Linear_Regression_with_a_Real_Dataset

August 23, 2021

[ ]:
```
!pip install jupyterthemes
from jupyterthemes import jtplot
```

[ ]:
```
!jt -t gruvboxd
jtplot.style()
```

[ ]:

1 Linear Regression with a Real Dataset

This Colab uses a real dataset to predict the prices of houses in California.

1.1 Learning Objectives:

After doing this Colab, you'll know how to do the following:

- Read a .csv file into a pandas DataFrame.
- Examine a dataset.
- Experiment with different features in building a model.
- Tune the model's hyperparameters.

1.2 The Dataset

The dataset for this exercise is based on 1990 census data from California. The dataset is old but still provides a great opportunity to learn about machine learning programming.

1.3 Use the right version of TensorFlow

The following hidden code cell ensures that the Colab will run on TensorFlow 2.X.

```
[ ]: #@title Run on TensorFlow 2.x
     %tensorflow_version 2.x
```

1.4 Import relevant modules

The following hidden code cell imports the necessary code to run the code in the rest of this Colaboratory.

```
[ ]: #@title Import relevant modules
     import pandas as pd
     import tensorflow as tf
     from matplotlib import pyplot as plt

     # The following lines adjust the granularity of reporting.
     pd.options.display.max_rows = 10
     pd.options.display.float_format = "{:.1f}".format
```

1.5 The dataset

Datasets are often stored on disk or at a URL in .csv format.

A well-formed .csv file contains column names in the first row, followed by many rows of data. A comma divides each value in each row. For example, here are the first five rows of the .csv file file holding the California Housing Dataset:

```
"longitude","latitude","housing_median_age","total_rooms","total_bedrooms","population","housel
-114.310000,34.190000,15.000000,5612.000000,1283.000000,1015.000000,472.000000,1.493600,66900.(
-114.470000,34.400000,19.000000,7650.000000,1901.000000,1129.000000,463.000000,1.820000,80100.(
-114.560000,33.690000,17.000000,720.000000,174.000000,333.000000,117.000000,1.650900,85700.000
-114.570000,33.640000,14.000000,1501.000000,337.000000,515.000000,226.000000,3.191700,73400.00
```

1.5.1 Load the .csv file into a pandas DataFrame

This Colab, like many machine learning programs, gathers the .csv file and stores the data in memory as a pandas Dataframe. pandas is an open source Python library. The primary datatype in pandas is a DataFrame. You can imagine a pandas DataFrame as a spreadsheet in which each row is identified by a number and each column by a name. pandas is itself built on another open source Python library called NumPy. If you aren't familiar with these technologies, please view these two quick tutorials:

- NumPy
- Pandas DataFrames

The following code cell imports the .csv file into a pandas DataFrame and scales the values in the label (`median_house_value`):

[ ]:
```
# Import the dataset.
training_df = pd.read_csv(filepath_or_buffer="https://download.mlcc.google.com/
↪mledu-datasets/california_housing_train.csv")

# Scale the label.
training_df["median_house_value"] /= 1000.0

# Print the first rows of the pandas DataFrame.
training_df.head()
```

Scaling `median_house_value` puts the value of each house in units of thousands. Scaling will keep loss values and learning rates in a friendlier range.

Although scaling a label is usually *not* essential, scaling features in a multi-feature model usually *is* essential.

1.6 Examine the dataset

A large part of most machine learning projects is getting to know your data. The pandas API provides a `describe` function that outputs the following statistics about every column in the DataFrame:

- `count`, which is the number of rows in that column. Ideally, `count` contains the same value for every column.
- `mean` and `std`, which contain the mean and standard deviation of the values in each column.
- `min` and `max`, which contain the lowest and highest values in each column.
- `25%`, `50%`, `75%`, which contain various quantiles.

[ ]:
```
# Get statistics on the dataset.
training_df.describe()
```

1.6.1 Task 1: Identify anomalies in the dataset

Do you see any anomalies (strange values) in the data?

[ ]:
```
#@title Double-click to view a possible answer.

# The maximum value (max) of several columns seems very
# high compared to the other quantiles. For example,
# example the total_rooms column. Given the quantile
# values (25%, 50%, and 75%), you might expect the
# max value of total_rooms to be approximately
# 5,000 or possibly 10,000. However, the max value
# is actually 37,937.

# When you see anomalies in a column, become more careful
# about using that column as a feature. That said,
# anomalies in potential features sometimes mirror
# anomalies in the label, which could make the column
# be (or seem to be) a powerful feature.
# Also, as you will see later in the course, you
# might be able to represent (pre-process) raw data
# in order to make columns into useful features.
```

1.7 Define functions that build and train a model

The following code defines two functions:

- `build_model(my_learning_rate)`, which builds a randomly-initialized model.
- `train_model(model, feature, label, epochs)`, which trains the model from the examples (feature and label) you pass.

Since you don't need to understand model building code right now, we've hidden this code cell. You may optionally double-click the following headline to see the code that builds and trains a model.

[ ]:
```
#@title Define the functions that build and train a model
def build_model(my_learning_rate):
  """Create and compile a simple linear regression model."""
  # Most simple tf.keras models are sequential.
  model = tf.keras.models.Sequential()

  # Describe the topography of the model.
  # The topography of a simple linear regression model
  # is a single node in a single layer.
  model.add(tf.keras.layers.Dense(units=1,
                                  input_shape=(1,)))

  # Compile the model topography into code that TensorFlow can efficiently
  # execute. Configure training to minimize the model's mean squared error.
  model.compile(optimizer=tf.keras.optimizers.
↪RMSprop(learning_rate=my_learning_rate),
                loss="mean_squared_error",
                metrics=[tf.keras.metrics.RootMeanSquaredError()])

  return model


def train_model(model, df, feature, label, epochs, batch_size):
  """Train the model by feeding it data."""

  # Feed the model the feature and the label.
  # The model will train for the specified number of epochs.
  history = model.fit(x=df[feature],
                      y=df[label],
                      batch_size=batch_size,
                      epochs=epochs)

  # Gather the trained model's weight and bias.
  trained_weight = model.get_weights()[0]
  trained_bias = model.get_weights()[1]

  # The list of epochs is stored separately from the rest of history.
  epochs = history.epoch

  # Isolate the error for each epoch.
  hist = pd.DataFrame(history.history)

  # To track the progression of training, we're going to take a snapshot
  # of the model's root mean squared error at each epoch.
  rmse = hist["root_mean_squared_error"]

  return trained_weight, trained_bias, epochs, rmse

print("Defined the create_model and traing_model functions.")
```

1.8 Define plotting functions

The following matplotlib functions create the following plots:

- a scatter plot of the feature vs. the label, and a line showing the output of the trained model
- a loss curve

You may optionally double-click the headline to see the matplotlib code, but note that writing matplotlib code is not an important part of learning ML programming.

[ ]:
```
#@title Define the plotting functions
def plot_the_model(trained_weight, trained_bias, feature, label):
  """Plot the trained model against 200 random training examples."""

  # Label the axes.
  plt.xlabel(feature)
  plt.ylabel(label)

  # Create a scatter plot from 200 random points of the dataset.
  random_examples = training_df.sample(n=200)
  plt.scatter(random_examples[feature], random_examples[label])

  # Create a red line representing the model. The red line starts
  # at coordinates (x0, y0) and ends at coordinates (x1, y1).
  x0 = 0
  y0 = trained_bias
  x1 = 10000
  y1 = trained_bias + (trained_weight * x1)
  plt.plot([x0, x1], [y0, y1], c='r')

  # Render the scatter plot and the red line.
  plt.show()


def plot_the_loss_curve(epochs, rmse):
  """Plot a curve of loss vs. epoch."""

  plt.figure()
  plt.xlabel("Epoch")
  plt.ylabel("Root Mean Squared Error")

  plt.plot(epochs, rmse, label="Loss")
  plt.legend()
  plt.ylim([rmse.min()*0.97, rmse.max()])
  plt.show()

print("Defined the plot_the_model and plot_the_loss_curve functions.")
```

1.9 Call the model functions

An important part of machine learning is determining which features correlate with the label. For example, real-life home-value prediction models typically rely on hundreds of features and synthetic features. However, this model relies on only one feature. For now, you'll arbitrarily use `total_rooms` as that feature.

[ ]:
```
# The following variables are the hyperparameters.
learning_rate = 0.01
epochs = 30
batch_size = 30

# Specify the feature and the label.
my_feature = "total_rooms"  # the total number of rooms on a specific city␣
 ↪block.
my_label="median_house_value" # the median value of a house on a specific city␣
 ↪block.
# That is, you're going to create a model that predicts house value based
# solely on total_rooms.

# Discard any pre-existing version of the model.
my_model = None

# Invoke the functions.
my_model = build_model(learning_rate)
weight, bias, epochs, rmse = train_model(my_model, training_df,
                                         my_feature, my_label,
                                                    epochs, batch_size)

print("\nThe learned weight for your model is %.4f" % weight)
print("The learned bias for your model is %.4f\n" % bias )

plot_the_model(weight, bias, my_feature, my_label)
plot_the_loss_curve(epochs, rmse)
```

A certain amount of randomness plays into training a model. Consequently, you'll get different results each time you train the model. That said, given the dataset and the hyperparameters, the trained model will generally do a poor job describing the feature's relation to the label.

1.10 Use the model to make predictions

You can use the trained model to make predictions. In practice, you should make predictions on examples that are not used in training. However, for this exercise, you'll just work with a subset of the same training dataset. A later Colab exercise will explore ways to make predictions on examples not used in training.

First, run the following code to define the house prediction function:

```
[ ]: def predict_house_values(n, feature, label):
       """Predict house values based on a feature."""

       batch = training_df[feature][10000:10000 + n]
       predicted_values = my_model.predict_on_batch(x=batch)

       print("feature   label          predicted")
       print("  value   value          value")
       print("          in thousand$   in thousand$")
       print("--------------------------------------")
       for i in range(n):
         print ("%5.0f %6.0f %15.0f" % (training_df[feature][10000 + i],
                                        training_df[label][10000 + i],
                                        predicted_values[i][0] ))
```

Now, invoke the house prediction function on 10 examples:

```
[ ]: predict_house_values(10, my_feature, my_label)
```

1.10.1 Task 2: Judge the predictive power of the model

Look at the preceding table. How close is the predicted value to the label value? In other words, does your model accurately predict house values?

```
[ ]: #@title Double-click to view the answer.

     # Most of the predicted values differ significantly
     # from the label value, so the trained model probably
     # doesn't have much predictive power. However, the
     # first 10 examples might not be representative of
     # the rest of the examples.
```

1.11 Task 3: Try a different feature

The `total_rooms` feature had only a little predictive power. Would a different feature have greater predictive power? Try using `population` as the feature instead of `total_rooms`.

Note: When you change features, you might also need to change the hyperparameters.

[ ]:
```
my_feature = "population"   # Replace the ? with population or possibly
                   # a different column name.

# Experiment with the hyperparameters.
learning_rate = 0.05
epochs = 20
batch_size = 120

# Don't change anything below this line.
my_model = build_model(learning_rate)
weight, bias, epochs, rmse = train_model(my_model, training_df,
                                         my_feature, my_label,
                                         epochs, batch_size)
plot_the_model(weight, bias, my_feature, my_label)
plot_the_loss_curve(epochs, rmse)

predict_house_values(15, my_feature, my_label)
```

[ ]:
```
#@title Double-click to view a possible solution.

my_feature = "population" # Pick a feature other than "total_rooms"

# Possibly, experiment with the hyperparameters.
learning_rate = 0.05
epochs = 18
batch_size = 3

# Don't change anything below.
my_model = build_model(learning_rate)
weight, bias, epochs, rmse = train_model(my_model, training_df,
                                         my_feature, my_label,
                                         epochs, batch_size)

plot_the_model(weight, bias, my_feature, my_label)
plot_the_loss_curve(epochs, rmse)

predict_house_values(10, my_feature, my_label)
```

Did `population` produce better predictions than `total_rooms`?

[ ]:
```
#@title Double-click to view the answer.

# Training is not entirely deterministic, but population
# typically converges at a slightly higher RMSE than
# total_rooms.  So, population appears to be about
# the same or slightly worse at making predictions
# than total_rooms.
```

1.12 Task 4: Define a synthetic feature

You have determined that `total_rooms` and `population` were not useful features. That is, neither the total number of rooms in a neighborhood nor the neighborhood's population successfully predicted the median house price of that neighborhood. Perhaps though, the *ratio* of `total_rooms` to `population` might have some predictive power. That is, perhaps block density relates to median house value.

To explore this hypothesis, do the following:

1. Create a synthetic feature that's a ratio of `total_rooms` to `population`. (If you are new to pandas DataFrames, please study the Pandas DataFrame Ultraquick Tutorial.)
2. Tune the three hyperparameters.
3. Determine whether this synthetic feature produces a lower loss value than any of the single features you tried earlier in this exercise.

[ ]:
```
# Define a synthetic feature named rooms_per_person
training_df["rooms_per_person"] = training_df["total_rooms"] /␣
 ↪training_df["population"] # write your code here.

# Don't change the next line.
my_feature = "rooms_per_person"

# Assign values to these three hyperparameters.
learning_rate = 0.2
epochs = 30
batch_size = 64

# Don't change anything below this line.
my_model = build_model(learning_rate)
weight, bias, epochs, rmse = train_model(my_model, training_df,
                                         my_feature, my_label,
                                         epochs, batch_size)

plot_the_loss_curve(epochs, rmse)
predict_house_values(15, my_feature, my_label)
```

[ ]:
```
#@title Double-click to view a possible solution to Task 4.
```

```
# Define a synthetic feature
training_df["rooms_per_person"] = training_df["total_rooms"] /␣
 ↪training_df["population"]
my_feature = "rooms_per_person"

# Tune the hyperparameters.
learning_rate = 0.06
epochs = 24
batch_size = 30

# Don't change anything below this line.
my_model = build_model(learning_rate)
weight, bias, epochs, mae = train_model(my_model, training_df,
                                        my_feature, my_label,
                                        epochs, batch_size)

plot_the_loss_curve(epochs, mae)
predict_house_values(15, my_feature, my_label)
```

Based on the loss values, this synthetic feature produces a better model than the individual features you tried in Task 2 and Task 3. However, the model still isn't creating great predictions.

1.13 Task 5. Find feature(s) whose raw values correlate with the label

So far, we've relied on trial-and-error to identify possible features for the model. Let's rely on statistics instead.

A **correlation matrix** indicates how each attribute's raw values relate to the other attributes' raw values. Correlation values have the following meanings:

- `1.0`: perfect positive correlation; that is, when one attribute rises, the other attribute rises.
- `-1.0`: perfect negative correlation; that is, when one attribute rises, the other attribute falls.
- `0.0`: no correlation; the two columns are not linearly related.

In general, the higher the absolute value of a correlation value, the greater its predictive power. For example, a correlation value of -0.8 implies far more predictive power than a correlation of -0.2.

The following code cell generates the correlation matrix for attributes of the California Housing Dataset:

```
[ ]: # Generate a correlation matrix.
     training_df.corr()
```

The correlation matrix shows nine potential features (including a synthetic feature) and one label (`median_house_value`). A strong negative correlation or strong positive correlation with the label suggests a potentially good feature.

Your Task: Determine which of the nine potential features appears to be the best candidate for a feature?

```
[ ]: #@title Double-click here for the solution to Task 5

     # The `median_income` correlates 0.7 with the label
     # (median_house_value), so median_income` might be a
     # good feature. The other seven potential features
     # all have a correlation relatively close to 0.

     # If time permits, try median_income as the feature
     # and see whether the model improves.
```

Correlation matrices don't tell the entire story. In later exercises, you'll find additional ways to unlock predictive power from potential features.

Note: Using `median_income` as a feature may raise some ethical and fairness issues. Towards the end of the course, we'll explore ethical and fairness issues.