I decided to download the iris dataset from the above reference (this is from Final Project – Spark Project). After downloading the data from UCI Irvine Machine learning Repository, I ended up having 4 datasets: bezdekIris.data, Index, iris.data, iris.names. I decided to load “iris.data” into Excel and use the “Convert Text to Columns Wizard Delimited Option” within the “Text to Columns” section in “Data” tab.

I saved the file as a csv file; however I ended up getting missing features. Due to this I finally decided to store the file as an excel file.

This is 60% data will go to the training set, 20% to the validation set, and another 20% will go to the test set. Since we are not using SKLearn to train\_test\_split the data, we have to find another solution to this task.

We are going to random split the data using the “randomSplit()” method from PySpark. Also, seed = 42 is added just to mimic the SKlearn random\_state = 42.

The purpose of the following code is to iterate over each row in the training, validation, and testing sets in order to collect the “class” column.

Let’s calculate and collect the mean and std values for each column within the feature\_columns. The code “xxx” calculates the mean of each column and the code “xxx” calculate the std of each column. We finally use the “xxx” to collect the values and store them in variables.

The above plot shows that Setosa has smaller sepal length but larger sepal width. Versicolor class lies in the middle of the other tw classes in terms of sepal length and width. Virginica has larger sepal legnths but smaller spela widths

The above plot shows that class Setosa has smaller sepal length, but larger sepal width. Versicolor is between the other 2 classes in terms of sepal lengths and widths. Virginica is the class that has larger lengths, but smaller widths.

This plot shows that Setosa has the smaller petal lengths and widths. Again. Versicolor lies in between, but its superior to Setosa in term of petal lengths and widths. Finally, the plot shows that class Virginica has the largest measurements for petal lengths and widths.

Even though the KNN Algorithm can be used for both classification and regression, this task is a multi-classification problem. In this Iris dataset, the column “class” is our categorical data containing 2 different 3 different classes: Setosa, Vericolor, and Virginica.

Our goal here is to try and predict classes.

It is important to state that I decided to switch to Jupyter Notebook because Google Colab wasn't compatible with HBase. Since it is a cloud-based platform, it doesn't have native support for HBase connection. Jupyter Notebook does supports this connection.

- Mac has JAVA 1.17 as the default Java version. Because he HBase nature, I needed to change it to version 1.8 for it to work. (This is done by downloading the java version in java website)

- Download Apache HBase 2.4.17 bin not src. (You do this by going to Apache HBase website)

- I store the HBase 2.4.17 file on my Desktop and proceeded to create two new files: hbase-data (hbase table storage) and zookepeer-data. Go to the "hbase-2.4.17" file and open the hbase-site.xml in order to change configurations to where I want to store my Hbase tables.

Encode() will be used to convert strings to bytes. HBase does not have any notion of *data types*; all row keys, column names and column values are simply treated as raw byte strings (HBase uses bytes).

Tables (class Table) provide the main API to retrieve and manipulate data in HBase. Encode() will be used to convert strings to bytes. HBase does not have any notion of data types; all row keys, column names and column values are simply treated as raw byte strings (HBase uses bytes). Also, HBase tables are made of column families which are the logical and physical grouping of columns. The columns in one family are stored separately from the columns in another family. A column family is a group of columns in a table that are stored as a single key-value pair in the underlying key-value store. Column families reduce the number of keys stored in the key-value store, resulting in improved performance during operations.

We are going to iterate all rows using the df.iterrows() and apply str.encode() to get the strings encoded to bytes. Manually putting all the csv columns in the HBase table using the previous methods mentioned.

This custom function came from previous HomeWorks. It contains spark.read.option() and it’s also design to read the show the top 5 rows just like df.head(). This last step can also be edited to show the numbers of rows you want.

Let’s calculate and collect the mean and std values for each column within the feature*\_columns. The technique we are using here to normalize the data will transform the features values from those coumns to a scale where the mean is 0 and the std is 1. This will help our model to actually learn and get better results. We are doing this because we have different sepal and petal lengths and widths range values. Let’s say we decide to feed the KNN model this data without normalization. The model might give mixed results, because it might take sepal vs petal lengths and favor the petal lengths for being bigger values (KNN might favor features with larger values). We must normalize the data to have a better performance and more accurate results.*

*The code `train\_set.agg({column: ‘mean’}).collect()[0][0]`*calculates the aggregate mean of the passed parameter set (train\_set) and

the code *`train\_set.agg({column: ‘stddev’}).collect()[0][0]`* calculate the aggregate std of the passed parameter set (train\_set).

We finally use the collect()[0][0] to collect the values and store them in variables. The sets: train, validation, and test sets are the result of subtracting the mean of the respective column from each cell, and dividing that with the standard deviation of that column. It is also important to state that we need to normalize the evaluation and testing sets (new unseen data) using the mean and std of the training set.

Retrieve “class” column form the train\_set, validation\_set, and test\_set. Store values in separate lists that will help us make predictions later on.

The function does the following:

* For each row in the validation data set, it calculates the Euclidean distance to every row in the training set.
* Once we have all the distances, select the k minimum distance with each row of the train set. We will sort the Euclidean distance for each validation row in ascending order (taking into account the “class” label for each training row). We will sort using Lambda here.
* Stores tuples of the “class “labels for the k nearest training set neighbors for each validation row.
* Perform list slicing. Select K values (nearest neighbors) for each validation row and adds them to distance list.

Different k values will influence our model’s prediction, reason why we have to test different k values (hyper parameter tuning – Applied Machine Learning).

The most\_common\_class function uses the Python Counter approach. This approach basically returns the count of each element in the list.

The function `predict\_classes` takes the above “neighbors” list and for each row of this list, it will extract the second element (class label) from each row in neighbors list by doing list comprehension. You can view a small DataFrame above which display the first and second element of the neighbors list. We also apply the most\_common\_class function within the predict\_classes function to get the most common “class” label for that row of neighbors. Finally, we store the prediction results in a list called “predictions”.