Chapter 9: Data Processing

The materials in this chapter and the next are from the LinkedIn Online Course *Python: Working with Predictive Analytics*. You are required to complete this course. It's 1 hour 22 minutes long, and cost $49.99 if you are not a premium member on LinkedIn, but you have a free one-month trial. You can start the free trial and finish in a month and cancel to get a free online course.

The link to the course is here
https://www.linkedin.com/learning/python-working-with-predictive-analytics

You'll receive a certificate upon completion. You are required to submit the certificate with your name on it, and it counts 10% towards your grade in FIN 695.

In this chapter, you'll learn four different data types, how to fill in missing values. You'll also learn how to convert categorical data into numbers. You'll learn to divide data into training set and test set. Finally, you'll learn how to normalize or standardize data to reduce the impact of outliers on the results.

Differentiate data types

In order to make good predictions, you need to understand the types of data we are working with.

Data can be either numerical or categorical. Numerical data can be expressed as an interval or as a ratio. Categorical data can be broken down as nominal or ordinal. I'll explain each of these with an example.

Let's say we own a landscaping business, and we need to gather data on all the different locations we work. In order for our employees to know how much tree removing equipment to take with them, we need to know how many trees are at each location. One home with a small yard may have three trees, and another larger home might have eight trees. This is the numerical data. We also need to know where each place is located. For this data, we need information like street or the city. This is the categorical data.

Nominal Data

Starting with the first type of categorical data, let's discuss the nominal scale. For our landscaping business data, let's say we have data on what the exterior color is for each house, blue, red, and green. In a nominal scale, we can only compare if the data is equal or not equal.

We cannot order, add or subtract, or multiply or divide nominal data, like blue is not larger than red, or green cannot be divided into blue.

Ordinal Data

Categorical data can be on an ordinal scale, and for this, maybe we have the addresses of each house. Door numbers can be 1210, 1211, and 1212, and so forth. In this case, we can decide if the values are the same or not. We can order the houses by their numbers, but we cannot add or subtract, or multiply or divide the ordinal data.

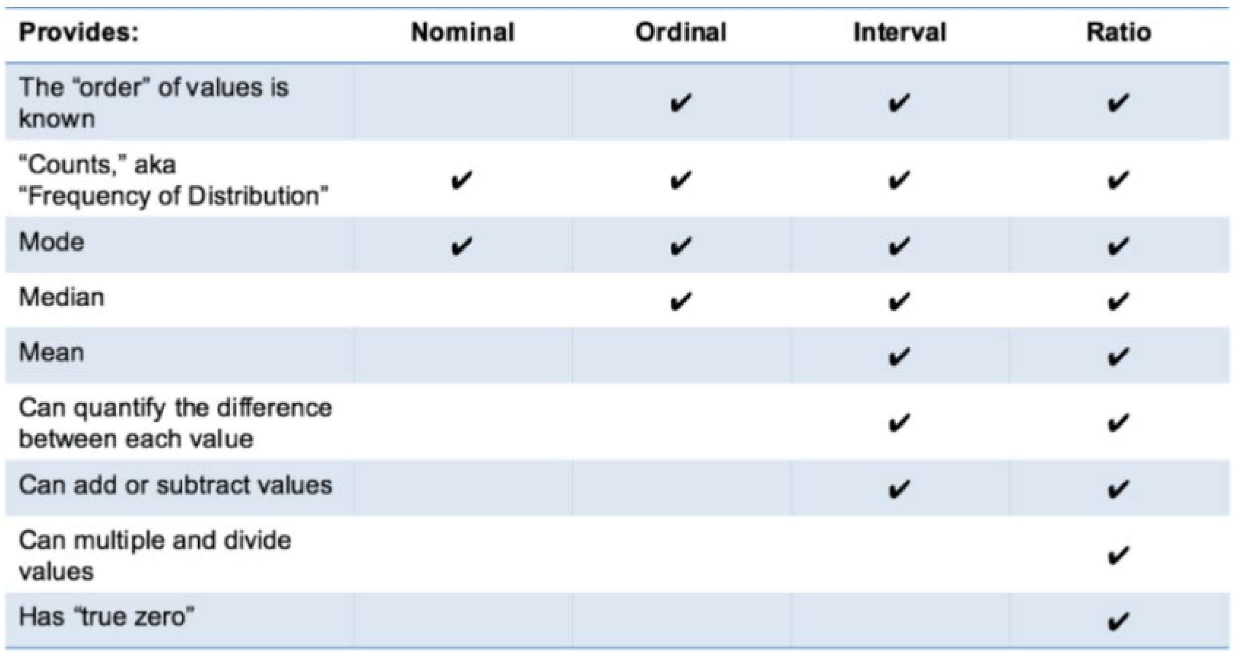
Interval Data

Numerical data on the other hand can be expressed as an interval or ratio. Let's start with the interval scale. In our houses, maybe we need to know the temperature readings at each home to know what kinds of plants will work there. Our homes can be 23, 24, and minus three degrees Celsius. We can decide if these values are the same or not, we can order the houses by temperature, and we can add, subtract the temperature. For example, the red house is 27 degrees Celsius warmer than the green one. But, interval scale does not have a true zero point. For example, zero degrees Celsius is still a temperature reading. It is not the absence of temperature. We cannot multiply or divide this data with meaning.

Ratio Data

Let's look at the other class of numerical data, which is ratio. Ratio is very similar to the interval scale, with the difference that it has a true zero point. This scale is commonly used for values that are measured in numbers, such as length, height, weight, or monetary values like cost and revenue. For example, each of our houses has a square footage, but due to the true zero point property of ratio scale, it doesn't make sense to say a house has minus 400 square feet. We can do all the operations for the ratio data. We can make equal, unequal comparison, we can order them from the largest to the smallest home, we can add and subtract, and multiply and divide this ratio scale.

Figure 1 summarizes the properties of the four types of data.



Provides:	Nominal	Ordinal	Interval	Ratio
The "order" of values is known		✔	✔	✔
"Counts," aka "Frequency of Distribution"	✔	✔	✔	✔
Mode	✔	✔	✔	✔
Median		✔	✔	✔
Mean			✔	✔
Can quantify the difference between each value			✔	✔
Can add or subtract values			✔	✔
Can multiple and divide values				✔
Has "true zero"				✔

Figure 1: Summary of data types and scale measures

In Figure 1, we see a summary of the mathematical operations we can perform with each data type. Remember the interval scale example where we had data of temperature readings at each house? Well, yes, we can compare this data, like one house's temperature is not equal to another one, and yes, we can determine if one house is warmer than another house. And yes, we can add or subtract them, and say that one house is 27 degrees warmer than the other house. But, interval scale does not make sense for us to multiply or divide this data, and have data that will help us. When working with prediction models, it's important for you to know that they cannot process categorical data. They need numbers. So, in order to work with predictive analytics, we will need to find ways to convert categorical data into numerical data.

Handle missing values

We will now start the Data Preparation step. It's 80% of what data science is. As they say, "Garbage in, garbage out," if not careful during the data prep. In real life, we seldom have completely full data sets to work with. In the Python world, missing values are represented as *NaN*, which is "not a number". Most prediction methods cannot work with missing data, thus, we need to fix the problem of missing values. We have quite a few methods to handle this. Three options we will mention here are:

First, drop the entire column where the NaN values exist.

Secondly, drop the rows with NaN values.

And finally, fill in the NaN values.

There's no right answer for every data set. One or the other may be appropriate, depending on the conditions.

In the subfolder */ExerciseFiles/01_02*, open the Python program *01_02_Finish.py* in Spyder. The program is as follows:

```
import pandas as pd
import numpy as np
from sklearn.model_selection import train_test_split
from sklearn.impute import SimpleImputer
import matplotlib.pyplot as plt

#import data
data = pd.read_csv("../Datasets/insurance.csv")

#see the first 15 lines of data
print(data.head(15))
```

Before you run the program, you should install several packages. In the Anaconda prompt (Windows) or a terminal (Mac or Linux), first activate the virtual environment, then execute the following commands:

```
conda install pandas
conda install matplotlib
conda install numpy
conda install scikit-learn
```

If you run the above program, you'll see an output of the following:

```
    age     sex     bmi  children smoker     region      charges
0    19  female  27.900         0    yes  southwest  16884.92400
1    18    male  33.770         1     no  southeast   1725.55230
2    28    male  33.000         3     no  southeast   4449.46200
3    33    male  22.705         0     no  northwest  21984.47061
4    32    male  28.880         0     no  northwest   3866.85520
5    31  female  25.740         0     no  southeast   3756.62160
```

```
6     46  female  33.440         1     no  southeast   8240.58960
7     37  female  27.740         3     no  northwest   7281.50560
8     37    male  29.830         2     no  northeast   6406.41070
9     60  female  25.840         0     no  northwest  28923.13692
10    25    male  26.220         0     no  northeast   2721.32080
11    62  female  26.290         0    yes  southeast  27808.72510
12    23    male  34.400         0     no  southwest   1826.84300
13    56  female  39.820         0     no  southeast  11090.71780
14    27    male     NaN         0    yes  southeast  39611.75770
```

As you can see the 15th observation of the *bmi* variable is missing (again, Python uses zero indexing, so the 15th observation has an index of 14). Next, we'll discuss how to handle missing values.

In the subfolder */ExerciseFiles/01_03*, open the Python program *01_03_Finish.py* in Spyder. The program is as follows:

```
--snip--
#check how many values are missing (NaN) before we apply the methods below

count_nan = data.isnull().sum() # the number of missing values for every
column

print(count_nan[count_nan > 0])


#fill in the missing values

data['bmi'].fillna(data['bmi'].mean(), inplace = True)


#check how many values are missing (NaN) - after we filled in the NaN

count_nan = data.isnull().sum() # the number of missing values for every
column

print(count_nan[count_nan > 0])
```

Let's look first how many missing values each column has. For that, let's type `count_nan=data.isnull().sum()`. Then, we will print the *count_nan* values where the *count_nan* values are larger than zero. Let's click run, and we will see that we are missing five values from the *bmi* column as we can see in the console. So this time we will be filling in `NaN` values with the mean value of the *bmi* column.

In `data['bmi'].fillna()` we are filling in the missing values with the mean value. The argument `inplace=True` will make the changes on the data frame. Okay, let's select this cell. Right click and run cell. So we just filled in the values. To make sure, we will check if we have any missing values left. You will see that we have zero missing values.

As I've mentioned earlier, there are quite a few other methods to fill in the *NaN* values. In fact, this is a very extensive topic which can be studied as its own class. Next we'll go through the *Finish_ visualize* version of the code, to learn more about the other methods to drop or fill in the *NaN* values.

Visualize data

A picture is worth a thousand words. After we've filled in the missing values, it's important to visualize the data we are working with in order to absorb the data quickly and understand the next steps for making good predictions.

In the subfolder */ExerciseFiles/01_03*, open the Python program *01_03_Finish_visualize.py* in Spyder. The program is as follows:

```
--snip--
figure, ax = plt.subplots(4,2, figsize=(12,24))
#See the distrubution of the data
sns.distplot(data['charges'],ax= ax[0,0])
sns.distplot(data['age'],ax=ax[0,1])
sns.distplot(data['bmi'],ax= ax[1,0])
sns.distplot(data['children'],ax= ax[1,1])
sns.countplot(data['sex'],ax=ax[2,0])
sns.countplot(data['smoker'],ax= ax[2,1])
sns.countplot(data['region'],ax= ax[3,0])
#visualizeing skewness
sns.pairplot(data)
#Lets look at smokers vs non-smokers on age vs charges:
sns.lmplot(x="age", y="charges", hue="smoker", data=data, palette = 'muted', 
height = 7)
plt.show(sns)
#Lets look at correlation:
```

```
corr = data.corr()
sns.heatmap(corr, cmap = 'Wistia', annot= True)
plt.show(sns)
```

Some examples of the next steps may include removal of outliers and finding trends and so on. Let's run the code and enlarge the console to see the visualization.

We use the `distplot()` function to see the distribution of numerical data. Figure 2 shows the distribution of the four variables *charges*, *age*, *bmi*, and *children*.

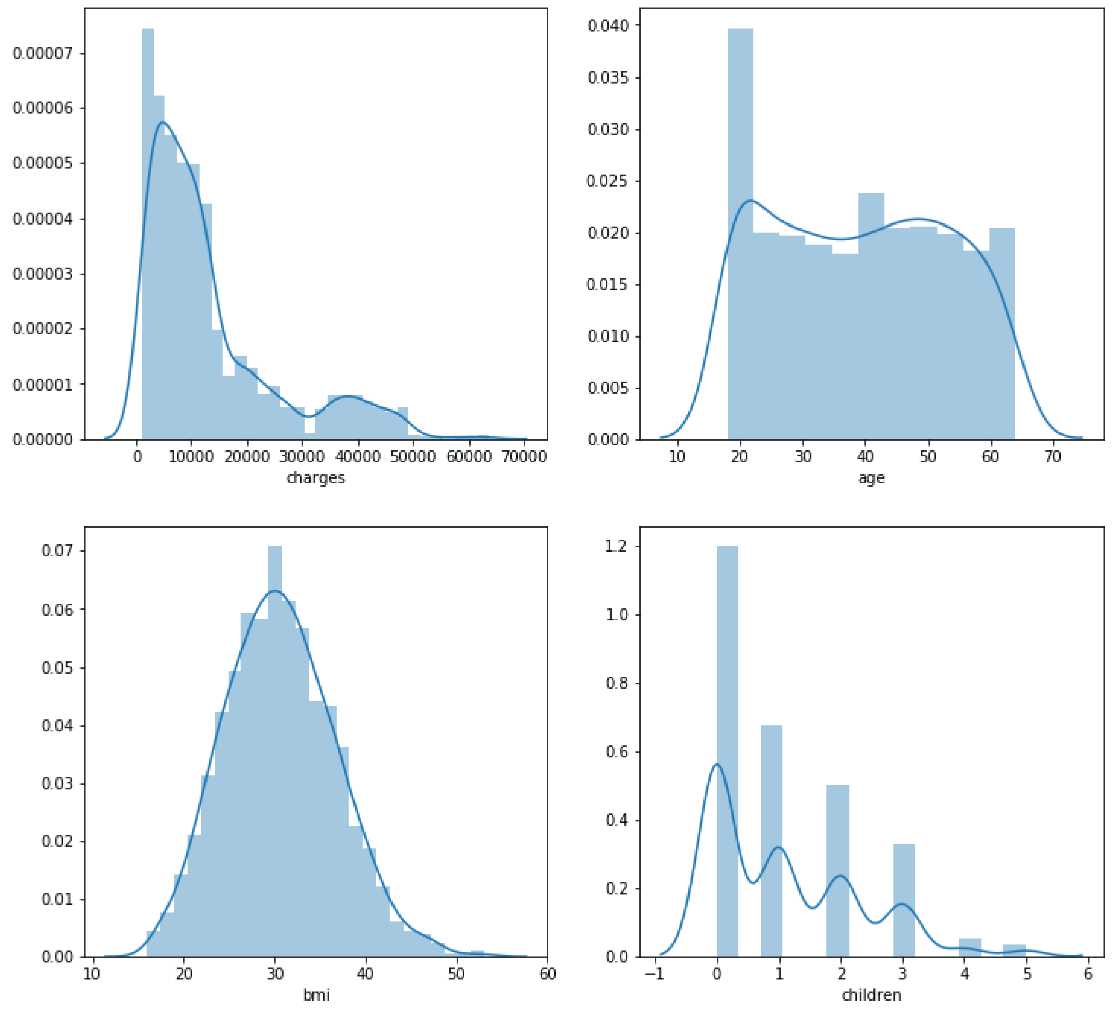


Figure 2: The distribution of the four variables charges, age, bmi, and children

Let's first see the distribution of the charges. The variable, as we see, is right skewed.

We use the `countplot()` function to see the distribution of categorical data. Figure 3 shows the distribution of the following three variables *sex*, *smoker*, and *region*.

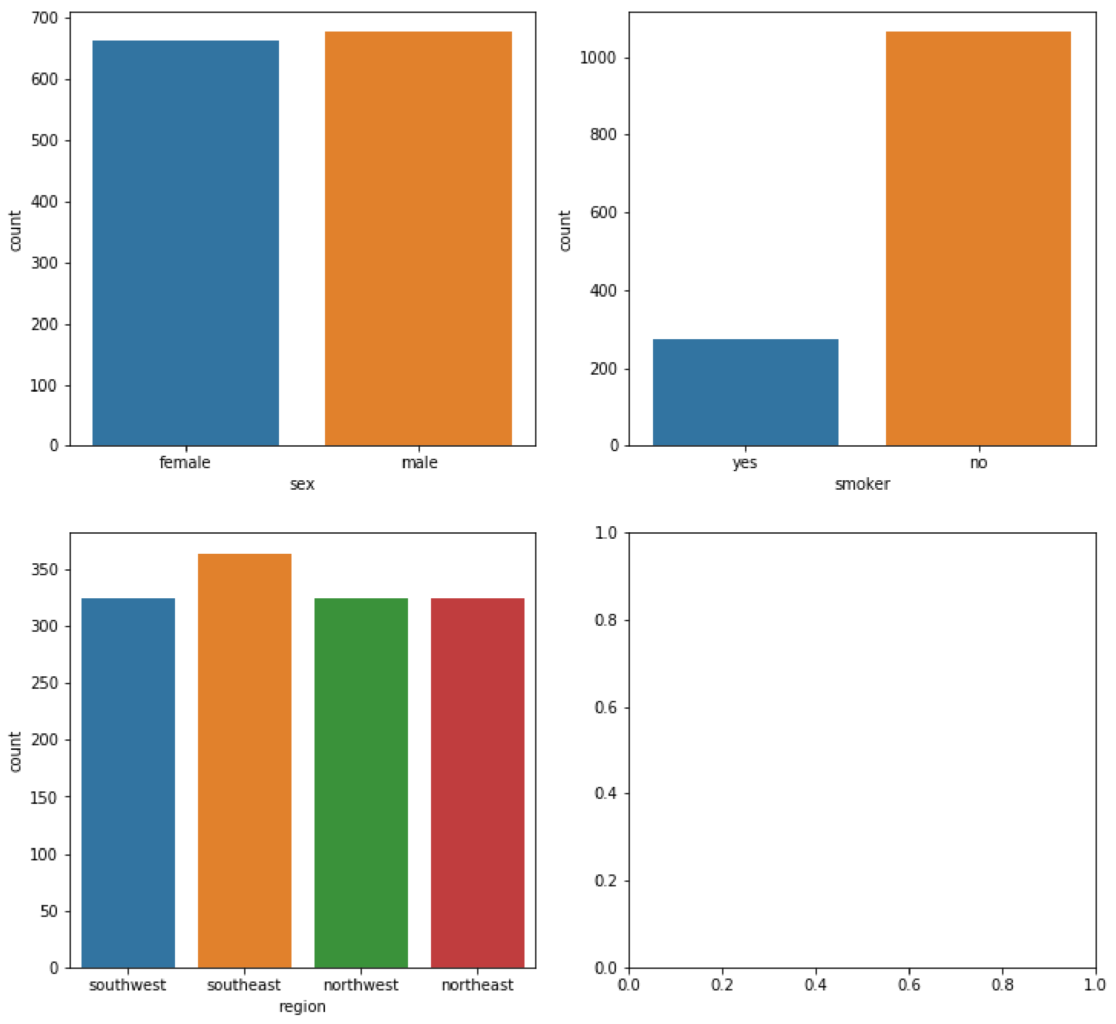


Figure 3: The distribution of the three categorical variables sex, smoker, and region

We see that we have a smaller number of smokers versus non-smokers. The variable *sex* is more or less evenly distributed between female and male. The variable *region* is more or less evenly distributed among four areas, with southeast slightly more than others.

We use the `pairplot()` function to create a plot between any pair of numerical variables in the dataset. Figure 4 shows the result with the pair-wise plot among the four variables *charges*, *age*, *bmi*, and *children*.

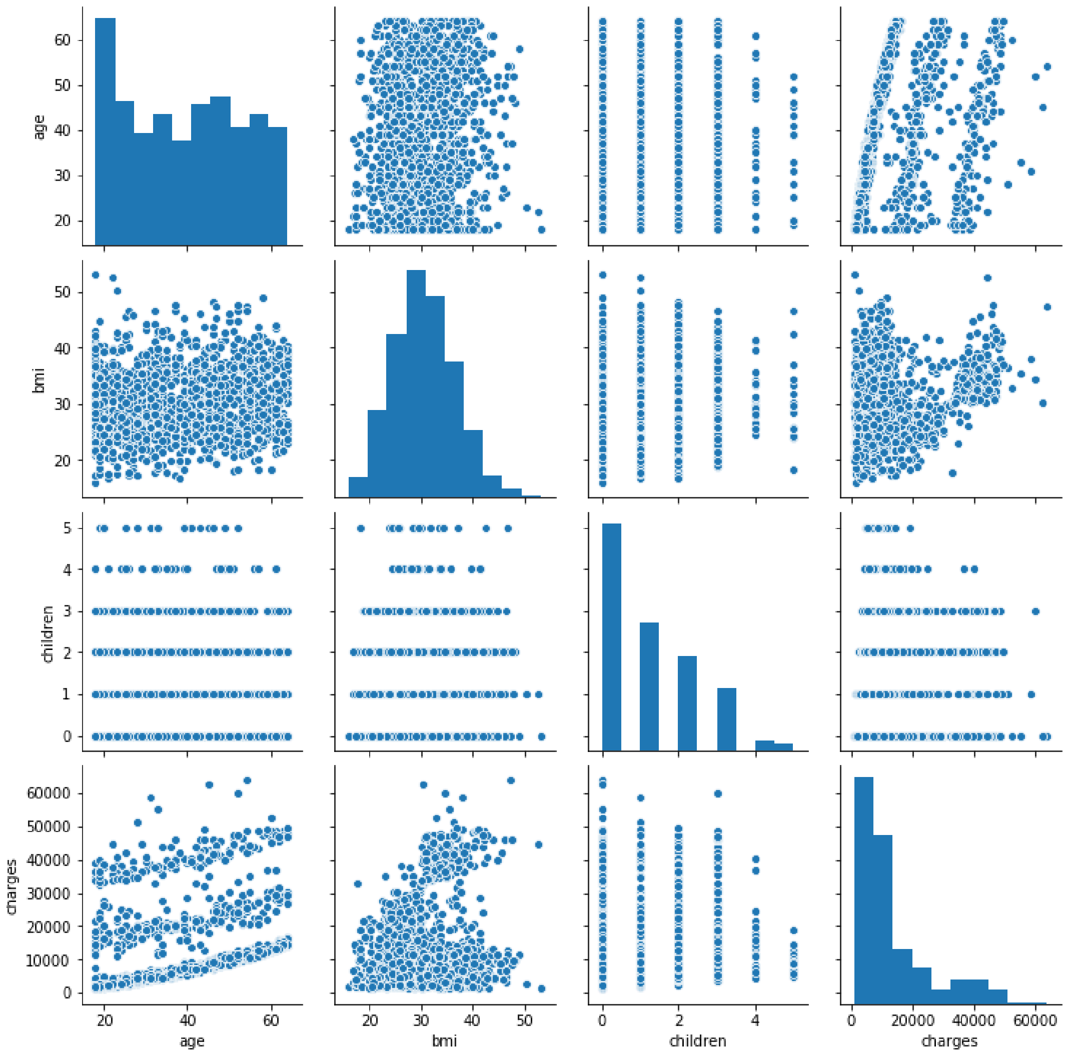


Figure 4: The pair-wise plot among the four variables *charges*, *age*, *bmi*, and *children*

In the pair plots, in the lower left corner, age versus charges, it seems to have a good correlation with layers.

We use the `lmplot()` function to create a linear model plot between *age* and *charges* among smokers and non-smokers. Figure 5 shows the result.

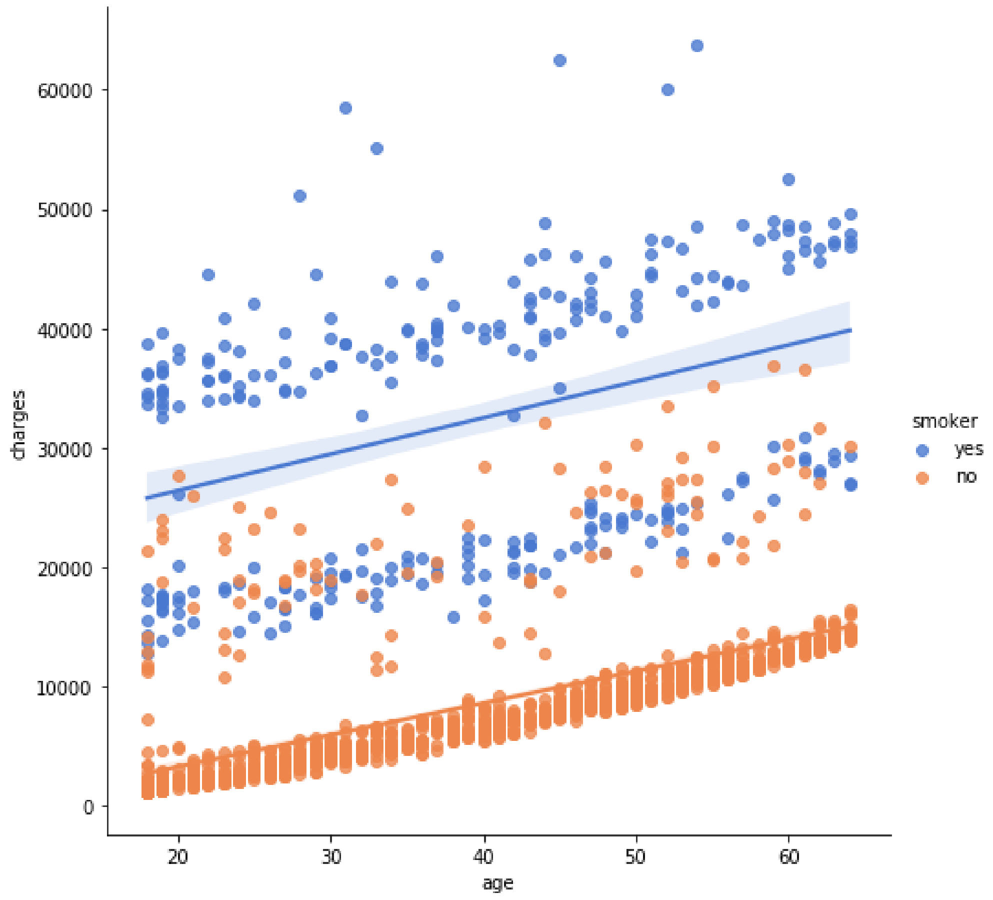


Figure 5: A linear model plot between age and charges among smokers and non-smokers

From the linear model plot, we can see that smokers clearly do have higher charges.

We use the `heatmap()` function to visualize the correlation between any pair of numerical variables in the dataset. Figure 6 shows the result.

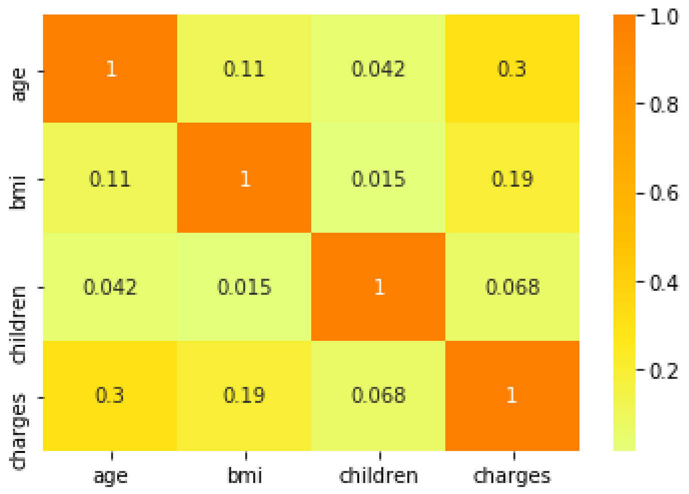


Figure 6: The correlation between any pair of numerical variables in the dataset

Looking at the correlation matrix, we can say that the biggest correlation is between age and charges, with .3 on the lower left corner. Note that the correlation matrix is symmetric since the correlation between x and y is the same as correlation between y and x. The diagonal values are 1 since the correlation between a variable and itself is always 1.

Convert categorical data into numbers

We need to convert categorical data into numbers, because prediction models only accept numerical data.

We have two ways to handle this. One is *label encoding*, the other way is *one hot encoding*. Label encoding works well if we have two distinct values. One hot encoding works well if we have three or more distinct values.

In our insurance data set we used before, we have two distinct values for smoker, yes or no. Which means we can handle with label encoder by replacing yes with one, and no with zero. Now, let's look at what happens if we have more than two distinct values. For example, colors, blue, red and green. When we applied label encoding here, we end up having numbers as zero, one, and two. And in this case, two is larger than zero, which means green is larger than blue. We cannot make such correlation between categorical values, thus we need another method.

One hot encoding is the saver here. Unlike the label encoder, we are adding new columns here labeled as the categorical data distinct values. In this case, blue, red, and green. We place one where we satisfy the presence of that color, and zero where we don't. This shows examples of how the *pandas* and *sklearn* Python libraries handle label encoding and one hot encoding.

What about code? Let's apply what we learned so far to Python code using *sklearn* library.

In the subfolder */ExerciseFiles/01_04*, open the Python program *01_04_Finish.py* in Spyder. The program is as follows:

```
--snip--
#option0: pandas factorizing: maps each category to a different integer

#create series for pandas
region = data["region"] # series
region_encoded, region_categories = pd.factorize(region)
factor_region_mapping = dict(zip(region_categories, region_encoded))
#mapping of encoded numbers and original categories.

print("Pandas factorize function for label encoding with series")
print(region[:10]) #original version
print(region_categories) #list of categories
print(region_encoded[:10]) #encoded numbers for categories
print(factor_region_mapping) # print factor mapping

#option1: pandas get_dummies: maps each category to 0 (cold) or 1 (hot)
#create series for pandas
region = data["region"] # series
region_encoded = pd.get_dummies(region, prefix='')
print("Pandas get_dummies function for one hot encoding with series")
print(region[:10]) #original version
print(region_encoded[:10]) #encoded numbers for categories

#option2: sklearn label encoding: maps each category to a different integer
#create ndarray for label encodoing (sklearn)
sex = data.iloc[:,1:2].values
smoker = data.iloc[:,4:5].values

#label encoder = le
## le for sex
le = LabelEncoder()
```

```
sex[:,0] = le.fit_transform(sex[:,0])
sex = pd.DataFrame(sex)
sex.columns = ['sex']
le_sex_mapping = dict(zip(le.classes_, le.transform(le.classes_)))
print("Sklearn label encoder results for sex:")
print(le_sex_mapping)
print(sex[:10])

## le for smoker
le = LabelEncoder()
smoker[:,0] = le.fit_transform(smoker[:,0])
smoker = pd.DataFrame(smoker)
smoker.columns = ['smoker']
le_smoker_mapping = dict(zip(le.classes_, le.transform(le.classes_)))
print("Sklearn label encoder results for smoker:")
print(le_smoker_mapping)
print(smoker[:10])

#option3: sklearn one hot encoding: maps each category to 0 (cold) or 1 (hot)

#one hot encoder = ohe

#create ndarray for one hot encodoing (sklearn)
region = data.iloc[:,5:6].values #ndarray

## ohe for region
ohe = OneHotEncoder()

region = ohe.fit_transform(region).toarray()
region = pd.DataFrame(region)
region.columns = ['northeast', 'northwest', 'southeast', 'southwest']
print("Sklearn one hot encoder results for region:")
print(region[:10])
```

We create a label encoder function for sex. The goal is to map each category to a different integer. We then fit and transform this function to sex. Let's run this code, and we will see that we have a mapping, as female will be encoded to zero, and male will be equal to one. And the result of variable for sex is as seen here in the console.

Let's now repeat label encoding for smoker as well. First, we will define the label encoder, here. And then, we will fit and transform that for smoker. Finally, we will apply one hot encoding for region.

You can have a look at the printed out data. For example, the label encoder results for *sex* is as follows:

```
Sklearn label encoder results for sex:
{'female': 0, 'male': 1}
  sex
0   0
1   1
2   1
3   1
4   1
5   0
6   0
7   0
8   1
9   0
```

As you can see, the variable *sex* now takes values of 0 or 1, with 0 representing female and 1 representing male.

You can also have a look at the label encoder results for *smoker*:

```
Sklearn label encoder results for smoker:
{'no': 0, 'yes': 1}
  smoker
0       1
1       0
2       0
3       0
4       0
5       0
6       0
7       0
```

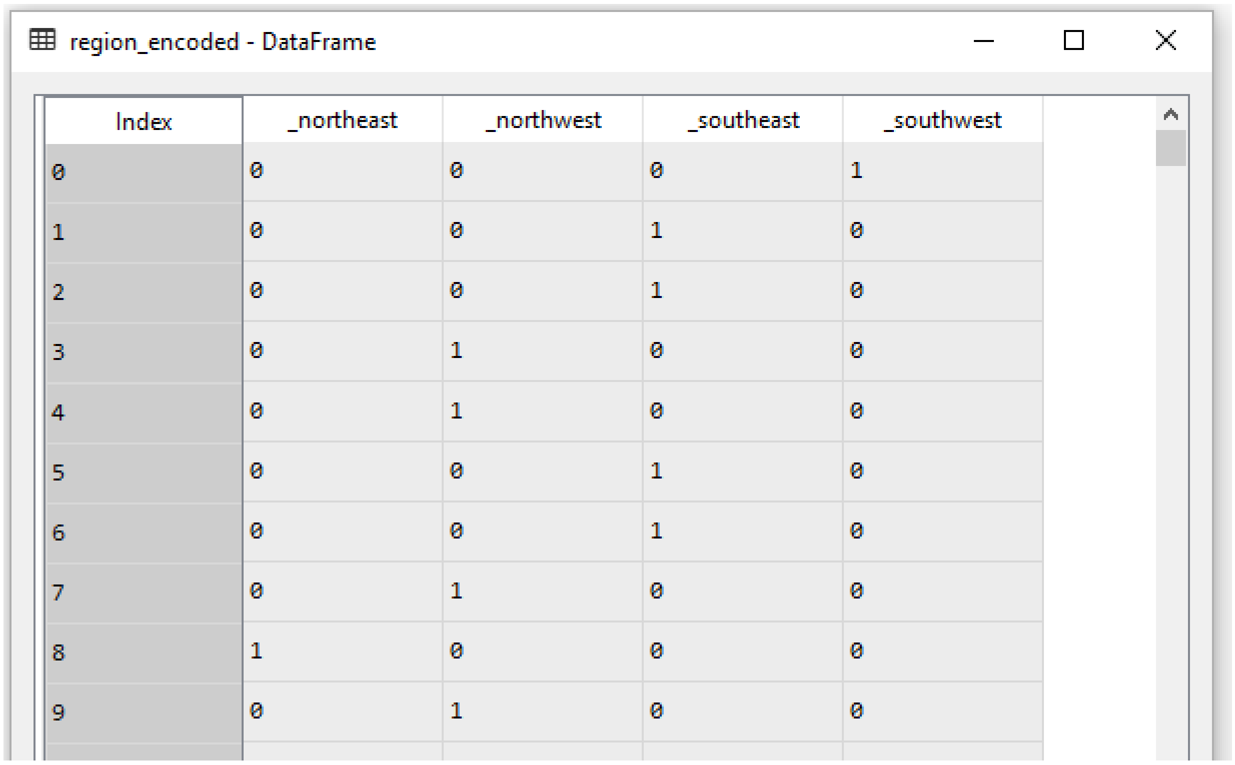
```
8           0
9           0
```

As you can see, the variable *smoker* now has values either 0 or 1, with 0 representing nonsmoker and 1 representing smoker.

NOTE: A dummy variable is a variable that takes a value of either 0 or 1.

Because the original variable *region* has four values: *southwest*, *northwest*, *southeast*, and *northeast*, four dummy variables are created when we perform the one hot encoder. The four dummy variables are _*southwest*, _*northwest*, _*southeast*, and _ *northeast* (note that there is an underscore in front of each dummy variable). If the variable *region* has an original value of *southwest*, the dummy variable _*southwest* will have a value of 1 and the other three dummy variables _*northwest*, _*southeast*, and _ *northeast* will have a value of 0. Similarly, if the variable region has an original value of *southeast*, the dummy variable _*southeast* will have a value of 1 and the other three dummy variables _*northwest*, _*northeast*, and _*southwest* will have a value of 0. And so on.

You can also have a look at the one hot encoder results for *region* from the Variable explorer pane:



region_encoded - DataFrame

Index	_northeast	_northwest	_southeast	_southwest
0	0	0	0	1
1	0	0	1	0
2	0	0	1	0
3	0	1	0	0
4	0	1	0	0
5	0	0	1	0
6	0	0	1	0
7	0	1	0	0
8	1	0	0	0
9	0	1	0	0

As you can see, the variable *region* now is now encoded with four dummy variables *_southwest*, *_northwest*, *_southeast*, and *_ northeast*.

As most predicted models accept numerical data as input, we need to convert categorical data into numerical data. This is how we do it in Python using label encoding and one hot encoding.

Divide the data into test and train

We need to divide the data into what's known as train and test. The training set contains a known output and the model learns on this data. We have the test dataset in order to test our model's prediction.

Now all the data is numerical. Imagine our data now looks like separate wooden blocks stacked up as columns, like individual data frames. Then stacking multiple data frames together gives us a final data frame. In some cases, we may want to reduce the dimensions from the data to reduce process time and increase efficiency of the model, which is called dimensionality reduction. We will not talk about this concept in this course, but it's good to know this is something you can use, especially if your data has many features and comparatively few training samples.

Now that we've put together the independent variables and assigned the response, which is a dependent variable, it's time to divide the data into test and train. Well we just put the data together, why do we need to split it? The training data set is the initial data set used to train a prediction model. The test data set however, is used to assess how well our prediction model was trained.

Imagine we are in line at a fast food restaurant and the next in line is a six year old kid. You have a pretty good idea that kid will order from the kid's menu. While this guess was a result of different samples you have witnessed in the past, which is the training data which created a prediction model in your brain. Test data on the other hand, that six year old in the line, is a new data different from the training data which is used to test the success of the prediction model. The common division percentages are two thirds for training and one third for test data.

Let's now go and look at this in Python.

In the subfolder */ExerciseFiles/01_05*, open the Python program *01_05_Finish.py* in Spyder. The program is as follows:

```
--snip--
```

```
#putting the data together:

##take the numerical data from the original data
X_num = data[['age', 'bmi', 'children']].copy()

##take the encoded data and add to numerical data
X_final = pd.concat([X_num, region, sex, smoker], axis = 1)

#define y as being the "charges column" from the original dataset
y_final = data[['charges']].copy()

#Test train split
X_train, X_test, y_train, y_test = train_test_split(X_final, y_final,
test_size = 0.33, random_state = 0 )
```

Looking at our initial numerical variables first, they are age, BMI, and children. So `X_num` will include age, BMI, and children. We will add the encoded data we prepared earlier, which are region, sex, and smoker. We will concatenate the data and call it `X_final`. We will then call the response variable charges from the data set and assign the variable name `Y_final`.

Now let's divide the data into train and test in Python. We will use the SK Learn model selection library. We will assign the X_train, X_test, Y_train, and Y_test respectively. We will provide a test size. Here in this case we will use 0.33, that is, 33% of data for test and 67% for training.

It's important that you don't use the test data for training your model because we want the model to perform well on the unseen test data.

Feature scaling

We need to apply feature scaling to our data, mainly to prevent the features with larger magnitudes from dominating the prediction model.

Let’s go back to the dinosaurs age and imagine we will need to count the number of living species in an island from top down, with a drone. When we look down from a higher distance bigger animals like dinosaurs, will be quite visible. However, it will be rather difficult to count the smaller animals, like ants. In order to have a fair glass to see all variables from the same lands, we will apply some methods. This way the prediction models will preform better.

Two methods we will discuss here are called normalization and standardization. These methods are commonly applied to the variables X, and scaling the target values Y, is commonly not required.

Let us start with the first method, normalization. Also referred as min max scaling. Subtract the minimum value from the number and divide by maximum minus minimum. Values will be rescaled to be between zero and one, unless we define a different range using the feature range hyper parameter.

In the other method, standardization, we first subtract the mean value from the number and divide by the standard deviation. In this case we do not have a specific range we are staying in, unlike the case with the normalization. This method is less affected by the outliers.

Lets do feature scaling in Python.

In the subfolder */ExerciseFiles/01_06*, open the Python program *01_06_Finish.py* in Spyder. The program is as follows:

```
--snip—
###normalized scaler (fit transform on train, fit only on test)
#n_scaler = MinMaxScaler()
#X_train = n_scaler.fit_transform(X_train.astype(np.float))
#X_test= n_scaler.transform(X_test.astype(np.float))


#standard scaler (fit transform on train, fit only on test)
s_scaler = StandardScaler()
X_train = s_scaler.fit_transform(X_train.astype(np.float))
X_test= s_scaler.transform(X_test.astype(np.float))
```

Note that we standardized the variables. If you want to use the min max scaling, remove the # in the corresponding lines of code above.

Extra Material: Deep Neural Networks

The following material is not in the LinkedIn online course.

The most successful new development in the field of machine learning is deep neural networks (DNN). Prof Geoffrey Hinton at the University of Toronto and his coauthors published a paper in 2006 showing how to train a deep neural network capable of recognizing handwritten digits with high accuracy. Training a deep neural network was considered impossible at the time before and most researchers had abandoned the idea since the 1990s. Now it’s the most powerful tool in the field of Machine Learning.

DNN is now powering many of today’s high-tech products: ranking your web search results, recommending videos, and beating the world champion at the game of Go. It’s also the most promising tool for a self-driving car.

Researchers and practitioners have also applied DNN to the business field in recent years, including algorithm trading in many hedge funds. In this chapter, I will use some very simple examples to show you how DNN works and how to build your own neural network.

Anatomy of a Neural Network

Despite its widespread use, many people complain that DNNs are like black boxes and hard to understand. Indeed, nowadays, when building a neural network, you can simply use existing methods in the TensorFlow library without fully understanding what’s going on within the model.

To look under the hood and have a better understanding of the logic behind a neural network, let’s create one from scratch and dissect its components.

The neural network is trying to mimic human brains, which consists of billions of neurons that are connected by synapses. DNN models this process by creating a neural network on a computer, which consists of an input layer, an output layer, and some hidden layers in between. The powerful DNNs usually have many hidden layers, hence the name “Deeping neural network.”

We’ll start with a shallow one with no hidden layers, just one input layer with and one output layer.

Your First Neural Network

The problem we're trying to solve is as follows. Suppose that you have a list x, with values [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]. You also have ten values of y, [5, 7, 9, 11, 13, 15, 17, 19, 21, 23]. You may have noticed a linear relation between x and y in the form of

$$y = 2X + 3$$

You can build a simple neural network, use the ten pairs of data to train, and see if the model can detect the simple linear relationship.

Install tensorflow

First, pip install the *tensorflow* library in your virtual environment by using the following command:

```
pip install tensorflow
```

Follow the instructions to finish installation.

After that, enter the following lines of code in Spyder, and save it as *first_nn.py*.

```
import numpy as np
import tensorflow as tf
from tensorflow import keras

X = np.array([1,2,3,4,5,6,7,8,9,10], dtype=float)
y = np.array([5,7,9,11,13,15,17,19,21,23], dtype=float)

model = keras.models.Sequential()
model.add(keras.layers.Dense(1,input_shape=[1]))
model.compile(loss='mean_squared_error',
              optimizer=keras.optimizers.Adam(lr=0.1))

model.fit(X, y, epochs=1000)
pred = model.predict([[11],
                      [12],
                      [13],
                      [14],
                      [15]])
print(pred)
```

If you run the script, you'll get an import that looks like below

```
--snip--
Epoch 997/1000
1/1 [==============================] - 0s 0s/step - loss: 1.2392e-11
Epoch 998/1000
1/1 [==============================] - 0s 0s/step - loss: 1.2392e-11
Epoch 999/1000
1/1 [==============================] - 0s 0s/step - loss: 1.2392e-11
Epoch 1000/1000
1/1 [==============================] - 0s 0s/step - loss: 1.2392e-11
[[25.000006]
 [27.000008]
 [29.000008]
 [31.00001 ]
 [33.00001 ]]
```

At the end, you'll see the prediction of the model: the model predicts that for X = 11, 12, 13, 14, and 15, the predicted y values are around 25, 27, 29, 31, and 33, respectively, and the relation is almost exactly

$$y = 2X + 3$$

Wow, that is cool! How did a single layer of neural network manage to do that?

NOTE: The results you get may differ slightly from the above results due to the stochastic nature of neural networks. I'll explain later in this chapter the source of the randomness in results and how you can fix the random state so that the neural network generates replicable results.

How Does A Neural Network Come Up with A Solution?

To come up with the correct answer, the neural network model follows the steps below:

1. The model takes a batch of the examples from the training dataset, put them in the input layer.

2. The examples are passed though from the input layer to the first layer of neurons. Each neuron is assigned a weight. By default, a bias neuron is also assigned a weight and added to the layer.
3. Based on the values of the examples and the weights, the model calculates the output using a formula.
4. Compare the output with the actual values and calculate the loss based on the loss function.
5. Use the optimizer to adjust weights based on the losses from the previous iteration;

Now let's explain each step with a little more detail. Before we do that, you can go to the tensorflow documentation website and look at the Dense layer that we use in this model: https://www.tensorflow.org/api_docs/python/tf/keras/layers/Dense

A dense layer is a neural network layer in which every neuron is connected to all other neurons in the next layer. It's also called a fully-connected layer. The layer has the following arguments:

```
tf.keras.layers.Dense(
    units, activation=None, use_bias=True,
    kernel_initializer='glorot_uniform',
    bias_initializer='zeros', kernel_regularizer=None,
    bias_regularizer=None, activity_regularizer=None, kernel_constraint=None,
    bias_constraint=None, **kwargs
)
```

units: The first argument is the number of neurons in the layer. In our example above, we use 1. But it's common to use tens or hundreds of neurons in each layer.

use_bias: The second argument is weather to use a bias neuron in the layer. We ignored the argument because the default option is to use the bias neuron. We choose to use it since there is a constant intercept term in the linear relation.

kernel_initializer: As we mentioned above, the model first assigns a weight to each neuron in the first layer. The *kernel_initializer* argument specifies how the initial weights are assigned. We choose the default setting, which means the model randomly generates a weight from the *glorot_uniform* distribution. The range of the uniform distribution is *[-limit, limit]*, where *limit = sqrt(6/(fin_in+fin_out))*. In our case, both *fin_in* and *fin_out* have values of 1 since we have one input and one out put. So the weight is randomly drawn from a uniform distribution in the range [-1.732, 1.732].

bias_initializer: The initial weight on the bias neuron, which is set to 0 by default.

kernel_regularizer: You can use l1 or l2 regularizers to control the magnitude of the weights so that the model doesn't overfit the model.

The model calculates the output as follows:

$$Ouput = weight * \text{input} + \text{bias}$$

We'll look at the exact numbers in the neural network example you just built.

Looking under the hood

We'll dive deeper into the model and see different parts of the neural network in detail.

Open Spyder, enter the following lines of code, and save it as *dissect_nn.py*.

```
import numpy as np
import tensorflow as tf
from tensorflow import keras
from tensorflow.keras.utils import plot_model

tf.random.set_seed(0)
X = np.array([1,2,3,4,5,6,7,8,9,10], dtype=float)
y = np.array([5,7,9,11,13,15,17,19,21,23], dtype=float)

model = keras.models.Sequential()
model.add(keras.layers.Dense(1,input_shape=[1]))
model.compile(loss='mean_squared_error',
              optimizer=keras.optimizers.Adam(lr=0.1))
# print model summary
print(model.summary())
# Save the plotted model
plot_model(model, to_file="first_nn.png",show_shapes=True)
# Print out the initial weight and bias
```

```
w0, bias0 = model.layers[-1].get_weights()
print("the initial weight is", w0)
print("the initial bias is", bias0)
# Train the model
model.fit(X, y, epochs=1000, verbose=0)
# Use the model to predicdt
X_test = [[11], [12], [13], [14], [15]]
pred = model.predict(X_test)
print("the model predictions are\n", pred)
# Print out final weight and bias
w1, bias1 = model.layers[-1].get_weights()
print("the final weight is", w1)
print("the final bias is", bias1)
# Use final weight and bias to calcualte predictions
manual_pred = bias1 + w1 * X_test
print("the manually calcualted predictions are\n", manual_pred)
```

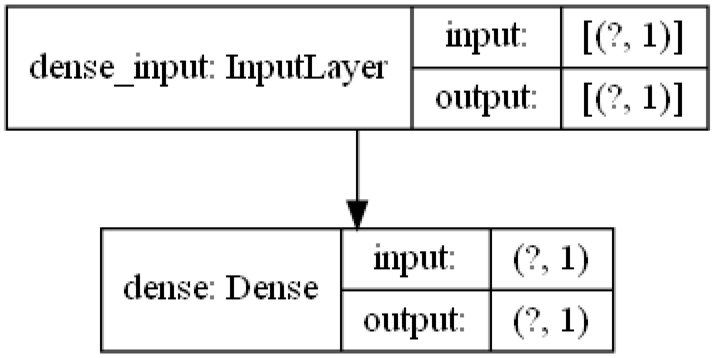
I have set the random state to 0 so that you'll see the same output as I do.

Once the model is built, I use the *model.summary()* to generate a summary of the model, which is shown below:

```
Model: "sequential"
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
dense (Dense)                (None, 1)                 2
=================================================================
Total params: 2
Trainable params: 2
Non-trainable params: 0
_________________________________________________________________
None
```

The model is a sequential model from the keras API. The API provides an interface for the Tensorflow library to make the creation of neural networks easy. The output shows that the model uses a dense layer with output shape of 1 and the number of parameters is 2 (weight and bias).

We use the *plot_model()* method to plot the model in a graph. The graph is saved as a picture first_nn.png in your chapter folder. If you open the graph, it look as follows:



The graphs shows that the input is passed through an input layer, with the shape of (?,1), which means it can have any number of observations, and each observation has one feature. The output has the same number of observations and each observation also have one value.

Before we train the model, we print out the initial weight and bias, which has the following output:

```
the initial weight is [[-0.7206192]]
the initial bias is [0.]
```

The initial weight is -0.7206192, a random drawn from the uniform distribution [-1.732, 1.732]. The initial bias is set to 0 as the default value of the dense layer.

We then train the model using the values of X and y as before. Note here I put an argument *verbose=0* in the *fit()* method. As a result, you don't see the statistics from each epoch of training. We then use the *predict()* method to predict y values when the values of X are [[11], [12], [13], [14], [15]]. The results are:

```
the model predictions are
 [[25.000023]
 [27.000029]
 [29.000034]
 [31.000038]
 [33.000042]]
```

As you can see, the model did a very good job at predicting the y values.

After we train the model, we print out the final weight and bias, which has the following output:

```
the final weight is [[2.0000052]]
the final bias is [2.9999652]
```

The final weight and bias are very close to 2 and 3, which means the DNN has correctly identified the relation between y and X as y=2X+3.

You can also manually calculate the model predictions based on the inputs, final weight, and final bias. For example, when X=11, the prediction should be

$$Ouput = weight * \text{input} + \text{bias}$$
$$= 2.0000052 * 11 + 2.9999652$$
$$= 25.000023$$

The script prints out the results as follows:

```
the manually calcualted predictions are
[[25.00002289]
 [27.00002813]
 [29.00003338]
 [31.00003862]
 [33.00004387]]
```

They are almost identical to the outputs using model predictions, with some small discrepancies due to number rounding

Use DNN to Predict Insurance Charges

Below I am using DNN to predict the insurance charges, the code *insurance_dnn.py* is on Canvas.

```
from sklearn.model_selection import train_test_split
import pandas as pd
import numpy as np
from sklearn.preprocessing import StandardScaler
from sklearn.preprocessing import OneHotEncoder
from sklearn.preprocessing import LabelEncoder
from sklearn.metrics import r2_score
import tensorflow as tf

data = pd.read_csv("insurance.csv")
data['bmi'].fillna(data['bmi'].mean(), inplace = True)

sex = data.iloc[:,1:2].values
smoker = data.iloc[:,4:5].values

le = LabelEncoder()
sex[:,0] = le.fit_transform(sex[:,0])
sex = pd.DataFrame(sex)
sex.columns = ['sex']
le_sex_mapping = dict(zip(le.classes_, le.transform(le.classes_)))

le = LabelEncoder()
smoker[:,0] = le.fit_transform(smoker[:,0])
smoker = pd.DataFrame(smoker)
smoker.columns = ['smoker']
le_smoker_mapping = dict(zip(le.classes_, le.transform(le.classes_)))
print("Sklearn label encoder results for smoker:")
print(le_smoker_mapping)
print(smoker[:10])

region = data.iloc[:,5:6].values #ndarray
ohe = OneHotEncoder()
region = ohe.fit_transform(region).toarray()
region = pd.DataFrame(region)
region.columns = ['northeast', 'northwest', 'southeast', 'southwest']

X_num = data[['age', 'bmi', 'children']].copy()
X_final = pd.concat([X_num, region, sex, smoker], axis = 1)
y_final = data[['charges']].copy()

# Split
```

```
X_train, X_test, y_train, y_test = train_test_split(X_final, y_final,
test_size = 0.33, random_state = 0 )
# standarize variables
s_scaler = StandardScaler()
X_train = s_scaler.fit_transform(X_train.astype(np.float))
X_test= s_scaler.transform(X_test.astype(np.float))

# DNN method
X_train = np.array(X_train).reshape((-1,9))
print(X_train.shape)

y_train = np.array(y_train).reshape((-1,1))
print(y_train.shape)

X_test = np.array(X_test).reshape((-1,9))
print(X_test.shape)

y_test = np.array(y_test).reshape((-1,1))

print(y_test.shape)

dnn = tf.keras.Sequential()
dnn.add(tf.keras.layers.Dense(128,input_shape=[9],activation="relu"))
dnn.add(tf.keras.layers.Dense(64,activation="relu"))
dnn.add(tf.keras.layers.Dense(32,activation="relu"))
dnn.add(tf.keras.layers.Dense(1, activation='relu'))

dnn.compile(optimizer="adam", loss="mean_squared_error")

dnn.fit(X_train, y_train,verbose=1, epochs=50)

y_train_pred_dnn=dnn.predict(X_train)

y_test_pred_dnn=dnn.predict(X_test)

y_true=np.array(y_test)
dif=y_true-y_test_pred_dnn
```

```
ss_total=np.square(y_true-np.mean(y_true))
ss_res = np.square(dif)

rsq = 1-sum(ss_res)/sum(ss_total)
print("R-squared based on 1-sum(ss_res)/sum(ss_total) is", rsq)

print("R-squared for training data is", r2_score(y_train, y_train_pred_dnn))
print("R-squared based on sklearn r2_score is", r2_score(y_true,
y_test_pred_dnn))
```

The output shows that

```
R-squared for training data is 0.776695240963212

R-squared for testing data is 0.8261840872837964
```

This means that in the testing dataset, 82.62% of the variation can be explained by using the DNN prediction model.

In Chapter 10, I will explain more on the above program, and how DNN performs compared to other machine learning methods.