Machine Learning Final Project

Team 6: Erik Ringen, Yaping Zhang, Mayur Kumar, Allegra Noto, Carl Xi

Ad Click Predictive Analytics

Data Understanding

The purpose of our project is to predict whether a customer will click on an ad or not. The dataset given to us contains 32 million entries, with 24 variables that describe everything from a person's device to the website the ad is found on. The target variable (click or no click) is binary.

In terms of understanding the target variable, 17% is comprised of clicks while 83% is comprised of no clicks. This is fairly imbalanced. However, to ensure the training, validation and test subsets contained similar proportions and to maintain simplicity, we used a random-sampling. We decided not to use stratified sampling, upsampling, or downsampling because despite the smaller percentage of clicks, the size of our data was sufficiently large to collect the information we needed.

When looking at the rest of the data, we can see that there are no NA entries within the entire dataset, which is lucky. The features are of class numeric and character. We converted all character features to factors.

After factor conversion, we checked the number of levels for each categorical variable. In Figure 1, we see the number of levels for each categorical variable in descending order. The variables Device_id and Device_ip each contain more than two million levels.

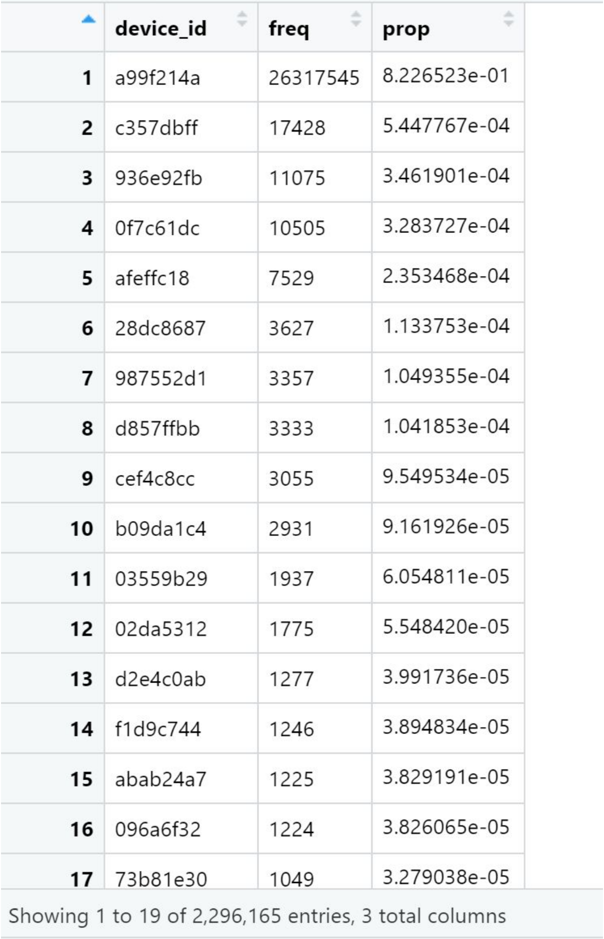
Figure 1: Variable Factor Levels

Variable	Number of Levels
device_ip	5762925
device_id	2296165
app_id	8088
device_model	8058
site_domain	7341
site_id	4581
C14	2465
app_domain	526
C17	407
hour	216
C20	171

Variable	Number of Levels
C19	66
C21	55
app_category	36
Site_category	26
C16	9
C15	8
C1	7
Banner_pos	7
Device_type	5
Device_conn_type	4
C18	4

It's important for us to also understand the distribution of the levels of these categorical variables. To do this, we created a table for each categorical variable to show the proportion of data for each level. This allowed us to understand the distribution of levels for each categorical variable. For example, in device_id, the level “a99f214a” accounts for approximately 82% of the data, with all the rest making up the final 18%. While in device_ip, there is one major level and the rest take up less than 0.005%, this is not always the case. In Figure 2, we see only one example of the 22 tables we created to show these distributions. This table specifically represents our device_id level distribution.

Figure 2: Factor Level Distribution (device_id)



	device_id	freq	prop
1	a99f214a	26317545	8.226523e-01
2	c357dbff	17428	5.447767e-04
3	936e92fb	11075	3.461901e-04
4	0f7c61dc	10505	3.283727e-04
5	afeffc18	7529	2.353468e-04
6	28dc8687	3627	1.133753e-04
7	987552d1	3357	1.049355e-04
8	d857ffbb	3333	1.041853e-04
9	cef4c8cc	3055	9.549534e-05
10	b09da1c4	2931	9.161926e-05
11	03559b29	1937	6.054811e-05
12	02da5312	1775	5.548420e-05
13	d2e4c0ab	1277	3.991736e-05
14	f1d9c744	1246	3.894834e-05
15	abab24a7	1225	3.829191e-05
16	096a6f32	1224	3.826065e-05
17	73b81e30	1049	3.279038e-05

Showing 1 to 19 of 2,296,165 entries, 3 total columns

Through our observations, we also found that for variables with a large number of levels (over 200), most of those levels have very few occurrences (less than 0.05% of the total). Since our models cannot handle more than 32 levels, we left only the 31 most populated levels, converting the remaining to “Other”.

Data Cleaning

As stated earlier, we converted all categorical variables, which were originally class character, to factors. Then, we jumped right into feature engineering.

First, we manipulated the hours variable. In its original timestamp format, it was not useful. We extracted from the timestamp hour of the day and day of the week. We believe those are the most relevant to find the likelihood of an ad click. This is because they can help distinguish when people are at work from when they are at home.

Besides extracting time features, we created a few totally new features based on interactions with ‘day’. This was based on the knowledge that individuals may be more likely to click an ad on particular days and that particular websites may get more clicks on a given day (for example, if they are having a sale). Therefore, we created interaction terms between day and app_id, site_id, device_ip, and device_id. This created 27 total prediction features.

Before beginning our modelling process, we then had to split the main training dataset into three subsets: training, validation, and test. We split this data through random sampling, as outlined earlier.

When it came to deciding encoding the data for logistic regression and neural nets, we looked at two methods: one hot encoding and likelihood encoding. When we used one hot encoding, it gave over 500 columns once all the categorical variables were converted to dummy variables, which would be very time consuming to fit models on. For this reason, we attempted it using only one million instances for the training dataset, with two million for validation and test, respectively. We ensured that the target variable ratio was representative of the population and that all levels of categorical variables were represented in each of the subsets.

Since most categorical features have a huge number of levels, we also attempted likelihood encoding. This approach involves creating new synthetic variables based on the in-sample click rate for each level. This approach has elements of Naive Bayes, but the assumption of conditional independence between features is instead relaxed. In addition, instead of calculating P(Feature | Y), we calculate P(Y | Feature). This is quite similar to autoregressive models and is popular in time series analysis. Here, past values are regressed onto the present to predict the future.

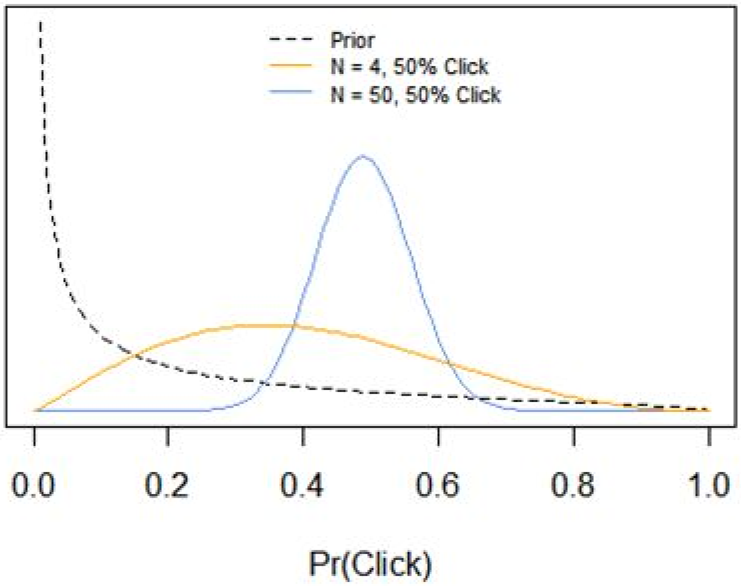
In our case, we found likelihood encoding performed over one hundred times faster than one hot encoding. For our logistic regression, a training set of four million instances was able to fit the model in 2 minutes. In Figure 3, you can see the data split sizes we tested.

Figure 3: Encoding Technique Data Splits

One Hot Encoding	Train	1 Million
	Validation	2 Million
	Test	2 Million
Likelihood Encoding	Train	8 Million
	Calibration	4 Million
	Test	4 Million

To limit overfitting of sparse feature labels (unique categories with a low number of observations), we also imposed a regularizing prior on the likelihood encoding. Assuming our outcome is well-described by the binomial distribution, the beta distribution is the conjugate prior – which means we can get a closed-form expression for the posterior. For likelihood encoding we use the training sample mean probability of a click as the prior mean, with a variance = 0.05, to regularize in the case of small N. In Figure 4, we see the regularizing effect of the prior.

Figure 4: Regularizing Effect of the Prior



Model Building

We decided that the best way to most accurately predict clicks would be to use an ensemble method. To do this, we tested out a number of different predictive models. We chose the best 2 models, and averaged together the predictions they generated. This gave us our final predictions. First, we had to check the accuracy of each individual function.

Naive Bayes

The first model we tried was the Naive Bayes model. This simplistic model simply uses as a probability the total rate of clicks in the dataset. Since 17% of our data had a 1 value for click, every instance has a 17% of a click. This naturally produced atrocious results, so we did not consider it for a potential final model.

Logistic Regression

Logistic regression is a predictive regression model that is used for binary target variables. It predicts the probability of the target taking on the value of 1. For this model, we used likelihood encoding, as stated earlier.

For likelihood encoding, we are first given a data frame of N observations of p features, and a response click. Next, we split the initial dataset into three subsets: training, calibration, and validation. After we decided upon likelihood encoding, we used a final split of 70% training, 15% calibration, and 15% validation from a starting sample of 15,000,000 observations.

For p in N features of the training dataset, we first aggregate by each unique value of p (levels of the categorical variable). Then, we record the number of observations for each level, and the number of clicks. Given some prior probability of click θ_prior, update the beta distribution using the training data for each level to get the posterior probability of a click, θ_post. These are values are represented by:

$$\boldsymbol{\theta\_prior \sim Beta(\alpha\_prior,\beta\_prior)}$$

$$\boldsymbol{\theta\_post \sim Beta(\alpha\_post,\beta\_post)}$$

$$\boldsymbol{\alpha\_post=N\_clicks + \alpha\_prior}$$

$$\boldsymbol{\beta\_post=N\_obs\text{-}N\_clicks + \beta\_prior}$$

$$\boldsymbol{E[\theta\_post]=\alpha\_post/(\alpha\_post+\beta\_post)}$$

Next, we transformed all values of θ_post to the log-odds scale, using the equation log(θ_post) - log(1 - θ_post), and subtracted the sample mean probability of a click. This centered the mean posterior probabilities at 0.

After we're done with our training set, we cycle through p in our N features of the calibration and validation sets, respectively. For each, we create new features called p_ratio by matching the feature labels in the calibration and validation datasets to the corresponding logit-transformed mean posterior probabilities in the training dataset. If the feature label in these datasets has no match in the training dataset, the new variables get a value of 0, which corresponds to the training sample mean probability of a click.

Using the calibration data, we then fit a logistic regression predicting clicks as a function of the newly encoded features, e.g., glm(click_cal ~ p_ratio, family="binomial", data=d_cal). This calibration step tells us how much the past performance of each feature helps predict out-of-sample.

With this fit model, we then predict clicks in the validation dataset using the corresponding features, e.g., predict(fit_glm, newdata=d_val). Lastly, we convert the predicted values back to the probability scale with the inverse logit function, and calculate the log loss.

Random Forest

Our next model was a random forest. Random forests are, in themselves, ensemble methods. They consist of an averaging of many individual decision trees, and help avoid the overfitting that can occur with a singular tree. The individual trees are differentiated through choosing random subsets of features at each split. This forces trees to go down different paths and reach different results. When they are all averaged together, this eliminates some of the noise that may result in choosing just one containing all attributes.

Due to this increased stability, we decided to avoid doing a singular decision tree and to just use the random forest. We believe that if a decision tree were to perform better than the random forest method, it would likely be due to chance and may not translate well to the test data. For that reason, we decided to stick strictly to two random forest models: one with bagging and one without.

Random forests are particularly useful in predicting click rates because they can handle binary features, categorical features, and numerical features at once while also being indifferent to nonlinear features. There is very little pre-processing that needs to be done, and the data does not need to be rescaled or transformed. Our data set is large with many features. However, random forests are great with high dimensional data since they use subsets of the data.

As we saw in our exploration, our data is unbalanced. Since click rate is low, we are in danger of having a biased prediction. Random forests, however, handle unbalanced data easily by balancing errors. To minimize overall error rate, when we have an unbalanced data set, the larger class will have a low error rate while smaller classes will likely have larger error rates.

Lastly, random forests have low biases and moderate variance predictions. As we know. Individual decision trees have a high variance and low bias. Since we average individual trees in random forests, we are averaging the variance as well. This way, we have both a low bias and a moderate variance model.

The procedure for data cleaning, data encoding and feature engineering is the same as for logistic regression. There are only two differences. Firstly, we used a much smaller dataset for training and validation because the more complicated procedure makes random forests have a much longer computation time too long. However, this is not a problem, as we don't need much data to train our model. The second difference is that we our dependent variable, "click", is transformed into factor. These differences allow random forests to properly deal with our data and give effective results.

In our modeling process, we tried random forests (without bagging) with different tree sizes, finally deciding on 500. In our model using bagging, we played with the parameter "mtry". This parameter determines the number of randomly selected attributes used at each split. We decided on 17, which means that we pick 17 features randomly each time when determining which attribute to split on.

Neural Networks

For predictions using neural networks, we built several models with varied widths and heights. Since running each of these models took very long, we first used small training sets and the one hot encoding method to figure out what combination of width and height was best for our data. After that we found our favorite model, we ran it on a larger training dataset using likelihood encoding. For our models, we chose to use a sigmoid as a neuron in our output layer. We did this as opposed to using softmax as a neuron as softmax was taking much more time, which was incredibly valuable in this process.

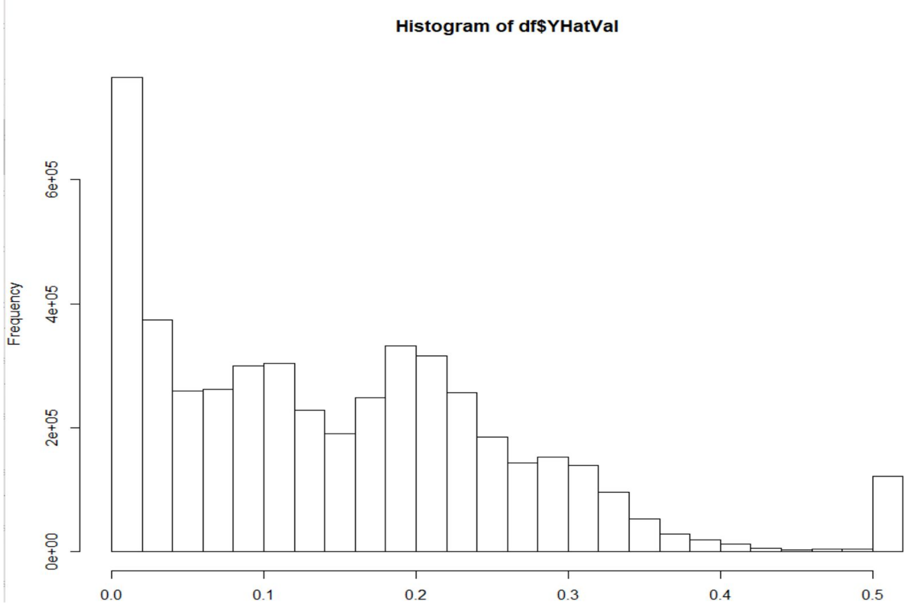
As displayed in Figure 5, we found that using 4 neurons in each of the 4 hidden layers gave us the best out-of-sample log loss. We then used that same setup with likelihood encoding and five million training instances to get our out-of-sample log loss of 0.38.

Figure 5: Neural Network Setups and Results

	Training Dataset Size	Neural Network Setup	Training Duration	In-Sample Log Loss	Out-of-Sample LogLoss
One Hot Encoding	500,000	4 Neurons in each of the 4 Hidden layers	~4hours	0.405	0.411
	500,000	10 Neurons in each of the 2 Hidden layers	~3hours	0.414	0.429
	500,000	10Neurons in each of the 5 Hidden Layers	~5hours	0.395	0.412
	1Million	4 Neurons in each of the 4 Hidden Layers	~6hours	0.403	0.417
Likelihood Encoding with interactive terms	5 Million	4 Neurons in each of the 4 Layers	~15hours	0.3834	0.3800

From our results, we see some models underfit while others overfit. Here, using likelihood encoding, we wound up with an exceptional log loss. However, when we explored these results further, the probability predictions on the validation dataset were in the range of 0 to 0.51. In Figure 6, we see a plot that shows the distribution of probability predictions on the validation set. Due to this, we chose not to use neural nets in our final ensemble method, as we are skeptical as to how it might perform on the test data with these odd ranges.

Figure 6: Distribution of Validation Probability Predictions



Conclusion & Further Analysis

In deciding which models to use for an ensemble prediction, we first rule out a neural network due to issues stated earlier. Then, if we look at the performance of our models in Figure 7, we can see that random forest and logistic regression produced the best performances at only 0.3881 and 0.3906 out-sample log loss for the two models respectively. The results are also very similar, with a Δ Log Loss of only 0.0025. Rather than simply select one model and in order to further enhance our performance, we opted for an ensemble approach to the test data, averaging over the predictions of the two models for our final test predictions.

Figure 7: Model Performance

Model	Out-of-Sample Log Loss
Naive Bayes	Excluded due to bad performance
Intercept-Only Model	0.4557
Logistic Regression	0.3906
Random Forest	0.3881
Neural Networks	0.3800

Appendix

analysis.R - Splits the random sample, creates the likelihood encoded variables, fits the models, and calculates the log loss.

Data_exploration.R - Contains our data exploration process

Neural_Net_Data_Prep.R - Prepares the data for a neural net fit

NN_Likelihood_encoding.ipynb - Uses likelihood encoding to prepare the data for a neural net model

NN_main.ipynb - Contains the main code for our neural net model

predict_ensemble.R - Creates predictions for the test data

sample_draws.R - Creates a random shuffled sample of 15,000,000 rows from ProjectTrainingData.csv

test_encode.R - Creates likelihood encodings for the test data

References

Vprokopev. “Mean (Likelihood) Encodings: a Comprehensive Study.” *Kaggle*, Kaggle, 7 Oct. 2018, https://www.kaggle.com/vprokopev/mean-likelihood-encodings-a-comprehensive-study.