CS 395
Homework 8 – Regression and Clustering
27 Points Total
Due in Canvas by 11:59 PM on Sunday, April 21, 2019

To date, most of our work has been examining problems with classification. In this final lab, we will look at two others: regression and clustering.

Regression

In a regression problem, we aim to predict the output of a continuous value, like a price or a probability. Contrast this with a classification problem, where we aim to select a class from a list of classes (for example, where a picture contains an apple or an orange, recognizing which fruit is in the picture).

It is helpful in making a prediction of a number based on other inputs. We explored this in the example of setting a price for a house based on other features (number of bedrooms, number of bathrooms, square footages, etc.).

Data Preparation

This lab uses the classic Auto MPG Dataset and builds a model to predict the fuel efficiency of late-1970s and early 1980s automobiles. To do this, we'll provide the model with a description of many automobiles from that time period. This description includes attributes like: cylinders, displacement, horsepower, and weight.

```
from __future__ import absolute_import, division, print_function

import pathlib
import matplotlib.pyplot as plt
import pandas as pd
import seaborn as sns
import tensorflow as tf
from tensorflow import keras
from tensorflow.keras import layers
```

The Auto MPG dataset is available from the UCI Machine Learning Repository. First we need to download the dataset.

```
dataset_path = keras.utils.get_file("auto-mpg.data",
"https://archive.ics.uci.edu/ml/machine-learning-databases/auto-mpg/auto-
mpg.data")
dataset_path
```

Import it using pandas

```
column_names = ['MPG','Cylinders','Displacement','Horsepower','Weight',
                'Acceleration', 'Model Year', 'Origin']
raw_dataset = pd.read_csv(dataset_path, names=column_names,
                      na_values = "?", comment='\t',
                      sep=" ", skipinitialspace=True)

dataset = raw_dataset.copy()
dataset.tail()
```

Clean the data

The dataset contains a few unknown values.

```
dataset.isna().sum()
```

```
MPG             0
Cylinders       0
Displacement    0
Horsepower      6
Weight          0
Acceleration    0
Model Year      0
Origin          0
dtype: int64
```

To keep this initial tutorial simple drop those rows.

```
dataset = dataset.dropna()
```

The "Origin" column is really categorical, not numeric. So convert that to a one-hot:

```
origin = dataset.pop('Origin')

dataset['USA'] = (origin == 1)*1.0
dataset['Europe'] = (origin == 2)*1.0
dataset['Japan'] = (origin == 3)*1.0
dataset.tail()
```

1. Run all of the above code segments and show your code and all outputs (1 point)

Split the data into train and test

Now split the dataset into a training set and a test set. We will use the test set in the final evaluation of our model. Run the following.

```
train_dataset = dataset.sample(frac=0.8,random_state=0)
test_dataset = dataset.drop(train_dataset.index)
```

Inspect the data

Have a quick look at the joint distribution of a few pairs of columns from the training set.

```
sns.pairplot(train_dataset[["MPG", "Cylinders", "Displacement",
"Weight"]], diag_kind="kde")
```

2. Run the above code and show the output. (1 point).
3. What insight can you gain from this output. Explain. (2 points)

Now, let's look at the overall statistics:

```
train_stats = train_dataset.describe()
train_stats.pop("MPG")
train_stats = train_stats.transpose()
train_stats
```

4. Run the above. What observations can you make from this data? (1 point)

Split features from labels

Now we separate the target value, or "label", from the features. This label is the value that you will train the model to predict.

```
train_labels = train_dataset.pop('MPG')
test_labels = test_dataset.pop('MPG')
```

Normalize the data

Look again at the train_stats block above and note how different the ranges of each feature are.

As we explored in earlier labs, ilt is good practice to normalize features that use different scales and ranges. Although the model might converge without feature normalization, it makes training more difficult, and it makes the resulting model dependent on the choice of units used in the input.

Note: Although we intentionally generate these statistics from only the training dataset, these statistics will also be used to normalize the test dataset. We need to do that to project the test dataset into the same distribution that the model has been trained on.

```
def norm(x):
  return (x - train_stats['mean']) / train_stats['std']
normed_train_data = norm(train_dataset)
normed_test_data = norm(test_dataset)
```

This normalized data is what we will use to train the model.

Caution: The statistics used to normalize the inputs here (mean and standard deviation) need to be applied to any other data that is fed to the model, along with the one-hot encoding that we did earlier. That includes the test set as well as live data when the model is used in production.

Building the model

Let's build our model. Here, we'll use a Sequential model with two densely connected hidden layers, and an output layer that returns a single, continuous value. The model building steps are wrapped in a function, build_model, since we'll create a second model, later on.

```
def build_model():
  model = keras.Sequential([
    layers.Dense(64, activation=tf.nn.relu,
input_shape=[len(train_dataset.keys())]),
```

```
    layers.Dense(64, activation=tf.nn.relu),
    layers.Dense(1)
  ])

  optimizer = tf.keras.optimizers.RMSprop(0.001)

  model.compile(loss='mean_squared_error',
                optimizer=optimizer,
                metrics=['mean_absolute_error', 'mean_squared_error'])
  return model

model = build_model()
```

Inspect the model

Use the .summary method to print a simple description of the model

```
model.summary()
```

Now try out the model. Take a batch of 10 examples from the training data and call model.predict on it.

```
example_batch = normed_train_data[:10]
example_result = model.predict(example_batch)
example_result
```

5. Run all of the above code. Show all code and outputs. (1 point)

Train the model

Train the model for 1000 epochs, and record the training and validation accuracy in the history object.

```
# Display training progress by printing a single dot for each completed
epoch
class PrintDot(keras.callbacks.Callback):
  def on_epoch_end(self, epoch, logs):
    if epoch % 100 == 0: print('')
    print('.', end='')

EPOCHS = 1000

history = model.fit(
  normed_train_data, train_labels,
  epochs=EPOCHS, validation_split = 0.2, verbose=0,
  callbacks=[PrintDot()])
```

Visualize the model's training progress using the stats stored in the history object.

```
hist = pd.DataFrame(history.history)
hist['epoch'] = history.epoch
hist.tail()
```

6. Run all of the above code. Show all code and outputs. (1 point)

```
def plot_history(history):
  hist = pd.DataFrame(history.history)
  hist['epoch'] = history.epoch

  plt.figure()
  plt.xlabel('Epoch')
  plt.ylabel('Mean Abs Error [MPG]')
  plt.plot(hist['epoch'], hist['mean_absolute_error'],
           label='Train Error')
  plt.plot(hist['epoch'], hist['val_mean_absolute_error'],
           label = 'Val Error')
  plt.ylim([0,5])
  plt.legend()

  plt.figure()
  plt.xlabel('Epoch')
  plt.ylabel('Mean Square Error [$MPG^2$]')
  plt.plot(hist['epoch'], hist['mean_squared_error'],
           label='Train Error')
  plt.plot(hist['epoch'], hist['val_mean_squared_error'],
           label = 'Val Error')
  plt.ylim([0,20])
  plt.legend()
  plt.show()


plot_history(history)
```

7. Run all of the above code. Show all code and outputs. (1 point)

This graph shows little improvement, or even degradation in the validation error after about 100 epochs. Let's update the model.fit call to automatically stop training when the validation score doesn't improve. We'll use an EarlyStopping callback that tests a training condition for every epoch. If a set amount of epochs elapses without showing improvement, then automatically stop the training.

```
model = build_model()

# The patience parameter is the amount of epochs to check for improvement
early_stop = keras.callbacks.EarlyStopping(monitor='val_loss',
patience=10)

history = model.fit(normed_train_data, train_labels, epochs=EPOCHS,
                    validation_split = 0.2, verbose=0,
callbacks=[early_stop, PrintDot()])

plot_history(history)
```

When we run the above code, we should see a graph. The graph shows that on the validation set, the average error is usually around +/- 2 MPG. Is this good? We'll leave that decision up to you.

Let's see how well the model generalizes by using the test set, which we did not use when training the model. This tells us how well we can expect the model to predict when we use it in the real world.

```
loss, mae, mse = model.evaluate(normed_test_data, test_labels, verbose=0)

print("Testing set Mean Abs Error: {:5.2f} MPG".format(mae))
```

8. Run all of the above code. Show all code and outputs. (1 point)

Make predictions

Finally, predict MPG values using data in the testing set:

```
test_predictions = model.predict(normed_test_data).flatten()

plt.scatter(test_labels, test_predictions)
plt.xlabel('True Values [MPG]')
plt.ylabel('Predictions [MPG]')
plt.axis('equal')
plt.axis('square')
plt.xlim([0,plt.xlim()[1]])
plt.ylim([0,plt.ylim()[1]])
_ = plt.plot([-100, 100], [-100, 100])
```

Now let's look at the error distribution:

```
error = test_predictions - test_labels
plt.hist(error, bins = 25)
plt.xlabel("Prediction Error [MPG]")
_ = plt.ylabel("Count")
```

9. Run all of the above code. Show all code and outputs. (1 point)

It's not quite Gaussian, but we might expect that because the number of samples is very small.

Clustering

Now we will examine clustering. Cluster analysis or clustering is the task of grouping a set of objects in such a way that objects in the same group are more similar to each other than to those in other groups. You can read more about it here: https://www.geeksforgeeks.org/clustering-in-machine-learning/

At this point, it might be helpful to save the regression notebook and open a new clustering notebook.

First, let's generate random data points with a uniform distribution and assign them to a 2D tensor constant. Then, randomly choose initial centroids from the set of data points.

```
import matplotlib.pyplot as plt
import numpy as np
import tensorflow as tf

points_n = 200
```

```
clusters_n = 3
iteration_n = 100

points = tf.constant(np.random.uniform(0, 10, (points_n, 2)))
centroids = tf.Variable(tf.slice(tf.random_shuffle(points), [0, 0],
[clusters_n, -1]))
```

For the next step, we want to be able to do element-wise subtraction of points and centroids that are 2D tensors. As the tensors have different shape, let's expand points and centroids into 3 dimensions, which enables us to use the broadcasting feature of subtraction operation.

```
points_expanded = tf.expand_dims(points, 0)
centroids_expanded = tf.expand_dims(centroids, 1)
```

Then, calculate the distances between points and centroids and determine the cluster assignments.

```
distances = tf.reduce_sum(tf.square(tf.subtract(points_expanded,
centroids_expanded)), 2)
assignments = tf.argmin(distances, 0)
```

Next, we can compare each cluster with a cluster assignments vector, get points assigned to each cluster, and calculate mean values. These mean values are refined centroids, so let's update the centroids variable with the new values.

```
means = []
for c in range(clusters_n):
    means.append(tf.reduce_mean(
      tf.gather(points,
                tf.reshape(
                  tf.where(
                    tf.equal(assignments, c)
                  ),[1,-1])
               ),reduction_indices=[1]))

new_centroids = tf.concat(means, 0)

update_centroids = tf.assign(centroids, new_centroids)
```

It's time to run the graph. For each iteration, we update the centroids and return their values along with the cluster assignments values.

```
with tf.Session() as sess:
  sess.run(init)
  for step in range(iteration_n):
    [_, centroid_values, points_values, assignment_values] =
sess.run([update_centroids, centroids, points, assignments])
```

Lastly, we display the coordinates of the final centroids and a multi-colored scatter plot showing how the data points have been clustered.

```
plt.scatter(points_values[:, 0], points_values[:, 1],
c=assignment_values, s=50, alpha=0.5)
```

```
plt.plot(centroid_values[:, 0], centroid_values[:, 1], 'kx',
markersize=15)
plt.show()
```

The data in a training set is grouped into clusters as the result of implementing the k-means algorithm in TensorFlow.

Here is the complete code:

```
import matplotlib.pyplot as plt
import numpy as np
import tensorflow as tf

points_n = 200
clusters_n = 3
iteration_n = 100

points = tf.constant(np.random.uniform(0, 10, (points_n, 2)))
centroids = tf.Variable(tf.slice(tf.random_shuffle(points), [0, 0],
[clusters_n, -1]))

points_expanded = tf.expand_dims(points, 0)
centroids_expanded = tf.expand_dims(centroids, 1)

distances = tf.reduce_sum(tf.square(tf.subtract(points_expanded,
centroids_expanded)), 2)
assignments = tf.argmin(distances, 0)

means = []
for c in range(clusters_n):
    means.append(tf.reduce_mean(
      tf.gather(points,
                tf.reshape(
                  tf.where(
                    tf.equal(assignments, c)
                  ),[1,-1])
               ),reduction_indices=[1]))

new_centroids = tf.concat(means, 0)

update_centroids = tf.assign(centroids, new_centroids)
init = tf.global_variables_initializer()

with tf.Session() as sess:
  sess.run(init)
  for step in range(iteration_n):
    [_, centroid_values, points_values, assignment_values] =
sess.run([update_centroids, centroids, points, assignments])

  print("centroids" + "\n", centroid_values)

plt.scatter(points_values[:, 0], points_values[:, 1],
c=assignment_values, s=50, alpha=0.5)
plt.plot(centroid values[:, 0], centroid values[:, 1], 'kx',
markersize=15)
```

```
plt.show()
```

10. Run the above code, showing the code and your output (1 point).

11. Change the number of clusters from 3 to 4. Show your output. Is increasing the number of clusters a better result? Explain. (2 points)

12. We want to determine the amount of error. Write a script to calculate the amount of error (distance between the centroid and each point) with 3 clusters and with 4 clusters. Show the code and the output calculation (10 points)

13. Explain your observation of how the reduction of error and the change in the number of clusters affects the *informativeness* of the data. Is there a tradeoff? Explain and show output from your program to illustrate your point. You may need to run your clustering algorithm with different cluster sizes. (4 points).

Conclusion

- This lab introduced a few techniques to handle a regression problem.
- Mean Squared Error (MSE) is a common loss function used for regression problems (different loss functions are used for classification problems).
- Similarly, evaluation metrics used for regression differ from classification. A common regression metric is Mean Absolute Error (MAE).
- When numeric input data features have values with different ranges, each feature should be scaled independently to the same range.
- If there is not much training data, one technique is to prefer a small network with few hidden layers to avoid overfitting.
- Early stopping is a useful technique to prevent overfitting.
- With clustering, we are addressing a different kind of problem. No labels are given to the learning algorithm, leaving it on its own to find structure in its input. Unsupervised learning can be a goal in itself (discovering hidden patterns in data) or a means towards an end (feature learning).
- In some pattern recognition problems, the training data consists of a set of input vectors x without any corresponding target values. The goal in such unsupervised learning problems may be to discover groups of similar examples within the data, where it is called clustering, or to determine how the data is distributed in the space, known as density estimation.
- The number of clusters, which is a parameter we set in advance in k-means clustering (it is the k), affects our solution.