Code Explanation Report

1. Importing Libraries and Loading Data

```
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
import pandas_ta as ta
btc_data = pd.read_csv("btc_daily_data.csv")
```

This section imports necessary libraries:

- `pandas` for data manipulation
- `numpy` for numerical operations
- `matplotlib.pyplot` for plotting
- `pandas_ta` for technical analysis indicators

It then loads Bitcoin daily data from a CSV file into a pandas DataFrame called `btc_data`.

2. Heikin-Ashi Calculations

```
def calculate_heikin_ashi(df):
    df['HA_Close'] = (df['Open'] + df['High'] + df['Low'] + df['Close']) / 4
    df['HA_Open'] = (df['Open'] + df['Close']) / 2
    df['HA_High'] = df[['High', 'HA_Open', 'HA_Close']].max(axis=1)
    df['HA_Low'] = df[['Low', 'HA_Open', 'HA_Close']].min(axis=1)
    return df
```

This function calculates Heikin-Ashi candles, which are a type of financial chart used to filter out market noise. It creates new columns in the DataFrame:

- `HA_Close`: Average of Open, High, Low, and Close
- `HA_Open`: Average of Open and Close
- `HA_High`: Maximum of High, HA_Open, and HA_Close
- `HA_Low`: Minimum of Low, HA_Open, and HA_Close

3. Smooth Heikin-Ashi Calculations

```
def smooth_heikin_ashi(df, period=9):
    df = calculate_heikin_ashi(df)
    df['SHA_Close'] = df['HA_Close'].rolling(window=period).mean()
    df['SHA_Open'] = df['HA_Open'].rolling(window=period).mean()
    df['SHA_High'] = df[['High', 'SHA_Open', 'SHA_Close']].max(axis=1)
    df['SHA_Low'] = df[['Low', 'SHA_Open', 'SHA_Close']].min(axis=1)
    return df
```

This function creates smoothed Heikin-Ashi candles by applying a rolling mean to the Heikin-Ashi values. It:

1. Calls `calculate_heikin_ashi()`
2. Calculates rolling means for Close and Open values
3. Determines new High and Low values based on the smoothed data

4. Trade Execution Function

```
def execute_trade(price_data):
    # Initialize the DataFrame
    trade_book = pd.DataFrame(columns=["Long ID", "Short ID", "Entry time", "Exit time", "Entry Price", "Exit Price", "Signal", "Long
    cash = 100000
    pnl = 0
    long_t = False
    short_t = False
    short_price = 0
    long_price = 0
    short_quantity = 0
    long_quantity = 0
    long_ID = 0
    short_ID = 0
    # Loop through each row in the price history dataframe
    for i in range(len(price_data)):
        # Long Trade Management
        if long_t:
            price = price_data["Open"].iloc[i]
            if price_data["Signals"].iloc[i] == 2:
                long_t = False
                # Add exit signals to trade book
                price_data.loc[price_data.index[i], "Close Signals"] = -1
                price_data.loc[price_data.index[i], "Remarks"] = "Exit Long Algo"
                trade_book.loc[trade_book["Long ID"] == long_ID, "Exit time"] = price_data.index[i]
                trade_book.loc[trade_book["Long ID"] == long_ID, "Exit Price"] = price
                # Further trading logic here...
```

This function simulates trade execution based on signals in the price data. It:

1. Initializes a trade book to record all trades
2. Sets up initial values for cash, profit/loss, and trade states
3. Loops through the price data
4. Manages long trades, including:
    - Checking for exit signals
    - Updating the trade book when exiting a trade
5. (The function is incomplete in the provided code, but would likely include similar logic for short trades and other trade management tasks)

5. Genetic Algorithm Setup

```
import random
from deap import base, creator, tools, algorithms

# Set up the DEAP environment
creator.create("FitnessMax", base.Fitness, weights=(1.0,))
creator.create("Individual", list, fitness=creator.FitnessMax)
toolbox = base.Toolbox()

# Parameter ranges
param_ranges = {
    "sha_len": (5, 20),
    "adx_len": (5, 20),
    "rsi_len": (5, 20),
    "atr_len": (5, 20),
    "rsi_long": (30, 70),
    "rsi_short": (30, 70),
    "adx_long": (20, 50),
    "adx_short": (20, 50)
}

# Register attributes
for i, (param, (low, high)) in enumerate(param_ranges.items()):
    toolbox.register(f"attr_{i}", random.randint, low, high)

# Create individual and population
toolbox.register("individual", tools.initCycle, creator.Individual,
                 (toolbox.attr_0, toolbox.attr_1, toolbox.attr_2, toolbox.attr_3,
                  toolbox.attr_4, toolbox.attr_5, toolbox.attr_6, toolbox.attr_7), n=1)
toolbox.register("population", tools.initRepeat, list, toolbox.individual)
```

This section sets up a genetic algorithm using the DEAP library to optimize trading parameters:

1. Defines fitness and individual types
2. Sets parameter ranges for various indicators (e.g., RSI length, ADX length)
3. Registers attribute generators for each parameter
4. Creates functions to generate individuals and populations

6. Evaluation and Genetic Operators

```
# Define evaluation function
def evaluate(individual):
    sha_len, adx_len, rsi_len, atr_len, rsi_long, rsi_short, adx_long, adx_short = individual
    *, total_ret = backtest(btc_data, sha_len, adx_len, rsi_len, atr_len, rsi_long, rsi_short, adx_long, adx_short)
    return (total_ret,)

toolbox.register("evaluate", evaluate)
toolbox.register("mate", tools.cxTwoPoint)
toolbox.register("mutate", tools.mutUniformInt, low=[r[0] for r in param_ranges.values()],
                 up=[r[1] for r in param_ranges.values()], indpb=0.2)
toolbox.register("select", tools.selTournament, tournsize=3)
```

This section defines the genetic algorithm's core components:

1. An evaluation function that runs a backtest with given parameters and returns the total return
2. Crossover (mating) operation using two-point crossover
3. Mutation operation using uniform integer mutation
4. Selection method using tournament selection

7. Optimization Function

```
def optimize(population_size=50, generations=30):
    pop = toolbox.population(n=population_size)
    hof = tools.HallOfFame(1)
    stats = tools.Statistics(lambda ind: ind.fitness.values)
    stats.register("avg", np.mean)
    stats.register("min", np.min)
    stats.register("max", np.max)

    pop, logbook = algorithms.eaSimple(pop, toolbox, cxpb=0.7, mutpb=0.2,
                                       ngen=generations, stats=stats, halloffame=hof, verbose=True)

    return hof[0], hof[0].fitness.values[0]
```

This function runs the genetic algorithm optimization:

1. Creates an initial population
2. Sets up a Hall of Fame to track the best individual
3. Configures statistics to track during evolution
4. Runs the simple evolutionary algorithm for a specified number of generations
5. Returns the best individual and its fitness value

8. Running the Optimization and Displaying Results

```
# Run optimization
best_params, best_return = optimize()

# Print results
param_names = list(param_ranges.keys())
print("Best parameters:")
for name, value in zip(param_names, best_params):
    print(f"{name}: {value}")
print(f"Best return: {best_return}")

# Verify the results
final_trade_book, final_total_ret = backtest(btc_data, *best_params)
```

This final section:

1. Runs the optimization process
2. Prints the best parameters found and the corresponding return
3. Verifies the results by running a backtest with the optimized parameters

This code implements a sophisticated trading strategy optimization system using genetic algorithms. It aims to find the best combination of technical indicator parameters to maximize returns on Bitcoin trading.