Quick Introduction to pandas

Learning Objectives:

- Gain an introduction to the `DataFrame` and `Series` data structures of the *pandas* library
- Access and manipulate data within a `DataFrame` and `Series`
- Import CSV data into a *pandas* `DataFrame`
- Reindex a `DataFrame` to shuffle data

pandas (http://pandas.pydata.org/) is a column-oriented data analysis API. It's a great tool for handling and analyzing input data, and many ML frameworks support *pandas* data structures as inputs. Although a comprehensive introduction to the *pandas* API would span many pages, the core concepts are fairly straightforward, and we'll present them below. For a more complete reference, the *pandas* docs site (http://pandas.pydata.org/pandas-docs/stable/index.html) contains extensive documentation and many tutorials.

Basic Concepts

The following line imports the *pandas* API and prints the API version:

In [1]:

```
import pandas as pd
%pylab inline
```

```
Populating the interactive namespace from numpy and matplotlib
```

The primary data structures in *pandas* are implemented as two classes:

- **`DataFrame`**, which you can imagine as a relational data table, with rows and named columns.
- **`Series`**, which is a single column. A `DataFrame` contains one or more `Series` and a name for each `Series`.

The data frame is a commonly used abstraction for data manipulation. Similar implementations exist in Spark (https://spark.apache.org/) and R (https://www.r-project.org/about.html).

One way to create a `Series` is to construct a `Series` object. For example:

In [2]:

```
pd.Series(['San Francisco', 'San Jose', 'Sacramento'])
```

Out[2]:

```
0    San Francisco
1         San Jose
2       Sacramento
dtype: object
```

`DataFrame` objects can be created by passing a `dict` mapping `string` column names to their respective `Series`. If the `Series` don't match in length, missing values are filled with special NA/NaN (http://pandas.pydata.org/pandas-docs/stable/missing_data.html) values. Example:

In [3]:

```
city_names = pd.Series(['San Francisco', 'San Jose', 'Sacramento'])
population = pd.Series([852469, 1015785, 485199])

pd.DataFrame({ 'City name': city_names, 'Population': population })
```

Out[3]:

	City name	Population
0	San Francisco	852469
1	San Jose	1015785
2	Sacramento	485199

But most of the time, you load an entire file into a `DataFrame`. The following example loads a file with California housing data. Run the following cell to load the data and create feature definitions:

In [4]:

```
california_housing_dataframe = pd.read_csv("data/california_housing_train.csv", sep=","
)
california_housing_dataframe.describe()
```

Out[4]:

	longitude	latitude	housing_median_age	total_rooms	total_bed
count	17000.000000	17000.000000	17000.000000	17000.000000	17000.00
mean	-119.562108	35.625225	28.589353	2643.664412	539.4108
std	2.005166	2.137340	12.586937	2179.947071	421.4994
min	-124.350000	32.540000	1.000000	2.000000	1.000000
25%	-121.790000	33.930000	18.000000	1462.000000	297.0000
50%	-118.490000	34.250000	29.000000	2127.000000	434.0000
75%	-118.000000	37.720000	37.000000	3151.250000	648.2500
max	-114.310000	41.950000	52.000000	37937.000000	6445.000

The example above used `DataFrame.describe` to show interesting statistics about a `DataFrame`. Another useful function is `DataFrame.head`, which displays the first few records of a `DataFrame`:

In [8]:

```
california_housing_dataframe.tail(5)
```

Out[8]:

	longitude	latitude	housing_median_age	total_rooms	total_bedrooms	popul
16995	-124.26	40.58	52.0	2217.0	394.0	907.0
16996	-124.27	40.69	36.0	2349.0	528.0	1194.0
16997	-124.30	41.84	17.0	2677.0	531.0	1244.0
16998	-124.30	41.80	19.0	2672.0	552.0	1298.0
16999	-124.35	40.54	52.0	1820.0	300.0	806.0

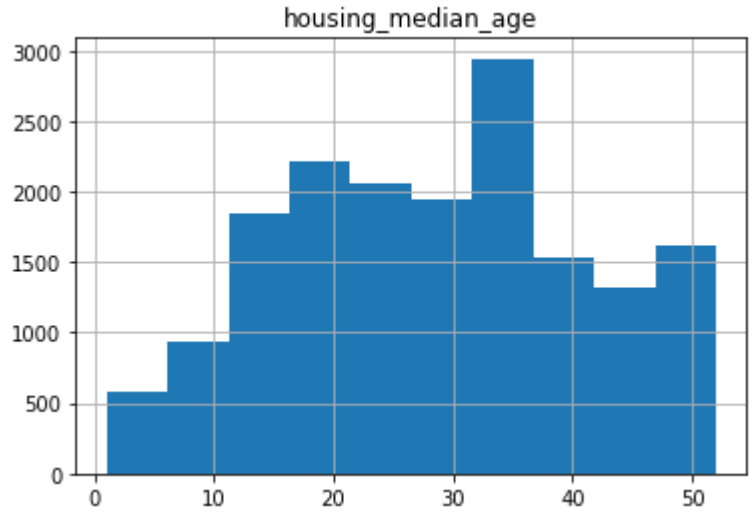
Another powerful feature of *pandas* is graphing. For example, `DataFrame.hist` lets you quickly study the distribution of values in a column:

In [13]:

```
#plt.title("something")
california_housing_dataframe.hist('housing_median_age')
```

Out[13]:

```
array([[<matplotlib.axes._subplots.AxesSubplot object at 0x000002A247A7308
0>]],
      dtype=object)
```



Accessing Data

You can access `DataFrame` data using familiar Python dict/list operations:

In [14]:

```
cities = pd.DataFrame({ 'City name': city_names, 'Population': population })
print(type(cities['City name']))
cities['City name']
```

```
<class 'pandas.core.series.Series'>
```

Out[14]:

```
0    San Francisco
1         San Jose
2       Sacramento
Name: City name, dtype: object
```

In [15]:

```
print(type(cities['City name'][1]))
cities['City name'][1]
```

```
<class 'str'>
```

Out[15]:

```
'San Jose'
```

In [16]:

```
print(type(cities[0:2]))
cities[0:2]
```

```
<class 'pandas.core.frame.DataFrame'>
```

Out[16]:

	City name	Population
0	San Francisco	852469
1	San Jose	1015785

In addition, *pandas* provides an extremely rich API for advanced indexing and selection (http://pandas.pydata.org/pandas-docs/stable/indexing.html) that is too extensive to be covered here.

Manipulating Data

You may apply Python's basic arithmetic operations to `Series`. For example:

In [17]:

```
population / 1000.
```

Out[17]:

```
0     852.469
1    1015.785
2     485.199
dtype: float64
```

NumPy (http://www.numpy.org/) is a popular toolkit for scientific computing. *pandas* `Series` can be used as arguments to most NumPy functions:

In [19]:

```
population
```

Out[19]:

```
0     852469
1    1015785
2     485199
dtype: int64
```

In [18]:

```
import numpy as np

np.log(population)
```

Out[18]:

```
0    13.655892
1    13.831172
2    13.092314
dtype: float64
```

For more complex single-column transformations, you can use `Series.apply`. Like the Python map function (https://docs.python.org/2/library/functions.html#map), `Series.apply` accepts as an argument a lambda function (https://docs.python.org/2/tutorial/controlflow.html#lambda-expressions), which is applied to each value.

The example below creates a new `Series` that indicates whether `population` is over one million:

In [20]:

```
population.apply(lambda val: val > 1000000)
```

Out[20]:

```
0    False
1     True
2    False
dtype: bool
```

Modifying `DataFrames` is also straightforward. For example, the following code adds two `Series` to an existing `DataFrame`:

In [21]:

```
cities['Area square miles'] = pd.Series([46.87, 176.53, 97.92])
cities['Population density'] = cities['Population'] / cities['Area square miles']
cities
```

Out[21]:

	City name	Population	Area square miles	Population density
0	San Francisco	852469	46.87	18187.945381
1	San Jose	1015785	176.53	5754.177760
2	Sacramento	485199	97.92	4955.055147

Exercise #1

Modify the `cities` table by adding a new boolean column that is True if and only if *both* of the following are True:

- The city is named after a saint.
- The city has an area greater than 50 square miles.

Note: Boolean `Series` are combined using the bitwise, rather than the traditional boolean, operators. For example, when performing *logical and*, use & instead of `and`.

Hint: "San" in Spanish means "saint."

In [ ]:

```
# Your code here
```

Solution

Click below for a solution.

In [22]:

```
cities['Is wide and has saint name'] = (cities['Area square miles'] > 50) & cities['Cit
y name'].apply(lambda name: name.startswith('San'))
cities
```

Out[22]:

	City name	Population	Area square miles	Population density	Is wide and has saint name
0	San Francisco	852469	46.87	18187.945381	False
1	San Jose	1015785	176.53	5754.177760	True
2	Sacramento	485199	97.92	4955.055147	False

Indexes

Both `Series` and `DataFrame` objects also define an `index` property that assigns an identifier value to each `Series` item or `DataFrame` row.

By default, at construction, *pandas* assigns index values that reflect the ordering of the source data. Once created, the index values are stable; that is, they do not change when data is reordered.

In [23]:

```
city_names.index
```

Out[23]:

```
RangeIndex(start=0, stop=3, step=1)
```

In [24]:

```
cities.index
```

Out[24]:

```
RangeIndex(start=0, stop=3, step=1)
```

Call `DataFrame.reindex` to manually reorder the rows. For example, the following has the same effect as sorting by city name:

In [25]:

```
cities.reindex([2, 0, 1])
```

Out[25]:

	City name	Population	Area square miles	Population density	Is wide and has saint name
2	Sacramento	485199	97.92	4955.055147	False
0	San Francisco	852469	46.87	18187.945381	False
1	San Jose	1015785	176.53	5754.177760	True

Reindexing is a great way to shuffle (randomize) a `DataFrame`. In the example below, we take the index, which is array-like, and pass it to NumPy's `random.permutation` function, which shuffles its values in place. Calling `reindex` with this shuffled array causes the `DataFrame` rows to be shuffled in the same way. Try running the following cell multiple times!

In [26]:

```
cities.reindex(np.random.permutation(cities.index))
```

Out[26]:

	City name	Population	Area square miles	Population density	Is wide and has saint name
0	San Francisco	852469	46.87	18187.945381	False
2	San Jose	1015785	176.53	5754.177760	True
1	Sacramento	485199	97.92	4955.055147	False

For more information, see the Index documentation (http://pandas.pydata.org/pandas-docs/stable/indexing.html#index-objects).

Exercise #2

The `reindex` method allows index values that are not in the original `DataFrame`'s index values. Try it and see what happens if you use such values! Why do you think this is allowed?

In [ ]:

```
# Your code here
```

Solution

Click below for the solution.

If your `reindex` input array includes values not in the original `DataFrame` index values, `reindex` will add new rows for these "missing" indices and populate all corresponding columns with `NaN` values:

In [ ]:

```
cities.reindex([0, 4, 5, 2])
```

This behavior is desirable because indexes are often strings pulled from the actual data (see the *pandas* reindex documentation (http://pandas.pydata.org/pandas-docs/stable/generated/pandas.DataFrame.reindex.html) for an example in which the index values are browser names).

In this case, allowing "missing" indices makes it easy to reindex using an external list, as you don't have to worry about sanitizing the input.

First Steps with TensorFlow

Learning Objectives:

- Learn fundamental TensorFlow concepts
- Use the `LinearRegressor` class in TensorFlow to predict median housing price, at the granularity of city blocks, based on one input feature
- Evaluate the accuracy of a model's predictions using Root Mean Squared Error (RMSE)
- Improve the accuracy of a model by tuning its hyperparameters

The data (https://developers.google.com/machine-learning/crash-course/california-housing-data-description) is based on 1990 census data from California.

Setup

In this first cell, we'll load the necessary libraries.

In [27]:

```
from __future__ import print_function

import math

from IPython import display
from matplotlib import cm
from matplotlib import gridspec
from matplotlib import pyplot as plt
import numpy as np
import pandas as pd
from sklearn import metrics
import tensorflow as tf
from tensorflow.python.data import Dataset

tf.logging.set_verbosity(tf.logging.ERROR)
pd.options.display.max_rows = 10
pd.options.display.float_format = '{:.1f}'.format
```

Next, we'll load our data set.

In [28]:

```
california_housing_dataframe = pd.read_csv("data/california_housing_train.csv", sep=","
)
```

We'll randomize the data, just to be sure not to get any pathological ordering effects that might harm the performance of Stochastic Gradient Descent. Additionally, we'll scale `median_house_value` to be in units of thousands, so it can be learned a little more easily with learning rates in a range that we usually use.

In [29]:

```
california_housing_dataframe = california_housing_dataframe.reindex(
    np.random.permutation(california_housing_dataframe.index))
california_housing_dataframe["median_house_value"] /= 1000.0
california_housing_dataframe
```

Out[29]:

	longitude	latitude	housing_median_age	total_rooms	total_bedrooms	popul
9049	-114.3	34.2	15.0	5612.0	1283.0	1015.0
10220	-114.5	34.4	19.0	7650.0	1901.0	1129.0
9505	-114.6	33.7	17.0	720.0	174.0	333.0
4362	-114.6	33.6	14.0	1501.0	337.0	515.0
5855	-114.6	33.6	20.0	1454.0	326.0	624.0
...	...	...	...	...	...	...
4160	-124.3	40.6	52.0	2217.0	394.0	907.0
574	-124.3	40.7	36.0	2349.0	528.0	1194.0
7660	-124.3	41.8	17.0	2677.0	531.0	1244.0
6970	-124.3	41.8	19.0	2672.0	552.0	1298.0
87	-124.3	40.5	52.0	1820.0	300.0	806.0

17000 rows × 9 columns

Examine the Data

It's a good idea to get to know your data a little bit before you work with it.

We'll print out a quick summary of a few useful statistics on each column: count of examples, mean, standard deviation, max, min, and various quantiles.

In [30]:

```
california_housing_dataframe.describe()
```

Out[30]:

	longitude	latitude	housing_median_age	total_rooms	total_bedrooms	popul
count	17000.0	17000.0	17000.0	17000.0	17000.0	17000
mean	-119.6	35.6	28.6	2643.7	539.4	1429.6
std	2.0	2.1	12.6	2179.9	421.5	1147.9
min	-124.3	32.5	1.0	2.0	1.0	3.0
25%	-121.8	33.9	18.0	1462.0	297.0	790.0
50%	-118.5	34.2	29.0	2127.0	434.0	1167.0
75%	-118.0	37.7	37.0	3151.2	648.2	1721.0
max	-114.3	42.0	52.0	37937.0	6445.0	35682

Build the First Model

In this exercise, we'll try to predict `median_house_value`, which will be our label (sometimes also called a target). We'll use `total_rooms` as our input feature.

NOTE: Our data is at the city block level, so this feature represents the total number of rooms in that block.

To train our model, we'll use the LinearRegressor (https://www.tensorflow.org/api_docs/python/tf/estimator/LinearRegressor) interface provided by the TensorFlow Estimator (https://www.tensorflow.org/get_started/estimator) API. This API takes care of a lot of the low-level model plumbing, and exposes convenient methods for performing model training, evaluation, and inference.

Step 1: Define Features and Configure Feature Columns

In order to import our training data into TensorFlow, we need to specify what type of data each feature contains. There are two main types of data we'll use in this and future exercises:

- **Categorical Data**: Data that is textual. In this exercise, our housing data set does not contain any categorical features, but examples you might see would be the home style, the words in a real-estate ad.
- **Numerical Data**: Data that is a number (integer or float) and that you want to treat as a number. As we will discuss more later sometimes you might want to treat numerical data (e.g., a postal code) as if it were categorical.

In TensorFlow, we indicate a feature's data type using a construct called a **feature column**. Feature columns store only a description of the feature data; they do not contain the feature data itself.

To start, we're going to use just one numeric input feature, `total_rooms`. The following code pulls the `total_rooms` data from our `california_housing_dataframe` and defines the feature column using `numeric_column`, which specifies its data is numeric:

In [31]:

```
# Define the input feature: total_rooms.
my_feature = california_housing_dataframe[["total_rooms"]]

# Configure a numeric feature column for total_rooms.
feature_columns = [tf.feature_column.numeric_column("total_rooms")]
```

NOTE: The shape of our `total_rooms` data is a one-dimensional array (a list of the total number of rooms for each block). This is the default shape for `numeric_column`, so we don't have to pass it as an argument.

Step 2: Define the Target

Next, we'll define our target, which is `median_house_value`. Again, we can pull it from our `california_housing_dataframe`:

In [32]:

```
# Define the label.
targets = california_housing_dataframe["median_house_value"]
```

Step 3: Configure the LinearRegressor

Next, we'll configure a linear regression model using LinearRegressor. We'll train this model using the `GradientDescentOptimizer`, which implements Mini-Batch Stochastic Gradient Descent (SGD). The `learning_rate` argument controls the size of the gradient step.

NOTE: To be safe, we also apply gradient clipping (https://developers.google.com/machine-learning/glossary/#gradient_clipping) to our optimizer via `clip_gradients_by_norm`. Gradient clipping ensures the magnitude of the gradients do not become too large during training, which can cause gradient descent to fail.

In [33]:

```
# Use gradient descent as the optimizer for training the model.
my_optimizer=tf.train.GradientDescentOptimizer(learning_rate=0.0000001)
my_optimizer = tf.contrib.estimator.clip_gradients_by_norm(my_optimizer, 5.0)

# Configure the linear regression model with our feature columns and optimizer.
# Set a learning rate of 0.0000001 for Gradient Descent.
linear_regressor = tf.estimator.LinearRegressor(
    feature_columns=feature_columns,
    optimizer=my_optimizer
)
```

Step 4: Define the Input Function

To import our California housing data into our `LinearRegressor`, we need to define an input function, which instructs TensorFlow how to preprocess the data, as well as how to batch, shuffle, and repeat it during model training.

First, we'll convert our *pandas* feature data into a dict of NumPy arrays. We can then use the TensorFlow Dataset API (https://www.tensorflow.org/programmers_guide/datasets) to construct a dataset object from our data, and then break our data into batches of `batch_size`, to be repeated for the specified number of epochs (num_epochs).

NOTE: When the default value of `num_epochs=None` is passed to `repeat()`, the input data will be repeated indefinitely.

Next, if `shuffle` is set to `True`, we'll shuffle the data so that it's passed to the model randomly during training. The `buffer_size` argument specifies the size of the dataset from which `shuffle` will randomly sample.

Finally, our input function constructs an iterator for the dataset and returns the next batch of data to the LinearRegressor.

In [34]:

```
def my_input_fn(features, targets, batch_size=1, shuffle=True, num_epochs=None):
    """Trains a linear regression model of one feature.

    Args:
      features: pandas DataFrame of features
      targets: pandas DataFrame of targets
      batch_size: Size of batches to be passed to the model
      shuffle: True or False. Whether to shuffle the data.
      num_epochs: Number of epochs for which data should be repeated. None = repeat ind
efinitely
    Returns:
      Tuple of (features, labels) for next data batch
    """

    # Convert pandas data into a dict of np arrays.
    features = {key:np.array(value) for key,value in dict(features).items()}


    # Construct a dataset, and configure batching/repeating.
    ds = Dataset.from_tensor_slices((features,targets)) # warning: 2GB limit
    ds = ds.batch(batch_size).repeat(num_epochs)

    # Shuffle the data, if specified.
    if shuffle:
      ds = ds.shuffle(buffer_size=10000)

    # Return the next batch of data.
    features, labels = ds.make_one_shot_iterator().get_next()
    return features, labels
```

NOTE: We'll continue to use this same input function in later exercises. For more detailed documentation of input functions and the `Dataset` API, see the TensorFlow Programmer's Guide (https://www.tensorflow.org/programmers_guide/datasets).

Step 5: Train the Model

We can now call `train()` on our `linear_regressor` to train the model. We'll wrap `my_input_fn` in a `lambda` so we can pass in `my_feature` and `target` as arguments (see this TensorFlow input function tutorial (https://www.tensorflow.org/get_started/input_fn#passing_input_fn_data_to_your_model) for more details), and to start, we'll train for 100 steps.

In [35]:

```
_ = linear_regressor.train(
    input_fn = lambda:my_input_fn(my_feature, targets),
    steps=100
)
```

Step 6: Evaluate the Model

Let's make predictions on that training data, to see how well our model fit it during training.

NOTE: Training error measures how well your model fits the training data, but it ***does not*** measure how well your model ***generalizes to new data***. In later exercises, you'll explore how to split your data to evaluate your model's ability to generalize.

In [36]:

```
# Create an input function for predictions.
# Note: Since we're making just one prediction for each example, we don't
# need to repeat or shuffle the data here.
prediction_input_fn =lambda: my_input_fn(my_feature, targets, num_epochs=1, shuffle=Fal
se)

# Call predict() on the linear_regressor to make predictions.
predictions = linear_regressor.predict(input_fn=prediction_input_fn)

# Format predictions as a NumPy array, so we can calculate error metrics.
predictions = np.array([item['predictions'][0] for item in predictions])

# Print Mean Squared Error and Root Mean Squared Error.
mean_squared_error = metrics.mean_squared_error(predictions, targets)
root_mean_squared_error = math.sqrt(mean_squared_error)
print("Mean Squared Error (on training data): %0.3f" % mean_squared_error)
print("Root Mean Squared Error (on training data): %0.3f" % root_mean_squared_error)
```

```
Mean Squared Error (on training data): 56367.026
Root Mean Squared Error (on training data): 237.417
```

Is this a good model? How would you judge how large this error is?

Mean Squared Error (MSE) can be hard to interpret, so we often look at Root Mean Squared Error (RMSE) instead. A nice property of RMSE is that it can be interpreted on the same scale as the original targets.

Let's compare the RMSE to the difference of the min and max of our targets:

In [37]:

```
min_house_value = california_housing_dataframe["median_house_value"].min()
max_house_value = california_housing_dataframe["median_house_value"].max()
min_max_difference = max_house_value - min_house_value

print("Min. Median House Value: %0.3f" % min_house_value)
print("Max. Median House Value: %0.3f" % max_house_value)
print("Difference between Min. and Max.: %0.3f" % min_max_difference)
print("Root Mean Squared Error: %0.3f" % root_mean_squared_error)
```

```
Min. Median House Value: 14.999
Max. Median House Value: 500.001
Difference between Min. and Max.: 485.002
Root Mean Squared Error: 237.417
```

Our error spans nearly half the range of the target values. Can we do better?

This is the question that nags at every model developer. Let's develop some basic strategies to reduce model error.

The first thing we can do is take a look at how well our predictions match our targets, in terms of overall summary statistics.

In [38]:

```
calibration_data = pd.DataFrame()
calibration_data["predictions"] = pd.Series(predictions)
calibration_data["targets"] = pd.Series(targets)
calibration_data.describe()
```

Out[38]:

	predictions	targets
count	17000.0	17000.0
mean	0.1	207.3
std	0.1	116.0
min	0.0	15.0
25%	0.1	119.4
50%	0.1	180.4
75%	0.2	265.0
max	1.9	500.0

Okay, maybe this information is helpful. How does the mean value compare to the model's RMSE? How about the various quantiles?

We can also visualize the data and the line we've learned. Recall that linear regression on a single feature can be drawn as a line mapping input *x* to output *y*.

First, we'll get a uniform random sample of the data so we can make a readable scatter plot.

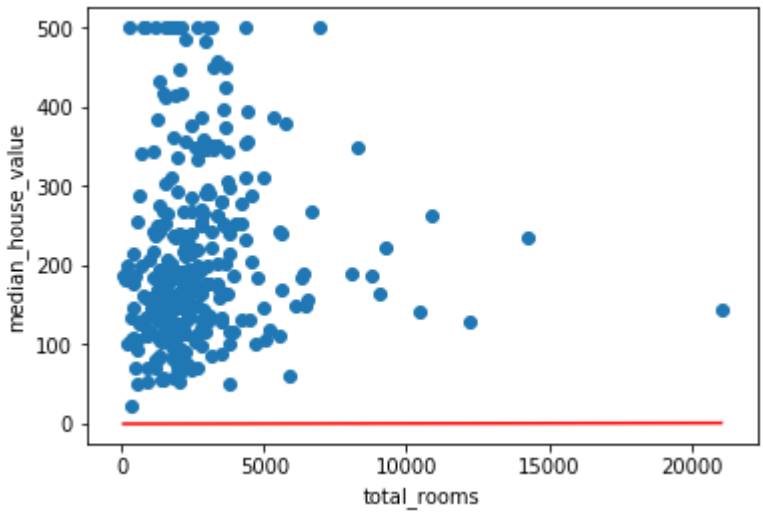
In [39]:

```
sample = california_housing_dataframe.sample(n=300)
```

Next, we'll plot the line we've learned, drawing from the model's bias term and feature weight, together with the scatter plot. The line will show up red.

In [40]:

```
# Get the min and max total_rooms values.
x_0 = sample["total_rooms"].min()
x_1 = sample["total_rooms"].max()

# Retrieve the final weight and bias generated during training.
weight = linear_regressor.get_variable_value('linear/linear_model/total_rooms/weights')
[0]
bias = linear_regressor.get_variable_value('linear/linear_model/bias_weights')

# Get the predicted median_house_values for the min and max total_rooms values.
y_0 = weight * x_0 + bias
y_1 = weight * x_1 + bias

# Plot our regression line from (x_0, y_0) to (x_1, y_1).
plt.plot([x_0, x_1], [y_0, y_1], c='r')

# Label the graph axes.
plt.ylabel("median_house_value")
plt.xlabel("total_rooms")

# Plot a scatter plot from our data sample.
plt.scatter(sample["total_rooms"], sample["median_house_value"])

# Display graph.
plt.show()
```



This initial line looks way off. See if you can look back at the summary stats and see the same information encoded there.

Together, these initial sanity checks suggest we may be able to find a much better line.

Tweak the Model Hyperparameters

For this exercise, we've put all the above code in a single function for convenience. You can call the function with different parameters to see the effect.

In this function, we'll proceed in 10 epochs so that we can observe the model improvement at each epoch.

For each epoch, we'll compute and graph training loss. This may help you judge when a model is converged, or if it needs more iterations.

We'll also plot the feature weight and bias term values learned by the model over time. This is another way to see how things converge.

In [41]:

```
def train_model(learning_rate, steps, batch_size, input_feature="total_rooms"):
  """Trains a linear regression model of one feature.

  Args:
    learning_rate: A `float`, the learning rate.
    steps: A non-zero `int`, the total number of training steps. A training step
      consists of a forward and backward pass using a single batch.
    batch_size: A non-zero `int`, the batch size.
    input_feature: A `string` specifying a column from `california_housing_dataframe`
      to use as input feature.
  """

  epochs = 10
  steps_per_epoch = steps / epochs

  my_feature = input_feature
  my_feature_data = california_housing_dataframe[[my_feature]]
  my_label = "median_house_value"
  targets = california_housing_dataframe[my_label]

  # Create feature columns.
  feature_columns = [tf.feature_column.numeric_column(my_feature)]

  # Create input functions.
  training_input_fn = lambda:my_input_fn(my_feature_data, targets, batch_size=batch_siz
e)
  prediction_input_fn = lambda: my_input_fn(my_feature_data, targets, num_epochs=1, shu
ffle=False)

  # Create a linear regressor object.
  my_optimizer = tf.train.GradientDescentOptimizer(learning_rate=learning_rate)
  my_optimizer = tf.contrib.estimator.clip_gradients_by_norm(my_optimizer, 5.0)
  linear_regressor = tf.estimator.LinearRegressor(
      feature_columns=feature_columns,
      optimizer=my_optimizer
  )

  # Set up to plot the state of our model's line each epoch.
  plt.figure(figsize=(15, 6))
  plt.subplot(1, 2, 1)
  plt.title("Learned Line by Epoch")
  plt.ylabel(my_label)
  plt.xlabel(my_feature)
  sample = california_housing_dataframe.sample(n=300)
  plt.scatter(sample[my_feature], sample[my_label])
  colors = [cm.coolwarm(x) for x in np.linspace(-1, 1, epochs)]

  # Train the model, but do so inside a loop so that we can periodically assess
  # loss metrics.
  print("Training model...")
  print("RMSE (on training data):")
  root_mean_squared_errors = []
  for epoch in range (0, epochs):
    # Train the model, starting from the prior state.
    linear_regressor.train(
        input_fn=training_input_fn,
        steps=steps_per_epoch
    )
    # Take a break and compute predictions.
```

```
    predictions = linear_regressor.predict(input_fn=prediction_input_fn)
    predictions = np.array([item['predictions'][0] for item in predictions])

    # Compute loss.
    root_mean_squared_error = math.sqrt(
        metrics.mean_squared_error(predictions, targets))
    # Occasionally print the current loss.
    print("  epoch %02d : %0.2f" % (epoch, root_mean_squared_error))
    # Add the loss metrics from this epoch to our list.
    root_mean_squared_errors.append(root_mean_squared_error)
    # Finally, track the weights and biases over time.
    # Apply some math to ensure that the data and line are plotted neatly.
    y_extents = np.array([0, sample[my_label].max()])

    weight = linear_regressor.get_variable_value('linear/linear_model/%s/weights' % inp
ut_feature)[0]
    bias = linear_regressor.get_variable_value('linear/linear_model/bias_weights')

    x_extents = (y_extents - bias) / weight
    x_extents = np.maximum(np.minimum(x_extents,
                                      sample[my_feature].max()),
                           sample[my_feature].min())
    y_extents = weight * x_extents + bias
    plt.plot(x_extents, y_extents, color=colors[epoch])
  print("Model training finished.")

  # Output a graph of loss metrics over epochs.
  plt.subplot(1, 2, 2)
  plt.ylabel('RMSE')
  plt.xlabel('Epochs')
  plt.title("Root Mean Squared Error vs. Epochs")
  plt.tight_layout()
  plt.plot(root_mean_squared_errors)

  # Output a table with calibration data.
  calibration_data = pd.DataFrame()
  calibration_data["predictions"] = pd.Series(predictions)
  calibration_data["targets"] = pd.Series(targets)
  display.display(calibration_data.describe())

  print("Final RMSE (on training data): %0.2f" % root_mean_squared_error)
```

Task 1: Achieve an RMSE of 180 or Below

Tweak the model hyperparameters to improve loss and better match the target distribution. If, after 5 minutes or so, you're having trouble beating a RMSE of 180, check the solution for a possible combination.

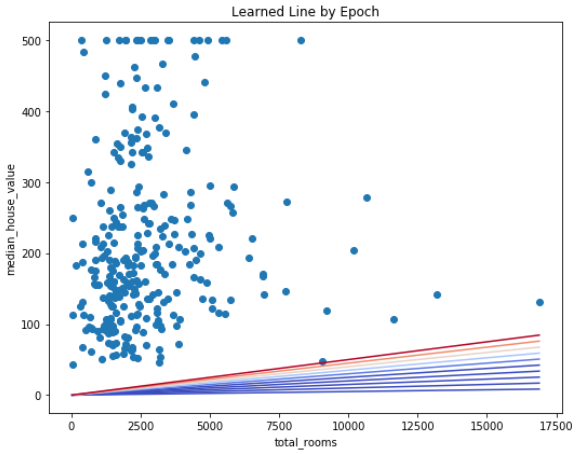
In [42]:

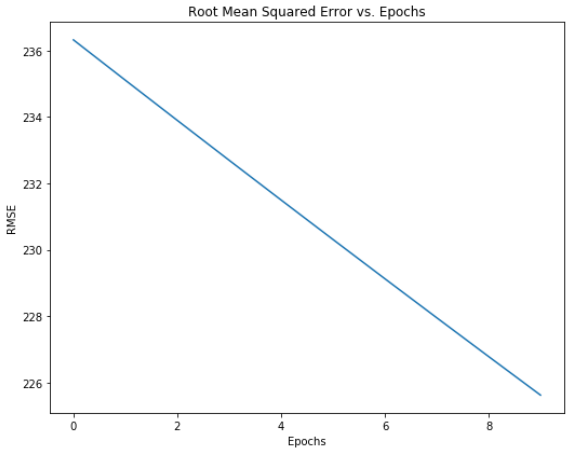
```
train_model(
    learning_rate=0.00001,
    steps=100,
    batch_size=1
)
```

```
Training model...
RMSE (on training data):
  epoch 00 : 236.32
  epoch 01 : 235.11
  epoch 02 : 233.90
  epoch 03 : 232.70
  epoch 04 : 231.50
  epoch 05 : 230.31
  epoch 06 : 229.13
  epoch 07 : 227.96
  epoch 08 : 226.79
  epoch 09 : 225.63
Model training finished.
```

	predictions	targets
count	17000.0	17000.0
mean	13.2	207.3
std	10.9	116.0
min	0.0	15.0
25%	7.3	119.4
50%	10.6	180.4
75%	15.8	265.0
max	189.7	500.0

```
Final RMSE (on training data): 225.63
```





Solution

Click below for one possible solution.

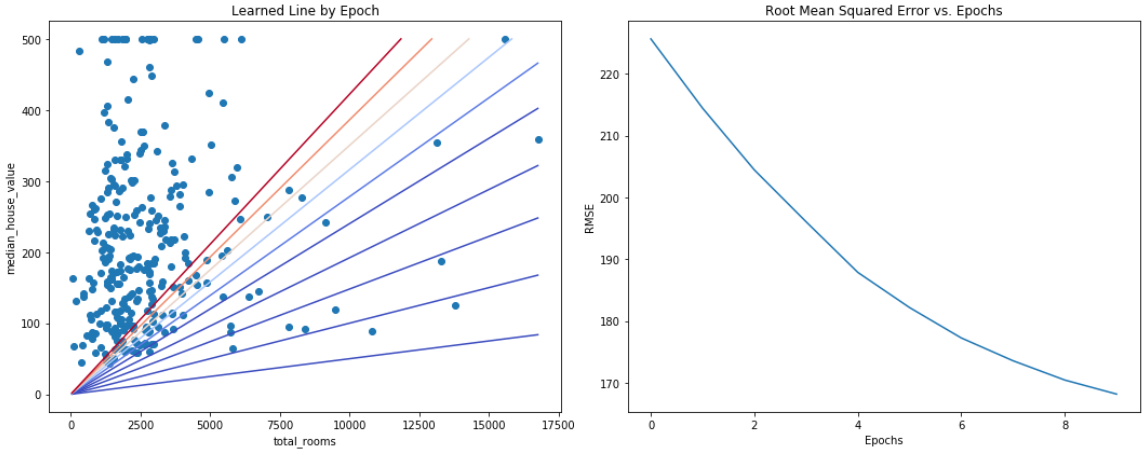
In [43]:

```
train_model(
    learning_rate=0.00002,
    steps=500,
    batch_size=5
)
```

```
Training model...
RMSE (on training data):
  epoch 00 : 225.63
  epoch 01 : 214.42
  epoch 02 : 204.44
  epoch 03 : 196.05
  epoch 04 : 187.86
  epoch 05 : 182.17
  epoch 06 : 177.26
  epoch 07 : 173.57
  epoch 08 : 170.45
  epoch 09 : 168.18
Model training finished.
```

	predictions	targets
count	17000.0	17000.0
mean	111.6	207.3
std	92.0	116.0
min	0.1	15.0
25%	61.7	119.4
50%	89.8	180.4
75%	133.0	265.0
max	1600.9	500.0

```
Final RMSE (on training data): 168.18
```



This is just one possible configuration; there may be other combinations of settings that also give good results. Note that in general, this exercise isn't about finding the *one best* setting, but to help build your intutions about how tweaking the model configuration affects prediction quality.

Is There a Standard Heuristic for Model Tuning?

This is a commonly asked question. The short answer is that the effects of different hyperparameters are data dependent. So there are no hard-and-fast rules; you'll need to test on your data.

That said, here are a few rules of thumb that may help guide you:

- Training error should steadily decrease, steeply at first, and should eventually plateau as training converges.
- If the training has not converged, try running it for longer.
- If the training error decreases too slowly, increasing the learning rate may help it decrease faster.
  - But sometimes the exact opposite may happen if the learning rate is too high.
- If the training error varies wildly, try decreasing the learning rate.
  - Lower learning rate plus larger number of steps or larger batch size is often a good combination.
- Very small batch sizes can also cause instability. First try larger values like 100 or 1000, and decrease until you see degradation.

Again, never go strictly by these rules of thumb, because the effects are data dependent. Always experiment and verify.

Task 2: Try a Different Feature

See if you can do any better by replacing the `total_rooms` feature with the `population` feature.

Don't take more than 5 minutes on this portion.

In [ ]:

```
# YOUR CODE HERE
```

Solution

Click below for one possible solution.

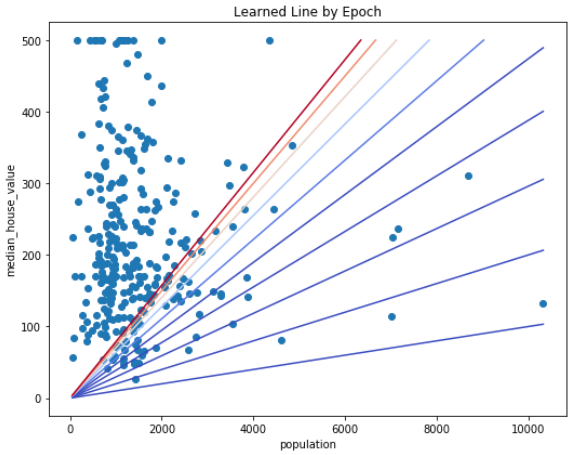
In [44]:

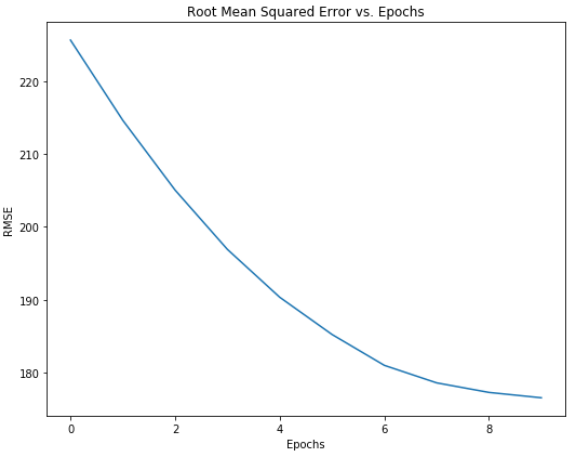
```
train_model(
    learning_rate=0.00002,
    steps=1000,
    batch_size=5,
    input_feature="population"
)
```

```
Training model...
RMSE (on training data):
  epoch 00 : 225.63
  epoch 01 : 214.62
  epoch 02 : 205.05
  epoch 03 : 196.92
  epoch 04 : 190.35
  epoch 05 : 185.25
  epoch 06 : 181.01
  epoch 07 : 178.61
  epoch 08 : 177.30
  epoch 09 : 176.57
Model training finished.
```

	predictions	targets
count	17000.0	17000.0
mean	112.7	207.3
std	90.5	116.0
min	0.2	15.0
25%	62.3	119.4
50%	92.0	180.4
75%	135.6	265.0
max	2811.8	500.0

```
Final RMSE (on training data): 176.57
```





Thank you

Shout out to our sponsor

visit their website (http://deepanalytics.ai/)

Like their Facebook page (https://www.facebook.com/DeepAnalyticsAI/)

Join our Facebook Group (https://www.facebook.com/groups/harareschoolofai/)