d in many labeled problems.

unknown function of ge or distance from it, and other parameters. It's based on the principle of "information gain"—the algorithm finds out which is most suitable to predict an unknown value.

In this article, we're going to explore key concepts behind the KNN algorithm and analyze a real-world KNN use case.

Contents:

The lazy learning paradigm and KNN algorithm

KNN is widely known as an ML algorithm that doesn't need any training on data. This is much different from eager learning approaches that rely on a training dataset to perform predictions on unseen data. With KNN, you don't need a training phase at all.

KNN relies on observable data similarities and sophisticated distance metrics to generate accurate predictions. This technique may seem a bit counterintuitive and not trustworthy at first, but it's actually very reliable. It's popular in many fields, including:

- *Computer Vision*: KNN performs classification tasks. It handles image data well, and it's considered a fine option for classifying a bunch of diverse images based on similarities.
- *Content Recommendation*: KNN is great for content recommendation. It's used in many recommendation system engines and continues to be relevant even though there are newer, more powerful systems already available.

[…]n the number of features […]cause no one knows which piece of noise will contribute to the model. KNN performs better with low dimensionality (as demonstrated by a study by Gu and Shao in 2014).

KNN inner workings

Surprisingly enough, the KNN algorithm is quite accessible and easy to understand. For an observation that's not in the dataset, the algorithm will simply look for the K number of instances defined as similar based on the closest perimeter to that observation. Any data point falls under a specific group if it's close enough to it.

For **K** neighbors, the algorithm will use their output to calculate the variable y of the observation that we want to predict.

As such:

- If KNN is used for regression tasks, the predictions will be based on the ***mean*** or ***median*** of the K closest observations.
- If KNN is used for classification purposes, the ***mode*** of the closest observations will serve for prediction.

A close look at the structure of KNN

Suppose we have:

- a dataset ***D***,
- a defined distance metric that we'll be using to measure the distance between the set of observations,
- and an integer ***K*** representing the minimum number of near neighbors we should consider to establish proximity.

In order to predict the output ***y*** for a new observation ***X***, will follow these steps:

1. Calculate the total distances between the ***X*** observable and all the data points.
2. Retain the ***K*** observations that constitute the smaller distances to the observable point ***X***.
3. With the ***y*** outputs taken from the ***K*** observations:
   1. apply the mean of the ***y*** deductions if it's a regression problem,
   2. use the mode of ***y*** deductions if it's a classification problem.
4. The final prediction will be the value calculated in step 3.
5. *A detailed version of the algorithm can be found in pseudo-code*:

```
Current ← s
for i ← 2 to n do
    Find the lowest element in row current and unmarked column j containing the
    element.
    Current ← j
    Visited[j] ← true
    Add j to the end of list Path
Add s to the end of list Path
return Path
```

Source: chegg.com

How distances and similarities are carried out in KNN

At its heart, KNN uses different sorts of distance metrics to evaluate the proximity of two data points (their similarity). A core assumption of KNN is:

> ***The closer two given points are to each other, the more related and similar they are.***

Several distance metrics determine correlation and similarity. Even though there are plenty of distance functions to choose from, we should always use the functions that best fit the nature of our data. Notable metrics include:

Distance Metric	Purpose
Euclidean Distance	Mostly used for quantitative data
Taxicab Geometry	Used when the data types are heterogenous
Minkowski distance	Intended for real-valued vector spaces
Jaccard index	Often used in applications when dealing with binarized data
Hamming distance	Typically used with data transmitted over computer networks. And also used with categorical variables.

***Note**: I highly encourage you to look up this article about the effects of distance measure choices when using KNN for classification tasks.*

Most ML libraries offer these metrics out of the box. So, you don't need to code them from scratch, but you might want to do it just to understand how they work.

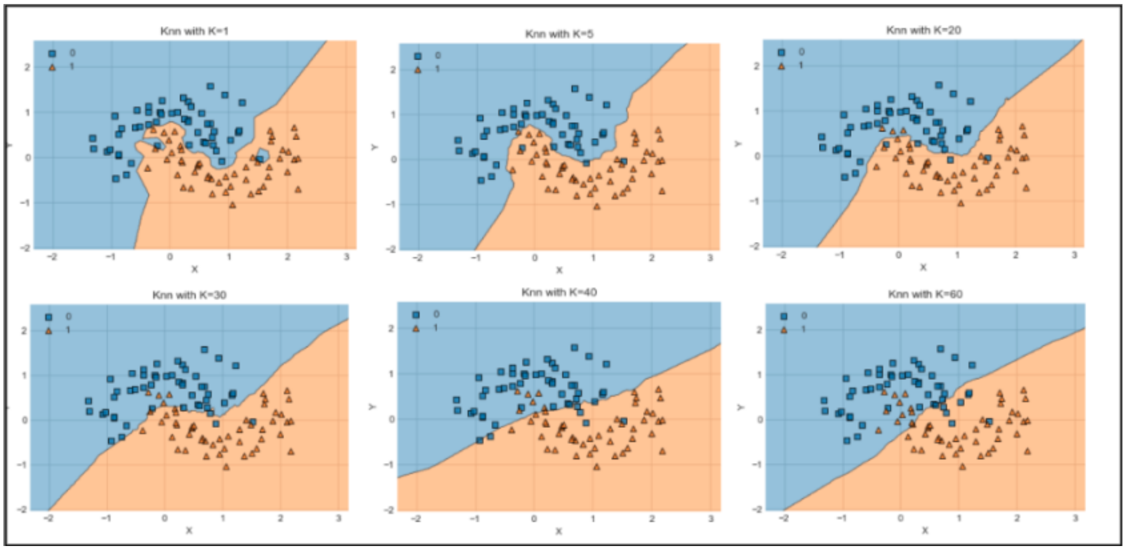
Choose the K value

To select the value of K that fits your data, we run the KNN algorithm multiple times with different K values. We'll use accuracy as the metric for evaluating K performance. If the value of accuracy changes proportionally to the change in K, then it's a good candidate for our K value.

When it comes to choosing the best value for K, we must keep in mind the number of features and sample size per group. The more features and groups in our data set, the larger a selection we need to make in order to find an appropriate value of K.

Also, you shouldn't forget to take into account the effect of the K value on the sample class distribution. If you tend to have many people in one group, then you should increase K. Conversely if your data set often has a significant number of people in one group, you need to decrease K.

Here are some examples of varying the value of K for a specific dataset:



Source: Deepthi A R, [KNN visualization in just 13 lines of code]

As you can see, the more neighbors you use, the more accurate the segmentation. However, as we increase the **K** value until reaching **N** (the total number of data points), we seriously risk overfitting our model, leaving it unable to generalize well on unseen observations.

A practical use case of the KNN algorithm

To illustrate what we've been explaining so far, we'll try to use KNN against a well-known dataset recording the symptoms of breast cancer of clinical patients from Wisconsin in the US.

First, let's download the dataset from UCI Machine Learning Repository. You'll find the data folder with a detailed explanation of each attribute and the target variable we'll try to predict.

Data Set Characteristics:	Multivariate	Number of Instances:	699	Area:	Life
Attribute Characteristics:	Integer	Number of Attributes:	10	Date Donated	1992-07-15
Associated Tasks:	Classification	Missing Values?	Yes	Number of Web Hits:	708362

Source:

Creator:

Dr. WIlliam H. Wolberg (physician)
University of Wisconsin Hospitals
Madison, Wisconsin, USA

Donor:

Olvi Mangasarian (mangasarian '@' cs.wisc.edu)
Received by David W. Aha (aha '@' cs.jhu.edu)

Source: UCI Machine Learning Repository

Set up the project

Download the dataset and install all required packages:

```
pip install scikit-learn
pip install matplotlib
pip install pandas
```

Import the dataset and read it as csv:

```
import pandas as pd

data = pd.read_csv('breast-cancer-wisconsin.data')
data.info()
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 698 entries, 0 to 697
Data columns (total 11 columns):
 #   Column   Non-Null Count  Dtype
---  ------   --------------  -----
 0   1000025  698 non-null    int64
 1   5        698 non-null    int64
 2   1        698 non-null    int64
 3   1.1      698 non-null    int64
 4   1.2      698 non-null    int64
 5   2        698 non-null    int64
 6   1.3      698 non-null    object
 7   3        698 non-null    int64
 8   1.4      698 non-null    int64
 9   1.5      698 non-null    int64
 10  2.1      698 non-null    int64
```

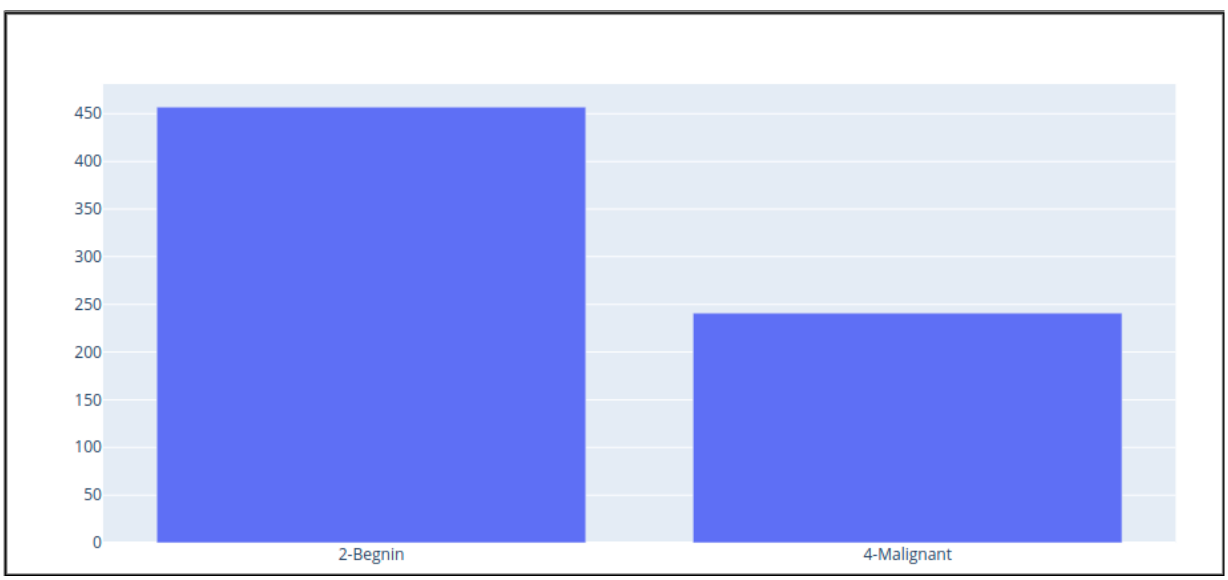
```
.fomrmity
›matin',
```

Visualize the data using the Plotly library

The dataset is clearly unbalanced and unevenly distributed. If we plot the two groups of the target variable, the Benign group records largely more cases than the Malignant one. That can be explained and correlated to the fact that some events are less likely to happen than others.

Here is a plot comparing the balance between Benign and Malignant records:

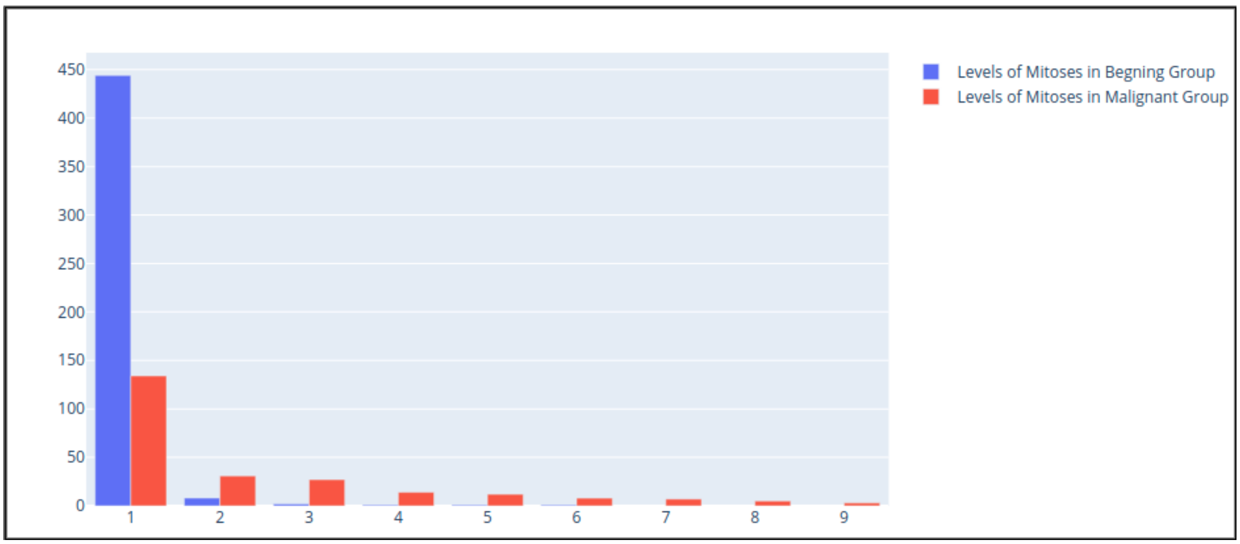
```
import matplotlib.pyplot as plt
import chart_studio.plotly as py
import plotly.graph_objects as go
import plotly.offline as pyoff

target_balance = data['Class'].value_counts().reset_index()
target_balance

target_class = go.Bar(
    name = 'Target Balance',
    x = ['2-Benign, '4-Malignant'],
    y = target_balance['Class']
)

fig = go.Figure(target_class)
pyoff.iplot(fig)
```



Beginning and Malignant Group Classes | Credit: Author

```
()
Mith_10_mal = mal_class_pat['Mitoses'].value_counts().reset_index()
```

```
# Grouping both results:
fig = go.Figure(data=[
    go.Bar(name='Levels of Mitoses in Begnin Group', x=['1', '2', '3', '4', '5',
'6', '7', '8', '9', '10'],
           y=Mith_10_beg['Mitoses']),
    go.Bar(name='Levels of Mitoses in Malignant Group', x=['1', '2', '3', '4',
'5', '6', '7', '8', '9', '10'],
           y=Mith_10_mal['Mitoses']),
])
fig.update_layout(barmode='group')
fig.show()
```



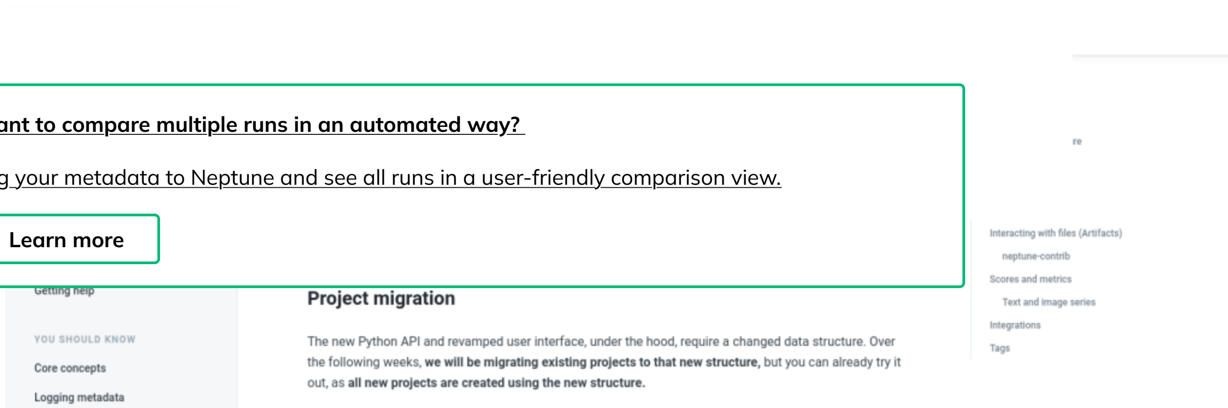
Level of Mitosis in Both clinical Groups | Credit: Author

Initialize your Neptune AI experiment

I usually like to start by creating a virtual environment where I'll be installing and required packages for the project.

```
conda create --name neptune python=3.6
```

- Then, install the Neptune client library with all its dependencies. A newer version is already released and it contains a lot of cool new features and ML integrations. Check it out here: Neptune Doc Home



New Documentation Website, migrate your old projects to the newest API | Source: Neptune Docs

- Install Neptune and its dependencies and enable jupyter integration:

```
pip  install neptune-client
pip install -U neptune-notebooks
jupyter nbextension enable --py neptune-notebooks
```

You could also check the Installation and setup guide on Neptune's official documentation website: Neptune Docs

- Start by creating your project in Neptune 👉 read how.
- Get your API token and connect your notebook with your Neptune session 👉 read how.
- Enable connection with Neptune:

```
import neptune.new as neptune

run = neptune.init(
api_token='YOUR_TOKEN_API',
project='aymane.hachcham/KNN-Thorough-Tour',
)
```

- Start with your experiment. Set up the required parameters we'll be working with:

```
run["Algorithm"] = "KNN"

params = {
    "algorithm": auto,
    "leaf_size": 30,
    "metric": minkowski,
    "metric_params": None,
    "N_jobs": None,
    "N_neighbors": None,
    "P": 2
    "weight": uniform
}
run["parameters"] = params
```

```
features = features.loc[:, features.columns != 'Id']
target = data['Class']


# Splitting the data
x_train, x_test, y_train, y_test = train_test_split(features, target,
test_size=0.2, random_state=123)
```

Training the model

Choosing the best K value

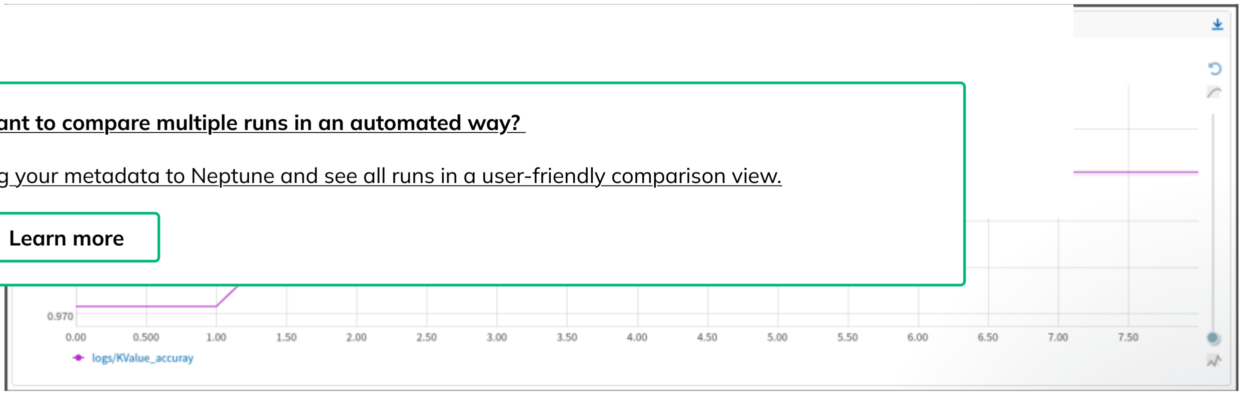
We'll iterate through a range of three different K values and try to see which K will best fit our case and data. First, let's try to understand what exactly does K influences the algorithm. If we see the last example, given that all the 6 training observations remain constant, with a given K value we can make boundaries of each class. Now, that's a nice and useful property of K for the algorithm to use. But, as you may know, the value of K isn't static. The value of K changes with each successive iteration. This means that we'll have a different set of boundary values for each class the second time around.

USEFUL

See how to keep track of your model training in different frameworks.

We'll be logging each K iteration in Neptune using neptune.log_metric().

```
# Logging K values to Neptune:
accuracy_K = []
for k in range(1, 10):
    knn = KNeighborsClassifier(n_neighbors=k)
    knn.fit(x_train, y_train)
    preds = knn.predict(x_test)
    accuracy = metrics.accuracy_score(y_test, y_pred=preds)
    accuracy_K.append(accuracy)
    run['KValue_accuray'].log(accuracy)
```



KNN log value in Neptune | Credit: Author

We observe that the maximum value reached is 0.992 and it appears for K = 6. Other values for K = {2, 4, 5} are 0.98. Since we have more than 3 candidates sharing the same value, we can conclude that the optimal K value is 5.

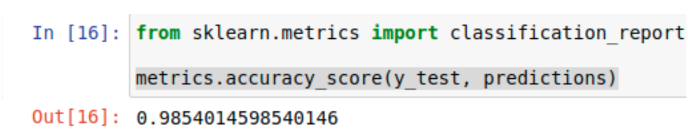
In this particular case, we're using the Minkowski distance for the KNN model. But it could be the case that if you try different distances, you could obtain other K values.

KNN Classifier appears as follow:

```
KNeighborsClassifier(algorithm='auto', leaf_size=30, metric='minkowski',
metric_params=None, n_jobs=None, n_neighbors=5, p=2, weights='uniform')
```

Once we have decided that the best value K is 5, we'll proceed to train the model with the data and check its overall accuracy score.

```
knn = KNeighborsClassifier(n_neighbors=5)
knn.fit(x_train, y_train)
predictions = knn.predict(x_test)


metrics.accuracy_score(y_test, predictions)
```



```
In [16]: from sklearn.metrics import classification_report
         metrics.accuracy_score(y_test, predictions)
Out[16]: 0.9854014598540146
```

Final accuracy score | Credit: Author

KNN limitations

KNN is a fairly simple algorithm to understand. It doesn't rely on any ML model that works inside and makes predictions. KNN is a classification algorithm that only needs to know the number of categories (one or more). This means it can easily determine if a new category should be added without any data on how many other categories there may be.

The downside to this simplicity is that it doesn't make predictions for rare things (like new diseases), where KNN can't predict because it has no idea what the prevalence of a rare thing would be in an otherwise healthy population.

Although KNN produces good accuracy on the testing set, the classifier remains slower and costlier in terms of time and above.

...ncepts about how KNN

KNN is one of the many lazy learning algorithms that don't base predictions on a learning model. KNN makes predictions on the fly (just in time) by averaging the similarity between an input observation and the data already present.

I'll leave you with some useful resources to expand your understanding of KNN even more:

- 9 Distance Measures in Data Science
- Understand the Fundamentals of the K-Nearest Neighbors (KNN) Algorithm
- Introduction to k-Nearest Neighbors: A powerful Machine Learning Algorithm (with implementation in Python & R)
- KNN Classification using Scikit-learn

Thank you for reading!

Aymane Hachcham

"If you don't measure it you can't improve it."

But what should you keep track of?

I have never found myself in a situation where I thought that I had logged too many metrics for my machine learning experiment.

Also, in a real-wor... ...cifications, so logging more metr...

Either way, my suggestion is:

"Log more metrics than you think you need."

Ok, but how do you do that exactly?

Tracking metrics that are a single number

Neptune Integrations

Resources

Pricing

Roadmap

Service Status

Legal

Terms of service

Privacy policy

MLOps

MLOps at a Reasonable Scale

Competitor Comparison

ML Experiment Tracking Tools

Neptune vs Weights & Biases

Neptune vs MLflow

Neptune vs TensorBoard

Other Comparisons

Best MLflow Alternatives

Best TensorBoard Alternatives

Company

About us

Careers We are hiring!