CarterKaoLetcherPhilippou_w261_FM_CTR

December 13, 2018

1 Click Through Rate (CTR) Prediction using Factorization Machines at Scale

1.1 MIDS w261 Final Project

1.2 December 12, 2018

Authors: Colby Carter, Kalvin Kao, Adam Letcher, Jennifer Philippou

2 1. Project Purpose

2.0.1 *Introduction*

Since the explosion of global internet usage, the advertising industry has adapted to utilize the internet to entice users to purchase their products, subscribe to their services, or generally perceive their brand in a positive light. Display advertising is a form of advertising on third party sites (such as social networks) or search engines that allow companies to market directly to consumers. This industry has grown rapidly with the adoption of internet usage in homes and on mobile devices worldwide, and a key metric in determining the effectiveness of such marketing campaigns is a "click through rate". Naturally, given the huge financial incentives, models which are able to identify important drivers of click through rates, or even predict the click through rate of a campaign, are highly desirable.

The Criteo Labs display ad challenge was a machine learning competition hosted on Kaggle that provided a week's worth of data to participants and asked them to build a predictive model that would answer the question "Given a user and the page they are visiting, what is the probability that they will click on a given ad?". The competition was scored by calculating a log-loss for submitted predictions based on a held-out test dataset. While the challenge concluded in 2014, this project will use the same dataset and scoring to demonstrate a scalable machine learning model that is capable of handling the large amount of Criteo data efficiently by using Apache Spark running on Google's cloud engine.

https://www.kaggle.com/c/criteo-display-ad-challenge

2.0.2 *Data Description*

The training dataset consists of a portion of Criteo's traffic over a period of 7 days. Each row corresponds to a display ad served by Criteo. Positive (clicked) and negatives (non-clicked) examples have both been subsampled at different rates in order to reduce the dataset size. Downloading the dataset requires signing a user agreement so it was not an automated step (link provided below). The datafields consist of the following:

- Label - Target variable that indicates if an ad was clicked (1) or not (0).
- I1-I13 - A total of 13 columns of integer features (mostly count features).
- C1-C26 - A total of 26 columns of categorical features. The values of these features have been hashed onto 32 bits for anonymization purposes.

The total training dataset consists of approximately 46 million records. There are a large number of categorical features which require a one-hot encoding approach (which will be discussed further in later sections), which result in a transformed dataset with approximately 35 million unique features, with only an average of 48 non-zero entries per record. This is a large, sparse dataset that will require special treatment in order to efficiently handle and take advantage of Spark's ability to process data in parallel and in memory (primarily). Additionally, in order to be useful from a practical point of view, the methodologies for modeling based on these data need to continue to be scalable far beyond just the seven day snapshot provided for the challenge, and lightweight enough to make predictions on future display ads in a timeframe that supports the ultimate business objectives.

http://labs.criteo.com/2014/09/kaggle-contest-dataset-now-available-academic-use/

2.0.3 *Project Outline*

In the remainder of this report, we will walk through an explanation of our chosen algorithmic approach using a toy example, provide an exploratory data analysis (EDA) using a sample of the full dataset, and then describe our implementation of the algorithm at scale. Finally, we will discuss the practical concepts of scalability utilized in our approach, highlight any challenges that arose during the course of the project, and identify any future exploration or modifications that could be made to our approach in order to improve its predictive power or training performance.

3 2. Factorization Machines

For this project, our team chose to take an approach that utilized Factorization Machines (FM), similar to other successful challenge participants. FMs are a model class that combine the advantages of matrix factorization models with Support Vector Machines (SVM). FMs are a general predictor that work with any real valued feature vector, but they improve on SVMs by modeling all interaction terms between variables using factorized variables. This allows FMs to model complex interactions even in situations with a large amount of sparsity, such as this particular dataset (described above). Additionally, as shown in the original paper by Rendle, FMs can be calculated in linear time through a reformulation of the model equation, making them very attractive in terms of a scalable solution.

https://www.csie.ntu.edu.tw/~b97053/paper/Rendle2010FM.pdf

https://www.csie.ntu.edu.tw/~r01922136/slides/ffm.pdf

The model equation for an FM of degree equal to two is shown in the image below. As described in the original Rendle paper, a 2-way FM models all single and pairwise iteractions between features where:

- w_0 represents the global bias
- w_i represents the strength of the inidividual feature i

$$\hat{y}(\mathbf{x}) := w_0 + \sum_{i=1}^{n} w_i \, x_i + \sum_{i=1}^{n} \sum_{j=i+1}^{n} \langle \mathbf{v}_i, \mathbf{v}_j \rangle \, x_i \, x_j$$

FM Equation

$$= \frac{1}{2} \sum_{f=1}^{k} \left(\left(\boxed{\sum_{i=1}^{n} v_{i,f} \, x_i} \right)^2 - \sum_{i=1}^{n} v_{i,f}^2 \, x_i^2 \right)$$

FM factor

- $< v_i, v_j >$ represents the interaction between the i-th and j-th features

The key difference between the FM model and a general polynomial model is the third term, which models the interactions as the dot product of two weights, rather than an individual weight w_i, j. This factorization means that no model parameter depends on any two variables and therefore can be reformulated to an equation that has complexity $O(k \cdot n)$ instead of $O(n^2)$. The image below shows the right hand summands, which have been rewritten as the average of differences between dot products for each factor k:

This closed form equation can be solved in linear time, and therefore can be learned efficiently using forms of Gradient Descent. In later sections of this document, we will describe our implementation of gradient descent to learn a 2-way FM model. However, first we will proceed with a toy problem to illustrate the mathematical description of FMs above in a practical (although simple) setting.

3.1 Toy Example

Our team will use the toy data below to try and demonstrate the logic and flow of the FM approach, where the data is a set of observations in tab separated rows. In each row, the first value is a binary label where 1 represents a positive click-through and 0 represents no click-through. The remaining values are the numeric and categorical features associated with each record, which must be parses and stored in a tuple of (label, features).

```
In [10]: import numpy as np

         toyDataRaw = ['1\t0\t5\t\t1\t26\tcat\tblue\t\tpizza',
                       '0\t1\t10\t1\t\t12\tdog\tyellow\t\t',
                       '0\t0\t\t0.5\t2\t45\tdog\t\tcar\tsteak']

In [11]: # parse out label and features
         toyDataParsed = []
         for row in toyDataRaw:
             splitRow = row.split('\t')
             toyDataParsed.append((splitRow[0], splitRow[1:]))
```

```
        print("Toy data made up of label followed by numeric and categorical features:")
        toyDataParsed
```

```
Toy data made up of label followed by numeric and categorical features:
```

```
Out[11]: [('1', ['0', '5', '', '1', '26', 'cat', 'blue', '', 'pizza']),
          ('0', ['1', '10', '1', '', '12', 'dog', 'yellow', '', '']),
          ('0', ['0', '', '0.5', '2', '45', 'dog', '', 'car', 'steak'])]
```

3.1.1 *Summarize Data*

In addition to the format of the parsed data, it is useful to examine the number of non-zero features in the data. This metric will become even more important in later processing steps when we one-hot encode features, which will result in a highly sparse data structure with a very low average number of non-zero features relative to the total number of derived features.

```
In [12]: ncol = len(toyDataParsed[0][1])
         nrow = len(toyDataParsed)
         print(f'This toy example contains {nrow} rows and {ncol} columns, plus a label in inde
```

```
This toy example contains 3 rows and 9 columns, plus a label in index 0.
```

```
In [13]: def avgFeatures(row):
             count = 0
             feats = row[1][:]
             for feat in feats:
                 if feat != '':
                     count += 1
             return count

         nonSparse = [avgFeatures(row) for row in toyDataParsed]

         print("There is an average of", str(round(np.mean(nonSparse),2)), "populated features
```

```
There is an average of 6.67 populated features per observation.
```

3.1.2 *One-Hot Encode Features*

A basic approach to dealing with categorical features is to one-hot encode them, where instead of a single column with many possible values, we transform the data to capture all possible values as features, and the values are simple binary values. In this toy example, we first take a step to create a list of features that is simply a string which concatenates the original feature index and then the feature value. This is done to maintain a distinction between any two columns in the original dataset which might coincidentally have the same feature value.

```
In [16]: # binarize
         def makeString(data):
             """Get list of features and make them into distinct strings according to column i
              #include label for SGD
             newData = []
             for r, row in enumerate(data):
                 label = row[0]
                 id_feats = []
                 for i, value in enumerate(row[1], 1):
                     if value=='':
                         add='NA'
                     else:
                         add=value
                     id_feats.append("v"+str(i)+"="+add)
                 newData.append((label, id_feats))

             return newData

         stringData = makeString(toyDataParsed)
         print("Example of string-indexed features:")
         stringData[0]
```

```
Example of string-indexed features:
```

```
Out[16]: ('1',
          ['v1=0',
           'v2=5',
           'v3=NA',
           'v4=1',
           'v5=26',
           'v6=cat',
           'v7=blue',
           'v8=NA',
           'v9=pizza'])
```

Next, we need to take the list of features for each record, and then one-hot encode them so that each record has a list of binary values for all possible features, not simply the ones it has.

```
In [19]: def oneHotEncode(data):
             """turn indexed-string features into one-hot encoded features"""

             setFeats = set()
             for row in data:
                 setFeats.update(row[1])
             listFeats = list(setFeats)
             print("Features:")
             print(listFeats)
             newData = np.zeros(shape=(len(data), len(listFeats)+1))
```

```
        for r, row in enumerate(data):
            newData[r][0] = row[0]     #first index is the label
            for var in row[1]:
                newData[r][listFeats.index(var)+1] = 1

        return newData, len(listFeats)

    oneHotData, numFeats = oneHotEncode(stringData)
    print("\nOne-hot encoded features (first element is label):")
    oneHotData[0]
```

```
Features:
['v1=0', 'v7=blue', 'v9=NA', 'v5=26', 'v8=car', 'v5=12', 'v7=yellow', 'v2=5', 'v9=pizza', 'v2=

One-hot encoded features (first element is label):


Out[19]: array([1., 1., 1., 0., 1., 0., 0., 0., 1., 1., 0., 0., 0., 1., 1., 0., 0.,
                0., 1., 0., 0., 0., 0., 0., 1.])
```

3.1.3 *Model Updates using Gradient Descent*

With the data transformed into our required format, we now initialize the weights for the 2-way factorization machine. Since we are using gradient descent, it shouldn't really matter where we initialize these weight vectors, so we choose a bias equal to zero and two weight vectors filled with random numbers.

```
In [7]: # initialize model
        b = 0.0
        w_vector = np.random.normal(0.0, 0.02, (1, numFeats))
        k = 2    #number of latent factors
        V_matrix = np.random.normal(0.0, 0.02, (k, numFeats))   #k factors

        print("Initialized weight vector W:")
        w_vector
```

```
Initialized weight vector W:


Out[7]: array([[-0.03328396,  0.03109567, -0.00353963,  0.00108691,  0.01301888,
                0.00522236,  0.01511192, -0.00106579, -0.00058697,  0.00502548,
                0.00981188, -0.00124704,  0.00902628, -0.00041259, -0.00939172,
               -0.01471153,  0.02622656, -0.00783223, -0.00401789,  0.00080727,
                0.03522305, -0.02387439, -0.00679598, -0.00943945]])
```

Using logarithmic-loss as our cost function along with the chain rule, we can use the product of the following partial derivatives with the loss function's derivative, $(\hat{p}_i - y_i)$, to estimate gradients by parameter. The highlighted summand has already been calculated in our $\hat{y}(x)$ equation from

$$\frac{\partial}{\partial\theta}\hat{y}(\mathbf{x}) = \begin{cases} 1, & \text{if } \theta \text{ is } w_0 \\ x_i, & \text{if } \theta \text{ is } w_i \\ x_i \boxed{\sum_{j=1}^{n} v_{j,f} x_j} - v_{i,f} x_i^2, & \text{if } \theta \text{ is } v_{i,f} \end{cases}$$

FM factor

above, saving us computation time. In our later description of the scaled implementation with Spark, we will also show how we can save additional computation time by only calculating the updated partial gradients for the non-zero values. In an extremely large feature set built with a sparse-represented data structure, this saves considerable time.

```
In [2]: def estimateGradientToy(record, k, b, w, V):
            """
                Compute the predicted probability AND return the gradients
                Args:
                    record - label followed by binarized feature values
                Model:
                    b - bias term (scalar)
                    w - linear weight vector (array)
                    k - number of factors (def=2)
                    V - factor matrix of size (d dimensions, k=2 factors)
                Returns:
                    pair - ([label, predicted probability], [set of weight vectors in csr_matr
            """

            label = record[0]
            feats = record[1:]

            # calculate P-hat
            # start with linear weight dot product (X dot W)
            linear_sum = np.dot(w, feats)

            # factor matrix interaction sum
            factor_sum = 0.0
            lh_factor = [0.0]*k
            rh_factor = [0.0]*k
            for f in range(0, k):
                lh_factor[f] = np.dot(V[f][:], feats)  # we take dot product of ALL elements o
                rh_factor[f] = np.dot(V[f][:]**2, feats**2)
                factor_sum += (lh_factor[f]**2 - rh_factor[f])
            factor_sum = 0.5 * factor_sum

            y_hat = b + linear_sum + factor_sum

            p_hat = 1.0 / (1 + float(np.exp(-y_hat)))  #logit transformation
            #compute Gradients
            b_grad = p_hat - label    #the partial derivative of log-loss function wrt constan

            w_grad = b_grad*feats

            v_data = np.array([])
            for f in range(0, k):
                # this would be too many (unnecessary) computations on a very large, sparse fe
                v_data = np.append(v_data, b_grad*(lh_factor[f]*feats - np.multiply(V[f][:], fe
            v_grad = np.reshape(v_data, newshape=(k, V.shape[1]))

            return ([label, p_hat], [b_grad, w_grad, v_grad])
```

```
In [26]: # for one example
         gradient = estimateGradientToy(oneHotData[0], k, b, w_vector, V_matrix)
         print("(Label, predicted probability), [beta, w vector, V matrix]:")
         gradient
```

```
(Label, predicted probability), [beta, w vector, V matrix]:


Out[26]: ([1.0, 0.5024168636288922],
          [-0.49758313637110785,
           array([-0.49758314, -0.        , -0.49758314, -0.        , -0.49758314,
                  -0.49758314, -0.        , -0.49758314, -0.        , -0.        ,
                  -0.49758314, -0.        , -0.        , -0.        , -0.        ,
                  -0.        , -0.49758314, -0.        , -0.        , -0.49758314,
                  -0.        , -0.        , -0.        , -0.49758314]),
           array([[-0.051016  , -0.        , -0.03645439, -0.        , -0.03854385,
                   -0.02986771, -0.        , -0.03737504, -0.        , -0.        ,
                   -0.02509388, -0.        , -0.        , -0.        , -0.        ,
                   -0.        , -0.05390807, -0.        , -0.        , -0.01865564,
                   -0.        , -0.        , -0.        , -0.04345672],
                  [ 0.0244058 , -0.        ,  0.02071585, -0.        ,  0.02966948,
                    0.03270131,  0.        ,  0.01629193,  0.        , -0.        ,
                    0.03281537, -0.        , -0.        , -0.        , -0.        ,
                   -0.        ,  0.02478096,  0.        ,  0.        ,  0.02006145,
                    0.        ,  0.        ,  0.        ,  0.03295628]])])
```

We calculate the log-loss based on our current set of predictions and their actual click outcomes. This will be useful later when we iteratively update the model until we see a decline in further improvement.

$$logloss = -\frac{1}{N}\sum_{i=1}^{N}[y_i \cdot log p_i + (1 - p_i) \cdot log(1 - p_i)]$$

```
In [10]: def logLossToy(pair):
             """parallelize log loss
```

```
            input: ([label, prob], [b_grad, w_grad, v_grad])
        """
        y = pair[0][1]

        eps = 1.0e-16
        if pair[0][1] == 0:
            p_hat = eps
        elif pair[0][1] == 1:
            p_hat = 1-eps
        else:
            p_hat = pair[0][1]

        return float(-(y * np.log(p_hat) + (1-y) * np.log(1-p_hat)))
```

```
In [11]: logLossToy(gradient)
```

```
Out[11]: 0.6931354980548503
```

We then use the partial gradients calculated in `estimateGradientToy` and combine those using a reduce and then update our weight vectors by subtracting off the gradient times a learning rate. This would then feed back into a new set of model predictions, a new calculation of loss, and then continued updates of the model, either until the loss converges or a set number of iterations.

```
In [24]: # update weights
         learningRate = 0.1

         # initialize gradient
         wGrad_reduce = np.zeros((1, numFeats))

         # aggregate partial gradients while iterating over rows
         for r in range(0, nrow):
             gradient = estimateGradientToy(oneHotData[r], k, b, w_vector, V_matrix)
             wGrad_reduce += gradient[1][1]

         # calculate average gradient
         w_update = wGrad_reduce / nrow

         # update weight vector
         w_new = w_vector - learningRate*w_update

         print("New weight vector W")
         w_new.T
```

```
New weight vector W
```

```
Out[24]: array([[-0.03310685],
                [ 0.01428775],
                [-0.00376144],
                [-0.015721  ],
                [ 0.02960499],
                [ 0.02180846],
                [-0.001696  ],
                [ 0.01552031],
                [-0.01699597],
                [-0.01138352],
                [ 0.02639799],
                [-0.01805496],
                [-0.00738271],
                [-0.01722051],
                [-0.02619963],
                [-0.03151945],
                [ 0.04281266],
                [-0.04104915],
                [-0.02042689],
                [ 0.01739337],
                [ 0.01881405],
                [-0.04028339],
                [-0.02320497],
                [ 0.00714665]])
```

```
In [23]: # repeat process to update V matrix

         vGrad_reduce = np.zeros((k, numFeats))
         for r in range(0, nrow):
             gradient = estimateGradientToy(oneHotData[r], k, b, w_vector, V_matrix)
             vGrad_reduce += gradient[1][2]
         v_update = vGrad_reduce / nrow

         V_new = V_matrix - learningRate*v_update

         print("New factor matrix V weights:")
         V_new.T
```

```
New factor matrix V weights:
```

```
Out[23]: array([[-0.01659269, -0.0107129 ],
               [-0.00526753, -0.01770845],
               [ 0.01061032, -0.01639128],
               [ 0.03818184, -0.00625369],
               [ 0.00782152, -0.00024604],
               [ 0.02496888,  0.00574602],
               [ 0.01363172,  0.01560516],
               [ 0.01013153, -0.02668518],
               [ 0.01761522,  0.00018196],
               [-0.06089243, -0.03704973],
               [ 0.03440378,  0.00597145],
               [ 0.01331686, -0.01696164],
               [ 0.02983592, -0.01582817],
               [ 0.02259963, -0.04301197],
               [ 0.00912995, -0.01519263],
               [-0.00842256, -0.00671412],
               [-0.02254404, -0.00990762],
               [-0.01389515,  0.01140745],
               [-0.00252248,  0.00297071],
               [ 0.04712819, -0.01923517],
               [-0.01277656,  0.01345092],
               [ 0.0081833 ,  0.02907576],
               [ 0.02244913,  0.00251953],
               [-0.00188818,  0.00624993]])
```

4 3. Exploration of Criteo Training Dataset

We begin by exploring a sample of our training dataset's numeric and categorical features, understanding their distributions and potential sparsity, and consider how we can best transform them into features efficiently used by FM.

```
In [5]: # imports
        import numpy as np
        import pandas as pd

        # visuals
        import seaborn as sns
        import matplotlib.pyplot as plt
        from matplotlib.pyplot import figure

        import time
        from scipy.sparse import csr_matrix
        from pyspark.sql import Row
        from pyspark.ml.feature import CountVectorizer
        from pyspark.sql import DataFrame
```

```
In [6]: # start Spark Session
        from pyspark.sql import SparkSession
        app_name = "w261FinalProject"
        master = "local[*]"
        spark = SparkSession\
                .builder\
                .appName(app_name)\
                .master(master)\
                .getOrCreate()
        sc = spark.sparkContext
```

Our team performed our EDA using a sample of ~230,000 observations from the training data. The EDA results are first presented on the numerical data in the Criteo display advertising dataset in order to understand the distributions of their values, their correlations, and the extent of missing data. We then present the EDA results for the categorical data in order to also understand their frequency distributions and missing values. Most of these analyses were performed with a sub-sample of 5000 observations. These details and results informed our transformations and representation of the data.

4.0.1 Load Training Data Sample for Local Testing

```
In [7]: original_trainRDD = sc.textFile('data/train.txt')

        splits = 0.005

        largeRDD, smallTestRDD, smallTrainRDD = original_trainRDD.randomSplit([1-2*splits, spli
        smallTrainRDD.cache()

        ncol = len(smallTrainRDD.take(1)[0].split('\t'))
        nrow = smallTrainRDD.count()
        print(f'This sample contains {nrow} rows and {ncol} columns')
```

```
This sample contains 229937 rows and 40 columns
```

4.0.2 *Numeric Variables*

Below, the numerical features are extracted for EDA.

```
In [5]: def parse_numbers(line):
            """
            This function selects the numerical variables from the dataset for EDA
            """
            fields = np.array(line.split('\t')[:14])
            label,features = fields[0], fields[1:]
            return(features, label)
```

```
In [6]: numeric_trainRDDCached = smallTrainRDD.map(parse_numbers).cache()
        numeric_sample = np.array(numeric_trainRDDCached.map(lambda x: np.append(x[0], [x[1]])
        numeric_sample_df = pd.DataFrame(np.array(numeric_sample))
```

The first 5 rows of the EDA sample (numerical only) are shown below to give an example of the numerical features.

```
In [8]: print("Example of Numerical Features Extracted:")
        numeric_sample_df.head()
```

```
Example of Numerical Features Extracted:
```

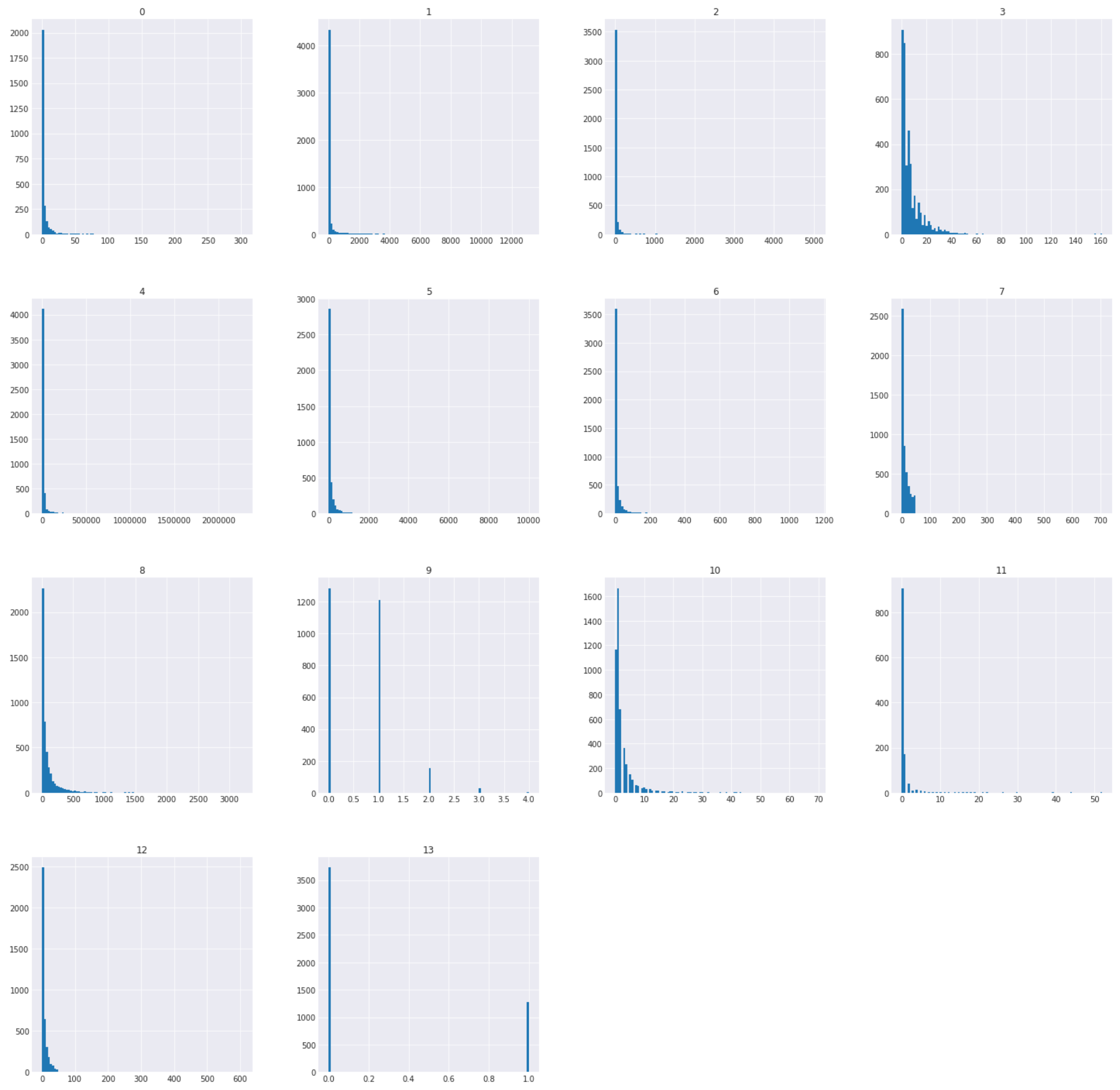
```
Out[8]:    0   1   2   3     4   5   6   7    8   9   10  11   12  13
        0  0  -1   5   2    12  61   4  45  102   0   2        2   0
```

```
1  6  19   6   9     24  10  25   9    9  2  11  1   9  0
2  1  24   2   1    210  11   8  12  260  1   6      1  1
3      7  35  17  11232       0  17   45      0     17  0
4     84  25  11    217       0  15   14      0  1  14  1
```

```
In [16]: numeric_sample_df = numeric_sample_df.apply(pd.to_numeric)
```

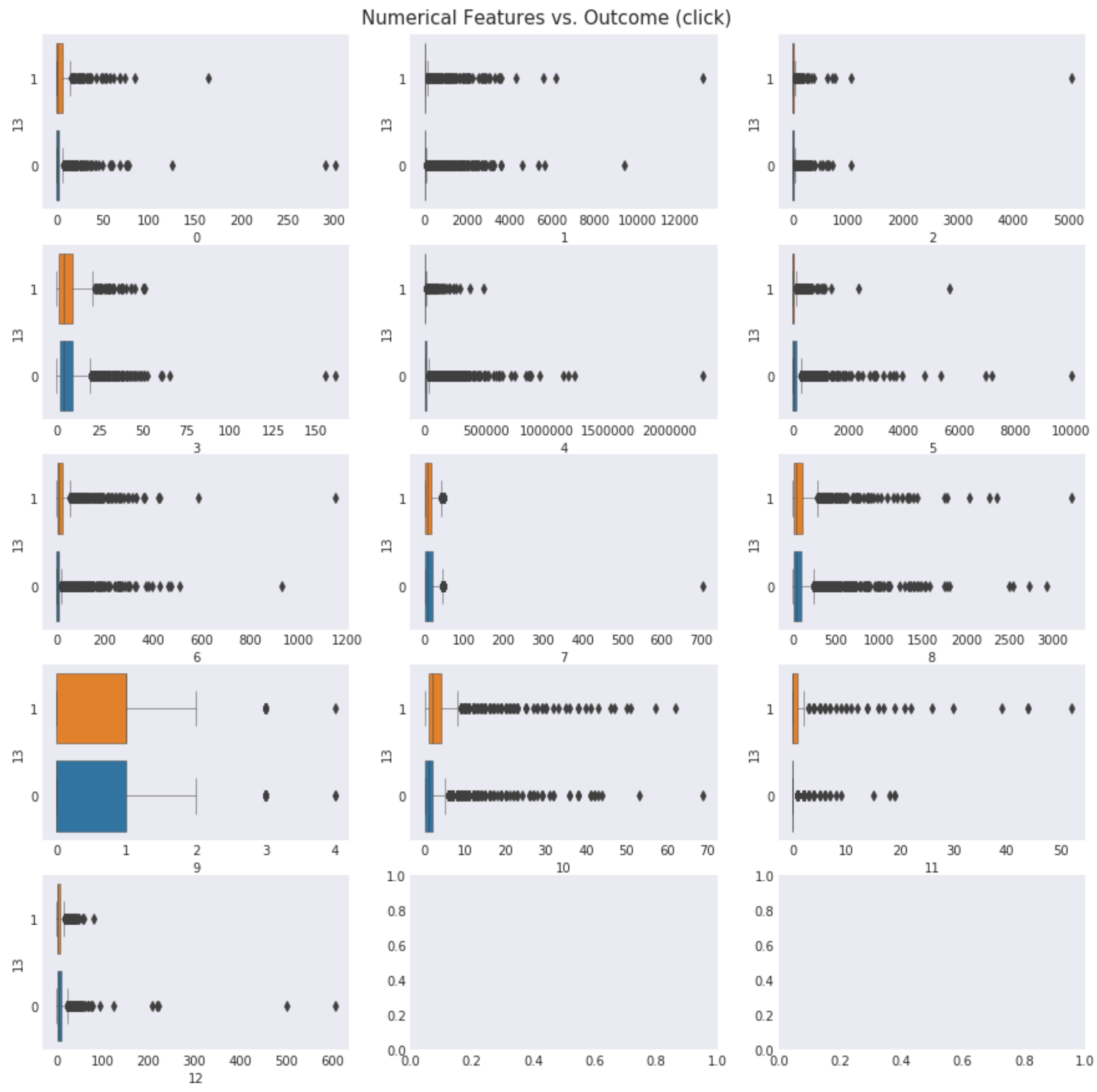
The histograms of each numerical feature are shown below. The 13th column holds the outcome variable.

```
In [17]: numeric_sample_df.hist(figsize=(25,25), bins=100)
         plt.show()
```



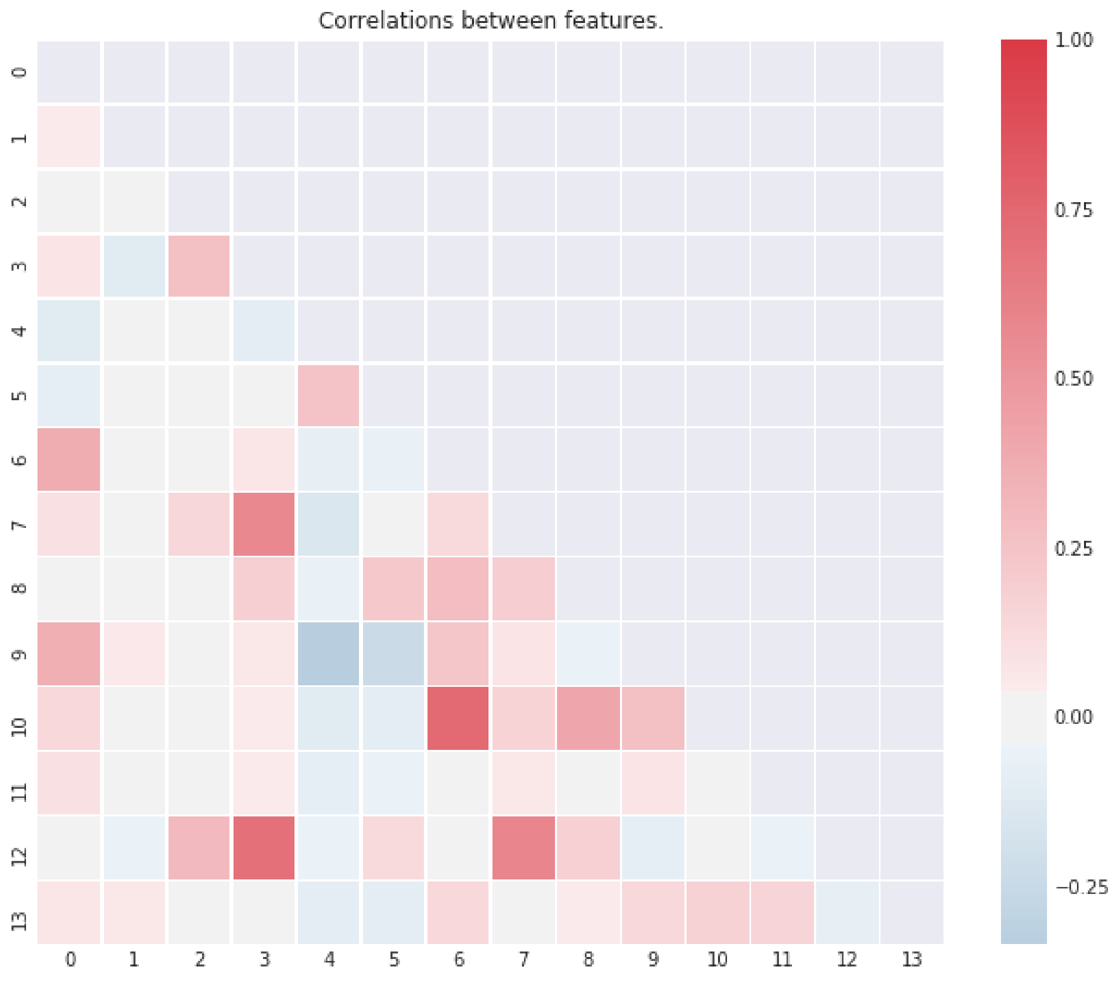
All of the independent numerical variables, with the exception of the 10th numerical variable (index 9), show a very strong right (positive) skew, which suggests that a log-transform might be helpful. The above distributions also provide guidance on bucketing criteria in the subsequent data transformation. Next, we use a series of boxplots to identify any obvious correlations between the independent numerical variables and the click through response.

```
In [28]: fig, ax_grid = plt.subplots(5, 3, figsize=(15,15))
         y = numeric_sample_df.loc[:,13]#.astype("float")
         for idx in range(len(numeric_sample_df.columns)-1):
             x = numeric_sample_df.loc[:,idx]
             sns.boxplot(x, y, ax=ax_grid[idx//3][idx%3], orient='h', linewidth=.5)
             ax_grid[idx//3][idx%3].invert_yaxis()
         fig.suptitle("Numerical Features vs. Outcome (click)", fontsize=15, y=0.9)
         plt.show()
```



There is some minor evidence that the 11th numerical feature (index 10) has a positive correlation with click response, but the above boxplots otherwise show no obvious correlations with the target variable. Below, we show a heatmap of pair-wise correlations between the independent variables to identify any strong collinearity within the dataset.

```
In [29]: corr = numeric_sample_df.loc[:,:14].corr()
         fig, ax = plt.subplots(figsize=(11, 9))
         mask = np.zeros_like(corr, dtype=np.bool)
         mask[np.triu_indices_from(mask)] = True
         cmap = sns.diverging_palette(240, 10, as_cmap=True)
         sns.heatmap(corr, mask=mask, cmap=cmap, center=0, linewidths=.5)
         plt.title("Correlations between features.")
         plt.show()
```



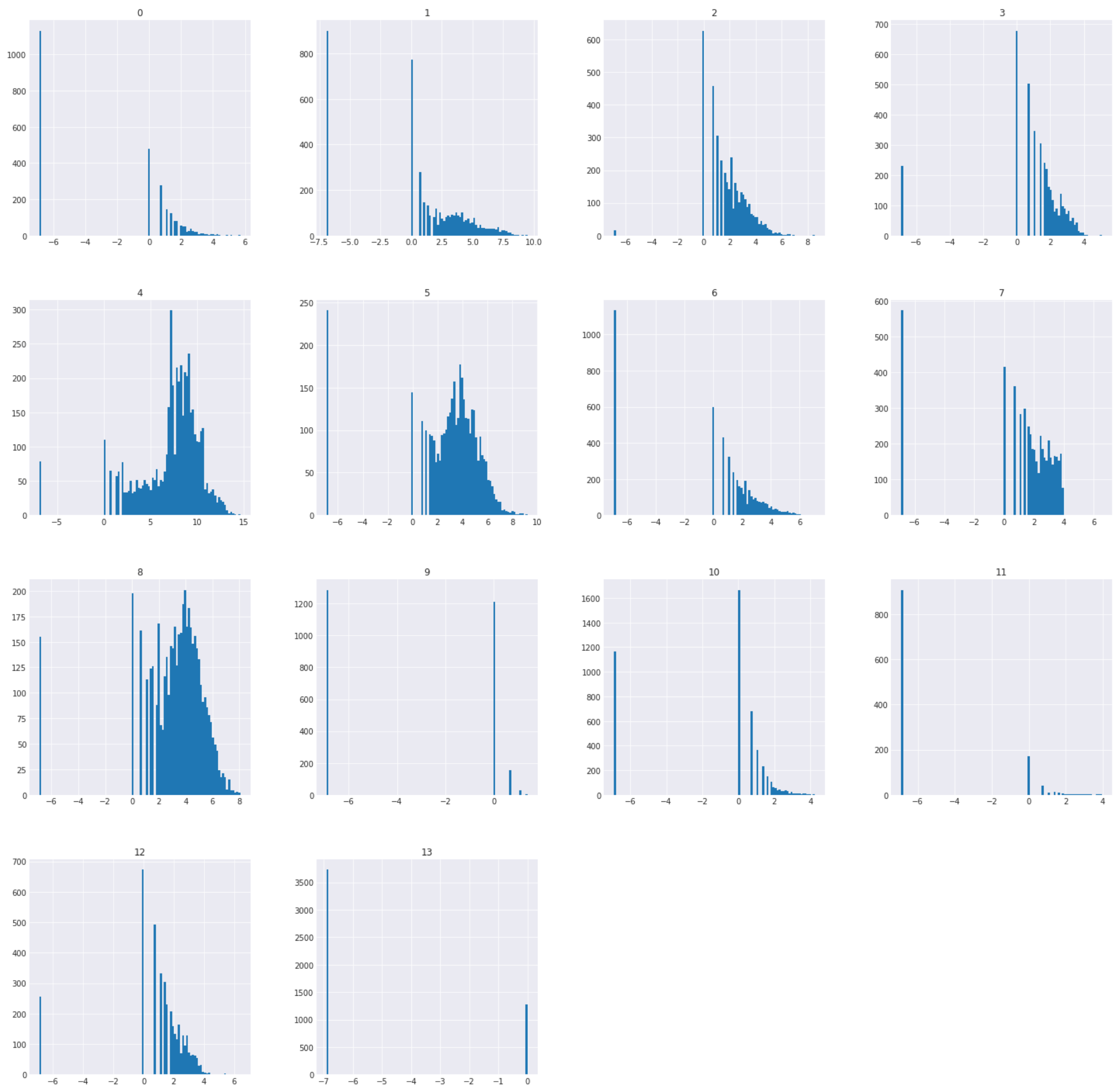
Numerical feature pairs 6 - 10 and 3 - 12 (by index) show evidence of a positive correlation, but the remaining features otherwise do not indicate strong collinearity.

4.0.3 *Log Transform*

Due to the high positive skew in the histograms of the numerical features, we have repeated the prior EDA using the log form to assess its potential to improve their predictive power.

```
In [30]: small_constant = 0.001 #to be added since these variables contain a high number of ze
         log_numeric_sample_df = numeric_sample_df.apply(lambda x: x+small_constant).apply(np.l
```

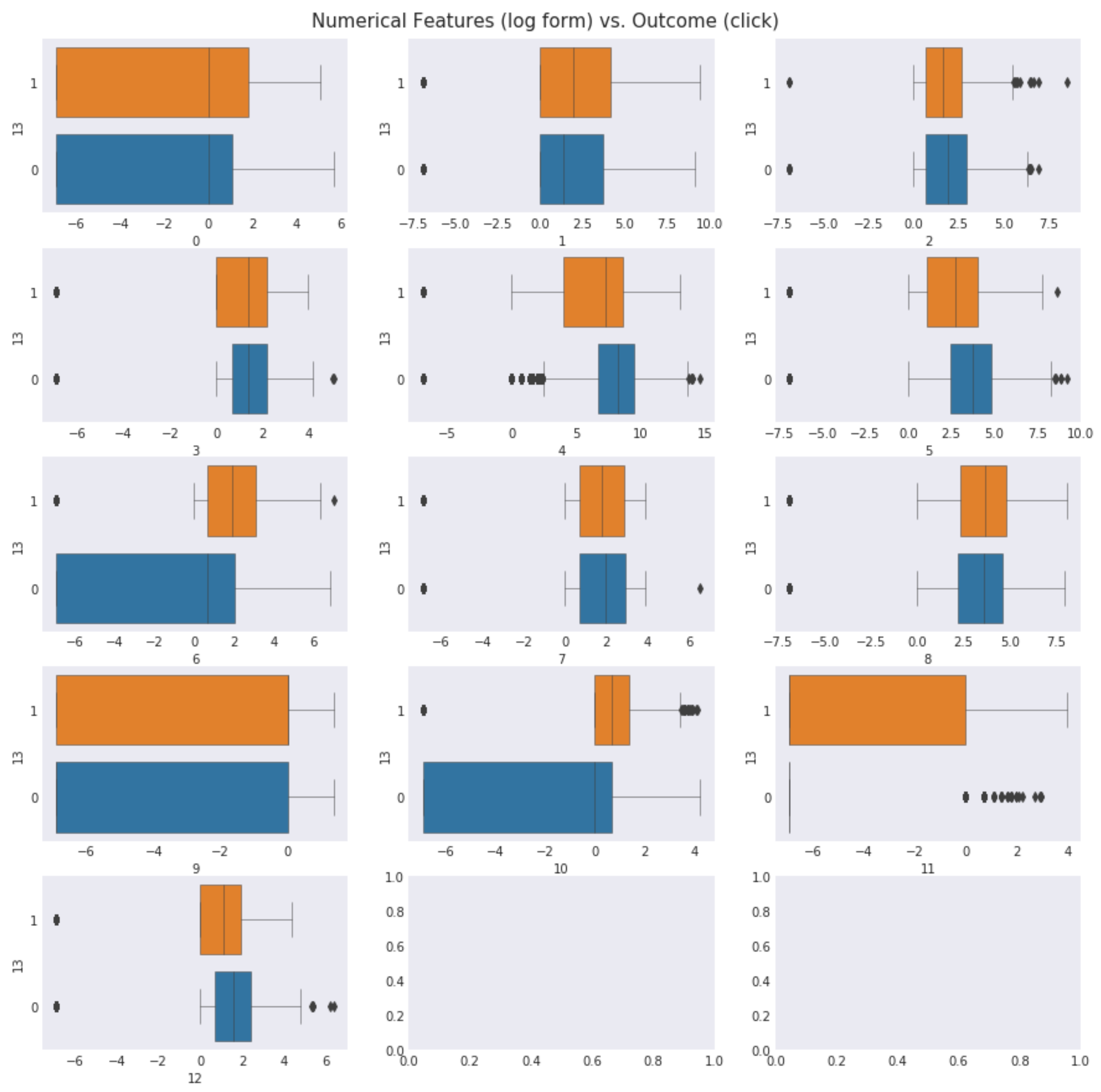
```
In [31]: log_numeric_sample_df.hist(figsize=(25,25), bins=100)
         plt.show()
```



The log transform indeed addresses the skew in the numerical features, however, the high number of zeros in these variables results in a peak that is distant from the distribution of non-zero raw values. Again, we show the boxplots for each of the log-transformed variables below.

```
In [32]: fig, ax_grid = plt.subplots(5, 3, figsize=(15,15))
         y = numeric_sample_df.loc[:,13]#.astype("float")
         for idx in range(len(numeric_sample_df.columns)-1):
             x = log_numeric_sample_df.loc[:,idx]
             sns.boxplot(x, y, ax=ax_grid[idx//3][idx%3], orient='h', linewidth=.5)
```

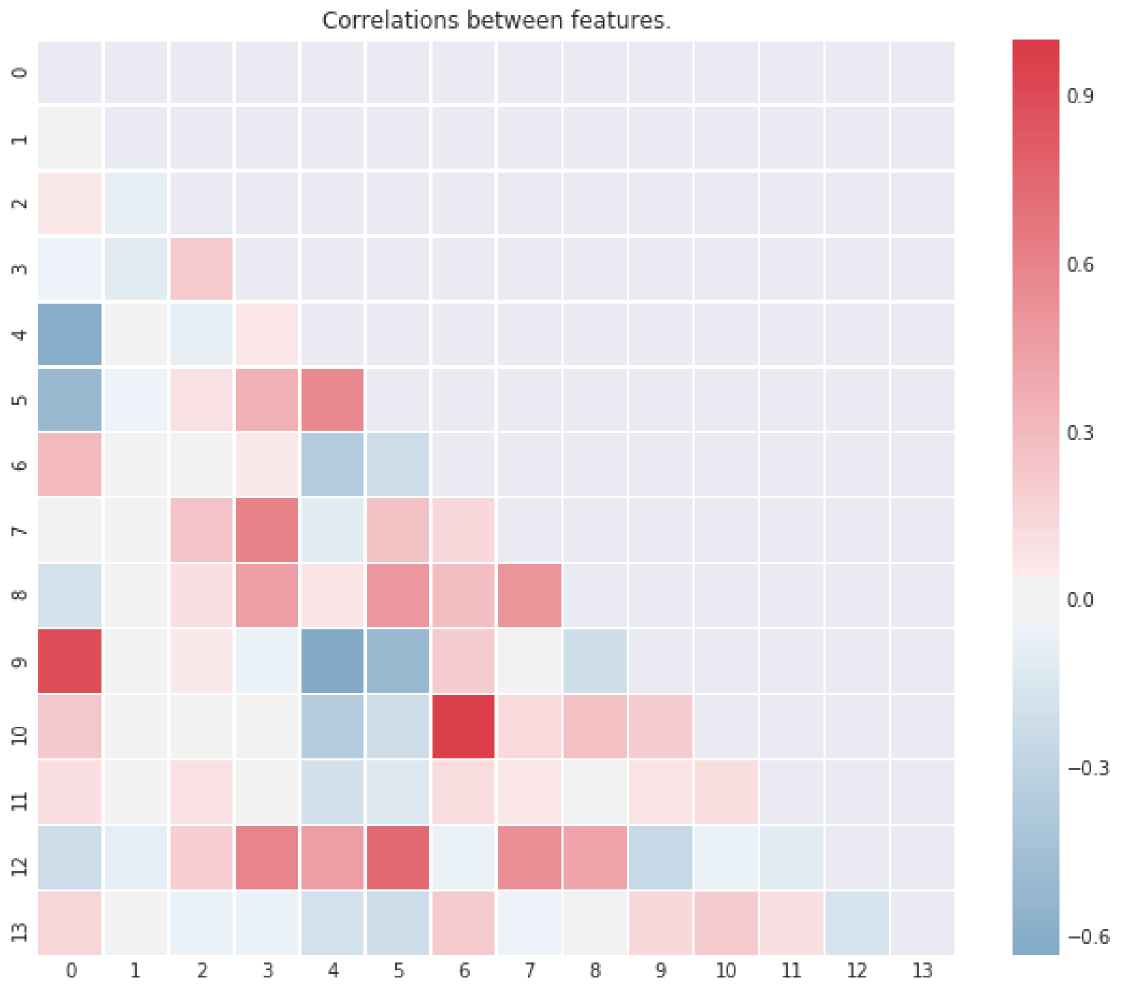
```
        ax_grid[idx//3][idx%3].invert_yaxis()
    fig.suptitle("Numerical Features (log form) vs. Outcome (click)", fontsize=15, y=0.9)
    plt.show()
```



Similar to before, the boxplots above suggest a positive correlation between the log form of the 11th numerical feature (index 10) and the outcome, and additionally suggest a potential negative correlation between the log form of the 6th numerical feature (index 5) and the outcome. Next, we show a heatmap for pair-wise correlations of the log transforms of the independent variables.

```
In [33]: corr = log_numeric_sample_df.loc[:,:14].corr()
         fig, ax = plt.subplots(figsize=(11, 9))
         mask = np.zeros_like(corr, dtype=np.bool)
         mask[np.triu_indices_from(mask)] = True
         cmap = sns.diverging_palette(240, 10, as_cmap=True)
```

```
sns.heatmap(corr, mask=mask, cmap=cmap, center=0, linewidths=.5)
plt.title("Correlations between features.")
plt.show()
```



The log forms of the numerical features again show a strong positive correlation in feature pair 6 - 10 (by index), but not for feature pair 3 - 12. Instead, feature pair 0 - 9 becomes positively correlated when in log form. But again, the remaining variable pairs do not show evidence of high collinearity.

4.0.4 *Categorical Variables*

Below, we extract the categorical features for EDA.

```
In [8]: def parse_categories(line):
            """
            Map record_csv_string --> (tuple,of,fields)
            """
            fields = np.array(line.split('\t'))
```

```
        label,features = fields[0], fields[14:]
        return(features, label)
```

```
In [9]: categorical_trainRDDCached = smallTrainRDD.map(parse_categories).cache()
```

```
In [10]: categorical_sample = np.array(categorical_trainRDDCached.map(lambda x: np.append(x[0]
         categorical_sample_df = pd.DataFrame(np.array(categorical_sample))
```

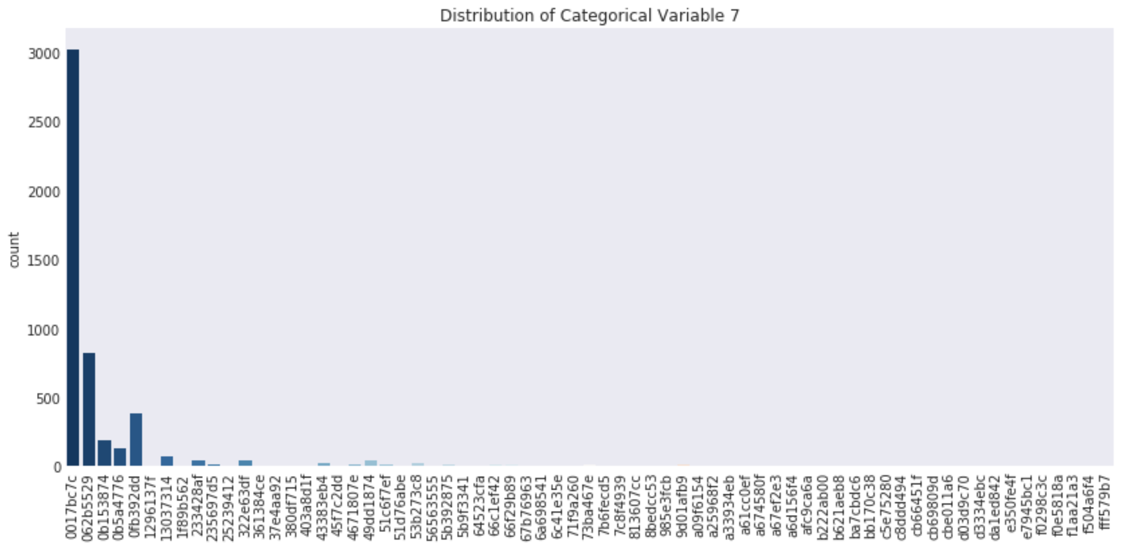
The first 5 rows of the EDA sample (categorical only) are shown below to provide an example of the categorical features.

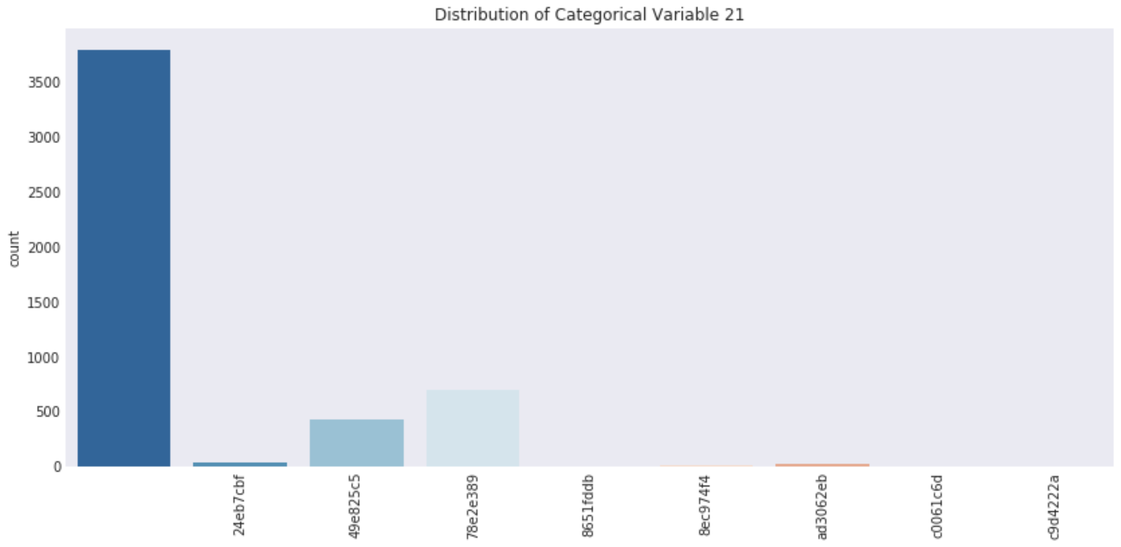
```
In [11]: categorical_sample_df.head()  #26 features + label
```

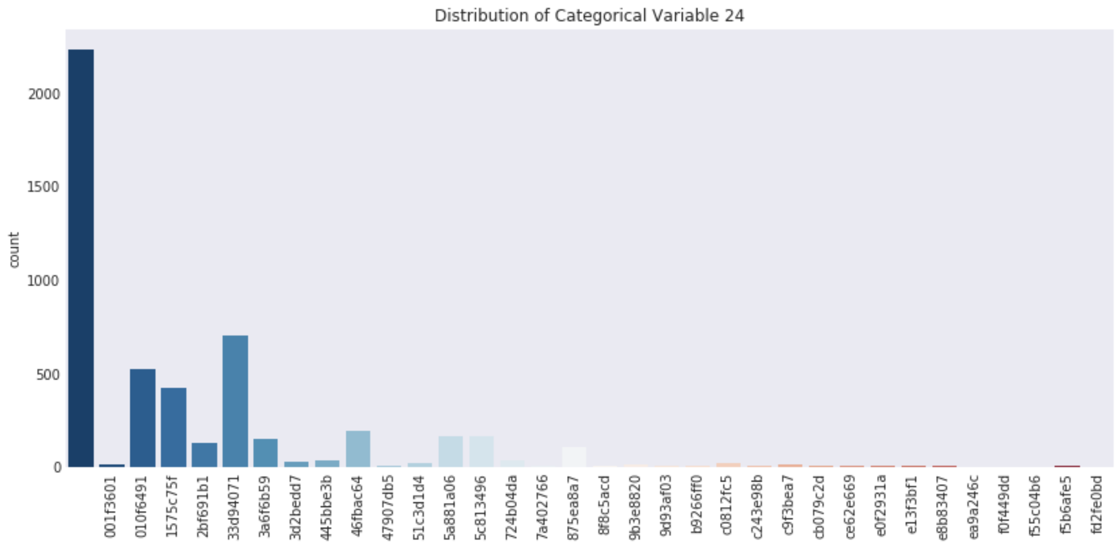
```
Out[11]:          0         1         2         3         4         5         6   \
         0  be589b51  f3139f76  b90edd83  bf0b19a8  25c83c98  7e0ccccf  3965ff35
         1  241546e0  38a947a1  5905f6e3  11f7f740  25c83c98  6f6d9be8  60db3a7e
         2  be589b51  8aade191  d82a5184  55699589  43b19349  fe6b92e5  66ad28b2
         3  68fd1e64  c44e8a72  0f0f773d  a6bd88d7  25c83c98  7e0ccccf  2575d83f
         4  05db9164  09e68b86  aa8c1539  85dd697c  25c83c98  7e0ccccf  372a0c4c

                  7         8         9 ...        17        18        19        20  \
         0  0b153874  a73ee510  267caf03 ...  78db103b                      3b226dea
         1  5b392875  a73ee510  712eb033 ...  f92d697a                      45c5cb57
         2  0b153874  a73ee510  3b08e48b ...  eef7297e                      f2f1547c
         3  0b153874  a73ee510  fbbf2c95 ...  456d734d  8733cf72  b1252a9d  aad9b4ce
         4  37e4aa92  a73ee510  a08eee5a ...  63cdbb21  21ddcdc9  5840adea  5f957280

           21        22        23        24        25 26
         0       32c7478e  4fcc135f                    0
         1       32c7478e  10864bee                    1
         2       32c7478e  8d4a9014                    1
         3       3a171ecb  3a586084  724b04da  ad323355  0
         4       3a171ecb  1793a828  e8b83407  b7d9c3bc  0

         [5 rows x 27 columns]
```

The count distribution across values was examined for each categorical variable in the 5,000-observation EDA sample. For conciseness, only 3 example histograms are shown below, demonstrating the high skew in the frequency distributions that is even more extreme across a number of the other categorical features. Note that `blanks` are often the most frequent entry within a categorical variable.

```
In [14]: for idx in [7, 21, 24]:
             columnName = categorical_sample_df.columns[idx]
             labels = np.unique(categorical_sample_df[columnName].values)
             fig = plt.figure(figsize=(14,6))
             ax = fig.add_subplot(111)
             sns.countplot(x=categorical_sample_df[columnName].values,data=categorical_sample_
             ax.set_xticklabels(labels, rotation=90)
             ax.set_title("Distribution of Categorical Variable " + str(columnName))
```

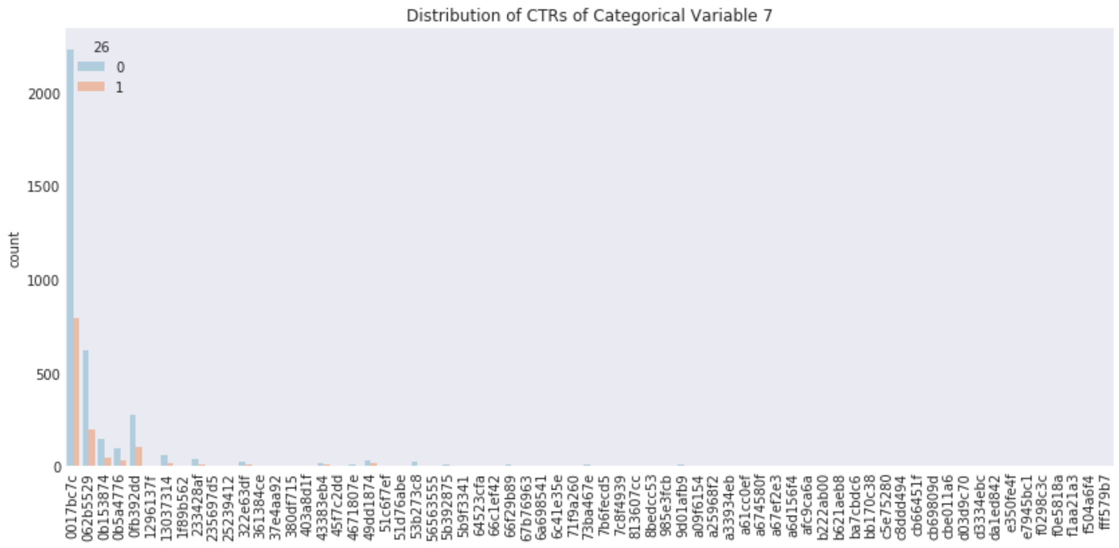

Distribution of Categorical Variable 7
count
3000
2500
2000
1500
1000
500
0
0017bc7c
062b5529
0b153874
0b5a4776
0fb392dd
1296137f
13037314
1f89b562
233428af
235697d5
25239412
322e63df
361384ce
37e4aa92
380df715
403a8d1f
43383eb4
45f7c2dd
4671807e
49dd1874
51c6f7ef
51d76abe
53b273c8
56563555
5b392875
5b9f3341
64523cfa
66c1ef42
66f29b89
67b76963
6a698541
6c41e35e
71f9a260
73ba467e
7b6fecd5
7c8f4939
813607cc
8bedcc53
985e3fcb
9d01afb9
a09f6154
a25968f2
a33934eb
a61cc0ef
a674580f
a67ef2e3
a6d156f4
afc9ca6a
b222ab00
b621aeb8
ba7cbdc6
bb170c38
c5e75280
c8dd494
cb66451f
cb69809d
cbe011a6
d03d9c70
d3334ebc
da1ed842
e350fe4f
e7945bc1
f0298c3c
f0e5818a
f1aa21a3
f504a6f4
fff579b7

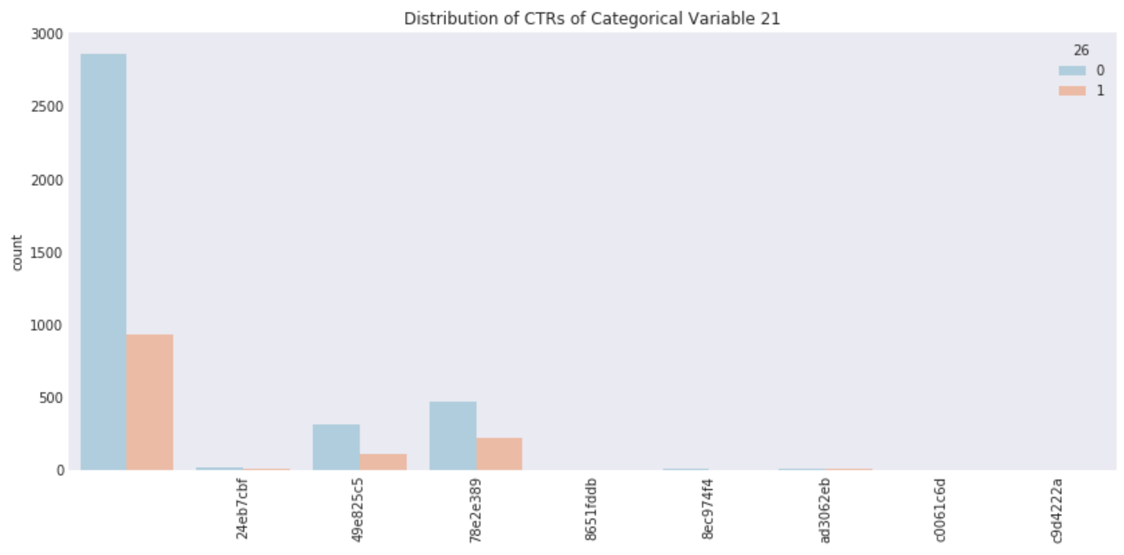

Distribution of Categorical Variable 21
count
3500
3000
2500
2000
1500
1000
500
0
24eb7cbf
49e825c5
78e2e389
8651fddb
8ec974f4
ad3062eb
c0061c6d
c9d4222a

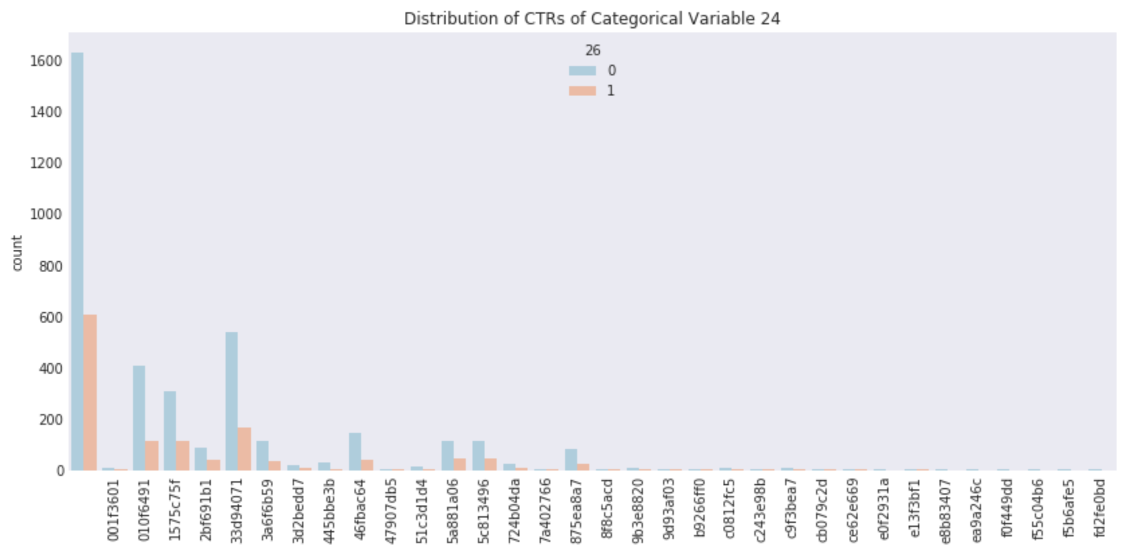


Below, the **outcome** distribution across values is shown for two of these categorical features from the sample. This outcome distribution was also examined to gain a sense of the class balance within each feature.

```
In [13]: for idx in [7, 21, 24]:
             columnName = categorical_sample_df.columns[idx]
             labels = np.unique(categorical_sample_df[columnName].values)
             fig = plt.figure(figsize=(14,6))
             ax = fig.add_subplot(111)
             sns.countplot(x=categorical_sample_df[columnName].values,data=categorical_sample_
             ax.set_xticklabels(labels, rotation=90)
             ax.set_title("Distribution of CTRs of Categorical Variable " + str(columnName))
```



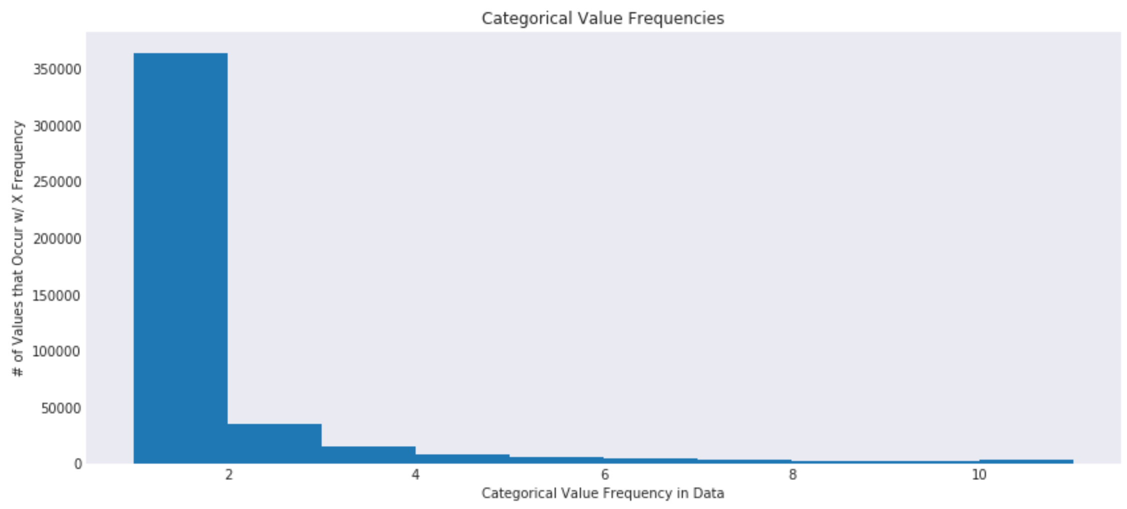




To help illustrate how skewed the categorical distributions are in general, we plot the frequency of each categorical *value* in the sample in a histogram below (the plot is cut-off beyond 11 counts for easier visualization). Note that the frequency distribution shows that the overwhelming majority of categorical values in the data only appear once. The plot below uses the greater 230,000-observation EDA sample.

```
In [14]: catFreq = smallTrainRDD.map(lambda x: x.split('\t')[14:]) \
                                .flatMap(lambda x: [(i,1) for i in x]) \
                                .reduceByKey(lambda x,y: x+y) \
                                .values().collect()
         plt.figure(figsize=(14,6))
```

```
        plt.hist(catFreq, bins=10, range=(1,11))
        plt.xlabel('Categorical Value Frequency in Data')
        plt.ylabel('# of Values that Occur w/ X Frequency')
        plt.title('Categorical Value Frequencies')
        plt.show()
```



4.0.5 *Binarization, Dimensionality and Sparsity*

Below we examine the dimensionality and sparsity of our features to better inform the subsequent choice of dimension reduction technique.

```
In [45]: def avgFeatures(line):
             count = 0
             feats = line.split('\t')[1:]
             for feat in feats:
                 if feat != '':
                     count += 1
             return count

         print("There is an average of", str(round(smallTrainRDD.map(avgFeatures).mean(),2)), '
```

```
There is an average of 33.46 populated features per observation.
```

What is the percentage of NaN values in each numerical feature?

```
In [48]: def parse(line):
             """
             Map record_csv_string --> (tuple,of,fields)
             """
             raw_values = line.split('\t')
```

```
        label = [int(raw_values[0])]
        numerical_values = list(pd.Series(raw_values[1:14]).apply(pd.to_numeric))
        categorical_values = list([str(idx)+"_MISSINGVALUE" if str(value)=="" else str(id
        return(numerical_values + categorical_values + label)
```

```
In [49]: parsed_smallTrainRDDCached = smallTrainRDD.map(parse)
         numericalFeatures = parsed_smallTrainRDDCached.map(lambda x: list(x[:13])).cache()
         nanCounts = numericalFeatures.map(lambda line: 1.0*np.isnan(line)).reduce(lambda x,y:
         nonNanCounts = numericalFeatures.map(lambda line: 1.0*~np.isnan(line)).reduce(lambda 
```

```
In [52]: print("Percentage of NaN values in each numerical feature:")
         print(pd.DataFrame(list(np.round(100*np.divide(nanCounts, np.add(nanCounts, nonNanCou
```

```
Percentage of NaN values in each numerical feature:
     0    1     2     3    4     5    6    7    8     9   10    11    12
0  45.2  0.0  21.6  21.6  2.6  22.3  4.3  0.0  4.3  45.2  4.3  76.6  21.6
```

The percentages listed above indicate that the numerical training data will be very sparse (e.g. >75% of values for the 12th numerical variable (index 11) are missing).

In this sample, the number of distinct values for each category are calculated below to provide a sense of the size of the feature space in the dataset. Evidence of duplicate values is also found below, and confirmed when counting the distinct values.

```
In [43]: total_categories = 0
         all_categorical_values = pd.Series()
         for feature_num in range(len(categorical_sample_df.columns)):
             distinct_values = categorical_sample_df.loc[:,feature_num].unique()
             num_distinct = len(distinct_values)
             all_categorical_values= all_categorical_values.append(pd.Series(distinct_values))
             if '55dd3565' in distinct_values:
                 print("Duplicate of 55dd3565 found in feature",feature_num+1)
             if feature_num < 26:
                 total_categories += num_distinct
                 print("Distinct values in categorical feature", str(feature_num+1) + ":", str
         print("Total categories in sample:", str(total_categories))
```

```
Distinct values in categorical feature 1: 132
Distinct values in categorical feature 2: 339
Distinct values in categorical feature 3: 3240
Distinct values in categorical feature 4: 2443
Distinct values in categorical feature 5: 50
Distinct values in categorical feature 6: 7
Distinct values in categorical feature 7: 2205
Distinct values in categorical feature 8: 73
Distinct values in categorical feature 9: 2
Distinct values in categorical feature 10: 2061
Distinct values in categorical feature 11: 1576
Distinct values in categorical feature 12: 3071
Distinct values in categorical feature 13: 1349
Distinct values in categorical feature 14: 22
Distinct values in categorical feature 15: 1522
Distinct values in categorical feature 16: 2794
Distinct values in categorical feature 17: 9
Distinct values in categorical feature 18: 924
Duplicate of 55dd3565 found in feature 19
Distinct values in categorical feature 19: 349
Distinct values in categorical feature 20: 4
Distinct values in categorical feature 21: 2955
Distinct values in categorical feature 22: 9
Duplicate of 55dd3565 found in feature 23
Distinct values in categorical feature 23: 12
Distinct values in categorical feature 24: 1641
Distinct values in categorical feature 25: 35
Distinct values in categorical feature 26: 1206
Total categories in sample: 28030
```

The presence of duplicate values across these categories was initially detected as shown below-- duplication of categorical values across the columns impacts how one-hot encoding is done for the categorical values.

```
In [41]: all_categorical_values[all_categorical_values.duplicated()]
```

```
Out[41]: 29
         3
         30
         1109    780bcb50
         30
         0
         10      4632bcdc
         266     83552c76
         0
         30
         0
         3       ccfd4002
         0       c7dc6720
         3       55dd3565
         5       423fab69
         27
         0
         0
         dtype: object
```

```
In [42]: all_categorical_values[all_categorical_values=='55dd3565']
```

```
Out[42]: 9    55dd3565
         3    55dd3565
         dtype: object
```

The EDA provided several important observations that inform our modeling decisions:

- Although the numerical features show strong positive skew, their log transforms do not provide nor show clear potential to have improved predictive power and may additionally obfuscate the way 0 values are represented
- High collinearity is not anticipated to be a major problem in this model
- The distributions of numerical features provide guidance on how to bucket them into categories for one-hot encoding
- The count distribution of categorical values is in general highly skewed across a very wide range of values
- Since the meanings of the categorical values are hidden from view, they should be one-hot encoded, suggesting that the feature space will thus be extremely large, even before generating features such as interactive terms
- Some of the categorical values are duplicated across columns, so the encoding needs to take the column or feature number into account
- After one-hot encoding, the data representation will be extremely sparse, suggesting a factorization machines approach to reducing the dimensionality, due to its ability to handle sparse data

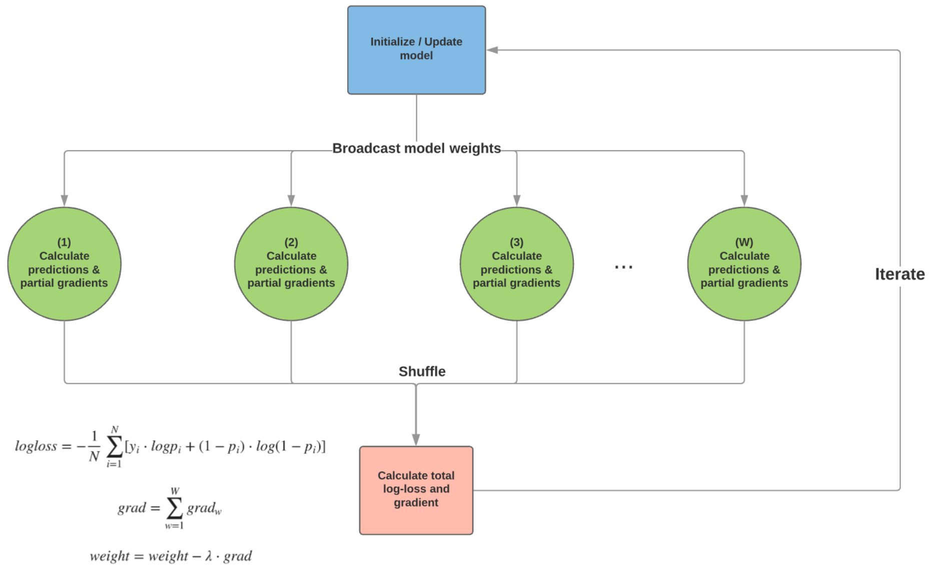
5 4. FM Implementation

In our implementation of this algorithm, we must keep several conditions in mind that will make training on the full dataset possible, which we demonstrate below again using a sample of the training data. In particular, given the extremely wide feature space and high volume of records, we are concerned with sparsely representing our feature vectors, only performing vector and matrix operations on features for which we have populated values, and using caching and broadcasting tactfully such that we do not overload the master or worker nodes, or worse, run out of memory while transferring the data or the parameter objects. The flow of our model training will be as follows, where we initialize our weight parameters, map our model function and loss calculation to the parallelized data across the worker nodes, and then shuffle our gradients back together to reduce to a single, average update for every weight element in the parameter space:

5.0.1 *Pre-Processing*

We begin by parsing our training data, appending the index number of the raw column as a prefix to each value (e.g. categorical variable 12 with value "online" becomes "c12_online"). Similarly, we append "NA" to the end of values that would otherwise be missing, and for numeric variables we bucket them according to the raw distributions examined during our exploratory analysis; this has the added benefit of limiting our feature space and mitigating some risk of overfitting to the training data.

```
In [9]: # function to parse raw data and tag feature values with type and feature indices
        def parseCV(line):
            """
            Map text records to --> (label, features)
            Bucket (string) numeric features
            Add variable index prefix to feature values for binarization
            """
```


Initialize / Update model
Broadcast model weights
(1) Calculate predictions & partial gradients
(2) Calculate predictions & partial gradients
(3) Calculate predictions & partial gradients
...
(W) Calculate predictions & partial gradients
Iterate
Shuffle
Calculate total log-loss and gradient
$logloss = -\frac{1}{N}\sum_{i=1}^{N}[y_i \cdot log p_i + (1-p_i) \cdot log(1-p_i)]$
$grad = \sum_{w=1}^{W} grad_w$
$weight = weight - \lambda \cdot grad$

FM factor

```
# start of categorical features
col_start = 14

raw_values = line.split('\t')
label = int(raw_values[0])

# parse numeric features
numericals = []
for idx, value in enumerate(raw_values[1:col_start]):
    if value == '':
        append_val = 'NA'
    elif value == '0':
        append_val = '0'
    else:
        # continues variables
        if idx in [0,3,6,7]:
            if float(value)<10:
                append_val = '<10'
            elif float(value)<25:
                append_val = '<25'
            else:
                append_val = '>25'
        elif idx in [1,2,5]:
            if float(value)<100:
                append_val = '<100'
            else:
                append_val = '>100'
        elif idx==4:
            if float(value)<10000:
                append_val = '<10k'
            elif float(value)<50000:
                append_val = '<50k'
            else:
                append_val = '>50k'
        elif idx==8:
            if float(value)<100:
                append_val = '<100'
            elif float(value)<500:
                append_val = '<500'
            else:
                append_val = '>500'
        elif idx in [10,11]:
            if float(value)<3:
                append_val = '<3'
            elif float(value)<6:
                append_val = '<6'
            else:
                append_val = '>6'
        elif idx==12:
            if float(value)<5:
                append_val = '<5'
            elif float(value)<10:
                append_val = '<10'
            elif float(value)<25:
                append_val = '<25'
            else:
                append_val = '>25'
        # ordinal/binary cases
        else:
            append_val = str(value)

    numericals.append('n' + str(idx) + '_' + append_val)

# parse categorical features
categories = []
for idx, value in enumerate(raw_values[col_start:]):
    if value == '':
        categories.append('c'+ str(idx) + '_NA')
    else:
        categories.append('c'+ str(idx) + '_' + str(value))

return Row(label=label, raw=numericals + categories)
```

```
In [10]: # call functions
```

```
         parsedDF = smallTrainRDD.map(parseCV).toDF()
```

5.0.2 *One-Hot Encode All Features using CountVectorizer for Sparse Representation*

In order to efficiently move our data beginning in our driver program to our worker nodes, and then perform computations on them, we need to represent the features sparsely, or as vectors containing the indices of the populated feature values. We can generate these features one time on the master node by utilizing MLlib's CountVectorizer class to fit the sparse representation and transform our string features as a new data structure, which can later be added, or reduced following parallelization.

```
In [11]: # function to one hot encode all features using a count vectorizer
         def vectorizeCV(DF):

             vectorizer = CountVectorizer()
             cv = CountVectorizer(minDF=1, inputCol="raw", outputCol="features")

             model = cv.fit(DF)
             result = model.transform(DF)

             return result, model

In [12]: vectorizedDF, cvModel = vectorizeCV(parsedDF)
         vectorizedDF.show(truncate=True)
```

```
+-----+--------------------+--------------------+
|label|                 raw|            features|
+-----+--------------------+--------------------+
|    0|[n0_NA, n1_<100, ...|(25739,[0,1,2,3,4...|
|    0|[n0_NA, n1_<100, ...|(25739,[0,1,2,3,5...|
|    1|[n0_NA, n1_<100, ...|(25739,[0,2,3,4,5...|
|    0|[n0_<10, n1_<100,...|(25739,[0,1,4,5,6...|
|    1|[n0_<10, n1_<100,...|(25739,[0,1,2,3,4...|
|    1|[n0_NA, n1_<100, ...|(25739,[1,2,3,5,6...|
|    0|[n0_<10, n1_<100,...|(25739,[0,1,2,5,6...|
|    0|[n0_NA, n1_<100, ...|(25739,[0,1,2,3,5...|
|    0|[n0_0, n1_>100, n...|(25739,[0,1,3,4,7...|
|    0|[n0_NA, n1_0, n2_...|(25739,[0,2,3,4,8...|
|    0|[n0_0, n1_<100, n...|(25739,[0,1,2,4,5...|
|    0|[n0_<10, n1_<100,...|(25739,[0,1,2,4,5...|
|    0|[n0_NA, n1_>100, ...|(25739,[1,2,3,6,7...|
|    0|[n0_0, n1_>100, n...|(25739,[0,1,4,6,8...|
|    0|[n0_0, n1_<100, n...|(25739,[0,1,2,5,6...|
|    0|[n0_NA, n1_>100, ...|(25739,[1,2,3,6,7...|
|    0|[n0_NA, n1_>100, ...|(25739,[0,1,2,3,7...|
|    0|[n0_NA, n1_<100, ...|(25739,[1,2,3,5,6...|
|    1|[n0_<25, n1_<100,...|(25739,[0,1,2,3,4...|
|    0|[n0_NA, n1_>100, ...|(25739,[0,1,2,3,7...|
+-----+--------------------+--------------------+
only showing top 20 rows
```

```
In [13]: vectorizedRDD = vectorizedDF.select(['label', 'features']).rdd.cache()
```

```
In [15]: num_feats = vectorizedRDD.take(1)[0][1].size
         percent_pos = vectorizedRDD.map(lambda x: x[0]).mean()

         print("Number of total expanded features:", num_feats)
         print("Relative frequency of positive class:", percent_pos)
```

```
Number of total expanded features: 25739
Relative frequency of positive class: 0.25949084412684253
```

5.1 *Predicting the Outcome and Estimating Gradients*

Next, with our improved data structure and our model equation discussed in section 2., we can strategically perform computations such as dot-products *only on the populated feature values and the corresponding elements of our parameter vector and matrix*, i.e. only for those weight vector indices that correspond to our sparse feature indices. Where our data only have an average on the order of 30 populated feature values per row, this is a fractional amount of computation compared to the entire feature set or parameter matrix size.

```
In [17]: def predictGrad(pair, k_br, b_br, w_br, V_br):
             """
                 Compute the predicted probability for average loss AND return the gradients
                 Args:
                     pair - records are in (label, sparse feature set) format
                 Broadcast:
                     b - bias term (scalar)
                     w - linear weight vector (array)
                     k - number of factors (def=2)
                     V - factor matrix of size (d dimensions, k=2 factors)
                 Returns:
                     predRDD - pair of ([label, predicted probability], [set of weight vectors
             """

             label = pair[0]
             feats = pair[1]

             # start with linear weight dot product
             linear_sum = np.dot(w_br.value[0][feats.indices], feats.values)

             # factor matrix interaction sum
             factor_sum = 0.0
             lh_factor = [0.0]*k_br.value
             rh_factor = [0.0]*k_br.value
```

```
        for f in range(0, k_br.value):
            lh_factor[f] = np.dot(V_br.value[f][feats.indices], feats.values)  #KEY--this
            rh_factor[f] = np.dot(V_br.value[f][feats.indices]**2, feats.values**2)
            factor_sum += (lh_factor[f]**2 - rh_factor[f])
        factor_sum = 0.5 * factor_sum

        y_hat = b_br.value + linear_sum + factor_sum    #full model equation

        prob = 1.0 / (1 + np.exp(-y_hat))  #logit transformation

        #compute Gradients
        b_grad = prob - label    # bias term

        #linear term
        w_grad = csr_matrix((b_grad*feats.values, (np.zeros(feats.indices.size), feats.ind

        #factor matrix (d*k)
        v_data = np.array([], dtype=np.float32)
        v_rows = np.array([], dtype=int)
        v_cols = np.array([], dtype=int)
        for i in range(0, k_br.value):
            v_data = np.append(v_data, b_grad*(lh_factor[i]*feats.values - np.multiply(V_
            v_rows = np.append(v_rows, [i]*feats.indices.size)
            v_cols = np.append(v_cols, feats.indices)
        v_grad = csr_matrix((v_data, (v_rows, v_cols)), shape=(k_br.value, V_br.value.sha

        return ([label, prob], [b_grad, w_grad, v_grad])
```

At each step, we calculate our (negative) log-loss for each observation, which will give us a sense of on average how close our predicted probabilities are to their true outcome value (0 or 1).

```
In [18]: def logLoss(pair):
             """parallelize log-loss calculation
                 argument: ([label, prob], [b_grad, w_grad, v_grad])
                 out: -(log-loss)
             """
             y = pair[0][1]

             eps = 1.0e-16
             if pair[0][1] == 0:
                 y_hat = eps
             elif pair[0][1] == 1:
                 y_hat = 1-eps
             else:
                 y_hat = pair[0][1]

             return -(y * np.log(y_hat) + (1-y) * np.log(1-y_hat))
```

Where the probability, gradient and log-loss can all be calculated in parallel, we will then need to shuffle our observations back together and reduce our newly-estimated gradients, which conveniently can be efficiently added even in sparse vector representation.

```
In [ ]: def reduceFct(x, y):
            """function for aggregating bias and weight matrices
                arguments: ([label, pred], [bias, weight, V matrix])
                out:       [sum bias b, sum weight w, sum matrix V]
            """
            b = x[0] + y[0]
            w = x[1] + y[1]
            V = x[2] + y[2]
            return [b, w, V]
```

5.1.1 *Model Training*

Now that we have our key map and reduce functions, we can iteratively estimate our gradients and update our weight parameter estimates until we achieve diminishing returns to our average log-loss. To do this, however, we will need to broadcast our weight data structures to each node, as well as keep our primary RDD containing features and labels cached in worker memory, in order to minimize shuffling; however, given the size of our weight vectors and full dataset, this will be memory-intensive and ultimately require computation on a cluster.

```
In [20]: def iterateSGD(dataRDD, k, bInit, wInit, vInit, nIter = 2, learningRate = 0.1, useReg

             k_br = sc.broadcast(k)
             b_br = sc.broadcast(bInit)
             w_br = sc.broadcast(wInit)
             V_br = sc.broadcast(vInit)

             losses = []
             N = dataRDD.count()

             for i in range(0, nIter):
                 print('-' * 25 + 'Iteration ' + str(i+1) + '-' * 25)
                 predRDD = dataRDD.map(lambda x: predictGrad(x, k_br, b_br, w_br, V_br)).cache

                 loss = predRDD.map(logLoss).reduce(lambda a,b: a+b)/N + \
                         int(useReg)*(regParam/2)*(np.linalg.norm(w_br.value)**2 + np.linalg.no
                 losses.append(loss)
                 print(f'Current log-loss: {loss}')

                 # reduce step
                 gradRDD = predRDD.values().reduce(reduceFct)
                 bGrad = gradRDD[0]/N
                 wGrad = gradRDD[1]/N
                 vGrad = gradRDD[2]/N
```

```
            print(f"Bias: {bGrad}")
            print(f"wGrad shape: {wGrad.shape}")
            print(f"vGrad shape: {vGrad.shape}")

            ############## update weights ##############
            # first, unpersist broadcasts
            predRDD.unpersist()
            b_br.unpersist()
            w_br.unpersist()
            V_br.unpersist()

            # update and re-broadcast
            b_br = sc.broadcast(b_br.value - learningRate * bGrad)
            w_br = sc.broadcast(w_br.value - learningRate * (wGrad.toarray()+int(useReg)*
            V_br = sc.broadcast(V_br.value - learningRate * (vGrad.toarray()+int(useReg)*

        return losses, b_br, w_br, V_br
```

Below is again a sample of the data but run in a scaled manner, where model parameters are initialized and broadcast, the training loop is set to a fixed number of iterations, and there is no regularization. Here, k is the order of variable interactions we wish the model to consider, which is a tunable parameter, where the run-time is a function of k; a lower k may obscure possible interactions, but in a sparse setting greatly reduces the likelihood of the model overfitting. This model can accomodate L2 regularization, where the penalty is a function of lambda and the magnitude of our weight vectors; however, preliminary testing showed that this penalty did not help performance on the validation set where our k is already conservative, and was thus excluded from the full run. We also note that including the L2 regularization computation did not seem to substantially increase runtime.

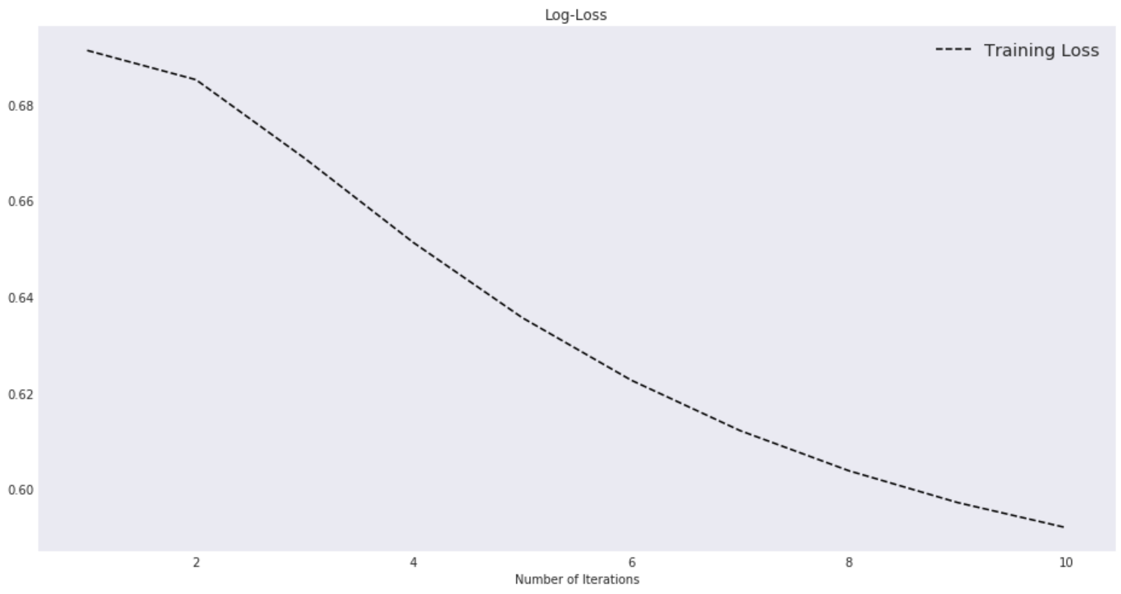
```
In [25]: # initialize weights
         np.random.seed(24)
         k = 2
         b = 0.0
         w = np.random.normal(0.0, 0.02, (1, num_feats))
         V = np.random.normal(0.0, 0.02, (k, num_feats))

         # train sample model
         nIter = 10
         start = time.time()
         losses, b_br, w_br, V_br = iterateSGD(vectorizedRDD, k, b, w, V, nIter)
         print(f'Performed {nIter} iterations in {time.time() - start} seconds')
```

```
-------------------------Iteration 1-------------------------
Current log-loss: 0.6912273009710425
Bias: 0.25436576408125167
wGrad shape: (1, 25739)
vGrad shape: (2, 25739)
-------------------------Iteration 2-------------------------
```

```
Current log-loss: 0.6851338435935052
Bias: 0.18375948102046125
wGrad shape: (1, 25739)
vGrad shape: (2, 25739)
-------------------------Iteration 3------------------------
Current log-loss: 0.6687666092907115
Bias: 0.13441639427762675
wGrad shape: (1, 25739)
vGrad shape: (2, 25739)
-------------------------Iteration 4------------------------
Current log-loss: 0.6512226506271765
Bias: 0.09992019607042654
wGrad shape: (1, 25739)
vGrad shape: (2, 25739)
-------------------------Iteration 5------------------------
Current log-loss: 0.6356066208636996
Bias: 0.07543655596140342
wGrad shape: (1, 25739)
vGrad shape: (2, 25739)
-------------------------Iteration 6------------------------
Current log-loss: 0.622623423080275
Bias: 0.05774713712429709
wGrad shape: (1, 25739)
vGrad shape: (2, 25739)
-------------------------Iteration 7------------------------
Current log-loss: 0.6121412223222497
Bias: 0.04475662909105663
wGrad shape: (1, 25739)
vGrad shape: (2, 25739)
-------------------------Iteration 8------------------------
Current log-loss: 0.6037891852325022
Bias: 0.03508540602284818
wGrad shape: (1, 25739)
vGrad shape: (2, 25739)
-------------------------Iteration 9------------------------
Current log-loss: 0.5971714200881342
Bias: 0.02780504477504894
wGrad shape: (1, 25739)
vGrad shape: (2, 25739)
-------------------------Iteration 10------------------------
Current log-loss: 0.5919359790827723
Bias: 0.022275786379377382
wGrad shape: (1, 25739)
vGrad shape: (2, 25739)
Performed 10 iterations in 404.480101108551 seconds
```

```
In [27]: fig, ax = plt.subplots(1,1,figsize = (16,8))
```

```
        x = list(range(1, len(losses)+1))
        ax.plot(x, losses, 'k--', label='Training Loss')
        ax.legend(loc='upper right', fontsize='x-large')
        plt.xlabel('Number of Iterations')
        plt.title("Log-Loss")
        plt.show()
```



5.2 *Sample Evaluation of Holdout Development Set*

To confirm that we are not overfitting our model to our sample training set, we next calculate the log-loss on a holdout set of labeled data, using the same CountVectorizer transformation and FM model equation to estimate our $\hat{p}$.

```
In [28]: def predictProb(pair, k_br, b_br, w_br, V_br):
             """
                 Compute the predicted probability AND return the gradient (?)
                 Args:
                     pair - records are in (label, sparse feature set) format
                 Broadcast:
                     b - bias term (scalar)
                     w - linear weight vector (array)
                     k - number of factors (def=2)
                     V - factor matrix of size (d dimensions, k=2 factors)
                 Returns:
                     predRDD - pair of (label, predicted probability)
             """

             label = pair[0]
```

```
            feats = pair[1]

            # start with linear weight dot product
            linear_sum = np.dot(w_br.value[0][feats.indices], feats.values)

            # factor matrix interaction sum
            factor_sum = 0.0
            lh_factor = [0.0]*k_br.value
            rh_factor = [0.0]*k_br.value

            for f in range(0, k_br.value):
                lh_factor[f] = np.dot(V_br.value[f][feats.indices], feats.values)
                rh_factor[f] = np.dot(V_br.value[f][feats.indices]**2, feats.values**2)
                factor_sum += (lh_factor[f]**2 - rh_factor[f])
            factor_sum = 0.5 * factor_sum

            pre_prob = b_br.value + linear_sum + factor_sum

            prob = 1.0 / (1 + np.exp(-pre_prob))  #logit transformation

            return (label, prob)
```

```
In [29]: def testLoss(pair):
            """parallelize log loss
                input: (label, prob)
            """
            y = pair[0]

            eps = 1.0e-16
            if pair[1] == 0:
                y_hat = eps
            elif pair[1] == 1:
                y_hat = 1-eps
            else:
                y_hat = pair[1]

            return -(y * np.log(y_hat) + (1-y) * np.log(1-y_hat))
```

```
In [30]: k_br = sc.broadcast(k)

         parsedTestDF = smallTestRDD.map(parseCV).toDF()
         vectorizedTestDF = cvModel.transform(parsedTestDF)
         testLoss = vectorizedTestDF.select(['label', 'features']).rdd \
                                            .map(lambda x: predictProb(x, k_br, b_br, w_br, V
                                            .map(testLoss).mean()

         print("Log-loss on the hold-out development set is:", testLoss)
```

```
Log-loss on the hold-out development set is: 0.5588072612641066
```

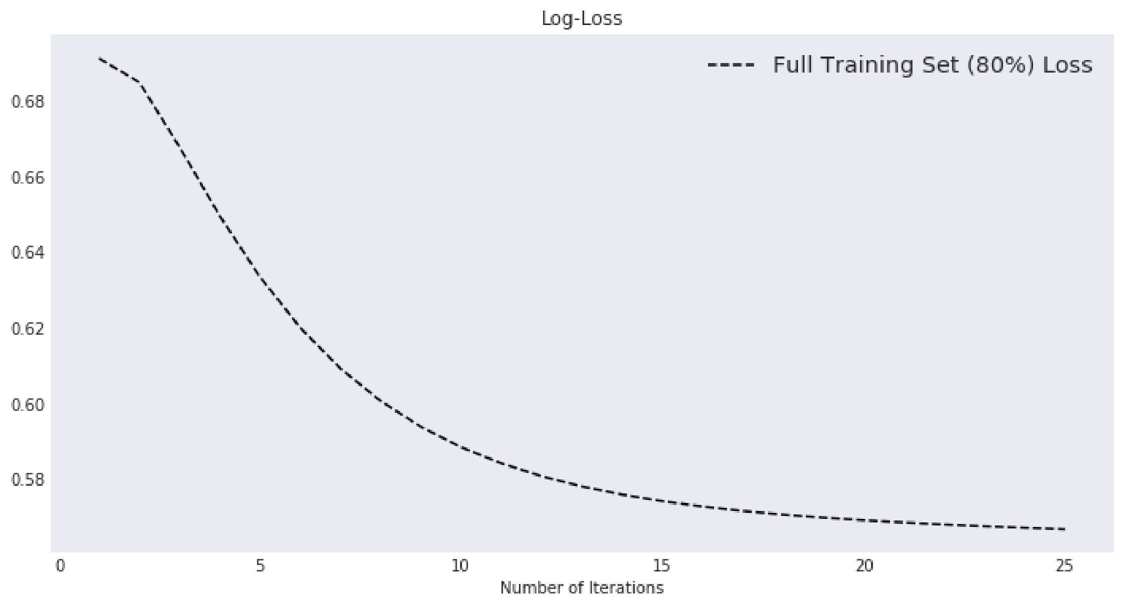
5.3 *Full Model Training on GCP Cluster*

The previous code was inserted into a pyspark file for submission to a GCP dataproc cluster, named `fm_on_cluster.py`. We used helper code provided by the instructors, `submit_job_to_cluster.py`, to specify the cluster creation and submit the pyspark job, and we additionally modified the code to allow the specification of different machine types for the master and the workers.

The results shown below are for a dataproc training job using a random sample of 80% of the `training.txt` dataset over 25 learning iterations, and evaluated on the remaining 20% of held-out user observations. This split contains close to 37 million training observations (plus 7 million in the holdout set) and 262,000 binarized features. Accordingly, our parameters are our scalar bias term `b`, linear weight vector w, and $k = 2$-wide factor matrix $\mathbf{V}$.

```
In [1]: fullLoss_txt = open("FINAL/results/results_train_loss.txt", "r")
        fullLoss_list = fullLoss_txt.read().split('\t')
        fullLoss = [float(loss) for loss in fullLoss_list[:-1]]
```

```
In [3]: fig, ax = plt.subplots(1,1,figsize = (12,6))
        x = list(range(1,len(fullLoss)+1))
        ax.plot(x, fullLoss, 'k--', label='Full Training Set (80%) Loss')
        ax.legend(loc='upper right', fontsize='x-large')
        plt.xlabel('Number of Iterations')
        plt.title("Log-Loss")
        plt.show()
```



Over the first ten iterations, we see rapid improvement in our average log-loss for our training set over the first ten iterations, with a leveling out thereafter. On our holdout set, we estimate an average loss on a similar order but slightly lower, which suggests our model could benefit from generating additional features, as the current is underfit.

```
In [4]: testLoss_txt = open("FINAL/results/results_test_loss.txt", "r")
        testLoss = float(testLoss_txt.read())
        print("Final loss on our holdout data of the train.txt set:", testLoss)
```

```
Final loss on our holdout data of the train.txt set: 0.548992525753779
```

5.4 *Score Unlabeled Test Data*

As an extension of our holdout set accuracy evaluation, we score the unlabeled `test.txt` data using the model trained on the cluster. This requires again first transforming our raw features using the save CV model:

```
In [31]: unlabeledRDD = sc.textFile('data/test.txt')
         scoreTest = 0.001
         smallUnlabeledRDD = unlabeledRDD.sample(False, scoreTest)
```

```
In [32]: parsedUnlabeledDF = smallUnlabeledRDD.map(lambda x: "0\t"+x).map(parseCV).toDF()
         vectorUnlabeledDF = cvModel.transform(parsedUnlabeledDF)
         vectorUnlabeledDF.show(truncate=True)
```

```
+-----+--------------------+--------------------+
|label|                 raw|            features|
+-----+--------------------+--------------------+
|    0|[n0_<10, n1_<100,...|(25739,[0,1,4,5,6...|
|    0|[n0_NA, n1_<100, ...|(25739,[0,1,2,3,4...|
|    0|[n0_0, n1_0, n2_<...|(25739,[0,1,2,3,4...|
|    0|[n0_NA, n1_<100, ...|(25739,[3,5,6,8,1...|
|    0|[n0_0, n1_<100, n...|(25739,[0,1,2,3,4...|
|    0|[n0_<10, n1_0, n2...|(25739,[0,1,3,4,8...|
|    0|[n0_NA, n1_<100, ...|(25739,[0,1,3,5,6...|
|    0|[n0_0, n1_<100, n...|(25739,[0,2,4,5,1...|
|    0|[n0_NA, n1_<100, ...|(25739,[0,2,3,4,5...|
|    0|[n0_NA, n1_>100, ...|(25739,[1,2,3,6,1...|
|    0|[n0_0, n1_0, n2_>...|(25739,[0,1,4,6,9...|
|    0|[n0_<10, n1_<100,...|(25739,[0,1,2,3,4...|
|    0|[n0_NA, n1_<100, ...|(25739,[1,2,5,6,7...|
|    0|[n0_NA, n1_0, n2_...|(25739,[1,2,3,7,9...|
|    0|[n0_NA, n1_<100, ...|(25739,[0,1,2,3,4...|
|    0|[n0_<10, n1_<100,...|(25739,[0,1,2,3,4...|
|    0|[n0_<25, n1_<100,...|(25739,[0,1,3,4,5...|
|    0|[n0_<10, n1_<100,...|(25739,[0,2,3,4,5...|
|    0|[n0_NA, n1_<100, ...|(25739,[0,1,2,3,4...|
|    0|[n0_<10, n1_0, n2...|(25739,[0,1,2,4,7...|
+-----+--------------------+--------------------+
only showing top 20 rows
```

Using our full model trained and validated with the labeled data on our cluster, we evaluate the robustness of our model's scoring, given what we know and expect about the distribution of our CTR outcome variable.

```
In [6]: import pandas as pd
        unlabeledScores = pd.read_csv("FINAL/results/results_test_predictions.csv", names = ["
        unlabeledScores.head(10)
```

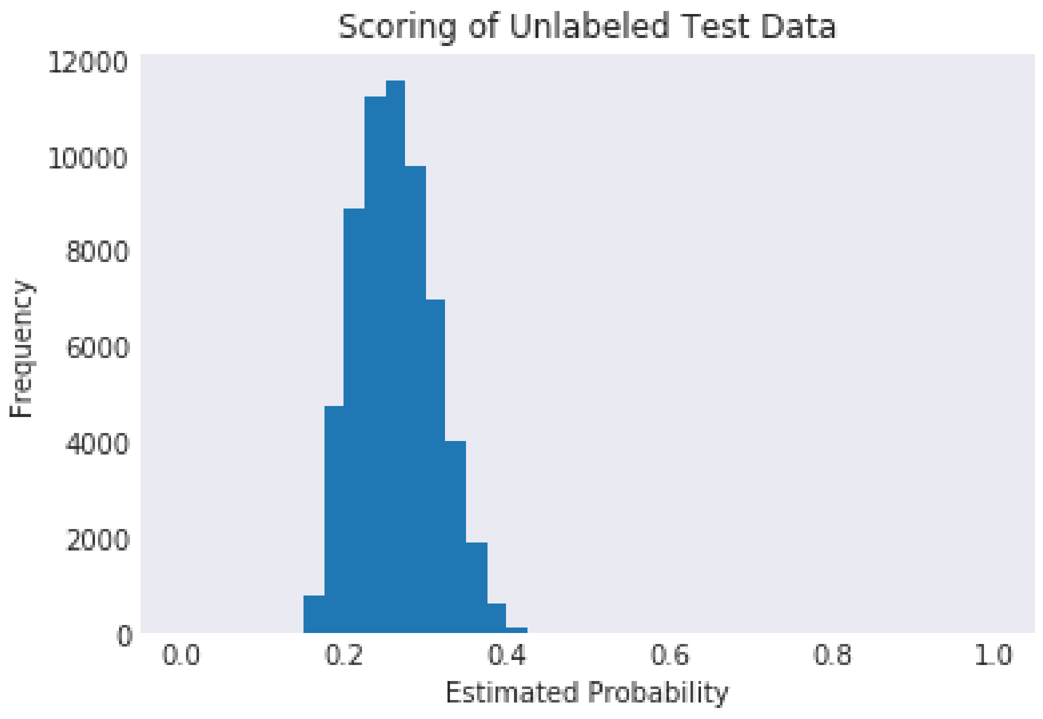
```
Out[6]:                                One-hot encoded features     Score
        0  ['n0_<25', 'n1_<100', 'n2_<100', 'n3_<10', 'n4...  0.322991
        1  ['n0_0', 'n1_<100', 'n2_<100', 'n3_<25', 'n4_<...  0.240020
        2  ['n0_NA', 'n1_<100', 'n2_<100', 'n3_<25', 'n4_...  0.183301
        3  ['n0_<10', 'n1_<100', 'n2_>100', 'n3_NA', 'n4_...  0.240142
        4  ['n0_NA', 'n1_0', 'n2_<100', 'n3_<10', 'n4_<10...  0.205011
        5  ['n0_0', 'n1_<100', 'n2_<100', 'n3_>25', 'n4_<...  0.272889
        6  ['n0_<10', 'n1_<100', 'n2_<100', 'n3_<10', 'n4...  0.279814
        7  ['n0_<10', 'n1_<100', 'n2_<100', 'n3_<10', 'n4...  0.287328
        8  ['n0_0', 'n1_0', 'n2_<100', 'n3_NA', 'n4_<50k'...  0.231372
        9  ['n0_<10', 'n1_<100', 'n2_<100', 'n3_<10', 'n4...  0.272297
```

```
In [7]: unlabeledScores.shape
```

```
Out[7]: (60738, 2)
```

Looking at a randomly drawn sample of roughly 60,000 user observations from the unlabeled test set, we see a narrow distribution of predicted probabilities centered around our mean training sample CTR. This suggests that although our model improved in its log-loss after each iteration, the trained model is lacking strong features and possibly more latent factors that could represent additional interactions between independent variables. There is a large opportunity for improvement toward practical accuracy from additional feature engineering.

```
In [9]: plt.hist(unlabeledScores["Score"], range=(0,1), bins=40)
        plt.xlabel('Estimated Probability')
        plt.ylabel('Frequency')
        plt.title('Scoring of Unlabeled Test Data')
        plt.show()
```



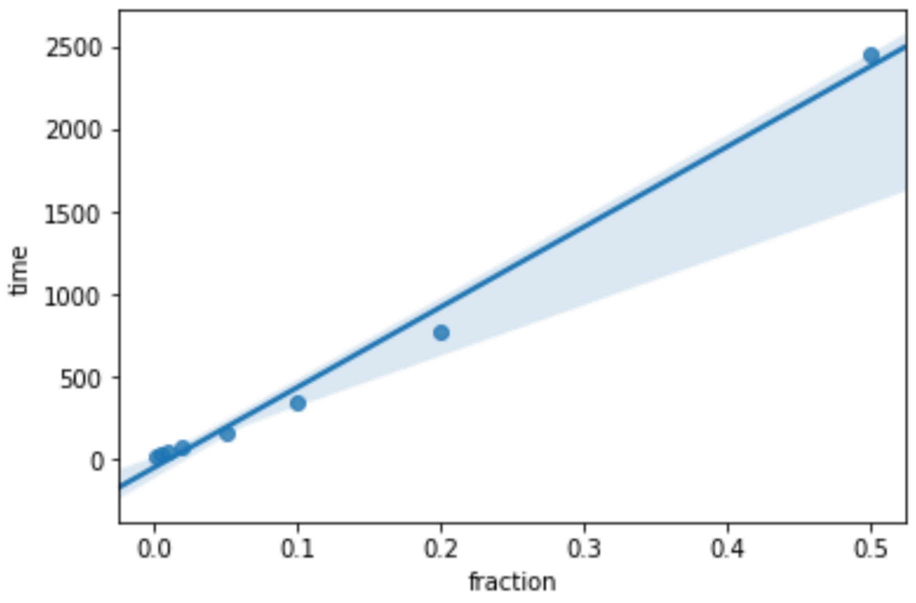
5.5 *Training Time at Scale*

Evaluating our run-time using different training sample proportions, we see strong evidence that we have achieved roughly linear run-time purely as a function of the fraction of training data used when using a parameter of $k = 2$. This is encouraging because even as the size of our one-hot encoded feature space is also growing (and thus the number of our order=2 feature interactions), we are not led to an exponential increase in run-time. This suggests that we have achieved scalability under the wide, sparse feature problem, and if we were to increase the number of latent factors k, we would expect only an increase in run-time by a factor of each additional k; for example, if we went from two to four latent factors, we expect twice as many computations per row in our mapper, and thus a doubling of run-time. Note that the regression line shows that the training time is not strictly linear, but is clearly not growing exponentially. We suspect that there may be additional overhead (e.g., time required to broadcast larger and larger model weights) that are increasing the training time with more and more data.

6 5. Key Concepts of Machine Learning at Scale

The direction of this project was determined by several core concepts from this course:

- functional programming / higher order functions / map reduce paradigm
- associative / commutative operations
- broadcasting / caching
- sparse representation (pairs vs stripes)



FM factor

- one-hot encoding / vector embeddings
- normalization

The model presented here takes advantage of the fact that gradient descent is parallelizable. Calculating the gradient for each example in the training data depends only on the broadcast model weights and factor matrix, and not on information captured by other examples. The final gradient is an average of these estimates across all examples. Since the first step of taking this average is to sum all gradient calculations together, the preceding individual gradient calculations are not impacted by the order in which gradients are calculated (making them commutative operations) and do not depend on which partitions calculate which gradients (associative), making this task embarrassingly parallel. To take advantage of parallelization, we utilize the functional programming paradigm of map-reduce and Spark, such as using the higher-order `map` function of RDDs to estimate the new gradient of each example. From a technical standpoint, the function depends only on the example and the broadcast model variables passed to it, although in our case those broadcast variables are sizeable data structures. But we do note that from a conceptual standpoint, usage of the broadcast variables is a departure from the 'statelessness' of the functional programming paradigm. Distributing these parallel computations across a cluster allows true gradient descent to be accomplished in significantly reduced execution time, thus eliminating the need to approximate gradients with an approach such as stochastic gradient descent.

As mentioned previously, broadcasting the weight matrix and factor matrix allows every executor in the cluster to perform the same gradient calculation with its partition of data. However, both full model parameters have hundreds of thousands of elements, so broadcasting these was a concern in terms of both data transfer and memory usage of the executors. Since high memory usage of these broadcasts was anticipated, caching of RDDs was only used tactfully in order to minimize memory usage. Two RDDs are accessed multiple times during training iterations (such as the one-hot encoded `vectorizedRDD`) and are therefore cached in order to avoid their recalculation. This avoids duplication of RDD calculations, and is used in accordance with `RDD.unpersist()`

where appropriate to release these RDDs and broadcast variables from memory when they are no longer needed. Careful caching and unpersisting permitted the high memory demand of large broadcasts while eliminating unnecessary calculations.

The model makes heavy use of vector embeddings. The original dataset started very close to a "stripes" format-- the label (outcome) of an example is accompanied by an array of features associated with that label. `CountVectorizer` is used to convert this dense representation into a sparse vector representation of the data, in which all feature values become one-hot encoded. The columns represent each unique feature value, and each row is a web user, where an entry in the feature matrix takes value 1 if that example contains the feature in that column, and 0 if not. Given the enormous number of categorical values throughout the data, as well as the fact that their meanings (i.e. ordering) are unknown, the one-hot encoding allows a numerical representation of the data that can be efficiently processed by many machine learning algorithms (e.g. similarity between examples can be calculated). The factorization machines method also creates a vector representation of the strength of each variable (w), as well as the single and pairwise interactions between features (V, when $k = 2$). Because of the extremely large but sparse feature space, the matrix representation of interactions V, while large and typically a significant drag on computation time, is only selectively used to perform operations with the *populated feature values* of x. As a result, we gain substantial computational benefit from representing our features with compressed sparse row vectors, which also have the benefit of being reducible when determining each training iteration's gradient.

Early on in the process of transforming the data, we attempted to normalize the values to the (0, 1) range to expedite convergence on gradient descent and regularization (which are both more efficient when features are on the same scale). In order to handle the missing values, the columns needed to be transposed to keys, bringing the data representation back to a 'pairs' approach, and we considered this to be too inefficient for processing. Furthermore, the EDA showed that a) most numerical variables were highly skewed towards zero across a large range (values close to zero would be hard to discern when scaling down), and b) a log transform could misconstrue the way zero values are represented. The numerical data was instead bucketed into categories, which simplified the one-hot encoded representation of the data. One-hot encoding the buckets of numerical variables was considered a sufficient representation of those variables that allows proper gradient descent and regularization while also simplifying the dataset.

Understanding the above concepts is crucial to determining the data representations, model choices, and algorithm design that result in a machine learning application that can scale well. The model created effectively analyzes a week's worth of data in about a day, meaning it could be put into production for Criteo's regular use with a computation cost of only one hundred dollars a week or just over five thousand dollars each year. A few future improvements to increase the model accuracy could include parameter tuning the size of the interaction matrix, normalizing the numeric features, and leveraging libraries such as Glint to more efficiently deal with sharing and updating large model weights and vectors across many worker nodes.