**DESCRIPTION**

K Nearest Neighbors is an effective and simple to use machine learning algorithm that provides surprising accurate results given the simplicity of the model. The KNN model is plagued by the curse of dimensionality. Given that it is a search algorithm, the computation time is dependent on the number of points and the dimension of the feature vectors as it compares distance to it’s neighbors. Therefore, it follows that approaches to speed up the algorithm should attempt to reduce the dimensionality and the number of points while minimizing the loss in accuracy. One family of methods involve generating more informative basis vectors that allow for dimensionality compression, examples include PCA and LDA. Another approach is to tackle the number of samples used to build the search tree by using prototype points. These prototypes can either be a subset of the training data set “M” or well selected proxies for the data in the training set. The prototype training points should outperform a randomly selected subset of the data.

For the purposes of this assignment, I will investigate utilizing the K-means clustering approach to prototype selection. K-means clustering is an unsupervised learning method that groups datapoints that share spatial proximity. The algorithm works by initializing a set of K random clusters and computes the distances from the data points to all the clusters, assigning the datapoints to the nearest cluster. The cluster center location is then updated to be the mean of the points assigned to the cluster and then the algorithm repeats the loop of distance calculation and center update. The algorithm terminates once it has converged or some other termination criterion has been met. This method is effective in selecting prototype points due to the fact that it summarizes the spatial distribution of our data points in space through the K cluster centers. Therefore, if the points of a specific class are distributed around a large continuous hypercube, this algorithm will place cluster locations more or less evenly distributed throughout that large sub domain. Alternatively is there are disconnected areas in the Rd then the algorithm can place cluster centers in the centers of the disconnected regions.

**ALGORITHM:**

INPUTS: Training Set X, Labels Y, Number of prototype points M, C is our set of classes

OUTPUTS: Prototype Points X\_p, and their labels Y\_p

Assumptions: Number of training points per class are equal, algorithm can be modified if this assumption is invalidated

PROCEDURE:

# PCA Dimensionality Reduction

Apply PCA to your training set X

Identify eigenvectors that retain 99% of the variance of the data by comparing cumulative sum of eigenvalues

Project Training Set X onto new basis using the chosen eigenvectors

Training Points Per Class T\_PC = (# of prototype points M)/(# of classes C)

For each class in C:

Run K-Means clustering with T\_PC clusters centers on subdomain of X Omega:X->class

Assign Clusters centers as Prototype Points X\_p\_i with class in Y\_pi

Return X\_P, Y\_P

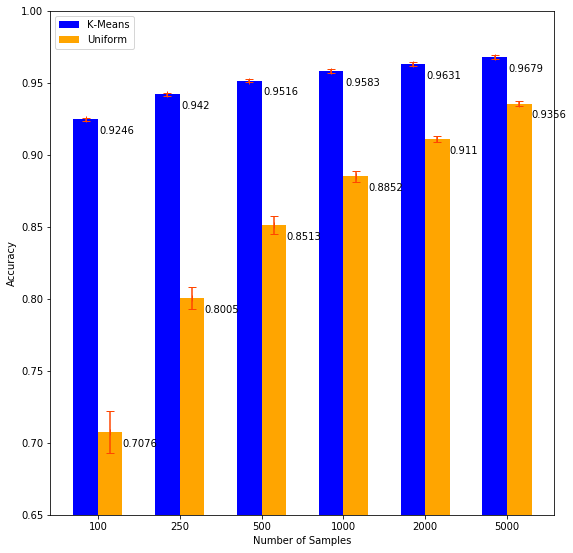
Run Nearest Neighbor Algorithm using X\_P, Y\_P as the training set

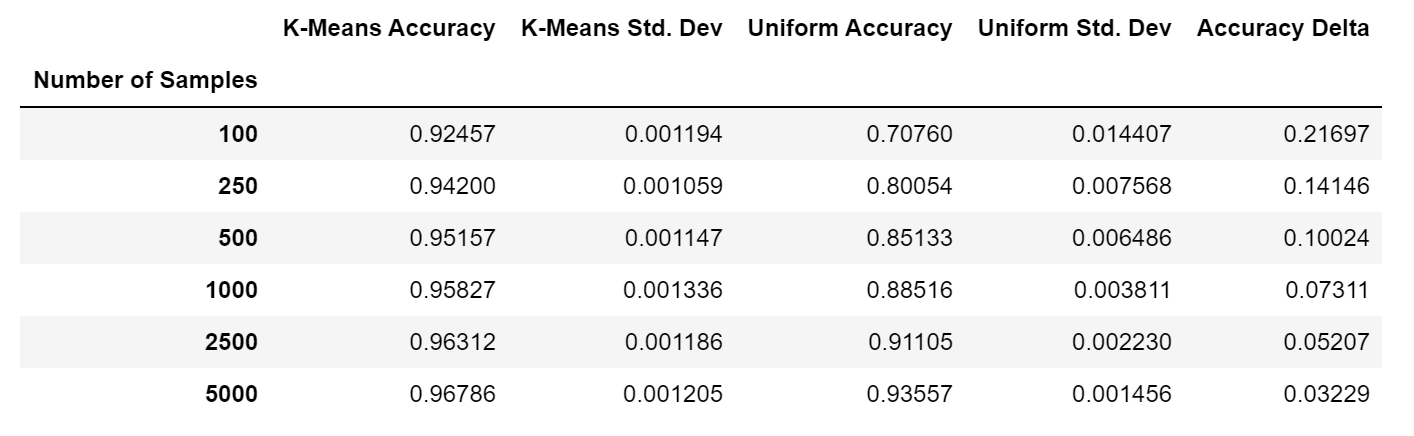
Evaluate performance on the test set

The algorithm below was run using PCA to reduce dimensionality. I determined that maintaining 330 eigenvectors would retain 99$\%$ of the variation information by looking at the cummulative sum of the eigenvectors. This dimensionality reduction was also utilized when drawing random samples so that the performance was compared on a level playing field. The scikit-learn python library was used to implement K-means clustering algorithm and return the cluster centers for each class. It should be noted that the K-Means function has a function parameter called n\_init that specifies the number of times that the cluster seeds are initialized, the default value of n\_init is 10. The function then returns the top performing cluster centers. Therefore, the algorithm is already pre-selecting the higher performing cluster centers prior to selecting prototype points. This effect is evident in the consistency of the clustering method even when selecting 100 total prototype points. After returning the K-means prototype points, the training set is sampled randomly. The sample points are used to construct kd trees and their performance is evaluated on the test set, which is also projected onto an the eigenbasis.

**EXPERIMENTAL RESULTS**

The performance of the k-means prototype method was compared against the uniform sampling over a series of sample quantities ranging from 100 points up to 10000 points. To examine the repeatability and robustness of the two methods, 10 experiments were conducted at each of the sample points. Results are tabulated in the graph and table below. The error bars on the graph indicate a single standard deviation for the accuracy of the respective methods at each sample quantity.





**CRITICAL EVALUATION**

**Prototype Performance:**

It can be seen that the K-Means prototype method significantly outperforms the uniform sampling approach, especially at values <1000, where we are sampling less than 2% of the possible training set. Additionally, the deviation of the K-means method is nearly identical regardless of the number of samples, speaking to the robustness of the K-means clustering method at picking cluster centers that accurately summarize the spatial distribution of the data. An interesting result of the K-means method is the 92% accuracy at 100 samples, meaning 10 prototype points per class. Therefore, necessarily, the improvements are incremental as we increase sample size.

As expected, the uniform random sampling method increases in accuracy and precision as the samples increase – approaching the accuracy of the K-means prototype method. As the number of samples go up, we begin to more accurately approximate the distribution of the various classes, and we are also more likely to sample points near decision boundaries, which will be especially important in a 1NN search.

**Why it works:**

Intuitively it makes sense that if we want to represent a broad space of points, we can attempt to summarize it by picking points that are the centers of clusters of points. And that since we are sampling from a large training set, the points in the test set will also likely fall into one of these clusters. By operating in this way, we are also making the assumption that points are spread relatively evenly throughout their respective subdomains, and that points from classes aren’t generally densely clustered near decision boundaries. As a result the prototype points will in theory have some spread between classes which will improve the accuracy of a NN search. In a way, it acts like its own intraclass KD tree, except the hypercubes aren’t true hypercubes and they are defined implicitly by the closest distance rule. And each hypercube is represented sparsely by a single point within the cube.

**Where it doesn’t work**

Because these clusters are created agnostic to the locations of the other clusters, it is possible for these hypercubes of different classes to intersect and as a result you get false matches. This would be the case if points for one class – call it class A – were heavily distributed near a decision boundary, with multiple cluster centers near a decision boundary, and another class – class B – has sparsely located points near that same decision boundary. We could easily see scenarios where points in class B are falsely assigned to class A due to the cluster centers near the boundary.

**Improvements:**

The least vector quantization approach in this book address the issue of having context of positions of other points, and also prevents prototypes from aggregating along a decision boundary. It starts with prototype initializations as the k-means cluster centers and then adjusts the location of the prototype points closer or farther from sample points depending on the class.

For the clustering method, one thing we could address to improve performance is to prevent the cluster hypercubes from intersecting the hypercubes of different classes. One method could be defining a hypersphere around the cluster, where the radius is the average distance of all points assigned to that cluster. One could then require that spheres from different classes do not intersect, and could adjust the position of the points accordingly so that the spheres do not intersect.

Spitballing on different approaches – One could build a KD tree and then parse through the leaves created on the tree, finding cubes that are homogenous in their classes and then remove all but one of the datapoints on the leaf.