Practical machine learning

By Shrey Mishra
R00183262

● Part -1 (KNN) 40 marks

Theory:

KNN stands for k nearest neighbor, it is one of the simplest machine learning technique that can be used for both classification and regression.

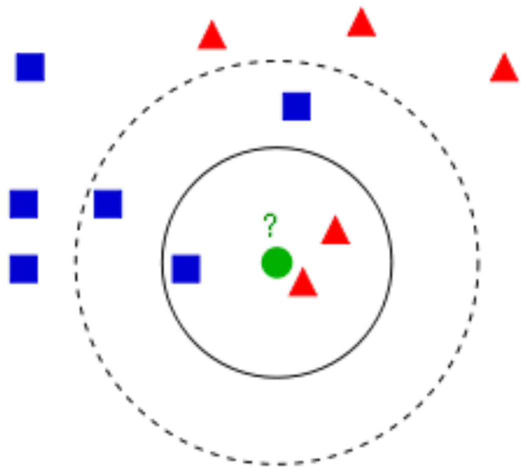
Before I begin explaining the algorithm itself I would like to explain a general prospective of the algorithm and what is trying to capture.

Usually, KNN tries to capture the neighborhood in the sense of distances, many other algorithms try to capture a different essence, for example - Naive Bayes tries to capture probability hence it's called probabilistic model.

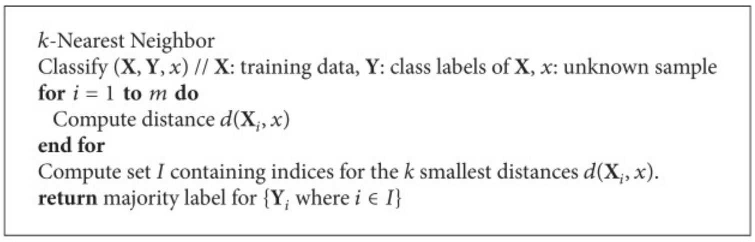
The Pseudo code of simple KNN,

Task - Suppose we are given a query point Q(x1, x2) , I am trying to predict it's label,
Then I will look at the nearest point to it and then say that the label is the nearest point's label
Because we are looking at only one nearest point to decide the label of our query point hence
I can say K=1 (this means how many nearest neighbors I want to look in).

Example 1 : Suppose I have K=3, In image given below then this means I will look 3 nearest neighbors to decide the label of query point.



The above explanation can be better explained with this Pseudo code:



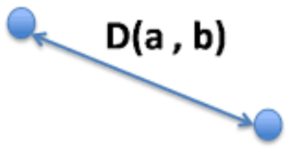
```
k-Nearest Neighbor
Classify (X, Y, x) // X: training data, Y: class labels of X, x: unknown sample
for i = 1 to m do
    Compute distance d(X_i, x)
end for
Compute set I containing indices for the k smallest distances d(X_i, x).
return majority label for {Y_i where i ∈ I}
```

But first we have to look at how to decide which neighbor is near and which is not?

Basically how to calculate distance between two points in the samples?

We need to define distances well, Hence we typically use "D" as Euclidean distance between 2 points.

$$D(a,b) = \sqrt{\sum_{i=1}^{n}(b_i - a_i)^2}$$

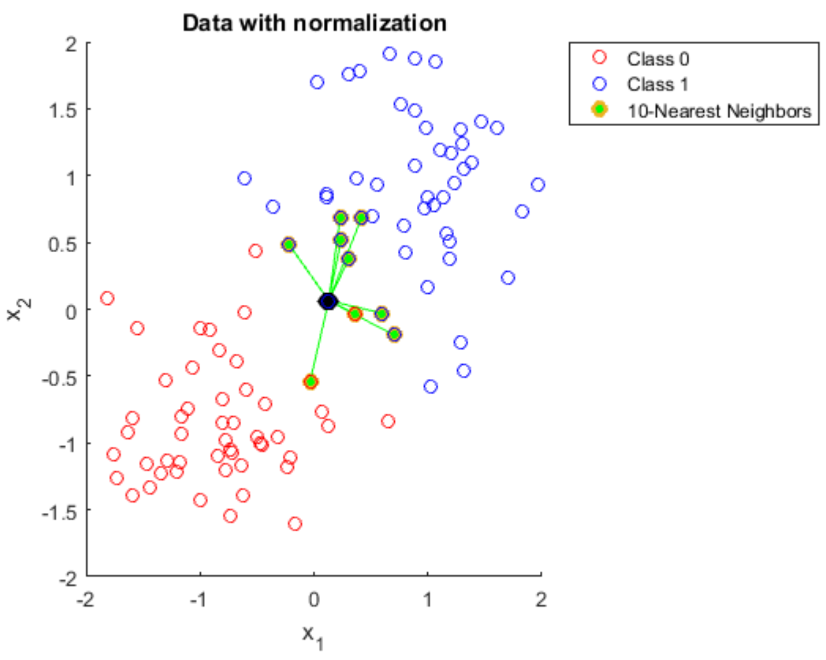


In the image above we can see the query point lie and is taking 3 nearest neighbors based on

the lowest distance value, I will then look at the labels of these 3 neighbors I have received that are closest.

Then I will perform a majority vote that determines the label of the query point.

Example 2 : The below Image is another example where it's calculating distances unto 10 nearest neighbors.



Euclidean is the most commonly used distance metric but not the only distance metric. This falls under a category of metric called Minkowski distance metric.

Where when I substitute p=1 becomes Manhattan and p=2 becomes Euclidean.

Given below is the general Minkowski metric:

$$D(\mathbf{x}_i, \mathbf{x}_j) = \left(\sum_{l=1}^{d} |x_{il} - x_{jl}|^{1/p} \right)^p .$$

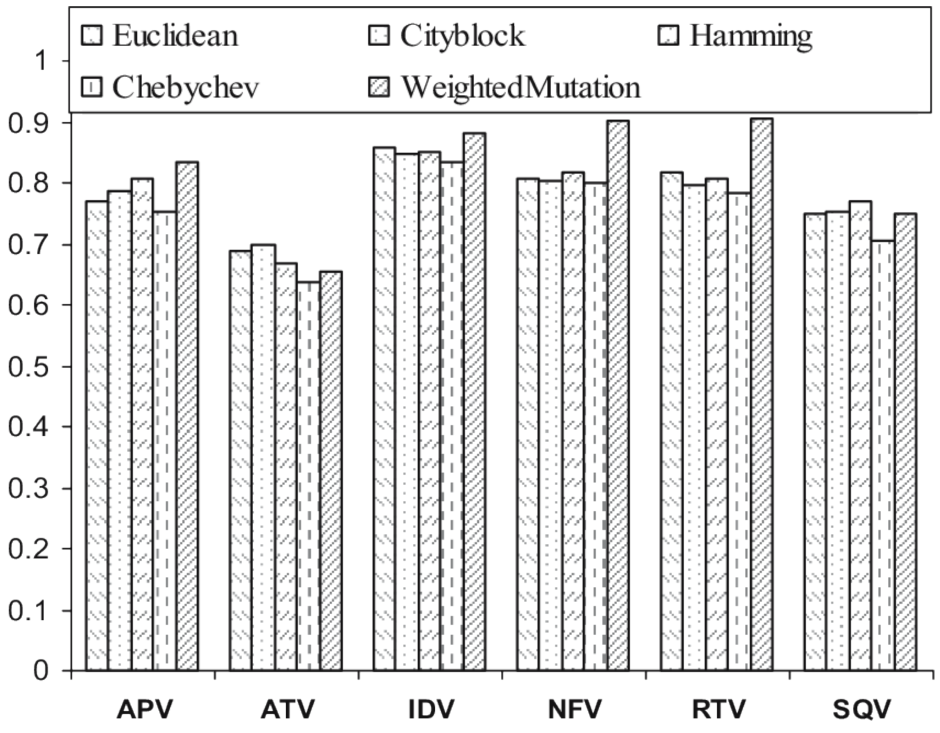
There are some other distances metrics that can be used such as hamming distance, Chebychev, Cosine distance, etc.

But which one to choose still entirely represents on the data we get, a different distance measures tries to capture neighborhood function in a different way.

For example, some researchers found that on SQV dataset the performance of different distance measures does differently on producing the accuracy.

(Source : https://www.researchgate.net/figure/kNN-accuracy-results-using-Euclidean-City-Block-Hamming-and-Chebychev-distances-for_fig4_225329485)

Given below are their results on one particular dataset.



After we have decided the distance metric what we are going to use we will then look for the K value that we can take. So in the first part, I will only focus on how to implement on K=1.

Implementation on code:

- Euc_dist (arr1, arr2) : takes two Numpy array's and calculate the Euclidean distance between them.
- Calculate distances (array2d, array1d): takes a query point array in 1d and takes all the training point to compute the distance of this query point by calling euc_dist.

(The implementation above does not use any for loops and works on the concept of vectorization)

- Vectored code executes faster as compared to for loops execution.

- Accuracy score (arr1, arr2): takes two arrays' and returns all the instances where their values matched at the same index. This is the evaluation metric that measures the performance of our model.

- Now we load the data and for every instance in the test data that I want to make prediction I will call Calculate_distance function which will return me the nearest distance value and the index number of that distance
- After I have received the index number, I will look at the corresponding neighbor's label in train.

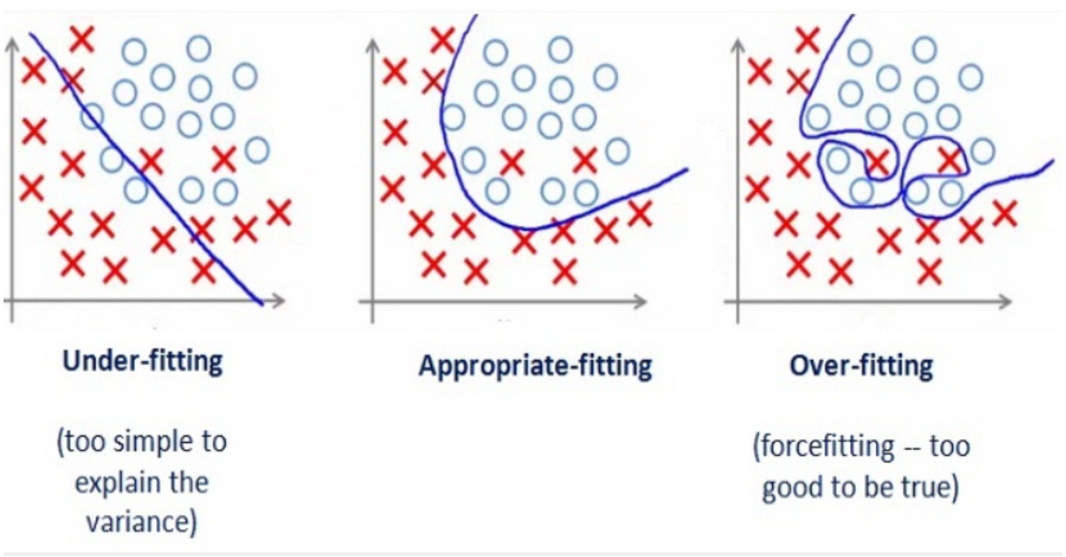
- Since in this analysis we are only considering the first nearest neighbor (K=1), I will conclude this is my label for the query point.

<u>Accuracy of the model:</u> upon running the model I found my accuracy to be 89.5%

(when compared this with scikit's high level library it was same as my result)

Limitations of the model:

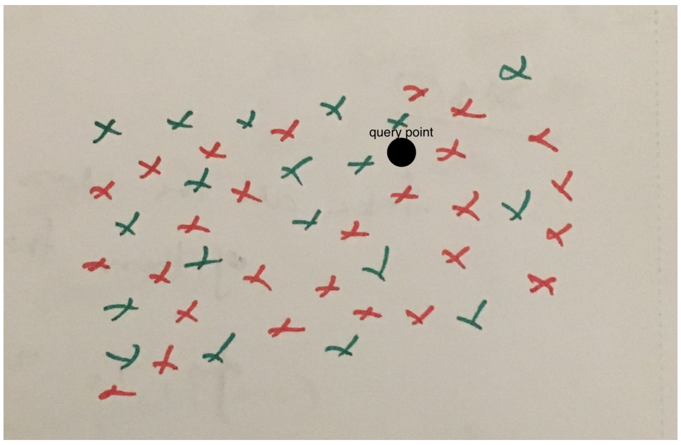
- We are only introducing K=1 this can over fit our data and is very prone to noise (impact of outliers)


Under-fitting
(too simple to explain the variance)
Appropriate-fitting
Over-fitting
(forcefitting -- too good to be true)

imagine if a decision boundary is then for neighborhood, hypothetically the three images then explain the impact of k=too large, K= right value, K=1 (overly sensitive to the data), respectively.

- If the data set is imbalanced and my K value is too high, then my majority vote will always be dominated by the class that has maximum number of points. This is unfair advantage.

- When neighborhood holds no meaning in the data.



<u>Example 1:</u> in the above Image I cannot firmly be sure that the neighborhood of query point has green in it because it is very tightly packed.

Example 2: when the query point is too far from my class labels

Example 3: when neighborhood is too small to predict the label, as discussed in the first point of limitation.

- The space complexity of simple KNN is still a challenge as we need to store all of the x_train in the memory to compute the distance of the query point with all the training points in my data. Hence the space complexity of simple KNN is O(nd) ,where n is the number of rows and d is the number of features.

  Example 1: if my dataset has 364K reviews and 100K features then I would require 36400 Megabytes (roughly) of memory that approximately equals to 36 GB which is insane ?

  It is not at all cheap to buy desktops with 36GB of RAM as of 2019

  Amazon fine food reviews is a real world example that has over 364K reviews on food products. Imagine running it on Tech product section where it's even harder as the review sizes would go up to 1 million. Increasing the RAM is always not the best solution. (see Part 3)

- The time complexity could be a problem where the data is large as at the time of execution I am again calculating the distance of the query point with all the points in my training sample. If the size of training data is too large my training Time will vary vastly, and sometimes in low latency systems such as Sentiment analysis on twitter the user wouldn't wait that much time so the prediction task has to be performed with in milliseconds, imagine training an ML model that works on Finance industry, the decisions have to be taken in micro seconds which can create huge impact on the trading. Hence a simple KNN would fail to deliver me. (See part 3)

● Part -2 (Investigating KNN parameters/ weights) 35 marks

I will first discuss the results in this section and then the methodology,

Results: The maximum accuracy received in weighted KNN is at K=9 which is 92.7%, while at K=10 got an accuracy of 91.9%.

(In Scikit learn's internal implementation the maximum accuracy received in weighted KNN at K=9 as 93.0 % and at K=10 as 92.7 %)

In this Section we have added mainly two important things to our existing model,

1. Our model now accounts many k values and then returns as the right K value that returns the highest accuracy.

2. Our model now takes a weighted majority vote instead of a simple majority vote where every neighbour in the neighborhood has weight assigned related to the distance from the query.

   Example : If I have three neighbors in the neighbour hood the value now assigned is now dependent upon the weight of the neighbors. The father points from the query point now will have less role in determining the label whereas the near points will play major role.

How to pick the right weight ?

Given a query instance xq,

$$f(\mathbf{x}_q) := \frac{\sum_{i=1}^{k} w_i f(\mathbf{x}_i)}{\sum_{i=1}^{k} w_i}$$

<u>Where</u>

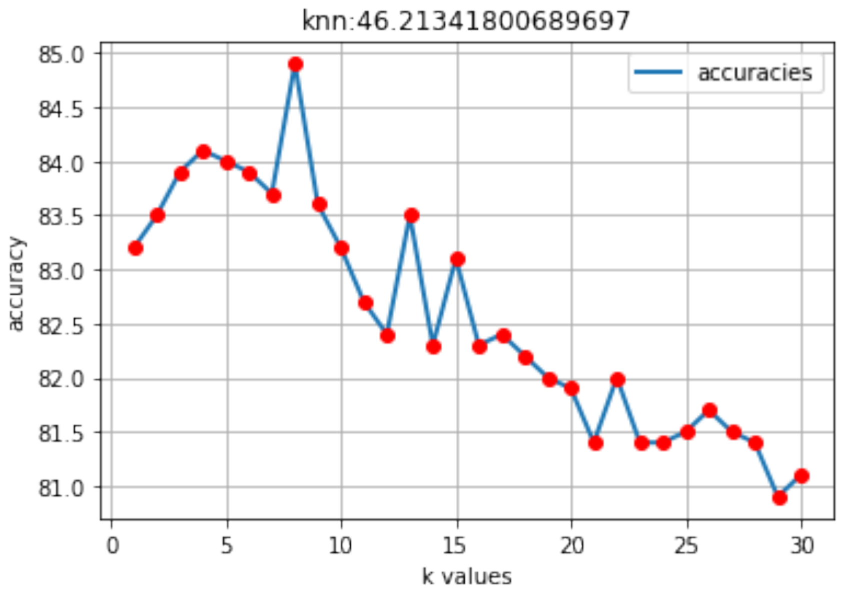
$$w_i = \frac{1}{d(\mathbf{x}_q, \mathbf{x}_i)^2}$$

Given above is the metric from class slides that is used to calculate the weights if you notice the denominator is raised to a power of 2 (call it n for general term), this power defines how much we want to penalize our farther features, the more I increase the n value the more are my farther features getting penalized.

The above results produced at K=10 were made with n=1, meaning weight is equal to the inverse of the distance.

Let's try to see what is the best values we can get with only <u>varying K</u> in the simple model as discussed in Part 1.

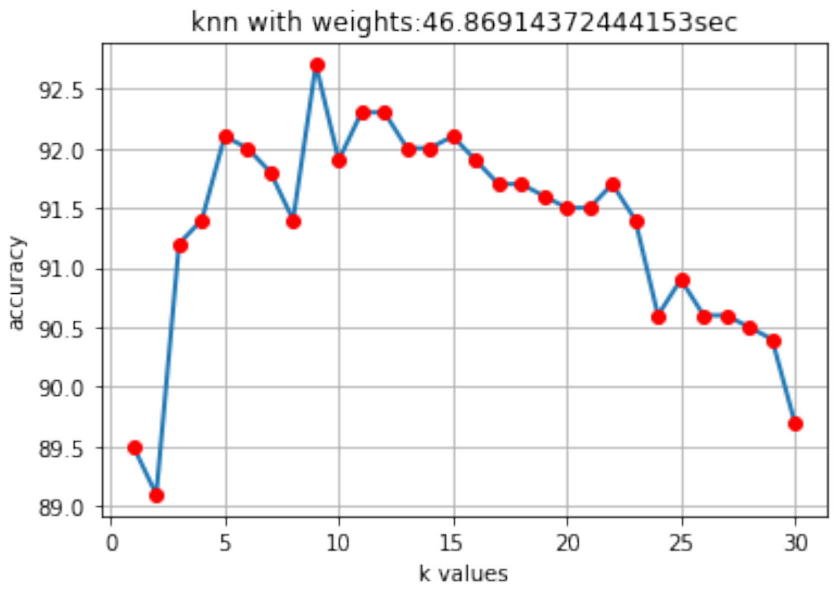
The title also mentions the execution time of the simple KNN.



```
the best k value is:
8, and the accuracy
is 84.89999999999999
```

```
Now let's see the results on weighted KNN,
```

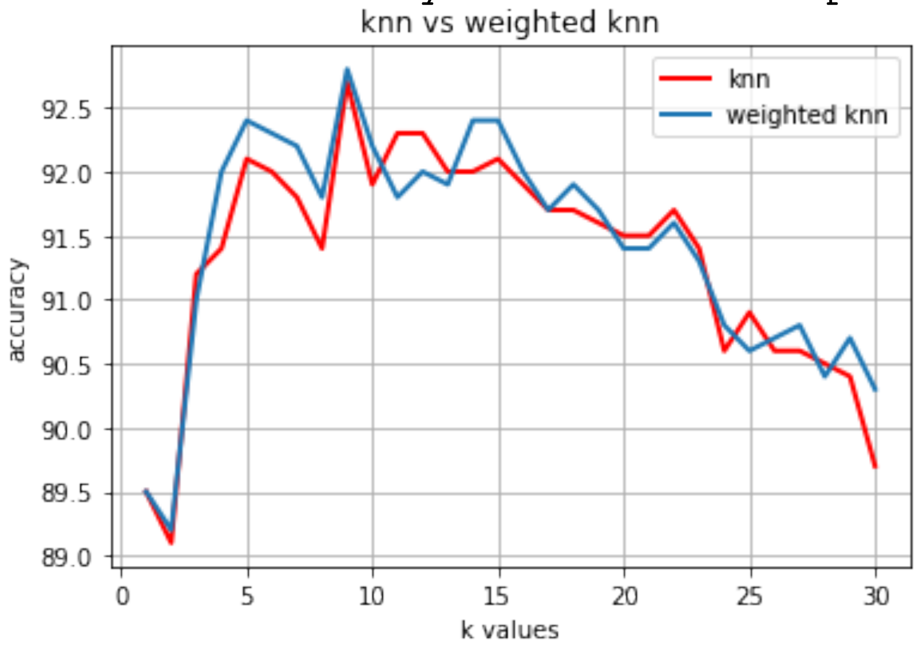
In the graph plotted below we can see the accuracy of KNN with weight is much better.



the best k value is: 9, and the accuracy is 92.7

Clearly with a slight increase in execution time we get much better increase in the accuracy.

Let's side by side compare the two before we conclude anything.



As we can see in most of the instances weighted KNN performs much better than simple KNN, but not necessarily always, because we can see some of the instances where the red line dominates.

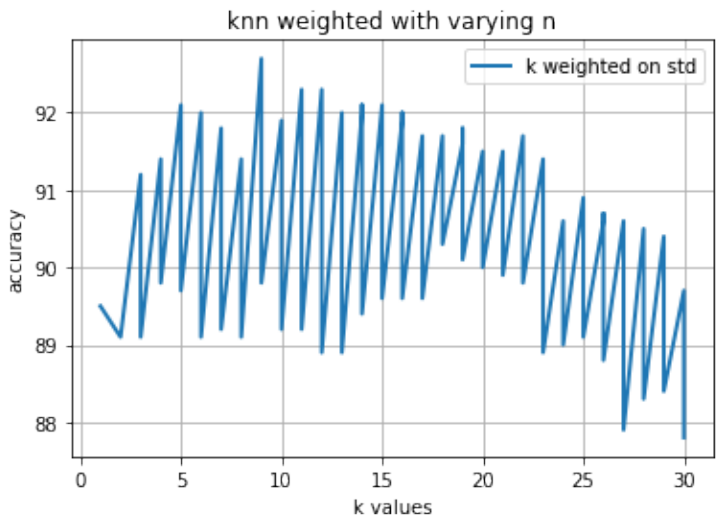
<u>Conclusion 1:</u> KNN with weights is generally better than simple KNN in *<u>most</u>* of the cases

Let's try to vary the power in the denominator of weight (n) and check if penalizing farther features from the query can help us get better accuracy.

This graph shows us when I try to penalize the features even more then my accuracy drops it only increases when I change the k value and then change n back to 1.

The graph plots K=1 n=1, K=1 n=2,
K=3 n=3,…………..and so on

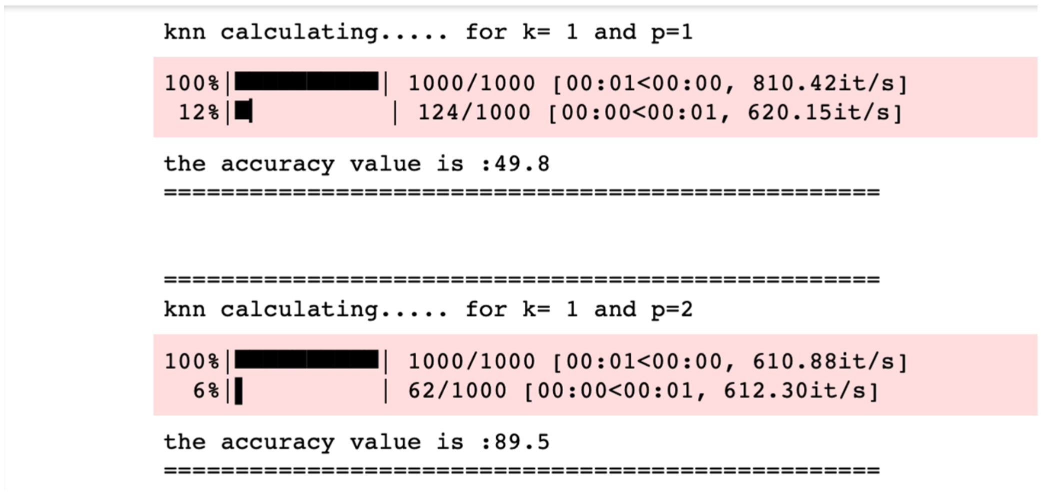
I have demonstrated the results for up till n=5 for every k values till 30



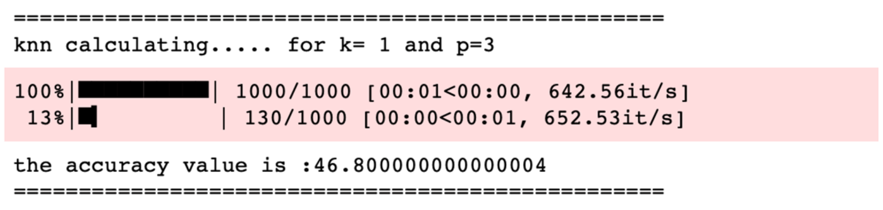
<u>Conclusion 2:</u> for every K value increasing the n more than 1 resulted in a drop in accuracy.

How about we change the Minkowski metric discussed in part 1 by varying the p value

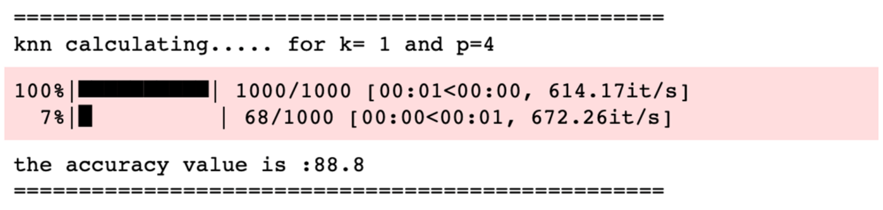
What is the impact when I use Manhattan distance or some other distance variation of Minkowski with a different p value.



```
knn calculating..... for k= 1 and p=1

100%|██████████| 1000/1000 [00:01<00:00, 810.42it/s]
 12%|█▏         | 124/1000 [00:00<00:01, 620.15it/s]

the accuracy value is :49.8
==================================================


==================================================
knn calculating..... for k= 1 and p=2

100%|██████████| 1000/1000 [00:01<00:00, 610.88it/s]
  6%|▌          | 62/1000 [00:00<00:01, 612.30it/s]

the accuracy value is :89.5
==================================================
```



```
==================================================
knn calculating..... for k= 1 and p=3

100%|██████████| 1000/1000 [00:01<00:00, 642.56it/s]
 13%|█▎         | 130/1000 [00:00<00:01, 652.53it/s]

the accuracy value is :46.800000000000004
==================================================
```

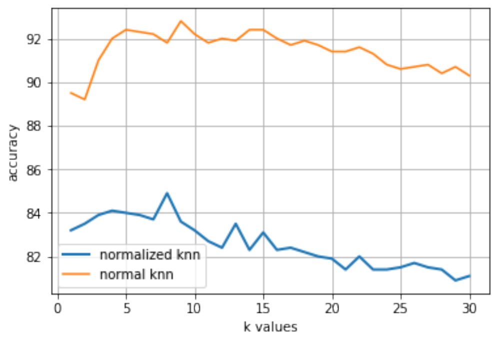


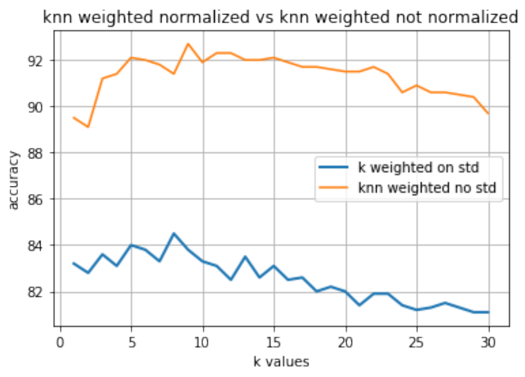
```
==================================================
knn calculating..... for k= 1 and p=4

100%|██████████| 1000/1000 [00:01<00:00, 614.17it/s]
  7%|▋          | 68/1000 [00:00<00:01, 672.26it/s]

the accuracy value is :88.8
==================================================
```

In the pictures above from the console output proves that increasing the p or decreasing it only worsens the accuracy score.

<u>Conclusion 3:</u> for every K value increasing/decreasing the p more than 1 resulted in a drop in accuracy.

Sometimes normalizing the data can help in achieving better accuracies , lets try to compare normalized training vs not normalized training.(will be discussed later in the part 3)





The above graph compares the effect of normalization and we can clearly see that the accuracy drops on normalized features.

(usually normalization should help me get better accuracy but in this example we prove that it did not)

Upon further research I found the reason why normalization didn't help us

<u>Conclusion 4:</u> normalization not always helps as some times it destroys diversity in the data and lower numerical stability

(Source - https://www.researchgate.net/post/why_I_do_not_get_good_results_with_normalized_data)

Before I finish this Part I have a question of a baseline before we compare our model we should always have a baseline to improve, in our case what could be the simplest model that works without doing any machine learning.

It would be the model that output always the majority class, such a naive model can perform very well on imbalanced data hence it is accuracy is never a good measure for performance of model.

In our case a baseline model that always predicts class two which is the majority class having 1342 instances will have an accuracy of 33.55%.

Despite of the fact that we have a pretty balanced dataset of class labels occurrences as:

```
{0.0: 1317, 1.0: 1341, 2.0: 1342}
```

Conclusion 5: Accuracy itself is not a good measure. We can have better evaluation metrics.

{Note: in this section I will not explain my code as it will be too lengthy for a pdf to be read,
I have written the comments and doc-strings inside the function making a crisp understanding of the concept also I have compared my results with Scikit learn’s internal implementation and have mentioned it in every section}

● Part -3 (KNN for regression) 25 marks

For the evaluation of this part we will not use accuracy, because we don’t have categorized labels Like in Part 1 and Part 2 Instead we will use R2 metric

R2 metric is defined as :

$$R^2 = 1 - \frac{sum\ of\ squared\ residuals}{total\ sum\ of\ squares}$$

Where

$$sum\ of\ squared\ residuals = \sum_{i=0}^{m} (f(x^i) - y^i)^2$$

$$total\ sum\ of\ squares = \sum_{i=0}^{m} (\bar{y} - y^i)^2$$

Sum of the squared residual (SSresidual) = the mean error in the deviation from the predicted and the actual values

Sum of the square (SStotal) = mean error deviation

How to interpret this score?

Case 1: an r2 score with value 0 means that my model is as good as the model that is predicting the mean of all the y_true all the time.

Case 2: an r2 score is negative when my model is performing worse than the simple mean based model

Case 3:an r2 score is maximum 1 when I have my all the labels exactly matched ssresidual will be 0 and then the ratio term becomes 0 so the r2 metric will return 1

Case 4: an r2 score between 0 and 1 when my model better than the simple mean based model but still less than a perfect model.

Here the benchmark is a simple model that always tries to predict the mean of y_train, we want to do better than that model.

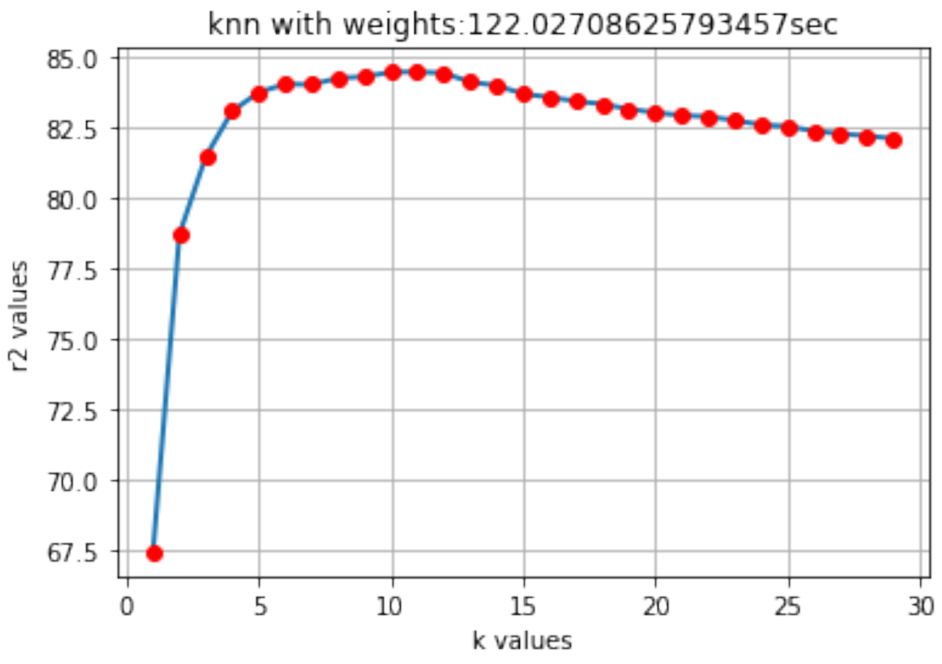
A regression label is decided by the weighted mean of the neighbors with respect to the feature values so the regression label expression becomes, equivalent to

$$f(\mathbf{x}_q) := \frac{\sum_{i=1}^{k} w_i f(\mathbf{x}_i)}{\sum_{i=1}^{k} w_i}$$

Results: maximum r2 score is 84.45761460645686%
At k value :11, and the r2 score at k=10 is 84.44423491032562 %

(In Scikit learn's internal implementation the maximum accuracy received in weighted KNN at K=10 is 84.84% and at K=11 is 84.7756771821375 %)

The below graph plots the my r2 score for KNN regression.



<u>The problem with Euclidean distance:</u> the problem with the Euclidean distance is that the features with heavy range can dominate.

For example: if I have a data points (1,790), (2,890) and the feature 1 lies between the range of 1-3 while feature 2 lies in the range of 100-1000 then the variation is 33% more in the feature 1 while variation in feature 2 is 10% more.

So technically feature 1 should impact the distance more than feature 2, but this might not be true,

Now what will happen is my second feature will dominate over the distance metric as

Which will now be – sort $((1)^2 + 10^2)$ so in order to give features equal weightage and not based on their size or range, In the above example the distance of the two points is not as large with what we will get when we standardize or normalize meaning these two points are very close if we give equal feature importance. Directly using Euclidean distance without doing normalization will give more importance to the feature that has a bigger value.

1. We can introduce normalization and standardization but as seen in Part 2 this practice may not always be helpful in improving accuracy but usually a good practice if we are using Euclidean distance as a measure. With a mean of 0 and std of 1 we can relocate all of our data points to this new scale.

2. There are certain distance measure that no matter how much we try to improve will never give an appropriate result because they are failing to capture the neighborhood idea in the data.

   So we could try to base our results based on completely different distance metric such as hamming or cosine distance, trying a wide variety of distance can help us get better accuracy.

Cos-distance (x1, x2) = 1- Cos-similarity(x1, x2)

Where cos similarity is the measure of Cos angle between the two vectors.

Cos similarity is related to Euclidean if x1 and x2 are unit vector,

Euc-dist (x1,x2)^2= 2* Cos distance

3. When the dimensionality is huge then these distances are very hard to interpret hence a feature selection can be applied that removes feature that do not necessarily contribute.

   This will not only help in easy visualization but also help us in getting faster results.

Are there better versions of KNN available?

A. KD tree can be extremely helpful as they are based on the concept of Binary search trees where the intersection of axis parallel lines is considered. I will not go in detail as why KD tree works but I will try to answer why it didn't for our case which is even more interesting

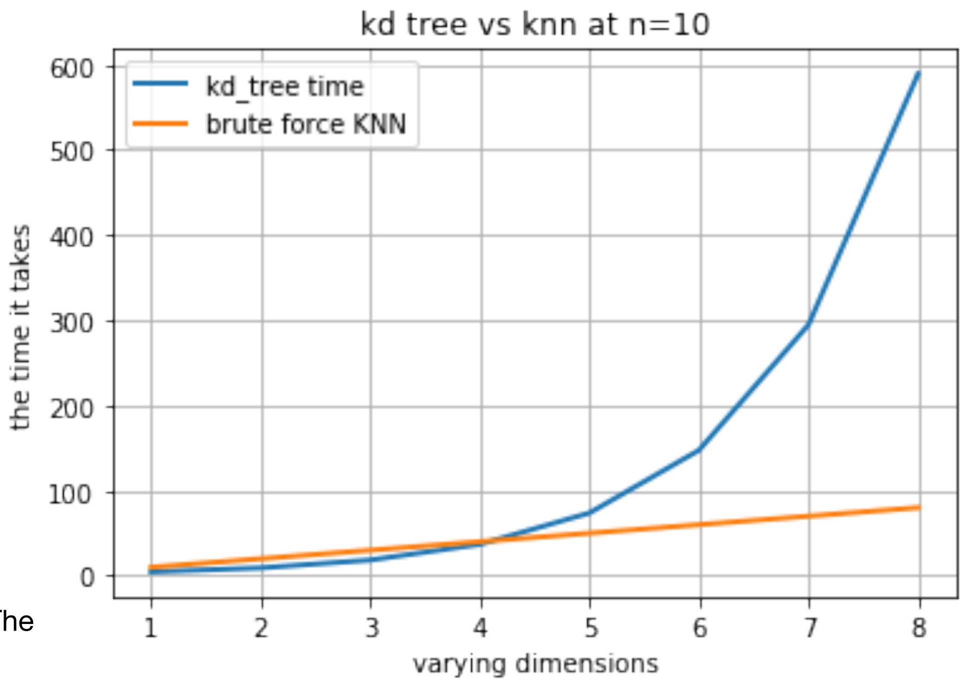
KD tree relies on intersection of planes, and the concept of Binary search tree where the order of finding the node has a time complexity of O(log n) instead of O(n) in linear search.

KD tree exploits the same concept,

In KNN we took a query point and did comparisons with it in the neighborhood and then took the majority vote. In K-D tree we only make log(n) comparisons but there is a catch this is only for one axis region and thus If I account my time complexity with all the axis combined with the total comparisons that we have to make it is O(2^d * log n)

So when d is small say 2,3,4 or low dimensions it is still better than brute force approach of KNN which has time complexity of O(nd)

But as dimensions increases the time effect is clearly visible, first I will prove this mathematically and then in scikit's learn implementation as in our x_train in part 3 has 12 dimensions which is large for a kd tree.

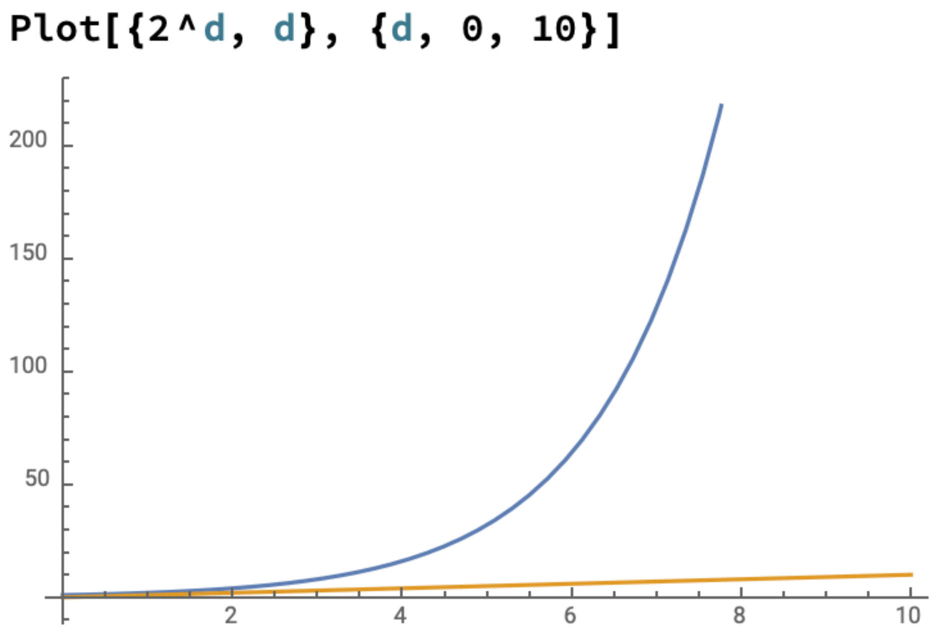


The plot that I have

used to show only accounts d=8 in our data we have 12 dimensions which clearly prove that time can be even worse than normal KNN.

I have plotted this graph using marplotlib code will be left in the python notebook. In the code kd-tree took 5.77 seconds while brute force took 4.58 sec

A much cleaner graph obtained from wolfram alpha



```
Plot[{2^d, d}, {d, 0, 10}]
```

Conclusion:

KD tree is a better solution but only if my dimensionality in data is low. It can take even worse time than brute force approach for large number of features

To be honest KD tree wasn't even made for machine learning, it was developed for computer graphics where the dimensions are typically restricted within 2d and 3d.

Another drawback of KD tree is that it only works on uniformly distributed numbers, I will not go to prove this.

Conclusion: for low dimensions we can have a model that runs kd-tree and when the dimensions increase we can try the basic KNN to save time.

There are other versions of KNN such as ball_tree, I will not go in much detail about them, as they are variations of basic KD tree.

Perfect we saved us some time of execution for low dimensions, can we do even better than this ?

This idea is called **LSH (locality sensitive Hashing)** in which we exploit the python dictionary data structure where searching an element has time complexity in O(1) which is even better than my KD tree's implementation where searching a node can take order of log n.

In this approach we save all the neighbors in a dictionary

In this section I will highlight how LSH works (Pseudo code), unfortunately this algorithm is not implemented in Scikit-Learn learn and couldn't find any other packages that had it working.

Pseudo code:

1. We calculate the hash value for each point.

2. We save the hash value as key in the dictionary and value as the query point

3. If we need to look for the nearest neighbour we can look it across the keys in dictionary and this will save time on execution

Before we conclude anything, wait!!

In order to make such dictionary, I need to construct it, suppose I also want to incorporate the m planes in which I will be looking up my query point then

O (m d +n'd)   where m= number of hyper planes , d= dimensions n'=elements in the bucket size of my dictionary.

typically m is approximately equal to log(n)

This will result in a time Complexity of O(d*logn)

But what if my query point falls into the other set of hyper plane?

Solution > I can make another dictionary of another set of m hyper planes and I can make n such dictionaries.

This means we will have n such dictionaries or hash tables and then we can take a set union of all the values present in all the dictionary at a given key, and then these are all my K neighbors.

But still there is a catch, if two points are placed together on the same plane their projections will lie on the same plane.

Hence LSH is not perfect, it depends upon the m number of hyper planes constructed, It is also a probabilistic model, randomized algorithm.

This idea is used extensively in Computer Vision.

<u>Conclusion:</u> The execution time of LSH is significantly very low in just O (1) we can look our element in the dictionary and return its nearest neighbors. For M such dictionaries I will have O(M) hence if LSH is implemented, all I need is all the look up tables and I can immediately tell the nearest neighbors and this is the solution to the problem discussed in limitations of the first part where I have low latency systems and want to implement KNN. This also comes with a drawback to construct the dictionary but run time execution is really fast.

What could it be if we could use any dimensionality reduction algorithm and then feed the data to a KD tree?

Tried: but the accuracy was even lower than the base line mean predicting model.