Lab_4_Using_a_Bucketized_Numerical_Feature

April 24, 2019


```
In [0]: # Licensed under the Apache License, Version 2.0 (the "License");
        # you may not use this file except in compliance with the License.
        # You may obtain a copy of the License at
        #
        # https://www.apache.org/licenses/LICENSE-2.0
        #
        # Unless required by applicable law or agreed to in writing, software
        # distributed under the License is distributed on an "AS IS" BASIS,
        # WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied.
        # See the License for the specific language governing permissions and
        # limitations under the License.
```

1 Lab 4: Using Bucketized Numerical Features

Learning Objectives: * Create bucketized numerical features in TF and use them to train a model * Use visualizations to understand the value of using bucketized features

1.0.1 Standard Set-up

We start by reading in the data from the Automobile Data Set. In this lab we are going to look at using `compression-ratio` to predict `city-mpg`.

```
In [0]: import fnmatch
        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 mpl_toolkits.mplot3d import Axes3D
        from sklearn import metrics
        import tensorflow as tf
        from tensorflow.contrib.learn.python.learn import learn_io, estimator
```

```
# This line increases the amount of logging when there is an error. You can
# remove it if you want less logging.
tf.logging.set_verbosity(tf.logging.ERROR)

# Set the output display to have two digits for decimal places, for display
# readability only and limit it to printing 15 rows.
pd.options.display.float_format = '{:.2f}'.format
pd.options.display.max_rows = 15

# Provide the names for the columns since the CSV file with the data does
# not have a header row.
cols = ['symboling', 'losses', 'make', 'fuel-type', 'aspiration', 'num-doors',
        'body-style', 'drive-wheels', 'engine-location', 'wheel-base',
        'length', 'width', 'height', 'weight', 'engine-type', 'num-cylinders',
        'engine-size', 'fuel-system', 'bore', 'stroke', 'compression-ratio',
        'horsepower', 'peak-rpm', 'city-mpg', 'highway-mpg', 'price']


# Load in the data from a CSV file that is comma seperated.
car_data = pd.read_csv('https://storage.googleapis.com/ml_universities/cars_dataset/ca
                       sep=',', names=cols, header=None, encoding='latin-1')

# We'll then randomize the order of the rows.
car_data = car_data.reindex(np.random.permutation(car_data.index))
```

1.0.2 Functions to Help Visualize Our Results

As in the past we define functions to generate a scatter plot, calibration plot, and learning curve.

```
In [0]: def make_scatter_plot(dataframe, input_feature, target,
                              slopes=[], biases=[], model_names=[]):
          """ Creates a scatter plot of input_feature vs target along with the models.

          Args:
            dataframe: the dataframe to visualize
            input_feature: the input feature to be used for the x-axis
            target: the target to be used for the y-axis
            slopes: list of model weights (slopes)
            bias: list of model biases (same length as slopes)
            model_names: list of model_names to use for legend (same length as slopes)
          """
          # Define some colors to use that go from blue towards red
          colors = [cm.coolwarm(x) for x in np.linspace(0, 1, len(slopes))]

          # Generate the scatter plot
          x = dataframe[input_feature]
          y = dataframe[target]
```

```
    plt.ylabel(target)
    plt.xlabel(input_feature)
    plt.scatter(x, y, color='black', label="")

    # Add the lines corresponding to the provided models
    for i in range (0, len(slopes)):
      y_0 = slopes[i] * x.min() + biases[i]
      y_1 = slopes[i] * x.max() + biases[i]
      plt.plot([x.min(), x.max()], [y_0, y_1],
               label=model_names[i], color=colors[i])
    if len(model_names) >= 1:
      plt.legend(bbox_to_anchor=(1.05, 1), loc=2, borderaxespad=0.)

  def make_calibration_plot(predictions, targets):
    """ Creates a calibration plot.

    Args:
      predictions: a list of values predicted by the model being visualized
      targets: a list of the target values being predicted that must be the
               same length as predictions.
    """
    calibration_data = pd.DataFrame()
    calibration_data["predictions"] = pd.Series(predictions)
    calibration_data["targets"] = pd.Series(targets)
    calibration_data.describe()
    min_val = calibration_data["predictions"].min()
    max_val = calibration_data["predictions"].max()
    plt.ylabel("target")
    plt.xlabel("prediction")
    plt.scatter(predictions, targets, color='black')
    plt.plot([min_val, max_val], [min_val, max_val])

  def plot_learning_curve(training_losses):
    """ Plot the learning curve.

    Args:
      training_loses: a list of losses to plot.
    """
    plt.ylabel('Loss')
    plt.xlabel('Training Steps')
    plt.plot(training_losses)
```

Let's begin by looking at a scatter plot to understand the relationship between `compression-ratio` and `city-mpg`.

```
In [0]: make_scatter_plot(car_data, "compression-ratio", "city-mpg")
```

Would you expect a linear model to make a good prediction?

1.0.3 Functions for defining the linear regression model and training it

We use the same functions as in the last lab to define the input function, feature columns, linear regression model, and train the model.

In [0]:
```
CATEGORICAL_COLUMNS = []
NUMERICAL_COLUMNS = ["city-mpg", "compression-ratio"]

def input_fn(dataframe):
  """Constructs a dictionary for the feature columns.

  Args:
    dataframe: The Pandas DataFrame to use for the input.
  Returns:
    The feature columns and the associated labels for the provided input.
  """
  # Creates a dictionary mapping from each numeric feature column name (k) to
  # the values of that column stored in a constant Tensor.
  numerical_cols = {k: tf.constant(dataframe[k].values)
                    for k in NUMERICAL_COLUMNS}
  # Creates a dictionary mapping from each categorical feature column name (k)
  # to the values of that column stored in a tf.SparseTensor.
  categorical_cols = {k: tf.SparseTensor(
      indices=[[i, 0] for i in range(dataframe[k].size)],
      values=dataframe[k].values,
      dense_shape=[dataframe[k].size, 1])
                      for k in CATEGORICAL_COLUMNS}
  # Merges the two dictionaries into one.
  feature_cols = dict(numerical_cols.items() + categorical_cols.items())
  # Converts the label column into a constant Tensor.
  label = tf.constant(dataframe[LABEL].values)
  # Returns the feature columns and the label.
  return feature_cols, label

def train_input_fn():
  """Sets up the input function using the training data.

  Returns:
     The feature columns to use for training and the associated labels.
  """
  return input_fn(training_examples)

def construct_feature_columns():
  """Construct TensorFlow feature columns.

  Returns:
    A set of feature columns.
  """
  feature_set = set([tf.contrib.layers.real_valued_column(feature)
```

```
                      for feature in NUMERICAL_FEATURES])
  return feature_set

def define_linear_regression_model(learning_rate):
  """ Defines a linear regression model of one feature to predict the target.

  Args:
    learning_rate: A `float`, the learning rate.

  Returns:
    A linear regressor created with the given parameters.
  """
  linear_regressor = tf.contrib.learn.LinearRegressor(
    feature_columns=construct_feature_columns(),
    optimizer=tf.train.GradientDescentOptimizer(learning_rate=learning_rate),
    gradient_clip_norm=5.0
  )
  return linear_regressor

def compute_loss(predictions, targets):
  """ Computes the loss (RMSE) for linear regression.

  Args:
    predictions: a list of values predicted by the model.
    targets: a list of the target values being predicted that must be the
             same length as predictions.

  Returns:
    The RMSE for the provided predictions and targets.
  """
  return math.sqrt(metrics.mean_squared_error(predictions, targets))

def train_model(linear_regressor, steps):
  """Trains a linear regression model.

  Args:
    linear_regressor: The regressor to train
    steps: A positive `int`, the total number of training steps.

  Returns:
    The trained regressor.
  """
  # In order to see how the model evolves as we train it, we divide the
  # steps into periods and show the model after each period.
  periods = 10
  steps_per_period = steps / periods

  # Train the model, but do so inside a loop so that we can periodically assess
```

```
# loss metrics.  We store the loss, slope (feature weight), bias, and a name
# for the model when there is a single feature (which then allows us
# to plot the model in a scatter plot).
print "Training model..."
training_losses = []
slopes = []
biases = []
model_names = []

for period in range (0, periods):
  # Call fit to train the regressor for steps_per_period steps
  linear_regressor.fit(input_fn=train_input_fn, steps=steps_per_period)

  # Use the predict method to compute the predictions from the current model
  predictions = np.array(list(linear_regressor.predict(
      input_fn=train_input_fn)))

  # Compute the loss between the predictions and the correct labels, append
  # the loss to the list of losses used to generate the learning curve after
  # training is complete and print the current loss.
  loss = compute_loss(predictions, training_examples[LABEL])
  training_losses.append(loss)
  print "  Loss after period %02d : %0.3f" % (period, loss)

  # When there is a single input feature, add slope, bias and model_name to
  # the lists to be used later to plot the model after each training period.
  if len(NUMERICAL_FEATURES) == 1 and len(CATEGORICAL_FEATURES) == 0:
    feature_weight = fnmatch.filter(linear_regressor.get_variable_names(),
                                    'linear/*/weight')
    slopes.append(linear_regressor.get_variable_value(
        feature_weight[0])[0])
    biases.append(linear_regressor.get_variable_value(
        'linear/bias_weight')[0])
    model_names.append("period_" + str(period))

# Now that training is done, print the final loss.
print "Final Loss (RMSE) on the training data: %0.3f" % loss

# Generate a figure with the learning curve on the left and either a scatter
# plot or calibration plot (when more than 2 input features) on the right.
plt.figure(figsize=(10, 5))
plt.subplot(1, 2, 1)
plt.title("Learning Curve (RMSE vs time)")
plot_learning_curve(training_losses)
plt.subplot(1, 2, 2)
plt.tight_layout(pad=1.1, w_pad=3.0, h_pad=3.0)

if len(NUMERICAL_FEATURES) > 1 or len(CATEGORICAL_FEATURES) != 0:
```

```
    plt.title("Calibration Plot")
    make_calibration_plot(predictions, training_examples[LABEL])

  else:
    plt.title("Learned Model by Period on Scatter Plot")
    make_scatter_plot(training_examples, NUMERICAL_FEATURES[0], LABEL,
                      slopes, biases, model_names)

  return linear_regressor
```

1.0.4 Generate the Training Examples

We generate the training examples by calling `prepare_features` on the `car_data` DataFrame.

```
In [0]: def prepare_features(dataframe):
          """Prepares the features for provided dataset.

          Args:
            dataframe: A Pandas DataFrame expected to contain data from the
              desired data set.
          Returns:
            A new DataFrame that contains the features to be used for the model.
          """
          processed_features = dataframe.copy()
          return processed_features

        # Generate the training examples.
        training_examples = prepare_features(car_data)
```

1.1 Task 1 - Train a Linear Regression Model (1 point)

Tune the hyperparameters to train the best linear regression model you can to predict `city-mpg` from `compression-ratio`. **What do you observe?**

```
In [0]: """
        Very briefly describe what you observe after training the model in this comment.
        (There is no need to run this code block but it also won't hurt if you do).




        """
```

```
In [0]: NUMERICAL_FEATURES = ["compression-ratio"]
        CATEGORICAL_FEATURES = []
        LABEL = "city-mpg"

        LEARNING_RATE = .25
```

```
STEPS = 50

linear_regressor = define_linear_regression_model(learning_rate = LEARNING_RATE)
linear_regressor = train_model(linear_regressor, steps=STEPS)
```

1.2 Task 2: Use a Bucketized Feature to Create a Better Model (2 points)

As you might have observed when looking at the scatter plot, there are two different behaviors happening: One with a low compression ratio, and another with a high compression ratio. So really we want to learn a separate offset (the bias) for these two regions.

A Bucketized Column for a numerical feature is designed exactly for this situation. Below we revise `construct_feature_columns` to introduce a bucketized feature column. Observe that since the bucketized features are binary (1 if in the bucket and 0 otherwise), adding the bucketized features allows the model to learn an independent bias for each bucket. If you also want to learn a slope for the model, you also still want to include the `compression_ratio` as a numerical feature.

Since the two bucketized features will have a value of 0 or 1, you might also want to do some linear scaling of compression ratio and then use 0.5 as your threshold. If you were not able to get a linear scaling function to work in Lab 3, come to the office hours to get help writing this function if you decide that you'd like to use it in this lab.

In the below code box we've provided the functions that you are likely to want to change. You'll need to make the needed changes in these functions. You are welcome to add/remove functions as it fits your needs.

```
In [0]: # Make any needed changes to prepare_features
        def prepare_features(dataframe):
          """Prepares the features for provided dataset.

          Args:
            dataframe: A Pandas DataFrame expected to contain data from the
              desired data set.
          Returns:
            A new DataFrame that contains the features to be used for the model.
          """
          processed_features = dataframe.copy()
          ## Add whatever processing you'd like to use here.
          return processed_features

        # Generate the training examples with your revised version of prepare_features
        training_examples = prepare_features(car_data)

        # Below, we modify construct_feature_columns to use compression_ratio, a
        # bucketized feature for compression_ratio with two buckets using the provided
        # threshold. Make sure that the threshold choice is appropriate for whatever
        # feature processing you choose to use.

        threshold = 0.5

        def construct_feature_columns():
```

```
    """Construct the TensorFlow feature columns.

    Returns:
      A set of feature columns
    """
    compression_ratio = tf.contrib.layers.real_valued_column("compression-ratio")
    bucketized_compression_ratio = tf.contrib.layers.bucketized_column(
        compression_ratio, [threshold])

    return set([compression_ratio, bucketized_compression_ratio])
```

1.2.1 Now Train a Model

Once you've completed the changes above you can train the model.

```
In [0]: # Putting an empty categorical feature to generate a calibration plot instead
        # of a scatter plot.
        CATEGORICAL_FEATURES = [""]
        LABEL = "city-mpg"

        LEARNING_RATE = 0.25
        STEPS = 50

        linear_regressor = define_linear_regression_model(learning_rate = LEARNING_RATE)
        linear_regressor = train_model(linear_regressor, steps=STEPS)
```

1.2.2 Look at the Model Weights

If you ever want to see the weights stored in a model you can use the `get_variable_names` method. Below is a code box that prints the weights trained in this model.

```
In [0]: w = linear_regressor.get_variable_value("linear/compression-ratio/weight")[0]
        w_low = linear_regressor.get_variable_value("linear/compression-ratio_bucketized/weight
        w_high = linear_regressor.get_variable_value("linear/compression-ratio_bucketized/weigh
        b = linear_regressor.get_variable_value("linear/bias_weight")
        print "weight for compression ratio:", w
        print "weight for compression ratio small bucket:", w_low
        print "weight for compression ratio large bucket:", w_high
        print "weight for model bias:", b
```

1.3 Task 3: Understand How to Make Predictions With This Model (1 point)

For the model you just trained there are 3 variables: * x the `compression_ratio` * x_{low} the value for the bucket corresponding to low compression ratio that is 1 if the compression ratio is less than the threshold and 0 otherwise. * x_{high} the value for the bucket corresponding to high compression ratio that is 1 if the compression ratio is above the threshold and 0 otherwise.

When training the model there are 4 weights learned

- w the weight for `compression-ratio`

- w_{low} the weight for the bucket corresponding to low compression ratio
- w_{high} weight for the bucket corresponding to high compression ratio
- b the bias

So the prediction made by the trained model for an example will be: $w * x + w_{low} * x_{low} + w_{high} * x_{high} + b$.

For this task, answer the following four questions. * A) What is the prediction of your model when the original compression ratio is 20? Remember to apply any feature normalization you did to this input value. * B) What is the prediction of your model when the original compression ratio is 5? * C) Derive a formula (simplified as much as you can) for the predicted `city-mpg` when the compression ratio x is less than the threshold. * D) Derive a formula (simplified as much as you can) for the predicted `city-mpg` when the compression ratio x is greater than the threshold.

```
In [0]: """
        Put your answers in the comment of this code block.  You need not worry about
        formatting the answers to C and D being neatly.  As long as we can understand
        your answer that is fine.

        Also feel free to add another code block or some code outside the comment area
        of this code block to perform any needed computations to help you out.

        Answer to A)

        Answer to B)

        Answer to C)

        Answer to D)

        """
```

1.4 Task 4: Draw a Scatter Plot Showing the Predictions of Your Model (1 point)

Unlike the scatter plots we've seen before, with the bucketized feature you will have a different linear model in the region for each bucket. Here you are going to draw the model you trained with two buckets for `compression-ratio`. You can just put direct code to do this. There is no need to modify `make_scatter_plot` but rather just use that code to guide you here where you are going to draw one line segment from x_min to your threshold for the bucketized feature and a second line segment going from your threshold to x_max. We've got you started.

```
In [0]: # Generate the scatter plot
        x = training_examples["compression-ratio"]
        y = training_examples["city-mpg"]
        plt.ylabel("city-mpg")
        plt.xlabel("compression-ratio")
        plt.scatter(x, y, color='black', label="")


        # Plot line for x.min() to threshold
```

```
    # Plot line from threshold to x.max()
```