Lab_7_Feature_Engineering_Creating_Synthetic_Features

April 24, 2019

1 Lab 7: Feature Engineering - Creating Synthetic Features

Learning Objectives: * Gain more experience with the `LinearRegressor` class in TensorFlow by using it to predict median housing price, at the granularity of city blocks * Use a validation data set and test set to make sure that our model will generalize and is not overfitting the training data. * Use test data only after tuning hyperparameters as a measure of how the model will generalize to new data * Create synthetic features from the existing features (e.g., taking a ratio of two other features) * More practice with feature transformations including identifying and clipping (removing) outliers out of the input data to obtain the best model

1.0.1 Standard Set-up

We begin with the same set-up as in the last lab.

```
In [0]: 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
```

Read the data and randomize the order.

```
In [0]: california_housing_dataframe = pd.read_csv("https://storage.googleapis.com/ml_universi

        california_housing_dataframe = california_housing_dataframe.reindex(
            np.random.permutation(california_housing_dataframe.index))

        california_housing_dataframe.describe()
```

1.1 Prepare Features

As our learning models get more sophisticated we will want to do some computation on the features and even generate new features from the existing features. You will change this later in the lab. For now this method will just make a copy of the portion of the dataframe we plan to use, and re-scale the median-house value (to make it a bit easier to work with).

```
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()

          # Modifying median_house_value to be in scale of $1000.  So a value of 14.0
          # will correspond to $14,000.  This will make it a bit easier to work with.
          processed_features["median_house_value"] /= 1000.0

          return processed_features
```

1.1.1 Divide the provided data for training our model into a training and validation set

As in the last lab we use the first 14000 examples (after randomization) for the ***training set*** and the remaining 3000 examples for the ***validation set***.

```
In [0]: training_examples = prepare_features(california_housing_dataframe.head(14000))
        validation_examples = prepare_features(california_housing_dataframe.tail(3000))
```

1.1.2 Load the Test Data

As in the last lab we load the test data from here.

```
In [0]: california_housing_test_data = pd.read_csv(
            "https://storage.googleapis.com/ml_universities/california_housing_test.csv",
            sep=",")

        test_examples = prepare_features(california_housing_test_data)
```

1.1.3 Compute Loss

Here is a simple method to compute the loss on the given input function and targets.

```
In [0]: def compute_loss(model, input_fn, targets):
          """ Computes the loss (RMSE) for linear regression.

          Args:
            model: the trained model to use for making the predictions.
            input_fn: the input_fn to use to make the predictions.
            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.
          """
          predictions = list(model.predict(input_fn=input_fn))
          return math.sqrt(metrics.mean_squared_error(predictions, targets))
```

1.1.4 Setting Up the Feature Columns and Input Function for TensorFlow

As in the last lab we define `input_fn` to create a real-valued feature for each provided numerical column, and then define `train_input_fn` to use the training data, `eval_input_fn` to use the validation data, and `test_input_fn` to use the test data.

```
In [0]: CATEGORICAL_COLUMNS = []
        NUMERICAL_COLUMNS = ["longitude", "latitude", "housing_median_age",
                             "total_rooms", "total_bedrooms", "population",
                             "households", "median_income"]
        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 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 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():
  return input_fn(training_examples)

def eval_input_fn():
  return input_fn(validation_examples)

def test_input_fn():
  return input_fn(test_examples)
```

1.1.5 Functions to help visualize our results

We will use our functions from the last lab to generate a calibration plot and learning curve (with both training and validation losses).

```
In [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, validation_losses):
  """ Plot the learning curve.

  Args:
    training_loses: a list of training losses to plot.
    validation_losses: a list of validation losses to plot.
  """
  plt.ylabel('Loss')
  plt.xlabel('Training Steps')
  plt.plot(training_losses, label="training")
  plt.plot(validation_losses, label="validation")
  plt.legend(loc=1)
```

1.1.6 Defining the features, linear regression model, and function to train the model

These functions are just like the last lab except now we include all of the available numerical features.

```
In [0]: NUMERICAL_FEATURES = ["longitude", "latitude", "housing_median_age",
                              "total_rooms", "total_bedrooms", "population",
                              "households"]
        LABEL = "median_income"

        def construct_feature_columns():
          """Construct TensorFlow Feature Columns for features.

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

          Args:
            linear_regressor: The regressor to train.
            steps: A non-zero `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 training and validation losses so we can
          # generate a learning curve.
          print "Training model..."
          training_losses = []
          validation_losses = []

          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)

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

          # Now that training is done print the final training and validation losses.
          print "Final Training Loss (RMSE): %0.3f" % training_loss
          print "Final Validation Loss (RMSE): %0.3f" % validation_loss

          # Generate a figure with the learning curve on the left and a
          # calibration plot 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, validation_losses)

          plt.subplot(1, 2, 2)
          plt.tight_layout(pad=1.1, w_pad=3.0, h_pad=3.0)
          plt.title("Calibration Plot on Validation Data")
          validation_predictions = np.array(list(linear_regressor.predict(
              input_fn=eval_input_fn)))
          make_calibration_plot(validation_predictions, validation_examples[LABEL])

          return linear_regressor
```

1.1.7 Training a model with one feature.

In the last lab, you trained a model to predict the `median_house_value` from a single feature. Before we explore what can be done by introducing additional features, let's just train a good model to use a single feature. Feel free to change the below, but as is it should give a pretty good result (given the constraint of using a single feature).

```
In [0]: NUMERICAL_FEATURES = ["median_income"]
        LABEL = "median_house_value"

        LEARNING_RATE = 0.1
        STEPS = 250

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

Once you've adjusted the hyperparameters (if you'd like to see if you can reduce the validation loss), let's check the loss on the test data.

```
In [0]: print "loss on test data is", compute_loss(
            linear_regressor, test_input_fn, test_examples[LABEL])
```

1.2 Task 1: Introduce Synthetic Features (1 Point)

Both the total_rooms and population features count totals for a given city block. But what if one city block were more densely populated than another? Then just using `total_rooms` and `population` directly might not be very useful. Instead what we really want is the **quadratic feature** obtain by dividing `total_rooms` by the `population` to give the average number of rooms per person for that city block.

We've got you started by creating a feature called rooms_per_person. Create others, and use them along with whatever other features you think are useful to train a model.

```
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()

    # Modifying median_house_value to be in scale of $1000.  So a value of 14.0
    # will correspond to $14,000.  This will make it a bit easier to work with.
    processed_features["median_house_value"] /= 1000.0

    # Add your synthetic features here. We've got you started by defining
    # rooms_per_person
    processed_features["rooms_per_person"] = (
      dataframe["total_rooms"] / dataframe["population"])

    return processed_features

  # Generate the training, validation and test examples
  training_examples = prepare_features(california_housing_dataframe.head(14000))
  validation_examples = prepare_features(california_housing_dataframe.tail(3000))
  test_examples = prepare_features(california_housing_test_data)

  validation_examples.describe()
```

Since we added new features, we need to add them to NUMERICAL_COLUMNS so that they are included in the dictionary created by input_fn.

```
In [0]: # Add any other synthetic features you create to this list.
        NUMERICAL_COLUMNS = ["longitude", "latitude", "housing_median_age",
                             "total_rooms", "total_bedrooms", "population",
                             "households", "median_income", "rooms_per_person"]
```

1.3 Task 2 -- Add a Clip Feature Transformation (1 point)

Recall that there are two characteristics we'd like of numerical features when used together to train a linear model: * The range of the features is roughly the same. * To the extent possible the histogram of the features kind of resembles a bell curve. Sometimes the data will fit this very well and other times it won't.

Below are the methods to perform linear scaling and log scaling. For this data set it will also be useful to have a feature transformation to cap the features to within a minimum and/or maximum value. Most likely you will want to then linearly scale or log scale the feature after clipping it.

```
In [0]: # Linearly rescales to the range [0, 1]
        def linear_scale(series):
          min_val = series.min()
          max_val = series.max()
```

```
    scale = 1.0 * (max_val - min_val)
    return series.apply(lambda x:((x - min_val) / scale))

  # Perform log scaling
  def log_scale(series):
    return series.apply(lambda x:math.log(x+1.0))

  # Clip all features to given min and max
  def clip(series, clip_to_min, clip_to_max):
    # You need to modify this to actually do the clipping versus just returning
    # the series unchanged.
    return series.apply(lambda x: x)
```

You can use this function to draw a histogram to help decide what kind of scaling is best to use for `households` and also to confirm your implementation of `clip` works as you intended.

```
In [0]: clip_min = -np.inf
        clip_max = np.inf

        def draw_histograms(dataframe, feature_name,
                            clip_min = -np.inf, clip_max = np.inf):
          plt.figure(figsize=(20, 4))
          plt.subplot(1, 3, 1)
          plt.title(feature_name)
          histogram = dataframe[feature_name].hist(bins=50)

          plt.subplot(1, 3, 2)
          plt.title("linear_scaling")
          scaled_features = pd.DataFrame()
          scaled_features[feature_name] = linear_scale(
              clip(dataframe[feature_name], clip_min, clip_max))
          histogram = scaled_features[feature_name].hist(bins=50)

          plt.subplot(1, 3, 3)
          plt.title("log scaling")
          log_normalized_features = pd.DataFrame()
          log_normalized_features[feature_name] = log_scale(dataframe[feature_name])
          histogram = log_normalized_features[feature_name].hist(bins=50)

        draw_histograms(training_examples, 'households')
```

```
In [0]: print "Now let's clip between 0 and 1000 before scaling."

        draw_histograms(training_examples, 'households', 0, 1000)
```

The leftmost histogram is unchanged since that just shows the raw feature. You should see a very visible change when you clip the feature before applying linear scaling. As you will often (but not always) find, for this particular feature log scaling worked well without the need to first clip the data.

Run the above code with features other than `households` to help decide what feature normalization to use for each feature. Feel free to duplicate the code box that calls `draw_histogram` if you'd like to show the histograms for multiple features.

1.4 Task 3: Train the Best Model You Can (3 points)

We expect you to take some time exploring the feature transformations and hyperparameters.

Select any number of the provided features, create synthetic features you think will be informative, and modify the hyperparmaters to get a better model. You will want to edit preprocess_features to do some feature normalization like you saw in the Using Multiple Numerical Features and Feature Scaling. See how well you are able to do.

DO NOT APPLY ANY ADDITIONAL FEATURE TRANSFORMATION TO THE TARGET `median_price` since that would change the scale for RMSE.

- Summarize the changes you made that were the most important.
- Once you find a model, try training 20 times more steps. Does overfitting occur if you do that?

```
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()

          # Modifying median_house_value to be in scale of $1000.  So a value of 14.0
          # will correspond to $14,000.  This will make it a bit easier to work with.
          processed_features["median_house_value"] /= 1000.0

          # Perform your feature scaling here


          # Add your synthetic features here along with the feature scaling you'd like
          # to them. As a starting point linear scaling is used for rooms_per_person.
          # You are encouraged to experiment with different scaling options.
          processed_features["rooms_per_person"] = (
            linear_scale(dataframe["total_rooms"] / dataframe["population"]))

          return processed_features

        # Generate the training, validation and test examples
        training_examples = prepare_features(california_housing_dataframe.head(14000))
        validation_examples = prepare_features(california_housing_dataframe.tail(3000))
        test_examples = prepare_features(california_housing_test_data)
```

```
training_examples.describe()
```

Here's the method to train the model. You'l need to fill in the features you want to use.

```
In [0]: # Fill in the features you want to use
        NUMERICAL_FEATURES = ["total_rooms"]
        LABEL = "median_house_value"

        LEARNING_RATE = 0.5
        STEPS = 100

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

You can look at the weights of the trained model.

```
In [0]: # Let's also look at the weights and bias
        for feature in NUMERICAL_FEATURES:
          print "weight for", feature, ":", linear_regressor.get_variable_value(
            "linear/" + feature + "/weight")[0]
        print "bias:",  linear_regressor.get_variable_value("linear/bias_weight")
```

Check the loss on the test data after you are done selecting the learning rate and number of steps to run.

```
In [0]: print "loss on test data is", compute_loss(
            linear_regressor, test_input_fn, test_examples[LABEL])
```

Here's a code box for you to train your model with the same learning rate that worked best but to see what happens if you train it 20 times longer.

```
In [0]: # Fill in the features you want to use
        NUMERICAL_FEATURES = ["total_rooms"]
        LABEL = "median_house_value"

        LEARNING_RATE = 0.5
        STEPS = 100

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

```
In [0]: """
        Here's a place for you to answer these questions:


        A) Summarize the changes you made that were the most important.
```

```
B) Once you find a model, try training 20 times more steps.  Does overfitting
   occur if you do that?

"""
```