UNIVERSITY OF CALIFORNIA,
IRVINE

# Selecting Strong Gravitationally Lensed Quasar Candidates with Ensembles of Supervised Machine Learning Methods

THESIS

A thesis submitted for the degrees of

*B.S. in Physics with Honors*
*B.S. in Computer Science with Honors*

by

Kyler Frazier

Thesis Committee:
Alberto Krone-Martins, Chair
Manoj Kaplinghat

2021

# TABLE OF CONTENTS

# LIST OF FIGURES

# LIST OF TABLES

# ACKNOWLEDGMENTS

I would like to thank my project advisors for their seemingly endless knowledge, wisdom, and guidance. I would also like to thank my family, friends, and colleagues for their unwavering support.

# ABSTRACT OF THE THESIS

Selecting Strong Gravitationally Lensed Quasar Candidates with
Ensembles of Supervised Machine Learning Methods

By

Kyler Frazier

B.S. in Physics with Honors
B.S. in Computer Science with Honors

University of California, Irvine, 2021

Alberto Krone-Martins, Chair

The discovery of strong gravitational lenses aids in cosmological studies, including providing independent inferences for the Hubble's constant and Dark Matter properties. However, selecting them out of the other astronomical sources has proven to be a challenging process. As the numbers of discovered lenses begins to grow, machine learning methods as supervised learning have begun to emerge. This project aims to study the selection of strong gravitationally lensed quasar candidates using small training sets and a stacked ensemble method consisting of a random forest, support vector machine, and gradient boosted tree as base models and another random forest as a meta model. The data in this project originates from the ZTF photometric and astrometric timeseries before going through feature extraction and selection. After hyperparameter optimization, the resulting model was successfully able to filter out all quasars and identify $\sim 30\%$ of lenses from the testing set. This model will later be run on a larger, unlabeled data set for lens candidate selection where the final selected lens candidates will be confirmed manually.

# Chapter 1

# Introduction

## 1.1 Motivation

Strong gravitational lenses (GLs) (Einstein 1936; Zwicky 1937) are an illusive but powerful tool for probing many mysteries of astronomy. They allow us to model the dark matter in galaxies based on their gravitational influences (Koopmans et al. 2009) and independently calculate the controversial value of Hubble's Constant (Refsdal 1964). The usefulness of GLs grows as more of them are discovered, as the conclusions we draw from studying them become increasingly statistically significant. However, as a consequence of the difficulty in isolating them from millions of other luminous celestial bodies, the number of confirmed GLs is on the order of magnitude of low hundreds since the first discovery in 1979 (Walsh et al. 1979). The goal is to discover more GLs while increasing the rate of their discoveries with assistance from modern machine learning and informational retrieval methods. This project focuses on developing a machine learning model that works with a small training set which can select strong lens candidates from images of quasars to be finally inspected by a human.

## 1.2 Strong Gravitational Lenses

Similar to how glass can refract light, gravity can act as a lens (Einstein 1936; Zwicky 1937). A sufficiently massive celestial body can lens other luminous celestial bodies enough to distort images that we observe. The former body is known as the lens and the latter as the source. In reality, all light is affected by gravity to an extent, though it is typically only observable on the cosmic scale.

One of the more prominent quantifiers of lensing is the deflection angle, $\hat{\alpha}$, which measures the angle between a ray of light that is observed and the same ray if it had not been affected by gravity (Congdon & Keeton 2018).

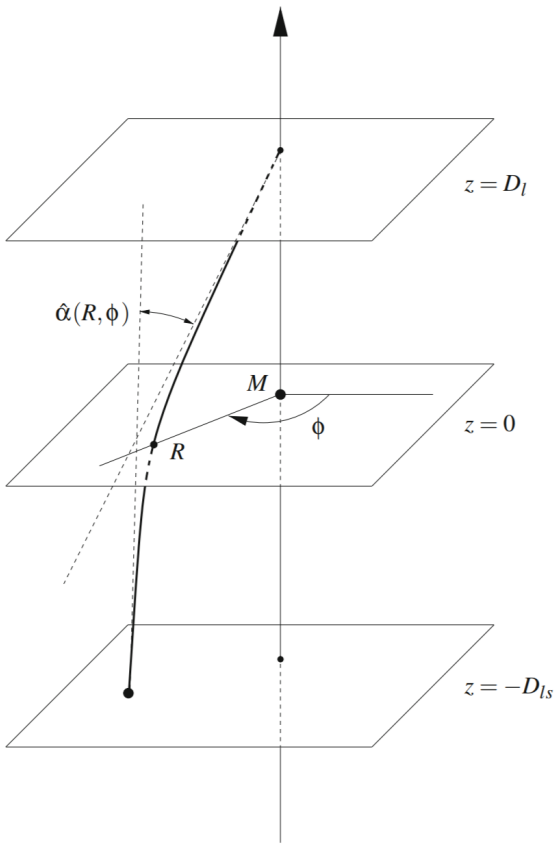


Figure 1.1: An illustration of the deflection angle with a point mass as the lens in the deflector plane at $z = 0$, a point as the source at $z = -D_{ls}$, and the observer at $z = D_l$ (Congdon & Keeton 2018).

The deflection angle can be approximated with conservation of energy from Newtonian physics; its derivation reveals that the deflection angle of a light ray as it passes a point mass lens is

$$\hat{\alpha} = \frac{2GM}{Rc^2} \tag{1.1}$$

where $G$ is the gravitational constant, $M$ is the mass of the lens, $R$ is the impact parameter of the light ray (see figure 1.1), and $c$ is the speed of light. This can be written more compactly as

$$\hat{\alpha} = \frac{R_S}{R} \tag{1.2}$$

where $R_S = \frac{2GM}{c^2}$ is the Schwarzschild radius of the lens. This Newtonian approximation does a fair job of describing the path of the light ray; it correctly predicts that in the extreme case, when the impact parameter is infinitely large, that there is no deflection, and that the deflection angle increases as the impact parameter decreases. The corrected version of this equation, as derived with General Relativity, predicts that the deflection angle is actually

$$\hat{\alpha} = \frac{2R_S}{R} \tag{1.3}$$

given that spacetime is very close to flat, described by the weak-field limit (Congdon & Keeton 2018). Its derivation can be reconstructed by solving Fermat's principle

$$\delta \left\{ \int_{P_1}^{P_2} n(r)\, ds \right\} = 0 \tag{1.4}$$

where the light ray travels from $P_1$ to $P_2$ through refractive index $n(r)$, using the Euler-Lagrange equation and choosing a pseudo-refractive index

$$n(r) = 1 + \frac{R_S}{r}. \tag{1.5}$$

In this case, $n(r)$ is dubbed "pseudo-refractive index" because, in principle, this form of

gravitational lensing closely resembles that of atmospheric lensing, though the concept of refraction is held loosely. The equation for the deflection angle can be generalized for more arbitrarily shaped lenses under the thin-lens approximation, that is, the lens is constituted of one mass concentration and is small relative to the distances that the light ray travels. The defector plane is defined to be the plane that the center of mass of the lens sits in. Under the thin-lens approximation, equation 1.3 evolves into

$$\vec{\hat{\alpha}}(\vec{\zeta}) = \frac{4G}{c^2} \int \int \sum (\vec{\zeta'}) \frac{\vec{\zeta} - \vec{\zeta'}}{|\vec{\zeta} - \vec{\zeta'}|^2} d\xi' d\eta' \tag{1.6}$$

where $\vec{\zeta} = (\xi, \eta)$ represents the location the light ray passes through the defector plane and $\vec{\zeta'}$ represents the locations of the infinitesimally small point mass elements which constitute the total surface mass density of the lens (Surdej 2021).

Strong lensing is a form of gravitational lensing defined by several resolved images and occurs when the deflection angle is sufficiently large, i.e. when the lens is dense and the impact parameter is small. Galaxies can be good candidates for lenses because they are very massive, aren't too bright, and can be relatively small. Quasars are generally good candidates for sources because they are incredibly bright and dense. Examples of strong lensing with this lens and source pair include: two-image lenses (Walsh et al. 1979), Einstein Crosses (Huchra et al. 1985), and Einstein Rings (Hewitt et al. 1988). As their names imply, they respectively form two images, four images, and a Ring-shaped image from a single source.

Figure 1.2: An image of an Einstein Cross formed from the galaxy UZC J224030.2+032131, the diffuse point in the center, lensing a QSO, seen as the 4 images around the galaxy. (ESA/Hubble 2012).

Multiple images form when there is more than one path for light rays to travel to reach the observer given a single source. In the context of gravitational lensing in the thin-lens approximation, this means that there is more than one deflection angle which correctly redirects the light rays in a single defector plane.

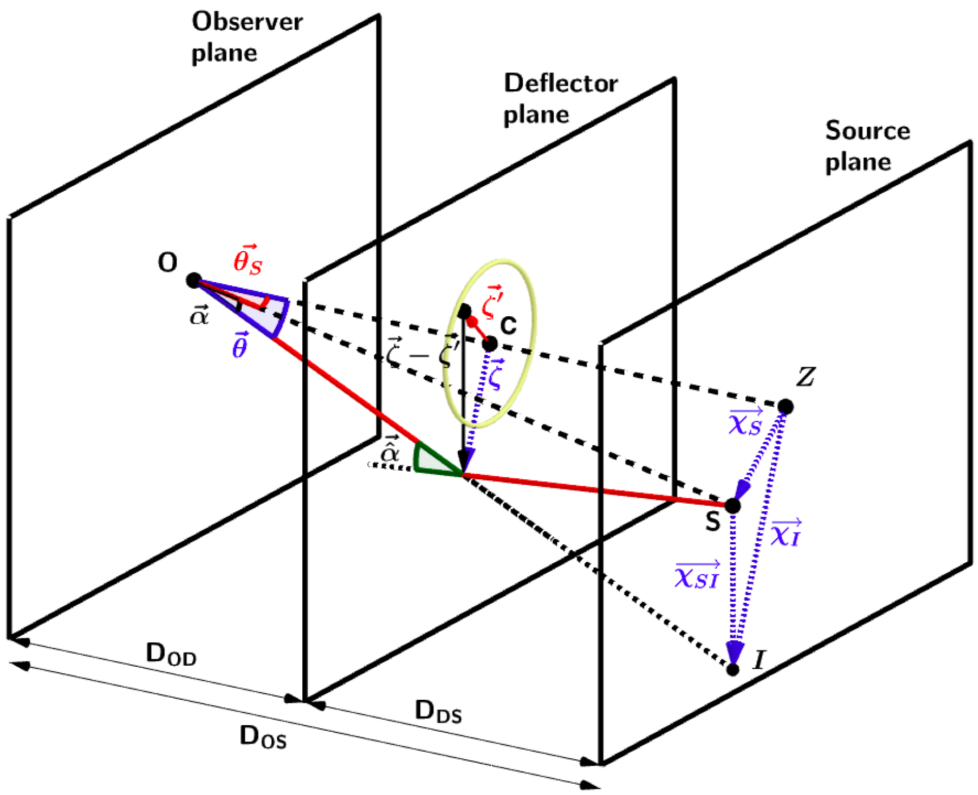


Figure 1.3: Visualization of the variables in equation 1.7. The new angle vectors introduced, $\vec{\theta}$ and $\vec{\theta_S}$, represent the location of an observed image and the actual location of the source relative to the location of the lens respectively. $O$, $D$, and $S$ represent the parallel planes that contain the observer, defector plane, and source respectively. $D_{AB}$ is equal to the distance between plane $A$ and plane $B$ (Surdej 2021).

This can be described mathematically with the gravitational lens equation (see figure 1.3 for variable definitions)

$$\vec{\theta} = \vec{\theta_S} + \vec{\hat{\alpha}}(\vec{\theta} D_{OD}) \frac{D_{DS}}{D_{OS}}. \tag{1.7}$$

It is possible to have more than one unique value of $\vec{\theta}$ which solves the equation, each with their own deflection angle; these give rise to the different images observed in strong lensing, though there is no guarantee that they will be resolved (Surdej 2021). In reality, the effects of lensing are less straightforward, and include other attributes such as shear and magnification. Lenses are generally more complex, with sources often being influenced by multiple lenses before reaching the observer.

## 1.3 Quasars as Sources

Quasars, also known as quasi-stellar objects or QSOs, are a type of supermassive black hole and are the most luminous type of active galactic nuclei. They are roughly defined to have luminosities greater than or equal to $10^{44}$ erg $s^{-1}$; their luminosities come from an accretion disk. The accretion disk of a quasar contains particles which orbit together and radiate as the particles interact with each other, eventually falling into the black hole as they lose energy (Maoz 2016). The light curve, which is the light intensity as a function of time, of quasars can be modeled as a damped random walk (Kelly et al. 2009; Zu et al. 2013). If a quasar is lensed and the resulting images are resolved, it is possible to identify the images as a lens by the light curves of each image. This is because the light curves of each image come from the same source, but will have a time-lag since the light in each image travels different distances to reach the observer. However, if the images are not resolved, then the light curves with different time-lags will be superimposed on each other. Because the light curves from each unresolved image have the same characteristics, the entropy of the superimposed light curve will go down; checking the entropy of quasars can assist in identifying lenses (Krone-Martins et al. 2019).

## 1.4 The Zwicky Transient Facility

Located at the Palomar Observatory, the Zwicky Transient Facility (ZTF) is an optical time-domain survey. It uses the the 48-inch Samuel Oschin Schmidt Telescope with a 47 $deg^2$ field of view and an 8 second readout time (Bellm et al. 2018; Graham et al. 2019). The telescope is a catadioptric astrophotographic telescope, combining both refraction and reflection in one system. The focal plane of the ZTF contains 16 $6k \times 6k$ science CCDs, as well as four $2k \times 2k$ science CCDs on the perimeter for guidance and control (Bellm et al. 2018).

One of the primary goals of ZTF is to map out as much of the sky as possible multiple times per year. It is able to take a full scan of the northern sky every night. The data from this survey has a wide range of applications, which includes studying: gravitational waves, type Ia Supernovae, and gravitational lenses (Bellm et al. 2018; Graham et al. 2019).

# Chapter 2

# Data and Methods

The original image data used in this project comes from the ZTF, and was further preprocessed before being used in the machine learning models. All of the machine learning models were derived from those defined by the Scikit-learn 0.24.1 library (Pedregosa et al. 2011) in Python 3.7.4 (Van Rossum & Drake Jr 1995), with additional usage of the python libraries Numpy 1.19.5 (Harris et al. 2020), Matplotlib 3.4.1 (Hunter 2007), and Pandas 0.25.1 (McKinney et al. 2010) for data manipulation. The final model used was a stacked ensemble using a random forest, SVM, and gradient boosted tree as the base models and another random forest as the meta model.

## 2.1 Data

The raw data from ZTF comes in two forms: photometric timeseries and astrometric timeseries. These contain information on how the fluxes of resolved images change over time and how their positions change over time.

From this data, several features can be extracted: entropy, Fourier powerspectra, slopes of

significant frequencies in the powerspectra, and correlations between the photometric and astrometric timeseries. The entropy of the timeseries was useful for identifying lenses that are not fully resolved, as was noted in section 1.3. The Fourier powerspectra was taken by squaring the Lomb-Scargle periodogram of the timeseries, and was further transformed with principal component analysis (PCA). This was used for finding periodic signals in the timeseries (where observations were not necessarily taken at even time intervals). Similarly, the slopes of the significant frequencies of the Fourier powerspectra were added, where significance was determined by the p-value. Lastly, it was expected that lensed QSOs have higher correlations between the photometric and astrometric timeseries than non-lenses.

Once the features were extracted, two-sample Anderson-Darling tests were performed between all dimensions of the feature space using known lenses in order to rank the significance of the features for feature selection. The more dissimilar the Anderson-Darling test indicates that the distributions of lenses and non-lenses in a certain dimension are, the higher the rank is for that dimension in solving a classification problem. The top 44 features were selected to be used as the data set of this project.

These pre-processing activities of the ZTF data were performed in the context of the Gaia Gravitational Lens group (GraL; see Krone-Martins et al. 2018, and further papers). The data of this BSc thesis refers to the data after this feature extraction and selection: 450 data points of 44 dimensions each. This data is balanced, with 225 data points per class of "QSO" or "Lens" which contain QSOs that are not lensed and QSOs that are lensed respectively. The training data and testing data were split on a roughly $3:1$ ratio, with 337 data points in the training set and 113 data points in the testing set.

## 2.2 Influencing Pure Classification of Lenses

While the training data for this model is balanced, the model will be selecting strong lens candidates from an unbalanced data set; there are significantly more QSOs that are not lensed than those that are. Because the lens candidates need to be verified manually, having too many false positives of lenses will make it difficult to find any of the actual lenses. The solution is to make a model that is "picky" about what it selects to be lens. The model should only select lenses when it is very confident. This can be influenced in several ways, including defining an error metric that encourages a very pure selection of lenses and finding a sufficiently high threshold for the model when it labels a data point as a lens.

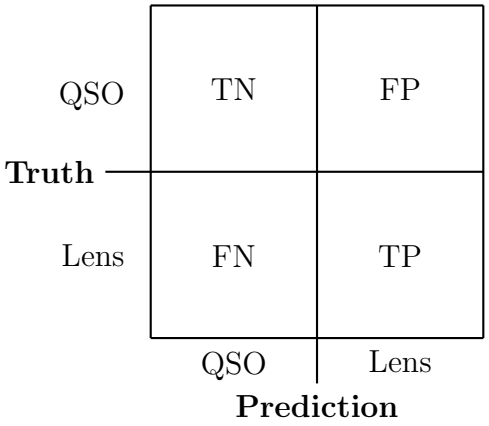


Figure 2.1: A confusion matrix with 2 labels, QSO and Lens. In the matrix, TN, FN, FP, and TP are true negative, false negative, false positive, and true positive respectively. A model which classifies all data points correctly will have a diagonal confusion matrix.

### 2.2.1 Defining an Error Function

An error function was made for the models in this project in a way as to discourage incorrect mislabeling of quasars, keeping false positives at a minimum. It was defined as

$$Error = 1 - F_\beta \tag{2.1}$$

where $F_\beta$ is the F-score of the model's prediction on the testing data using weighting parameter $\beta = 0.01$. The F-score is defined as

$$F_\beta = \frac{(1+\beta^2)\cdot \mathrm{TP}}{(1+\beta^2)\cdot \mathrm{TP} + \beta^2 \cdot \mathrm{FN} + \mathrm{FP}} \tag{2.2}$$

where TP, FN, and FP are defined in figure 2.1.

### 2.2.2 Confidence Thresholds

All of the models used in this project have soft predict functions: after the model has been trained, the model can output a probability (or more formally, a score) for how confident it is that a given data point is either a QSO or a Lens. The predefined threshold for determining if a model is a Lens or not is 0.5. However, to remove more quasars from the data points that were selected as lenses, a higher threshold was used. The hard predictions of the models then becomes

$$\mathrm{Class} = \begin{cases} \mathrm{Lens} & P(x = Lens) \geq \mathrm{Threshold} \\ \mathrm{QSO} & P(x = Lens) < \mathrm{Threshold} \end{cases} \tag{2.3}$$

for some data point $x$. A threshold is further tuned to minimize the error.

## 2.3 Models

### 2.3.1 Random Forests

Decision trees are used as a base in many different machine learning models. As a form of supervised learning, they create a tree structure of splits given a training set, where the splits are based off of values of the parameters and the leaves are the predicted classes. To

determine what splits are made down the tree, the splits closest to the root are performed to minimize the gini impurity (or maximize the information gain) based on a single parameter in the training set, with subsequent splits down the branches of the tree improving the model less than higher splits (Breiman et al. 1984).

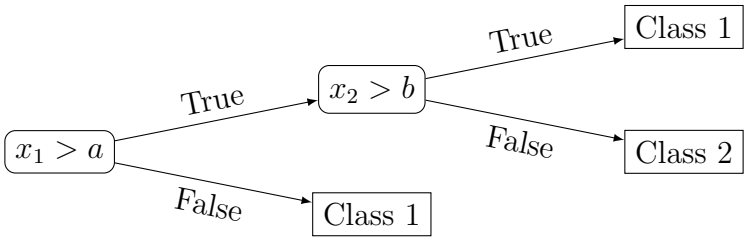


Figure 2.2: An example of a decision tree. The tree was trained on data that had continuous parameters $x_1$ and $x_2$ which outputs one of two classes.

The random forest model was developed as a method to reduce the high variance of deep decision trees, as decision trees tend to overfit. Random forests are ensemble models that use bagging on decision trees; the forest contains multiple decision trees which are each trained individually on data that is sampled, with replacement, from the original training data. This means that each decision tree, even if they are relatively deep, will only be overfit on a subspace of the parameters. The random forest then takes the average of its trees as the final output (Ho 1995).

Because random forest models do not overfit, determining the number of trees to use in the model becomes a balance of reducing the error metric and increasing the performance. The model's error will begin to reach an asymptote when increasing the number of trees no longer significantly improves the model, at which point no further trees should be added to save on performance.

### 2.3.2 Support Vector Machines

The support vector machine (SVM) learning model aims to separate the classes in the parameter space or some transformation of the parameter space with a hyperplane. When the data is not linearly separable, the SVM can use a soft margin which allows for overlap (Cortes & Vapnik 1995).

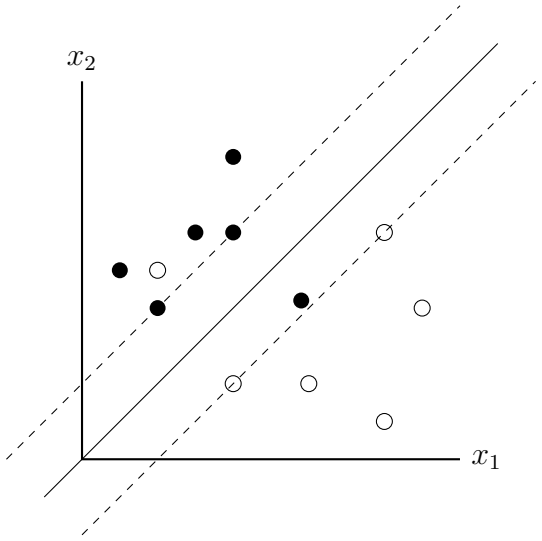


Figure 2.3: An example of a linear support vector machine with a soft-margin. The support vectors are the points used to build the lines which define the margin, seen as the dotted lines, with the decision boundary as the hard line. The example contains a 2D parameter space with two classes seen as white points and black points.

An easy to visualize model of a SVM is one that is linear in a 2D parameter space, as shown in figure 2.3. The model attempts to separate the data while leaving a margin, where the margin boundaries are called the support vectors. The SVM model starts with an initial set of weights which define the separation of the classes based on the input parameters. It uses a hinge loss function to determine how accurate the weights are, and iteratively uses stochastic gradient descent to converge on an optimal set of weights.

When the separability of the data is nonlinear or unclear, as is the case in this project, the SVM model can be abstracted by using the kernel trick which transforms the input

parameters space (Cortes & Vapnik 1995). A linear kernel is just the linear SVM, and polynomial kernels will effectively apply a polynomial function to the input parameters. The radial basis function (RBF) kernel is useful when the separability of the data is not apparent and has a similar behavior to K-nearest neighbor models. The RBF kernel is defined as

$$K(x, x') = \exp\left(-\gamma \|x - x'\|^2\right) \tag{2.4}$$

given two data points, $x$ and $x'$, and a kernel coefficient $\gamma$. The RBF kernel effectively is the summation of all polynomial kernels from power 1 to power $\infty$ (this can be shown by taking the Taylor expansion of the RBF kernel).

Using an RBF kernel, the SVM model in this project was trained after tuning the $\gamma$ hyper-parameter to achieve a low error.

### 2.3.3 Gradient Boosted Trees

Similar to random forests, gradient boosted trees are an ensemble model derived from decision trees. However, rather than making many trees separately and putting them together, gradient boosted trees are made iteratively. Additionally, the individual trees used in the iterative process are relatively shallow, usually only having a few splits; the decision trees of the gradient boosted tree model in this project have a maximum depth of 3. In the iterative building process, the gradient boosted tree models typically start with a single leaf as a guess. A decision tree is then made in order to minimize the loss of the initial guess. Several loss functions can be used; this project uses binomial deviance loss function, which is defined as

$$L(y, \gamma) = y\gamma + \ln\left(1 + e^{\gamma}\right) \tag{2.5}$$

for a known class, $y$, and the decision tree's log odds, $\gamma$ (which is not the same variable as that of the RBF kernel). Subsequent trees are made to reduce the loss of their successor trees and can be generated using the gradient of the loss function. At each step, the tree is weighted with the learning weight (which is the same for all trees). The final model is the sum of all of the iteratively made models: the initial leaf and each of the trees multiplied by the learning weight. Predictions will be in units of log odds, which can be converted to probabilities using the logistic function (Friedman 2001).

Similar to the random forest model, gradient boosted trees are resistant to overfitting as a function of the number of trees; trees can be added until performance no longer improves. The other notable parameters to tune are the the max tree depth and the learning rate of the model.

### 2.3.4 Stacked Ensembles

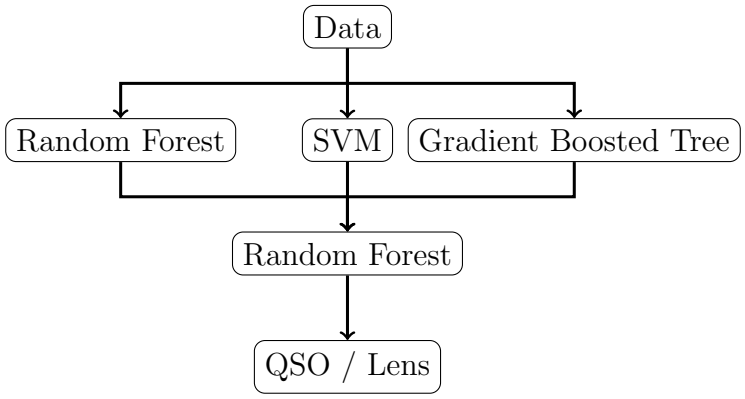


Figure 2.4: A diagram of the stacked ensemble pipeline using a random forest, SVM, and gradient boosted tree as the base models and a random forest as the meta model. The training data and testing data passed to each base model was identical. Each of their outputs were passed as a combined input to the meta layer, which made the final prediction.

Ensembles combine multiple models into a single one, as was the case for the random forest and gradient boosted tree, to create a classifier that works better than any single classifier by combining the strengths while ignoring the weakness of the individual models. A stacked

ensemble has two layers consisting of the base models and the meta model respectively. The base models can consist of any classifier that can make a soft prediction. Each base model is fitted with the same training data, and their soft predict outputs are then passed as an input into the meta model. The meta model then fits with the predictions of the base models on the training data with the same training classes. Testing data travels down the pipeline in a similar manner; the base models make soft predictions, and the meta model takes those and makes a classification.

When a random forest is used as a meta model, the ensemble's performance does not drop below that of any base model. This is because if all of the other models hurt the performance, the decision trees in the random forest will only split on the soft predictions of the best model. However, this is only true if the split condition for the decision trees minimizes the correct error function; the random forest model used in this project splits to minimize the gini impurity, which does not accurately reflect the error function in equation 2.1. Similarly, because the meta model takes the soft predictions of the base layer, the threshold for each base model is unused. To compensate for this, the meta model can be influenced to pick higher thresholds by assigning sample weights to the different classes in the training data. When the weights for the QSOs is larger, the meta model will place more value on correctly classifying the QSOs. Tuning the sample weights of the QSOs relative to those of the lenses is important for keeping the selected lenses pure. This can be further tuned by adding a threshold on the predictor of the meta model.

# Chapter 3

# Analysis and Predicted Results

## 3.1 Hyperparameter Optimization

The hyperparameters of the base models were tuned individually for performance and to make estimations of their base level performances. The stacked ensemble's hyperparameters, which consists of the hyperparameters of the meta model, was then tuned on the best models found for the base models.

### 3.1.1 Base Models

The base models were primarily tuned manually. For the random forest and gradient boosted tree, the number of trees were tuned separately from the other parameters because it was only necessary to increase them until performance was growing too slow; the number of trees was increased exponentially. The hyperparameter $\gamma$ for the SVM was tuned over an exponential grid space and the learning rate for the gradient boosted tree was tuned over a linear grid space. The regularization parameter for the SVM was left at the default of

1.0 and the max depth of the decision trees in the gradient boosted forest was left at the default of 3; deviations from these values did not show significant improvements to the models. The threshold was then tuned over a linear space for each base model after all other hyperparameters were set.

| **Random Forest** | **Tuned Value** |
|---|---|
| $n$ | 64 |
| Threshold | 0.68 |

Table 3.1: The tuned hyperparameters of the random forest base model, where $n$ is the number of trees in the forest.

| **SVM** | **Tuned Value** |
|---|---|
| $\gamma$ | $2.0 \times 10^{-8}$ |
| Threshold | 0.93 |

Table 3.2: The tuned hyperparameters of the support vector machine base model, where $\gamma$ is the kernel coefficient for the radial basis function kernel.

| **Gradient Boosted Tree** | **Tuned Value** |
|---|---|
| $n$ | 64 |
| Learning Rate | $7.0 \times 10^{-2}$ |
| Threshold | 0.87 |

Table 3.3: The tuned hyperparameters of the gradient boosted tree base model, where $n$ is the number of trees.

### 3.1.2 Final Model

The final model, which was the stacked ensemble using the aforementioned base models as well as the random forest meta model, had three hyperparameters to tune: the number of

trees in the meta model, the sample weights, and the threshold. Because increasing the number of trees does not result in overfitting, the number of trees was increased over an exponential grid space until the error was at a minima.

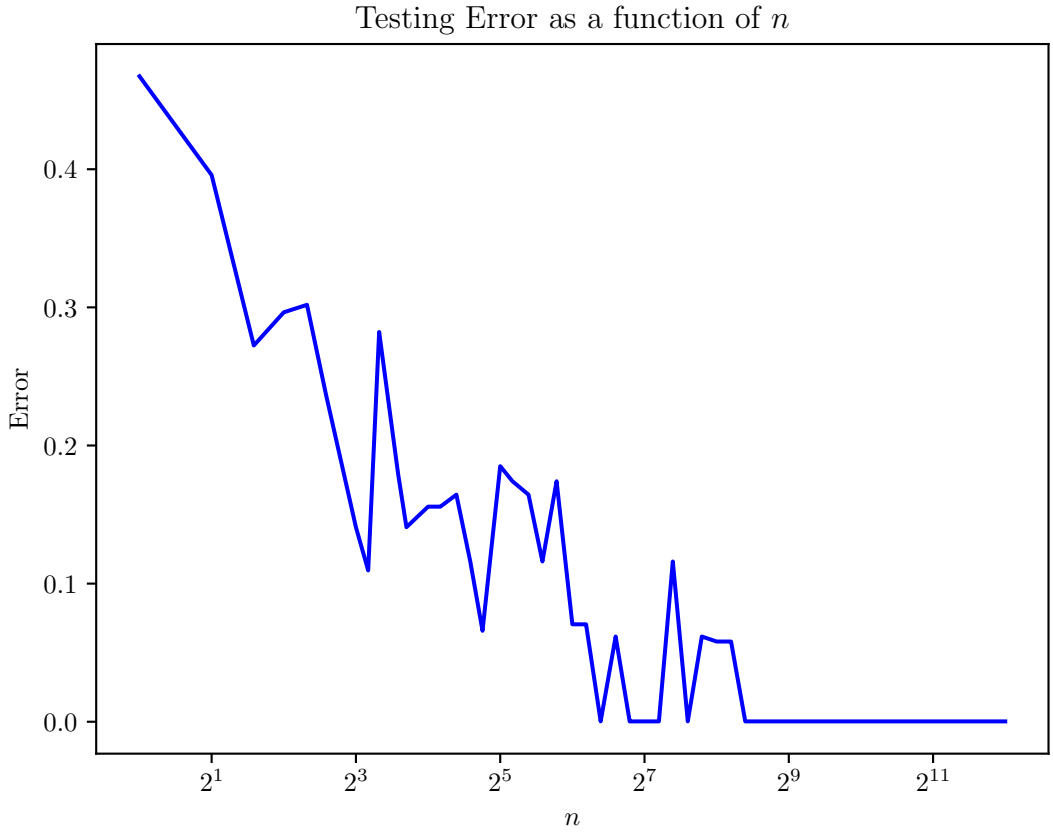


Figure 3.1: The testing error as a function of the number of decision trees used in the random forest meta model.

The sample weights and threshold were tuned together over a dense linear grid space. Several minima were found in the grid space as depicted in figure 3.2. The sample weight with the highest value was chosen because it is expected that QSOs should be on the order of magnitude of 100's of times more important to classify than lenses, as was defined by the $\beta$ parameter in the error function. Out of those minima, the value for the threshold was simply averaged and rounded, which can be seen in table 3.4.

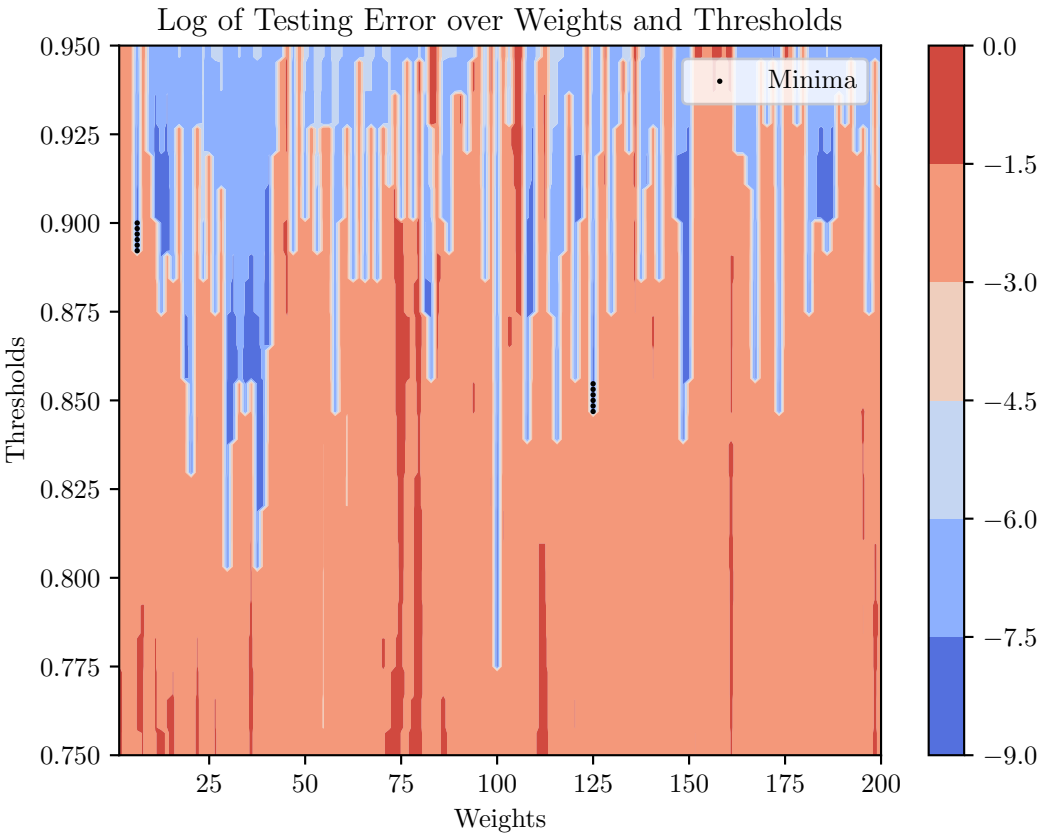


Figure 3.2: The natural logarithm of the testing error as a contour plot of the cutoff threshold and the sample weights for the QSOs. Points represent the minima of the plot.

| **Meta Class** | **Tuned Value** |
|---|---|
| $n$ | 111 |
| QSO Sample Weight | 125 |
| Threshold | 0.85 |

Table 3.4: The tuned hyperparameters of the final model, where $n$ is the number of trees in the meta class. The QSO sample weight is applied to all QSO data points, while the sample weight for lenses is left at 1.

## 3.2 Testing Data Performance and Predictions

| Model | Error |
|---|---|
| Random Forest | $3.66 \times 10^{-4}$ |
| SVM | $5.469 \times 10^{-3}$ |
| Gradient Boosted Tree | $1.019 \times 10^{-3}$ |
| Stacked Ensemble | $2.29 \times 10^{-4}$ |

Table 3.5: The errors of the base models and the final model on the testing data as defined by equation 2.1 after hyperparameter tuning.

After tuning all of the hyperparameters, the testing errors for the respective models were calculated (see table 3.5). The stacked ensemble outperforms all of the base models and was selected as the final model. On the testing data, it was able to filter out all QSOs and select $\sim 30\%$ of the known lenses.

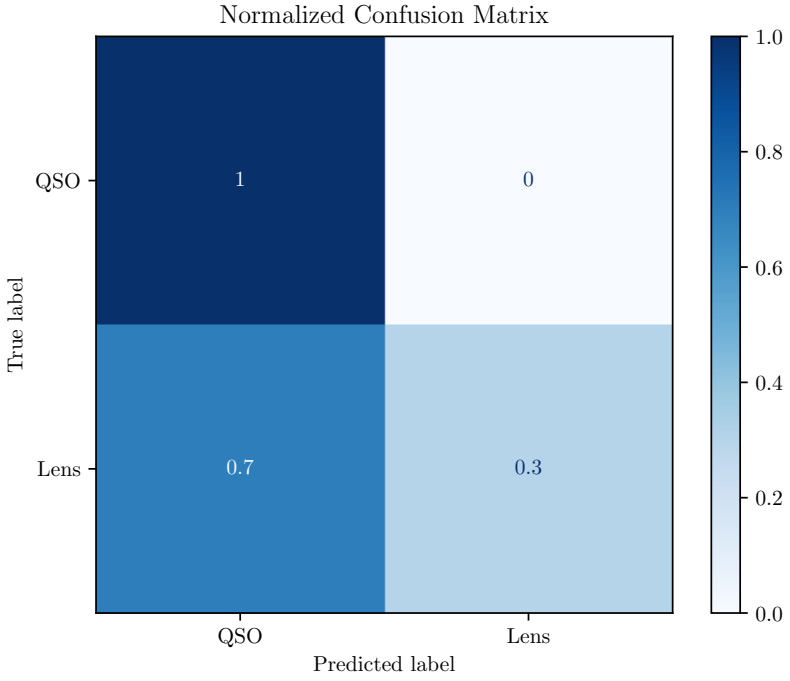


Figure 3.3: Confusion matrix of the final model on the testing data. The normalized values are rounded to one decimal place.

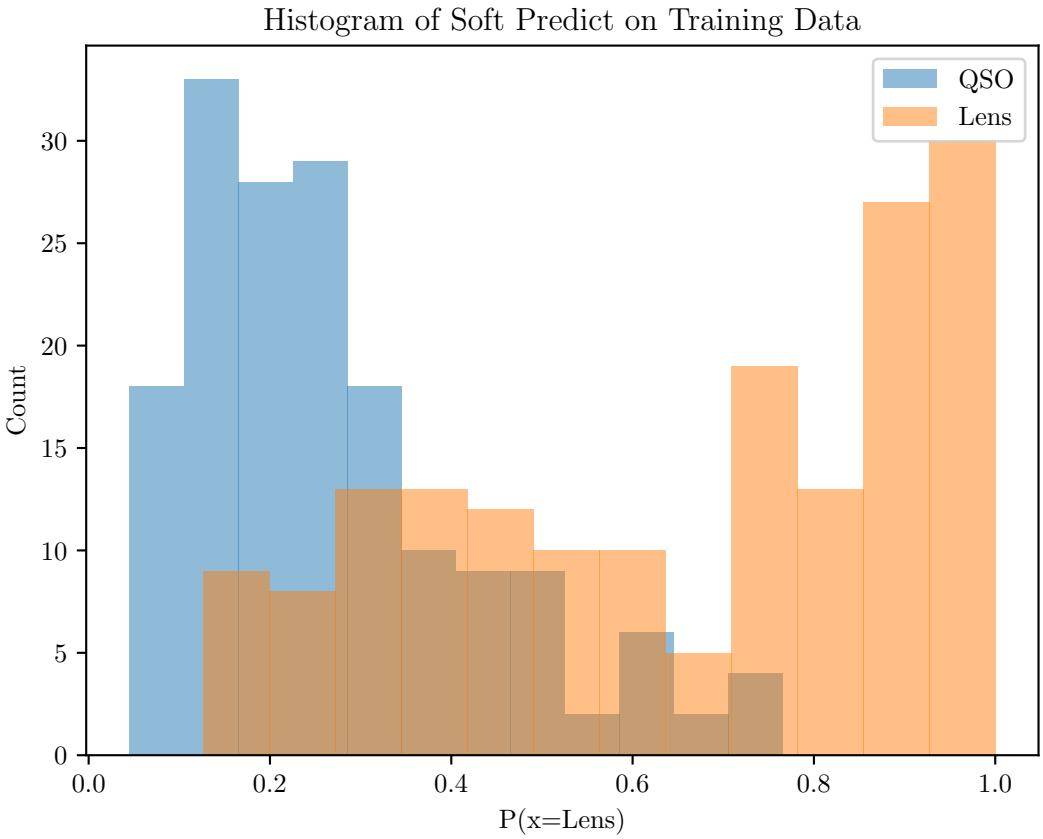


Figure 3.4: Histogram of the final model's soft predictions on the training data. The true labels are separated by color; each label is given 12 bins on the histogram.

In figure 3.4, the separation of QSOs and lenses is not too extreme; it is not likely that the model overfit on the training data. As for the testing predictions in figure 3.5, the unimodal distribution of QSOs is separate from the bimodal distribution of lenses. It is possible that the QSO distribution has a tail end on the right which is not captured in this data set, however it appears that it would be significantly smaller than the overlapping lens data points.

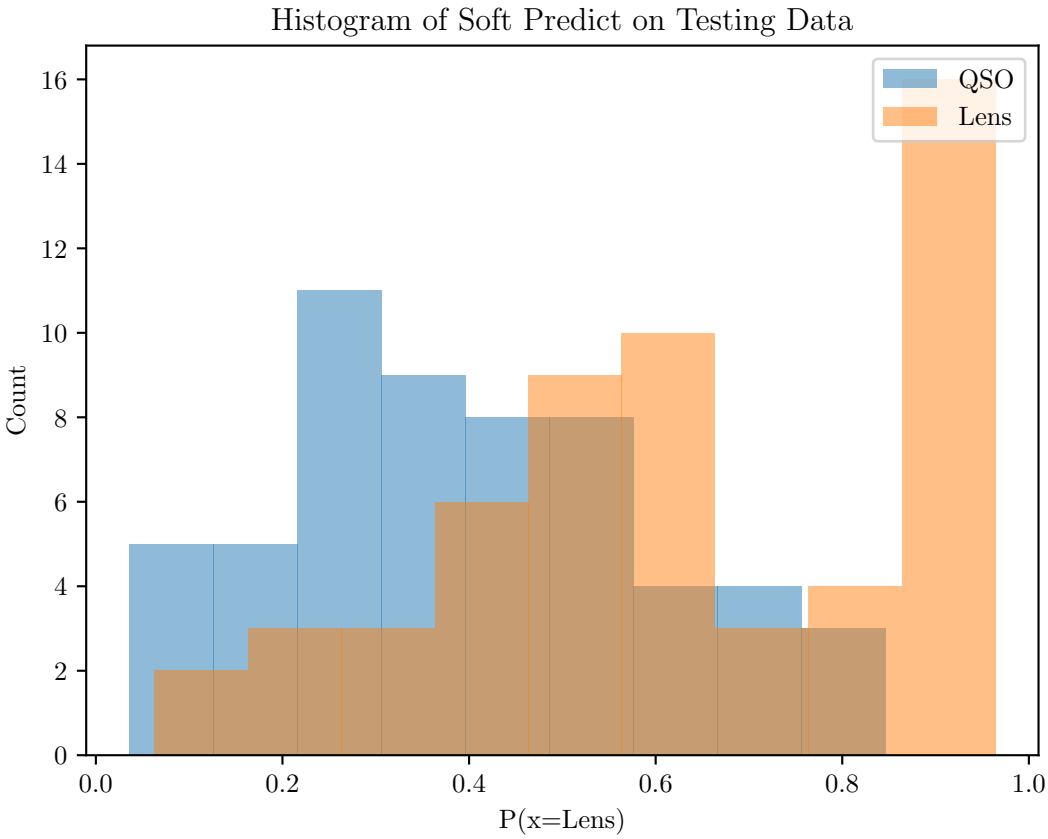


Figure 3.5: Histogram of the final model's soft predictions on the testing data. The true labels are separated by color; each label is given 9 bins on the histogram.

The final model will later be run on a large set of roughly $2 \times 10^6$ unlabeled data points. An estimation of the number of existing lenses (doubly + quadruply imaged ones) in this set is roughly $2 \times 10^3$. Considering that these numbers are accurate and that the distributions of 3.5 remain the same in the larger data set, it is predicted that the model of this project will be able to select roughly 607 lenses, though they may not all be previously undiscovered. More realistically however, the distributions of 3.5 will not hold perfectly, and there will be a right tail to the QSO distribution. In this case, the number of selected lenses will not be perfectly pure, and may even be too large to manually filter. If the selection set is too large, the selected points which the model is most confident in can be sampled, essentially shifting the threshold to higher values until the number of points is manageable.

# Chapter 4

# Conclusion

The discovery of strong gravitational lenses has a fundamental importance in modern cosmology. However, selecting them out of the other astronomical sources has proven to be a challenging process. In this BSc thesis we build a stacked model that was successfully able to filter out all of the QSOs from both the training and testing data sets and selected more lenses than any of the base models, achieving a normalized true positive rate for lenses of roughly $\sim 30\%$ and a false positive rate that tends to null. Our proposed stacked model uses a random forest, support vector machine, and gradient boosted tree as base models and another random forest as a meta model. Using sample weights, the stacked model was successfully influenced into choosing higher thresholds for the base models, as higher thresholds showed improved performance while tuning hyperparameters for each base model.

Our model adopts astrometric and photometric time series, and it does not depends on the lenses being spatially resolved. We now aim to analyse larger data sets and to check if it will result in successful selection of lens candidates, contributing then to the discovery of these rare and physically important, astronomical phenomena from present day data from ZTF as well as preparing for future time-resolved surveys as the Rubin/LSST.

# Bibliography


Bellm, E. C., Kulkarni, S. R., Graham, M. J., et al. 2018, Publications of the Astronomical Society of the Pacific, 131, 018002

Breiman, L., Friedman, J., Stone, C. J., & Olshen, R. 1984, Classification and Regression Trees (Chapman and Hall/CRC)

Congdon, A. & Keeton, C. 2018, Principles of Gravitational Lensing: Light Deflection as a Probe of Astrophysics and Cosmology (Springer)

Cortes, C. & Vapnik, V. 1995, Machine learning, 20

Einstein, A. 1936, Science, 84, 506

ESA/Hubble. 2012, Seeing quadruple, `https://esahubble.org/images/potw1204a/`

Friedman, J. H. 2001, The Annals of Statistics, 29, 1189

Graham, M. J., Kulkarni, S. R., Bellm, E. C., et al. 2019, Publications of the Astronomical Society of the Pacific, 131, 078001

Harris, C. R., Millman, K. J., van der Walt, S. J., et al. 2020, Nature, 585, 357–362

Hewitt, J., Turner, E., Schneider, D., et al. 1988, Nature, 333, 537

Ho, T. K. 1995, in Proceedings of 3rd International Conference on Document Analysis and Recognition, Vol. 1, 278–282 vol.1

Huchra, J., Gorenstein, M., Kent, S., et al. 1985, The Astronomical Journal, 90, 691

Hunter, J. D. 2007, Computing in Science & Engineering, 9, 90

Kelly, B. C., Bechtold, J., & Siemiginowska, A. 2009, The Astrophysical Journal, 698, 895

Koopmans, L. V. E., Auger, M., Barnabe, M., et al. 2009, Strong Gravitational Lensing as a Probe of Gravity, Dark-Matter and Super-Massive Black Holes

Krone-Martins, A., Delchambre, L., Wertz, O., et al. 2018, Astronomy and Astrophysics, 616, L11

Krone-Martins, A., Graham, M. J., Stern, D., et al. 2019, arXiv e-prints, arXiv:1912.08977

Maoz, D. 2016, Astrophysics in a Nutshell: Second Edition (Princeton University Press)

McKinney, W. et al. 2010, in Proceedings of the 9th Python in Science Conference, Vol. 445, Austin, TX, 51–56

Pedregosa, F., Varoquaux, G., Gramfort, A., et al. 2011, Journal of Machine Learning Research, 12, 2825

Refsdal, S. 1964, Monthly Notices of the Royal Astronomical Society, 128, 307

Surdej, J. 2021, Lecture Notes in Introduction to gravitational lensing

Van Rossum, G. & Drake Jr, F. L. 1995, Python tutorial (Centrum voor Wiskunde en Informatica Amsterdam, The Netherlands)

Walsh, D., Carswell, R. F., & Weymann, R. J. 1979, Nature, 279, 381

Zu, Y., Kochanek, C. S., Kozłowski, S., & Udalski, A. 2013, The Astrophysical Journal, 765, 106

Zwicky, F. 1937, Phys. Rev., 51, 290

# Appendix A

# Source Code

The source code is written in Python 3.7.4. The source code file structure contains a file `main.py` and a directory `utils` at the base with the remaining files inside `utils`. See chapter 2 for a list of all used libraries and sources.

The source code can also be found on GitHub at:

`https://github.com/KylerFrazier/Strong-Gravitational-Lenses-and-Machine-Learning`

## A.1 `main.py`

```
# Modules for processing, math, and graphing
import numpy as np
from matplotlib import pyplot as plt
from os import path
from sklearn.metrics import ( confusion_matrix,
                              plot_confusion_matrix )

# Hide warnings from sklearn about convergence
```

```
from sklearn.exceptions import ConvergenceWarning
import warnings
warnings.filterwarnings("ignore", category=ConvergenceWarning)

# Change plot fonts and enable LaTeX
plt.rcParams.update({
    "text.usetex": True,
    "font.family": "serif",
    "font.serif": ["Computer Modern Roman"]
})

# Modules for machine learning and other utilities
from utils import ( EnsembleBT,
                    RandomForestBT,
                    SupportVectorMachineBT,
                    GradientBoostingBT )
from utils import utils

def main():

    ### Setting up data and base models ###

    output_path = "./output/ensemble/"

    # Approximations of the larger data set
    n_real_data_size = 2000000
    n_predicted_lenses = 2000

    # Renaming classes to be binary to keep things clean
```

```
classes = np.array(["QSO", "Lens"])
QSO = np.where(classes == "QSO")[0][0]
LENS = np.where(classes == "Lens")[0][0]

# Loading data
x_tr, x_te, y_tr, y_te = utils.load_binary_data(
    "data/ADavailableData.csv",
    "data/ADlabel.csv",
    classes, verbose=True )

base_models = [
    ('rf', RandomForestBT(
        n_estimators = 64,
        random_state = 0 )),
    ('svm', SupportVectorMachineBT(
        gamma = 2e-8,
        kernel = 'rbf',
        probability=True,
        random_state = 0 )),
    ('gbt', GradientBoostingBT(
        learning_rate = 0.07,
        n_estimators = 64,
        loss = "deviance",
        random_state = 0 ))
]

trainer = utils.EnsembleTrainer( base_models, x_tr, y_tr, x_te, y_te,
                                 file_path = output_path )
```

```
### Tuning n ###

temp_weight = 100

ns = np.unique((2.0**np.linspace(0,12,12*5+1)[1:]).astype(int))
n_best = trainer.find_n_linear(ns, weight=temp_weight)

### Tuning weights and thresholds ###

thresholds = np.linspace(0.75, 0.95, 2**7+1).astype(float).round(decimals=10)
weights   = np.linspace( 0,  200, 2**7+1).astype(float).round(decimals=10)[1:]
weight_best, threshold_best = trainer.find_weights_and_threshold_grid(
                                    thresholds, weights )

### Displaying results ###

# Best results so far:
#     n = 111
#     weight = 125
#     threshold = 0.85
#     tp = 0.3036

print("Best Error:", trainer.model_best.error(x_te, y_te))
print("Best n:", n_best)
print("Best weight:", weight_best)
print("Best threshold:", threshold_best)
fp_best, tp_best = np.ravel(confusion_matrix(y_te,
    trainer.model_best.predict(x_te), normalize="true")[:,1])
print()
```

```
    print("Best False Positive:", fp_best)
    print("Best True Positive:", tp_best)
    print()
    print("Expected findings:")
    print("   Selected QSOs  =", int(n_real_data_size*fp_best))
    print("   Selected Lenses =", int(n_predicted_lenses*tp_best))

    disp = plot_confusion_matrix(trainer.model_best, x_te, y_te,
        cmap=plt.cm.Blues, normalize="true", display_labels=classes)
    disp.ax_.set_title("Ensenmble Model || CM Normalized")
    plt.savefig(path.join(output_path, "CM.pdf"))
    plt.clf()


if __name__ == "__main__":
    main()
```

## A.2 utils/

### A.2.1 __init__.py

```
from .EnsembleBT             import EnsembleBT
from .RandomForestBT         import RandomForestBT
from .SupportVectorMachineBT import SupportVectorMachineBT
from .GradientBoostingBT     import GradientBoostingBT

__all__ = [ "EnsembleBT",
```

```
            "RandomForestBT",
            "SupportVectorMachineBT",
            "GradientBoostingBT" ]
```

### A.2.2 utils.py

```
import pickle
import numpy as np
from matplotlib import pyplot as plt
from matplotlib import cm
from pandas import read_csv
from sklearn.model_selection import train_test_split
from os import path
from utils import EnsembleBT, RandomForestBT


def load_binary_data(feature_path, class_path, classes=[0,1], verbose=False):
    data_features = read_csv( feature_path, header=0, quotechar='"',
                            skipinitialspace=True ).to_numpy()
    data_classes = read_csv( class_path, header=0, quotechar='"',
                           skipinitialspace=True ).to_numpy()

    # The below assumes that the features and classes are ordered properly
    x, y = data_features[:,1:], np.ravel(data_classes[:,1:])

    num_entries_per_class = 0
    for i in range(len(classes)):
        y[y == classes[i]] = i
```

```
        num_entries_per_class += len(y[y == i])
    y = y.astype(int)
    assert num_entries_per_class == len(y) , f"The classes should be in {classes}"

    x_tr, x_te, y_tr, y_te = train_test_split(x, y, random_state=0, stratify=y)

    if verbose:
        print("Classes:", classes)
        print("x_tr.shape =", x_tr.shape)
        print("x_te.shape =", x_te.shape)
        print("y_tr.shape =", y_tr.shape)
        print("y_te.shape =", y_te.shape)

    return x_tr, x_te, y_tr, y_te

def plot_errors(x, x_label, error_tr, error_te, output_path):
    plt.semilogx(x, error_tr, c="blue", base=2, label="Training Error")
    plt.semilogx(x, error_te, c="red", base=2, label="Testing Error" )
    plt.title(f"Model error as a function of {x_label}")
    plt.xlabel(x_label)
    plt.ylabel("Error")
    plt.legend()
    plt.savefig(output_path)
    plt.clf()

def plot_errors_3D(x, y, x_label, y_label, error_tr, error_te, path1, path2):
    plt.contourf(x, y, error_tr.T, cmap=cm.coolwarm)
    plt.title(f"Model training error as a function of {x_label} and {y_label}")
    plt.xlabel(x_label)
```

```
    plt.ylabel(y_label)
    plt.savefig(path1)
    plt.clf()

    plt.contourf(x, y, error_te.T, cmap=cm.coolwarm)
    plt.title(f"Model testing error as a function of {x_label} and {y_label}")
    plt.xlabel(x_label)
    plt.ylabel(y_label)
    plt.savefig(path2)
    plt.clf()

class ProgressBar(object):

    def __init__(self, length):
        self.length = length
        self.tics = 0
        print(' '*11 + '_'*(length*2+1))
        print("Progress: [ ", end='', flush=True)

    def increment(self):
        if self.tics < self.length:
            print('> ', end='', flush=True)
            self.tics += 1
            if self.tics >= self.length:
                print(']')

class EnsembleTrainer(object):

    def __init__( self, base_models, x_tr, y_tr, x_te, y_te,
```

```
            file_path=".", file_name="EnsembleData.p",
            model_name="Model.p" ):

    assert path.exists(file_path), f"File path {file_path} does not exist."

    self.base_models = base_models
    self.x_tr, self.y_tr = x_tr, y_tr
    self.x_te, self.y_te = x_te, y_te

    self.data = {}
    self.file_path = file_path
    self.file_name = file_name
    self.model_name = model_name

    self.model_best = None
    self.error_best = np.inf

    self.n_best = None
    self.weight_best = None
    self.threshold_best = None

def save(self, file_path=None, file_name=None):
    if file_path == None or file_name == None:
        file_path = self.file_path
        file_name = self.file_name
    with open(path.join(file_path, file_name), 'wb') as handle:
        pickle.dump(self, handle)

def save_model(self, file_path=None, model_name=None):
```

```
    if file_path == None or model_name == None:
        file_path = self.file_path
        model_name = self.model_name
    with open(path.join(file_path, model_name), 'wb') as handle:
        pickle.dump(self.model_best, handle)

def find_n_linear( self, ns, threshold=None, weight=None,
                verbose=True, save=True, plot=True ):
    models = np.empty((len(ns)), dtype=object)
    error_tr = np.zeros((len(ns)))
    error_te = np.zeros((len(ns)))

    if self.threshold_best != None:
        threshold = self.threshold_best
    elif threshold == None:
        threshold = 0.80
    if self.weight_best != None:
        weight = self.weight_best
    elif threshold == None:
        weight = 1

    sample_weight = np.ones(self.y_tr.shape)
    sample_weight[self.y_tr == 0] = weight

    if verbose: bar = ProgressBar(len(ns))
    for i, n in enumerate(ns):
        ens_wrapper = RandomForestBT(n_estimators = n, random_state = 0)
        model = EnsembleBT( estimators = self.base_models,
                        final_estimator = ens_wrapper,
```

```
                    threshold = threshold )
    model.fit(self.x_tr, self.y_tr, sample_weight = sample_weight)
    error_tr[i] = model.error(self.x_tr, self.y_tr)
    error_te[i] = model.error(self.x_te, self.y_te)
    models[i] = model
    if verbose: bar.increment()

model_best = models[np.argmin(error_te)]
n_best = ns[np.argmin(error_te)]
self.n_best = n_best

if np.min(error_te) < self.error_best:
    self.error_best = np.min(error_te)
    self.model_best = model_best

if verbose:
    print(f"Best n: {n_best}\n")

n_data = {
    "ns"        : ns,
    "error_tr"  : error_tr,
    "error_te"  : error_te,
    "model_best" : model_best,
    "n_best"    : n_best
}
self.data["n"] = n_data

if save:
    self.save()
```

```
    if plot:
        plot_errors( ns, "$n$", error_tr, error_te,
                    path.join(self.file_path, "n.pdf") )

    return n_best

def find_weights_and_threshold_grid( self, thresholds, weights, n=None,
                                    verbose=True, save=True, plot=True ):
    error_tr = np.zeros((len(weights), len(thresholds)))
    error_te = np.zeros((len(weights), len(thresholds)))

    if self.n_best != None:
        n = self.n_best
    elif n == None:
        n = 2**7

    error_best = np.inf
    model_best = None
    ind_best = (0,0)

    if verbose: bar = ProgressBar(len(weights))
    for i, weight in enumerate(weights):
        sample_weight = np.ones(self.y_tr.shape)
        sample_weight[self.y_tr == 0] = weight

        for j, threshold in enumerate(thresholds):
            ens_wrapper = RandomForestBT( n_estimators = n,
                                         random_state = 0 )
```

```
        model = EnsembleBT( estimators = self.base_models,
                            final_estimator = ens_wrapper,
                            threshold = threshold )
        model.fit(self.x_tr, self.y_tr, sample_weight = sample_weight)

        error_test = model.error(self.x_te, self.y_te)
        error_tr[i,j] = model.error(self.x_tr, self.y_tr)
        error_te[i,j] = error_test
        if error_test < error_best:
            error_best = error_test
            model_best = model
            ind_best = (i,j)
    if verbose: bar.increment()

ind = np.unravel_index(np.argmin(error_te), error_te.shape)
if ind != ind_best:
    print("The selected model didn't match the selected error values!")

weight_best = weights[ind[0]]
threshold_best = thresholds[ind[1]]
self.weight_best = weight_best
self.threshold_best = threshold_best

if error_te[ind] < self.error_best:
    self.error_best = error_te[ind]
    self.model_best = model_best

if verbose:
    print(f"Best weights: {weight_best}")
```

```
            print(f"Best threshold: {threshold_best}\n")


        weights_and_threshold_data = {
            "weights"       : weights,
            "thresholds"    : thresholds,
            "error_tr"      : error_tr,
            "error_te"      : error_te,
            "model_best"    : model_best,
            "weight_best"   : weight_best,
            "threshold_best" : threshold_best
        }
        self.data["weights_and_threshold"] = weights_and_threshold_data


        if save:
            self.save()


        if plot:
            plot_errors_3D(
                weights, thresholds, "Weights", "Thresholds",
                error_tr, error_te,
                path.join(self.file_path, "weights_thresholds_training.pdf"),
                path.join(self.file_path, "weights_thresholds_testing.pdf")
            )
        return weight_best, threshold_best
```

### A.2.3 BaseBinaryThreshold.py

```
from abc import ABC
```

```
import inspect
import numpy as np

class BaseBinaryThreshold(ABC):

    def __init__(self, threshold = 0.5, *args, **kwargs):
        super().__init__(*args, **kwargs)
        self.threshold = threshold

    def predict(self, X):
        # Predicts class for X with an offset threshold
        return (self.predict_proba(X)[:,1] >= self.threshold).astype(int)

    def error(self, x, y, soft = False, beta = 0.01):
        return 1 - self.fscore(x, y, soft, beta)

    def fscore(self, x, y, soft = False, beta = 1.0):
        p = self.precision(x, y, soft)
        r = self.recall(x, y, soft)
        b = beta**2
        return (1 + b) * (p * r) / (b * p + r) if b*p+r != 0 else 0

    def precision(self, x, y, soft = False):
        y_p = self.predict_proba(x).T[1] if soft else self.predict(x)
        fp = np.mean(y_p[y == 0])
        tp = np.mean(y_p[y == 1])
        return tp / (tp + fp) if tp+fp != 0 else 0

    def recall(self, x, y, soft = False):
```

```
        y_n = self.predict_proba(x).T[0] if soft else 1-self.predict(x)
        y_p = self.predict_proba(x).T[1] if soft else self.predict(x)
        fn = np.mean(y_n[y == 1])
        tp = np.mean(y_p[y == 1])
        return tp / (tp + fn) if tp+fn != 0 else 0
```

### A.2.4 EnsembleBT.py

```
from .BaseBinaryThreshold import BaseBinaryThreshold
from sklearn.ensemble import StackingClassifier
from sklearn.utils.validation import _deprecate_positional_args

class EnsembleBT(BaseBinaryThreshold, StackingClassifier):

    @_deprecate_positional_args
    def __init__(self, estimators, final_estimator=None, *, cv=None,
                 stack_method='auto', n_jobs=None, passthrough=False,
                 verbose=0, threshold=0.5):
        super().__init__(estimators=estimators, final_estimator=final_estimator,
                         cv=cv, stack_method=stack_method, n_jobs=n_jobs,
                         passthrough=passthrough, verbose=verbose,
                         threshold=threshold)
```

### A.2.5 RandomForestBT.py

```
from .BaseBinaryThreshold import BaseBinaryThreshold
from sklearn.ensemble import RandomForestClassifier
from sklearn.utils.validation import _deprecate_positional_args
```

```
class RandomForestBT(BaseBinaryThreshold, RandomForestClassifier):


    @_deprecate_positional_args
    def __init__(self, n_estimators=100, *, criterion="gini", max_depth=None,
                 min_samples_split=2, min_samples_leaf=1,
                 min_weight_fraction_leaf=0., max_features="auto",
                 max_leaf_nodes=None, min_impurity_decrease=0.,
                 min_impurity_split=None, bootstrap=True, oob_score=False,
                 n_jobs=None, random_state=None, verbose=0, warm_start=False,
                 class_weight=None, ccp_alpha=0.0, max_samples=None,
                 threshold=0.5):
        super().__init__(n_estimators=n_estimators, criterion=criterion,
                         max_depth=max_depth,
                         min_samples_split=min_samples_split,
                         min_samples_leaf=min_samples_leaf,
                         min_weight_fraction_leaf=min_weight_fraction_leaf,
                         max_features=max_features,
                         max_leaf_nodes=max_leaf_nodes,
                         min_impurity_decrease=min_impurity_decrease,
                         min_impurity_split=min_impurity_split,
                         bootstrap=bootstrap, oob_score=oob_score,
                         n_jobs=n_jobs, random_state=random_state,
                         verbose=verbose, warm_start=warm_start,
                         class_weight=class_weight, ccp_alpha=ccp_alpha,
                         max_samples=max_samples, threshold=threshold)
```

### A.2.6 SupportVectorMachineBT.py

```
from .BaseBinaryThreshold import BaseBinaryThreshold
from sklearn.svm import SVC
from sklearn.utils.validation import _deprecate_positional_args

class SupportVectorMachineBT(BaseBinaryThreshold, SVC):

    @_deprecate_positional_args
    def __init__(self, *, C=1.0, kernel='rbf', degree=3, gamma='scale',
                 coef0=0.0, shrinking=True, probability=False, tol=1e-3,
                 cache_size=200, class_weight=None, verbose=False, max_iter=-1,
                 decision_function_shape='ovr', break_ties=False,
                 random_state=None, threshold=0.5):
        super().__init__(C=C, kernel=kernel, degree=degree, gamma=gamma,
                         coef0=coef0, shrinking=shrinking,
                         probability=probability, tol=tol,
                         cache_size=cache_size, class_weight=class_weight,
                         verbose=verbose, max_iter=max_iter,
                         decision_function_shape=decision_function_shape,
                         break_ties=break_ties, random_state=random_state,
                         threshold=threshold)
```

### A.2.7 GradientBoostingBT.py

```
from .BaseBinaryThreshold import BaseBinaryThreshold
from sklearn.ensemble import GradientBoostingClassifier
from sklearn.utils.validation import _deprecate_positional_args
```

```
class GradientBoostingBT(BaseBinaryThreshold, GradientBoostingClassifier):


    @_deprecate_positional_args
    def __init__(self, *, loss='deviance', learning_rate=0.1, n_estimators=100,
                 subsample=1.0, criterion='friedman_mse', min_samples_split=2,
                 min_samples_leaf=1, min_weight_fraction_leaf=0.,
                 max_depth=3, min_impurity_decrease=0., min_impurity_split=None,
                 init=None, random_state=None, max_features=None, verbose=0,
                 max_leaf_nodes=None, warm_start=False, validation_fraction=0.1,
                 n_iter_no_change=None, tol=1e-4, ccp_alpha=0.0, threshold=0.5):
        super().__init__(loss=loss, learning_rate=learning_rate,
                         n_estimators=n_estimators, subsample=subsample,
                         criterion=criterion,
                         min_samples_split=min_samples_split,
                         min_samples_leaf=min_samples_leaf,
                         min_weight_fraction_leaf=min_weight_fraction_leaf,
                         max_depth=max_depth,
                         min_impurity_decrease=min_impurity_decrease,
                         min_impurity_split=min_impurity_split, init=init,
                         random_state=random_state, max_features=max_features,
                         verbose=verbose, max_leaf_nodes=max_leaf_nodes,
                         warm_start=warm_start,
                         validation_fraction=validation_fraction,
                         n_iter_no_change=n_iter_no_change, tol=tol,
                         ccp_alpha=ccp_alpha, threshold=threshold)
```