DS Growth Case Study

Bijan Seyednasrollah

Nov 2020

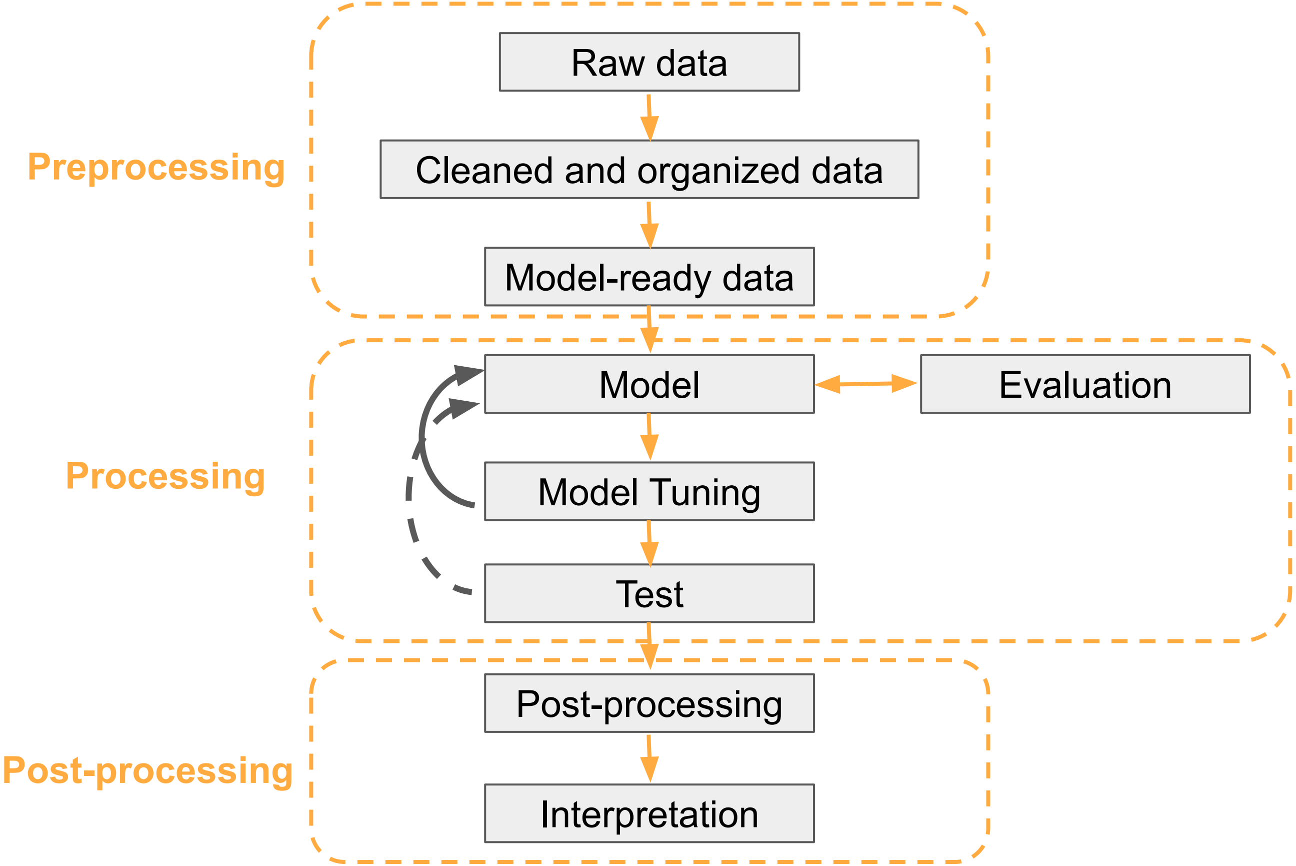
Table of content

Problem Statement

We want to predict the probability of “activation” based on the following data per click using data from Google Adwords.

1. **created_date** - date on which the click was recorded
2. **location_in_query** - city name that appeared in search query
3. **platform** - platform such as web/mobile
4. **campaign_state** - US state where the ad campaign was run
5. **in_city** - whether the ad campaign was run in a city or not
6. **is_prime** - whether the word fico prime appeared in the search query
7. **is_hardship** - whether the word hardship appeared in the search query
8. **category_debt_type** - type of debt solution that appeared in the search query
9. **is_activated**[TARGET] - whether the user talked to FFN call center or not

I developed the workflow in TensorFlow, summarized below.



Preprocessing

There are several challenges regarding the dataset that should be properly addressed before feeding it into the model:

1. Missing data
2. Imbalanced classes
3. Incorporating date and time
4. Multi data types and hyper-categorical data

We address each challenge by performing several steps as follows

Preprocessing- Missing Data (Imputation)

The table below shows number of missing values for each variable:

- `created_date            0`
- `location_in_query      15`
- `platform                0`
- `campaign_state          6`
- `in_city                 0`
- `is_prime                0`
- `is_hardship             0`
- `is_activated            0`
- `category_debt_type   9859`
- `time_index              0`

Several columns contain missing data points that should be imputed, including: location_in_query, campaign_state, category_debt_type.

For the most part, I used majority voting to impute missing values for the above columns. I also replaced invalid state name with 'fl' which was the majority class. The missing `location_in_query` were replaced by '`unknown_location`'

Preprocessing- Imputation (Code)

```
states = ['al', 'ak', 'az', 'ar', 'ca', 'co', 'ct', 'dc', 'de', 'fl', 'ga', 'hi',
          'id', 'il', 'in', 'ia', 'ks', 'ky', 'la', 'me', 'md', 'ma', 'mi', 'mn',
          'ms', 'mo', 'mt', 'ne', 'nv', 'nh', 'nj', 'nm', 'ny', 'nc', 'nd', 'oh',
          'ok', 'or', 'pa', 'ri', 'sc', 'sd', 'tn', 'tx', 'ut', 'vt', 'va', 'wa',
          'wv', 'wi', 'wy']

# also imputing states where the value is a number
clicks.loc[~clicks['campaign_state'].isin(states),'campaign_state'] = np.nan

#imputing campaign_state with the most frequent state
clicks['campaign_state'].fillna(clicks['campaign_state'].mode()[0], inplace = True)

#imputing location_in_query with the unknown_location
clicks['location_in_query'].fillna('unknown_location', inplace = True)

#imputing category_debt_type with the most frequent value
clicks['category_debt_type'].fillna(clicks['category_debt_type'].mode()[0], inplace = True)
```

Preprocessing- Imbalanced Classes

The label data is highly imbalanced:

- 87407 as is_activated=1
- 6787 a is_activcated=0

This could be a major problem if properly not resolved. There are several ways to tackle this issue: oversampling the minority class, under-sampling the majority class and assigning weights to each class in the loss function.

I took a hybrid approach: To avoid removing information, I picked the **oversampling** methods rather than the under-sampling method; but I also slightly adjusted the weights to improve the convergence of the model towards giving more importance to activated data (0.75 vs. 1). This could vary based on the goal of the model, that I will later discuss.

Preprocessing- Imbalanced Classes (Code)

```
from sklearn.utils import resample

data = clicks.copy()

#check the length of each class
print('oringal data class size:\n',data.is_activated.value_counts(), '\n')

#creating subsets
data_not_activated = data[np.array(data.is_activated) == 0]
data_activated = data[np.array(data.is_activated) == 1]

#upsampling of the minorty class
n_samples = len(data_activated)

#downsampling of the minorty class
#n_samples = len(data_not_activated)

data_not_activated_resampled = resample(data_not_activated, n_samples = n_samples, random_state = 0)
data_activated_resampled = resample(data_activated, n_samples = n_samples, random_state = 0)

#merged data back together
data_resampled = pd.concat([data_not_activated_resampled, data_activated_resampled])

#shuffle the data
data_resampled = data_resampled.sample(frac = 1)
```

Preprocessing- Incorporating Date/Time

There is date column (created_date) in the dataset that include information about the time of the application. However there are 32 unique values across the entire over 94,000 observations.

I used datetime data represented in two different formats:

1- As a categorical variable:

I will later explain how to handle categorical data

2- As a continuous number from 0 to 1 (named as ‘time_index’):

First converting to seconds, then scaling between 0 and 1.

```
scaler = MinMaxScaler()
clicks[['time_index']] = scaler.fit_transform(clicks[['time_index']])
```

Preprocessing- Multi-datatype

As the features include numerical, binary and categorical data we need to make the ready for our model.

To accomplish this, I used the feature_column module from tensorflow and converted all variables either to numerical vectors, multi-dimensional embedded matrices, and crossed hashed data, as follows:

- Vectors: for numeric variables: `'in_city', 'is_prime', 'is_hardship', 'time_index'`
- Categorical matrics: for multicategorical variables: `platform', 'category_debt_type'`
- Embedded matrices: for hyper-categorical variables: `'created_date', 'campaign_state'`
- Crossed columns: for combination of `'campaign_state' and ‘location_in_query’`

Preprocessing- Multi-datatype (Code)

```
from tensorflow import feature_column

feature_columns = []

#first numerical columns
for col_name in ['in_city', 'is_prime', 'is_hardship', 'time_index']:
  feature_columns.append(feature_column.numeric_column(col_name))

# indicator columns
indicator_column_names = ['platform', 'category_debt_type']
for col_name in indicator_column_names:
  categorical_column = feature_column.categorical_column_with_vocabulary_list(col_name, clicks[col_name].unique())
  indicator_column = feature_column.indicator_column(categorical_column)
  feature_columns.append(indicator_column)

# embedding cateogorical columns
embeding_columns = ['created_date', 'campaign_state']
for col_name in embeding_columns:
  vocab = clicks[col_name].unique()
  categorical_column = feature_column.categorical_column_with_vocabulary_list(col_name, vocab)
  column_embedding = feature_column.embedding_column(categorical_column, dimension= 10)#int(len(vocab) *.3))
  feature_columns.append(column_embedding)


# crossed columns
campaign_state = feature_column.categorical_column_with_vocabulary_list('campaign_state', clicks.campaign_state.unique())
location_in_query = feature_column.categorical_column_with_vocabulary_list('location_in_query', clicks.location_in_query.unique())
state_location = feature_column.crossed_column([campaign_state, location_in_query], hash_bucket_size=500)
feature_columns.append(feature_column.indicator_column(state_location))
```

Preprocessing - Train-Validation-Test Split

It is important to split the data into three parts:

1. Training data: used to train the model
2. Validation data: used to validate the model on the fly
3. Test data: used to evaluate the model after it is fitted.

```
# splitting the data

# the test data will be used only after the model is trained
train_valid, test = train_test_split(data_resampled, test_size = 0.2, shuffle = True)

# the valid data will be used during the training but not for the training, only for evaluations per epoch
train, valid = train_test_split(train_valid, test_size = 0.25, shuffle = True)

#printig their length
print('training data size:', len(train))
print('validation data size:', len(valid))
print('testing data size:', len(test))
```

```
training data size: 104888
validation data size: 34963
testing data size: 34963
```

Preprocessing- TensorFlow Ready Datasets

To prepare the input data for the TensorFlow data pipeline, we need to put to in sliced batches of tensors.

```
def create_dataset(data, batch_size=512):
  df = data.copy()
  labels = df.pop('is_activated')

  dataset = tf.data.Dataset.from_tensor_slices((dict(df), labels))

  # shuffle the dataset
  dataset = dataset.shuffle(buffer_size=len(df))

  dataset = dataset.batch(batch_size)
  return dataset
```

```
train_data = create_dataset(train)
val_data = create_dataset(valid)
test_data = create_dataset(test)
```

Processing - Model Structure

I used a neural network model with four main layers:

- An input layer of all features
- A dense layer of 1024 neurons with 50% dropout and ‘ReLU’ activation
- A dense layer of 128 neurons with 50% dropout and ‘ReLU’ activation
- A final single neuron layer of outputs with ‘sigmoid’ activation

```
model = tf.keras.Sequential()

model.add(layers.DenseFeatures(feature_columns))

model.add(layers.Dense(1024, activation='relu'))
model.add(layers.Dropout(0.5))

model.add(layers.Dense(128, activation='relu'))
model.add(layers.Dropout(0.5))

model.add(layers.Dense(1, activation='sigmoid'))
```

Processing - Model Evaluation

I used four main evaluation metrics to monitor the model performance:

- Accuracy: To observe the overall accuracy of the model
- AUC: To evaluate the classification of the model
- Precision: To evaluate True-Positive vs False-Negative
- Recall: To evaluate how the model performed in identifying relevant class.

```
metrics = [
      tf.keras.metrics.BinaryAccuracy(name='accuracy'),
      tf.keras.metrics.AUC(name='auc'),
      tf.keras.metrics.Precision(name='precision'),
      tf.keras.metrics.Recall(name='recall'),
]

metrics_names =  ['loss','accuracy', 'auc', 'precision', 'recall']
```

Processing- Modeling Approach

- I used the binary cross entropy function as my loss function that should be optimized.

- Because this is a binary classification, the Adam (Adaptive Moment Estimation) optimizer function should perform better than the generic SGD (Stochastic Gradient Descent).

- After tuning the 0.001 learning rate showed a reasonable performance

```
#model compile
model.compile(optimizer = tf.keras.optimizers.Adam(learning_rate= 0.001),
              loss = tf.keras.losses.BinaryCrossentropy(),
              metrics = metrics)
```

Processing - Model fitting

I used class weights to give more importance to `recall` over precision. This helps the model to start with higher recall and then it converges to higher precision.

```
class_weight = {1: 1,
                0: 0.75}

history = model.fit(train_data,
                    validation_data = val_data,
                    class_weight = class_weight,
                    epochs=100)
```

Processing - Overfitting

One of the main challenges with ANN models is overfitting. To avoid overfitting, I took several steps:

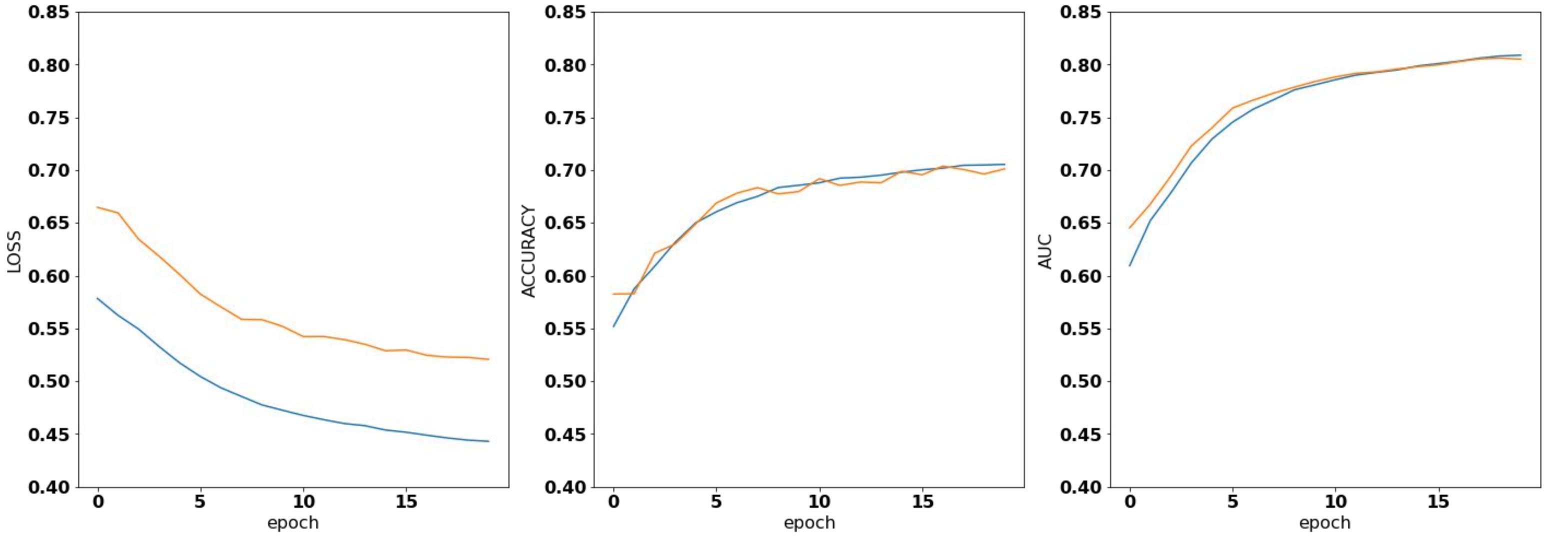
- **Carefully monitoring validation convergence:**
  The final model showed great performance on both training and validation data

- **Applying DropOut layers after large dense layers:**
  I used 50% drop out rate that helped a lot with over-fitting

- **Using multi-metric evaluation:**
  Instead of just relying on the loss and accuracy values, I also quantified recall, precision and AUC to make sure the model is on the right track.

Post-processing: Initial Performance

Only with 20 epochs the tuned model was already on a satisfying trajectory.



Post-processing: Initial Performance

I ran the model for a total of 200 epochs. The model run is relatively fast.

Data	Accuracy	AUC	Recall	Precision
Training				
Validation				
Test				

Engineering Architecture and Trade-Offs:

Question: What does the engineering architecture to productionize one of the solutions looks like? What trade-offs would you use to productionize promising algorithms faster?

Answer:

Business Performance and ROI

Question: Discuss how will you analyze the business performance of the algorithm once in production using metrics such as marketing ROI etc.?

Answer:

Code, Presentation and Repository

Code: The code and a self explanatory Jupyter Notebook is available from here:

https://colab.research.google.com/drive/1le1emKWg3kDgibay0SIrE6v0-VB460UZ?usp=sharing

Presentation: This presentation can be found from the following link:

https://docs.google.com/presentation/d/1bchwCkkt2tYCuIVC7vNqV2exXWL7I7Q15mca6QOwx_8/edit?usp=sharing

Repository: All the data and codes can be found from the following repository: