Electric Dream Machine CS3270 Spring 2024 Kaggle Report
(T. Lamont, N. Sacandy, D. Emmons)

We are Electric Dream Machine, and this is our report.

The year is 2912, and the maiden voyage of the Spaceship Titanic has run afoul of an iceberg-flavored black hole. For our Kaggle competition, we were tasked with solving a sci-fi disaster: what are the common attributes amongst those teleported to another dimension after the collision?

We need to look at the end and work backwards a bit to see what is required. What are we trying to accomplish? We have said common attributes, but the way we test our findings of "common attributes" is by applying our model to unlabeled passengers and announcing which of those passengers would be transported and which would not. We have "transported" and "not transported" as possible labels, so we have a binary classification problem. ⅔ have labels (we will call this training data), ⅓ does not (test data), so we take the model we have created with the training data and apply it to the test data. Easy peasy; however, we can only test against the test data 10 times daily. This limits the amount we can fine-tune our data **without overfitting**.

Luckily, our brilliant and talented professor reminded us of k-fold testing and advised that we create a dev set and labeled test set. This allowed us to do some editing of our algorithms in a non-black box sort of way.

K-fold is easy enough to implement by just splitting the data into 5 or 10 slices; however, you are unlikely to get good stratification that way. We used sklearn's StratifiedKFold function to bust up the data into usable chunks.

Let's take a look at some of the features that *may or may not* be important in solving this mystery.

- PassengerID - in the format ####_##
- Home Planet - one of a limited number of planets
- Cryosleep - binary value indicating whether or not the passenger spent the trip in suspended animation

Electric Dream Machine CS3270 Spring 2024 Kaggle Report
(T. Lamont, N. Sacandy, D. Emmons)

- Cabin Number - in the form deck/num/side, side being port or starboard
- Destination - one of a limited number of planets
- Age - an int
- VIP - binary value indicating whether or not passenger bought VIP service
- RoomService, FoodCourt, ShoppingMall, Spa, VRDeck
    - Decimal value indicating amount spent at each of these places
- Name - First and last names of passenger, probably very sci-fi names
- Transported - finally our labels/class, indicating whether or not the passenger was transported during the wreck

Just perusing the above data, some problems immediately jump out: how are we going to compare names? How do we compare planets? How do we compare completely different data types vis-a-vis something like Naive Bayes that only takes presence or absence into account?
Great news! Pandas has a function called .get_dummies that solves this exceedingly common problem. In a small collection (like an enumerated type), the get_dummies function turns the small list planets into a binary value, like the following:

I live on Earth. I do not live on any of the other 7 planets in the solar system, so for all those planets, I would have a value of 0, and for Earth, I would have a value of 1.

.get_dummies also automatically corrects incorrect data types like “Not a Number” and missing values.

While the .get_dummies function was a fantastic boon, there are elements that we did not drill down into as much as we would have liked. Two problem attributes are “Cabin Number” and “amounts spent” (for lack of a better term). Cabin number contains numbers that correspond to what could be called an X and Y axis on a map, but *also has a binary Z value.* With a great deal of time and skill that we do not have, we could have split this data up into *at least* two fields. Furthermore, the X and Y

values might have a specific relation to the Z values vis-a-vis passenger teleportation. A separate model could be made on location in the ship, alone.

The “amounts spent” attribute is similarly complicated. There are multiple places to spend money, and we don’t know the relation between amount spent and influence on teleportation. Is it a binary question? Does spending $0.01 at a specific store cause teleportation? Is it a total amount spread across all stores? Is it a sigmoid function? Once again, a separate model might be necessary to answer such questions.

Before any models were made, we took the list of passengers in the test data and made a submission asserting that all passengers were teleported (a 1 value for all cases in the test data.) Why? To get an idea of the split we should be looking at. If it came back that we had 25% accuracy, we would know that our model should predict roughly 75% of passengers having a 0 value. What we got back was 50.689% accuracy. This told us three things: one, we can’t just submit a file with all true or all false and get any higher than 50% accuracy; two, we should have a model that splits the data into roughly 50/50; three, the difference will most likely be discrete.

As mentioned before, one of the trickier details of our dataset was how to handle the “cabin” attribute. With each entry representing essentially three sub-attributes it was difficult to determine the best way to present these values to the model. We decided the best approach would be to separate out these three sub-attributes. While this did lengthen our dataset by adding three new features, the effects this had on our predictions were beneficial, as we now had even more usable data to train our models on.

Another detail we believe contributed to the accuracy of our models was a nuance we found in the “PassengerID” attribute. These ID numbers had a format of “xxxx_yy”, where the first four digits correspond to a group number. Noticing this provided some further insight into the dataset, as passengers in the same group were more likely to be together.

The original dataset from Kaggle is also incomplete, with some fields missing from random passengers; this makes sense as in the real world, we will not always have complete data for every data member, but we do not want to throw out valuable data just because we don't know if they are a VIP or not. To combat some of the incompleteness, we made a few assumptions: anyone who had spent 0 dollars with missing cryosleep data was assumed to be in cryosleep. Likewise, anyone in cryosleep's missing spending data was set to 0. People spending money were not in cryosleep, and people were not VIPs if that information was missing. Doing our due diligence before running tests of splitting and refining the data allowed for higher-quality data while training.

Given that we are still minors in the adult world of AI, we used sklearn prebuilt algorithms to create our models. Below is a summation of our data:

Algorithm	**Accuracy**
Gaussian Naive-Bayes	76.4%
Multinomial Naive-Bayes	61.9%
Decision Tree	77.4%
Random Forest	79.2%
Logistic Regression	78.4%
Neural Network	80.1%

As you can see, the more "sophisticated" algorithms won out. Why? Our best guess is that the random forest, decision tree, and neural net take more factors into account than merely presence and absence, like naive Bayes. A random forest will subdivide and dive into those subdivisions. A neural net combs back and forth, redistributing weights. The more simplistic algorithms may be capable of getting similar numbers, but the data would most likely need to be presented differently.

Electric Dream Machine CS3270 Spring 2024 Kaggle Report
(T. Lamont, N. Sacandy, D. Emmons)

There were a couple of ways we selected our model hyperparameter configurations. The neural network took by far the longest to run, so we only tested 5 sets of hyperparameters at first. Once the model finished running, there was an issue where the maximum iterations would be reached before the iterations converged. We tried increasing the maximum iterations so that the data would always converge, but this led to lower accuracy. As we increased the iterations higher and higher, overfitting became a larger problem, leading to lower accuracy; as such, we kept the max iterations to a reasonable number and saw improvements as a result. For all the other models, we used sklearn's GridSearchCV. this allows us to put in a few options for each model's parameters and test every possible combination. Essentially, this leads to not having to guess at which parameters might work best at the expense of computational resources. This tradeoff seemed worthwhile; for instance, we could test 480 configurations of decision trees and see the top 5 configurations of hyperparameters.

Our Top Three configurations were:

Multi-Layer Perceptron (Hidden Layers = 100, Max Iterations = 300, Alpha = 0.0001, Random State = 3270) 80.13% Average CV Accuracy, 81.14% Final Dataset Accuracy

Random Forest (n estimators = 300, min samples split = 2, min samples leaf = 1, max features = 'sqrt', max depth = 10, random state = 3270) 79.17% Average CV Accuracy, 80.28% Final Dataset Accuracy

Logistic Regression (classifier C = 100, classifier max iterations = 300, classifier solver = liblinear) 78.43% Average CV Accuracy, 79.29% Final Dataset Accuracy

The results on the final dataset compared to our results in stratified testing imply that we did a good job of not overfitting the test data.

As you can see from the results, Multi-Layer Perceptron emerged as our top performer. However, we had one last trick up our sleeves.

Combining our top three configurations, we created an ensemble model that would take predictions from each configuration, and provide a single prediction based on the majority vote. This meta-algorithm gave us our best results:

80.71% Accuracy and a Placement of 335 out of 2572.