Activity 1.1 – Training a Regression Model Using Linear Regression

In this activity, we will learn:

- ❑ use scikit-learn to train a machine learning model to estimate the price of a house

Note: use aaip_intro_py37 virtual environment
Installation:
conda create –name aaip_intro_py37 python=3.7 pandas scikit-learn
pip install tensorflow (use version 2.0)

1) Examining the Training Data

See source file house_data.csv. let’s look at our training data. In house_data.csv, we have 10,000 rows of data that looks like this:

sq feet	num bedrooms	num bathrooms	sale price
785	2	2	170461
1477	2	2	271651
712	1	1	139912

Our job is to use the size (sq_feet), the number of bedrooms, and the number of bathrooms in each house to try to predict the house’s final sale price.

If you open up the file and skim through it, you’ll see that there are no missing values and no bad data. You won’t always be so lucky, though. It’s important to always check your data for problems and inconsistencies. Bad data is the most common source of problems in machine learning models.

2) Creating the model

a) The first step of our program is to import all the libraries we’ll be using:

```
import pandas as pd
from sklearn.model_selection import train_test_split
from sklearn.linear_model import LinearRegression
from sklearn.metrics import mean_absolute_error
from sklearn.externals import joblib
```

b) Next, we’ll use the pandas library to load the contents of the CSV into a table in memory. This will make the data easy to work with:

```
#Load our data set
df = pd.read_csv("house_data.csv")
```

c) Our training data has both the inputs (sq_feet, num_bedrooms, num_bathrooms) and the expected output (sale_price) in the same table. We need to split the data into two separate tables. By convention, machine learning programmers give these two tables special names.

- X is the table that has all the input features that the model will use to make predictions.
- y has the matching expected output for each row of training data in the X table.

d) We can create X and y by grabbing the columns we want out of the data we already loaded and saving them as separate tables:

```
# Create the X and y arrays
X = df[["sq_feet", "num_bedrooms", "num_bathrooms"]]
y = df["sale_price"]
```

e) Right now, X contains all of our house details and y contains all of the corresponding prices. But if we use all 50,000 house prices to train the model, we won’t have a good way to test the model to verify that it’s actually working.
The solution is to split the training data into two groups:
i) 75 percent of the data will be used to train the model. The model will get to see the y values for these houses. This is called the Training Data Set.
ii) 25 percent of the data will be kept secret and used to test the model. The model will only be allowed to see the X values and we’ll check its prediction for each one against the real y values. This is called the Test Data Set.
Getting this right is critical. You never want to test a model with data it saw during the training process. Otherwise, it can just memorize each answer without having to actually solve the problem.

f) Scikit-learn provides a function that splits the data into a training set and test set that we can use:

```
# Split the data set in a training set (75%) and a test set (25%)
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.25)
```

g) Now we are ready to create our machine learning model. Any model that uses a set of input values to predict an output number is called a regression model. We can choose from any machine learning regression algorithm that our libraries support to solve this problem.

Earlier, we talked about how linear regression works, so let’s try out that algorithm. To use it in scikit-learn, you just create a new LinearRegression object.

```
# Create the Linear Regression model
model = LinearRegression()
```

That's it! Now we are ready to train the model. You always do that by calling fit() on the model and passing in the X and y values to use for training:

h) Train the model

```
# Train the model
model.fit(X_train, y_train)
```

This will train the model to look at each row in the X_train table and predict the matching value in y_train array.

i) Next, we can save the trained model object to disk so that we can use it to make predictions later. In Python, the joblib library makes it easy to save objects to disk and re-load them later:

```
# Save the trained model to a file so we can use it to make predictions later
joblib.dump(model, 'house_value_model.pkl')
```

j) Finally, we can check the model to see how well it performs on the training data and the test data. Scikit-learn has a helpful function called mean_absolute_error that does this for us. It compares the real answers from the y array against the predictions from our model and figures out how wrong the model is, on average.

```
# Report how well the model is performing
print("Model training results:")

# Report an error rate on the training set
mse_train = mean_absolute_error(y_train, model.predict(X_train))
print(f" - Training Set Error: {mse_train}")

# Report an error rate on the test set
mse_test = mean_absolute_error(y_test, model.predict(X_test))
print(f" - Test Set Error: {mse_test}")
```

k) Run the code. You will get the following output.

```
Model training results:
 - Training Set Error: 9148.6731064472
 - Test Set Error: 8924.501184661423
```

This means that the model can predict the value of any house in the training data set with an average error of plus or minus $9,148. For the test set, it has nearly the same average error amount. That means things are working great!

3) Using the model

a) Our trained model is saved in a file called house_value_model.pkl. Let's use the model to make a value prediction for a new house! In a new Python file, let's use joblib to reload our model back into a variable:

```
from sklearn.externals import joblib

# Load our trained model
model = joblib.load('house_value_model.pkl')
```

b) Now we need to define the attributes of the house that we want to value. To do this, we just need to create an array with the values in the same order as they appeared in the training data file:

```
# Define the house that we want to value (with the values in the same order as in the training data)
house_1 = [
    2000,  # Size in Square Feet
    3,  # Number of Bedrooms
    2,  # Number of Bathrooms
]
```

c) Scikit-learn assumes we want to predict values for multiple houses at once, so it expects us to pass in an array of arrays. We only want to value a single house, so we'll just put that single house inside another array:

```
# scikit-learn assumes you want to predict the values for multiple of houses at once, so it expects an
array.
# We only want to estimate the value of a single house, so there will only be one item in our array.
homes = [
    house_1
]
```

d) Making a prediction is as easy as calling the predict() function on our model and passing in the new house details.

```
# Make a prediction for each house in the homes array (we only have one)
home_values = model.predict(homes)

# Since we are only predicting the price of one house, grab the first prediction returned
predicted_value = home_values[0]
```

e) And let's print out the results.

```
# Print the results
print("House details:")
print(f"- {house_1[0]} sq feet")
print(f"- {house_1[1]} bedrooms")
print(f"- {house_1[2]} bathrooms")
print(f"Estimated value: ${predicted_value:,.2f}")
```

f) When you run it, you should get similar results as below. But there is some randomness involved in training this kind of model so the numbers won't be exactly the same:

```
House details:
- 2000 sq feet
- 3 bedrooms
- 2 bathrooms
Estimated value: $383,765.87
```

The model thinks this house is worth about $383,765. Try changing the house's attributes and running it again to see how it affects the value of the house.

Activity wrap-up:
We learn how to
❑ use scikit-learn to train a machine learning model to estimate the price of a house

Activity 1.2 – Training a Regression Model Using Neural Network

In this activity, we will learn:
❑ use tensorflow keras to train and use a neural network model to estimate the price of a house

1) Training the Model

a) First, we need to import all the libraries that we need. We'll use several of the same scikit-learn helper functions, but we'll also pull in a bunch of Keras classes. We'll use Keras to define the neural network.

```
import pandas as pd
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import *
from sklearn.model_selection import train_test_split
from sklearn.preprocessing import MinMaxScaler
from sklearn.externals import joblib
from sklearn.metrics import mean_absolute_error
```

b) Next, we'll load our dataset and split it into X and y values using pandas exactly the same way we did in the last project.

```
# Load our data set
df = pd.read_csv("house_data.csv")

# Create the X and y arrays
X = df[["sq_feet", "num_bedrooms", "num_bathrooms"]]
y = df[["sale_price"]]
```

c) But here's where things start getting more complicated with neural networks. All of our input and output data needs to be scaled to values between 0 to 1 for the neural network to train correctly. To do that, we will create two MinMaxScaler classes and tell them to scale values between 0 and 1:

```
# Data needs to be scaled to  0 to 1 for the neural network to train correctly.
X_scaler = MinMaxScaler(feature_range=(0, 1))
y_scaler = MinMaxScaler(feature_range=(0, 1))

# Scale both the training inputs and outputs
X[X.columns] = X_scaler.fit_transform(X[X.columns])
y[y.columns] = y_scaler.fit_transform(y[y.columns])
```

The reason we need to create a separate scaler for the input values and the output values is that we'll want to be able to use them separately when we make real predictions with the model.

d) We also need to convert the X and y arrays from pandas dataframes into numpy arrays. Newer versions of pandas provide a to_numpy() function on dataframes to make this easy.

```
# Convert the inputs and outputs to numpy format so TensorFlow doesn't complain
X = X.to_numpy()
y = y.to_numpy()
```

This conversion doesn't change the values of the data at all. The only reason we need to convert the datatype is because Keras runs on top of TensorFlow and TensorFlow 2.0 doesn't support pandas dataframes directly. It's a work-around for a capability issue.

e) Next, we'll split the data into a training set and a test set. It's important to do this after we scale the data to make sure that the training and test sets are scaled exactly the same way.

```
# Split the data set in a training set (75%) and a test set (25%)
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.25)
```

f) Now we are ready to define our neural network. Keras has different APIs for defining neural networks that we can choose between. The most commonly used is the Sequential API. The Sequential API is the version in which we define it one layer at a time, in sequence. It's the easiest way to do it, so that's what we'll use too. So, first we create a new Sequential() model.

```
# Create a Neural Network model
model = Sequential()
```

g) This is the model object to which we'll add our layers. Now we can de- fine the layers of our neural network calling the model.add() function and passing in the kind of layer we want to add.
So far, we've only learned about the simplest kind of neural network layer where every node in the layer is connected to every node in the following layer. These are called dense layers because the nodes are densely

connected.
To create a dense layer in Keras, we can create new Dense() objects and add them to the model. Here's how we'll define the first layer:

```
model.add(Dense(50, input_dim=3, activation='relu'))
```

This single line actually will create two layers—a three-node input layer and a 50-node hidden layer. This is because the first layer in Keras has a special input_shape parameter. This tells Keras that we want three input nodes in the neural network. Instead of declaring the input layer on its own line, this is how Keras specifies the size of the input layer. The 50 parameter tells Keras to create the dense layer with 50 nodes. Earlier, we talked about hidden layers with four nodes. But in real life, we'll often use hundreds of nodes in a single layer.
Finally, each layer needs to have an activation function. This controls how much signal passes from each node in this layer to nodes in the next layer. Here are the two activation functions that we'll use:

- ***relu*** is a rectified linear unit. This activation function is non-linear but it has some nice mathematical properties that make it train more quickly than other activation functions. This is what you'll see used most often these days.
- ***linear*** is the same thing as having no activation function for the layer and letting the value for each node pass to the next layer unchanged.

h) Next, let's add two more hidden layers, one with 100 nodes and one with 50 nodes:

```
model.add(Dense(100, activation='relu'))
model.add(Dense(50, activation='relu'))
```

As usual, these hidden layers use a relu activation function.

i) Now let's add the output layer:

```
model.add(Dense(1, activation='linear'))
```

Since we are only predicting one final value (the value of a house), we only need one output node. And since we are predicting a real-world value, we'll ask it to use a linear activation function, which is the same thing as doing nothing. This is because we want the final prediction to be a linear number so we don't need to do anything special to it.

j) We are using the Keras API to define our neural network, but it uses TensorFlow behind the scenes to do all the math. Now that we've declared all the layers, we need to tell Keras to construct the neural network inside of TensorFlow for us. To do that, we'll call the compile() function:

```
model.compile(
    loss='mean_squared_error',
    optimizer='SGD'
)
```

There are two important parameters that we need to specify:

- **loss** is the loss or cost function we are using to measure how wrong our neural network currently is. The Keras documentation lists out all the possible loss functions that you can choose from. Since we are predicting continuous values, mean squared error is a good choice.
- **optimizer** is which numerical optimization algorithm we will use to train the neural network. We will use **stochastic gradient descent**, or SGD for short. This is the exact same thing as the normal gradient descent algorithm we talked about, except that it works with batches of training data instead of processing all the training data at once.

k) Now we're ready to train the model. Keras models its syntax on scikit-learn, so its training function is also called fit():

```
# Train the model
model.fit(
    X_train,
    y_train,
    epochs=50,
    batch_size=8,
    shuffle=True,
    verbose=2
)
```

First, we pass in the training data and the matching answers for each training example. Then we have several parameters that we can control:

- **epochs** is how many times we will loop through the entire training dataset before ending the gradient descent training process.
- **batch_size** controls how many training examples are considered at once during each gradient descent update pass.
- **shuffle=True** tells Keras to randomize the order of the input data it sees. This is super important. You always want to randomize the order of your training data unless you know for sure that you have it stored in random order already.
- **verbose** just controls how much Keras prints on the screen during the training process. Setting it to 2 makes it print less output.

l) Then we need to save both scalers to a file since we'll need them again later to make predictions.

```
# Save the scalers to files so we can use it to pre-process new data later
joblib.dump(X_scaler, "X_scaler.pkl")
joblib.dump(y_scaler, "y_scaler.pkl")
```

m) Then we'll save the trained neural network itself to a file so that we can use it to make predictions later.

```
# Save the trained model to a file so we can use it to make predictions later
model.save("house_value_model.h5")
```

n) And just like when we used the linear regression model, we'll print out training accuracy results. The only difference here is we'll use the y_scaler to unscale the accuracy numbers so that the results are in dollar amounts instead of a number between 0 and 1:

```
# Report how well the model is performing
print("Model training results:")

# Report mean absolute error on the training set in a value scaled back to dollars so it's easier to
#  understand.
predictions_train = model.predict(X_train, verbose=0)

mse_train = mean_absolute_error(
    y_scaler.inverse_transform(predictions_train),
    y_scaler.inverse_transform(y_train)
)
print(f" - Training Set Error: {mse_train}")
```

o) That covers the training data set results. We'll do the same for the test data set.

```
# Report mean absolute error on the test set in a value scaled back to dollars so it's easier to
# understand.
predictions_test = model.predict(X_test, verbose=0)

mse_test = mean_absolute_error(
    y_scaler.inverse_transform(predictions_test),
    y_scaler.inverse_transform(y_test)
)
print(f" - Test Set Error: {mse_test}")
```

p) Run the code.

```
….
Epoch 47/50
7500/7500 - 3s - loss: 3.2279e-04
Epoch 48/50
7500/7500 - 1s - loss: 3.2239e-04
Epoch 49/50
7500/7500 - 2s - loss: 3.2327e-04
Epoch 50/50
7500/7500 - 2s - loss: 3.2289e-04
Model training results:
 - Training Set Error: 9197.581994791668
 - Test Set Error: 9307.667750000002
```

In the end, we got nearly the same results as we got using the linear regression model, but the training process took a lot longer. This is because neural networks are much more complex mathematically than linear regression models so they take more computational power to train.
This data set is really simple, so we didn't really need the extra modelling power that the neural network provided to solve it. But in lots of cases with real-world data, a linear regression model won't work at all because the data is too complex. In that case, the extra processing time required for a neural network would be worth it.

Also, notice that the training set and test set errors are different. Models nearly always do worse on the test data than the training data. That's because the model can cheat and memorize answers in the training dataset, but it can't do that with the test data since it never sees those answers.
That means that Test Set Error is a much better predictor of real-world performance than Training Set Error. But the distance between the two numbers also gives us tons of clues of how well your model is working:

- If the Test Set Error is reasonably low and only a little higher than your Training Set Error, then everything is working correctly!
- If your Training Set Error and Test Set Error are both really high, the model is failing to learn at all. This is called **underfitting**.
- If your Training Set Error is low but your Test Set Error is high, the model is memorizing the training data and not really learning anything. This is called **overfitting**.

Overfitting and underfitting are the bane of machine learning. Training a model is a balancing act where you are trying to find the best spot between those two conditions.
There are several strategies you can use to fix underfitting and overfitting.

Strategies for fixing underfitting:

- Make sure the output value that you want to predict can reasonably be determined by the training inputs. Machine learning isn't magic!
- Try running the training process longer.

- Increase model complexity by adding layers or adding nodes to layers.
- Try a different machine learning algorithm that is better suited to the problem.

Strategies for fixing overfitting:

- Reduce the model complexity by removing layers or reducing the number of nodes in layers.
- Use regularization techniques like dropout.
- Increase the amount of training data.

It's important to be systematic when evaluating a model. Following these guidelines will save you a lot of time and money. For example, there's absolutely no reason to collect more training data if your model is underfitting. In that case, no amount of extra data will help! Instead, make sure your model can overfit the training data you already have before adding more data.

2) Using the Model

a) Now that we've trained the neural network, let's use it to make new predictions from a new Python program. First, we need to import some libraries so we can load the model file and the data scalers that we created earlier.

```
from tensorflow.keras.models import load_model
from sklearn.externals import joblib
```

b) Load the model that we just trained.

```
# Load our trained model
model = load_model('house_value_model.h5')
```

c) Load the data scalers.

```
# Load the data scalers so that we can transform new data and prediction the same way as training
data.
X_scaler = joblib.load('X_scaler.pkl')
y_scaler = joblib.load('y_scaler.pkl')
```

d) Now let's specify the house that we want to value.

```
Define the house that we want to value (with the values in the same order as in the training data).
house_1 = [
    2000,  # Size in Square Feet
    3,  # Number of Bedrooms
    2,  # Number of Bathrooms
]
```

e) Keras assumes we want to make multiple price predictions at once, so we need to put our single house inside an array.

```
# Keras assumes we want to predict the values for multiple of houses at once, so it expects an array.
# We only want to value a single house, so it will be the only item in our array.
homes = [
    house_1
]
```

f) But we need to remember that the house we just defined needs to have those values scaled to values between 0 and 1 so the model will understand the values. We can do that using the X_scaler we created during the training process.

```
# Scale the new data like the training data
scaled_home_data = X_scaler.transform(homes)
```

g) Prediction in Keras has nearly the same syntax as in scikit-learn. Just call model.predict() and pass in the data for the house.

```
# Make a prediction for each house in the homes array (but we only have one)
home_values = model.predict(scaled_home_data)
```

h) The prediction we get back from the neural network will be a value between 0 and 1. To turn it back into a dollar amount, we need to undo the scaling. The y_scaler we created during training can do that with its inverse_transform() function.

```
# The prediction from the neural network will be scaled 0 to 1 just like the training data
# We need to unscale it using the same factor as we used to scale the training data
unscaled_home_values = y_scaler.inverse_transform(home_values)
```

i) Now we can just print out the results.

```
# Since we are only predicting the price of one house, grab the first prediction returned
predicted_value = unscaled_home_values[0][0]

# Print the results
print("House details:")
print(f"- {house_1[0]} sq feet")
print(f"- {house_1[1]} bedrooms")
print(f"- {house_1[2]} bathrooms")
print(f"Estimated value: ${predicted_value:,.2f}")
```

j) Run the code. You should get the following output.

```
House details:
- 2000 sq feet
- 3 bedrooms
- 2 bathrooms
```

```
05  Estimated value: $384,112.53
```

Try playing around with the characteristics of the house and seeing what values it will predict. You can also compare the neural network's predictions against the linear regression model we made earlier to see where they agree and where they disagree.

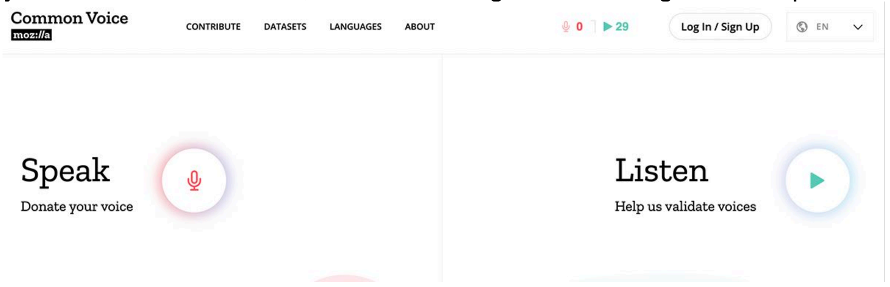
Activity wrap-up:
We learn how to
❑ use tensorflow keras to train and use a neural network model to estimate the price of a house

Activity 1.3 – Recognising Speech with DeepSpeech

In this activity, we will learn:
❑ Build a speech recogniser with DeepSpeech pre-trained model

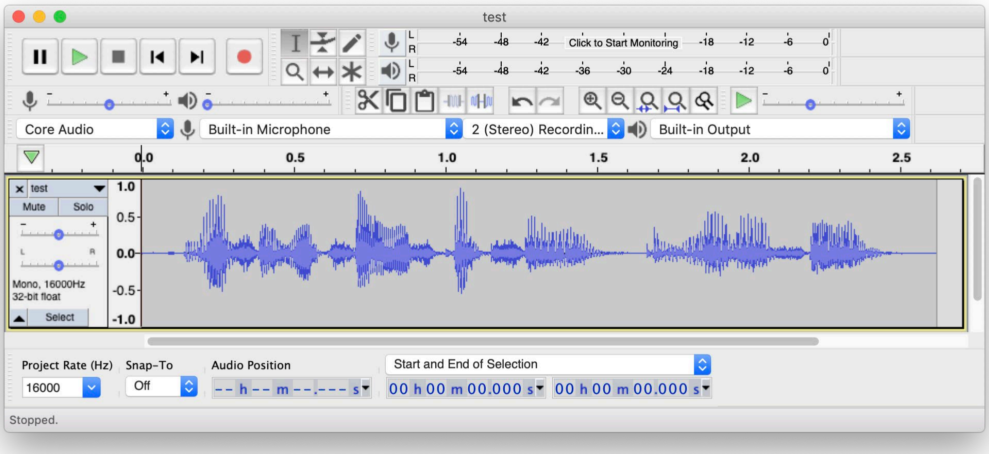
Note: Use aaip_sr_py37 virtual environment (pip install deepspeech==0.5.1)

1) Speech recognition systems are hard to build because they require massive amounts of training data combined with complex models. However, the folks at Mozilla have done excellent work to make both training data and models more easily available to everyone.
First, they created Common Voice, a website that gathers training data for speech recognition.



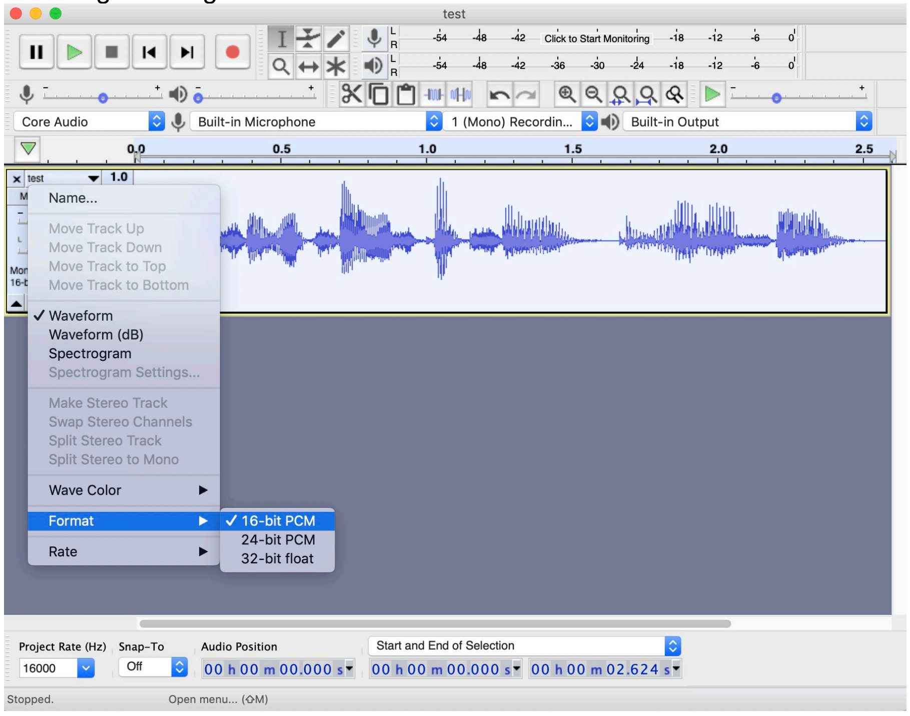
a) The Common Voice website allows volunteers to record themselves speaking and to transcribe audio that others have recorded. All this is done right on the website without any extra tools to download. They've made it simple for anyone to contribute speech recognition training data and the results are shared with anyone who wants to train their own speech recognition system!
b) Second, Mozilla has also built DeepSpeech, a reference implementation of the speech recognition pipeline we learned about in the last chapter. It is implemented with TensorFlow and it's easy to use in our own Python code. They also provide pre-trained models for DeepSpeech based on the Common Voice dataset.
c) Let's use DeepSpeech to transcribe audio and discuss when it might be useful to retrain it with custom training data.

2) **Preparing Audio to Be Processed with Deep Speech**
a) The DeepSpeech model can only process audio files stored in the WAV format. In addition, the audio has to be recorded at 16,000 samples per second at 16-bit resolution. If your audio isn't in exactly this format, you'll need to convert it before you can use it with DeepSpeech.
b) To record new audio or convert existing audio to this format, you can install the free program Audacity. **Audacity** (https://www.fosshub.com/Audacity.html) is an open source audio editing tool. This is what Audacity looks like with an audio file open.



c) To prepare an audio file for DeepSpeech, we need to change these settings:

• The ProjectRate(Hz) setting at the bottom of the screen needs to be set to 16000.
• The audio track needs to be set to Mono,16000Hz,16-bit PCM. This is not the default setting. You can change it using the Select button at the bottom of the audio track.



Project Rate (Hz): 16000
Snap-To: Off
Audio Position: 00 h 00 m 00.000 s
Start and End of Selection: 00 h 00 m 00.000 s – 00 h 00 m 02.624 s

d) When settings are configured, you can record yourself saying anything you want. When you are done, choose File > Export> Export as WAV from the menu and save the file in the same folder as the code for this project.
e) If you don't want to create your own audio file, you can use the provided sample audio file called test.wav that is already in the correct format.

3) Downloading the Pre-Trained Model Data

a) Before we write any code, we need to download the pre-trained DeepSpeech model data from Mozilla's Github account. This data includes both a speech model that converts sounds into letters and a language model that picks the most likely word being spoken in context.
b) We can use wget or curl to download the model data right into the folder with the project code. Run these commands in a Terminal window:

```
curl -LO https://github.com/mozilla/DeepSpeech/releases/download/v0.5.1/deepspeech-0.5.1-
models.tar.gz

wget https://github.com/mozilla/DeepSpeech/releases/download/v0.5.1/deepspeech-0.5.1-models.tar.gz
```

Make sure you download the same version of the model data as the version of the DeepSpeech python library that is installed. For this activity, we are using version 0.5.1.'
c) Next, we need to uncompress the model data. Run this command:

```
tar -zxvf deepspeech-0.5.1-models.tar.gz
```

We will end up with several files that make up the different parts of DeepSpeech.

- alphabet.txt, a list of letters understood by the CTC algorithm used by the speech model.
- output_graph.pb, the pre-trained speech model.
- output_graph.pbmm, the same pre-trained speech model in a memory-mapped file format which is more efficient to use.
- output_graph.tflite, the pre-trained speech model in a format optimized for low-power devices like mobile phones.
- lm.binary, the pre-trained language model.
- trie, an index of the language model stored as a trie data structure which is quick to search.

Of these, we'll need *alphabet.txt, lm.binary, trie and output_graph. pbmm* to use DeepSpeech in a Python script.

4) Recognizing Audio with the Pre-Trained Model

a) Using DeepSpeech to transcribe audio is a three-step process. 1. Load the pre-trained DeepSpeech audio and language models into memory. 2. Load the audio file to transcribe. 3. Run the audio file through the model to create a transcription.
b) To start off, we need to install deepspeech library.

```
pip install deepspeech==0.5.1
pip install deepspeech==0.5.0a8 (for windows)
```

c) import necessary libraries.

```
import numpy as np
import wave
from deepspeech import Model as DeepSpeechModel
from pathlib import Path
```

a) Next, we need to configure how the speech recognition and language models will work. These parameters have all been tuned by Mozilla to perform best with their pre-trained model.

```
# Configurable parameters
CTC_BEAM_WIDTH = 500
LANGUAGE_MODEL_WEIGHT = 0.75
LANGUAGE_MODEL_WORD_INS_BONUS = 1.85

# Parameters that can only be changed by re-training the model
NUM_SAMPLES_PER_WINDOW = 9
NUM_MFCC_FEATURES = 26
```

b) Next, we need to tell DeepSpeech the names of each of the model data files that we will be using. These filenames need to match the data that you downloaded.

```
# Folder paths to pre-trained model data files
CURRENT_FOLDER = Path(__file__).parent.absolute()
SPEECH_MODEL = "deepspeech-0.5.1-models/output_graph.pbmm"
LANGUAGE_MODEL = "deepspeech-0.5.1-models/lm.binary"
LANGUAGE_MODEL_TRIE = "deepspeech-0.5.1-models/trie"
ALPHABET_CONFIG = "deepspeech-0.5.1-models/alphabet.txt"
```

c) We also need to pick an audio file to transcribe. I've included a test.wav file or you can use a different file you've created as long as you've created it using the correct settings in Audacity.

```
# The audio file we want to transcribe
AUDIO_FILE = "test.wav"
```

d) One issue with DeepSpeech is that it needs the full path and filename of the files that we pass to it. To make things easy, we can use Python's Pathlib to combine the folder name where we are running this script with the name of each file inside of it, instead of writing out the full filenames by hand.

```
# Convert the model files to absolute paths (required by DeepSpeech)
speech_model_path = str(CURRENT_FOLDER / SPEECH_MODEL)
language_model_path = str(CURRENT_FOLDER / LANGUAGE_MODEL)
language_model_trie_path = str(CURRENT_FOLDER / LANGUAGE_MODEL_TRIE)
alphabet_config_path = str(CURRENT_FOLDER / ALPHABET_CONFIG)
audio_file_path = str(CURRENT_FOLDER / AUDIO_FILE)
```

e) Now we are ready to create the speech model. This is what will convert sounds into letters using the CTC algorithm. To do this, we create a new speech model object and pass in all the required parameters including the trained speech model itself, the alphabet used by the model, and the model tuning parameters (such as the number of audio samples per CTC detection window).

```
# Load the pre-trained speech model
deepspeech_model = DeepSpeechModel(
    speech_model_path,
    NUM_MFCC_FEATURES,
    NUM_SAMPLES_PER_WINDOW,
    alphabet_config_path,
    CTC_BEAM_WIDTH
)
```

f) The second part of the speech recognition system is the language model that helps the speech recognition system pick the most likely transcriptions based on which words are most likely to be written together in normal English usage. To create this model, we call enableDecoderWithLM() and pass in the alphabet, the pre-trained language model itself (and the trie index file), and two parameters that control how aggressive the language model is in influencing the final transcription.

```
# Load the pre-trained language model
deepspeech_model.enableDecoderWithLM(
    alphabet_config_path,
    language_model_path,
    language_model_trie_path,
    LANGUAGE_MODEL_WEIGHT,
    LANGUAGE_MODEL_WORD_INS_BONUS,
)
```

The language model will almost always improve the quality of the final transcription, but keep in mind that it is totally optional. You can comment it out and see how the transcription looks when the transcription is based entirely on sounds and not on language usage.

g) Next, we need to load the audio sample data from the WAV file we want to transcribe. We can do this using the wave Python library. We also need to grab the sample rate (which should be 16000!) and the length of the audio file.

```
# Load audio file using the wave library
with wave.open(audio_file_path, 'rb') as input_wave_file:
    # Get the sample rate of the audio file
    sample_rate = input_wave_file.getframerate()

    # Get the length of the audio file
    num_samples = input_wave_file.getnframes()

    # Grab the samples from the audio file
    audio_binary_data = input_wave_file.readframes(num_samples)

    # Convert the audio data into a numpy array
    audio = np.frombuffer(audio_binary_data, np.int16)
```

Notice that the last line converts the audio data we loaded into a numpy array. This is because DeepSpeech is implemented in TensorFlow and expects the audio sample data as a numpy array instead of as raw bytes of data.

h) The last step is to transcribe the audio by feeding the audio sample data through the speech and language models. With everything we've already set up, that is as simple as calling the stt() function, which is short for "speech to text."

```
# Transcribe the audio with the model!
text_transcription = deepspeech_model.stt(audio, sample_rate)

print(text_transcription)
```

At the end, we just printed out the transcription to the screen. But you could re-use this code to save the transcription to a file or even call another function based on what the user said.

i) Run the code. You should get a transcription printed out that matches the audio file. If you made your own audio file, you might get a transcription that isn't completely accurate. In that case, you can help out by contributing some data to the Common Voice project, which will help improve the next version of the pre-trained model!

j) For most uses, the pre-trained DeepSpeech model provided by Mozilla will give you better results than retraining the speech and language models yourself. This is because the model is trained on the entire Common Voice dataset, which is more data than you could likely collect yourself. The Common Voice data also collects audio in languages other than English and volunteers are starting to provide pre-trained DeepSpeech models for a variety of languages.

However if you want to build a very specialized tool, like a voice-controlled robot that only understands a limited set of commands, you have the option of training a custom model.

The first step is to collect short audio files of all the phrases you want the model to understand. You need to collect these using different microphones and in different environments that mimic how the final model will be used. For example, if you want to recognize speech on a busy street, the training audio needs to have lots of examples of speech recorded on busy streets. At a minimum, you'll need several thousand audio files recorded under a variety of conditions and in a variety of types of rooms.

The second step is to transcribe all your audio files to text. To do this, you need to create a CSV file with filename, size, and transcription columns and list out all of your training data.

This file should continue for thousands of lines listing each of the thousands of sound files that you recorded and a matching transcription for each file. Make sure your transcriptions only include letters listed in the alphabet.txt file. You can edit the alphabet file if you are working in another language.

From there, you are ready to retrain the speech model by running the commands in in the TRAIN.rst file that comes with the Mozilla DeepSpeech code on Github. You'll need a GPU to retrain it in a reasonable amount of time.

You'll also have the option of retraining the language model. But if you are using English, that probably isn't necessary. And for custom applications like detecting single command words, you probably don't need to use a language model at all.

At the end of all this work, you'll have a custom output_graph.pbmm file that you can use with the same transcription code that we created above. But the results still probably won't be that accurate unless you have access to massive amounts of transcribed data. In most cases, I'd recommend contributing to the Common Voice project and helping improve the shared model that everyone is using instead of building your own mode

<u>Activity wrap-up:</u>
We learn how to
- ❑ Build a speech recogniser with DeepSpeech pre-trained model

Activity 1.4 – Project: Extracting Facts with an NLP Pipeline

In this activity, we will learn:
- ❑ Extracting Entities from text
- ❑ Building a Data Scrubber
- ❑ Extracting Facts from Text
- ❑ Extracting Noun Chunks

Note: Installation of packages:

a) conda install spacy

b) download core model

```
python -m spacy download en_core_web_lg
```

c) conda install textacy.

d) fastText - Facebook's fastText is a text classification library. Text classification is a natural language technique that can be used almost like a secret weapon to solve lots of complex problems. And like the name claims, fastText is really fast. You can classify gigabytes of text with it in mere seconds. (https://fasttext.cc/docs/en/support.html)

```
$ git clone https://github.com/facebookresearch/fastText.git
$ cd fastText
$ make
```

e) Building fasttext python module

```
$ sudo pip install .
$ # or :
$ sudo python setup.py install
```

OR pip install fasttext
conda install -c conda-forge textacy

1) Extracting Entities from Text

a) Several Python libraries implement NLP pipelines similar to the one that we talked about. For our activity, we will be using spaCy. It's fast, easy to use, and among the most accurate NLP libraries available. To use spaCy to do named entity recognition on a piece of text, first, we need to import spaCy:

```
import spacey
```

b) Next, we need to load a trained model for the language that we want to parse. SpaCy provides trained models for several languages. For some languages like English, it even gives you a choice between small, medium, and large models. The large models are bigger and take longer to load when your program starts, but they offer extra features and are sometimes more accurate. We'll use the large English model. So let's load it.

```
# Load the large English NLP model
nlp = spacy.load('en_core_web_lg')
```

c) By convention, when we load the model, we store it in a variable called nlp . This variable is really an object that gives us access to all the functions of spaCy. Let's use it to parse some text.

```
# The text we want to examine
text = """London is the capital and most populous city of England and
the United Kingdom.  Standing on the River Thames in the south east
of the island of Great Britain, London has been a major settlement
for two millennia. It was founded by the Romans, who named it Londinium.
"""

# Parse the text with spaCy. This runs the entire pipeline.
doc = nlp(text)
```

d) At this point, the doc object is a parsed version of the text. We can use this object to access any data created by the NLP pipeline, including named entities and sentence parse trees. Let's just print out a list of all the named entities by looping over doc.ents.

```
# 'doc' now contains a parsed version of text. We can use it to do anything we want!
# For example, this will print out all the named entities that were detected:
for entity in doc.ents:
    print(f"{entity.text} ({entity.label_})")
```

e) Run the code. You will get the following output.

```
London (GPE)
England (GPE)
the United Kingdom (GPE)
the River Thames (LOC)
the south east (LOC)
Great Britain (GPE)
London (GPE)
two millennia (DATE)
Romans (NORP)
Londinium (PERSON)
```

With just a line or two of code, we've already extracted named entities from the text! Named entities are one of the easiest ways to quickly get value out of NLP.

Side note: When working with NLP libraries, you'll run into lots of abbreviations for everything from entity types (like GPE and NORP here) to parts of speech and word dependency types. There are too many possibilities to list here, but you can refer to the spaCy docs online if any of them aren't clear.

However, we see that the NLP pipeline made a mistake on "Londinium" and labelled it as the name of a person instead of as a place. This is probably because there was nothing in the training dataset similar to that and it made a best guess. Named entity detection often requires a little bit of model fine tuning if you are parsing text that has unique or specialized terms like this.

2) Building a Data Scrubber

a) Let's take the idea of detecting entities and twist it around to build a data scrubber. Let's say you are trying to comply with user privacy regulations and you've discovered that you have thousands of documents with personally identifiable information in them like people's names. You've been given the task of removing any and all names from your documents. Going through thousands of documents and trying to redact all the names by hand could take years. But with NLP, it's a breeze. We can use NLP to detect all the names and simply blank them out.

```
import spacy

# Load the large English NLP model
nlp = spacy.load('en_core_web_lg')
```

b) Next, let's create a function called replace_name_with_placeholder that will take in a spaCy token and replace it with the word [REDACTED] if it is a person's name. If the token isn't a name, we'll return it unchanged.

```
# Replace a token with "REDACTED" if it is a name
def replace_name_with_placeholder(token):
    if token.ent_iob != 0 and token.ent_type_ == "PERSON":
        return "[REDACTED] "
    else:
        return token.string
```

c) Remember the tokenization step in our NLP pipeline? In spaCy, we get access to those tokens. Each token is an object that represents a single word (or in some cases, a group of words like a proper name). The token object gives us access to lots of properties that tell us more about the token.

In this case, we used token.ent_iob to check if the current token is part of an entity. The term ent_iob is short for "inside, outside, or beginning of an entity." NLP is chock-full of abbreviations! In any case, token.ent_iob will be set to a non-zero number if the token was part of a named entity.

If the token is part of an entity, we can check token.ent_type_ to see what type of entity it was. If it is set to "PERSON", a person's name was detected. These are the same type names we saw in the last example when we printed out the list of detected entities.

d) Now we can write the scrub() function itself that will run the text through the NLP pipeline and then call the replace_name_with_placeholder() function on each token.

```
# Loop through all the entities in a document and check if they are names
def scrub(text):
    doc = nlp(text)
    for ent in doc.ents:
        ent.merge()
    tokens = map(replace_name_with_placeholder, doc)
    return "".join(tokens)
```

There's one special trick in this function. We are using a for loop to loop through all the entities in the document and calling ent.merge() on each one.
This is a common trick that you'll do a lot. It combines into one single token any tokens that are part of the same entity. This way we can replace each entire person's name with a single [REDACTED] string instead of replacing the first name and last name separately with two different [REDACTED] strings.

e) All that's left to do is call the scrub function on some text.

```
s = """
In 1950, Alan Turing published his famous article "Computing Machinery and Intelligence". In 1957, 
Noam Chomsky's
Syntactic Structures revolutionized Linguistics with 'universal grammar', a rule based system of 
syntactic structures.
"""

print(scrub(s))
```

f) Run the code. You will get the following output.

```
In 1950, [REDACTED] published his famous article "Computing Machinery and Intelligence". In 1957, [REDACTED]
Syntactic Structures revolutionized Linguistics with 'universal grammar', a rule based system of syntactic struc
```

This is a great use of entities! By taking advantage of something our NLP pipeline is good at, we were able to build a really useful program with just a few lines of code.

3) Extracting Facts from Text

a) Let's use NLP to extract facts from our text instead of just names and places. Let's take the Wikipedia article on London and see how much we can learn about London using NLP.

b) The simplest way to find facts about London is to search the parse tree for sentences where the subject is "London" and the main verb is a form of the verb "to be." In those sentences, the target clause of the "be" should be a fact. Here's an example sentence:

London is the capital of England.

In this sentence, the subject is "London" and the "is" is a conjugated form of "be." And the object clause of the sentence is "the capital of England," which is a useful fact we can extract!

We could code this by examining the spaCy parse tree of each sentence and checking for this pattern. But this is a common task called **semi-structured statement extraction**. Instead of re-implementing it ourselves, we can use an off-the-shelf implementation. There's a python library called **textacy** that implements several common data extraction algorithms on top of spaCy, including semi-structured statement extraction. Let's use the implementation included in textacy.

To start off, we'll import the libraries that we'll use and then we'll load our spaCy language model.

```
import spacy
import textacy.extract
from pathlib import Path

# Load the large English NLP model
nlp = spacy.load('en_core_web_lg')
```

c) Now we need some text to examine. I've included a copy of the text of the London Wikipedia article in a file called london.txt so we can load that.

```
# The text we want to examine
text = Path("london.txt").read_text()
```

d) Now we can parse the document with spaCy like we normally would and then use textacy's semistructured_statements function to pull facts out of the parse tree data.

```
# Parse the document with spaCy
doc = nlp(text)

# Extract semi-structured statements
statements = textacy.extract.semistructured_statements(doc, "London")
```

The first parameter to semistructured_statements is the parsed document we want to check. The second is the subject we should look for in each sentence—in this case, London. By default, the function looks for sentences with a form of the verb "to be," but you can also pass in an optional cue="verb" parameter with a different verb to check.

e) Finally, we'll print out the results.

```
# Print the results
print("Here are the things I know about London:")

for statement in statements:
    subject, verb, fact = statement
    print(f" - {fact}")
```

f) Run the code. You will get the following output.

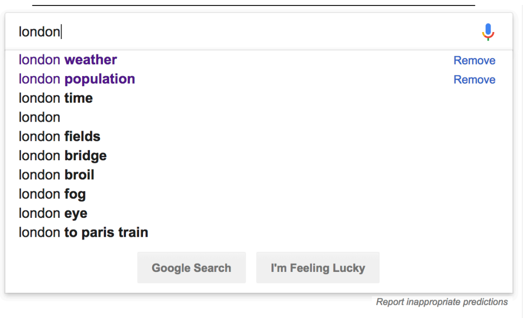
```
Here are the things I know about London:
 - the capital and most populous city of England and the United Kingdom.

 - a major settlement for two millennia
 - the world's most populous city from around 1831 to 1925
 - beyond all comparison the largest
town in England
 - still very compact
 - the world's largest city from about 1831 to 1925
 - the seat of the
Government of the United Kingdom
 - one of nine regions of England and the top-level subdivision
covering most of the city's metropolis
 - vulnerable to flooding.
```

4) Extracting Noun Chunks

a) By looking through the spaCy docs and textacy docs, you'll see lots of examples of the ways you can work with parsed text. What we've seen so far is just the starting point.

b) Here's another practical example: Imagine that you were building a website that lets the user view information for every city in the world using the information we extracted in the last example. If you had a search feature on the website, it might be nice to autocomplete common search queries like Google does.



But to do this, we need a list of possible completions to suggest to the user. We can use NLP to quickly generate this data. One of the algorithms included in textacy is called **noun chunk extraction**. It looks for chunks of words that seem to belong together and seem to refer to a single idea. These kinds of words are often the kinds of keywords that a user will type into a search box. We can use them to populate our autocomplete system.

c) First, let's import our libraries, load our model, and load our text again.

```
import spacy
import textacy.extract
from pathlib import Path

# Load the large English NLP model
nlp = spacy.load('en_core_web_lg')

# The text we want to examine
text = Path("london.txt").read_text()
```

d) Now we can parse the document with spaCy and then run textacy's ***noun_chunks*** function to extract noun chunks.

```
# Parse the document with spaCy
doc = nlp(text)

# Extract semi-structured statements
noun_chunks = textacy.extract.noun_chunks(doc, min_freq=3)
```

The min_freq=3 parameter tells textacy to ignore any noun chunks that don't appear at least three times in the document. The idea is that we are collecting common terms that a user would likely care about, so we don't need every possible noun in the document.

e) The list of noun chunks we get back will be spaCy tokens and the words will be a mix of uppercase and lowercase. Let's convert them to lowercase strings and print out any noun chunks that have at least two words.

```
# Convert noun chunks to lowercase strings
noun_chunks = map(str, noun_chunks)
noun_chunks = map(str.lower, noun_chunks)

# Print out any nouns that are at least 2 words long
for noun_chunk in set(noun_chunks):
    if len(noun_chunk.split(" ")) > 1:
        print(noun_chunk)
```

f) Run the code. You will get the following output (similar).

```
epping forest
trafalgar square
greater london authority
2011 census
office space
london school
city of
london
london eye
south london
tate modern
large number
other city
central london

national gallery
greater london's population
major centre
st paul's cathedral
new york city
river thames
```

we've got some great data to seed our autocomplete tool!

Activity wrap-up:
We learn how to

- ❑ Extracting Entities from text
- ❑ Building a Data Scrubber
- ❑ Extracting Facts from Text
- ❑ Extracting Noun Chunks

Activity 1.5 – Project: Training and Using a Text Classifier

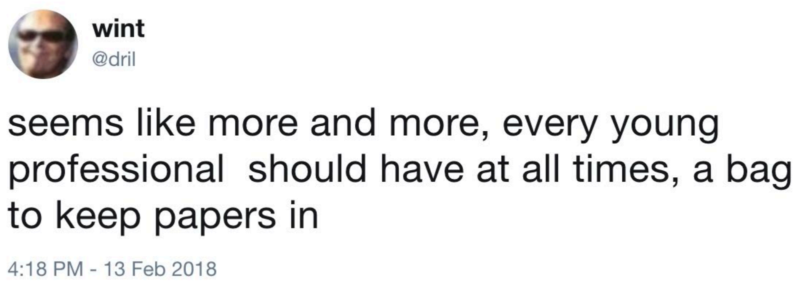
In this activity, we will learn:
- ❑ Training and Using a Text Classifier
- ❑ Using Classification Models to Extract Meaning
- ❑ Training and Using a Text Classifier
- ❑ Writing the Data Preparation Script
- ❑ Training the Model
- ❑ Testing the Model
- ❑ Iterating on the Model
- ❑ Using the Model in a Program

1) Working Bottom Up instead of Top Down

a) So far, we've processed text with an NLP pipeline. However, a lot of text- based problems can be solved with a much simpler approach called text classification. It's a much simpler solution, but often it works very well!

The NLP pipeline that we set up in the last section processes text in a top- down way. First, we split the text into sentences, then we break sentences down into nouns and verbs, then we figure out the relationships between those words, and so on. It's a very logical approach and logic just feels right, but logic isn't necessarily the best way to go about extracting data from text.

A lot of user-created text is messy, unstructured, and, some might even say, nonsensical.


wint
@dril

seems like more and more, every young professional should have at all times, a bag to keep papers in

4:18 PM - 13 Feb 2018

Extracting data from messy text by analysing its grammatical structure is very challenging, because messy text doesn't follow normal grammatical rules. We can often get better results using dumber models that work from the bottom up. Instead of analysing sentence structure and grammar, we'll just look for statistical patterns in word use.

2) Using Classification Models to Extract Meaning

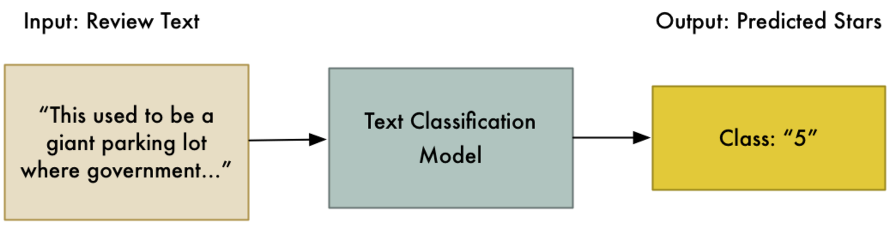
Let's look at user reviews, one of the most common types of online data that you might want to parse with a computer. Here is one of a real five-star Yelp reviews for a public park.

This used to be a giant parking lot where government employees that worked in the county building would park. They moved all the parking underground and built an awesome park here instead. It's literally the reverse of the Joni Mitchell song.

Although the above is a five-star review. But if this is posted without a star rating, you would still automatically understand that the author liked the park from how he described it.

How can we write a program that can read this text and understand that the author liked the park even though he never directly said, "I like this park," in the text? The trick is to reframe this complex language understanding task as a simple classification problem.

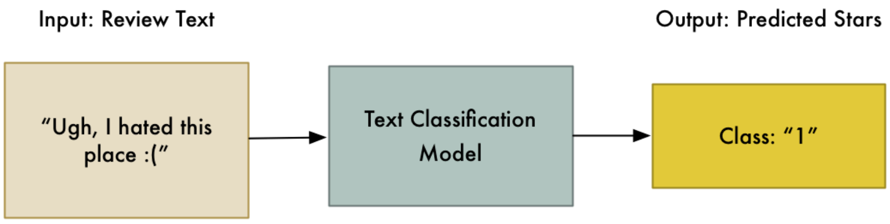
Let's set up a simple linear classifier that takes in words. The input to the classifier is the text of the review. The output is one of five fixed labels—"1 star", "2 stars", "3 stars", "4 stars", or "5 stars".


Input: Review Text
"This used to be a giant parking lot where government…"
Text Classification Model
Output: Predicted Stars
Class: "5"

If the classifier was able to take in the text and reliably predict the correct label, that means it must somehow understand the text enough to extract the overall meaning of whether or not the author liked the park.

To train our text classification model, we will collect a lot of user reviews of similar places (parks, businesses, landmarks, hotels, whatever we can find) where the user wrote a text review and assigned a similar star rating. And by lots, we mean millions of reviews! Then we will train the model to predict a star rating based on the corresponding text.

Once the model is trained, we can use it to make predictions for new text. Just pass in a new piece of text and get back a score.

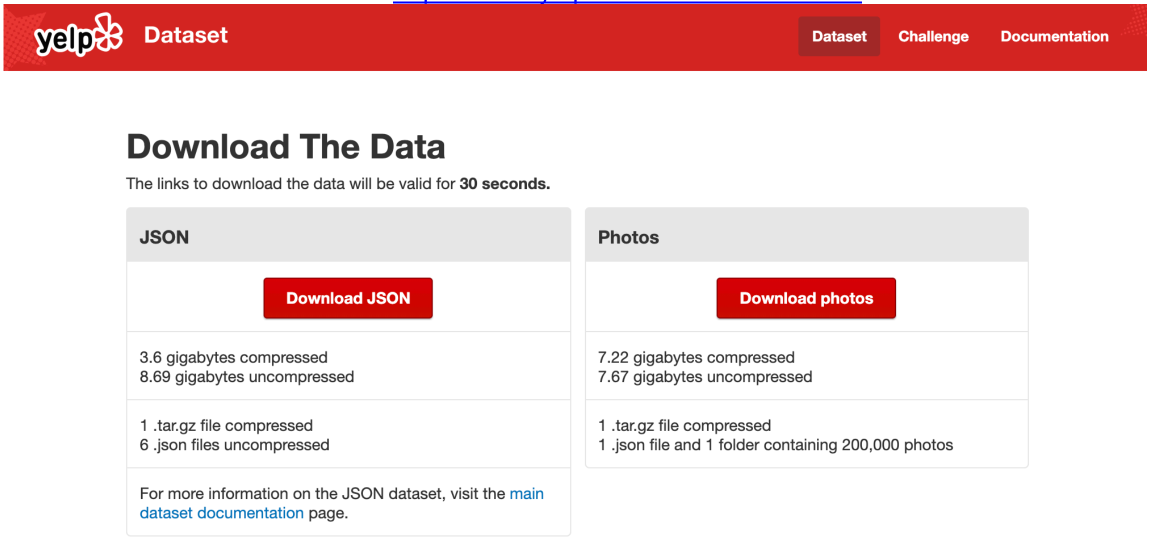


With this simplistic model, we can do all kinds of useful things. For example, we could start a company that analyses social media trends. Companies would hire us to track how their brand is perceived online and to alert them to negative trends in perception.

To build that, we could just scan for any tweets that mentioned our customer's business. Then we would feed all those tweets into the text classification model to predict if each user likes or dislikes the business. Once we have numerical ratings representing each user's feelings, we could track changes of average score over time. We could even automatically trigger an action whenever someone posts something very negative about the business.

3) Training and Using a Text Classifier

a) Let's build a text classifier that can parse any restaurant review and assign a star rating.
b) The tool for building text classification models in this section is Facebook's fastText program. It's open source and you can run it as a command line tool or call it from Python. There other great options like *Vowpal Wabbit* that also work well and are potentially more flexible, but fastText easier to use.
c) **Getting the Training Data**.
To build a reliable user review model, we'll need a lot of training data. Luckily, the popular restaurant review website Yelp provides a research dataset of 4.7 million user reviews that we can use. You can download load the data from their website here: https://www.yelp.com/dataset/download



Download The Data

The links to download the data will be valid for **30 seconds.**

JSON	Photos
3.6 gigabytes compressed 8.69 gigabytes uncompressed	7.22 gigabytes compressed 7.67 gigabytes uncompressed
1 .tar.gz file compressed 6 .json files uncompressed	1 .tar.gz file compressed 1 .json file and 1 folder containing 200,000 photos
For more information on the JSON dataset, visit the main dataset documentation page.	

d) Unzip the tar.gz file. One of the files is a four-gigabyte JSON named review.json. If you get a file named yelp_academic_dataset_review.json instead, just rename it to review.json. You can delete all of the other files included in the download. We won't need them. Each line in the file is a JSON object with data like this:

```
{
   "review_id": "abc123",
   "user_id": "xyy123",
   "business_id": "1234",
   "stars": 5,
   "date":" 2015-01-01",
   "text": "This restaurant is great!",
   "useful":0,
   "funny":0,
   "cool":0
}
```

e) **Formatting the training data**
The first step is to convert this data into the format that fastText expects.
FastText requires a text file with each piece of text on a line by itself. The be- ginning of each line needs to have a special prefix of __label__YOURLABEL that assigns the label to that piece of text.
In other words, our restaurant review data needs to be reformatted like this:

```
1 __label__5 This restaurant is great!
2 __label__1 This restaurant is terrible :'
```

f) **Normalizing the Data**
But reformatting the data isn't all that we need to do. Just like with any other machine learning model, we need to normalize the data before we train the model.
Normalization is a little bit different with text than with neural networks. Instead of rescaling numbers so they are all in a 0-to-1 range, we have to get rid of any patterns in the text that will make it harder to learn from the data.

This is because fastText is totally oblivious to any English language conventions (or the conventions of any other language). As far as fastText knows, the words Hello, hello and hello! are all totally different words because they aren't exactly the same characters. To fix this, we want to do a quick pass through our text to convert everything to lowercase and to put spaces before punctuation marks. This will make it a lot easier for fastText to pick up on statistical patterns in the data.

This means that the text This restaurant is great! should become this restaurant is great!.

g) **Splitting the Data into a Training Set and a Test Set**.
Just like with any other machine learning model, we need a training data set and a test data set. So we need to extract some of the strings from the training dataset and keep them in a separate test data file. Then we can test the trained model's performance with that held-back data to get a real-world measure of how well the model performs.

4) Writing the Data Preparation Script

a) To reformat our data into fastText format, normalize it, and split it into a training a test set, we need to write a little bit of Python code. First, we'll import the libraries we'll use.

```
import json
from pathlib import Path
import re
import random
```

b) Next, we'll set the paths to our input and output files.:

```
reviews_data = Path("review.json")
training_data = Path("fasttext_dataset_training.txt")
test_data = Path("fasttext_dataset_test.txt")
```

The first variable, reviews_data, is the path to the data file you downloaded. You'll need to update the path if you didn't copy that file into the same folder where you are writing the code.
The other two variables, training_data and test_data, are the paths to two new files that we'll create with this script. Those are the training data and test data files that we'll use to train our fastText classifier model.

c) Next, let's decide how much data to hold back as test data. A good starting point is 10 percent.

```
# What percent of data to save separately as test data
percent_test_data = 0.10
```

d) Let's write the string normalization function. We need to convert everything to lowercase and add blank space around any punctuation marks.

```
def strip_formatting(string):
    string = string.lower()
    string = re.sub(r"([.!?,'/()])", r" \1 ", string)
    return string
```

e) Now we are ready to process the data. First, let’s open up the input JSON file and create the training data and test data output files.

```
with reviews_data.open() as input, \
     training_data.open("w") as train_output, \
     test_data.open("w") as test_output:
```

f) We can process the review data by looping over the JSON file one line at a time. Each line is a JSON record containing a separate restaurant review, so we’ll parse its contents and then pull out the rating and review text. We can also go ahead and call our strip_formatting() function on the review text to normalize it.

```
    for line in input:
       review_data = json.loads(line)

       rating = int(review_data['stars'])
       text = review_data['text'].replace("\n", " ")
       text = strip_formatting(text)
```

g) All that’s left to do is to output the normalized text in fastText format. We’ll use a random number generator to decide if the line should end up in the training data file or the test data file.

```
       fasttext_line = "__label__{} {}".format(rating, text)

       if random.random() <= percent_test_data:
           test_output.write(fasttext_line + "\n")
       else:
           train_output.write(fasttext_line + "\n")
```

Run that and you will end up with two new files, fasttext_dataset_training .txt and fasttext_dataset_test.txt. The script is processing gigabytes of review data so it might take several minutes to run. But when it’s done, we are ready to train the model!

5) Training the Model

a) We’ll train the classifier using the fasttext command line tool. Just call fasttext, pass in the supervised keyword to tell it to train a supervised classification model, and then give it the training file and an output name for the model.
Type this command in a terminal window open to the same folder where your code is located:

```
fasttext supervised -input fasttext_dataset_training.txt -output reviews_model
```

Here’s what the training process looks like:

```
Read 797M words
Number of words:  1337432
Number of labels: 5
Progress: 100.0% words/sec/thread: 2043147 lr:  0.000000 avg.loss:  0.766230 ETA:   0h 0m 0s
```

Depending on the speed of your computer, this model should finish training in five to 10 minutes. Not bad for 800 million words of data!

6) Testing the Model

a) Let’s see how accurate the model is by checking it against our test data. We can do that right from the fasttext command by passing in test, the name of the model, and the name of the test data file.

```
fasttext test reviews_model.bin fasttext_dataset_test.txt
```

Here are the results it gives us:

```
N  668309
P@1     0.696
R@1     0.696
```

Note: Since we randomized the test and training data split, your results should be similar but slightly different.
The P@1 score is the precision score and the R@1 score is the recall score. These work exactly the same way as when we used them to evaluate neural networks.

It guessed the user’s exact star rating 69.6 percent of the time. Not bad for a first attempt at training the model.

You can also ask fastText to check how often the correct star rating was in one of its first two predictions. When we are training classifiers, sometimes getting the exact prediction isn’t necessary. If the model’s top two most likely guesses were “5” and “4” and the real user said “4”, that might be close enough.

```
fasttext test reviews_model.bin fasttext_dataset_test.txt 2
```

And here are the results:

```
N  668309
P@2     0.458
R@2     0.916
```

That means that 91.6 percent of the time, the model recalled the user’s star rating as one of its top two guesses. That’s a good indication that the model is not far off in most cases even when it isn’t exactly correct. But since it wasn’t getting the correct score on the first guess, this, in turn, lowers the model’s precision score.

b) You can also try out the model interactively by running the fasttext predict command and then typing in your own reviews. When you hit enter, it will tell you its prediction for each one.

```
fasttext predict reviews_model.bin -

this is a terrible restaurant . i hate it so much .
__label__1
this is a very good restaurant .
__label__4
this is the best restaurant i have ever tried .
__label__5**
```

When you are done, hit Ctrl-D to end the program.

7) Iterating on the Model

a) With the default training settings, fastText considers each word independently and doesn't care at all about word order. But when you have a large training dataset, you can ask it to take the order of words into consideration by using the wordNgrams parameter. That will make it track groups of words instead of just individual words.
For a dataset of millions of words, tracking two-word pairs (also called bi- grams) instead of single words is a good starting point for improving the model.

Let's train a new model with the -wordNgrams 2 parameter and see how it performs.

```
fasttext supervised -input fasttext_dataset_training.txt -output reviews_model_ngrams -wordNgrams 2
```

And here are the training results:

```
Read 797M words
Number of words:  1337432
Number of labels: 5
Progress: 100.0% words/sec/thread: 1034046 lr:  0.000000 avg.loss:  0.668431 ETA:   0h 0m 0s
```

Now the model will create a feature for every possible two-word pair instead of single words. This will make training take a bit longer and it will make the model file much larger, but it can be worth it if it gives us higher accuracy.

b) Once the training completes, you can re-run the test command in the same way as before and see how using bigrams affected accuracy.

```
fasttext test reviews_model_ngrams.bin fasttext_dataset_test.txt
```

Here are the results it returns:

```
N  668309
P@1     0.729
R@1     0.729
```

By using bigrams, our model's precision and recall scores jumped from 69.6 percent to 72.9 percent. That's an improvement of over 3 percent!

When you use the model, you might also notice that this seems to reduce the number of obvious errors that the model makes because now it cares more about the context of each word. It's harder to trick the model by creating a sentence that includes a negative word in an otherwise positive sentence.

There are other ways to improve your model, too. One of the simplest but most effective ways is to skim your training data file by hand and make sure that the preprocessing code is formatting your text in the most useful way possible.

For example, our text preprocessing code will turn the common restaurant name P.F. Chang's into p . f . chang 's. That appears as seven separate words to fastText.

If you have cases like that where important words representing a single concept are getting split up, you can write custom code to fix it. In this case, we could add code to look for common restaurant names and replace them with placeholders like p_f_changs so that fastText sees each as a single word.

8) Using the Model in a Program

a) Now that we've trained a model, we can use it inside a program to make new predictions by using the Python fastText wrapper library created by Facebook called simply fasttext. Here's how to use it. First, we'll import the libraries that we need.

```
import fasttext
import re
```

b) Next, we need to include the exact same string normalization function that we used to create the training data. We'll use this function to normalize text before we run it through the model.

```
def strip_formatting(string):
    string = string.lower()
    string = re.sub(r"([.!?,'/()])", r" \1 ", string)
    return string
```

c) Now we need new restaurant reviews to run through the model. Here are a few made-up reviews, but you can create your own if you want:

```
# Reviews to check
reviews = [
    "This restaurant literally changed my life. This is the best food I've ever eaten!",
    "I hate this place so much. They were mean to me.",
    "I don't know. It was ok, I guess. Not really sure what to say.",
]
```

d) Next, normalize the text in those reviews with our strip_formatting function.

```
# Pre-process the text of each review so it matches the training format
preprocessed_reviews = list(map(strip_formatting, reviews))
```

e) With the data normalized, we can load our fastText model and use it to make predictions for those reviews.

```
# Load the model
classifier = fasttext.load_model('reviews_model_ngrams.bin')

# Get fastText to classify each review with the model
labels, probabilities = classifier.predict(preprocessed_reviews, 1)
```

Loading the model is simple. Just call fasttext.load_model() and pass in the filename of the model. That creates a classifier object that we can use to make new predictions.

The classifier.predict() function takes in two parameters:
• The first parameter is the array of text to use as input to the model. Just like with scikit-learn and Keras, the fastText library assumes you will pass in an array of data to process instead of a single piece of data.
• The second parameter is how many guesses to generate for each piece of text. The value 1 means we just want it to return the model's single best guess for each review.

f) All that's left to do is to print out the results.

```
# Print the results
for review, label, probability in zip(reviews, labels, probabilities):
    stars = int(label[0][-1])

    print("{} ({}% confidence)".format("*" * stars, int(probability[0] * 100)))
    print(review)
    print()
```

g) Run the code. You will get the following output:

```
***** (100% confidence)
This restaurant literally changed my life. This is the best food I've ever eaten!

* (97% confidence)
I hate this place so much. They were mean to me.

*** (76% confidence)
I don't know. It was ok, I guess. Not really sure what to say.
```

Because we have so much training data, the model seems to be really good at scoring each review. This is why machine learning is so cool. Once we figured out a good way to pose the problem as a classification problem, the algorithm did all the hard work of extracting meaning from the training data. You can then call that model from your code with just a couple of lines of code. And just like that, your program seemingly gains superpowers.

Try creating your own imaginary restaurant reviews and seeing how well the model performs!

<u>Activity wrap-up:</u>
We learn how to
- ❑ Training and Using a Text Classifier
- ❑ Using Classification Models to Extract Meaning
- ❑ Training and Using a Text Classifier
- ❑ Writing the Data Preparation Script
- ❑ Training the Model
- ❑ Testing the Model
- ❑ Iterating on the Model
- ❑ Using the Model in a Program