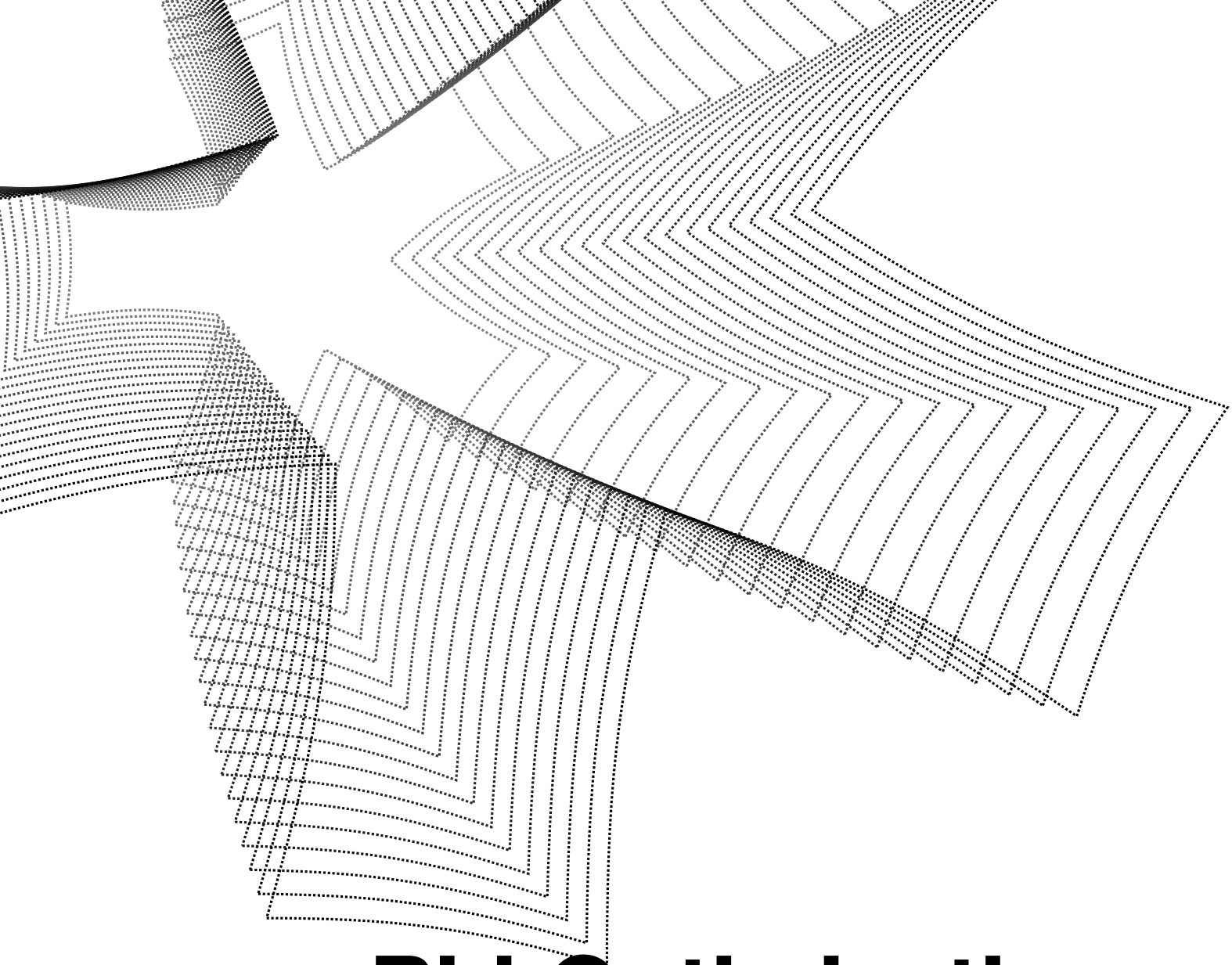


Bid Optimization

AMS Bid Optimization Using ML

Prepared by: **Nuvoretail Enlytical**
Date: **December 26, 2023**
Visit: **Bid Optimization**

THE TOOLS

1. TOOLS OVERVIEW

1.1. OVERVIEW

This document comprehensively details the algorithms and codebase employed in the bid optimization process for Amazon Marketing Services (AMS). This part of the document covers some tools used for AMS and some scripts that are not directly related to AMS. The next part will discuss the codebase for AMS in detail.

1.2. MATERIALS COVERED IN THIS PART

The document encompasses diverse types of materials, including:

1. **Tools:** This section outlines the tools utilized to facilitate the execution of code iterations and to manage other related tasks. For example:
   - Airflow
   - Flask
2. **Non-AMS Related Scripts:** Some scripts related that are not directly related to AMS bid optimization. For example:
   - OSA Availability Mailer
   - Category Benchmark
   - Event Insight

2. AIRFLOW

2.1. INTRODUCTION

Apache Airflow (or simply Airflow) is a platform to programmatically author, schedule, and monitor workflows. In our case, we use Airflow to schedule the various tasks for bid optimization and budget optimization. A detail on how to install Airflow on your system will be discussed in this section.

2.2. BENEFITS OF USING APACHE AIRFLOW

2.2.1. Workflow Orchestration

Apache Airflow provides a powerful platform for orchestrating complex workflows. It allows users to define, schedule, and monitor workflows as directed acyclic graphs (DAGs), enabling the automation of data pipelines and processes.

2.2.2. Extensibility and Customization

Airflow is highly extensible and allows users to define custom operators, sensors, and hooks. This extensibility makes it suitable for a wide range of use cases and integration with various systems and tools.

2.2.3. Dynamic and Scalable

One of the key benefits of Airflow is its ability to handle dynamic workflows and scale horizontally. It can efficiently manage a large number of tasks and parallelize execution, ensuring optimal resource utilization.

2.2.4. Built-in Monitoring and Logging

Airflow provides built-in tools for monitoring and logging, making it easier to track the progress and performance of workflows. This includes a web-based user interface for visualizing DAG runs, task logs, and execution metadata.

2.2.5. Dependency Management

Airflow allows users to express dependencies between tasks in a DAG, ensuring that tasks are executed in the correct order. This makes it easy to represent complex workflows with intricate dependencies and manage data flow between tasks.

2.2.6. Distributed Execution

With its distributed architecture, Apache Airflow can execute tasks on multiple worker nodes, allowing for parallel and distributed processing. This feature enhances performance and enables the handling of large-scale data processing tasks.

2.2.7. Community and Ecosystem

Airflow has a vibrant and active community that contributes to its development and maintenance. The ecosystem around Airflow includes a variety of plugins and integrations, providing users with a wide range of pre-built components for their workflows.

2.3. INSTALLING APACHE AIRFLOW

Airflow can not be installed on Windows directly. You have to use WSL to be able to install Airflow on Windows. However, it is suggested that you use Linux machine for this. This section provides a step-by-step approach on how to install Airflow on Windows using WSL. If you are using Linux, skip the first three steps and go directly to Update Package Lists.

2.3.1. Install Windows Subsystem for Linux (WSL)

Ensure that you have WSL enabled on your Windows machine. You can enable WSL by following the official Microsoft documentation: https://docs.microsoft.com/en-us/windows/wsl/install

2.3.2. Install a Linux Distribution on WSL

Choose and install a Linux distribution of your choice from the Microsoft Store. Popular choices include Ubuntu, Debian, or CentOS. Follow the installation instructions provided for the selected distribution.

2.3.3. Open WSL Terminal

Open the WSL terminal by launching the installed Linux distribution from the Start menu or using the command line.

2.3.4. Update Package Lists

Update the package lists for your Linux distribution using the following commands:

```
1 sudo apt-get update
2 sudo apt-get upgrade
```

2.3.5. Install Dependencies

Install the required dependencies for Apache Airflow by running the following command:

```
sudo apt-get install -y python3-pip python3-dev libssl-dev libffi-dev
    libmysqlclient-dev libjpeg8-dev zlib1g-dev
```

It is also advisable to create a virtual environment, If you already have Python and Miniconda/Anaconda, you can use that, and you do not need to install Python separately.

2.3.6. Install Apache Airflow

Install Apache Airflow using the Python package manager, pip. Run the following commands:

```
pip3 install apache-airflow
```

2.3.7. Initialize Airflow Database

Initialize the Airflow metadata database by running the following commands:

```
airflow db init
```

2.3.8. Start Airflow Web Server

Start the Airflow web server by running the following command:

```
airflow webserver
```

Visit http://localhost:8080 in your web browser to access the Airflow web interface.

2.3.9. Start Airflow Scheduler

In a new terminal window, start the Airflow scheduler by running the following command:

```
airflow scheduler
```

2.3.10. Verify Installation

Verify the successful installation of Apache Airflow by checking the web interface and exploring the example DAGs provided.

2.4. USING AIRFLOW

2.4.1. Some Useful Commands for Airflow

Once the webserver and scheduler are running, you can use the web interface of Airflow to keep a tab on which tasks failed and which were successful. Here are some commands that will be useful:

1. **Connecting to Airflow Via SSH:** You can use SSH via puTTy, git bash or powershell to connect through the VM on which airflow is running and access the web interface on your machine by port-forwarding.

```
ssh -L 8080:127.0.0.1:8080 <HOST>
```

   Running the command will ask for the password.
2. **Running the webserver and/or scheduler in background:**

```
airflow scheduler -D --pid scheduler.pid

airflow webserver -p 8080 -D --pid webserver.pid
```

3. We can use the `kill` command on the pid to stop the service `kill -9 `cat webserver.pid``

2.4.2. DAGs

In Apache Airflow, a Directed Acyclic Graph (DAG) is a fundamental concept used to define, schedule, and manage workflows. A DAG represents a collection of tasks with defined dependencies and a specific execution order. DAGs serve as a powerful abstraction for defining, managing, and executing workflows in Apache Airflow, making it a versatile platform for orchestrating data pipelines, ETL processes, and various automation tasks.

2.4.3. A Sample DAG

The following is a sample DAG using the TaskFlow API.

```
import json
import pendulum

from airflow.decorators import dag, task
@dag(
    schedule=None,
    start_date=pendulum.datetime(2021, 1, 1, tz="UTC"),
    catchup=False,
    tags=["example"],
)
def tutorial_taskflow_api():
    """
    ### TaskFlow API Tutorial Documentation
    This is a simple data pipeline example which demonstrates the use of
    the TaskFlow API using three simple tasks for Extract, Transform, and Load.
    Documentation that goes along with the Airflow TaskFlow API tutorial is
    located
    [here](https://airflow.apache.org/docs/apache-airflow/stable/
        tutorial_taskflow_api.html)
    """
    @task()
    def extract():
```

```
        """
        #### Extract task
        A simple Extract task to get data ready for the rest of the data
        pipeline. In this case, getting data is simulated by reading from a
        hardcoded JSON string.
        """
        data_string = '{"1001": 301.27, "1002": 433.21, "1003": 502.22}'

        order_data_dict = json.loads(data_string)
        return order_data_dict
    @task(multiple_outputs=True)
    def transform(order_data_dict: dict):
        """
        #### Transform task
        A simple Transform task which takes in the collection of order data and
        computes the total order value.
        """
        total_order_value = 0

        for value in order_data_dict.values():
            total_order_value += value

        return {"total_order_value": total_order_value}
    @task()
    def load(total_order_value: float):
        """
        #### Load task
        A simple Load task which takes in the result of the Transform task and
        instead of saving it to end user review, just prints it out.
        """

        print(f"Total order value is: {total_order_value:.2f}")
    order_data = extract()
    order_summary = transform(order_data)
    load(order_summary["total_order_value"])
tutorial_taskflow_api()
```

Here is another example using the standard API:

```
from datetime import datetime, timedelta
from airflow import DAG
from airflow.operators.python_operator import PythonOperator

# Step 1: Define default_args for the DAG
default_args = {
    'owner': 'airflow',
    'start_date': datetime(2023, 1, 1),  # Specify the start date of the DAG
    'retries': 1,
    'retry_delay': timedelta(minutes=5),
}

# Step 2: Create a DAG instance
dag = DAG(
    'example_taskflow_dag',
    default_args=default_args,
    schedule_interval='20 10 * * *',  # CRON expression for daily execution at
        10:20 AM
    catchup=False,  # Prevent catching up on historical DAG runs
)

# Step 3: Define three Python functions to be executed by the tasks

def task_a_func():
    # Task A logic goes here
    print("Executing Task A")
```

```
26
27 def task_b_func():
28     # Task B logic goes here
29     print("Executing Task B")
30
31 def task_c_func():
32     # Task C logic goes here
33     print("Executing Task C")
34
35 # Step 4: Create three PythonOperator tasks, each executing one of the
       functions
36
37 task_a = PythonOperator(
38     task_id='task_a',
39     python_callable=task_a_func,
40     dag=dag,
41 )
42
43 task_b = PythonOperator(
44     task_id='task_b',
45     python_callable=task_b_func,
46     dag=dag,
47 )
48
49 task_c = PythonOperator(
50     task_id='task_c',
51     python_callable=task_c_func,
52     dag=dag,
53 )
54
55 # Step 5: Set up task dependencies
56
57 task_a >> [task_b, task_c]  # Task A is followed by tasks B and C
58
59 # Explanation of the DAG structure:
60
61 # - The DAG is named 'example_taskflow_dag'.
62 # - It starts on January 1, 2023.
63 # - It has three tasks: A, B, and C.
64 # - Tasks B and C are dependent on Task A.
65 # - The DAG is scheduled to run every day at 10:20 AM.
66 # - The catchup parameter is set to False to prevent catching up on missed DAG
       runs.
```

2.4.4. CRON Expression

A CRON expression is a string representation of a schedule, commonly used in Unix-like operating systems to define the timing of recurring tasks. In the context of Apache Airflow, CRON expressions are employed to schedule the execution of Directed Acyclic Graphs (DAGs). The syntax of a CRON expression consists of five fields (minute, hour, day of the month, month, and day of the week) and supports flexible patterns for specifying time intervals. For example, the expression `0 2 * * *` represents a schedule to run a task every day at 2:00 AM. The use of CRON expressions in Airflow allows users to precisely define when their workflows should be triggered, providing a convenient and powerful way to automate recurring tasks with fine-grained control over the scheduling.

In Apache Airflow, scheduling DAGs using CRON expressions is straightforward and flexible. The `schedule_interval` parameter in the DAG definition allows users to specify the timing pattern for DAG runs. By setting `schedule_interval` to a CRON expression, users can dictate when the DAG should be triggered. For instance, to schedule a DAG

to run every weekday at 8:30 AM, the expression would be `30 8 * * 1-5`. This level of granularity enables users to tailor the execution frequency to meet the requirements of their workflows. Additionally, Airflow's support for catch-up and backfilling allows the system to compensate for any missed or delayed DAG runs, ensuring that the scheduled tasks are executed reliably. The use of CRON expressions in Airflow provides users with a powerful mechanism to automate and orchestrate complex workflows with precision and flexibility.

For more detail on CRON, see the **website**. You can use the **crontab guru** to check your CRON expression.

3. FLASK

3.1. INTRODUCTION

Flask is renowned for its simplicity and ease of use, making it an excellent choice for projects of various scales. To create a Flask app, developers typically define routes, each corresponding to a specific URL endpoint, and associated functions to handle requests to those endpoints. Developers can easily incorporate features like URL parameters, request handling, and form processing. With Flask, it's straightforward to implement RESTful APIs, serving as a foundation for building modern, data-driven applications. Additionally, Flask's extensive ecosystem of extensions provides solutions for tasks such as authentication, database integration, and more, allowing developers to extend their applications' capabilities effortlessly.

3.1.1. The Flask Microservice

A simple flask microservice was created to facilitate some simple interaction of the system generated output and the team handling the client. The microservice is used with the main **Enlytical website**. The app, for now, provides a number of features such as:

1. **Bid Modification:** The system generated bids are pushed to MongoDB and they are reflected to the enlytical web app. From their, team can change the bids and then the updated bids are pushed to Amazon API using a route of the flask app.
2. **State Modification:** Same as the bid, the state of a target can be updated by the team. They are pushed to Amazon API via another route of the flask app.
3. **Onboarding:** This is still going on. Once done, the flask app will be used to automate a lot of steps involved while onboarding a client such as:
   - Product Master
   - Seller Master
   - Competition Master
   - The Month's Projection
   - Top Sale and Category Keywords

3.1.2. Understanding `flask_jwt_extended` in Flask

The `flask_jwt_extended` library is a Flask extension designed to enhance Flask applications by providing support for JSON Web Tokens (JWT). JWT is a compact, URL-safe means of representing claims that can be transferred between two parties securely. Here are some of the features of JWT:

1. **Token-Based Authentication:** Enables token-based authentication in Flask applications. Users can obtain a JWT by providing valid credentials, such as a username and password.
2. **Token Creation and Verification:** Simplifies the process of creating and verifying JWTs, typically using a secret key for signing and verifying the tokens.
3. **Token Refresh:** Supports token refresh mechanisms, allowing users with a valid refresh token to obtain a new access token without re-entering credentials.

4. **Custom Claims:** Allows inclusion of custom claims in the JWT payload for storing additional user information or implementing custom authorization logic.
5. **Token Revocation:** Provides mechanisms for token revocation, enabling blacklisting or revoking tokens as needed.

3.1.3. Example Usage

Here's a basic example of how to use `flask_jwt_extended` in a Flask application:

```python
from flask import Flask, jsonify
from flask_jwt_extended import JWTManager, jwt_required, create_access_token

app = Flask(__name__)

# Configure Flask-JWT-Extended
app.config['JWT_SECRET_KEY'] = 'your-secret-key'
jwt = JWTManager(app)

# Example route that requires authentication
@app.route('/protected', methods=['GET'])
@jwt_required()
def protected():
    current_user = get_jwt_identity()
    return jsonify(logged_in_as=current_user), 200

# Example route for user login and token creation
@app.route('/login', methods=['POST'])
def login():
    access_token = create_access_token(identity='username')
    return jsonify(access_token=access_token), 200

if __name__ == '__main__':
    app.run(debug=True)
```

All the routes that use the decorator `@jwt_required()` is protected in a sense that you must pass `"Authorization": f"Bearer <ACCESS TOKEN>"` in the header.

3.2. CURRENT FLASK ROUTES

As of now, here is a list of routes and what are they used for as well as what input is the route expects.

3.2.1. Route: /

- **Function Name:** `index`
- **Input Schema:** None
- **Summary:** Returns a simple greeting.

3.2.2. Route: /login

- **Function Name:** `login`

- **Input Schema:**

```
{
  "email": str,
  "password": str
}
```

- **Summary:** Authenticates users and returns JWTs.

3.2.3. Route: `/push_bid_changes`

- **Function Name:** `push_bid_changes`
- **Input Schema:**

```
{
  "client_profile_id": str,
  "debug": bool (optional, default=True),
  "update_status": bool (optional, default=False),
  "update_by_name": bool (optional, default=True),
  "push_to_db": bool (optional, default=True),
  "push_to_api": bool (optional, default=True),
  "hours": tuple (optional, default=(current_hour - 1, current_hour)),
  "check_for_existing_changes": bool (optional, default=True)
}
```

- **Summary:** Pushes bid changes to the API.

3.2.4. Route: `/push_status_changes`

- **Function Name:** `push_status_changes`
- **Input Schema:**

```
{
  "client_profile_id": str,
  "debug": bool (optional, default=True),
  "push_to_api": bool (optional, default=True),
  "hours": tuple (optional),
  "changed_only": bool (optional, default=True),
  "check_for_existing_changes": bool (optional, default=True),
  "update_status": bool (optional, default=True),
  "update_by_name": bool (optional, default=True)
}
```

- **Summary:** Pushes status changes to the API.

3.2.5. Route: `/hello`

- **Function Name:** `hello`
- **Input Schema:** None
- **Summary:** Returns a simple greeting.

3.2.6. Route: `/test_connection`

- **Function Name:** `test_connection`
- **Input Schema:** None

- **Summary:** Tests database connection.

3.2.7. Route: `/product_master`

- **Function Name:** `product_master`
- **Input Schema:**

```
{
  "client_profile_id": str,
  "debug": bool (optional, default=False)
}
```

- **Summary:** Updates product master data.

3.2.8. Route: `/seller_master`

- **Function Name:** `seller_master`
- **Input Schema:**

```
{
  "client_profile_id": str,
  "debug": bool (optional, default=False)
}
```

- **Summary:** Updates seller master data.

3.3. DEPLOYMENT OF THE FLASK APP

Since the app needs to connect to the database and has some requirements, we are using a simple CI/CD based on Github actions to deploy the app. This GitHub Actions workflow is triggered on push events to the master branch and manual workflow dispatch. It consists of two jobs: `build` and `deploy`. Here are all the steps:

3.3.1. Job: Build

Step 1: Checkout Code

```
- uses: actions/checkout@v4
```

Checks out the source code from the repository.

Step 2: Set up Python version

```
- name: Set up Python version
  uses: actions/setup-python@v1
  with:
    python-version: "3.11"
```

Sets up the specified Python version.

Step 3: Create and start virtual environment

```yaml
- name: Create and start virtual environment
  run: |
    python -m venv venv
    source venv/bin/activate
```

Creates and activates a virtual environment.

Step 4: Install dependencies

```yaml
- name: Install dependencies
  run: pip install -r requirements.txt
```

Installs the Python dependencies specified in `requirements.txt`.

Step 5: Install ODBC Server

```yaml
- name: Install ODBC Server
  run: |
    sudo apt-get update
    sudo ACCEPT_EULA=Y apt-get install -y msodbcsql18
    sudo ACCEPT_EULA=Y apt-get install -y mssql-tools18
    echo 'export PATH="$PATH:/opt/mssql-tools18/bin"' >> ~/.
       bashrc
    source ~/.bashrc
```

Installs the ODBC Server for database connectivity.

Step 6: Zip artifact for deployment

```yaml
- name: Zip artifact for deployment
  run: zip release.zip ./* -r
```

Zips the application code and dependencies for deployment.

Step 7: Upload artifact for deployment jobs

```yaml
- name: Upload artifact for deployment jobs
  uses: actions/upload-artifact@v3
  with:
    name: python-app
    path: |
      release.zip
      !venv/
```

Uploads the zipped artifact for later deployment.

3.3.2. Job: Deploy

Step 1: Download artifact from build job

```yaml
- name: Download artifact from build job
  uses: actions/download-artifact@v3
  with:
    name: python-app
```

Downloads the zipped artifact from the build job.

Step 2: Unzip artifact for deployment

```yaml
- name: Unzip artifact for deployment
  run: unzip release.zip
```

Unzips the artifact for deployment.

Step 3: Deploy to Azure Web App

```yaml
- name: "Deploy to Azure Web App"
  uses: azure/webapps-deploy@v2
  id: deploy-to-webapp
  with:
    app-name: "flask-app"
    slot-name: "Production"
    publish-profile: ${{ secrets.XXX }}
```

Deploys the application to Azure Web App using the specified publish profile.

4. MISCELLANEOUS HANDOVERS

4.1. INTRODUCTION

This chapter has some of the code that are not exactly related to AMS. These were handed over to me by other members of the team. There are three such handovers.

4.2. OSA AVAILABILITY MAILER

This is a mailer that creates an HTML report on the availability of various products and then send it to the team handling the client.

4.2.1. `get_campaign_asin_status_data()`

The function retrieves data related to campaign ASIN status. It performs the following steps:

- Retrieves data from the ”osa” table where the date is within the last 24 hours.
- Modifies the ”osa” DataFrame by assigning statuses (”lbb”, ”oos”, ”ins”) based on specific conditions and removes duplicates.
- Retrieves campaign ASIN mapping data and expands the ”asin” column to multiple rows.
- Merges the expanded campaign ASINs with the modified ”osa” DataFrame based on the ”asin” column.
- Renames the ”sp” column to ”price” and returns the resulting DataFrame.

```
def get_campaign_asin_status_data():
    # Code snippet for data retrieval and manipulation
    # ...

    return df
```

4.2.2. `get_sb_availability(data)`

The function processes data related to Sponsored Brands (SB) campaigns and calculates various availability metrics. It includes the following steps:

- Filters the input data to include only SB campaigns and removes duplicate rows based on specific columns.
- Creates columns for different types of ASINs.
- Further filters data to obtain subsets of ASINs with different statuses (available, OOS, lost buybox, unknown).
- Calculates and aggregates counts of available, OOS, lost buybox, and unknown ASINs for each campaign.
- Merges the calculated metrics into a final DataFrame and returns it.

```
def get_sb_availability(data):
    # Code snippet for data filtering and aggregation
```

```
    # ...

    return df5
```

4.2.3. `get_non_sb_availability(data)`

The function filters out SB campaigns from the input data and focuses on ASINs with the "lbb" status. It involves the following steps:

- Filters out SB campaigns from the input data.
- Obtains data on sellers from the "osa_consolidated" table for ASINs with "lbb" status.
- Merges the filtered data with seller information based on the "asin" column.
- Removes duplicate rows and returns the final DataFrame.

```
def get_non_sb_availability(data):
    # Code snippet for data filtering and merging
    # ...

    return final
```

4.2.4. The `Mailer` Class

The provided Python code defines a `Mailer` class for email generation and sending. The constructor initializes instance variables, and the `body` method performs data processing, HTML generation, and status determination.

```
    class Mailer:
        def __init__(self, client_profile_id, client):
            self.client_profile_id = client_profile_id
            self.client = client

        def body(self):
            ...
```

4.2.5. Constructor (`__init__`)

```
    def __init__(self, client_profile_id, client):
        self.client_profile_id = client_profile_id
        self.client = client
```

4.2.6. Body Method

```
    def body(self):
        ...
```

The `body` method involves extensive data processing, HTML generation, and conditional logic. It returns a status code and HTML content based on two dataframes: `sb_df` and `non_sb_df`.

4.2.7. Data Processing and HTML Generation

```
        sb_df = sb_df_orig[sb_df_orig["client_profile_id"] == self.
            client_profile_id]
        sb_df = sb_df[["campaignName", "campaignType", "ASINs in campaign", "
            available ASINs", "available ASINs count", "OOS ASINs", "OOS ASINs
            count", "Lost buybox ASINs", "Lost buybox ASINs count"]]
        sb_df = sb_df[sb_df["available ASINs count"] != 3]
        sb_df["ASINs in campaign"] = sb_df["ASINs in campaign"].fillna(0)
        ...
```

The code processes data in `sb_df` and `non_sb_df`, performs HTML generation, and sets up various HTML elements.

4.2.8. Email Sending

```
    def send_email(self):
        status, comp_html = self.body()
        s = smtplib.SMTP("smtp-mail.outlook.com", 587)
        s.ehlo()
        s.starttls()
        s.login(EMAIL, PASSWORD)
        ...
```

The `send_email` method initiates an SMTP connection, logs in, and sends emails using the generated HTML content.

4.3. CATEGORY BENCHMARK

Amazon provides, each month, a list of keywords and their benchmark performance. This data is saved to SQLVM1 each month for future reference. Earlier, the dataset was pushed to SQLVM5 too but the format of the table in both the table was different and hence data pushing in VM5 was stopped.

4.3.1. preprocess_sheet Function

This preprocesses the sheets from the Excel file. The preprocessing includes:

- Removing columns with too many NaNs
- Removing rows before the header row (which are none)
- Renaming columns
- Adding a column for adType
- Converting columns to float

4.3.2. Overall Summary

The Python script aims to process data from an Excel file, perform necessary preprocessing, and store the refined data in a database. It begins by loading an Excel file, extracting individual sheets, and converting them into Pandas DataFrames. The preprocessing involves eliminating columns with a high percentage of missing values, discarding rows

before the header row, renaming columns, adding an ”adType” column, and converting specific columns to float format. The script then combines the preprocessed DataFrames into a single merged DataFrame and removes rows with all-null values. Postprocessing steps include handling missing values, rounding numeric columns to two decimal places, converting certain columns to lowercase, and adding a ”Date” column with the current date. The data is subsequently connected to a database, and after some column renaming and metric calculations (like ROAS), it is pushed into the database. The script concludes with querying the database and potentially updating specific values. Overall, the script serves as a comprehensive pipeline for data loading, preprocessing, database interaction, and data storage.

> The notebook is not sufficient on its own. I have tried to make the script as robust as possible, however, as the input Excel sheet is not very consistent (the column name or the rows before the actual data starts keep changing), you might need to tweak the code to make it work.

4.4. EVENT INSIGHT

This script has some methods that can be used to analyze the trend of various keywords before and after some date period. This is mainly used to analyze how the search volume of a keyword changes during a sale period. Here is detail on some main functions available in the script.

4.4.1. `get_unique_searches_plot(variations_list)`

```
def get_unique_searches_plot(variations_list):
    # Code for getting unique searches plot
    # ...

    return fig
```

1. **Description:** Generates a plot showing the count of unique searches during an event, comparing overall unique searches and event-related unique searches.
2. **Input:**
   - `variations_list`: List of variations related to searches
3. **Output:** Plot comparing overall and event-related unique searches

4.4.2. `get_volume_plot(var_l_join)`

```
def get_volume_plot(var_l_join):
    # Code for getting volume plot
    # ...

    return fig
```

1. **Description:** Generates a plot illustrating the search volume during an event, comparing overall search volume and event-related search volume.
2. **Input:**
   - `var_l_join`: Variable related to joining data
3. **Output:** Plot comparing overall and event-related search volume

4.4.3. get_event_top_searches(var_l_join)

```
def get_event_top_searches(var_l_join):
    # Code for getting top event-related searches
    # ...

    return top_searches, top_10_keywords
```

1. **Description:** Retrieves the top event-related search terms, their main categories, last nodes, brands, and search volumes. Also generates top 10 keywords for each main category and exports data to Excel files.
2. **Input:**
   - `var_l_join`: Variable related to joining data
3. **Output:**
   - DataFrame containing top event-related search terms
   - DataFrame containing top 10 search terms for each main category
   - Excel files: "overall_searches.xlsx" and "top_10_searches.xlsx"

4.4.4. get_top_5_trend(data)

```
def get_top_5_trend(data):
    # Code for getting top 5 trend
    # ...
```

1. **Description:** Retrieves the most clicked ASINs for the top 5 search terms and generates a trend plot.
2. **Input:**
   - `data`: DataFrame containing search term data
3. **Output:** Trend plot of the top 5 search terms

4.4.5. get_most_clicked_asin_top10(top_10_df)

```
def get_most_clicked_asin_top10(top_10_df):
    # Code for getting most clicked ASINs of top 10 keywords
    # ...

    return final_df1
```

1. **Description:** Retrieves the most clicked ASINs for the top 10 search terms and their corresponding brands.
2. **Input:**
   - `top_10_df`: DataFrame containing top 10 search terms data
3. **Output:** DataFrame containing top 10 search terms, most clicked ASINs, and their brands

4.4.6. get_top_10_main_categories_volume()

```
def get_top_10_main_categories_volume():
    # Code for getting top 10 main categories based on search volume
    # ...

    return main_cat_top10
```

1. **Description:** Retrieves the top 10 main categories based on search volume and generates a bar plot.
2. **Output:** DataFrame containing top 10 main categories based on search volume

4.4.7. `get_top_10_main_categories_unique_searches()`

```
def get_top_10_main_categories_unique_searches():
    # Code for getting top 10 main categories based on unique searches count
    # ...
```

1. **Description:** Retrieves the top 10 main categories based on unique searches count and generates a bar plot.
2. **Output:** None (Displays the plot)

4.4.8. `get_new_dates(start_date, end_date, past)`

```
def get_new_dates(start_date, end_date, past):
    # Code for getting new dates based on input dates
    # ...

    return past_start_date, past_end_date
```

1. **Description:** Gets new dates for past or future dates based on the input dates.
2. **Input:**
   - `start_date`: Start date in the format YYYY-MM-DD.
   - `end_date`: End date in the format YYYY-MM-DD.
   - `past`: If True, returns past dates. If False, returns future dates.
3. **Output:**
   - `past_start_date`: Past start date in the format YYYY-MM-DD.
   - `past_end_date`: Past end date in the format YYYY-MM-DD.

4.4.9. `get_top_10_main_categories_with_highest_jump(past=False)`

```
def get_top_10_main_categories_with_highest_jump(past=False):
    # Code for getting top 10 main categories with highest jump
    # ...
```

1. **Description:** Compares the search volume of main categories before and after the event, identifying the top 10 with the highest percentage jump.
2. **Input:**
   - `past` (optional): If True, compares with past dates. If False, compares with future dates.
3. **Output:** Bar plot showing the top 10 main categories with the highest percentage jump in search volume.

4.4.10. `get_top_10_brands_with_highest_jump(past=False)`

```
def get_top_10_brands_with_highest_jump(past=False):
    # Code for getting top 10 brands with highest jump
    # ...
```

1. **Description:** Compares the search volume of brands before and after the event, identifying the top 10 with the highest percentage jump.
2. **Input:**
   - `past` (optional): If True, compares with past dates. If False, compares with future dates.
3. **Output:** Bar plot showing the top 10 brands with the highest percentage jump in search volume.

4.4.11. `get_brand_type_jump(past=False)`

```
def get_brand_type_jump(past=False):
    # Code for getting brand type jump
    # ...
```

1. **Description:** Compares the search volume of brand types (brand or generic term) before and after the event, showing the percentage jump.
2. **Input:**
   - `past` (optional): If True, compares with past dates. If False, compares with future dates.
3. **Output:** Bar plot showing the brand type jump in search volume.

4.4.12. `get_top_10_brands_top10main_with_highest_jump(top_main_df, past=False)`

```
def get_top_10_brands_top10main_with_highest_jump(top_main_df, past=False):
    # Code for getting top 10 brands in top 10 main categories with highest
        jump
    # ...
```

1. **Description:** Compares the search volume of brands within the top 10 main categories before and after the event, identifying the top 10 brands with the highest percentage jump in each category.
2. **Input:**
   - `top_main_df`: DataFrame containing top 10 main categories data.
   - `past` (optional): If True, compares with past dates. If False, compares with future dates.
3. **Output:** Bar plots showing the top 10 brands with the highest percentage jump in each top 10 main category.

4.4.13. `get_top_5_subcat_top10main_with_highest_jump(top_main_df, past=False)`

```
def get_top_5_subcat_top10main_with_highest_jump(top_main_df, past=False):
    # Code for getting top 5 sub-categories in top 10 main categories with
        highest jump
    # ...
```

1. **Description:** Compares the search volume of sub-categories within the top 10 main categories before and after the event, identifying the top 5 sub-categories with the highest percentage jump in each category.
2. **Input:**
   - `top_main_df`: DataFrame containing top 10 main categories data.

- `past` (optional): If True, compares with past dates. If False, compares with future dates.

3. **Output:** Bar plots showing the top 5 sub-categories with the highest percentage jump in each top 10 main category.

To use this notebook, you start with a list of keywords that you need to track, then add negative keywords to avoid unintended keywords getting into the list. For example, suppose you want to analyze the keywords during the prime days. You can start with keywords like `prime day`, `sale` etc. and then negative keywords like `warehouse sales`, `wholsale` etc.

THE ALGORITHM

5. ALGORITHM OVERVIEW

5.1. OVERVIEW

This part of the document discusses all the aspects of AMS bid optimization and budget optimization. The initial chapters of the part will discuss various components used to construct the algorithm that powers bid optimization. Once this is done, a couple of chapters will be used to bring it all together.

5.2. COMPONENTS

Here is a list of some components used in budget optimization. They will be covered in detail later on.

1. **Hygiene Scoring:** This is a score that is assigned to each ASIN. It is a measure of how well the parameters like delivery days, ratings etc. as well as how relevant the ASIN is to the keyword it is targeted by. See the chapter Hygiene Scoring for more details.
2. **Optimum Bid Range:** This module uses the last 60 days data to create an optimum bid range for each targets. This bid range is used to make sure that the bids are not very high. See the chapter Optimum Bid Range for more details.
3. **Bucketing:** This module is used to bucket the targets into various buckets. The buckets are decided based on the past performance of the targets. See the chapter Buckets for more details.
4. **Calculation of Desired Rank:** The bid of a target depend on the desired rank of the target. This module is used to calculate the desired rank of each target. See the chapter Desired Rank for more details.
5. **Rule Based Bidding:** This module is used to bid on the targets based on some rules that were created by the team. See the chapter Rule Based Bidding for more details.
6. **Visibility Based Bidding:** This module is used to bid on the targets based on the visibility of the targets. See the chapter Visibility Based Bidding for more details.
7. **Other Rules:** There are a number of other rules that are applied on the targets once it has passed through rule based bidding and visibility based bidding. See the chapter Other Rules for more details.
8. **The Configuration File:** This is a file that contains all the parameters that are used in the algorithm. See the chapter The Configuration File for more details.

5.3. HOW BID OPTIMIZATION IS DONE?

There are a number of steps involved in bid optimization. The following is a list of steps that are followed to optimize the bids:

1. We start by creating a list of ASINs that are to be promoted. This is done so that some ASINs that are, say, out of buybox are not promoted, and we do not spend money on them.
2. Then we create hygiene scores for each promotable ASINs. This is required as Amazon uses not only bid but the relevancy score to show the ads.
3. Next, the performance for a number of time windows is calculated. A number of time

Odd Clients	Even Clients
nykaa	kiro
so good	curatio
taali	indica
vip	petterati
streax	salon
minelab india	chik
uppercase	spinz

Odd and Even Clients

windows are required because the buckets are decided based on the performance of the targets in the past.

4. Now, we are ready to create the buckets. The targets are divided into buckets based on the performance of the targets in the past. The buckets are used for various purposes like deciding which targets should be used as visibility targets or which targets to pause first when the budget is exhausted.
5. Next step is calculating the desired rank of each target. The desired rank is calculated based on the performance and the hygiene score of the targets. The desired rank is used to calculate the bid of the targets and to mismatch the targets. See the chapter Desired Rank for more details.
6. Now, we are ready to calculate the bid of each target. Calculation of bids is divided into two categories: rule based bidding is done for all those targets that are not visibility targets. See the chapter Rule Based Bidding for more details. Visibility based bidding is done for all those targets that are visibility targets. For this, we use the calculated desired rank. See the chapter Visibility Based Bidding for more details.
7. After this, the targets are passed through a number of rules. Some of them are done to ensure that we have proper bids of the targets and some are to make sure that the bid is neither too high nor too low. See the chapter Other Rules for more details.

5.4. TIMINGS OF THE ALGORITHM

The algorithm is run at a number of times during the day. There are mainly three codes as part of the overall algorithms. The timing is set based on whether the client correspond to odd or even. As of writing this document, here is the list of odd and even clients:

5.4.1. Morning Changes

This is run in the morning and does most of the heavy lifting. It does everything mention in the previous section. The time of running this iteration is at 0900 IST (for odd clients) and 1000 IST (for even clients). The later iterations are dependent on this iteration.

5.4.2. Hourly Changes

This is run at 3-4 times depending on how frequently the visibility is crawled by the crawler team. The only purpose of this iteration is to change the bids of the visibility targets based on the desired rank and the current rank. The time of running this iteration is at 1200, 1500 and 1800 IST (for odd clients) and 1300, 1600 and 1900 IST (for even clients).

5.4.3. Pause Targets

This iteration is run to ensure that the projections provided by the clients is met. We do this by pausing the targets that are not performing well when budget starts to exhaust. The time of running this iteration is every two hour 0200 IST to 2200 IST. For now, both the odd and even clients are run at the same time. In the future, this might change.

Here is the a table that summarizes the timings of the algorithm:

Iteration	**Odd Clients**	**Even Clients**
Morning Changes	0900 IST	1000 IST
Hourly Changes	1200, 1500, 1800 IST	1300, 1600, 1900 IST
Pause Targets	Every Two Hours	Every Two Hours

Timings of the Algorithm

6. HYGIENE SCORING

6.1. AIM

Amazon uses two parameters while shoeing relevant ads: the bid and the relevancy of the search term (or product) with the target. This means that even if bid is very high for a keyword, it might not show up if the relevancy is low. Due to this, while calculating the bid, we need to take into account the relevancy of the product with the search term. This is where the hygiene score comes into play. The hygiene score is a number out of 100 that is given to an ASIN based on a number of parameters like rating, delivery days, etc.

6.2. ASSUMPTIONS

We make the following assumptions while calculating the hygiene score:

1. The parameters that we use to calculate the hygiene score are independent of each other.
2. The parameters are latest and are updated regularly. They do not change within the day.
3. The parameters used provide a good enough estimate of the hygiene of the product.

6.3. PARAMETERS

This section details the parameters that we use to calculate the hygiene score and how the score will change if the parameter changes. We create a score between 0 and 100 for each of the parameters and then take a weighted average of the scores to get the final hygiene score. What this means that if w_i is the weight of the i^{th} parameter and s_i is the score of the i^{th} parameter, then the hygiene score is given by:

$$\text{Hygiene Score} = \sum_{i=1}^{n} w_i \times s_i \tag{6.1}$$

where n is the number of parameters.

6.3.1. Rating

Rating is the average rating of the product. It is a number between 0 and 5. The higher the rating, the better the hygiene score. The hygiene score is directly proportional to the rating. Since we need a score out of 100, the following formula is used to generate the rating score:

$$\text{Rating Score} = \frac{\text{Rating}}{5} \times 100 = \text{Rating} \times 20 \tag{6.2}$$

6.3.2. Delivery Score

Delivery score is the number of days it takes for the product to be delivered. The lower the number of days, the better the hygiene score. The hygiene score is inversely proportional to the delivery score. Tthe following formula is used to generate the delivery score:

$$\text{Delivery Score} = \frac{1}{\text{Delivery Days}} \times 100 \tag{6.3}$$

Deal Score

Deal is a promotion that is run on the product. This can be something like *deal of the day*. This is a binary parameter. If the product has a deal, then the deal score is 100, else it is 0.

Discount Score

Discount is the percentage discount on the product. The higher the discount, the better the hygiene score. We use the percentage of discount as the discount score. This means that if the discount is 50%, then the discount score is 50. Given the maximum retail price (MRP) and the selling price (SP), the discount score is given by:

$$\text{Discount Score} = \frac{\text{MRP} - \text{SP}}{\text{MRP}} \times 100 \tag{6.4}$$

6.3.3. Keyword and Attribute Relevancy

The keyword and attribute relevancy were used earlier to calculate the hygiene score. However, they are not used anymore. In the future, if they are used, we can directly use the relevancy score as the hygiene score.

6.3.4. The Final Score

We use the following weights for the parameters:

Parameter	Weight
Rating	0.3
Delivery	0.3
Deal	0.15
Discount	0.25

Weights of the parameters in the hygiene score

Using this, we get the following formula for the hygiene score:

$$\text{Hygiene Score} = 0.3\times\text{Rating Score}+0.3\times\text{Delivery Score}+0.15\times\text{Deal Score}+0.25\times\text{Discount Score}$$

6.4. THE CODE FOR HYGIENE SCORE

The `create_asin_scores()` function implements the logic for the hygiene score. Here are the details of the function:

```
def create_asin_scores(client_profile_id, use_prev_day=False):
    # Code for creating asin scores based on osa data
    # ...
```

1. **Description:** Creates asin scores using osa data and various parameters such as rating, delivery days, deal, discount, etc. It also includes a status column indicating whether the asin is promoted or not.
2. **Parameters:**
   - `client_profile_id`: The client_profile_id for which the asin scores are to be created.
   - `use_prev_day` (optional): If True, uses data from the previous day if no data is found for today.
3. **Returns:**
   - `osa_df`: The osa data with asin scores.
4. **Raises:**
   - `QueryReturnedNoData`: If no data is found in the osa table for today (or previous day if `use_prev_day` is True).
   - `UnexpectedResultFound`: If the osa data does not seem to be correct.
5. **Notes:**
   - Detail about asin is obtained from the osa and product master tables.
   - The score is calculated based on parameters such as rating, delivery days, deal, discount, etc. All scores are normalized to 100.
   - If a parameter is null or empty, the corresponding score is set to 0.

6.5. SQL SCHEMA

6.5.1. Tables From Which Data is Used

1. **osa:** This table contains the osa data for the client profile id. The following main columns are used:
   - `asin`: Amazon Standard Identification Number (ASIN) of the product.
   - `parent_asin`: Parent ASIN of the product.
   - `sp`: Selling price of the product.
   - `mrp`: Maximum Retail Price (MRP) of the product.
   - `rating`: Rating of the product.
   - `delivery_days`: Number of days it takes for the product to be delivered.
   - `deal`: Indicates whether the product has a deal or not.
   - `available`: Indicates product availability.
   - `buybox`: Indicates whether the product is in the buy box or not.
2. **onboarding_product_master_updated:** This table contains the onboarding product master updated data. The following columns are used:
   - `asin`: Amazon Standard Identification Number (ASIN) of the product.
   - `mrp (as float)`: Maximum Retail Price (MRP) of the product, casted as a floating-point number.

The data is not pushed to any table. It is used directly in the algorithm.

6.6. LIMITATIONS

1. The hygiene score is calculated only once a day. This means that if the hygiene score changes during the day, the algorithm will not take it into account.
2. A number of parameters are not used in the hygiene score. These include keyword and attribute relevancy, etc.

7. OPTIMUM BID RANGE

7.1. AIM

The rule based bids do not take into consideration the fact that a lot of time, increasing bids is not the best way to get better performance. A number of times, it happens that a target performs the best (in terms of ACOS) in a range of bids. This range is called the optimum bid range. This chapter describes the method to find the optimum bid range for a target.

7.2. ASSUMPTIONS

A number of assumptions are inherent in the method described in this chapter. These assumptions are listed below:

1. The optimal bid of the targets is independent of the date. That is, if the targets has been performing best 5 INR during the last two months, it will continue to perform best at 5 INR in the future as well.
2. The optimal bid does not depend on the spend or sales of a single day but only on the average spend and sales of the target.
3. Data for last 60 days is available.

7.3. ALGORITHM

The algorithm uses the last 60 days of data. The idea is to use the data for last 60 days to calculate a mean and standard deviation of the bids that minimize or maximize the required metrics. Later, we can use these two values to create the bid range.

7.3.1. Metrics Available in the Algorithm

The algorithm supports the following metrics:

1. **ACOS**: Minimize the ACOS.
2. **clicks**: Maximize the spend.
3. **Sales**: Maximize the sales.
4. **Impressions**: Maximize the impressions.

Other metrics can be added easily. Here is the algorithm:

7.3.2. Remove the Outliers

The first step involves removing the outliers. This is necessary because the outliers can skew the results. The outliers are removed from both the end, that is, the items that are very low from the minimum and very high from the maximum are removed. The outliers are removed using the quartiles. The method is as follows:

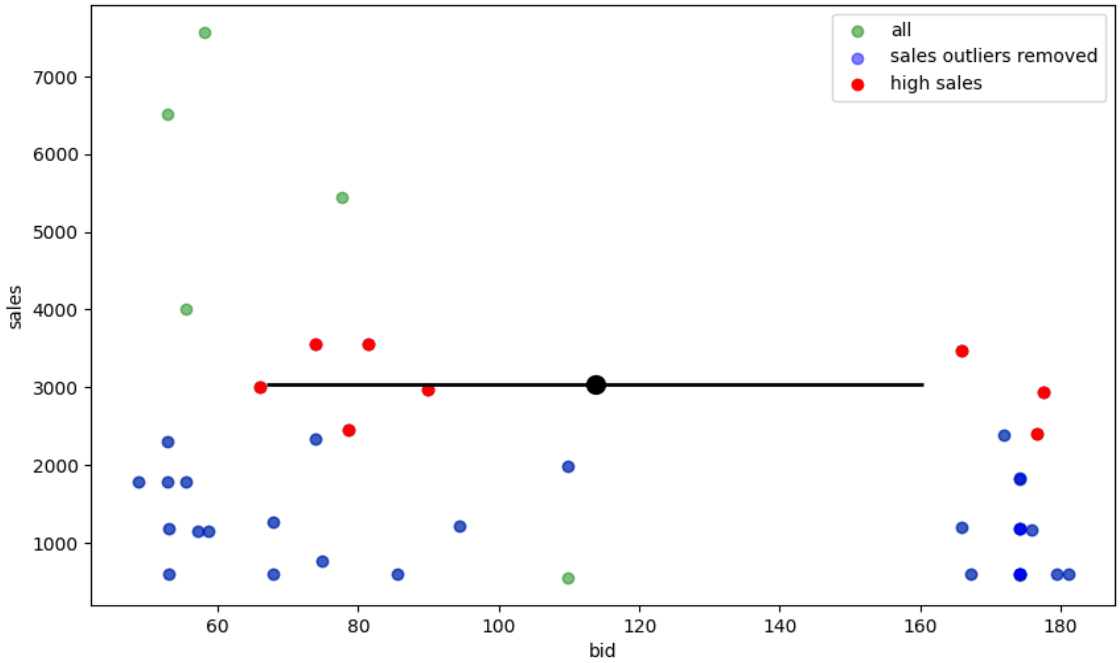
1. Start with an epsilon value of, say 0.2. Call it ϵ.
2. This value is used to remove the top and bottom ϵ fraction of the data.
3. If less than 3 items are left, then decrease the value of ϵ by 0.05 and repeat the process until either 3 items are left or ϵ becomes 0.05.

> At least three items are required to calculate the optimum bid range because we want to determine the mean and standard deviation of the data. If there are less than three items, then the mean and standard deviation will not be good.

7.3.3. Get The Best Values

Next, we take 10 or 20% of the items with the best metrics. For example, if the metrics is sales, we will take the top 10 or 20% of the items with the highest sales and get the bids corresponding the items. We will use these bids to calculate the mean and standard deviation. The algorithm also supports getting the minimum and maximum values instead of the mean and standard deviation. This is done by setting the `data_return_type` parameter to `min_max`.

Here is a visualization of the algorithm:

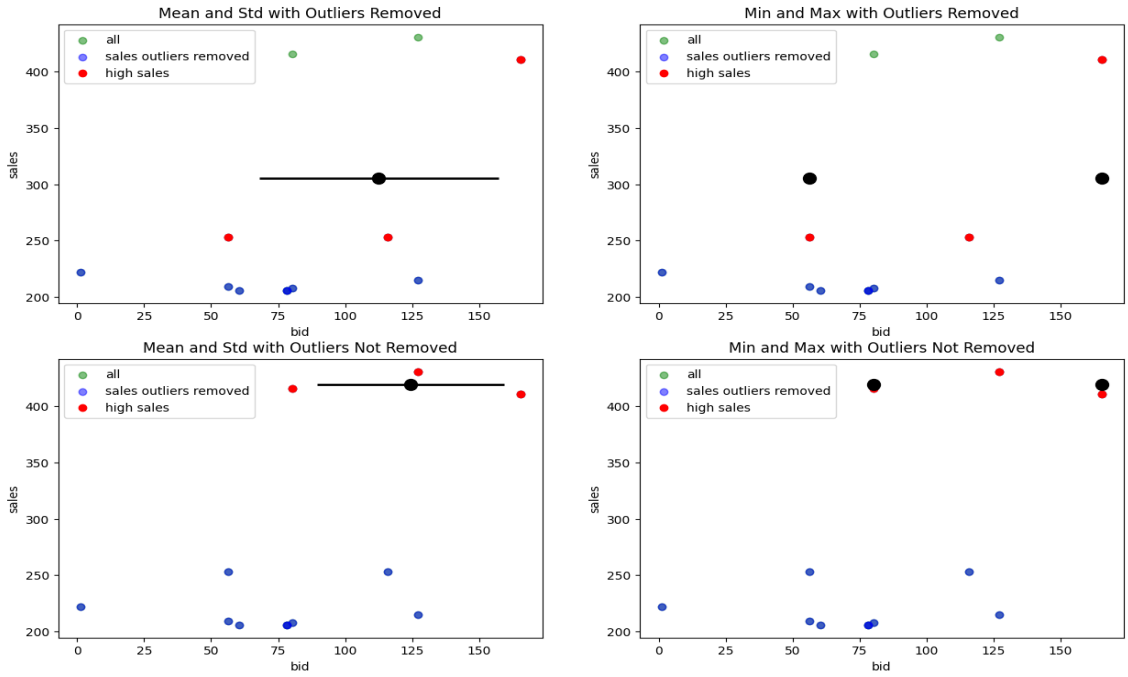


The Algorithm to Get the Best Values

7.4. EFFECT OF PARAMETERS

7.4.1. Effect of Outliers

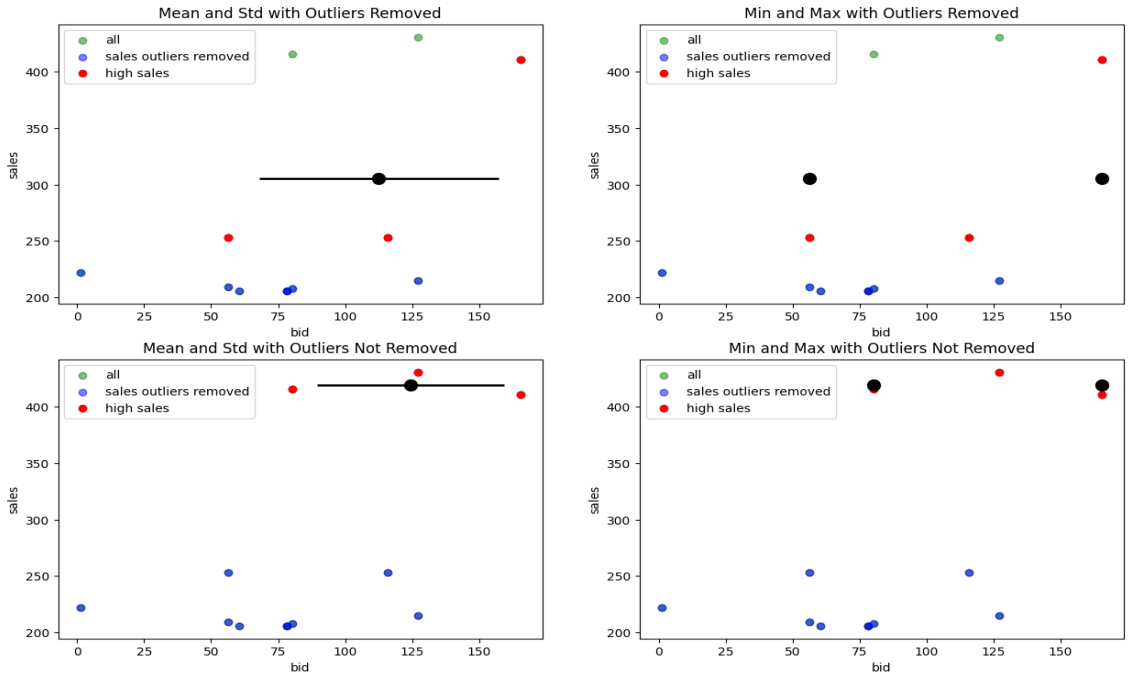
The following figure shows the effect of outliers on the bid range. The outliers are removed using the method described above. The figure shows that the outliers have a significant effect on the bid range. The bid range is much higher when the outliers are not removed.



Effect of Outliers on the Bid Range

7.4.2. Effect of Metrics

The bid range changes based on what metrics is used. The following figure shows the effect of metrics on the bid range. The bid range changes significantly based on the metrics.



Effect of Metrics

7.5. BIDRANGE CLASS

The BidRange class is designed to predict a bid range based on a specified column name. The column name can be sales, clicks, impressions, or acos. The class is initialized with a pandas DataFrame, a column name to use, and an optional logger. If no logger is provided, a new one is created.

```
class BidRange:
    def __init__(self, data, column_to_get="bid", logger=None):
        # Code implementation
```

7.5.1. _remove_outliers Method

The _remove_outliers method removes outliers from an array based on a specified quantile.

```
def _remove_outliers(self, array, eps=0.1, verbose=False):
    # Code implementation
```

7.5.2. _get_bids_for_top_value Method

The _get_bids_for_top_value method returns the ids of values that are within a certain quantile range.

```
def _get_bids_for_top_value(self, array, eps=0.80, verbose=False):
    # Code implementation
```

7.5.3. _get_bids_for_bottom_value Method

The _get_bids_for_bottom_value method returns the ids of values that are within a certain quantile range.

```
def _get_bids_for_bottom_value(self, array, eps=0.10, verbose=False):
    # Code implementation
```

7.5.4. visualize_one Method

The visualize_one method visualizes the bid range for a single targeting id.

```
def visualize_one(self, sample_id, ax=None, legend=False, column_name="sales",
    strategy="maximize", data_return_type="mean_std", remove_outliers=True):
    # Code implementation
```

7.5.5. _bid_range_one Method

The _bid_range_one method calculates the mean and standard deviation of the bids for a single targeting id based on a specified column name and strategy.

```
def _bid_range_one(self, targeting_id, column_name="sales", strategy="maximize", min_rows_to_use=3, min_rows_breach_behaviour="warn", data_return_type="mean_std", remove_outliers=True):
    # Code implementation
```

7.5.6. _bid_range Method

The _bid_range method calculates the mean and standard deviation of the bids for all column names and strategies.

```
def _bid_range(self, targeting_id, min_rows_to_use=3, min_rows_breach_behaviour="leave", data_return_type="mean_std", remove_outliers=True):
    # Code implementation
```

7.5.7. bid_range_all Method

The bid_range_all method applies the _bid_range method to all targeting ids in the DataFrame and returns a DataFrame with the results.

```
def bid_range_all(self, min_rows_to_use=3, min_rows_breach_behaviour="leave", data_return_type="mean_std", remove_outliers=True):
    # Code implementation
```

8. BUCKETS

8.1. AIM

Bucketing the targets is one of the most important steps in the bid optimization process. Not all the targets perform the same and hence we have divided the targets into a number of buckets. These buckets are used to decide the bid optimization strategy for each target. These buckets are also responsible for deciding when to pause the targets when the budget starts to run out.

8.2. ASSUMPTIONS

Some assumptions related to the buckets are as follows:

- The buckets are independent of each other.
- The buckets are mutually exclusive. (This does not always hold true and hence we also define a priority for each bucket.)
- The buckets are exhaustive. What this means that each target will fall into one of the buckets. (Again, this does not always hold true and hence we have defined a bucket called *no bucket* for targets that do not fall into any of the buckets.)

8.3. BUCKETS AND THEIR DESCRIPTION

8.3.1. Parameters Used for Bucketing

A number of parameters are used for bucketing. These parameters are as follows:

- **Sales**: The number of sales.
- **ACOS**: The ACOS.
- **Spend**: The spend.
- **Clicks**: The number of clicks.
- **Impression**: The number of impressions.
- **Frequency of Sales**: Whether the sales is regular, intermittent or no sales.

There are mainly three types of buckets:

8.3.2. Performing Buckets

These are the buckets that are performing well and the acos is either within limit or is very close to the target acos. There are three families of performing buckets:

1. **Star Buckets**: These are the buckets that are performing well and the acos is within the target. There are four star buckets:
   - **Star_1**: These are the targets that are performing well and the acos is within the target. Also, the sales are regular.

- **Star_2**: These are the targets that are performing well and the acos is within the target. However, the sales are intermittent.
- **Star_3**: These are the targets that were performing well for the last 45 days but have no sale for the last 15 days. Sales is intermittent.
- **Star_4**: These are the targets that are have a high acos yesterday but the average acos for the last 15 days is within the target. Sales are regular.

2. **Crucial Buckets**: Star bucket uses at least 15 days of data and this may result in leaving out some opportunities for bid optimization. Crucial buckets are used to capture these opportunities by considering last day and the day before yesterday data. There are four crucial buckets:
   - **Crucial_1**: Last day acos less than target and less than last seven days average acos.
   - **Crucial_2**: Last day acos greater than target and less than last seven days average acos.
   - **Crucial_3**: Last day acos greater than target and greater than last seven days average acos.
   - **Crucial_4**: Last day acos less than target and greater than last seven days average acos.
3. **Inflated Buckets**: These are the buckets that are performing well but the acos is more than the target. There are two inflated buckets:
   - **Inflated_1**: These are the targets that are performing well but the acos is more than the target.
   - **Inflated_2**: These are the targets that are performing well but the acos is more than the target. Also, the sales are intermittent.

8.3.3. Non-Performing Buckets

These are the buckets that are either not giving any sale or have very high acos. There are two families of non-performing buckets:

1. **Alarming Buckets**: These are the buckets that are not performing well and the acos is more than the target. There are three alarming buckets:
   - **Alarming_1**: The target have spend a lot of money in the last seven days but have not given any sale.
   - **Alarming_2**: The target have spend a lot of money in the last 45 days but have not given any sale.Also, the acos is more than three times the target.
   - **Alarming_3**: The targets have brought sales but the acos is more than three times the target.
2. **Growth Buckets**: These are the buckets that have not brought any sales/clicks/impressions. Here are the buckets in this family:
   - **Growth_1**: Clicks are less than 10 with non-zero impressions in the last seven days.
   - **Growth_2**: No impressions in the last seven days.

8.3.4. No Bucket

This is the bucket for targets that do not fall into any of the above buckets. Ideally, this should not happen but it does happen. This bucket is used to decide the bid optimization strategy for targets that do not fall into any of the above buckets.

8.4. BUCKETS AND THEIR PROPERTIES

Bucket	Priority
Growth_1	1
Growth_2	2
Alarming_1	3
Inflated_1	4
Inflated_2	5
Star_4	6
Star_2	7
Star_1	8
Star_3	9
Crucial_1	10
Crucial_2	11
Crucial_3	12
Crucial_4	13
Alarming_2	14
Alarming_3	15
No Bucket	16

Bucket priority

8.4.1. Priority

As discussed, it is not guaranteed that a target will fall into only one bucket. Hence, we have defined a priority for each bucket. The buckets are as follows:

8.4.2. Time Window

Each bucket has a time window. The metric for this time window is used to decide the bucket for a target. The time window for each bucket is as follows:

8.4.3. Actionable

Each bucket has a set of actions associated with it. These actions are used to decide the bid optimization strategy for a target. The actions for each bucket are as follows:

Bucket	Time Window
Growth_1	Last 7 days
Growth_2	Last 7 days
Star_1	Last 15 days and Last 30 days
Star_2	Last 15 days and Last 30 days
Star_3	Last 15 days and Last 60 days
Star_4	Last day, Last 7 days and Last 15 days
Inflated_1	Last 15 days and Last 30 days
Inflated_2	Last 15 days and Last 30 days
Crucial_1	Last day and Last 7 days
Crucial_2	Last day, Last 7 days and Last 15 days
Crucial_3	Last day and Last 7 days
Crucial_4	Last day and Last 7 days
Alarming_1	Last 7 days
Alarming_2	Last 45 days (only on Monday and Thursday)
Alarming_3	Last 45 days (only on Monday and Thursday)

Buckets and Time Windows

8.4.4. Bucket and Filter

Each bucket has a filter associated with it. This filter is used to decide whether a target falls into a bucket or not. The filter for each bucket is as follows:

8.5. BUCKETING FUNCTION

8.5.1. _create_bucket Function

```
def _create_bucket(data, mask, bucket_name, action, priority):
    bucket = data[mask].reset_index(drop=True)
    bucket["bucket"] = bucket_name
    bucket["action"] = action
    bucket["bucket_priority"] = priority
    return bucket
```

This function filters rows from the input data using a specified mask and creates a bucket with a given name, action, and priority. The function takes a pandas DataFrame as input data, a pandas Series as a mask, a string as the bucket name, another string as the action, and an integer as the priority. It returns a DataFrame that represents the bucket.

8.5.2. bucketing Function

Bucket	Action Taken
Growth_1	Increase gradually
Growth_2	Increase gradually
Alarming_1	Decrease by 20%
Inflated_1	Decrease gradually
Inflated_2	Decrease gradually
Star_4	No action
Star_1	Increase gradually
Star_2	Increase gradually
Star_3	Increase gradually
Crucial_1	Increase gradually
Crucial_2	No action
Crucial_3	Decrease gradually
Crucial_4	No action
Alarming_2	Pause
Alarming_3	Pause

Buckets and Actions Taken

```
def bucketing(data, brand_bidding_strategy="boost", performing_buckets=None,
    pause_non_performing_targets=False):
    # ... (code omitted for brevity)
    return merged_df
```

This function creates buckets for the input data using the _create_bucket function. It takes a pandas DataFrame as input data, a string as the brand bidding strategy (default is ”boost”), a list of performing buckets, and a boolean to indicate whether to pause non-performing targets or not (default is False). The function returns a DataFrame with the input data bucketed according to the specified rules.

Bucket	Mask Used
Growth_1	Last 7 days clicks less than or equal to 10 and Last 7 days impressions greater than 0
Growth_2	Last 7 days impressions equal to 0
Alarming_1	Last 7 days sales equal to 0 and (Last 7 days clicks greater than 100 divided by Last 7 days CVR or Last 7 days spend greater than SP times target ACOS divided by 100)
Inflated_1	Last 30 days frequency equals regular and Last 15 days ACOS greater than target ACOS plus 20
Inflated_2	Last 15 days ACOS greater than target ACOS plus 20 and Last 30 days frequency equals intermittent
Star_4	Last day ACOS greater than target ACOS plus 20 and Last 15 days ACOS less than or equal to target ACOS plus 20 and Last 15 days ACOS greater than 0
Star_2	Last 30 days frequency equals intermittent and Last 15 days ACOS less than or equal to target ACOS plus 20 and Last 15 days ACOS greater than 0
Star_1	Last 30 days frequency equals regular and Last 15 days ACOS less than or equal to target ACOS plus 20 and Last 15 days ACOS greater than 0
Star_3	Last 60 minus 15 days ACOS less than or equal to target ACOS plus 20 and Last 15 days sales equal to 0 and Last 60 minus 15 days ACOS greater than 0
Crucial_1	Last day ACOS less than target ACOS plus 20 and Last day ACOS less than Last 7 days ACOS and Last day ACOS greater than 0
Crucial_2	Last day ACOS greater than or equal to target ACOS plus 20 and Last day ACOS less than Last 7 days ACOS and Last 7 days ACOS less than target ACOS plus 20 and Last day ACOS greater than 0
Crucial_3	Last day ACOS greater than or equal to target ACOS plus 20 and Last day ACOS greater than or equal to Last 7 days ACOS
Crucial_4	Last day ACOS less than target ACOS plus 20 and Last day ACOS greater than or equal to Last 7 days ACOS and Last day ACOS greater than 0
Alarming_2	Last 45 days sales equal to 0 and (Last 45 days clicks greater than 100 divided by Last 45 days CVR or Last 45 days spend greater than SP times target ACOS divided by 100) and keyword type not equal to brand
Alarming_3	Last 45 days ACOS greater than 3 times target ACOS and keyword type not equal to brand

Buckets and Filter

9. DESIRED RANK

9.1. AIM

In order to decide which target should be given higher bid, we define a concept of desired rank. The desired rank of a target is the rank that is allocated based on the performance of the target. A target that has a higher desired rank should be given a higher bid.

9.2. ASSUMPTIONS

Here are some of the assumptions that we make in order to define the desired rank.

1. Desired rank only depend on some parameters of the target.
2. Higher bid results in greater performance

9.3. CALCULATING THE DESIRED RANK

The desired rank depends on a number of parameters of the target. The idea is simple, we group by the targets based on the keyword, match type and ad type. Then we sort the targets in each group based on the performance of the targets and the highest performing target is given the lowest desired rank.

9.3.1. calculate_desired_rank Function

```
def calculate_desired_rank(data):
    # ... (code omitted for brevity)
    return final_df
```

This function generates the desired rank for the data using sales, hygiene score, and other metrics. The function takes as input a pandas DataFrame ‘data‘ which should contain the data with buckets.

The function first sorts the data by the following metrics: ‘l45d_sales‘, ‘hygiene_score‘, ‘l45d_acos‘, ‘l45d_clicks‘, and ‘l45d_impressions‘. The data is sorted in descending order by ‘l45d_sales‘, ‘hygiene_score‘, ‘l45d_clicks‘, and ‘l45d_impressions‘ and in ascending order by ‘l45d_acos‘.

The function then groups the data by ‘Targeting‘, ‘Match_Type‘, and ‘campaignType‘. For each group, it assigns a new rank to the data in the group. The rank is assigned in ascending order, starting from 1. The results are concatenated into a new DataFrame ‘final_df‘.

Finally, the function checks if the length of ‘final_df‘ is equal to the length of the input data. If it is not, it raises a ‘MergeNotCorrect‘ error. The function then returns ‘final_df‘.

9.3.2. Sorting Metrics

The data is sorted by the following metrics:

1. **Last 45 Days Sales** The sales in the last 45 days. The data is sorted in descending order by this metric.
2. **Hygiene score** The hygiene score of the data. The data is sorted in descending order by this metric.
3. **Last 45 Days ACOS** The ACOS (Advertising Cost of Sales) in the last 45 days. The data is sorted in ascending order by this metric.
4. **Last 45 Days Clicks** The number of clicks in the last 45 days. The data is sorted in descending order by this metric.
5. **Last 45 Days Impression** The number of impressions in the last 45 days. The data is sorted in descending order by this metric.

9.3.3. Grouping Parameters

The data is grouped by the following metrics:

1. **Targeting:** The targeting method used for the data.
2. **Match Type:** The match type used for the data.
3. **Campaign Type:** The type of campaign used for the data.

The ‘calculate_desired_rank‘ function then assigns a rank to each group based on the sorting metrics. This rank is used to determine which targets are given preference during the optimization.

9.4. UPDATING DESIRED RANK

The desired rank is later updated based on other parameters, such as if the target belongs to spotlight ads. Here are some ways the desired rank can be updated:

9.4.1. desired_rank_for_spotlight Function

```
1 def desired_rank_for_spotlight(data, return_merged=False):
2     # ... (code omitted for brevity)
3     return data_updated
```

This function updates the desired rank for spotlight targets so that they are ranked higher than Sponsored Brand (SB) targets. The function takes as input a pandas DataFrame ‘data‘ and an optional boolean ‘return_merged‘. The DataFrame should contain the data with buckets. If ‘return_merged‘ is set to True, the function returns the merged data, else it returns the updated data.

The function first creates a new column ‘new_campaignType‘ which is a copy of the ‘campaignType‘ column. It then creates a mask for the Sponsored Brand Video Campaign (SBVC) spotlight targets. If no SBVC spotlight targets are found, the function logs a warning and returns None and the input data.

If SBVC spotlight targets are found, the function updates the ‘new_campaignType‘ for these targets to ”SB_SPOT” and sets their ‘desired_rank‘ to 0. This is a placeholder value that will be updated later.

The function then updates the ‘new_campaignType‘ for SB targets that have the same targeting as the SBVC spotlight targets to ”SB_SPOT”.

The function then groups the data by ‘Targeting‘, ‘Match_Type‘, and ‘new_campaignType‘ and for each group, it sorts the data by ‘desired_rank‘ in ascending order. This ensures that the spotlight targets come first. It then assigns new ranks to the data in the group, with the spotlight targets getting the highest ranks.

Finally, if ‘return_merged‘ is set to True, the function merges the updated data with the rest of the data and returns the merged data. If the length of the merged data is not equal to the length of the input data, the function raises a ‘MergeNotCorrect‘ error.

10. RULE BASED BIDDING

10.1. AIM

A large number of targets are those that are not being tracked for the visibility and hence these targets are updated using the rules defined by the team. These are simple rule such as what percentage of the bid should be increased/decreased if the match type/target type and the last bid of the target is known.

10.2. ASSUMPTIONS

1. The performance of a target depend on the bid. Higher the bid, higher the sales.
2. The bid change reflects in the performance of the target very quickly.

10.3. THE RULES

The rules are mainly divided into three parts:

1. **Increase Gradually**
2. **Decrease Gradually**
3. **No Changes**

The bids are increased or decreased based on these three actions. We use the average CPC of the whole category for this.

10.3.1. Average CPC

calculate_avg_cpc Function

```
def calculate_avg_cpc(client_profile_id):
    # ... (code omitted for brevity)
    return cpc
```

This function calculates the average cost per click (CPC) for a given client profile id for the last 7 days. The function takes as input a string ‘client_profile_id‘ which is the client profile id of the client.

The function first constructs a SQL query to calculate the average CPC from the ‘AMS_Automation_Consolidated.dbo.table_targeting‘ table where the ‘client_profile_id‘ matches the input and the date is between the current date minus 8 and the current date minus 1.

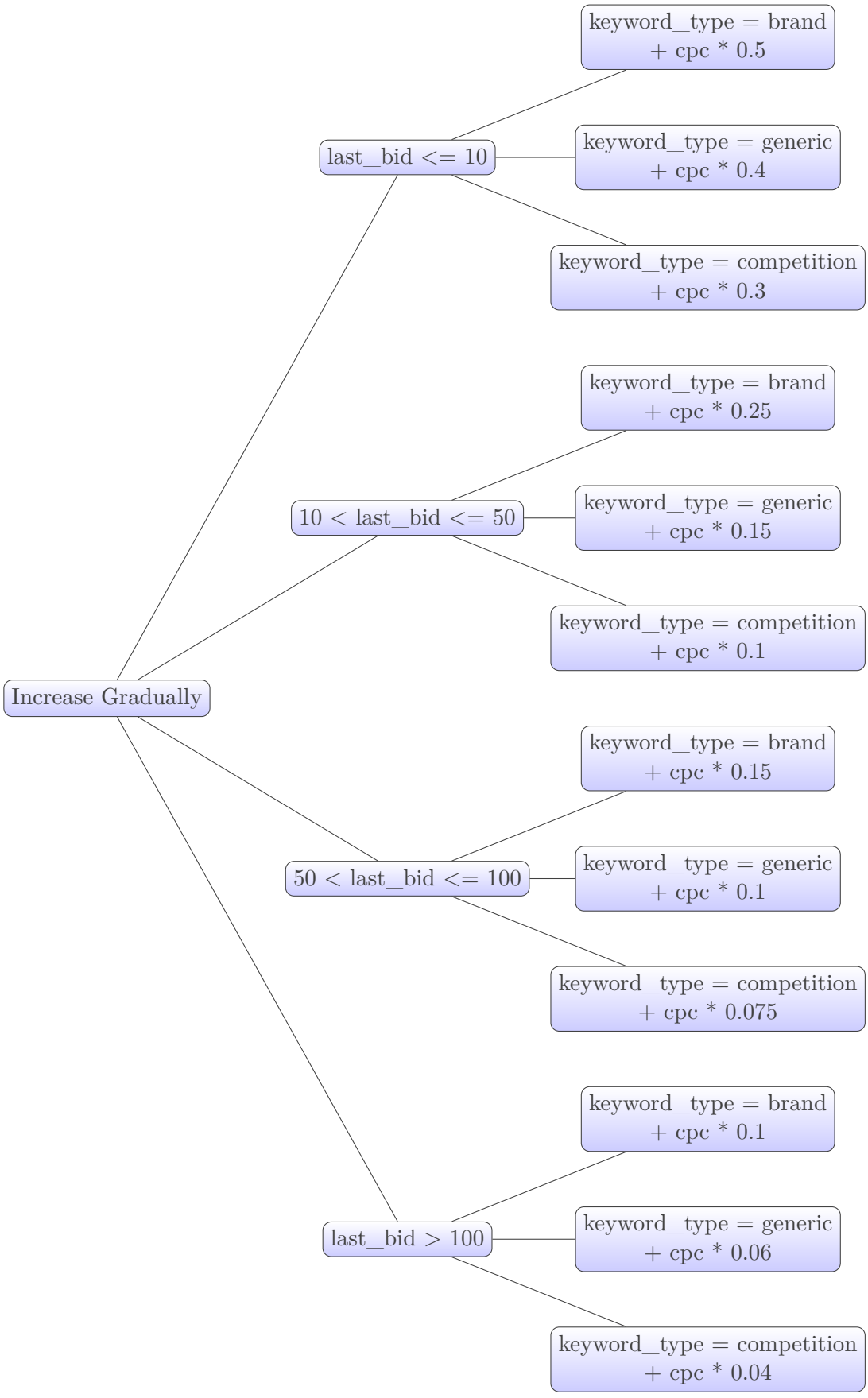
The function then runs the query and retrieves the average CPC. If the average CPC is null or zero, the function raises an ‘UnexpectedResultFound‘ error. If the average CPC is not null or zero, the function logs the CPC and returns it.

10.3.2. Increase Gradually

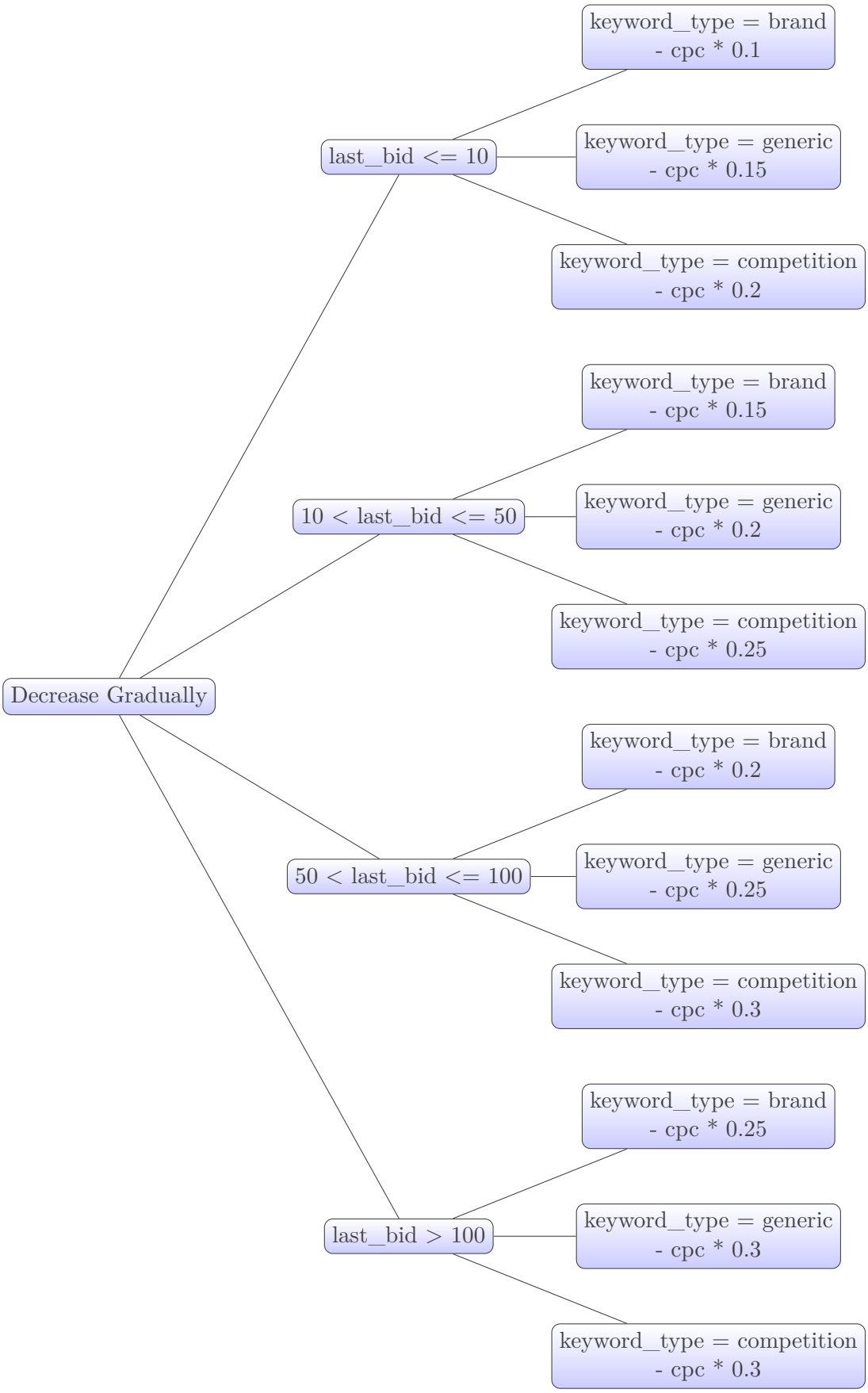
The bids are increased gradually based on the last bid of the target. Here is a figure that shows the percentage increase in the bid based on the last bid of the target. See the figure 10.1.

10.3.3. Decrease Gradually

The below figure shows the percentage decrease in the bid based on the last bid of the target. See the figure 10.2.



Percentage increase in bid



Percentage decrease in bid

11. VISIBILITY BASED BIDDING

11.1. AIM

Keywords, that are giving the most portion of the overall sales are usually tracked for visibility. These keywords are updated using the visibility rules. The rules for visibility are pretty simple, using only the current rank and the desired rank.

11.2. ASSUMPTIONS

1. The placement of a target depend on the bid. Higher the bid, the higher the placement.
2. Increasing bids results in higher placement and decreasing bids results in lower placement.

11.3. VISIBILITY RULES

The rules are simple. We just compare the desired rank with the current rank and increase or decrease the bid based on the difference between the two. We use the average CPC of the whole category for this. The following figure shows the rules.

11.3.1. get_visible_keywords Function

```
def get_visible_keywords(client_profile_id):
    # ... (code omitted for brevity)
    return visible_keywords
```

This function retrieves a list of keywords that are visible and creates a list of ASINs (Amazon Standard Identification Numbers) that are visible with the keyword. The function takes as input a string ‘client_profile_id‘ which is the client profile id of the client.

The function first constructs a SQL query to retrieve the distinct keywords, ASINs, campaign IDs, and campaign types from the ‘visibility_clean‘ table where the ‘client_profile_id‘ matches the input and the date is greater than the current date minus 1.

The function then runs the query and retrieves the data into a DataFrame ‘visibility_df‘. It then groups the data by ‘Targeting‘ and ‘campaignType_v‘ and aggregates the ‘asin‘ and ‘campaignId_v‘ columns using a custom aggregation function ‘agg_function‘.

Finally, the function renames the columns of the DataFrame and returns it.

11.3.2. get_asin_visible_precentage Function

```
def get_asin_visible_precentage(target_asins, visible_asins):
    # ... (code omitted for brevity)
    return len(intersection) / len(target_asins)
```

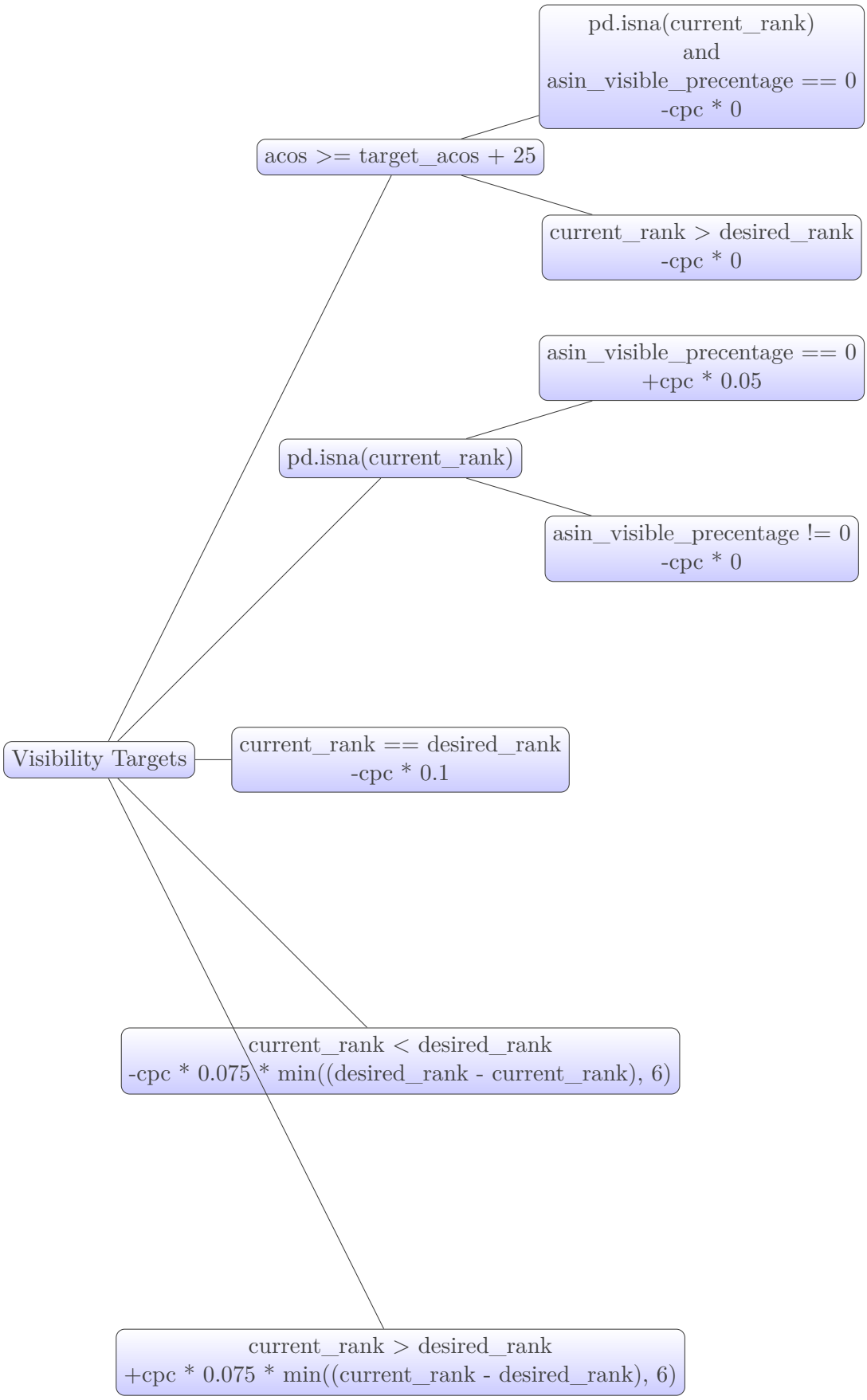
This function calculates the percentage of ASINs that are visible for a keyword. The function takes as input two strings ‘target_asins‘ and ‘visible_asins‘ which are the ASINs that are targeted and the ASINs that are visible, respectively.

The function first checks if ‘target_asins‘ or ‘visible_asins‘ is null. If either is null, it returns 0. If neither is null, it splits ‘target_asins‘ and ‘visible_asins‘ into lists.

The function then finds the intersection of the ‘target_asins‘ and ‘visible_asins‘ lists and calculates the length of the intersection divided by the length of ‘target_asins‘. This is the percentage of ASINs that are visible. The function then returns this percentage.

11.3.3. Flow Chart for Visibility Rules

The 11.1 shows the flow chart for the visibility rules.



Visibility Bids

12. OTHER RULES

12.1. OVERVIEW

This chapter discusses some rules for bid changes that are not the rule based on rule-based bidding or visibility-based bidding. These rules are applied after the rule-based bidding and visibility-based bidding. The main part of this is the mismatch that we do, the capping of minimum and maximum bids and applying any filters that are required.

12.2. OPTIMUM BID RANGE

The optimum bid is the bid that is calculated by the algorithm. While calculating the bid range, we make sure that the bids are not very high compared to the optimum bid.

12.2.1. _comply_bid_range_one Function

```
1 def _comply_bid_range_one(row):
2     # ... (code omitted for brevity)
3     return round(bid, 2)
```

This function complies the bid range for a single row of data. The function takes as input a Series `row` which contains the data for a single row.

The function first checks if the `action` value in `row` is null. If it is, the function sets `bid` to the `sys_bid` value in `row` and returns it.

The function then checks if the `clicks` or `impressions` value in `row` is 0. If either is 0, the function sets `bid` to the `sys_bid` value in `row` and returns it.

The function then checks if the `action` value in `row` is "visibility targets". If it is, the function sets `bid` to the minimum of the `sys_bid` value in `row` and 1.4 times the `mean_bid` value in `row`. If `action` is not "visibility targets", the function sets `bid` to the minimum of the `sys_bid` value in `row` and 1.15 times the `mean_bid` value in `row`.

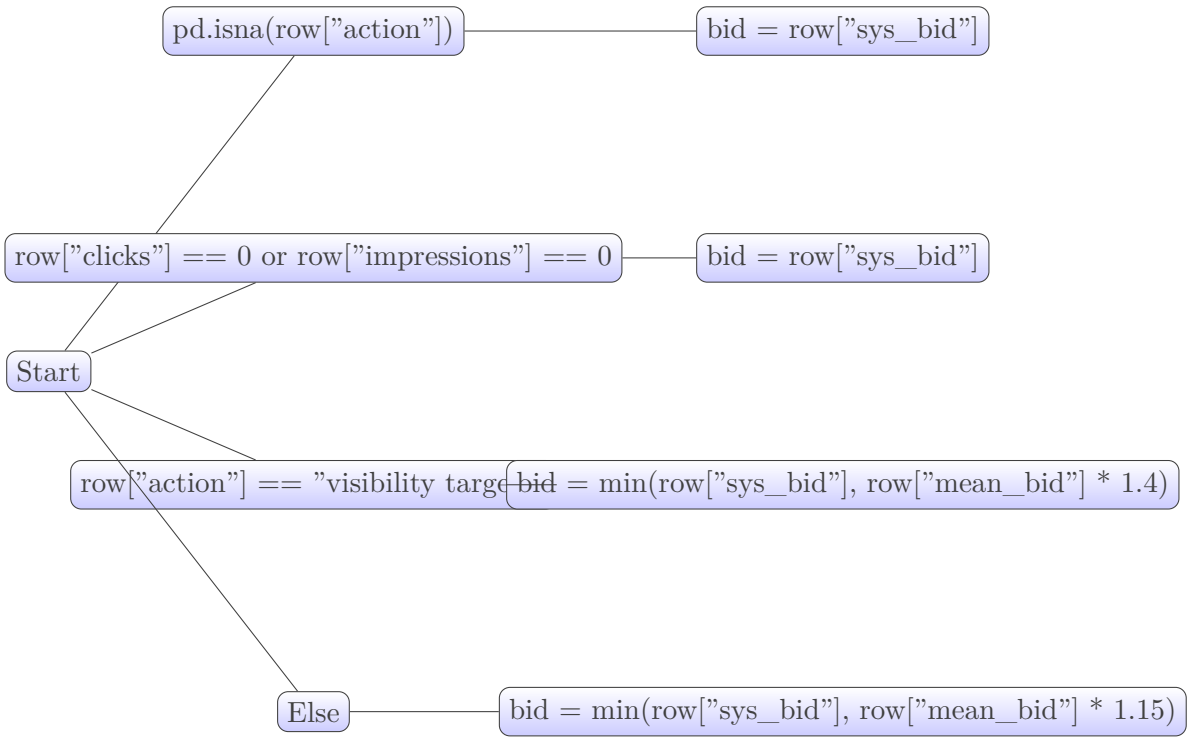
Finally, the function returns `bid` rounded to 2 decimal places.

The figure 12.1 shows a tree diagram of the logic of the function:

12.2.2. set_bid_range Function

```
1 def set_bid_range(data_with_sys_bid, client_profile_id):
2     # ... (code omitted for brevity)
3     return data_with_sys_bid
```

This function sets the bid so that the bid complies with the bid range. The function takes as input a DataFrame `data_with_sys_bid` which contains the data with system bids generated using the `calculate_sys_bid` function, and a string `client_profile_id` which is the client profile id.



Logic For Complying Bid

The function first retrieves a list of unique `Targeting_ID` values from `data_with_sys_bid`. It then constructs a SQL query to retrieve the `Targeting_ID`, `client_profile_id`, `mean_bid`, `min_bid`, `max_bid`, `type` as `bid_range_type`, and `Date` from the `max_bid_range` table where the `client_profile_id` matches the input and the `Date` is the maximum date for the `client_profile_id` and the `Targeting_ID` is in the list of unique `Targeting_ID` values.

The function then runs the query and retrieves the data into a DataFrame `bid_range_df`. It logs the shape of `bid_range_df` and the date for `bid_range_df`, and then drops the `Date` column from `bid_range_df`.

The function then merges `data_with_sys_bid` and `bid_range_df` on `Targeting_ID` and `client_profile_id`, and applies the `_comply_bid_range_one` function to the `sys_bid` column of `data_with_sys_bid`.

12.3. MISMATCH

Mismatch in bids is done because we do not want the same bids and/or we want to give higher bids in a group of targets based on some parameters. The following functions are used to do the mismatch.

12.3.1. mismatch_bids Function

This is the main function to be called for mismatch.

```
1 def mismatch_bids(
2     buckets_with_bid_range,
3     maximum_drop=0.4,
```

```
    \=["Targeting", "Match_Type", "campaignType"],
    order_by=["desired_rank"],
    ascending=[True],
):
    # ... (code omitted for brevity)
    return final_df
```

This function creates a mismatch of bids based on the `order_by` columns when grouped on `mismatch_by` columns. The bids are distributed so that the maximum bid is the max bid for the group and the minimum bid is the `max bid * (1-maximum_drop)`.

The function takes as input a DataFrame `buckets_with_bid_range` which contains the data with buckets and bids after `set_bid_range`, a float `maximum_drop` which is the maximum drop in bids, a list `mismatch_by` which are the columns to mismatch by, a list `order_by` which are the columns to order by, and a list `ascending` which determines whether the sorting is ascending.

The function first logs the mismatch and order by columns. It then groups `buckets_with_bid_range` by the `mismatch_by` columns and creates a list of DataFrames `groups`.

The function then iterates over each group in `groups`, sorts the group by the `order_by` columns, calculates the maximum bid in the group, calculates the minimum bid as the maximum bid times `1 - maximum_drop`, creates a list of bids from the maximum bid to the minimum bid with the number of steps equal to the number of rows in the group, and updates the `sys_bid` column of the group with the bids in the list. It then appends the group to the list `mismatched_groups`.

The function then concatenates `mismatched_groups` into a DataFrame `final_df`. It checks if the number of rows in `final_df` is not equal to the number of rows in `buckets_with_bid_range`. If they are not equal, it raises a `MergeNotCorrect` error.

12.3.2. mismatch_for_pt_visibilities Function

```
def mismatch_for_pt_visibilities(data, client_profile_id, maximum_drop=0.4):
    # ... (code omitted for brevity)
    return final_df
```

This function determines which visibilities are due to PT and mismatches them. The function takes as input a DataFrame `data` which contains the data with bids, a string `client_profile_id` which is the client profile id, and a float `maximum_drop` which is the maximum drop in bids.

The function first constructs a SQL query to retrieve the `Keyword` and count of distinct `targeting_type` from the `visibility_clean` and `campaign_master` tables where the `client_profile_id` matches the input and the date is greater than the current date minus 1.

The function then runs the query and retrieves the data into a DataFrame `vis`. It then retrieves a list of `Targeting` values where the count of distinct `targeting_type` is greater than 1.

The function then retrieves the data in `data` where the `action` is ”visibility” and the `Targeting` is in the list of `Targeting` values, and logs the number of rows in this data. It then retrieves the data in `data` where the `Targeting_ID` is not in the list of `Targeting_ID`

values in this data.

The function then checks if there is any data to mismatch. If there is, it calls the `mismatch_bids` function to mismatch the bids, and concatenates the mismatched data with the other data into a DataFrame `final_df`. If there is no data to mismatch, it sets `final_df` to `data`.

The function then checks if the number of rows in `final_df` is not equal to the number of rows in `data`. If they are not equal, it raises a `MergeNotCorrect` error.

12.3.3. mismatch_for_spotlight Function

```
def mismatch_for_spotlight(data, maximum_drop=0.4):
    # ... (code omitted for brevity)
    return final_df
```

This function mismatches bids for spotlight campaigns. The function takes as input a DataFrame `data` which contains the data with bids, and a float `maximum_drop` which is the maximum drop in bids.

The function first calls the `desired_rank_for_spotlight` function to retrieve the spotlight and non-spotlight data. If there is no spotlight data, it returns `data`.

The function then calls the `mismatch_bids` function to mismatch the bids for the spotlight data, and concatenates the mismatched data with the non-spotlight data into a DataFrame `final_df`.

The function then checks if the number of rows in `final_df` is not equal to the number of rows in `data`. If they are not equal, it raises a `MergeNotCorrect` error.

12.4. DROPPING BIDS

Bids are dropped for a few reasons. These are:

12.4.1. Dropping Bids Due to High PT Visibility

We need to reduce the bids of pt targets so that we can control the visibility from kt. This is done by dropping the bids of pt targets.

drop_pt_bid Function

```
def drop_pt_bid(data, client_profile_id):
    # ... (code omitted for brevity)
    return data
```

This function drops bids on pt targets that have high `tos_is`. The function takes as input a DataFrame `data` which contains the data with bids, and a string `client_profile_id` which is the client profile id.

The function first constructs a SQL query to retrieve the `Targeting_ID` and `tos_is` from the `table_targeting` table where the date is the current date minus 1, the `client_profile_id` matches the input, the `campState` is "enabled", the `adGroup state` is in ["enabled", "na"], and the `AM-State` is in ["enabled", "paused"].

The function then runs the query and retrieves the data into a DataFrame `tos_is_df`. It then drops duplicate rows from `tos_is_df`, and merges `tos_is_df` with `data` on `Targeting_ID`.

The function then reduces the `sys_bid` of pt targets that have `tos_is` less than 50 and greater than or equal to 10 by 15%, and reduces the `sys_bid` of pt targets that have `tos_is` greater than or equal to 50 by 25%.

The function then calls the `_add_reason` function to add a bid change reason to `data`, and returns `data`.

12.4.2. Dropping Bids Because of High CPC

Keywords with high CPC have their bids reduced so that the CPC and ACOS is reduced.

drop_high_cpc_bid Function

```
def drop_high_cpc_bid(data, performing_buckets):
    # ... (code omitted for brevity)
    return data
```

This function drops bids on targets that have high cpc. The function takes as input a DataFrame `data` which contains the data with bids, and a list `performing_buckets` which are the performing buckets.

The function first creates a mask where the `cpc` is greater than 1.5 times the `desired_cpc`, the `acos` is greater than 1.2 times the `target_acos`, the `sys_bid` is greater than 0.8 times the `last_bid`, and the `bucket` is not in `performing_buckets`.

The function then logs the number of rows to drop bids due to high CPC, and reduces the `sys_bid` of the rows where the mask is True by 20

The function then calls the `_add_reason` function to add a bid change reason to `data`, and returns `data`.

13. THE CONFIGURATION FILE

13.1. AIM

There are a number of configuration that can change based on client. One such example is the timing of the iterations. Another will be the strategy under which the algorithm is run. Since it is not possible to change all this for each client from the function itself, I created a config file, in `yaml` that can be used to change the configuration for each client. This is the aim of this chapter.

13.2. THE CONFIG FILE

There are mainly four sections of the config file:

13.2.1. Client Name and Client Profile ID

This is the name of the client and the profile ID of the client. This is used to get the data from the database as well as to identify the clients in any part of the code. This part of the config file also details about which client is automated and if it is automated, whether it is odd or an even client. The section is used to create the list of clients that are eligible for automation. Here is this part of the config file as per the time of writing this document:

```yaml
all_clients:
  "bsc": "1703833694712446"
  "bombae": "1067792793219800"
  "chik": "3720135559731580"
  "curatio": "886919425817994"
  "indica": "28004434171406"
  "kiro": "209014597912770"
  "minelab_india": "3864805791405473"
  "nykaa": "1468124741501045"
  "petterati": "1040943915031731"
  "salon": "4158041936950591"
  "so_good": "4422554122982285"
  "spinz": "3659898280444209"
  "streax": "3529562790090541"
  "taali": "3145569791150530"
  "vip": "944318880609262"
  "uppercase": "169276395281531"

automated_clients:
  - curatio
  - kiro
  - minelab_india
  - nykaa
  - taali
  - so_good
  - uppercase
```

```
27
28 odd_clients:
29   - nykaa
30   - so_good
31   - taali
32   - vip
33   - streax
34   - minelab_india
35   - uppercase
36
37 even_clients:
38   - kiro
39   - curatio
40   - indica
41   - petterati
42   - salon
43   - chik
44   - spinz
```

13.2.2. Timing and the Strategy

This part of the config file is used to set the timing of the iterations. It also sets the strategy under which the algorithm is run. The strategy is set based on the client. This strategy is used to classify whether the bucket will be considered performing or non-performing. Here is this part of the config file as per the time of writing this document:

```
1     timing:
2   morning:
3     odd: 9
4     even: 10
5   hourly:
6     odd:
7       - 12
8       - 15
9       - 18
10    even:
11      - 13
12      - 16
13      - 19
14
15 strategy:
16   boost: []
17   balanced:
18     # - nykaa
19     - curatio
20   roi:
21     - indica
22     - kiro
23     - minelab_india
24     - nykaa
25     - petterati
26     - salon
```

```
    - vip
    - streax
    - chik
    - spinz
    - taali
    - uppercase
    - so_good
```

13.2.3. Defining the Performing Buckets

Based on client strategy, we use a set of bucket as performing buckets and the remaining buckets as non-performing buckets. This is used to decide which targets to pause first if the budget is running out. Here is this part of the config file as per the time of writing this document:

```
# Performing buckets are those that will be given a larger
   portion of budget. The buckets will depend on the client
   strategy.
performing_buckets:
    roi:
    - star_1
    - star_2
    - star_3
    - star_4
    - crucial_1
    - crucial_2
    - crucial_3
    - crucial_4
    balanced:
    - star_1
    - star_2
    - star_3
    - star_4
    - crucial_1
    - crucial_2
    - crucial_3
    - crucial_4
    - growth_1
    boost:
    - star_1
    - star_2
    - star_3
    - star_4
    - crucial_1
    - crucial_2
    - crucial_3
    - crucial_4
    - growth_1
    - growth_2
    - inflated_1
    - inflated_2
```

13.2.4. Other Configurations

This section has some other configurations, notably, the maximum bid to drop while mismatching the bids (see the chapter 12 for more details). Also, this section notes the filter function to be used. Here is this part of the config file as per the time of writing this document:

```
# default maximum drop to use
default_maximum_drop: 0.4

# Change the maximum drop for specific clients if required
# Do not need to change if the default is to be used
maximum_drop: {}

# filter functions for clients. Other than providing names of clients that have filter function,
# the said function must be implemented in the file filter_functions.py with name as `filter_for_client_<client_name>`.
clients_with_filter_function:
  - vip
```

13.3. THE CONFIG MODULE

The above config file goes hands in hand with the `config.py` module. This module is used to read the config file and create a dictionary that can be used to access the configuration. The module also provides a number of functions and classes that can be used to access the configuration. Some functions and classes discussed here:

13.3.1. BaseConfig Class

The `BaseConfig` class is a base class for the configuration classes. It is responsible for parsing the configuration file and creating the automated clients.

__init__ Method

```
def __init__(self, logger=None):
    if logger is None:
        self.logger = get_simple_logger("config")
    else:
        self.logger = logger

    self.config_data = parse_config(self.config_file_path)
    self.all_clients = self.config_data["all_clients"]
    self.automated_clients = self._create_automated_clients()
    self.all_clients_reverse = self._reverse_dict(self.all_clients)
    self.automated_clients_reverse = self._reverse_dict(self.automated_clients)
```

The `__init__` method initializes the class, parses the configuration file, and creates the automated clients. If a logger is not provided, it creates a simple logger. It then parses

the configuration file and stores the data in `self.config_data`. It retrieves the list of all clients and the list of automated clients from `self.config_data`, and creates reverse dictionaries for both.

_create_automated_clients Method

```
def _create_automated_clients(self):
    automated_client_names = self.config_data["automated_clients"]
    automated_clients = {}
    for client_name in automated_client_names:
        if client_name in self.all_clients:
            client_profile_id = self.all_clients[client_name]
            automated_clients[client_profile_id] = client_name
        else:
            self.logger.error(
                f"Client {client_name} not found in all clients. See the config file."
            )
            raise ClientNotFound(
                f"Client {client_name} not found in all clients. See the config file."
            )
    return automated_clients
```

The `_create_automated_clients` method retrieves the list of automated client names from `self.config_data` and creates a dictionary of automated clients. If a client name is not found in the list of all clients, it logs an error and raises a `ClientNotFound` error.

_reverse_dict Method

```
def _reverse_dict(self, dict):
    return {v: k for k, v in dict.items()}
```

The `_reverse_dict` method reverses a dictionary so that the keys become values and vice versa. This is used to create reverse dictionaries for all clients and automated clients.

13.3.2. ClientConfig Class

The `ClientConfig` class is a subclass of the `BaseConfig` class. It is used to parse the configuration for a particular client.

__init__ Method

```
def __init__(self, client_profile_id, logger=None):
    super().__init__(logger=logger)
    self.client_profile_id = client_profile_id
    self._parse_name()
```

The `__init__` method initializes the class and parses the client name from the client profile id. It calls the `super()` function to inherit the methods and properties from the parent class.

_parse_name Method

```
def _parse_name(self):
    if self.client_profile_id in self.all_clients_reverse:
        self.client_name = self.all_clients_reverse[self.client_profile_id]
    else:
        raise ClientNotFound(...)
```

The `_parse_name` method parses the client name from the client profile id. If the client profile id is not found in the config file, it raises a `ClientNotFound` error.

_parse_client_timing_type Method

```
def _parse_client_timing_type(self):
    odd_clients = self.config_data["odd_clients"]
    even_clients = self.config_data["even_clients"]
    if self.client_name in odd_clients:
        self.client_timing_type = "odd"
    elif self.client_name in even_clients:
        self.client_timing_type = "even"
    else:
        raise ConfigNotFound(...)
```

The `_parse_client_timing_type` method parses the client timing type from the client name. If the client timing type is not found in the config file, it raises a `ConfigNotFound` error.

create_client_configurations Method

```
def create_client_configurations(self, raise_error_if_not_automated=True):
    self._parse_client_timing_type()
    self._parse_client_timing()
    self._parse_strategy()
    buckets = self._parse_performing_buckets(self.client_strategy)
    maximum_drop = self._parse_maximum_drop()
    is_automated = self._is_automated()
    filter_function = self._parse_filter_function()
    if not is_automated and raise_error_if_not_automated:
        raise ClientNotOnAutomation(...)
    self.client_configurations = {...}
    return self.client_configurations
```

The `create_client_configurations` method creates the client configurations. It parses the client timing type, client timing, and strategy, and checks if the client is automated. If the client is not automated and `raise_error_if_not_automated` is True, it raises a `ClientNotOnAutomation` error. It then creates a dictionary of client configurations and returns it.

13.3.3. RunnerConfig Class

The `RunnerConfig` class is a subclass of the `BaseConfig` class. It is used to run the iterations on Airflow.

__init__ Method

```
def __init__(self, logger=None) -> None:
    super().__init__(logger=logger)
    self.all_clients = self.config_data["all_clients"]
    self.odd_clients = self.config_data["odd_clients"]
    self.even_clients = self.config_data["even_clients"]
    self.automated_clients = self.config_data["automated_clients"]
```

The `__init__` method initializes the class and parses the config file. It calls the `super()` function to inherit the methods and properties from the parent class. It then retrieves the list of all clients, odd clients, even clients, and automated clients from `self.config_data`.

get_client_details Method

```
def get_client_details(self, client_names, automated_only=True, raise_error=
    True):
    # ... (code omitted for brevity)
    return details
```

The `get_client_details` method gets the client name and client profile id for the given client names. If a client name is not found in the list of clients and `raise_error` is True, it raises a `ClientNotFound` error.

create_odd_and_even_clients Method

```
def create_odd_and_even_clients(self, only_automated=True):
    # ... (code omitted for brevity)
    return odd_clients, even_clients, self.all_clients
```

The `create_odd_and_even_clients` method creates the odd and even clients. If `only_automated` is True, it only includes the automated clients.

get_clients Method

```
def get_clients(
    self,
    type_,
    to_remove=[],
    to_add=[],
    only_automated=True,
    id_first=False,
):
    # ... (code omitted for brevity)
    return clients
```

The `get_clients` method gets the clients for the given type. It can remove clients from the list and add clients to the list. If `id_first` is True, it returns the dictionary with client profile id as keys and client name as values.

get_all_clients Method

```
def get_all_clients(
    self,
    automated_only=True,
    to_remove=[],
    id_first=False,
):
    # ... (code omitted for brevity)
    return clients
```

The `get_all_clients` method gets all the clients. If `automated_only` is True, it only includes the automated clients. If `id_first` is True, it returns the dictionary with client profile id as keys and client name as values.

get_timing Method

```
def get_timing(self, type_):
    # ... (code omitted for brevity)
    return morning_time_str, hourly_times_str
```

The `get_timing` method returns the tuple of morning and hourly timings for the given type.

THE CODEBASE

14. CODEBASE OVERVIEW

14.1. OVERVIEW

This part gives detail about the existing codebase. The part starts with the code for morning and the hourly changes and then moves on to discuss other parts of the codebase.

15. MORNING CHANGES

15.1. AIM

16. HOURLY CHANGES

16.1. AIM

17. PAUSE ITEMS

17.1. AIM

THE ONBOARDING

18. ONBOARDING OVERVIEW

18.1. OVERVIEW

APPENDICES

LIST OF FIGURES

LIST OF TABLES