Hazardous Asteroid Classification

Achieving 0.99 Recall on NEOs

Peter Brown
brown.pet@northeastern.edu

William Emerson Tower IV
william.emerson.tower@gmail.com

22 August 2022

Contents

1 Introduction

The National Aeronautics and Space Agency(NASA) maintains a list of Near Earth Objects (NEO) to monitor for impact risk and danger to planet Earth. A dataset is made available for analysis. The group has built and analyzed a number of classifiers to identify potentially dangerous asteroids. Considering the danger a large asteroid poses to Earth on impact, correct detection is important. Using data collected by space agencies we construct several classifiers to determine asteroids which are potentially hazardous.

2 Data Preprocessing

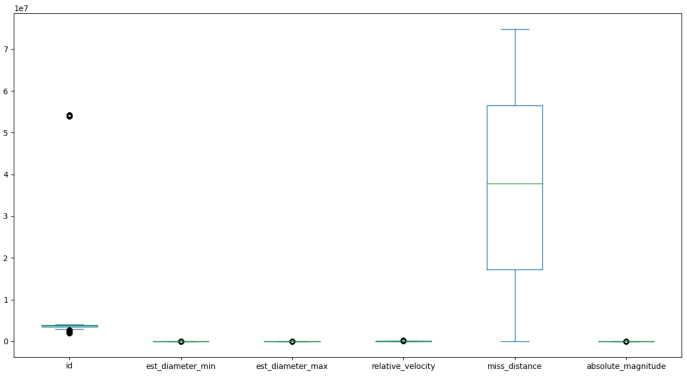
The raw asteroids dataset is available at kaggle.

The analysis begins by importing the dataset as a pandas dataframe.

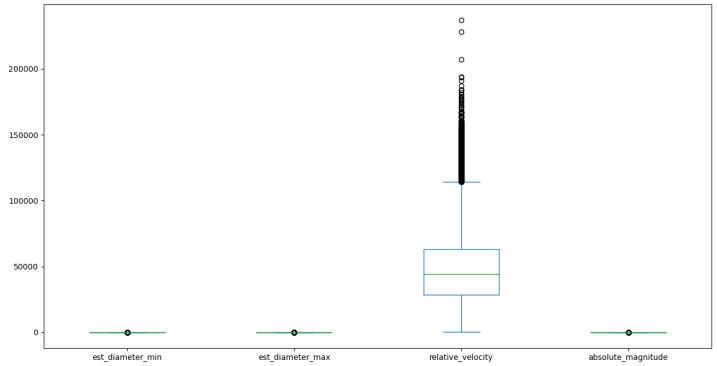
asteroids.info() reveals:

```
    Data columns (total 10 columns):
 #   Column              Non-Null Count  Dtype
---   ------              --------------  -----
 0   id                  90836 non-null  int64
 1   name                90836 non-null  object
 2   est_diameter_min    90836 non-null  float64
 3   est_diameter_max    90836 non-null  float64
 4   relative_velocity   90836 non-null  float64
 5   miss_distance       90836 non-null  float64
 6   orbiting_body       90836 non-null  object
 7   sentry_object       90836 non-null  bool
 8   absolute_magnitude  90836 non-null  float64
 9   hazardous           90836 non-null  bool
dtypes: bool(2), float64(5), int64(1), object(2)
memory usage: 5.7+ MB
```

and examining the distributions as a boxplot gives:



It appears the miss distance feature spans several orders of magnitude more than any other feature, it should be dropped to get a better idea of what the other features look like. Regenerating the boxplot we find:



All our features are of highly disparate magnitude. A numerical inspection of the data would be more revealing:

	id	d. min	d. max	rel. velocity	miss distance	abs. magnitude
count	90836	90836	90836	90836	90836	90836
mean	1.438288e+07	0.127	0.284947	48066.918	3.706e+07	23.527
std	2.087202e+07	0.298	0.667491	25293.296	2.23e+07	2.894
min	2.000433e+06	0.0006	0.001362	203.346	6.745e+03	9.23
25	3.448110e+06	0.019	0.043057	28619.020	1.721e+07	21.34
50	3.748362e+06	0.048	0.108153	44190.117	3.784e+07	23.7
75	3.884023e+06	0.143	0.320656	62923.604	5.654e+07	25.7
max	5.427591e+07	37.892	84.730541	236990.128	7.479e+07	33.2

It is clear this data will require scaling to work with many machine learning models. It is also obvious there are several unnecessary columns present, namely 'id', 'name', 'orbiting_body' and 'sentry_object'. The unnecessary columns are dropped from the dataframe, and the set is passed to an instance of SKlearn's train_test_split, generating train and test sets at a 3:1 data ratio. A standard scaler is fit on the training data, and then used to transform training and test data.

3 Dataset Considerations

Preliminary inspection of the data revealed that features are of vastly different magnitudes, but what does the class distribution look like? Running the snippet:

```
neg, pos = np.bincount(asteroids['hazardous'])
total = neg + pos
print('Total: {}\n    Positive: {} ({:.2f}% of total)\n'.format(
    total, pos, 100 * pos / total))
```

Shows us: Total: 90836 Positive: 8840 (9.73% of total)

This is problematic, our dataset is very unbalanced, so we'll need to be wary of metrics like accuracy, which may return high values despite performing poorly on the positive class. It is possible to remedy this by downsampling the data, specifying class weights, and selecting loss functions which account for the imbalance, each of which will be explored.

3.1 Success Metrics

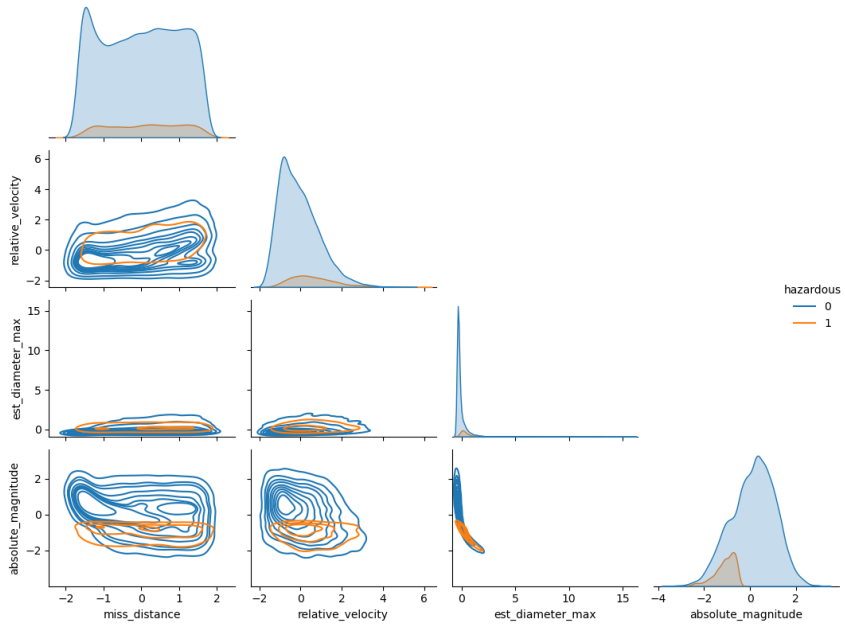
The really important thing about classifying hazardous asteroids is not to miss the dangerous ones. It's life and death! We want to worry about maximizing recall for the positive class, and then optimizing for whole dataset

accuracy under that, so that the model provides a useful filter for discarding obviously non-threatening space rocks.

4 Exploratory Data Analysis

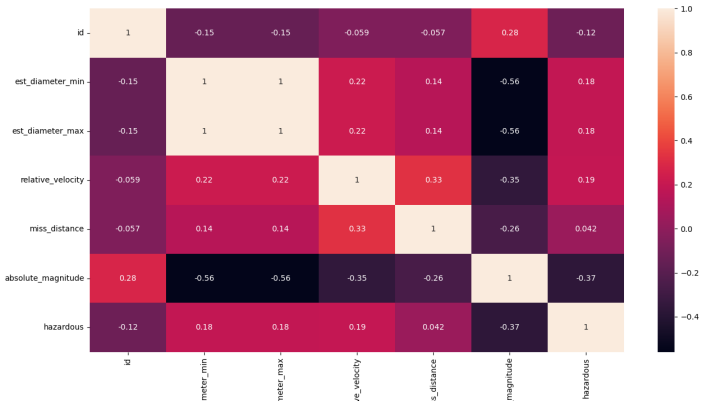
4.1 Dataset Visualization

A KDE pairplot illuminates a number of important features about the dataset:



From this we can glean that absolute magnitude, diameter, and miss distance are very useful features.

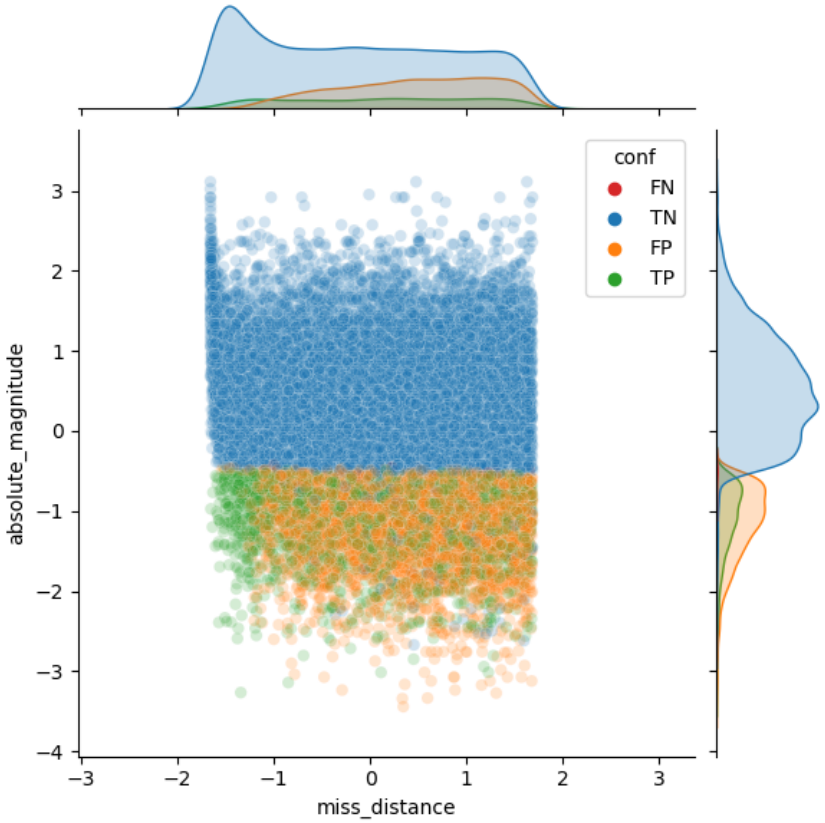
4.2 Dataset Correlation

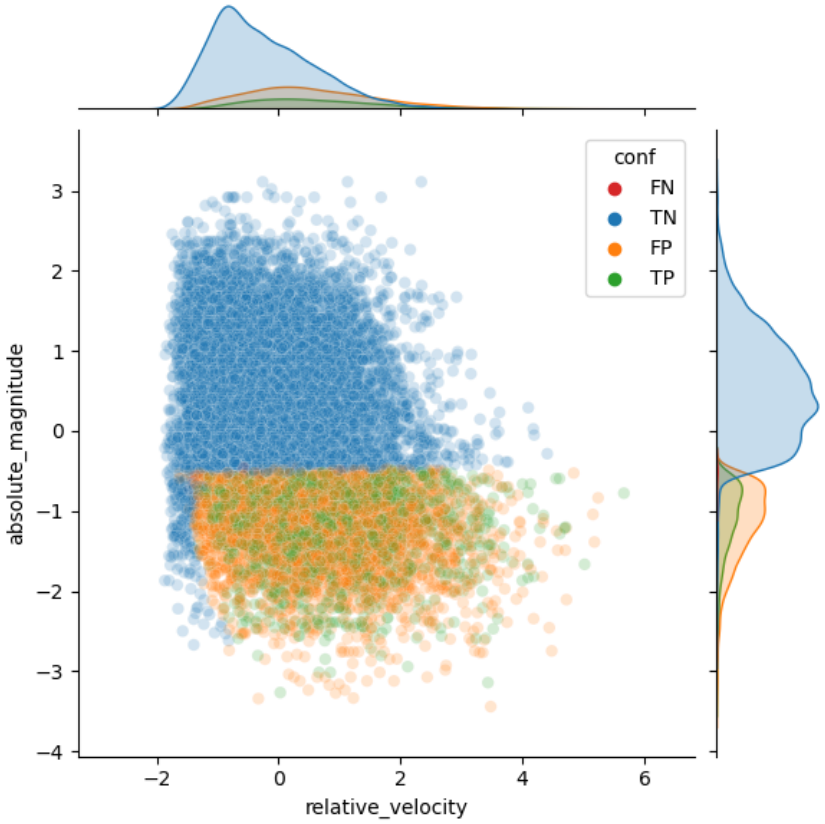


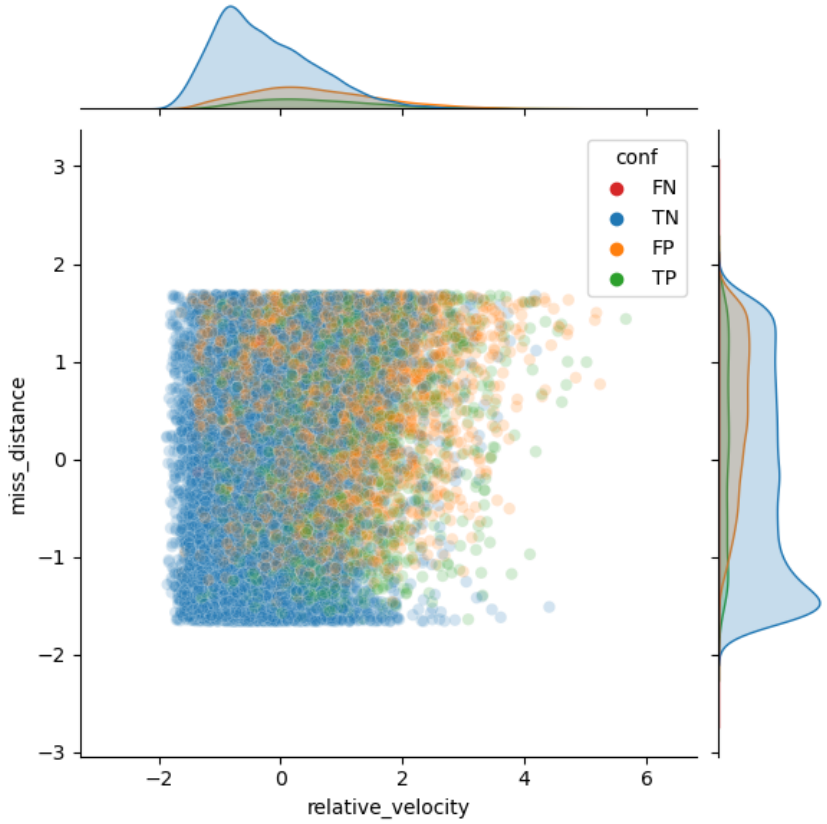
The correlation heatmap confirms eyeball-estimation from previous section about relative feature importance, it appears absolute magnitude is the strongest feature for determining if an object is hazardous. It is also clear that the estimated diameter variables are tied together.

4.3 Visualization from Toy Models

Examining some results from a neural net with high recall which appears later, it is possible to generate clear visualizations of the data, giving an impression of where the positive class is, and which section of the negative class is misidentified as positive.

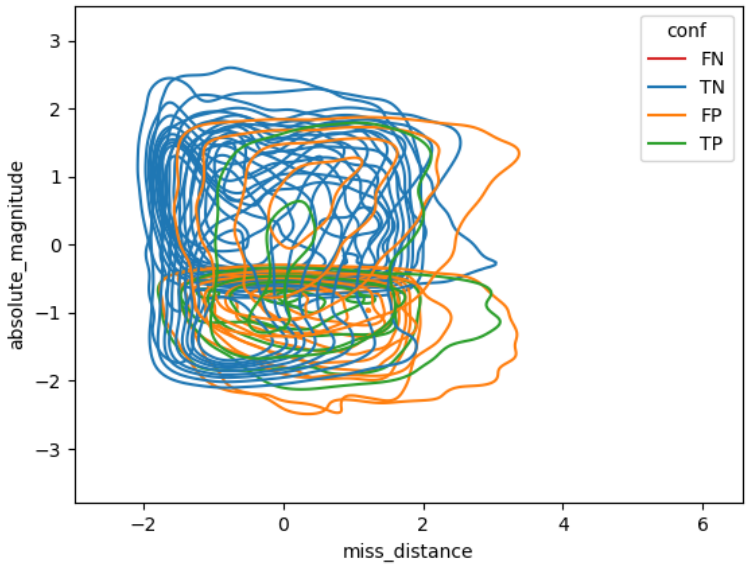

conf
FN
TN
FP
TP
absolute_magnitude
miss_distance
3
2
1
0
−1
−2
−3
−4
−3
−2
−1
0
1
2
3

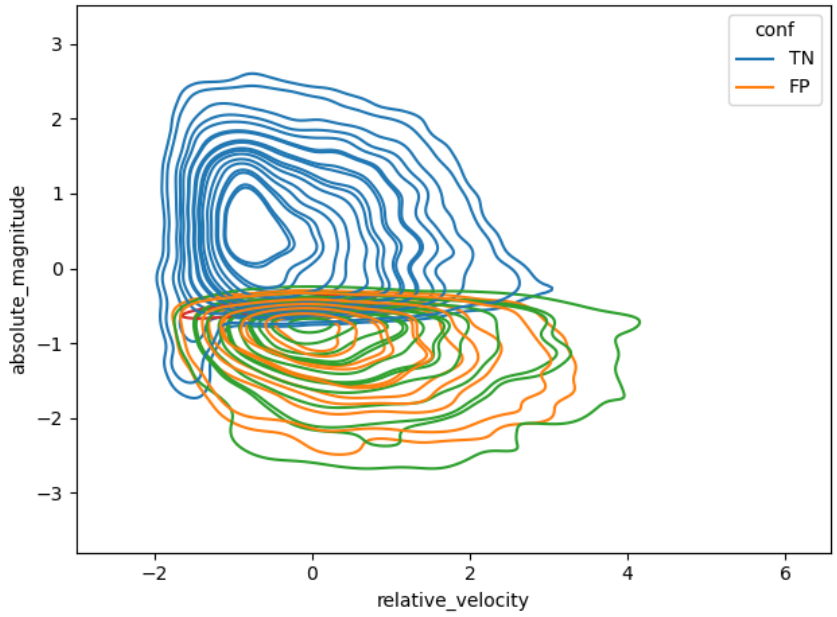

conf
FN
TN
FP
TP
absolute_magnitude
relative_velocity
3
2
1
0
−1
−2
−3
−4
−2
0
2
4
6

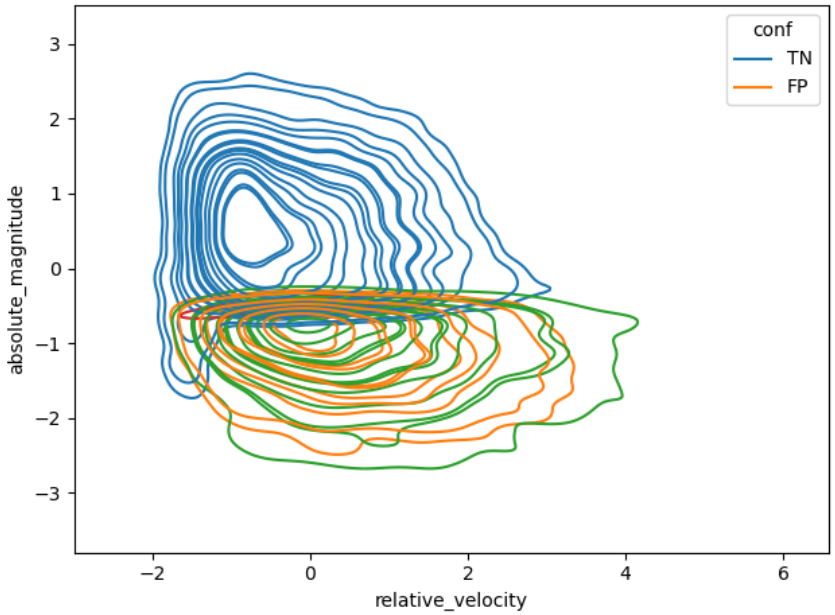


Check out those sharp lines on the absolute magnitude axes! The neural net used to generate these classifications has very high recall on this dataset, so we can be sure that the line represents a clear boundary for marking a body hazardous. It also appears there's some consideration toward an object moving substantially quickly away from earth being non-hazardous.

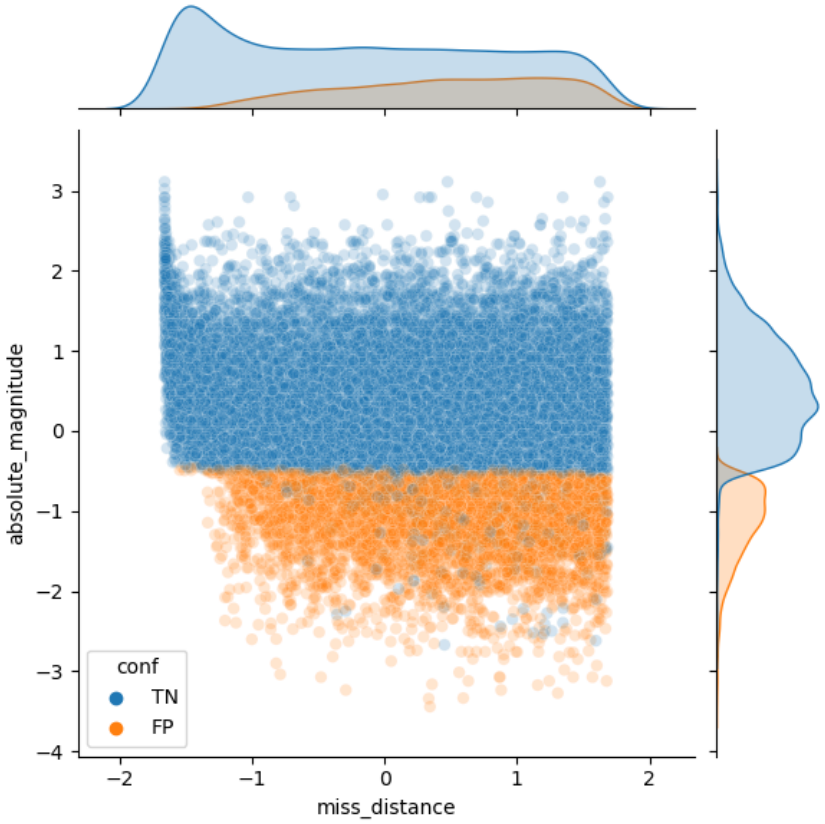
KDE plots of the classified data provide some further insight into what the contouring of the data looks like:

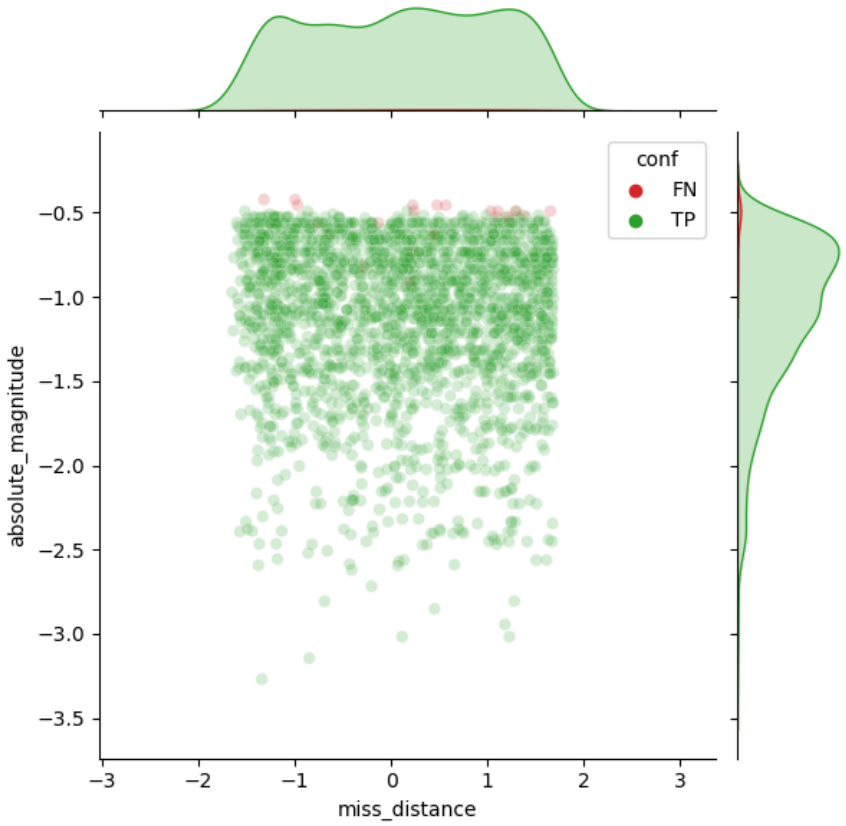

conf
FN
TN
FP
TP
absolute_magnitude
miss_distance
3
2
1
0
−1
−2
−3
−2
0
2
4
6

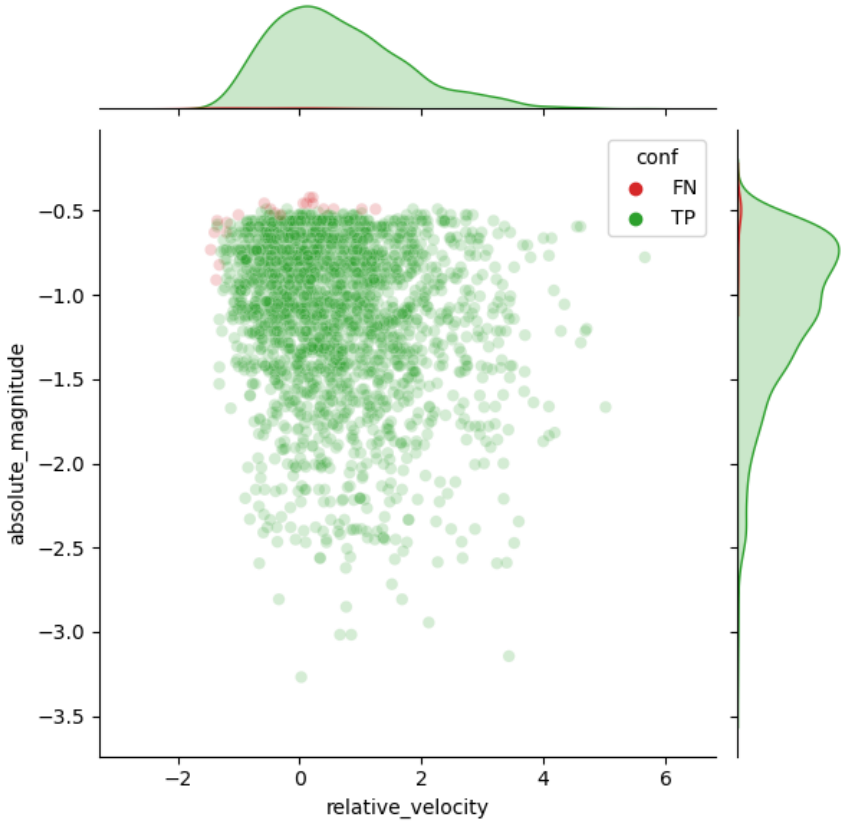

conf
TN
FP
absolute_magnitude
relative_velocity
3
2
1
0
−1
−2
−3
−2
0
2
4
6

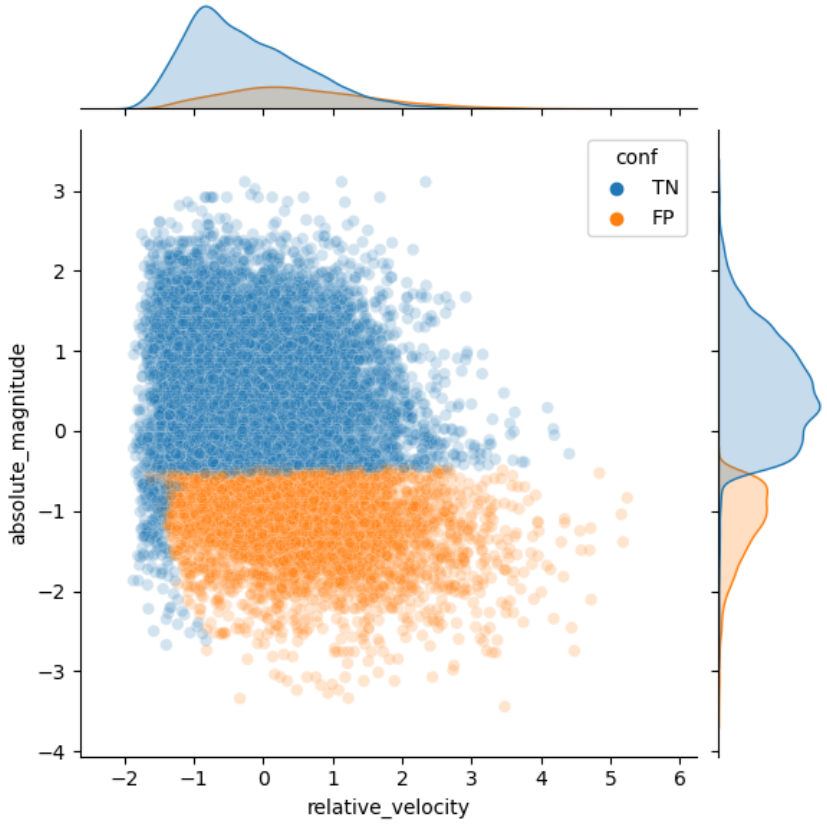


And to increase clarity, pairplots bifurcated into positive class only and negative class only:


absolute_magnitude
miss_distance
conf
TN
FP
3
2
1
0
−1
−2
−3
−4
−2
−1
0
1
2


conf
FN
TP
absolute_magnitude
−0.5
−1.0
−1.5
−2.0
−2.5
−3.0
−3.5
miss_distance
−3
−2
−1
0
1
2
3


conf
FN
TP
absolute_magnitude
−0.5
−1.0
−1.5
−2.0
−2.5
−3.0
−3.5
−2
0
2
4
6
relative_velocity



From these visualizations, it's very clear that the positive and negative classes have a significant overlap, and that we will have to make some active choices about the precision-recall tradeoff.

5 Downsampling

Because our dataset is unbalanced, we need to apply a correction, perhaps by sampling each class differently. We can undersample the non-hazardous class, or we can oversample the hazardous class. We do this to prevent the classifier's from ignoring the smaller class (several initial models did this, to our amusement). For example, a model that guesses "False" on this dataset achieves 90% accuracy, which is good as far as default param SKlearn is concerned. However, this would never detect any hazardous asteroids, and the model would be worthless. While undersampling is a valid technique for countering this issue, it has faults. It is possible to oversample the smaller

class (in this case, the hazardous class), and as a result will drag down the accuracy of the classifier as it attempts to classify a smaller class more frequently than it should. We define class weights for each classifier through Cross Validation to ensure that we do not encounter this issue.

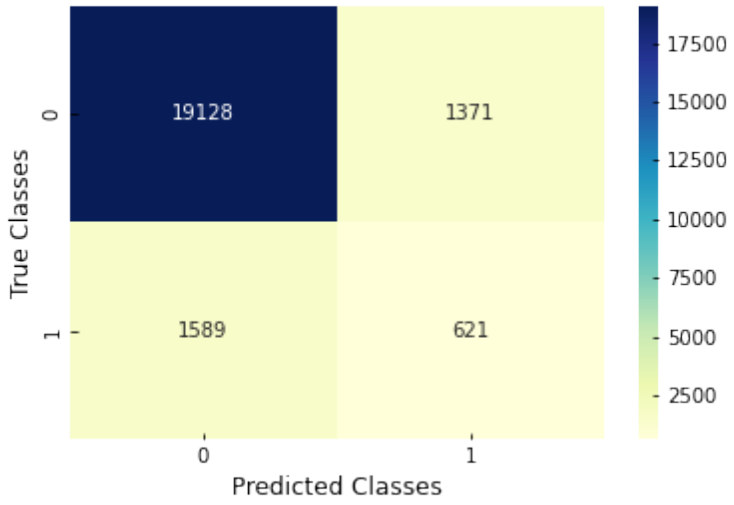
6 Logistic Regression Classifier

Since the asteroid dataset is a binary classification problem (an asteroid is either hazardous or not), we can use a logistic regression model to classify each asteroid as safe or not. Using the LogisticRegression class in sklearn, we can quickly construct a regression model to train on our data. First, we would like to tune some of the hyperparameters of the model. The two we chose to tune were λ and class_Weights. The λ, or "C" parameter, focuses on the regularization of the dataset, and helps to prevent overfitting or underfitting the model. Class weights are assigned to the hazardous/non-hazardous data classes to weight them properly when training. This is to ensure that the regressor does not overfit the non-hazardous dataset, or underfit the hazardous dataset. To determine the best parameters for the logistic regressor, we ran a grid search Cross Validation using K-Fold Cross Validation. From this validator, we can determine the best values to use are $\lambda = 0.8$, and the class weights as $\{$ True : 0.2, False : 0.8$\}$. Training a Logistic Regression Classifier using these parameters gives us the following results:

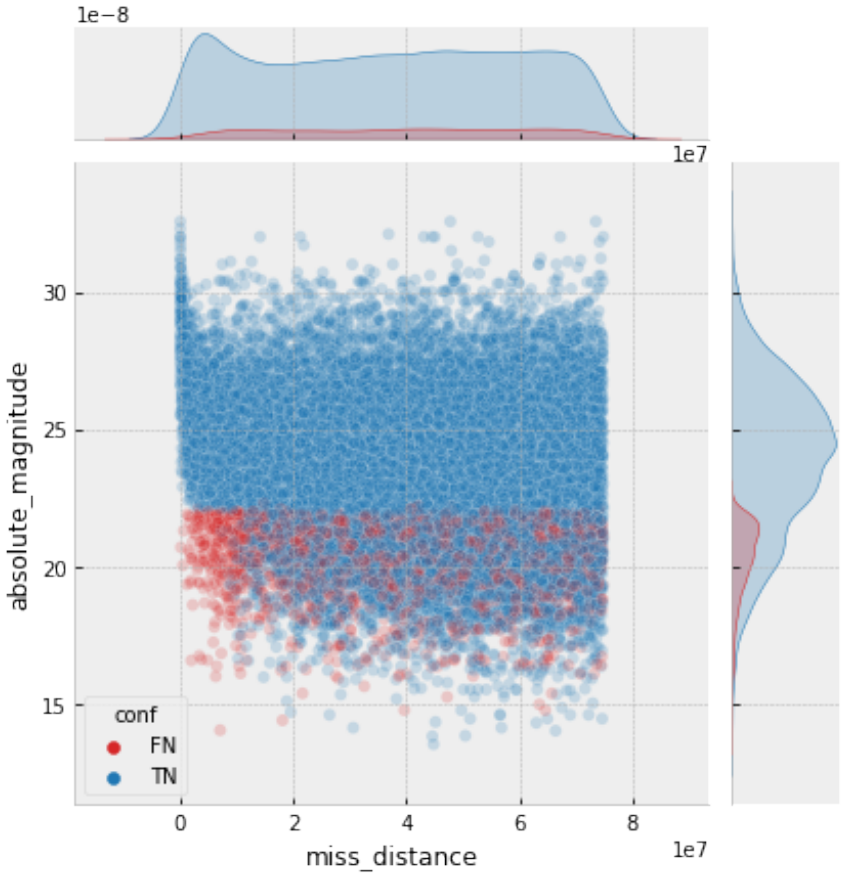
	precision	recall	f1-score	support
False	0.92	0.93	0.93	20499
True	0.31	0.28	0.30	2210
accuracy			0.87	22709
macro avg	0.62	0.61	0.61	22709
weighted avg	0.86	0.87	0.87	22709

From the metrics provided by sklearn, we see the logistic regression classifier performs well on classifying non-hazardous asteroids, with both precision, recall, and an f1 score above 90% for class 1. For predicting the hazardous asteroids, the model fails to do so with any accuracy. All three metrics to measure the accuracy of the model are below 32%, making this worse than a coin flip. In fact, if we look at the confusion matrix output of the Logistic

Regression classifier, we can see that it has more False Positives and False Negatives than it does True Positives.



While the Logistic Regression Classifier serves as a decent benchmark, there is room for improvement in accuracy and precision for the hazard class.



False Negatives vs True Negatives from Logistic Regression Classifier

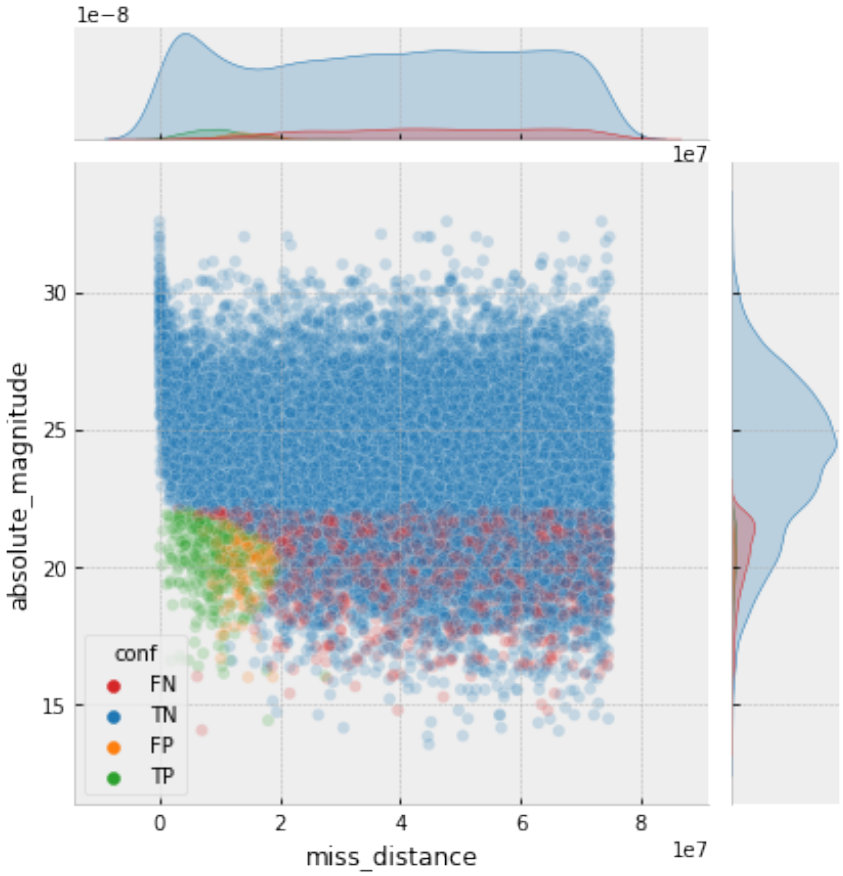
7 SVC Classifier

The SVC classifier is favorable for this dataset for a few reasons, but it does have problems as well. Typically, an SVC can perform well in edge case scenarios where even a more advanced model such as a neural network would be incorrect. However, SVCs tend to favor the majority class in a dataset, which is not what we want when our dataset it split 90/10. This can be offset by tuning the hyperparameters λ and class_weights. Similarly to the Logistic Regression Classifier, the SVC can tune its regularization parameter. Using the same method as before, we use Grid search Cross Validation to arrive at $\lambda = 1.2$ and class_weight = True : 0.6, False : 0.4.

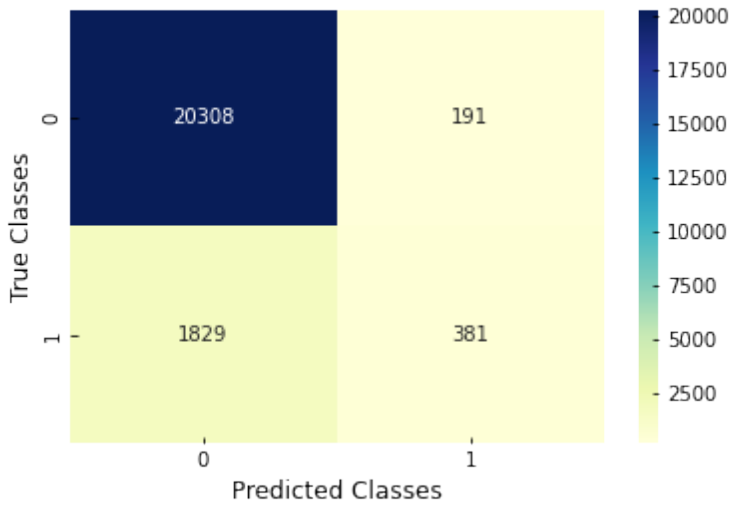
With these values, the SVC was trained and produced these results:

	precision	recall	f1-score	support
False	0.92	0.99	0.95	20499
True	0.67	0.17	0.27	2210
accuracy			0.91	22709
macro avg	0.79	0.58	0.61	22709
weighted avg	0.89	0.91	0.89	22709

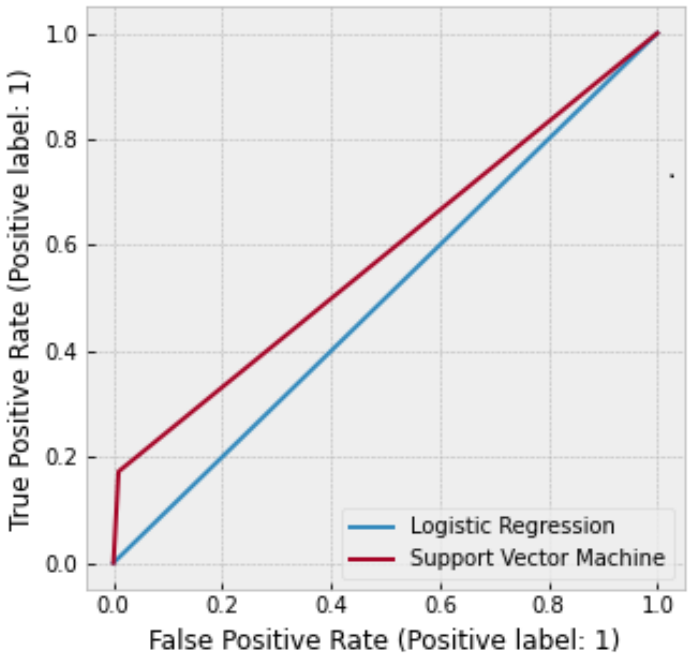
As expected, the SVC performs better on the majority non-hazardous class, with a 99% recall and a 95% f1 score. However, it performs abysmally on the hazardous class, scoring just 17% for recall. Overall, the model has 79% precision, 58% recall, and 61% f1 with a 91% accuracy. Compared to the Logistic Regression Classifier, the SVC attains a higher accuracy at the cost of some recall.



By plotting the predictions of the SVC with respect to miss distance and absolute magnitude, we can see where the model struggles. For the most part, the SVC is able classifying True Negatives accurately, but struggles to classify True Positives. This is also reflected by the confusion matrix for the SVC



8 ROC Curve of Models



This is the ROC curve of both the Logistic Regression model as well as the Support Vector Machine. From this curve, we can see that the Logistic Regression model is about as helpful as a coin flip, producing no better results than a coin flip. While the SVM produces slightly better results, they both clearly are inefficient models to classify this dataset. We now define a neural net that will classify the asteroid dataset with higher precision and accuracy than either of these models.

9 Neural Net Implementation

9.1 Naive Attempt

Code for the naive implementation of our neural net:

```
# does what it says on the box
METRICS = [
keras.metrics.TruePositives(name='tp'),
```

```
keras.metrics.FalsePositives(name='fp'),
keras.metrics.TrueNegatives(name='tn'),
keras.metrics.FalseNegatives(name='fn'),
keras.metrics.BinaryAccuracy(name='accuracy'),
keras.metrics.Precision(name='precision'),
keras.metrics.Recall(name='recall'),
keras.metrics.AUC(name='auc'),
keras.metrics.AUC(name='prc', curve='PR'), # precision-recall curve
]

# good for wrapping our model in sklearn API, quick builds
def build_model(nNeurons=30, lr = .035, input_shape=[5], gamma=2):
model = Sequential()
model.add(keras.layers.InputLayer(input_shape=[5]))
model.add(keras.layers.Dense(nNeurons, activation='tanh'))
model.add(keras.layers.Dense(nNeurons, activation='tanh'))
model.add(keras.layers.Dense(nNeurons, activation='tanh'))
model.add(keras.layers.Dense(1, activation='sigmoid'))
optimizer = keras.optimizers.SGD(learning_rate=lr, momentum=0.9, \
nesterov=True, decay=1e-5)
model.compile(loss=BinaryFocalLoss(gamma=gamma), \
optimizer=optimizer, metrics=METRICS)
return model

#set up callbacks and misc. model variables
tensorboard = TensorBoard(log_dir="logs/{}".format((time())))
callback = [ \
tf.keras.callbacks.EarlyStopping(monitor='loss', patience=5), tensorboard]

class_weight = {
0 : 1,
1 : 10}

# train model
model = build_model(nNeurons=20, gamma=1)
keras_clf = KerasClassifier(model=model)

print(model.summary())

keras_clf.fit(X_train, y_train, epochs=45, \
```

```
validation_data=(X_test, y_test),  callbacks=[callback], class_weight = class_weight)
```

This model architecture was selected based on prior experience with binary classification tasks. The metrics blocks and weight variable are derived with inspiration from a TensorFlow tutorial on imbalanced datasets. The number of internal layers, neurons, and learning rate were rule-of-thumbed. The tanh activation function was selected for its ability to fit complicated curves, and the optimizer was selected for the excellent convergence properties of SGD with momentum and look-ahead.

9.2 Loss Function Selection

We had originally selected a binary-crossentropy loss function, but were disappointed with its performance on the dataset, even with the provided weights parameter. After some shopping around, we settled on a Binary Focal Loss metric, defined:

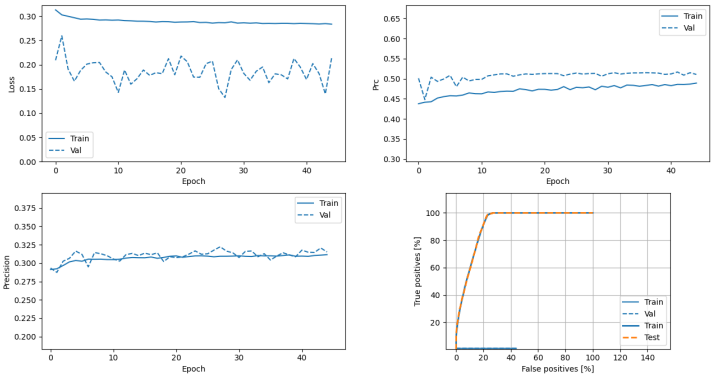
$$L(y, \hat{p}) = -\alpha y \, (1 - \hat{p})^{\gamma} \log(\hat{p}) - (1 - y)\hat{p}^{\gamma} \log(1 - \hat{p}) \tag{1}$$

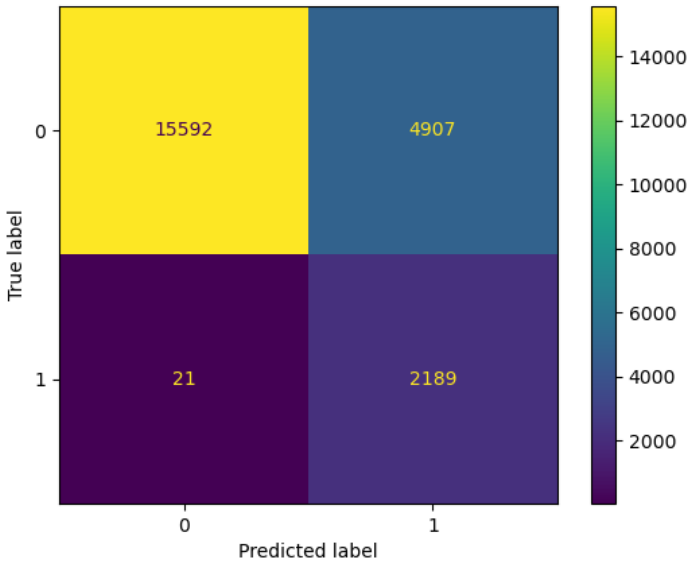
where γ is the focusing parameter, specifying how high-confidence correct predictions contribute to the overall loss (higher γ, translates to down-weighted easy-to-classify samples), and α is a hyperparameter that controls the precision-recall tradeoff by setting weights for errors on the positive class (α=1 is no weighting)[1]

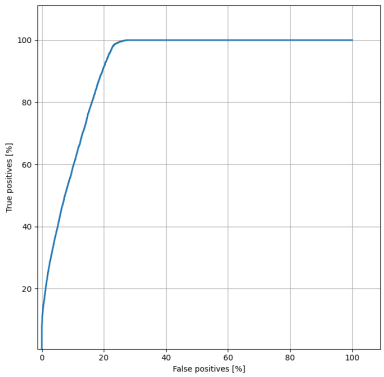
This loss allows us to focus on the hard to classify samples, and to value recall over precision, exactly what we want.

9.3 Base Model Performance

The base model does fairly well on a number of metrics, and was used to generate the plots seen in the Visualization from Toy Models subsection.

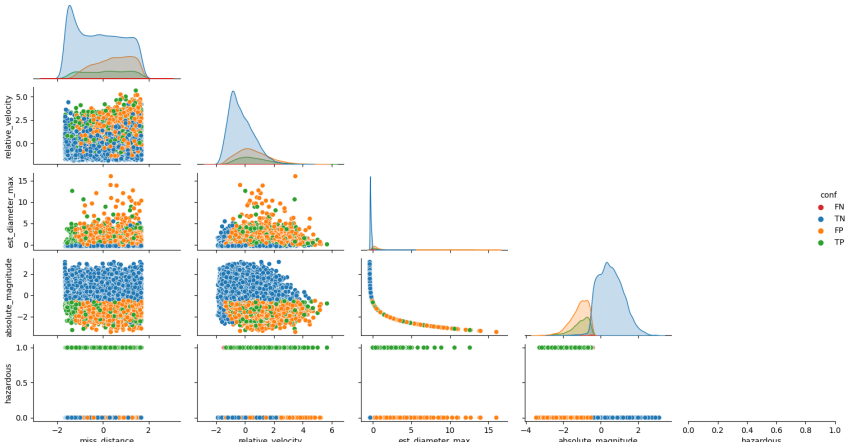

Loss
Epoch
Train
Val
Prc
Epoch
Train
Val
Precision
Epoch
Train
Val
True positives [%]
False positives [%]
Train
Val
Train
Test


15592
4907
21
2189
0
1
True label
0
1
Predicted label
14000
12000
10000
8000
6000
4000
2000



From a visual analysis of the ROC curve, it looks like achieving high precision will cost a false positive rate in the low 20s. From inspection of the confusion matrix, we can calculate a false postive rate for this model 23.9%, just as expected. Our classification report for this model is:

	precision	recall	f1-score	support
0	1.00	0.77	0.87	20499
1	0.31	0.99	0.48	2210
accuracy			0.79	22709
macro avg	0.66	0.88	0.67	22709
weighted avg	0.93	0.79	0.83	22709



Classifications and True Labels for Various Features

9.4 Hyperparameter Tuning with Keras Tuner

To select the best hyperparameters for the model, we pass it through a Keras Tuner.

```
def keras_build_model(hp):
model = keras.Sequential()
model.add(keras.layers.InputLayer(input_shape=[5]))
# Tune the number of layers.
for i in range(hp.Int("num_layers", 2, 4)):
model.add(keras.layers.Dense(hp.Int("units", min_value=15,\
max_value=30, step=5), activation='tanh'))
model.add(keras.layers.Dense(1, activation='sigmoid'))
lr = hp.Float("lr", min_value=1e-4, max_value=0.03, sampling="log")
gamma = hp.Float("gamma", min_value=0.05, max_value = 5, sampling="log")

optimizer = keras.optimizers.SGD(learning_rate=lr, \
momentum=0.9, nesterov=True, decay=1e-5)
model.compile(loss=BinaryFocalLoss(gamma=gamma), \
optimizer=optimizer, metrics=METRICS)
return model

import keras_tuner

# model = keras_build_model(keras_tuner.HyperParameters())

tuner = keras_tuner.RandomSearch(
hypermodel=keras_build_model,
objective=keras_tuner.Objective("val_recall", direction="max"),
max_trials=3,
executions_per_trial=5,
overwrite=True,
directory="asteroids_nn_tuning",
project_name="asteroids_tuning{}".format((time())),
)

tuner.search(X_train,
y_train,
epochs=10,
validation_data=(X_test, y_test),
callbacks=[keras.callbacks.TensorBoard("logs/tuner/{}".format((time())))],
```

```
class_weight = class_weight)
```

The above tuner works by calling the build_model function over the randomized search space, where the set of randomly tunable parameters is defined by various hp.* calls to build_model itself. We find the best parameters are:

```
best_hp = tuner.get_best_hyperparameters()[0]
best_hp.values
\{'num_layers': 3, 'units': 30, 'lr': 0.009263303924328512, 'gamma': 0.118359836150857
```

The tuner reports search space and results

```
Results summary
Results in asteroids_nn_tuning/asteroids
Showing 3 best trials
<keras_tuner.engine.objective.Objective object at 0x7fbe842c3e50>
Trial summary
Hyperparameters:
num_layers: 3
units: 30
lr: 0.009263303924328512
gamma: 0.11835983615085716
Score: 0.9974660515785218
Trial summary
Hyperparameters:
num_layers: 3
units: 15
lr: 0.0004000695122985253
gamma: 0.06352943477687838
Score: 0.9941176414489746
Trial summary
Hyperparameters:
num_layers: 4
units: 25
lr: 0.0004926783901540332
gamma: 1.6928927504819127
Score: 0.992941164970398

tuner.search_space_summary()
Search space summary
```

```
Default search space size: 4
num_layers (Int)
{'default': None, 'conditions': [], 'min_value': 2, 'max_value': 4, 'step': 1,
'sampling': None}
units (Int)
{'default': None, 'conditions': [], 'min_value': 15, 'max_value': 30, 'step': 5,
'sampling': None}
lr (Float)
{'default': 0.0001, 'conditions': [], 'min_value': 0.0001, 'max_value': 0.03,
'step': None, 'sampling': 'log'}
gamma (Float)
{'default': 0.05, 'conditions': [], 'min_value': 0.05, 'max_value': 5.0,
'step': None, 'sampling': 'log'}
```

Results can be further visualized at the associated Tensorboard.dev

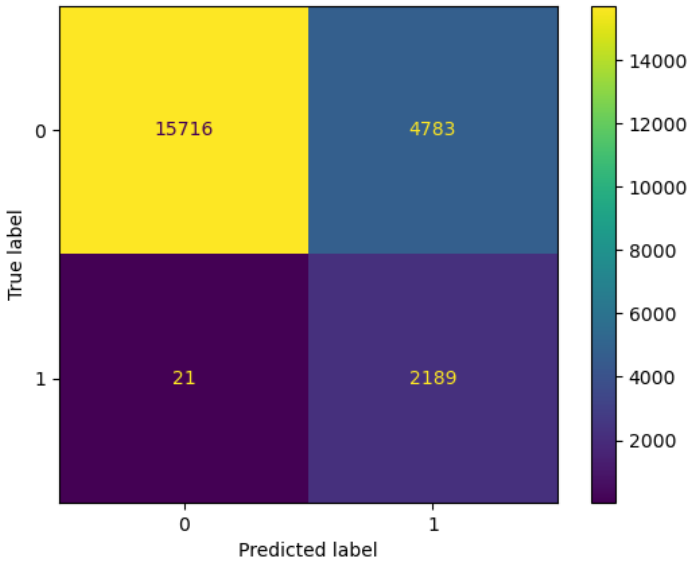
9.5 Tuned Model Performance

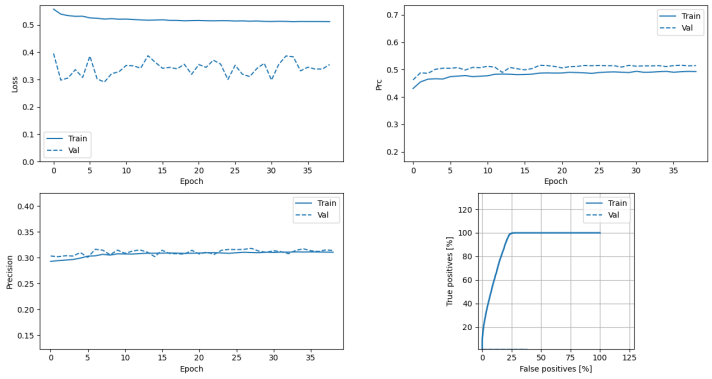
With the results of our tuner in hand, it is time to train a full model on the best parameters discovered.

```
model = build_model(nNeurons=30, gamma=0.118359, lr=0.00926)
history = model.fit(
X_train,
y_train,
epochs=45,
validation_data=(X_test, y_test),
callbacks=[callback],
class_weight = class_weight)
```

This gives:

	precision	recall	f1-score	support
0	1.00	0.77	0.87	20499
1	0.31	0.99	0.48	2210
accuracy			0.79	22709
macro avg	0.66	0.88	0.67	22709
weighted avg	0.93	0.79	0.83	22709





Tuned Model Metrics

It seems like our naive model was close to the optimal solution. The improvements in the tuned model are limited to slightly better performance on class 0 accuracy. This makes sense, it's clear from above visualizations that the decision boundary should be straightforward and simple, so as long we're optimizing for close to max precision then we will tend to converge on the same model without too much effort.

Results can be further visualized at the associated Tensorboard.dev

10 Conclusion

Unbalanced datasets present a unique problem in machine learning, and ask us to think hard about how to value type I and type II errors. We outline several methods to classify asteroids as hazardous or non-hazardous, and because misidentifying an hazardous body is so dangerous, we optimize for correctly identifying nearly all of them. From implementing logistic regression and SVM models we understood that their built-in losses are not flexible enough to deliver the high recall we desire. With this in mind, we designed a neural network that achieves 99% recall on hazardous asteroids, with an f-1 score of 83%

11 References

[1] Lin, T. Y., et al. "Focal loss for dense object detection." arXiv 2017." arXiv preprint arXiv:1708.02002 (2002).