Speeding up K-Nearest Neighbors Classification with Data Space Latticing and Precomputation using Distance Weighting

Leonardo Valli and Alan Zhu

January 15, 2025
Dr. Yilmaz
Period 5
Quarter 2 Project

Table of Contents

1. Abstract **3**
2. Introduction and Goals **3**
3. Related Work **3**
4. Dataset and Features **4**
4.1. Dataset Overview 4
4.2. Attributes 5
4.3. Preprocessing 5
4.4. Train-test split 5
5. Methods **6**
5.1. Overview of canonical KNN algorithm 6
5.2 Dataspace latticing 6
5.3 Proximity search 7
5.4 Distance weighting 8
6. Experiments/Results/Discussion **8**
6.1. Performance metrics 8
6.2. Models 9
6.3. Runtime results 9
6.4. Accuracy results 10
6.5. Confusion matrices 10
7. Conclusions **11**
8. Project Contributions **11**
9. References **11**

1. Abstract

The KNN classifier is a simple supervised machine learning algorithm that performs well for classifying many clustered datasets. However, one major flaw with the KNN algorithm is its O(N) classification time complexity, making KNN impracticable for large datasets. In this project, we use data space latticing to enable O(1) classification using KNN. To speed up the latticing process, we use a tree-based proximity search method (also known as approximate nearest-neighbors, or ANN). We demonstrate that these techniques make KNN classification significantly faster without sacrificing accuracy.

2. Introduction and Goals

The KNN algorithm is one used to classify an unknown instance using the majority labels of the closest other points to it in the training dataset. The main computation involved in this algorithm is the distance calculations between the testing instance and each of the training instances, as well as sorting those distances in order to find the training instances that are closest to the new instance. As the KNN algorithm does not build a model, this process is repeated for each testing instance that is classified, making KNN computationally expensive and time consuming, especially as the amount of training data increases. In order to combat this, our goal was to create a model out of the KNN algorithm by latticing the dataspace that the training dataset is located in, running the KNN algorithm on the center-point of each one of the hypercubes of dataspace, and then storing the classifications into a look-up table that would act as our KNN model. We anticipated that this would be a major improvement on the runtime of KNN, with only a negligible reduction of model classification performance.

We also experimented with a proximity-search implementation of KNN and weighting the distance calculations. The proximity search was another attempt to speed up the KNN algorithm, by using a method to calculate the distances to only the closest few points to the testing instance, instead of making a distance calculation for each of the points in the dataset and sorting them afterward. We were aware that proximity search has some flaws in its accuracy, but we hoped that the KNN speed-up would outweigh the lost model accuracy. Finally, we tested a distance weighting equation to see if it would improve the accuracy of our model in situations where the dataspace latticing or proximity search may have diminished the model's performance.

3. Related Work

There have been variations of KNN precomputation, proximity searching, and distance weighting performed by researchers in the past. In particular, Jeroen Van Der Donckt's (2021) Product Quantization k-Nearest Neighbors (PQKNN) served as inspiration for our version of a precomputed KNN. Donckt's implementation involves finding centroids in the training data, classifying them with KNN, and then using only those centroids during classification. This reduces the amount of data that needs to be stored, and limits the number of euclidean distance calculations performed to the number of centroids. With this, training time was increased, but testing time was heavily reduced. This was also our motivation for attempting a version of KNN precomputation that involved latticing the training data space. However, PQKNN's time complexity is O(Nmw/k'), where N is the number of instances, m is the number of attributes, and w and k' are algorithm-specific hyperparameters. This is still an O(N) linear algorithm, although it can be sped up with large k'.

Spotify (Li et al., 2016) and Meta (Douze et al., 2024) both implement their own versions of proximity search. Their runtime speedups of O(log N) inspired us to use proximity search in an attempt to further reduce the runtime of our KNN implementation. Finally, we tried different weighting functions to see if weighted distances would help reduce the performance-loss of our KNN algorithm by dealing with other central KNN issues like certain k values not performing well. Related works to this are Gou et al. (2012) and Zuo et al. (2008).

4. Dataset and Features

We used the same dataset as our Quarter 1 project.

4.1. Dataset Overview

We use the Transiting Exoplanet Survey Satellite (TESS) dataset, which gathers data from a satellite and telescopic search for exoplanets. The dataset includes certain characteristics of the exoplanet, the star the exoplanet orbits around ("host star"), the position of the exoplanet with respect to earth, the time the data was collected, and identifying information of the exoplanet.

The TESS satellite identifies exoplanet candidates. Because the satellite might not be accurate, the candidates require follow-up validation using ground-based telescopes to verify it is a confirmed exoplanet, or if it is a false positive.

Link to dataset: https://exofop.ipac.caltech.edu/tess/view_toi.php.

The dataset is maintained by the Exoplanet Follow-up Observing Program, NASA Exoplanet Science Institute, operated by the California Institute of Technology, Infrared Processing and Analysis Center.

4.2. Attributes

The raw dataset has 61 attributes and 1 class. At the time of dataset acquisition, there were 7217 TESS candidates (instances).

The attributes are divided into 5 categories: TESS identification, position, planet properties, stellar properties, dates.

Attributes found in the planetary and stellar properties attribute categories generally were found to be normally distributed when plotted. This is why we chose to use a Gaussian distribution model for our lattice, as discussed in the Methods section.

4.3. Preprocessing

We did all of our data preprocessing in Python, in a Google Colab notebook.

First, we included in our dataset only the exoplanets that have already had a determination made to their TFOPWG status. This means we only used instances that had a class of confirmed planet (CP), known planet (KP), false positive (FP), or false alarm (FA) — we discarded PC (planetary candidate) and APC (ambiguous planetary candidate). We then transformed the class into a boolean: true means it is an actual exoplanet, false means it is not an exoplanet.

Then, we removed attributes that we know intuitively have no predictive power. We also removed attributes that are based on calculations by astronomers intended to predict whether the candidate is an actual exoplanet (our model should operate only on the raw data). This brought us down to 22 attributes, from the original 62. We also removed one attribute because it consisted of mostly empty values, and filled in the rest of missing values with attribute means.

We normalized all of the attributes using z-score normalization due to the presence of outliers in several of our columns. Normalization also simplifies our lattice algorithm as discussed in the Methods section.

Finally, we used the CFS subset attribute selection method, found in our Q1 project to perform best, to select 12 attributes to use: Time Series Observations, Spectroscopy Observations, Imaging Observations, Period, Depth, Planet Radius, Planet Insolation, Planet Equil Temp, Planet SNR, Stellar Distance, Stellar log(g), and Stellar Mass.

4.4. Train-test split

On each run, we randomly split the dataset into training (70%) and testing (30%). There is no validation dataset since KNN does not use one. We did not have a fixed train and test dataset because we are trying to evaluate the speed of KNN. This means that we need a large number of instances to test on — just one pass on our dataset alone is not enough. Instead, we ran many tests on our dataset, randomly splitting into training and testing each time. To evaluate our algorithm, we ran 10,000 tests on our dataset of 2165 instances, which means the algorithm classified nearly 22 million instances. The large number of tests makes the statistics for the random variation in accuracy between individual tests average out for our overall accuracies of the different models.

5. Methods

5.1. Overview of canonical KNN algorithm

KNN is a supervised learning algorithm that classifies a testing instance by calculating its euclidean distance to each training data point and assigning it the label of the most commonly occurring class among the k nearest training points to it. First, the distances are calculated and stored. Then, they are sorted, and the smallest k distances are taken. Finally, the testing instance is classified based on the majority-vote of those k training instances. The k-value used in the KNN algorithm can be tuned in training.

5.2 Dataspace latticing

The core of our project is to lattice the dataspace into hypercubes so that any point within the boundaries of that hypercube is assigned the classification for that hypercube. We chose to use a lattice because our major goal is to enable O(1) classification, which can be achieved with a lattice. Other fast implementations of KNN often use proximity search, but have O(log N) classification (see section 6.3).

The high-dimensional nature of our dataset means the lattice will have an enormous number of points. Therefore, it is very important to select the points of the lattice to be at the most useful locations, i.e. to split the dataspace up evenly and ensure a large proportion of points within each hypercube have the same class. We could split up the dataspace using equidistance latticing, but that does not match our dataset's distribution. Knowing that our dataset is well-modeled by a normal distribution, we chose lattice points on a multivariate Gaussian distribution to evenly divide up the dataspace. Note that our algorithm could be modified to use any statistical distribution, not just Gaussian. Specifically, consider one feature we call x with mean μ and standard deviation σ. We want to chose points $x_1, x_2, x_3 ... x_n$, where n is the number of lattice points per dimension, such that

$$normalcdf(-\infty, x_1) = normalcdf(x_1, x_2) = normalcdf(x_2, x_3) = \ldots normalcdf(x_n, \infty) = \frac{1}{n}$$

where

$$normalcdf(a, b) = \int_a^b \frac{1}{\sigma\sqrt{2\pi}} e^{-\frac{(x-\mu)^2}{2\sigma^2}} dx$$

This splits the dataspace up into n^m hypercubes, where m is the number of dimensions (features).

Extending this to the full dimensionality of the dataset, the probability an instance randomly chosen from the distribution lies within any given hypercube is given by

$$\int \cdots \int_m \frac{e^{\left(-\frac{1}{2}(\mathbf{x}-\boldsymbol{\mu})^{\mathrm{T}}\boldsymbol{\Sigma}^{-1}(\mathbf{x}-\boldsymbol{\mu})\right)}}{\sqrt{(2\pi)^m|\boldsymbol{\Sigma}|}} d^m x = \frac{1}{n^m}$$

where x and μ are now vectors and Σ is the covariance matrix, and with bounds on each integral of consecutive x instances ($\int_{x_i}^{x_{i+1}}$).

To allow for easy O(1) access, each lattice point x_i is stored in the lattice data structure, a dictionary, as the number

$$L_i = \frac{normalcdf(-\infty, x_i) \times (n+1)}{S}$$

where S is an arbitrary scale factor > n chosen to make roundoff and interpretation easier. For example, with S = 10, and n = 3, the lattice points for each dimension would be stored as 0.1, 0.2, 0.3. For implementation in code, L_i does need to be clipped between an arbitrary maximum and minimum value to handle infinities, and rounded to account for roundoff error.

The precomputation algorithm iterates over every lattice point, computes the KNN classification for that point, and stores it into the lattice. Then, when classifying a point, that classification is retrieved from the lattice in O(1) time. Additionally, this lattice can be stored and loaded for future iterations without computing it again.

5.3 Proximity search

Given the enormous number of lattice points, it is impracticable to precompute each point using the canonical O(N) KNN algorithm, as that would take an extremely long time. Instead, we use an approximate nearest-neighbors (ANN) implementation called Annoy, created by Spotify (Li et al., 2016). This algorithm does not guarantee to find the exact *k* nearest neighbors, but it is a close approximation that runs in O(log N), a significant speedup. We used Annoy to precompute the classifications of the lattice points.

Annoy uses a forest of random projection trees to divide up the dataspace randomly using hyperplanes. The hyperspaces left in the end are likely to have the same classification. During classification, an instance is classified using this forest of trees to find the hyperspace it belongs to, followed by a short KNN search within the hyperspace, to obtain the resulting classification.

5.4 Distance weighting

Beyond speeding up KNN, we also tried to improve its accuracy by weighting the nearest-neighbor classes by distance. Neighbors closer to the instance to be classified are weighted higher, making selecting their class more likely. The weighting was done using the formula:

$$w = \frac{1}{(d + b_1)^{b_2}}$$

where *d* is the distance metric between the neighbor and instance (e.g. Euclidean, Manhattan, etc.) and b_1 and b_2 are hyperparameters.

The goal of this distance weighting was to combat one of the main flaws with the KNN algorithm: the simplicity of a majority-rule voting system between the k nearest neighbors to the testing instance. Our hope was that this would help mitigate the effects of using ANN on our model's performance, as proximity search tends to be quite variable and we did not want for the runtime decrease to cost us model accuracy.

6. Experiments/Results/Discussion

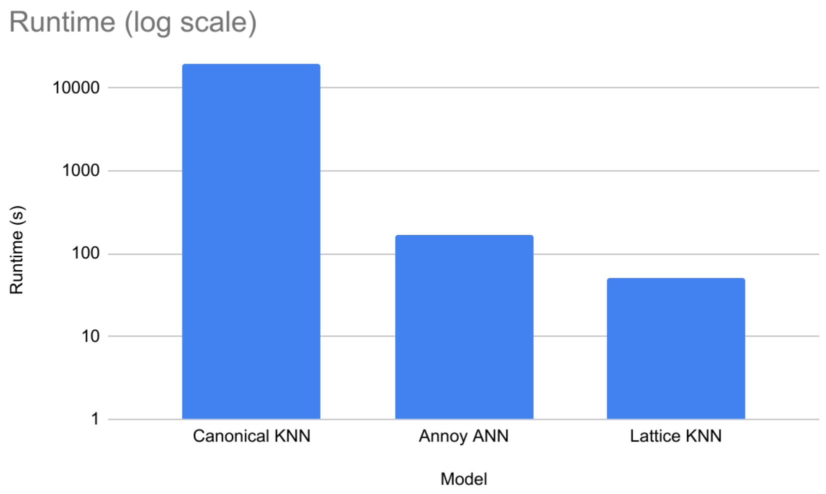
6.1. Performance metrics

Our main performance metrics were classification accuracy and classification runtime. We aim to achieve a much lower classification runtime at the same classification accuracy. We also used confusion matrices to visualize if there were any class-specific performance alterations for the model across the algorithms we tested.

6.2. Models

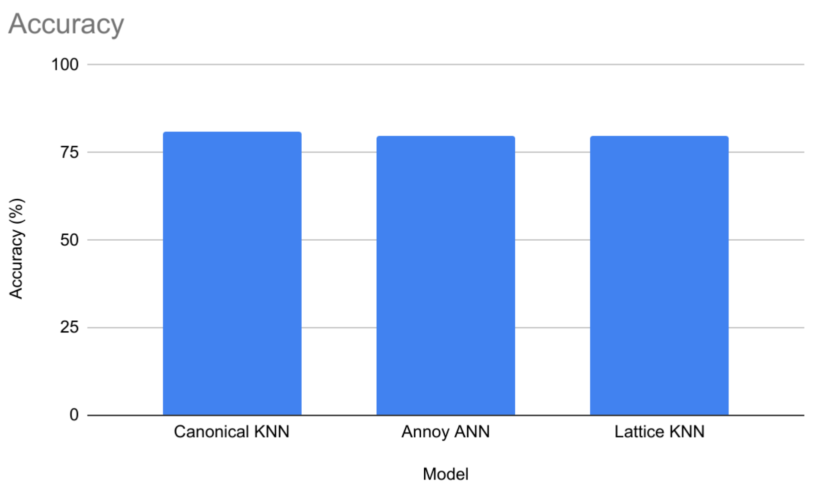
We evaluated three models. First is the canonical KNN algorithm, the default one. This is the control. Second is the Annoy ANN using proximity search. We expect the ANN to be significantly faster with slightly reduced accuracy. This is another control. Finally, our algorithm: the lattice KNN. We aim to achieve significantly faster runtime than both ANN and canonical KNN while at the same or slightly-improved accuracy as ANN. We chose a constant k-value of 3 to use with all of these algorithms, as this value seemed to yield good results when we tuned it on the training dataset.

6.3. Runtime results



For 10000 tests on our dataset, each with their own unique train-test-split, our canonical KNN model took 19500 seconds; the proximity search KNN took 171 seconds, and the dataspace latticing KNN took 51 seconds to classify the testing instances. Note the log scale on the graph, meaning the runtime differences are even larger than they appear. Because canonical KNN was taking so long, we ran 1000 tests with it and extrapolated the runtime for 10000 tests. The 3-slice latticing model took 5.2 seconds to build, while the 4-slice one took 220 seconds, reflecting the large increase in lattice points with base.

6.4. Accuracy results



The canonical KNN had an accuracy of 0.808; the proximity search KNN had a slightly lower accuracy of 0.799; the 3-slice lattice KNN model had an accuracy of 0.777, and the 4-slice lattice had an accuracy of 0.796 over the 10000 tests. This demonstrated that our lattice KNN had no significant decrease in accuracy from the default KNN while running much faster.

6.5. Confusion matrices

Default KNN

	0	**1**
0	254	77
1	59	260

Annoy ANN

	0	**1**
0	268	75
1	58	249

Lattice KNN

	0	**1**
0	253	90
1	44	263

Overall, the performance across the three models were very similar. The default KNN's confusion matrices remained relatively consistent across runs because base KNN is generally deterministic, as long as there are no ties in euclidean distance calculations. However, the proximity search KNN experienced significant variation between trials, and on this trial, this model performed the best on the 0 (False Exoplanet) class and worst on the 1 (True Exoplanet) class. Finally, our 3-slice latticing KNN performed very similarly to the default KNN algorithm for both the 0 and 1 classes.

7. Conclusions

Our goal to create a version of KNN with a computational complexity independent of number of instances was a success. Using only 12 of the attributes collected at the moment of the detection of a potential exoplanet by a satellite, our 4-slice dataspace latticing KNN model could distinguish between true positive and false positive exoplanet detections with 79.6% accuracy, only a small reduction from base KNN's 80.8%. It did this with a classification time of 51 seconds, compared to base KNN's run time of 19500 seconds for 10000 tests. Our predictions were correct, and both our latticing KNN and proximity search implementations exhibited negligible performance losses in order to achieve faster runtimes. If we had more time, we would research more into other KNN precomputation methods, like Jeroen Van Der Donckt's PQKNN, and implement them ourselves..

8. Project Contributions

Alan came up with the lattice KNN algorithm idea, and Leo did research into previous implementations with similar approaches. Alan coded the latticing precomputation and proximity search, and Leo played around with distance weighting methods in the code. Alan and Leo wrote the sections of the report corresponding to the parts each researched/coded.

9. References

Jeroen Van Der Donckt. *pqknn* [GitHub repository]. GitHub. https://github.com/jvdd/pqknn

Li, W., Zhang, Y., Sun, Y., Wang, W., Zhang, W., & Lin, X. (2021). *Approximate nearest neighbor search on high-dimensional data: Experiments, analyses, and improvement (v1.0)*. arXiv. http://arxiv.org/pdf/1610.02455

Douze, M., Guzhva, A., Deng, C., Johnson, J., Szilvasy, G., Mazaré, P.-E., Lomeli, M., Hosseini, L., & Jégou, H. (2024). *The Faiss library*. arXiv. https://arxiv.org/abs/2401.08281

Gou, J., Du, L., Zhang, Y., & Xiong, T. (2014). A new distance-weighted k-nearest neighbor classifier. *ResearchGate*.

https://www.researchgate.net/profile/Jianping-Gou/publication/266872328_A_New_Dist ance-weighted_k_-nearest_Neighbor_Classifier/links/5451acdf0cf2bf864cba99fc/A-New-Distance-weighted-k-nearest-Neighbor-Classifier.pdf

Zuo, W., Zhang, D. & Wang, K. On kernel difference-weighted *k*-nearest neighbor classification. *Pattern Anal Applic* 11, 247–257 (2008). https://doi.org/10.1007/s10044-007-0100-z