/* Most common used flex styles*/ /* Basic flexbox reverse styles */ /* Flexbox alignment */ /* Non-flexbox positioning helper styles */

The Pima are a group of Native Americans living in Arizona. A genetic predisposition allowed this group to survive normally to a diet poor of carbohydrates for years. In the recent years, because of a sudden shift from traditional agricultural crops to processed foods, together with a decline in physical activity, made them develop the highest prevalence of type 2 diabetes and for this reason they have been subject of many studies.

The dataset includes data from 768 women with 8 features:

Number of times pregnant

Plasma glucose concentration a 2 hours in an oral glucose tolerance test

Diastolic blood pressure (mm Hg)

Triceps skin fold thickness (mm)

2-Hour serum insulin (mu U/ml)

Body mass index (weight in kg/(height in m)^2)

Diabetes pedigree function

Age (years)

The last column of the dataset indicates if the person has been diagnosed with diabetes (1) or not (0)

We will start with importing a few key libraries

```
import numpy as np #linear algebra library of Python
import pandas as pd # build on top of numpy for data analysis, data manipulation
import matplotlib.pyplot as plt #plotting library of Python
```

Now let's mount Google drive so that we can upload the diabetes.csv file. You can find the code in the 'Code snippets' tab of Colab

```
from google.colab import drive
drive.mount('/content/gdrive')
```

First thing that we do is take a look at the shape of the dataframe (df.shape) and take a look at first 5 lines through df.head()

```
df=pd.read_csv('/content/gdrive/My Drive/Colab Notebooks/diabetes.csv') #import
df.head() #shows first 5 lines including column namesdf.shape # number of rows a
```

```
df.shape # provides # rows and # columns of the dataframe df - 768 rows and 9 cc
```

Now we will assess if the dataset has the same proportion of diabetes vs. non-diabetes cases. At the same time we will look if there are missing values. In our dataset we note that woman #2 has a skin thickness of zero and this is not realistic. It leads us to believe that there are a few zero entries that signal that no data was available. This does not apply to columns columns 1 and 9 for obvious reasons.

We use a trick to count the non-zero values of the columns. We convert the data type of the dataframe df to to Boolean using df.astype(bool) coverting all zero values to false=0 and all other entries to true=1 . We subsequently add up all True entries per column.

```
df.astype(bool).sum(axis=0) # counts the number of non-zeros for each column whi
```

The dataframe is unbalanced as we have 268 ones (diabetes) and thus 500 zeros (no diabetes).

The easiest option could be to eliminate all those patients with zero values, but in this way we would eliminate a lot of important data.

Another option is to calculate the median value for a specific column and substitute the zero values for the columns by that median value.

```
median_BMI=df['BMI'].median()
df['BMI']=df['BMI'].replace(to_replace=0, value=median_BMI)

median_BloodPressure=df['BloodPressure'].median()
df['BloodPressure']=df['BloodPressure'].replace(to_replace=0, value=median_Blood

median_Glucose=df['Glucose'].median()
df['Glucose']=df['Glucose'].replace(to_replace=0, value=median_Glucose)

median_SkinThickness=df['SkinThickness'].median()
df['SkinThickness']=df['SkinThickness'].replace(to_replace=0, value=median_SkinT

median_Insulin=df['Insulin'].median()
df['Insulin']=df['Insulin'].replace(to_replace=0, value=median_Insulin)
```

```
df.head() #shows first 5 lines including column names
```

The skin thickness of woman #2 is now 23 (median of that column)

Let's create numpy arrays, one for the features (X) and one for the label (y)

```
X=df.drop('Outcome', 1).values #drop 'Outcome' column but you keep the index col
y=df['Outcome'].values
```

We import the train_test_split function from sklearn to split the arrays or matrices into random train and test subsets>

Parameters:
test_size : in our case 20% (default=0.25)

random_state: is basically used for reproducing your problem the same every time it is run. If you do not use a random_state in train_test_split, every time you make the split you might get a different set of train and test data points and will not help you in debugging in case you get an issue. We used random_state=42 but number does not matter

stratify : array-like or None (default=None) If the number of values belonging to each class are unbalanced, using stratified sampling is a good thing. You are basically asking the model to take the training and test set such that the class proportion is same as of the whole dataset, which is the right thing to do.

```
from sklearn.model_selection import train_test_split #method to split training a
X_train, X_test, y_train, y_test=train_test_split(X, y, test_size=0.2, random_st
```

```
print(X_train) # X_train is an array now!
print(type(X_train))
```

The last preprocessing step is feature normalization transforming the data to have mean=0 and standard deviation=1. It is good practice and since we use distance as the similarity measure in KNN we should not forget this step.

```
from sklearn.preprocessing import StandardScaler
sc=StandardScaler()
X_train=sc.fit_transform(X_train)
X_test=sc.transform(X_test) # we will use exact same transformation for X_test (
```

Now we have finished pre-processing and we are ready to apply the KNN algorithm

```
from sklearn.neighbors import KNeighborsClassifier # we import the K-Nearest Nei
neighbors=np.arange(1,30) #we will try different k - default step size is 1 - re

train_accuracy=np.empty(len(neighbors)) # creates an array that will be used for
test_accuracy=np.empty(len(neighbors))  # creates an array that will be used for
```

```
print(neighbors)
```

A lot of times when dealing with iterators, we also get a need to keep a count of iterations. Python eases the programmers' task by providing a built-in function enumerate() for this task. Enumerate() method adds a counter to an iterable and returns it in a form of enumerate object. This enumerate object can then be used directly in 'for loops' or be converted into a list of tuples using list() method.

```
for i,k in enumerate(neighbors): #k goes from 1 to 19 en i is de counter
  knn=KNeighborsClassifier(n_neighbors=k)
  knn.fit(X_train, y_train)
  train_accuracy[i]=knn.score(X_train, y_train)
  test_accuracy[i]=knn.score(X_test, y_test)
```

```
plt.title('k-NN Varying number of neighbors')
plt.plot(neighbors, test_accuracy, label='Testing Accuracy')
plt.plot(neighbors, train_accuracy, label='Training Accuracy')
plt.legend()
plt.xlabel('Number of neighbors')
plt.ylabel('Accuracy')
plt.show()
```

We get maximum testing accuracy for k=26, so we will setup a KNN classifier with hyperparameter k=26 (we will look at 26 neighbors in order to decide on the label of a test point)

```
knn=KNeighborsClassifier(n_neighbors=26)
knn.fit(X_train, y_train)
knn.score(X_test, y_test) #score method represents accuracy
```

Now we will look at other classification KPIs that we discussed in our lessons: Confusion Matrix, ROC, AUC, F1-Score

```
from sklearn.metrics import confusion_matrix
y_pred=knn.predict(X_test)
confusion_matrix(y_test,y_pred)
```

Classifier not so good: true positives=30, true negatives=90, false positives=10 and false negatives=24. We cannot accept the numerous False Negatives (FNs) in this case as we would tell a women that she is not diabetic whereas she actually is diabetic. One option to reduce the FNs is to change the threshold of the classifier but this will increase the amount of False Positives (FPs) as we have seen in lesson 3. Recall in our case is TP/(TP+FN) = 55% and this is too high. Precission is TP/(TP+FP)=75%

ROC (Receiver Operating Characteristic) curve

It is a plot of Recall vs. False Positive Rate (FPR) for the different possible thresholds of the classifier. It shows the tradeoff between Recall and Precision. The closer the curve follows the left-hand border and then the top border of the ROC space, the more accurate the test. The closer the curve comes to the 45-degree diagonal of the ROC space, the less accurate the test. The area under the curve is a measure of test accuracy.

```
from sklearn.metrics import roc_curve
y_pred_proba=knn.predict_proba(X_test)[:,1]
fpr, tpr, thresholds=roc_curve(y_test, y_pred_proba)
```

```
plt.plot([0,1], [0,1], 'k--')
plt.plot(fpr, tpr, label='knn')
plt.xlabel('fpr')
plt.ylabel('tpr')
plt.title('knn(Neighbors=8) ROC curve')
plt.show()
```

```
from sklearn.metrics import roc_auc_score #area under the ROC curve
roc_auc_score(y_test, y_pred_proba)
```

We build a KNN classifier with 20 blocks of code!