Predicting the Weather in Siberia

CS 4641 – Machine Learning

19 April 2020

Introduction to the dataset

The dataset I chose consists of weather data measured in Siberia over a span of two years, posted on Kaggle.com, an online community for data scientists. Each row in the data frame lists the following for the specified recording: water level, precipitation, temperature, humidity, visibility, wind level, fire level, weather type, pressure, change in water level, change in temperature, and change in pressure. The attribute I wanted to focus on is temperature; I used regression algorithms to predict the temperature at a given instance based on historic weather data.

In order to get the best prediction results, I had to preprocess the dataset to minimize any unwanted biases. The "weather" column in the dataset classified each instance using nine options (cloudy, clear, stormy, etc.). Since the options were ordered from 0 to 8 with no apparent correlation on how the numbers were assigned, the regression algorithms did not perform very well as the column somewhat counteracted the other data. I tried using one-hot encoding to change the column into a binary representation of itself, but it only slightly improved the regression scores. Therefore, I decided to remove the column, as the regression scores vastly improved without it and the rest of the data is all quantitative rather than qualitative. I used the rest of the data normally in my algorithms.

As mentioned previously, the underlying supervised learning problem that I am trying to solve is to use regression to predict the temperature at a given instance based on historic weather data. I feel that this is important because weather plays a huge impact in our lives, as everyone relies on weather predictions to dictate our activities. Using machine learning is a common tactic to predict weather in the near future, so I was curious to see if I could predict the temperature of a given day. To do this, I used three algorithms: Kernel Ridge Regression (using a non-linear kernel), *k*-Neighbors Regression, and a Neural Network (using at least 2 hidden layers). I ran these algorithms using the scikit-learn machine learning library for Python and used Matplotlib's plotting library to visually represent the differences between the three algorithms.

Description of the algorithms

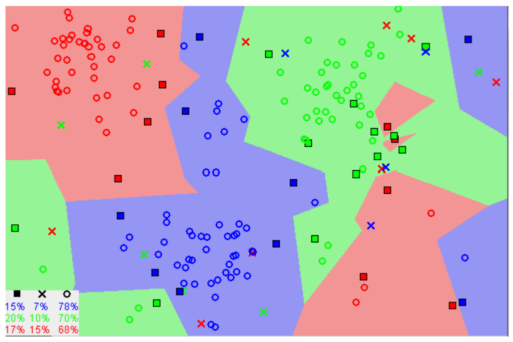
A) Kernel Ridge Regression

The first algorithm I implemented to predict temperature was Kernel Ridge Regression using the "sklearn.kernel_ridge" package. This algorithm works by using a combination of least squares and l2-norm regularization to learn a linear or non-linear function, depending on whether a linear or non-linear kernel was used, respectively. In this case, I used a non-linear kernel.

The three hyperparameters I tuned were alpha, gamma, and the kernel type. This was an important step because not only would using the default hyperparameters use a linear kernel, but various hyperparameters yielded vastly different scores. The other parameters (degree, zero coefficient) were specific to a kernel and ignored by the rest, so I decided to leave them at their default values.

B) k-Neighbors Regression

The second algorithm I implemented to predict temperature was *k*-Neighbors Regression using the "sklearn.neighbors" package. This supervised learning algorithm works by picking a number of neighbors (the *k* value) and calculating the distance between a query point and the example point while looking only at the *k* closest query points to the example point. This works because the *k*-Neighbors algorithm assumes that similar data points are closest to each other in a 2D plane; therefore, the algorithm is used to find these closest points. Shown below is an example of this phenomenon:



The three hyperparameters I tuned were the number of neighbors (*k* value), weights, and power (*p*) parameter. As mentioned in the previous paragraph, using a reasonable *k* value is imperative to having a successful *k*-Neighbors algorithm because having too low of a value will generate inconsistent results, while having too high of a value will cause outliers to skew the data. The weight hyperparameter defines how the distances measured will be weighed during prediction, and the *p* parameter chooses the metric used for the distance calculations (either Manhattan or Euclidean).

C) Neural Network

The third and final algorithm I implemented to predict temperature was a Neural Network using the "sklearn.neural_network" package, specifically the MLPRegressor. This algorithm works by using layers of perceptrons (algorithms that take in multiple inputs, multiplies them by a weight, and passes them through an activation function to generate an output) to create a Neural Network. These perceptrons are the key to a functioning Neural Network; they are tuned as part of the hyperparameters of the Neural Network algorithm.

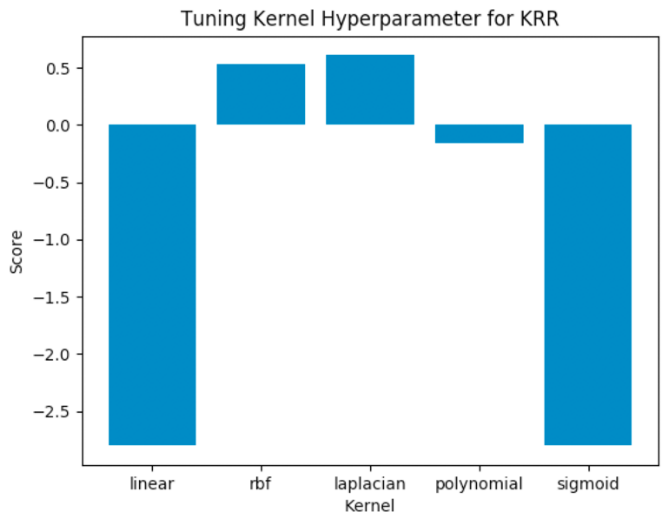
The two hyperparameters I tuned were alpha and hidden layer size. For the number of hidden layers, I consistently used a size of 3, while I changed the number of neurons in the layer. As with the Kernel Ridge Regression algorithm, the alpha value was important to act as the regularization parameter, so I made sure to test with a variety of values. I tried changing some of the other hyperparameters when testing, but they all either worked best at their default values or did not affect the performance in a significant way.

Tuning Hyperparameters

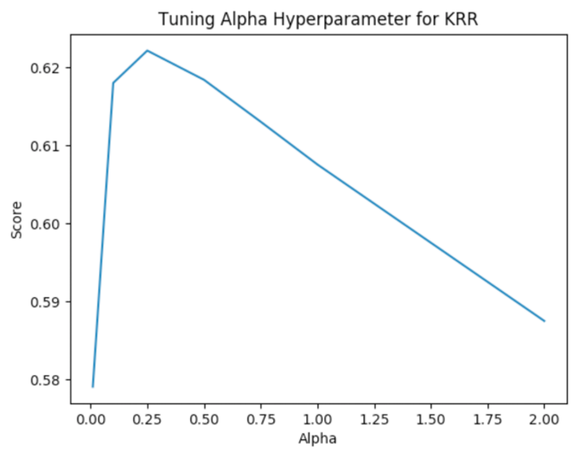
One of the requirements for this project was that the data must have a non-trivial distribution. While the Kernel Ridge algorithm proved this by having a better score with a non-linear kernel versus a linear kernel, I decided to run Linear Regression on the dataset to test its performance against the other algorithms. Running Linear Regression returned a score of 0.3911, which is much worse than the other algorithms. Therefore, this indicates that the dataset I chose has a non-trivial distribution.

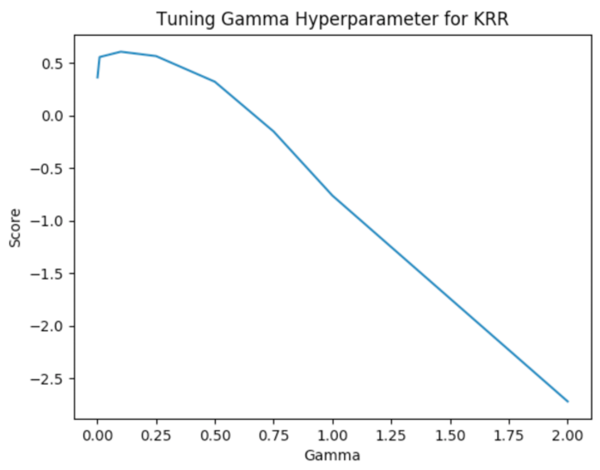
A) Kernel Ridge Regression

The three hyperparameters that I thought were most significant for the performance of Kernel Ridge Regression were alpha, gamma, and the kernel type. There were five different kernel options I could choose from for my dataset: linear, RBF (radial basis function), Laplacian, polynomial, and sigmoid. As previously mentioned, I chose data that would be non-trivial; therefore, the linear kernel performed the worst out of the five options. Because the default kernel for Kernel Ridge Regression is the linear kernel, I had to find the best kernel first before testing for the other hyperparameters to make sure that I got the most accurate alpha and gamma hyperparameters. The Laplacian kernel ended up performing the best.



For the other two hyperparameters (alpha and gamma), I tested a range of values from 0.001 to 2. From the graphs generated, I noticed that the alpha value was the best around 0.25, while the gamma value was the best around 0.1; the scores dropped significantly with smaller and larger values.

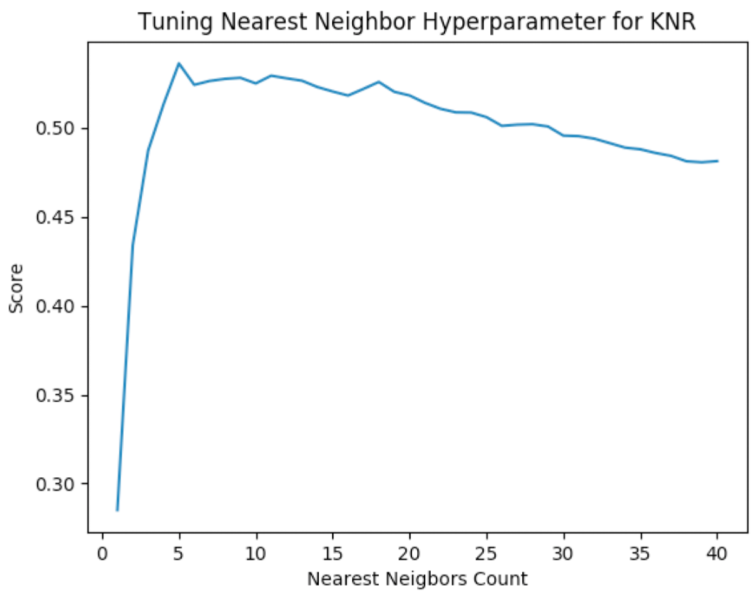




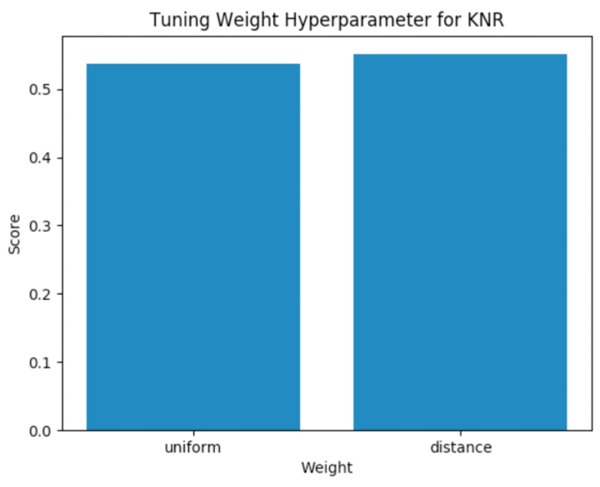
In order to make sure I got the best combination of hyperparameters for the Kernel Ridge Regression algorithm, I used GridSearchCV to tune the hyperparameters, which provides the best set of hyperparameters to run regression. After using GridSearchCV, the best hyperparameters I got were an alpha score of 0.25, a gamma score of 0.1, and a Laplacian kernel. Using these hyperparameters yielded a final score of 0.6221 for Kernel Ridge Regression. The total computation time to tune the hyperparameters and train the data was 57.6279 seconds.

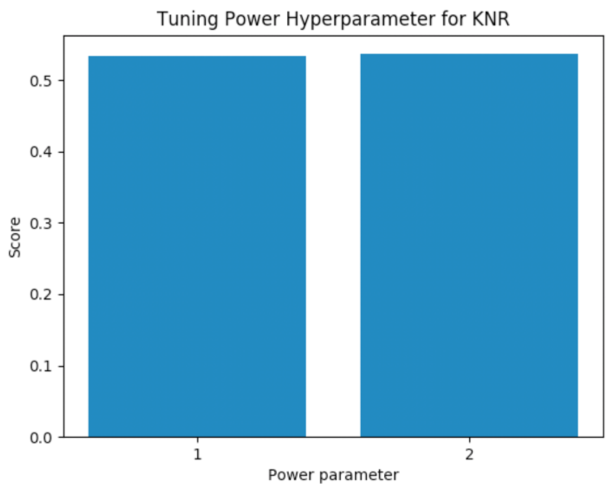
B) *k-Neighbors Regression*

The three hyperparameters that I thought were the most significant were the number of nearest neighbors (*k*), weights, and power (*p*) parameter. The number of nearest neighbors was easily the most significant hyperparameter, as using different k values vastly changed the performance of the algorithm. As previously mentioned, using too few nearest neighbors did not analyze enough data, which resulted in inaccurate performance, while using too many nearest neighbors caused outliers to diminish performance. The data I collected reflected, as the best number of nearest neighbors seemed to be around 5-10 when testing the performance of the algorithm with 1-40 nearest neighbors.



The other two hyperparameters I tuned were the weight (either uniform or distance) and the power parameter, which decided whether to use Manhattan distance or Euclidean distance to calculate distance (this was designated by a 1 or 2 for the *p* value, respectively). These two hyperparameters barely affected the performance of the algorithm on the dataset but tuning them still slightly helped the algorithm get a better score.

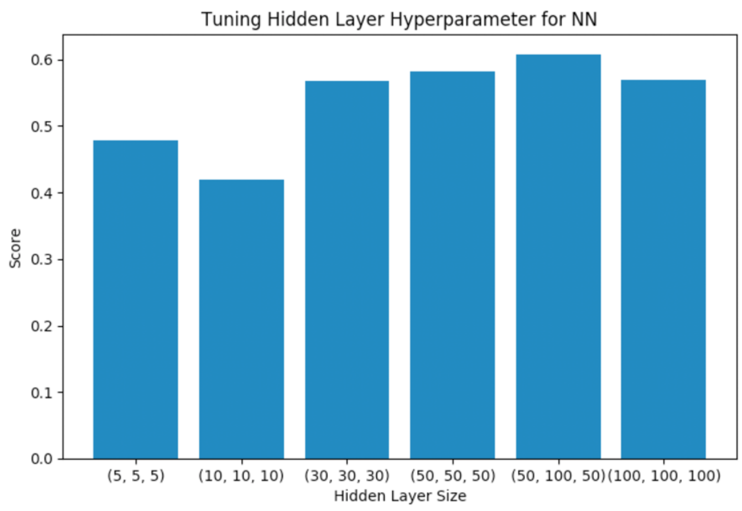


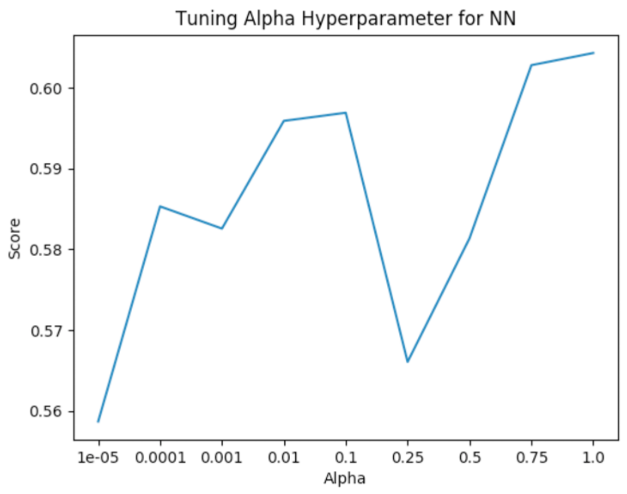


As with the Kernel-Ridge Regression algorithm, I used GridSearchCV to tune the three hyperparameters for *k*-Neighbors Regression. With the values I tested, the algorithm performed best when it used 6 nearest neighbors, used a distance weight function, and set the power parameter to 1. Using these hyperparameters yielded a final score of 0.5623 for *k*-Neighbors Regression. The total computation time to tune the hyperparameters and train the data was 4.1599 seconds.

C) Neural Network

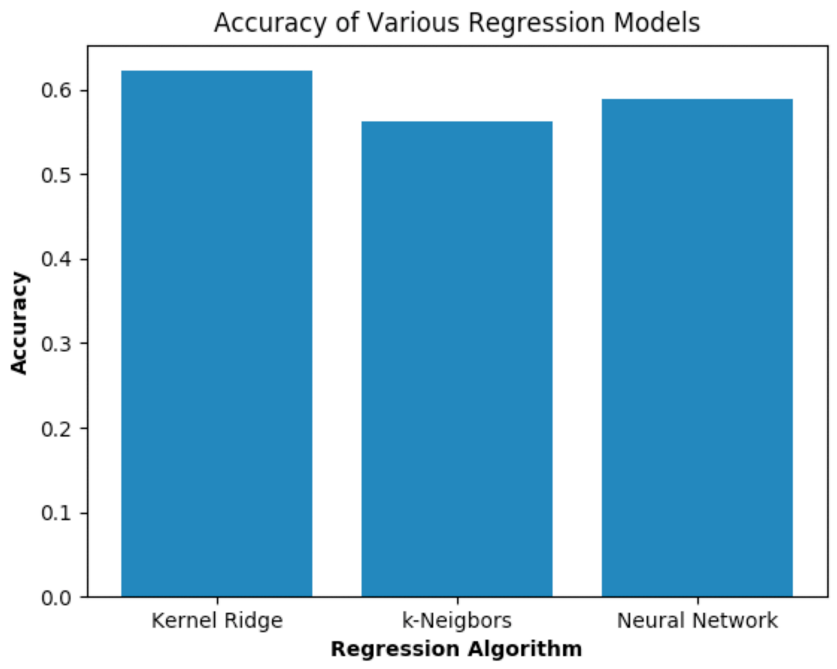
The three hyperparameters that I thought were the most significant were alpha and the hidden layer size. I felt that these hyperparameters were the most difficult to tune because my Neural Network never converged to my dataset no matter which hyperparameters I used, so the values for the best hyperparameters slightly varied. However, I was able to generate a range of values that worked best with the MLPRegressor. I observed that 3 layers of around 30-50 neurons each worked the best, while the best alpha value was around 0.75 to 1.





As with the other two algorithms, I used GridSearchCV to tune the two hyperparameters for the Neural Network. With the values I tested, the algorithm performed best when it used 3 hidden layers of 50 neurons each and had an alpha value of 0.75. Using these hyperparameters yielded a final score of 0.5889 for the Neural Network using MLPRegressor. The total computation time to tune the hyperparameters and train the data was 408.3008 seconds, or 6.8 minutes.

Comparing algorithm performance



After tuning all the hyperparameters for the three Regression algorithms, I ran the three algorithms against an unseen test dataset. I got a score of 0.6221 with Kernel Ridge Regression, 0.5623 with *k*-Neighbors Regression, and 0.5889 with a Neural Network using MLPRegressor. Kernel-Ridge Regression took 57.6279 seconds to run, *k*-Neighbors Regression 4.1599 seconds to run, and the Neural Network took 408.3008 seconds to run. In order to quantify the quality of my predictions, I used cross-validation on the models, which splits the training set into smaller folds. To compare my results to this, I split the training set into ten folds, ran cross-validation on them, and took the standard deviation of the scores to check the general accuracy on how the models perform. With this, I got standard deviation values of 0.0555 with Kernel Ridge Regression, 0.1011 with *k*-Neighbors Regression, and 0.1220 with the Neural Network.

I believe that the Neural Network easily is the worst overall of the three algorithms on the dataset. While it had the second-best score, it simply took way long to train the data (almost 7 minutes). Also, the optimization never converged even when testing with 1000 maximum iterations, which would take even longer to run. Therefore, the results would be inconsistent as GridSearchCV could never make a definitive choice in choosing the hyperparameters. While *k*-Neighbors got the worst score out of the three algorithms, I feel that it performs better overall than the Neural Network. The accuracy improvement of the Neural Network compared to *k*-Neighbors is less than 5%, while the speed difference between *k*-Neighbors and the Neural Network is almost 100%. *k*-Neighbors is a relatively simple algorithm that works best with simple and continuous data, so it makes sense that it would work well. Kernel Ridge Regression performed the best overall out of the three algorithms on this dataset. While it took about a minute to run tune the hyperparameters and execute the algorithm, it consistently performed better than the other two algorithms. This may have been because unlike the other two algorithms, I was able to fine tune all of its hyperparameters with a wide variety of values with GridSearchCV, which allowed me to get higher accuracy rates.

Conclusion

Regression Model	**Training Time (seconds)**	**Number of Hyperparameters**	**Final Accuracy Score with Tuned Parameters**
Kernel Ridge	57.6279	3	0.6221
k-Neighbors	4.1599	3	0.5623
Neural Network	408.3008	2	0.5889

If I were to use one of these algorithms in practice on real-world data, I would use the Kernel Ridge Regression algorithm. It performs well with both trivial and non-trivial data depending on the kernel passed in. When the best kernel to use is chosen, all that is needed is to tune the alpha and gamma values, which results in hyperparameters that consistently output a relatively high accuracy score compared to the other algorithms. Kernel Ridge Regression also does not take too long to tune and train, which makes it efficient for practical uses. I believe that the *k*-Neighbors Regression algorithm is the next best algorithm to use for regression. While it performed the worst out of the three algorithms, it was by far the simplest to use, as the main hyperparameter required to tune was the number of nearest neighbors; the other two hyperparameters helped with accuracy but not significantly. It also ran much faster than the other two algorithms, making it very efficient to use in practice. Finally, I believe that a Neural Network would be the worst to use in practice. While similar to *k*-Neighbors Regression in that only one hyperparameter significantly matters (number of hidden layers and neurons), it performs incredibly inconsistently and frequently picks different numbers of neurons to use as a hyperparameter each run. Also, it took significantly longer than the other two algorithms to tune and train, which makes it highly inefficient given that it did not even perform the best. Given the findings of this exercise, I believe that Kernel Ridge Regression performs the best overall on this dataset and I would use it for practical use on real-world data.

Acknowledgments

Cross Validation Score
https://scikit-learn.org/stable/modules/model_evaluation.html
https://scikit-learn.org/stable/modules/cross_validation.html
https://scikit-learn.org/stable/modules/generated/sklearn.model_selection.cross_val_score.html
https://towardsdatascience.com/train-test-split-and-cross-validation-in-python-80b61beca4b6

Dataset
https://www.kaggle.com/dwdkills/weather

GridSearchCV
https://scikit-learn.org/stable/modules/grid_search.html
https://scikit-learn.org/stable/modules/generated/sklearn.model_selection.GridSearchCV.html

Hyperparameter Tuning
https://towardsdatascience.com/hyperparameter-tuning-the-random-forest-in-python-using-scikit-learn-28d2aa77dd74

Kernel-Ridge
https://scikit-learn.org/stable/modules/kernel_ridge.html
https://scikit-learn.org/stable/modules/generated/sklearn.kernel_ridge.KernelRidge.html
https://scikit-learn.org/stable/auto_examples/plot_kernel_ridge_regression.html

k-Neighbors
https://www.saedsayad.com/k_nearest_neighbors_reg.htm
https://scikit-learn.org/stable/modules/generated/sklearn.neighbors.KNeighborsRegressor.html
https://towardsdatascience.com/machine-learning-basics-with-the-k-nearest-neighbors-algorithm-6a6e71d01761

Matplotlib
https://matplotlib.org/3.2.1/api/_as_gen/matplotlib.pyplot.plot.html

Neural Networks/MLP Regressor
https://www.springboard.com/blog/beginners-guide-neural-network-in-python-scikit-learn-0-18/
https://scikit-learn.org/stable/modules/generated/sklearn.neural_network.MLPRegressor.html