Udacity Machine Learning Engineering Nanodegree
Capstone Project

Gary Foreman

Domain background:

Gamma rays (the highest energy forms of light) give astronomers insight into some of the most extreme events in our universe. Given their high energies, gamma rays are not produced as part of normal thermal processes, e.g., stellar fusion, because the temperatures required would be much too high. Rather, gamma rays are produced in non-thermal processes such as black hole accretion and shockwaves created by stellar explosions (also known as supernovae).

Gamma rays lie well beyond the range of light that is visible to our human eyes, and their detection requires substantial efforts of science and engineering. Gamma-ray telescopes rely upon measurements of interaction products created when gamma rays pass through another medium. For ground-based telescopes such as the one providing data for this capstone, that medium is Earth’s atmosphere, whereas for space-based telescopes, heavy metals are used to induce electron-positron pair creation (beyond the scope of this capstone).

(Side note: this project of interest to me because I used measurements from a gamma-ray telescope to carry out my PhD research. I have been removed from the field for a few years, but this project will be a nice way to reconnect.)

Problem Statement:

The detection of gamma rays requires the filtering of noise from other high energy, charged, particles, usually protons and electrons (also known as cosmic rays). Cosmic rays produce similar, but not exactly the same, characteristics within a gamma ray detector; however, cosmic rays originate from different sources, and confuse the signal from gamma ray sources. This cosmic-ray confusion makes gamma ray sources appear brighter than they truly are if the cosmic-ray signal is not properly removed.

The Major Atmospheric Gamma Imaging Cherenkov (MAGIC) Telescopes form a ground-based array used for measuring gamma-ray light. Ultimately, they detect interaction products from gamma rays and particles in Earth’s atmosphere. As mentioned above, cosmic rays produce similar interaction products, but their signal must be removed before accurate gamma ray measurements can be made. Here, we build a product to distinguish gamma rays from cosmic rays based on data from MAGIC.

Data Set:

We leverage the publicly available MAGIC Gamma Telescope Data Set to build our model. This data is provided by the University of California Irvine Machine Learning Repository. Original work with this data set dates back to Bock et al. (2004), when scientists were experimenting with machine learning models to use as part of MAGIC’s data preprocessing pipeline.

As documented on the University of California Irvine site, there are 19,020 rows in this data set; 12,332 of which are gamma-ray events and 6,688 of which are cosmic-ray events. The target is imbalanced with 65% of records being gamma rays. All feature columns are continuous.

In Figure 1, we plot feature distributions compared against each other and their correlation with the target variable. The plot below does indicate a degree of correlation between some of our features, most notably Conc and Conc1. Correlation will not necessarily be a concern for the models built here, but would be a problem if we were interested in investigating feature importance. The univariate distributions plotted on the diagonal do not indicate any particularly strong correlation between individual features and the target. This suggests a well-performing model will likely need to leverage multiple features simultaneously to make predictions.

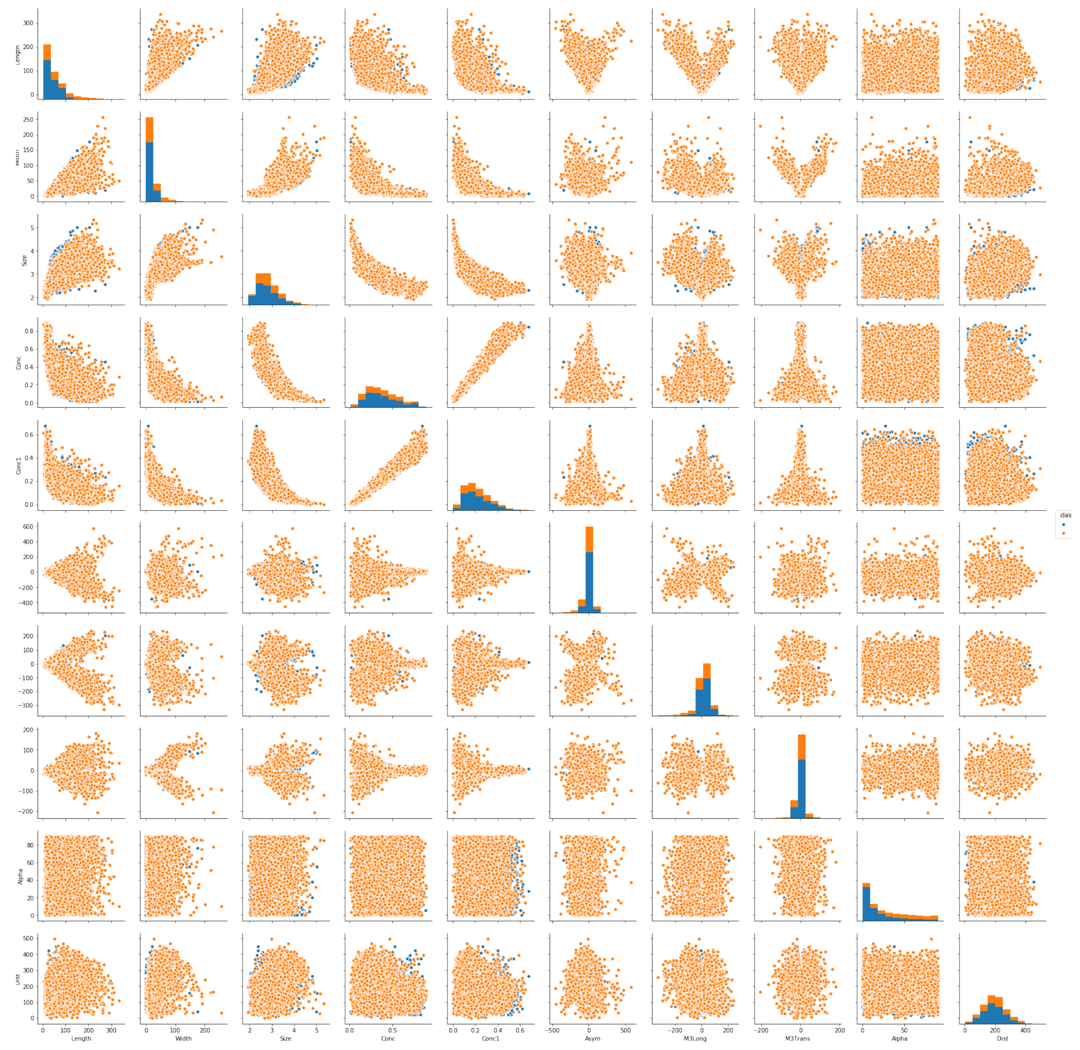


Figure 1: Pairwise feature plot (please see the eda/eda.ipynb notebook for code used to generate this plot).

In Figure 2, we compute and visualize the Pearson correlation coefficients of our feature variables. As seen in the previous plot, the Conc and Conc1 features are highly correlated. Because we are not specifically interested in determining feature importance here, we note the correlation, but keep both features in our training data set.

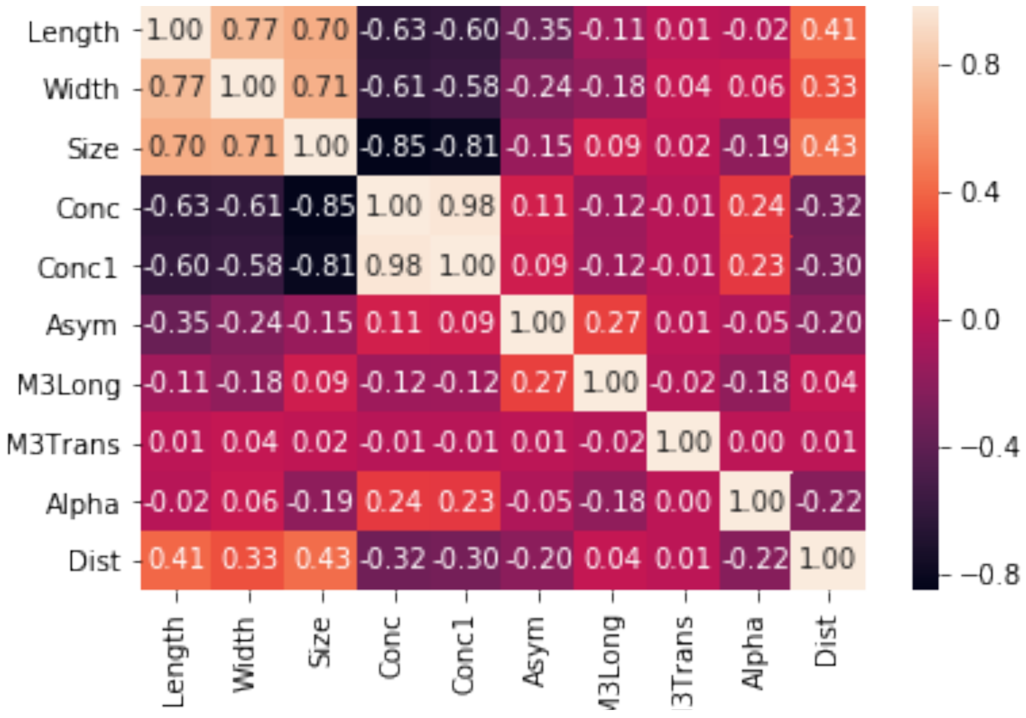


Figure 2: Pearson's correlation coefficient (please see the eda/eda.ipynb notebook for code used to generate this plot).

For the k-nearest neighbors and logistic regression algorithms presented below, features must be scaled so that they are distributed over approximately the same range of values. We can see from the pair plot above that the Conc feature, for example, lies in a range of about 0 to 1, whereas the Dist feature lies in a range from 0 to 500. Left unadjusted, these types of discrepancies will lead to poor performance of the k-nearest neighbors and logistic regression algorithms. However, this is not an issue for the tree-based gradient boosting classifier algorithm.

Benchmark models:

The benchmark models reported here were trained in the eda/eda.ipynb notebook that can be found in the GitHub repository that accompanies this report.

As a benchmark, we train a k-nearest neighbors classifier. We optimize the n_neighbors hyperparameter using 4-fold cross validation. The results of the grid search indicate that the k-nearest neighbors algorithm performs surprisingly well. The optimal mean test AUC of just under 0.9 across validation folds was achieved using 37 neighbors.

We also train a logistic regression classifier. We optimize the L2 regularization strength hyperparameter C using 4-fold cross validation. Logistic regression performs substantially worse than k-nearest neighbors, which may be due to what we discovered in the pairwise plots above: individual features do not distinguish the target very well. We see below that performance is independent of regularization strength probably because our model includes so few features. The logistic model consistently performs at just under 0.84 AUC across validation folds.

Finally, we train a gradient boosting classifier. Using 4-fold cross validation, we optimize the n_estimators, learning_rate and max_depth hyperparameters. An optimal solution with mean test AUC of 0.93 is achieved using a learning_rate of 0.1, a max_depth of 5, and with 300 n_estimators. This model performs quite a bit better that the k-nearest neighbor model trained above. We propose this model for deployment via Amazon Sagemaker, where we will retrain using their xgboost implementation.

Evaluation metrics:

We test performance of the three models presented above using a test set created by holding out a random 20% of the original data. We find AUC of 0.84, 0.90, and 0.94 for the logistic regression, k-nearest neighbors, and gradient boosting classifier, respectively; these results are consistent with the AUCs reported while performing cross validation during hyperparameter optimization.

We also use this random test set to create Figure 3, the the receiver operating characteristic (ROC) curve for the three models trained here. This plot is meant to be directly compared with Figure 5 of Bock et al. (2004). Black vertical lines are places at 1%, 2%, 5%, 10%, and 20% false positive rate to highlight key thresholds outlined under the Attribute Information section of the MAGIC Gamma Telescope Data Set site hosted by the University of California, Irvine. These results are presented in Tables 1 through 3. Our gradient boosting model outperforms all methods presented by Bock et al. (2004) at each of the required false positive thresholds.

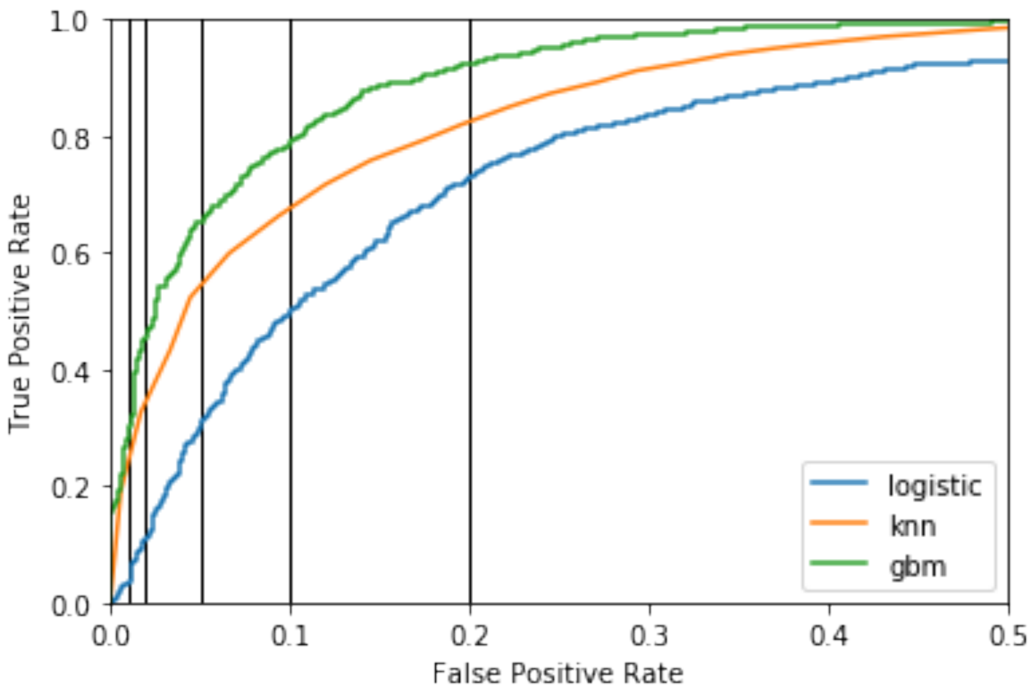


Figure 3: Receiver operating characteristic. Black vertical lines indicate thresholds identified by the data contributors. (Please see the eda/eda.ipynb notebook for code used to generate this plot.)

False Positive Rate	True Positive Rate
0.01	0.03
0.02	0.11
0.05	0.29
0.10	0.49
0.20	0.73

Table 1: ROC for logistic regression model.

False Positive Rate	True Positive Rate
0.01	0.19
0.02	0.33
0.05	0.52
0.10	0.66
0.20	0.82

Table 2: ROC for k-nearest neighbors model.

False Positive Rate	True Positive Rate
0.01	0.27
0.02	0.45
0.05	0.64
0.10	0.78
0.20	0.92

Table 3: ROC for gradient boosting classifier.

Production Model:

The code to develop and deploy the production model to AWS can be found in the producton_model/production_model.ipynb notebook in the GitHub repository that accompanies this document.

We port the optimized hyperparameters determined in the previous section for our gradient boosting classifier model to an xgboost model trained using the Amazon SageMaker interface. As a side note, we chose not to perform exploratory data analysis and initial model training/hyperparameter optimization using SageMaker because throughout this course, we found SageMaker inhibits the iterative process. The overhead associated with working in the cloud, especially for a data set of this size, didn't makes sense and development was faster on our laptop.

To perform retraining of the production model on AWS, we needed to reformat our training and testing data sets for compliance with S3 and the requirements of SageMaker's implementation of the XGBoost algorithm. This includes uploading csv files for both the train and test sets. These files must contain no headers or row indices, and the train set must have the target as its first column.

We leverage an image based on the XGBoost algorithm for the deployment of our production model. As reported above, we leverage hyperparameters optimized during local training for our deployed model so that we do not need to use SageMaker for the iterative portion of our model build process. We set max_depth=5, eta=0.1 (eta corresponds to learning_rate), and num_rounds=300 (num_rounds corresponds to num_estimators); we leave all other parameters set to default. While the Scikit-Learn and XGBoost implementations are based on the same underlying Gradient Boosting algorithm, their exact implementations differ in the details. We were encouraged to find consistent performance of both implementations; the AUC computed on the test set using our XGBoost model trained on SageMaker is 0.94.

Next we set up the infrastructure for an API that can be used to send data to and receive predictions from our deployed model. The first necessary decision was how we expect input data to be formatted. We have chosen a JSON format as follows:

'{“Length”:21.0483,“Width”:17.6112,“Size”:2.4409,“Conc”:0.4529,“Conc1”: 0.2373,“Asym”:29.591,“M3Long”:-9.6241,“M3Trans”:14.2623,“Alpha”:32.4415,“Dist”: 173.549}'

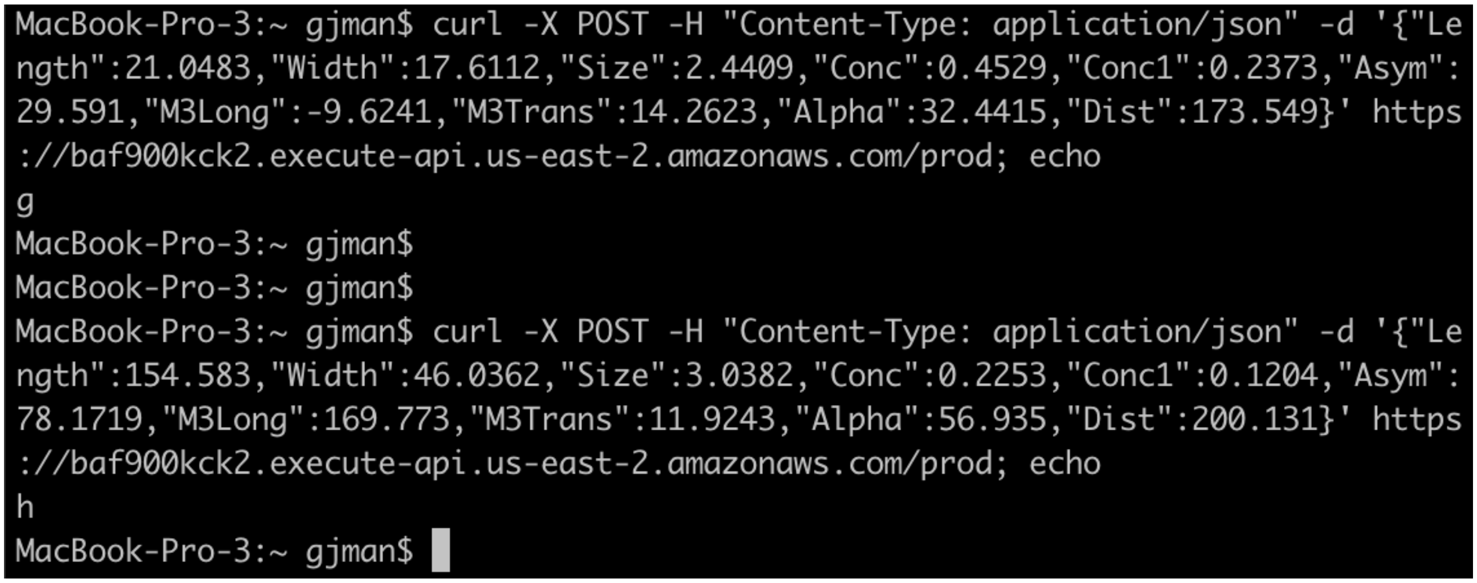
with the feature name as the key and the feature itself as the value. We the wrote a method that parses this JSON string to produce a csv string of simply the feature values ordered as expected by the model. We run through the prediction steps for a single event from our test set (the one listed above) as a means to determine the necessary code to include in our Lambda function.

Finally, we include the code we implement in our Lambda function at the end of production_model/production_model.ipynb. This function is responsible for parsing the input JSON string mentioned above, transforming that string to a format consumable by our model endpoint, calling the model, and returning the result to the API gateway for display to the end user. Our Lambda function returns a “g” for gamma ray (signal) if the model prediction probability is above 0.5, otherwise it returns “h” for hadron (noise).

Our model API is hosted at https://baf900kck2.execute-api.us-east-2.amazonaws.com/prod, and is intended to be called using curl with a POST command. Figure 4 shows two example API calls including the response. The first call is an example from our test set that is labeled “g,” and the second is an example from our test set labeled “h.” We see in both cases that the model prediction is correct. We will leave the model, Lambda function, and API deployed until a response is received from our reviewer. This way they reviewer may test the API if they choose.

Conclusion:

We have trained and deployed a gradient boosting classifier to distinguish gamma-ray from cosmic-ray (i.e. hadron) events. While our model was optimized locally using Scikit-Learn’s implementation of this algorithm, we found optimal hyperparameters translated will to SageMaker’s XGBoost implementation. In both cases, the AUC on our test set was 0.94. This is well above anything that was achieved by Bock et al. (2004), and is also comfortably above our two benchmark models, logistic regression(AUC of 0.84) and k-nearest neighbors (AUC of 0.9). We have also written and deployed an API interface for our model using an AWS Lambda function and the AWS API Gateway. The API is hosted at https://baf900kck2.execute-api.us-east-2.amazonaws.com/prod and can be called using the curl unix function.



```
MacBook-Pro-3:~ gjman$ curl -X POST -H "Content-Type: application/json" -d '{"Le
ngth":21.0483,"Width":17.6112,"Size":2.4409,"Conc":0.4529,"Conc1":0.2373,"Asym":
29.591,"M3Long":-9.6241,"M3Trans":14.2623,"Alpha":32.4415,"Dist":173.549}' https
://baf900kck2.execute-api.us-east-2.amazonaws.com/prod; echo
g
MacBook-Pro-3:~ gjman$
MacBook-Pro-3:~ gjman$
MacBook-Pro-3:~ gjman$ curl -X POST -H "Content-Type: application/json" -d '{"Le
ngth":154.583,"Width":46.0362,"Size":3.0382,"Conc":0.2253,"Conc1":0.1204,"Asym":
78.1719,"M3Long":169.773,"M3Trans":11.9243,"Alpha":56.935,"Dist":200.131}' https
://baf900kck2.execute-api.us-east-2.amazonaws.com/prod; echo
h
MacBook-Pro-3:~ gjman$
```

Figure 4: POST requests to the API hosting the XGBoost model trained on the MAGIC-gamma data. The first request is an example from the test set that is labeled “g,” and the second request is a test set record labeled “h.” Our model predicts both cases correctly. Note: the echo command is included purely so that a new line character is created after the API response; this is for readability purposes only.