W261 Final Project

Display Ads Challenge

Andrew Carlson
Mark Gin
Michelle Liu
Kathleen Wang

Our Goals

- **Display advertising** - a billion dollar effort to target potential customers most effectively
- **Kaggle competition** - CriteoLabs’s data on the user and the page he is visiting to develop models predicting ad click-through rate (CTR)
- **Goal** - beat 75% accuracy

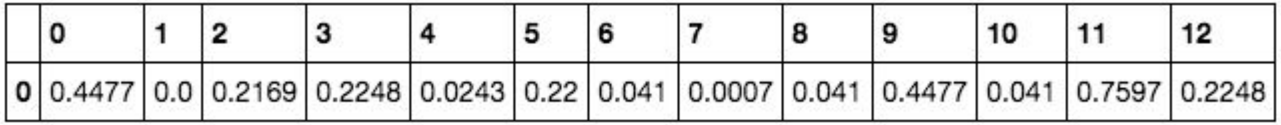
Exploratory Data Analysis

1. Data quality and distribution
    - Data labels indicate if the user click on the ad
    - Columns 1-13 are numerical
    - Columns 14-26 are categorical (string) data
    - All data are anonymized
2. Statistics of the most prevalent instances of categorical data
3. Systematic differences in features per label

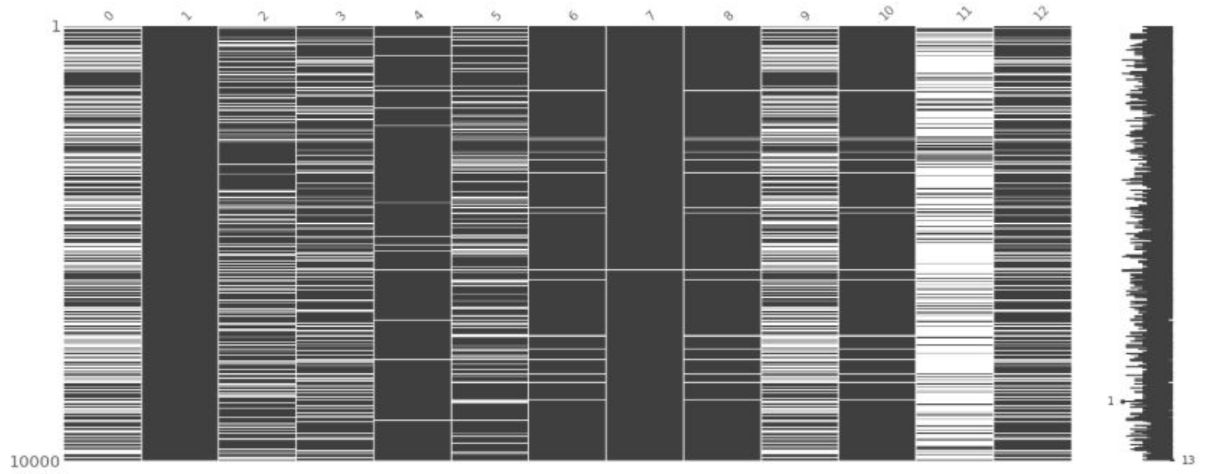
EDA: Numeric Columns (1/4)

1. Fraction of missing values for each columns:

1.

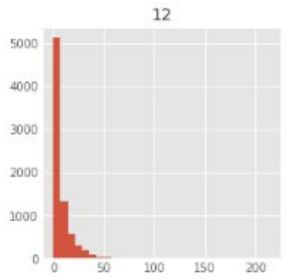


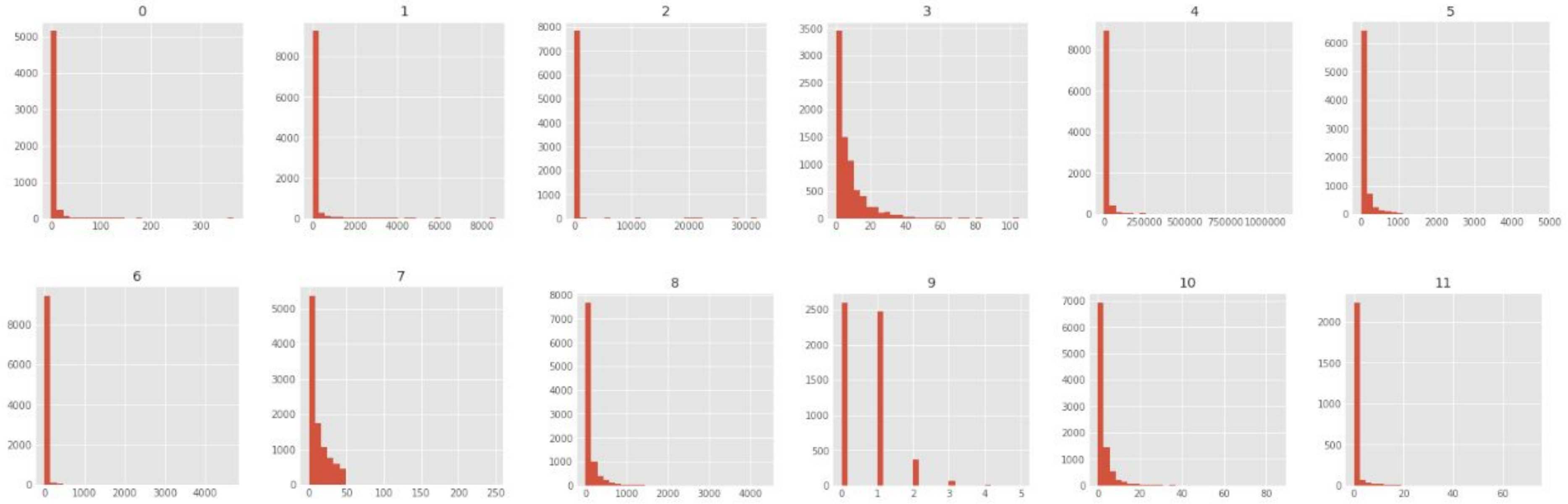
	0	1	2	3	4	5	6	7	8	9	10	11	12
0	0.4477	0.0	0.2169	0.2248	0.0243	0.22	0.041	0.0007	0.041	0.4477	0.041	0.7597	0.2248



EDA: Numeric Columns (2/4)

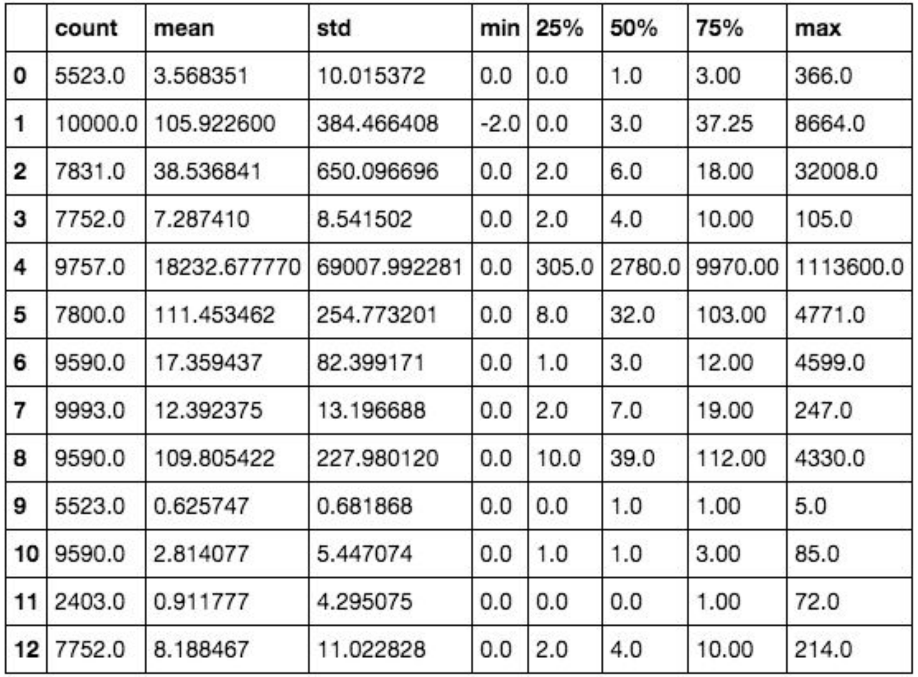
1. All the numeric features are very right-skewed





EDA: Numeric Columns (3/4)

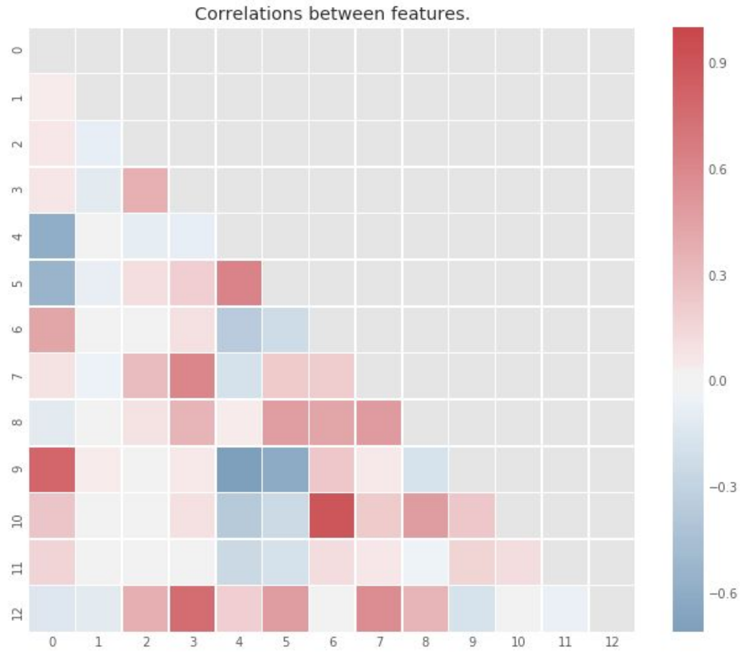
1. Aggregate statistics to check for anomalies



	count	mean	std	min	25%	50%	75%	max
0	5523.0	3.568351	10.015372	0.0	0.0	1.0	3.00	366.0
1	10000.0	105.922600	384.466408	-2.0	0.0	3.0	37.25	8664.0
2	7831.0	38.536841	650.096696	0.0	2.0	6.0	18.00	32008.0
3	7752.0	7.287410	8.541502	0.0	2.0	4.0	10.00	105.0
4	9757.0	18232.677770	69007.992281	0.0	305.0	2780.0	9970.00	1113600.0
5	7800.0	111.453462	254.773201	0.0	8.0	32.0	103.00	4771.0
6	9590.0	17.359437	82.399171	0.0	1.0	3.0	12.00	4599.0
7	9993.0	12.392375	13.196688	0.0	2.0	7.0	19.00	247.0
8	9590.0	109.805422	227.980120	0.0	10.0	39.0	112.00	4330.0
9	5523.0	0.625747	0.681868	0.0	0.0	1.0	1.00	5.0
10	9590.0	2.814077	5.447074	0.0	1.0	1.0	3.00	85.0
11	2403.0	0.911777	4.295075	0.0	0.0	0.0	1.00	72.0
12	7752.0	8.188467	11.022828	0.0	2.0	4.0	10.00	214.0

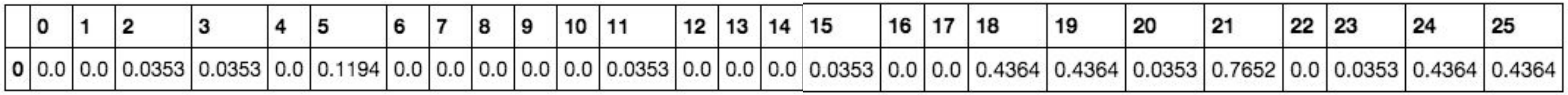
EDA: Numeric Columns (4/4)

1. Few features seem to be correlated with each other

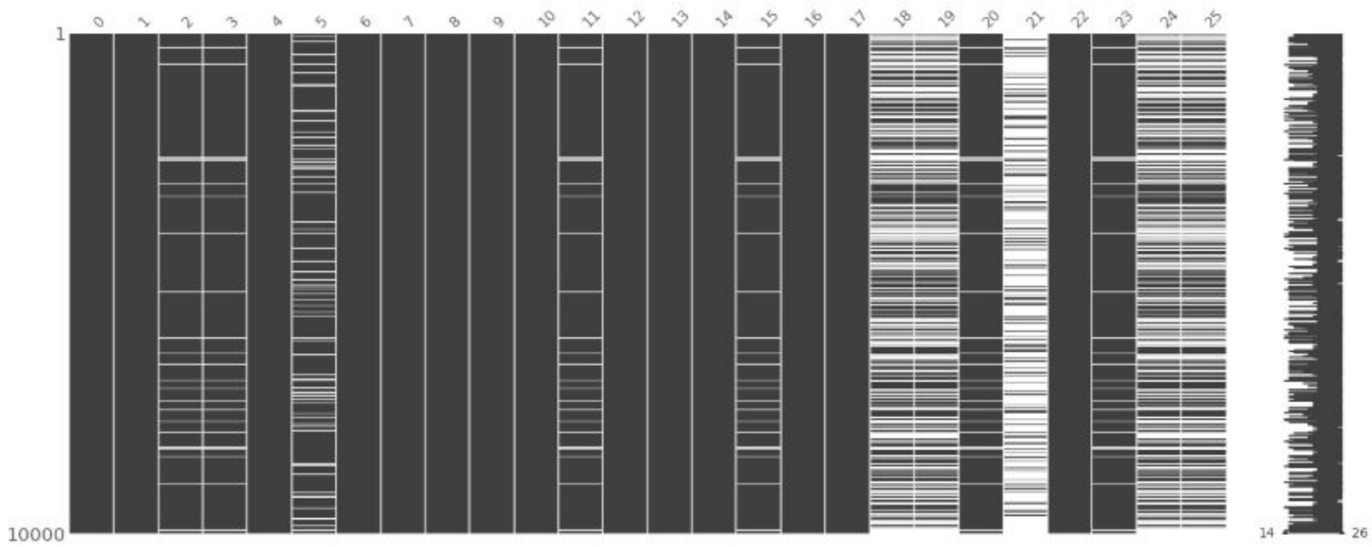


EDA: Categorical Columns (1/4)

1. More complete, missing values are more correlated

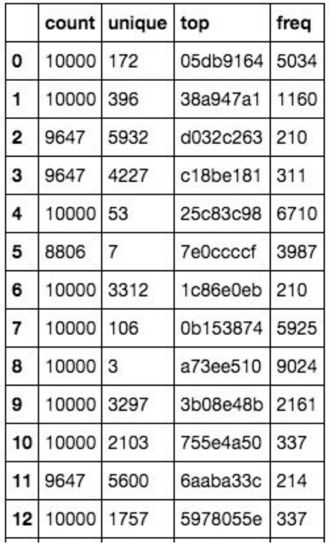
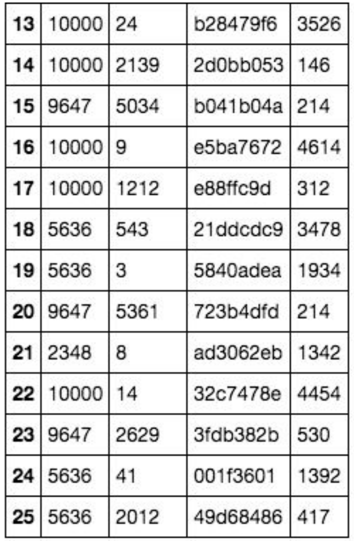


	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25
0	0.0	0.0	0.0353	0.0353	0.0	0.1194	0.0	0.0	0.0	0.0	0.0	0.0353	0.0	0.0	0.0	0.0353	0.0	0.0	0.4364	0.4364	0.0353	0.7652	0.0	0.0353	0.4364	0.4364

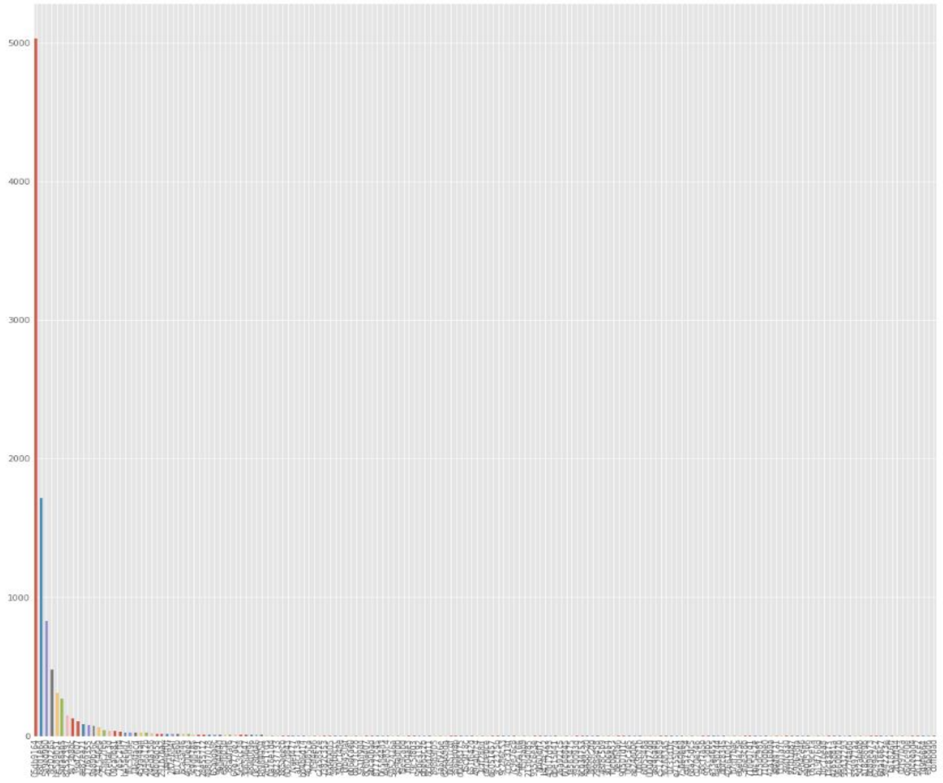


EDA: Categorical Columns (2/4)

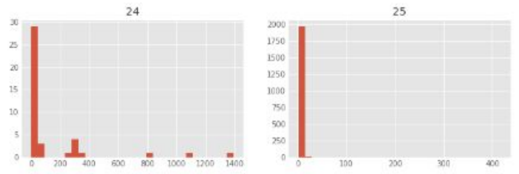
1. Large numbers of rare categories across all categorical columns
2. Top most common category is sometimes relatively rare itself

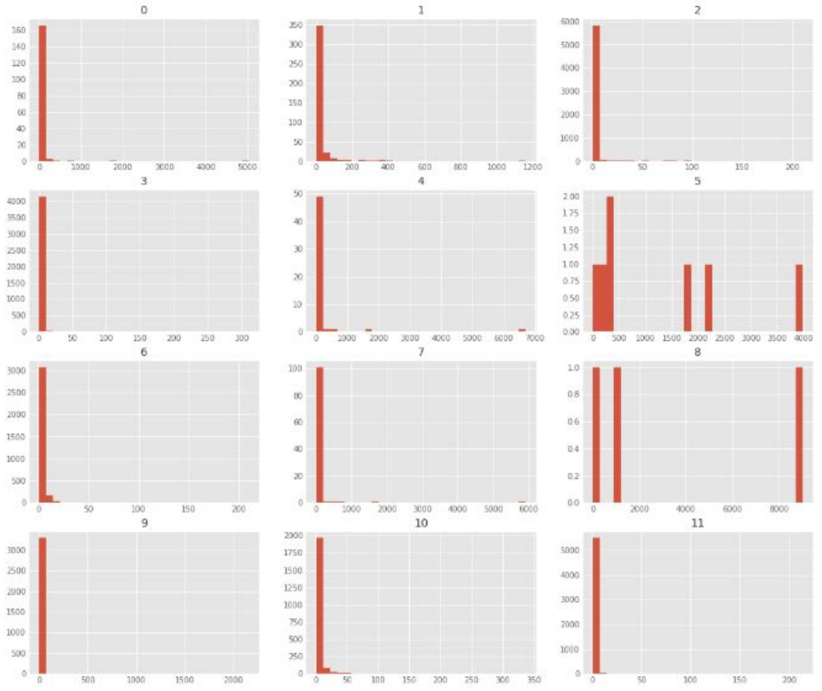
	count	unique	top	freq
0	10000	172	05db9164	5034
1	10000	396	38a947a1	1160
2	9647	5932	d032c263	210
3	9647	4227	c18be181	311
4	10000	53	25c83c98	6710
5	8806	7	7e0ccccf	3987
6	10000	3312	1c86e0eb	210
7	10000	106	0b153874	5925
8	10000	3	a73ee510	9024
9	10000	3297	3b08e48b	2161
10	10000	2103	755e4a50	337
11	9647	5600	6aaba33c	214
12	10000	1757	5978055e	337
13	10000	24	b28479f6	3526
14	10000	2139	2d0bb053	146
15	9647	5034	b041b04a	214
16	10000	9	e5ba7672	4614
17	10000	1212	e88ffc9d	312
18	5636	543	21ddcdc9	3478
19	5636	3	5840adea	1934
20	9647	5361	723b4dfd	214
21	2348	8	ad3062eb	1342
22	10000	14	32c7478e	4454
23	9647	2629	3fdb382b	530
24	5636	41	001f3601	1392
25	5636	2012	49d68486	417

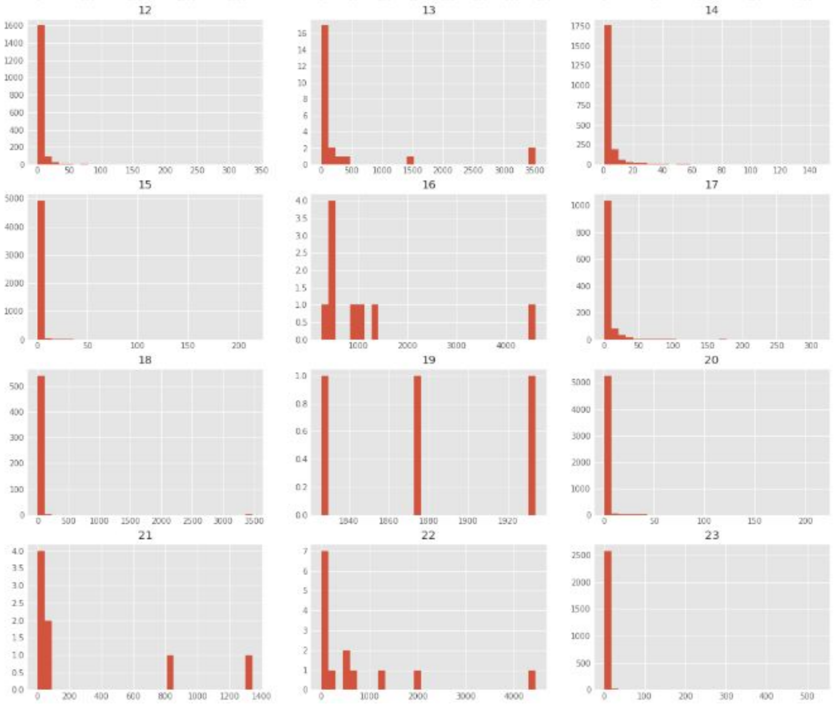


EDA: Categorical Columns (3/4)



1. Histograms show how common the rare-category problem is across all features



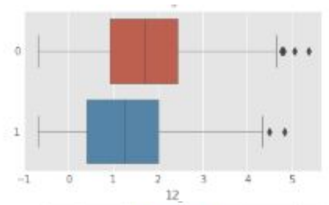


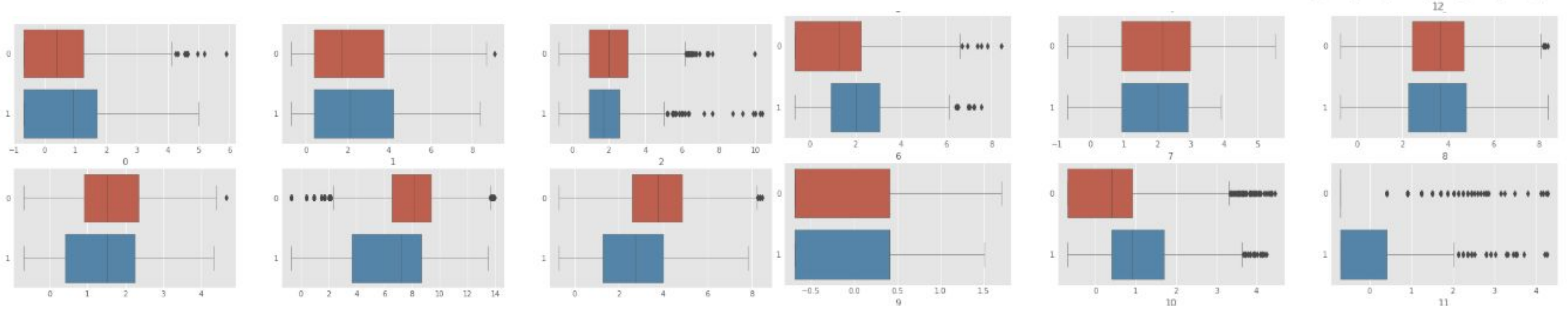
EDA: Categorical Columns (4/4)

1. We identify the 50 most frequent categories, everything outside of that is replaced with the value *“RARE”*

EDA: Correlation of Labels & Features

1. We check if the values are distributed differently within each class label
   - Doesn't seem to be much systematic differences across classes
2. For categorical features, we use a χ^2 test to check if the distribution of categories are different per class
   - Most p-values are very close to 0: there are differences in the distribution
   - Skeptical since the χ^2 test reject the H_0 too easily when the count is small





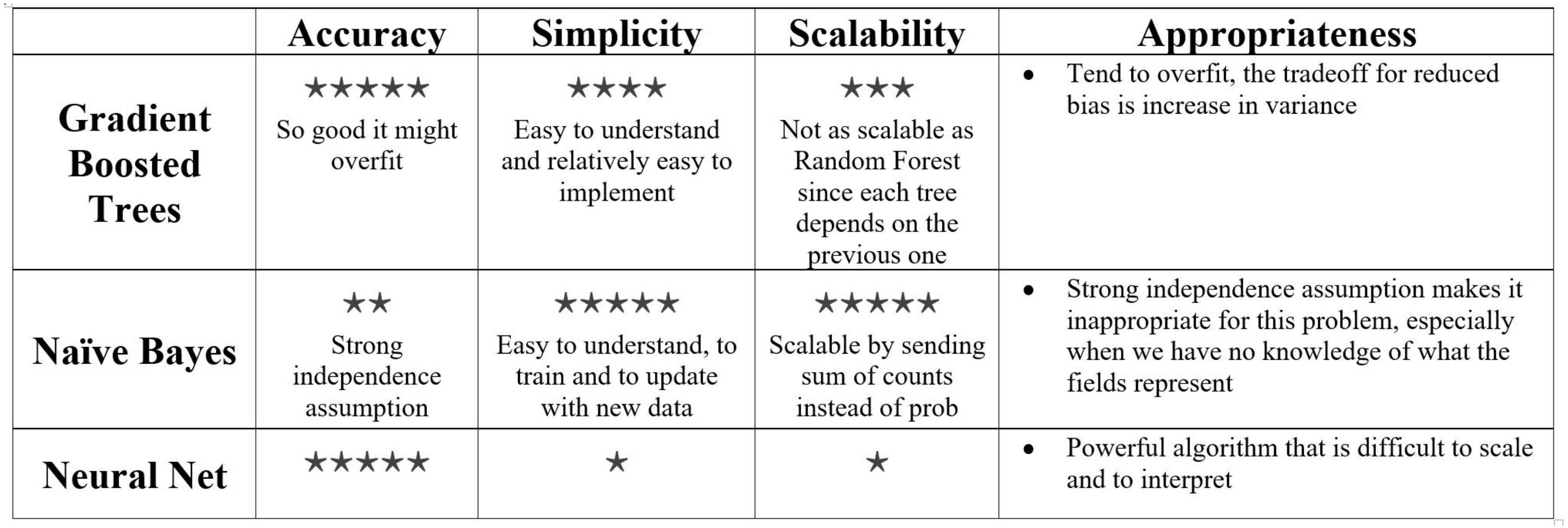
Insights from EDA

1. Some columns have few nulls, while others have many. Sometimes columns have nulls correlated with other columns
   - Won't be able to simply omit rows with missing data
2. Most numerical values are very right-skewed
3. Little correlation between the numerical columns
4. Categorical columns consist mostly of a disarrangement of rare categories
5. The classes are imbalanced: about 75% of the examples are negative
   - Models will favor high accuracy on the majority class unless we can weight the examples or re-sample.
6. Not much evidence of any columns correlating strongly with the target

Algorithm Comparison (1/2)

	Accuracy	Simplicity	Scalability	Appropriateness
Logistic Regression	★★★ Good feature engineering boosts accuracy but increases complexity	★★★★ Easy to understand and implement, but need to normalize and impute missing value	★★★★★ Scalable with gradient descent	• Can't feature engineer effectively without knowing what the features actually mean • Too many missing values • Choosing the right learning rate could be difficult
Decision Tree	★★★ A single tree is not very accurate, but boosting and bagging helps significantly	★★★★★ Easy to understand and implement.	★★★★★ Highly scalable, all nodes in the same frontier can be trained in parallel	• Handles missing values; No normalization needed, suitable for this problem • Starting point for both Random Forest and Gradient boosted tree • Simple enough to demonstrate in notebook
Random Forest	★★★★ Work when the forest is large and trees are random	★★★★ Easy to understand and relatively easy to implement	★★★★★ Scalable, each tree can be trained independently	• Easily scalable • Would need to down-sample the negative classes • Can be made more complicated by combining with other algorithms

Algorithm Comparison (2/2)



	Accuracy	Simplicity	Scalability	Appropriateness
Gradient Boosted Trees	★★★★★ So good it might overfit	★★★★ Easy to understand and relatively easy to implement	★★★ Not as scalable as Random Forest since each tree depends on the previous one	• Tend to overfit, the tradeoff for reduced bias is increase in variance
Naïve Bayes	★★ Strong independence assumption	★★★★★ Easy to understand, to train and to update with new data	★★★★★ Scalable by sending sum of counts instead of prob	• Strong independence assumption makes it inappropriate for this problem, especially when we have no knowledge of what the fields represent
Neural Net	★★★★★	★	★	• Powerful algorithm that is difficult to scale and to interpret

Why Trees?

- Little preprocessing/transformation required, no normalization
- Handles both numeric and categorical data
- Handles missing values and NAs naturally
- Highly scalable
- Decision tree serves as the basis of powerful ensemble algorithms
- Great for demonstrating parallelized computing

Algorithm Explanation (1/2)

1. Parallelize across data and across frontier:
    - Trees can easily be parallelized, each layer depends only on prior path, not prior values
    - parallelize along each frontier when we build an entire tree layer in each iteration.
2. Master-node: Initialize, Broadcast and Update:
    - Tree structure, frontier, list of possible splits
    - After each iteration, all three must be updated and re-broadcasted with the new splits information
3. Worker-nodes: Map data to node and find best split
    - For each record, decide which node the record is in based on frontier
    - Then for each of the possible splits:
        i. Decide if this record satisfies the split
        ii. Decide if the y-value is 1
    - Sum up counts, calculate entropy:
        i. Compute the total counts associated with each possible split

Algorithm Explanation (2/2)

- Calculate weighted entropy for each possible split:

$$H_{split1} = -(p_{y_1|split_1} log p_{y_1|split_1} + p_{y_0|split_1} log p_{y_0|split_1})$$
$$H_{split0} = -(p_{y_1|split_0} log p_{y_1|split_0} + p_{y_0|split_0} log p_{y_0|split_0})$$
$$p_{y_1|split_1} = \frac{\text{count of y = 1 for those with split = 1}}{\text{count of those with split = 1}}$$

- For each node, select the split that produces the minimum entropy
- Other metrics: Gini, SSE

4. Update tree structure, frontier and split listStop growing tree if criteria are met
    - Stop after x iterations
    - Stop when <n samples in leaf

Algorithm Implementation - Feature Hashing

1. Massive feature space for categorical data
2. convert the anonymized categorical data from strings into a numeric type
3. In order to reduce the feature space, we apply a hash function:
   - simple modulus operation, to reduce the feature size to 16 bits.
4. Since the data is anonymized, it is impractical to come up with a "logical" hash function to group "like" categories

Algorithm Implementation - Data Engineering

1. Numerical columns are pretty much used directly
   - NULLs are encoded with the value -10
2. Categorical values are kept if they are in the common categories
   - All rare values are converted to a special ID
   - NULLs are converted to yet another special ID
3. Transform our input data into LabeledPoint object

Implementing Trees - Homegrown

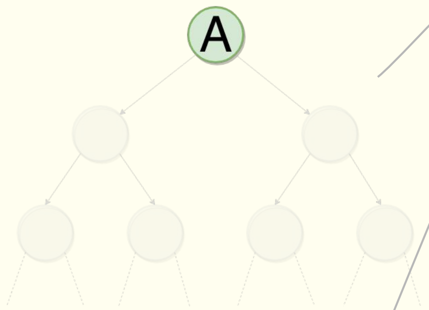
1. We implemented a “homegrown” implementation of the tree in Spark

Homegrown Trees Challenges

1. We need to parallelize:
    - Aggregation and calculations of the bins or categories we wish to split upon
    - Calculation of the entropy effect in determining which points offer the most information gain in tree construction

1. Difficult given a large dataset
    - a significant amount of cross communication between nodes in calculation is needed
2. Can be parallelized with Map-Reduce
    - mapping values across nodes, reducing in counting the values up

Master



```
{
   "A": [ (0, (0, 10)),
          (1, (0, 10)) ]
}
```

broadcast

flatMap

reduceByKey

label	features
0	[0, 2, 62, 0, …]
0	[2, 0, 304, 1, …]
1	[1, 15, 0, 0 …]

0	[1, 1, 54, 2, …]
1	[0, 15, 0, 0 …]
0	[1, 1, 54, 2, …]

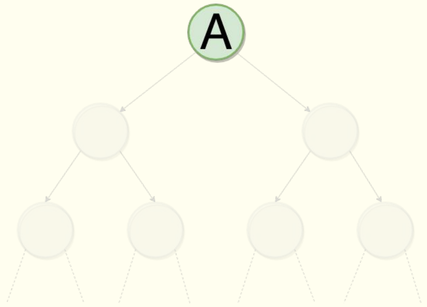
```
(A, 0, 0) => [ [1, 0], [0, 0] ]
(A, 0, 1) => [ [0, 0], [1, 0] ]
(A, 0, 2) => [ [0, 0], [1, 0] ]
...
(A, 0, 0) => [ [1, 0], [0, 0] ]
(A, 0, 1) => [ [1, 0], [0, 0] ]
(A, 0, 2) => [ [0, 0], [1, 0] ]
…
(A, 0, 0) => [ [1, 0], [0, 0] ]
(A, 0, 1) => [ [0, 0], [1, 0] ]
(A, 0, 2) => [ [0, 0], [1, 0] ]
…
…
```

```
(A, 0, 0) => [ [500, 250], [700, 550] ]
(A, 0, 1) => [ [200, 600], [1000, 200] ]
(A, 0, 2) => [ [  66, 790], [1134, 10] ]
...
(A, 1, 0) => [ … ]
...
```

mapValues

```
(A, 0, 0) => ( (0.44, 0.33, 750), (0.49, 0.46, 1250) )
(A, 0, 1) => ( (0.38, 0.75, 800), (0.28, 0.17, 1200) )
(A, 0, 2) => ( (0.14, 0.92, 856), (0.02, 0.01, 1144) )
...
(A, 1, 0) => ( … )
...
```

Master



```
{
   "A": [ (0, (0, 10)),
          (1, (0, 10)) ]
}
```

```
(A, 0, 0) => ( (0.44, 0.33, 750), (0.49, 0.46, 1250) )
(A, 0, 1) => ( (0.38, 0.75, 800), (0.28, 0.17, 1200) )
(A, 0, 2) => ( (0.14, 0.92, 856), (0.02, 0.01, 1144) )
...
(A, 1, 0) => ( … )
...
```

map

```
A => ( (0, 0), (0.44, 0.33, 750), (0.49, 0.46, 1250) )
A => ( (0, 1), (0.38, 0.75, 800), (0.28, 0.17, 1200) )
A => ( (0, 2), (0.14, 0.92, 856), (0.02, 0.01, 1144) )
...
A => ( (1, 0), … )
...
```

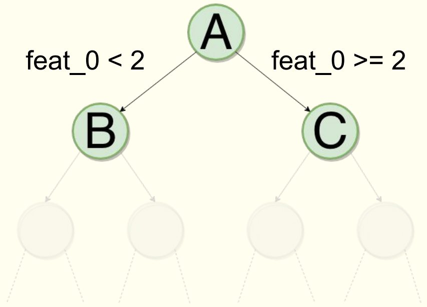
reduceByKey

```
A => ( (0, 2), (0.14, 0.92, 856), (0.02, 0.01, 1144) )
```

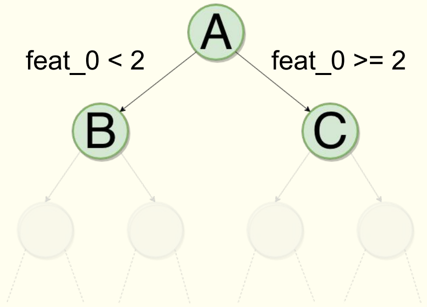
collectAsMap

Split A on feature 0 at the value 2

Master



Master



```
{
   "B": [ (0, (0, 2)),
          (1, (0, 10)) ],
   "C": [ (0, (3, 10)),
          (1, (0, 10)) ]
}
```

Split B on feat 1 at val 5 and C on feat 0 at val 7

label	features
0	[0, 2, 62, 0, …]
0	[2, 0, 304, 1, …]
1	[1, 15, 0, 0 …]

0	[1, 1, 54, 2, …]
1	[0, 15, 0, 0 …]
0	[1, 1, 54, 2, …]

flatMap

```
(B, 0, 0) => [ [1, 0], [0, 0] ]
(B, 0, 1) => [ [0, 0], [1, 0] ]
(C, 0, 1) => [ [0, 0], [1, 0] ]
...
(B, 0, 0) => [ [1, 0], [0, 0] ]
(C, 0, 1) => [ [1, 0], [0, 0] ]
(C, 0, 2) => [ [0, 0], [1, 0] ]
…
(B, 0, 0) => [ [1, 0], [0, 0] ]
(C, 0, 1) => [ [0, 0], [1, 0] ]
(B, 0, 2) => [ [0, 0], [1, 0] ]
…
…
```

reduceByKey

```
(B, 1, 0) => [ [250, 125], [350, 275] ]
(B, 1, 1) => [ [100, 300], [500, 100] ]
(C, 0, 0) => [ [  33, 395], [567, 5] ]
...
(C, 1, 0) => [ … ]
...
```

mapValues

```
(B, 1, 0) => ( (0.44, 0.33, 375), (0.49, 0.46, 625) )
(B, 1, 1) => ( (0.38, 0.75, 400), (0.28, 0.17, 600) )
(C, 0, 0) => ( (0.14, 0.92, 428), (0.02, 0.01, 572) )
...
(C, 1, 0) => ( … )
...
```

map, reduceByKey

```
B => ( (1, 5), (...), (...) )
C => ( (0, 7), (...), (...) )
```

Implementing MLlib Trees

1. In our sampling cases, both models yielded similar accuracy:

   Gradient Boosted Trees: Depth = 9, Max Bins = 25, Iterations = 5 accuracy = 0.773

	Precision	**Recall**	**F1-Score**	**Support**
0	0.81	0.92	0.86	745
1	0.51	0.26	0.35	221
avg/total	0.74	0.77	0.74	966

Random Forest: Max Bins = 52, Max Depth = 17, Num Trees = 25 accuracy = 0.781

	Precision	**Recall**	**F1-Score**	**Support**
0	0.80	0.96	0.87	800
1	0.55	0.18	0.27	234
avg/total	0.74	0.78	0.73	1034

Implementing MLlib Trees - Gradient Boosted Tree

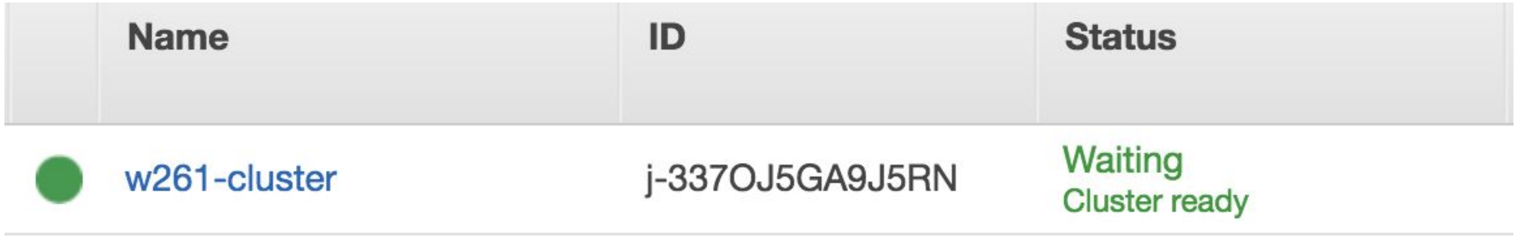
1. 2 parameters tuned that yielded the greatest improvement in accuracy:
   - Number of bins for splitting
   - Tree depth
2. Not a significant improvement in training trees with depths from 6 to 9 layers, despite greatly increased computation time
3. With increased binning, we run the risk of overfitting

Implementing MLlib Trees - Random Forest

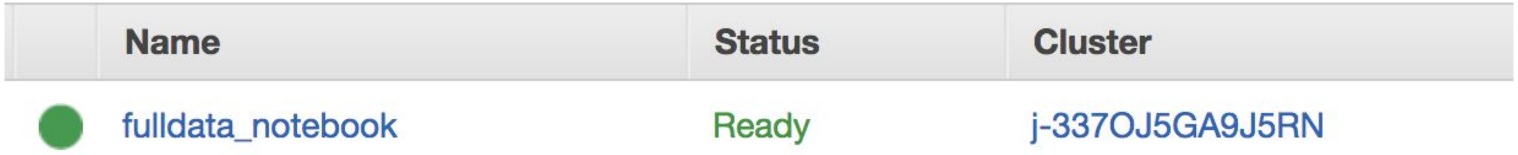
1. 2 parameters tuned that yielded the greatest improvement in accuracy:
    - Number of splits
    - Tree depth
2. The number of trees (> 10 trees) in the forest did not yield significant improvements despite the increased compute time
3. Increasing tree depth did improve accuracy as we are again able to delve into more variables in our tree decision path

Running on the Cluster

1. Ran on AWS EMR
   - A Hadoop framework that process large amount of data across Amazon EC2 instances
   - Spark can run in EMR and interact with data stored in Amazon S3

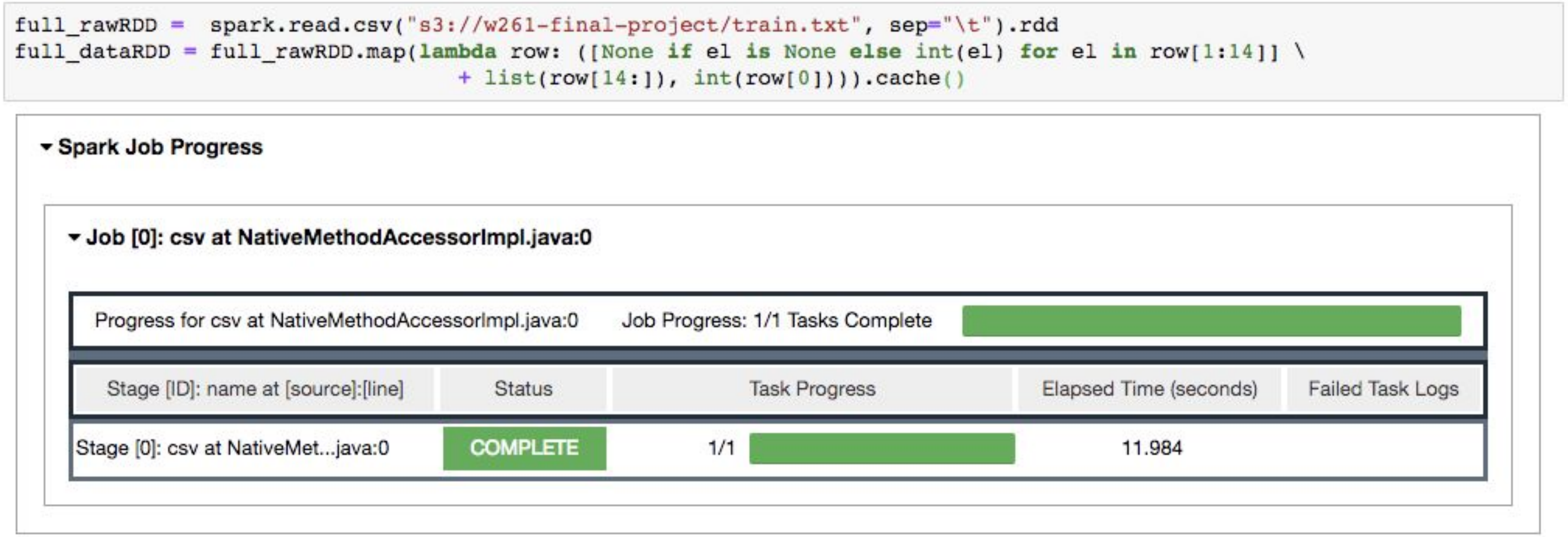


	Name	ID	Status
●	w261-cluster	j-337OJ5GA9J5RN	Waiting Cluster ready



	Name	Status	Cluster
●	fulldata_notebook	Ready	j-337OJ5GA9J5RN

EMR Notebook



```
full_rawRDD =  spark.read.csv("s3://w261-final-project/train.txt", sep="\t").rdd
full_dataRDD = full_rawRDD.map(lambda row: ([None if el is None else int(el) for el in row[1:14]] \
                               + list(row[14:]), int(row[0]))).cache()
```

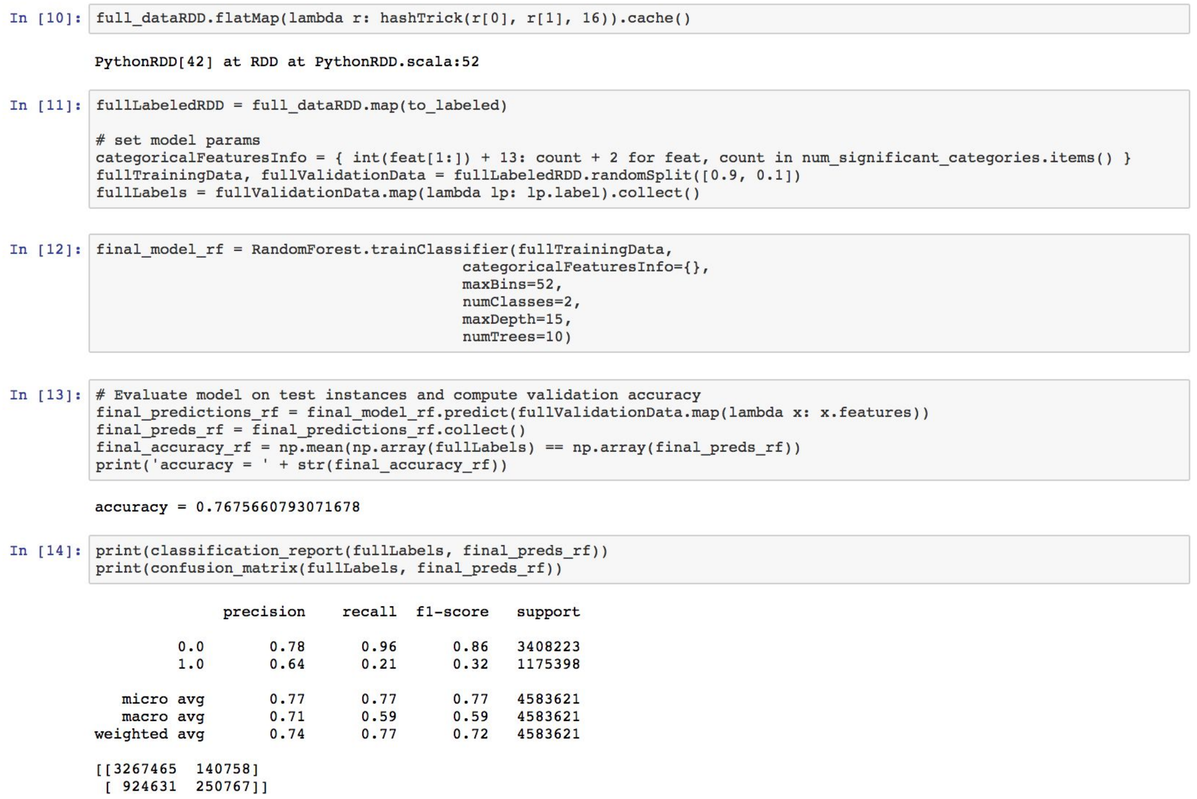
▾ Spark Job Progress

▾ Job [0]: csv at NativeMethodAccessorImpl.java:0

Progress for csv at NativeMethodAccessorImpl.java:0 Job Progress: 1/1 Tasks Complete

Stage [ID]: name at [source]:[line]	Status	Task Progress	Elapsed Time (seconds)	Failed Task Logs
Stage [0]: csv at NativeMet...java:0	COMPLETE	1/1	11.984	

Random Forest on the Full Data Set



```
In [10]: full_dataRDD.flatMap(lambda r: hashTrick(r[0], r[1], 16)).cache()

         PythonRDD[42] at RDD at PythonRDD.scala:52

In [11]: fullLabeledRDD = full_dataRDD.map(to_labeled)

         # set model params
         categoricalFeaturesInfo = { int(feat[1:]) + 13: count + 2 for feat, count in num_significant_categories.items() }
         fullTrainingData, fullValidationData = fullLabeledRDD.randomSplit([0.9, 0.1])
         fullLabels = fullValidationData.map(lambda lp: lp.label).collect()

In [12]: final_model_rf = RandomForest.trainClassifier(fullTrainingData,
                                                        categoricalFeaturesInfo={},
                                                        maxBins=52,
                                                        numClasses=2,
                                                        maxDepth=15,
                                                        numTrees=10)

In [13]: # Evaluate model on test instances and compute validation accuracy
         final_predictions_rf = final_model_rf.predict(fullValidationData.map(lambda x: x.features))
         final_preds_rf = final_predictions_rf.collect()
         final_accuracy_rf = np.mean(np.array(fullLabels) == np.array(final_preds_rf))
         print('accuracy = ' + str(final_accuracy_rf))

         accuracy = 0.7675660793071678

In [14]: print(classification_report(fullLabels, final_preds_rf))
         print(confusion_matrix(fullLabels, final_preds_rf))

                       precision    recall  f1-score   support

                  0.0       0.78      0.96      0.86   3408223
                  1.0       0.64      0.21      0.32   1175398

            micro avg       0.77      0.77      0.77   4583621
            macro avg       0.71      0.59      0.59   4583621
         weighted avg       0.74      0.77      0.72   4583621

         [[3267465  140758]
          [ 924631  250767]]
```

Key Concepts from w261

- Bias Variance Tradeoff / Model Complexity
  - ↑ # of bins and/or ↑ tree depth can ↑ the complexity of our model, but also ↑ bias (overfitting)
  - ↑ complexity by ↑ the # of iterations for Gradient Boosted trees or # of trees in the forest for Random Forest allows us to generalize better (↓ bias)
- Broadcasting / DAGs / Lazy Evaluation
  - Tree structures are broadcasted in the Spark job to all the clusters after every iteration
  - Using DAGs to minimize shuffling data around & Lazy Evaluation to reduce the execution time of the RDD operations in Spark
- Feature Engineering
  - Hashing - dimensionality reduction
  - Implemented a simple & space-efficient way to vectorize features

Questions?