Machine Learning with Python

Machine Learning

*The Regression/Estimation technique is used for predicting a continuous value. For example, predicting things like the price of a house based on its characteristics, or to estimate the Co2 emission from a car’s engine. A Classification technique is used for Predicting the class or category of a case, for example, if a cell is benign or malignant, or whether or not a customer will churn. Clustering groups of similar cases, for example, can find similar patients, or can be used for customer segmentation in the banking field. Association technique is used for finding items or events that often co-occur, for example, grocery items that are usually bought together by a particular customer. Anomaly detection is used to discover abnormal and unusual cases, for example, it is used for credit card fraud detection. Sequence mining is used for predicting the next event, for instance, the click-stream in websites. Dimension reduction is used to reduce the size of data. And finally, recommendation systems, this associates people's preferences with others who have similar tastes, and recommends new items to them, such as books or movies. We will cover some of these techniques in the next videos.*

*By this point, I’m quite sure this question has crossed your mind, “What is the difference between these buzzwords that we keep hearing these days, such as Artificial intelligence (or AI), Machine Learning and Deep Learning?” Well, let me explain what is different between them. In brief, AI tries to make computers intelligent in order to mimic the cognitive functions of humans. So, Artificial Intelligence is a general field with a broad scope including: Computer Vision, Language Processing, Creativity, and Summarization. Machine Learning is the branch of AI that covers the statistical part of artificial intelligence. It teaches the computer to solve problems by looking at hundreds or thousands of examples, learning from them, and then using that experience to solve the same problem in new situations. And Deep Learning is a very special field of Machine Learning where computers can actually learn and make intelligent decisions on their own. Deep learning involves a deeper level of automation in comparison with most machine learning algorithms. Now that we’ve completed the introduction to Machine Learning, subsequent videos will focus on reviewing two main components: First, you’ll be learning about the purpose of Machine Learning and where it can be applied in the real world; and Second, you’ll get a general overview of Machine Learning topics, such as supervised vs unsupervised learning, model evaluation and various Machine Learning algorithms. So now that you have a sense with what’s in store on this journey, let’s continue our exploration of Machine Learning! Thanks for watching!*

|  |
| --- |
| Python Packages for Data Science  A Python library is a collection of functions and methods that allow you to perform lots of actions without writing any code. The libraries usually contain built in modules providing different functionalities which you can use directly. There are extensive libraries offering a broad range of facilities. Libraries are broadly divided in three groups.  Scientifics computing libraries:   * + **Pandas:** offers *data structure and tools* for effective data manipulation and analysis. Offers data structure and tools for effective data manipulation and analysis. It provides fast access to structured data. The primary instrument of Pandas is a two-dimensional table consisting of columns and rows labels which are called a DataFrame. It is designed to provide an easy indexing function   + **Numpy:** Uses arrays as their inputs and outputs. It can be extended to objects for matrices, and with a little change of coding, developers perform fast array processing Arrays & Matrices   + **SciPy:** includes functions for some advanced math problems as listed here, as well as data visualization. Integrals, solving differential equations and optimisation   Visualization libraries: *These libraries enable you to create graphs, charts and maps*   * + **Matplotlib:** The Matplotlib package is the most well-known library for data visualization. The graphs are also highly customizable.   + **Seaborn:** Another high-level visualization library is Seaborn. it is based on Matplotlib. It's very easy to generate various plots such as heat maps, time series and violin plots.   Machine learning algorithms: we're able to develop a model using our data set and obtain predictions. Here we introduce two packages   * + **Scikit-learn:** the Scikit-learn library contains tools statistical modelling, including regression, classification, clustering, and so on. This library is built on NumPy, SciPy and Matplotib.   + **Statsmodels:** is also a Python module that allows users to explore data, estimate statistical models, and perform statistical tests. |
| Simple Linear Regression  *In this notebook, we learn how to use scikit-learn to implement simple linear regression. We download a dataset that is related to fuel consumption and Carbon dioxide emission of cars. Then, we split our data into training and test sets, create a model using training set, evaluate your model using test set, and finally use model to predict unknown value.*  # Importing Needed packages  import matplotlib.pyplot as plt  import pandas as pd  import pylab as pl  import numpy as np  %matplotlib inline  # Downloading Data, To download the data, we will use *!wget* to download it from IBM Object Storage.  !wget -O FuelConsumption.csv <https://s3-api.us-geo.objectstorage.softlayer.net/cf-courses-data/CognitiveClass/ML0101ENv3/labs/FuelConsumptionCo2.csv>  Understanding the Data – FuelConsumption.csv:  *We have downloaded a fuel consumption dataset, FuelConsumption.csv, which contains model-specific fuel consumption ratings and estimated carbon dioxide emissions for new light-duty vehicles* [*for retail sale in Canada*](http://open.canada.ca/data/en/dataset/98f1a129-f628-4ce4-b24d-6f16bf24dd64)*.*  Modelyear e.g. 2014, Make e.g. Acura, Model e.g. ILX, Vehical Class e.g. SUV, Enfine Size e.g. 4.7, Cylinders e.g 6, Transmission e.g. A6, Fuel Consumption in City (L/100 km) e.g. 9.9, Fuel Consumption in Hwy (L/100 km) e.g. 8.9, Fuel Consumption Comb (L/100 km) e.g. 9.2, CO2 Emission (g/km) e.g. 182 --> low --> 0  # Reading the data  df = pd.read\_csv("FuelConsumption.csv")  ​df.head() # take a look at the dataset  ​# Data Exploration – Lets first have a descriptive exploration on our data.  df.describe() # summarize the data   |  |  |  |  |  |  |  |  |  | | --- | --- | --- | --- | --- | --- | --- | --- | --- | |  | Modelyear | Enginesize | Cylinders | Fuelcons.  City | Fuelcons.  Hwy | Fuelcons.  Comb | Fuelcons.  Comb\_Mpg | Co2 Emissions | | count | 1067 | 1067 | 1067 | 1067 | 1067 | 1067 | 1067 | 1067 | | mean | 2014 | 3.346298 | 5.794752 | 13.29653 | 9.474602 | 11.58088 | 26.441425 | 256.2287 | | std | 0 | 1.415895 | 1.797447 | 4.101253 | 2.79451 | 3.485595 | 7.468702 | 63.3723 | | min | 2014 | 1 | 3 | 4.6 | 4.9 | 4.7 | 11 | 108 | | 25% | 2014 | 2 | 4 | 10.25 | 7.5 | 9 | 21 | 207 | | 50% | 2014 | 3.4 | 6 | 12.6 | 8.8 | 10.9 | 26 | 251 | | 75% | 2014 | 4.3 | 8 | 15.55 | 10.85 | 13.35 | 31 | 294 |   # Lets select some features to explore more  cdf = df[['ENGINESIZE','CYLINDERS','FUELCONSUMPTION\_COMB','CO2EMISSIONS']]  cdf.head(9)   |  |  |  |  |  | | --- | --- | --- | --- | --- | |  | ENGINESIZE | CYLINDERS | FUELCONSUMPTION\_COMB | CO2EMISSIONS | | 0 | 2.0 | 4 | 8.5 | 196 | | 1 | 2.4 | 4 | 9.6 | 221 | | 2 | 1.5 | 4 | 5.9 | 136 | | 3 | 3.5 | 6 | 11.1 | 255 | | 4 | 3.5 | 6 | 10.6 | 244 | | 5 | 3.5 | 6 | 10.0 | 230 | | 6 | 3.5 | 6 | 10.1 | 232 | | 7 | 3.7 | 6 | 11.1 | 255 | | 8 | 3.7 | 6 | 11.6 | 267 |   # We can plot each of these features:  viz = cdf[['CYLINDERS','ENGINESIZE','CO2EMISSIONS','FUELCONSUMPTION\_COMB']]  viz.hist()  plt.show()  # Now, lets plot each of these features vs the Emission, to see how linear is their relation:  plt.scatter(cdf.FUELCONSUMPTION\_COMB, cdf.CO2EMISSIONS, color='blue')  plt.xlabel("FUELCONSUMPTION\_COMB")  plt.ylabel("Emission")  plt.show()  plt.scatter(cdf.ENGINESIZE, cdf.CO2EMISSIONS, color='blue')  plt.xlabel("Engine size")  plt.ylabel("Emission")  plt.show()  # Practice: plot CYLINDER vs the Emission, to see how linear is their relation:    plt.scatter(cdf.CYLINDERS, cdf.CO2EMISSIONS, color='blue')  plt.xlabel("Cylinders")  plt.ylabel("Emission")  plt.show()  Creating train and test dataset  *Train/Test Split involves splitting the dataset into training and testing sets respectively, which are mutually exclusive. After which, you train with the training set and test with the testing set. This will provide a more accurate evaluation on out-of-sample accuracy because the testing dataset is not part of the dataset that have been used to train the data. It is more realistic for real world problems.*  *This means that we know the outcome of each data point in this dataset, making it great to test with! And since this data has not been used to train the model, the model has no knowledge of the outcome of these data points. So, in essence, it is truly an out-of-sample testing.*  *Lets split our dataset into train and test sets, 80% of the entire data for training, and the 20% for testing. We create a mask to select random rows using np.random.rand() function:*  np.random.rand(len(df)) < 0.8  msk = np.random.rand(len(df)) < 0.8  train = cdf[msk]  test = cdf[~msk]  Simple Regression Model  *Linear Regression fits a linear model with coefficients 𝜃=(𝜃1,...,𝜃𝑛) to minimize the 'residual sum of squares' between the independent x in the dataset, and the dependent y by the linear approximation.*  # Train data distribution  plt.scatter(train.ENGINESIZE, train.CO2EMISSIONS, color='blue')  plt.xlabel("Engine size")  plt.ylabel("Emission")  plt.show()  # Modelling, Using *sklearn* package to model data.  from sklearn import linear\_model  regr = linear\_model.LinearRegression()  *As mentioned before, Coefficient and Intercept in the simple linear regression, are the parameters of the fit line. Given that it is a simple linear regression, with only 2 parameters, and knowing that the parameters are the intercept and slope of the line, sklearn can estimate them directly from our data. Notice that all of the data must be available to traverse and calculate the parameters.*  train\_x = np.asanyarray(train[['ENGINESIZE']])  train\_y = np.asanyarray(train[['CO2EMISSIONS']])  regr.fit (train\_x, train\_y)  # The coefficients  print ('Coefficients: ', regr.coef\_)  print ('Intercept: ',regr.intercept\_)  Coefficients: [[39.04229689]]  Intercept: [125.08561806]  # Plot outputs, we can plot the fit line over the data:  plt.scatter(train.ENGINESIZE, train.CO2EMISSIONS, color='blue')  plt.plot(train\_x, regr.coef\_[0][0]\*train\_x + regr.intercept\_[0], '-r')  plt.xlabel("Engine size")  plt.ylabel("Emission")  # Evaluation  *we compare the actual values and predicted values to calculate the accuracy of a regression model. Evaluation metrics provide a key role in the development of a model, as it provides insight to areas that require improvement.*  *There are different model evaluation metrics, lets use MSE here to calculate the accuracy of model based on the test set:*   * Mean absolute error: It is the mean of the absolute value of the errors. This is the easiest of the metrics to understand since it’s just average error. * Mean Squared Error (MSE): Mean Squared Error (MSE) is the mean of the squared error. It’s more popular than Mean absolute error because the focus is geared more towards large errors. This is due to the squared term exponentially increasing larger errors in comparison to smaller ones. * Root Mean Squared Error (RMSE): This is the square root of the Mean Square Error. * R-squared is not error, but is a popular metric for accuracy of your model. It represents how close the data are to the fitted regression line. *The higher the R-squared, the better the model fits your data. Best possible score is 1.0 and it can be negative (because the model can be arbitrarily worse).*   # Compute model evaluation metrics  from sklearn.metrics import r2\_score  ​  test\_x = np.asanyarray(test[['ENGINESIZE']])  test\_y = np.asanyarray(test[['CO2EMISSIONS']])  test\_y\_hat = regr.predict(test\_x)  ​  print("Mean absolute error: %.2f" % np.mean(np.absolute(test\_y\_hat - test\_y)))  print("Residual sum of squares (MSE): %.2f" % np.mean((test\_y\_hat - test\_y) \*\* 2))  print("R2-score: %.2f" % r2\_score(test\_y\_hat , test\_y) )  Mean absolute error: 24.62  Residual sum of squares (MSE): 1003.04  R2-score: 0.69 |
| Multiple Regression Model  *In reality, there are multiple variables that predict the Co2emission. When more than one independent variable is present, the process is called multiple linear regression. For example, predicting co2emission using FUELCONSUMPTION\_COMB, EngineSize and Cylinders of cars. The good thing here is that Multiple linear regression is the extension of simple linear regression model.*  *Lets split our dataset into train and test sets, 80% of the entire data for training, and the 20% for testing. We create a mask to select random rows using np.random.rand() function:*  np.random.rand(len(df)) < 0.8; msk = np.random.rand(len(df)) < 0.8; train = cdf[msk]; test = cdf[~msk]  from sklearn import linear\_model  regr = linear\_model.LinearRegression()  x = np.asanyarray(train[['ENGINESIZE','CYLINDERS','FUELCONSUMPTION\_COMB']])  y = np.asanyarray(train[['CO2EMISSIONS']])  regr.fit (x, y)  # The coefficients  print ('Coefficients: ', regr.coef\_); print ('Intercept: ',regr.intercept\_)  Coefficients: [[9.67722355 8.3572085 9.55760278]]; Intercept: [64.06275301]  *As mentioned before, Coefficient and Intercept, are the parameters of the fit line. Given that it is a multiple linear regression, with 3 parameters, and knowing that the parameters are the intercept and coefficients of hyperplane, sklearn can estimate them from our data. Scikit-learn uses plain Ordinary Least Squares method to solve this problem.*  Ordinary Least Squares (OLS)  *OLS is a method for estimating the unknown parameters in a linear regression model. OLS chooses the parameters of a linear function of a set of explanatory variables by minimizing the sum of the squares of the differences between the target dependent variable and those predicted by the linear function. In other words, it tries to minimizes the sum of squared errors (SSE) or mean squared error (MSE) between the target variable (y) and our predicted output ( 𝑦̂ ) over all samples in the dataset.*  *OLS can find the best parameters using of the following methods: - Solving the model parameters analytically using closed-form equations - Using an optimization algorithm (Gradient Descent, Stochastic Gradient Descent, Newton’s Method, etc.)*  Prediction  y\_hat= regr.predict(test[['ENGINESIZE','CYLINDERS','FUELCONSUMPTION\_COMB']])  x = np.asanyarray(test[['ENGINESIZE','CYLINDERS','FUELCONSUMPTION\_COMB']])  y = np.asanyarray(test[['CO2EMISSIONS']])  print("Residual sum of squares: %.2f" % np.mean((y\_hat - y) \*\* 2))  ​print('Variance score: %.2f' % regr.score(x, y))# Explained variance score: 1 is perfect prediction  Residual sum of squares: 613.85  Variance score: 0.86  Explained variance regression score:  *If 𝑦̂ is the estimated target output, y the corresponding (correct) target output, and Var is Variance, the square of the standard deviation, then the explained variance is estimated as follow:*  *explainedVariance() = The best possible score is 1.0, lower values are worse.*  *Practice:*  *Try to use a multiple linear regression with the same dataset but this time use \_\_FUEL CONSUMPTION in CITY\_\_ and \_\_FUEL CONSUMPTION in HWY\_\_ instead of FUELCONSUMPTION\_COMB. Does it result in better accuracy.*  # Write your code here  regr = linear\_model.LinearRegression()  x = np.asanyarray(train[['ENGINESIZE','CYLINDERS','FUELCONSUMPTION\_CITY','FUELCONSUMPTION\_HWY']])  y = np.asanyarray(train[['CO2EMISSIONS']])  regr.fit (x, y)  print ('Coefficients: ', regr.coef\_)  y\_hat1= regr.predict(test[['ENGINESIZE','CYLINDERS','FUELCONSUMPTION\_CITY','FUELCONSUMPTION\_HWY']])  x = np.asanyarray(test[['ENGINESIZE','CYLINDERS','FUELCONSUMPTION\_CITY','FUELCONSUMPTION\_HWY']])  y = np.asanyarray(test[['CO2EMISSIONS']])  print("Residual sum of squares: %.2f"% np.mean((y\_hat1 - y) \*\* 2))  print('Variance score: %.2f' % regr.score(x, y))  ​Coefficients: [[9.7022013 7.95168466 6.34783963 2.86527264]]  Residual sum of squares: 617.66  Variance score: 0.86  Want to learn more?  IBM SPSS Modeler is a comprehensive analytics platform that has many machine learning algorithms. It has been designed to bring predictive intelligence to decisions made by individuals, by groups, by systems – by your enterprise as a whole. A free trial is available through this course, available [here](http://cocl.us/ML0101EN-SPSSModeler): |
| Polynomial regression  *Sometimes, the trend of data is not really linear, and looks curvy. In this case we can use Polynomial regression methods. In fact, many different regressions exist that can be used to fit whatever the dataset looks like, such as quadratic, cubic, and so on, and it can go on and on to infinite degrees.*  *In essence, we can call all of these, polynomial regression, where the relationship between the independent variable x and the dependent variable y is modeled as an nth degree polynomial in x. Lets say you want to have a polynomial regression (let's make 2 degree polynomial):*  *Now, the question is: how we can fit our data on this equation while we have only x values, such as Engine Size? Well, we can create a few additional features:*  *PloynomialFeatures() function in Scikit-learn library, drives a new feature sets from the original feature set. That is, a matrix will be generated consisting of all polynomial combinations of the features with degree less than or equal to the specified degree. For example, lets say the original feature set has only one feature, ENGINESIZE. Now, if we select the degree of the polynomial to be 2, then it generates 3 features, degree=0, degree=1 and degree=2:*  from sklearn.preprocessing import PolynomialFeatures  from sklearn import linear\_model  *Lets split our dataset into train and test sets, 80% of the entire data for training, using np.random.rand() function:*  np.random.rand(len(df)) < 0.8; msk = np.random.rand(len(df)) < 0.8; train = cdf[msk]; test = cdf[~msk]  train\_x = np.asanyarray(train[['ENGINESIZE']])  train\_y = np.asanyarray(train[['CO2EMISSIONS']])  ​  test\_x = np.asanyarray(test[['ENGINESIZE']])  test\_y = np.asanyarray(test[['CO2EMISSIONS']])  ​  ​poly = PolynomialFeatures(degree=2)  train\_x\_poly = poly.fit\_transform(train\_x)  train\_x\_poly  *fit\_transform takes our x values, and output a list of our data raised from power of 0 to power of 2 (since we set the degree of our polynomial to 2).*  *It looks like feature sets for multiple linear regression analysis, right? Yes. It Does. Indeed, Polynomial regression is a special case of linear regression, with the main idea of how do you select your features.*  *Just consider replacing the and so on. Then the degree 2 equation would be turn into:*  *Now, we can deal with it as 'linear regression' problem. Therefore, this polynomial regression is considered to be a special case of traditional multiple linear regression. So, you can use the same mechanism as linear regression to solve such a problems*.  *so we can use LinearRegression() function to solve it:*  clfreg = linear\_model.LinearRegression()  clfreg.fit(train\_x\_poly, train\_y)  # The coefficients  print ('Coefficients: ', clfreg.coef\_)  print ('Intercept: ',clfreg.intercept\_)  Coefficients: [[ 0. 48.05089754 -1.06210879]]  Intercept: [109.76986807]  *As mentioned before, Coefficient and Intercept , are the parameters of the fit curvy line. Given that it is a typical multiple linear regression, with 3 parameters, and knowing that the parameters are the intercept and coefficients of hyperplane, sklearn has estimated them from our new set of feature sets. Lets plot it:*  plt.scatter(train.ENGINESIZE, train.CO2EMISSIONS, color='blue')  XX = np.arange(0.0, 10.0, 0.1)  yy = clfreg.intercept\_[0]+ clfreg.coef\_[0][1]\*XX+ clfreg.coef\_[0][2]\*np.power(XX, 2)  plt.plot(XX, yy, '-r' )  plt.xlabel("Engine size")  plt.ylabel("Emission")  Evaluation  from sklearn.metrics import r2\_score  ​test\_x\_poly = poly.fit\_transform(test\_x)  test\_y\_ = clf.predict(test\_x\_poly)  ​print("Mean absolute error: %.2f" % np.mean(np.absolute(test\_y\_ - test\_y)))  print("Residual sum of squares (MSE): %.2f" % np.mean((test\_y\_ - test\_y) \*\* 2))  print("R2-score: %.2f" % r2\_score(test\_y\_ , test\_y) )  Mean absolute error: 22.87  Residual sum of squares (MSE): 860.26  R2-score: 0.74  Practice: Try to use a polynomial regression - degree three (cubic) on same dataset. Does it result in better accuracy?  poly3 = PolynomialFeatures(degree=3)  train\_x\_poly3 = poly3.fit\_transform(train\_x)  clf3 = linear\_model.LinearRegression()  train\_y3\_ = clf3.fit(train\_x\_poly3, train\_y)  print ('Coefficients: ', clf3.coef\_)  print ('Intercept: ',clf3.intercept\_)  plt.scatter(train.ENGINESIZE, train.CO2EMISSIONS, color='blue')  XX = np.arange(0.0, 10.0, 0.1)  yy = clf3.intercept\_[0]+ clf3.coef\_[0][1]\*XX + clf3.coef\_[0][2]\*np.power(XX, 2) + clf3.coef\_[0][3]\*np.power(XX, 3)  plt.plot(XX, yy, '-r' )  plt.xlabel("Engine size")  plt.ylabel("Emission")  test\_x\_poly3 = poly3.fit\_transform(test\_x)  test\_y3\_ = clf3.predict(test\_x\_poly3)  print("Mean absolute error: %.2f" % np.mean(np.absolute(test\_y3\_ - test\_y)))  print("Residual sum of squares (MSE): %.2f" % np.mean((test\_y3\_ - test\_y) \*\* 2))  print("R2-score: %.2f" % r2\_score(test\_y3\_ , test\_y) )  ​  Coefficients: [[ 0. 18.01452904 7.15238731 -0.67531591]]  Intercept: [141.84021144]  Mean absolute error: 23.16  Residual sum of squares (MSE): 878.25  R2-score: 0.74 |
| Non-Linear Regression Analysis  *If the data shows a curvy trend, then linear regression will not produce very accurate results when compared to a non-linear regression because, as the name implies, linear regression presumes that the data is linear. Let's learn about non linear regressions and apply an example on python. In this notebook, we fit a non-linear model to the datapoints corrensponding to China's GDP from 1960 to 2014.*  # Importing required libraries  import numpy as np  import matplotlib.pyplot as plt  %matplotlib inline  *Though Linear regression is very good to solve many problems, it cannot be used for all datasets. First recall how linear regression, could model a dataset. It models a linear relation between a dependent variable y and independent variable x. It had a simple equation, of degree 1, for example y = 2𝑥 + 3.*  x = np.arange(-5.0, 5.0, 0.1)  y = 2\*(x) + 3  y\_noise = 2 \* np.random.normal(size=x.size)  ydata = y + y\_noise  #plt.figure(figsize=(8,6))  plt.plot(x, ydata, 'bo')  plt.plot(x,y, 'r')  plt.ylabel('Dependent Variable')  plt.xlabel('Indepdendent Variable')  plt.show()  *Non-linear regressions are a relationship between independent variables 𝑥 and a dependent variable 𝑦 which result in a non-linear function modeled data. Essentially any relationship that is not linear can be termed as non-linear, and is usually represented by the polynomial of 𝑘 degrees (maximum power of 𝑥 ).*      Non-linear functions can have elements like exponentials, logarithms, fractions, and others. For example:  Or even, more complicated such as :  # Let's take a look at a cubic function's graph.  x = np.arange(-5.0, 5.0, 0.1)  y = 1\*(x\*\*3) + 1\*(x\*\*2) + 1\*x + 3  y\_noise = 20 \* np.random.normal(size=x.size)  ydata = y + y\_noise  plt.plot(x, ydata, 'bo')  plt.plot(x,y, 'r')  plt.ylabel('Dependent Variable')  plt.xlabel('Indepdendent Variable')  plt.show()  As you can see, this function has as independent variables. Also, the graphic of this function is not a straight line over the 2D plane. So this is a non-linear function.  Some other types of non-linear functions are:  Quadratic:  x = np.arange(-5.0, 5.0, 0.1)  y = np.power(x,2)  y\_noise = 2 \* np.random.normal(size=x.size)  ydata = y + y\_noise  plt.plot(x, ydata, 'bo')  plt.plot(x,y, 'r')  plt.ylabel('Dependent Variable')  plt.xlabel('Indepdendent Variable')  plt.show()    Exponential: An exponential function with base c is defined by:  where b ≠0, c > 0 , c ≠1, and x is any real number. The base, c, is constant and the exponent, x, is a variable.  X = np.arange(-5.0, 5.0, 0.1)  ​Y= np.exp(X)  ​plt.plot(X,Y)  plt.ylabel('Dependent Variable')  plt.xlabel('Indepdendent Variable')  plt.show()  Logarithmic: The response 𝑦 is a results of applying logarithmic map from input 𝑥 's to output variable 𝑦 . It is one of the simplest form of log(): i.e.  *Please consider that instead of 𝑥 , we can use 𝑋 , which can be polynomial representation of the 𝑥 's. In general form it would be written as*  X = np.arange(-5.0, 5.0, 0.1)  ​Y = np.log(X)  ​plt.plot(X,Y)  plt.ylabel('Dependent Variable')  plt.xlabel('Indepdendent Variable')  plt.show()  Sigmoidal/Logistic:  X = np.arange(-5.0, 5.0, 0.1)  ​Y = 1-4/(1+np.power(3, X-2))  ​plt.plot(X,Y)  plt.ylabel('Dependent Variable')  plt.xlabel('Indepdendent Variable')  plt.show()  Non-Linear Regression example:  *For an example, we're going to try and fit a non-linear model to the datapoints corresponding to China's GDP from 1960 to 2014. We download a dataset with two columns, the first, a year between 1960 and 2014, the second, China's corresponding annual gross domestic income in US dollars for that year.*  import numpy as np  import pandas as pd  ​#downloading dataset  !wget -nv -O china\_gdp.csv https://s3-api.us-geo.objectstorage.softlayer.net/cf-courses-data/CognitiveClass/ML0101ENv3/labs/china\_gdp.csv  df = pd.read\_csv("china\_gdp.csv")  df.head()  Year Value  0 1960 5.918412e+10  1 1961 4.955705e+10  2 1962 4.668518e+10  3 1963 5.009730e+10  4 1964 5.906225e+10  # Plotting the Dataset    This is what the datapoints look like. It kind of looks like an either logistic or exponential function. The growth starts off slow, then from 2005 on forward, the growth is very significant. And finally, it decelerate slightly in the 2010s.  plt.figure(figsize=(8,5))  x\_data, y\_data = (df["Year"].values, df["Value"].values)  plt.plot(x\_data, y\_data, 'ro')  plt.ylabel('GDP')  plt.xlabel('Year')  plt.show()  # Choosing a model  *From an initial look at the plot, we determine that the logistic function could be a good approximation, since it has the property of starting with a slow growth, increasing growth in the middle, and then decreasing again at the end; as illustrated below:*  X = np.arange(-5.0, 5.0, 0.1)  Y = 1.0 / (1.0 + np.exp(-X))  ​plt.plot(X,Y)  plt.ylabel('Dependent Variable')  plt.xlabel('Indepdendent Variable')  plt.show()  *The formula for the logistic function is the following:*    𝛽1 : Controls the curve's steepness,  𝛽2 : Slides the curve on the x-axis.  # Building The Model  *Now, let's build our regression model and initialize its parameters.*  def sigmoid(x, Beta\_1, Beta\_2):  y = 1 / (1 + np.exp(-Beta\_1\*(x-Beta\_2)))  return y  *Lets look at a sample sigmoid line that might fit with the data:*  beta\_1 = 0.10  beta\_2 = 1990.0  ​#logistic function  Y\_pred = sigmoid(x\_data, beta\_1 , beta\_2)  ​#plot initial prediction against datapoints  plt.plot(x\_data, Y\_pred\*15000000000000.)  plt.plot(x\_data, y\_data, 'ro')  *Our task here is to find the best parameters for our model. Lets first normalize our x and y:*  # Lets normalize our data  xdata =x\_data/max(x\_data)  ydata =y\_data/max(y\_data)  *How we find the best parameters for our fit line?*  *we can use curve\_fit which uses non-linear least squares to fit our sigmoid function, to data. Optimal values for the parameters so that the sum of the squared residuals of sigmoid(xdata, \*popt) - ydata is minimized.*  *popt are our optimized parameters.*  from scipy.optimize import curve\_fit  popt, pcov = curve\_fit(sigmoid, xdata, ydata)  #print the final parameters  print("beta\_1 = %f, beta\_2 = %f" % (popt[0], popt[1]))  beta\_1 = 690.447527, beta\_2 = 0.997207  *Now we plot our resulting regression model.*  x = np.linspace(1960, 2015, 55)  x = x/max(x)  plt.figure(figsize=(8,5))  y = sigmoid(x, \*popt)  plt.plot(xdata, ydata, 'ro', label='data')  plt.plot(x,y, linewidth=3.0, label='fit')  plt.legend(loc='best')  plt.ylabel('GDP')  plt.xlabel('Year')  plt.show()  Practice:  *Can you calculate what is the accuracy of our model?*  # split data into train/test  msk = np.random.rand(len(df)) < 0.8  train\_x = xdata[msk]  test\_x = xdata[~msk]  train\_y = ydata[msk]  test\_y = ydata[~msk]  ​  # Build the model using train set  popt, pcov = curve\_fit(sigmoid, train\_x, train\_y)  ​  # predict using test set  y\_hat = sigmoid(test\_x, \*popt)  ​  # evaluation  print("Mean absolute error: %.2f" % np.mean(np.absolute(y\_hat - test\_y)))  print("Residual sum of squares (MSE): %.2f" % np.mean((y\_hat - test\_y) \*\* 2))  from sklearn.metrics import r2\_score  print("R2-score: %.2f" % r2\_score(y\_hat , test\_y) )  ​  Mean absolute error: 0.03  Residual sum of squares (MSE): 0.00  R2-score: 0.96 |
| K-Nearest Neighbors  *In this Lab you will load a customer dataset, fit the data, and use K-Nearest Neighbors to predict a data point. But what is K-Nearest Neighbors?*  *K-Nearest Neighbors is an algorithm for supervised learning. Where the data is 'trained' with data points corresponding to their classification. Once a point is to be predicted, it takes into account the 'K' nearest points to it to determine it's classification.*  *Here's an visualization of the K-Nearest Neighbors algorithm.*  *In this case, we have data points of Class A and B. We want to predict what the star (test data point) is. If we consider a k value of 3 (3 nearest data points) we will obtain a prediction of Class B. Yet if we consider a k value of 6, we will obtain a prediction of Class A.*  *In this sense, it is important to consider the value of k. But hopefully from this diagram, you should get a sense of what the K-Nearest Neighbors algorithm is. It considers the 'K' Nearest Neighbors (points) when it predicts the classification of the test point.*  # Lets load required libraries  import itertools  import numpy as np  import matplotlib.pyplot as plt  from matplotlib.ticker import NullFormatter  import pandas as pd  import numpy as np  import matplotlib.ticker as ticker  from sklearn import preprocessing  %matplotlib inline  # About the dataset  *Imagine a telecommunications provider has segmented its customer base by service usage patterns, categorizing the customers into four groups. If demographic data can be used to predict group membership, the company can customize offers for individual prospective customers. It is a classification problem. That is, given the dataset, with predefined labels, we need to build a model to be used to predict class of a new or unknown case.*  *The example focuses on using demographic data, such as region, age, and marital, to predict usage patterns.*  *The target field, called custcat, has four possible values that correspond to the four customer groups, as follows:*  *1- Basic Service, 2- E-Service, 3- Plus Service, 4- Total Service*  *Our objective is to build a classifier, to predict the class of unknown cases. We will use a specific type of classification called K nearest neighbour.*  # Lets download the dataset. To download the data, we will use !wget to download it from IBM Object Storage.  !wget -O teleCust1000t.csv https://s3-api.us-geo.objectstorage.softlayer.net/..../labs/teleCust1000t.csv  # Load Data From CSV File  df = pd.read\_csv('teleCust1000t.csv')  df.head()   |  |  |  |  |  |  |  |  |  |  |  |  |  | | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | |  | region | tenure | age | marital | address | income | ed | employ | retire | gender | reside | custcat | | 0 | 2 | 13 | 44 | 1 | 9 | 64.0 | 4 | 5 | 0.0 | 0 | 2 | 1 | | 1 | 3 | 11 | 33 | 1 | 7 | 136.0 | 5 | 5 | 0.0 | 0 | 6 | 4 | | 2 | 3 | 68 | 52 | 1 | 24 | 116.0 | 1 | 29 | 0.0 | 1 | 2 | 3 | | 3 | 2 | 33 | 33 | 0 | 12 | 33.0 | 2 | 0 | 0.0 | 1 | 1 | 1 | | 4 | 2 | 23 | 30 | 1 | 9 | 30.0 | 1 | 2 | 0.0 | 0 | 4 | 3 |   # Data Visualization and Analysis, Let’s see how many of each class is in our data set  df['custcat'].value\_counts()  3 281  1 266  4 236  2 217  Name: custcat, dtype: int64  *i.e. 281 Plus Service, 266 Basic-service, 236 Total Service, and 217 E-Service customers.*    # You can easily explore your data using visualization techniques:  df.hist(column='income', bins=50)  Feature set:  Lets define feature sets, X:  df.columns  Index(['region', 'tenure', 'age', 'marital', 'address', 'income', 'ed',  'employ', 'retire', 'gender', 'reside', 'custcat'], dtype='object')  *To use scikit-learn library, convert the data frame to a Numpy array:*  #astype(float)  X = df[['region', 'tenure','age', 'marital', 'address', 'income', 'ed', 'employ','retire', 'gender', 'reside']] .values  X[0:5]  # What are our labels?  y = df['custcat'].values  y[0:5]  array([1, 4, 3, 1, 3])  # Normalize Data  *Data Standardization give data zero mean and unit variance, it is good practice, especially for algorithms such as KNN which is based on distance of cases:*  X = preprocessing.StandardScaler().fit(X).transform(X.astype(float))  X[0:5]  # Train Test Split  *Out of Sample Accuracy is the percentage of correct predictions that the model makes on data that that the model has NOT been trained on. Doing a train and test on the same dataset will most likely have low out-of-sample accuracy, due to the likelihood of being over-fit.*  *It is important that our models have a high, out-of-sample accuracy, because the purpose of any model, of course, is to make correct predictions on unknown data. So how can we improve out-of-sample accuracy? One way is to use an evaluation approach called Train/Test Split. Train/Test Split involves splitting the dataset into training and testing sets respectively, which are mutually exclusive. After which, you train with the training set and test with the testing set.*  *This will provide a more accurate evaluation on out-of-sample accuracy because the testing dataset is not part of the dataset that have been used to train the data. It is more realistic for real world problems.*  from sklearn.model\_selection import train\_test\_split  X\_train, X\_test, y\_train, y\_test = train\_test\_split( X, y, test\_size=0.2, random\_state=4)  print ('Train set:', X\_train.shape, y\_train.shape)  print ('Test set:', X\_test.shape, y\_test.shape)  Train set: (800, 11) (800,)  Test set: (200, 11) (200,)  Classification: K nearest neighbor (KNN)  # Import library, Classifier implementing the k-nearest neighbors vote.  from sklearn.neighbors import KNeighborsClassifier  # Training, Lets start the algorithm with k=4 for now:  k = 4  #Train Model and Predict  neigh = KNeighborsClassifier(n\_neighbors = k).fit(X\_train,y\_train)  neigh  KNeighborsClassifier(algorithm='auto', leaf\_size=30, metric='minkowski',  metric\_params=None, n\_jobs=None, n\_neighbors=4, p=2,  weights='uniform')  # Predicting, we can use the model to predict the test set:  yhat = neigh.predict(X\_test)  yhat[0:5]  array([1, 1, 3, 2, 4])  Accuracy evaluation:  *In multilabel classification, accuracy classification score is a function that computes subset accuracy. This function is equal to the jaccard\_similarity\_score function. Essentially, it calculates how closely the actual labels and predicted labels are matched in the test set.*  from sklearn import metrics  print("Train set Accuracy: ", metrics.accuracy\_score(y\_train, neigh.predict(X\_train)))  print("Test set Accuracy: ", metrics.accuracy\_score(y\_test, yhat))  Train set Accuracy: 0.5475  Test set Accuracy: 0.32  Practice: Can you build the model again, but this time with k=6?  # Write your code here  k = 6  # Train Model and Predict  neigh1 = KNeighborsClassifier(n\_neighbors = k).fit(X\_train,y\_train)  yhat1 = neigh1.predict(X\_test)  print("Train set Accuracy: ", metrics.accuracy\_score(y\_train, neigh1.predict(X\_train)))  print("Test set Accuracy: ", metrics.accuracy\_score(y\_test, yhat1))  ​  Train set Accuracy: 0.51625  Test set Accuracy: 0.31  *What about other K?*  *K in KNN, is the number of nearest neighbors to examine. It is supposed to be specified by the User. So, how can we choose right value for K? The general solution is to reserve a part of your data for testing the accuracy of the model. Then chose k =1, use the training part for modeling, and calculate the accuracy of prediction using all samples in your test set. Repeat this process, increasing the k, and see which k is the best for your model.*  *We can calculate the accuracy of KNN for different Ks.*  Ks = 10  mean\_acc = np.zeros((Ks-1))  std\_acc = np.zeros((Ks-1))  ConfustionMx = [];  for n in range(1,Ks):    #Train Model and Predict  neigh = KNeighborsClassifier(n\_neighbors = n).fit(X\_train,y\_train)  yhat=neigh.predict(X\_test)  mean\_acc[n-1] = metrics.accuracy\_score(y\_test, yhat)  std\_acc[n-1]=np.std(yhat==y\_test)/np.sqrt(yhat.shape[0])  ​  mean\_acc  array([0.3 , 0.29 , 0.315, 0.32 , 0.315, 0.31 , 0.335, 0.325, 0.34 ])  # Plot model accuracy for Different number of Neighbors  plt.plot(range(1,Ks),mean\_acc,'g')  plt.fill\_between(range(1,Ks),mean\_acc - 1 \* std\_acc,mean\_acc + 1 \* std\_acc, alpha=0.10)  plt.legend(('Accuracy ', '+/- 3xstd'))  plt.ylabel('Accuracy ')  plt.xlabel('Number of Nabors (K)')  plt.tight\_layout()  plt.show()  print( "The…", mean\_acc.max(), "with k=", mean\_acc.argmax()+1)  The best accuracy was with 0.34 with k= 9 |
| Decision Trees  *In this lab exercise, you will learn a popular machine learning algorithm, Decision Tree. You will use this classification algorithm to build a model from historical data of patients, and their response to different medications. Then you use the trained decision tree to predict the class of a unknown patient, or to find a proper drug for a new patient.*  # Import the Following Libraries: numpy (as np), pandas, DecisionTreeClassifier from sklearn.tree  import numpy as np  import pandas as pd  from sklearn.tree import DecisionTreeClassifier  About the dataset:  *Imagine that you are a medical researcher compiling data for a study. You have collected data about a set of patients, all of whom suffered from the same illness. During their course of treatment, each patient responded to one of 5 medications, Drug A, Drug B, Drug c, Drug x and y.*  *Part of your job is to build a model to find out which drug might be appropriate for a future patient with the same illness. The feature sets of this dataset are Age, Sex, Blood Pressure, and Cholesterol of patients, and the target is the drug that each patient responded to.*  *It is a sample of binary classifier, and you can use the training part of the dataset to build a decision tree, and then use it to predict the class of a unknown patient, or to prescribe it to a new patient.*  # Downloading the Data  *To download the data, we will use !wget to download it from IBM Object Storage.*  !wget -O drug200.csv <https://s3-api.us-geo.objectstorage.softlayer.net..../labs/drug200.csv>  # *Now, read data using pandas dataframe:*  my\_data = pd.read\_csv("drug200.csv", delimiter=",")  my\_data[0:5]   |  |  |  |  |  |  |  | | --- | --- | --- | --- | --- | --- | --- | |  | Age | Sex | BP | Cholesterol | Na\_to\_K | Drug | | 0 | 23 | F | HIGH | HIGH | 25.355 | drugY | | 1 | 47 | M | LOW | HIGH | 13.093 | drugC | | 2 | 47 | M | LOW | HIGH | 10.114 | drugC | | 3 | 28 | F | NORMAL | HIGH | 7.798 | drugX | | 4 | 61 | F | LOW | HIGH | 18.043 | drugY |   # Practice: What is the size of data?  my\_data.describe()  my\_data.shape  (200,6)  my\_data.columns  Index(['Age', 'Sex', 'BP', 'Cholesterol', 'Na\_to\_K', 'Drug'], dtype='object')  # Pre-processing  *Using my\_data as the Drug.csv data read by pandas, declare the following variables:*  *X as the Feature Matrix (input data for classification)*  *y as the response vector (target, original classification )*  Remove the column containing the target name since it doesn't contain numeric values.  X = my\_data[['Age', 'Sex', 'BP', 'Cholesterol', 'Na\_to\_K']].values  y = my\_data["Drug"]  X[0:5]  array([ [23, 'F', 'HIGH', 'HIGH', 25.355], [47, 'M', 'LOW', 'HIGH', 13.093], [47, 'M', 'LOW', 'HIGH', 10.113999999999999],  [28, 'F', 'NORMAL', 'HIGH', 7.797999999999999], [61, 'F', 'LOW', 'HIGH', 18.043] ], dtype=object)  *As you may figure out, some features in this dataset are categorical such as Sex or BP. Unfortunately, Sklearn Decision Trees do not handle categorical variables. But still we can convert these features to numerical values. pandas.get\_dummies() Convert categorical variable into dummy/indicator variables.*  # pd.get\_dummies(my\_data['BP'])  from sklearn import preprocessing  le\_sex = preprocessing.LabelEncoder()  le\_sex.fit(['F','M'])  X[:,1] = le\_sex.transform(X[:,1])  le\_BP = preprocessing.LabelEncoder()  le\_BP.fit([ 'LOW', 'NORMAL', 'HIGH'])  X[:,2] = le\_BP.transform(X[:,2])  le\_Chol = preprocessing.LabelEncoder()  le\_Chol.fit([ 'NORMAL', 'HIGH'])  X[:,3] = le\_Chol.transform(X[:,3])  X[0:5]  array([ [23, 0, 0, 0, 25.355], [47, 1, 1, 0, 13.093], [47, 1, 1, 0, 10.113999999999999], [28, 0, 2, 0, 7.797999999999999],  [61, 0, 1, 0, 18.043] ], dtype=object)  # Setting up the Decision Tree, *We will be using train/test split on our decision tree.*  # Let's import train\_test\_split from sklearn.cross\_validation.  from sklearn.model\_selection import train\_test\_split  # Now train\_test\_split will return 4 different parameters. name them: X\_trainset, X\_testset, y\_trainset, y\_testset   * *The train\_test\_split will need the parameters: X, y, test\_size=0.3, and random\_state=3.* * *The X and y are the arrays required before the split, the test\_size represents the ratio of the testing dataset, and the random\_state ensures that we obtain the same splits.*   X\_trainset, X\_testset, y\_trainset, y\_testset = train\_test\_split(X, y, test\_size=0.3, random\_state=3)  # Print the shape of X\_trainset, y\_trainset and X\_testset, y\_testset . Ensure that the dimensions match  print ('Train set:', X\_trainset.shape, y\_trainset.shape)  ​print ('Test set:', X\_testset.shape, y\_testset.shape)  Train set: (140, 5) (140,)  Test set: (60, 5) (60,)  Modelling:  *We will first create an instance of the DecisionTreeClassifier called drugTree.*  *Inside of the classifier, specify criterion="entropy" so we can see the information gain of each node.*  drugTree = DecisionTreeClassifier(criterion="entropy", max\_depth = 4)  drugTree # it shows the default parameters  DecisionTreeClassifier(class\_weight=None, criterion='entropy', max\_depth=4,  max\_features=None, max\_leaf\_nodes=None,  min\_impurity\_decrease=0.0, min\_impurity\_split=None,  min\_samples\_leaf=1, min\_samples\_split=2,  min\_weight\_fraction\_leaf=0.0, presort=False, random\_state=None,  splitter='best')  *Next, we will fit the data with the training feature matrix X\_trainset and training response vector y\_trainset*  drugTree.fit(X\_trainset,y\_trainset)  # Prediction  *Let's make some predictions on the testing dataset and store it into a variable called predTree.*  predTree = drugTree.predict(X\_testset)  *You can print out predTree and y\_testset if you want to visually compare the prediction to the actual values.*  print (predTree [0:5])  print (y\_testset [0:5].tolist())  ​  ['drugY' 'drugX' 'drugX' 'drugX' 'drugX']  ['drugY', 'drugX', 'drugX', 'drugX', 'drugX']  Evaluation  *Next, let's import metrics from sklearn and check the accuracy of our model.*  from sklearn import metrics  import matplotlib.pyplot as plt  print("DecisionTrees's Accuracy: ", metrics.accuracy\_score(y\_testset, predTree))  DecisionTrees's Accuracy: 0.9833333333333333  *Accuracy classification score computes subset accuracy: the set of labels predicted for a sample must exactly match the corresponding set of labels in y\_true.*  *In multilabel classification, the function returns the subset accuracy. If the entire set of predicted labels for a sample strictly match with the true set of labels, then the subset accuracy is 1.0; otherwise it is 0.0.*  Can you calculate the accuracy score without sklearn ?  # your code here  std\_acc = y\_testset.tolist()==predTree  sum(std\_acc)/len(std\_acc)  ​  #org=y\_testset.tolist()  #pred=predTree.tolist()  #std\_acc=[]  #for n in range(0,len(org)):  #print(n)  #val = (int(org[n]==pred[n]),)  #std\_acc = std\_acc + (val)  #val = int(y\_testset.tolist()[n]==predTree[n])  #std\_acc.extend([val])  #std\_acc  #sum(std\_acc)/len(std\_acc)  #print("DecisionTrees's Accuracy: ", r2\_score(y\_testset, predTree))  0.9833333333333333  Visualization  # You might need to uncomment and install the pydotplus and graphviz libraries if you have not installed these before  !conda install -c conda-forge pydotplus -y  !conda install -c conda-forge python-graphviz -y  from sklearn.externals.six import StringIO  import pydotplus  import matplotlib.image as mpimg  from sklearn import tree  %matplotlib inline  dot\_data = StringIO()  filename = "drugtree.png"  featureNames = my\_data.columns[0:5]  targetNames = my\_data["Drug"].unique().tolist()  out=tree.export\_graphviz(drugTree,  feature\_names=featureNames,  out\_file=dot\_data,  class\_names= np.unique(y\_trainset),  filled=True,  special\_characters=True,  rotate=False)  graph = pydotplus.graph\_from\_dot\_data(dot\_data.getvalue())  graph.write\_png(filename)  img = mpimg.imread(filename)  plt.figure(figsize=(100, 200))  plt.imshow(img,interpolation='nearest') |
| Logistic Regression with Python  *In this notebook, you will learn Logistic Regression, and then, you'll create a model for a telecommunication company, to predict when its customers will leave for a competitor, so that they can take some action to retain the customers.*  *What is the difference between Linear and Logistic Regression?*  *While Linear Regression is suited for estimating continuous values (e.g. estimating house price), it is not the best tool for predicting the class of an observed data point. In order to estimate the class of a data point, we need some sort of guidance on what would be the most probable class for that data point. For this, we use Logistic Regression.*  *Recall linear regression:*  *As you know, Linear regression finds a function that relates a continuous dependent variable, y, to some predictors (independent variables 𝑥1, 𝑥2, etc.). For example, Simple linear regression assumes a function of the form:*  *and finds the values of parameters etc, where the term is the "intercept". It can be generally shown as:*  *Logistic Regression is a variation of Linear Regression, useful when the observed dependent variable, y, is categorical. It produces a formula that predicts the probability of the class label as a function of the independent variables.*  *Logistic regression fits a special s-shaped curve by taking the linear regression and transforming the numeric estimate into a probability with the following function, which is called sigmoid function 𝜎:*  *Or:*  *In this equation, is the regression result (the sum of the variables weighted by the coefficients), exp is the exponential function and is the sigmoid or logistic function, also called logistic curve. It is a common "S" shape (sigmoid curve).*  *So, briefly, Logistic Regression passes the input through the logistic/sigmoid but then treats the result as a probability:*  *The objective of Logistic Regression algorithm, is to find the best parameters , in such a way that the model best predicts the class of each case.*  Customer churn with Logistic Regression  *A telecommunications company is concerned about the number of customers leaving their land-line business for cable competitors. They need to understand who is leaving. Imagine that you are an analyst at this company and you have to find out who is leaving and why.*  # Lets first import required libraries:  import pandas as pd  import pylab as pl  import numpy as np  import scipy.optimize as opt  from sklearn import preprocessing  %matplotlib inline  import matplotlib.pyplot as plt  About the dataset:  *We will use a telecommunications dataset for predicting customer churn. This is a historical customer dataset where each row represents one customer. The data is relatively easy to understand, and you may uncover insights you can use immediately. Typically, it is less expensive to keep customers than acquire new ones, so the focus of this analysis is to predict the customers who will stay with the company.*  *This data set provides information to help you predict what behaviour will help you to retain customers. You can analyse all relevant customer data and develop focused customer retention programs.*  *The dataset includes information about:*   * *Customers who left within the last month – the column is called Churn* * *Services that each customer has signed up for – phone, multiple lines, internet, online security, online backup, device protection, tech support, and streaming TV and movies* * *Customer account information – how long they had been a customer, contract, payment method, paperless billing, monthly charges, and total charges* * *Demographic info about customers – gender, age range, and if they have partners and dependents*   # Load the Telco Churn data  *Telco Churn is a hypothetical data file that concerns a telecommunications company's efforts to reduce turnover in its customer base. Each case corresponds to a separate customer and it records various demographic and service usage information. Before you can work with the data, you must use the URL to get the ChurnData.csv.*  # To download the data, we will use, !wget to download it from IBM Object Storage.  !wget -O ChurnData.csv <https://s3-api.us-geo.objectstorage.softlayer.net/.../ML0101ENv3/labs/ChurnData.csv>  # Load Data From CSV File  churn\_df = pd.read\_csv("ChurnData.csv")  churn\_df.head()  # Data pre-processing and selection  Let’s select some features for the modelling. Also, we change the target data type to be integer, as it is a requirement by the skitlearn algorithm:  churn\_df = churn\_df[['tenure', 'age', 'address', 'income', 'ed', 'employ', 'equip', 'callcard', 'wireless','churn']]  churn\_df['churn'] = churn\_df['churn'].astype('int')  churn\_df.head()   |  |  |  |  |  |  |  |  |  |  |  | | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | |  | tenure | age | address | income | ed | employ | equip | callcard | wireless | churn | | 0 | 11.0 | 33.0 | 7.0 | 136.0 | 5.0 | 5.0 | 0.0 | 1.0 | 1.0 | 1 | | 1 | 33.0 | 33.0 | 12.0 | 33.0 | 2.0 | 0.0 | 0.0 | 0.0 | 0.0 | 1 | | 2 | 23.0 | 30.0 | 9.0 | 30.0 | 1.0 | 2.0 | 0.0 | 0.0 | 0.0 | 0 | | 3 | 38.0 | 35.0 | 5.0 | 76.0 | 2.0 | 10.0 | 1.0 | 1.0 | 1.0 | 0 |   # Practice  How many rows and columns are in this dataset in total? What are the names of columns?  churn\_df.shape  ​  # Let’s define X, and y for our dataset:  X = np.asarray(churn\_df[['tenure', 'age', 'address', 'income', 'ed', 'employ', 'equip']])  X[0:5]  array([ [ 11., 33., 7., 136., 5., 5., 0.], [ 33., 33., 12., 33., 2., 0., 0.], [ 23., 30., 9., 30., 1., 2., 0.],  [ 38., 35., 5., 76., 2., 10., 1.], [ 7., 35., 14., 80., 2., 15., 0.]])  y = np.asarray(churn\_df['churn'])  y [0:5]  array([1, 1, 0, 0, 0])  # Also, we normalize the dataset:  from sklearn import preprocessing  X = preprocessing.StandardScaler().fit(X).transform(X)  X[0:5]  # Train/Test dataset, Okay, we split our dataset into train and test set:  from sklearn.model\_selection import train\_test\_split  X\_train, X\_test, y\_train, y\_test = train\_test\_split( X, y, test\_size=0.2, random\_state=4)  print ('Train set:', X\_train.shape, y\_train.shape)  print ('Test set:', X\_test.shape, y\_test.shape)  Train set: (160, 7) (160,)  Test set: (40, 7) (40,)  Modelling (Logistic Regression with Scikit-learn)  *Let’s build our model using LogisticRegression from Scikit-learn package. This function implements logistic regression and can use different numerical optimizers to find parameters, including ‘newton-cg’, ‘lbfgs’, ‘liblinear’, ‘sag’, ‘saga’ solvers. You can find extensive information about the pros and cons of these optimizers if you search it in internet.*  *The version of Logistic Regression in Scikit-learn, support regularization. Regularization is a technique used to solve the overfitting problem in machine learning models. C parameter indicates inverse of regularization strength which must be a positive float. Smaller values specify stronger regularization. Now let’s fit our model with train set:*  from sklearn.linear\_model import LogisticRegression  from sklearn.metrics import confusion\_matrix  LR = LogisticRegression(C=0.01, solver='liblinear').fit(X\_train,y\_train)  LR LogisticRegression(C=0.01, class\_weight=None, dual=False, fit\_intercept=True,  intercept\_scaling=1, max\_iter=100, multi\_class='warn',  n\_jobs=None, penalty='l2', random\_state=None, solver='liblinear',  tol=0.0001, verbose=0, warm\_start=False)  # Now we can predict using our test set:  yhat = LR.predict(X\_test)  yhat array([0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 1, 1, 0, 0, 0, 1, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 1, 0, 0, 0])  # predict\_proba returns estimates for all classes, ordered by the label of classes. So, the first column is the probability of class 1, , and second column is probability of class 0,  yhat\_prob = LR.predict\_proba(X\_test)  yhat\_prob  array([ [0.54132919, 0.45867081], [0.60593357, 0.39406643], [0.56277713, 0.43722287],  [0.63432489, 0.36567511], […………………, …………………], [0.51141720, 0.48858280] ])  # jaccard index  *Let’s try jaccard index for accuracy evaluation. we can define jaccard as the size of the intersection divided by the size of the union of two label sets. If the entire set of predicted labels for a sample strictly match with the true set of labels, then the subset accuracy is 1.0; otherwise it is 0.0.*  from sklearn.metrics import jaccard\_similarity\_score  jaccard\_similarity\_score(y\_test, yhat)  0.75  # A*nother way of looking at accuracy of classifier is to look at confusion matrix.*  from sklearn.metrics import classification\_report, confusion\_matrix  import itertools  cnf\_matrix = confusion\_matrix(y\_test, yhat, labels=[1,0])  print (cnf\_matrix)  np.set\_printoptions(precision=2)  [[ 6 9] [ 1 24]]  # Plot confusion matrix  classes=['churn=1','churn=0']  plt.imshow(cnf\_matrix, interpolation='nearest', cmap=plt.cm.Blues)  plt.title('Confusion matrix')  plt.colorbar()  tick\_marks = np.arange(len(classes))  plt.xticks(tick\_marks, classes, rotation=45)  plt.yticks(tick\_marks, classes)  fmt = '.2f' #if normalize else 'd'  thresh = cnf\_matrix.max() / 2.  for i, j in itertools.product(range(cnf\_matrix.shape[0]), range(cnf\_matrix.shape[1])):  plt.text(j, i, format(cnf\_matrix[i, j], fmt),  horizontalalignment="center",  color="white" if cnf\_matrix[i, j] > thresh else "black")  plt.tight\_layout()  plt.ylabel('True label')  plt.xlabel('Predicted label')  # Confusion matrix Plotting Function  *This function prints and plots the confusion matrix. Normalization can be applied by setting `normalize=True*  def plot\_confusion\_matrix(cm, classes,  normalize=False,  title='Confusion matrix',  cmap=plt.cm.Blues):  """  This function prints and plots the confusion matrix. Normalization can be applied by setting `normalize=True`.  """  if normalize:  cm = cm.astype('float') / cm.sum(axis=1)[:, np.newaxis]  print("Normalized confusion matrix")  else:  print('Confusion matrix, without normalization')  ​  print(cm)  ​  plt.imshow(cm, interpolation='nearest', cmap=cmap)  plt.title(title)  plt.colorbar()  tick\_marks = np.arange(len(classes))  plt.xticks(tick\_marks, classes, rotation=45)  plt.yticks(tick\_marks, classes)  ​  fmt = '.2f' if normalize else 'd'  thresh = cm.max() / 2.  for i, j in itertools.product(range(cm.shape[0]), range(cm.shape[1])):  plt.text(j, i, format(cm[i, j], fmt),  horizontalalignment="center",  color="white" if cm[i, j] > thresh else "black")  ​  plt.tight\_layout()  plt.ylabel('True label')  plt.xlabel('Predicted label')  print(confusion\_matrix(y\_test, yhat, labels=[1,0]))  [[ 6 9] [ 1 24]]  # Compute confusion matrix  cnf\_matrix = confusion\_matrix(y\_test, yhat, labels=[1,0])  np.set\_printoptions(precision=2)  ​  ​# Plot non-normalized confusion matrix using above function.  plt.figure()  plot\_confusion\_matrix(cnf\_matrix, classes=['churn=1','churn=0'],normalize= False, title='Confusion matrix')  Confusion matrix, without normalization  [[ 6 9] [ 1 24]]  *Look at first row. The first row is for customers whose actual churn value in test set is 1. As you can calculate, out of 40 customers, the churn value of 15 of them is 1. And out of these 15, the classifier correctly predicted 6 of them as 1, and 9 of them as 0.*  *It means, for 6 customers, the actual churn value was 1 in test set, and classifier also correctly predicted those as 1. However, while the actual label of 9 customers were 1, the classifier predicted those as 0, which is not very good. We can consider it as error of the model for first row.*  *What about the customers with churn value 0? Let’s look at the second row. It looks like there were 25 customers whom their churn value was 0.*  *The classifier correctly predicted 24 of them as 0, and one of them wrongly as 1. So, it has done a good job in predicting the customers with churn value 0. A good thing about confusion matrix is that shows the model’s ability to correctly predict or separate the classes. In specific case of binary classifier, such as this example, we can interpret these numbers as the count of true positives, false positives, true negatives, and false negatives.*  print (classification\_report(y\_test, yhat))   |  |  |  |  |  | | --- | --- | --- | --- | --- | |  | Precision | recall | f1-score | support | | 0 | 0.73 | 0.96 | 0.83 | 25 | | 1 | 0.86 | 0.40 | 0.55 | 15 | | micro avg | 0.75 | 0.75 | 0.75 | 40 | | macro avg | 0.79 | 0.68 | 0.69 | 40 | | weighted avg | 0.78 | 0.75 | 0.72 | 40 |   *Based on the count of each section, we can calculate precision and recall of each label:*   * Precision is a measure of the accuracy provided that a class label has been predicted. It is defined by:   precision = TP / (TP + FP)   * Recall is true positive rate. It is defined as:   Recall = TP / (TP + FN)  So, we can calculate precision and recall of each class.  *F1 score: Now we are in the position to calculate the F1 scores for each label based on the precision and recall of that label.*  *The F1 score is the harmonic average of the precision and recall, where an F1 score reaches its best value at 1 (perfect precision and recall) and worst at 0. It is a good way to show that a classifier has a good value for both recall and precision.*  *And finally, we can tell the average accuracy for this classifier is the average of the F1-score for both labels, which is 0.72 in our case.*  log loss  *Now, let’s try log loss for evaluation. In logistic regression, the output can be the probability of customer churn is yes (or equals to 1). This probability is a value between 0 and 1. Log loss (Logarithmic loss) measures the performance of a classifier where the predicted output is a probability value between 0 and 1.*  from sklearn.metrics import log\_loss  log\_loss(y\_test, yhat\_prob)  0.6017092478101185 |
| SVM (Support Vector Machines)  *In this notebook, you will use SVM (Support Vector Machines) to build and train a model using human cell records, and classify cells to whether the samples are benign or malignant.*  *SVM works by mapping data to a high-dimensional feature space so that data points can be categorized, even when the data are not otherwise linearly separable. A separator between the categories is found, then the data is transformed in such a way that the separator could be drawn as a hyperplane. Following this, characteristics of new data can be used to predict the group to which a new record should belong.*  # Load the Cancer data  *The example is based on a dataset that is publicly available from the UCI Machine Learning Repository (Asuncion and Newman, 2007)-[http://mlearn.ics.uci.edu/MLRepository.html]. The dataset consists of several hundred human cell sample records, each of which contains the values of a set of cell characteristics. The fields in each record are:*   |  |  | | --- | --- | | Field name | Description | | ID | Clump thickness | | Clump | Clump thickness | | UnifSize | Uniformity of cell size | | UnifShape | Uniformity of cell shape | | MargAdh | Marginal adhesion | | SingEpiSize | Single epithelial cell size | | BareNuc | Bare nuclei | | BlandChrom | Bland chromatin | | NormNucl | Normal nucleoli | | Mit | Mitoses | | Class | Benign or malignant |   For the purposes of this example, we're using a dataset that has a relatively small number of predictors in each record. To download the data, we will use !wget to download it from IBM Object Storage.  !wget -O cell\_samples.csv <https://s3-api.us-geo.objectstorage.softlayer.net..../labs/cell_samples.csv>  cell\_df = pd.read\_csv("cell\_samples.csv")  cell\_df.head()   |  |  |  |  |  |  |  |  |  |  |  |  | | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | |  | ID | Clump | UnifSize | UnifShape | MargAdh | SingEpiSize | BareNuc | BlandChrom | NormNucl | Mit | Class | | 0 | 1000025 | 5 | 1 | 1 | 1 | 2 | 1 | 3 | 1 | 1 | 2 | | 1 | 1002945 | 5 | 4 | 4 | 5 | 7 | 10 | 3 | 2 | 1 | 2 | | 2 | 1015425 | 3 | 1 | 1 | 1 | 2 | 2 | 3 | 1 | 1 | 2 | | 3 | 1016277 | 6 | 8 | 8 | 1 | 3 | 4 | 3 | 7 | 1 | 2 | | 4 | 1017023 | 4 | 1 | 1 | 3 | 2 | 1 | 3 | 1 | 1 | 2 |   *The ID field contains the patient identifiers. The characteristics of the cell samples from each patient are contained in fields Clump to Mit. The values are graded from 1 to 10, with 1 being the closest to benign.*  *The Class field contains the diagnosis, as confirmed by separate medical procedures, as to whether the samples are benign (value = 2) or malignant (value = 4).*  # Let’s look at the distribution of the classes based on Clump thickness and Uniformity of cell size:  ax = cell\_df[cell\_df['Class'] == 4][0:50].plot(kind='scatter', x='Clump', y='UnifSize', color='DarkBlue', label='malignant');  cell\_df[cell\_df['Class'] == 2][0:50].plot(kind='scatter', x='Clump', y='UnifSize', color='Yellow', label='benign', ax=ax);  plt.show()  # Data pre-processing and selection, Let’s first look at columns data types:  cell\_df.dtypes  # cell\_df['BareNuc'].value\_counts()  # cell\_df['BareNuc'].isnull()  # cell\_df[pd.to\_numeric(cell\_df['BareNuc'], errors='coerce').isnull()]  *It looks like the BareNuc column includes some values that are not numerical. We can drop those rows:*  cell\_df = cell\_df[pd.to\_numeric(cell\_df['BareNuc'], errors='coerce').notnull()]  cell\_df['BareNuc'] = cell\_df['BareNuc'].astype('int')  cell\_df.dtypes  *We want the model to predict the value of Class (that is, benign (=2) or malignant (=4)). As this field can have one of only two possible values, we need to change its measurement level to reflect this.*  cell\_df['Class'] = cell\_df['Class'].astype('int')  y = np.asarray(cell\_df['Class'])  y [0:5]  array([2, 2, 2, 2, 2])  # Train/Test dataset, Okay, we split our dataset into train and test set:  X\_train, X\_test, y\_train, y\_test = train\_test\_split( X, y, test\_size=0.2, random\_state=4)  print ('Train set:', X\_train.shape, y\_train.shape)  print ('Test set:', X\_test.shape, y\_test.shape)  Train set: (546, 9) (546,)  Test set: (137, 9) (137,)  Modeling (SVM with Scikit-learn)  *The SVM algorithm offers a choice of kernel functions for performing its processing. Basically, mapping data into a higher dimensional space is called kernelling. The mathematical function used for the transformation is known as the kernel function, and can be of different types, such as:*  *1.Linear*  *2.Polynomial*  *3.Radial basis function (RBF)*  *4.Sigmoid*  *Each of these functions has its characteristics, its pros and cons, and its equation, but as there's no easy way of knowing which function performs best with any given dataset, we usually choose different functions in turn and compare the results. Let's just use the default, RBF (Radial Basis Function) for this lab.*  from sklearn import svm  clf = svm.SVC(kernel='rbf')  clf.fit(X\_train, y\_train)  *After being fitted, the model can then be used to predict new values:*  yhat = clf.predict(X\_test)  yhat [0:5]  array([2, 4, 2, 4, 2])  Evaluation  # Create function to plot confusion matrix  from sklearn.metrics import classification\_report, confusion\_matrix  import itertools  def plot\_confusion\_matrix(cm, classes,  normalize=False,  title='Confusion matrix',  cmap=plt.cm.Blues):  """  This function prints and plots the confusion matrix.  Normalization can be applied by setting `normalize=True`.  """  if normalize:  cm = cm.astype('float') / cm.sum(axis=1)[:, np.newaxis]  print("Normalized confusion matrix")  else:  print('Confusion matrix, without normalization')  ​  print(cm)  ​  plt.imshow(cm, interpolation='nearest', cmap=cmap)  plt.title(title)  plt.colorbar()  tick\_marks = np.arange(len(classes))  plt.xticks(tick\_marks, classes, rotation=45)  plt.yticks(tick\_marks, classes)  ​  fmt = '.2f' if normalize else 'd'  thresh = cm.max() / 2.  for i, j in itertools.product(range(cm.shape[0]), range(cm.shape[1])):  plt.text(j, i, format(cm[i, j], fmt),  horizontalalignment="center",  color="white" if cm[i, j] > thresh else "black")  ​  plt.tight\_layout()  plt.ylabel('True label')   |  |  |  |  |  | | --- | --- | --- | --- | --- | |  | Precision | recall | f1-score | support | | 2 | 1.00 | 0.94 | 0.97 | 90 | | 4 | 0.90 | 1.00 | 0.95 | 47 | | micro avg | 0.96 | 0.96 | 0.96 | 137 | | macro avg | 0.95 | 0.97 | 0.96 | 137 | | weighted avg | 0.97 | 0.96 | 0.96 | 137 |   plt.xlabel('Predicted label')  # Compute confusion matrix  cnf\_matrix = confusion\_matrix(y\_test, yhat, labels=[2,4])  np.set\_printoptions(precision=2)  print (classification\_report(y\_test, yhat))  # Plot non-normalized confusion matrix  plt.figure()  plot\_confusion\_matrix(cnf\_matrix, classes=['Benign(2)','Malignant(4)'],normalize= False, title='Confusion matrix')  *You can also easily use the f1\_score from sklearn library:*  from sklearn.metrics import f1\_score  f1\_score(y\_test, yhat, average='weighted') 0.9639038982104676  # Let’s try jaccard index for accuracy:  from sklearn.metrics import jaccard\_similarity\_score  jaccard\_similarity\_score(y\_test, yhat) 0.9635036496350365  Practice:  Can you rebuild the model, but this time with a linear kernel ? You can use kernel='linear’ option, when you define the svm. How the accuracy changes with the new kernel function?  # write your code here  clf2 = svm.SVC(kernel='linear')  clf2.fit(X\_train, y\_train)  yhat2 = clf2.predict(X\_test)  print("Avg F1-score: %.4f" % f1\_score(y\_test, yhat2, average='weighted'))  print("Jaccard score: %.4f" % jaccard\_similarity\_score(y\_test, yhat2))  Avg F1-score: 0.9639  Jaccard score: 0.9635 |
| K-Means Clustering  *There are many models for clustering out there. In this notebook, we will be presenting the model that is considered one of the simplest models amongst them. Despite its simplicity, the K-means is vastly used for clustering in many data science applications, especially useful if you need to quickly discover insights from unlabelled data. In this notebook, you will learn how to use k-Means for customer segmentation.*  *Some real-world applications of k-means:*   * *Customer segmentation* * *Understand what the visitors of a website are trying to accomplish* * *Pattern recognition* * *Machine learning* * *Data compression*   *In this notebook we practice k-means clustering with 2 examples:*   1. *k-means on a random generated dataset* 2. *Using k-means for customer segmentation*   # Import libraries  Let’s first import the required libraries. Also run %matplotlib inline since we will be plotting in this section.  import random  import numpy as np  import matplotlib.pyplot as plt  from sklearn.cluster import KMeans  from sklearn.datasets.samples\_generator import make\_blobs  %matplotlib inline  k-Means on a randomly generated dataset  *Let’s create our own dataset for this lab!, First, we need to set up a random seed.*  # Use numpy's random.seed() function, where the seed will be set to 0  np.random.seed(0)  *Next, we will be making random clusters of points by using the make\_blobs class. The make\_blobs class can take in many inputs, but we will be using these specific ones.*  # Input   * n\_samples: The total number of points equally divided among clusters.   Value will be: 5000   * centers: The number of centers to generate, or the fixed center locations.   Value will be: [[4, 4], [-2, -1], [2, -3],[1,1]]   * cluster\_std: The standard deviation of the clusters.   Value will be: 0.9  # Output   * X: Array of shape [n\_samples, n\_features]. (Feature Matrix)   The generated samples.   * y: Array of shape [n\_samples]. (Response Vector)   The integer labels for cluster membership of each sample.  X, y = make\_blobs(n\_samples=5000, centers=[[4,4], [-2, -1], [2, -3], [1, 1]], cluster\_std=0.9)  # Display the scatter plot of the randomly generated data.  plt.scatter(X[:, 0], X[:, 1], marker='.')  # Setting up K-Means  *Now that we have our random data, let's set up our K-Means Clustering.*  *The KMeans class has many parameters that can be used, but we will be using these three:*   * *init: Initialization method of the centroids.*   *Value will be: "k-means++"*  *k-means++: Selects initial cluster canter’s for k-mean clustering in a smart way to speed up convergence.*   * *n\_clusters: The number of clusters to form as well as the number of centroids to generate.*   *Value will be: 4 (since we have 4 centres’)*   * *n\_init: Number of times the k-means algorithm will be run with different centroid seeds. The final results will be the best output of n\_init consecutive runs in terms of inertia.*   *Value will be: 12*  # Initialize KMeans with these parameters, where the output parameter is called k\_means.  k\_means = KMeans(init = "k-means++", n\_clusters = 4, n\_init = 12)  # Now let's fit the KMeans model with the feature matrix we created above, X  k\_means.fit(X)  KMeans(algorithm='auto', copy\_x=True, init='k-means++', max\_iter=300,  n\_clusters=4, n\_init=12, n\_jobs=None, precompute\_distances='auto',  random\_state=None, tol=0.0001, verbose=0)  # Now let's grab the labels for each point in the model using KMeans' .labels\_ attribute and save it as k\_means\_labels  k\_means\_labels = k\_means.labels\_  k\_means\_labels  array([0, 3, 3, ..., 1, 0, 0], dtype=int32)  *We will also get the coordinates of the cluster centers using KMeans' .cluster\_centers\_ and save it as:*  k\_means\_cluster\_centers = k\_means.cluster\_centers\_  k\_means\_cluster\_centers  Creating the Visual Plot  *So now that we have the random data generated and the KMeans model initialized, let's plot them. Please read through the code and comments to understand how to plot the model.*  # Initialize the plot with the specified dimensions.  fig = plt.figure(figsize=(6, 4))  ​  # Colors uses a color map, which will produce an array of colors based on the number of labels there are.  # We use set(k\_means\_labels) to get the unique labels.  colors = plt.cm.Spectral(np.linspace(0, 1, len(set(k\_means\_labels))))  ​  # Create a plot  ax = fig.add\_subplot(1, 1, 1)  ​  # For loop that plots the data points and centroids.  # k will range from 0-3, which will match the possible clusters that each data point is in.  for k, col in zip(range(len([[4,4], [-2, -1], [2, -3], [1, 1]])), colors):  ​  # Create a list of all data points, where the data points that are in the cluster (ex. cluster 0) are labelled as true, else they are labelled as false.  my\_members = (k\_means\_labels == k)    # Define the centroid, or cluster center.  cluster\_center = k\_means\_cluster\_centers[k]    # Plots the datapoints with colour col.  ax.plot(X[my\_members, 0], X[my\_members, 1], 'w', markerfacecolor=col, marker='.')    # Plots the centroids with specified color, but with a darker outline  ax.plot(cluster\_center[0], cluster\_center[1], 'o', markerfacecolor=col, markeredgecolor='k', markersize=6)  ​  ax.set\_title('KMeans') # Title of the plot  # Remove x-axis ticks, y-axis ticks  ax.set\_xticks(())​  ax.set\_yticks(())  ​  plt.show() # Show the plot  *Try to cluster the above dataset into 3 clusters., Notice: do not generate data again, use the same dataset as above.*  k\_means3 = KMeans(init = "k-means++", n\_clusters = 3, n\_init = 12)  k\_means3.fit(X)  fig = plt.figure(figsize=(6, 4))  colors = plt.cm.Spectral(np.linspace(0, 1, len(set(k\_means3.labels\_))))  ax = fig.add\_subplot(1, 1, 1)  for k, col in zip(range(len(k\_means3.cluster\_centers\_)), colors):  my\_members = (k\_means3.labels\_ == k)  cluster\_center = k\_means3.cluster\_centers\_[k]  ax.plot(X[my\_members, 0], X[my\_members, 1], 'w', markerfacecolor=col, marker='.')  ax.plot(cluster\_center[0], cluster\_center[1], 'o', markerfacecolor=col, markeredgecolor='k', markersize=6)  plt.show()  Customer Segmentation with K-Means  *Imagine that you have a customer dataset, and you need to apply customer segmentation on this historical data. Customer segmentation is the practice of partitioning a customer base into groups of individuals that have similar characteristics. It is a significant strategy as a business can target these specific groups of customers and effectively allocate marketing resources. For example, one group might contain customers who are high-profit and low-risk, that is, more likely to purchase products, or subscribe for a service. A business task is to retaining those customers. Another group might include customers from non-profit organizations. And so on.*  # Let’s download the dataset. To download the data  !wget -O Cust\_Segmentation.csv <https://s3-api.us-geo.objectstorage.softlayer.net/.../labs/Cust_Segmentation.csv>  # Load Data from CSV File  import pandas as pd  cust\_df = pd.read\_csv("Cust\_Segmentation.csv")  cust\_df.head()  # Pre-processing  *As you can see, Address in this dataset is a categorical variable. k-means algorithm isn't directly applicable to categorical variables because Euclidean distance function isn't really meaningful for discrete variables. So, lets drop this feature and run clustering.*  df = cust\_df.drop('Address', axis=1)  df.head()   |  |  |  |  |  |  |  |  |  |  | | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | |  | Customer Id | Age | Edu | Years Employed | Income | Card Debt | Other Debt | Defaulted | DebtIncomeRatio | | 0 | 1 | 41 | 2 | 6 | 19 | 0.124 | 1.073 | 0.0 | 6.3 | | 1 | 2 | 47 | 1 | 26 | 100 | 4.582 | 8.218 | 0.0 | 12.8 | | 2 | 3 | 33 | 2 | 10 | 57 | 6.111 | 5.802 | 1.0 | 20.9 | | 3 | 4 | 29 | 2 | 4 | 19 | 0.681 | 0.516 | 0.0 | 6.3 | | 4 | 5 | 47 | 1 | 31 | 253 | 9.308 | 8.908 | 0.0 | 7.2 |   # Normalizing over the standard deviation  *Now let's normalize the dataset. But why do we need normalization in the first place? Normalization is a statistical method that helps mathematical-based algorithms to interpret features with different magnitudes and distributions equally. We use StandardScaler() to normalize our dataset.*  from sklearn.preprocessing import StandardScaler  X = df.values[:,1:]  X = np.nan\_to\_num(X)  Clus\_dataSet = StandardScaler().fit\_transform(X)  Clus\_dataSet  Modelling  *In our example (if we didn't have access to the k-means algorithm), it would be the same as guessing that each customer group would have certain age, income, education, etc, with multiple tests and experiments. However, using the K-means clustering we can do all this process much easier.*  # Let’s apply k-means on our dataset, and take look at cluster labels.  clusterNum = 3  k\_means = KMeans(init = "k-means++", n\_clusters = clusterNum, n\_init = 12)  k\_means.fit(X)  labels = k\_means.labels\_  print(labels)  # Insights, note that each row in our dataset represents a customer, and therefore, each row is assigned a label.  df["Clus\_km"] = labels  df.head(5)   |  |  |  |  |  |  |  |  |  |  |  | | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | |  | Customer Id | Age | Edu | Years Employed | Income | Card Debt | Other Debt | Defaulted | DebtIncomeRatio | Clus\_km | | 0 | 1 | 41 | 2 | 6 | 19 | 0.124 | 1.073 | 0.0 | 6.3 | 0 | | 1 | 2 | 47 | 1 | 26 | 100 | 4.582 | 8.218 | 0.0 | 12.8 | 2 | | 2 | 3 | 33 | 2 | 10 | 57 | 6.111 | 5.802 | 1.0 | 20.9 | 1 | | 3 | 4 | 29 | 2 | 4 | 19 | 0.681 | 0.516 | 0.0 | 6.3 | 0 | | 4 | 5 | 47 | 1 | 31 | 253 | 9.308 | 8.908 | 0.0 | 7.2 | 2 |   We can easily check the centroid values by averaging the features in each cluster.  df.groupby('Clus\_km').mean()   |  |  |  |  |  |  |  |  |  |  | | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | | Clus\_km | Cust\_Id | Age | Edu | Yrs Employed | Income | Card Debt | Oth. Debt | Defaulted | DebtIncome  Ratio | | 0 | 432.468413 | 32.964561 | 1.614792 | 6.374422 | 31.164869 | 1.032541 | 2.104133 | 0.285185 | 10.094761 | | 1 | 410.166667 | 45.388889 | 2.666667 | 19.555556 | 227.166667 | 5.678444 | 10.907167 | 0.285714 | 7.322222 | | 2 | 402.295082 | 41.333333 | 1.956284 | 15.256831 | 83.928962 | 3.103639 | 5.765279 | 0.171233 | 10.724590 |   # Now, let’s look at the distribution of customers based on their age and income:    area = np.pi \* ( X[:, 1])\*\*2  plt.scatter(X[:, 0], X[:, 3], s=area, c=labels.astype(np.float), alpha=0.5)  plt.xlabel('Age', fontsize=18)  plt.ylabel('Income', fontsize=16)  ​plt.show()  ​  # 3d Graph  from mpl\_toolkits.mplot3d import Axes3D  fig = plt.figure(1, figsize=(8, 6))  plt.clf()  ax = Axes3D(fig, rect=[0, 0, .95, 1], elev=48, azim=134)  plt.cla()  # plt.ylabel('Age', fontsize=18)  # plt.xlabel('Income', fontsize=16)  # plt.zlabel('Education', fontsize=16)  ax.set\_xlabel('Education')  ax.set\_ylabel('Age')  ax.set\_zlabel('Income')  ax.scatter(X[:, 1], X[:, 0], X[:, 3], c= labels.astype(np.float))  *k-means will partition your customers into mutually exclusive groups, for example, into 3 clusters. The customers in each cluster are similar to each other demographically. Now we can create a profile for each group, considering the common characteristics of each cluster. For example, the 3 clusters can be:*  AFFLUENT, EDUCATED AND OLD AGED  MIDDLE AGED AND MIDDLE INCOME  YOUNG AND LOW INCOME |
| Hierarchical Clustering  *Welcome to Lab of Hierarchical Clustering with Python using Scipy and Scikit-learn package.*  Hierarchical Clustering - Agglomerative  *We will be looking at a clustering technique, which is Agglomerative Hierarchical Clustering. Remember that agglomerative is the bottom up approach.*  *In this lab, we will be looking at Agglomerative clustering, which is more popular than Divisive clustering.*  *We will also be using Complete Linkage as the Linkage Criteria.*  *NOTE: You can also try using Average Linkage wherever Complete Linkage would be used to see the difference!*  import numpy as np  import pandas as pd  from scipy import ndimage  from scipy.cluster import hierarchy  from scipy.spatial import distance\_matrix  from matplotlib import pyplot as plt  from sklearn import manifold, datasets  from sklearn.cluster import AgglomerativeClustering  from sklearn.datasets.samples\_generator import make\_blobs  %matplotlib inline  Generating Random Data: *We will be generating a set of data using the make\_blobs class.*  # Input these parameters into make\_blobs:  *n\_samples: The total number of points equally divided among clusters*   * *Choose a number from 10-1500*   *centers: The number of centers to generate, or the fixed center locations.*   * *Choose arrays of x,y coordinates for generating the centers. ex. centers = [[1,1], [2,5]])*   *cluster\_std: The standard deviation of the clusters. The larger the number, the further apart the clusters*   * *Choose a number between 0.5-1.5*   # Save the result to X1 and y1.  X1, y1 = make\_blobs(n\_samples=50, centers=[[4,4], [-2, -1], [1, 1], [10,4]], cluster\_std=0.9)  # Plot the scatter plot of the randomly generated data  plt.scatter(X1[:, 0], X1[:, 1], marker='o')  Agglomerative Clustering  *We will start by clustering the random data points we just created.*  *The Agglomerative Clustering class will require two inputs:*  *n\_clusters: The number of clusters to form as well as the number of centroids to generate.*   * *Value will be: 4*   *linkage: Which linkage criterion to use. The linkage criterion determines which distance to use between sets of observation. The algorithm will merge the pairs of clusters that minimize this criterion.*   * *Value will be: 'complete'* * *Note: It is recommended you try everything with 'average' as well*   # Save the result to a variable called agglom  agglom = AgglomerativeClustering(n\_clusters = 4, linkage = 'average')  # Fit the model with X1 and y1 from the generated data above.  agglom.fit(X1,y1)  *Run the following code to show the clustering!*  # Create a figure of size 6 inches by 4 inches.  plt.figure(figsize=(6,4))  ​  # These two lines of code is used to scale the data points down, or else the data points will be scattered very far apart.  ​  x\_min, x\_max = np.min(X1, axis=0), np.max(X1, axis=0) # Create a minimum and maximum range of X1.  ​  X1 = (X1 - x\_min) / (x\_max - x\_min) # Get the average distance for X1.  ​  # This loop displays all of the datapoints.  for i in range(X1.shape[0]):  # Replace the data points with their respective cluster value  # (ex. 0) and is color coded with a colormap (plt.cm.spectral)  plt.text(X1[i, 0], X1[i, 1], str(y1[i]),  color=plt.cm.nipy\_spectral(agglom.labels\_[i] / 10.),  fontdict={'weight': 'bold', 'size': 9})    # Remove the x ticks, y ticks, x and y axis  plt.xticks([])  plt.yticks([])  #plt.axis('off')  ​  ​# Display the plot of the original data before clustering  plt.scatter(X1[:, 0], X1[:, 1], marker='.')  plt.show() # Display the plot  Dendrogram Associated for the Agglomerative Hierarchical Clustering  *Remember that a distance matrix contains the distance from each point to every other point of a dataset.*  *Use the function distance\_matrix, which requires two inputs. Use the Feature Matrix, X2 as both inputs and save the distance matrix to a variable called dist\_matrix*  *Remember that the distance values are symmetric, with a diagonal of 0's. This is one way of making sure your matrix is correct.*  (print out dist\_matrix to make sure it's correct)  dist\_matrix = distance\_matrix(X1,X1)  print(dist\_matrix)  [[0. 0.78282765 0.54605494 ... 0.21581511 0.11556985 0.28037145]  [0.78282765 0. 0.34525758 ... 0.7281957 0.86250104 0.73577491]  [0.54605494 0.34525758 0. ... 0.42330176 0.59282486 0.41389637]  ...  [0.21581511 0.7281957 0.42330176 ... 0. 0.19321347 0.06542633]  [0.11556985 0.86250104 0.59282486 ... 0.19321347 0. 0.24522097]  [0.28037145 0.73577491 0.41389637 ... 0.06542633 0.24522097 0. ]]  *Using the linkage class from hierarchy, pass in the parameters:*   * The distance matrix * 'complete' for complete linkage   # Save the result to a variable called Z  Z = hierarchy.linkage(dist\_matrix, 'complete')  *A Hierarchical clustering is typically visualized as a dendrogram as shown in the following cell. Each merge is represented by a horizontal line. The y-coordinate of the horizontal line is the similarity of the two clusters that were merged, where cities are viewed as singleton clusters. By moving up from the bottom layer to the top node, a dendrogram allows us to reconstruct the history of merges that resulted in the depicted clustering.*  *Next, we will save the dendrogram to a variable called dendro. In doing this, the dendrogram will also be displayed. Using the dendrogram class from hierarchy, pass in the parameter:* Z  dendro = hierarchy.dendrogram(Z)  Practice  *We used complete linkage for our case, change it to average linkage to see how the dendogram changes.*  # Write your code here  Z1 = hierarchy.linkage(dist\_matrix, 'average')  dendro1 = hierarchy.dendrogram(Z1)  Clustering on Vehicle dataset    *Imagine that an automobile manufacturer has developed prototypes for a new vehicle. Before introducing the new model into its range, the manufacturer wants to determine which existing vehicles on the market are most like the prototypes--that is, how vehicles can be grouped, which group is the most similar with the model, and therefore which models they will be competing against.*  *Our objective here, is to use clustering methods, to find the most distinctive clusters of vehicles. It will summarize the existing vehicles and help manufacturers to make decision about the supply of new models.*  # Download data  *To download the data, we will use, !wget to download it from IBM Object Storage.*  !wget -O cars\_clus.csv https://s3-api.us-geo.objectstorage.softlayer.net/.../labs/cars\_clus.csv  # Read data  *Let's read dataset to see what features the manufacturer has collected about the existing models.*  filename = 'cars\_clus.csv'  ​# Read csv  pdf = pd.read\_csv(filename)  print ("Shape of dataset: ", pdf.shape)  ​pdf.head(5)  *The feature sets include price in thousands (price), engine size (engine\_s), horsepower (horsepow), wheelbase (wheelbas), width (width), length (length), curb weight (curb\_wgt), fuel capacity (fuel\_cap) and fuel efficiency (mpg).*  #Data Cleaning  *Let’s simply clear the dataset by dropping the rows that have null value:*  print ("Shape of dataset before cleaning: ", pdf.size)  pdf[[ 'sales', 'resale', 'type', 'price', 'engine\_s', 'horsepow', 'wheelbas', 'width', 'length', 'curb\_wgt', 'fuel\_cap', 'mpg', 'lnsales']] = pdf[[from.. 'sales' to 'lnsales' i.e. same as above highlighted]].apply (pd.to\_numeric, errors='coerce') # Assigns nulls to errors  pdf = pdf.dropna()  pdf = pdf.reset\_index(drop=True)  print ("Shape of dataset after cleaning: ", pdf.size)  pdf.head(5)  # Feature selection  *Let’s select our feature set:*  featureset = pdf[['engine\_s', 'horsepow', 'wheelbas', 'width', 'length', 'curb\_wgt', 'fuel\_cap', 'mpg']]  engine\_s horsepow wheelbas width length curb\_wgt fuel\_cap mpg  0 1.8 140.0 101.2 67.3 172.4 2.639 13.2 28.0  1 3.2 225.0 108.1 70.3 192.9 3.517 17.2 25.0  2 3.5 210.0 114.6 71.4 196.6 3.850 18.0 22.0  3 1.8 150.0 102.6 68.2 178.0 2.998 16.4 27.0  4 2.8 200.0 108.7 76.1 192.0 3.561 18.5 22.0  # Normalization  *Now we can normalize the feature set. MinMaxScaler transforms features by scaling each feature to a given range. It is by default (0, 1). That is, this estimator scales and translates each feature individually such that it is between zero and one.*  from sklearn.preprocessing import MinMaxScaler  x = featureset.values #returns a numpy array  min\_max\_scaler = MinMaxScaler()  feature\_mtx = min\_max\_scaler.fit\_transform(x)  feature\_mtx [0:5]  array([[0.11428571, 0.21518987, 0.18655098, 0.28143713, 0.30625832, 0.2310559 , 0.13364055, 0.43333333],  [0.31428571, 0.43037975, 0.3362256 , 0.46107784, 0.5792277 , 0.50372671, 0.31797235, 0.33333333],  [0.35714286, 0.39240506, 0.47722343, 0.52694611, 0.62849534, 0.60714286, 0.35483871, 0.23333333],  [0.11428571, 0.24050633, 0.21691974, 0.33532934, 0.38082557, 0.34254658, 0.28110599, 0.4 ],  [0.25714286, 0.36708861, 0.34924078, 0.80838323, 0.56724368, 0.5173913 , 0.37788018, 0.23333333]])  # Clustering using Scipy  *In this part we use Scipy package to cluster the dataset: First, we calculate the distance matrix.*  import scipy  leng = feature\_mtx.shape[0] # 117 here as feature\_mtx.shape = (117, 8)  D = scipy.zeros([leng,leng])  for i in range(leng):  for j in range(leng):  D[i,j] = scipy.spatial.distance.euclidean(feature\_mtx[i], feature\_mtx[j]) – check the distance formula  *In agglomerative clustering, at each iteration, the algorithm must update the distance matrix to reflect the distance of the newly formed cluster with the remaining clusters in the forest. The following methods are supported in Scipy for calculating the distance between the newly formed cluster and each: - single - complete - average - weighted - centroid*  *We use complete for our case, but feel free to change it to see how the results change.*  import pylab  import scipy.cluster.hierarchy  Z = hierarchy.linkage(D, 'complete')  *Essentially, Hierarchical clustering does not require a pre-specified number of clusters. However, in some applications we want a partition of disjoint clusters just as in flat clustering. So, you can use a cutting line:*5  from scipy.cluster.hierarchy import fcluster  max\_d = 5  clusters = fcluster(Z, max\_d, criterion='distance')  clusters  *Also, you can determine the number of clusters directly:*  from scipy.cluster.hierarchy import fcluster  k = 5  clusters = fcluster(Z, k, criterion='maxclust')  clusters  ​  *Now, plot the dendrogram:*  fig = pylab.figure(figsize=(18,50))  def llf(id):  return '[%s %s %s]' % (pdf['manufact'][id], pdf['model'][id], int(float(pdf['type'][id])) )    dendro = hierarchy.dendrogram(Z, leaf\_label\_func=llf, leaf\_rotation=0, leaf\_font\_size =12, orientation = 'right')  # Clustering using scikit-learn  *Let’s redo it again, but this time using scikit-learn package:*  dist\_matrix = distance\_matrix(feature\_mtx,feature\_mtx)  print(dist\_matrix)  [[0. 0.57777143 0.75455727 ... 0.28530295 0.24917241 0.18879995]  [0.57777143 0. 0.22798938 ... 0.36087756 0.66346677 0.62201282]  [0.75455727 0.22798938 0. ... 0.51727787 0.81786095 0.77930119]  ...  [0.28530295 0.36087756 0.51727787 ... 0. 0.41797928 0.35720492]  [0.24917241 0.66346677 0.81786095 ... 0.41797928 0. 0.15212198]  [0.18879995 0.62201282 0.77930119 ... 0.35720492 0.15212198 0. ]]  *Now, we can use the 'AgglomerativeClustering' function from scikit-learn library to cluster the dataset. This function performs a hierarchical clustering using a bottom up approach. The linkage criteria determine the metric used for the merge strategy:*  *Ward minimizes the sum of squared differences within all clusters. It is a variance-minimizing approach and in this sense is similar to the k-means objective function but tackled with an agglomerative hierarchical approach.*   * *Maximum or complete linkage minimizes the maximum distance between observations of pairs of clusters.* * *Average linkage minimizes the average of the distances between all observations of pairs of clusters.*   agglom = AgglomerativeClustering(n\_clusters = 6, linkage = 'complete')  agglom.fit(feature\_mtx)  agglom.labels\_  array([1, 2, 2, 1, 2, 3, 1, 2, 2, 2, 2, 2, 3, 3, 2, 1, 1, 2, 2, 2, 5, 1,  4, 1, 1, 2, 1, 2, 1, 1, 1, 5, 0, 0, 0, 3, 2, 1, 2, 1, 2, 3, 2, 3,  0, 3, 0, 1, 1, 1, 2, 3, 1, 1, 1, 2, 1, 1, 2, 2, 2, 3, 3, 3, 1, 1,  1, 2, 1, 2, 2, 1, 1, 2, 3, 2, 3, 1, 2, 3, 5, 1, 1, 2, 3, 2, 1, 3,  2, 3, 1, 1, 2, 1, 1, 2, 2, 2, 1, 1, 1, 1, 1, 1, 1, 1, 2, 1, 1, 1,  2, 0, 1, 1, 1, 1, 1])  *And, we can add a new field to our dataframe to show the cluster of each row:*  pdf['cluster\_'] = agglom.labels\_  pdf.head()   |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  | | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | |  | manufact | model | sales | resale | type | price | engine\_s | horsepow | wheelbas | width | length | curb\_wgt | fuel\_cap | mpg | lnsales | partition | cluster\_ | | 0 | Acura | Integra | 16.919 | 16.360 | 0.0 | 21.50 | 1.8 | 140.0 | 101.2 | 67.3 | 172.4 | 2.639 | 13.2 | 28.0 | 2.828 | 0.0 | 1 | | 1 | Acura | TL | 39.384 | 19.875 | 0.0 | 28.40 | 3.2 | 225.0 | 108.1 | 70.3 | 192.9 | 3.517 | 17.2 | 25.0 | 3.673 | 0.0 | 2 | | 2 | Acura | RL | 8.588 | 29.725 | 0.0 | 42.00 | 3.5 | 210.0 | 114.6 | 71.4 | 196.6 | 3.850 | 18.0 | 22.0 | 2.150 | 0.0 | 2 | | 3 | Audi | A4 | 20.397 | 22.255 | 0.0 | 23.99 | 1.8 | 150.0 | 102.6 | 68.2 | 178.0 | 2.998 | 16.4 | 27.0 | 3.015 | 0.0 | 1 | | 4 | Audi | A6 | 18.780 | 23.555 | 0.0 | 33.95 | 2.8 | 200.0 | 108.7 | 76.1 | 192.0 | 3.561 | 18.5 | 22.0 | 2.933 | 0.0 | 2 |   import matplotlib.cm as cm  n\_clusters = max(agglom.labels\_)+1 # 6  colors = cm.rainbow(np.linspace(0, 1, n\_clusters))  cluster\_labels = list(range(0, n\_clusters)) # [0, 1, 2, 3, 4, 5]  ​  # Create a figure of size 6 inches by 4 inches.  plt.figure(figsize=(16,14))  ​  for color, label in zip(colors, cluster\_labels):  subset = pdf[pdf.cluster\_ == label] # condition for subset  for i in subset.index:  plt.text(subset.horsepow[i], subset.mpg[i],str(subset['model'][i]), rotation=25)  plt.scatter(subset.horsepow, subset.mpg, s= subset.price\*10, c=color, label='cluster'+str(label),alpha=0.5)  # plt.scatter(subset.horsepow, subset.mpg)  plt.legend()  plt.title('Clusters')  plt.xlabel('horsepow')  plt.ylabel('mpg')  *As you can see, we are seeing the distribution of each cluster using the scatter plot, but it is not very clear where is the centroid of each cluster. Moreover, there are 2 types of vehicles in our dataset, "truck" (value of 1 in the type column) and "car" (value of 0 in the type column). So, we use them to distinguish the classes, and summarize the cluster. First, we count the number of cases in each group:*  pdf.groupby(['cluster\_','type'])['cluster\_'].count()  *Now we can look at the characteristics of each cluster:*  agg\_cars = pdf.groupby(['cluster\_','type'])['horsepow','engine\_s','mpg','price'].mean()  agg\_cars   |  |  |  |  |  |  | | --- | --- | --- | --- | --- | --- | | Cluster\_ | type | horsepow | engine\_s | mpg | price | | 0 | 1.0 | 211.666667 | 4.483333 | 16.166667 | 29.024667 | | 1 | 0.0 | 146.531915 | 2.246809 | 27.021277 | 20.306128 | | 1 | 1.0 | 145.000000 | 2.580000 | 22.200000 | 17.009200 | | 2 | 0.0 | 203.111111 | 3.303704 | 24.214815 | 27.750593 | | 2 | 1.0 | 182.090909 | 3.345455 | 20.181818 | 26.265364 | | 3 | 0.0 | 256.500000 | 4.410000 | 21.500000 | 42.870400 | | 4 | 0.0 | 55.000000 | 1.000000 | 45.000000 | 9.235000 | | 5 | 0.0 | 365.666667 | 6.233333 | 19.333333 | 66.010000 |   *It is obvious that we have 3 main clusters with the majority of vehicles in those.*  Cars:  *Cluster 1: with almost high mpg, and low in horsepower.*  *Cluster 2: with good mpg and horsepower, but higher price than average.*  *Cluster 3: with low mpg, high horsepower, highest price.*  Trucks:  Cluster 1: with almost highest mpg among trucks, and lowest in horsepower and price.  Cluster 2: with almost low mpg and medium horsepower, but higher price than average.  Cluster 3: with good mpg and horsepower, low price.  *Please notice that we did not use type, and price of cars in the clustering process, but Hierarchical clustering could forge the clusters and discriminate them with quite high accuracy.*  plt.figure(figsize=(16,10))  for color, label in zip(colors, cluster\_labels):  subset = agg\_cars.loc[(label,),]  for i in subset.index:  plt.text(subset.loc[i][0]+5, subset.loc[i][2], 'type='+str(int(i)) + ', price='+str(int(subset.loc[i][3]))+'k')  plt.scatter(subset.horsepow, subset.mpg, s=subset.price\*20, c=color, label='cluster'+str(label))  plt.legend()  plt.title('Clusters')  plt.xlabel('horsepow')  plt.ylabel('mpg')  ​ |
| Density-Based Clustering  *Most of the traditional clustering techniques, such as k-means, hierarchical and fuzzy clustering, can be used to group data without supervision.*  *However, when applied to tasks with arbitrary shape clusters, or clusters within cluster, the traditional techniques might be unable to achieve good results. That is, elements in the same cluster might not share enough similarity or the performance may be poor. Additionally, Density-based Clustering locates regions of high density that are separated from one another by regions of low density. Density, in this context, is defined as the number of points within a specified radius.*  *In this section, the main focus will be manipulating the data and properties of DBSCAN and observing the resulting clustering.*  # For visualization of map, you need basemap package.  #if you dont have basemap install on your machine, you can use the following line to install it  #!conda install -c conda-forge basemap==1.1.0 matplotlib==2.2.2 -y  #Notice: you maight have to refresh your page and re-run the notebook after installation  # Import the following libraries:  import numpy as np  from sklearn.cluster import DBSCAN  from sklearn.datasets.samples\_generator import make\_blobs  from sklearn.preprocessing import StandardScaler  import matplotlib.pyplot as plt  %matplotlib inline # remember %matplotlib inline to display plots  Data generation  *The function below will generate the data points and requires these inputs:*   * *centroidLocation: Coordinates of the centroids that will generate the random data.*   *Example: input: [[4, 3], [2, -1], [-1, 4]]*   * *numSamples: The number of data points we want generated, split over the number of centroids (# of centroids defined in centroidLocation), Example: 1500* * *clusterDeviation: The standard deviation between the clusters. The larger the number, the further the spacing.*   *Example: 0.5*  def createDataPoints(centroidLocation, numSamples, clusterDeviation):  # Create random data and store in feature matrix X and response vector y.  X, y = make\_blobs(n\_samples=numSamples, centers=centroidLocation, cluster\_std=clusterDeviation)    # Standardize features by removing the mean and scaling to unit variance  X = StandardScaler().fit\_transform(X)  return X, y  *Use createDataPoints with the 3 inputs and store the output into variables X and y.*  X, y = createDataPoints([[4,3], [2,-1], [-1,4]] , 1500, 0.5)  Modelling  *DBSCAN stands for Density-Based Spatial Clustering of Applications with Noise. This technique is one of the most common clustering algorithms which works based on density of object. The whole idea is that if a particular point belongs to a cluster, it should be near to lots of other points in that cluster.*  *It works based on two parameters: Epsilon and Minimum Points*   * **Epsilon** determine a specified radius that if includes enough number of points within, we call it dense area * **minimumSamples** determine the minimum number of data points we want in a neighbourhood to define a cluster   epsilon = 0.3  minimumSamples = 7  db = DBSCAN(eps=epsilon, min\_samples=minimumSamples).fit(X)  labels = db.labels\_  labels  array([0, 1, 1, ..., 0, 1, 2])  Distinguish outliers  *Let’s Replace all elements with 'True' in core\_samples\_mask that are in the cluster, 'False' if the points are outliers.*  # First, create an array of booleans using the labels from db.  core\_samples\_mask = np.zeros\_like(db.labels\_, dtype=bool)  core\_samples\_mask[db.core\_sample\_indices\_] = True  core\_samples\_mask  array([ True, True, True, ..., True, True, True])  # Number of clusters in labels, ignoring noise if present.  n\_clusters\_ = len(set(labels)) - (1 if -1 in labels else 0)  n\_clusters\_  3  Notice: labels 🡪 set(labels) is array([0, 1, 1, ..., 0, 1, 2]) 🡪 {-1, 0, 1, 2} # remember set only keeps distinct value  # Remove repetition in labels by turning it into a set.  unique\_labels = set(labels)  unique\_labels  {-1, 0, 1, 2}  Data visualization  # Create colors for the clusters.  colors = plt.cm.Spectral(np.linspace(0, 1, len(unique\_labels)))  ​  array([[0.61960784, 0.00392157, 0.25882353, 1. ],  [0.99346405, 0.74771242, 0.43529412, 1. ],  [0.74771242, 0.89803922, 0.62745098, 1. ],  [0.36862745, 0.30980392, 0.63529412, 1. ]])  # Plot the points with colors  for k, col in zip(unique\_labels, colors):  if k == -1:  # Black used for noise.  col = 'k'  class\_member\_mask = (labels == k)  # Plot the datapoints that are clustered  xy = X[class\_member\_mask & core\_samples\_mask]  plt.scatter(xy[:, 0], xy[:, 1],s=50, c=[col], marker=u'o', alpha=0.5)  # Plot the outliers  xy = X[class\_member\_mask & ~core\_samples\_mask]  plt.scatter(xy[:, 0], xy[:, 1],s=50, c=[col], marker=u'o', alpha=0.5)  Practice  *To better understand differences between partitional and density-based clustering, try to cluster the above dataset into 3 clusters using k-Means. Notice: do not generate data again, use the same dataset as above.*  # write your code here  from sklearn.cluster import KMeans  k = 3  k\_means3 = KMeans(init = "k-means++", n\_clusters = k, n\_init = 12)  k\_means3.fit(X)  fig = plt.figure(figsize=(6, 4))  ax = fig.add\_subplot(1, 1, 1)  for k, col in zip(range(k), colors):  my = (k\_means3.labels\_ == k)  plt.scatter(X[my, 0], X[my, 1], c=col, marker=u'o', alpha=0.5)  plt.show()  Weather Station Clustering using DBSCAN & scikit-learn  *DBSCAN is specially very good for tasks like class identification on a spatial context. The wonderful attribute of DBSCAN algorithm is that it can find out any arbitrary shape cluster without getting affected by noise. For example, this following example cluster the location of weather stations in Canada.*  *DBSCAN can be used here, for instance, to find the group of stations which show the same weather condition. As you can see, it not only finds different arbitrary shaped clusters, can find the denser part of data-cantered samples by ignoring less-dense areas or noises.*  *let's start playing with the data. We will be working according to the following workflow:*   * Loading data * Overview data * Data cleaning * Data selection * Clustering   # About the dataset  *Environment Canada Monthly Values for July - 2015*   |  |  | | --- | --- | | Column Name | Meaning | | **Stn\_Name** | **Station Name** | | **Lat** | **Latitude (North+, degrees)** | | **Long** | **Longitude (West - , degrees)** | | Prov | Province | | Tm | Mean Temperature (°C) | | DwTm | Days without Valid Mean Temperature | | D | Mean Temperature difference from Normal (1981-2010) (°C) | | Tx | Highest Monthly Maximum Temperature (°C) | | DwTx | Days without Valid Maximum Temperature | | Tn | Lowest Monthly Minimum Temperature (°C) | | DwTn | Days without Valid Minimum Temperature | | S | Snowfall (cm) | | DwS | Days without Valid Snowfall | | S%N | Percent of Normal (1981-2010) Snowfall | | **P** | **Total Precipitation (mm)** | | DwP | Days without Valid Precipitation | | P%N | Percent of Normal (1981-2010) Precipitation | | S\_G | Snow on the ground at the end of the month (cm) | | Pd | Number of days with Precipitation 1.0 mm or more | | BS | Bright Sunshine (hours) | | DwBS | Days without Valid Bright Sunshine | | BS% | Percent of Normal (1981-2010) Bright Sunshine | | HDD | Degree Days below 18 °C | | CDD | Degree Days above 18 °C | | Stn\_No | Climate station identifier (first 3 digits indicate drainage basin, last 4 characters are for sorting alphabetically). | | NA | Not Available |   1- Download data  !wget -O weather-stations20140101-20141231.csv https://s3-api.us-geo.. labs/weather-stations…-20141231.csv  2- Load the dataset  We will import the .csv then we creates the columns for year, month and day.  import csv; import pandas as pd; import numpy as np  ​  filename='weather-stations20140101-20141231.csv'  ​  #Read csv  pdf = pd.read\_csv(filename)  pdf.head(5)  3-Cleaning  Let’s remove rows that dont have any value in the Tm field.  pdf = pdf[pd.notnull(pdf["Tm"])]  pdf = pdf.reset\_index(drop=True)  pdf.head(5)  4-Visualization  *Visualization of stations on map using basemap package. The matplotlib basemap toolkit is a library for plotting 2D data on maps in Python. basemap does not do any plotting on it’s own, but provides the facilities to transform coordinates to a map projections.*  *Please notice that the size of each data points represents the average of maximum temperature for each station in a year.*  from mpl\_toolkits.basemap import Basemap; import matplotlib.pyplot as plt; from pylab import rcParams  %matplotlib inline  rcParams['figure.figsize'] = (14,10)  llon=-140; ulon=-50; llat=40; ulat=65  pdf = pdf[(pdf['Long'] > llon) & (pdf['Long'] < ulon) & (pdf['Lat'] > llat) &(pdf['Lat'] < ulat)]  my\_map = Basemap(projection='merc', resolution = 'l', area\_thresh = 1000.0,  llcrnrlon=llon, llcrnrlat=llat, #min longitude (llcrnrlon) and latitude (llcrnrlat)  urcrnrlon=ulon, urcrnrlat=ulat) #max longitude (urcrnrlon) and latitude (urcrnrlat)  my\_map.drawcoastlines()  my\_map.drawcountries()  # my\_map.drawmapboundary()  my\_map.fillcontinents(color = 'white', alpha = 0.3)  my\_map.shadedrelief()  # To collect data based on stations  xs,ys = my\_map(np.asarray(pdf.Long), np.asarray(pdf.Lat))  pdf['xm']= xs.tolist()  pdf['ym'] =ys.tolist()  #Visualization1  for index,row in pdf.iterrows():  # x,y = my\_map(row.Long, row.Lat)  my\_map.plot(row.xm, row.ym,markerfacecolor =([1,0,0]), marker='o', markersize= 5, alpha = 0.75)  #plt.text(x,y,stn)  plt.show()    5- Clustering of stations based on their location i.e. Lat & Lon  *DBSCAN form sklearn library can runs DBSCAN clustering from vector array or distance matrix. In our case, we pass it the Numpy array Clus\_dataSet to find core samples of high density and expands clusters from them.*  from sklearn.cluster import DBSCAN  import sklearn.utils  from sklearn.preprocessing import StandardScaler  sklearn.utils.check\_random\_state(1000)  Clus\_dataSet = pdf[['xm','ym']]  Clus\_dataSet = np.nan\_to\_num(Clus\_dataSet)  Clus\_dataSet = StandardScaler().fit\_transform(Clus\_dataSet)  # Compute DBSCAN  db = DBSCAN(eps=0.15, min\_samples=10).fit(Clus\_dataSet)  core\_samples\_mask = np.zeros\_like(db.labels\_, dtype=bool)  core\_samples\_mask[db.core\_sample\_indices\_] = True  labels = db.labels\_  pdf["Clus\_Db"]=labels  realClusterNum=len(set(labels)) - (1 if -1 in labels else 0)  clusterNum = len(set(labels))  # A sample of clusters  pdf[["Stn\_Name","Tx","Tm","Clus\_Db"]].head(5)   |  |  |  |  |  | | --- | --- | --- | --- | --- | | Stn\_Name | Tx | Tm | Clus\_Db |  | | 0 | CHEMAINUS | 13.5 | 8.2 | 0 | | 1 | COWICHAN LAKE FORESTRY | 15.0 | 7.0 | 0 | | 2 | LAKE COWICHAN | 16.0 | 6.8 | 0 | | 3 | DUNCAN KELVIN CREEK | 14.5 | 7.7 | 0 | | 4 | ESQUIMALT HARBOUR | 13.1 | 8.8 | 0 |   *As you can see for outliers, the cluster label is -1*  set(labels)  {-1, 0, 1, 2, 3, 4}  6- Visualization of clusters based on location  *Now, we can visualize the clusters using basemap:*  from mpl\_toolkits.basemap import Basemap  import matplotlib.pyplot as plt  from pylab import rcParams  %matplotlib inline  rcParams['figure.figsize'] = (14,10)  ​  my\_map = Basemap(projection='merc',  resolution = 'l', area\_thresh = 1000.0,  llcrnrlon=llon, llcrnrlat=llat, #min longitude (llcrnrlon) and latitude (llcrnrlat)  urcrnrlon=ulon, urcrnrlat=ulat) #max longitude (urcrnrlon) and latitude (urcrnrlat)  ​  my\_map.drawcoastlines()  my\_map.drawcountries()  #my\_map.drawmapboundary()  my\_map.fillcontinents(color = 'white', alpha = 0.3)  my\_map.shadedrelief()  ​  # To create a color map  colors = plt.get\_cmap('jet')(np.linspace(0.0, 1.0, clusterNum))  ​​  #Visualization1  for clust\_number in set(labels):  c=(([0.4,0.4,0.4]) if clust\_number == -1 else colors[np.int(clust\_number)])  clust\_set = pdf[pdf.Clus\_Db == clust\_number]  my\_map.scatter(clust\_set.xm, clust\_set.ym, color =c, marker='o', s= 20, alpha = 0.85)  if clust\_number != -1:  cenx=np.mean(clust\_set.xm)  ceny=np.mean(clust\_set.ym)  plt.text(cenx,ceny,str(clust\_number), fontsize=25, color='red',)  print ("Cluster "+str(clust\_number)+', Avg Temp: '+ str(np.mean(clust\_set.Tm)))    7- Clustering of stations based on their location, mean, max, and min Temperature  In this section we re-run DBSCAN, but this time on a 5-dimensional dataset:  from sklearn.cluster import DBSCAN  import sklearn.utils  from sklearn.preprocessing import StandardScaler  sklearn.utils.check\_random\_state(1000)  Clus\_dataSet = pdf[['xm','ym','Tx','Tm','Tn']]  Clus\_dataSet = np.nan\_to\_num(Clus\_dataSet)  Clus\_dataSet = StandardScaler().fit\_transform(Clus\_dataSet)  ​  # Compute DBSCAN  db = DBSCAN(eps=0.3, min\_samples=10).fit(Clus\_dataSet)  core\_samples\_mask = np.zeros\_like(db.labels\_, dtype=bool)  core\_samples\_mask[db.core\_sample\_indices\_] = True  labels = db.labels\_  pdf["Clus\_Db"]=labels  ​  realClusterNum=len(set(labels)) - (1 if -1 in labels else 0)  clusterNum = len(set(labels))  ​  # A sample of clusters  pdf[["Stn\_Name","Tx","Tm","Clus\_Db"]].head(5)   |  |  |  |  |  | | --- | --- | --- | --- | --- | | Stn\_Name | Tx | Tm | Clus\_Db |  | | 0 | CHEMAINUS | 13.5 | 8.2 | 0 | | 1 | COWICHAN LAKE FORESTRY | 15.0 | 7.0 | 0 | | 2 | LAKE COWICHAN | 16.0 | 6.8 | 0 | | 3 | DUNCAN KELVIN CREEK | 14.5 | 7.7 | 0 | | 4 | ESQUIMALT HARBOUR | 13.1 | 8.8 | 0 |   8- Visualization of clusters based on location and Temperture  from mpl\_toolkits.basemap import Basemap  import matplotlib.pyplot as plt  from pylab import rcParams  %matplotlib inline  rcParams['figure.figsize'] = (14,10)  ​  my\_map = Basemap(projection='merc',  resolution = 'l', area\_thresh = 1000.0,  llcrnrlon=llon, llcrnrlat=llat, #min longitude (llcrnrlon) and latitude (llcrnrlat)  urcrnrlon=ulon, urcrnrlat=ulat) #max longitude (urcrnrlon) and latitude (urcrnrlat)  ​  my\_map.drawcoastlines()  my\_map.drawcountries()  #my\_map.drawmapboundary()  my\_map.fillcontinents(color = 'white', alpha = 0.3)  my\_map.shadedrelief()  ​  # To create a color map  colors = plt.get\_cmap('jet')(np.linspace(0.0, 1.0, clusterNum))  ​​  #Visualization1  for clust\_number in set(labels):  c=(([0.4,0.4,0.4]) if clust\_number == -1 else colors[np.int(clust\_number)])  clust\_set = pdf[pdf.Clus\_Db == clust\_number]  my\_map.scatter(clust\_set.xm, clust\_set.ym, color =c, marker='o', s= 20, alpha = 0.85)  if clust\_number != -1:  cenx=np.mean(clust\_set.xm)  ceny=np.mean(clust\_set.ym)  plt.text(cenx,ceny,str(clust\_number), fontsize=25, color='red',)  print ("Cluster "+str(clust\_number)+', Avg Temp: '+ str(np.mean(clust\_set.Tm))) |
| Recommendation systems    CONTENT-BASED FILTERING  *Recommendation systems are a collection of algorithms used to recommend items to users based on information taken from the user. These systems have become ubiquitous, and can be commonly seen in online stores, movies databases and job finders. In this notebook, we will explore Content-based recommendation systems and implement a simple version of one using Python and the Pandas library.*   * Acquiring the Data * Pre-processing * Content-Based Filtering   Acquiring the Data  *To acquire and extract the data, simply run the following Bash scripts:*  *Dataset acquired from GroupLens. Let’s download the dataset. To download the data, we will use !wget.*  !wget -O moviedataset.zip https://s3-api.us-geo.objectstorage.softlayer.net/.../labs/moviedataset.zip  print('unziping ...')  !unzip -o -j moviedataset.zip  Now you're ready to start working with the data!  Pre-processing  *First, let's get all of the imports out of the way:*  # Dataframe manipulation library  import pandas as pd  # Math functions, we'll only need the sqrt function so let's import only that  from math import sqrt  import numpy as np  import matplotlib.pyplot as plt  %matplotlib inline  *Now let's read each file into their Dataframes:*  # Storing the movie information into a pandas dataframe  movies\_df = pd.read\_csv('movies.csv')  # Storing the user information into a pandas dataframe  ratings\_df = pd.read\_csv('ratings.csv')  # Head is a function that gets the first N rows of a dataframe. N's default is 5.  movies\_df.head()   |  |  |  |  | | --- | --- | --- | --- | |  | movieId | title | genres | | 0 | 1 | Toy Story (1995) | Adventure|Animation|Children|Comedy|Fantasy | | 1 | 2 | Jumanji (1995) | Adventure|Children|Fantasy | | 2 | 3 | Grumpier Old Men (1995) | Comedy|Romance | | 3 | 4 | Waiting to Exhale (1995) | Comedy|Drama|Romance | | 4 | 5 | Father of the Bride Part II (1995) | Comedy |   Let's also remove the year from the title column by using pandas replace function and store in a new year column.  # Using regular expressions to find a year stored between parentheses  # We specify the parentheses so we don't conflict with movies that have years in their titles  movies\_df['year'] = movies\_df.title.str.extract('(\(\d\d\d\d\))',expand=False)  # Removing the parentheses  movies\_df['year'] = movies\_df.year.str.extract('(\d\d\d\d)',expand=False)  # Removing the years from the 'title' column  movies\_df['title'] = movies\_df.title.str.replace('(\(\d\d\d\d\))', '')  # Applying the strip function to get rid of any ending whitespace characters that may have appeared  movies\_df['title'] = movies\_df['title'].apply(lambda x: x.strip())  movies\_df.head()   |  |  |  |  |  | | --- | --- | --- | --- | --- | |  | movieId | title | genres | year | | 0 | 1 | Toy Story | Adventure|Animation|Children|Comedy|Fantasy | 1995 | | 1 | 2 | Jumanji | Adventure|Children|Fantasy | 1995 | | 2 | 3 | Grumpier Old Men | Comedy|Romance | 1995 | | 3 | 4 | Waiting to Exhale | Comedy|Drama|Romance | 1995 | | 4 | 5 | Father of the Bride Part II | Comedy | 1995 |   *With that, let's also split the values in the Genres column into a list of Genres to simplify future use. This can be achieved by applying Python's split string function on the correct column.*  # Every genre is separated by a | so we simply have to call the split function on |  movies\_df['genres'] = movies\_df.genres.str.split('|')   |  |  |  |  |  | | --- | --- | --- | --- | --- | |  | movieId | title | genres | year | | 0 | 1 | Toy Story | [Adventure, Animation, Children, Comedy, Fantasy] | 1995 | | 1 | 2 | Jumanji | [Adventure, Children, Fantasy] | 1995 | | 2 | 3 | Grumpier Old Men | [Comedy, Romance] | 1995 | | 3 | 4 | Waiting to Exhale | [Comedy, Drama, Romance] | 1995 | | 4 | 5 | Father of the Bride Part II | [Comedy] | 1995 |   *Since keeping genres in a list format isn't optimal for the content-based recommendation system technique, we will use the One Hot Encoding technique to convert the list of genres to a vector where each column corresponds to one possible value of the feature. This encoding is needed for feeding categorical data. In this case, we store every different genre in columns that contain either 1 or 0. 1 shows that a movie has that genre and 0 shows that it doesn't. Let's also store this dataframe in another variable since genres won't be important for our first recommendation system.*  # Copying the movie dataframe into a new one since we won't need to use the genre information in our first case.  moviesWithGenres\_df = movies\_df.copy()  ​  # For every row in the dataframe, iterate through the list of genres and place a 1 into the corresponding column  for index, row in movies\_df.iterrows():  for genre in row['genres']:  moviesWithGenres\_df.at[index, genre] = 1  # Filling in the NaN values with 0 to show that a movie doesn't have that column's genre  moviesWithGenres\_df = moviesWithGenres\_df.fillna(0)  moviesWithGenres\_df.head()   |  |  |  |  |  |  |  |  |  |  |  |  |  |  | | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | |  | movieId | title | genres | year | Adventure | Animation | Children | Comedy | Fantasy | ... | Horror | Musical | nogenres | | 0 | 1 | Toy Story | [Adventure, Animation, Children, Comedy, Fantasy] | 1995 | 1.0 | 1.0 | 1.0 | 1.0 | 1.0 | ... | 0.0 | 0.0 | 0.0 | | 1 | 2 | Jumanji | [Adventure, Children, Fantasy] | 1995 | 1.0 | 0.0 | 1.0 | 0.0 | 1.0 | ... | 0.0 | 0.0 | 0.0 | | 2 | 3 | Grumpier Old Men | [Comedy, Romance] | 1995 | 0.0 | 0.0 | 0.0 | 1.0 | 0.0 | ... | 0.0 | 0.0 | 0.0 | | 3 | 4 | Waiting to Exhale | [Comedy, Drama, Romance] | 1995 | 0.0 | 0.0 | 0.0 | 1.0 | 0.0 | ... | 0.0 | 0.0 | 0.0 | | 4 | 5 | Father of the Bride Part II | [Comedy] | 1995 | 0.0 | 0.0 | 0.0 | 1.0 | 0.0 | ... | 0.0 | 0.0 | 0.0 |   5 rows × 24 columns  *Next, let's look at the ratings dataframe.*  ratings\_df.head()   |  |  |  |  |  | | --- | --- | --- | --- | --- | |  | userId | movieId | rating | timestamp | | 0 | 1 | 169 | 2.5 | 1204927694 | | 1 | 1 | 2471 | 3.0 | 1204927438 | | 2 | 1 | 48516 | 5.0 | 1204927435 | | 3 | 2 | 2571 | 3.5 | 1436165433 | | 4 | 2 | 109487 | 4.0 | 1436165496 |   *Every row in the ratings dataframe has a user id associated with at least one movie, a rating and a timestamp showing when they reviewed it. We won't be needing the timestamp column, so let's drop it to save on memory.*  # Drop removes a specified row or column from a dataframe  ratings\_df = ratings\_df.drop('timestamp', 1)  ratings\_df.head()   |  |  |  |  | | --- | --- | --- | --- | |  | userId | movieId | rating | | 0 | 1 | 169 | 2.5 | | 1 | 1 | 2471 | 3.0 | | 2 | 1 | 48516 | 5.0 | | 3 | 2 | 2571 | 3.5 | | 4 | 2 | 109487 | 4.0 |   Content-Based recommendation system  *Now, let's take a look at how to implement Content-Based or Item-Item recommendation systems. This technique attempts to figure out what a user's favourite aspects of an item is, and then recommends items that present those aspects. In our case, we're going to try to figure out the input's favorite genres from the movies and ratings given.*  *Let's begin by creating an input user to recommend movies to:*  *Notice: To add more movies, simply increase the number of elements in the userInput. Feel free to add more in! Just be sure to write it in with capital letters and if a movie starts with a "The", like "The Matrix" then write it in like this: 'Matrix, The'.*  userInput = [  {'title':'Breakfast Club, The', 'rating':5},  {'title':'Toy Story', 'rating':3.5},  {'title':'Jumanji', 'rating':2},  {'title':"Pulp Fiction", 'rating':5},  {'title':'Akira', 'rating':4.5}  ]  inputMovies = pd.DataFrame(userInput)  inputMovies   |  |  |  | | --- | --- | --- | |  | title | rating | | 0 | Breakfast Club, The | 5.0 | | 1 | Toy Story | 3.5 | | 2 | Jumanji | 2.0 | | 3 | Pulp Fiction | 5.0 | | 4 | Akira | 4.5 |   # Add movieId to input user  *With the input complete, let's extract the input movie's ID's from the movies dataframe and add them into it.*  *We can achieve this by first filtering out the rows that contain the input movie's title and then merging this subset with the input dataframe. We also drop unnecessary columns for the input to save memory space.*  # Filtering out the movies by title  inputId = movies\_df[movies\_df['title'].isin(inputMovies['title'].tolist())]  # Then merging it so we can get the movieId. It's implicitly merging it by title.  inputMovies = pd.merge(inputId, inputMovies)  # Dropping information, we won't use from the input dataframe  inputMovies = inputMovies.drop('genres', 1).drop('year', 1)  # Final input dataframe  inputMovies   |  |  |  |  | | --- | --- | --- | --- | |  | movieId | title | rating | | 0 | 1 | Toy Story | 3.5 | | 1 | 2 | Jumanji | 2.0 | | 2 | 296 | Pulp Fiction | 5.0 | | 3 | 1274 | Akira | 4.5 | | 4 | 1968 | Breakfast Club, The | 5.0 |   # If a movie you added in above isn't here, then it might not be in the original  # dataframe or it might spelled differently, please check capitalisation.  *We're going to start by learning the input's preferences, so let's get the subset of movies that the input (user,groups) has watched from the Dataframe containing genres defined with binary values.*  # Filtering out the movies from the input  userMovies = moviesWithGenres\_df[moviesWithGenres\_df['movieId'].isin(inputMovies['movieId'].tolist())]  userMovies   |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  | | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | |  | movieId | title | genres | year | Adventure | Animation | Children | Comedy | Fantasy | Romance | ... | Horror | Western | Film-Noir | nogenres | | 0 | 1 | Toy Story | [Adventure, Animation, Children, Comedy, Fantasy] | 1995 | 1.0 | 1.0 | 1.0 | 1.0 | 1.0 | 0.0 | ... | 0.0 | 0.0 | 0.0 | 0.0 | | 1 | 2 | Jumanji | [Adventure, Children, Fantasy] | 1995 | 1.0 | 0.0 | 1.0 | 0.0 | 1.0 | 0.0 | ... | 0.0 | 0.0 | 0.0 | 0.0 | | 293 | 296 | Pulp Fiction | [Comedy, Crime, Drama, Thriller] | 1994 | 0.0 | 0.0 | 0.0 | 1.0 | 0.0 | 0.0 | ... | 0.0 | 0.0 | 0.0 | 0.0 | | 1246 | 1274 | Akira | [Action, Adventure, Animation, Sci-Fi] | 1988 | 1.0 | 1.0 | 0.0 | 0.0 | 0.0 | 0.0 | ... | 0.0 | 0.0 | 0.0 | 0.0 | | 1885 | 1968 | Breakfast Club, The | [Comedy, Drama] | 1985 | 0.0 | 0.0 | 0.0 | 1.0 | 0.0 | 0.0 | ... | 0.0 | 0.0 | 0.0 | 0.0 |   5 rows × 24 columns  *We'll only need the actual genre table, so let's clean this up a bit by resetting the index and dropping the movieId, title, genres and year columns.*  # Resetting the index to avoid future issues  userMovies = userMovies.reset\_index(drop=True)  # Dropping unnecessary issues due to save memory and to avoid issues  userGenreTable = userMovies.drop('movieId', 1).drop('title', 1).drop('genres', 1).drop('year', 1)  userGenreTable   |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  | | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | |  | Adventure | Animation | Children | Comedy | Fantasy | Romance | Drama | Action | Crime | Thriller | Horror | Mystery | Sci-Fi | IMAX | Documentary | War | Musical | Western | Film-Noir | No genres | | 0 | 1.0 | 1.0 | 1.0 | 1.0 | 1.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | | 1 | 1.0 | 0.0 | 1.0 | 0.0 | 1.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | | 2 | 0.0 | 0.0 | 0.0 | 1.0 | 0.0 | 0.0 | 1.0 | 0.0 | 1.0 | 1.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | | 3 | 1.0 | 1.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 1.0 | 0.0 | 0.0 | 0.0 | 0.0 | 1.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | | 4 | 0.0 | 0.0 | 0.0 | 1.0 | 0.0 | 0.0 | 1.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 |   *Now we're ready to start learning the input's preferences!*  **userProfile:**  Adventure 10.0  Animation 8.0  Children 5.5  Comedy 13.5  Fantasy 5.5  Romance 0.0  Drama 10.0  Action 4.5  Crime 5.0  Thriller 5.0  …. 0.0  Sci-Fi 4.5  Documentary 0.0  War 0.0  Musical 0.0  Western 0.0  Film-Noir 0.0  no genres 0.0  dtype: float64  *To do this, we're going to turn each genre into weights. We can do this by using the input's reviews and multiplying them into the input's genre table and then summing up the resulting table by column. This operation is actually a dot product between a matrix and a vector, so we can simply accomplish by calling Pandas's "dot" function.*  inputMovies['rating']  0 3.5  1 2.0  2 5.0  3 4.5  4 5.0  Name: rating, dtype: float64  # Dot produt to get weights  userProfile = userGenreTable.transpose().dot(inputMovies['rating'])  # The user profile  userProfile 🡪  Now, we have the weights for every of the user's preferences. This is known as the User Profile. Using this, we can recommend movies that satisfy the user's preferences.  *Let's start by extracting the genre table from the original dataframe:*  # Now let's get the genres of every movie in our original dataframe  genreTable = moviesWithGenres\_df.set\_index(moviesWithGenres\_df['movieId'])  # And drop the unnecessary information  genreTable = genreTable.drop('movieId', 1).drop('title', 1).drop('genres', 1).drop('year', 1)  genreTable.head()   |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  | | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | | movieId | Adventure | Animation | Children | Comedy | Fantasy | Romance | Drama | Action | Crime | Thriller | Horror | Mystery | Sci-Fi | IMAX | Documentary | War | Musical | Western | Film-Noir | No genres | | 1 | 1.0 | 1.0 | 1.0 | 1.0 | 1.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | | 2 | 1.0 | 0.0 | 1.0 | 0.0 | 1.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | | 3 | 0.0 | 0.0 | 0.0 | 1.0 | 0.0 | 1.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | | 4 | 0.0 | 0.0 | 0.0 | 1.0 | 0.0 | 1.0 | 1.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | | 5 | 0.0 | 0.0 | 0.0 | 1.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 |   genreTable.shape  (34208, 20)  *With the input's profile and the complete list of movies and their genres in hand, we're going to take the weighted average of every movie based on the input profile and recommend the top twenty movies that most satisfy it.*  # Multiply the genres by the weights and then take the weighted average  recommendationTable\_df = ((genreTable\*userProfile).sum(axis=1))/(userProfile.sum())  recommendationTable\_df.head()  movieId  1 0.594406  2 0.293706  3 0.188811  4 0.328671  5 0.188811  dtype: float64  # Sort our recommendations in descending order  recommendationTable\_df = recommendationTable\_df.sort\_values(ascending=False)  # Just a peek at the values  recommendationTable\_df.head()  movieId  5018 0.748252  26093 0.734266  27344 0.720280  148775 0.685315  6902 0.678322  dtype: float64  *Now here's the recommendation table!*  # The final recommendation table  movies\_df.loc[movies\_df['movieId'].isin(recommendationTable\_df.head(20).keys())]  Advantages and Disadvantages of Content-Based Filtering  *Advantages*   * *Learns user's preferences* * *Highly personalized for the user*   *Disadvantages*   * *Doesn't take into account what others think of the item, so low-quality item recommendations might happen* * *Extracting data is not always intuitive* * *Determining what characteristics of the item the user dislikes or likes is not always obvious* |
| CONTENT-BASED FILTERING  *Recommendation systems are a collection of algorithms used to recommend items to users based on information taken from the user. These systems have become ubiquitous can be commonly seen in online stores, movies databases and job finders. In this notebook, we will explore recommendation systems based on Collaborative Filtering and implement simple version of one using Python and the Pandas library.*  Acquiring the Data  *Dataset acquired from GroupLens. Let’s download the dataset. To download the data, we will use !wget.*  !wget -O moviedataset.zip https://s3-api.us-geo.objectstorage.softlayer.net/.../labs/moviedataset.zip  print('unziping ...')  !unzip -o -j moviedataset.zip  Preprocessing  *First, let's get all of the imports out of the way:*  # Dataframe manipulation library  import pandas as pd  # Math functions, we'll only need the sqrt function so let's import only that  from math import sqrt  import numpy as np  import matplotlib.pyplot as plt  %matplotlib inline  *Now let's read each file into their Dataframes:*  # Storing the movie information into a pandas dataframe  movies\_df = pd.read\_csv('movies.csv')  # Storing the user information into a pandas dataframe  ratings\_df = pd.read\_csv('ratings.csv')  *Let's also take a peek at how each of them is organized:*  # Head is a function that gets the first N rows of a dataframe. N's default is 5.  movies\_df.head()   |  |  |  |  | | --- | --- | --- | --- | |  | movieId | title | genres | | 0 | 1 | Toy Story (1995) | Adventure|Animation|Children|Comedy|Fantasy | | 1 | 2 | Jumanji (1995) | Adventure|Children|Fantasy | | 2 | 3 | Grumpier Old Men (1995) | Comedy|Romance | | 3 | 4 | Waiting to Exhale (1995) | Comedy|Drama|Romance | | 4 | 5 | Father of the Bride Part II (1995) | Comedy |   *So, each movie has a unique ID, a title with its release year along with it (Which may contain unicode characters) and several different genres in the same field. Let's remove the year from the title column and place it into its own one by using the handy extract function that Pandas has.*  *Let's remove the year from the title column by using pandas' replace function and store in a new year column.*  # Using regular expressions to find a year stored between parentheses  # We specify the parantheses so we don't conflict with movies that have years in their titles  movies\_df['year'] = movies\_df.title.str.extract('(\(\d\d\d\d\))',expand=False)  # Removing the parentheses  movies\_df['year'] = movies\_df.year.str.extract('(\d\d\d\d)',expand=False)  # Removing the years from the 'title' column  movies\_df['title'] = movies\_df.title.str.replace('(\(\d\d\d\d\))', '')  # Applying the strip function to get rid of any ending whitespace characters that may have appeared  movies\_df['title'] = movies\_df['title'].apply(lambda x: x.strip())   |  |  |  |  |  | | --- | --- | --- | --- | --- | |  | movieId | title | genres | year | | 0 | 1 | Toy Story | Adventure|Animation|Children|Comedy|Fantasy | 1995 | | 1 | 2 | Jumanji | Adventure|Children|Fantasy | 1995 | | 2 | 3 | Grumpier Old Men | Comedy|Romance | 1995 | | 3 | 4 | Waiting to Exhale | Comedy|Drama|Romance | 1995 | | 4 | 5 | Father of the Bride Part II | Comedy | 1995 |   *With that, let's also drop the genres column since we won't need it for this particular recommendation system.*  # Dropping the genres column  movies\_df = movies\_df.drop('genres', 1)  movies\_df.head()  *Next, let's look at the ratings dataframe.*  ratings\_df.head()   |  |  |  |  |  | | --- | --- | --- | --- | --- | |  | userId | movieId | rating | timestamp | | 0 | 1 | 169 | 2.5 | 1204927694 | | 1 | 1 | 2471 | 3.0 | 1204927438 | | 2 | 1 | 48516 | 5.0 | 1204927435 |   *Every row in the ratings dataframe has a user id associated with at least one movie, a rating and a timestamp showing when they reviewed it. We won't be needing the timestamp column, so let's drop it to save on memory.*  # Drop removes a specified row or column from a dataframe  ratings\_df = ratings\_df.drop('timestamp', 1)  ratings\_df.head()   |  |  |  |  | | --- | --- | --- | --- | |  | userId | movieId | rating | | 0 | 1 | 169 | 2.5 | | 1 | 1 | 2471 | 3.0 | | 2 | 1 | 48516 | 5.0 | | 3 | 2 | 2571 | 3.5 | | 4 | 2 | 109487 | 4.0 |   Collaborative Filtering  *Now, time to start our work on recommendation systems.*  *The first technique we're going to take a look at is called Collaborative Filtering, which is also known as User-User Filtering. As hinted by its alternate name, this technique uses other users to recommend items to the input user. It attempts to find users that have similar preferences and opinions as the input and then recommends items that they have liked to the input. There are several methods of finding similar users (Even some making use of Machine Learning), and the one we will be using here is going to be based on the Pearson Correlation Function.*  *The process for creating a User Based recommendation system is as follows:*   * *Select a user with the movies the user has watched* * *Based on his rating to movies, find the top X neighbours* * *Get the watched movie record of the user for each neighbour.* * *Calculate a similarity score using some formula* * *Recommend the items with the highest score*   *Let's begin by creating an input user to recommend movies to:*  *Notice: To add more movies, simply increase the number (amount) of elements in the userInput. Feel free to add more in! Just be sure to write it in with capital letters and if a movie starts with a "The", like "The Matrix" then write it in like this: 'Matrix, The'.*  userInput = [  {'title':'Breakfast Club, The', 'rating':5},  {'title':'Toy Story', 'rating':3.5},  {'title':'Jumanji', 'rating':2},  {'title':"Pulp Fiction", 'rating':5},  {'title':'Akira', 'rating':4.5}  ]  inputMovies = pd.DataFrame(userInput)  inputMovies   |  |  |  | | --- | --- | --- | |  | title | rating | | 0 | Breakfast Club, The | 5.0 | | 1 | Toy Story | 3.5 | | 2 | Jumanji | 2.0 |   # Add movieId to input user  *With the input complete, let's extract the input movie's ID's from the movies dataframe and add them into it.*  *We can achieve this by first filtering out the rows that contain the input movie's title and then merging this subset with the input dataframe. We also drop unnecessary columns for the input to save memory space.*  # Filtering out the movies by title  inputId = movies\_df[movies\_df['title'].isin(inputMovies['title'].tolist())]  # Then merging it so we can get the movieId. It's implicitly merging it by title.  inputMovies = pd.merge(inputId, inputMovies)  # Dropping information, we won't use from the input dataframe  inputMovies = inputMovies.drop('genres', 1).drop('year', 1)  # Final input dataframe  inputMovies   |  |  |  |  | | --- | --- | --- | --- | |  | movieId | title | rating | | 0 | 1 | Toy Story | 3.5 | | 1 | 2 | Jumanji | 2.0 | | …. | …. | …. | …. | | 4 | 1968 | Breakfast Club, The | 5.0 |   # If a movie you added in above isn't here, then it might not be in the original dataframe or it might spelled differently, please check capitalisation.  The users who has seen the same movies  *Now with the movie ID's in our input, we can now get the subset of users that have watched and reviewed the movies in our input.*  # Filtering out users that have watched movies that the input has watched and storing it  userSubset = ratings\_df[ratings\_df['movieId'].isin(inputMovies['movieId'].tolist())]  userSubset.head()  userId movieId rating  19 4 296 4.0  441 12 1968 3.0  479 13 2 2.0  531 13 1274 5.0  681 14 296 2.0  *We now group up the rows by user ID.*  # Group-by creates sub data-frames where they all have the same value in the column specified as the parameter  userSubsetGroup = userSubset.groupby(['userId'])  *let’s look at one of the users, e.g. the one with userID=1130*  userSubsetGroup.get\_group(1130)  userId movieId rating  104167 1130 1 0.5  104168 1130 2 4.0  104214 1130 296 4.0  104363 1130 1274 4.5  104443 1130 1968 4.5  *Let's also sort these groups so the users that share the most movies in common with the input have higher priority. This provides a richer recommendation since we won't go through every single user.*  # Sorting it so users with movie most in common with the input will have priority  userSubsetGroup = sorted(userSubsetGroup, key=lambda x: len(x[1]), reverse=True)  *Now let’s look at the first user*  userSubsetGroup[0:2]  [(75,  userId movieId rating  7507 75 1 5.0  7508 75 2 3.5  7540 75 296 5.0  7633 75 1274 4.5  7673 75 1968 5.0),  (106,  userId movieId rating  9083 106 1 2.5  9084 106 2 3.0  9115 106 296 3.5  9198 106 1274 3.0  9238 106 1968 3.5),  (106,  userId movieId rating  9083 106 1 2.5  9084 106 2 3.0  9115 106 296 3.5  9198 106 1274 3.0  9238 106 1968 3.5)]  Similarity of users to input user  *Next, we are going to compare all users (not really all !!!) to our specified user and find the one that is most similar.*  *we're going to find out how similar each user is to the input through the Pearson Correlation Coefficient. It is used to measure the strength of a linear association between two variables. The formula for finding this coefficient between sets X and Y with N values can be seen in the image below.*  *Why Pearson Correlation?*  *Pearson correlation is invariant to scaling, i.e. multiplying all elements by a nonzero constant or adding any constant to all elements. For example, if you have two vectors X and Y, then, pearson(X, Y) == pearson(X, 2 \* Y + 3). This is a pretty important property in recommendation systems because for example two users might rate two series of items totally different in terms of absolute rates, but they would be similar users (i.e. with similar ideas) with similar rates in various scales.*  *The values given by the formula vary from r = -1 to r = 1, where 1 forms a direct correlation between the two entities (it means a perfect positive correlation) and -1 forms a perfect negative correlation.*  *In our case, a 1 means that the two users have similar tastes while a -1 means the opposite.*  *We will select a subset of users to iterate through. This limit is imposed because we don't want to waste too much time going through every single user.*  userSubsetGroup = userSubsetGroup[0:100]  *Now, we calculate the Pearson Correlation between input user and subset group, and store it in a dictionary, where the key is the user Id and the value is the coefficient*  # Store the Pearson Correlation in a dictionary, where the key is the user Id and the value is the coefficient  pearsonCorrelationDict = {}  ​  # For every user group in our subset  for name, group in userSubsetGroup:  # Let's start by sorting the input and current user group so the values aren't mixed up later on  group = group.sort\_values(by='movieId')  inputMovies = inputMovies.sort\_values(by='movieId')  #Get the N for the formula  nRatings = len(group)  #Get the review scores for the movies that they both have in common  temp\_df = inputMovies[inputMovies['movieId'].isin(group['movieId'].tolist())]  #And then store them in a temporary buffer variable in a list format to facilitate future calculations  tempRatingList = temp\_df['rating'].tolist()  #Let's also put the current user group reviews in a list format  tempGroupList = group['rating'].tolist()  #Now let's calculate the pearson correlation between two users, so called, x and y  Sxx = sum([i\*\*2 for i in tempRatingList]) - pow(sum(tempRatingList),2)/float(nRatings)  Syy = sum([i\*\*2 for i in tempGroupList]) - pow(sum(tempGroupList),2)/float(nRatings)  Sxy = sum( i\*j for i, j in zip(tempRatingList, tempGroupList)) - sum(tempRatingList)\*sum(tempGroupList)/float(nRatings)  # If the denominator is different than zero, then divide, else, 0 correlation.  if Sxx != 0 and Syy != 0:  pearsonCorrelationDict[name] = Sxy/sqrt(Sxx\*Syy)  else:  pearsonCorrelationDict[name] = 0  ​  pearsonCorrelationDict.items()  dict\_items([(75, 0.8272781516947562), (106, 0.5860090386731182), (686, 0.8320502943378437), (815, 0.5765566601970551), (1040, 0.9434563530497265), (1130, 0.2891574659831201), (1502, 0.8770580193070299), (1599, 0.4385290096535153), (1625, 0.716114874039432),….(18509, 0.1322214713369862)])  pearsonDF = pd.DataFrame.from\_dict(pearsonCorrelationDict, orient='index')  pearsonDF.columns = ['similarityIndex']  pearsonDF['userId'] = pearsonDF.index  pearsonDF.index = range(len(pearsonDF))  pearsonDF.head()  similarityIndex userId  0 0.827278 75  1 0.586009 106  2 0.832050 686  3 0.576557 815  4 0.943456 1040  The top x similar users to input user  *Now let's get the top 50 users that are most similar to the input.*  topUsers=pearsonDF.sort\_values(by='similarityIndex', ascending=False)[0:50]  topUsers.head()  similarityIndex userId  64 0.961678 12325  34 0.961538 6207  55 0.961538 10707  67 0.960769 13053  4 0.943456 1040  *Now, let's start recommending movies to the input user.*  Rating of selected users to all movies  *We're going to do this by taking the weighted average of the ratings of the movies using the Pearson Correlation as the weight. But to do this, we first need to get the movies watched by the users in our pearsonDF from the ratings dataframe and then store their correlation in a new column called \_similarityIndex". This is achieved below by merging of these two tables.*  topUsersRating=topUsers.merge(ratings\_df, left\_on='userId', right\_on='userId', how='inner')  topUsersRating  similarityIndex userId movieId rating  0 0.961678 12325 1 3.5  1 0.961678 12325 2 1.5  2 0.961678 12325 3 3.0  3 0.961678 12325 5 0.5  4 0.961678 12325 6 2.5  ... ... ... ... ...  47235 0.576557 815 146350 2.5  47236 0.576557 815 146656 3.5  47237 0.576557 815 148238 2.5  47238 0.576557 815 148626 3.5  47239 0.576557 815 148652 2.0  47240 rows × 4 columns  *Now all we need to do is simply multiply the movie rating by its weight (The similarity index), then sum up the new ratings and divide it by the sum of the weights.*  *We can easily do this by simply multiplying two columns, then grouping up the dataframe by movieId and then dividing two columns:*  *It shows the idea of all similar users to candidate movies for the input user:*  # Multiplies the similarity by the user's ratings  topUsersRating['weightedRating'] = topUsersRating['similarityIndex']\*topUsersRating['rating']  topUsersRating.head()  similarityIndex userId movieId rating weightedRating  0 0.961678 12325 1 3.5 3.365874  1 0.961678 12325 2 1.5 1.442517  2 0.961678 12325 3 3.0 2.885035  3 0.961678 12325 5 0.5 0.480839  4 0.961678 12325 6 2.5 2.404196  # Applies a sum to the topUsers after grouping it up by userId  tempTopUsersRating = topUsersRating.groupby('movieId').sum()[['similarityIndex','weightedRating']]  tempTopUsersRating.columns = ['sum\_similarityIndex','sum\_weightedRating']  tempTopUsersRating.head()  movieId sum\_similarityIndex sum\_weightedRating  1 38.376281 140.800834  2 38.376281 96.656745  3 10.253981 27.254477  4 0.929294 2.787882  5 11.723262 27.151751  # Creates an empty dataframe  recommendation\_df = pd.DataFrame()  # Now we take the weighted average  recommendation\_df['weighted average recommendation score'] = tempTopUsersRating['sum\_weightedRating']/tempTopUsersRating['sum\_similarityIndex']  recommendation\_df['movieId'] = tempTopUsersRating.index  recommendation\_df.head()  movieId weighted average recommendation score  1 3.668955  2 2.518658  … …  *Now let's sort it and see the top 20 movies that the algorithm recommended!*  recomm..\_df = recomm..\_df.sort\_values(by='weighted average recommendation score', ascending=False)  recommendation\_df.head(10)  movieId weighted average recommendation score movieId  5073 5.0 5073  2284 5.0 2284  3329 5.0 3329  28 5.0 28  945 5.0 945  2848 5.0 2848  987 5.0 987  1024 5.0 1024  55067 5.0 55067  32792 5.0 32792  movies\_df.loc[movies\_df['movieId'].isin(recommendation\_df.head(10)['movieId'].tolist())]  movieId title year  2200 2284 Bandit Queen 1994  3243 3329 Year My Voice Broke, The 1987  3669 3759 Fun and Fancy Free 1947  3679 3769 Thunderbolt and Lightfoot 1974  3685 3775 Make Mine Music 1946  4978 5073 Son's Room, The (Stanza del figlio, La) 2001  6563 6672 War Photographer 2001  6667 6776 Lagaan: Once Upon a Time in India 2001  9064 26801 Dragon Inn (Sun lung moon hak chan) 1992  18106 90531 Shame 2011  Advantages and Disadvantages of Collaborative Filtering  *Advantages*   * Takes other user's ratings into consideration * Doesn't need to study or extract information from the recommended item * Adapts to the user's interests which might change over time   *Disadvantages*   * Approximation function can be slow * There might be a low of amounts of users to approximate * Privacy issues when trying to learn the user's preferences |