```
    """Generates features and labels for training or evaluation.
    This uses the input pipeline based approach using file name queue
    to read data so that entire data is not loaded in memory.
    """
    dataset = tf.data.TextLineDataset(filenames).skip(skip_header_lines).map(
      _decode_csv)

    if shuffle:
        dataset = dataset.shuffle(buffer_size=batch_size * 10)
    iterator = dataset.repeat(num_epochs).batch(
        batch_size).make_one_shot_iterator()
    features = iterator.get_next()
    return features, parse_label_column(features.pop(LABEL_COLUMN))
```

The code for the most part is self-explanatory; however, the reader should take note of the following points:

- The function 'build_estimator' uses the canned Estimator API to train a 'DNNClassifier' model on Cloud MLE. The learning rate and hidden units of the model can be adjusted and tuned as a hyper-parameter during training.
- The methods 'csv_serving_input_fn' and 'json_serving_input_fn' define the serving inputs for CSV and JSON serving input formats.
- The method 'input_fn' uses the TensorFlow Dataset API to build the input pipelines for training and evaluation on Cloud MLE. This method calls the private method _decode_csv() to convert the CSV columns to Tensors.

The Application Logic

Let's see the application logic in the file '**task.py**'.

```
import argparse
import json
import os
```

```
import tensorflow as tf
from tensorflow.contrib.training.python.training import hparam

import trainer.model as model

def _get_session_config_from_env_var():
    """Returns a tf.ConfigProto instance that has appropriate device_
    filters set.
    """

    tf_config = json.loads(os.environ.get('TF_CONFIG', '{}'))

    if (tf_config and 'task' in tf_config and 'type' in tf_config['task'] and
       'index' in tf_config['task']):
        # Master should only communicate with itself and ps
        if tf_config['task']['type'] == 'master':
            return tf.ConfigProto(device_filters=['/job:ps', '/job:master'])
        # Worker should only communicate with itself and ps
        elif tf_config['task']['type'] == 'worker':
            return tf.ConfigProto(device_filters=[
                '/job:ps',
                '/job:worker/task:%d' % tf_config['task']['index']
            ])
    return None

def train_and_evaluate(hparams):
    """Run the training and evaluate using the high level API."""

    train_input = lambda: model.input_fn(
        hparams.train_files,
        num_epochs=hparams.num_epochs,
        batch_size=hparams.train_batch_size
    )

    # Don't shuffle evaluation data
    eval_input = lambda: model.input_fn(
        hparams.eval_files,
        batch_size=hparams.eval_batch_size,
        shuffle=False
    )
```

```
    train_spec = tf.estimator.TrainSpec(
        train_input, max_steps=hparams.train_steps)

    exporter = tf.estimator.FinalExporter(
        'iris', model.SERVING_FUNCTIONS[hparams.export_format])
    eval_spec = tf.estimator.EvalSpec(
        eval_input,
        steps=hparams.eval_steps,
        exporters=[exporter],
        name='iris-eval')

    run_config = tf.estimator.RunConfig(
        session_config=_get_session_config_from_env_var())
    run_config = run_config.replace(model_dir=hparams.job_dir)
    print('Model dir %s' % run_config.model_dir)
    estimator = model.build_estimator(
        learning_rate=hparams.learning_rate,
        # Construct layers sizes with exponential decay
        hidden_units=[
            max(2, int(hparams.first_layer_size * hparams.scale_factor**i))
            for i in range(hparams.num_layers)
        ],
        config=run_config)

    tf.estimator.train_and_evaluate(estimator, train_spec, eval_spec)

if __name__ == '__main__':
    parser = argparse.ArgumentParser()
    # Input Arguments
    parser.add_argument(
        '--train-files',
        help='GCS file or local paths to training data',
        nargs='+',
        default='gs://iris-dataset/train_data.csv')
    parser.add_argument(
        '--eval-files',
        help='GCS file or local paths to evaluation data',
```

```
        nargs='+',
        default='gs://iris-dataset/test_data.csv')
    parser.add_argument(
        '--job-dir',
        help='GCS location to write checkpoints and export models',
        default='/tmp/iris-estimator')
    parser.add_argument(
        '--num-epochs',
        help="""\
        Maximum number of training data epochs on which to train.
        If both --max-steps and --num-epochs are specified,
        the training job will run for --max-steps or --num-epochs,
        whichever occurs first. If unspecified will run for --max-steps.\
        """,
        type=int)
    parser.add_argument(
        '--train-batch-size',
        help='Batch size for training steps',
        type=int,
        default=20)
    parser.add_argument(
        '--eval-batch-size',
        help='Batch size for evaluation steps',
        type=int,
        default=20)
    parser.add_argument(
        '--learning_rate',
        help='The training learning rate',
        default=1e-4,
        type=int)
    parser.add_argument(
        '--first-layer-size',
        help='Number of nodes in the first layer of the DNN',
        default=256,
        type=int)
```

```
parser.add_argument(
    '--num-layers', help='Number of layers in the DNN', default=3,
    type=int)
parser.add_argument(
    '--scale-factor',
    help='How quickly should the size of the layers in the DNN decay',
    default=0.7,
    type=float)
parser.add_argument(
    '--train-steps',
    help="""\
    Steps to run the training job for. If --num-epochs is not specified,
    this must be. Otherwise the training job will run indefinitely.\
    """,
    default=100,
    type=int)
parser.add_argument(
    '--eval-steps',
    help='Number of steps to run evalution for at each checkpoint',
    default=100,
    type=int)
parser.add_argument(
    '--export-format',
    help='The input format of the exported SavedModel binary',
    choices=['JSON', 'CSV'],
    default='CSV')
parser.add_argument(
    '--verbosity',
    choices=['DEBUG', 'ERROR', 'FATAL', 'INFO', 'WARN'],
    default='INFO')

args, _ = parser.parse_known_args()

# Set python level verbosity
tf.logging.set_verbosity(args.verbosity)
# Set C++ Graph Execution level verbosity
```

```
    os.environ['TF_CPP_MIN_LOG_LEVEL'] = str(
        tf.logging.__dict__[args.verbosity] / 10)

    # Run the training job
    hparams = hparam.HParams(**args.__dict__)
    train_and_evaluate(hparams)
```

Note the following in the preceding code:

- The method '_get_session_config_from_env_var()' defines the configuration for the runtime environment on Cloud MLE for the Estimator.
- The method 'train_and_evaluate()' does a number of orchestration events including
  - Routing training and evaluation datasets to the model function in 'model.py'
  - Setting up the runtime environment of the Estimator
  - Passing hyper-parameters to the Estimator model
- The line of code "if __name__ == '__main__':" defines the entry point of the Python script via the terminal session. In this script, the code will receive inputs from the terminal through the 'argparse.ArgumentParser()' method.

Training on Cloud MLE

The training execution codes are bash commands stored in a shell script. Shell scripts end with the suffix '.sh.'

Running a Single Instance Training Job

The bash codes for executing training on a single instance on Cloud MLE is shown in the following. Change the bucket names accordingly.

```
DATE=`date '+%Y%m%d_%H%M%S'`
export JOB_NAME=iris_$DATE
```