Optimizing *k* in K-Nearest Neighbors Using Adaptive Gradient Descent

Nikki Kudamik
Department of Computer Science, Slippery Rock University
Slippery Rock, United States
nak1005@sru.edu

Eli Podgorski
Department of Computer Science, Slippery Rock University
Slippery Rock, United States
ejp1008@sru.edu

***Abstract*— The performance of the K-Nearest Neighbors (KNN) algorithm is closely tied to the choice of *k*, the number of nearest neighbors used in predictions. Selecting the optimal *k* is often done through grid search or manual tuning, which can be time-consuming and inefficient, especially for large datasets. This paper introduces an alternative approach using gradient descent to optimize *k*, combined with an adaptive learning rate to refine the optimization process. The method aims to minimize a cost function that reflects the accuracy of the KNN model, enabling a systematic way to find a suitable *k*.**

Experimental results show that the proposed approach performs comparably to traditional methods, offering similar levels of accuracy and occasionally achieving slight improvements. This work explores how modern optimization techniques can be applied to classical algorithms like KNN, providing insights into the potential and limitations of gradient-based hyperparameter tuning.

Keywords: K-Nearest Neighbors, gradient descent, adaptive learning rate, hyperparameter optimization.

I. Introduction

The K-Nearest Neighbor Clustering algorithm (KNN) is one of the most widely used, fundamental supervised machine learning techniques today. KNN is generally considered to be a non-parametric classifier method, however it is also frequently utilized in regression problems. Non-parametric methods, meaning methods that make no assumptions about the statistical distribution of data (i.e. a bell curve distribution), are a great way to manage unstructured/real-world data. As the name suggests, in order for KNN to work, we assume that points of homogeneity in a dataset neighbor each other, and in that way, we can assign similar points a class label.

Implementing the KNN algorithm involves four basic steps [2]:

1. Select the right k value

Deciding the correct k value can be somewhat tricky, so it is important to preview the dataset being analyzed before determining this factor. In simple terms, however, the k value in KNN should represent the number of neighboring data points being considered.

Think of the test-point you have in mind; radiating from that point are a few circles of area that each encompass an increasing number of other, pre-labeled points. Since k is the number of neighboring data points surrounding the chosen test-point, the larger the k (and, consequently, the larger the radius from the test-point) the more accurate the data, typically. Outliers can affect this variable's accuracy.

2. Calculate distance

Once a test-point is chosen and a k value determined, the next step is to calculate the distance between the test point and its k-trained nearest neighbors. This is done using a distance metric, usually either Euclidean or Manhattan distance. Euclidean distance is the measurement of the length of a straight line made between two points in space (the blue line in Fig. 1). Manhattan distance, on the other hand, measures the difference between points using only vertical and horizontal segments (the red line in Fig. 1).

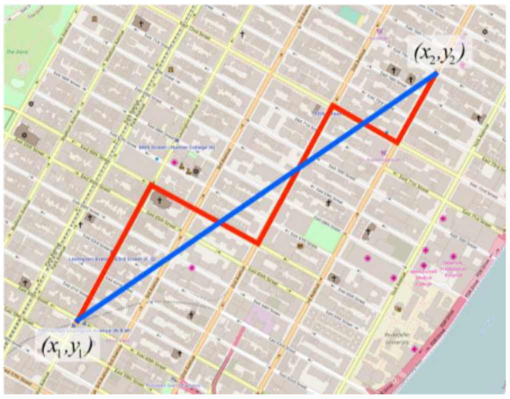

Fig. 1. Comparing Euclidean (blue) and Manhattan (red) distance metrics. [8]

3. Locate nearest neighbors

   Of the k-trained nearest neighbors determined in the previous step, identify those with the smallest distances from the test-point.

4. Decide: classification or regression

   Establish the purpose of the task at hand. For classification problems, assign the test-point a class label based on the majority vote of its surrounding neighbors. For regression problems, take the average of the test-points of the surrounding neighbors [3].

Like anything, even when it produces quality results, the K-Nearest Neighbor clustering algorithm possesses many strengths and weaknesses. As has been previously mentioned, KNN is a clean and easy algorithm to use, oftentimes being one of the first learned by aspiring data analysts (and related fields) due to its relative simplicity in implementation. Not only does it adapt well to new data, but it handles data of different varieties with ease. These key features allow for the algorithm's employment in a broad spectrum of applications [3].

Advantages:

- Simple Hyperparameter (k): KNN has the advantage of using only one main hyperparameter, "k," which is the number of nearest neighbors considered when making predictions. This makes the algorithm simple to understand and implement, especially when compared to models that require the use of multiple hyperparameters. By adjusting "k," you control how the model behaves. A higher "k" value makes the decision boundaries smoother, while a lower "k" makes the model more sensitive to specific and or new data points. In most cases, a small dataset with minimal outliers would call for a small k value to be used. A larger dataset with many outliers would benefit from a large k value.
- Flexible Distance Metrics**:** One of KNN's strengths is its flexibility in choosing how to measure the distance between data points. Depending on the problem, you can select different distance metrics, such as Euclidean, Manhattan, or Hamming distances. This flexibility means that KNN can be adjusted to fit a wide range of datasets and problems.
- No Need to Retrain: KNN has the advantage of not requiring retraining when new data is added. Because the algorithm stores all data points in memory, it simply includes new points as part of the future predictions. This makes KNN favorable in dynamic environments where new data is constantly coming in.

Disadvantages:

- Scalability Problems: KNN does not scale well when working with large datasets. The algorithm needs to calculate the distance between every data point, and as the size of the dataset grows, the time it takes and resources required increase significantly. This makes it impractical for large datasets.
- Curse of Dimensionality: KNN can struggle with high-dimensional data because of the curse of dimensionality. In these cases, the distances between data points become less meaningful and performance suffers. For example, in high-dimensional spaces, distance metrics like Euclidean distance may not provide the most useful information, as data points can all seem equally far from each other.
- Difficulties Choosing the Right "k": Selecting the optimal value for "k" can be difficult and has a big impact on how well the model performs. If "k" is too low, the model will overfit, while a "k" that is too high can cause underfitting, where the model doesn't capture the details in the data well enough. There is no method for choosing "k" that works for every scenario, so you often must try several values and evaluate their performance, which can be time-consuming.

While KNN is a versatile and intuitive algorithm, it still comes with its drawbacks. Its reliance on a single hyperparameter and flexibility in distance metrics make it usable for different applications, but scalability, the curse of dimensionality, and the difficulty in choosing the ideal k value remain obstacles. Having a good understanding of the advantages and disadvantages of KNN is key in effectively applying it to projects.

Our research and experiments were performed with the intention of developing a new, better way of tuning the hyperparameter k. We proposed implementing gradient descent and an adaptive learning rate within the KNN regression algorithm to return an optimal k value that is comparable to other methods of finding k.

The code we developed yielded positive results in the area of accuracy. However, arguments can be made against the algorithm's efficiency, since the computational complexity and runtime of our method may negate any potential positives. Gradient-based k-tuning using an adaptive learning rate proved, by means of our own programming, slow, and its scalability less than desirable. It is important to note that the experiments conducted used only KNN regression and have not been tested using KNN classification.

II. Literature Review

As mentioned, one of the biggest disadvantages of the k-Nearest Neighbor algorithm is that the user must define the variable k themself. The k-value is integral to the accuracy of the algorithm, and the correct choice relies on several different factors.

One of those factors is Bias-Variance Trade-Off. In machine learning, 'bias' refers to the difference between the algorithm's predicted values and the correct values. As for variance: "Variance in machine learning models depends on the ability of the model to accurately predict the targets of unseen data…This does not focus on the overall accuracy of the model but instead simply measures the spread, or uncertainty, in model estimates" [4]. Therefore, the bias-variance trade-off is the resulting mixture of the two in assessing the outcome of an algorithm such as KNN. This trade-off relies heavily on the k-value chosen. Too small a k-value or too large a k-value can skew this evaluation.

Another, more obvious factor in choosing a k-value would include a dataset's 'noisiness' or the number of outliers it contains. Choosing a smaller k-would not allow for many outliers to be in the dataset, as any outliers would more significantly impact the final output of the algorithm as compared to a larger k-value. A larger k, however, could mean that the output data is more generalized. Deciding which k to use when considering this factor is entirely dependent on what kind of answers the researcher is looking for.

The computational cost of the algorithm is an especially crucial factor to consider, especially if KNN is being used in some official capacity for a business. A larger k-value correlates to a higher computational cost, since the more neighbors there are, the more computations must occur.

It might be worthwhile to run KNN multiple times using varying k-values, after inspecting the data, to compare the outcomes and find which works best for the dataset at hand later. However, as data becomes increasingly complex, researchers have developed many different methods to address the problem of choosing the correct k. One of the most well-known of those being cross-validation.

A. Cross Validation

Typically, when deciding a value for k it is considered a rule of thumb to start out with a value of either 5 or 10. With cross validation, though, it is more common to submit a range of k values. Cross-validation is then performed to find the accuracy scores of those values [5]. In the example used to test this method, a list of 30 k-values was submitted to the scikit-learn's cross_val_score method. An instance of the KNN model, the data, and a cv value of 5 are passed to the function. The cv value is the number of splits that will be made. The article explains: "the model will split the data into five equal-sized groups and use 4 to train and 1 to test the result. It will loop through each group and give an accuracy score, which we average to find the best model" [5]. A chart is then created comparing the k-values to their accuracy scores. Out of the 30 possible k-values, values k=9, 10, 11, 12, and 13 reported the highest accuracy scores of just under 95%. Since it was a 5-way tie for the most accurate k-value, it is recommended to use the smallest, odd value. This is justified since it is more cost and time-efficient to use less calculations during the execution of the algorithm.

Cross-validation is considered a boon within many machine learning spaces. However, it is not desirable in all applications. For example, within the medical field data can oftentimes be scarce, especially data surrounding rare diseases. Splitting data like this into multiple folds using cross-validation may create small and inaccurate training datasets—worsening KNN (or other model) performance.

B. Elbow Method

Maulana and Roestam's paper [6], "Optimizing KNN Algorithm Using Elbow Method for Predicting Voter Participation Using Fixed Voter List Data (DPT)", investigates improving the accuracy and efficiency of the K-Nearest Neighbors algorithm for predicting voter participation. The authors focus on optimizing 'k,' which is the number of neighbors considered by the algorithm.

In their study, Maulana and Roestam incorporate the elbow method to find the ideal k value. The elbow method is a technique in cluster analysis that plots the sum of squared errors (SSE) against various k-values. While visualizing the plot, they look for the point where the SSE reduction starts to level out (elbow), and this is considered the optimal k. In this study, they plotted first against k values 1-100. They found that the decrease in error value came from points 20-60. They then plotted SSE v K again with the k range of 20-40. In the new visualization, they saw the sharp decrease (elbow) at K=31, thus making the optimal K value equal to 31.

The study conducted uses real-world voter data to predict participation. The application of KNN shows reliable results in predicting behavior based on historical patterns. The elbow method in this situation makes the model adaptable to diverse types of data. Maulana and Roestam also touch on the importance of hyperparameter tuning in machine learning as well as stating that methods like the elbow method can improve algorithm performance.

To conclude, the paper provides valuable insight into improving KNN's ability in prediction tasks, especially in areas with structured and categorical data. It reinforces the need to tune your model and shows the elbow method as a viable method to use in classification models.

C. *Grid Search*

The paper, "KNN Optimization Using Grid Search Algorithm for Preeclampsia Imbalance Class" by Sukamto, Kadiyanto, and Kurnianingsih investigates the optimization of KNN using the Grid Search method, to improve their ability to predict pre-eclampsia [7]. Pre-eclampsia is a severe complication in pregnancy and is a leading cause of maternal death across the globe. If able to detect the condition early enough, maternal morbidity and mortality could be significantly reduced. Thus, making it essential to develop an accurate predictive model.

A significant problem when working with medical data is imbalanced datasets. One class, in this case pre-eclampsia patients, is underrepresented compared to the healthy individuals. Machine learning models struggle with imbalanced data and tend to have biased predictions towards the majority class. Their study aims to optimize hyperparameters for both KNN and Decision Tree models using Grid Search to deal with these imbalances better. The Grid Search method is a method for hyperparameter optimization that evaluates all combinations of hyperparameters to identify the best set. This study combines Cross Validation and Grid Search. The grid search technique will construct different versions of the model and will return the best one. The KNN algorithm, which is known for its simplicity, is non-parametric and effective for classification problems. Its downside is that its performance is highly dependent on the value of k, which can be optimized through hyperparameter tuning.

Their results showed that KNN outperformed Decision tree in both accuracy and cross validation score. The KNN model had a cross-validation score of 63.45% compared to 62.05% for the Decision tree. Results also showed that Grid search can be a powerful tool for optimizing machine learning models in imbalanced datasets. It ensures that the best combination of parameters is selected for the task at hand.

In conclusion, this study shows that hyperparameter tuning through Grid Search can improve performance of machine learning models, especially when working with imbalanced datasets. The optimized KNN model provided better prediction for pre-eclampsia classification and showed it could be used in real world applications.

III. Describing the Data

Since KNN uses distance to classify points, we decided to use a dataset containing information about housing prices in California. Usually, houses within a neighborhood resemble each other in build and value--making this the perfect dataset for our purpose. Specifically, each house in this dataset contains values such as longitude, latitude, total_rooms, total_bedrooms, median_house_value, and so on. All the features within this dataset are float types except for one: ocean_proximity. There are means to encode this ordinal attribute. However, for simplicity's sake, we decided to drop this column.

This dataset in particular came from Kaggle. However, the CA housing dataset is well-known within the machine learning community and is used frequently for education and practice. The shape of the dataset is 20,640 rows by 10 columns. Gradient descent benefits from larger, standardized datasets. The preprocessing of this dataset remained simple. First, we derived descriptive information to help visualize the data's structure using pandas' .info() and .describe() methods. Next, we found any instances of missing values within the columns and filled them. We then created a correlation matrix to test which values best correlated positively with the median_house_value feature. Median_income, total_rooms, and housing_median_age were the top three most correlated attributes. Finally, we standardized the data manually by using the z-score mean. From there, we split the data into testing and training and tested our Gradient-based k-tuning method using KNN Regression.

After using the CA housing dataset, we wanted to see how our method would perform on a dataset with more dimensionality. So, we used another dataset, also from Kaggle, that contains data about automobiles. This dataset is significantly smaller with only 205 rows and 25 columns. Using KNN Regression to estimate the price of each vehicle, the algorithm would have to consider many more features—which one would expect to impact the model's performance significantly. We performed the same preprocessing steps on this dataset used previously with the CA housing dataset.

IV. Algorithm Breakdown

The primary goal of this work is to optimize *k*, the number of neighbors considered in the KNN algorithm, to minimize classification or regression error. Traditional approaches, such as grid search, can be computationally expensive and do not adapt dynamically. This method uses gradient descent to iteratively adjust *k* based on the direction of the steepest decrease in a defined cost function, enabling a more efficient and systematic search of the hyperparameter space.

A. *Technique Overview*

To optimize the hyperparameter *k in* KNN, this project leverages two primary techniques:

1. Gradient descent for *k* optimization: Gradient descent, an optimization algorithm, is adapted to dynamically adjust *k* based on the gradient of a defined cost function (Mean Squared Error in this

project). While gradient descent is commonly used in continuous scenarios, *k,* being a discrete integer, requires unique adaptation. Numerical differentiation is used to estimate the gradient with respect to *k*, ensuring the algorithm can identify the direction of steepest error reduction.

2. Adaptive Learning Rate with RMSProp: RMSProp is an optimization algorithm that adjusts the learning rate based on the magnitude of recent gradients. This is incorporated to stabilize and speed up convergence to the optimal *k.* It computes a moving average of squared gradients and uses this to normalize updates, preventing overshooting or becoming stagnant. RMSProps dynamic adjustment ensures more efficient tuning of *k.*

These methods together create an interesting way to search k-values in comparison to those that are nonadaptive.

B. Cost Function

The cost function used in this project is the Mean Squared Error (MSE), a widely used metric for evaluating the performance of models. MSE quantifies the average squared difference between the actual target values and the predicted values, providing a measure of how well the model is performing. A lower MSE indicates more accurate predictions, as the predicted values are closer to the actual values.

The MSE formula is defined as follows:

$$MSE = \frac{1}{n}\sum_{i=1}^{n}(y_i - \hat{y}_i)^2$$

where n is the number of test samples, y_i is the actual value, and $\hat{y}_i$ is the predicted value.

In this project, MSE is computed during each iteration of the gradient descent process. The function *knn_cost* is used to calculate the MSE for a given value of *k.* It trains the K-Nearest Neighbors (KNN) regressor on the training data, predicts outcomes on the test data, and computes the MSE using the *mean_squared_error* function from Scikit-learn. By comparing the MSE for different values of *k*, the algorithm determines the gradient direction to iteratively update *k* and minimize the error.

C. Gradient, RMSProp, and Adaptive Learning Rate

In the algorithm, numerical differentiation is used to approximate the gradient of the cost function with respect to *k*. the gradient provides the direction of the steepest decrease in the cost function and is a crucial component of gradient descent.

Since *k* is a discrete hyperparameter (an integer), its gradient cannot be computed using analytical methods, so numerical differentiation is used, as shown in the code.

1. A small change (Δk = 0.01) is added to the current value of *k* to compute the approximate gradient.

$$Gradient = \frac{\left(MSE(k + \Delta k) - MSE(k)\right)}{\Delta k}$$

2. To handle the discrete nature of *k*, the ceiling function (math.ceil) is applied to *k* + Δk, ensuring the number remains a valid integer.

This estimation is sufficient for guiding the updates of *k* because the changes in MSE due to small variations in *k* are smooth enough to estimate the gradient effectively. Care is still taken to force *K* into integer values after each update, as fractional values of *k* are invalid in the KNN algorithm.

RMSProp (Root Mean Square Propagation) is an adaptive learning rate optimization algorithm that adjusts the learning rate based on the magnitude of recent gradients. It is especially beneficial for hyperparameter optimization, like tuning *k*, where the cost surface can be non-linear and have areas with varying gradients.

In RMSProp:

RMSprop maintains an exponentially decaying average of the squared gradients, denoted as E$[g^2]$ This is calculated as:

$$E[g^2]_t = \beta E[g^2]_{t-1} + (1-\beta)g_t^2$$

Where:

a. $E[g^2]_t$ is the exponentially decaying average of the squared gradients at the current iteration t
b. β Is the decay rate
c. g_t the gradient of the cost function at iteration t
d. $E[g^2]_{t-1}$ is the exponentially decaying average of squared gradients from the previous iteration

The moving average helps control the optimization process by accounting for the magnitude of recent gradients.

The learning rate for each update is dynamically scaled by the square root of the moving average of the squared gradients:

$$\alpha_t = \frac{\alpha}{\sqrt{E[g^2]_t + \varepsilon}}$$

Where:

a. α_t is the dynamically adjusted learning rate at iteration t

b. α is the base learning rate

c. $E[g^2]_t$ is the exponentially decaying average of squared gradients at iteration t

d. ε Is a small constant

This formula scales the learning rate inversely to the square root of the moving average of squared gradients. Larger gradients reduce the learning rate while smaller rates allow for larger steps.

This approach ensures stability by normalizing the step size based on the gradient history. Larger gradients result in smaller step sizes, reducing the risk of overshooting, while smaller gradients allow for larger step sizes, improving convergence efficiency.

D. Code Explanation

The code for our method contains two main functions, knn_cost and error_gradient_k_tuning. The knn_cost function evaluates the performance of a KNN model for a given number of neighbors (k), while the error_gradient_k_tuning function applies gradient descent with RMSprop to optimize the value of k for the best model performance.

The knn_cost function calculates the mean squared error for a KNN model given a specific k value. This function ensures k is an integer, initializes the model, trains it on the input data, and predicts the outputs for the test set. The MSE is then computed to evaluate the model's performance. The code is as follows:

```
def knn_cost(x_train, y_train, x_test, y_test, k):
  """Calculates the error for a given k-value in KNN."""
  # Diagnostic: Print k to check if it's an integer
  print("knn_cost:", k)
  # Force k to be an integer to avoid issues with array-like values
  k = int(k)
  knn = KNeighborsRegressor(n_neighbors=k)
  knn.fit(x_train, y_train)
  y_pred = knn.predict(x_test)
  return mean_squared_error(y_test, y_pred)
```

The error_gradient_k_tuning function uses gradient descent with RMSProp to optimize the k parameter for KNN model. It starts with an initial guess for k and iteratively adjusts it based on the gradient of the error function with respect to k. RMSProp ensures stable updates by scaling the learning rate using the running average of squared gradients. Each iteration calculates the gradients, updates k, and records the error. The process stops early if the gradient becomes significantly small. The code is as follows:

```
def error_gradient_k_tuning(x_train, y_train, x_test, y_test, initial_k, initial_alpha, iterations, rmsprop_beta=0.9, epsilon=1e-8):
  # Performs gradient descent with RMSProp to find the optimal k for KNN.
  k = initial_k
  k_dict = {}
  cost_history = np.zeros(iterations)
  squared_gradients_avg = 0.0  # Initialize as a float for RMSProp
  for iter in range(iterations):
    # Diagnostic: Print current k value and type
    print("\nIteration:", iter, "Current k:", k, "Type of k:", type(k), "\n")
    # Calculate the current error with k forced to an integer
    current_error = knn_cost(x_train, y_train, x_test, y_test, int(k))
    print("Current Error:", current_error)
    # Numerical gradient approximation
    k_plus_delta = k + 0.01
    error_plus_delta = knn_cost(x_train, y_train, x_test, y_test, math.ceil(k_plus_delta)) # use ceiling function here
    print("Delta Error:", error_plus_delta)
    # Gradient calculation
    gradient = (error_plus_delta - current_error) / (k_plus_delta - k)
    # Update the running average of squared gradients
    squared_gradients_avg = rmsprop_beta * squared_gradients_avg + (1 - rmsprop_beta) * (gradient ** 2)
    # Adjust learning rate based on RMSProp formula
    adjusted_alpha = initial_alpha / (np.sqrt(squared_gradients_avg) + epsilon)
```

```
        # Print out the current adjusted learning rate
        print("Adjusted Learning Rate:", adjusted_alpha)
        # Update k using RMSProp-adjusted learning rate
        k = k - adjusted_alpha * gradient


        # Round and track k
        k = np.ceil(k)
        print("Rounded/Updated k:", k)
        k_dict[k] = current_error
        # Record the cost
        cost_history[iter] = current_error
        # Optional: Stop if gradient is minimal
        if abs(gradient) < 1e-5:
            break
    min_k = min(k_dict, key=k_dict.get)
    return min_k, cost_history
```

Together, these functions work together to efficiently determine the optimal number of neighbors for a KNN model.

E. Complexity Analysis

This algorithm uses gradient descent with RMSProp to optimize the *k* value for KNN. While it offers a dynamic alternative to traditional methods like GridSearchCV, it tends to scale poorly as the size of the dataset, or the dimensionality, increases. This is because the algorithm requires repeated computations of the KNN cost and gradient calculations throughout the iterative process.

In comparison, GridSearchCV precomputes model performance for a predefined set of *k* values, which can be more efficient for larger datasets. However, our approach provides finer control over the tuning process, making it more suitable for smaller datasets where runtime is not a big concern. For larger datasets, GridSearchCV or other scalable optimization techniques may be a better choice.

F. Advantages / Tradeoffs

The gradient descent algorithm with an adaptive learning rate demonstrated both significant advantages and tradeoffs in our implementation. One of the primary advantages is its ability to achieve high accuracy in optimizing the number of *k* for the KNN algorithm. By dynamically adjusting the learning rate during training, the algorithm effectively navigates the space, converging on an optimal *k* value. This adaptability reduces the likelihood of overshooting the minimum or getting stuck in nonoptimal solutions, which is an issue seen in static learning rate implementations.

However, this precision comes at the cost of computational efficiency. The adaptive nature of the learning rate requires additional calculations at every iteration to evaluate and adjust the step size based on the gradient behavior. Combined with the iterative nature of gradient descent, this leads to significantly longer processing times, especially for large or complex datasets.

Despite these tradeoffs, the algorithm's ability to deliver accurate and reliable results makes it a compelling choice for scenarios where accuracy is prioritized over speed. This balance between precision and speed must be weighed depending on the requirements of the application.

V. Results

The accuracy score of our KNN regression model using Gradient-based k-tuning is comparable to those of established methods. The training and testing scores for the CA Housing dataset using our proposed method are 0.7807120994070865 and 0.7309625524789645, respectively. These scores are based off an optimal k value of 8. GridSearchCV returned an optimal k value of 9, which resulted in a training score of 0.7746535502538328 and a testing score of 0.7307519646093426. Although the differences between the two are minute, our KNN regression model outperforms GridSearchCV in terms of accuracy on this dataset. The Elbow Method, on the other hand, returned an optimal k value of 5. This resulted in training and testing scores of 0.8070633939687645 and 0.7228732646101284, respectively. Compared to our Gradient-based k-Tuning approach, the Elbow method trained and tested better. While our method's accuracy scores may present a positive front, the computational complexity of the method is unfavorable. Since our method tests and fits a different k value in each iteration, the overall scalability is poor. In fact, we timed the runtime difference between our model and GridSearchCV using python's time library. KNN regression using Gradient-based k-tuning took approximately 12.5 seconds to run while GridSearchCV took less than a second. With such minimal differences in accuracy, we conclude that GridSearchCV and the Elbow Method remain superior methods of optimizing the hyperparameter k in KNN regression.

VI. Conclusion

This study presented an alternative method for optimizing the k parameter in the K-Nearest Neighbors algorithm using gradient descent with an adaptive learning rate. The approach was designed to provide a systematic way to tune k and achieve accuracy comparable to traditional methods like grid search and the elbow method. In some cases, it even delivered slightly better results. However, the increased computational cost and longer runtime highlight the tradeoffs of this method.

The experiments showed that the gradient based k-tuning algorithm works well for small to medium datasets but struggles with scalability, especially when applied to larger or more complex datasets. While grid search remains faster and more efficient for larger datasets, and the elbow method offers simplicity, this gradient based approach provides an alternative for situations where precision and adaptability are prioritized.

Overall, this research demonstrates how modern optimization techniques can be applied to traditional algorithms like KNN, offering new insights into hyperparameter tuning. While this method is not a one-size-fits-all solution, it adds another tool to the range of options available for optimizing k. Future work could focus on improving the scalability of this method and exploring its effectiveness in other applications.

References

[1] BM, "What is the k-nearest neighbors algorithm? | IBM," *www.ibm.com*, 2023. https://www.ibm.com/topics/knn

[2] GeeksforGeeks, "K-Nearest Neighbours - GeeksforGeeks," *GeeksforGeeks*, Nov. 13, 2018. https://www.geeksforgeeks.org/k-nearest-neighbours/

[3] P. Miesle, "What is the K-Nearest Neighbors (KNN) Algorithm?," *DataStax*, Sep. 06, 2024. https://www.datastax.com/guides/what-is-k-nearest-neighbors-knn-algorithm (accessed Sep. 19, 2024).

[4] P. Wilkinson, "Bias and Variance in Machine Learning - Towards Data Science," *Medium*, Jan. 06, 2022. https://towardsdatascience.com/bias-and-variance-for-machine-learning-in-3-minutes-4e5770e4bf1b#:~:text=Variance%20in%20machine%20learning%20models%20depends%20on%20the (accessed Sep. 19, 2024).

[5] A. Shafi, "KNN Classification Tutorial using Sklearn Python," *www.datacamp.com*, Feb. 2023. https://www.datacamp.com/tutorial/k-nearest-neighbor-classification-scikit-learn

[6] Maulana, I., & Roestam, R. . (2024). Optimizing KNN Algorithm Using Elbow Method for Predicting Voter Participation Using Fixed Voter List Data (DPT). *Jurnal Sosial Teknologi*, *4*(7), 441–451. https://doi.org/10.59188/jurnalsostech.v4i7.1308

[7] KNN Optimization Using Grid Search Algorithm for Preeclampsia Imbalance Class Sukamto, Hadiyanto, Kurnianingsih E3S Web Conf. 448 02057 (2023) DOI: 10.1051/e3sconf/202344802057

[8] *Packt-cdn.com*, 2024. https://static.packt-cdn.com/products/9781787121515/graphics/bd978c4c-8251-489d-bcda-5ce7b7b825dd.png (accessed Nov. 26, 2024).