logistic_regression

April 24, 2019

1 Logistic Regression

Learning Objectives: * Reframe the median house value predictor (from the preceding exercises) as a binary classification model * Compare the effectiveness of logisitic regression vs linear regression for a binary classification problem

As in the prior exercises, we're working with the California housing data set, but this time we will turn it into a binary classification problem by predicting whether a city block is a high-cost city block. We'll also revert to the default features, for now.

1.1 Frame the Problem as Binary Classification

The target of our dataset is `median_house_value` which is a numeric (continuous-valued) feature. We can create a boolean label by applying a threshold to this continuous value.

Given features describing a city block, we wish to predict if it is a high-cost city block. To prepare the targets for train and eval data, we define a classification threshold of the 75%-ile for median house value (a value of approximately 265000). All house values above the threshold are labeled `1`, and all others are labeled `0`.

1.2 Setup

Run the cells below to load the data and prepare the input features and targets.

```
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))
```

Note how the code below is slightly different from the previous exercises. Instead of using `median_house_value` as target, we create a new binary target, `median_house_value_is_high`.

```
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()
  # Create a boolean categorical feature representing whether the
  # median_house_value is above a set threshold.
  output_targets["median_house_value_is_high"] = (
    california_housing_dataframe["median_house_value"] > 265000).astype(float)
  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())
```

1.3 How Would Linear Regression Fare?

To see why logistic regression is effective, let us first train a naive model that uses linear regression. This model will use labels with values in the set {`0, 1`} and will try to predict a continuous value that is as close as possible to `0` or `1`. Furthermore, we wish to interpret the output as a probability, so it would be ideal if the output will be within the range `(0, 1)`. We would then apply a threshold of `0.5` to determine the label.

Run the cells below to train the linear regression model using LinearRegressor.

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_linear_regressor_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_is_high"
                                            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
```

```
In [0]: linear_regressor = train_linear_regressor_model(
          learning_rate=0.000001,
          steps=200,
          batch_size=20,
          training_examples=training_examples,
          training_targets=training_targets,
          validation_examples=validation_examples,
          validation_targets=validation_targets)
```

1.4 Task 1: Can We Calculate LogLoss for These Predictions?

Examine the predictions and decide whether or not we can use them to calculate LogLoss.

`LinearRegressor` uses the L2 loss, which doesn't do a great job at penalizing misclassifications when the output is interpreted as a probability. For example, there should be a huge difference whether a negative example is classified as positive with a probability of 0.9 vs 0.9999, but L2 loss doesn't strongly differentiate these cases.

In contrast, `LogLoss` penalizes these "confidence errors" much more heavily. Remember, `LogLoss` is defined as:

$$LogLoss = \sum_{(x,y)\in D} -y \cdot log(y_{pred}) - (1-y) \cdot log(1-y_{pred})$$

But first, we'll need to obtain the prediction values. We could use `LinearRegressor.predict` to obtain these.

Given the predictions and the targets, can we calculate `LogLoss`?

1.4.1 Solution

Click below to display the solution.

```
In [0]: predict_validation_input_fn = lambda: my_input_fn(validation_examples,
                                                  validation_targets["median_house_valu
                                                  num_epochs=1,
                                                  shuffle=False)

        validation_predictions = linear_regressor.predict(input_fn=predict_validation_input_fn)
        validation_predictions = np.array([item['predictions'][0] for item in validation_predic

        _ = plt.hist(validation_predictions)
```

1.5 Task 2: Train a Logistic Regression Model and Calculate LogLoss on the Validation Set

To use logistic regression, simply use LinearClassifier instead of `LinearRegressor`. Complete the code below.

NOTE: When running `train()` and `predict()` on a `LinearClassifier` model, you can access the real-valued predicted probabilities via the `"probabilities"` key in the returned dict—e.g., `predictions["probabilities"]`. Sklearn's log_loss function is handy for calculating LogLoss using these probabilities.

```
In [0]: def train_linear_classifier_model(
            learning_rate,
            steps,
            batch_size,
            training_examples,
            training_targets,
            validation_examples,
            validation_targets):
          """Trains a linear classification 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 `LinearClassifier` object trained on the training data.
  """

  periods = 10
  steps_per_period = steps / periods

  # Create a linear classifier object.
  my_optimizer = tf.train.GradientDescentOptimizer(learning_rate=learning_rate)
  my_optimizer = tf.contrib.estimator.clip_gradients_by_norm(my_optimizer, 5.0)
  linear_classifier = # YOUR CODE HERE: Construct the linear classifier.

  # Create input functions.
  training_input_fn = lambda: my_input_fn(training_examples,
                                          training_targets["median_house_value_is_high"
                                          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("LogLoss (on training data):")
  training_log_losses = []
```

```
    validation_log_losses = []
    for period in range (0, periods):
      # Train the model, starting from the prior state.
      linear_classifier.train(
          input_fn=training_input_fn,
          steps=steps_per_period
      )
      # Take a break and compute predictions.
      training_probabilities = linear_classifier.predict(input_fn=predict_training_input_
      training_probabilities = np.array([item['probabilities'] for item in training_proba

      validation_probabilities = linear_classifier.predict(input_fn=predict_validation_in
      validation_probabilities = np.array([item['probabilities'] for item in validation_p

      training_log_loss = metrics.log_loss(training_targets, training_probabilities)
      validation_log_loss = metrics.log_loss(validation_targets, validation_probabilities
      # Occasionally print the current loss.
      print("  period %02d : %0.2f" % (period, training_log_loss))
      # Add the loss metrics from this period to our list.
      training_log_losses.append(training_log_loss)
      validation_log_losses.append(validation_log_loss)
    print("Model training finished.")

    # Output a graph of loss metrics over periods.
    plt.ylabel("LogLoss")
    plt.xlabel("Periods")
    plt.title("LogLoss vs. Periods")
    plt.tight_layout()
    plt.plot(training_log_losses, label="training")
    plt.plot(validation_log_losses, label="validation")
    plt.legend()

    return linear_classifier
```

```
In [0]: linear_classifier = train_linear_classifier_model(
            learning_rate=0.000005,
            steps=500,
            batch_size=20,
            training_examples=training_examples,
            training_targets=training_targets,
            validation_examples=validation_examples,
            validation_targets=validation_targets)
```

1.5.1 Solution

Click below to see the solution.

```
In [0]: def train_linear_classifier_model(
            learning_rate,
```

```
  steps,
  batch_size,
  training_examples,
  training_targets,
  validation_examples,
  validation_targets):
"""Trains a linear classification 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 `LinearClassifier` object trained on the training data.
"""

periods = 10
steps_per_period = steps / periods

# Create a linear classifier object.
my_optimizer = tf.train.GradientDescentOptimizer(learning_rate=learning_rate)
my_optimizer = tf.contrib.estimator.clip_gradients_by_norm(my_optimizer, 5.0)
linear_classifier = tf.estimator.LinearClassifier(
    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_is_high"
                                        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("LogLoss (on training data):")
    training_log_losses = []
    validation_log_losses = []
    for period in range (0, periods):
      # Train the model, starting from the prior state.
      linear_classifier.train(
          input_fn=training_input_fn,
          steps=steps_per_period
      )
      # Take a break and compute predictions.
      training_probabilities = linear_classifier.predict(input_fn=predict_training_input_
      training_probabilities = np.array([item['probabilities'] for item in training_proba

      validation_probabilities = linear_classifier.predict(input_fn=predict_validation_in
      validation_probabilities = np.array([item['probabilities'] for item in validation_p

      training_log_loss = metrics.log_loss(training_targets, training_probabilities)
      validation_log_loss = metrics.log_loss(validation_targets, validation_probabilities
      # Occasionally print the current loss.
      print("  period %02d : %0.2f" % (period, training_log_loss))
      # Add the loss metrics from this period to our list.
      training_log_losses.append(training_log_loss)
      validation_log_losses.append(validation_log_loss)
    print("Model training finished.")

    # Output a graph of loss metrics over periods.
    plt.ylabel("LogLoss")
    plt.xlabel("Periods")
    plt.title("LogLoss vs. Periods")
    plt.tight_layout()
    plt.plot(training_log_losses, label="training")
    plt.plot(validation_log_losses, label="validation")
    plt.legend()

    return linear_classifier
```

In [0]:
```
linear_classifier = train_linear_classifier_model(
    learning_rate=0.000005,
    steps=500,
    batch_size=20,
```

```
        training_examples=training_examples,
        training_targets=training_targets,
        validation_examples=validation_examples,
        validation_targets=validation_targets)
```

1.6 Task 3: Calculate Accuracy and plot a ROC Curve for the Validation Set

A few of the metrics useful for classification are the model accuracy, the ROC curve and the area under the ROC curve (AUC). We'll examine these metrics.

`LinearClassifier.evaluate` calculates useful metrics like accuracy and AUC.

```
In [0]: predict_validation_input_fn = lambda: my_input_fn(validation_examples,
                                                          validation_targets["median_house_valu
                                                          num_epochs=1,
                                                          shuffle=False)

        evaluation_metrics = linear_classifier.evaluate(input_fn=predict_validation_input_fn)

        print("AUC on the validation set: %0.2f" % evaluation_metrics['auc'])
        print("Accuracy on the validation set: %0.2f" % evaluation_metrics['accuracy'])
```

You may use class probabilities, such as those calculated by `LinearClassifier.predict`, and Sklearn's roc_curve to obtain the true positive and false positive rates needed to plot a ROC curve.

```
In [0]: validation_probabilities = linear_classifier.predict(input_fn=predict_validation_input_
        # Get just the probabilities for the positive class.
        validation_probabilities = np.array([item['probabilities'][1] for item in validation_p

        false_positive_rate, true_positive_rate, thresholds = metrics.roc_curve(
            validation_targets, validation_probabilities)
        plt.plot(false_positive_rate, true_positive_rate, label="our model")
        plt.plot([0, 1], [0, 1], label="random classifier")
        _ = plt.legend(loc=2)
```

See if you can tune the learning settings of the model trained at Task 2 to improve AUC.

Often times, certain metrics improve at the detriment of others, and you'll need to find the settings that achieve a good compromise.

Verify if all metrics improve at the same time.

```
In [0]: # TUNE THE SETTINGS BELOW TO IMPROVE AUC
        linear_classifier = train_linear_classifier_model(
            learning_rate=0.000005,
            steps=500,
            batch_size=20,
            training_examples=training_examples,
            training_targets=training_targets,
            validation_examples=validation_examples,
            validation_targets=validation_targets)
```

```
evaluation_metrics = linear_classifier.evaluate(input_fn=predict_validation_input_fn)

print("AUC on the validation set: %0.2f" % evaluation_metrics['auc'])
print("Accuracy on the validation set: %0.2f" % evaluation_metrics['accuracy'])
```

1.6.1 Solution

Click below for a possible solution.

One possible solution that works is to just train for longer, as long as we don't overfit.

We can do this by increasing the number the steps, the batch size, or both.

All metrics improve at the same time, so our loss metric is a good proxy for both AUC and accuracy.

Notice how it takes many, many more iterations just to squeeze a few more units of AUC. This commonly happens. But often even this small gain is worth the costs.

```
In [0]: linear_classifier = train_linear_classifier_model(
            learning_rate=0.000003,
            steps=20000,
            batch_size=500,
            training_examples=training_examples,
            training_targets=training_targets,
            validation_examples=validation_examples,
            validation_targets=validation_targets)

        evaluation_metrics = linear_classifier.evaluate(input_fn=predict_validation_input_fn)

        print("AUC on the validation set: %0.2f" % evaluation_metrics['auc'])
        print("Accuracy on the validation set: %0.2f" % evaluation_metrics['accuracy'])
```