Final Project Report

CS 6350 Machine Learning

Ryan Dalby

21st December, 2021

Project Repository

Commit to reference for this report (see ClassProject directory):

`https://github.com/dalbyryan3/cs-6350-machine-learning/tree/c57639a822e018ff0da16a81af52e862db3c8f32/ClassProject`

Problem Definition and Motivation

For my project I chose to do the competitive Kaggle project. This project involves predicting if a person makes over 50k a year using data from the 1994 census. I chose this project because I thought it would be interesting to compete in a Kaggle competition as well as have a chance to apply the algorithms learned in this class to a real-world problem. This problem interestingly incorporates both categorical and numerical features as well as a fairly large set of 25000 samples (as well as 5000 testing samples) where some features/attributes are missing.

Machine learning is a viable way to approach creating a model for predicting if a persons income is over 50k a year because of the fairly large set of labelled data and the task being a binary classification task, a common machine learning problem. This machine learning problem can formally be considered supervised binary classification of tabular data. For this project I primarily used scikit-learn [1], a Python library with implementations of common machine learning algorithms, many of which I have implemented or learned about in this class.

In order to create a model that could accurately predict income my initial steps involved data pre-processing to get the data in an easily trainable form. I then found value in normalizing the feature values before training a model. Finally, I experimented with and analyzed various machine learning algorithms including support vector machines, logistic regression, fully-connected neural networks (also known as Multi-layer Perception (MLP)), and histogram based gradient boosted trees. I will discuss the results of using these models and some possible future steps to put this model into practice.

One last key note, I found that I had been submitting the predicted values for the Kaggle competition incorrectly as I was not submitting probability-like values for which the AUC score is generally calculated from but rather just the predicted label (a 0 or 1). Once submitting the values as a probability-like value (utilizing a sigmoid of a score value or just directly using a non-rounded probabilistic output (depending on model)) I got much better results in terms of AUC score (even with the same models as the midterm report).

Data Pre-processing

My first step of this project was to get familiar with the data, this was the primary focus of my project up to the midterm project report. I will discuss some of the midterm data pre-processing results as the same data pre-processing was used for the final report. The training data I was provided with contains approximately 25000 labelled examples and 14 features made up of 8 categorical features and 6 numerical features. The first pre-processing task was to convert the 8 categorical features to numerical features that could be used with many common scikit-learn algorithms. There are many ways to encode discrete class values as numerical ones. The most straight forward way is called ordinal encoding or label encoding, this involves simply mapping a given class to an integer. This technique is simple but introduces undesirable artificial ordering into the data [2].

I chose to utilize one-hot encoding, a simple technique that overcomes the artificial ordering problems of label encoding [2]. This technique maps a given class of a feature to a unique binary vector representing the class. In practice this expands the number of features since every categorical feature becomes represented by new features representing all of the possible values it can take on. For this machine learning task, one-hot encoding expanded the number of features from 14 to 108.

In my midterm report I also found the great benefit of normalizing the input data (transforming the values of each feature to be of zero-mean and unit-variance) as it improved model performance and convergence by allowing the model to better "focus" on the data relationships rather than the relative scale of the data values.

Finally with my one-hot encoded and normalized data, I randomly split the data into training and test sets using an 80% training, 20% test split. Some other things to note about this split are that I made sure to keep the test data completely isolated

Models and Results

After data pre-processing I was left with a training and test set that I could use to train various machine learning models. I will discuss the models and their results and analyze the models primarily by means of learning curves. Learning curves give a good indication of the bias and variance of a hypothesis by training on varying sizes of data and observing how the score/accuracy changes [3].

Support Vector Machine Classification

In the midterm report I had investigated using support vector machine based classification. Support vector classification (SVC) has an objective of maximizing the margin between a decision hyperplane and examples [4]. The objective can be stated as

$$\min_{w,b,\zeta} \frac{1}{2} w^T w + C \sum_{i=1}^{n} \zeta_i \quad \text{Subject to: } y_i(w^T \phi(x_i) + b) \geq 1 - \zeta_i, \;\; \zeta_i \geq 0, i = 1, ..., n$$

where w is the weight vector, C is a regularization constant (inverse regularization parameter), ζ_i controls distance from boundary, b is a bias term, y_i is a vector of labels and x_i is an example. A kernel can be used to transform the data to be linear, a kernel is defined by $\phi(x_i)$ as $K(x_i, x_j) = \phi(x_i)^T \phi(x_j)$.

SVC with non-linear kernels can be used to learn non-linear relationships in data by transforming the data in a way that it can appear as linear to the SVC classifier. In the midterm report, I trained an SVC with a linear kernel and found results equivalent or worse than logistic regression so I moved

onto using a polynomial kernel where I got no better results. I then tried a radial basis kernel (rbf) and got comparable but slightly better results than logistic regression. The results for SVC with an rbf kernel can be seen in Table 1 where a test AUC of 0.880 was achieved (the AUC values are correctly displayed when compared to the midterm report because I used probability-like outputs to determine the AUC score).

Looking at the learning curves for this model it is clear that this model overfits the training data to a higher degree than the logistic regression model. This can be seen in Figure 1 where the learning curve shows a high variance as evidenced by the gap between training and validation score. I experimented with different regularization values but found that the test AUC score did not change much, getting worse with less or more regularization. I did not attempt to create a bagged SVM classifier because of the large computational time during training and during prediction relative to similarly performing logistic regression.

Bagged Logistic Regression Classification

In the midterm report I utilized logistic regression, a linear binary classification model whose objective is to maximize loss entropy. Logistic regression provides an interpretable probabilistic output that is generalized by a softmax function for more than 2 output classes [5]. The loss function I utilized has l_2 regression and is

$$\min_{w,c} \frac{1}{2} w^T w + C \sum_{i=1}^{n} \log(\exp(-y_i(X_i^T w + c)) + 1)$$

where w is the weight vector, C is a regularization constant, c is a bias term, y_i is a vector of labels and X is a matrix of number of examples by number of features (X_i is a vector in X indexed by example i). I found using a direct approach of finding a the optima using limited-memory Broyden–Fletcher–Goldfarb–Shanno algorithm (lbfgs) was more consistent than stochastic gradient descent (sgd). Lbfgs uses an estimated of the Hessian (convexity) to optimize rather than just the gradient (thus less hyperparameters than sgd).

For the final report I took this model further by utilizing a bagged model approach where multiple classifiers built from random sampling of the original dataset to be averaged to give a classifier with similar bias but a lower variance.

I trained the bagging logistic model using 10 estimators with bootstrap sampling and each model built from the same number of samples and features as the original training set. The trained results are shown in Table 1 where the model achieved a test AUC of 0.904. The results can also be seen in Figure 2 which is the learning curve for bagged logistic regression on the data. This learning curve shows that we have relatively small variance (training and validation scores are close to each other), but a somewhat large bias (since we want to get closer .90 accuracy), implying we may need a more powerful model or more data. When comparing to the non-bagged model I found that the results in terms of error and AUC score were virtually the same, but the bagged classifier generally produced a more consistent model that did not change as much as I re-trained or split the data differently.

Multi-layer Perceptron

Next, I explored fully connected neural networks, also known as multi-layer perceptrons. Multi-layer perceptrons can powerfully learn non-linear decision boundaries if trained correctly [6]. These models can be complex to train effectively because of how the optimization objective of neural networks are generally non-convex. A neural network consists of many connected nodes which can

be described as a directed acyclic graph. The function (denoted as f) of a single node can be represented as transforming inputs from previous nodes (denoted as $\boldsymbol{z}$) by some weights associated with the edges of the inputs into the nodes (denoted as $\boldsymbol{w}$) and then transformed by some activation function (denoted as g)

$$f(\boldsymbol{x}) = g(\boldsymbol{w}^\top \boldsymbol{z}).$$

Note that for the input layer the output $\boldsymbol{z}$ can just bee seen as the input features $\boldsymbol{x}$ themselves.

A multi-layer perception is trained efficiently using backpropagation as well as with batch or stochastic gradient decent [6]. There are many hyperparameters to select including the hidden layer sizes of each layer, the batch size, learning rate and more. I selected a 100 first hidden layer size and a 50 second hidden layer size as well as a batch size of 128, ReLU activations, an initial learning rate of 0.008 and the adam optimizer, an advanced form of stochastic gradient descent. I found these results after much experimenting, although there are likely much better hyperparameter selection if experimented with more.

In the end I found I could get a test AUC of 0.902 as seen in Table 1 which was no better than bagged logistic regression. The learning curve can be seen in Figure 3 which shows how it was fairly easy to get high variance with this model and overfit the training data, as evidenced by the farily large gap between the training and validation score. This model provided a low bias on the training data, but not as much so on the validation data. The main difficulty with this model was how many hyperparameters there were to tune and the general computational cost to explore the various configurations, if the variance was reduced through more tuning this would have been a more effective model. I found hyperparameter turning a fairly difficult to effectively explore the hypermeter space and the results were not as consistent between changes in hyperparameter as they were with gradient boosted trees which will be seen next.

Histogram-based Gradient Boosting

Finally, after some research into optimization methods for classifying tabular binary labelled data I came across the ensemble method of gradient boosted trees a method similar to the idea of boosting in AdaBoost but generalizing boosting to be a differentiable function that can be optimized. In scikit-learn the implementation of gradient boosted trees utilizes the implementation methodologies of Light GBM [7]. I specifically used a histogram-based gradient boosted trees because of the size of the dataset being fairly large. Histogram gradient boosting uses histograms (which essentially threshold values into bins) rather than features values to make splits. This results in a computationally more efficient algorithm that gives approximately the same accuracy but allows me to better explore the hyperparameter space [8].

I was able to optimize the performance of the histogram-based gradient boosted using a grid search over hyperparameters of learning rate (a shrinkage factor for the values of the leaf nodes), max depth, L2 regularization, max leaf nodes, and min samples per leaf. I found that a learning rate of 0.1, a max depth of 31, L2 regularization of 15, max leaf nodes of 15, and a min samples per leaf of 10 gave optimal results for the parameter ranges searched over.

I was able to achieve a test AUC of 0.927 as can be seen in Table 1. The learning curves for this model can also be seen in Figure 4. The learning curves show relatively low bias when compared to other models and reasonable variance, thus this model performs fairly well for this data distribution.

In an attempt to further reduce variance I attempted to bag the histogram-based gradient boosted trees. I was able to get slightly less variance which resulted in a test AUC of 0.928 as can be seen in Table 1. The learning curves for this model can also be seen in Figure 5 that illustrate slightly less variance with similar bias when compared to the non-bagged model, although the difference

is small enough it may be due to other factors. This was my best performing model for this final project and this is the model I would proceed forward with.

Next Steps

In conclusion, I feel like I was able to produce an effective bagged histogram-based gradient boosted tree model that can predict if a person makes over 50k a year given features identified by the 1994 census. This model was effective as it achieved a test accuracy of 0.876 and a test AUC of 0.928.

If I had more time I would continue to explore hyperparameters and other ensemble-based gradient boosting methods, potentially using libraries outside of sci-kit learn. I might even attempt to use a deep-learning framework with convolutional layers to see if convolution is able to better extract high-level features from the dataset.

Some other reasonable next steps would be to see if I could get more 1994 census data and use it to further evaluate model performance. From here I would determine what the intended use of the model is and then proceed to make the model more "production-ready" which may mean creating a training or data pipeline to make it possible to update the model in a systematic fashion and also make using the model for prediction easier.

	SVC	Bagged Logistic	MLP	Gradient Boosting	Bagged Gradient Boosting
Train accuracy	0.889	0.854	0.873	0.883	0.882
Train AUC	0.937	0.910	0.934	0.940	0.941
Test accuracy	0.847	0.850	0.855	0.875	0.876
Test AUC	0.880	0.904	0.902	0.927	0.928

Table 1: Performance of Tested Models (All with Normalized Data)

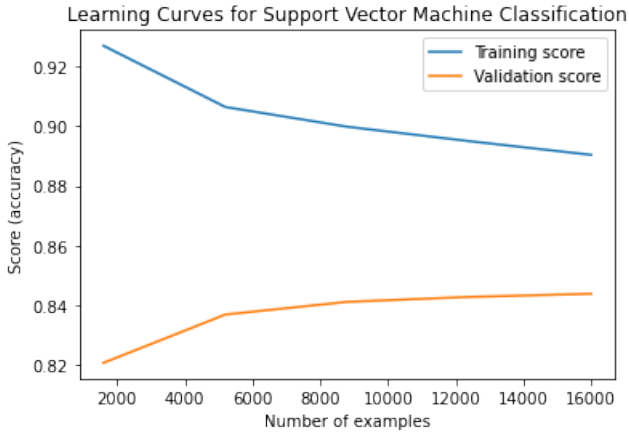

Figure 1: Learning Curve for Support Vector Classification

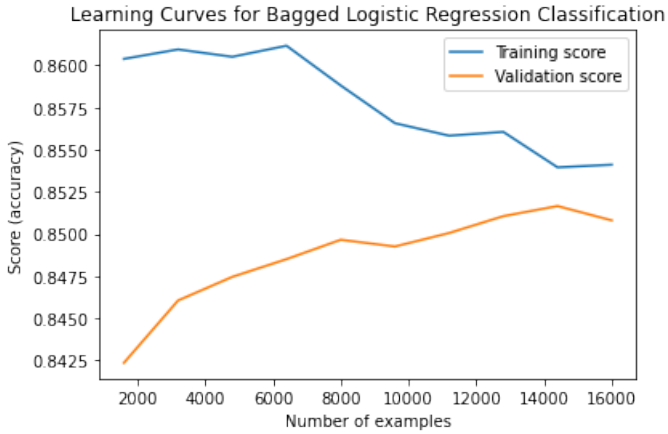

Figure 2: Learning Curve for Bagged Logistic Regression Classification

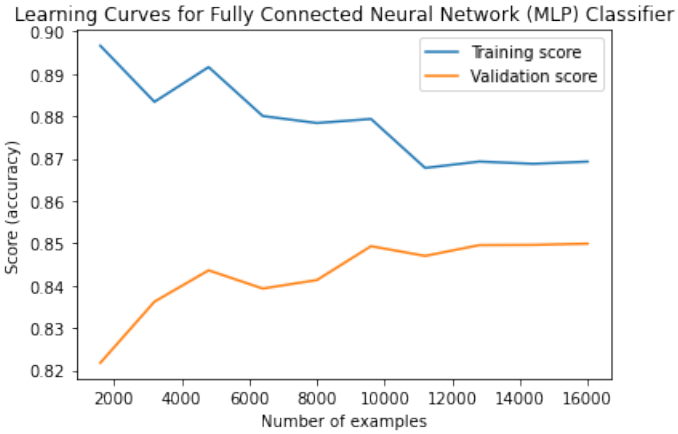

Figure 3: Learning Curve for Multi-layer Perceptron Classification

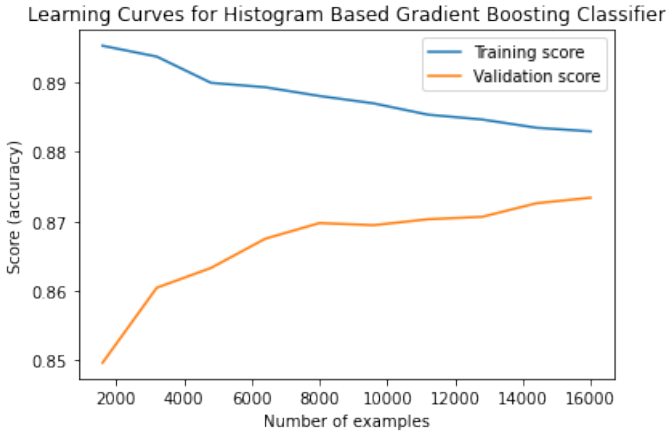

Figure 4: Learning Curve for Histogram Based Gradient Boosting Classification

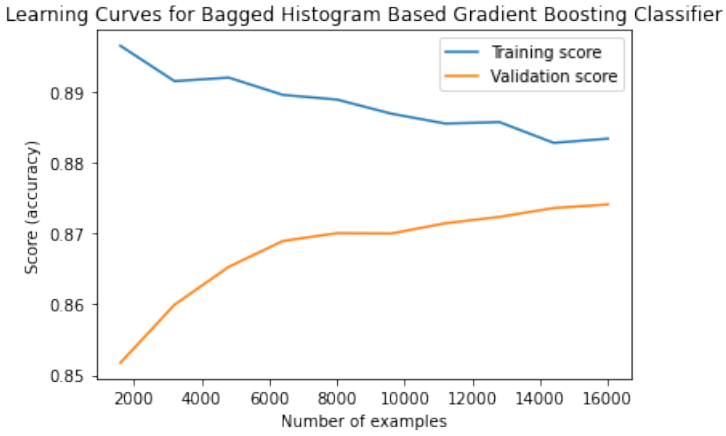

Figure 5: Learning Curve for Bagged Histogram Based Gradient Boosting Classification

References

[1] "scikit-learn: machine learning in Python — scikit-learn 1.0 documentation." [Online]. Available: https://scikit-learn.org/stable/

[2] J. T. Hancock and T. M. Khoshgoftaar, "Survey on categorical data for neural networks," *Journal of Big Data*, vol. 7, no. 1, p. 28, Apr. 2020. [Online]. Available: https://doi.org/10.1186/s40537-020-00305-w

[3] A. Ng, "Advice for applying Machine Learning," p. 30.

[4] "1.4. Support Vector Machines." [Online]. Available: https://scikit-learn/stable/modules/svm.html

[5] "1.1. Linear Models." [Online]. Available: https://scikit-learn/stable/modules/linear_model.html

[6] H. Ramchoun, M. Amine, J. Idrissi, Y. Ghanou, and M. Ettaouil, "Multilayer Perceptron: Architecture Optimization and Training," *International Journal of Interactive Multimedia and Artificial Inteligence*, vol. 4, pp. 26–30, Jan. 2016.

[7] G. Ke, Q. Meng, T. Finley, T. Wang, W. Chen, W. Ma, Q. Ye, and T.-Y. Liu, "LightGBM: A Highly Efficient Gradient Boosting Decision Tree," in *Advances in Neural Information Processing Systems*, vol. 30. Curran Associates, Inc., 2017. [Online]. Available: https://proceedings.neurips.cc/paper/2017/hash/6449f44a102fde848669bdd9eb6b76fa-Abstract.html

[8] "1.11. Ensemble methods." [Online]. Available: https://scikit-learn/stable/modules/ensemble.html