PD5: Click-Through Rate Prediction

MTI850 - Big Data Analytics

Fall 2024

Rev 1.0 Fall 2025 - Spark 4.0

Équipe	9
Nicolas WEILL	WEIN64330301
Jean-Baptiste VANHERPEN	VANJ64340301
Matis MEBAZAA	MEBM77320301

This assignment covers the steps for creating a click-through rate (CTR) prediction pipeline. You will work with the Criteo Labs dataset that was used for a Kaggle competition.

This assignment covers:

- *Part 1:* Featurize categorical data using one-hot-encoding (OHE)
- *Part 2:* Construct an OHE dictionary
- *Part 3:* Parse CTR data and generate OHE features
  - *Visualization 1:* Feature frequency
- *Part 4:* CTR prediction and logloss evaluation
  - *Visualization 2:* ROC curve
- *Part 5:* Reduce feature dimension via feature hashing

> Note that, for reference, you can look up the details of:
>
> - the relevant Spark methods in PySpark's DataFrame API
> - the relevant NumPy methods in the NumPy Reference

Part 0: Preparing the environment

By default, when a shuffle operation occurs with DataFrames, the post-shuffle partition count is 200. This is controlled by Spark configuration value `spark.sql.shuffle.partitions` . 200 is a little too high for this dataset, so we set the post-shuffle partition count to twice the number of available cores in commodity machines. We are setting it to 6.

shuffle = données redistribuées ente les partitions --> une des opérations les plus couteuses dans spark Apparait quand : regroupement des données selon une clé / tri des données / jointure des tables par clé / calcul d'un groupby ou reduce / créer un dictionnaire OHE etc ... ici on change la size du shuffle car trop de répartitions en sortie --> inutilement lent pour notre dataset

In [1]:
```
import os
import findspark
findspark.init()

# Test module for MTI850
import testmti850

# Util module for MTI850
import utilmti850

import pyspark
from pyspark.sql import SparkSession

# on force spark a utiliser 6 paritions après le shuffle
spark = SparkSession.builder \
.master("local") \
.appName("Click Rate Prediction") \
.config("spark.sql.shuffle.partitions", "6") \
.config("spark.driver.maxResultSize", "10g") \
.getOrCreate()
```

```
WARNING: Using incubator modules: jdk.incubator.vector
Using Spark's default log4j profile: org/apache/spark/log4j2-defaults.prop
erties
Setting default log level to "WARN".
To adjust logging level use sc.setLogLevel(newLevel). For SparkR, use setL
ogLevel(newLevel).
25/11/12 14:10:30 WARN NativeCodeLoader: Unable to load native-hadoop libr
ary for your platform... using builtin-java classes where applicable
25/11/12 14:10:30 WARN Utils: Service 'SparkUI' could not bind on port 404
0. Attempting port 4041.
```

Let's check whether the `spark.sql.shuffle.partitions` was properly set.

In [2]:
```
spark.sparkContext.getConf().getAll()
```

Out[2]:
```
[('spark.rdd.compress', 'True'),
 ('spark.hadoop.fs.s3a.vectored.read.min.seek.size', '128K'),
 ('spark.master', 'local'),
 ('spark.app.submitTime', '1762956629466'),
 ('spark.app.id', 'local-1762956631036'),
 ('spark.executor.extraJavaOptions',
  '-Djava.net.preferIPv6Addresses=false -XX:+IgnoreUnrecognizedVMOptions
--add-modules=jdk.incubator.vector --add-opens=java.base/java.lang=ALL-U
NNAMED --add-opens=java.base/java.lang.invoke=ALL-UNNAMED --add-opens=ja
va.base/java.lang.reflect=ALL-UNNAMED --add-opens=java.base/java.io=ALL-
UNNAMED --add-opens=java.base/java.net=ALL-UNNAMED --add-opens=java.bas
e/java.nio=ALL-UNNAMED --add-opens=java.base/java.util=ALL-UNNAMED --ad
d-opens=java.base/java.util.concurrent=ALL-UNNAMED --add-opens=java.bas
e/java.util.concurrent.atomic=ALL-UNNAMED --add-opens=java.base/jdk.inte
rnal.ref=ALL-UNNAMED --add-opens=java.base/sun.nio.ch=ALL-UNNAMED --add-
opens=java.base/sun.nio.cs=ALL-UNNAMED --add-opens=java.base/sun.securit
y.action=ALL-UNNAMED --add-opens=java.base/sun.util.calendar=ALL-UNNAMED
--add-opens=java.security.jgss/sun.security.krb5=ALL-UNNAMED -Djdk.refle
ct.useDirectMethodHandle=false -Dio.netty.tryReflectionSetAccessible=tru
e'),
 ('spark.sql.artifact.isolation.enabled', 'false'),
 ('spark.executor.id', 'driver'),
 ('spark.submit.pyFiles', ''),
 ('spark.driver.extraJavaOptions',
  '-Djava.net.preferIPv6Addresses=false -XX:+IgnoreUnrecognizedVMOptions
--add-modules=jdk.incubator.vector --add-opens=java.base/java.lang=ALL-U
NNAMED --add-opens=java.base/java.lang.invoke=ALL-UNNAMED --add-opens=ja
va.base/java.lang.reflect=ALL-UNNAMED --add-opens=java.base/java.io=ALL-
UNNAMED --add-opens=java.base/java.net=ALL-UNNAMED --add-opens=java.bas
e/java.nio=ALL-UNNAMED --add-opens=java.base/java.util=ALL-UNNAMED --ad
d-opens=java.base/java.util.concurrent=ALL-UNNAMED --add-opens=java.bas
e/java.util.concurrent.atomic=ALL-UNNAMED --add-opens=java.base/jdk.inte
rnal.ref=ALL-UNNAMED --add-opens=java.base/sun.nio.ch=ALL-UNNAMED --add-
opens=java.base/sun.nio.cs=ALL-UNNAMED --add-opens=java.base/sun.securit
y.action=ALL-UNNAMED --add-opens=java.base/sun.util.calendar=ALL-UNNAMED
--add-opens=java.security.jgss/sun.security.krb5=ALL-UNNAMED -Djdk.refle
ct.useDirectMethodHandle=false -Dio.netty.tryReflectionSetAccessible=tru
e'),
 ('spark.hadoop.fs.s3a.vectored.read.max.merged.size', '2M'),
 ('spark.driver.port', '39827'),
 ('spark.submit.deployMode', 'client'),
 ('spark.app.startTime', '1762956629995'),
 ('spark.driver.host', 'MTI850'),
 ('spark.serializer.objectStreamReset', '100'),
 ('spark.sql.shuffle.partitions', '6'),
 ('spark.app.name', 'Click Rate Prediction'),
 ('spark.driver.maxResultSize', '10g'),
 ('spark.ui.showConsoleProgress', 'true')]
```

In [3]:
```
# or
current_value = spark.conf.get("spark.sql.shuffle.partitions")
print(f"The current value of spark.sql.shuffle.partitions is: {current_va
```

```
The current value of spark.sql.shuffle.partitions is: 6
```

Part 1: Featurize categorical data using one-hot-encoding

(1a) One-hot-encoding

We would like to develop code to convert categorical features to numerical ones, and to build intuition, we will work with a sample unlabeled dataset with three data points, with each data point representing an animal.

- The first feature indicates the type of animal (bear, cat, mouse);
- The second feature describes the animal's color (black, tabby); and
- The third (optional) feature describes what the animal eats (mouse, salmon).

In a one-hot-encoding (OHE) scheme, we want to represent each tuple of `(featureID, category)` via its own binary feature. We can do this in Python by creating a dictionary that maps each tuple to a distinct integer, where the integer corresponds to a binary feature.

To start, manually enter the entries in the OHE dictionary associated with the sample dataset by mapping the tuples to consecutive integers starting from zero, ordering the tuples first by featureID and next by category.

Later in this assignement, you will use OHE dictionaries to transform data points into compact lists of features that can be used in machine learning algorithms.

In [4]:
```
from collections import defaultdict

# Data for manual OHE
# Note: the first data point does not include any value for the optional
sample_one = [(0, 'mouse'), (1, 'black')]
sample_two = [(0, 'cat'), (1, 'tabby'), (2, 'mouse')]
sample_three =  [(0, 'bear'), (1, 'black'), (2, 'salmon')]

def sample_to_row(sample):
    tmp_dict = defaultdict(lambda: None) # init a non-existing key None i
    tmp_dict.update(sample)
    return [tmp_dict[i] for i in range(3)] # using range, it is possible

spark.createDataFrame(
    map(sample_to_row, [sample_one, sample_two, sample_three]), # feature
    ['animal', 'color', 'food']                                 # feature
).show()

sample_data_df = spark.createDataFrame([(sample_one,), (sample_two,), (sa

sample_data_df.show(truncate=False)
```

```
+------+-----+------+
|animal|color|  food|
+------+-----+------+
| mouse|black|  NULL|
|   cat|tabby| mouse|
|  bear|black|salmon|
+------+-----+------+

+-----------------------------------+
|features                           |
+-----------------------------------+
|[{0, mouse}, {1, black}]           |
|[{0, cat}, {1, tabby}, {2, mouse}] |
|[{0, bear}, {1, black}, {2, salmon}]|
+-----------------------------------+
```

In [5]:
```
# TODO: Replace <FILL IN> with appropriate code

sample_ohe_dict_manual = {}
sample_ohe_dict_manual[(0, 'bear')] = 0
sample_ohe_dict_manual[(0, 'cat')] = 1
sample_ohe_dict_manual[(0, 'mouse')] = 2
sample_ohe_dict_manual[(1, 'black')] = 3
sample_ohe_dict_manual[(1, 'tabby')] = 4
sample_ohe_dict_manual[(2, 'mouse')] = 5
sample_ohe_dict_manual[(2, 'salmon')] = 6

print(sample_ohe_dict_manual)
```

```
{(0, 'bear'): 0, (0, 'cat'): 1, (0, 'mouse'): 2, (1, 'black'): 3, (1, 'tab
by'): 4, (2, 'mouse'): 5, (2, 'salmon'): 6}
```

In [6]:
```
# TEST One-hot-encoding (1a)
testmti850.Test.assertEqualsHashed(sample_ohe_dict_manual[(0, 'bear')],
                        'b6589fc6ab0dc82cf12099d1c2d40ab994e8410c',
                        "incorrect value for sample_ohe_dict_manual[(0,'b
testmti850.Test.assertEqualsHashed(sample_ohe_dict_manual[(0, 'cat')],
                        '356a192b7913b04c54574d18c28d46e6395428ab',
                        "incorrect value for sample_ohe_dict_manual[(0,'c
testmti850.Test.assertEqualsHashed(sample_ohe_dict_manual[(0, 'mouse')],
                        'da4b9237bacccdf19c0760cab7aec4a8359010b0',
                        "incorrect value for sample_ohe_dict_manual[(0,'m
testmti850.Test.assertEqualsHashed(sample_ohe_dict_manual[(1, 'black')],
                        '77de68daecd823babbb58edb1c8e14d7106e83bb',
                        "incorrect value for sample_ohe_dict_manual[(1,'b
testmti850.Test.assertEqualsHashed(sample_ohe_dict_manual[(1, 'tabby')],
                        '1b6453892473a467d07372d45eb05abc2031647a',
                        "incorrect value for sample_ohe_dict_manual[(1,'t
testmti850.Test.assertEqualsHashed(sample_ohe_dict_manual[(2, 'mouse')],
                        'ac3478d69a3c81fa62e60f5c3696165a4e5e6ac4',
                        "incorrect value for sample_ohe_dict_manual[(2,'m
testmti850.Test.assertEqualsHashed(sample_ohe_dict_manual[(2, 'salmon')],
                        'c1dfd96eea8cc2b62785275bca38ac261256e278',
                        "incorrect value for sample_ohe_dict_manual[(2,'s
testmti850.Test.assertEquals(len(sample_ohe_dict_manual.keys()), 7,
                    'incorrect number of keys in sample_ohe_dict_manual')
```

```
1 test passed.
1 test passed.
1 test passed.
1 test passed.
1 test passed.
1 test passed.
1 test passed.
1 test passed.
```

(1b) Sparse vectors

Data points can typically be represented with a small number of non-zero OHE features relative to the total number of features that occur in the dataset. By leveraging this sparsity and using sparse vector representations for OHE data, we can reduce storage and computational burdens. Below are a few sample vectors represented as dense numpy arrays.

Use SparseVector to represent them in a sparse fashion, and verify that both the sparse and dense representations yield the same results when computing dot products (we will later use MLlib to train classifiers via gradient descent, and MLlib will need to compute dot products between SparseVectors and dense parameter vectors).

Use `SparseVector(size, *args)` to create a new sparse vector where size is the length of the vector and args is either:

1. A list of indices and a list of values corresponding to the indices. The indices list must be sorted in ascending order. For example, SparseVector(5, [1, 3, 4], [10, 30, 40]) will represent the vector [0, 10, 0, 30, 40]. The non-zero indices are 1, 3 and 4. On the other hand, SparseVector(3, [2, 1], [5, 5]) will give you an error because the indices list [2, 1] is not in ascending order. Note: you cannot simply sort the indices list, because otherwise the values will not correspond to the respective indices anymore.
2. A list of (index, value) pair. In this case, the indices need not be sorted. For example, SparseVector(5, [(3, 1), (1, 2)]) will give you the vector [0, 2, 0, 1, 0].

SparseVectors are much more efficient when working with sparse data because they do not store zero values (only store non-zero values and their indices). You'll need to create a sparse vector representation of each dense vector `a_dense` and `b_dense`.

In [7]:
```
import numpy as np
from pyspark.ml.linalg import SparseVector
```

In [8]:
```
# TODO: Replace <FILL IN> with appropriate code

a_dense = np.array([0., 3., 0., 4.])

a_sparse = SparseVector(4,[1,3], [3., 4.])

b_dense = np.array([0., 0., 0., 1.])

b_sparse = SparseVector(4, [3], [1.]) # 4 elements: only the index 3 elem
```

```
w = np.array([0.4, 3.1, -1.4, -.5])

print(a_dense.dot(w))
print(a_sparse.dot(w))
print(b_dense.dot(w))
print(b_sparse.dot(w))
```

```
7.300000000000001
7.300000000000001
-0.5
-0.5
```

In [9]:
```
# TEST Sparse Vectors (1b)
testmti850.Test.assertTrue(isinstance(a_sparse, SparseVector), 'a_sparse
testmti850.Test.assertTrue(isinstance(b_sparse, SparseVector), 'a_sparse
testmti850.Test.assertTrue(a_dense.dot(w) == a_sparse.dot(w),
                'dot product of a_dense and w should equal dot product of
testmti850.Test.assertTrue(b_dense.dot(w) == b_sparse.dot(w),
                'dot product of b_dense and w should equal dot product of
testmti850.Test.assertTrue(a_sparse.numNonzeros() == 2, 'a_sparse should
testmti850.Test.assertTrue(b_sparse.numNonzeros() == 1, 'b_sparse should
```

```
1 test passed.
1 test passed.
1 test passed.
1 test passed.
1 test passed.
1 test passed.
```

(1c) OHE features as sparse vectors

Now let's see how we can represent the OHE features for points in our sample dataset. Using the mapping defined by the OHE dictionary from Part (1a), manually define OHE features for the three sample data points using SparseVector format. Any feature that occurs in a point should have the value 1.0. For example, the `DenseVector` for a point with features 2 and 4 would be `[0.0, 0.0, 1.0, 0.0, 1.0, 0.0, 0.0]`. Following the table below, this vector represents `[(0, "mouse"), (1, "tabby")]`.

Feat. id	Feat. value	Onehot bit
0	bear	0
0	cat	1
0	mouse	2
1	black	3
1	tabby	4
2	mouse	5
2	salmon	6

In [10]:
```
# The 3 lines below are a reminder of the sample features
# sample_one = [(0, 'mouse'), (1, 'black')]
# sample_two = [(0, 'cat'), (1, 'tabby'), (2, 'mouse')]
# sample_three =  [(0, 'bear'), (1, 'black'), (2, 'salmon')]
```

```
# TODO: Replace <FILL IN> with appropriate code

sample_one_ohe_feat_manual = SparseVector(7, [2,3], [1.0, 1.0])

sample_two_ohe_feat_manual = SparseVector(7, [1,4,5], [1.0, 1.0, 1.0])

sample_three_ohe_feat_manual = SparseVector(7, [0,3,6], [1.0, 1.0, 1.0])
```

In [11]:
```
# TEST OHE Features as sparse vectors (1c)
testmti850.Test.assertTrue(isinstance(sample_one_ohe_feat_manual, SparseV
                'sample_one_ohe_feat_manual needs to be a SparseVector')
testmti850.Test.assertTrue(isinstance(sample_two_ohe_feat_manual, SparseV
                'sample_two_ohe_feat_manual needs to be a SparseVector')
testmti850.Test.assertTrue(isinstance(sample_three_ohe_feat_manual, Spars
                'sample_three_ohe_feat_manual needs to be a SparseVector'
testmti850.Test.assertEqualsHashed(sample_one_ohe_feat_manual,
                        'ecc00223d141b7bd0913d52377cee2cf5783abd6',
                        'incorrect value for sample_one_ohe_feat_manual')
testmti850.Test.assertEqualsHashed(sample_two_ohe_feat_manual,
                        '26b023f4109e3b8ab32241938e2e9b9e9d62720a',
                        'incorrect value for sample_two_ohe_feat_manual')
testmti850.Test.assertEqualsHashed(sample_three_ohe_feat_manual,
                        'c04134fd603ae115395b29dcabe9d0c66fbdc8a7',
                        'incorrect value for sample_three_ohe_feat_manual
```

```
1 test passed.
1 test passed.
1 test passed.
1 test passed.
1 test passed.
1 test passed.
```

(1d) Define a OHE function

Next we will use the OHE dictionary from Part (1a) to programatically generate OHE features from the original categorical data. First write a function called `one_hot_encoding` that creates OHE feature vectors in `SparseVector` format. Then use this function to create OHE features for the first sample data point and verify that the result matches the result from Part (1c).

> Note: We'll pass in the OHE dictionary as a Broadcast variable, which will greatly improve performance when we call this function as part of a UDF. **When accessing a broadcast variable, you *must* use `.value`.** For instance: `ohe_dict_broadcast.value`.

In [12]:
```
# the commented line below may help you to accomplish this task
sample_one = [(0, 'mouse'), (1, 'black')]
# SparseVector(7, [(sample_ohe_dict_manual[x], 1) for x in sample_one]) #

# TODO: Replace <FILL IN> with appropriate code

def one_hot_encoding(raw_feats, ohe_dict_broadcast, num_ohe_feats):
    """Produce a one-hot-encoding from a list of features and an OHE dict

    Note:
```

```
        You should ensure that the indices used to create a SparseVector

    Args:
        raw_feats (list of (int, str)): The features corresponding to a s
            feature consists of a tuple of featureID and the feature's va
        ohe_dict_broadcast (Broadcast of dict): Broadcast variable contai
            (featureID, value) to unique integer.
        num_ohe_feats (int): The total number of unique OHE features (com
            value).

    Returns:
        SparseVector: A SparseVector of length num_ohe_feats with indices
            identifiers for the (featureID, value) combinations that occu
            with values equal to 1.0.
    """

    ohe_dict = ohe_dict_broadcast.value
    indices = [ohe_dict[feat] for feat in raw_feats]
    indices = sorted(indices)
    values = [1.0] * len(indices)

    return SparseVector(num_ohe_feats, indices, values)

# Calculate the number of features in sample_ohe_dict_manual (varibable t
num_sample_ohe_feats = len(sample_ohe_dict_manual)

sample_ohe_dict_manual_broadcast = spark.sparkContext.broadcast(sample_oh

# Run one_hot_encoding() on sample_one.  Make sure to pass in the Broadca
sample_one_ohe_feat = one_hot_encoding(sample_one, sample_ohe_dict_manual

print(sample_one)

print(sample_one_ohe_feat)
```

```
[(0, 'mouse'), (1, 'black')]
(7,[2,3],[1.0,1.0])
```

In [13]:
```
# TEST Define an OHE Function (1d)
testmti850.Test.assertTrue(sample_one_ohe_feat == sample_one_ohe_feat_man
                'sample_one_ohe_feat should equal sample_one_ohe_feat_man
testmti850.Test.assertEquals(sample_one_ohe_feat, SparseVector(7, [2, 3],
                  'incorrect value for sample_one_ohe_feat')
testmti850.Test.assertEquals(one_hot_encoding([(1, 'black'), (0, 'mouse')
                                  num_sample_ohe_feats), SparseVector(7,
                  'incorrect definition for one_hot_encoding')
```

```
1 test passed.
1 test passed.
1 test passed.
```

(1e) Apply OHE to a dataset

Finally, use the function from Part (1d) to create OHE features for all three data points in the sample dataset. You will need to generate a UDF that can be used in a `DataFrame` `select` statement.

> Note: Your implemenation of `ohe_udf_generator` needs to call your `one_hot_encoding` function.

In [14]:
```
# TODO: Replace <FILL IN> with appropriate code
from pyspark.sql.functions import udf
from pyspark.ml.linalg import VectorUDT

def ohe_udf_generator(ohe_dict_broadcast):
    """Generate a UDF that is setup to one-hot-encode rows with the given

    Note:
        We'll reuse this function to generate a UDF that can one-hot-enco
        one-hot-encoding dictionary built from the training data.  Also,
        the number of features before calling the one_hot_encoding functi

    Args:
        ohe_dict_broadcast (Broadcast of dict): Broadcast variable contai
            (featureID, value) to unique integer.

    Returns:
        UserDefinedFunction: A UDF can be used in `DataFrame` `select` st
            function on each row in a given column.  This UDF should call
            function with the appropriate parameters.
    """
    length = len(ohe_dict_broadcast.value)

    return udf(lambda x: one_hot_encoding(x, ohe_dict_broadcast, length),

sample_ohe_dict_udf = ohe_udf_generator(sample_ohe_dict_manual_broadcast)

sample_ohe_df = sample_data_df.select(sample_ohe_dict_udf("features").ali

sample_ohe_df.show(truncate=False)
```

```
[Stage 2:>                                                          (0 +
1) / 1]
+-------------------------+
|ohe_features             |
+-------------------------+
|(7,[2,3],[1.0,1.0])      |
|(7,[1,4,5],[1.0,1.0,1.0])|
|(7,[0,3,6],[1.0,1.0,1.0])|
+-------------------------+
```

In [15]:
```
# TEST Apply OHE to a dataset (1e)
sample_ohe_data_values = sample_ohe_df.collect()
testmti850.Test.assertTrue(len(sample_ohe_data_values) == 3, 'sample_ohe_
testmti850.Test.assertEquals(sample_ohe_data_values[0], (SparseVector(7,
```

```
                    'incorrect OHE for first sample')
testmti850.Test.assertEquals(sample_ohe_data_values[1], (SparseVector(7,
                    'incorrect OHE for second sample')
testmti850.Test.assertEquals(sample_ohe_data_values[2], (SparseVector(7,
                    'incorrect OHE for third sample')
testmti850.Test.assertTrue('one_hot_encoding' in sample_ohe_dict_udf.func
                  'ohe_udf_generator should call one_hot_encoding')
```

```
1 test passed.
1 test passed.
1 test passed.
1 test passed.
1 test passed.
```

Part 2: Construct an OHE dictionary

(2a) DataFrame with rows of `(featureID, category)`

To start, create a DataFrame of distinct `(feature_id, category)` tuples. In our sample dataset, the seven items in the resulting DataFrame are:

- `(0, 'bear')`,
- `(0, 'cat')`,
- `(0, 'mouse')`,
- `(1, 'black')`,
- `(1, 'tabby')`,
- `(2, 'mouse')`,
- `(2, 'salmon')`.

Notably `'black'` appears twice in the dataset but only contributes one item to the DataFrame: `(1, 'black')`, while `'mouse'` also appears twice and contributes two items: `(0, 'mouse')` and `(2, 'mouse')`. Use explode and distinct.

In [16]:
```
# TODO: Replace <FILL IN> with appropriate code
from pyspark.sql.functions import explode

sample_distinct_feats_df_0 = sample_data_df.select(explode("features").al

sample_distinct_feats_df = (sample_distinct_feats_df_0.select("feat._1",
)

sample_distinct_feats_df.show(truncate=False)
```

```
+---------+--------+
|featureID|category|
+---------+--------+
|0        |cat     |
|0        |mouse   |
|1        |black   |
|2        |mouse   |
|2        |salmon  |
|0        |bear    |
|1        |tabby   |
+---------+--------+
```

(2b) OHE Dictionary from distinct features

Next, create an RDD of key-value tuples, where each `(feature_id, category)` tuple in `sample_distinct_feats_df` is a key and the values are distinct integers ranging from 0 to (number of keys - 1). Then convert this RDD into a dictionary, which can be done using the `collectAsMap` action. Note that there is no unique mapping from keys to values, as all we require is that each `(featureID, category)` key be mapped to a unique integer between 0 and the number of keys. In this exercise, any valid mapping is acceptable. Use zipWithIndex followed by collectAsMap.

In our sample dataset, one valid list of key-value tuples is: `[((0, 'bear'), 0), ((2, 'salmon'), 1), ((1, 'tabby'), 2), ((2, 'mouse'), 3), ((0, 'mouse'), 4), ((0, 'cat'), 5), ((1, 'black'), 6)]`. The dictionary defined in Part (1a) illustrates another valid mapping between keys and integers.

> Note: We provide the code to convert the DataFrame to an RDD.

In [17]:
```
# TODO: Replace <FILL IN> with appropriate code
sample_distinct_feats_df.show()

sample_ohe_dict = sample_ohe_dict = (sample_distinct_feats_df.rdd.map(lam
# extraction du tuble marchait pas je l'ai réarrangée

print(sample_ohe_dict)
```

```
+---------+--------+
|featureID|category|
+---------+--------+
|        0|     cat|
|        0|   mouse|
|        1|   black|
|        2|   mouse|
|        2|  salmon|
|        0|    bear|
|        1|   tabby|
+---------+--------+

{(0, 'cat'): 0, (0, 'mouse'): 1, (1, 'black'): 2, (2, 'mouse'): 3, (2, 'sa
lmon'): 4, (0, 'bear'): 5, (1, 'tabby'): 6}
```

In [18]:
```
# TEST OHE Dictionary from distinct features (2b)
testmti850.Test.assertEquals(
    sorted(sample_ohe_dict.keys()),
```

```
    [(0, 'bear'), (0, 'cat'), (0, 'mouse'), (1, 'black'), (1, 'tabby'), (
    'sample_ohe_dict has unexpected keys'
)
testmti850.Test.assertEquals(sorted(sample_ohe_dict.values()), list(range
```

```
1 test passed.
1 test passed.
```

(2c) Automated creation of an OHE dictionary

Now use the code from Parts (2a) and (2b) to write a function that takes an input dataset and outputs an OHE dictionary. Then use this function to create an OHE dictionary for the sample dataset, and verify that it matches the dictionary from Part (2b).

In [19]:
```
# TODO: Replace <FILL IN> with appropriate code

def create_one_hot_dict(input_df):
    """Creates a one-hot-encoder dictionary based on the input data.

    Args:
        input_df (DataFrame with 'features' column): A DataFrame where ea
            (featureID, value) tuples.

    Returns:
        dict: A dictionary where the keys are (featureID, value) tuples a
            unique integers.
    """

    return (
        input_df.select(explode("features").alias("feat")).select("feat._
    )


sample_ohe_dict_auto = create_one_hot_dict(sample_data_df)

print(sample_ohe_dict_auto)

sample_data_df.show(truncate=False)
```

```
{(0, 'cat'): 0, (0, 'mouse'): 1, (1, 'black'): 2, (2, 'mouse'): 3, (2, 'sa
lmon'): 4, (0, 'bear'): 5, (1, 'tabby'): 6}
+-----------------------------------+
|features                           |
+-----------------------------------+
|[{0, mouse}, {1, black}]           |
|[{0, cat}, {1, tabby}, {2, mouse}] |
|[{0, bear}, {1, black}, {2, salmon}]|
+-----------------------------------+
```

Note that here `(0, 'bear')` maps to 5 instead of 0, as in the table in Section 1c.

In [20]:
```
# TEST Automated creation of an OHE dictionary (2c)
testmti850.Test.assertEquals(
    sorted(sample_ohe_dict_auto.keys()),
    [(0, 'bear'), (0, 'cat'), (0, 'mouse'), (1, 'black'), (1, 'tabby'), (
    'sample_ohe_dict_auto has unexpected keys'
```

```
)
testmti850.Test.assertEquals(sorted(sample_ohe_dict_auto.values()), list(
```

```
1 test passed.
1 test passed.
```

Part 3: Parse CTR data and generate OHE features

Before we can proceed, you will first need to obtain the data from Criteo (or download directly from Moodle).

- Run the following command in a terminal shell cell to download the dataset and make it available in the hdfs filesystem.

```
start-dfs.sh && start-yarn.sh
mkdir dac_sample
wget http://labs.criteo.com/wp-content/uploads/2015/04/
dac_sample.tar.gz
mkdir dac_sample && tar -zxf dac_sample.tar.gz --directory
dac_sample
hdfs dfs -put dac_sample /dac_sample
```

- The next cell make available the dataset as a `DataFrame` in variable `raw_df` .

In [23]:
```
# By default, it removes the line separators characters (\r and \n)
raw_df = spark.read.text("hdfs://localhost:9000/dac_sample/dac_sample.txt
raw_df.show()
```

```
+--------------------+
|                text|
+--------------------+
|0\t1\t1\t5\t0\t13...|
|0\t2\t0\t44\t1\t1...|
|0\t2\t0\t1\t14\t7...|
|0\t\t893\t\t\t439...|
|0\t3\t-1\t\t0\t2\...|
|0\t\t-1\t\t\t1282...|
|0\t\t1\t2\t\t3168...|
|1\t1\t4\t2\t0\t0\...|
|0\t\t44\t4\t8\t19...|
|0\t\t35\t\t1\t337...|
|0\t\t2\t632\t0\t5...|
|0\t0\t6\t6\t6\t42...|
|1\t0\t-1\t\t\t146...|
|1\t\t2\t11\t5\t10...|
|0\t0\t51\t84\t4\t...|
|0\t\t2\t1\t18\t20...|
|1\t1\t987\t\t2\t1...|
|0\t0\t1\t\t0\t165...|
|0\t0\t24\t4\t2\t2...|
|0\t7\t102\t\t3\t7...|
+--------------------+
only showing top 20 rows
```

(3a) Loading and splitting the data

We are now ready to start working with the actual CTR data, and our first task involves splitting it into training, validation, and test sets.

Use the randomSplit method with the specified weights and seed to create DFs storing each of these datasets, and then cache each of these DFs, as we will be accessing them multiple times in the remainder of this assignment.

Finally, compute the size of each dataset.

In [24]:
```
# TODO: Replace <FILL IN> with appropriate code

weights = [.8, .1, .1]
seed = 42

# Use randomSplit with weights and seed (note that randomSplit perform so
raw_train_df, raw_validation_df, raw_test_df = raw_df.randomSplit(weights

# Cache and count the DataFrames
n_train = raw_train_df.cache().count()
n_val = raw_validation_df.cache().count()
n_test = raw_test_df.cache().count()

print("Train samples = "+str(n_train), "\nValidation samples = "+str(n_va
      "\nTest samples = "+str(n_test), "\nTotal number of samples = "+str

raw_df.show(1)
```

```
Train samples = 79901
Validation samples = 10037
Test samples = 10062
Total number of samples = 100000
+--------------------+
|                text|
+--------------------+
|0\t1\t1\t5\t0\t13...|
+--------------------+
only showing top 1 row
```

In [25]:
```
# TEST Loading and splitting the data (3a)

testmti850.Test.assertTrue(all([raw_train_df.is_cached, raw_validation_df
                'you must cache the split data')
testmti850.Test.assertEquals(n_train, 79901, 'incorrect value for n_train
testmti850.Test.assertEquals(n_val, 10037, 'incorrect value for n_val')
testmti850.Test.assertEquals(n_test, 10062, 'incorrect value for n_test')
```

```
1 test passed.
1 test passed.
1 test passed.
1 test passed.
```

(3b) Extract features

We will now parse the raw training data to create a `DataFrame` that we can subsequently use to create an OHE dictionary. Note from the `show()` command in Part (3a) that each raw data point is a string containing several fields separated by some delimiter. For now, we will ignore the first field (which is the 0-1 label), and parse the

remaining fields (or raw features). To do this, complete the implemention of the `parse_point` function.

In [28]:
```
# TODO: Replace <FILL IN> with appropriate code

def parse_point(point):
    """Converts a comma separated string into a list of (featureID, value

    Note:
        featureIDs should start at 0 and increase to the number of featur

    Args:
        point (str): A comma separated string where the first value is th
            are features.

    Returns:
        list: A list of (featureID, value) tuples.
    """
    fields = point.split("\t")
    features = fields[1:]
    return [(i, features[i]) for i in range(len(features))]

pt = raw_df.select('text').first()[0]

print(pt)

print(parse_point(raw_df.select('text').first()[0])[: 6])
```

```
0       1       1       5       0       1382    4       15      2       18
1       1       2               2       68fd1e64        80e26c9b        fb
936136  7b4723c4        25c83c98        7e0ccccf        de7995b8        1f
89b562  a73ee510        a8cd5504        b2cb9c98        37c9c164        28
24a5f6  1adce6ef        8ba8b39a        891b62e7        e5ba7672        f5
4016b9  21ddcdc9        b1252a9d        07b5194c                3a171ecb
        c5c50484        e8b83407        9727dd16
[(0, '1'), (1, '1'), (2, '5'), (3, '0'), (4, '1382'), (5, '4')]
```

Note that these features are similar to that involving animals. Rather than strings with animal names, now we have strings with numeric content.

In [29]:
```
# TEST Extract features (3b)
testmti850.Test.assertEquals(
    parse_point(raw_df.select('text').first()[0])[: 3],
    [(0, u'1'), (1, u'1'), (2, u'5')],
    'incorrect implementation of parse_point'
)
```

```
1 test passed.
```

(3c) Extracting features continued

Next, we will create a `parse_raw_df` function that creates a `label` column from the first value in the text and a `feature` column from the rest of the values. The `feature` column will be created using `parse_point_udf`, which we've provided and is based on your `parse_point` function. Note that to name your columns you should use alias.

You can split the `text` field in `raw_df` using split and retrieve the first value of the resulting array with getItem.

Be sure to call cast to cast the column value to `double`. Your `parse_raw_df` function should also cache the DataFrame it returns.

In [31]:
```
# TODO: Replace <FILL IN> with the appropriate code
from pyspark.sql.functions import udf, split, monotonically_increasing_id
from pyspark.sql.types import ArrayType, StructType, StructField, LongTyp

parse_point_udf = udf(parse_point, ArrayType(StructType([StructField('_1'
                                                         StructField('_2'

def parse_raw_df(raw_df):
    """Convert a DataFrame consisting of rows of tab separated text into

    Args:
        raw_df (DataFrame with a 'text' column): DataFrame containing the

    Returns:
        DataFrame: A DataFrame with 'label' and 'feature' columns.
    """

    # Remember that each row comprises a 0-or-1 label (item 0) and a sequ
    label_col = split(raw_df['text'], '\t').getItem(0).cast(DoubleType())
    features_col = parse_point_udf(raw_df['text']).alias('feature')
    parsed_df = raw_df.select(label_col, features_col)

    return parsed_df.cache()


# Parse the raw training DataFrame
parsed_train_df = parse_raw_df(raw_train_df)

from pyspark.sql.functions import explode, col
num_categories = (parsed_train_df.select(explode('feature').alias('featur
                    .select(col('feature').getField('_1').alias('featureN
                    .groupBy('featureNumber')
                    .sum()
                    .orderBy('featureNumber')
                    .collect())


# show the number of categories associated with the featureNumber 2
print(num_categories[2][1])

print('first rows parsed_train_df')
parsed_train_df.show(2, truncate=40)

print('last rows of parsed_train_df')
parsed_train_df_inverted = parsed_train_df.withColumn(
    "index", monotonically_increasing_id()
).sort('index', ascending=False).select('label', 'feature')
parsed_train_df_inverted.show(2, truncate=40)

raw_train_df.show(2, truncate=40)
```

```
1702
first rows parsed_train_df
+-----+--------------------------------------+
|label|                               feature|
+-----+--------------------------------------+
|  0.0|[{0, }, {1, -1}, {2, }, {3, }, {4, },...|
|  0.0|[{0, }, {1, -1}, {2, }, {3, }, {4, },...|
+-----+--------------------------------------+
only showing top 2 rows
last rows of parsed_train_df
+-----+--------------------------------------+
|label|                               feature|
+-----+--------------------------------------+
|  1.0|[{0, 98}, {1, 0}, {2, 2}, {3, 0}, {4,...|
|  1.0|[{0, 96}, {1, 3}, {2, 8}, {3, 5}, {4,...|
+-----+--------------------------------------+
only showing top 2 rows
+--------------------------------------+
|                                  text|
+--------------------------------------+
|0\t\t-1\t\t\t\t\t\t\t\t\t\t\t\t05db91...|
|0\t\t-1\t\t\t\t\t\t\t\t\t\t\t\t05db91...|
+--------------------------------------+
only showing top 2 rows
```

Notice that the first two rows have several features with blank/null value (e.g., [2,], [3,], [4,]).

In [32]:
```
# TEST Extract features (3c)
testmti850.Test.assertTrue(parsed_train_df.is_cached, 'parse_raw_df shoul
testmti850.Test.assertEquals(num_categories[2][1], 1702, 'incorrect imple
testmti850.Test.assertEquals(num_categories[32][1], 128, 'incorrect imple
```

```
1 test passed.
1 test passed.
1 test passed.
```

(3d) Create an OHE dictionary from the dataset

Note that `parse_point` returns a data point as a list of `(featureID, category)` tuples, which is the same format as the sample dataset studied in Parts 1 and 2 of this assignment. Using this observation, create an OHE dictionary from the parsed training data using the function implemented in Part (2c).

Note that we will assume for simplicity that all features in our CTR dataset are categorical.

In [34]:
```
# TODO: Replace <FILL IN> with appropriate code
ctr_ohe_dict = create_one_hot_dict(parsed_train_df.withColumnRenamed("fea
num_ctr_ohe_feats = len(ctr_ohe_dict)

print(num_ctr_ohe_feats)

print(ctr_ohe_dict[(0, '')])
```

```
[Stage 56:>                                                         (0 +
1) / 1]
```

```
234658
0
```

In [35]:
```
# TEST Create an OHE dictionary from the dataset (3d)
testmti850.Test.assertEquals(num_ctr_ohe_feats, 234658, 'incorrect number
testmti850.Test.assertTrue((0, '') in ctr_ohe_dict, 'incorrect features i
```

```
1 test passed.
1 test passed.
```

(3e) Apply OHE to the dataset

Now let's use this OHE dictionary, by starting with the training data that we've parsed into `label` and `feature` columns, to create one-hot-encoded features.

Recall that we created a function `ohe_udf_generator` that can create the UDF that we need to convert row into `features`. Make sure that `ohe_train_df` contains a `label` and `features` column and is cached.

In [36]:
```
# TODO: Replace <FILL IN> with the appropriate code

ohe_dict_broadcast = spark.sparkContext.broadcast(ctr_ohe_dict)

ohe_dict_udf = ohe_udf_generator(ohe_dict_broadcast)

# each row of parsed_train_df is a list, the suitable format for ohe_dict
# generated for each row and gives rise to a SparseVector
ohe_train_df = parsed_train_df.select(parsed_train_df["label"],ohe_dict_u

print(ohe_train_df.count())

print(ohe_train_df.take(1))
```

```
[Stage 57:>                                                         (0 +
1) / 1]
79901
[Row(label=0.0, features=SparseVector(234658, {0: 1.0, 1: 1.0, 2: 1.0, 3:
1.0, 4: 1.0, 5: 1.0, 6: 1.0, 37412: 1.0, 37413: 1.0, 37414: 1.0, 37415: 1.
0, 67076: 1.0, 78303: 1.0, 78304: 1.0, 78305: 1.0, 78306: 1.0, 78307: 1.0,
78308: 1.0, 78309: 1.0, 78310: 1.0, 117312: 1.0, 117313: 1.0, 117314: 1.0,
117315: 1.0, 117316: 1.0, 156453: 1.0, 156454: 1.0, 156455: 1.0, 156456: 1
.0, 156457: 1.0, 195456: 1.0, 195457: 1.0, 195458: 1.0, 195459: 1.0, 19546
0: 1.0, 195461: 1.0, 195462: 1.0, 195463: 1.0, 195464: 1.0}))]
```

In [37]:
```
# TEST Apply OHE to the dataset (3e)
testmti850.Test.assertTrue('label' in ohe_train_df.columns and 'features'
testmti850.Test.assertTrue(ohe_train_df.is_cached, 'ohe_train_df should b
num_nz = sum(parsed_train_df.rdd.map(lambda r: len(r[1])).take(5))
num_nz_alt = sum(ohe_train_df.rdd.map(lambda r: len(r[1].indices)).take(5
testmti850.Test.assertEquals(num_nz, num_nz_alt, 'incorrect value for ohe
```

```
1 test passed.
1 test passed.


1 test passed.
```

Visualization 1: Feature frequency

We will now visualize the number of times each of the 234,658 OHE features appears in the training data. We first compute the number of times each feature appears, then bucket the features by these counts.

The buckets are sized by powers of 2, so the first bucket corresponds to features that appear exactly once (2^0), the second to features that appear twice (2^1), the third to features that occur between three and four (2^2) times, the fifth bucket is five to eight (2^3) times and so on.

The scatter plot below shows the logarithm of the bucket thresholds versus the logarithm of the number of features that have counts that fall in the buckets.

In [38]:
```
from pyspark.sql.types import ArrayType, IntegerType
from pyspark.sql.functions import log2

# indices of the 1's in the one-hot encoded features
get_indices = udf(lambda sv: list(map(int, sv.indices)), ArrayType(Intege
feature_counts_df = (ohe_train_df
                     .select(explode(get_indices('features')))
                     .groupBy('col')
                     .count()
                     .withColumn('bucket', log2('count').cast('int'))
                     .groupBy('bucket')
                     .count()
                     .orderBy('bucket'))

feature_counts_df.show()

feature_counts = feature_counts_df.collect()
```

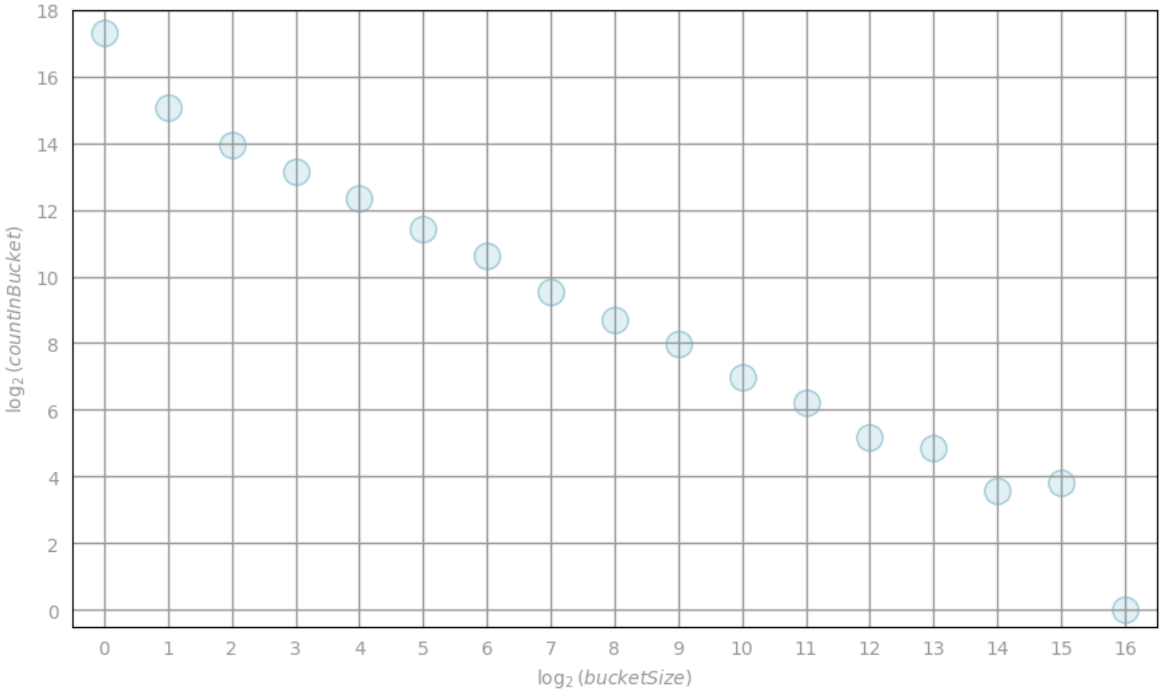
```
+------+------+
|bucket| count|
+------+------+
|     0|164533|
|     1| 34042|
|     2| 15891|
|     3|  9045|
|     4|  5153|
|     5|  2736|
|     6|  1554|
|     7|   743|
|     8|   416|
|     9|   253|
|    10|   126|
|    11|    74|
|    12|    36|
|    13|    29|
|    14|    12|
|    15|    14|
|    16|     1|
+------+------+
```

In [39]:
```
import matplotlib.pyplot as plt

x, y = zip(*feature_counts) # bucket x count
x, y = x, np.log2(y)

def prepare_plot(xticks, yticks, figsize=(10.5, 6), hide_labels=False, gr
                 grid_width=1.0):
    """Template for generating the plot layout."""
    plt.close()
    fig, ax = plt.subplots(figsize=figsize, facecolor='white', edgecolor=
    ax.axes.tick_params(labelcolor='#999999', labelsize='10')
    ax.set_xlim([-.5, xticks[-1] + .5])
    ax.set_ylim([-.5, yticks[-1]])
    for axis, ticks in [(ax.get_xaxis(), xticks), (ax.get_yaxis(), yticks
        axis.set_ticks_position('none')
        axis.set_ticks(ticks)
        axis.label.set_color('#999999')
        if hide_labels: axis.set_ticklabels([])
    plt.grid(color=grid_color, linewidth=grid_width, linestyle='-')
    map(lambda position: ax.spines[position].set_visible(False), ['bottom
    return fig, ax

# generate layout and plot data
fig, ax = prepare_plot(np.arange(0, max(x) + 1, 1), np.arange(0, max(y) +
ax.set_xlabel(r'$\log_2(bucketSize)$')
ax.set_ylabel(r'$\log_2(countInBucket)$')
ax.scatter(x, y, s=14**2, c='#d6ebf2', edgecolors='#8cbfd0', alpha=0.75)
plt.show()
```



(3f) Handling unseen features

We naturally would like to repeat the process from Part (3e), to compute OHE features for the validation and test datasets. However, we must be careful, as some categorical values will likely appear in new data that did not exist in the training data. To deal with this situation, update the `one_hot_encoding()` function from Part (1d) to ignore

previously unseen categories, and then compute OHE features for the validation data. Remember that you can parse a raw DataFrame using `parse_raw_df`.

> Note: you will have to generate a new UDF using `ohe_udf_generator` so that the updated `one_hot_encoding` function is used. And make sure to cache `ohe_validation_df`.

In [41]:
```
# TODO: Replace <FILL IN> with appropriate code

def one_hot_encoding(raw_feats, ohe_dict_broadcast, num_ohe_feats):
    """Produce a one-hot-encoding from a list of features and an OHE dict

    Note:
        You should ensure that the indices used to create a SparseVector
        function handles missing features.

    Args:
        raw_feats (list of (int, str)): The features corresponding to a s
            feature consists of a tuple of featureID and the feature's va
        ohe_dict_broadcast (Broadcast of dict): Broadcast variable contai
            (featureID, value) to unique integer.
        num_ohe_feats (int): The total number of unique OHE features (com
            value).

    Returns:
        SparseVector: A SparseVector of length num_ohe_feats with indices
            identifiers for the (featureID, value) combinations that occu
            with values equal to 1.0.
    """
    ohe_dict = ohe_dict_broadcast.value
    indices = [ohe_dict[x] for x in raw_feats if x in ohe_dict]
    indices = sorted(indices)
    values = [1.0] * len(indices)
    return SparseVector(num_ohe_feats, indices, values)

ohe_dict_broadcast = spark.sparkContext.broadcast(ctr_ohe_dict)
ohe_dict_missing_udf = ohe_udf_generator(ohe_dict_broadcast)


parsed_validation_df = parse_raw_df(raw_validation_df)

ohe_validation_df = parsed_validation_df.select(
    parsed_validation_df["label"],
    ohe_dict_missing_udf(parsed_validation_df["feature"]).alias("features
).cache()


ohe_validation_df.count()

ohe_validation_df.show(1, truncate=False)
```

```
25/11/12 14:32:29 WARN CacheManager: Asked to cache already cached data.
[Stage 83:>                                                         (0 +
1) / 1]
```

```
+-----
+--------------------------------------------------------------------------------
--------------------------------------------------------------------------------
--------------------------------------------------------------------------------
--------------------------------------------------------------------------------
--------------------------------------------------------------------------------
-------------------+
|label|features
|
+-----
+--------------------------------------------------------------------------------
--------------------------------------------------------------------------------
--------------------------------------------------------------------------------
--------------------------------------------------------------------------------
--------------------------------------------------------------------------------
-------------------+
|0.0  |(234658,[0,1,2,3,4,6,8,16,37412,37414,37416,37417,67076,67077,7830
3,78304,78305,78308,78310,78312,85226,89584,117312,117313,117314,117315,11
7317,156453,156454,156456,156457,195456,195457,195458,195459,195460,19546
1,195462,195463],
[1.0,1.0,1.0,1.0,1.0,1.0,1.0,1.0,1.0,1.0,1.0,1.0,1.0,1.0,1.0,1.0,1.0,1.0,1
.0,1.0,1.0,1.0,1.0,1.0,1.0,1.0,1.0,1.0,1.0,1.0,1.0,1.0,1.0,1.0,1.0,1.0,1.0
,1.0,1.0])|
+-----
+--------------------------------------------------------------------------------
--------------------------------------------------------------------------------
--------------------------------------------------------------------------------
--------------------------------------------------------------------------------
--------------------------------------------------------------------------------
-------------------+
only showing top 1 row
```

In [42]:
```
# TEST Handling unseen features (3f)
from pyspark.sql.functions import size, sum as sqlsum

testmti850.Test.assertTrue(ohe_validation_df.is_cached, 'you need to cach
num_nz_val = (ohe_validation_df
                .select(sqlsum(size(get_indices('features'))))
                .first()[0])

nz_expected = 371557
testmti850.Test.assertEquals(num_nz_val, nz_expected, 'incorrect number o
```

```
1 test passed.
1 test passed.
```

Part 4: CTR prediction and logloss evaluation

(4a) Logistic regression

We are now ready to train our first CTR classifier. A natural classifier to use in this setting is logistic regression, since it models the probability of a click-through event rather than returning a binary response, and when working with rare events, probabilistic predictions are useful.

First use LogisticRegression from the pyspark.ml package to train a model using

`ohe_train_df` with the given hyperparameter configuration. `LogisticRegression.fit` returns a LogisticRegressionModel.

Next, we will use the `LogisticRegressionModel.coefficients` and `LogisticRegressionModel.intercept` attributes to print out some details of the model's parameters.

Note that these are the names of the object's attributes and should be called using a syntax like `model.coefficients` for a given `model`.

In [44]:
```
# TODO: Replace <FILL IN> with appropriate code

standardization = False
elastic_net_param = 0.0
reg_param = .01
max_iter = 20

from pyspark.ml.classification import LogisticRegression

lr = LogisticRegression(
    featuresCol="features",
    labelCol="label",
    maxIter=max_iter,
    regParam=reg_param,
    elasticNetParam=elastic_net_param,
    standardization=standardization
)

lr_model_basic =  lr.fit(ohe_train_df)

print('intercept: {0}'.format(lr_model_basic.intercept))

print('length of coefficients: {0}'.format(len(lr_model_basic.coefficient

sorted_coefficients = sorted(lr_model_basic.coefficients)[:5]

print(sorted_coefficients)
```

```
25/11/12 14:35:26 WARN InstanceBuilder: Failed to load implementation fro
m:dev.ludovic.netlib.blas.JNIBLAS

intercept: -1.1775972524451497
length of coefficients: 234658
[np.float64(-0.11267143173152902), np.float64(-0.10163425803641195), np.fl
oat64(-0.10063997333013269), np.float64(-0.10045027889721173), np.float64
(-0.09975078904281737)]
```

In [45]:
```
# TEST Logistic regression (4a)

# Values updated for Spark 4. [A25]
testmti850.Test.assertTrue(np.allclose(lr_model_basic.intercept, -1.17759
testmti850.Test.assertTrue(np.allclose(sorted_coefficients, [-0.112671431

# Old tests for Spark 3.
#testmti850.Test.assertTrue(np.allclose(lr_model_basic.intercept, -1.2182
#testmti850.Test.assertTrue(np.allclose(sorted_coefficients, [-0.11580845
```

```
1 test passed.
1 test passed.
```

(4b) Log loss

Throughout this assignment, we will use log loss to evaluate the quality of models.

Log loss is defined as:

$$\ell_{log}(p, y) = \begin{cases} -\log(p) & \text{if } y = 1 \\ -\log(1-p) & \text{if } y = 0 \end{cases}$$

where p is a probability between 0 and 1 and y is a label of either 0 or 1.

Log loss is a standard evaluation criterion when predicting rare-events such as click-through rate prediction (it is also the criterion used in the Criteo Kaggle competition).

Write a function `add_log_loss` to a DataFrame, and evaluate it on some sample inputs. This does not require a UDF. You can perform conditional branching with DataFrame columns using when.

In [46]:
```
# Some example data
example_log_loss_df = spark.createDataFrame(
    [(.5, 1), (.5, 0), (.99, 1), (.99, 0), (.01, 1), (.01, 0), (1., 1), (
    ['p', 'label'] # columns
)
example_log_loss_df.show()
```

```
+----+-----+
|   p|label|
+----+-----+
| 0.5|    1|
| 0.5|    0|
|0.99|    1|
|0.99|    0|
|0.01|    1|
|0.01|    0|
| 1.0|    1|
| 0.0|    1|
| 1.0|    0|
+----+-----+
```

In [47]:
```
# TODO: Replace <FILL IN> with appropriate code

from pyspark.sql.functions import when, log, col

epsilon = 1e-16

def add_log_loss(df):
    """Computes and adds a 'log_loss' column to a DataFrame using 'p' and

    Note:
        log(0) is undefined, so when p is 0 we add a small value (epsilon
        p is 1 we subtract a small value (epsilon) from it.

    Args:
        df (DataFrame with 'p' and 'label' columns): A DataFrame with a p
```

```
            'p' and a 'label' column that corresponds to y in the log los

    Returns:
        DataFrame: A new DataFrame with an additional column called 'log_
        column contains the loss value as explained above.
    """

    safe_p = when(col('p') <= 0, epsilon).when(col('p') >= 1, 1 - epsilon
    log_loss_col = - (col('label') * log(safe_p) + (1 - col('label')) * l

    return df.withColumn('log_loss', log_loss_col)

add_log_loss(example_log_loss_df).show()
```

```
+----+-----+--------------------+
|   p|label|            log_loss|
+----+-----+--------------------+
| 0.5|    1|  0.6931471805599453|
| 0.5|    0|  0.6931471805599453|
|0.99|    1| 0.01005033585350145|
|0.99|    0|    4.60517018598809|
|0.01|    1|   4.605170185988091|
|0.01|    0| 0.01005033585350145|
| 1.0|    1|1.110223024625156...|
| 0.0|    1|  36.841361487904734|
| 1.0|    0|    36.7368005696771|
+----+-----+--------------------+
```

In [48]:
```
# TEST Log loss (4b)
log_loss_values = add_log_loss(example_log_loss_df).select('log_loss').rd
testmti850.Test.assertTrue(np.allclose(log_loss_values[:-2],
                                       [0.6931471805599451, 0.6931471805599451, 0.01
                                        4.605170185988081, 0.010050335853501338, -0.
testmti850.Test.assertTrue(not(any(map(lambda x: x is None, log_loss_valu
                                   'log loss needs to bound p away from 0 and 1 by epsilon')
```

```
1 test passed.
1 test passed.
```

(4c) Baseline log loss

Next we will use the function we wrote in Part (4b) to compute the baseline log loss on the training data.

A very simple yet natural baseline model is one where we always make the same prediction independent of the given datapoint, setting the predicted value equal to the fraction of training points that correspond to click-through events (i.e., where the label is one).

Compute this value (which is simply the mean of the training labels), and then use it to compute the training log loss for the baseline model.

> Note: you'll need to add a `p` column to the `ohe_train_df` DataFrame so that it can be used in your function from Part (4b). To represent a constant value as a column you can use the lit function to wrap the value.

In [49]:
```
# TODO: Replace <FILL IN> with appropriate code
# Note that our dataset has a very high click-through rate by design
# In practice click-through rate can be one to two orders of magnitude lo

from pyspark.sql.functions import lit, avg # creates a column with litera

class_one_frac_train = ohe_train_df.select(avg('label')).first()[0]
print('Training class one fraction = {0:.3f}'.format(class_one_frac_train

baseline_df = ohe_train_df.withColumn('p', lit(class_one_frac_train)).sel
log_loss_df = add_log_loss(baseline_df)

log_loss_tr_base = log_loss_df.select(avg('log_loss')).first()[0]
print('Baseline Train Logloss = {0:.3f}\n'.format(log_loss_tr_base))
```

```
Training class one fraction = 0.228
Baseline Train Logloss = 0.537
```

In [50]:
```
# TEST Baseline log loss (4c)
expected_frac = 0.22835759252074442
expected_log_loss = 0.5372841736106333

testmti850.Test.assertTrue(np.allclose(class_one_frac_train, expected_fra
testmti850.Test.assertTrue(np.allclose(log_loss_tr_base, expected_log_los
```

```
1 test passed.
1 test passed.
```

(4d) Predicted probability

In order to compute the log loss for the model we trained in Part (4a), we need to write code to generate predictions from this model. Write a function that computes the raw linear prediction from this logistic regression model and then passes it through a sigmoid function $\sigma(t) = (1+e^{-t})^{-1}$ to return the model's probabilistic prediction. Then compute probabilistic predictions on the training data.

Note that when incorporating an intercept into our predictions, we simply add the intercept to the value of the prediction obtained from the weights and features. Alternatively, if the intercept was included as the first weight, we would need to add a corresponding feature to our data where the feature has the value one. This is not the case here.

In [51]:
```
# TODO: Replace <FILL IN> with appropriate code
from pyspark.sql.types import DoubleType
from math import exp #  exp(-t) = e^-t

def add_probability(df, model):
    """Adds a probability column ('p') to a DataFrame given a model"""

    coefficients_broadcast = spark.sparkContext.broadcast(model.coefficie

    intercept = model.intercept

    def get_p(features):
        """Calculate the probability for an observation given a list of f
```

```
        Note:
            We'll bound our raw prediction between 20 and -20 for numeric

        Args:
            features: the features

        Returns:
            float: A probability between 0 and 1.
        """
        # Compute the raw value
        raw_prediction = float(features.dot(coefficients_broadcast.value)

        # Bound the raw value between 20 and -20
        raw_prediction = max(min(raw_prediction, 20), -20)

        # Return the probability
        p = 1.0 / (1.0 + exp(-raw_prediction))
        return p

    get_p_udf = udf(get_p, DoubleType())

    return df.withColumn('p', get_p_udf('features'))

add_probability_model_basic = lambda df: add_probability(df, lr_model_bas

training_predictions = add_probability_model_basic(ohe_train_df).cache()

training_predictions.show(5)
```

```
[Stage 128:>                                                        (0 +
1) / 1]
+-----+--------------------+--------------------+
|label|            features|                   p|
+-----+--------------------+--------------------+
|  0.0|(234658,[0,1,2,3,...|0.16596341053725525|
|  0.0|(234658,[0,1,2,3,...|0.12034039629457038|
|  0.0|(234658,[0,1,2,3,...| 0.1371762379486666|
|  0.0|(234658,[0,1,2,3,...|0.13056624370141812|
|  0.0|(234658,[0,1,2,3,...|0.17194892029175524|
+-----+--------------------+--------------------+
only showing top 5 rows
```

In [52]:
```
# TEST Predicted probability (4d)

# Updated to Spark 4. [A25]
expected = 19301.06209855696

# Old value for Spark 3.
# expected = 18333.7950991934

got = training_predictions.selectExpr('sum(p)').first()[0]
testmti850.Test.assertTrue(np.allclose(got, expected),
                'incorrect value for training_predictions. Got {0}, expec
```

```
1 test passed.
```

(4e) Evaluate the model

We are now ready to evaluate the quality of the model we trained in Part (4a). To do this, first write a general function that takes as input a model and data, and outputs the log loss.

Note that the log loss for multiple observations is the mean of the individual log loss values. Then run this function on the OHE training data, and compare the result with the baseline log loss.

In [54]:
```
# TODO: Replace <FILL IN> with appropriate code

def evaluate_results(df, model, baseline = None):
    """Calculates the log loss for the data given the model.

    Note:
        If baseline has a value the probability should be set to baseline
        the log loss is calculated.  Otherwise, use add_probability to ad
        appropriate probabilities to the DataFrame.

    Args:
        df (DataFrame with 'label' and 'features' columns): A DataFrame c
            labels and features.
        model (LogisticRegressionModel): A trained logistic regression mo
            can be None if baseline is set.
        baseline (float): A baseline probability to use for the log loss

    Returns:
        float: Log loss for the data.
    """
    if baseline is not None:
        with_probability_df = df.withColumn('p', lit(baseline))
    else:
        with_probability_df = add_probability(df, model)

    with_log_loss_df = add_log_loss(with_probability_df)

    log_loss = with_log_loss_df.select(avg('log_loss')).first()[0]

    return log_loss

log_loss_train_model_basic = evaluate_results(ohe_train_df, lr_model_basi

print ('OHE Features Train Logloss:\n\tBaseline = {0:.3f}\n\tLogReg = {1:
       .format(log_loss_tr_base, log_loss_train_model_basic))
```

```
[Stage 132:>                                                        (0 +
1) / 1]
OHE Features Train Logloss:
        Baseline = 0.537
        LogReg = 0.478
```

In [55]:
```
# TEST Evaluate the model (4e)

# Updated value for Spark 4. [A25]
expected_log_loss = 0.4783396327932997

# Old values for Spark 3.
# expected_log_loss = 0.4772118368120353
```

```
testmti850.Test.assertTrue(np.allclose(log_loss_train_model_basic, expect
                'incorrect value for log_loss_train_model_basic. Got {0},
expected_res = 0.693147180558829
res = evaluate_results(ohe_train_df, None,  0.5)
testmti850.Test.assertTrue(np.allclose(res, expected_res),
                'evaluate_results needs to handle baseline models. Got {0
```

```
1 test passed.
1 test passed.
```

(4f) Validation log loss

Next, using the `evaluate_results` function compute the validation log loss for both the baseline and logistic regression models.

Notably, the baseline model for the validation data should still be based on the label fraction from the training dataset.

In [56]:
```
# TODO: Replace <FILL IN> with appropriate code
log_loss_val_base = evaluate_results(ohe_validation_df, model=None, basel

log_loss_val_l_r0 = evaluate_results(ohe_validation_df, model=lr_model_ba

print ('OHE Features Validation Logloss:\n\tBaseline = {0:.3f}\n\tLogReg
       .format(log_loss_val_base, log_loss_val_l_r0))
```

```
OHE Features Validation Logloss:
        Baseline = 0.526
        LogReg = 0.471
```

In [57]:
```
# TEST Validation log loss (4f)
expected_val_base = 0.5256351707557996

testmti850.Test.assertTrue(np.allclose(log_loss_val_base, expected_val_ba
                'incorrect value for log_loss_val_base. Got {0}, expected

# Value updated for Spark 4. [A25]
expected_val_model_basic = 0.4711432904191549

# Old value for Spark 3.
# expected_val_model_basic = 0.4692840176843362

testmti850.Test.assertTrue(np.allclose(log_loss_val_l_r0, expected_val_mo
                'incorrect value for log_loss_val_l_r0. Got {0}, expected
```

```
1 test passed.
1 test passed.
```

Visualization 2: ROC curve

We will now visualize how well the model predicts our target. To do this we generate a plot of the ROC curve.

The ROC curve shows us the trade-off between the false positive rate and true positive rate, as we liberalize the threshold required to predict a positive outcome.

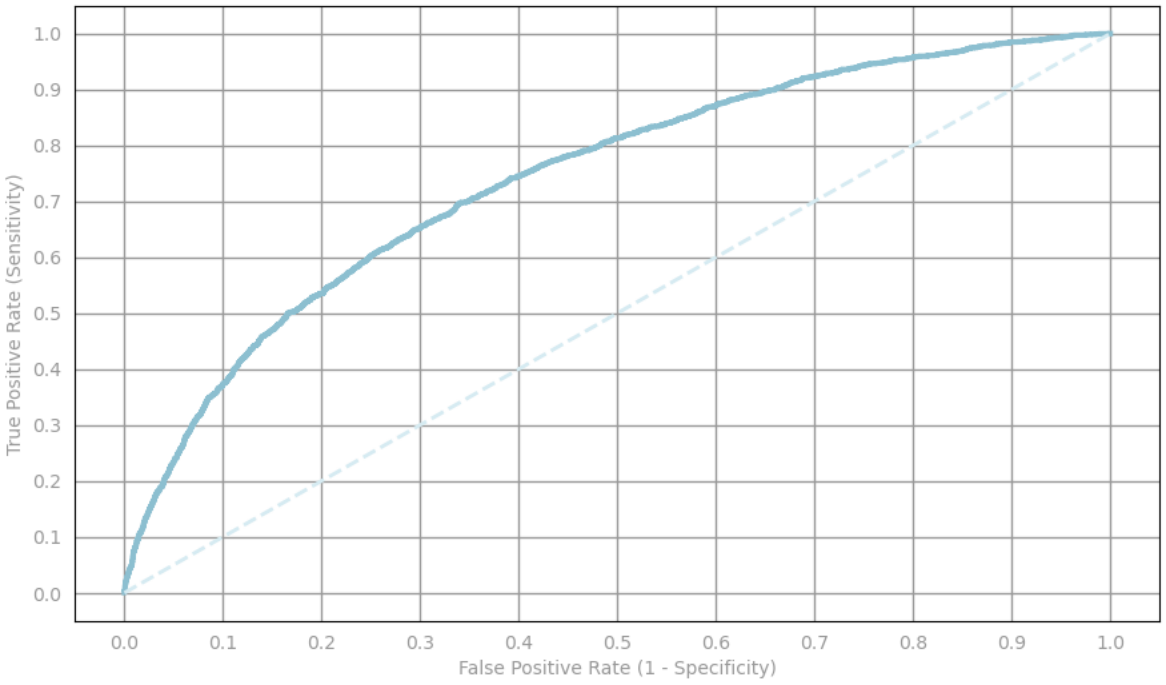
A random model is represented by the dashed line.

In [58]:
```
labels_and_scores = add_probability_model_basic(ohe_validation_df).select
labels_and_weights = labels_and_scores.collect()
labels_and_weights.sort(key=lambda kv: kv[1], reverse=True)
labels_by_weight = np.array([k for (k, v) in labels_and_weights])

length = labels_by_weight.size
true_positives = labels_by_weight.cumsum()
num_positive = true_positives[-1]
false_positives = np.arange(1.0, length + 1, 1.) - true_positives

true_positive_rate = true_positives / num_positive
false_positive_rate = false_positives / (length - num_positive)

# Generate layout and plot data
fig, ax = prepare_plot(np.arange(0., 1.1, 0.1), np.arange(0., 1.1, 0.1))
ax.set_xlim(-.05, 1.05), ax.set_ylim(-.05, 1.05)
ax.set_ylabel('True Positive Rate (Sensitivity)')
ax.set_xlabel('False Positive Rate (1 - Specificity)')
plt.plot(false_positive_rate, true_positive_rate, color='#8cbfd0', linest
plt.plot((0., 1.), (0., 1.), linestyle='--', color='#d6ebf2', linewidth=2
plt.show()
```



Part 5: Reduce feature dimension via feature hashing

(5a) Hash function

As we just saw, using a one-hot-encoding (OHE) featurization can yield a model with good statistical accuracy. However, the number of distinct categories across all features is quite large -- recall that we observed 234k categories in the training data in Part (3c).

Moreover, the full Kaggle training dataset includes more than 33M distinct categories, and the Kaggle dataset itself is just a small subset of Criteo's labeled data. Hence, featurizing via a OHE representation would lead to a very large feature vector. To reduce the dimensionality of the feature space, we will use feature hashing.

Below is the hash function that we will use for this part of the assignment. We will first use this hash function with the three sample data points from Part (1a) to gain some intuition. Specifically, run code to hash the three sample points using two different values for `numBuckets` and observe the resulting hashed feature dictionaries.

In [59]:
```
from collections import defaultdict
import hashlib

def hash_function(raw_feats, num_buckets, print_mapping=False):
    """Calculate a feature dictionary for an observation's features based

    Note:
        Use print_mapping=True for debug purposes and to better understan

    Args:
        raw_feats (list of (int, str)): A list of features for an observa
            (featureID, value) tuples.
        num_buckets (int): Number of buckets to use as features.
        print_mapping (bool, optional): If true, the mappings of featureS
            printed.

    Returns:
        dict of int to float:  The keys will be integers which represent
            features have been hashed to.  The value for a given key will
            (featureID, value) tuples that have hashed to that key.
    """
    encode = lambda s: s.encode('utf-8')

    mapping = { category + ':' + str(ind):
                int(int(hashlib.md5((category + ':' + str(ind)).encode('u
                for ind, category in raw_feats}

    if(print_mapping): print(mapping)

    sparse_features = defaultdict(float)

    for value in mapping.values():
        sparse_features[value] += 1

    return dict(sparse_features)

# Reminder of the sample values:
# sample_one = [(0, 'mouse'), (1, 'black')]
# sample_two = [(0, 'cat'), (1, 'tabby'), (2, 'mouse')]
```

```
# sample_three =  [(0, 'bear'), (1, 'black'), (2, 'salmon')]
```

In [60]:
```
# TODO: Replace <FILL IN> with appropriate code

# Use four buckets
samp_one_four_buckets = hash_function(sample_one, 4, True)
samp_two_four_buckets = hash_function(sample_two, 4,  True)
samp_three_four_buckets = hash_function(sample_three, 4, True)

# Use one hundred buckets
samp_one_hundred_buckets = hash_function(sample_one, 100, True)
samp_two_hundred_buckets = hash_function(sample_two, 100, True)
samp_three_hundred_buckets = hash_function(sample_three, 100, True)

print('\n\t\t 4 Buckets \t\t  100 Buckets')
print('SampleOne:\t {0}\t\t  {1}'.format(samp_one_four_buckets, samp_one_
print('SampleTwo:\t {0}\t  {1}'.format(samp_two_four_buckets, samp_two_hu
print('SampleThree:\t {0} {1}'.format(samp_three_four_buckets, samp_three
```

```
{'mouse:0': 3, 'black:1': 3}
{'cat:0': 1, 'tabby:1': 0, 'mouse:2': 1}
{'bear:0': 2, 'black:1': 3, 'salmon:2': 0}
{'mouse:0': 99, 'black:1': 51}
{'cat:0': 21, 'tabby:1': 72, 'mouse:2': 9}
{'bear:0': 82, 'black:1': 51, 'salmon:2': 80}

                 4 Buckets                 100 Buckets
SampleOne:       {3: 2.0}                  {99: 1.0, 51: 1.0}
SampleTwo:       {1: 2.0, 0: 1.0}          {21: 1.0, 72: 1.0, 9: 1.0}
SampleThree:     {2: 1.0, 3: 1.0, 0: 1.0} {82: 1.0, 51: 1.0, 80: 1.0}
```

In [61]:
```
# TEST Hash function (5a)
testmti850.Test.assertEquals(samp_one_four_buckets, {3: 2.0}, 'incorrect
testmti850.Test.assertEquals(samp_three_hundred_buckets, {80: 1.0, 82: 1.
                  'incorrect value for samp_three_hundred_buckets')
```

```
1 test passed.
1 test passed.
```

(5b) Creating hashed features

Next we will use this hash function to create hashed features for our CTR datasets. Use the provided UDF to create a function that takes in a DataFrame and returns labels and hashed features. Then use this function to create new training, validation and test datasets with hashed features.

In [63]:
```
# TODO: Replace <FILL IN> with appropriate code

from pyspark.ml.linalg import Vectors
num_hash_buckets = 2 ** 15

# UDF that returns a vector of hashed features given an Array of tuples
tuples_to_hash_features_udf = udf(lambda x: Vectors.sparse(num_hash_bucke
                                                           hash_function(

def add_hashed_features(df):
    """Return a DataFrame with labels and hashed features.
```

```
    Note:
        Make sure to cache the DataFrame that you are returning.

    Args:
        df (DataFrame with 'tuples' column): A DataFrame containing the t

    Returns:
        DataFrame: A DataFrame with a 'label' column and a 'features' col
            SparseVector of hashed features.
    """
    hashed_df = df.select(df['label'],tuples_to_hash_features_udf(df['fea
    return hashed_df.cache()

hash_train_df = add_hashed_features(parsed_train_df)

hash_validation_df = add_hashed_features(parse_raw_df(raw_validation_df))

hash_test_df = add_hashed_features(parse_raw_df(raw_test_df))

hash_train_df.show()
```

```
25/11/12 14:49:47 WARN CacheManager: Asked to cache already cached data.
[Stage 145:>                                                        (0 +
1) / 1]
+-----+--------------------+
|label|            features|
+-----+--------------------+
|  0.0|(32768,[1613,1659...|
|  0.0|(32768,[400,1613,...|
|  0.0|(32768,[492,1408,...|
|  0.0|(32768,[492,1408,...|
|  0.0|(32768,[1408,1613...|
|  0.0|(32768,[1408,1613...|
|  0.0|(32768,[274,590,1...|
|  0.0|(32768,[1408,1613...|
|  0.0|(32768,[300,1613,...|
|  0.0|(32768,[1613,2034...|
|  0.0|(32768,[40,463,16...|
|  0.0|(32768,[1408,1613...|
|  0.0|(32768,[1408,1613...|
|  0.0|(32768,[1418,1613...|
|  0.0|(32768,[1383,1613...|
|  0.0|(32768,[1613,1659...|
|  0.0|(32768,[980,1613,...|
|  0.0|(32768,[1613,2034...|
|  0.0|(32768,[1408,1613...|
|  0.0|(32768,[1613,1659...|
+-----+--------------------+
only showing top 20 rows
```

In [64]:
```
# TEST Creating hashed features (5b)
hash_train_df_feature_sum = sum(hash_train_df
                                .rdd
                                .map(lambda r: sum(r[1].indices))
                                .take(10))
hash_validation_df_feature_sum = sum(hash_validation_df
                                     .rdd
                                     .map(lambda r: sum(r[1].indices))
                                     .take(10))
```

```
hash_test_df_feature_sum = sum(hash_test_df
                                   .rdd
                                   .map(lambda r: sum(r[1].indices))
                                   .take(10))

expected_train_sum = 6569082
testmti850.Test.assertEquals(hash_train_df_feature_sum, expected_train_su
                  'incorrect number of features in hash_train_df. Got {0}

expected_validation_sum = 6923755
testmti850.Test.assertEquals(hash_validation_df_feature_sum, expected_val
                  'incorrect number of features in hash_validation_df. Go

expected_test_sum = 6635530
testmti850.Test.assertEquals(hash_test_df_feature_sum, expected_test_sum,
```

```
[Stage 148:>                                                        (0 +
1) / 1]
1 test passed.
1 test passed.
1 test passed.
```

(5c) Sparsity

Since we have 33k hashed features versus 233k OHE features, we should expect OHE features to be sparser. Verify this hypothesis by computing the average sparsity of the OHE and the hashed training datasets.

Note that if you have a `SparseVector` named `sparse`, calling `len(sparse)` returns the total number of features, not the number features with entries. `SparseVector` objects have the attributes `indices` and `values` that contain information about which features are nonzero.

Continuing with our example, these can be accessed using `sparse.indices` and `sparse.values`, respectively.

In [65]:
```
# TODO: Replace <FILL IN> with appropriate code
def vector_feature_sparsity(sparse_vector):
    """Calculates the sparsity of a SparseVector.

    Args:
        sparse_vector (SparseVector): The vector containing the features.

    Returns:
        float: The ratio of features found in the vector to the total num
    """
    return float(len(sparse_vector.indices)) / float(len(sparse_vector))

feature_sparsity_udf = udf(vector_feature_sparsity, DoubleType())

a_sparse_vector = Vectors.sparse(5, {0: 1.0, 3: 1.0})

a_sparse_vector_sparsity = vector_feature_sparsity(a_sparse_vector)

print('This vector should have sparsity 2/5 or .4.')
```

```
print('Sparsity = {0:.2f}.'.format(a_sparse_vector_sparsity))
```

```
This vector should have sparsity 2/5 or .4.
Sparsity = 0.40.
```

In [66]:
```
# TEST Sparsity (5c)
testmti850.Test.assertEquals(a_sparse_vector_sparsity, .4,
                'incorrect value for a_sparse_vector_sparsity')
```

```
1 test passed.
```

(5d) Sparsity continued

Now that we have a function to calculate vector sparsity, we will wrap it in a UDF and apply it to an entire DataFrame to obtain the average sparsity for features in that DataFrame.

We will use the function to find the average sparsity of the one-hot-encoded training DataFrame and of the hashed training DataFrame.

In [67]:
```
# TODO: Replace <FILL IN> with appropriate code
feature_sparsity_udf = udf(vector_feature_sparsity, DoubleType())

def get_sparsity(df):
    """Calculates the average sparsity for the features in a DataFrame.

    Args:
        df (DataFrame with 'features' column): A DataFrame with sparse fe

    Returns:
        float: The average feature sparsity.
    """
    return df.select(avg(feature_sparsity_udf(col('features')))).first()[

average_sparsity_ohe = get_sparsity(ohe_train_df)

average_sparsity_hash = get_sparsity(hash_train_df)

print('Average OHE Sparsity: {0:.7e}'.format(average_sparsity_ohe))

print('Average Hash Sparsity: {0:.7e}'.format(average_sparsity_hash))
```

```
[Stage 152:>                                                        (0 +
1) / 1]
Average OHE Sparsity: 1.6619932e-04
Average Hash Sparsity: 1.1896642e-03
```

In [68]:
```
# TEST Sparsity (5d)
expected_ohe = 1.6619932e-04
testmti850.Test.assertTrue(np.allclose(average_sparsity_ohe, expected_ohe
                'incorrect value for average_sparsity_ohe. Got {0}, expec
expected_hash = 1.1896642e-03
testmti850.Test.assertTrue(np.allclose(average_sparsity_hash, expected_ha
                'incorrect value for average_sparsity_hash. Got {0}, expe
```

```
1 test passed.
1 test passed.
```

(5e) Logistic model with hashed features

Now let's train a logistic regression model using the hashed training features. Use the hyperparameters provided, fit the model, and then evaluate the log loss on the training set.

In [70]:
```
# TODO: Replace <FILL IN> with appropriate code
standardization = False
elastic_net_param = 0.7
reg_param = .001
max_iter = 20

lr_hash = LogisticRegression(
    featuresCol="features",
    labelCol="label",
    maxIter=max_iter,
    regParam=reg_param,
    elasticNetParam=elastic_net_param,
    standardization=standardization
)

lr_model_hashed = lr_hash.fit(hash_train_df)

print('intercept: {0}'.format(lr_model_hashed.intercept))

print(len(lr_model_hashed.coefficients))

log_loss_train_model_hashed = evaluate_results(hash_train_df, lr_model_ha

print ('OHE Features Train Logloss:\n\tBaseline = {0:.3f}\n\thashed = {1:
       .format(log_loss_tr_base, log_loss_train_model_hashed))
```

```
intercept: -1.1284649878707151
32768
[Stage 180:>                                                        (0 +
1) / 1]
OHE Features Train Logloss:
        Baseline = 0.537
        hashed = 0.468
```

In [71]:
```
# TEST Logistic model with hashed features (5e)

# Update values for Spark 4. [A25]
expected = 0.4678969394413947

# Old value for Spark 3.
```

```
# expected = 0.468052083881297

testmti850.Test.assertTrue(np.allclose(log_loss_train_model_hashed, expec
                'incorrect value for log_loss_train_model_hashed. Got {0}
```

```
1 test passed.
```

(5f) Evaluate on the test set

Finally, evaluate the model from Part (5e) on the test set. Compare the resulting log loss with the baseline log loss on the test set, which can be computed in the same way that the validation log loss was computed in Part (4f).

In [76]:
```
# TODO: Replace <FILL IN> with appropriate code
# Log loss for the best model from (5e)

log_loss_test = evaluate_results(hash_test_df, lr_model_hashed)

# Log loss for the baseline model
class_one_frac_test = hash_test_df.select(avg('label')).first()[0]

print('Class one fraction for test data: {0}'.format(class_one_frac_test)

log_loss_test_baseline = evaluate_results(hash_test_df, model=None, basel

print ('Hashed Features Test Log Loss:\n\tBaseline = {0:.3f}\n\tLogReg =
       .format(log_loss_test_baseline, log_loss_test))
```

```
[Stage 201:>                                                        (0 +
1) / 1]
Class one fraction for test data: 0.22073146491751142
Hashed Features Test Log Loss:
        Baseline = 0.528
        LogReg = 0.459
```

In [77]:
```
# TEST Evaluate on the test set (5f)
expected_test_baseline = 0.5278321942418345
testmti850.Test.assertTrue(np.allclose(log_loss_test_baseline, expected_t
                'incorrect value for log_loss_test_baseline. Got {0}, exp

# Old values for Spark 4. [A25]
expected_test = 0.45898741847256846

# Old values for Spark 3.
# expected_test = 0.4590996248004832

testmti850.Test.assertTrue(np.allclose(log_loss_test, expected_test),
                'incorrect value for log_loss_test. Got {0}, expected {1}
```

```
1 test passed.
1 test passed.
```

Notebook Ended