PD6: Prediction of the Release Year of a Song

MTI850 - Big Data Analytics

Fall 2025

Fixed for mllib to ml

Équipe	9

This assignment covers a common supervised learning pipeline, using a subset of the Million Song Dataset from the UCI Machine Learning Repository. Our goal is to train a linear regression model to predict the release year of a song given a set of audio features. The subset has 6,724 samples out of 515,345 in the original dataset.

This assignment will cover:

- *Part 1:* Read and parse the initial dataset
  - *Visualization 1:* Features
  - *Visualization 2:* Shifting labels
- *Part 2:* Create and evaluate a baseline model
  - *Visualization 3:* Predicted vs. actual
- *Part 3:* Train (via gradient descent) and evaluate a linear regression model
  - *Visualization 4:* Training error
- *Part 4:* Train using SparkML library and tune hyperparameters via grid search
  - *Visualization 5:* Best model's predictions
  - *Visualization 6:* Hyperparameter heat map
- *Part 5:* Add interactions between features

> Note that, for reference, you can look up the details of:

> - the relevant Spark methods in Spark's RDD Python API and Spark's DataFrame Python API
> - the relevant NumPy methods in the NumPy Reference

Part 0: Preparing the enviroment

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

spark = SparkSession.builder \
.master("local") \
.appName("Music Year Prediction") \
.getOrCreate()
```

```
WARNING: Using incubator modules: jdk.incubator.vector
Using Spark's default log4j profile: org/apache/spark/log4j2-defaults.prop
erties
25/11/28 00:04:56 WARN Utils: Your hostname, mti850vm, resolves to a loopb
ack address: 127.0.1.1; using 10.0.2.15 instead (on interface enp0s8)
25/11/28 00:04:56 WARN Utils: Set SPARK_LOCAL_IP if you need to bind to an
other address
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).
25/11/28 00:04:56 WARN NativeCodeLoader: Unable to load native-hadoop libr
ary for your platform... using builtin-java classes where applicable
```

Part 1: Read and parse the initial dataset

(1a) Download, load, and check the data

The raw data is currently stored in a text file. Let's download it from Moodle and make it available in the hdfs filesystem.

```
start-dfs.sh && start-yarn.sh
hdfs dfs -put millionsong.txt /millionsong.txt
```

Then, we will store this raw data in as a DataFrame, with each element of the DataFrame representing a data point as a comma-delimited string. Each string starts with the label (a year) followed by numerical audio features. Use the DataFrame count method to check how many data points we have. Then use the take method to create and print out a list of the first 5 data points in their initial string format.

In [3]:
```
raw_data_df = spark.read.text("./millionsong.txt")

raw_data_df.show(5)
```

```
+--------------------+
|               value|
+--------------------+
|2001.0,0.88412373...|
|2001.0,0.85441194...|
|2001.0,0.90898297...|
|2001.0,0.84252521...|
|2001.0,0.90930328...|
+--------------------+
only showing top 5 rows
```

In [6]:
```
# TODO: Replace <FILL IN> with appropriate code

num_points = raw_data_df.count()
print(num_points)

sample_points = raw_data_df.take(5)
print(sample_points)
```

```
6724
[Row(value='2001.0,0.884123733793,0.610454259079,0.600498416968,0.47466921
2493,0.247232680947,0.357306088914,0.344136412234,0.339641227335,0.6008588
40135,0.425704689024,0.60491501652,0.419193351817'), Row(value='2001.0,0.8
54411946129,0.604124786151,0.593634078776,0.495885413963,0.26630783093
6,0.261472105188,0.506387076327,0.464453565511,0.665798573683,0.5429689887
66,0.58044428577,0.445219373624'), Row(value='2001.0,0.908982970575,0.6320
63159227,0.557428975183,0.498263761394,0.276396052336,0.312809861625,0.448
530069406,0.448674249968,0.649791323916,0.489868662682,0.591908113534,0.45
00023818'), Row(value='2001.0,0.842525219898,0.561826888508,0.50871525969
2,0.443531142139,0.296733836002,0.250213568176,0.488540873206,0.3605087476
59,0.575435243185,0.361005878554,0.678378718617,0.409036786173'), Row(valu
e='2001.0,0.909303285534,0.653607720915,0.585580794716,0.473250503005,0.25
1417011835,0.326976795524,0.40432273022,0.371154511756,0.629401917965,0.48
2243251755,0.566901413923,0.463373691946')]
```

In [7]:
```
# TEST Load and check the data (1a)
testmti850.Test.assertEquals(num_points, 6724, 'incorrect value for num_p
testmti850.Test.assertEquals(len(sample_points), 5, 'incorrect length for
```

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

(1b) Using `LabeledPoint`

In MLlib, labeled training instances are stored using the LabeledPoint object. Write the `parse_points` function that takes, as input, a DataFrame of comma-separated strings. We'll pass it the `raw_data_df` DataFrame.

It should parse each row in the DataFrame into individual elements, using Spark's select and split methods.

For example, split `"2001.0,0.884,0.610,0.600,0.474,0.247,0.357,0.344,0.33,0.600,0.425,0` into `['2001.0', '0.884', '0.610', '0.600', '0.474', '0.247', '0.357', '0.344', '0.33', '0.600', '0.425', '0.60', '0.419']`.

The first value in the resulting list (`2001.0` in the example, above) is the label. The remaining values (`0.884` , `0.610` , etc., in the example) are the features.

After splitting each row, map it to a `LabeledPoint` . You'll have to step down to an RDD (using `.rdd`) or use a DataFrame user-defined function to convert to the `LabeledPoint` object. (See **Hint**, below.) If you step down to an RDD, you'll have to use toDF() to convert back to a DataFrame.

Use this new `parse_points` function to parse `raw_data_df` . Then print out the features and label for the first training point, using the `features` and `label` attributes. Finally, calculate the number of features for this dataset.

Hint: Running Arbitrary Lambdas on a DataFrame

To solve this problem, you need a way to run your `parse_points` function on a DataFrame. There are two ways to do this, which we will illustrate with an extremely simple example.

Suppose you have a DataFrame consisting of a first name and a last name, and you want to add a unique SHA-256 hash to each row.

```
df = spark.createDataFrame([("John", "Smith"), ("Ravi",
"Singh"), ("Julia", "Jones")], ("first_name",
"last_name"))
```

Here's a simple function to calculate such a hash, using Python's built-in `hashlib` library:

```
def make_hash(first_name, last_name):
    import hashlib
    m = hashlib.sha256()
    # Join the first name and last name by a blank and
hash the resulting
    # string.
    full_name = ' '.join((first_name, last_name))
    m.update(full_name)
    return m.hexdigest()
```

Okay, that's great. But, how do we use it on our DataFrame? We can use a UDF:

```
from pyspark.sql.functions import udf
u_make_hash = udf(make_hash)
df2 = df.select(df['*'], u_make_hash(df['first_name'],
df['last_name']))
# could run df2.show() here to prove it works
```

Or we can step down to an RDD, use a lambda to call `make_hash` and have the lambda return a `Row` object, which Spark can use to "infer" a new DataFrame.

```
    from pyspark.sql import Row
    def make_hash_from_row(row):
        hash = make_hash(row[0], row[1])
        return Row(first_name=row[0], last_name=row[1],
    hash=hash)

    df2 = (df.rdd
             .map(lambda row: make_hash_from_row(row))
             .toDF())
```

These methods are roughly equivalent. You'll need to do something similar to convert *your* `raw_data_df` DataFrame into a new DataFrame of `LabeledPoint` objects.

In [9]:
```
from pyspark.mllib.regression import LabeledPoint
import numpy as np

# Here is a sample raw data point:
# '2001.0,0.884,0.610,0.600,0.474,0.247,0.357,0.344,0.33,0.600,0.425,0.60
# In this raw data point, 2001.0 is the label, and the remaining values a
```

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

def parse_points(df):
    """Converts a DataFrame of comma separated unicode strings into a Dat

    Args:
        df: DataFrame where each row is a comma separated unicode string.
            is the label and the remaining elements are the features.

    Returns:
        DataFrame: Each row is converted into a `LabeledPoint`, which con
            features. To convert an RDD to a DataFrame, simply call toDF(
    """
    def parse_line(row):
        raw_string = row[0]
        values = [float(x) for x in raw_string.split(',')]

        return LabeledPoint(values[0], values[1:])

    return df.rdd.map(parse_line).toDF()


parsed_points_df = parse_points(raw_data_df)
first_point = parsed_points_df.first()
first_point_features = first_point.features
first_point_label = first_point.label

print(first_point_features, first_point_label)

d = len(first_point_features)

print(d)
```

```
[0.884123733793,0.610454259079,0.600498416968,0.474669212493,0.24723268094
7,0.357306088914,0.344136412234,0.339641227335,0.600858840135,0.4257046890
24,0.60491501652,0.419193351817] 2001.0
12
Exception ignored in: <_io.BufferedWriter name=5>
Traceback (most recent call last):
  File "/opt/spark/python/lib/pyspark.zip/pyspark/daemon.py", line 200, in
manager
BrokenPipeError: [Errno 32] Broken pipe
```

In [15]:

```
# TEST Using LabeledPoint (1b)
testmti850.Test.assertTrue(isinstance(first_point_label, float), 'label m
expectedX0 = [0.8841,0.6105,0.6005,0.4747,0.2472,0.3573,0.3441,0.3396,0.6
testmti850.Test.assertTrue(np.allclose(expectedX0, first_point_features,
                'incorrect features for firstPointFeatures')
testmti850.Test.assertTrue(np.allclose(2001.0, first_point_label), 'incor
testmti850.Test.assertTrue(d == 12, 'incorrect number of features')
```

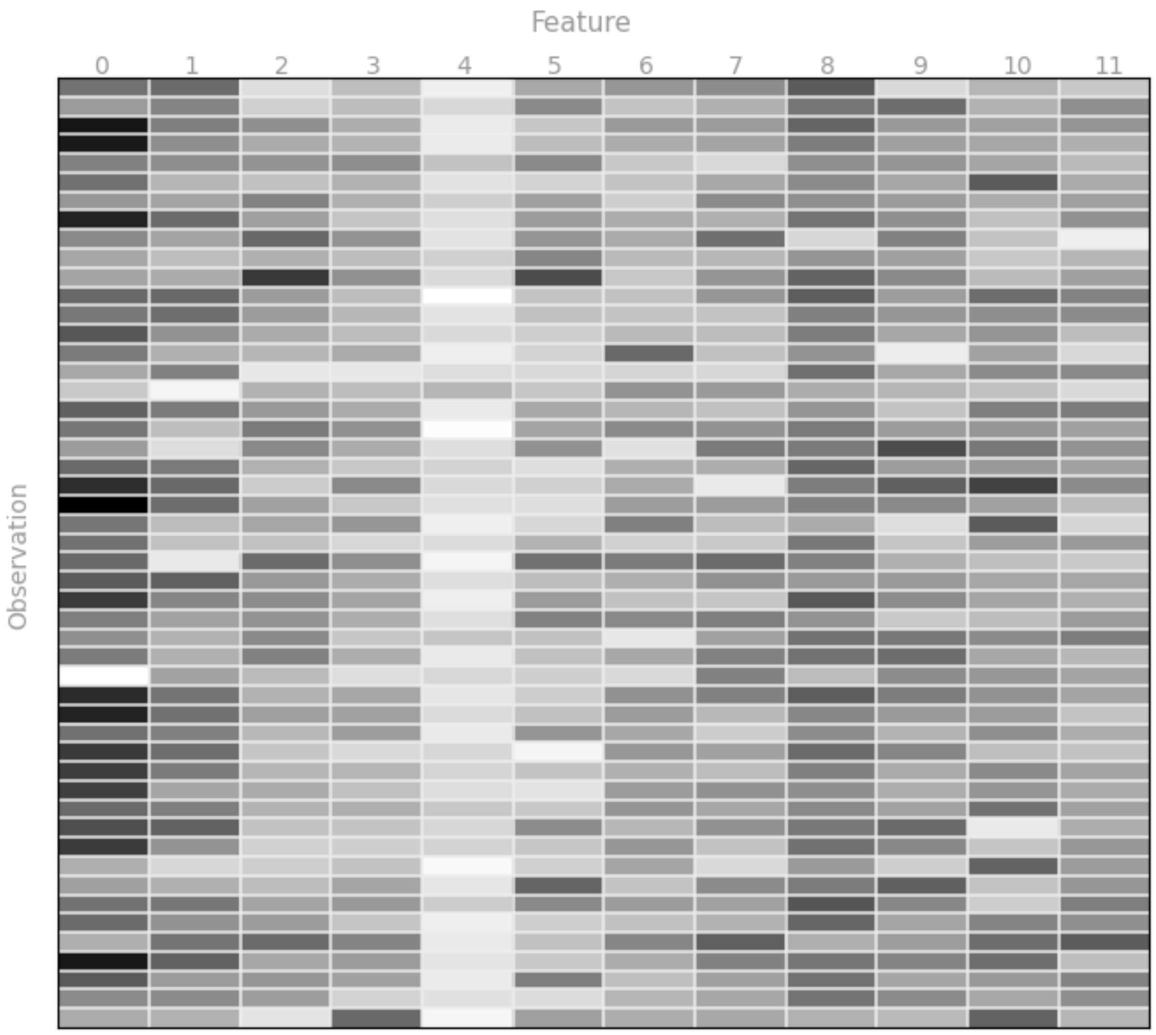
```
1 test passed.
1 test passed.
1 test passed.
1 test passed.
```

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 [16]:

```
import matplotlib.pyplot as plt
import matplotlib.cm as cm
from utilmti850 import prepareSubplot


# takeSample(withReplacement, num, [seed]) randomly selects num elements
# optional seed parameter that one can set for reproducible results

data_values = (parsed_points_df
              .rdd
              .map(lambda lp: lp.features.toArray())
              .takeSample(False, 50, 47))

# You can uncomment the line below to see randomly selected features.  Th
# selected each time you run the cell because there is no set seed.  Note
# this cell with the line commented out when answering the lab quiz quest
# data_values = (parsedPointsDF
#                .rdd
#                .map(lambda lp: lp.features.toArray())
#                .takeSample(False, 50))

# generate layout and plot
fig, ax = prepareSubplot(
    np.arange(.5, 11, 1), np.arange(.5, 49, 1), figsize=(8,7),
    hideLabels=True, gridColor='#eeeeee', gridWidth=1.1
)
image = ax.imshow(data_values,interpolation='nearest', aspect='auto', cma
for x, y, s in zip(np.arange(-.125, 12, 1), np.repeat(-.75, 12), [str(x)
    ax.text(x, y, s, color='#999999', size='10')
ax.text(4.7, -3, 'Feature', color='#999999', size='11'), ax.set_ylabel('O
plt.show()
```



(1c) Find the range

Now let's examine the labels to find the range of song years. To do this, find the smallest and largest labels in the `parsed_points_df` .

We will use the min and max functions that are native to the DataFrames, and thus can be optimized using Spark's Catalyst Optimizer and Project Tungsten (don't worry about the technical details). This code will run faster than simply using the native min and max functions in Python. Use selectExpr to retrieve the min and max label values.

In [26]:
```
# TODO: Replace <FILL IN> with appropriate code

content_stats = parsed_points_df.selectExpr("min(label) as y_min", "max(l

min_year = content_stats['y_min']

max_year = content_stats['y_max']

print(min_year, max_year)
```

```
1922.0 2011.0
```

In [27]:
```
# TEST Find the range (1c)
testmti850.Test.assertEquals(len(parsed_points_df.first().features), 12,
                  'unexpected number of features in sample point')
sum_feat_two = parsed_points_df.rdd.map(lambda lp: lp.features[2]).sum()
testmti850.Test.assertTrue(np.allclose(sum_feat_two, 3158.96224351), 'par
year_range = max_year - min_year
testmti850.Test.assertTrue(year_range == 89, 'incorrect range for minYear
```

```
1 test passed.
Exception ignored in: <_io.BufferedWriter name=5>
Traceback (most recent call last):
  File "/opt/spark/python/lib/pyspark.zip/pyspark/daemon.py", line 200, in
manager
BrokenPipeError: [Errno 32] Broken pipe
1 test passed.
1 test passed.
```

(1d) Shift labels

As we just saw, the labels are years in the 1900s and 2000s. In learning problems, it is often natural to shift labels such that they start from zero. Starting with `parsed_points_df`, create a new DataFrame in which the labels are shifted such that smallest label equals zero (hint: use `select`). After, use withColumnRenamed to rename the appropriate columns to `features` and `label`.

In [33]:
```
# TODO: Replace <FILL IN> with appropriate code
parsed_data_df = parsed_points_df.select((parsed_points_df.label-min_year

# View the first point
print('\n{0}'.format(parsed_data_df.first()))
```

```
Row(label=79.0, features=DenseVector([0.8841, 0.6105, 0.6005, 0.4747, 0.24
72, 0.3573, 0.3441, 0.3396, 0.6009, 0.4257, 0.6049, 0.4192]))
Exception ignored in: <_io.BufferedWriter name=5>
Traceback (most recent call last):
  File "/opt/spark/python/lib/pyspark.zip/pyspark/daemon.py", line 200, in
manager
BrokenPipeError: [Errno 32] Broken pipe
```

In [34]:
```
# TEST Shift labels (1d)
old_sample_features = parsed_points_df.first().features
new_sample_features = parsed_data_df.first().features
testmti850.Test.assertTrue(np.allclose(old_sample_features, new_sample_fe
                'new features do not match old features')
sum_feat_two = parsed_data_df.rdd.map(lambda lp: lp.features[2]).sum()
testmti850.Test.assertTrue(np.allclose(sum_feat_two, 3158.96224351), 'par
min_year_new = parsed_data_df.groupBy().min('label').first()[0]
max_year_new = parsed_data_df.groupBy().max('label').first()[0]
testmti850.Test.assertTrue(min_year_new == 0, 'incorrect min year in shif
testmti850.Test.assertTrue(max_year_new == 89, 'incorrect max year in shi
Exception ignored in: <_io.BufferedWriter name=5>
Traceback (most recent call last):
  File "/opt/spark/python/lib/pyspark.zip/pyspark/daemon.py", line 200, in
manager
BrokenPipeError: [Errno 32] Broken pipe
Exception ignored in: <_io.BufferedWriter name=5>
Traceback (most recent call last):
  File "/opt/spark/python/lib/pyspark.zip/pyspark/daemon.py", line 200, in
manager
BrokenPipeError: [Errno 32] Broken pipe
1 test passed.

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

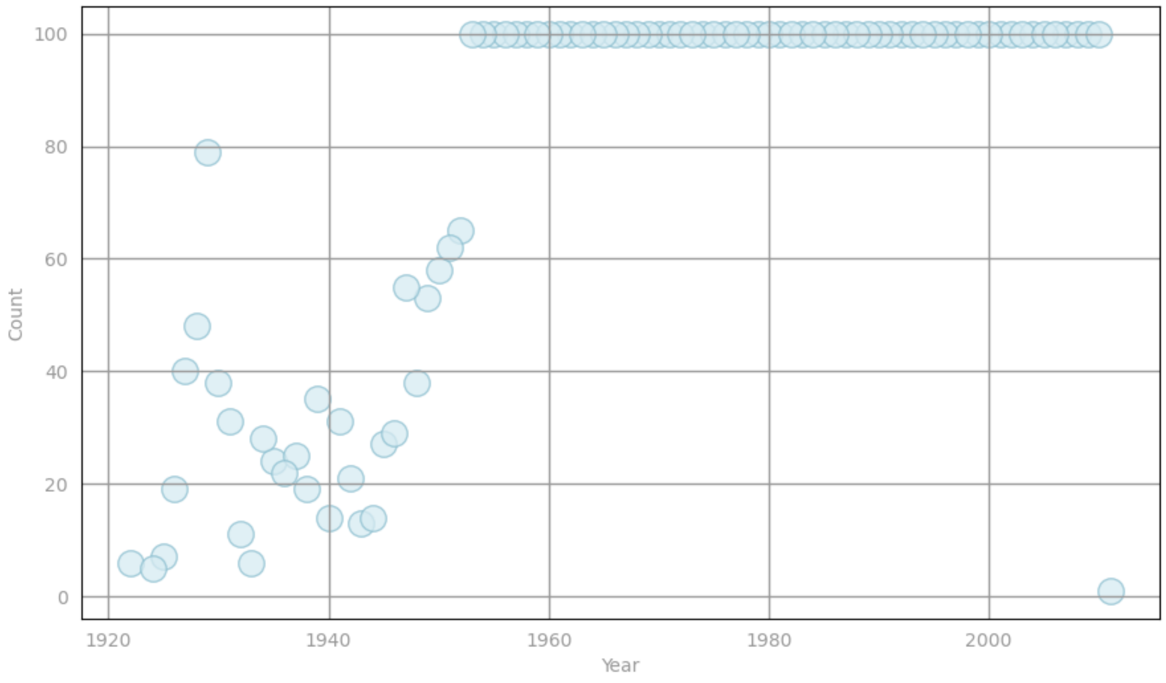
Visualization 2: Shifting labels

We will look at the labels before and after shifting them. Both scatter plots below visualize tuples storing:

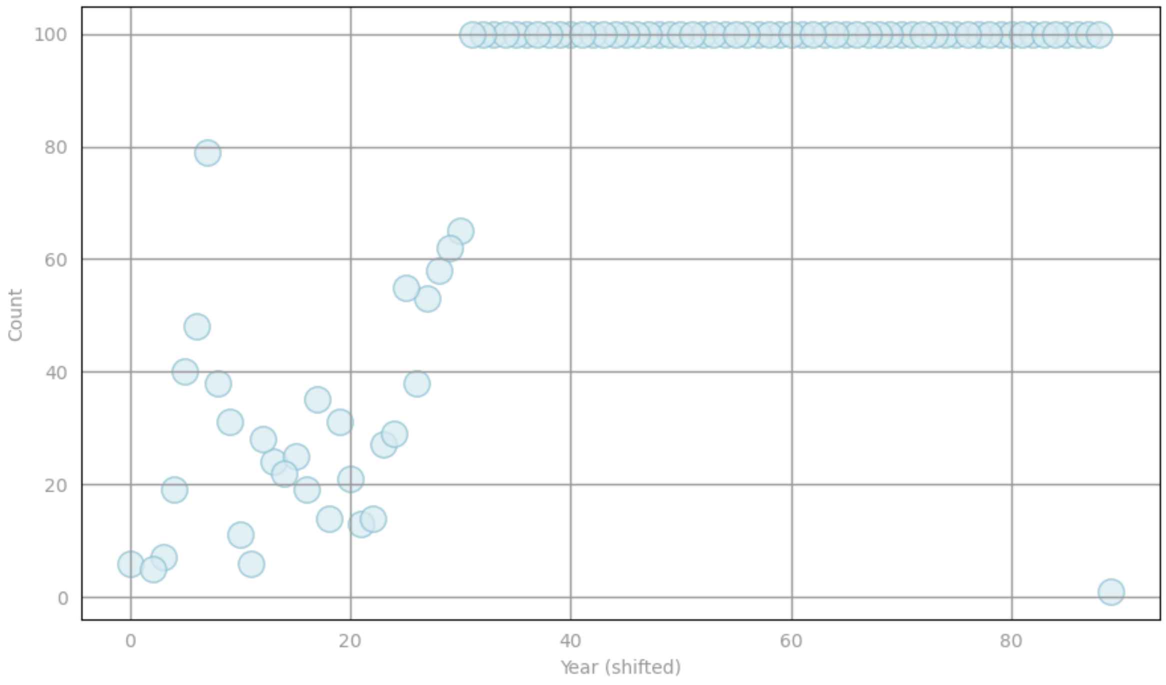
- a label value and
- the number of training points with this label.

The first scatter plot uses the initial labels, while the second one uses the shifted labels. Note that the two plots look the same except for the labels on the x-axis.

In [35]:
```python
# from operator import add

# get data for plot
old_data = (parsed_points_df
             .rdd
             .map(lambda lp: (lp.label, 1))
             .reduceByKey(lambda x, y: x + y) # alternatively, one can us
             .collect())                       # instead of the lambda fun
x, y = zip(*old_data)

# generate layout and plot data
fig, ax = prepareSubplot(np.arange(1920, 2050, 20), np.arange(0, 150, 20)
ax.scatter(x, y, s=14**2, c='#d6ebf2', edgecolors='#8cbfd0', alpha=0.75)
ax.set_xlabel('Year')
ax.set_ylabel('Count')
plt.show()
```



In [36]:
```
# get data for plot
new_data = (parsed_data_df
            .rdd
            .map(lambda lp: (lp.label, 1))
            .reduceByKey(lambda x, y: x + y)
            .collect())
x, y = zip(*new_data)

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



(1e) Training, validation, and test sets

We're almost done parsing our dataset, and our final task involves spliting the dataset into training, validation and test sets. Use the randomSplit method with the specified weights and seed to create DataFrames storing each of these datasets. Next, cache each of these DataFrames, as we will be accessing them multiple times in the remainder of this lab. Finally, compute the size of each dataset and verify that the sum of their sizes equals the value computed in Part (1a).

In [37]:
```
# TODO: Replace <FILL IN> with appropriate code
weights = [.8, .1, .1]
seed = 42

parsed_train_data_df, parsed_val_data_df, parsed_test_data_df = parsed_da

parsed_train_data_df.cache

parsed_val_data_df.cache

parsed_test_data_df.cache

n_train = parsed_train_data_df.count()

n_val = parsed_val_data_df.count()

n_test = parsed_test_data_df.count()

print(n_train, n_val, n_test, n_train + n_val + n_test)
print(parsed_data_df.count())
```

```
5437 612 675 6724
6724
```

In [38]:
```
# TEST Training, validation, and test sets (1e)
testmti850.Test.assertEquals(len(parsed_train_data_df.first().features),
                  'parsed_train_data_df has wrong number of features')
sum_feat_two = (parsed_train_data_df
                 .rdd
                 .map(lambda lp: lp.features[2])
                 .sum())
sum_feat_three = (parsed_val_data_df
                   .rdd
                   .map(lambda lp: lp.features[3])
                   .reduce(lambda x, y: x + y))
sum_feat_four = (parsed_test_data_df
                  .rdd
                  .map(lambda lp: lp.features[4])
                  .reduce(lambda x, y: x + y))
testmti850.Test.assertTrue(np.allclose([sum_feat_two, sum_feat_three, sum
                            2526.87757656, 297.340394298, 184.235876654),
                 'parsed Train, Val, Test data has unexpected values')
testmti850.Test.assertTrue(n_train + n_val + n_test == 6724, 'unexpected

testmti850.Test.assertEquals(n_train, 5437, 'unexpected value for nTrain'

testmti850.Test.assertEquals(n_val, 612, 'unexpected value for nVal')

testmti850.Test.assertEquals(n_test, 675, 'unexpected value for nTest')
```

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

Part 2: Create and evaluate a baseline model

(2a) Average label

A very simple yet natural baseline model is one where we always make the same prediction independent of the given data point, using the average label in the training set as the constant prediction value. Compute this value, which is the average (shifted) song year for the training set. Use `selectExpr` and `first()` from the DataFrame API.

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

average_train_year = parsed_train_data_df.selectExpr('avg(label)').first(

print(average_train_year)
```

```
53.54919992643001
```

In [45]:
```
# TEST Average label (2a)
testmti850.Test.assertTrue(
    np.allclose(average_train_year, 53.54919992643001), 'incorrect value
)
```

```
1 test passed.
```

(2b) Root mean squared error

We naturally would like to see how well this naive baseline performs. We will use root mean squared error (RMSE) for evaluation purposes. Using Regression Evaluator, compute the RMSE given a dataset of *(prediction, label)* tuples.

In [41]:
```
# TODO: Replace <FILL IN> with appropriate code
from pyspark.ml.evaluation import RegressionEvaluator

preds_and_labels = [(1., 3.), (2., 1.), (2., 2.)]

preds_and_labels_df = spark.createDataFrame(preds_and_labels, ["predictio

evaluator = RegressionEvaluator(predictionCol="prediction", labelCol="lab

def calc_RMSE(dataset):
    """Calculates the root mean squared error for an dataset of (predicti

    Args:
        dataset (DataFrame of (float, float)): A `DataFrame` consisting o

    Returns:
        float: The square root of the mean of the squared errors.
    """
    return evaluator.evaluate(dataset)

example_rmse = calc_RMSE(preds_and_labels_df)

print(example_rmse)
# RMSE = sqrt[((1-3)^2 + (2-1)^2 + (2-2)^2) / 3] = 1.291
```

```
1.2909944487358058
```

In [42]:
```
testmti850.Test.assertTrue(np.allclose(example_rmse, 1.2909944487358058),
```

```
1 test passed.
```

(2c) Training, validation and test RMSE

Now let's calculate the training, validation and test RMSE of our baseline model. To do this, first create DataFrames of *(prediction, label)* tuples for each dataset, and then call `calc_RMSE()` . Note that each RMSE can be interpreted as the average prediction error for the given dataset (in terms of number of years). You can use createDataFrame to make a DataFrame with the column names of "prediction" and "label" from an RDD.

In [46]:
```
# TODO: Replace <FILL IN> with appropriate code

preds_and_labels_train = parsed_train_data_df.rdd.map(lambda row: (averag

preds_and_labels_train_df = spark.createDataFrame(preds_and_labels_train,

rmse_train_base = calc_RMSE(preds_and_labels_train_df)


preds_and_labels_val = parsed_val_data_df.rdd.map(lambda row: (average_tr

preds_and_labels_val_df = spark.createDataFrame(preds_and_labels_val, ["p

rmse_val_base = calc_RMSE(preds_and_labels_val_df)


preds_and_labels_test = parsed_test_data_df.rdd.map(lambda row: (average_

preds_and_labels_test_df = spark.createDataFrame(preds_and_labels_test, [

rmse_test_base = calc_RMSE(preds_and_labels_test_df)

print('Baseline Train RMSE = {0:.3f}'.format(rmse_train_base))
print('Baseline Validation RMSE = {0:.3f}'.format(rmse_val_base))
print('Baseline Test RMSE = {0:.3f}'.format(rmse_test_base))
```

```
Baseline Train RMSE = 21.598
Baseline Validation RMSE = 20.759
Baseline Test RMSE = 20.547
```

In [48]:
```
# TEST Training, validation and test RMSE (2c)
testmti850.Test.assertTrue(
    np.allclose(
        [rmse_train_base, rmse_val_base, rmse_test_base],
        [21.598132, 20.759189, 20.546997]
    ),
    'incorrect RMSE values'
)
```

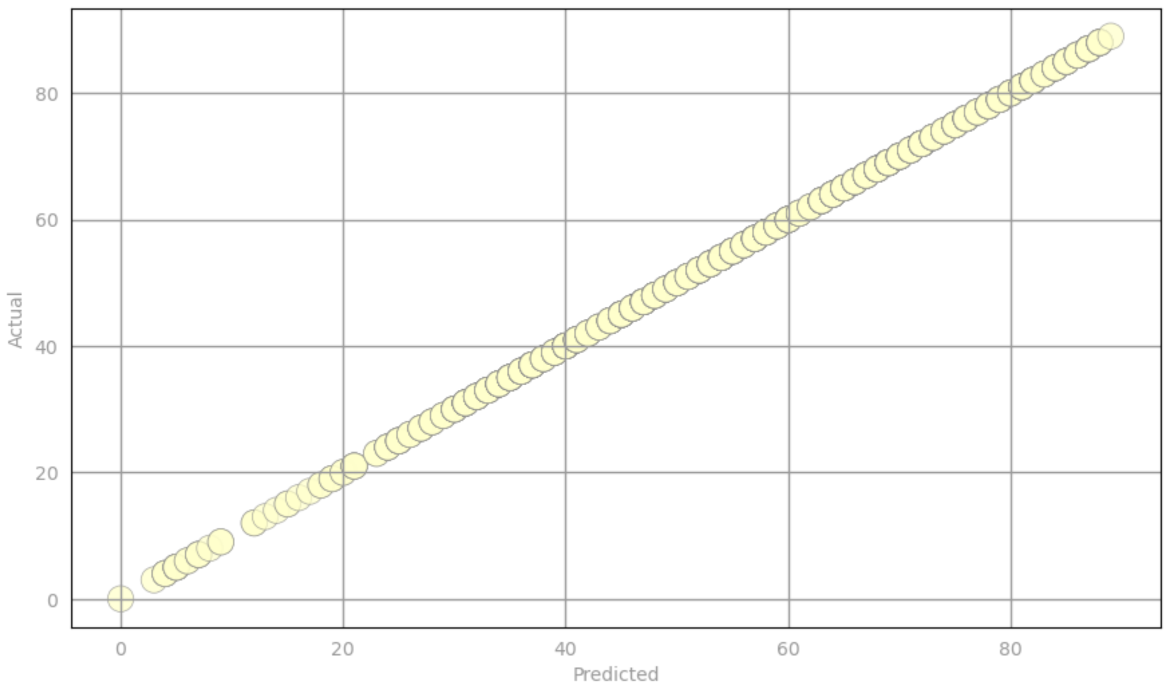
```
1 test passed.
```

Visualization 3: Predicted vs. actual

We will visualize predictions on the validation dataset. The scatter plots below visualize tuples storing i) the predicted value and ii) true label. The first scatter plot represents the ideal situation where the predicted value exactly equals the true label, while the second plot uses the baseline predictor (i.e., `average_train_year`) for all predicted values. Further note that the points in the scatter plots are color-coded, ranging from light yellow when the true and predicted values are equal to bright red when they drastically differ.

In [49]:
```python
from matplotlib.colors import ListedColormap, Normalize
from matplotlib.cm import get_cmap
cmap = get_cmap('YlOrRd')
norm = Normalize()

def squared_error(label, prediction):
    """Calculates the squared error for a single prediction."""
    return float((label - prediction)**2)

actual = np.asarray(parsed_val_data_df
                    .select('label')
                    .collect())
error = np.asarray(parsed_val_data_df
                   .rdd
                   .map(lambda lp: (lp.label, lp.label))
                   .map(lambda lp: squared_error(lp[0], lp[1]))
                   .collect())
clrs = cmap(np.asarray(norm(error)))[:, :3]

fig, ax = prepareSubplot(np.arange(0, 100, 20), np.arange(0, 100, 20))
ax.scatter(actual, actual, s=14**2, c=clrs, edgecolors='#888888', alpha=0
ax.set_xlabel('Predicted')
ax.set_ylabel('Actual')
plt.show()
```

```
/tmp/ipykernel_22549/2990054735.py:3: MatplotlibDeprecationWarning: The ge
t_cmap function was deprecated in Matplotlib 3.7 and will be removed in 3.
11. Use ``matplotlib.colormaps[name]`` or ``matplotlib.colormaps.get_cmap
()`` or ``pyplot.get_cmap()`` instead.
  cmap = get_cmap('YlOrRd')
```



In [50]:
```
def squared_error(label, prediction):
    """Calculates the squared error for a single prediction."""
    return float((label - prediction) ** 2)

predictions = np.asarray(parsed_val_data_df
                         .rdd
                         .map(lambda lp: average_train_year)
                         .collect())
error = np.asarray(parsed_val_data_df
                   .rdd
                   .map(lambda lp: (lp.label, average_train_year['avg(lab
                   .map(lambda lp: squared_error(lp[0], lp[1]))
                   .collect())

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

fig, ax = prepareSubplot(np.arange(53.0, 55.0, 0.5), np.arange(0, 100, 20
ax.set_xlim(53, 55)
ax.scatter(predictions, actual, s=14**2, c=clrs, edgecolors='#888888', al
ax.set_xlabel('Predicted')
ax.set_ylabel('Actual')
plt.show()
```

```
25/11/28 01:59:06 ERROR Executor: Exception in task 0.0 in stage 80.0 (TID
67)
org.apache.spark.api.python.PythonException: Traceback (most recent call l
ast):
  File "/opt/spark/python/lib/pyspark.zip/pyspark/worker.py", line 2044, i
n main
    process()
    ~~~~~~~^^
  File "/opt/spark/python/lib/pyspark.zip/pyspark/worker.py", line 2036, i
n process
    serializer.dump_stream(out_iter, outfile)
    ~~~~~~~~~~~~~~~~~~~~~~^^^^^^^^^^^^^^^^^^^
  File "/opt/spark/python/lib/pyspark.zip/pyspark/serializers.py", line 27
3, in dump_stream
    vs = list(itertools.islice(iterator, batch))
  File "/opt/spark/python/lib/pyspark.zip/pyspark/util.py", line 131, in w
rapper
    return f(*args, **kwargs)
  File "/tmp/ipykernel_22549/2256473714.py", line 11, in <lambda>
TypeError: 'float' object is not subscriptable

        at org.apache.spark.api.python.BasePythonRunner$ReaderIterator.han
dlePythonException(PythonRunner.scala:581)
        at org.apache.spark.api.python.PythonRunner$$anon$3.read(PythonRun
ner.scala:940)
        at org.apache.spark.api.python.PythonRunner$$anon$3.read(PythonRun
ner.scala:925)
        at org.apache.spark.api.python.BasePythonRunner$ReaderIterator.has
Next(PythonRunner.scala:532)
        at org.apache.spark.InterruptibleIterator.hasNext(InterruptibleIte
rator.scala:37)
        at scala.collection.mutable.Growable.addAll(Growable.scala:61)
        at scala.collection.mutable.Growable.addAll$(Growable.scala:57)
        at scala.collection.mutable.ArrayBuilder.addAll(ArrayBuilder.scal
a:75)
        at scala.collection.IterableOnceOps.toArray(IterableOnce.scala:150
5)
        at scala.collection.IterableOnceOps.toArray$(IterableOnce.scala:14
98)
        at org.apache.spark.InterruptibleIterator.toArray(InterruptibleIte
rator.scala:28)
        at org.apache.spark.rdd.RDD.$anonfun$collect$2(RDD.scala:1057)
        at org.apache.spark.SparkContext.$anonfun$runJob$5(SparkContext.sc
ala:2524)
        at org.apache.spark.scheduler.ResultTask.runTask(ResultTask.scal
a:93)
        at org.apache.spark.TaskContext.runTaskWithListeners(TaskContext.s
cala:171)
        at org.apache.spark.scheduler.Task.run(Task.scala:147)
        at org.apache.spark.executor.Executor$TaskRunner.$anonfun$run$5(Ex
ecutor.scala:647)
        at org.apache.spark.util.SparkErrorUtils.tryWithSafeFinally(SparkE
rrorUtils.scala:80)
        at org.apache.spark.util.SparkErrorUtils.tryWithSafeFinally$(Spark
ErrorUtils.scala:77)
        at org.apache.spark.util.Utils$.tryWithSafeFinally(Utils.scala:99)
        at org.apache.spark.executor.Executor$TaskRunner.run(Executor.scal
a:650)
        at java.base/java.util.concurrent.ThreadPoolExecutor.runWorker(Thr
eadPoolExecutor.java:1144)
```

```
        at java.base/java.util.concurrent.ThreadPoolExecutor$Worker.run(Th
readPoolExecutor.java:642)
        at java.base/java.lang.Thread.run(Thread.java:1583)
25/11/28 01:59:06 WARN TaskSetManager: Lost task 0.0 in stage 80.0 (TID 6
7) (10.0.2.15 executor driver): org.apache.spark.api.python.PythonExceptio
n: Traceback (most recent call last):
  File "/opt/spark/python/lib/pyspark.zip/pyspark/worker.py", line 2044, i
n main
    process()
    ~~~~~~~^^
  File "/opt/spark/python/lib/pyspark.zip/pyspark/worker.py", line 2036, i
n process
    serializer.dump_stream(out_iter, outfile)
    ~~~~~~~~~~~~~~~~~~~~~~^^^^^^^^^^^^^^^^^^^
  File "/opt/spark/python/lib/pyspark.zip/pyspark/serializers.py", line 27
3, in dump_stream
    vs = list(itertools.islice(iterator, batch))
  File "/opt/spark/python/lib/pyspark.zip/pyspark/util.py", line 131, in w
rapper
    return f(*args, **kwargs)
  File "/tmp/ipykernel_22549/2256473714.py", line 11, in <lambda>
TypeError: 'float' object is not subscriptable

        at org.apache.spark.api.python.BasePythonRunner$ReaderIterator.han
dlePythonException(PythonRunner.scala:581)
        at org.apache.spark.api.python.PythonRunner$$anon$3.read(PythonRun
ner.scala:940)
        at org.apache.spark.api.python.PythonRunner$$anon$3.read(PythonRun
ner.scala:925)
        at org.apache.spark.api.python.BasePythonRunner$ReaderIterator.has
Next(PythonRunner.scala:532)
        at org.apache.spark.InterruptibleIterator.hasNext(InterruptibleIte
rator.scala:37)
        at scala.collection.mutable.Growable.addAll(Growable.scala:61)
        at scala.collection.mutable.Growable.addAll$(Growable.scala:57)
        at scala.collection.mutable.ArrayBuilder.addAll(ArrayBuilder.scal
a:75)
        at scala.collection.IterableOnceOps.toArray(IterableOnce.scala:150
5)
        at scala.collection.IterableOnceOps.toArray$(IterableOnce.scala:14
98)
        at org.apache.spark.InterruptibleIterator.toArray(InterruptibleIte
rator.scala:28)
        at org.apache.spark.rdd.RDD.$anonfun$collect$2(RDD.scala:1057)
        at org.apache.spark.SparkContext.$anonfun$runJob$5(SparkContext.sc
ala:2524)
        at org.apache.spark.scheduler.ResultTask.runTask(ResultTask.scal
a:93)
        at org.apache.spark.TaskContext.runTaskWithListeners(TaskContext.s
cala:171)
        at org.apache.spark.scheduler.Task.run(Task.scala:147)
        at org.apache.spark.executor.Executor$TaskRunner.$anonfun$run$5(Ex
ecutor.scala:647)
        at org.apache.spark.util.SparkErrorUtils.tryWithSafeFinally(SparkE
rrorUtils.scala:80)
        at org.apache.spark.util.SparkErrorUtils.tryWithSafeFinally$(Spark
ErrorUtils.scala:77)
        at org.apache.spark.util.Utils$.tryWithSafeFinally(Utils.scala:99)
        at org.apache.spark.executor.Executor$TaskRunner.run(Executor.scal
a:650)
```

```
        at java.base/java.util.concurrent.ThreadPoolExecutor.runWorker(Thr
eadPoolExecutor.java:1144)
        at java.base/java.util.concurrent.ThreadPoolExecutor$Worker.run(Th
readPoolExecutor.java:642)
        at java.base/java.lang.Thread.run(Thread.java:1583)

25/11/28 01:59:06 ERROR TaskSetManager: Task 0 in stage 80.0 failed 1 time
s; aborting job
```

```
---------------------------------------------------------------------------
-
Py4JJavaError                             Traceback (most recent call las
t)
Cell In[50], line 13
      3     return float((label - prediction) ** 2)
      5 predictions = np.asarray(parsed_val_data_df
      6                          .rdd
      7                          .map(lambda lp: average_train_year)
      8                          .collect())
      9 error = np.asarray(parsed_val_data_df
     10                    .rdd
     11                    .map(lambda lp: (lp.label, average_train_year
['avg(label)']))
     12                    .map(lambda lp: squared_error(lp[0], lp[1]))
---> 13                    .collect())
     15 norm = Normalize()
     16 clrs = cmap(np.asarray(norm(error)))[:, : 3]

File /opt/spark/python/pyspark/core/rdd.py:1700, in RDD.collect(self)
   1698 with SCCallSiteSync(self.context):
   1699     assert self.ctx._jvm is not None
-> 1700     sock_info = self.ctx._jvm.PythonRDD.collectAndServe(self._jrd
d.rdd())
   1701 return list(_load_from_socket(sock_info, self._jrdd_deserializer))

File /opt/spark/python/lib/py4j-0.10.9.9-src.zip/py4j/java_gateway.py:136
2, in JavaMember.__call__(self, *args)
   1356 command = proto.CALL_COMMAND_NAME +\
   1357     self.command_header +\
   1358     args_command +\
   1359     proto.END_COMMAND_PART
   1361 answer = self.gateway_client.send_command(command)
-> 1362 return_value = get_return_value(
   1363     answer, self.gateway_client, self.target_id, self.name)
   1365 for temp_arg in temp_args:
   1366     if hasattr(temp_arg, "_detach"):

File /opt/spark/python/pyspark/errors/exceptions/captured.py:282, in captu
re_sql_exception.<locals>.deco(*a, **kw)
    279 from py4j.protocol import Py4JJavaError
    281 try:
--> 282     return f(*a, **kw)
    283 except Py4JJavaError as e:
    284     converted = convert_exception(e.java_exception)

File /opt/spark/python/lib/py4j-0.10.9.9-src.zip/py4j/protocol.py:327, in
get_return_value(answer, gateway_client, target_id, name)
    325 value = OUTPUT_CONVERTER[type](answer[2:], gateway_client)
    326 if answer[1] == REFERENCE_TYPE:
--> 327     raise Py4JJavaError(
    328         "An error occurred while calling {0}{1}{2}.\n".
    329         format(target_id, ".", name), value)
    330 else:
    331     raise Py4JError(
    332         "An error occurred while calling {0}{1}{2}. Trace:\n
{3}\n".
    333         format(target_id, ".", name, value))

Py4JJavaError: An error occurred while calling z:org.apache.spark.api.pyth
```

```
on.PythonRDD.collectAndServe.
: org.apache.spark.SparkException: Job aborted due to stage failure: Task
0 in stage 80.0 failed 1 times, most recent failure: Lost task 0.0 in stag
e 80.0 (TID 67) (10.0.2.15 executor driver): org.apache.spark.api.python.P
ythonException: Traceback (most recent call last):
  File "/opt/spark/python/lib/pyspark.zip/pyspark/worker.py", line 2044, i
n main
    process()
    ~~~~~~~^^
  File "/opt/spark/python/lib/pyspark.zip/pyspark/worker.py", line 2036, i
n process
    serializer.dump_stream(out_iter, outfile)
    ~~~~~~~~~~~~~~~~~~~~~~^^^^^^^^^^^^^^^^^^^
  File "/opt/spark/python/lib/pyspark.zip/pyspark/serializers.py", line 27
3, in dump_stream
    vs = list(itertools.islice(iterator, batch))
  File "/opt/spark/python/lib/pyspark.zip/pyspark/util.py", line 131, in w
rapper
    return f(*args, **kwargs)
  File "/tmp/ipykernel_22549/2256473714.py", line 11, in <lambda>
TypeError: 'float' object is not subscriptable

        at org.apache.spark.api.python.BasePythonRunner$ReaderIterator.han
dlePythonException(PythonRunner.scala:581)
        at org.apache.spark.api.python.PythonRunner$$anon$3.read(PythonRun
ner.scala:940)
        at org.apache.spark.api.python.PythonRunner$$anon$3.read(PythonRun
ner.scala:925)
        at org.apache.spark.api.python.BasePythonRunner$ReaderIterator.has
Next(PythonRunner.scala:532)
        at org.apache.spark.InterruptibleIterator.hasNext(InterruptibleIte
rator.scala:37)
        at scala.collection.mutable.Growable.addAll(Growable.scala:61)
        at scala.collection.mutable.Growable.addAll$(Growable.scala:57)
        at scala.collection.mutable.ArrayBuilder.addAll(ArrayBuilder.scal
a:75)
        at scala.collection.IterableOnceOps.toArray(IterableOnce.scala:150
5)
        at scala.collection.IterableOnceOps.toArray$(IterableOnce.scala:14
98)
        at org.apache.spark.InterruptibleIterator.toArray(InterruptibleIte
rator.scala:28)
        at org.apache.spark.rdd.RDD.$anonfun$collect$2(RDD.scala:1057)
        at org.apache.spark.SparkContext.$anonfun$runJob$5(SparkContext.sc
ala:2524)
        at org.apache.spark.scheduler.ResultTask.runTask(ResultTask.scal
a:93)
        at org.apache.spark.TaskContext.runTaskWithListeners(TaskContext.s
cala:171)
        at org.apache.spark.scheduler.Task.run(Task.scala:147)
        at org.apache.spark.executor.Executor$TaskRunner.$anonfun$run$5(Ex
ecutor.scala:647)
        at org.apache.spark.util.SparkErrorUtils.tryWithSafeFinally(SparkE
rrorUtils.scala:80)
        at org.apache.spark.util.SparkErrorUtils.tryWithSafeFinally$(Spark
ErrorUtils.scala:77)
        at org.apache.spark.util.Utils$.tryWithSafeFinally(Utils.scala:99)
        at org.apache.spark.executor.Executor$TaskRunner.run(Executor.scal
a:650)
        at java.base/java.util.concurrent.ThreadPoolExecutor.runWorker(Thr
```

```
eadPoolExecutor.java:1144)
        at java.base/java.util.concurrent.ThreadPoolExecutor$Worker.run(Th
readPoolExecutor.java:642)
        at java.base/java.lang.Thread.run(Thread.java:1583)

Driver stacktrace:
        at org.apache.spark.scheduler.DAGScheduler.$anonfun$abortStage$3(D
AGScheduler.scala:2935)
        at scala.Option.getOrElse(Option.scala:201)
        at org.apache.spark.scheduler.DAGScheduler.$anonfun$abortStage$2(D
AGScheduler.scala:2935)
        at org.apache.spark.scheduler.DAGScheduler.$anonfun$abortStage$2$a
dapted(DAGScheduler.scala:2927)
        at scala.collection.immutable.List.foreach(List.scala:334)
        at org.apache.spark.scheduler.DAGScheduler.abortStage(DAGSchedule
r.scala:2927)
        at org.apache.spark.scheduler.DAGScheduler.$anonfun$handleTaskSetF
ailed$1(DAGScheduler.scala:1295)
        at org.apache.spark.scheduler.DAGScheduler.$anonfun$handleTaskSetF
ailed$1$adapted(DAGScheduler.scala:1295)
        at scala.Option.foreach(Option.scala:437)
        at org.apache.spark.scheduler.DAGScheduler.handleTaskSetFailed(DAG
Scheduler.scala:1295)
        at org.apache.spark.scheduler.DAGSchedulerEventProcessLoop.doOnRec
eive(DAGScheduler.scala:3207)
        at org.apache.spark.scheduler.DAGSchedulerEventProcessLoop.onRecei
ve(DAGScheduler.scala:3141)
        at org.apache.spark.scheduler.DAGSchedulerEventProcessLoop.onRecei
ve(DAGScheduler.scala:3130)
        at org.apache.spark.util.EventLoop$$anon$1.run(EventLoop.scala:50)
        at org.apache.spark.scheduler.DAGScheduler.runJob(DAGScheduler.sca
la:1009)
        at org.apache.spark.SparkContext.runJob(SparkContext.scala:2484)
        at org.apache.spark.SparkContext.runJob(SparkContext.scala:2505)
        at org.apache.spark.SparkContext.runJob(SparkContext.scala:2524)
        at org.apache.spark.SparkContext.runJob(SparkContext.scala:2549)
        at org.apache.spark.rdd.RDD.$anonfun$collect$1(RDD.scala:1057)
        at org.apache.spark.rdd.RDDOperationScope$.withScope(RDDOperationS
cope.scala:151)
        at org.apache.spark.rdd.RDDOperationScope$.withScope(RDDOperationS
cope.scala:112)
        at org.apache.spark.rdd.RDD.withScope(RDD.scala:417)
        at org.apache.spark.rdd.RDD.collect(RDD.scala:1056)
        at org.apache.spark.api.python.PythonRDD$.collectAndServe(PythonRD
D.scala:203)
        at org.apache.spark.api.python.PythonRDD.collectAndServe(PythonRD
D.scala)
        at java.base/jdk.internal.reflect.NativeMethodAccessorImpl.invoke0
(Native Method)
        at java.base/jdk.internal.reflect.NativeMethodAccessorImpl.invoke
(NativeMethodAccessorImpl.java:75)
        at java.base/jdk.internal.reflect.DelegatingMethodAccessorImpl.inv
oke(DelegatingMethodAccessorImpl.java:52)
        at java.base/java.lang.reflect.Method.invoke(Method.java:580)
        at py4j.reflection.MethodInvoker.invoke(MethodInvoker.java:244)
        at py4j.reflection.ReflectionEngine.invoke(ReflectionEngine.java:3
74)
        at py4j.Gateway.invoke(Gateway.java:282)
        at py4j.commands.AbstractCommand.invokeMethod(AbstractCommand.jav
a:132)
```

```
        at py4j.commands.CallCommand.execute(CallCommand.java:79)
        at py4j.ClientServerConnection.waitForCommands(ClientServerConnect
ion.java:184)
        at py4j.ClientServerConnection.run(ClientServerConnection.java:10
8)
        at java.base/java.lang.Thread.run(Thread.java:1583)
Caused by: org.apache.spark.api.python.PythonException: Traceback (most re
cent call last):
  File "/opt/spark/python/lib/pyspark.zip/pyspark/worker.py", line 2044, i
n main
    process()
    ~~~~~~~^^
  File "/opt/spark/python/lib/pyspark.zip/pyspark/worker.py", line 2036, i
n process
    serializer.dump_stream(out_iter, outfile)
    ~~~~~~~~~~~~~~~~~~~~~~^^^^^^^^^^^^^^^^^^^
  File "/opt/spark/python/lib/pyspark.zip/pyspark/serializers.py", line 27
3, in dump_stream
    vs = list(itertools.islice(iterator, batch))
  File "/opt/spark/python/lib/pyspark.zip/pyspark/util.py", line 131, in w
rapper
    return f(*args, **kwargs)
  File "/tmp/ipykernel_22549/2256473714.py", line 11, in <lambda>
TypeError: 'float' object is not subscriptable

        at org.apache.spark.api.python.BasePythonRunner$ReaderIterator.han
dlePythonException(PythonRunner.scala:581)
        at org.apache.spark.api.python.PythonRunner$$anon$3.read(PythonRun
ner.scala:940)
        at org.apache.spark.api.python.PythonRunner$$anon$3.read(PythonRun
ner.scala:925)
        at org.apache.spark.api.python.BasePythonRunner$ReaderIterator.has
Next(PythonRunner.scala:532)
        at org.apache.spark.InterruptibleIterator.hasNext(InterruptibleIte
rator.scala:37)
        at scala.collection.mutable.Growable.addAll(Growable.scala:61)
        at scala.collection.mutable.Growable.addAll$(Growable.scala:57)
        at scala.collection.mutable.ArrayBuilder.addAll(ArrayBuilder.scal
a:75)
        at scala.collection.IterableOnceOps.toArray(IterableOnce.scala:150
5)
        at scala.collection.IterableOnceOps.toArray$(IterableOnce.scala:14
98)
        at org.apache.spark.InterruptibleIterator.toArray(InterruptibleIte
rator.scala:28)
        at org.apache.spark.rdd.RDD.$anonfun$collect$2(RDD.scala:1057)
        at org.apache.spark.SparkContext.$anonfun$runJob$5(SparkContext.sc
ala:2524)
        at org.apache.spark.scheduler.ResultTask.runTask(ResultTask.scal
a:93)
        at org.apache.spark.TaskContext.runTaskWithListeners(TaskContext.s
cala:171)
        at org.apache.spark.scheduler.Task.run(Task.scala:147)
        at org.apache.spark.executor.Executor$TaskRunner.$anonfun$run$5(Ex
ecutor.scala:647)
        at org.apache.spark.util.SparkErrorUtils.tryWithSafeFinally(SparkE
rrorUtils.scala:80)
        at org.apache.spark.util.SparkErrorUtils.tryWithSafeFinally$(Spark
ErrorUtils.scala:77)
        at org.apache.spark.util.Utils$.tryWithSafeFinally(Utils.scala:99)
```

```
        at org.apache.spark.executor.Executor$TaskRunner.run(Executor.scal
a:650)
        at java.base/java.util.concurrent.ThreadPoolExecutor.runWorker(Thr
eadPoolExecutor.java:1144)
        at java.base/java.util.concurrent.ThreadPoolExecutor$Worker.run(Th
readPoolExecutor.java:642)
        ... 1 more
```

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

(3a) Gradient summand

Now let's see if we can do better via linear regression, training a model via gradient descent (we'll omit the intercept for now). Recall that 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. Use the `DenseVector` dot method.

In [51]:
```
from pyspark.mllib.linalg import DenseVector

# OBS: You are advised to use pyspark.ml.linalg.DenseVector and adapt the
```

In [55]:
```
# TODO: Replace <FILL IN> with appropriate code
def gradient_summand(weights, lp):
    """Calculates the gradient summand for a given weight and `LabeledPoi

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

    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`.  Th
    """

    return (weights.dot(lp.features)-lp.label)*(lp.features)


example_w = DenseVector([1, 1, 1])
example_lp = LabeledPoint(2.0, [3, 1, 4])
# gradient_summand = (dot([1 1 1], [3 1 4]) - 2) * [3 1 4] = (8 - 2) * [3
summand_one = gradient_summand(example_w, example_lp)
print(summand_one)
```

```
example_w = DenseVector([.24, 1.2, -1.4])
example_lp = LabeledPoint(3.0, [-1.4, 4.2, 2.1])
summand_two = gradient_summand(example_w, example_lp)
print(summand_two)
```

```
[18.0,6.0,24.0]
[1.7304000000000002,-5.191200000000001,-2.5956000000000006]
```

In [56]:

```
# TEST Gradient summand (3a)
testmti850.Test.assertTrue(np.allclose(summand_one, [18., 6., 24.]), 'inc
testmti850.Test.assertTrue(np.allclose(summand_two, [1.7304,-5.1912,-2.59
```

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

(3b) Use weights to make predictions

Next, implement a `get_labeled_predictions` function that takes in weights and an observation's `LabeledPoint` and returns a *(prediction, label)* tuple. Note that we can predict by computing the dot product between weights and an observation's features.

In [60]:

```
# TODO: Replace <FILL IN> with appropriate code
def get_labeled_prediction(weights, observation):
    """Calculates predictions and returns a (prediction, label) tuple.

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

    Args:
        weights (np.ndarray): An array with one weight for each features
        observation (LabeledPoint): A `LabeledPoint` that contain the cor
            features for the data point.

    Returns:
        tuple: A (prediction, label) tuple. Convert the return type of th
    """

    return (float(weights.dot(observation.features)), float(observation.l

weights = np.array([1.0, 1.5])

prediction_example = spark.sparkContext.parallelize([LabeledPoint(2, np.a
                                                     LabeledPoint(1.5, np.array([.5, .5])

preds_and_labels_example = prediction_example.map(lambda lp: get_labeled_

print(preds_and_labels_example.collect())
```

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

In [61]:
```
# TEST Use weights to make predictions (3b)
testmti850.Test.assertTrue(isinstance(preds_and_labels_example.first()[0]
testmti850.Test.assertEquals(preds_and_labels_example.collect(), [(1.75,
                  'incorrect definition for getLabeledPredictions')
```

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

(3c) Gradient descent

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

In [65]:
```
# TODO: Replace <FILL IN> with appropriate code

def linreg_gradient_descent(train_data, num_iters):
    """Calculates the weights and error for a linear regression model tra

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

    Args:
        train_data (RDD of LabeledPoint): The labeled data for use in tra
        num_iters (int): The number of iterations of gradient descent to

    Returns:
        (np.ndarray, np.ndarray): A tuple of (weights, training errors).
            final weights (one weight per feature) for the model, and tra
            an error (RMSE) for each iteration of the algorithm.
    """

    # The length of the training data
    n = train_data.count()

    # The number of features in the training data
    d = len(train_data.first().features)
    w = np.zeros(d)
    alpha = 1.0

    # We will compute and store the training error after each iteration
    error_train = np.zeros(num_iters)
    for i in range(num_iters):
        # Use get_labeled_prediction from (3b) with trainData to obtain a
        # tuples.  Note that the weights all equal 0 for the first iterat
        # have large errors to start.

        preds_and_labels_train = train_data.map(lambda lp: get_labeled_pr

        preds_and_labels_train_df = spark.createDataFrame(preds_and_label

        error_train[i] = calc_RMSE(preds_and_labels_train_df)

        # Calculate the `gradient`.  Make use of the `gradient_summand` f
        # Note that `gradient` should be a `DenseVector` of length `d`.
        gradient = train_data.map(lambda lp: gradient_summand(w, lp)).red

        # Update the weights
        alpha_i = alpha / (n * np.sqrt(i + 1))

        w -= alpha_i*gradient

    return w, error_train

# create a toy dataset with n = 10, d = 3, and then run 5 iterations of g
# note: the resulting model will not be useful; the goal here is to verif
# linreg_gradient_descent is working properly
example_n = 10
example_d = 3
example_data = (
    spark
    .sparkContext
    .parallelize(parsed_train_data_df.take(example_n))
    .map(lambda lp: LabeledPoint(lp.label, lp.features[0:example_d]))
)
print(example_data.take(2))

example_num_iters = 5
example_weights, example_error_train = linreg_gradient_descent(example_da


print(example_weights)
print(example_error_train)
```

```
[LabeledPoint(0.0, [0.581044957651,0.233509721704,0.819061699014]), Labele
dPoint(0.0, [0.63944087626,0.65293056486,0.343546487474])]
[0.59874548 0.16755145 1.19239355]
[1.73205081 1.3096377  1.27962073 1.26123272 1.24682059]
```

In [67]:
```
# TEST Gradient descent (3c)
expected_output = [0.59874548, 0.16755145, 1.19239355]
testmti850.Test.assertTrue(np.allclose(example_weights, expected_output),
expected_error = [1.73205081, 1.3096377, 1.27962073, 1.26123272, 1.246820
testmti850.Test.assertTrue(np.allclose(example_error_train, expected_erro
                'value of exampleErrorTrain is incorrect')
```

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

(3d) Train the model

Now let's train a linear regression model on all of our training data and evaluate its accuracy on the validation set. Note that the test set will not be used here. If we evaluated the model on the test set, we would bias our final results.

We've already done much of the required work: we computed the number of features in Part (1b); we created the training and validation datasets and computed their sizes in Part (1e); and, we wrote a function to compute RMSE in Part (2b).

In [70]:
```
# TODO: Replace <FILL IN> with appropriate code
num_iters = 50

weights_LR0, error_train_LR0 = linreg_gradient_descent(parsed_train_data_

preds_and_labels = parsed_val_data_df.rdd.map(lambda lp: (get_labeled_pre
preds_and_labels_df = spark.createDataFrame(preds_and_labels, ["predictio

rmse_val_LR0 = calc_RMSE(preds_and_labels_df)

print('Validation RMSE:\n\tBaseline = {0:.3f}\n\tLR0 = {1:.3f}'.format(rm
```

```
Validation RMSE:
        Baseline = 20.759
        LR0 = 18.371
```

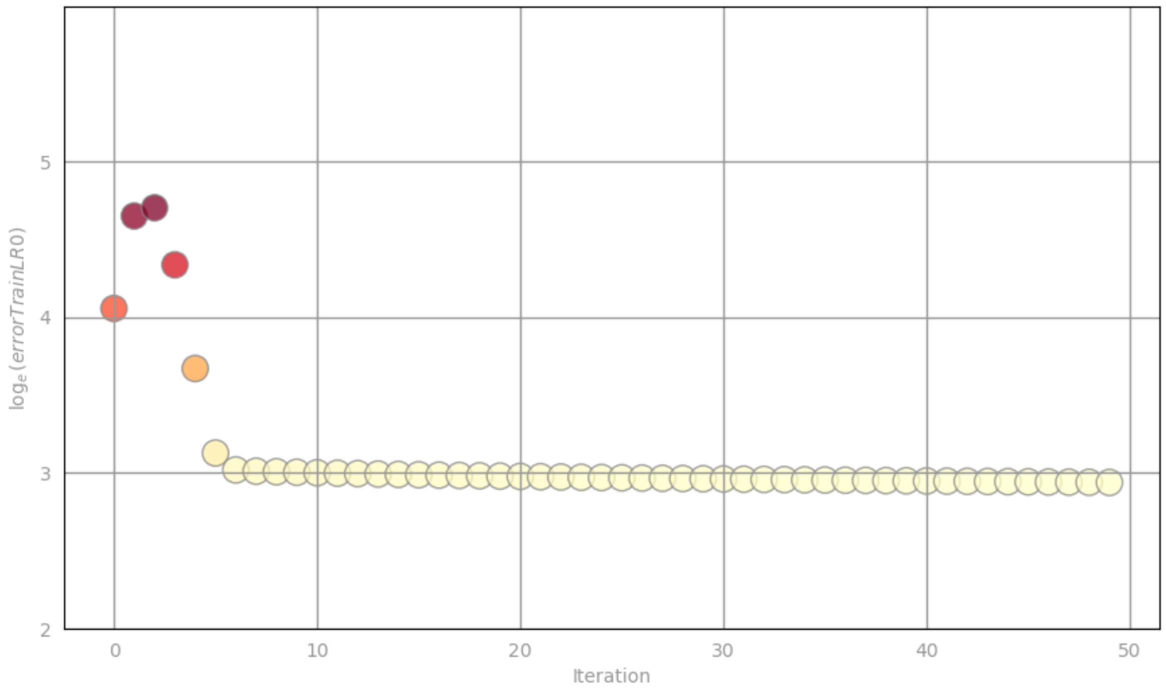
In [71]:
```
# TEST Train the model (3d)
expected_output = [22.63418505, 20.64417482, -0.39044365, 8.37584137, 5.8
                   3.6353787, 9.87147047, 5.80887121, 11.0129383, 3.48020
testmti850.Test.assertTrue(np.allclose(weights_LR0, expected_output), 'in
```

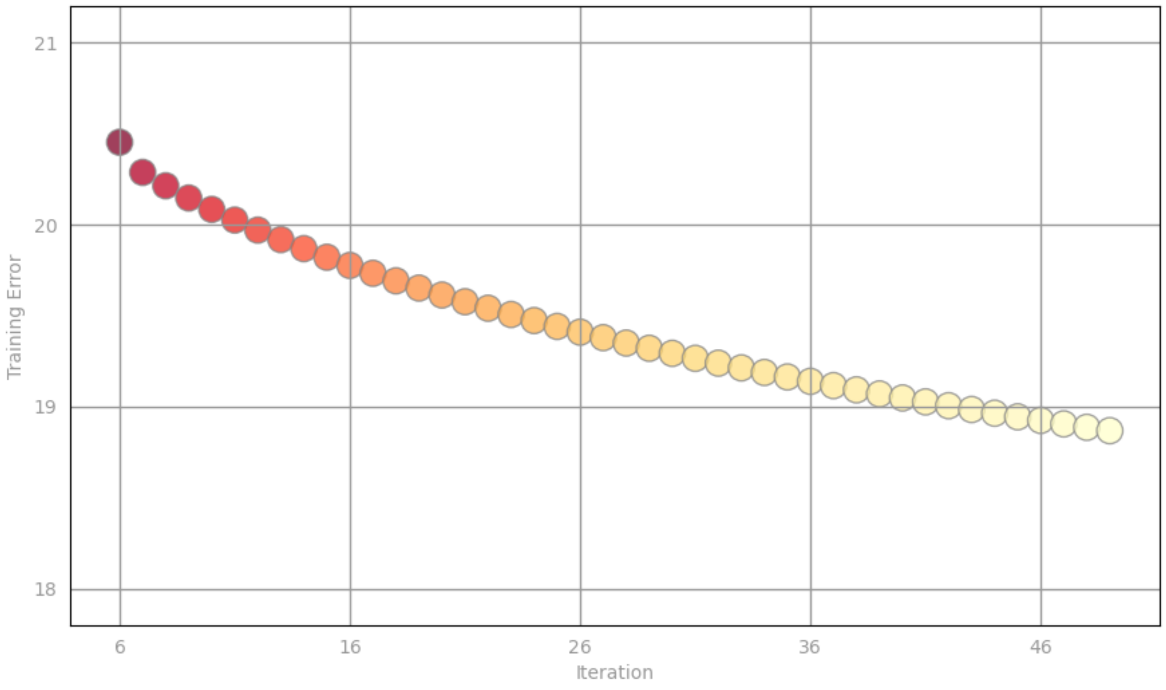
```
1 test passed.
```

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 [72]:
```
norm = Normalize()

clrs = cmap(np.asarray(norm(np.log(error_train_LR0))))[:,0:3]

fig, ax = prepareSubplot(np.arange(0, 60, 10), np.arange(2, 6, 1))
ax.set_ylim(2, 6)
ax.scatter(range(0, num_iters), np.log(error_train_LR0), s=14**2, c=clrs,
ax.set_xlabel('Iteration')
ax.set_ylabel(r'$\log_e(errorTrainLR0)$')
plt.show((fig))

norm = Normalize()

clrs = cmap(np.asarray(norm(error_train_LR0[6:])))[:,0:3]

fig, ax = prepareSubplot(np.arange(0, 60, 10), np.arange(17, 22, 1))
ax.set_ylim(17.8, 21.2)
ax.scatter(range(0, num_iters - 6), error_train_LR0[6:], s=14**2, c=clrs,
ax.set_xticklabels(map(str, range(6, 66, 10)))
ax.set_xlabel('Iteration')
ax.set_ylabel(r'Training Error')
plt.show()
```





Part 4: Train using SparkML and perform grid search

(4a) `LinearRegression`

We're already doing better than the baseline model, but let's see if we can do better by adding an intercept, using regularization, and (based on the previous visualization) training for more iterations. SparkML's LinearRegression essentially implements the same algorithm that we implemented in Part (3b), albeit more efficiently and with various additional functionality, such as including an intercept in the model and allowing L1, L2, or elastic net regularization. Elastic net regularization is a linear combination of L1 and L2 regularization. For $\alpha = 0$, the penalty is an L2 penalty. For $\alpha = 1$, it is an L1 penalty.

First use LinearRegression to train a model with elastic net regularization and an intercept. This method returns a LinearRegressionModel. Next, use the model's coefficients (weights) and intercept attributes to print out the model's parameters.

Important! Spark ML (`pyspark.ml`) expects the new vector type `pyspark.ml.linalg.VectorUDT` , but our DataFrame's features column is still using the **old MLlib** type: `pyspark.mllib.linalg.VectorUDT` . So, you need to convert your *parsed_train_data_df* dataframe to be fully compatible with the new Spark ML API before training our linear regression model.

Important! The same conversion will be required in (4c) for the validation set, and also for the training and test sets in (5b), (5c), and (5d) after adding interactions between features

In [73]:
```
from pyspark.ml.regression import LinearRegression
from pyspark.mllib.util import MLUtils
# Values to use when training the linear regression model

num_iters = 500  # iterations
reg = 1e-1  # regParam
alpha = .2  # elasticNetParam
use_intercept = True  # intercept
```

In [75]:
```
# TODO: Replace <FILL IN> with appropriate code

# Convert all vector columns to ML format
parsed_train_data_df = MLUtils.convertVectorColumnsToML(parsed_train_data

lin_reg =LinearRegression(maxIter=num_iters, regParam=reg, elasticNetPara

first_model = lin_reg.fit(parsed_train_data_df)

# coeffsLR1 stores the model coefficients; interceptLR1 stores the model

coeffs_LR1 = first_model.coefficients

intercept_LR1 = first_model.intercept

print(coeffs_LR1, intercept_LR1)
```

```
[23.1982638028573,26.98504523130359,-67.73180097557051,52.581253053537,-1
2.602263525137092,-48.387783459855676,36.41656162742634,-23.60729239535593
4,0.0,-3.010859551153598,-10.606012321267242,-14.977614644741953] 66.92674
87588785
```

In [76]:
```
# TEST LinearRegression (4a)
expected_intercept = 66.92674875888055
expected_weights = [23.198263802857262,26.985045231303086, -67.7318009755
                    -12.602263525139243, -48.387783459857786, 36.416561627
                     0.0,-3.0108595511539082,-10.606012321268773,-14.97761
testmti850.Test.assertTrue(np.allclose(intercept_LR1, expected_intercept)
testmti850.Test.assertTrue(np.allclose(coeffs_LR1, expected_weights), 'in
1 test passed.
1 test passed.
```

(4b) Transform

Now use the LinearRegressionModel.transform() method to make predictions on the `parsed_train_data_df` .

In [77]:
```
# TODO: Replace <FILL IN> with appropriate code

sample_prediction = first_model.transform(parsed_train_data_df)

sample_prediction.show()
```

```
+-----+--------------------+------------------+
|label|            features|        prediction|
+-----+--------------------+------------------+
|  0.0|[0.581044957651,0...|21.084909469789203|
|  0.0|[0.63944087626,0....| 63.08554683101235|
|  0.0|[0.665425875415,0...| 59.18426472167812|
|  0.0|[0.730673247594,0...| 67.41750658482327|
|  0.0|[0.791114423812,0...| 74.75083576946234|
|  2.0|[0.56977237281,0....| 14.55174964241526|
|  2.0|[0.595980071799,0...|13.269272145348076|
|  2.0|[0.615847938199,0...| 38.34650636878545|
|  3.0|[0.424574534176,0...|-3.291412687469176|
|  3.0|[0.444046053737,0...| 7.023831144339326|
|  3.0|[0.483297880403,0...| 12.67398590639872|
|  3.0|[0.54561164828,0....|12.865440995754824|
|  3.0|[0.552845959163,0...| 6.905799681612088|
|  3.0|[0.601494345313,0...|10.475204231854775|
|  4.0|[0.187672926543,0...| 7.790586512761337|
|  4.0|[0.359595235695,0...| 18.18359651791863|
|  4.0|[0.400744425698,0...|16.054230711571684|
|  4.0|[0.404665218137,0...| 9.184241129714351|
|  4.0|[0.421594084425,0...| 30.43956376158438|
|  4.0|[0.423538293212,0...|20.915148173273508|
+-----+--------------------+------------------+
only showing top 20 rows
```

In [78]:
```
# TEST Predict (4b)
testmti850.Test.assertTrue(np.allclose(sample_prediction.first().predicti
                'incorrect value for sample_prediction')
```

```
1 test passed.
```

(4c) Evaluate RMSE

Next evaluate the accuracy of this model on the validation set. Use the `transform()` method to create predictions, and then use the `calc_RMSE()` function from Part (2b).

In [81]:
```
# TODO: Replace <FILL IN> with appropriate code

parsed_val_data_df = MLUtils.convertVectorColumnsToML(parsed_val_data_df)
val_pred_df = first_model.transform(val_pred_df)

rmse_val_LR1 = calc_RMSE(val_pred_df)
```

```
print ('Validation RMSE:\n\tBaseline = {0:.3f} \n\tLR0 = {1:.3f}\n\tLR1 =
       .format(rmse_val_base, rmse_val_LR0, rmse_val_LR1))
```

```
25/11/28 03:28:22 WARN MLUtils: Vector column conversion has serialization
overhead. Please migrate your datasets and workflows to use the spark.ml p
ackage.
Validation RMSE:
        Baseline = 20.759
        LR0 = 18.371
        LR1 = 16.1152
```

In [82]:
```
# TEST Evaluate RMSE (4c)
testmti850.Test.assertTrue(np.allclose(rmse_val_LR1, 16.115204285984859),
```

```
1 test passed.
```

(4d) Grid search

We're already outperforming the baseline on the validation set by almost 2 years on average, but let's see if we can do better. Perform grid search to find a good regularization parameter. Try `regParam` values `1e-10` , `1e-5` , and `1.0` .

In [83]:
```
# TODO: Replace <FILL IN> with appropriate code
best_RMSE = rmse_val_LR1
best_reg_param = reg
best_model = first_model

num_iters = 500  # iterations
alpha = .2  # elasticNetParam
use_intercept = True  # intercept

regParams = [1e-10, 1e-5, 1]

for reg in regParams:
    lin_reg = LinearRegression(maxIter=num_iters, regParam=reg, elasticNe
    model = lin_reg.fit(parsed_train_data_df)
    val_pred_df = model.transform(parsed_val_data_df)

    rmse_val_grid = calc_RMSE(val_pred_df)
    print(rmse_val_grid)

    if rmse_val_grid < best_RMSE:
        best_RMSE = rmse_val_grid
        best_reg_param = reg
        best_model = model

rmse_val_LR_grid = best_RMSE
```

```
16.122518057355745
16.12251713674809
16.097420376208206
```

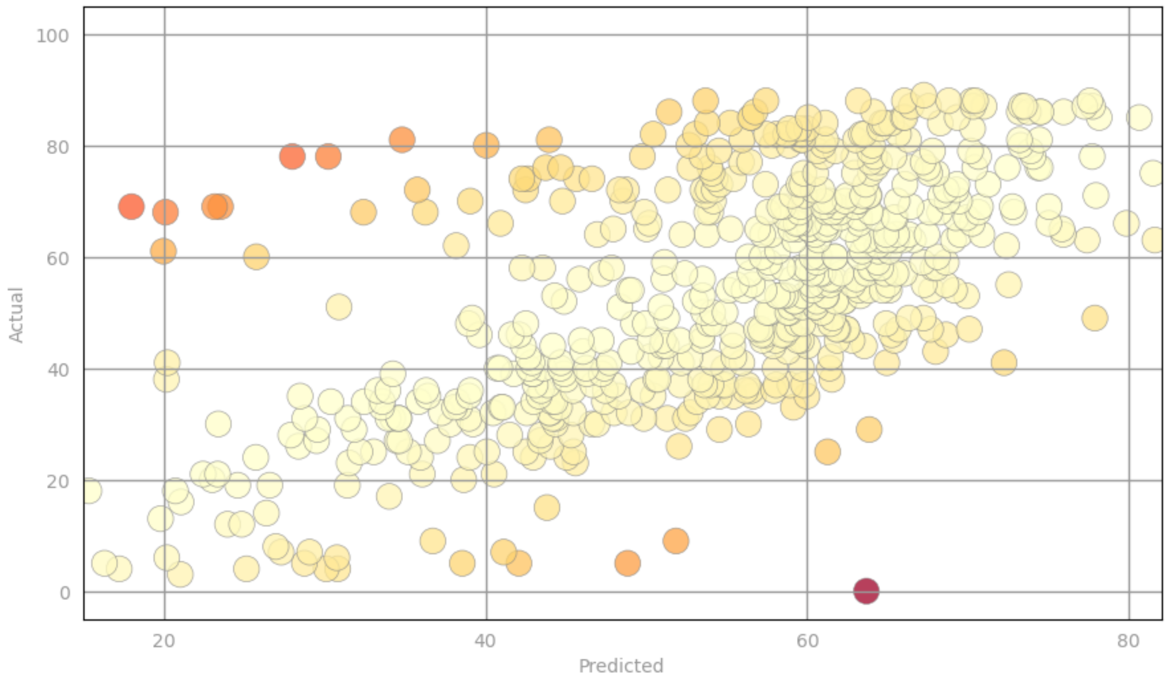
In [84]:
```
# TEST Grid search (4d)
testmti850.Test.assertTrue(np.allclose(16.097420376208206, rmse_val_LR_gr
```

```
1 test passed.
```

Visualization 5: Best model's predictions

Next, we create a visualization similar to 'Visualization 3: Predicted vs. actual' from Part 2 using 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 [85]:
```
parsed_val_df = best_model.transform(parsed_val_data_df)
predictions = np.asarray(parsed_val_df
                         .select('prediction')
                         .collect())
actual = np.asarray(parsed_val_df
                      .select('label')
                      .collect())
error = np.asarray(parsed_val_df
                     .rdd
                     .map(lambda lp: squared_error(lp.label, lp.predictio
                     .collect())

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

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

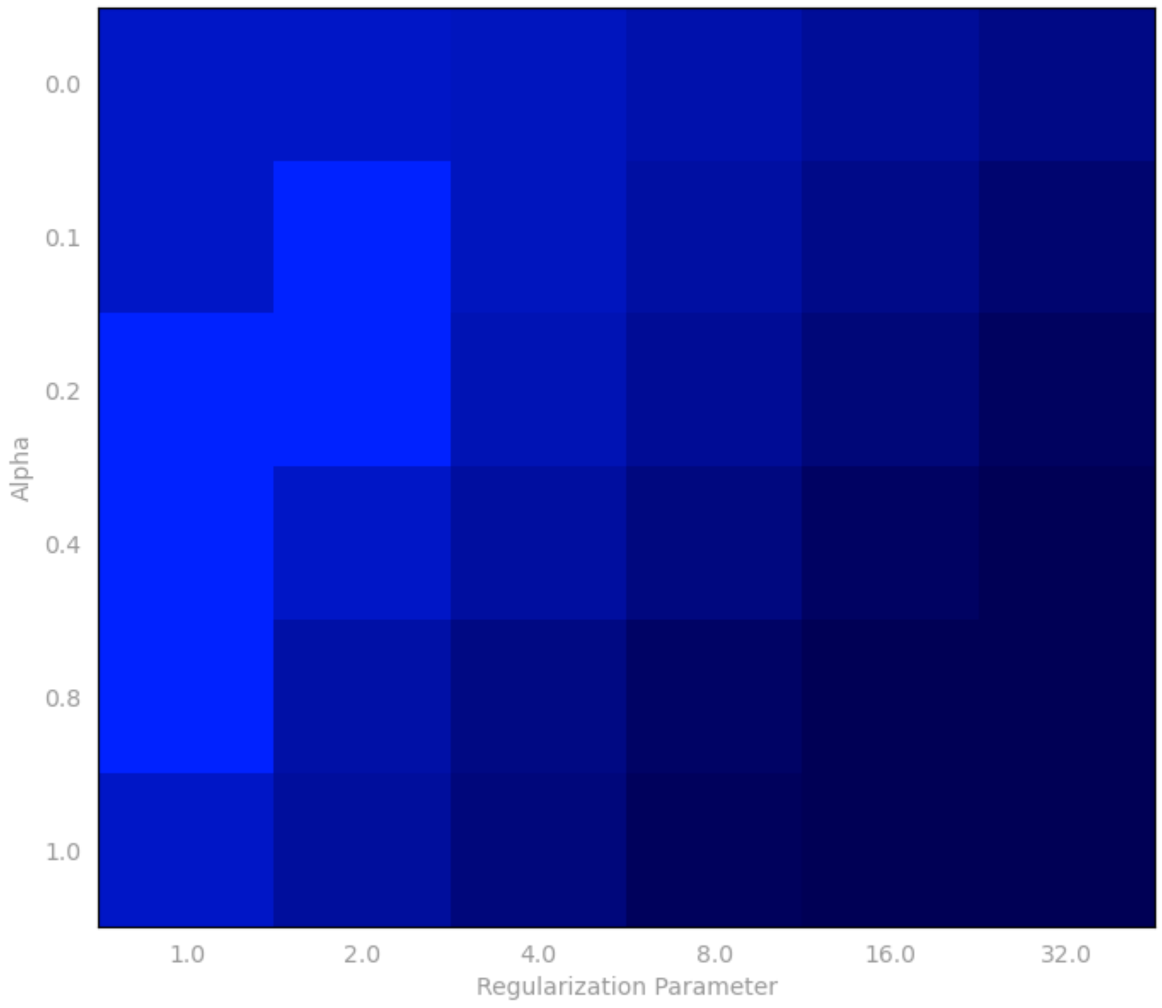


Visualization 6: Hyperparameter heat map

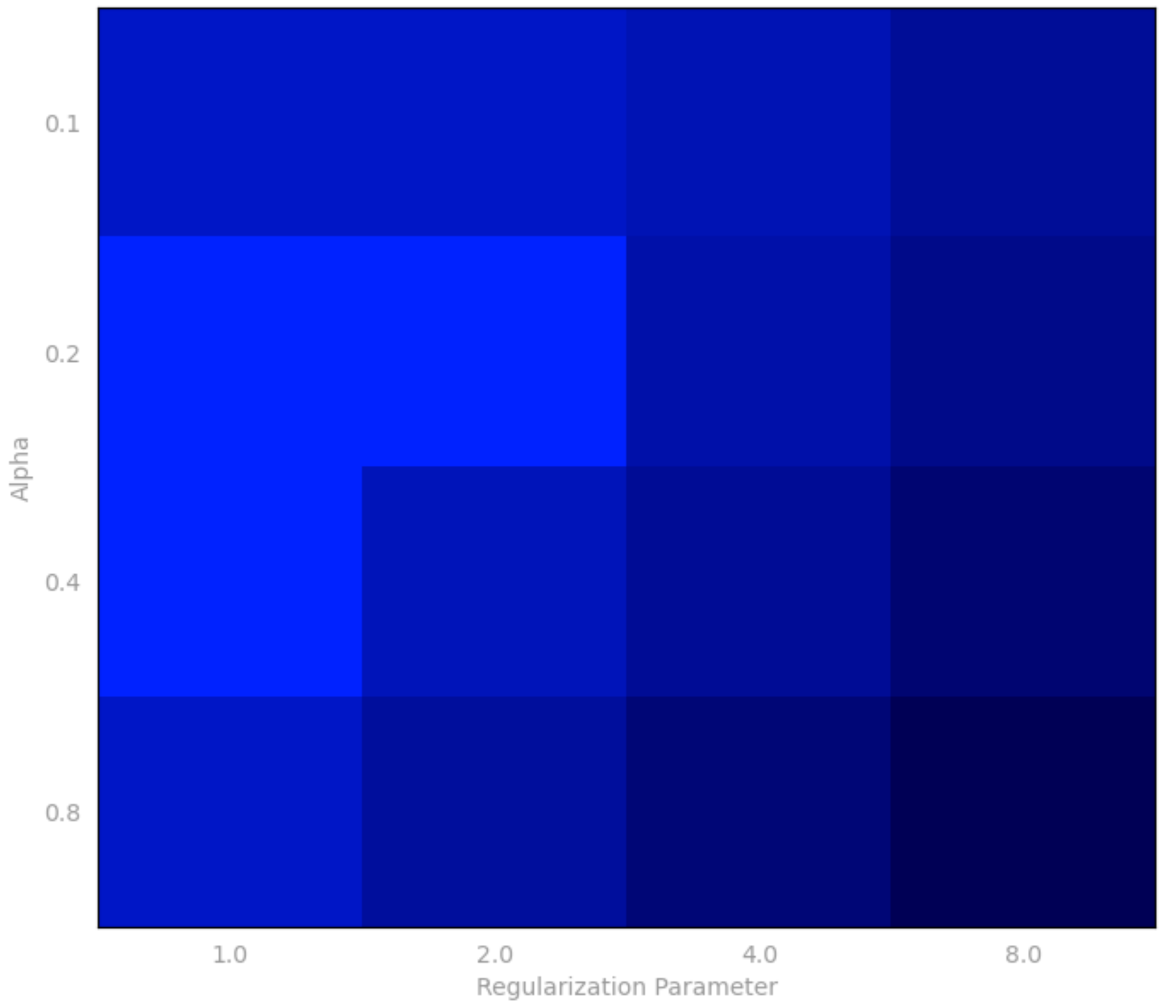
Next, we perform a visualization of hyperparameter search using a larger set of hyperparameters (with precomputed results). Specifically, we create a heat map where the brighter colors correspond to lower RMSE values. The first plot has a large area with brighter colors. In order to differentiate within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

In [86]:

```
from matplotlib.colors import LinearSegmentedColormap

# Saved parameters and results, to save the time required to run 36 model
num_iters = 500
reg_params = [1.0, 2.0, 4.0, 8.0, 16.0, 32.0]
alpha_params = [0.0, .1, .2, .4, .8, 1.0]
rmse_val = np.array([[ 15.317156766552452, 15.327211561989827, 15.3571529
                       16.36423857334287, 15.315019185101972, 15.30594921
                       16.231992712117222, 17.700179790697746, 15.3052663
                       15.824676190630191, 17.045905140628836, 19.3655583
                       15.620051033979871, 16.631757941340428, 18.9487868
                       15.522394576046239, 16.414106221093316, 18.6559787
                       15.33442896030322, 15.680134490745722, 16.86502909
                       20.91796910560631 ]])

num_rows, num_cols = len(alpha_params), len(reg_params)
rmse_val = np.array(rmse_val)
rmse_val.shape = (num_rows, num_cols)

fig, ax = prepareSubplot(np.arange(0, num_cols, 1), np.arange(0, num_rows
                       gridWidth=0.)
ax.set_xticklabels(reg_params), ax.set_yticklabels(alpha_params)
ax.set_xlabel('Regularization Parameter')
ax.set_ylabel('Alpha')

colors = LinearSegmentedColormap.from_list('blue', ['#0022ff', '#000055']
image = ax.imshow(rmse_val,interpolation='nearest', aspect='auto', cmap=c
plt.show()
```



In [88]:
```
# Zoom into the top left
alpha_params_zoom, reg_params_zoom = alpha_params[1:5], reg_params[:4]
rmse_val_zoom = rmse_val[1:5, :4]

num_rows, num_cols = len(alpha_params_zoom), len(reg_params_zoom)

fig, ax = prepareSubplot(np.arange(0, num_cols, 1), np.arange(0, num_rows
                         gridWidth=0.)
ax.set_xticklabels(reg_params_zoom), ax.set_yticklabels(alpha_params_zoom
ax.set_xlabel('Regularization Parameter'), ax.set_ylabel('Alpha')

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



Part 5: Add interactions between features

(5a) Add 2-way interactions

So far, we've used the features as they were provided. Now, we will add features that capture the two-way interactions between our existing features. Write a function `two_way_interactions` that takes in a `LabeledPoint` and generates a new `LabeledPoint` that contains the old features and the two-way interactions between them.

> Note:
>
> - A dataset with three features would have nine (3^2) two-way interactions.
> - You might want to use itertools.product to generate tuples for each of the possible 2-way interactions.
> - Remember that you can combine two `DenseVector` or `ndarray` objects using np.hstack.

In [94]:
```
# TODO: Replace <FILL IN> with appropriate code
import itertools

def two_way_interactions(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,
        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 a
            should include the features from `lp` followed by the two-way
    """
    features = lp.features
    interaction_features = [x * y for x, y in itertools.product(features,
    new_features = np.hstack((features, interaction_features))

    return LabeledPoint(lp.label, new_features)


#print(two_way_interactions((0.0, Vectors.dense([2, 3]))))


# Transform the existing train, validation, and test sets to include two-
# Remember to convert them back to DataFrames at the end.

train_data_interact_df = parsed_train_data_df.rdd.map(two_way_interaction

val_data_interact_df = parsed_val_data_df.rdd.map(two_way_interactions).t

test_data_interact_df = parsed_test_data_df.rdd.map(two_way_interactions)
```

In [95]:
```
# TEST Add two-way interactions (5a)
two_way_example = two_way_interactions((0.0, Vectors.dense([2, 3])))
testmti850.Test.assertTrue(np.allclose(sorted(two_way_example[1]),
                            sorted([2.0, 3.0, 4.0, 6.0, 6.0, 9.0])),
                'incorrect features generatedBy two_way_interactions')
two_way_point = two_way_interactions((1.0, Vectors.dense([1, 2, 3])))
testmti850.Test.assertTrue(np.allclose(sorted(two_way_point[1]),
                            sorted([1.0,2.0,3.0,1.0,2.0,3.0,2.0,4.0,6.0,3
                'incorrect features generated by twoWayInteractions')
testmti850.Test.assertEquals(two_way_point[0], 1.0, 'incorrect label gene
testmti850.Test.assertTrue(np.allclose(sum(train_data_interact_df.first()
                'incorrect features in train_data_interact_df')

testmti850.Test.assertTrue(np.allclose(sum(val_data_interact_df.first().f
                'incorrect features in val_data_interact_df')

testmti850.Test.assertTrue(np.allclose(sum(test_data_interact_df.first().
                'incorrect features in test_data_interact_df')
```

```
---------------------------------------------------------------------------
AttributeError                            Traceback (most recent call las
t)
Cell In[95], line 2
      1 # TEST Add two-way interactions (5a)
----> 2 two_way_example = two_way_interactions((0.0, Vectors.dense([2,
3])))
      3 testmti850.Test.assertTrue(np.allclose(sorted(two_way_example[1]),
      4                                  sorted([2.0, 3.0, 4.0, 6.0, 6.0,
9.0])),
      5                     'incorrect features generatedBy two_way_interactio
ns')
      6 two_way_point = two_way_interactions((1.0, Vectors.dense([1, 2,
3])))

Cell In[94], line 18, in two_way_interactions(lp)
      4 def two_way_interactions(lp):
      5     """Creates a new `LabeledPoint` that includes two-way interact
ions.
      6 
      7     Note:
   (...)
     16             should include the features from `lp` followed by the
two-way interaction features.
     17     """
---> 18     features = lp.features
     19     interaction_features = [x * y for x, y in itertools.product(fe
atures, features)]
     20     new_features = np.hstack((features, interaction_features))

AttributeError: 'tuple' object has no attribute 'features'
```

The function 'two_way_interactions' is supposed to take a LabeledPoint as Args, then I commented the test line with only a tuple and I didn't take into account the test code also with tuples.

(5b) Build interaction model

Now, let's build the new model. We've done this several times now. To implement this for the new features, we need to change a few variable names.

> Note:
>
> - Remember that we should build our model from the training data and evaluate it on the validation data.
> - You should re-run your hyperparameter search after changing features, as using the best hyperparameters from your prior model

> will not necessary lead to the best model.
> - For this exercise, we have already preset the hyperparameters to reasonable values.

In [98]:
```
# TODO: Replace <FILL IN> with appropriate code
num_iters = 500
reg = 1e-10
alpha = .2
use_intercept = True

train_data_interact_df = MLUtils.convertVectorColumnsToML(train_data_inte
val_data_interact_df = MLUtils.convertVectorColumnsToML(val_data_interact

lin_reg = LinearRegression(maxIter=num_iters, regParam=reg, elasticNetPar

model_interact = lin_reg.fit(train_data_interact_df)

preds_and_labels_interact_df = model_interact.transform(val_data_interact

rmse_val_interact = calc_RMSE(preds_and_labels_interact_df)

print ('Validation RMSE:\n\tBaseline = {0:.3f} \n\tLR0 = {1:.3f}\n\tLR1 =
                                                 rmse_val_LR_grid, rmse_
```

```
Exception ignored in: <_io.BufferedWriter name=5>
Traceback (most recent call last):
  File "/opt/spark/python/lib/pyspark.zip/pyspark/daemon.py", line 200, in
manager
BrokenPipeError: [Errno 32] Broken pipe

Validation RMSE:
        Baseline = 20.759
        LR0 = 18.371
        LR1 = 16.115
        LRGrid = 16.097
        LRInteract = 15.456
```

In [99]:
```
# TEST Build interaction model (5b)
testmti850.Test.assertTrue(np.allclose(rmse_val_interact, 15.456123194017
```

```
1 test passed.
```

(5c) Evaluate interaction model on test data

Our next step is to evaluate the new model on the test dataset. Note that we haven't used the test set to evaluate any of our models. Because of this, our evaluation provides us with an unbiased estimate for how our model will perform on new data. If we had changed our model based on viewing its performance on the test set, our estimate of RMSE would likely be overly optimistic.

We'll also print the RMSE for both the baseline model and our new model. With this information, we can see how much better our model performs than the baseline model.

In [100…
```
# TODO: Replace <FILL IN> with appropriate code

preds_and_labels_test_df = model_interact.transform(test_data_interact_df
rmse_test_interact = calc_RMSE(preds_and_labels_test_df)

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

```
Test RMSE:
        Baseline = 20.547
        LRInteract = 14.701
```

In [101…

```
# TEST Evaluate interaction model on test data (5c)
testmti850.Test.assertTrue(np.allclose(rmse_test_interact, 14.70091399650
                                'incorrect value for rmse_test_interact')
```

```
1 test passed.
```

(5d) Use a pipeline to create the interaction model

Our final step is to create the interaction model using a Pipeline. Note that Spark contains the PolynomialExpansion transformer which will automatically generate interactions for us. In this section, you'll need to generate the `PolynomialExpansion` transformer and set the stages for the `Pipeline` estimator. Make sure to use a degree of 2 for `PolynomialExpansion`, set the input column appropriately, and set the output column to "polyFeatures". The pipeline should contain two stages: the polynomial expansion and the linear regression.

In [103…

```
# TODO: Replace <FILL IN> with appropriate code
from pyspark.ml import Pipeline
from pyspark.ml.feature import PolynomialExpansion

num_iters = 500
reg = 1e-10
alpha = .2
use_intercept = True

polynomial_expansion = PolynomialExpansion(degree=2, inputCol='features',

linear_regression = LinearRegression(maxIter=num_iters, regParam=reg, ela
                                     fitIntercept=use_intercept, features

pipeline = Pipeline(stages=[polynomial_expansion, linear_regression])

pipeline_model = pipeline.fit(parsed_train_data_df)

parsed_test_data_df = MLUtils.convertVectorColumnsToML(parsed_test_data_d
predictions_df = pipeline_model.transform(parsed_test_data_df)

evaluator = RegressionEvaluator()

rmse_test_pipeline = evaluator.evaluate(predictions_df, {evaluator.metric

print('RMSE for test data set using pipelines: {0:.3f}'.format(rmse_test_
```

```
RMSE for test data set using pipelines: 14.675
```

```
25/11/28 04:34:13 WARN MLUtils: Vector column conversion has serialization
overhead. Please migrate your datasets and workflows to use the spark.ml p
ackage.
```

In [104…
```
# TEST Use a pipeline to create the interaction model (5d)
testmti850.Test.assertTrue(np.allclose(rmse_test_pipeline, 14.67520761192
                'incorrect value for rmse_test_pipeline')
```

```
1 test passed.
```

Notebook Ended