Predictive Modeling for Drunk Driving Incidents

Hongli Peng

May 10th, 2023

1 Introduction

In this report, we focused on predictive modeling, **Linear Model**, **Random Forest**, **Neural Network**, and **Transferred Learning**. Also, we discussed how to construct a **Choropleth Map** for United States based on a specific feature. The purpose for these models is to predict the percentage of fatal accidents involving drunk driving across 50 United States. We would perform the models and evaluate their performance in this report.

2 Data Overview

There are two datasets in our report, 'censustractsdatasetċsv' and 'raw_state_data_drunk_drivingċsv'. The target variable for two datasets are **DRUNK_DRIVING_PERCENTAGE** and **DRUNK_DRIVING_PERCENTAGE_2** respectively. The number of meaning features is 22.

3 Linear Model

We started with census-tracts-dataset.csv and performed the models with this dataset. We have 20,000 observations, 22 features, and 1 target variables. This can be achieved by removing two columns 'Unnamed: 0.1', 'Unnamed: 0'.

3.1 Build A Standard Linear Model

We built a standard liner model in this section. Firstly, we normalized the features data and split 30% of the data to testing set. Then we trained the linear model with train data and then predicted the test data . We would use mean square error (MSE) to evaluate the model performance. The MSE in the train set is 52.306692 while the MSE in the test set is lowest with 51.972846. Usually, we may expect a lower training MSE. The reason for this higher training MSE may be the model is not well trained, which is an indication of underfitting.

3.2 Histogram of True and Predicted Data

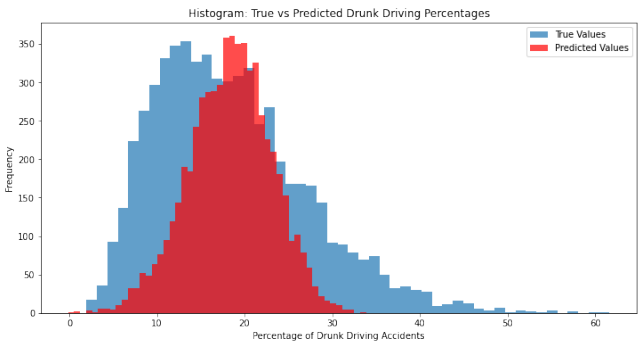

Figure 1: Histogram: True vs Predicted Drunk Driving Percentages

We plotted a histogram to compare the true values and predicted values of the percentage of fatal accidents involving drunk driving. From Figure 1, the predicted drunk driving percentage has a higher frequency around 15-20 percentage, while the true data is around 10-20. Also, there is a right skewed for the true data, while predicted data looks more normal, which means that we might lose some accuracy. But overall, the shape of predicted data is similar to the shape of the true data, which means that the predicted data is close to the test data. And the shape of predicted data is **narrower**.

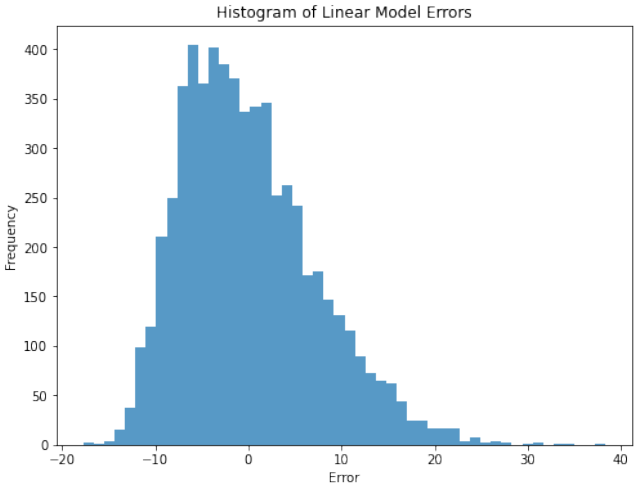

Figure 2: Histogram of Linear Model Errors

We also plotted the histogram of the linear model errors. From Figure 2, the distribution of the error looks normal with a slightly right skewed tail. And most of errors are between (-10, 10) and the histogram of the error seems to center at 0. Therefore, this linear model is reasonably accurate at some degree, but there are still some cases where the predictions are off by more than 10 percentage points. The slightly right-skewed tail suggests that there are some cases where the model is over-predicting the values, but it is not many of them.

3.3 L1 and L2 Regularization

Discussion of L1 and L2 could be found from pages 64-80 in lecture 6 logistic regression. For L1 regularization, we used lasso regression. For L2 regularization, we used ridge regression. These are two common regularization methods dealing with overfitting problem by adding the penatly to the loss function.

L1 adds a penalty equal to the absolute value of the magnitude of coefficients, while L2 adds a penalty equal to the square of the magnitude of coefficients. In general, when w_i is small, L2 regression stops paying attention to it. Weights are almost never completely eliminated. With L1 regression, we care about reducing big and small weights equally. So if a feature is useless, L1 will reduce it all the way to 0. Therefore, L1 encourages **feature selection and sparsity**, and L2 tends to tackle **multi-collinearity**.

We utilized the **GridSearchCV()** to find the optimal hyperparameter alpha for both lasso and ridge regression, and we choosed cv=5 to have 5 fold cross-validation. The one that yields the minimal MSE would be the the optimal alpha.
From table 1, we have the MSE for L1 and L2 with their optimal alpha. From this result, we did not see a substantial improvement of model performance because their MSE are really close to each other.

Method	MSE
L1 Lasso (Optimal Alpha = 0.00880)	51.982462
L2 Ridge (Optimal Alpha = 56.56206)	51.975972
Without Regularization	51.972846

Table 1: Comparison of Mean Squared Error (MSE) for Different Regression Methods

3.4 SGDregressor

In this section, we talked about how to perform Stochastic Gradient Descent method for linear regression model using **hyperparameter tuning**. The SGDRegressor algorithm iteratively optimizes the model's parameters by processing data in small sets. The model's parameters are adjusted based on the slope of the loss function relative to these parameters. There are several hyperparameters we adopted.

- 'loss': loss function such as 'squared_error' and 'huber'
- 'penalty': L1 and L2 regularization
- 'learning_rate': 'constant', 'optimal', 'adpative' and 'invscaling'
- 'alpha': a constant that multiplies the regularization term. The higher the value, the stronger the regularization, such as 0.001, 0.01, 0.1 and 1

Finally, we used GridSerachcv() with 6 cross-validation method and derived the optiaml hyperparameters alpha: 0.001, learning_rate: adaptive, loss: squared_error, penalty: l1. The goal of the **adaptive** learning rate is to decrease the learning rate as we get closer to the optimal solution, which can help the model converge faster and more accurately. We got MSE of the test set 51.97156, which is the smallest among all linear models above. Even though the MSE did not decrease a lot, this SGDregressor using GridSearchcv() is useful to enhance the model improvement by finding the optimal hyperparameters.

4 Random Forest

In this section, we would utilize the random forest model to predict the outcome variable. From pages 94-97 in lecture 7 random forests, random forests are an **ensemble** method that is sampling with replacement, which construct many decision trees and then average the results.

4.1 Steps of Performing the Random Forest Model with Hyperparameters Tuning

1. We firstly applied Random Hyperparameter Grid using RandomizedSearchCV(), which can narrow down the range for each hyperparameter.

1.1 Explained some popular or important hyperparameters.

- n_estimators: how many decision trees we want to build
- max_features: max number of features considered for splitting a node in a tree. Usually, we considered $\sqrt(p)$ where p is total number of features, while auto means the total number of features p.
- max_depth: max number of levels in each decision tree
- min_samples_split: minimum number of samples required to split an internal node
- min_samples_leaf: minimum number of samples required to be at a leaf node
- bootstrap: a method for sampling data points (with or without replacement)

Leaf nodes represent endpoints or outcomes, while internal nodes represent decision points or hypothetical ancestors.

1.2 Built the grid for these hyperparameters. For each parameter, we would consider every combination of each element. However, However, for the random search method, we were not looping over every combination due to the expensive computational cost. Hence, we just selected some combinations at **random**.

1.3 We set the n_iter = 100, which the number of different random combination is 100. The cv=5 means that we have 5 folds for cross validation. Increasing the number of folds may better avoid the overfitting problems.

2. Used GridSerachCV(). After narrowing down the range of hyperparameters using random search, We could adopt GridSerachCV() to loop through every combination for the narrow range. Then we could get the optimal hypeparameters. Finally, we could use it to train our random forest model.

4.2 Model Comparison of With and Without the Optimal Hyperparameters

From table 2, we could conclude that by comparing with the train and test MSE for both random forest models with and without the optimal hyperparameters, the MSE of test data for the random forest model with the optimal hyperparameters is lower than that without them. We reached an **improvement** by setting the optimal parameters. The train MSE with optimal hyperparameters is higher than that without optimal hyperparameters. This may indicate an **overfitting** problem that the original model did a good job on train set but performed poorly on test set. Overall, the two test MSEs were very close to each other and the improvement was not substantial. Hence, tuning the optimal hyperarameters only helped a little bit.

Model	**Train MSE**	**Test MSE**
Best Random Forest Model	14.494979	48.171884
Original Random Forest Model	6.925702	48.374571

Table 2: Comparison of Mean Squared Error (MSE) for the Best and Original Random Forest Models

4.3 Histograms of True and Predicted Data in Random Forest

By comparing the histograms and density plots of forest outputs and the true data, the true data has a wider range than the predicted values. And the predicted values has a higher density around 15-20 percentage of Drunk driving accidents.

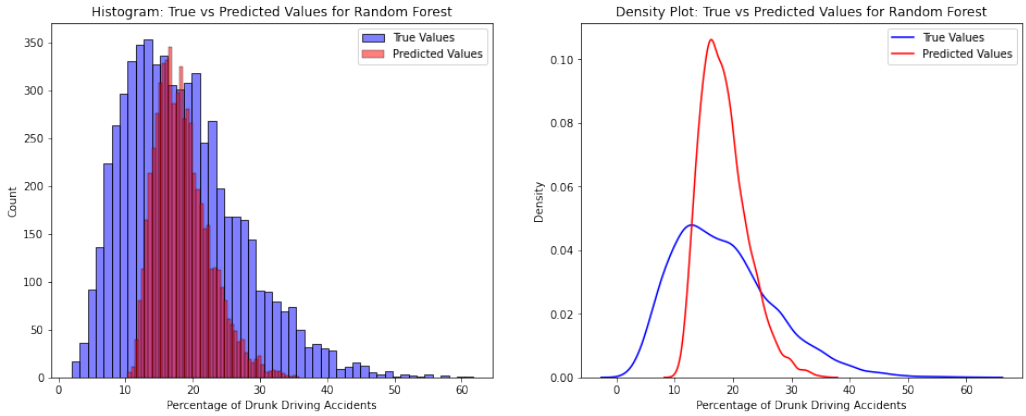

Figure 3: Histogram of True and Predicted Data in Random Forest

4.4 Feature Importance Selection

Random forest model can also be used for feature importance selection. During the training process, each decision tree in the random forest makes splits on features that most effectively separate the classes or reduce error for regression. These criterion could be **'Gini impurity'** or **'information gain'** such as cross entropy for classification, and MSE for regression. The more often a feature is being selected to make decisions, the higher its importance. The first five most important feature we found are stored in table 3. With these features importance information, researchers can train the model and pay more attention on the important features, which can reduce the complexity of the model. Also, they can create new features with a combination of the important features. What's more, they can collect more data for this important features for training because finding the data for all features together is a difficult task.

Feature	**Importance**
MEDIAN_INCOME	0.13615
BLACK_ALONE	0.09714
TOTAL_ACCIDENTS	0.09354
WHITE_ALONE	0.05615
DRUG_OVERDOSE_DEATH_RATE	0.05020

Table 3: Feature Importance Ranking from Random Forest Model

5 Neural Networks

In this section, we conducted Neural Networks using **PyTorch**. We mainly focused on 3 layer Network and evaluate the model performance with MSE.

5.1 3 Layers Networks with Sigmoid Activation

Firstly, we imported the necessary useful modules.

- **'torch'** imports the PyTorch library.
- **'nn'** module contains various neural network layers and functions
- **'TensorDataset'** and **'DataLoader'** create datasets and data loaders for training and testing the model

We implemented the model with following steps.

1. Converted the training and testing data into a PyTorch tensor using the torch.tensor() function
2. Create a TensorDataset from the tensors and then a DataLoader for easy batch processing
    - The TensorDataset combined the input data X_train_tensor and output data y_train_tensor into a single dataset that can be iterated over during training.
    - The DataLoader allowed for efficient loading and processing of the data in batches during training. The batch_size specified the number of samples in each batch, and shuffle=True specifies that the data should be shuffled at the beginning of each epoch to prevent overfitting and improve the training process. We set the batch_size = 100 at first.
3. Defined the sequentail model and loss function
    - The **'Sequential()'** from 'nn' allowed for a sequence of layers to be defined in order, with each layer passing its output to the next layer in the sequence. 3 layers with the order in this case are input layer (22 * 64), hidden layer (64 * 64) and the output layer (64 * 1)

- **'nn.MSELoss()'** defined the loss function to be MSE
- **'torch.optim.Adam()'** defined the optimizer to be **'Adam optimizer'**. The optimizer was to update the weights of the neural network during training in order to minimize the loss function MSE. It utilized SGD that computes adaptive learning rate for each parameter.
- **'model.parameters()'** returned the parameters such as the weights w_j and biases b_j of the neural network model, which were then passed to the optimizer for updating.

4. Trained the neural network model defined above

   - **'num_epochs='5** set the 5 number of times the entire dataset will be passed forward and backward through the neural network
   - **'outputs = model(inputs)'** adopted **forward propagation** to process the input data for all layers respectively and produced an output.
   - 'loss = criterion(outputs, targets)' computed MSE for each epoch, which allows us to see how MSE decays
   - **'optimizer.zero_grad()'** cleaned the previous existing gradients of parameters before calculating the gradients of the parameters with respect to the MSE.
   - **'loss.backward()'** applied **backward propagation** to calculate the gradient of the loss with respect to the output and then used chain rule to the gradient of the loss with respect to each parameter.
   - **'optimizer.step()' updated** the model's parameters after backward pass. The optimizer uses an Adam optimization to adjust each parameter slightly in the direction that reduces the loss.

5. Evaluated the model performance

   - **'model.eval()'** set the model to **evaluation mode**
   - **'torch.no_grad()'** built no computational graph
   - **'model(X_test_tensor) '** predicts the outputs for the test data. No computational graph is constructed here, and no gradients will be calculated in backpropagation based on 'model.eval()' and 'torch.no_grad()'
   - **'mse_nn.item()'** extracted the MSE for the test data using this trained neural network model. Using **'.item()'** to convert the MSE from PyTorch tensor to a regular **Python float**

With the steps above, the final MSE of the test data for our neural network is 62.6671. And the MSE for the train data is 62.01337.

5.2 Visualization and Hyperparameters Tuning for the Neural Network

From Figure 4 the histograms plot for the neural network model above, the predictions *converged to the mean output*, that is around 20 percentage of drunk driving accidents. Therefore, we considered tuning the hyperparameters to solve this problem and enhance the model performance. There were several hyperparameters I considered to adjsut.

- Adjusted the **'batch_size'** from 100 to 32
- Added more neurons for each layer such as 128 for the hidden layer
- Changed the activation function to **'ReLU()'** that is less prone to the vanishing gradient problem and might help the model learn better.

- Adjusted to a **smaller learning rate** 0.001 to take smaller steps when updating the weights during training, which prevents the model converge too quickly to get stuck in a suboptimal solution. Taking a smaller step would take more time to converge but it is better for finding an optimal point. The **learning rate** specifies the starting point and then is adaptively adjusted by the Adam optimizer during training based on the gradients.
- Increased training epochs from 5 to 10.

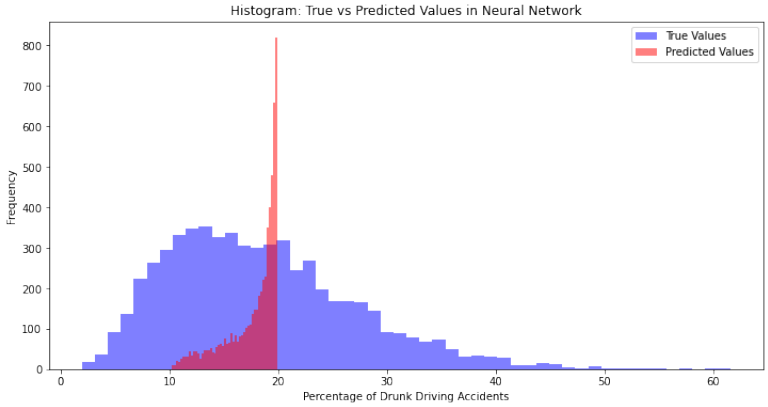

Figure 4: Histogram of True and Predicted Data in Neural Network

As a result, the test MSE and train MSE after hyperparameters tuning were 45.6291 and 42.4869, which the model performance was improved due to the lower test MSE. From Figure 6, the distribution of true and predicted data for this complex neural network, we solved the convergence issue and the plot has more interpretability that is similar with random forest and linear regression.

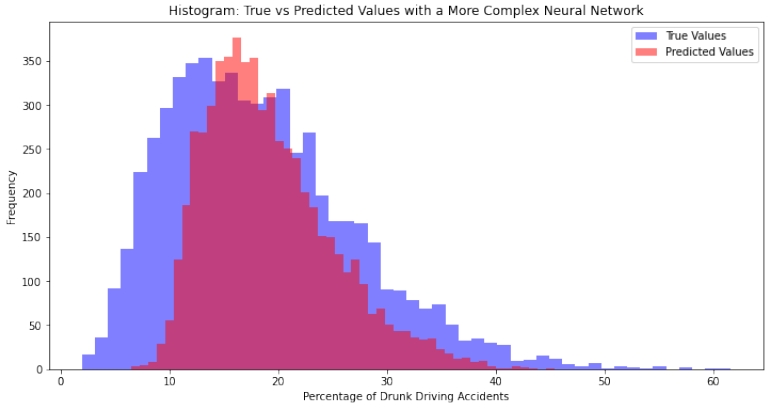

Figure 5: Histogram of True and Predicted Data in a more complex Neural Network

6 Transfer Learning

In this section, we turned our attention to **raw_state_data_drunk_driving.csv**. This is a smaller dataset that contains 50 states information. We would implement a transfer learning for this small data.

6.1 Linear, Random Forest, and Neural Network Model

We also used standard normalization for our 22 features and 1 target variable **'DRUNK_DRIVING_PERCENTAGE_2'**. Because we only had 50 data, we split 20% of the data to test set, in order to have more data to be trained.

1. For the linear model, the MSE for test data was 97.137759 and the MSE for train data was near 9.433096. This might indicate an **'overfitting'** problem that the model learned too many noise and trained well for the train set, but performed poorly for unseen data

2. For the random forest model, the MSE for test data was 60.081643 and the MSE for train data was 3.417113

3. For Neural Network, the MSE for test data was 106.67842 and the MSE for train data was 5.990657

We used MSE as our metrics. Random forest model performed the best with the smallest MSE for this small dataset. The smaller dataset tended to have higher test MSEs and lower train MSEs. This might imply an overfitting problem due to insufficient data for training the model.

6.2 Transfer Learning for the Linear Model

Transfer learning often involves using a model trained on a large dataset to initialize the weights for a new model that will be fine-tuned on a smaller related dataset. The **'sgd_grid_search()'** was already trained by SGD linear model and fitted by the larger dataset. We would apply this to predict the linear model for the smaller data. Finally, the MSE that predicted the whole small dataset was 149.631427 and the test MSE for the small dataset was 251.551168. We did not see an improvement by this transferred model due to the larger MSE.

6.3 Transfer Learning for the Neural Network Model

We firstly trained the neural network model using a large dataset. Then we used this to train our small dataset. After some hyperparameter tuning, we set the 'batch size' = 32, 'learning rate' = 0.005 and 'number of epochs' = 40 for this dataset. As a result, the MSE for test data was 86.651 and the MSE for train data was 1.813978, where both the train MSE and the test MSE decreased. Therefore, this transfer helped to predict the data in the neural network model.

7 Visualization

In this section, we constructed two Chorpleth map of the US States. The first map from Figure 6 was colored by the percentage of drunk driving accidents in that state. The Montana seemed to have the highest percentage.

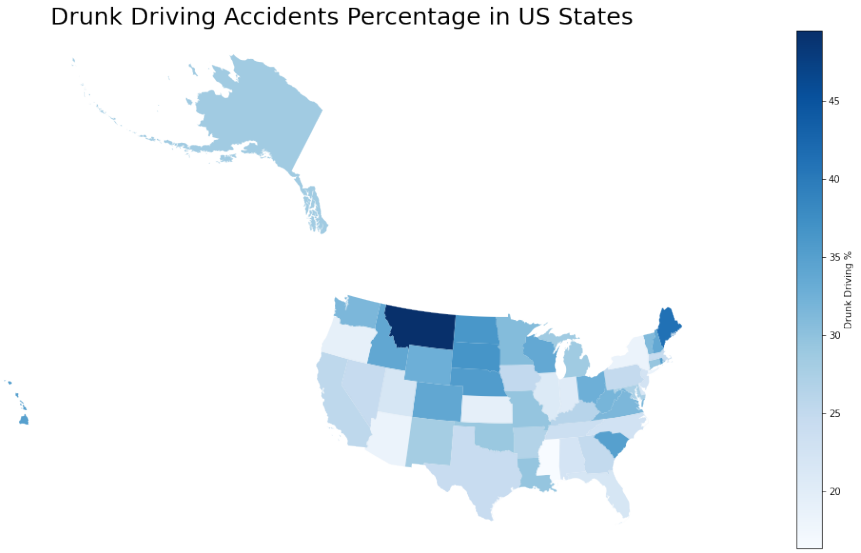

Figure 6: Drunk Driving Accidents Percentage in US States

The second map from Figure 7 was colored by the prediction errors. We picked the random forest model to calculate the prediction error. We utilized transfer learning here that used a random forest model trained by a larger dataset and then predicted the outputs using the smaller dataset. The Montana state seemed to have the highest prediction error while UTAH seemed to have the smallest prediction error. I also provided the top 5 states that had the highest and lowerest prediction errors from table 4 and table 5.

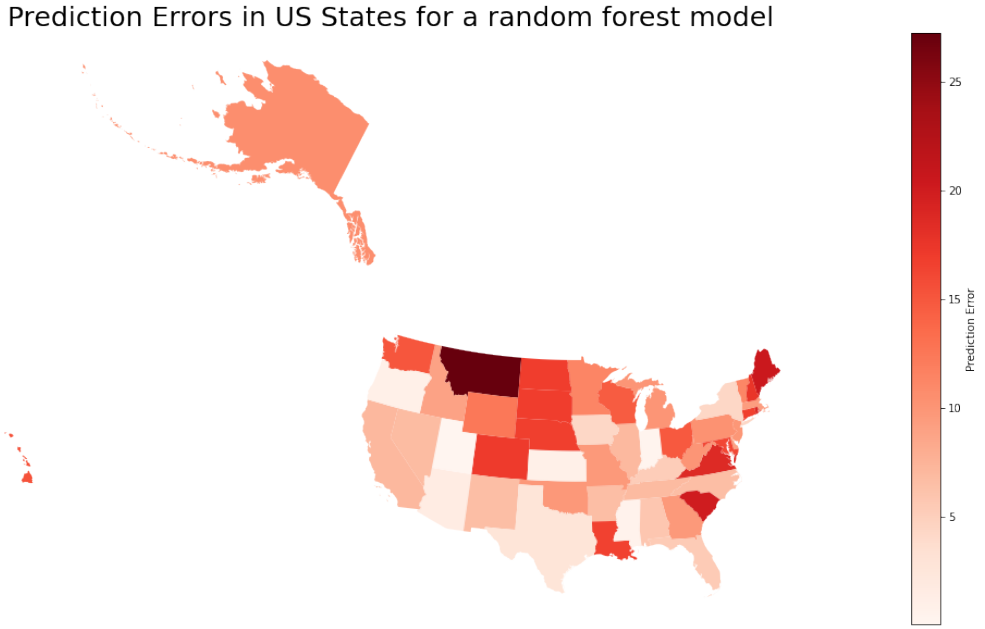

Figure 7: Prediction Errors in US States for a random forest model

STATE	True_perc	Pred_perc	Pred_error
Montana	49.480	22.266565	27.213435
Maine	41.173	20.727644	20.445356
South Carolina	35.097	15.029256	20.067744
Rhode Island	35.403	16.390717	19.012283
Virginia	31.605	12.959052	18.645948

Table 4: Top 5 States that had the highest prediction error

STATE	True_perc	Pred_perc	Pred_error
Utah	21.924	22.010797	0.086797
Indiana	20.442	20.130449	0.311551
Mississippi	16.369	15.756229	0.612771
Kansas	19.381	18.399081	0.981919
Oregon	19.326	20.326954	1.000954

Table 5: Top 5 States that had the lowerest prediction error