feature_crosses

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 Feature Crosses

Learning Objectives: * Improve a linear regression model with the addition of additional synthetic features (this is a continuation of the previous exercise) * Use an input function to convert pandas `DataFrame` objects to `Tensors` and invoke the input function in `fit()` and `predict()` operations * Use the FTRL optimization algorithm for model training * Create new synthetic features through one-hot encoding, binning, and feature crosses

1.1 Setup

First, as we've done in previous exercises, let's define the input and create the data-loading code.

```
In [0]: 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

california_housing_dataframe = pd.read_csv("https://download.mlcc.google.com/mledu-data

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

In [0]:
```
def preprocess_features(california_housing_dataframe):
  """Prepares input features from California housing data set.

  Args:
    california_housing_dataframe: A Pandas DataFrame expected to contain data
      from the California housing data set.
  Returns:
    A DataFrame that contains the features to be used for the model, including
    synthetic features.
  """
  selected_features = california_housing_dataframe[
    ["latitude",
     "longitude",
     "housing_median_age",
     "total_rooms",
     "total_bedrooms",
     "population",
     "households",
     "median_income"]]
  processed_features = selected_features.copy()
  # Create a synthetic feature.
  processed_features["rooms_per_person"] = (
    california_housing_dataframe["total_rooms"] /
    california_housing_dataframe["population"])
  return processed_features

def preprocess_targets(california_housing_dataframe):
  """Prepares target features (i.e., labels) from California housing data set.

  Args:
    california_housing_dataframe: A Pandas DataFrame expected to contain data
      from the California housing data set.
  Returns:
    A DataFrame that contains the target feature.
  """
  output_targets = pd.DataFrame()
  # Scale the target to be in units of thousands of dollars.
```

```
    output_targets["median_house_value"] = (
      california_housing_dataframe["median_house_value"] / 1000.0)
    return output_targets
```

In [0]:
```
# Choose the first 12000 (out of 17000) examples for training.
training_examples = preprocess_features(california_housing_dataframe.head(12000))
training_targets = preprocess_targets(california_housing_dataframe.head(12000))

# Choose the last 5000 (out of 17000) examples for validation.
validation_examples = preprocess_features(california_housing_dataframe.tail(5000))
validation_targets = preprocess_targets(california_housing_dataframe.tail(5000))

# Double-check that we've done the right thing.
print("Training examples summary:")
display.display(training_examples.describe())
print("Validation examples summary:")
display.display(validation_examples.describe())

print("Training targets summary:")
display.display(training_targets.describe())
print("Validation targets summary:")
display.display(validation_targets.describe())
```

In [0]:
```
def construct_feature_columns(input_features):
  """Construct the TensorFlow Feature Columns.

  Args:
    input_features: The names of the numerical input features to use.
  Returns:
    A set of feature columns
  """
  return set([tf.feature_column.numeric_column(my_feature)
              for my_feature in input_features])
```

In [0]:
```
def my_input_fn(features, targets, batch_size=1, shuffle=True, num_epochs=None):
    """Trains a linear regression model.

    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 in
    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(10000)

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

1.2 FTRL Optimization Algorithm

High dimensional linear models benefit from using a variant of gradient-based optimization called FTRL. This algorithm has the benefit of scaling the learning rate differently for different coefficients, which can be useful if some features rarely take non-zero values (it also is well suited to support L1 regularization). We can apply FTRL using the FtrlOptimizer.

```
In [0]: def train_model(
            learning_rate,
            steps,
            batch_size,
            feature_columns,
            training_examples,
            training_targets,
            validation_examples,
            validation_targets):
          """Trains a linear regression model.

          In addition to training, this function also prints training progress information,
          as well as a plot of the training and validation loss over time.

          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.
            feature_columns: A `set` specifying the input feature columns to use.
            training_examples: A `DataFrame` containing one or more columns from
              `california_housing_dataframe` to use as input features for training.
            training_targets: A `DataFrame` containing exactly one column from
              `california_housing_dataframe` to use as target for training.
            validation_examples: A `DataFrame` containing one or more columns from
              `california_housing_dataframe` to use as input features for validation.
            validation_targets: A `DataFrame` containing exactly one column from
              `california_housing_dataframe` to use as target for validation.
```

```
  Returns:
    A `LinearRegressor` object trained on the training data.
  """

  periods = 10
  steps_per_period = steps / periods

  # Create a linear regressor object.
  my_optimizer = tf.train.FtrlOptimizer(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
  )

  training_input_fn = lambda: my_input_fn(training_examples,
                                          training_targets["median_house_value"],
                                          batch_size=batch_size)
  predict_training_input_fn = lambda: my_input_fn(training_examples,
                                                  training_targets["median_house_value"
                                                  num_epochs=1,
                                                  shuffle=False)
  predict_validation_input_fn = lambda: my_input_fn(validation_examples,
                                                    validation_targets["median_house_va
                                                    num_epochs=1,
                                                    shuffle=False)

  # 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):")
  training_rmse = []
  validation_rmse = []
  for period in range (0, periods):
    # Train the model, starting from the prior state.
    linear_regressor.train(
        input_fn=training_input_fn,
        steps=steps_per_period
    )
    # Take a break and compute predictions.
    training_predictions = linear_regressor.predict(input_fn=predict_training_input_fn)
    training_predictions = np.array([item['predictions'][0] for item in training_predic
    validation_predictions = linear_regressor.predict(input_fn=predict_validation_input
    validation_predictions = np.array([item['predictions'][0] for item in validation_pr

    # Compute training and validation loss.
    training_root_mean_squared_error = math.sqrt(
```

```
            metrics.mean_squared_error(training_predictions, training_targets))
        validation_root_mean_squared_error = math.sqrt(
            metrics.mean_squared_error(validation_predictions, validation_targets))
        # Occasionally print the current loss.
        print("  period %02d : %0.2f" % (period, training_root_mean_squared_error))
        # Add the loss metrics from this period to our list.
        training_rmse.append(training_root_mean_squared_error)
        validation_rmse.append(validation_root_mean_squared_error)
    print("Model training finished.")


    # Output a graph of loss metrics over periods.
    plt.ylabel("RMSE")
    plt.xlabel("Periods")
    plt.title("Root Mean Squared Error vs. Periods")
    plt.tight_layout()
    plt.plot(training_rmse, label="training")
    plt.plot(validation_rmse, label="validation")
    plt.legend()

    return linear_regressor
```

```
In [0]: _ = train_model(
            learning_rate=1.0,
            steps=500,
            batch_size=100,
            feature_columns=construct_feature_columns(training_examples),
            training_examples=training_examples,
            training_targets=training_targets,
            validation_examples=validation_examples,
            validation_targets=validation_targets)
```

1.3 One-Hot Encoding for Discrete Features

Discrete (i.e. strings, enumerations, integers) features are usually converted into families of binary features before training a logistic regression model.

For example, suppose we created a synthetic feature that can take any of the values `0`, `1` or `2`, and that we have a few training points:

#	feature_value
0	2
1	0
2	1

For each possible categorical value, we make a new **binary** feature of **real values** that can take one of just two possible values: 1.0 if the example has that value, and 0.0 if not. In the example above, the categorical feature would be converted into three features, and the training points now

look like:

#	feature_value_0	feature_value_1	feature_value_2
0	0.0	0.0	1.0
1	1.0	0.0	0.0
2	0.0	1.0	0.0

1.4 Bucketized (Binned) Features

Bucketization is also known as binning.

We can bucketize `population` into the following 3 buckets (for instance): - `bucket_0` (`< 5000`): corresponding to less populated blocks - `bucket_1` (`5000 – 25000`): corresponding to mid populated blocks - `bucket_2` (`> 25000`): corresponding to highly populated blocks

Given the preceding bucket definitions, the following `population` vector:

```
[[10001], [42004], [2500], [18000]]
```

becomes the following bucketized feature vector:

```
[[1], [2], [0], [1]]
```

The feature values are now the bucket indices. Note that these indices are considered to be discrete features. Typically, these will be further converted in one-hot representations as above, but this is done transparently.

To define feature columns for bucketized features, instead of using `numeric_column`, we can use `bucketized_column`, which takes a numeric column as input and transforms it to a bucketized feature using the bucket boundaries specified in the `boundaries` argument. The following code defines bucketized feature columns for `households` and `longitude`; the `get_quantile_based_boundaries` function calculates boundaries based on quantiles, so that each bucket contains an equal number of elements.

```
In [0]: def get_quantile_based_boundaries(feature_values, num_buckets):
          boundaries = np.arange(1.0, num_buckets) / num_buckets
          quantiles = feature_values.quantile(boundaries)
          return [quantiles[q] for q in quantiles.keys()]

        # Divide households into 7 buckets.
        households = tf.feature_column.numeric_column("households")
        bucketized_households = tf.feature_column.bucketized_column(
          households, boundaries=get_quantile_based_boundaries(
            california_housing_dataframe["households"], 7))

        # Divide longitude into 10 buckets.
        longitude = tf.feature_column.numeric_column("longitude")
        bucketized_longitude = tf.feature_column.bucketized_column(
          longitude, boundaries=get_quantile_based_boundaries(
            california_housing_dataframe["longitude"], 10))
```

1.5 Task 1: Train the Model on Bucketized Feature Columns

Bucketize all the real valued features in our example, train the model and see if the results improve.

In the preceding code block, two real valued columns (namely `households` and `longitude`) have been transformed into bucketized feature columns. Your task is to bucketize the rest of the columns, then run the code to train the model. There are various heuristics to find the range of the buckets. This exercise uses a quantile-based technique, which chooses the bucket boundaries in such a way that each bucket has the same number of examples.

```
In [0]: def construct_feature_columns():
          """Construct the TensorFlow Feature Columns.

          Returns:
            A set of feature columns
          """
          households = tf.feature_column.numeric_column("households")
          longitude = tf.feature_column.numeric_column("longitude")
          latitude = tf.feature_column.numeric_column("latitude")
          housing_median_age = tf.feature_column.numeric_column("housing_median_age")
          median_income = tf.feature_column.numeric_column("median_income")
          rooms_per_person = tf.feature_column.numeric_column("rooms_per_person")

          # Divide households into 7 buckets.
          bucketized_households = tf.feature_column.bucketized_column(
            households, boundaries=get_quantile_based_boundaries(
              training_examples["households"], 7))

          # Divide longitude into 10 buckets.
          bucketized_longitude = tf.feature_column.bucketized_column(
            longitude, boundaries=get_quantile_based_boundaries(
              training_examples["longitude"], 10))

          #
          # YOUR CODE HERE: bucketize the following columns, following the example above:
          #
          bucketized_latitude =
          bucketized_housing_median_age =
          bucketized_median_income =
          bucketized_rooms_per_person =

          feature_columns = set([
            bucketized_longitude,
            bucketized_latitude,
            bucketized_housing_median_age,
            bucketized_households,
            bucketized_median_income,
            bucketized_rooms_per_person])
```

```
    return feature_columns
```

```
In [0]: _ = train_model(
          learning_rate=1.0,
          steps=500,
          batch_size=100,
          feature_columns=construct_feature_columns(),
          training_examples=training_examples,
          training_targets=training_targets,
          validation_examples=validation_examples,
          validation_targets=validation_targets)
```

1.5.1 Solution

Click below for a solution.

You may be wondering how to determine how many buckets to use. That is of course data-dependent. Here, we just selected arbitrary values so as to obtain a not-too-large model.

```
In [0]: def construct_feature_columns():
          """Construct the TensorFlow Feature Columns.

          Returns:
            A set of feature columns
          """
          households = tf.feature_column.numeric_column("households")
          longitude = tf.feature_column.numeric_column("longitude")
          latitude = tf.feature_column.numeric_column("latitude")
          housing_median_age = tf.feature_column.numeric_column("housing_median_age")
          median_income = tf.feature_column.numeric_column("median_income")
          rooms_per_person = tf.feature_column.numeric_column("rooms_per_person")

          # Divide households into 7 buckets.
          bucketized_households = tf.feature_column.bucketized_column(
            households, boundaries=get_quantile_based_boundaries(
              training_examples["households"], 7))

          # Divide longitude into 10 buckets.
          bucketized_longitude = tf.feature_column.bucketized_column(
            longitude, boundaries=get_quantile_based_boundaries(
              training_examples["longitude"], 10))

          # Divide latitude into 10 buckets.
          bucketized_latitude = tf.feature_column.bucketized_column(
            latitude, boundaries=get_quantile_based_boundaries(
              training_examples["latitude"], 10))

          # Divide housing_median_age into 7 buckets.
```

```
  bucketized_housing_median_age = tf.feature_column.bucketized_column(
    housing_median_age, boundaries=get_quantile_based_boundaries(
      training_examples["housing_median_age"], 7))

  # Divide median_income into 7 buckets.
  bucketized_median_income = tf.feature_column.bucketized_column(
    median_income, boundaries=get_quantile_based_boundaries(
      training_examples["median_income"], 7))

  # Divide rooms_per_person into 7 buckets.
  bucketized_rooms_per_person = tf.feature_column.bucketized_column(
    rooms_per_person, boundaries=get_quantile_based_boundaries(
      training_examples["rooms_per_person"], 7))

  feature_columns = set([
    bucketized_longitude,
    bucketized_latitude,
    bucketized_housing_median_age,
    bucketized_households,
    bucketized_median_income,
    bucketized_rooms_per_person])

  return feature_columns
```

```
In [0]: _ = train_model(
          learning_rate=1.0,
          steps=500,
          batch_size=100,
          feature_columns=construct_feature_columns(),
          training_examples=training_examples,
          training_targets=training_targets,
          validation_examples=validation_examples,
          validation_targets=validation_targets)
```

1.6 Feature Crosses

Crossing two (or more) features is a clever way to learn non-linear relations using a linear model. In our problem, if we just use the feature `latitude` for learning, the model might learn that city blocks at a particular latitude (or within a particular range of latitudes since we have bucketized it) are more likely to be expensive than others. Similarly for the feature `longitude`. However, if we cross `longitude` by `latitude`, the crossed feature represents a well defined city block. If the model learns that certain city blocks (within range of latitudes and longitudes) are more likely to be more expensive than others, it is a stronger signal than two features considered individually.

Currently, the feature columns API only supports discrete features for crosses. To cross two continuous values, like `latitude` or `longitude`, we can bucketize them.

If we cross the `latitude` and `longitude` features (supposing, for example, that `longitude` was bucketized into `2` buckets, while `latitude` has `3` buckets), we actually get six crossed binary features. Each of these features will get its own separate weight when we train the model.

1.7 Task 2: Train the Model Using Feature Crosses

Add a feature cross of `longitude` and `latitude` to your model, train it, and determine whether the results improve.

Refer to the TensorFlow API docs for `crossed_column()` to build the feature column for your cross. Use a `hash_bucket_size` of 1000.

```
In [0]: def construct_feature_columns():
          """Construct the TensorFlow Feature Columns.

          Returns:
            A set of feature columns
          """
          households = tf.feature_column.numeric_column("households")
          longitude = tf.feature_column.numeric_column("longitude")
          latitude = tf.feature_column.numeric_column("latitude")
          housing_median_age = tf.feature_column.numeric_column("housing_median_age")
          median_income = tf.feature_column.numeric_column("median_income")
          rooms_per_person = tf.feature_column.numeric_column("rooms_per_person")

          # Divide households into 7 buckets.
          bucketized_households = tf.feature_column.bucketized_column(
            households, boundaries=get_quantile_based_boundaries(
              training_examples["households"], 7))

          # Divide longitude into 10 buckets.
          bucketized_longitude = tf.feature_column.bucketized_column(
            longitude, boundaries=get_quantile_based_boundaries(
              training_examples["longitude"], 10))

          # Divide latitude into 10 buckets.
          bucketized_latitude = tf.feature_column.bucketized_column(
            latitude, boundaries=get_quantile_based_boundaries(
              training_examples["latitude"], 10))

          # Divide housing_median_age into 7 buckets.
          bucketized_housing_median_age = tf.feature_column.bucketized_column(
            housing_median_age, boundaries=get_quantile_based_boundaries(
              training_examples["housing_median_age"], 7))

          # Divide median_income into 7 buckets.
          bucketized_median_income = tf.feature_column.bucketized_column(
            median_income, boundaries=get_quantile_based_boundaries(
              training_examples["median_income"], 7))

          # Divide rooms_per_person into 7 buckets.
          bucketized_rooms_per_person = tf.feature_column.bucketized_column(
            rooms_per_person, boundaries=get_quantile_based_boundaries(
```

```
    training_examples["rooms_per_person"], 7))

  # YOUR CODE HERE: Make a feature column for the long_x_lat feature cross
  long_x_lat =

  feature_columns = set([
    bucketized_longitude,
    bucketized_latitude,
    bucketized_housing_median_age,
    bucketized_households,
    bucketized_median_income,
    bucketized_rooms_per_person,
    long_x_lat])

  return feature_columns
```

```
In [0]: _ = train_model(
          learning_rate=1.0,
          steps=500,
          batch_size=100,
          feature_columns=construct_feature_columns(),
          training_examples=training_examples,
          training_targets=training_targets,
          validation_examples=validation_examples,
          validation_targets=validation_targets)
```

1.7.1 Solution

Click below for the solution.

```
In [0]: def construct_feature_columns():
          """Construct the TensorFlow Feature Columns.

          Returns:
            A set of feature columns
          """
          households = tf.feature_column.numeric_column("households")
          longitude = tf.feature_column.numeric_column("longitude")
          latitude = tf.feature_column.numeric_column("latitude")
          housing_median_age = tf.feature_column.numeric_column("housing_median_age")
          median_income = tf.feature_column.numeric_column("median_income")
          rooms_per_person = tf.feature_column.numeric_column("rooms_per_person")

          # Divide households into 7 buckets.
          bucketized_households = tf.feature_column.bucketized_column(
            households, boundaries=get_quantile_based_boundaries(
              training_examples["households"], 7))
```

```
  # Divide longitude into 10 buckets.
  bucketized_longitude = tf.feature_column.bucketized_column(
    longitude, boundaries=get_quantile_based_boundaries(
      training_examples["longitude"], 10))

  # Divide latitude into 10 buckets.
  bucketized_latitude = tf.feature_column.bucketized_column(
    latitude, boundaries=get_quantile_based_boundaries(
      training_examples["latitude"], 10))

  # Divide housing_median_age into 7 buckets.
  bucketized_housing_median_age = tf.feature_column.bucketized_column(
    housing_median_age, boundaries=get_quantile_based_boundaries(
      training_examples["housing_median_age"], 7))

  # Divide median_income into 7 buckets.
  bucketized_median_income = tf.feature_column.bucketized_column(
    median_income, boundaries=get_quantile_based_boundaries(
      training_examples["median_income"], 7))

  # Divide rooms_per_person into 7 buckets.
  bucketized_rooms_per_person = tf.feature_column.bucketized_column(
    rooms_per_person, boundaries=get_quantile_based_boundaries(
      training_examples["rooms_per_person"], 7))

  # YOUR CODE HERE: Make a feature column for the long_x_lat feature cross
  long_x_lat = tf.feature_column.crossed_column(
  set([bucketized_longitude, bucketized_latitude]), hash_bucket_size=1000)

  feature_columns = set([
    bucketized_longitude,
    bucketized_latitude,
    bucketized_housing_median_age,
    bucketized_households,
    bucketized_median_income,
    bucketized_rooms_per_person,
    long_x_lat])

  return feature_columns
```

In [0]:
```
_ = train_model(
  learning_rate=1.0,
  steps=500,
  batch_size=100,
  feature_columns=construct_feature_columns(),
  training_examples=training_examples,
  training_targets=training_targets,
  validation_examples=validation_examples,
```

```
      validation_targets=validation_targets)
```

1.8 Optional Challenge: Try Out More Synthetic Features

So far, we've tried simple bucketized columns and feature crosses, but there are many more combinations that could potentially improve the results. For example, you could cross multiple columns. What happens if you vary the number of buckets? What other synthetic features can you think of? Do they improve the model?