

95-869: Big Data and Large-Scale Computing Homework 3

Click-Through Rate Prediction Assignment

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 recent [Kaggle competition](#).

About the Display Advertising Dataset

This dataset contains feature values and click feedback for millions of display ads. Its purpose is to benchmark algorithms for clickthrough rate (CTR) prediction.

The dataset consists of a portion of Criteo's traffic over a period of 7 days. Each row corresponds to a display ad served by Criteo and the first column indicates whether this ad has been clicked or not. The positive (clicked) and negatives (non-clicked) examples have both been subsampled (but at different rates) in order to reduce the dataset size.

There are 13 features taking integer values (mostly count features) and 26 categorical features. The values of the categorical features have been hashed onto 32 bits for anonymization purposes. The semantic of these features is undisclosed. Some features may have missing values.

The rows are chronologically ordered. When a value is missing, the field is just empty.

Dataset was assembled by Olivier Chapelle (o.chapelle@criteo.com)

This assignment will cover:

- *Part 1 (10 Points)*: Featurize categorical data using one-hot-encoding (OHE)
- *Part 2 (10 Points)*: Construct an OHE dictionary
- *Part 3 (10 Points)*: Parse CTR data and generate OHE features
 - *Visualization 1 (10 Points)*: Feature frequency
- *Part 4 (15 Points)*: CTR prediction and logloss evaluation
 - *Visualization 2 (15 Points)*: ROC curve
- *Part 5 (15 Points)*: Reduce feature dimension via feature hashing
 - *Visualization 3 (15 Points)*: Hyperparameter heat map

Note that, for reference, you can look up the details of the relevant Spark methods in [Spark's Python API](#) and the relevant NumPy methods in the [NumPy Reference](#)

Submission Instructions:

You will submit both a PDF and an IPython file on Canvas. No printout submission is expected.

Rename the notebook from "hw3_ctr_student.ipynb" to "groupid_hw3_ctr_student.ipynb" where "groupid" is your group id or your Andrew ID if you are working on your own. Complete the assignment, execute all cells in the completed notebook, and make sure all results show up. Export the contents of the notebook by choosing "File > Download as > HTML" and saving the resulting file as "groupid_hw3_ctr_student.html" Convert the exported HTML file to PDF by using a feature such as the "Save as PDF" feature on Mac. Submit the PDF solution to **gradescope**. Submit the IPython and exported PDF solution files on **Canvas**.

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 lab, we'll use OHE dictionaries to transform data points into compact lists of features that can be used in machine learning algorithms.

In [1]:

```
import os
import sys
import os.path
import warnings
warnings.filterwarnings('ignore')

sys.path.append("/opt/packages/spark/latest/python/lib/py4j-0.10.7-src.zip")
sys.path.append("/opt/packages/spark/latest/python/")
sys.path.append("/opt/packages/spark/latest/python/pyspark")
from pyspark import SparkConf, SparkContext
sc = SparkContext()
sc
```

Out[1]:

SparkContext

[Spark UI](#)

Version

v2.4.3

Master

local[*]

AppName

pyspark-shell

In [2]:

```
# Data for manual OHE
# Note: the first data point does not include any value for the optional third feature
sampleOne = [(0, 'mouse'), (1, 'black')]
sampleTwo = [(0, 'cat'), (1, 'tabby'), (2, 'mouse')]
sampleThree = [(0, 'bear'), (1, 'black'), (2, 'salmon')]
sampleDataRDD = sc.parallelize([sampleOne, sampleTwo, sampleThree])
```

In [3]:

```
# Manually construct OHE for all categorical features
sampleOHEDictManual = {}
sampleOHEDictManual[(0, 'bear')] = 0
sampleOHEDictManual[(0, 'cat')] = 1
sampleOHEDictManual[(0, 'mouse')] = 2
sampleOHEDictManual[(1, 'black')] = 3
sampleOHEDictManual[(1, 'tabby')] = 4
sampleOHEDictManual[(2, 'mouse')] = 5
sampleOHEDictManual[(2, 'salmon')] = 6
```

(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 of 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 a dictionary, a list of (index, value) pairs, or two separate arrays of indices and values (sorted by index). You'll need to create a sparse vector representation of each dense vector `aDense` and `bDense`.

In [4]:

```
import numpy as np
from pyspark.mllib.linalg import SparseVector
```

In [5]:

```
# TODO: Replace <FILL IN> with appropriate code
aDense = np.array([0., 3., 0., 4.])
aSparse = SparseVector(len(aDense), [(1,3), (3.,4.)])

bDense = np.array([0., 0., 0., 1.])
bSparse = SparseVector(len(bDense), [3], [1.0])

w = np.array([0.4, 3.1, -1.4, -.5])
print (aDense.dot(w))
print (aSparse.dot(w))
print (bDense.dot(w))
print (bSparse.dot(w))
```

```
7.3000000000000001
7.3000000000000001
-0.5
-0.5
```

(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]`.

In [6]:

```
# Reminder of the sample features
# sampleOne = [(0, 'mouse'), (1, 'black')]
# sampleTwo = [(0, 'cat'), (1, 'tabby'), (2, 'mouse')]
# sampleThree = [(0, 'bear'), (1, 'black'), (2, 'salmon')]
```

In [7]:

```
# TODO: Replace <FILL IN> with appropriate code
sampleOneOHEFeatManual = SparseVector(len(sampleOne)+len(sampleTwo)+len(sampleThree), [2,3], [1.0,1.0])
sampleTwoOHEFeatManual = SparseVector(len(sampleOne)+len(sampleTwo)+len(sampleThree), [1,4,5], [1.0,1.0,1.0])
sampleThreeOHEFeatManual = SparseVector(len(sampleOne)+len(sampleTwo)+len(sampleThree), [0,3,6], [1.0,1.0,1.0])
```

(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 `oneHotEncoding` 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).

In [8]:

```
# TODO: Replace <FILL IN> with appropriate code
def oneHotEncoding(rawFeats, OHEDict):
    """Produce a one-hot-encoding from a list of features and an OHE dictionary.

    Note:
        You should ensure that the indices used to create a SparseVector are sorted.

    Args:
        rawFeats (list of (int, str)): The features corresponding to a single observation. Each
            feature consists of a tuple of featureID and the feature's value. (e.g. sampleOne)
        OHEDict (dict): A mapping of (featureID, value) to unique integer.

    Returns:
        SparseVector: A SparseVector of length numOHEFeats with indicies equal to the unique
            identifiers for the (featureID, value) combinations that occur in the observation and
            with values equal to 1.0. Here numOHEFeats is the total number of unique OHE features
            (combinations of featureID and value).

    """
```

```

feature_list = []
for feature in rawFeats:
    feature_list.append((OHEDict[feature],1))
return SparseVector(len(OHEDict),feature_list)

# Run oneHotEncoding on sampleOne
sampleOneOHEFeat = oneHotEncoding(sampleOne, sampleOHEDictManual)

print (sampleOneOHEFeat)

```

```
(7, [2, 3], [1.0, 1.0])
```

(1e) Apply OHE to a dataset

Finally, use the function from Part (1d) to create OHE features for all 3 data points in the sample dataset.

In [9]:

```

# TODO: Replace <FILL IN> with appropriate code
sampleOHEData = sampleDataRDD.map(lambda x:oneHotEncoding(x,sampleOHEDictManual))
print (sampleOHEData.collect())

```

```
[SparseVector(7, {2: 1.0, 3: 1.0}), SparseVector(7, {1: 1.0, 4: 1.0, 5: 1.0}), SparseVector(7, {0: 1.0, 3: 1.0, 6: 1.0})]
```

Part 2: Construct an OHE dictionary

(2a) Pair RDD of (featureID, category)

To start, create an RDD of distinct (featureID, category) tuples. In our sample dataset, the 7 items in the resulting RDD 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 RDD: (1, 'black'), while 'mouse' also appears twice and contributes two items: (0, 'mouse') and (2, 'mouse'). Use [flatMap](#) and [distinct](#).

In [10]:

```

# TODO: Replace <FILL IN> with appropriate code
sampleDistinctFeats = sampleDataRDD.flatMap(lambda x:x).distinct()
print(sampleDistinctFeats.collect())

```

```
[(2, 'mouse'), (0, 'bear'), (1, 'black'), (1, 'tabby'), (0, 'mouse'), (2, 'salmon'), (0, 'cat')]
```

(2b) OHE Dictionary from distinct features

Next, create an RDD of key-value tuples, where each (featureID, category) tuple in sampleDistinctFeats 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.

In [11]:

```

# TODO: Replace <FILL IN> with appropriate code
sampleOHEDict = sampleDistinctFeats.zipWithIndex().collectAsMap()
print (sampleOHEDict)

```

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

(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 [12]:

```
# TODO: Replace <FILL IN> with appropriate code
def createOneHotDict(inputData):
    """Creates a one-hot-encoder dictionary based on the input data.

    Args:
        inputData (RDD of lists of (int, str)): An RDD of observations where each observation is
            made up of a list of (featureID, value) tuples.

    Returns:
        dict: A dictionary where the keys are (featureID, value) tuples and map to values that are
            unique integers.
    """

    return inputData.flatMap(lambda x:x).distinct().zipWithIndex().collectAsMap()

sampleOHEDictAuto = createOneHotDict(sampleDataRDD)
print (sampleOHEDictAuto)
```

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

Part 3: Parse CTR data and generate OHE features

Before we can proceed, let's load the data from Criteo. If you want to see what it looks like, you can check [here](#).

In [13]:

```
numPartitions = 10

rawData = sc.textFile('file:///pylon5/ci5619p/benh/dac_bigdata.txt', numPartitions)
rawData = rawData.map(lambda x: x.strip().replace('\t', ','))
```

(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 RDDs storing each of these datasets, and then [cache](#) each of these RDDs, as we will be accessing them multiple times in the remainder of this lab. Finally, compute the size of each dataset.

In [14]:

```
# TODO: Replace <FILL IN> with appropriate code
weights = [.8, .1, .1]
seed = 42
# Use randomSplit with weights and seed
rawTrainData, rawValidationData, rawTestData = rawData.randomSplit(weights, seed)
# Cache the data
rawTrainData.cache()
rawValidationData.cache()
rawTestData.cache()

nTrain = rawTrainData.count()
nVal = rawValidationData.count()
nTest = rawTestData.count()
print (nTrain, nVal, nTest, nTrain + nVal + nTest)
print (rawData.take(1))
```

```
80066 10020 9914 100000
['0,1,1,5,0,1382,4,15,2,181,1,2,,2,68fd1e64,80e26c9b,fb936136,7b4723c4,25c83c98,7e0ccccc,de7995b8,1
562,a73ee510,a8cd5504,b2cb9c98,37c9c164,2824a5f6,1adce6ef,8ba8b39a,891b62e7,e5ba7672,f54016b9,21ddc
b1252a9d,07b5194c,,3a171ecb,c5c50484,e8b83407,9727dd16']
```

(3b) Extract features

We will now parse the raw training data to create an RDD that we can subsequently use to create an OHE dictionary. Note from the `take()` 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 implementation of the `parsePoint` function.

In [15]:

```
# TODO: Replace <FILL IN> with appropriate code
def parsePoint(point):
    """Converts a comma separated string into a list of (featureID, value) tuples.

    Note:
        featureIDs should start at 0 and increase to the number of features - 1.

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

    Returns:
        list: A list of (featureID, value) tuples.
    """
    split_data = point.split(',')
    featureID = 0
    features = []
    for data in split_data:
        features.append((featureID, data))
        featureID += 1
    return features

parsedTrainFeat = rawTrainData.map(parsePoint)

numCategories = (parsedTrainFeat
    .flatMap(lambda x: x)
    .distinct()
    .map(lambda x: (x[0], 1))
    .reduceByKey(lambda x, y: x + y)
    .sortByKey()
    .collect())

print (numCategories[2][1])
```

841

(3c) Create an OHE dictionary from the dataset

Note that `parsePoint` 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 lab. Using this observation, create an OHE dictionary using the function implemented in Part (2c). Note that we will assume for simplicity that all features in our CTR dataset are categorical.

In [16]:

```
# TODO: Replace <FILL IN> with appropriate code
ctrOHEDict = createOneHotDict(rawTrainData.map(parsePoint))
numCtrOHEFeats = len(ctrOHEDict.keys())
print (numCtrOHEFeats)
print (ctrOHEDict[(0, '')])
```

234093

93886

(3d) Apply OHE to the dataset

Now let's use this OHE dictionary by starting with the raw training data and creating an RDD of `LabeledPoint` objects using OHE features. To do this, complete the implementation of the `parseOHEPoint` function. Hint: `parseOHEPoint` is an extension of the `parsePoint` function from Part (3b) and it uses the `oneHotEncoding` function from Part (1d).

In [17]:

In [18]:

Visualization 1: Feature frequency

In [19]:

```
from operator import add

def bucketFeatByCount(featCount):
    """Bucket the counts by powers of two."""
    for i in range(11):
        size = 2 ** i
        if featCount <= size:
            return size
    return -1

featCounts = (OHETrainData
               .flatMap(lambda lp: lp.features.indices)
               .map(lambda x: (x, 1))
               .reduceByKey(add))
```

```

featCountsBuckets = (featCounts
    .map(lambda x: (bucketFeatByCount(x[1]), 1))
    .filter(lambda t: t[0] != -1)
    .reduceByKey(add)
    .collect())
print (featCountsBuckets)

```

```

[(1, 163763), (32, 4780), (512, 415), (2, 23714), (1024, 257), (64, 2618), (4, 16818), (16, 7742),
(256, 742), (128, 1490), (8, 11465)]

```

In [21]:

```

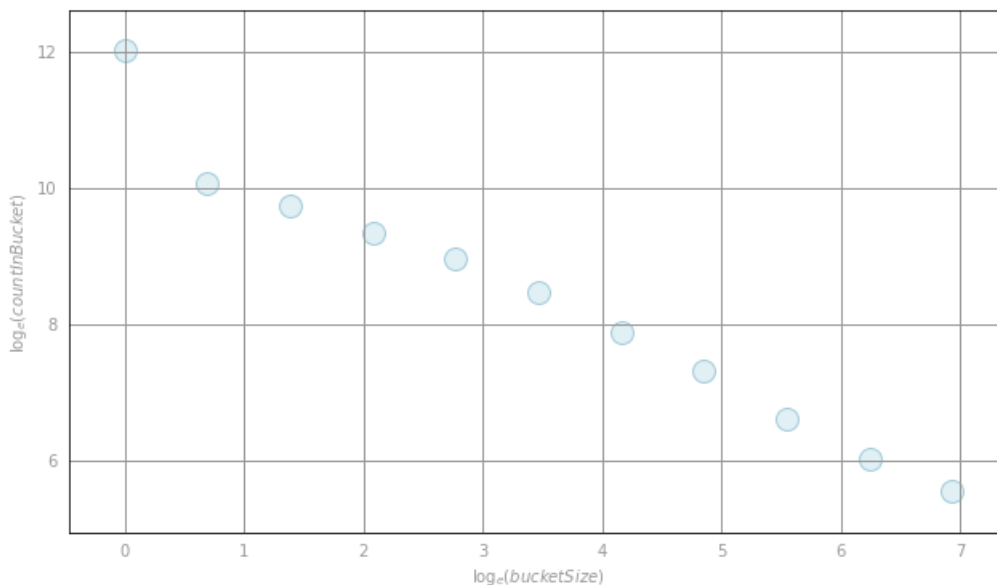
import matplotlib.pyplot as plt

x, y = zip(*featCountsBuckets)
x, y = np.log(x), np.log(y)

def preparePlot(xticks, yticks, figsize=(10.5, 6), hideLabels=False, gridColor='#999999',
    gridWidth=1.0):
    """Template for generating the plot layout."""
    plt.close()
    fig, ax = plt.subplots(figsize=figsize, facecolor='white', edgecolor='white')
    ax.axes.tick_params(labelcolor='#999999', labelsz='10')
    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 hideLabels: axis.set_ticklabels([])
    plt.grid(color=gridColor, linewidth=gridWidth, linestyle='-')
    map(lambda position: ax.spines[position].set_visible(False), ['bottom', 'top', 'left', 'right'])
    return fig, ax

# generate layout and plot data
fig, ax = preparePlot(np.arange(0, 10, 1), np.arange(4, 14, 2))
ax.set_xlabel(r'$\log_e(bucketSize)$'), ax.set_ylabel(r'$\log_e(countInBucket)$')
plt.scatter(x, y, s=14**2, c='#d6ebf2', edgecolors='#8cbfd0', alpha=0.75)
plt.show()

```



(3e) Handling unseen features

We naturally would like to repeat the process from Part (3d), e.g., 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 `oneHotEncoding()` function from Part (1d) to ignore previously unseen categories, and then compute OHE features for the validation data.

In [22]:

```

# TODO: Replace <FILL IN> with appropriate code
def oneHotEncoding(rawFeats, OHEDict):

```


(4b) Log loss

Throughout this lab, we will use log loss to evaluate the quality of models. Log loss is defined as:
$$\text{loss}_{\text{log}}(p, y) = \begin{cases} -\log(p) & \text{if } y = 1 \\ -\log(1-p) & \text{if } y = 0 \end{cases}$$
 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 to compute log loss, and evaluate it on some sample inputs.

In [26]:

```
# TODO: Replace <FILL IN> with appropriate code
from math import log

def computeLogLoss(p, y):
    """Calculates the value of log loss for a given probability and label.

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

    Args:
        p (float): A probability between 0 and 1.
        y (int): A label. Takes on the values 0 and 1.

    Returns:
        float: The log loss value.
    """
    epsilon = 10e-12
    if p == 1:
        p -= epsilon
    elif p == 0:
        p += epsilon

    if y == 1:
        return -log(p)
    else:
        return -log(1-p)

print (computeLogLoss(.5, 1))
print (computeLogLoss(.5, 0))
print (computeLogLoss(.99, 1))
print (computeLogLoss(.99, 0))
print (computeLogLoss(.01, 1))
print (computeLogLoss(.01, 0))
print (computeLogLoss(0, 1))
print (computeLogLoss(0, 0))
print (computeLogLoss(.25, 1))
print (computeLogLoss(.25, 0))
print (computeLogLoss(1, 1))
print (computeLogLoss(1, 0))
```

```
0.6931471805599453
0.6931471805599453
0.01005033585350145
4.605170185988091
4.605170185988091
0.01005033585350145
25.328436022934504
1.000000082745371e-11
1.3862943611198906
0.2876820724517809
1.000000082745371e-11
25.328435940194137
```

(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. The log loss for multiple observations is the mean of the individual log loss values.

In [27]:

```
# 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 lower
classOneFracTrain = OHETrainData.map(lambda x:x.label).mean()
print (classOneFracTrain)

logLossTrBase = OHETrainData.map(lambda x:computeLogLoss(classOneFracTrain, x.label)).mean()
print ('Baseline Train Logloss = {0:.3f}\n'.format(logLossTrBase))
```

0.2272375290385432

Baseline Train Logloss = 0.536

(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 [28]:

```
# TODO: Replace <FILL IN> with appropriate code
from math import exp # exp(-t) = e^-t

def getP(x, w, intercept):
    """Calculate the probability for an observation given a set of weights and intercept.

    Note:
        We'll bound our raw prediction between 20 and -20 for numerical purposes.

    Args:
        x (SparseVector): A vector with values of 1.0 for features that exist in this
            observation and 0.0 otherwise.
        w (DenseVector): A vector of weights (betas) for the model.
        intercept (float): The model's intercept.

    Returns:
        float: A probability between 0 and 1.
    """
    rawPrediction = (1 + exp(-(w.dot(x) + intercept)))**(-1)

    # Bound the raw prediction value
    rawPrediction = min(rawPrediction, 20)
    rawPrediction = max(rawPrediction, -20)
    return rawPrediction

trainingPredictions = OHETrainData.map(lambda x:getP(x.features, model0.weights, model0.intercept))

print (trainingPredictions.take(5))
```

[0.29127805566633835, 0.10213812172552388, 0.29253604392199584, 0.16861928189218603, 0.5719744057299744]

(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. Then run this function on the OHE training data, and compare the result with the baseline log loss.

In [29]:

```
# TODO: Replace <FILL IN> with appropriate code
def evaluateResults(model, data):
    """Calculates the log loss for the data given the model.

    Args:
```

```

    """
    model (LogisticRegressionModel): A trained logistic regression model.
    data (RDD of LabeledPoint): Labels and features for each observation.

    Returns:
        float: Log loss for the data.
    """
    return data.map(lambda x: computeLogLoss(getP(x.features, model.weights, model.intercept), x.label)).mean()

logLossTrLR0 = evaluateResults(model0, OHETrainData)
print ('OHE Features Train Logloss:\n\tBaseline = {0:.3f}\n\tLogReg = {1:.3f}'
        .format(logLossTrBase, logLossTrLR0))

```

```

OHE Features Train Logloss:
Baseline = 0.536
LogReg = 0.457

```

(4f) Validation log loss

Next, following the same logic as in Parts (4c) and 4(e), 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 [30]:

```

# TODO: Replace <FILL IN> with appropriate code
logLossValBase = OHEValidationData.map(lambda x: computeLogLoss(classOneFracTrain, x.label)).mean()

logLossValLR0 = evaluateResults(model0, OHEValidationData)
print ('OHE Features Validation Logloss:\n\tBaseline = {0:.3f}\n\tLogReg = {1:.3f}'
        .format(logLossValBase, logLossValLR0))

```

```

OHE Features Validation Logloss:
Baseline = 0.530
LogReg = 0.456

```

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 [31]:

```

labelsAndScores = OHEValidationData.map(lambda lp:
                                          (lp.label, getP(lp.features, model0.weights, model0.int
cept)))
labelsAndWeights = labelsAndScores.collect()
labelsAndWeights.sort(key=lambda t: t[1], reverse=True)
labelsByWeight = np.array([k for (k, v) in labelsAndWeights])

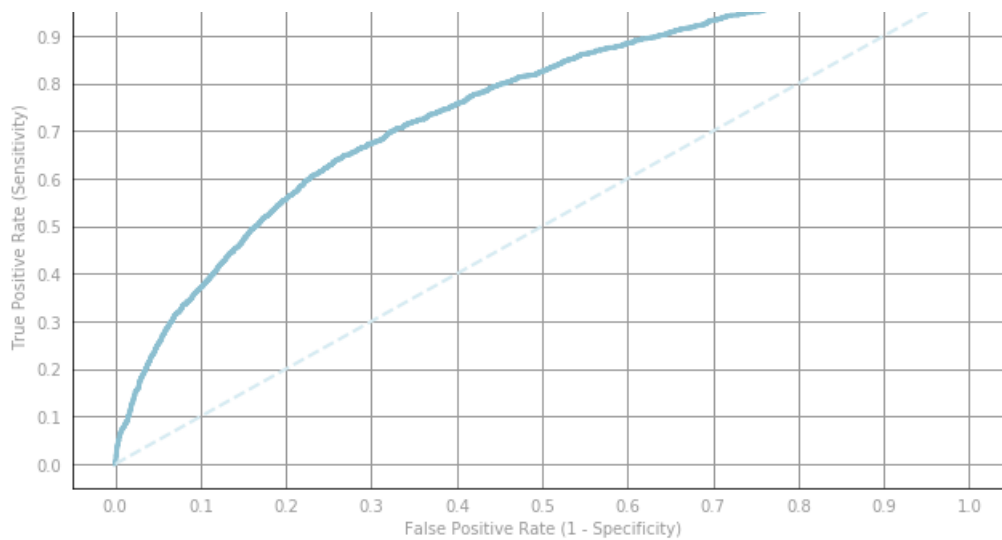
length = labelsByWeight.size
truePositives = labelsByWeight.cumsum()
numPositive = truePositives[-1]
falsePositives = np.arange(1.0, length + 1, 1.) - truePositives

truePositiveRate = truePositives / numPositive
falsePositiveRate = falsePositives / (length - numPositive)

# Generate layout and plot data
fig, ax = preparePlot(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(falsePositiveRate, truePositiveRate, color='#8cbfd0', linestyle='-', linewidth=3.)
plt.plot((0., 1.), (0., 1.), linestyle='--', color='#d6ebf2', linewidth=2.) # Baseline model
plt.show()

```





Part 5: Reduce feature dimension via feature hashing

(5a) Hash function

As we just saw, using a one-hot-encoding 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 one-hot-encoding 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 lab. 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 [32]:

```
from collections import defaultdict
import hashlib

def hashFunction(numBuckets, rawFeats, printMapping=False):
    """Calculate a feature dictionary for an observation's features based on hashing.

    Note:
        Use printMapping=True for debug purposes and to better understand how the hashing works.

    Args:
        numBuckets (int): Number of buckets to use as features.
        rawFeats (list of (int, str)): A list of features for an observation. Represented as
            (featureID, value) tuples.
        printMapping (bool, optional): If true, the mappings of featureString to index will be
            printed.

    Returns:
        dict of int to float: The keys will be integers which represent the buckets that the
            features have been hashed to. The value for a given key will contain the count of the
            (featureID, value) tuples that have hashed to that key.
    """
    mapping = {}
    for ind, category in rawFeats:
        featureString = category + str(ind)
        mapping[featureString] = int(int(hashlib.md5(featureString.encode('utf-8')).hexdigest(), 16)
    ) % numBuckets
    if printMapping: print(mapping)
    sparseFeatures = defaultdict(float)
    for bucket in mapping.values():
        sparseFeatures[bucket] += 1.0
    return dict(sparseFeatures)

# Reminder of the sample values:
# sampleOne = [(0, 'mouse'), (1, 'black')]
# sampleTwo = [(0, 'cat'), (1, 'tabby'), (2, 'mouse')]
# sampleThree = [(0, 'bear'), (1, 'black'), (2, 'salmon')]
```

In [33]:

```
# TODO: Replace <FILL IN> with appropriate code
# Use five buckets
sampOneFiveBuckets = hashFunction(5, sampleOne, True)
sampTwoFiveBuckets = hashFunction(5, sampleTwo, True)
sampThreeFiveBuckets = hashFunction(5, sampleThree, True)

# Use one hundred buckets
sampOneHundredBuckets = hashFunction(100, sampleOne, True)
sampTwoHundredBuckets = hashFunction(100, sampleTwo, True)
sampThreeHundredBuckets = hashFunction(100, sampleThree, True)

print ('\t\t 5 Buckets \t\t 100 Buckets')
print ('SampleOne:\t {0}\t\t {1}'.format(sampOneFiveBuckets, sampOneHundredBuckets))
print ('SampleTwo:\t {0}\t\t {1}'.format(sampTwoFiveBuckets, sampTwoHundredBuckets))
print ('SampleThree:\t {0}\t\t {1}'.format(sampThreeFiveBuckets, sampThreeHundredBuckets))
```

```
{'mouse0': 1, 'black1': 4}
{'cat0': 0, 'tabby1': 1, 'mouse2': 2}
{'bear0': 2, 'black1': 4, 'salmon2': 0}
{'mouse0': 31, 'black1': 14}
{'cat0': 40, 'tabby1': 16, 'mouse2': 62}
{'bear0': 72, 'black1': 14, 'salmon2': 5}
5 Buckets      100 Buckets
SampleOne: {1: 1.0, 4: 1.0}      {31: 1.0, 14: 1.0}
SampleTwo:  {0: 1.0, 1: 1.0, 2: 1.0}      {40: 1.0, 16: 1.0, 62: 1.0}
SampleThree: {2: 1.0, 4: 1.0, 0: 1.0}      {72: 1.0, 14: 1.0, 5: 1.0}
```

(5b) Creating hashed features

Next we will use this hash function to create hashed features for our CTR datasets. First write a function that uses the hash function from Part (5a) with $\text{numBuckets} = 2^{15} \approx 33\text{K}$ to create a `LabeledPoint` with hashed features stored as a `SparseVector`. Then use this function to create new training, validation and test datasets with hashed features. Hint: `parsedHashPoint` is similar to `parseOHEPoint` from Part (3d).

In [34]:

```
# TODO: Replace <FILL IN> with appropriate code
def parseHashPoint(point, numBuckets):
    """Create a LabeledPoint for this observation using hashing.

    Args:
        point (str): A comma separated string where the first value is the label and the rest are
            features.
        numBuckets: The number of buckets to hash to.

    Returns:
        LabeledPoint: A LabeledPoint with a label (0.0 or 1.0) and a SparseVector of hashed
            features.
    """
    features = hashFunction(numBuckets, parsePoint(point))
    return LabeledPoint(point.split(',')[0], SparseVector(numBuckets, features))

numBucketsCTR = 2 ** 15
hashTrainData = rawTrainData.map(lambda x: parseHashPoint(x, numBucketsCTR))
hashTrainData.cache()
hashValidationData = rawValidationData.map(lambda x: parseHashPoint(x, numBucketsCTR))
hashValidationData.cache()
hashTestData = rawTestData.map(lambda x: parseHashPoint(x, numBucketsCTR))
hashTestData.cache()

print (hashTrainData.take(1))
```

[illegible]

(5c) Sparsity

Since we have 33K hashed features versus 234K 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 [35]:

```
# TODO: Replace <FILL IN> with appropriate code
def computeSparsity(data, d, n):
    """Calculates the average sparsity for the features in an RDD of LabeledPoints.

    Args:
        data (RDD of LabeledPoint): The LabeledPoints to use in the sparsity calculation.
        d (int): The total number of features.
        n (int): The number of observations in the RDD.

    Returns:
        float: The average of the ratio of features in a point to total features.
    """
    return data.map(lambda x:len(x.features.indices)/d).mean()

averageSparsityHash = computeSparsity(hashTrainData, numBucketsCTR, nTrain)
averageSparsityOHE = computeSparsity(OHETrainData, numCtrOHEFeats, nTrain)

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

```
Average OHE Sparsity: 1.6293624e-04
Average Hash Sparsity: 1.1556060e-03
```

(5d) Logistic model with hashed features

Now let's train a logistic regression model using the hashed features. Run a grid search to find suitable hyperparameters for the hashed features, evaluating via log loss on the validation data. Note: This may take a few minutes to run. Use 1 and 10 for `stepSizes` and `1e-6` and `1e-3` for `regParams`.

In [36]:

```
numIters = 500
regType = 'l2'
includeIntercept = True

# Initialize variables using values from initial model training
bestModel = None
bestLogLoss = 1e10
```

In [37]:

```
# TODO: Replace <FILL IN> with appropriate code
stepSizes = [1,10]
regParams = [1e-6,1e-3]
for stepSize in stepSizes:
    for regParam in regParams:
        model = (LogisticRegressionWithSGD
                  .train(hashTrainData, numIters, stepSize, regParam=regParam, regType=regType,
                        intercept=includeIntercept))
        logLossVa = evaluateResults(model, hashValidationData)
        print ('\tstepSize = {0:.1f}, regParam = {1:.0e}: logloss = {2:.3f}'
              .format(stepSize, regParam, logLossVa))
        if (logLossVa < bestLogLoss):
            bestModel = model
            bestLogLoss = logLossVa

print ('Hashed Features Validation Logloss:\n\tBaseline = {0:.3f}\n\tLogReg = {1:.3f}'
      .format(logLossValBase, bestLogLoss))
```

```

stepSize = 1.0, regParam = 1e-06: logloss = 0.474
stepSize = 1.0, regParam = 1e-03: logloss = 0.474
stepSize = 10.0, regParam = 1e-06: logloss = 0.449
stepSize = 10.0, regParam = 1e-03: logloss = 0.451
Hashed Features Validation Logloss:
Baseline = 0.530
LogReg = 0.449

```

Visualization 3: Hyperparameter heat map

We will now perform a visualization of an extensive hyperparameter search. Specifically, we will create a heat map where the brighter colors correspond to lower values of `logLoss`.

The search was run using six step sizes and six values for regularization, which required the training of thirty-six separate models. We have included the results below, but omitted the actual search to save time.

In [38]:

```

from matplotlib.colors import LinearSegmentedColormap

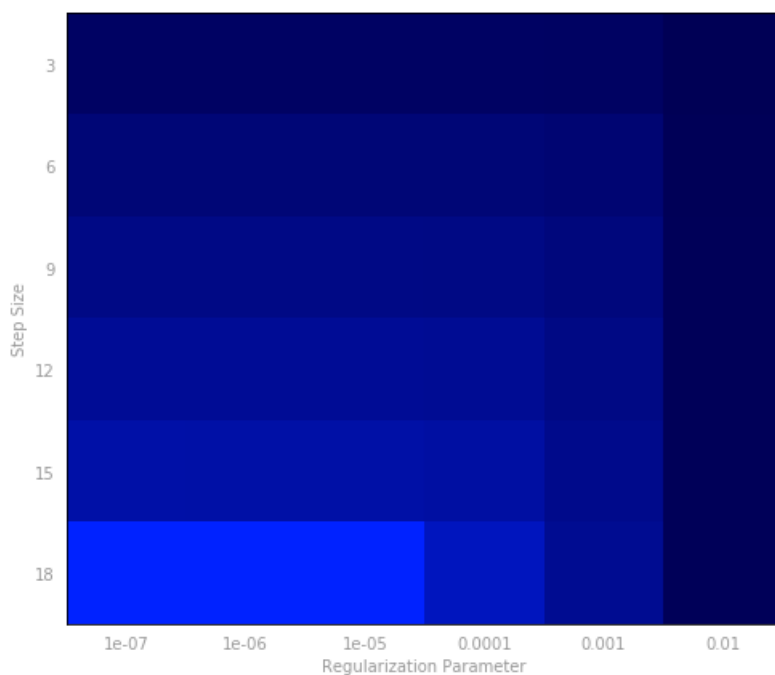
# Saved parameters and results. Eliminate the time required to run 36 models
stepSizes = [3, 6, 9, 12, 15, 18]
regParams = [1e-7, 1e-6, 1e-5, 1e-4, 1e-3, 1e-2]
logLoss = np.array([[ 0.45808431,  0.45808493,  0.45809113,  0.45815333,  0.45879221,  0.46556321],
                    [ 0.45188196,  0.45188306,  0.4518941,   0.4520051,   0.45316284,  0.46396068],
                    [ 0.44886478,  0.44886613,  0.44887974,  0.44902096,  0.4505614,   0.46371153],
                    [ 0.44706645,  0.4470698,   0.44708102,  0.44724251,  0.44905525,  0.46366507],
                    [ 0.44588848,  0.44589365,  0.44590568,  0.44606631,  0.44807106,  0.46365589],
                    [ 0.44508948,  0.44509474,  0.44510274,  0.44525007,  0.44738317,  0.46365405]])

numRows, numCols = len(stepSizes), len(regParams)
logLoss = np.array(logLoss)
logLoss.shape = (numRows, numCols)

fig, ax = preparePlot(np.arange(0, numCols, 1), np.arange(0, numRows, 1), figsize=(8, 7),
                      hideLabels=True, gridWidth=0.)
ax.set_xticklabels(regParams), ax.set_yticklabels(stepSizes)
ax.set_xlabel('Regularization Parameter'), ax.set_ylabel('Step Size')

colors = LinearSegmentedColormap.from_list('blue', ['#0022ff', '#000055'], gamma=.2)
image = plt.imshow(logLoss, interpolation='nearest', aspect='auto',
                   cmap = colors)
plt.show()

```



(5e) Evaluate on the test set

Finally, evaluate the best model from Part (5d) 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 [41]:

```
# TODO: Replace <FILL IN> with appropriate code
# Log loss for the best model from (5d)
logLossTest = evaluateResults(LogisticRegressionWithSGD.train(hashTrainData,
                                                                numIters, step = 18, regParam = 1e-7,
                                                                regType = regType, intercept =
                                                                includeIntercept)
                              , hashTestData)

# Log loss for the baseline model
logLossTestBaseline = hashTestData.map(lambda x: computeLogLoss(classOneFracTrain, x.label)).mean()

print ('Hashed Features Test Log Loss:\n\tBaseline = {0:.3f}\n\tLogReg = {1:.3f}'
      .format(logLossTestBaseline, logLossTest))
```

Hashed Features Test Log Loss:

Baseline = 0.535

LogReg = 0.451

In []: