GROUP INFORMATION
Canvas Group 99
Kaggle Team Name: epic cheese nuggets

Joshua Hsin, 13651420
Ryan Wong, 56903918
Marissa Lafreniere, 5781245

TABLE OF MODELS

	Training perf.	Validation perf.	Kaggle perf.
Linear Models	**Polynomial (no cross products)** Error: 0.29202 **Stochastic Gradient descent** Error: 0.49730	**Polynomial (no cross products)** Error: 0.35542 **Stochastic Gradient descent** Error: 0.49891	0.63989 N/A
Random Forests	Error: 0.36772	Error: 0.38515	0.73024
Kernel SVM	Error: 0.25835	Error: 0.38668	0.62195
Gradient Boost Ada Boost	AUC: 0.8071 AUC: 0.7772	AUC: 0.74435 AUC: 0.73185	1st: 0.64669 2nd: 0.73860 N/A
Bagging Ensemble Stacking Ensemble	N/A	AUC: 0.742981 AUC: 0.74688	0.73979 (public) 0.74520 (private) 0.74306 (public) 0.74902 (private)

MODEL 1: Polynomial model with no cross products

I split the data into half training and half validation data, and borrowed a function I made in homework 2, which takes a degree and makes polynomial features and cross-products. I slightly tweaked it to ignore cross-products since there were too many features, which caused errors, and I had the function print the training and validation error for the degree chosen. I then iterated through a loop, calling the function with degrees 1-40, and chose the model with the lowest validation error which was degree 27. Lastly, using a degree 27 polynomial model, I tested data by using sklearn library's feature selection and selected different amounts of features between 40 and 80. Using the 70 most important features worked best.

MODEL 2: Stochastic Gradient Descent

First, I selected the 70 best features since it gave the best performance for me in a linear model. I then wrote the function with parameters for number of iterations and alpha for the step size. In theory, the theta would become a better classifier by adjusting based on every point and averaging. At first, I used the linear trainers theta as a baseline and trained the data on different alphas like 0.3, 0.5, 0.7, 1.0, and 2.0 and a small iteration value. 0.7 worked the best for me, so I stuck with it. I did the same thing with iterations, testing values like 5, 10, 20, 50, 100, 200, and 500. I later also tested a baseline theta of all zeros, and overall, the errors were slightly less. With this theta, I repeated the process of picking an alpha and an iteration value, and ended up with the same optimal parameters - 0.7 for alpha and 30 iterations.

MODEL 3: Weighted Random Forest

First the data was split into training and validation using the default 80 | 20 split. Based on the performance graph in Homework 4, I decided to use 23 bagged decision trees. I tested this decision tree with different values for minParent, bootstrap data sizes, and minLeafs and plotting the results. This technique gave a bootstrap size of 2125, maxDepth=6, minParent=200, and minLeaf=8. I attempted to further improve accuracy by assigning a weight to each of the learners. This weight was calculated by evaluating each model's performance on the validation data and assigning weights accordingly. This technique performed slightly worse than my original random forest from Homework 4. I suspect this is because I split the data so many times. The final random forest was trained on subsets (bootstrapping data) of a subset (training data) of the original data.

MODEL 4: Kernel SVM

For this model, I used the sklearn library and specifically the svm module. First I used sklearn's feature selection module to subsample important features. I chose 70 since I tested the subsampled data on linear features and subsampling by 70 gave the best results. I then used a scalar function in the sklearn library called

GROUP INFORMATION
Canvas Group 99
Kaggle Team Name: epic cheese nuggets

Joshua Hsin, 13651420
Ryan Wong, 56903918
Marissa Lafreniere, 5781245

StandardScaler to normalize the data. After performing different kernels like linear, polynomial and rbf, rbf performed the best. I believe this is because rbf is the most flexible and "smoothest" of the three, suitable to the large dataset and large amount of features. I tried scale and auto values for gamma which use 1 / (n_features * X.var()) and 1 / n_features, but they did not have much impact on training and validation error, so I just chose auto. Finally, I changed the C value, or the regularization of the Kernel SVM. At first, I tried lower numbers and the training error decreased greatly, with little impact on my validation error. This made me realize the model was overfitting, so I lowered the C value to 2.0, which improved both training and validation error slightly.

MODEL 5: Gradient Boost with Bagging

I initially used different regressors and ended up using dtree.treeRegress with auc of 0.705 (first kaggle auc of 0.64 was from tree classification). Then I switched over to using the sklearn library to easily graph, cross validate, and test different hyperparameters. I first graphed test vs training auc graph over different numbers of tree regressors and learning rates (alpha) and found 75-100 regressions, alpha < 0.1 was optimal (any further started to slightly overfit as the train roc curve became much higher than test's). Using lower alphas with higher trees had the greatest improvement among all parameter changes. In hope of reducing "noise" from the feature space, I used linear regression to keep the top k features that had higher correlation between each x_i and y_i (This improved the auc by ~0.02 with k=80~90). Then, I used the subsampling option along with a low learning rate to allow for bagging / stochastic gradient boosting and set max_features to 30~60 for each regressor to further minimize overfitting and variance (This improved the auc by ~0.01). Lastly, I used an exhaustive grid search with cross validation to permutate various parameter combinations on the learner and feature selection parameters to fine tune the best parameters. This lead to subsampling: 0.8, n_estimators: 75~100, learning rate: 0.08, max_depth: 3, min_leafs: 3 or 4, max_features: 30~60 and feature selection: 90/107. AUC score for the validation split was 0.74435, which was the highest of all individual learners. (However, this score often fluctuated +- 0.015 depending on the random state used, indicating that it would sometimes choose a bad local minima even though I used a convex loss function - logistic regression. This further signified the importance of using cross validation)

OVERALL PREDICTION ENSEMBLE

We first ran all our learners and ranked best auc scores to worst. As such, we decided to use Gradient Boost, AdaBoost, Random Forest, and SVC.

First, we stacked all of the top performing learners by averaging the predictions based on weights. To get our weights, we used itertools cross product function to iterate possible weights from 0.02 ~ 1.00 (step=0.02) for each learner. After figuring out the best performing weights on the validation split, we used that same weight for the kaggle submission. We used the weights: [1.0, 0.46, 0.44, 0.02] in respective order with the learners mentioned above. This resulted in an AUC of 0.74902 (private), 0.74306 (public).

We also tried the bagging technique on the same models as above. Each model was trained on a subset of the data (2500 points) 30 times. Then we took the average of those 30 predictions to come up with 1 prediction for each model. To come up with the final prediction, we took a weighted average of all the models according to the best performing weights used on stacking the models. Ultimately, this was a very slow technique, but performed relatively well producing an AUC score of 0.74520 (private), 0.73979 (public).

It can be seen that the ensemble of learners scored higher than any of the individual learners. This demonstrates the concept of "mixture of experts".

CONCLUSION

The Stochastic Gradient Descent likely did not work well due to the data being noisy and non-differentiable. Kernel SVM likely did not work well due to noise and Xtest's large data set. Although the training X was biased towards a radial basis kernel function, due to noise, Xtest may have fit better in a polynomial or another type of kernel function. The Polynomial model likely did not work well due to overfitting. Although validation error showed good performance, the validation data and training data was only one-fourth the size of the testing data, making the model biased. The weighted random forest likely decreased performance from the normal random forest from the data being spread too thin between bagging and testing. Lastly, the gradient boost ultimately did the best of all learners likely due to its nature of learning/correcting losses/mistakes from previous iterations.