K-Nearest Neighbors

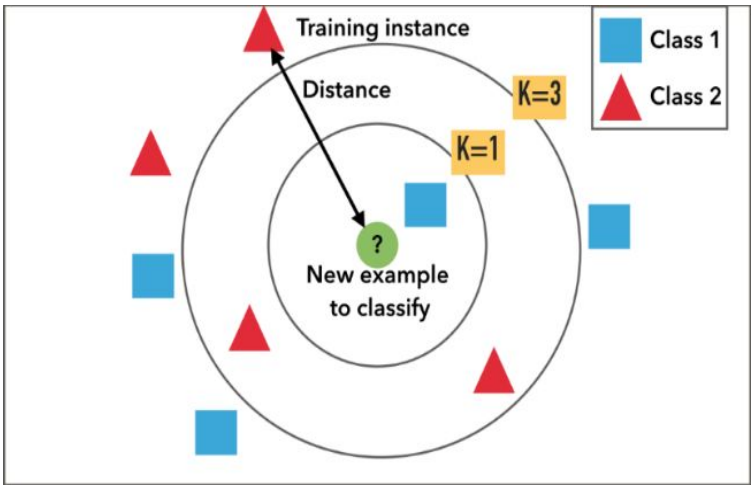
Introduction to K Nearest Neighbors

To understand the k-nearest neighbor algorithm, we first need to understand nearest neighbor. Nearest neighbor algorithm is an algorithm that can be used for regression and classification tasks but is usually used for classification because it is simple and intuitive.

At training time, the nearest neighbor algorithm simply memorizes all values of data for inputs and outputs. During test time when a data point is supplied and a class label is desired, it searches through its memory for any data point that has features which are most similar to the test data point, then it returns the label of the related data point as its prediction. A Nearest neighbor classifier has very quick training time as it is just storing all samples. At test time however, its speed is slower because it needs to search through all stored examples for the closest match. The time spent to receive a classification prediction increases as the dataset increases.

The k-nearest neighbor algorithm is a modification of the nearest neighbor algorithm in which a class label for an input is voted on by the k closest examples to it. That is the predicted label would be the label with the majority vote from the delegates close to it. So a k value of 5 means, get the five most similar examples to an input that is to be classified and choose the class label based on the majority class label of the five examples.

Let us now look at an example image to hone our knowledge:



The new example to be classified is placed in the vector space, when k = 1, the label of the closest example to it is chosen as its label. In this case, the new example is categorized as belonging to class 1. When k = 1, k-nearest neighbor algorithm is reduced to nearest neighbor algorithm.

From the image, when k = 3, we choose the 3 closest examples to the new example using a similarity metric known as the distance measure. We see that two close examples predict the class as being class 2 (red triangle) while the remaining example predicts the class to be class 1 (blue square). The predicted class of the new data point is therefore class 2 because it has the majority vote.

The distance metric used to measure proximity of examples may be L1 or L2 distance. L1 distance is the sum of the absolute of the difference between two points and is given by:

$$d_1(P,q) = \sum_{i=1}^{n} |P_i - q_i|$$

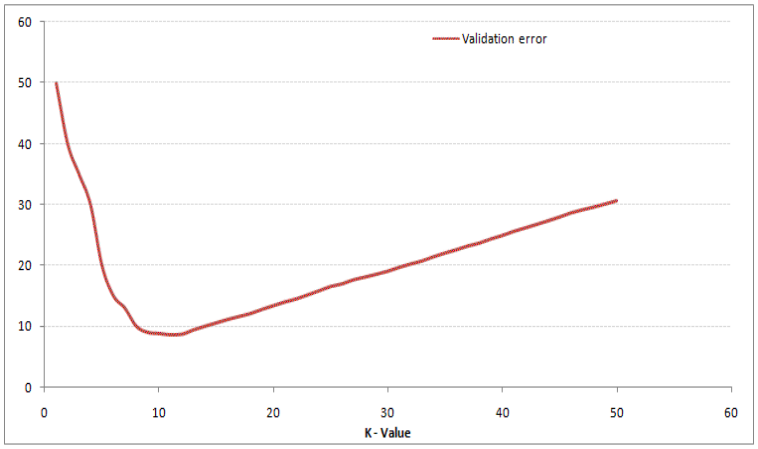
L1 distance is sometimes referred to as the Manhattan distance.

L2 is an alternative distance measure that may be used. It is the Euclidean distance between two points using a straight line. It is given as:

$$d_2(p,q) = \sqrt{\sum_{i=1}^{n} (q_i - P_i)^2}$$

A value of k = 1 would classify all training examples correctly since the most similar example to a point would be itself. This would be a sub-

optimal approach as the classifier would fail to learn anything and would have no power to generalize to data that it was not trained on. A better solution is to choose a value of k in a way that it performs well on the validation set. The validation set is normally used to tune the hyperparameter k. Higher values of k has a smoothing effect on the decision boundaries because outlier classes are swallowed up by the voting pattern of the majority. Increasing the value of k usually leads to greater accuracy initially before the value becomes too large and we reach the point of diminishing returns where accuracy drops and validation error starts rising.



The optimal value for k is the point where the validation error is lowest.

How to create and test the K Nearest Neighbor classifier

We would now apply what we have learnt so far to a binary classification problem. The dataset we would use is the Pima Indian Diabetes Database which is a dataset from the National Institute of Diabetes and Digestive and Kidney Diseases. The main purpose of this dataset is to predict whether a patient has diabetes or not based on diagnostic measurements carried out on patients. The patients in this study were female, of Pima Indian origin and at least 21 years old.

The dataset can be found at:

https://www.kaggle.com/uciml/pima-indians-diabetes-database/data

Since we are dealing with two mutually exclusive classes, a patient either has diabetes or not, this can be modelled as a binary classification task and for the purpose of our example we would use the k-nearest neighbor classifier for classification.

The first step is to import the libraries that we would use.

```
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
```

Next we load the data using Pandas.

```
dataset = pd.read_csv('diabetes.csv')
```

As always what we should do is get a feel of our dataset and the features that are available.

```
dataset.head(5)
```

	Pregnancies	Glucose	BloodPressure	SkinThickness	Insulin	BMI	DiabetesPedigreeFunction	Age	Outcome
0	6	148	72	35	0	33.6	0.627	50	1
1	1	85	66	29	0	26.6	0.351	31	0
2	8	183	64	0	0	23.3	0.672	32	1
3	1	89	66	23	94	28.1	0.167	21	0
4	0	137	40	35	168	43.1	2.288	33	1

We see that we have 8 features and 9 columns with **Outcome** being the binary label that we want to predict.

To know the number of observations in the dataset we run

```
dataset.shape
```

This shows **dataset** contains 768 observations.

Let's now get a summary of the data so that we can have an idea of the distribution of attributes.

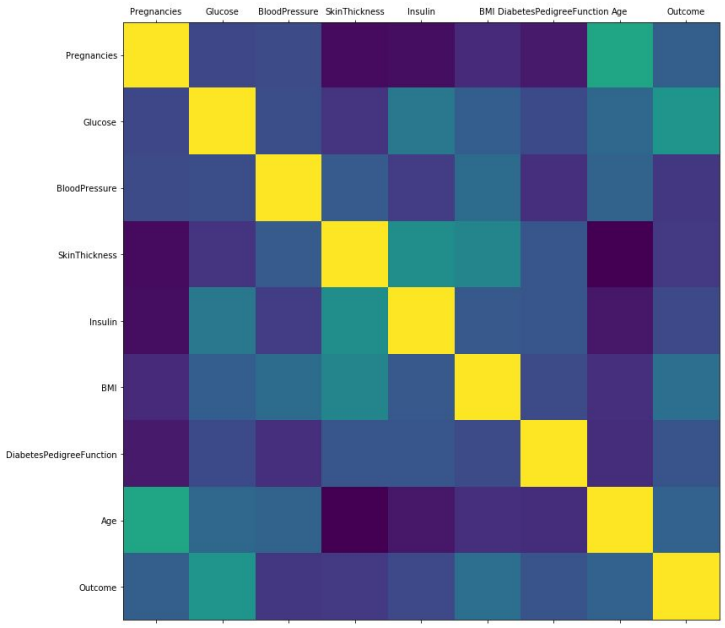
```
dataset.describe()
```

	Pregnancies	Glucose	BloodPressure	SkinThickness	Insulin	BMI	DiabetesPedigreeFunction	Age	Outcome
count	768.000000	768.000000	768.000000	768.000000	768.000000	768.000000	768.000000	768.000000	768.000000
mean	3.845052	120.894531	69.105469	20.536458	79.799479	31.992578	0.471876	33.240885	0.348958
std	3.369578	31.972618	19.355807	15.952218	115.244002	7.884160	0.331329	11.760232	0.476951
min	0.000000	0.000000	0.000000	0.000000	0.000000	0.000000	0.078000	21.000000	0.000000
25%	1.000000	99.000000	62.000000	0.000000	0.000000	27.300000	0.243750	24.000000	0.000000
50%	3.000000	117.000000	72.000000	23.000000	30.500000	32.000000	0.372500	29.000000	0.000000
75%	6.000000	140.250000	80.000000	32.000000	127.250000	36.600000	0.626250	41.000000	1.000000
max	17.000000	199.000000	122.000000	99.000000	846.000000	67.100000	2.420000	81.000000	1.000000

The **count** row shows a constant value of 768.0 across features, it would be remembered that this is the same number of rows in our dataset. It signifies that we do not have any missing values for any features. The quantities **mean, std** gives the mean and standard deviation respectively across attributes in our dataset. The mean is the average value of that feature while the standard deviation measures the variation in the spread of values.

Before going ahead with classification, we check for correlation amongst our features so that we do not have any redundant features

```
corr = dataset.corr()    # data frame correlation function
fig, ax = plt.subplots(figsize=(13, 13))
ax.matshow(corr)   # color code the rectangles by correlation value
plt.xticks(range(len(corr.columns)), corr.columns)  # draw x tick marks
plt.yticks(range(len(corr.columns)), corr.columns)  # draw y tick marks
```



The plot does not indicate any 1 to 1 correlation between features, so all features are informative and provide discriminability.

We need to separate our columns into features and labels

```
features = dataset.drop(['Outcome'], axis=1)
labels = dataset['Outcome']
```

We would once again split our dataset into training set and test set as we want to train our model on the train split, then evaluate its performance on the test split.

```
from sklearn.model_selection import train_test_split
features_train, features_test, labels_train, labels_test = train_test_split(features, labels, test_size=0.25)
```

features_train, features_test contains the attributes while **labels_train, labels_test** are the discrete class labels for the train split and test splits respectively. We use a **test_size** of 0.25 which indicates we want to use 75% of observations for training and reserve the remaining 25% for testing.

The next step is to use the k-nearest neighbor classifier from Scikit-Learn machine learning library.

```
# importing the model
from sklearn.neighbors import KNeighborsClassifier
classifier = KNeighborsClassifier()
```

The above code imports the k-nearest neighbor classifier and instantiates an object from it.

```
classifier.fit(features_train, labels_train)
```

We fit the classifier using the features and labels from the training set. To get predictions from the trained model we use the **predict** method on the **classifier** , passing in features from the test set.

```
pred = classifier.predict(features_test)
```

In order to access the performance of the model we use accuracy as a metric. Scikit-Learn contains a utility to enable us easily compute the accuracy of a trained model. To use it we import **accuracy_score** from **metrics** module.

```
from sklearn.metrics import accuracy_score
accuracy = accuracy_score(labels_test, pred)
print('Accuracy: {}'.format(accuracy))
```

We obtain an accuracy of 0.74, which means the predicted label was the same as the true label for 74% of examples.

Here is the code in full:

```
# import libraries
import numpy as np
```

```
import pandas as pd
import matplotlib.pyplot as plt

# read dataset from csv file
dataset = pd.read_csv('diabetes.csv')

# display first five observations
dataset.head(5)

# get shape of dataset, number of observations, number of features
dataset.shape

# get information on data distribution
dataset.describe()

# plot correlation between features
corr = dataset.corr()    # data frame correlation function
fig, ax = plt.subplots(figsize=(13, 13))
ax.matshow(corr)   # color code the rectangles by correlation value
plt.xticks(range(len(corr.columns)), corr.columns)  # draw x tick marks
plt.yticks(range(len(corr.columns)), corr.columns)  # draw y tick marks

# create features and labels
features = dataset.drop(['Outcome'], axis=1)
labels = dataset['Outcome']

# split dataset into training set and test set
from sklearn.model_selection import train_test_split
features_train, features_test, labels_train, labels_test = train_test_split(features, labels,
test_size=0.25)

# import nearest neighbor classifier
from sklearn.neighbors import KNeighborsClassifier
classifier = KNeighborsClassifier()

# fit data
```

```
classifier.fit(features_train, labels_train)

# get predicted class labels
pred = classifier.predict(features_test)

# get accuracy of model on test set
from sklearn.metrics import accuracy_score
accuracy = accuracy_score(labels_test, pred)
print('Accuracy: {}'.format(accuracy))
```

Another Application

The dataset we would use for this task is the Iris flower classification dataset. The dataset contains 150 examples of 3 classes of species of Iris flowers namely Iris Setosa, Iris Versicolor and Iris Virginica. The dataset can be downloaded from Kaggle (https://www.kaggle.com/saurabh00007/iriscsv/downloads/Iris.csv/1).

The first step of the data science process is to acquire data, which we have done. Next we need to handle the data or preprocess it into a suitable form before passing it off to a machine learning classifier.

To begin let's import all relevant libraries.

```
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
import scipy as sp
```

Next we use Pandas to load the dataset which is contained in a CSV file and print out the first few rows so that we can have a sense of what is contained in the dataset.

```
dataset = pd.read_csv('Iris.csv')
dataset.head(5)
```

	Id	SepalLengthCm	SepalWidthCm	PetalLengthCm	PetalWidthCm	Species
0	1	5.1	3.5	1.4	0.2	Iris-setosa
1	2	4.9	3.0	1.4	0.2	Iris-setosa
2	3	4.7	3.2	1.3	0.2	Iris-setosa
3	4	4.6	3.1	1.5	0.2	Iris-setosa
4	5	5.0	3.6	1.4	0.2	Iris-setosa

As we can see, there are 4 predictors namely sepal length, sepal width, petal length and petal width. Species is the target variable that we are interested in predicting. Since there are 3 classes what we have is a multi-classification problem.

In line with our observations, we separate the columns into features (X) and targets (y).

```
X = dataset.iloc[:, 1:5].values # select features ignoring non-informative column Id
y = dataset.iloc[:, 5].values # Species contains targets for our model
```

Our targets are currently stored as text. We need to transform them into categorical variables. To do this we leverage Scikit-Learn label encoder.

```
from sklearn.preprocessing import LabelEncoder
le = LabelEncoder()
y = le.fit_transform(y) # transform species names into categorical values
```

Next we split our dataset into a training set and a test set so that we can evaluate the performance of our trained model appropriately.

```
from sklearn.model_selection import train_test_split
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size = 0.3)
```

Calculating Similarity

In the last section, we successfully prepared our data and explained the inner workings of the K-NN algorithm at a high level. We would now implement a working version in Python. The most important part of K-NN algorithm is the similarity metric which in this case is a distance measure. There are several distance metrics but we would use Euclidean distance which is the straight line distance between two points in a Euclidean plane. The plane may be 2-dimensional, 3-dimensional etc. Euclidean distance is sometimes referred to as L2 distance. It is given by the formula below.

$$d(\mathbf{p}, \mathbf{q}) = d(\mathbf{q}, \mathbf{p}) = \sqrt{(q_1 - p_1)^2 + (q_2 - p_2)^2 + \cdots + (q_n - p_n)^2}$$

$$= \sqrt{\sum_{i=1}^{n} (q_i - p_i)^2}.$$

The L2 distance is computed from the test sample to every sample in the training set to determine how close they are. We can implement L2 distance in Python using Numpy as shown below.

```
def euclidean_distance(training_set, test_instance):
    # number of samples inside training set
    n_samples = training_set.shape[0]

    # create array for distances
    distances = np.empty(n_samples, dtype=np.float64)

    # euclidean distance calculation
    for i in range(n_samples):
        distances[i] = np.sqrt(np.sum(np.square(test_instance - training_set[i])))
```

```
    return distances
```

Locating Neighbors

Having implemented the similarity metric, we can build out a full fledged class that is capable of identifying nearest neighbors and returning a classification. It should be noted that the K-Nearest Neighbor algorithm has no training phase. It simply stores all data points in memory. It only performs computation during test time when it is calculating distances and returning predictions. Here is an implementation of the K-NN algorithm that utilizes the distance function defined above.

```
class MyKNeighborsClassifier():
    """
    Vanilla implementation of KNN algorithm.
    """

    def __init__(self, n_neighbors=5):
        self.n_neighbors=n_neighbors

    def fit(self, X, y):
        """
        Fit the model using X as array of features and y as array of labels.
        """
        n_samples = X.shape[0]
        # number of neighbors can't be larger then number of samples
        if self.n_neighbors > n_samples:
            raise ValueError("Number of neighbors can't be larger then number of samples in training
set.")

        # X and y need to have the same number of samples
        if X.shape[0] != y.shape[0]:
            raise ValueError("Number of samples in X and y need to be equal.")

        # finding and saving all possible class labels
        self.classes_ = np.unique(y)
```

```
        self.X = X
        self.y = y

    def pred_from_neighbors(self, training_set, labels, test_instance, k):
        distances = euclidean_distance(training_set, test_instance)

        # combining arrays as columns
        distances = sp.c_[distances, labels]
        # sorting array by value of first column
        sorted_distances = distances[distances[:,0].argsort()]
        # selecting labels associeted with k smallest distances
        targets = sorted_distances[0:k,1]

        unique, counts = np.unique(targets, return_counts=True)
        return(unique[np.argmax(counts)])


    def predict(self, X_test):

        # number of predictions to make and number of features inside single sample
        n_predictions, n_features = X_test.shape

        # allocationg space for array of predictions
        predictions = np.empty(n_predictions, dtype=int)

        # loop over all observations
        for i in range(n_predictions):
            # calculation of single prediction
            predictions[i] = self.pred_from_neighbors(self.X, self.y, X_test[i, :], self.n_neighbors)

        return(predictions)
```

The workflow of the class above is that during test time, a test sample (instance) is supplied and the Euclidean distance to every sample in the entire training set is calculated. Depending on the value of nearest

neighbors to consider, the labels of those neighbors participate in a vote to determine the class of the test sample.

Generating Response

In order to generate a response or create a prediction, we first have to initialize our custom classifier. The value of k, cannot exceed the number of samples in our dataset. This is to be expected because we cannot compare with a greater number of neighbors than what we have available in the training set.

```
# instantiate learning model (k = 3)
my_classifier = MyKNeighborsClassifier(n_neighbors=3)
```

Next we can train our model on the data. Remember in K-NN no training actually takes place.

```
# fitting the model
my_classifier.fit(X_train, y_train)
```

Evaluating Accuracy

To evaluate the accuracy of our model, we test its performance on examples which it has not seen such as those contained in the test set.

```
# predicting the test set results
my_y_pred = my_classifier.predict(X_test)
```

We then check the predicted classes against the ground truth labels and use Scikit-Learn accuracy module to calculate the accuracy of our classifier.

```
from sklearn.metrics import confusion_matrix, accuracy_score
accuracy = accuracy_score(y_test, my_y_pred)*100
print('Accuracy: ' + str(round(accuracy, 2)) + ' %.')
```

```
Accuracy: 97.78 %.
```

Our model achieves an accuracy of 97.8% which is impressive for such a simple and elegant model.

The Curse of Dimensionality

The K-Nearest Neighbor algorithm performs optimally when the dimension of the input space is small as in this example. We had four predictors (sepal length, sepal width, petal length, petal width) going into the algorithm. K-NN struggles in high dimensional input spaces like those encountered in images. This is because the similarity measure as expressed by the distance metric is very limited and cannot properly model this high dimensional space. In general, machine learning algorithms try to reduce the number of dimensions so that intuitions we have about low dimensional spaces would still hold true. The accuracy or performance of algorithms usually suffer when the number of dimensions increase. This is known as the curse of dimensionality.