Million Song dataset

This notebook covers a common supervised learning pipeline, using a subset of the Million Song Dataset (http://labrosa.ee.columbia.edu/millionsong/) from the UCI Machine Learning Repository (https://archive.ics.uci.edu/ml/datasets/YearPredictionMSD). The goal is to train a linear regression model to predict the release year of a song given a set of audio features.

Part 1: Read and parse the initial dataset

(1a) Load and check the data

The raw data is currently stored in text file. This raw data will be stored as an RDD, with each element of the RDD representing a data point as a comma-delimited string. Each string starts with the label (a year) followed by numerical audio features.

In [33]:

```
# load testing library
from test_helper import Test
import os.path

baseDir = os.path.join('data')
inputPath = os.path.join('cs190', 'millionsong.txt')
fileName = os.path.join(baseDir, inputPath)

numPartitions = 2
rawData = sc.textFile(fileName, numPartitions)
numPoints = rawData.count()
print numPoints
samplePoints = rawData.take(5)
print samplePoints[1]
```

```
6724
2001.0,0.854411946129,0.604124786151,0.593634078776,0.495885413963
,0.266307830936,0.261472105188,0.506387076327,0.464453565511,0.665
798573683,0.542968988766,0.58044428577,0.445219373624
```

(1b) Using `LabeledPoint`

The labeled training instances are stored using the [LabeledPoint]. The features are the audio attributes and the label is the year of the song.

In [16]:

```
from pyspark.mllib.regression import LabeledPoint
import numpy as np
```

In [43]:

```

def parsePoint(line):
    """Converts a comma separated unicode string into a `LabeledPoint`.

    Args:
        line (unicode): Comma separated unicode string where the first element
is the label and the
            remaining elements are features.

    Returns:
        LabeledPoint: The line is converted into a `LabeledPoint`, which consi
sts of a label and
            features.
    """
    tokens = line.split(',')
    label, features = tokens[0], tokens[1:]
    return LabeledPoint(label, features)


 parsedSamplePoints = [parsePoint(x) for x in samplePoints]
firstPointFeatures = parsedSamplePoints[0].features
firstPointLabel = parsedSamplePoints[0].label
print firstPointFeatures, firstPointLabel

d = len(firstPointFeatures)
print d
```

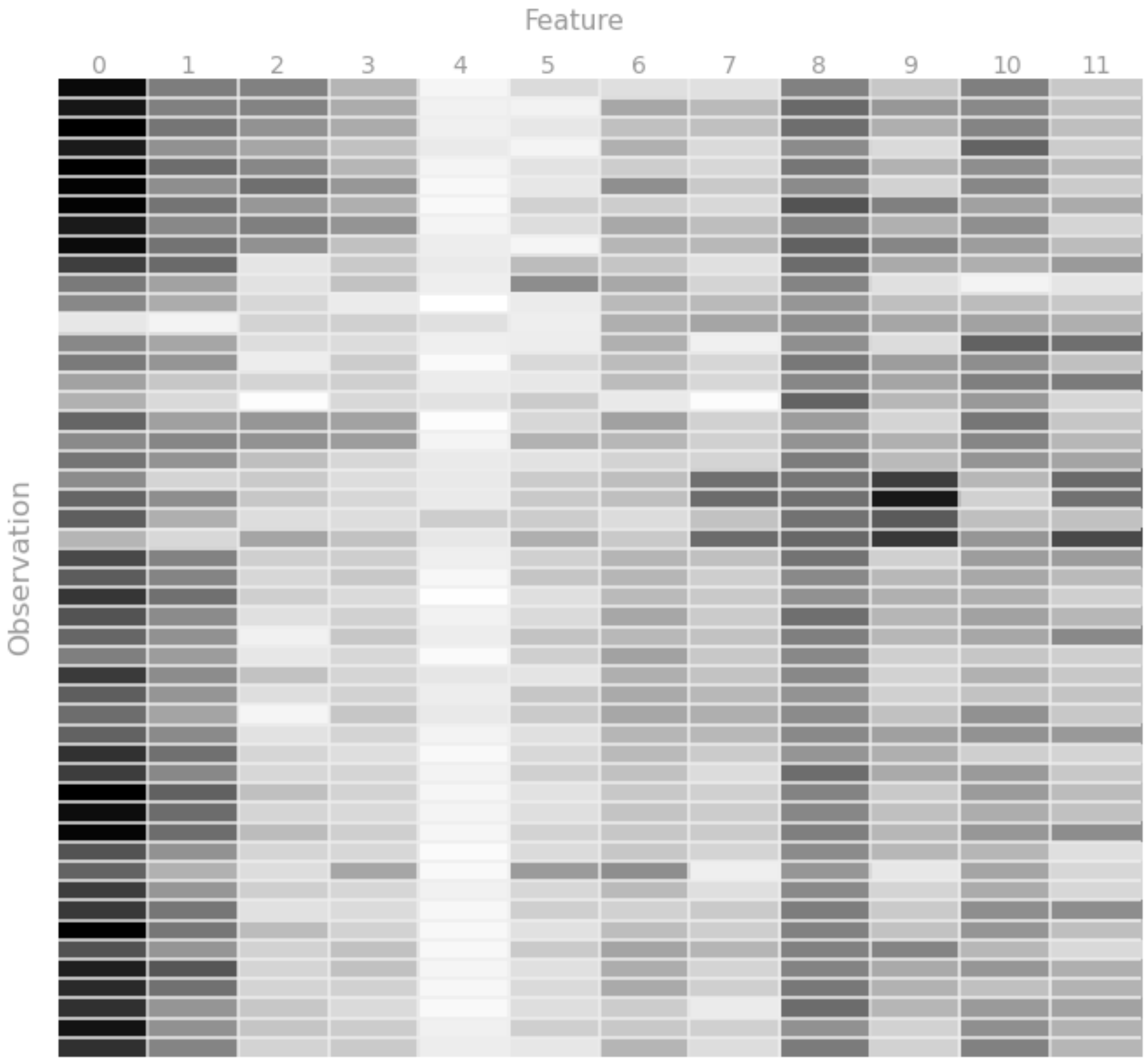
```
[0.884123733793,0.610454259079,0.600498416968,0.474669212493,0.247
232680947,0.357306088914,0.344136412234,0.339641227335,0.600858840
135,0.425704689024,0.60491501652,0.419193351817] 2001.0
12
```

Visualization 1: Features

First we will load and setup the visualization library. Then we will look at the raw features for 50 data points by generating a heatmap that visualizes each feature on a grey-scale and shows the variation of each feature across the 50 sample data points. The features are all between 0 and 1, with values closer to 1 represented via darker shades of grey.

In [45]:

```
import matplotlib.pyplot as plt
import matplotlib.cm as cm

sampleMorePoints = rawData.take(50)
# You can uncomment the line below to see randomly selected features.  These w
ill be randomly
# selected each time you run the cell.  Note that you should run this cell wit
h the line commented
# out when answering the lab quiz questions.
# sampleMorePoints = rawData.takeSample(False, 50)

parsedSampleMorePoints = map(parsePoint, sampleMorePoints)
dataValues = map(lambda lp: lp.features.toArray(), parsedSampleMorePoints)

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='whit
e')
    ax.axes.tick_params(labelcolor='#999999', labelsize='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', 't
op', 'left', 'right'])
    return fig, ax

# generate layout and plot
fig, ax = preparePlot(np.arange(.5, 11, 1), np.arange(.5, 49, 1), figsize=(8,7
), hideLabels=True,
                      gridColor='#eeeeee', gridWidth=1.1)
image = plt.imshow(dataValues,interpolation='nearest', aspect='auto', cmap=cm.
Greys)
for x, y, s in zip(np.arange(-.125, 12, 1), np.repeat(-.75, 12), [str(x) for x
in range(12)]):
    plt.text(x, y, s, color='#999999', size='10')
plt.text(4.7, -3, 'Feature', color='#999999', size='11'), ax.set_ylabel('Obser
vation')
pass
```



(1c) Find the range

The range of the years in the dataset.

In [53]:

```
parsedDataInit = rawData.map(parsePoint)
onlyLabels = parsedDataInit.map(lambda x: x.label).collect()

minYear = min(onlyLabels)
maxYear = max(onlyLabels)
print maxYear, minYear
```

```
2011.0 1922.0
```

(1d) Shift labels

Set the smallest year label 1922 to 0.

In [55]:

```
parsedData = parsedDataInit.map(lambda x: LabeledPoint(x.label-minYear, x.feat
ures))

# Should be a LabeledPoint
print type(parsedData.take(1)[0])
# View the first point
print '\n{0}'.format(parsedData.take(1))
```

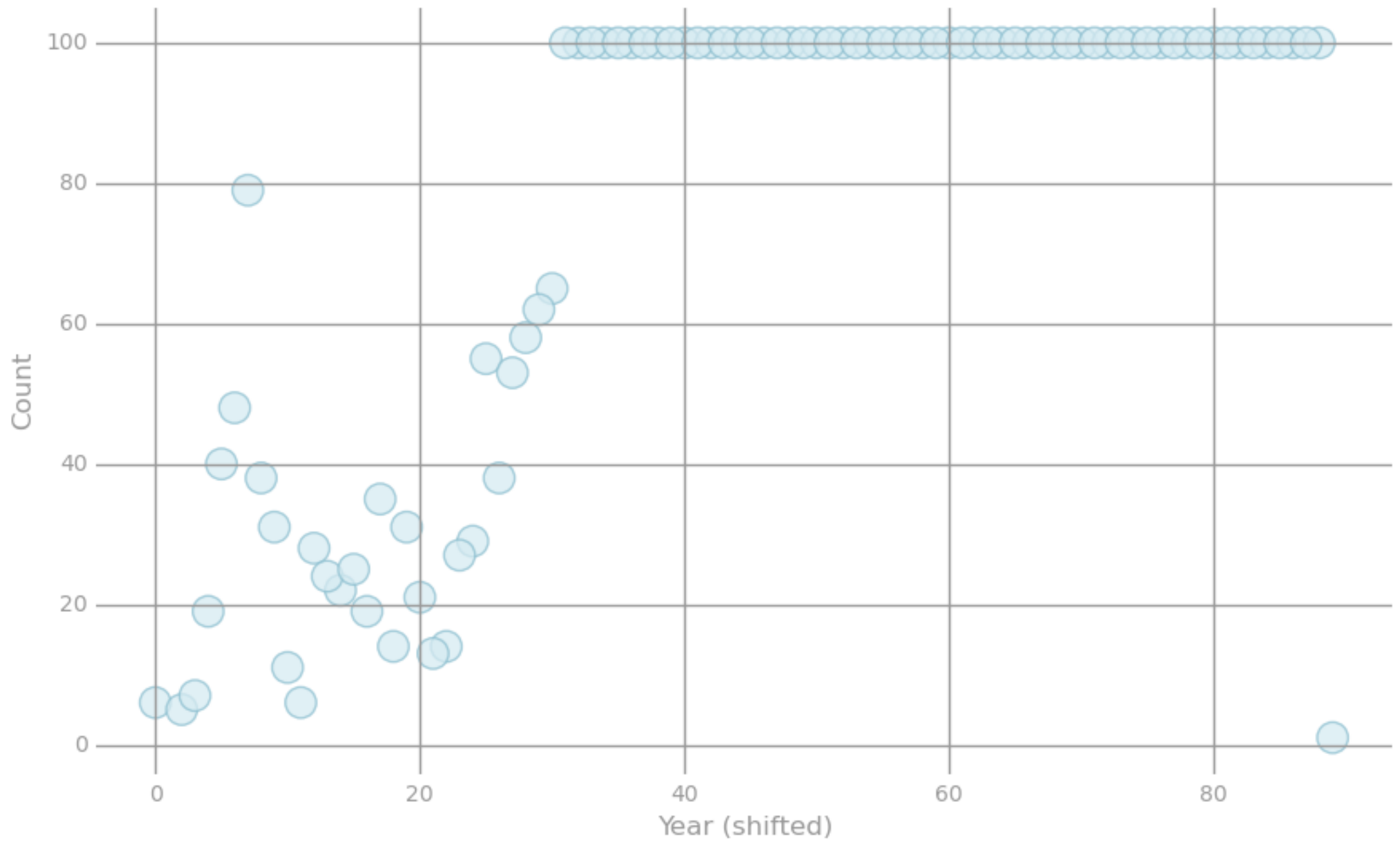
```
<class 'pyspark.mllib.regression.LabeledPoint'>

[LabeledPoint(79.0, [0.884123733793,0.610454259079,0.600498416968,
0.474669212493,0.247232680947,0.357306088914,0.344136412234,0.3396
41227335,0.600858840135,0.425704689024,0.60491501652,0.41919335181
7])]
```

Visualization 2: Shifting labels

The shifted years in the dataset.

In [58]:

```
# get data for plot
newData = (parsedData
           .map(lambda lp: (lp.label, 1))
           .reduceByKey(lambda x, y: x + y)
           .collect())
x, y = zip(*newData)

# generate layout and plot data
fig, ax = preparePlot(np.arange(0, 120, 20), np.arange(0, 120, 20))
plt.scatter(x, y, s=14**2, c='#d6ebf2', edgecolors='#8cbfd0', alpha=0.75)
ax.set_xlabel('Year (shifted)'), ax.set_ylabel('Count')
pass
```



(1e) Training, validation, and test sets

We now split the data into training, validation and test sets using the [randomSplit method] with the specified weights and seed to create RDDs storing each of these datasets. Next, cache each of these RDDs.

In [61]:

```
weights = [.8, .1, .1]
seed = 42
parsedTrainData, parsedValData, parsedTestData = parsedData.randomSplit(weight
s, seed)
parsedTrainData.cache()
parsedValData.cache()
parsedTestData.cache()
nTrain = parsedTrainData.count()
nVal = parsedValData.count()
nTest = parsedTestData.count()

print nTrain, nVal, nTest, nTrain + nVal + nTest
print parsedData.count()
```

```
5371 682 671 6724
6724
```

Part 2: Defining a loss function

(2a) Root mean squared error

We use root mean squared error [RMSE] for evaluation purposes. Implement a function to compute RMSE given an RDD of (label, prediction) tuples, and test out this function on an example.

In [123]:

```
def squaredError(label, prediction):
    """Calculates the the squared error for a single prediction.

    Args:
        label (float): The correct value for this observation.
        prediction (float): The predicted value for this observation.

    Returns:
        float: The difference between the `label` and `prediction` squared.
    """
    return (float(label)-float(prediction))**2


from operator import add
import math

def calcRMSE(X):
    """Calculates the root mean squared error for an `RDD` of (label, predicti
on) tuples.

    Args:
        labelsAndPred (RDD of (float, float)): An `RDD` consisting of (label,
prediction) tuples.

    Returns:
        float: The square root of the mean of the squared errors.
    """

    l = X.map(lambda x: float(x[0])).collect()
    p = X.map(lambda x: -float(x[1])).collect()
    c= [sum(x)**2 for x in zip(l, p)]
    return  math.sqrt(sum(c)/len(l))

labelsAndPreds = sc.parallelize([(3., 1.), (1., 2.), (2., 2.)])
# RMSE = sqrt[((3-1)^2 + (1-2)^2 + (2-2)^2) / 3] = 1.291
exampleRMSE = calcRMSE(labelsAndPreds)
print exampleRMSE
```

```
1.29099444874
```

Part 3: Train (via gradient descent) and evaluate a linear regression model

(3a) Gradient summand

We now apply linear regression, training a model via gradient descent. The gradient descent update for linear regression is:

$$\mathbf{w}_{i+1} = \mathbf{w}_i - \alpha_i \sum_j (\mathbf{w}_i^\top \mathbf{x}_j - y_j)\mathbf{x}_j \,.$$

where i is the iteration number of the gradient descent algorithm, and j identifies the observation.

First, implement a function that computes the summand for this update, i.e., the summand equals $(\mathbf{w}^\top \mathbf{x} - y)\mathbf{x}$, and test out this function on two examples, using the `DenseVector` method.

In [128]:

```
from pyspark.mllib.linalg import DenseVector
```

In [133]:

```


def gradientSummand(weights, lp):
    """Calculates the gradient summand for a given weight and `LabeledPoint`.

    Note:
        `DenseVector` behaves similarly to a `numpy.ndarray` and they can be u
sed interchangably
        within this function.  For example, they both implement the `dot` meth
od.

    Args:
        weights (DenseVector): An array of model weights (betas).
        lp (LabeledPoint): The `LabeledPoint` for a single observation.

    Returns:
        DenseVector: An array of values the same length as `weights`.  The gra
dient summand.
    """
    return (np.dot(weights,lp.features)-lp.label)*lp.features

exampleW = DenseVector([1, 1, 1])
exampleLP = LabeledPoint(2.0, [3, 1, 4])
print np.dot(exampleW,exampleLP.features)
# gradientSummand = (dot([1 1 1], [3 1 4]) - 2) * [3 1 4] = (8 - 2) * [3 1 4]
= [18 6 24]
summandOne = gradientSummand(exampleW, exampleLP)
print summandOne

exampleW = DenseVector([.24, 1.2, -1.4])
exampleLP = LabeledPoint(3.0, [-1.4, 4.2, 2.1])
summandTwo = gradientSummand(exampleW, exampleLP)
print summandTwo
```

```
8.0
[18.0,6.0,24.0]
[1.7304,-5.1912,-2.5956]
```

(3b) Use weights to make predictions

Next, implement a `getLabeledPredictions` function that takes in weights and an observation's `LabeledPoint` and returns a (label, prediction) tuple. The prediction is obtained by computing the dot product between weights and an observation's features.

In [135]:

```
def getLabeledPrediction(weights, observation):
    """Calculates predictions and returns a (label, prediction) tuple.

    Note:
        The labels should remain unchanged as we'll use this information to ca
lculate prediction
        error later.

    Args:
        weights (np.ndarray): An array with one weight for each features in `t
rainData`.
        observation (LabeledPoint): A `LabeledPoint` that contain the correct
label and the
            features for the data point.

    Returns:
        tuple: A (label, prediction) tuple.
    """
    return (np.dot(weights,observation.features), observation.label)

weights = np.array([1.0, 1.5])
predictionExample = sc.parallelize([LabeledPoint(2, np.array([1.0, .5])),
                                    LabeledPoint(1.5, np.array([.5, .5]))])
labelsAndPredsExample = predictionExample.map(lambda lp: getLabeledPrediction(
weights, lp))
print labelsAndPredsExample.collect()
```

```
[(1.75, 2.0), (1.25, 1.5)]
```

(3c) Gradient descent

Next, implement a gradient descent function for linear regression and test out this function on an example.

In [138]:

```
def linregGradientDescent(trainData, numIters):
    """Calculates the weights and error for a linear regression model trained
with gradient descent.

    Note:
        `DenseVector` behaves similarly to a `numpy.ndarray` and they can be u
sed interchangably
        within this function.  For example, they both implement the `dot` meth
```

```
od.

    Args:
        trainData (RDD of LabeledPoint): The labeled data for use in training
the model.
        numIters (int): The number of iterations of gradient descent to perfor
m.

    Returns:
        (np.ndarray, np.ndarray): A tuple of (weights, training errors).  Weig
hts will be the
            final weights (one weight per feature) for the model, and training
errors will contain
            an error (RMSE) for each iteration of the algorithm.
    """
    # The length of the training data
    n = trainData.count()
    # The number of features in the training data
    d = len(trainData.take(1)[0].features)
    w = np.zeros(d)
    alpha = 1.0
    # We will compute and store the training error after each iteration
    errorTrain = np.zeros(numIters)
    for i in range(numIters):
        # Use getLabeledPrediction from (3b) with trainData to obtain an RDD o
f (label, prediction)
        # tuples.  Note that the weights all equal 0 for the first iteration,
so the predictions will
        # have large errors to start.
        labelsAndPredsTrain = trainData.map(lambda x: getLabeledPrediction(w,x
))
        errorTrain[i] = calcRMSE(labelsAndPredsTrain)

        # Calculate the `gradient`.  Make use of the `gradientSummand` functio
n you wrote in (3a).
        # Note that `gradient` sould be a `DenseVector` of length `d`.
        gradient =  trainData.map(lambda x: gradientSummand(w, x)).reduce(lamb
da x,y: x+y)

        # Update the weights
        alpha_i = alpha / (n * np.sqrt(i+1))
        w -=  alpha_i*gradient
    return w, errorTrain

# create a toy dataset with n = 10, d = 3, and then run 5 iterations of gradie
nt descent
# note: the resulting model will not be useful; the goal here is to verify tha
t
# linregGradientDescent is working properly
exampleN = 10
exampleD = 3
exampleData = (sc
               .parallelize(parsedTrainData.take(exampleN))
               .map(lambda lp: LabeledPoint(lp.label, lp.features[0:exampleD])
))
print exampleData.take(2)
exampleNumIters = 5
exampleWeights, exampleErrorTrain = linregGradientDescent(exampleData, example
NumIters)

print exampleWeights
```

```
[LabeledPoint(79.0, [0.884123733793,0.610454259079,0.600498416968]
), LabeledPoint(79.0, [0.854411946129,0.604124786151,0.59363407877
6])]
[ 48.88110449  36.01144093  30.25350092]
```

(3d) Train the model

Next we train a linear regression model on all of our training data and evaluate its accuracy on the validation set.

In [140]:

```
numIters = 50
weightsLR0, errorTrainLR0 = linregGradientDescent(parsedTrainData,numIters)

labelsAndPreds = parsedValData.map(lambda x: getLabeledPrediction(weightsLR0,x
))
rmseValLR0 = calcRMSE(labelsAndPreds)

print 'Validation RMSE:\n\tBaseline = {0:.3f}\n\tLR0 = {1:.3f}'.format(rmseVal
Base,
                                                                       rmseVal
LR0)
```

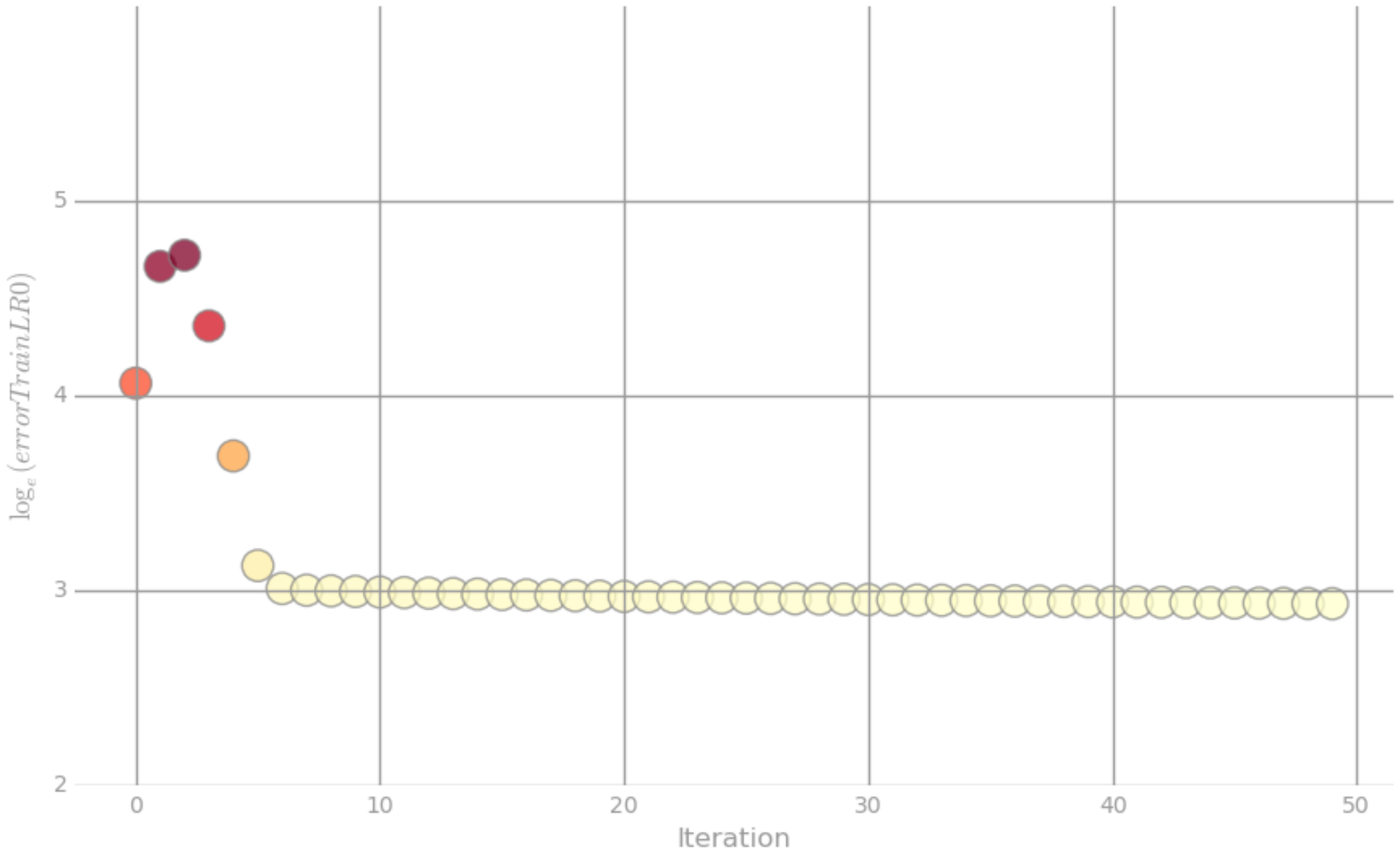
```
Validation RMSE:
        Baseline = 21.586
        LR0 = 19.192
```

Visualization 4: Training error

We will look at the log of the training error as a function of iteration. The first scatter plot visualizes the logarithm of the training error for all 50 iterations. The second plot shows the training error itself, focusing on the final 44 iterations.

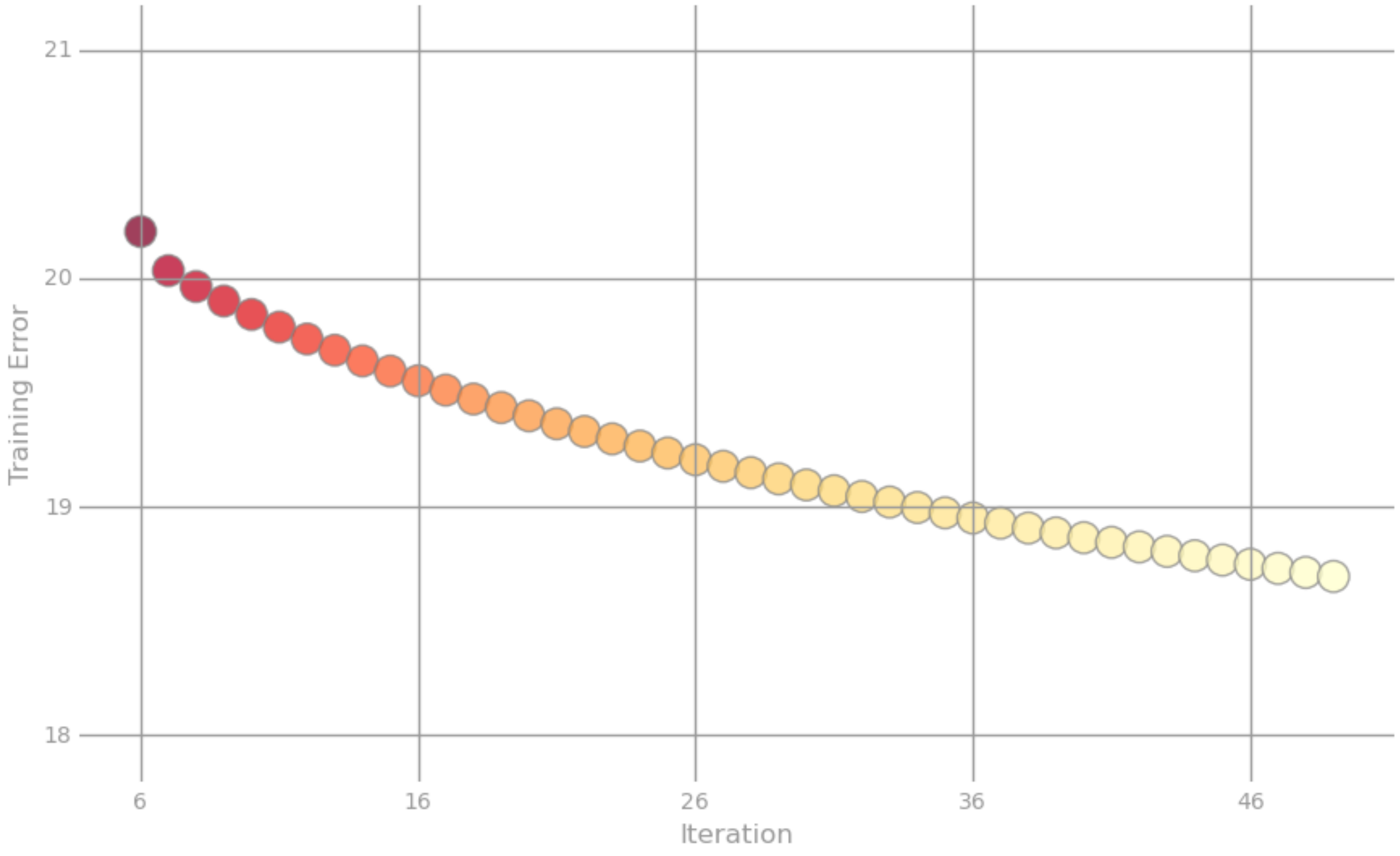
In [142]:

```
norm = Normalize()
clrs = cmap(np.asarray(norm(np.log(errorTrainLR0))))[:,0:3]

fig, ax = preparePlot(np.arange(0, 60, 10), np.arange(2, 6, 1))
ax.set_ylim(2, 6)
plt.scatter(range(0, numIters), np.log(errorTrainLR0), s=14**2, c=clrs, edgeco
lors='#888888', alpha=0.75)
ax.set_xlabel('Iteration'), ax.set_ylabel(r'$\log_e(errorTrainLR0)$')
pass
```



In [143]:

```
norm = Normalize()
clrs = cmap(np.asarray(norm(errorTrainLR0[6:])))[:,0:3]

fig, ax = preparePlot(np.arange(0, 60, 10), np.arange(17, 22, 1))
ax.set_ylim(17.8, 21.2)
plt.scatter(range(0, numIters-6), errorTrainLR0[6:], s=14**2, c=clrs, edgecolo
rs='#888888', alpha=0.75)
ax.set_xticklabels(map(str, range(6, 66, 10)))
ax.set_xlabel('Iteration'), ax.set_ylabel(r'Training Error')
pass
```



Part 4: Train using MLlib and perform grid search

(4a) `LinearRegressionWithSGD`

We can improve the model by adding an intercept, using regularization, and (based on the previous visualization) training for more iterations. First use LinearRegressionWithSGD to train a model with L2 regularization and with an intercept. This method returns a [LinearRegressionModel]. Next, use the model's [weights] and [intercept] attributes to print out the model's parameters.

In [144]:

```
from pyspark.mllib.regression import LinearRegressionWithSGD
# Values to use when training the linear regression model
numIters = 500  # iterations
alpha = 1.0  # step
miniBatchFrac = 1.0  # miniBatchFraction
reg = 1e-1  # regParam
regType = 'l2'  # regType
useIntercept = True  # intercept
```

In [147]:

```
firstModel = LinearRegressionWithSGD.train(parsedTrainData, iterations=numIter
s, step=alpha,
                                           miniBatchFraction=1.0, initialWeigh
ts=None,
                                           regParam=reg, regType='l2', interce
pt=True)

# weightsLR1 stores the model weights; interceptLR1 stores the model intercept
weightsLR1 =  firstModel.weights
interceptLR1 = firstModel.intercept
print weightsLR1, interceptLR1
```

```
[16.682292427,14.7439059559,-0.0935105608897,6.22080088829,4.01454
261926,-3.30214858535,11.0403027232,2.67190962854,7.18925791279,4.
46093254586,8.14950409475,2.75135810882] 13.3335907631
```

(4b) Predict

Now use the [LinearRegressionModel.predict() (http://spark.apache.org/docs/latest/api/python/pyspark.mllib.html#pyspark.mllib.regression.Linear](http://spark.apache.org/docs/latest/api/python/pyspark.mllib.html#pyspark.mllib.regression.Linear) method to make a prediction on a sample point. Pass the `features` from a `LabeledPoint` into

In [162]:

```
samplePoint = parsedTrainData.collect()[0]
print samplePoint.features

samplePrediction = firstModel.predict(samplePoint.features )
print samplePrediction
```

```
[0.884123733793,0.610454259079,0.600498416968,0.474669212493,0.247
232680947,0.357306088914,0.344136412234,0.339641227335,0.600858840
135,0.425704689024,0.60491501652,0.419193351817]
56.8013380112
```

(4c) Evaluate RMSE

Next we evaluate the accuracy of this model on the validation set. Use the `predict()` method to create a `labelsAndPreds` RDD, and then use the `calcRMSE()` function from Part (2a).

In [166]:

```
labelsAndPreds = parsedValData.map(lambda x: (x.label,firstModel.predict(x.fea
tures)))
print labelsAndPreds.top(1)
rmseValLR1 =  calcRMSE(labelsAndPreds)

print ('Validation RMSE:\n\tBaseline = {0:.3f}\n\tLR0 = {1:.3f}' +
       '\n\tLR1 = {2:.3f}').format(rmseValBase, rmseValLR0, rmseValLR1)
```

```
[(88.0, 61.074692008947231)]
Validation RMSE:
        Baseline = 21.586
        LR0 = 19.192
        LR1 = 19.691
```

(4d) Grid search

Perform grid search to find a good regularization parameter. We try `regParam` values `1e-10`, `1e-5`, and `1`.

In [168]:

```


bestRMSE = rmseValLR1
bestRegParam = reg
bestModel = firstModel

numIters = 500
alpha = 1.0
miniBatchFrac = 1.0
for reg in [1e-10, 1e-5, 1]:
    model = LinearRegressionWithSGD.train(parsedTrainData, numIters, alpha,
                                          miniBatchFrac, regParam=reg,
                                          regType='l2', intercept=True)
    labelsAndPreds = parsedValData.map(lambda lp: (lp.label, model.predict(lp.
features)))
    rmseValGrid = calcRMSE(labelsAndPreds)
    print rmseValGrid

    if rmseValGrid < bestRMSE:
        bestRMSE = rmseValGrid
        bestRegParam = reg
        bestModel = model
rmseValLRGrid = bestRMSE

print ('Validation RMSE:\n\tBaseline = {0:.3f}\n\tLR0 = {1:.3f}\n\tLR1 = {2:.3
f}\n' +
       '\tLRGrid = {3:.3f}').format(rmseValBase, rmseValLR0, rmseValLR1, rmseV
alLRGrid)
```

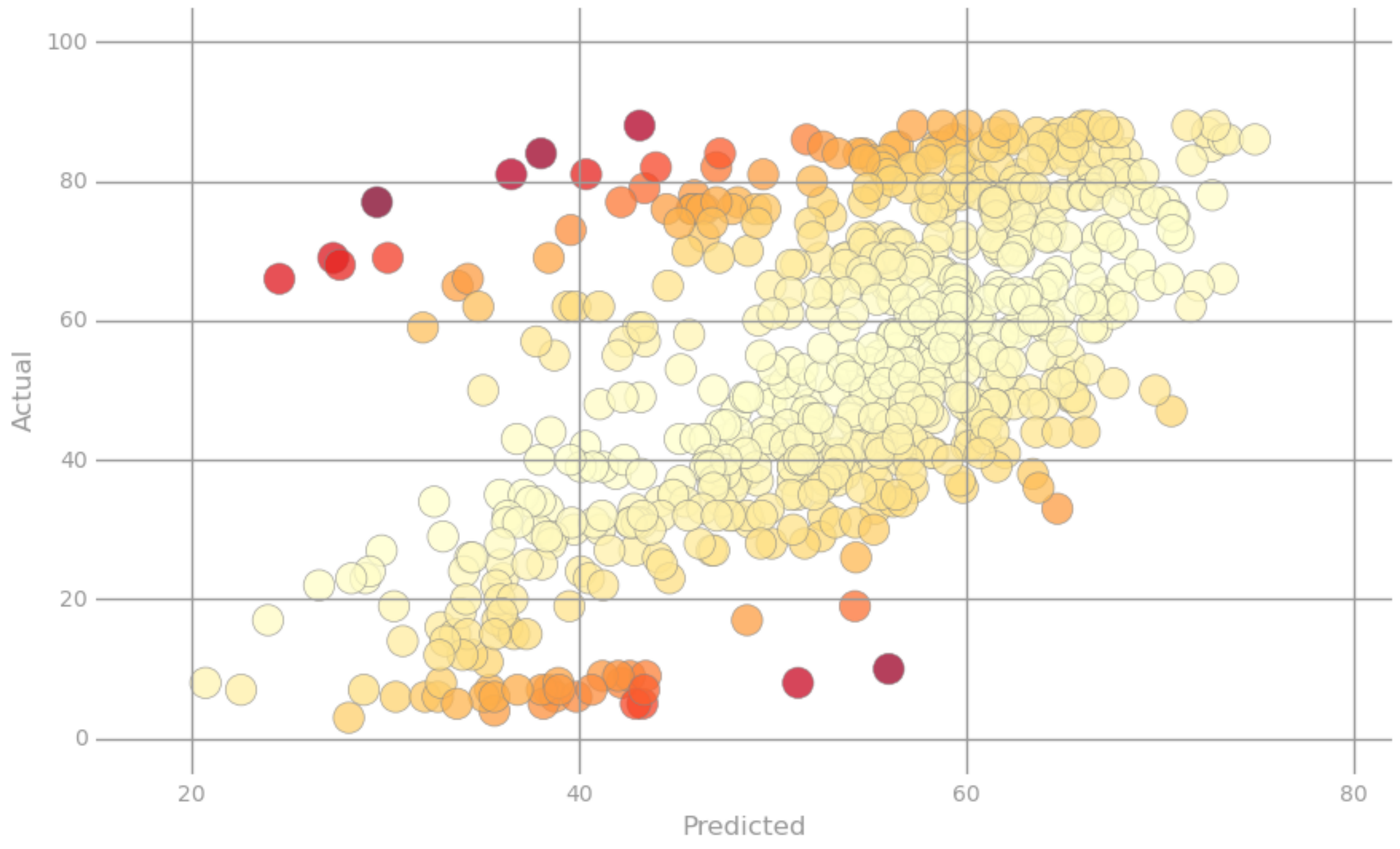
```
17.0171700716
17.0175981807
23.8007746698
Validation RMSE:
        Baseline = 21.586
        LR0 = 19.192
        LR1 = 19.691
        LRGrid = 17.017
```

Visualization 5: Best model's predictions

Next, we create a visualization 'Predicted vs. actual' fusing the predictions from the best model from Part (4d) on the validation dataset. Specifically, we create a color-coded scatter plot visualizing tuples storing i) the predicted value from this model and ii) true label.

In [170]:

```
predictions = np.asarray(parsedValData
                         .map(lambda lp: bestModel.predict(lp.features))
                         .collect())
actual = np.asarray(parsedValData
                    .map(lambda lp: lp.label)
                    .collect())
error = np.asarray(parsedValData
                   .map(lambda lp: (lp.label, bestModel.predict(lp.features)))
                   .map(lambda (l, p): squaredError(l, p))
                   .collect())

norm = Normalize()
clrs = cmap(np.asarray(norm(error)))[:,0:3]

fig, ax = preparePlot(np.arange(0, 120, 20), np.arange(0, 120, 20))
ax.set_xlim(15, 82), ax.set_ylim(-5, 105)
plt.scatter(predictions, actual, s=14**2, c=clrs, edgecolors='#888888', alpha=
0.75, linewidths=.5)
ax.set_xlabel('Predicted'), ax.set_ylabel(r'Actual')
pass
```

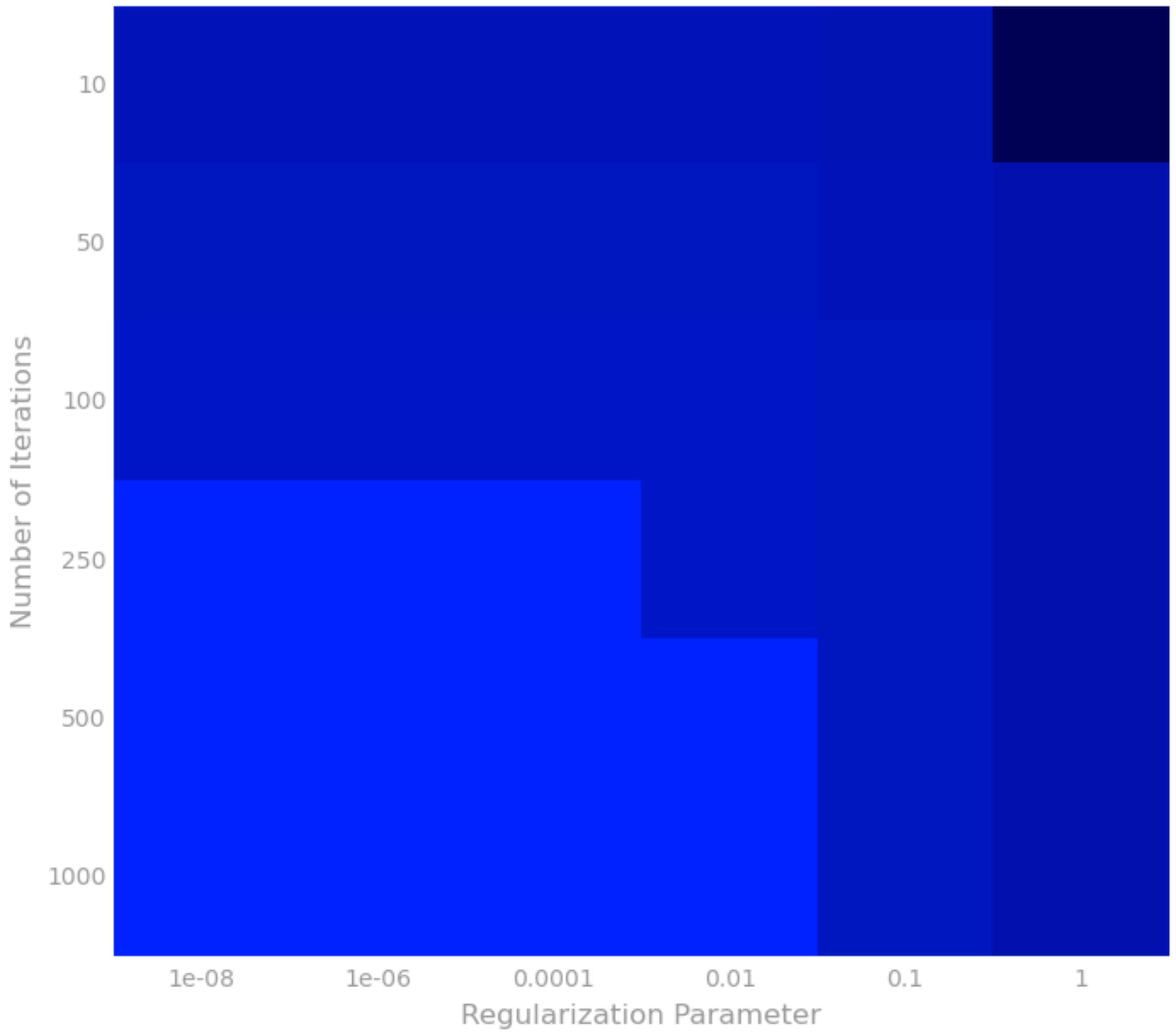


Visualization 6: Hyperparameter heat map

Next, we perform a visualization of hyperparameter search using a larger set of hyperparameters (with precomputed results). Specifically, we create a heat map where the brighter colors correspond to lower RMSE values. The first plot has a large area with brighter colors. In order to differentiate within the bright region, we generate a second plot corresponding to the hyperparameters found within that region.

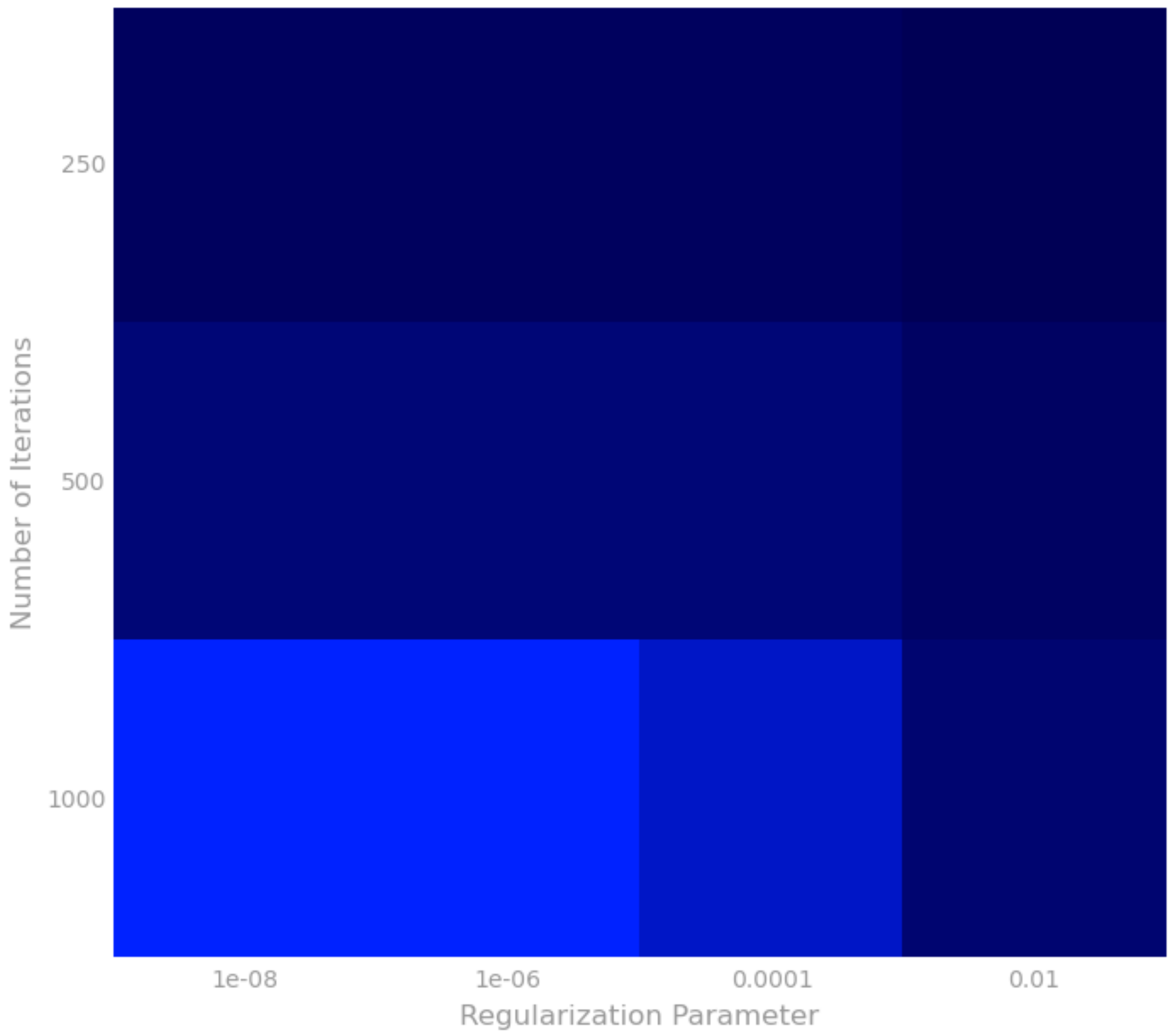
In [184]:

```
from matplotlib.colors import LinearSegmentedColormap

# Saved parameters and results, to save the time required to run 36 models
numItersParams = [10, 50, 100, 250, 500, 1000]
regParams = [1e-8, 1e-6, 1e-4, 1e-2, 1e-1, 1]
rmseVal = np.array([[  20.36769649,   20.36770128,   20.36818057,   20.4179535
4,  21.09778437,  301.54258421],
                    [  19.04948826,   19.0495    ,   19.05067418,   19.1651772
6,  19.97967727,   23.80077467],
                    [  18.40149024,   18.40150998,   18.40348326,   18.5945749
1,  19.82155716,   23.80077467],
                    [  17.5609346 ,   17.56096749,   17.56425511,   17.8844212
7,  19.71577117,   23.80077467],
                    [  17.0171705 ,   17.01721288,   17.02145207,   17.4451057
4,  19.69124734,   23.80077467],
                    [  16.58074813,   16.58079874,   16.58586512,   17.1146690
4,  19.6860931 ,   23.80077467]])

numRows, numCols = len(numItersParams), len(regParams)
rmseVal = np.array(rmseVal)
rmseVal.shape = (numRows, numCols)

fig, ax = preparePlot(np.arange(0, numCols, 1), np.arange(0, numRows, 1), figs
ize=(8, 7), hideLabels=True,
                      gridWidth=0.)
ax.set_xticklabels(regParams), ax.set_yticklabels(numItersParams)
ax.set_xlabel('Regularization Parameter'), ax.set_ylabel('Number of Iterations
')

colors = LinearSegmentedColormap.from_list('blue', ['#0022ff', '#000055'], gam
ma=.2)
image = plt.imshow(rmseVal,interpolation='nearest', aspect='auto',
                    cmap = colors)
```


10
50
100
250
500
1000
Number of Iterations
1e-08
1e-06
0.0001
0.01
0.1
1
Regularization Parameter

In [185]:

```
# Zoom into the bottom left
numItersParamsZoom, regParamsZoom = numItersParams[-3:], regParams[:4]
rmseValZoom = rmseVal[-3:, :4]

numRows, numCols = len(numItersParamsZoom), len(regParamsZoom)

fig, ax = preparePlot(np.arange(0, numCols, 1), np.arange(0, numRows, 1), figs
ize=(8, 7), hideLabels=True,
                      gridWidth=0.)
ax.set_xticklabels(regParamsZoom), ax.set_yticklabels(numItersParamsZoom)
ax.set_xlabel('Regularization Parameter'), ax.set_ylabel('Number of Iterations
')

colors = LinearSegmentedColormap.from_list('blue', ['#0022ff', '#000055'], gam
ma=.2)
image = plt.imshow(rmseValZoom,interpolation='nearest', aspect='auto',
                    cmap = colors)
pass
```



Part 5: Add interactions between features

(5a) Add 2-way interactions

Now, we will add features that capture the two-way interactions between our existing features. A function `twoWayInteractions` takes in a `LabeledPoint` and generates a new `LabeledPoint` that contains the old features and the two-way interactions between them.

In [200]:

```
import itertools

def twoWayInteractions(lp):
    """Creates a new `LabeledPoint` that includes two-way interactions.

    Note:
        For features [x, y] the two-way interactions would be [x^2, x*y, y*x,
y^2] and these
        would be appended to the original [x, y] feature list.

    Args:
        lp (LabeledPoint): The label and features for this observation.

    Returns:
        LabeledPoint: The new `LabeledPoint` should have the same label as `lp
`.  Its features
            should include the features from `lp` followed by the two-way inte
raction features.
    """
    lb = lp.label
    a = list(itertools.product(lp.features, repeat=2))
    b = [np.prod(i) for i in a]
    ft = np.hstack((lp.features, b))
    return LabeledPoint(lb,ft)

print twoWayInteractions(LabeledPoint(0.0, [2, 3]))

# Transform the existing train, validation, and test sets to include two-way i
nteractions.
trainDataInteract = parsedTrainData.map(twoWayInteractions)
valDataInteract = parsedValData.map(twoWayInteractions)
testDataInteract = parsedTestData.map(twoWayInteractions)
```

```
(0.0,[2.0,3.0,4.0,6.0,6.0,9.0])
```

(5b) Build interaction model

Now, we build the new model using the old features and the new interaction features.

In [202]:

```
numIters = 500
alpha = 1.0
miniBatchFrac = 1.0
reg = 1e-10

modelInteract = LinearRegressionWithSGD.train(trainDataInteract, numIters, alp
ha,
                                              miniBatchFrac, regParam=reg,
                                              regType='l2', intercept=True)
labelsAndPredsInteract = valDataInteract.map(lambda lp: (lp.label, modelIntera
ct.predict(lp.features)))
rmseValInteract = calcRMSE(labelsAndPredsInteract)

print ('Validation RMSE:\n\tBaseline = {0:.3f}\n\tLR0 = {1:.3f}\n\tLR1 = {2:.3
f}\n\tLRGrid = ' +
       '{3:.3f}\n\tLRInteract = {4:.3f}').format(rmseValBase, rmseValLR0, rmse
ValLR1,
                                                 rmseValLRGrid, rmseValInterac
t)
```

```
Validation RMSE:
        Baseline = 21.586
        LR0 = 19.192
        LR1 = 19.691
        LRGrid = 17.017
        LRInteract = 15.690
```

(5c) Evaluate interaction model on test data

We now evaluate the new model on the test dataset. This evaluation provides us with an unbiased estimate for how our model will perform on new data. We also compare with a Baseline model where all the predictions are just the average year.

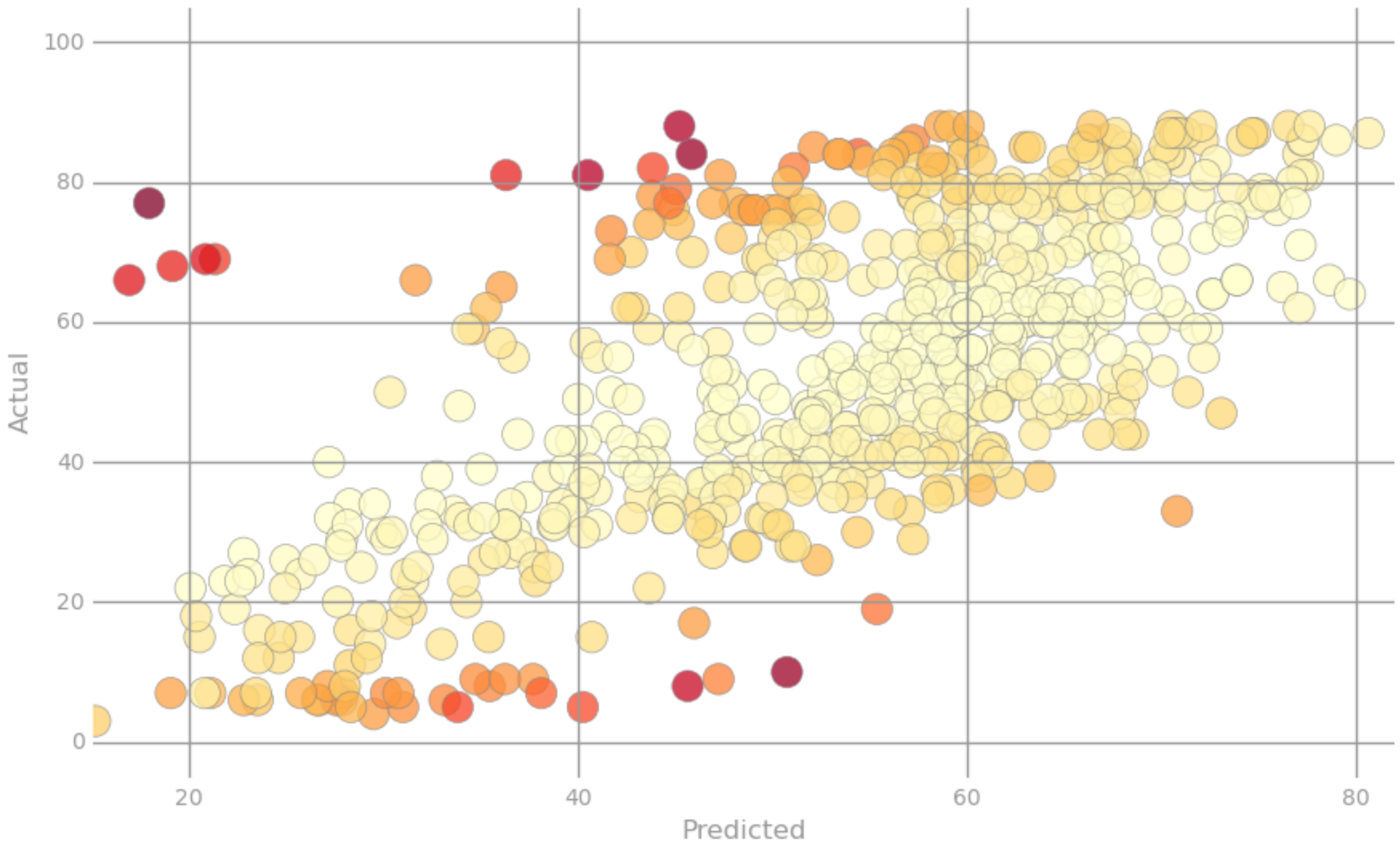
In [204]:

```


labelsAndPredsTest = testDataInteract.map(lambda lp: (lp.label, modelInteract.
predict(lp.features)))
rmseTestInteract = calcRMSE(labelsAndPredsTest)

print ('Test RMSE:\n\tBaseline = {0:.3f}\n\tLRInteract = {1:.3f}'
       .format(rmseTestBase, rmseTestInteract))
```

```
Test RMSE:
        Baseline = 22.137
        LRInteract = 16.327
```

In [206]:

```
predictions = np.asarray(valDataInteract
                         .map(lambda lp: modelInteract.predict(lp.features))
                         .collect())
actual = np.asarray(parsedValData
                    .map(lambda lp: lp.label)
                    .collect())
error = np.asarray(parsedValData
                   .map(lambda lp: (lp.label, bestModel.predict(lp.features)))
                   .map(lambda (l, p): squaredError(l, p))
                   .collect())

norm = Normalize()
clrs = cmap(np.asarray(norm(error)))[:,0:3]

fig, ax = preparePlot(np.arange(0, 120, 20), np.arange(0, 120, 20))
ax.set_xlim(15, 82), ax.set_ylim(-5, 105)
plt.scatter(predictions, actual, s=14**2, c=clrs, edgecolors='#888888', alpha=
0.75, linewidths=.5)
ax.set_xlabel('Predicted'), ax.set_ylabel(r'Actual')
pass
```


Actual
Predicted