FA23_ISOM_674_4103 AI & Machine Learning Final Project
Team 5: Prince Musonerwa, Foster Mosden, Jingyi Huang, Sunday Nwanyim

Exploratory Data Analysis:

We started by exploring the dataset to understand its structure and characteristics. Namely, we analyzed basic statistics, such as the overall number of clicks, the distribution of categorical variables, and any temporal patterns in the data. In addition, we visualized relationships between the variables through correlation matrices and bar plots to get an idea of how our variables would interact. It was important to gain an idea of any potential multicollinearity issues before modeling began.

Data Cleaning & Preparation:

To prepare the dataset for modeling, we first conducted feature engineering and feature selection. We retrieved the hour, the day of the week, and day of month from the hour column, aiding to capture temporal patterns. Extracting the hour, day of the week, and day of the month from the timestamp column proves beneficial for several reasons. These features enable the model to capture temporal patterns, enhancing its ability to recognize variations in ad click behavior during specific hours, weekdays, and days of the month. Including such relevant temporal information not only improves the model's generalization to new data, but also facilitates interpretability, providing insights into when ads are more likely to be clicked. Users' behavior exhibits distinct patterns based on time, and incorporating these temporal features contributes to a more refined and accurate predictive model for ad click prediction.

We then dropped 'id', 'hour', 'site_id', 'site_domain', 'app_id', 'app_domain', 'device_id', 'device_ip' to reduce dimensionality and focus the model on only the most important variables. We also dropped these columns because most of them were unique identifiers. These identifiers lack predictive power and may lead to overfitting by memorizing training data patterns that struggle to generalize to new data. Incorporating them increases dimensionality and computational complexity, potentially compromising confidence in the results of the model. The exclusion practice aligns with the goal of enhancing model generalization, mimizing overfitting, and upholding interpretability for accurate predictive outcomes.

We imputed the values which only exist in test data but not train data with the mode to avoid errors in model fitting. Imputing values in the test data while leaving the training data unchanged mimics real-world scenarios where models encounter new data with missing values during deployment. This approach ensures a realistic evaluation of the model's performance and guards against data leakage. By not imputing in the training data, the model is trained on the available information, fostering objectivity in predicting unseen data with missing values. The methodology reflects the model's ability to handle incomplete information, providing insights into its robustness in practical, deployment-oriented situations.

Next, we used label encoding to convert the categorical variables into numerical format. This step can be useful for keeping the consistency in data processing and

improving model performance. Label encoding proves advantageous in converting categorical variables to numerical format for machine learning. It preserves ordinal information when applicable, ensuring the model recognizes inherent hierarchies.

Finally, we used under sampling to handle the imbalance class problem of the training dataset. This greatly increased the working size of the dataset, and balanced the occurrences of the y-variable, which allowed us to both create and evaluate our models with greater confidence. Undersampling helps to improve generalization, mitigate model biases, and prevent overfitting.

Modeling:

XGBoost:

For this project, we conducted experiments with three different modeling algorithms.

Firstly, we utilized the XGBoost algorithm, optimizing the model through GridSearchCV. This involved tuning parameters such as n_estimators, max_depth, and learning_rate to minimize log loss. Leveraging the power of GPU accelerated computations with XGBoost enhanced the speed of our calculations. Due to computational constraints, we narrowed down the hyperparameter tuning to learning rate, max depth, and n_estimators (by providing range of values). GridSearchCV proved instrumental in determining the optimal combination of these parameters.

The second model in our experimentation was a Random Forest. Similar to the XGBoost approach, we employed GridSearchCV for hyperparameter tuning, focusing on n_estimators and max_depth.

Lastly, we explored the Decision Tree algorithm, following a similar methodology as described for the previous models. Throughout this process, we systematically tuned hyperparameters to enhance model performance.

Overall, our experimentation involved a strategic approach to hyperparameter tuning for each model, with a specific focus on optimizing performance metrics such as log loss.

Evaluation:

We employed Log Loss as a metric to assess the performance of our models. The XGBoost classifier achieved a log loss of 0.59295, outperforming the random forest classifier with a log loss of 0.59808, and the decision tree, which yielded a log loss of 0.6145. Consequently, we conclude that the XGBoost classifier excels in click prediction based on our evaluation.

Insights:

Throughout the course of this project, we were able to extract a great deal of insights from creating models on the dataset. First, we gained a greater understanding

of how best to handle categorical x-variables -- for our models, we employed label encoding to transform our categorical variables to numerical, which aided in the training process. We also performed feature engineering to extract the hour and the day of the week from the timestamp stored in the dataset. These features aided in the overall prediction of the distribution of clicks, which was important as the “peak time” of clicks was relevant to creating an effective model.

Challenges:

The dataset for this project posed a significant challenge, containing over 13 million rows for the test data and exceeding 30 million rows for the training data. Processing such vast amounts of data demanded substantial computing resources, surpassing the capabilities of our local machines. As a team, we made the strategic decision to invest in enhanced computing power.

Accordingly, we subscribed to Google Colab Pro, which provided us with shared access to substantial resources, including 50 GB of RAM, a 16 GB GPU, and a 225 GB disk. Despite this considerable computing power, we encountered challenges, particularly during hyperparameter tuning, where the sheer size of the dataset strained even these robust resources. In response, we opted to carefully limit the parameters used for tuning to optimize the use of our computing resources effectively. This approach allowed us to successfully navigate the computational demands of the project and complete it within the constraints of the available resources.

Suggestion for Future Work:

While we made what we believe to be the best approach in our modeling, we know that there are ways that we could improve the results. Some of the aspects of our process that we could change include:

i) **Compute Resources**: By using even more powerful computing resources, we could increase the granularity of our grid search, and thus hopefully improve our results. As it was, we were limited by the time it took to run each model, and thus it wasn’t realistic to explore as many parameters in tuning the model.

ii) **Hyperparameter Tuning**: To ensure a more comprehensive search for optimal configurations, we recommend expanding the range of hyperparameter values during tuning. This broader exploration could lead to a better-performing model.

iii) **Data Pre-processing**: A different approach to data pre-processing might have improved our predictions. Rather than use label encoding, we think hashing might have provided better results. This approach involves hashing the non-numeric variables to get a unique value for each of the strings. In addition to this, scaling the hashed values can be implemented to reduce the dimension.

iv) **Model Training**: It might have been beneficial to train our model on the different types of processed data and record the output.

Code Overview

`Data_Preprocessing.ipynb`

This notebook encapsulates the code for data cleaning, exploration, and feature engineering. It provides a step-by-step walkthrough of the preprocessing phase, showcasing the methods employed to transform raw data into a format suitable for modeling.

`XGBoost_Model.ipynb`

Focused on implementing the XGBoost model, this notebook details the optimization process using GridSearchCV. Hyperparameter tuning, GPU acceleration, and model evaluation procedures are thoroughly documented.

`Random_Forest_Model.ipynb`

This notebook mirrors the structure of the XGBoost implementation, presenting the application of a Random Forest model. It outlines the grid search for optimal hyperparameters and provides insights into the model's performance.

Appendix(code):

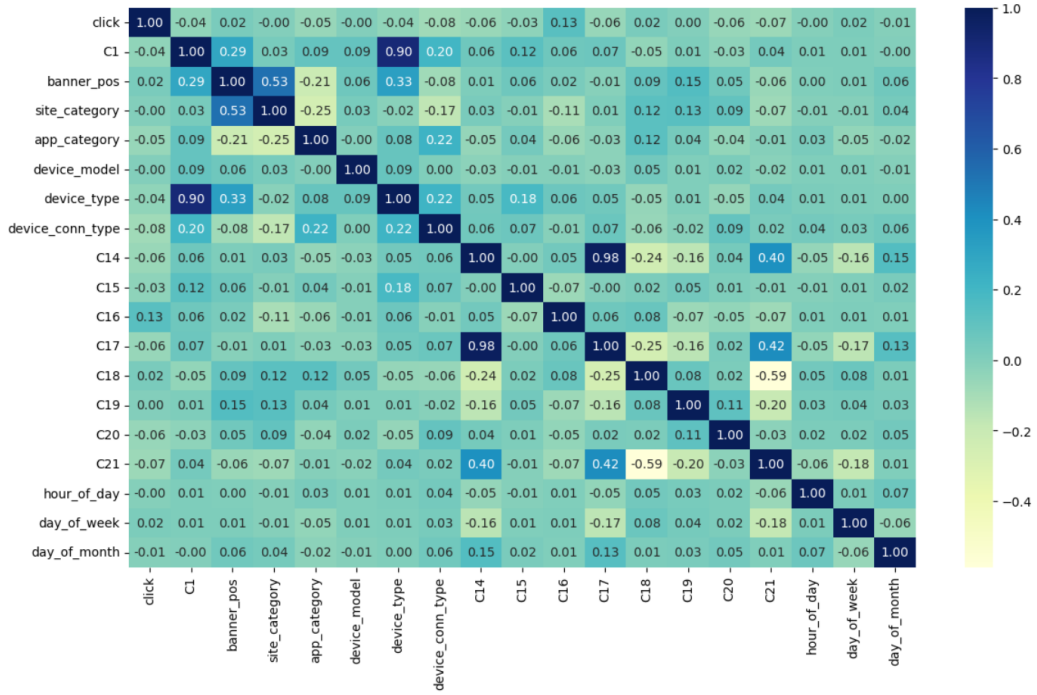
1. Exploratory Data Analysis
    1) Feature Engineering

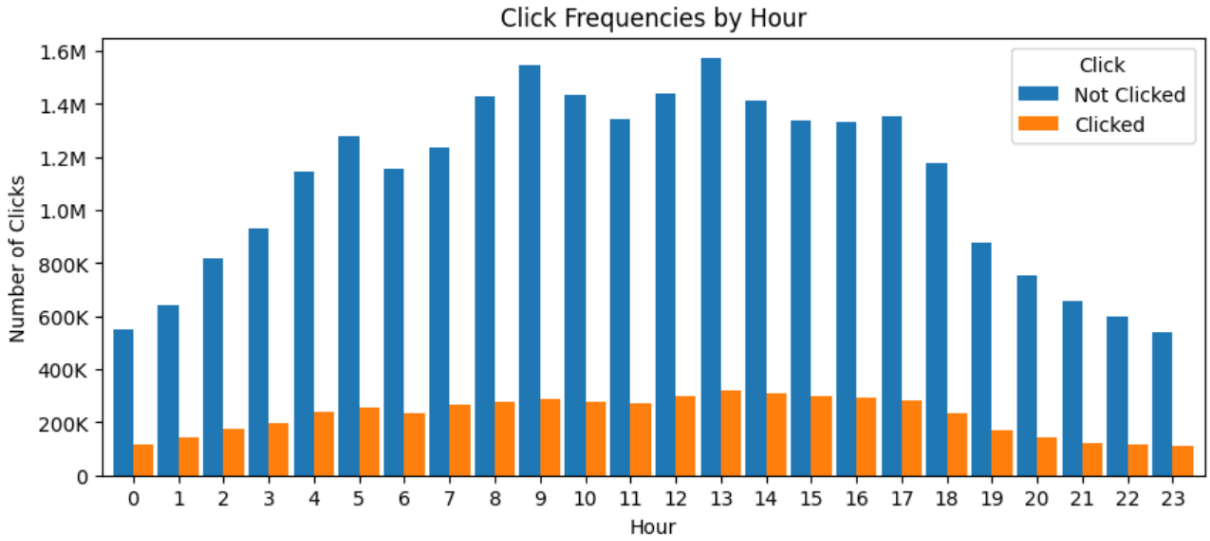
```
# Retrieve the hour, day of the week, and day of month
train_data["hour"] = pd.to_datetime(train_data['hour'], format='%y%m%d%H')
train_data['hour_of_day'] = train_data['hour'].dt.hour
train_data['day_of_week'] = train_data['hour'].dt.weekday + 1
train_data['day_of_month'] = train_data['hour'].dt.day
train_data.tail()
```

```
test_data["hour"] = pd.to_datetime(test_data['hour'], format='%y%m%d%H')
test_data['hour_of_day'] = test_data['hour'].dt.hour
test_data['day_of_week'] = test_data['hour'].dt.weekday + 1
test_data['day_of_month'] = test_data['hour'].dt.day
test_data.tail()
```

    2) Data Exploration

```
# Assuming 'train_data' is your DataFrame
corr = train_data.corr()
# Set the size of the figure
plt.figure(figsize=(14, 8))  # You can adjust the dimensions as needed
# Create the heatmap with annotations rounded to two decimals
sns.heatmap(corr, cmap='YlGnBu', annot=True, fmt=".2f")
# Rotate tick marks for visibility
plt.yticks(rotation=0)
plt.xticks(rotation=90)
# Show the plot
plt.show()
```





3) Feature Selection

```
# Define columns to drop - ensure these are exactly as they appear in your DataFrame
columns_to_drop = ['id', 'hour', 'site_id', 'site_domain', 'app_id',
                   'app_domain', 'device_id', 'device_ip']
# Drop the columns
train_data = train_data.drop(columns=columns_to_drop, axis=1)
```

```
# Drop the columns
test_data = test_data.drop(columns=columns_to_drop, axis=1 )
```

4) Data Cleaning

Finding values that exist in categorical values that exist in test data but not in the train data.

```
# Dictionary to hold the results
distinct_values_not_in_train = {}

for col in categorical:
    # Convert column values to sets
    train_values = set(train_data[col].unique())
    test_values = set(test_data[col].unique())

    # Find values in test not in train
    not_in_train = test_values - train_values

    # Store the result
    distinct_values_not_in_train[col] = not_in_train

# Display the results
for col, values in distinct_values_not_in_train.items():
    print(f"Values in test but not in train for '{col}': {values}")
```

```
Values in test but not in train for 'site_category': set()
Values in test but not in train for 'app_category': set()
Values in test but not in train for 'device_model': {'c7eae983', '4f4c3cdd', '969ee142', 'a32a9d22', 'eb7d45
```

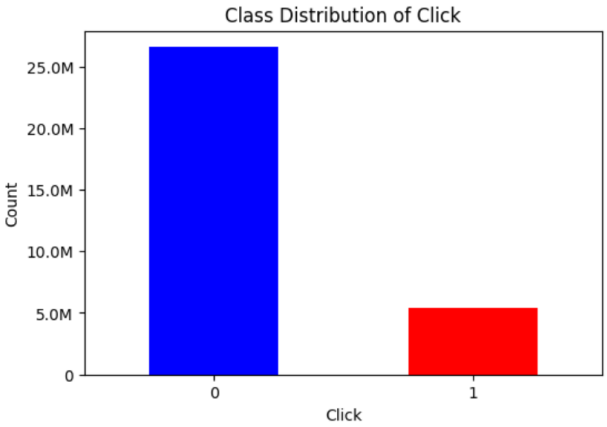
5) Data Imputing and Data Transformation using Label Encoding

```
for col in categorical:
    most_frequent_value = train_data[col].mode()[0]
    train_values = set(train_data[col].unique())
    test_values = set(test_data[col].unique())
    not_in_train = test_values - train_values
    test_data[col] = test_data[col].apply(lambda x: most_frequent_value if x in not_in_train else x)
```

```
le = LabelEncoder()
for column in categorical:
    # Fit and transform the training data
    train_data[column] = le.fit_transform(train_data[column])
    test_data[column] = le.transform(test_data[column])
```

6) Handling Class Imbalance

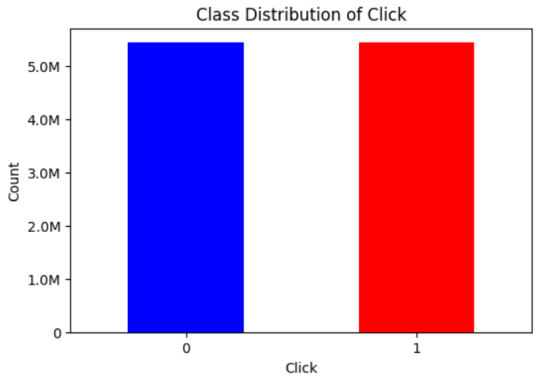
In our project, we employed under-sampling to address the significant class imbalance present in our dataset, where the majority class substantially outnumbered the minority class. To avoid the risk of biasing our predictive model towards the majority, we strategically reduced the size of the majority class through under-sampling.




```
# Handle class imbalance
from imblearn.under_sampling import RandomUnderSampler
# Initialize the RandomUnderSampler
rus = RandomUnderSampler(random_state=42)

# Resample the dataset
X_resampled, y_resampled = rus.fit_resample(X, y)

# Create a new DataFrame with the resampled data
train_data_resampled = pd.DataFrame(X_resampled, columns=X.columns)
train_data_resampled['click'] = y_resampled
```




```
train_data_resampled.to_csv('/content/drive/MyDrive/Final Project Dataset/ProjectTrainingDataBalanced.csv', index=False)
```

```
test_data.to_csv('/content/drive/MyDrive/Final Project Dataset/ProjectTestingEncoded.csv', index=False)
```

2. Modeling

1) Prepare the data for training

```
X = df_train.drop(['click'], axis=1)
y = df_train['click']
```

```
from sklearn.model_selection import train_test_split

# Split the data - 80% train, 20% test, for example
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.3, random_state=42)
```

2) Model Training–XGBoost

This part we used GridSearchCV to tune hyperparameters for the XGBoost classifier

```
from sklearn.model_selection import GridSearchCV
from xgboost import XGBClassifier
from sklearn.metrics import log_loss, accuracy_score

# Define your model with GPU support
model = XGBClassifier(
    use_label_encoder=False
    #tree_method='gpu_hist',     # Use GPU histogram algorithm
    #predictor='gpu_predictor'   # Use GPU for prediction
    #device = 'cuda'           # Use GPU for training
)
```

```
# Define the parameter grid
param_grid = {
    'n_estimators': [100,150],
    'max_depth': [12, 15, 18],
    'learning_rate': [0.01, 0.1, 0.2],
    # Add other parameters as needed
}

# Initialize GridSearchCV
grid_search = GridSearchCV(model, param_grid, cv=2, scoring='neg_log_loss', verbose=1, n_jobs=-1)

# Fit GridSearchCV
grid_search.fit(X_train, y_train)

# Best parameters and score
print("Best parameters found: ", grid_search.best_params_)
best_model = grid_search.best_estimator_

# Predictions and Evaluation
probabilities = best_model.predict_proba(X_test)
loss = log_loss(y_test, probabilities)
print(f'Log Loss: {loss}')
```

```
Fitting 2 folds for each of 18 candidates, totalling 36 fits
/usr/local/lib/python3.10/dist-packages/joblib/externals/loky/process_executor.py:752: UserWarning:
  warnings.warn(
Best parameters found:  {'learning_rate': 0.1, 'max_depth': 15, 'n_estimators': 150}
Log Loss: 0.5929530887288481
```

```
best_model
```

```
XGBClassifier(base_score=None, booster=None, callbacks=None,
              colsample_bylevel=None, colsample_bynode=None,
              colsample_bytree=None, device=None, early_stopping_rounds=None,
              enable_categorical=False, eval_metric=None, feature_types=None,
              gamma=None, grow_policy=None, importance_type=None,
              interaction_constraints=None, learning_rate=0.1, max_bin=None,
              max_cat_threshold=None, max_cat_to_onehot=None,
              max_delta_step=None, max_depth=15, max_leaves=None,
              min_child_weight=None, missing=nan, monotone_constraints=None,
              multi_strategy=None, n_estimators=150, n_jobs=None,
              num_parallel_tree=None, random_state=None, ...)
```

3) Model Training–RandomForest

```
from sklearn.model_selection import GridSearchCV
from sklearn.ensemble import RandomForestClassifier
model = RandomForestClassifier(random_state=42)
param_grid = {
   'n_estimators': [20, 22, 25],  # Number of trees in the forest
   'max_depth': [20, 25, 30],       # Maximum depth of the tree
   # Add other parameters as needed
}


grid_search = GridSearchCV(model, param_grid, cv=5, scoring='neg_log_loss', verbose=2, n_jobs=-1)

 # Load your training data here
grid_search.fit(X_train, y_train)

print("Best parameters:", grid_search.best_params_)
print("Best score:", grid_search.best_score_)
```

```
Fitting 5 folds for each of 9 candidates, totalling 45 fits
/usr/local/lib/python3.10/dist-packages/joblib/externals/loky/process_executor.py:752: UserWarning:
  warnings.warn(
Best parameters: {'max_depth': 20, 'n_estimators': 25}
Best score: -0.5986135209752769
```

```
from sklearn.metrics import log_loss, accuracy_score
best_model1 = grid_search.best_estimator_
# You can now use best_model for predictions or further analysis
# Predictions and Evaluation
probabilities2 = best_model1.predict_proba(X_test)
loss = log_loss(y_test, probabilities2)
print(f'Log Loss: {loss}')
```

```
Log Loss: 0.5980842064782914
```

4) Model Training–Decision Tree

```
# Define the parameter grid
param_grid = {
    'max_depth': [2, 5, 8, 12],
    'min_samples_split': [2, 5, 10],
    'min_samples_leaf': [1, 2, 5]
}
# Create a Decision Tree Classifier
dtree = DecisionTreeClassifier(random_state=42)
# Instantiate the Grid Search model
grid_search = GridSearchCV(dtree, param_grid, cv=5, verbose=2, n_jobs=-1)
# Fit the grid search to the data
grid_search.fit(X_train, y_train)
# Best parameters and best score
print("Best Parameters:", grid_search.best_params_)
print("Best Score:", grid_search.best_score_)

# Use the best model to make predictions
best_model = grid_search.best_estimator_
y_pred = best_model.predict(X_test)
y_pred_proba = best_model.predict_proba(X_test)
```

```
# Use the best model to make predictions
best_model = grid_search.best_estimator_
y_pred = best_model.predict(X_test)
y_pred_proba = best_model.predict_proba(X_test)

# Evaluate the model
print(classification_report(y_test, y_pred))
print("Accuracy:", accuracy_score(y_test, y_pred))

# Calculate and print log loss
log_loss_val = log_loss(y_test, y_pred_proba)
print(f'Log Loss: {log_loss_val:.4f}')
```

```
Fitting 5 folds for each of 36 candidates, totalling 180 fits
Best Parameters: {'max_depth': 12, 'min_samples_leaf': 5, 'min_samples_split': 2}
Best Score: 0.657212601993874
              precision    recall  f1-score   support

           0       0.68      0.60      0.64   1630536
           1       0.64      0.71      0.68   1630451

    accuracy                           0.66   3260987
   macro avg       0.66      0.66      0.66   3260987
weighted avg       0.66      0.66      0.66   3260987

Accuracy: 0.6570961491106834
Log Loss: 0.6145
```