EstimatorSpec

Use the EstimatorSpec object to organize training, evaluation, and prediction.

Chapter Goals:

- Learn how to evaluate a regression model
- Use the **EstimatorSpec** object to organize results from training, evaluation, and prediction

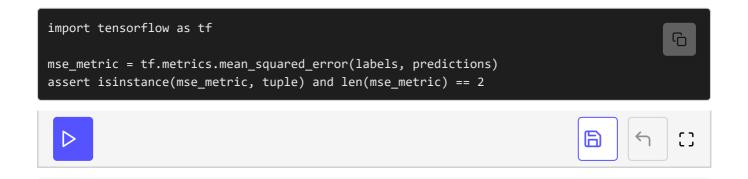
A. Regression evaluation

Unlike classification models, we can't use the accuracy metric to evaluate regression. Since the output of regression models is a real number, rather than a class prediction, there's no definite way to say what is "correct" or "incorrect".

However, we can tell how good a model output is based on its distance from the corresponding label. For example, if the label for some data observation was 0.2 and the model returned 0.199, the prediction is excellent. On the other hand, if the model returned 812.11, the prediction is likely very poor.

The metric that corresponds to this idea is mean squared error (MSE). The MSE is very similar to the L2-norm, in that both are based on the squared difference between labels and predictions.

In TensorFlow, we obtain the MSE metric using tf.metrics.mean_squared_error. The function takes in the labels and model predictions as its two required arguments.



Obtaining the MSE metric using tf.metrics.mean_squared_error. Note that the function's output is a tuple containing a tensor and an update operation.

The first element of the output tuple is a tensor representing the overall MSE after each evaluation step. The second tuple element is an operation that's used to update the overall MSE after each evaluation step. This gives us a cumulative MSE when we finish evaluating all the data observations.

B. Using EstimatorSpec

The upcoming chapters deal with TensorFlow's Estimator API, which encapsulates training, evaluating, and predicting into one compact object. However, in order to use the Estimator object, we need to first organize the model results in an EstimatorSpec.

The EstimatorSpec object is initialized with a single required argument, called mode. The mode can take one of three values:

- tf.estimator.ModeKeys.TRAIN
- tf.estimator.ModeKeys.EVAL
- tf.estimator.ModeKeys.PREDICT

The keyword arguments required to initialize the EstimatorSpec will differ depending on the mode.

Time to Code!

In this chapter you'll be completing the regressor_fn by creating helper functions for the EVAL and PREDICT blocks (the code for the TRAIN block is already filled in.

The first helper function you'll create is the eval_regressor function, for the EVAL block. It applies mean squared error as the evaluation metric.

Then set mse_metric equal to tf.metrics.mean_squared_error with labels as the first argument and self.predictions as the second argument.

Set eval_metric equal to a dictionary with 'mse' as the only key and mse_metric as the corresponding value.

When initializing <code>EstimatorSpec</code> in <code>EVAL</code> mode, the required keyword argument is <code>loss</code>, meaning that the evaluation will always use the model loss

as a metric.

We can add more evaluation metrics through the <code>eval_metric_ops</code> keyword argument. The argument takes in a dictionary, which maps string names for each metric to tuple values. Each tuple contains a tensor and update operation (e.g. <code>mse_metric</code>).

Set estimator_spec equal to tf.estimator.EstimatorSpec initialized with mode as the required argument. Use self.loss for the loss keyword argument and eval_metric for the eval_metric_ops keyword argument.

Then return estimator_spec.

```
import numpy as np
import tensorflow as tf
class RegressionModel(object):
   def __init__(self, output_size):
       self.output size = output size
   # Helper for regressor_fn
   def eval_regressor(self, mode, labels):
       # CODE HERE
       pass
   # Helper from previous chapter
   def set_predictions_and_loss(self, logits, labels):
       self.predictions = tf.squeeze(logits)
       if labels is not None:
            self.loss = tf.nn.l2_loss(labels - self.predictions)
   # The function for the regression model
   def regressor_fn(self, features, labels, mode, params):
        inputs = tf.feature_column.input_layer(features, params['feature_columns'])
       layer = inputs
       for num_nodes in params['hidden_layers']:
            layer = tf.layers.dense(layer, num_nodes,
                activation=tf.nn.relu)
       logits = tf.layers.dense(layer, self.output_size,
           name='logits')
       self.set_predictions_and_loss(logits, labels)
       if mode == tf.estimator.ModeKeys.TRAIN:
           self.global_step = tf.train.get_or_create_global_step()
           adam = tf.train.AdamOptimizer()
           self.train_op = adam.minimize(
                self.loss, global_step=self.global_step)
           return tf.estimator.EstimatorSpec(mode,
                loss=self.loss, train_op=self.train_op)
       if mode == tf.estimator.ModeKeys.EVAL:
           return self.eval_regressor(mode, labels)
       if mode == tf.estimator.ModeKeys.PREDICT:
           pass
```

The second helper function you'll create is the predict_regressor function, for the PREDICT block. It sets up the regression predictions for the model.

The features argument contains the names for each data observation. These names will be returned along with the model predictions, so we can easily identify each data observation.

Create a dictionary with two keys, 'predictions' and 'names', which map to self.predictions and features['name'], respectively. Store the dictionary in a variable named prediction_info.

When initializing EstimatorSpec in PREDICT mode, the required keyword argument is predictions. This represents a dictionary that contains the output values for prediction mode. The keys are string names used to identify each output, while the values are the output tensors.

Set estimator_spec equal to tf.estimator.EstimatorSpec with mode as the required argument and prediction_info as the predictions keyword argument.

Then return estimator_spec .

```
import numpy as np
                                                                                        G
import tensorflow as tf
class RegressionModel(object):
   def __init__(self, output_size):
       self.output_size = output_size
   # Helper for regressor_fn
   def predict regressor(self, mode, features):
       # CODE HERE
       pass
   # Helper from previous chapter
   def set_predictions_and_loss(self, logits, labels):
        self.predictions = tf.squeeze(logits)
       if labels is not None:
            self.loss = tf.nn.l2_loss(labels - self.predictions)
   # The function for the regression model
   def regressor_fn(self, features, labels, mode, params):
        inputs = tf.feature_column.input_layer(features, params['feature_columns'])
        layer = inputs
        for num_nodes in params['hidden_layers']:
            layer = tf.layers.dense(layer, num_nodes,
                activation=tf.nn.relu)
```







