Santander Customer Transaction Prediction

Isaac Kresse
Tejas Krishna Reddy
Bruno Costa Rendon
EECE 5698 Parallel Processing for Data Analytics

Problem Statement:

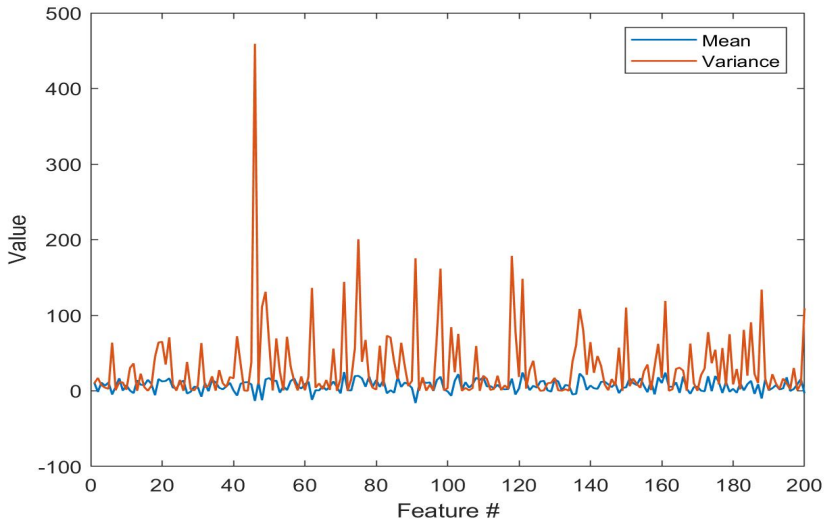
In this project we analyzed data provided by Kaggle regarding customer transactions. The data consists of 200,000 rows, and 200 columns. Throughout the paper we will reference the 200,000 rows as examples, and the 200 columns as features. Each example represents a customer, and each feature represents an unknown variable. Each variable is a float number, larger or smaller than zero. It is worth noting that the data set is dense, and we cannot take advantage of sparse vectors to further improve runtimes. The objective of classifying this dataset is to determine whether a customer will make a next transaction or not. The labels for this objective are given as a 1 or a 0. This is a binary classification objective.

Approach:

We start off by exploring the data. We note the skewed number of labels in the dataset: 180,000 examples are classified as 0, and 20,098 examples are classified as 1. In earlier attempts at classification, our classifiers were consistently calling all samples class 0, likely due to overfitting caused due to the imbalance in data set. From this we make a critical decision to approximately balance the number of examples per class we use in this project. All points in class 1 were kept, and an approximately equal amount of points were sampled randomly from class 0.

0	21,000
1	20,098

With the 41,098 examples in hand, we begin a preliminary analysis of the data. We calculate the mean, variance and standard deviation of the 200 features in all 41,098 examples. The calculated variance and mean per feature are plotted below.



Data Preprocessing:

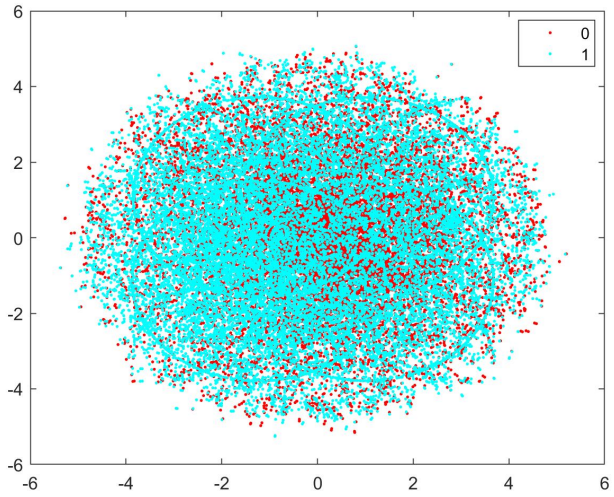
To address the large variance between features, we normalize the data using the *z-score*. The technique involves a process of scaling the dataset such that the features have zero mean and unit variance. This is achieved by subtracting the mean of a feature from every data-point and dividing the result by the standard deviation of that respective feature. This method of normalizing the data is commonly termed as Standard Scaling Technique. Standardization is a requirement for many machine learning estimators, i.e the models might behave erratically if the individual features do not more or less look like standard normally distributed data. Standard Scaling Technique is best suited for classification problems, especially binary classification problems. The mean and standard deviation are used calculate the *z-score*:

$$z = \frac{x - \mu}{\sigma}$$, where μ = mean, σ = standard deviation

We perform this *z-score* analysis with both *pyspark* and *scikit-learn* on the discovery cluster. When calculating this metric, we also measure the time it takes to run, over ten trials. We note how the overhead to parallelize the data is greater than the computation itself. Because of this, a single thread to compute the *z-score* with *scikit-learn* is faster than multiple partitions with *pyspark*.

scikit-learn	0.5763 s
pyspark	1.2824 s

As a final step to help us understand the data, we looked for methods to reduce the dimensionality of the data. We found the t-SNE Barnes Hut technique to be well suited for this, and we plotted it with a built in function in MatLab. From this we can observe the great similarity between both classes.



Classification:

To classify the data, we decided to compare several different approaches. We classified the data using:

1. Linear Regression
2. Random Forests
3. Neural Networks

Linear Regression: We attempted to classify the balanced dataset by minimizing the regularized linear loss, as described by the equation below. In this equation, y_i is the label of sample i (x_i), β is the vector of biases associated with each feature, and λ is the regularization factor.

$$l(x, y; \beta) = \sum_{i=1}^{n} (y_i - \beta_i{}^T x_i)^2 + \lambda ||\beta||_2{}^2$$

The loss function was minimized through gradient descent, where the gradient of the loss is shown below.

$$\nabla_\beta l(x, y; \beta) = \sum_{i=1}^{n} [-2(y_i - \beta_i{}^T x_i)x_i] + 2\lambda\beta$$

On each iteration of the gradient descent, backtracking line search was used to find the next values of β. The algorithm was repeated for either a fixed number of iterations, or until the gradient of the loss was below a threshold value (by default 1×10^{-99}). For determining the correct regularization parameter value, λ, the data set was split into five folds, and cross-validation was performed for λ values in the range [0, 20]. In this stage, the algorithm was run for a maximum of 20 iterations. The λ value with the lowest test RMSE across all folds was used for the final classification using the entire balanced dataset.

Random Forest: Random Forests is a flexible, easy to use and powerful supervised machine learning algorithm that could be used for both regression and classification problems. The 'forest' it builds is an ensemble of decision trees, which is trained from bagging method. To put them in simple words, Random Forest Algorithm builds multiple decision trees and merges them together to get a more accurate and stable prediction.

An important advantage of random forest is that it adds additional randomness to the model, while growing the trees. Instead of searching for the most important feature while splitting a node, it searches for the best feature among a random subset of features. This results in a wide diversity that generally results in a better model.

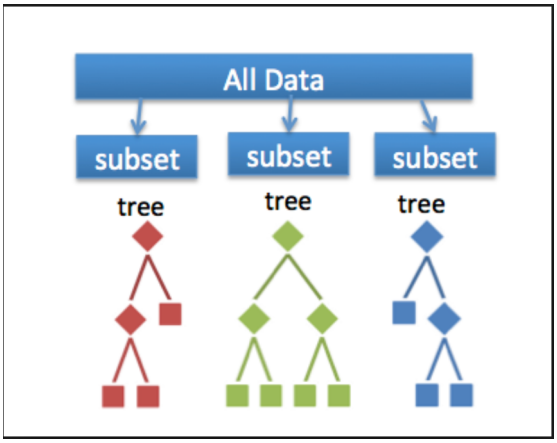
One of the most important feature of Random Forest is that, it computes the relative importance of each feature on the final prediction. The importance score is measured by looking at how much the tree nodes, which use that feature, reduce impurity across all trees in the forest. The sum of all the importance scores will add up to 1.

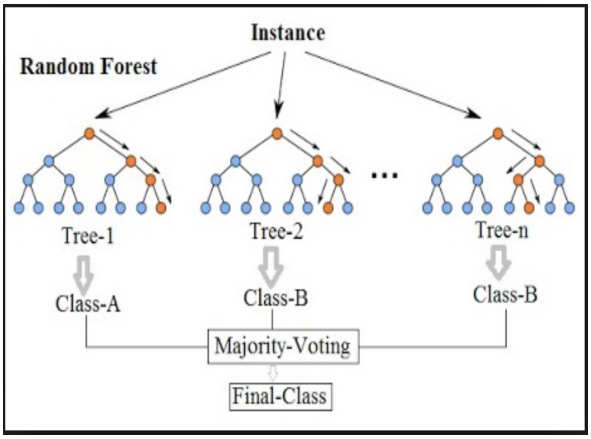
If the hyperparameter "*oob_score*" is set to be "*True*" then the Random Forest algorithm uses cross validation at each of its decision trees by calling few out-of-bag unseen samples which further checks on the performance of the model.

As mentioned above, random forests train each tree independently and later votes for the results. Since, each tree is trained independently, multiple trees can be trained in parallel in addition to the parallelization involved in single trees. This distributed learning of ensembles is of great use in terms of its fast execution and effective results. Distributed ensembles can be developed using *pyspark* through its API, namely *MLlib.* A variable number of subtrees are trained in parallel, where the number is optimized on each iteration based on the memory constraints.

A common issue with random forests is that, though the training of algorithm is fast, it takes significant amount of time to predict results over a large dataset. This problem can be alleviated by using parallel computing to a considerable extent.

In this project, we build two random forest models - with and without spark. We then compare the results obtained and their time of execution at each stage (training and testing period) with one another in order to have a better understanding of the capabilities of parallel computing.





Neural Networks: With the recent advances and popularity of neural networks, we wanted to try them out ourselves and test how they performed with respect to the other methods we used in this project. We built a neural network model consisting of fully connected layers. A layer consists of a series of neurons containing values. The input layer contains a number of neurons equal to the number input examples. In our case, the number of examples is 41,098. Hence, the first layer contains 41,098 neurons. Our model layers are designed in funnel form; every subsequent layer contains a smaller number of neurons than the previous one. This funneling design allows for feature reduction. By representing the input data at every deeper layer with fewer neurons, we are reducing the dimensionality of the data.

Each of the layers are connected in a fully connected fashion. This means every layer is multiplied with a matrix, which results in a smaller vector of neurons. Hence, every neuron in the next layer is an aggregation of matrix multiplications.

$$\sum_{i=1}^{n} (W_i^T \cdot x) + b$$

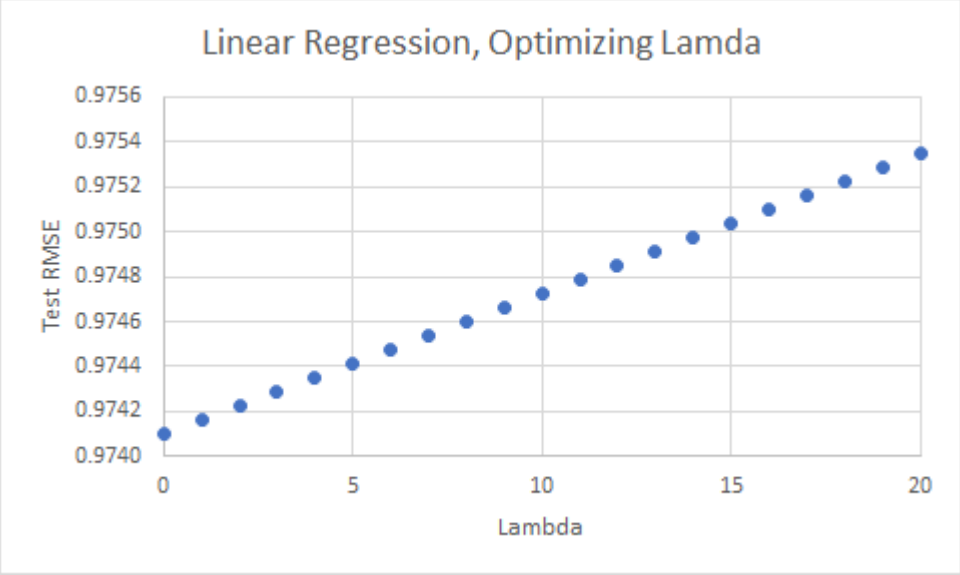
Including a bias term is an optional design choice that we decided not to use.

We implemented the model using the *keras* framework. The model includes 3 fully connected layers with the following sizes: 41098, 1024, 1. At the first two layers a Rectifying Linear Unit (ReLU) activation function is used. Activation functions are used the filter the output of a node. An ReLU outputs the maximum of either 0 or the node value x.

$$f(x) = max(0, x)$$

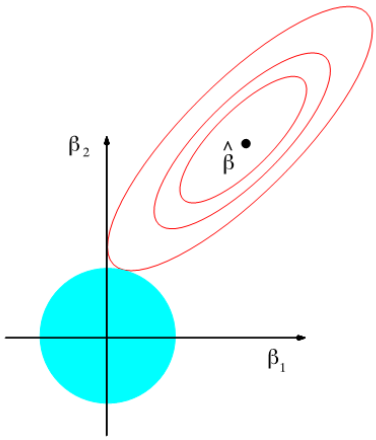
Validation:

Linear Regression: Before evaluating the results of the classifier, the value of the regularization parameter, λ, needed to be determined. The k-fold cross validation test RMSE is shown below for a variety of λ values.



Typically, we would expect the Test RMSE to follow the shape of the expected predicted error (EPE) and have a minimum value for some intermediate value of λ. However, in this case, we see the test RMSE gradually increase with λ and is its lowest when the regularization parameter is set to zero. However, it is also worth noting that the Test RMSE changes very little over this range of λ (0.9741 to 0.9754). This odd behaviour may demonstrate that linear

regression is not the best method of classification. When trying to understand why the regularization parameter did not improve the Test RMSE, we considered the following figure from our lecture:



In this figure, increasing lambda decreases the size of the boundary within which β is minimized. If β is very small along all dimensions, the global minimum may well lie within the boundary. As long as the global minimum lies within the boundary, we would expect no change in Test RMSE. The negligible change seen above may be a case of this happening. As further support, the magnitude of β_i for all i is less than 0.0016. Though it is difficult to say so without a sense of scale, this seems rather small, and may explain the lack of change in Test RMSE.

To evaluate the quality of the linear regression classifier, the area under the curve (AUC), accuracy, precision, and recall were measured using k-fold cross validation (shown below). Interestingly, these measurements were all the same for all values of λ that were tested. This indicates that the regularization parameter did not significantly affect the classification of the data.

Parameter	**Value**
Area Under the Curve	0.7468
Accuracy	0.6814
Precision	0.6737
Recall	0.6762

Overall, the classifier performed moderately. The AUC was around 0.75, indicating that it was about halfway between a random coin and a perfect classifier. The top 50 leading teams in the Kaggle competition from which this data was obtained have an AUC of 0.92. Clearly this simple

linear regression classifier does not perform as well. Without knowing the costs of false positives versus false negatives, we cannot say whether it is more important to have high precision or high recall. However, in this case, the two values are approximately the same.

Random Forest: The hyperparameter “oob_score” was set to be ‘True’ in the random forest model. This would let cross validation happen simultaneously during the training period of the model. Out of the bag samples which are unseen by the model are chosen randomly and are trained by each of the subtrees which acts as a cross-validation set.

Apart from this, the dataset formed was divided into training_set, cross_validation_set and testing_set in the ratio 70%, 15% and 15% respectively, where the model developed was tested over cross validation set and to tweak the hyperparameters of random forest model for better results. The best hyperparameter optimized model was then tested over testing set to produce the results as described below:

Parameter	Value
Cross-Validation Accuracy	0.87195621
Testing Accuracy	0.7749585
AUC Score	0.7756751
True Positive Rate	0.732463
False Positive Rate	0.181113
Precision	0.801664

The figure below shows the different accuracy measures for Random Forest Algorithm:

```
In [49]: from sklearn.metrics import r2_score, accuracy_score
    ...: from sklearn.metrics import roc_curve, auc
    ...:
    ...: print ("Training Accuracy: ", accuracy_score(y_train,
rf.predict(X_train)))
    ...: print ("Testing Accuracy: ",  accuracy_score(y_test,
rf.predict(X_test)))
    ...:
    ...: false_positive_rate, true_positive_rate, thresholds =
roc_curve(y_test, rf.predict(X_test))
    ...: roc_auc = auc(false_positive_rate, true_positive_rate)
    ...:
    ...: print ("AUC Score: ", roc_auc)   ## AUC score is better
    ...: print ("False Positive Rate = ", false_positive_rate[1])
    ...: print ("True Positive Rate = ", true_positive_rate[1])
Training Accuracy:  0.8719562163427769
Testing Accuracy:  0.7749585406301824
AUC Score:  0.7756751552231168
False Positive Rate =  0.18111298482293423
True Positive Rate =  0.732463295269168
```

Upon averaging several monte-carlo runs, we could see that AUC score was around 77.5% using random forests with 500 trees, max_features = 5, max_depth = 8, min_samples_leaf = 4.

Neural Networks: The model's loss is measured with *kera*'s built in *mean_squared_error* loss. The formula for mean squared loss is described as:

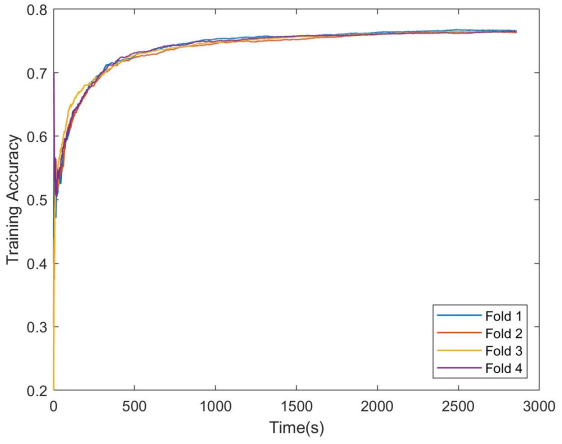
$$RMSE(x, y; \beta) = \frac{1}{N} \sqrt{\sum_{i=1}^{n} (y_i - \beta_i{}^T x_i)^2}$$

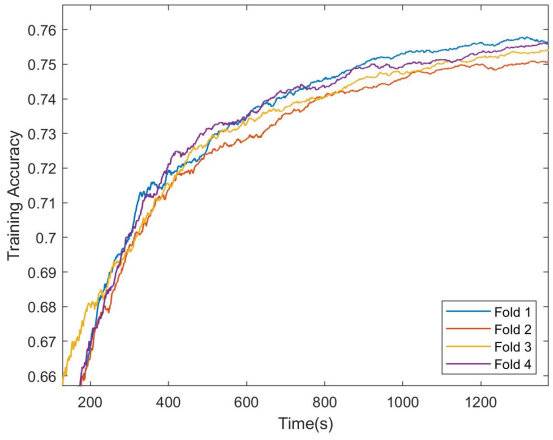
In order to improve the model, a stochastic gradient descent optimizer is used. Compared to gradient descent, SGD uses an estimate of the gradient $g(x^k, w^k)$ to decide where the next step will go.

$$x^{k+1} = x^k - \gamma^k * g(x^k, w^k)$$

SGD is iterated at every point of the dataset. A full iteration of the dataset is called an "epoch". In our case, when training the model, we used 1 epoch. For training we use a bach size of 10. Batch sizes define the number of examples used at input. The training and testing phases are done four separate times using a 4 k-fold cross validation. We plotted the training accuracies for each k fold below.

Accuracy fold 1:	77.27%
Accuracy fold 2:	77.69%
Accuracy fold 3:	77.74%
Accuracy fold 4:	77.59%
Average accuracy:	77.57%

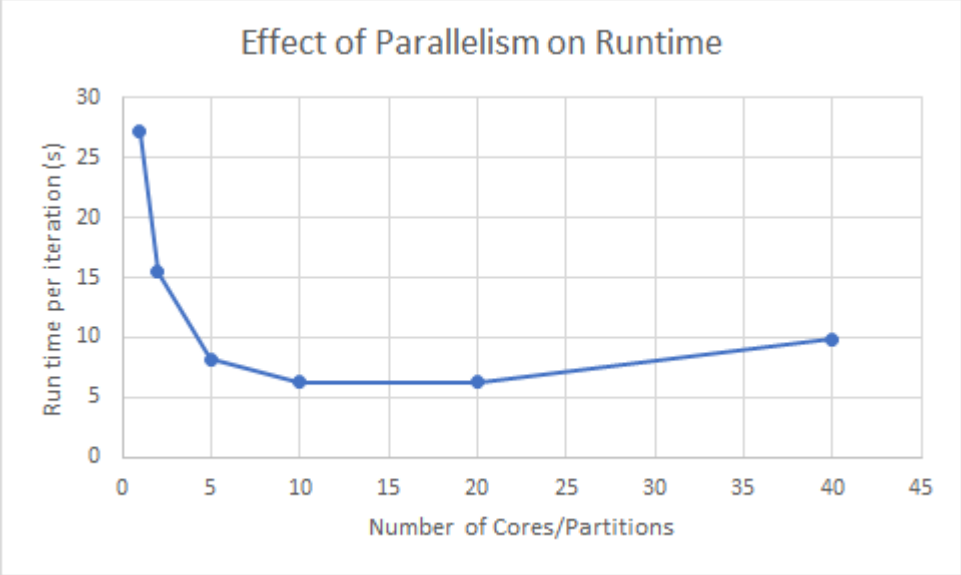




We can see every cross validation fold has very similar training accuracies. The small discrepancies between them are due to the random split of the data.

Performance:

Linear Regression: To demonstrate the benefit of running linear regression in parallel, we varied the degree of parallelism and measured the time per iteration of the gradient descent. To control the degree of parallelism, we gradually decreased the number of cores and partitions used. The number of partitions was set equal to the number of cores, so that each core operated on a single partition.



As is shown in the figure above, the runtime significantly drops when going from a single core (sequential operations) to 5 cores (parallel operations). After 5 cores, the runtime only decreases slightly. At 40 cores, the runtime per iteration actually increases. This increase in runtime may be explained by an increase in communication between cores. As the data becomes more spread out, more communication is necessary. Since this data set is somewhat high dimensional (200 dimensions), the communication cost may be higher than in other applications. However, even with this small increase, the runtime per iteration when using 40 cores is still significantly lower than without parallelism.

Random Forest: The time of execution between a random forest algorithm with and without parallelization can be acknowledged at different stages of the program in different ways. The *MLlib* API in *pyspark* is used to parallelize random forest algorithm in spark. This algorithm trains different number of subtrees in parallel at different stages. The number of trees trained in parallel is optimized based on the memory constraints regularly.

For random forest algorithm to be executed in parallel, the training data must be available to the program in a way the data can be segregated into parallel cores simultaneously. This cannot be achieved by a pandas dataframe, but a spark SQL dataframe is capable of handling parallelism.

Thus, the data must be preprocessed and contained in spark SQL dataframes before initializing random forest algorithm to train on that data.

Now, let us look upon the time difference at different stages of of the program of a random forest algorithm with and without parallelism.

Event	Without Parallelism	With Parallelism
Data Preprocessing - Standard Scaling Normalizer	0.5763 s	1.2824 s
Training Random Forest	14.2747 s	4.99256 s
Prediction of Test Set	0.8431 s	0.039254 s
Calculating AUC score	0.2390 s	0.01536 s

```
In [69]: start_random = time.time()
    ...: rf = RandomForestClassifier(labelCol="target",
featuresCol="features", numTrees=10)
    ...: model = rf.fit(training_data)
    ...: end_random = time.time()
    ...:
    ...: start_prediction = time.time()
    ...: predictions = model.transform(training_data)
    ...: end_prediction = time.time()
    ...: print("Random Forest training time = ", end_random -
start_random)
    ...: print("Random Forest testing time =", end_prediction -
start_prediction )
Random Forest training time =  4.992562532424927
Random Forest testing time = 0.03925466537475586
```

We can clearly see the drastic change in the time of execution for random forests with and without parallelism. Training period of random forest algorithm was reduced to almost 1/3rd of its original time.

Also, prediction time was reduced to almost 1/10th of its original time for the same data with parallelization. This proves us the efficacy of the results with parallelization property.

“featureImportances” attribute can be used to print the relative importance scores of the available 200 features in predicting final results. If we threshold the produced results to 0.04 importance, then the least 70 important features among 200 can be removed. The AUC score

obtained after selecting only important features was found to be 72.5% which is just 2% less than the actual accuracy as a trade off for removing 70 features.

Neural Networks: We trained and tested the neural network on the Discovery Cluster. Installing the required python modules was done through an anaconda environment. To run the neural network we allocated a node with 50 GB of available memory on the *general* partition. The total runtime per fold for the training and testing was:

	Training (mm:ss)	Testing (mm:ss)
Fold 1	10:21	00:25
Fold 2	10:05	00:20
Fold 3	10:12	00:23
Fold 4	10:14	00:24

As in the performance section, we can observe each fold taking about the same amount of time to be computed. When running this on our local machine with an Intel Core i7 7th Gen, we observed a slower training and testing time:

	Training (mm:ss)	Testing (mm:ss)
Fold 1	15:45	00:48
Fold 2	15:28	00:42
Fold 3	15:35	00:44
Fold 4	15:43	00:46

These results allow us to observe the parallelism occurring in the discovery cluster when we allocating a node.

Conclusion:

In this project we explored three different methods to classify the Santander Customer Transaction Prediction dataset from Kaggle. Each one of the team members was responsible for each method. Isaac Kresse used linear regression, Tejas Reddy used random forests, and Bruno Costa Rendon used neural networks. Throughout each individual analysis, different metrics for validation and time performance were measured. The shared metric among all techniques was accuracy.

	Linear Regression	Random Forests	Neural Network
Accuracy	68.14%	77.49%	77.57%

We can see linear regression was able to get a better performance than 50%, given we balanced the data fairly equally. However, compared to the other methods, it did not perform as well. Both random forests and neural networks got similar results. For future work, random forests and neural networks could be further optimized with hyperparameters. The neural network model could be tested with different batch sizes, different number of epochs, different optimizers, and different loss functions.

References:

1. https://www.kaggle.com/c/santander-customer-transaction-prediction/overview
2. https://www.mathworks.com/help/stats/visualize-high-dimensional-data-using-t-sne.html
3. https://keras.io/layers/core/
4. https://keras.io/losses/