Today's class will focus on the notion of using Monte Carlo simulations to forecast future results and make confident predictions supported by statistical evidence. Monte Carlo simulations are an important tool in emulating a real-world use case that involves a degree of randomness surrounding an event or outcome, and seeks to iterate `n` number of times to find the most probable result of a variable event as well as the range of results and their corresponding probabilities of occurring.

In particular, stocks prices also tend to move somewhat randomly in such a way that there are varying probabilities to where the price may go or deviate from its average return (daily, weekly, monthly). Therefore, this lesson will teach students how to apply the concept of Monte Carlo simulations to predict future stock prices and therefore forecast the potential stocks returns of an initial investment, either as a single stock investment or as an investment in a portfolio.

Class Objectives

By the end of class, students will be able to:

- Define what a simulation is and why it's used.
- Deconstruct the components of the Monte Carlo Simulation process: probability distributions and iterations.
- Interpret probability distributions (normal/bell curve) and random number generators.
- Comprehend the use of confidence intervals and what they suggest.
- Implement a single Monte Carlo simulation on the future price trajectory of a stock.
- Execute multiple Monte Carlo simulations on the future price trajectories of a stock.
- Break down Portfolio Forecasting in the context of Monte Carlo Simulations on stock price trajectories and portfolio returns.
- Implement multiple Monte Carlo simulations on the potential returns of a stock portfolio.

Instructor Notes

- Today's lesson deals heavily with statistical concepts, particularly probability. Try to be as clear as possible and be mindful of students who may become easily confused as this lesson will surely push the boundaries of most students' comfort levels when it comes to statistics.
- When overviewing the concept of probability distributions, also make sure to stress the notion of randomness. Probability merely implies that there is a chance that a specific result or event may occur but makes no guarantees, which is why results can differ with each iteration.
- Once students are comfortable with probability distributions, namely normal distributions, students should be able to process the idea that Monte Carlo simulations on stock investments seek to chart the different paths (and probabilities) in which a stock can vary about its average daily return. Overview the code in detail so that this becomes more apparent.
- Towards the end of class, students will begin applying Monte Carlo simulations to portfolio returns. Therefore, they will need to combine the concepts of portfolio optimization (taught in the Pandas unit) with the concept of portfolio forecasting (taught in today's lesson). Walk through the steps in detail as students can easily get lost in this myriad of technical concepts!

Sample Class Video (Highly Recommended)

- To watch an example class lecture, go here: 5.3 Class Video. Note that this video may not reflect the most recent lesson plan.

4. Students Do: Free Throw Simulation

In this activity, students execute a Monte Carlo simulation to analyze the probability distribution of free throws made (out of 10 shots) for a player with a 70% accuracy and determine the likelihood of the player making 9-10 free throws in a single session.

Circulate with TAs during this activity to provide students with assistance. Below are a couple scenarios to watch out for.

- Students might face difficulty working with the histograms. Histogram **bins** have a default value, and so if the **bins** are not configured properly, the charts might not look as expected (the **bin** edges will be off) and the ranges may deviate from what is being simulated.
- Also keep an eye out for any student issues related to missing data; if 'missed free throws' or 'made free throws' data is missing, this is most likely because 0 was not appended for the missing values. This could visually result in one side of the distribution being cut off (producing a non-normal distribution with no values for the first shot attempted).

Instructions: README

Files: free_throw_simulation_unsolved.ipynb

5. Instructor Do: Review Free Throw Simulation (10 min)

Files: free_throw_simulation_solved.ipynb

Open the solution and explain the following:

- The process of executing a Monte Carlo simulation remains similar even for a different use case (free throws vs. coin flips). At its core, the basis of Monte Carlo simulations is iteration (running tests and simulations) and saving the results of a random process. Thus, expect the programmatic structure of `for` loops and potentially nested `for` loops.

```
# Set number of simulations and free throws
num_simulations = 1000
num_throws = 10

# Set a list object acting as a throw: made basket or missed basket
throw = ["made", "missed"]

# Set probability of events
probability = [0.7, 0.3]

# Create an empty DataFrame to hold simulation results
monte_carlo = pd.DataFrame()

# Run n number of simulations
for n in range(num_simulations):

    # Print simulation iteration
    # print(f"Running Simulation {n+1}...")
    # Set an empty list to hold throw results
    throws = []
```

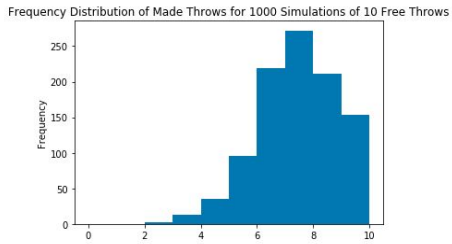
```
    # Shoot the ball `10` times
    for i in range(num_throws):

        # Randomly choose between `made` and `missed` with a `70%`
chance to make the throw and a `30%` chance the throw is missed
        free_throw = random.choice(throw, p=probability)

        # Print throw result
        # print(f" Throw {i+1}: {free_throw}")

        # Append throw result to list
        throws.append(free_throw)

    # Append column for each simulation and throw results
    monte_carlo[f"Simulation {n}"] = pd.Series(throws)

# Print the DataFrame
monte_carlo
```

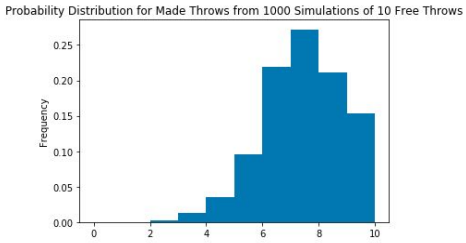
- The `choice` function from the `random` class of the `numpy` library has a `p` parameter that allows for setting a non-uniform probability to events. In this case, a player has a `70%` chance of making a shot and consequently a `30%` chance of missing the shot. Therefore, the `choice` function below reflects this.

```
# Randomly choose between `made` and `missed` with a `70%`
chance to make the throw and a `30%` chance the throw is missed
free_throw = random.choice(throw, p=probability)
```

- Because the random process has non-uniform probability (`70%` chance to make a shot and

30% chance to miss a shot) the corresponding frequency and probability distributions of made free throws show that a majority of the distribution lies within the 7, 8, 9, 10 range, while the rest of the distribution is spread out within the 0, 1, 2, 3, 4, 5, 6 range. Unlike the bell-curve of a normal distribution, this is called a skewed (in this case left-skewed) distribution.



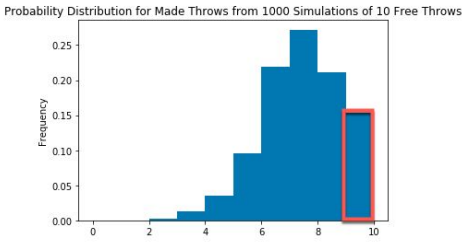


- The probability distribution of free throws made will change slightly with every run of the program; however, in this current run, the probability distribution shows that the likelihood of the player making 9-10 shots in a single session is approximately 15%.

Plot Probability Distribution of Made Free Throws for 1000 Simulations of 10 Free Throws

```
# Plot the data as a histogram with probabilities of
plot_title = f"Probability Distribution for Made Throws from {num_simulations} Simulations of 10 Free Throws"
freq_dist_df['made_throws'].plot.hist(density=True, title=plot_title, bins=[0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10])
```

```
<matplotlib.axes._subplots.AxesSubplot at 0x114197320>
```



End Section

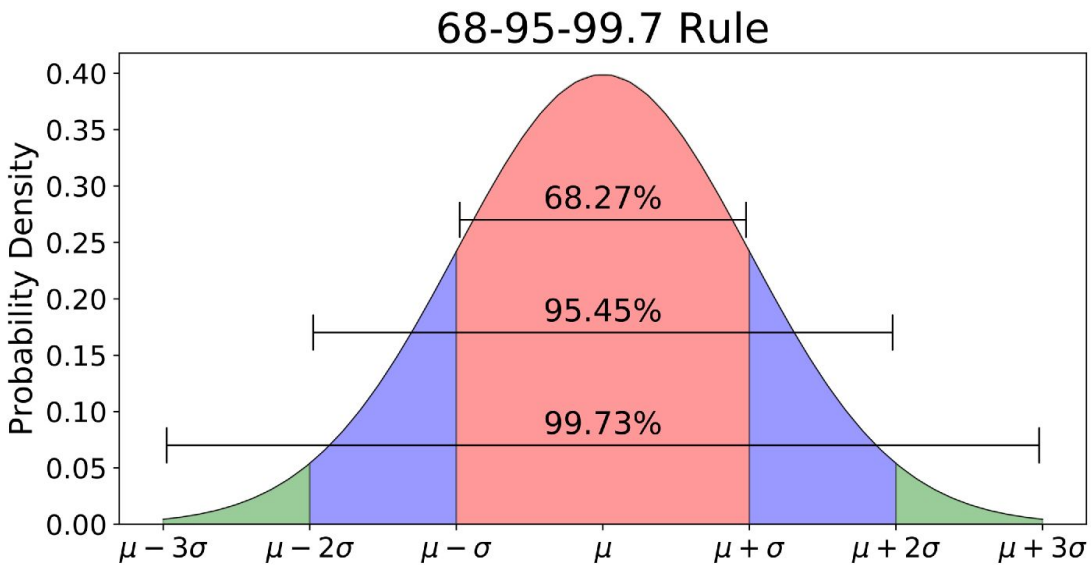
9. Instructor Do: Simulation of Stock Price Trajectory (10 min)

This activity exemplifies the use case where a Monte Carlo simulation can be applied to a historical data set such as daily closing stock prices, given the assumption that daily closing stock prices have a normal probability distribution. Stock data sets will be pulled in from the Alpaca API and used to generate a Monte Carlo simulation based off of a normally distributed random process using the data set's calculated average and standard deviation of daily returns.

Files: stock_price_simulation_solved.ipynb

Walk through the solution and highlight the following:

- Monte Carlo simulations can be executed not just on random processes with *discrete probabilities* (ex. `70%` to make a free throw and `30%` to miss a free throw), but also on *continuous probabilities* such as normal probability distributions.
- Normal probability distributions showcase the various probabilities of returning a value based on the number of standard deviations it is from the mean (how far the value may lie plus or minus from the average expected value); values far away from the mean are less common while values near the mean are more common. Monte Carlo simulation use this characteristic to simulate a random process' potential outcomes with respect to the variability around its mean.



- The daily closing stock price data will be pulled using the `Alpaca-trade-api` SDK that connects to the `Alpaca` API. Therefore, make sure to import the necessary libraries and dependencies before proceeding.

```
# Import libraries and dependencies
import numpy as np
import pandas as pd
import os
from datetime import datetime, timedelta
import matplotlib.pyplot as plt
import alpaca_trade_api as tradeapi

%matplotlib inline
```

- Now that you have imported the `alpaca-trade-api` and it's required dependencies we are going to list out the available, tradable assets.

- Use the `list_assets()` function from the `tradeapi` object to check the available stock ticker data that can be pulled from the `Alpaca` API. Then iterate over the data to only keep the currently tradable assets.

```
# Get all Asstes
assets = api.list_assets()

# Keep only tradeable assets
tradeable = [asset for asset in assets if asset.tradable ]
tradeable
```

```
[Asset({   'class': 'us_equity',
    'easy_to_borrow': True,
    'exchange': 'NYSE',
    'id': '27155f8a-2e13-4098-a54b-d63ac819aa9c',
    'marginable': True,
    'name': 'Main Street Capital Corporation',
    'shortable': True,
    'status': 'active',
    'symbol': 'MAIN',
    'tradable': True}), Asset({   'class': 'us_equity',
    'easy_to_borrow': True,
    'exchange': 'NYSE',
```

- Create a new empty DataFrame named `asset_info_df`. Convert the python list of assets to a panda's series and then define a new column in your DataFrame named `symbol` with

that series.

```
# Create a new empty DataFrame
asset_info_df = pd.DataFrame()
asset_info_df["symbol"] = pd.Series([asset.symbol for asset in assets])

# Display the first 10 asset tickers
display(asset_info_df.head(10))
```

	symbol
0	MAIN
1	OIL
2	CCA
3	CY
4	TRNX
5	NFO
6	CBPX
7	GRSH_DELISTED
8	N014776_DELISTED
9	KOR

- The `get_barset()` function from the Alpaca SDK takes in the following parameters:
    - ticker
    - timeframe
    - limit
    - start_date
    - end_date
    - after
    - until

- And returns an object containing a DataFrame of `AAPL` daily stock prices. The `start_date` and `end_date` variables, in this case, are set to `365` days from the current date and the current date, respectively. To correctly fetch the stock data, the Alpaca SDK works with dates in ISO format, so we transform the `start_date` and the `end_date` using the `Timestamp` and `isoformat` functions from Pandas.

```
# Set the ticker
ticker = "AAPL"

# Set timeframe to "1D"
timeframe = "1D"

# Set start and end datetimes of 1 year, between now and 365 days ago.
end_date = datetime.now()
start_date = end_date + timedelta(-365)

# Format end_date and start_date with isoformat
end_date = pd.Timestamp(end_date, tz="America/New_York").isoformat()
start_date = pd.Timestamp(start_date, tz="America/New_York").isoformat()

# Get 1 year"s worth of historical data for AAPL
df = api.get_barset(
    ticker,
    timeframe,
    limit=None,
    start=start_date,
    end=end_date,
    after=None,
    until=None,
).df

df.head()
```

	AAPL				
	open	high	low	close	volume
2019-05-06 00:00:00-04:00	204.290	208.8400	203.500	208.60	28949691
2019-05-07 00:00:00-04:00	205.880	207.4175	200.825	202.86	34328425
2019-05-08 00:00:00-04:00	201.900	205.3400	201.750	202.90	22729670
2019-05-09 00:00:00-04:00	200.400	201.6800	196.660	200.72	32427147
2019-05-10 00:00:00-04:00	197.419	198.8500	192.770	197.30	36118438

- The DataFrame object from the Alpaca SDK contains an outer level (`level 0`) that is not needed, drop this level using the `df.droplevel` function.
- Simulating stock price trajectory involves analyzing the closing prices of a stock. Therefore, it's best to drop the extraneous columns for the `AAPL` price data received from the `Alpaca` API.

```
# Drop Outer Table Level
df = df.droplevel(axis=1, level=0)

# Use the drop function to drop extra columns
df.drop(columns=["open", "high", "low", "volume"], inplace=True)

# Since this is daily data, we can keep only the date (remove the time) component of the data
df.index = df.index.date

df.head()
```

	close
2019-05-06	208.60
2019-05-07	202.86
2019-05-08	202.90
2019-05-09	200.72
2019-05-10	197.30

- To simulate `AAPL` stock prices for the next `252` trading days, the simulation must be framed in the context of a stock's growth. Therefore, the `pct_change` function is used to calculate the last year of daily returns for `AAPL`, and the `mean` and `std` functions are used to calculate the average daily return and the volatility of daily returns.

Calculate Daily Returns

```
# Use the `pct_change` function to calculate daily returns of AAPL
daily_returns = df.pct_change()
daily_returns.head()
```

	close
2019-05-06	NaN
2019-05-07	-0.027517
2019-05-08	0.000197
2019-05-09	-0.010744
2019-05-10	-0.017039

Calculate Value of Average Daily Returns

```
# Use the `mean` function to calculate the mean of daily returns for AAPL
avg_daily_return = daily_returns.mean()["close"]
avg_daily_return
```

```
0.0016430845809107236
```

Calculate Value of Standard Deviation of Daily Returns

```
# Use the `std` function to calculate the standard deviation of daily returns for AAPL
std_dev_daily_return = daily_returns.std()["close"]
std_dev_daily_return
```

```
0.025972760508558162
```

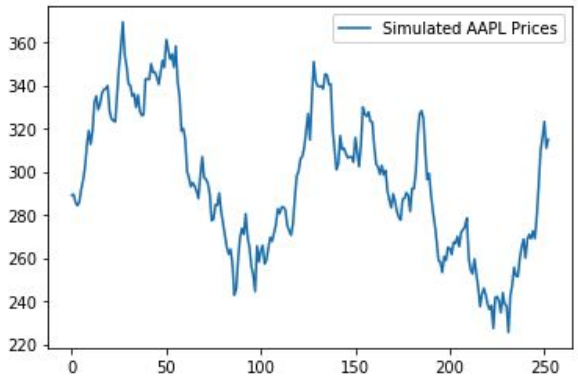
- The following code snippet exemplifies the simulation of stock price trajectory. The simulation calculates the next day's simulated closing price by multiplying the preceding day's closing price by a random selection of a range of values defined by the normal probability distribution of `AAPL` daily returns, given by the *mean* and *standard deviation* of daily returns.

```
# Simulate the returns for 252 days
for i in range(num_trading_days):
    # Calculate the simulated price using the last price within the list
    simulated_price = simulated_aapl_prices[-1] * (1 +
np.random.normal(avg_daily_return, std_dev_daily_return))
    # Append the simulated price to the list
    simulated_aapl_prices.append(simulated_price)
```

- Plotting the DataFrame of simulated `AAPL` closing prices for the next `252` trading days shows one potential pathway for how `AAPL` stock prices may behave in the next year.

```
# Use the `plot` function to plot the trajectory of AAPL stock based on a 252 trading day simulation
simulated_price_df.plot()
```

```
<matplotlib.axes._subplots.AxesSubplot at 0x118ac4cd0>
```



- Calculating the daily returns and cumulative returns of `AAPL` simulated prices allow for plotting the profits and losses of a potential investment in `AAPL` over the next trading year.

Calculate the Cumulative Profits/Losses of Simulated Stock Prices for AAPL

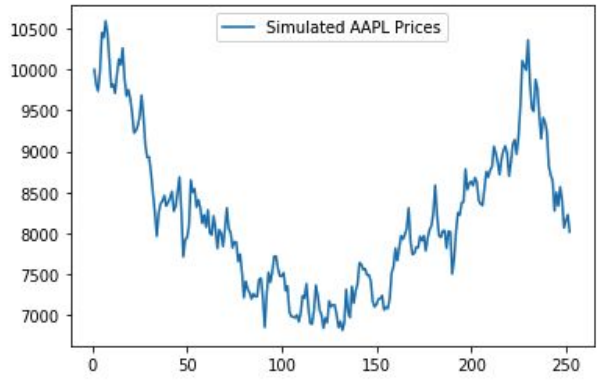
```
# Multiply an initial investment by the cumulative returns of simulative stock prices to
# return the progression of cumulative returns in terms of money
initial_investment = 10000
cumulative_pnl = initial_investment * simulated_cumulative_returns
cumulative_pnl.head()
```

	Simulated AAPL Prices
0	NaN
1	9998.024421
2	9817.829573
3	9736.095739
4	9972.161820

Plot the Cumulative Profits/Losses of `$10,000` in AAPL Over the Next `252` Trading Days

```
# Use the "plot" function to create a chart of the cumulative profits/losses
cumulative_pnl.plot()
```

```
<matplotlib.axes._subplots.AxesSubplot at 0x1267ecc50>
```



- Answer any questions before moving on.

End Section