Assignment 1 Report

By Brandon Grunes, Raymond Chen, Jordan Levy

Explanation of implementation

Classifiers:

- SVM -
    - Single Label - For this model we found that the SVC was more accurate than the LinearSVC, so we chose SVC with a kernel type of “linear” and we then chose a C value of 100. However, these models were very inaccurate, starting around 20%, and we could only increase its accuracy to around 46%. This large jump was because of the downsampling of the data and the increase in the value of C to 100.
    - Final Label - This model was very simple compared to the single data and the multi data. We again used the SVC instead of LinearSVC, but this time we had a very low value for C, 0.1, as the base prediction was already 97% accurate.

- KNN -
    - Single Label - This model was pretty simple as it performed very well without much modification of the data or the models parameters. It gave us an accuracy of 74% at first, but when we searched for the optimal number of neighbors, we found that 18 neighbors was optimal. We also had to update the weights to , weights = ‘distance’, to get an even greater accuracy of 90%.
    - Final Label -This model was based on our model for the Single Label and performed the best out of the 2 other models when it came to predicting who won the game. When looking for the optimal value of neighbors, we found that 4 neighbors gave an accuracy of 100%.
- MLP -
    - Single Label - For this model, the predictions were very accurate, 93%, with only one layer. After adding more layers and perceptrons, the accuracy increased to a value of 95%.
    - Final Label - This model was very accurate from the start with an accuracy of 97%. Once we added some more layers, each with the number perceptrons being a power of 2, this accuracy increased to 99%.

Regressors:

- Linear Regression - For this model we made 9 different models. We then rounded the decimal values in each index of the array to either 1 or 0 so it could be compared with the optimal values. Due to the extremely poor accuracy of Linear Regression, most decimal values were below 0.5, so most indexes were set to 0.
- KNN Regression - For this model, we needed to find the optimal amount of neighbors. So, we kept increasing the number of neighbors to 17 and stopped because we also found that any neighbor greater than 17 caused the accuracy to decrease. Furthermore, we modified parameters so that it took into account weights = ‘distance’. We did this because weight assigned to each neighbor is determined by the inverse of its distance, so closer neighbors contribute more to the prediction, while farther neighbors contribute less.

- MLP Regression - For this model, we found that if we increased the number of layers, increased the number of perceptrons at each layer, and decreased layers by powers of 2, our R-squared value could be improved. The new layers that we used are (1024, 128, 64, 32). However, we only noticed a difference in convergence speed when adding more layers. The more layers we added, the longer training took.

Tic-Tac-Toe Game - The game was fun to play and there was a clear winner on the best model for playing. MLP trained off of Single Label was impossible to beat as it would always pick the optimal move, resulting in either a draw or loss. The MLP model trained with Multi Label was also impossible to beat. The 2nd best was KNN trained off of the Multi Label because it sometimes picked a suboptimal choice and allowed us to sometimes win. The KNN trained off of Single Label did slightly worse but barely distinguishable. The worst was Linear Regression because it had poor accuracy and would tend to choose invalid indexes, resulting in an unplayable game.

Instructions to run our Code

To run the TicTacToe game, just run the last cell under “Main Function” to play the game, since the Jupyter notebook will have all the models trained already. It will ask for input, if you want to play type ‘1’, else type ‘2’ to quit. To select a gridbox to place your ‘X’, you are expected to input a row number and a column number. For example, inputting 0 for both row and column will place the ‘X’ in the top left corner of the board. There is a commented diagram and row-column labels everytime the board outputs to explain what to input. Play the game until you are satisfied with the results. If the kernel happens to crash and the game produces errors, please re-run the entire file. This will take approximately 20 to 25 minutes to complete.

Bugs and Difficulties

- Modifying the number of perceptrons for each layer was a challenge to balance. We wanted enough so that it could more accurately move. However, we found that adding too many perceptrons/layers not only increased training time but also caused overfitting, decreasing accuracy.
- When we first started training the KNN models, it was outputting a low accuracy and we couldn’t determine why. Later we figured out that we needed weights = ‘distance’ in the training parameters. This increased the accuracy because the weight assigned to each neighbor is determined by the inverse of its distance, so closer neighbors contribute more to the prediction, while farther neighbors contribute less.
- Since the SVM model is not capable of training off of large datasets, we had to figure out how to downsample the data to fit the requirements. Learning how to do this took some time and research in the Scikit-learn textbook.
- The SVM model and the Linear Regression models were both so inaccurate, that they kept predicting the same move infinitely. This would cause the kernel to crash, and we had to restart it as well as the long runtime.
- We came across several bugs throughout the development of the game:
    - Visual Studio Code kept deleting and misplacing cells when multiple people were collaborating (Live Share) on a project.
    - Output from main would not appear.
    - Random crashing of the Python kernel when interrupting the training.
    - Remote collaborators unable to input text into game