The aim of this Click Fraud detection model is to perform click fraud Detection along with this, providing time and space complexity. It will support huge data handling. For that, we have prepared the model. Did Data pre-processing, Feature Transformation/ Feature Engineering over it through which huge data was converted to normal size data and the "out of memory" error was resolved. After that, we performed operations on different algorithms such as Random Forests, LightGBM Algorithm, XGBoost Algorithm, etc.

A diagram of a machine

Description automatically generated

Flowchart architectural diagram of the system

**Data Gathering:**

The data we are using is sourced through Kaggle. The name of the Dataset is Talking Dataset which contains a total of 7.1GB of data. Comprising 8 columns and 184,903,890 rows. It is a Chinese dataset that has details of clicks, whether the action was performed on the site or not. The Column "is-attributed" shows that. 0 means yes, the action was performed and 1 means no action was not performed. The detailed Data characteristics are mentioned below:

1. IP: IP address of the system by which the website is accessed.
2. App: App used from which the user is visiting the website
3. Device: The Device user is using (e.g., PC, Mobile, Tablet etc.)
4. OS: The Operating System of that device (e.g., Android, Windows etc.)
5. Channel: The publisher’s channel ID
6. Click-time: The timestamp when the click was performed.
7. Attributed-Time: The timestamp when the action was performed.
8. is-attributed: Whether the action was performed or not. 0 means yes, the step was performed, and this is an actual used and non-fraudulent click, 1 means the move was not performed and is a fraudulent click.

At first, when we tried importing the complete dataset, we were facing a Memory Error "Unable to allocate 8.27 GiB for an array with shape (6,184903)"

**Data Pre-processing:**

After Data gathering, we checked the details of the data. Then we wanted to make this data work. As this is a huge amount of data and initially, we were facing Memory problems when we were just importing the data. To make this data work on the model we have to perform some Pre-Processing and Feature Transformation/Feature engineering and keep only important data which is Dimensionality reduction, filling the missing data values, and data cleaning, etc. For this the first thing we did was just import IP addresses and Channels by which the action was performed and then sorted the data in Ascending order by IP addresses along with this we did the indexing.

**Feature Transformation / Feature Engineering:**

Here comes the Feature transformation part. Our goal is to divide the dataset into 4 chunks and a single IP address should be present in one chunk only. If the same IP is present in more than 1 chunk, then the complexity will be increased. If we simply divide the dataset into 4 parts, we will see that one IP address is present in 2 parts. To solve this issue. First, we find the Window size. The window size shows "how many times a single IP address is repeating in the complete dataset". Then we will check the last 2 IPs on the chunk. For this we will select "Window Size + 1", this will give us the past 2 IP addresses of each chunk. Then we will simply take the Last IP address and merge it at the start of the next Chunk. Then we will Create a new data frame and transfer The IP address there and the number of times it is appearing (number of different channels it is appearing on). By this, we are creating a new Column that contains the IP-Counts. By this approach we will get 4 chunks, each containing only unique IP addresses with their IP-Counts.

The next step is to Create a new dataframe and transfer all the other column values for each IP address for this we divided our complete dataframe containing 5GB of data into 10 parts. Because while transferring the column values we must search for the IP address in the whole 5 GB data which is a time-consuming job. That’s why divided the whole data into 10 new dataframe and that is saved into an array. By searching the column values of an IP we don’t have to investigate the whole 5GB data, we just must check the start and end IP values of those 10 dataframes and just go and find column values on our concerned dataframe. The last step now is to set the column "is-attributed" value. Because this is the most important column that identifies fraudulent and non-fraudulent clicks. For this we selected the rows containing "is-attributed = 1" and fetched the unique IP addresses from them and then on the final dataframe we updated the values of the "is-attributed" column. In this way, we created a final dataframe that contains unique IP addresses, IP-counts, and other column values. Now our dataframe does not have 1000 repeating rows and then our dataframe is now of very less in size.

The final shape of the Transformed Data has a size of 8.5MB and a total of 277,394 rows and 9 columns. Here we can see we have an additional column named ip-count and each IP address is present only 1 time.

**Pre-processing on the Transformed data:**

On the Transformed data, we performed some Pre-Processing that includes Checking for missing data and then filling the missing data value with the most common values. After that check the target value distribution i.e., the "is-attributed" column. Which distinguishes between fraudulent and non-fraudulent clicks. Here the 11% clicks are fraudulent clicks in this Transformed dataset.

**Feature Engineering on the Transformed data:**

At this point, we will do the leftover Feature Engineering, that is.

* Renaming the "click-time" column to a new column named "datetime".
* Day of the week from "datetime" column to a new column named "day-of-week".
* Day of Year from "datetime" column to a new column named "day-of-year".
* Month from "datetime" column to a new column named "month".
* Hour from "datetime" column to a new column named "hour".

The next step is to change the type of the following columns to int16 for catering to the Space Complexity issue.

* IP
* App
* Device
* OS
* Channel

Deleting the unused columns

* click-time.
* datetime
* attributed-time.

**Classification Model:**

1. **Train - Test Split**:

Training and Testing data was split into 75-25 portions. 75% of the data is used as Training data and 25% of the data is used as Testing Data.

1. **AdaBoost Algorithm:**

For selecting the right Algorithm for our model, we worked on the Top 3 Mostly used algorithms that we have discussed in detail in Literature Review part. The First one is AdaBoost Algorithm. Which provided us 79.88% "roc-auc-score". One drawback of this model was that it took a lot of time to execute. Following are the code and details of the parameters that we first used for the model.

* 1. Here we used DescisionTreeClassifier with the max-depth = 3 as the base estimator for AdaBoostClassifier
  2. n-estimator is set to 600.
  3. learning-rate is set to 1.5.
  4. algorithm = "SAMME"

To get better results we performed Hyper-parameter tuning on it. Following are the code and details of the parameters that we first used for the hyper-parameter tuning of AdaBoostClassifier.

1. The base-estimator max-depth was set to 2,3
2. n-estimators were set at 100,300,500.
3. The learning rate is set to 0.9.
4. Algorithm = "SAMME"
5. cv = 3
6. For the scoring we chose "roc-auc-score

Results from the Hyper-parameter tuning of the AdaBoost Algorithm were then transferred to a dataframe and then we sorted it by ’rank-test-score’. Following are the code and the results of Hyper-parameter tuning.

Used the best parameters that we obtained after the hyper-parameter tuning of the AdaBoost Algorithm and it gave us 79.92% "roc-auc-score" which is good but let’s check the score from the remaining two Algorithms. Here one thing that was a minus point is that it takes a lot of time to execute. Following are the code and details of the parameters that we first used for the model.

1. Here we used DescisionTreeClassifier with the max-depth = 3 as the base estimator for AdaBoostClassifier
2. n-estimator is set to 500.
3. learning-rate is set to 1.5.
4. algorithm = "SAMME
5. **LightGBM Algorithm**:

The Second Algorithm that we are working on is LightGBM Algorithm. This is also widely used in Click Fraud Detection models. This algorithm is famous for its lightning-fast speed and good accuracy. Following is the code for Assigning the Train and Test data for LightGBM algorithm object. After that, we are setting the parameters for training the LightGBM Algorithm. Following are the code and details of the parameters that we first used for the model.

* 1. metric for scoring is again set to ’auc’.
  2. objective is set to ’binary.’
  3. num-leaves are set to 105.
  4. feature-fraction is set to 0.8.
  5. The bagging-fraction is set at 0.8.
  6. bagging-freq is set to 1.
  7. learning-rate is set to 0.08.
  8. and verbose is set to 1.

After using those parameters, the result we got was 81.33% "roc-auc-score" which is very good and as we know that this model is very fast in speed which is a plus point of this algorithm. Among the two Algorithms that we used LightGBM is the best. But let’s check the last Algorithm that we have in line before coming to any conclusion.

1. **XGBoost Algorithm:**

The last Algorithm that we will be testing out is XGBoost Algorithm. This algorithm. This algorithm is popular for its best performance in terms of Time and Space complexity as well as on less tendency of overfitting and accuracy. So now let’s check how this algorithm performs when we work with our complex data. At first, we just sent the Train and test data without explicitly setting the parameters. To check the initial ’roc-auc-score’. Following is the initial implementation of the XGBoost Algorithm.

The "roc-auc-score" from the initial implementation of the XGBoost algorithm is 81.03%. Which is good but is less than the score we got from LightGBM Algorithm.

1. **Hyper-parameter Tuning**:

We are performing Hyper-parameter Tuning on XGBoost Algorithm to get the best parameters for our model. The following are the code and details of the parameters that we first used for the hyper-parameter tuning of the XGBoost Algorithm.

* 1. learning-rate is set to 0.1, 0.3, 0.5.
  2. subsamples are set to 0.3, 0.6, 0.8.
  3. n-estimators are set to 100, 200, 300, 500.
  4. max-depth is set to 2, 3, 4.
  5. cv = 3
  6. verbose = 0

Results from the Hyper-parameter tuning of the XGBoost Algorithm were then transferred to a dataframe and then we sorted it by ’rank-test-score’. Following are the code and the results of Hyper-parameter tuning.

1. **Final Model:**

Applying the Best performing parameters that we have obtained after Hyperparameter tuning. Following are the code and details of the parameters that we first used for the model.

* 1. The max-depth is set to 4.
  2. n-estimators set to 500.
  3. learning-rate is set to 0.1.
  4. Evaluation metric is "auc".
  5. sub-samples 1

The ROC score after Hyper-parameter tuning is 81.45% which is the best result that we have received so far among all three algorithms. The Algorithm we will be selected for our Final model is XGBoost Algorithm as we have achieved the best accuracy with an almost similar amount of time in execution. XGBoost model provided us with the best results in less amount of time.