feature_sets

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 Sets

Learning Objective: Create a minimal set of features that performs just as well as a more complex feature set

So far, we've thrown all of our features into the model. Models with fewer features use fewer resources and are easier to maintain. Let's see if we can build a model on a minimal set of housing features that will perform equally as well as one that uses all the features in the data set.

1.1 Setup

As before, let's load and prepare the California housing data.

```
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())
```

1.2 Task 1: Develop a Good Feature Set

What's the best performance you can get with just 2 or 3 features?

A **correlation matrix** shows pairwise correlations, both for each feature compared to the target and for each feature compared to other features.

Here, correlation is defined as the Pearson correlation coefficient. You don't have to understand the mathematical details for this exercise.

Correlation values have the following meanings:

- `-1.0`: perfect negative correlation
- `0.0`: no correlation
- `1.0`: perfect positive correlation

In [0]:
```
correlation_dataframe = training_examples.copy()
correlation_dataframe["target"] = training_targets["median_house_value"]

correlation_dataframe.corr()
```

Features that have strong positive or negative correlations with the target will add information to our model. We can use the correlation matrix to find such strongly correlated features.

We'd also like to have features that aren't so strongly correlated with each other, so that they add independent information.

Use this information to try removing features. You can also try developing additional synthetic features, such as ratios of two raw features.

For convenience, we've included the training code from the previous exercise.

```
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

In [0]: def train_model(
            learning_rate,
            steps,
            batch_size,
            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.
  batch_size: A non-zero `int`, the batch size.
  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.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=construct_feature_columns(training_examples),
    optimizer=my_optimizer
)

# Create input functions.
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_predi

      validation_predictions = linear_regressor.predict(input_fn=predict_validation_inpu
      validation_predictions = np.array([item['predictions'][0] for item in validation_p

      # 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
```

Spend 5 minutes searching for a good set of features and training parameters. Then check the solution to see what we chose. Don't forget that different features may require different learning parameters.

```
In [0]: #
        # Your code here: add your features of choice as a list of quoted strings.
        #
```

```
minimal_features = [
]

assert minimal_features, "You must select at least one feature!"

minimal_training_examples = training_examples[minimal_features]
minimal_validation_examples = validation_examples[minimal_features]

#
# Don't forget to adjust these parameters.
#
train_model(
    learning_rate=0.001,
    steps=500,
    batch_size=5,
    training_examples=minimal_training_examples,
    training_targets=training_targets,
    validation_examples=minimal_validation_examples,
    validation_targets=validation_targets)
```

1.2.1 Solution

Click below for a solution.

```
In [0]: minimal_features = [
          "median_income",
          "latitude",
        ]

        minimal_training_examples = training_examples[minimal_features]
        minimal_validation_examples = validation_examples[minimal_features]

        _ = train_model(
            learning_rate=0.01,
            steps=500,
            batch_size=5,
            training_examples=minimal_training_examples,
            training_targets=training_targets,
            validation_examples=minimal_validation_examples,
            validation_targets=validation_targets)
```

1.3 Task 2: Make Better Use of Latitude

Plotting `latitude` vs. `median_house_value` shows that there really isn't a linear relationship there.

Instead, there are a couple of peaks, which roughly correspond to Los Angeles and San Francisco.

```
In [0]: plt.scatter(training_examples["latitude"], training_targets["median_house_value"])
```

Try creating some synthetic features that do a better job with latitude.

For example, you could have a feature that maps `latitude` to a value of `|latitude - 38|`, and call this `distance_from_san_francisco`.

Or you could break the space into 10 different buckets. `latitude_32_to_33`, `latitude_33_to_34`, etc., each showing a value of `1.0` if `latitude` is within that bucket range and a value of `0.0` otherwise.

Use the correlation matrix to help guide development, and then add them to your model if you find something that looks good.

What's the best validation performance you can get?

```
In [0]: #
        # YOUR CODE HERE: Train on a new data set that includes synthetic features based on la
        #
```

1.3.1 Solution

Click below for a solution.

Aside from `latitude`, we'll also keep `median_income`, to compare with the previous results.

We decided to bucketize the latitude. This is fairly straightforward in Pandas using `Series.apply`.

```
In [0]: def select_and_transform_features(source_df):
          LATITUDE_RANGES = zip(range(32, 44), range(33, 45))
          selected_examples = pd.DataFrame()
          selected_examples["median_income"] = source_df["median_income"]
          for r in LATITUDE_RANGES:
            selected_examples["latitude_%d_to_%d" % r] = source_df["latitude"].apply(
              lambda l: 1.0 if l >= r[0] and l < r[1] else 0.0)
          return selected_examples

        selected_training_examples = select_and_transform_features(training_examples)
        selected_validation_examples = select_and_transform_features(validation_examples)
```

```
In [0]: _ = train_model(
            learning_rate=0.01,
            steps=500,
            batch_size=5,
            training_examples=selected_training_examples,
            training_targets=training_targets,
            validation_examples=selected_validation_examples,
            validation_targets=validation_targets)
```