**EECS 16ML Machine Learning** **Fall 2020**

# Lecture 1: November 30

*Lecturer: Anirudhan Badrinath, Mehul Raheja, Andrew Lin* *Scribe: None*



## 1.1 Introduction to Keras

Keras is a machine learning framework with a strong focus on the development of neural networks in a fashion that abstracts many of the mathematical underpinnings required by TensorFlow away. It provides an introduction to the development of models such as the typical deep neural network, convolutional neural network, and recurrent neural network in less than 10 lines from model instantiation to training and testing. Typically, the learning curve for Keras is dominated by its more advanced features such as data generators, the functional model, and advanced activation functions. We will explore these more complex features in the later sections of this note.

We will begin development of neural networks in Keras by installing it with the TensorFlow backend (note: it is possible to use a Theano backend, but it is less conducive to GPU utilization and lacks features). To do so:

```
pip install tensorflow keras
```

### 1.1.1 Installation and Setup

The setup for Keras should be relatively painless as it comes bundled with Python's TensorFlow module, which has wide support for nearly all types of machines. Any online notebook system such as Google Colab, Jupyter DataHub, or Kaggle will be able to support TensorFlow, and by extension, Keras. The version of TensorFlow referenced in this note is 2.2.0 and the corresponding version of Keras is 2.3.1, but the examples provided below should work fairly consistently regardless of version.

Support for GPU utilization for training machine learning models requires more effort. We will not cover the installation of CUDA libraries, but there are readily available online tutorials that aid in installing the required system packages and drivers.

### 1.1.2 Creating a Simple Neural Network

We will demonstrate how to create a simple neural network, and eventually, we will be able to extend this simple design paradigm to much more complex neural networks with potentially tens or hundreds of layers. To do this, we use Keras's Sequential model paradigm which treats neural networks as a linearly growing stack of layers that don't have any skipped connections (i.e. the first layer doesn't connect to the last layer). By following this modeling paradigm, we are able to use a simple interface to add new layers to a neural network.

The following code snippet illustrates how to instantiate a simple Sequential model with an input layer, a hidden fully connected layer with 10 nodes and the hyperbolic tangent as the activation function, and an output layer. We assume that the input shape is a vector in $R^1 28$ and that we will output a scalar quantity. Refer to section 1.1.3 for more information about the layers themselves.

Note that the input shape is passed in as a tuple and describes the shape for each training reference $x_i$. Likewise, the output shape is described by the number of units in the final output layer, which is one in this case. The hidden and output layers can optionally specify an activation function, such as the hyperbolic tangent (tanh), rectified linear unit (RELU), sigmoid, or softmax.

```
import keras
from keras.layers import *
from keras.models import Sequential

model = Sequential()
model.add(Input(shape = (128,)))
model.add(Dense(units = 10, activation = 'tanh'))
model.add(Dense(units = 1, activation = 'linear'))
```

An additional mode of specifying the layers of the neural network without using the add function is by providing a list of layers. The following code behaves functionally identically to the previous example in all aspects, and all Sequential models can be constructed in either fashion.

```
# ...
model = Sequential([Input(shape = (128,)),
                    Dense(units = 10, activation = 'tanh'),
                    Dense(units = 1, activation = 'linear')])
```

The model construction can be verified using the following code snippet, which displays the structure of the neural network. For more complex networks, this can serve as a visualization and debugging tool. Along with general structural information, the following function displays information about the number of parameters, the names and orders of layers and the preceding layer (note: using the functional model, we are able to "skip" layers which is why the preceding layer is listed).

```
print(model.summary())
```

With the model construction complete, there are still some important aspects of the model that are left unspecified. These include the loss function, the chosen optimization technique (i.e. stochastic gradient descent), and the number of epochs to train the model.

We will fill in these blanks by compiling the model with an appropriate optimization technique and loss function. Then, we are able to train the model based on the chosen input data and observations, number of epochs and the chosen batch size. Note that the training process doesn't begin until the call to the fit function: we only fill in our desired loss function and optimizer in the call to compile.

There are many possible options for the optimizer and loss function that may drastically alter the performance of the training process. The most typical combination for a simple regression task is the Adam optimizer combined with the mean-squared error (MSE) whereas the most typical combination for a classification task is Adam combined with binary or categorical cross-entropy. We discuss these options in further detail in a later section.

The model fitting process can take either in the order of seconds or days depending on the size of the dataset, the depth and number of parameters within the network, and the number of epochs and batch size. Typically, the greater the epochs, number of parameters and depth in the network, the training time increases. The reverse is true of batch size, but after a certain threshold, the model's training efficacy will suffer if the batch size is too high.

```
model.compile(optimizer = 'Adam', loss = 'mse')
# epochs and batch_size are optional
model.fit(X_train, y_train, epochs = 10, batch_size = 32)
```

We are able to evaluate and predict using our fitted model as follows. Note that any preprocessing or postprocessing with regards to normalization of inputs needs to occur before and after the calls to these functions respectively. In this simple case, we assume that there is no normalization of inputs. The predict function will take the same sorts of input methods as the fit function, and it will return an array of predictions. The evaluate function predicts on the given dataset, then applies the loss function used during training time to evaluate the error of the model.

In this case, the manual calculation of the mean-squared error should be equivalent to Keras's evaluation.

```
def mse(y_pred, y_true):
    return np.mean((y_pred - y_true) ** 2)

y_pred = model.predict(X_val)
assert np.isclose(mse(y_pred, y_val), model.evaluate(X_val, y_val)), "Should never happen!"
```

This general process of constructing, compiling, fitting, and testing a model is constant throughout the development paradigm of Keras. The more complex Keras models will typically only affect the construction and fitting of the model in terms of additional code. We explore some of the more advanced layers in Keras that are useful in such complex models in the next section.

### 1.1.3 Creating a Convolutional Neural Network in Keras

Some datasets require an extension of the features and layers we have thus far observed for several good reasons. While Occam's Razor generally speculates that the simplest possible effective model is the better among those that are more complex, some types of datasets and machine learning tasks benefit from convolution. The idea behind the convolutional neural network (CNN) in Keras is to effectively use spatial features and dependency to generate features that may be more useful in the prediction task. Examples of such datasets that would demand a CNN framework would be image-based or spatially distributed datasets. Mathematically stated, there is some spatial interdependency between different elements in the input dataset in $R^2$.

To implement a convolutional neural network in Keras, the overall process does not change with the exception of the addition of two new types of layers, the convolutional and pooling layer. Optionally, we observe the effect of the Dropout layer, which serves as a randomized form of regularization. Some of the following code is copied from the previous section for consistency in showing a standalone example.

Note that the differences between the last example and the following example lie only in the convolutional and pooling layers. The idea behind the convolutional neural network in Keras is ultimately to follow the same pattern as any other model, but the usage and order of layers differs. Typically, we use convolutional

and pooling layers in succession followed by a final Dense layer to order our intermediate values in the form of the output (i.e. a number in this case since Dense(units = 1) specifies a scalar).

Specifically, the convolutional layer demands the number of filters, the kernel size, an optional activation function and a padding scheme whereas the pooling layer only demands a pooling size. Note that these parameters and the specific layers are explained in greater detail in Section 1.1.4.

Further note that the input shape is now a spatial field with one color channel as opposed to a vector of 128 elements. Typically, inputs to convolutional neural nets that process images have RGB color channels and hence, the input shape becomes the a tuple of the height, width, and number of channels.

```
import keras
from keras.layers import * # to save space
from keras.models import Sequential

model = Sequential()
model.add(Input(shape = (128, 128, 3)))
model.add(Conv2D(filters = 64, kernel_size = (3, 3), activation = 'relu', padding = 'same'))
model.add(MaxPooling2D(pool_size = (2, 2)))
model.add(Conv2D(filters = 128, kernel_size = (3, 3), activation = 'relu', padding = 'same'))
model.add(MaxPooling2D(pool_size = (2, 2)))
model.add(Flatten())
model.add(Dense(units = 1, activation = 'linear'))

model.compile(optimizer = 'Adam', loss = 'mse')
model.fit(X, y, epochs = 10, batch_size = 32)
```

Note that Keras does provide support for convolutional neural networks that operate in 3D (Conv3D) as well.

### 1.1.4 Layers in Keras

Keras provides a variety of layers to use in a neural network. We summarize all the main types of layers below with an example of the context in which they are typically used.

- Input Layer - This is the entry point to most neural networks created in Keras (technically, Sequential models can begin with any layer with an input_shape parameter, but we will keep it simple). They take one parameter: the input shape.
- Dense Layer - This is the Keras equivalent of the fully connected layer in neural networks. They can be accompanied by an optional activation function to apply after the layer. Given $n$ nodes at the input of the Dense layer and a desired $k$ nodes as the output, it introduces $(n + 1)k$ trainable parameters. Note that there is an extra node for the bias term. These are used in most regression and classification tasks alike, often at the very minimum as a way to format the output as a vector after operations such as convolutions.
- Activation Layer - This specifies a layer that applies an activation function. Most of the time, this is included along as a parameter with fully-connected and convolutional layers; hence, separate layers for activation are often omitted. This is not the case when there are more advanced activation functions such as the leaky rectified linear unit (LeakyRELU), where the activation layer has to be separately specified as a layer.

- Convolutional Layer - This layer (named Conv2D, Conv3D) is typically used with image-related datasets and helps capture spatial features. Its parameters include the number of convolutional filters, the kernel and stride size, an optional activation function, padding schemes, and regularization. Typically, anywhere from 16-512 filters are used depending on the magnitude of the machine lerning task, with the kernel size set to anywhere from (2, 2) to (16, 16). The most common activation function used with the convolutional layer is the rectified linear unit (RELU), with the 'same' padding scheme. Most of these parameters will default to a reasonable value except for the activation function, which must be specified explicitly if we wish to add an activation function. The number of parameters introduced is a linear function of the kernel size, number of filters, and stride.

- Dropout Layer - This layer serves as a Lasso-type regularization in the sense that it zeroes parameters introduced by other layers to choose the most relevant and effective features. However, the difference is that the Dropout layer randomly samples a given proportion of the parameters to zero as opposed to applying an absolute penalty over the parameter weights. The other parameters are rescaled accordingly. The only pertinent parameter for the Dropout layer is the proportion of parameters to zero, which is typically set from 0.2 to 0.5 depending on the type of network (CNNs typically use lower fractions, whereas typical NNs use higher). These tend to be used more prominently in large networks that are prone to overfitting to data.

- Pooling Layer - This layer serves to reduce the input space by accumulating over a spatial region by maximizing or averaging. Typically, this captures the most important spatial information generated by the convolutional layers and hence, it is placed after the convolutional layers. Its parameters include a pool size, which specifies the dimensions over which to apply the chosen pooling function (i.e. maximization or averaging). Typically, we do not apply an activation function after the pooling layer.

- Recurrent Layer - These layers tend to be used with models describing an evolving temporal input or output space. These include the LSTM, ConvLSTM, and simple RNN layers, but we don't explore these layers in detail. These typically take inputs that vary along a time axis, but in all other respects, are fairly similar to all other Keras models.

- Batch Normalization Layer - This layer provides normalization within the network that adaptively transforms the data as it is trained to be normalized. There are no parameters, and this is again typically used in deeper networks with more variability in the data.

### 1.1.5 Extending to More Advanced Models

Typically, more advanced models are implemented through the repetition of layer patterns. For example, the award-winning U-Net architecture convolutional neural network for segmentation map generation in biomedical imaging uses 4 nearly identical contracting layers, which consist of a convolutional and pooling layer. Typically, in Python, this can be achieved by simply copying the layers and changing some parameters such as the number of filters.

Another aspect of model complexity is the choice of more complex individual layers. Whereas the Dense layer is fairly straightforward, some of the recurrent and convolutional layers require more complex engineering of data and further thought into their utility. In that sense, we can extend a similar thought process for the use of more complex layers in Keras just as we did for extending a traditional neural network into a convolutional neural network. The following questions can aid in addressing those thoughts:

- Does the layer present value in the context of the input data type and the type of machine learning? An image-based dataset could benefit from the spatial featurizations that convolutional layers present.

A dataset with heavy temporal dependency (i.e. precipitation datasets) could benefit from a combination of temporal and spatial featurizations provided by a convolutional LSTM layer. Some feature engineering and deeper thought about the data is required to answer this question.

- Does this layer add unnecessary intractability into the model? By Occam's Razor, we assert that simpler efforts, when they yield a similar result, are always more fruitful. In the modern day, it's easy to spam convolutional layers to spatially break down every possible combination of pixels, but chances are that it won't perform nearly at all with an unseen dataset and that it will take many orders of magnitude more training time than a simpler model.
- Does this layer present challenges with interacting with the rest of the model? For example, if we wish to add a Dropout layer to increase the regularization in our neural network, this has a significant impact on the weights of the convolutional layer that may follow it. In many cases, when Dropout layers are used near the tail of a network, it presents significant challenges to the network and often results in poor performance.

### 1.1.6 Advanced Features

**Transfer Learning**

One of the most common downfalls of deep learning is the lack of "initializing" a neural network with any knowledge. For example, training a model to classify between cats and dogs, conceptually, may require it to first understand how to detect basic shapes like lines and circles before comprehending how to put them together to determine if an image is a cat or dog. Inevitably, this causes each problem to require massive amounts of data. Transfer learning attempts to remedy this by utiizing the idea that weights from a general models can effectively serve as weights for a more specific one. One of the most common uses of this idea is in image classification. Various different groups have trained keras models to perform surprisingly accurate classification on over 14 million images in the ImageNet database. One such models pre-trained weights can be found here: https://www.kaggle.com/keras/resnet50. In creating such a model, the top/output layer would be the aforementioned softmax layer to output a probability distribution over the hundreds of classes in ImageNet. We can load the model as follows:

```
ResNet50(weights=resnet_weights_path)
```

If, for instance, we are trying to solve the task of classifying cats and dogs, we no longer want the final layer for the ImageNet classes. Hence, we actually load the model like this:

```
ResNet50(include_top=False, weights=resnet_weights_path)
```

However, now the output is just the output of the second to last layer in this ResNet model which is fairly uninterepable. By the design of Keras, we can create a Sequential model which essentially appends the softmax layer that will output the probability distribution of the classes we want (cats and dogs) to ResNet:

```
model = Sequential()
model.add(ResNet50(include_top=False, weights=resnet_weights_path))
model.add(Dense(2, activation='softmax'))
```

Finally, we want to make sure that the weights of the ResNet itself are not modified, hence we can do:

```
model.layers[0].trainable = False
```

Now, if we have our input images as X and our data labels as y, calling model.fit(X,y) should train the weights of only the output layer and, theoretically, build a fairly accurate classification model in very few epochs and without too much data.

**Data Generators**

Typically, when machine learning models become more complex, we tend to receive higher volumes of data. In many systems, this data is too intractable to fit into memory all at once to train on, even with dedicated GPU and RAM. In this case, we will want to dynamically load in data as we use it for training purposes. For this, Keras supports the generator-style of specifying input training and test data.

We can accomplish this using simple Python generators or a dedicated class that extends one of Keras's or TensorFlow's dedicated Generator classes. For simplicity and Python style, we will demonstrate the former since it is more commonly used in Keras despite other machine learning frameworks preferring the latter. The following code shows an example of a generator function that loops infinitely through multiple datasets in files that are dynamically loaded. Assume that all the files in our dataset consist of csv files.

```
def data_gen(file_locs):
    while True:
        for file in file_locs:
            raw_df = pd.read_csv(file)
            X_train, y_train = load_process_data(raw_df)
            yield X_train, y_train
```

When we wish to use the generator to train the model, it's quite simple. Omit the observations array and instead pass in the generator itself as follows. The file locations parameter is not mandatory either and your custom generator could take any range of parameters or none at all.

```
# ...
train_locations = ["/home/foo/a.csv", "/home/foo/b.csv", "/home/foo/c.csv"] # and so on
model.fit(data_gen(train_locations), epochs = 10, batch_size = 32)

# ...

pred_locations = ["/home/foo/d.csv", "/home/foo/e.csv", "/home/foo/f.csv"] # and so on
y_preds = model.predict(data_gen(file_locations))
print(model.evaluate(data_gen(pred_locations)))
```

**Functional Model**

The Keras Sequential model is quite powerful in terms of the number and complexity of models it can construct, but many non-sequential models (i.e. those with skipped connections) are unable to be modeled thus far in Keras. Fortunately, Keras provides an entirely different way of specifying models which is based on Python functions as layers. Instead of adding layers sequentially, we tie sequential layers together manually by providing the input to each layer as fetched from another layer.

The following code snippet demonstrates this behaviour. Note that the inputs to subsequent layers are the outputs of previous layers. In this case, the input layer is fed into the argument of the function returned by the Conv2D function, which returns a new functional layer. This function is fed into the next layer, and so on.

```
inp = keras.Input(shape = (28, 28, 1))
c1 = Conv2D(32, (3, 3), activation = 'relu', padding = 'same')(inp)
c1 = Conv2D(32, (3, 3), activation = 'relu', padding = 'same')(c1)
inter = MaxPooling2D((2, 2))(c1)
c2 = Conv2D(32, (3, 3), activation = 'relu', padding = 'same')(inter)
c2 = Conv2D(32, (3, 3), activation = 'relu', padding = 'same')(c2)
inter = MaxPooling2D((2, 2))(inter)
inter = Dropout(0.2)(inter)
```

The process to train and test the model remain otherwise identical.

## 1.2 Introduction to Spark

Spark is a combined data processing and machine learning framework that provides easy-to-use models and parallelizable data wrangling tools. Similar to Pandas in its data processing tools and SciKit-Learn in its machine learning model utilization, it provides an efficient and easy-to-use interface to carry an end-to-end machine learning experience. With access to regressors and classifiers such as linear regression, logistic regression, decision trees, SVMs, and more, Spark provides a fully fledged module for students to orient themselves with efficient data processing and training machine learning models without breaking the abstraction barrier of most other tools.

### 1.2.1 Installation

The installation of Spark, similar to that of Keras, should be fairly painless. While Spark supports multithreading natively, it should be universally available regardless of platform. Spark is available for many different languages, including Python, Java and others. To install Spark on Python, we can again use pip in the same fashion as before:

```
pip install pyspark
```

### 1.2.2 Map and Reduce

The map and reduce functions that are provided with Spark are reminiscent of a Hadoop framework in which data processing is parallelized and work is distributed among different threads or servers. Often, the map function will demand a Python lambda function and mutates or converts the data into a particular format. The reduceByKey function will aggregate all the inputs based on the given function and modifies the value of the aggregated items.

The following code snippet illustrates a potential generic word-crunching task using Spark in which we aggregate all non-whitespace separated words and reduce by word such that we are able to count the number of instances of each word.

```
# assume imports are taken care of!
text_file = sc.textFile("/home/foo/a.txt")
counts = text_file.flatMap(lambda line: line.split()) \
                  .map(lambda x: (x, 1)) \
                  .reduceByKey(add)
```

We are able to use any type of Python function within the map and reduce operations. For example, the following code performs case insensitive counting over all unique words.

```
def split(line):
    return line.split()
def count(x):
    return (x.lower(), 1)
text_file = sc.textFile("/home/foo/a.txt")
counts = text_file.flatMap(split) \
                  .map(count) \
                  .reduceByKey(add)
```

The usage of map and reduce reach far beyond the scope of these examples, but for our purposes, these 3 commands should suffice in gaining an understanding of how to efficiently process data for use in machine learning models such as those in Spark or Keras.

### 1.2.3 DataFrames and Cleaning Tools

Spark supports DataFrames similar to Pandas; however, it presents a more computationally sustainable method for processing large datasets as it offers the ability to parallelize across both threads and other servers or computers. We explain preliminary functions to use for processing data within Spark DataFrames, but many functions are identical or quite similar to their usage in Pandas.

We begin with a simple example of processing data from lists into a Spark DataFrame.

```
# assume SQL context sqc
data = [('Mountain View', 'CA', 8), ('San Diego', 'CA', 6), ('Sacramento', 'CA', 5),
        ('Seattle', 'WA', 3), ('Los Angeles', 'CA', 8), ('Miami', 'FL', 2),
        ('Washington', 'DC', 10)]
rows = [Row(location = i, state = j, rating = k) for i, j, k in data]
df = sqc.createDataFrame(rows)
```

If we wanted to parallelize the above code, we could have applies the Row constructor as a map operation as follows. The resulting DataFrame is identical, but for large volumes of the data, the parallelization capabilities of Spark would be beneficial.

```
# ...
# assume Spark context sc
data_proc = sc.parallelize(data)
result = data_proc.map(lambda i, j, k: Row(location = i, state = j, rating = k))
df = sqc.createDataFrame(result)
```

We can use several Pandas-like functions to process and apply various functions to our DataFrame. The functions that are demonstrated below bear quite a bit of resemblance to Pandas functions, and hence, a lot of other functions that are useful are omitted and left as an exercise for the reader to discover. As such, we list the Pandas analogue along with the usage of the function.

Note that all the functions below require the previous steps to have been completed and for the columnn names to exist within the DataFrame.

```
# Spark -> Pandas equivalent
df.show() # -> repr(df)
df.select('location') # -> df['location']
df.sort('location') # -> df.sort_values(by = 'location')
df.filter(df.state == 'CA') # -> df[df['state'] == 'CA']
df.groupBy('location').agg(sum) # -> df.groupby('location').sum()
df.join(other_df, 'location', how = 'right') # -> df.merge(other_df, 'location', how = 'right')
```

Without a full Pandas background, it may be slightly challenging to parse the functions, but most behave similarly to their naive Python counterparts as well. We show some examples of its utilization along with the outputs below. There are additional examples of DataFrame manipulation in the problem set as well as the open-ended tasks.

```
# Spark -> Python equivalent
>>> df.show() # -> print(...)
+-------------+-----+------+
|     location|state|rating|
+-------------+-----+------+
|Mountain View|   CA|     8|
|    San Diego|   CA|     6|
|   Sacramento|   CA|     5|
|      Seattle|   WA|     3|
|  Los Angeles|   CA|     8|
|        Miami|   FL|     2|
|   Washington|   DC|    10|
+-------------+-----+------+

>>> df.sort('location').show() # -> sorted(...)
+-------------+-----+------+
|     location|state|rating|
+-------------+-----+------+
|  Los Angeles|   CA|     8|
|        Miami|   FL|     2|
|Mountain View|   CA|     8|
|   Sacramento|   CA|     5|
|    San Diego|   CA|     6|
|      Seattle|   WA|     3|
|   Washington|   DC|    10|
+-------------+-----+------+

>>> df.filter(df.state == 'CA').show() # -> filter(...)
+-------------+-----+------+
|     location|state|rating|
+-------------+-----+------+
|Mountain View|   CA|     8|
|    San Diego|   CA|     6|
|   Sacramento|   CA|     5|
|  Los Angeles|   CA|     8|
+-------------+-----+------+
```

### 1.2.4 Fitting a regression model

The machine learning models within Spark make use of its DataFrames. There are methods to implement many different regression and classification methods, such as linear regression, LASSO, ridge regression, and support vector machines. We first need to load and parse the data, then build a model using a method of our choice. We can call the train method on the parsed data, then call the predict method to use our model to predict a quantity. Below is an example of an implementation of ordinary least squares regression using stochastic gradient descent, which can be easily modified to implement other forms of regression or classification.

```
# Refer to https://spark.apache.org/docs/1.1.1/mllib-linear-methods.html
# import algorithms
from pyspark.mllib.regression import LabeledPoint, LinearRegressionWithSGD
from numpy import array

# Load and parse the data
data = sc.textFile("data/mllib/ridge-data/lpsa.data")
parsedData = data.map(parsePoint)

# Build the model
model = LinearRegressionWithSGD.train(parsedData)

# Evaluate model
valuesAndPreds = parsedData.map(lambda p: (p.label, model.predict(p.features)))
MSE = valuesAndPreds.map(lambda (v, p): (v - p)**2) \
                    .reduce(lambda x, y: x + y) / valuesAndPreds.count()
```

### 1.2.5 Creating a Decision Tree

To use a decision tree, first we must set up our inputs. We first load and parse the data, and convert it to a dataframe. We then index the labels and features, and also randomly split the data into training data and test data.

```
# Refer to https://spark.apache.org/docs/1.5.2/ml-decision-tree.html
# import algorithms
from pyspark.ml import Pipeline
from pyspark.ml.classification import DecisionTreeClassifier
from pyspark.ml.feature import StringIndexer, VectorIndexer
from pyspark.ml.evaluation import MulticlassClassificationEvaluator
from pyspark.mllib.util import MLUtils

# Load, parse, convert data to dataframe
data = MLUtils.loadLibSVMFile(sc, "data.txt").toDF()

# Index labels, adding metadata to the label column.
labelIndexer = StringIndexer(inputCol="label", outputCol="indexedLabel").fit(data)
# Identify categorical features, and index them.
featureIndexer = VectorIndexer(inputCol="features",
                               outputCol="indexedFeatures",
```

```
                                maxCategories=5).fit(data)
# Split the data into training and test sets
(trainingData, testData) = data.randomSplit([0.75, 0.25])
```

We use the DecisionTreeClassifier function to indicate the method we are using, chain the indexers together in a pipeline, and then train the model using the fit method. We use the transform method to use our trained model to make predictions.

```
# Train a DecisionTree model.
dt = DecisionTreeClassifier(labelCol="indexedLabel", featuresCol="indexedFeatures")

# Chain indexers and tree in a Pipeline
pipeline = Pipeline(stages=[labelIndexer, featureIndexer, dt])

# Train model
model = pipeline.fit(trainingData)

# Make predictions.
predictions = model.transform(testData)

# Select (prediction, true label) and compute test error
evaluator = MulticlassClassificationEvaluator(
    labelCol="indexedLabel", predictionCol="prediction", metricName="precision")
accuracy = evaluator.evaluate(predictions)
```