Machine Learning Approaches to the Detection of Exoplanet Transits

Anna Zuckerman, Leah Zuckerman, Ashutosh Gandhi, and Andrew Floyd

July 2023

1 Introduction and Motivation

Before the groundbreaking 1992 detection of the planetary system PSR1257 + 12 (Wolszczan and Frail, 1992), the existence of exoplanets (planets that orbit stars other than our sun) was only hypothetical. In the few decades since, detecting new exoplanetary systems has become vital to understanding the nature and variability of other worlds around distant stars. Characterizing the population of exoplanets is key to understanding the mechanisms which drive the formation and evolution of planetary systems both inside and beyond our solar system, and even to understanding the conditions that may allow life to originate on planetary bodies.

The first step in this endeavor is to observe the stars in our local galaxy, and efficiently determine which host exoplanets. Though several methods exist to accomplish this (for instance, measuring the tiny motions of stars due to the gravitational influence of their planets, or directly imaging the planets in the limited cases when this is possible), the method that has so far produced the most detections is called transit photometry. In this method, astrophysicists measure the flux (amount of light) received from a star over a period of time, and attempt to identify the periodic dips in starlight that signify the presence of a planet orbiting between it's host star and our telescopes on Earth.

Never before has the detection and characterization of exoplanets via transit photometry been as promising and feasible as it is now, due to the increasing breadth and sensitivity of time domain optical surveys. Visually identifying transits in stellar lightcurves (flux as a function of time) is impractically time-consuming and tedious, but machine learning is uniquely suited to the task of identifying which lightcurves contain transits. In this project, we explore and evaluate several supervised machine learning algorithms to classify lightcurves by whether or not they are likely to contain exoplanet transits. Instead of performing a "vetting" on pre-"triaged" curves (see section 2), we work with a full set of Kepler observations (see Section 4). To train and evaluate our algorithms, we make use of catalogs of previously-classified stars.

Extracting accurate transit predictions is key to future studies of population-wide exoplanet system statistics, and for follow-up studies of individual systems and planets. Machine learning approaches to detection via transit photometry will greatly enhance the efficiency and practicality of transit searches, paving the way for novel exoplanet science in the future.

2 Literature Review

For decades, transits were identified manually in lightcurves with tedious visual inspection (e.g. Charbonneau et al., 2000). This was slow and labor-intensive. The earliest machine-aided detection methods included Box-Fitting-Least-Squares (BLS) algorithms, which scan curves for box-like signals (e.g. Kovács et al., 2002; Grziwa et al., 2012), and Bayesian-based analysis techniques, which characterize the likelihood of a signal representing a transit (Aigrain and Favata, 2002). In recent years, interest in supervised machine-learning techniques has risen. These methods usually rely on previous bodies of human-labeled light curves for the generation of training data. Once trained, they can scan through hundreds of curves and flag promising sources for later visual inspection, dramatically reducing the amount of human labor required.

Past work has explored many different types of supervised machine-learning detection techniques, as well as various methods of pre-processing input data. Most-commonly, pure time-series light curves (flux measurements recorded over a series of timestamps), are input to an algorithm as features describing an observation. Other features can be derived by processing the light curves into the frequency domain, (for example with Fourier transforms or Wavelet transforms), or by phase-folding to fit them within a specified period (see e.g. Stumpe et al., 2014; Pearson et al., 2018). Often, simple (non-ML) algorithms are used to first "triage" curves, flagging transit-like signals for further inspection. Machine learning methods are then applied in a "vetting" phase to predict whether these signals are true transits. Many previously explored algorithms for this task are based on Decision Tree Classification, using simple Decision Trees (e.g. Coughlin et al., 2016; Catanzarite, 2015), Random Forests (e.g. Armstrong et al., 2015; McCauliff et al., 2015), or Gradient Boosted Trees (e.g. Malik et al., 2021). Support Vector Machines and K-Nearest-Neighbors Algorithms have also been implemented with good results (e.g. Schanche et al., 2018).

While these basic classification algorithms do perform well, further work has shown that more sophisticated techniques, such as deep learning methods, may achieve even better results in a more streamlined way. Early deep learning models focused on improving previous "vetter" models, starting with a convolutional neural network developed by Shallue and Vanderburg (2018). Other work (e.g. Ansdell et al., 2018; Yu et al., 2019) applied small modifications to this model to incorporate more domain knowledge. More recently, architectures have been developed to detect likely transits without previous triaging. Two one-dimensional convolutional neural network architectures were developed concurrently by Zucker and Giryes (2018) and Pearson et al. (2018). The latter uses phase-folded signals and has been shown to achieve better accuracy on simulated light curves than benchmark BLS and Support-Vector-Machine algorithms (the former was not well-tested against other algorithms). To build on the accuracy achieved with phase-folding, while addressing the issue that it can be difficult to accurately measure the period of a suspected transit, Chintarungruangchai and Jiang (2019) proposed a two-dimensional convolutional neural network that takes as input a 2D stack of all segments (cycles), as opposed to a single averaged phase-folded curve. This model is able to achieve good accuracy even when predicted transit periods are significantly inaccurate.

In this project, we attempt similar algorithms to those above, but without an initial triage phase. Our goal is to explore a large range of algorithms, to find a technique that can accurately detect transit signals in a full set of lightcurve observations.

3 Model Evaluation Metrics

Our task is a classification problem, so there are a few standard metrics that we can use to evaluate the performance of the models we construct. We are primarily interested in the following:

- **Accuracy**: Accuracy is the most straightforward metric and measures the overall correctness of the predictions by calculating the ratio of correctly classified instances to the total number of instances.
- **Precision**: Precision calculates the ratio of true positives (correctly predicted positive instances) to the sum of true positives and false positives (incorrectly predicted positive instances). It indicates how many of the predicted positive instances are actually relevant. High precision indicates that the model is correctly identifying exoplanets and minimizing false positives.
- **Recall** (Sensitivity or True Positive Rate): Recall calculates the ratio of true positives to the sum of true positives and false negatives (missed positive instances). It indicates the proportion of actual positive instances that are correctly identified. High recall indicates that the model effectively captures exoplanets and minimizes false negatives.
- **F1 Score**: The F1 score is the harmonic mean of precision and recall. It provides a single metric that balances both precision and recall. The F1 score is useful to find an optimal balance between precision and recall.

For our purposes, accuracy alone will be inadequate due to the imbalanced nature of our datasets. These datasets typically have a larger number of light curves without any transit signal compared to those with true transits. What's more, while focusing on precision would ensure that the predicted "planet candidates" are mostly true, this may not be very useful since it could be achieved by making only a few "planet candidate" predictions and ensuring their correctness. Thus, it may result in missing many true transits. Instead, we will prioritize recall for assessing the algorithms' performance in transit detection. Recall measures the proportion of actual transit signals that are correctly identified by the algorithm. In our case, it is preferable to accept a higher number of false positives (incorrectly identified planet candidates) rather than missing potential planet signals. Since the overall purpose of our work is to flag stars for further observation and study, it is more important to catch all true transits than to make sure no false transits are flagged.

While we will focus on recall, we not that the trade-off between precision and recall, commonly known as the precision-recall trade-off, is likely to be significant. A model with high recall may have a lower precision and vice versa. This trade-off is typically evaluated using the F1 score, which combines precision and recall in a single metric. Thus, we will also consider F1-score as we evaluate our models.

4 Data

We use stellar lightcurves from the Kepler mission (Ricker et al., 2015). We chose to use data from the Kepler mission because it is one of the largest exoplanet surveys to date, producing 2708 confirmed detections of transiting exoplanet systems[1]. The program ran from 2009 to 2013, observing approximately 150,000 stars in multiple 90-day quarters, at a cadence of either 30 or 60 seconds between observations. It also used a uniquely high exposure time, and was thus able to observe dimmer, farther away targets than other missions such as the Transiting Exoplanet Survey Satellite (TESS) mission (the other largest exoplanet survey). The mission prioritized Main Sequence stars for which Earth-like planets would be detectable (Batalha et al., 2010). Kepler lightcurves can be publicly downloaded from the Barbara A. Mikulski Archive for Space Telescopes (MAST) archive (DOI: 10.17909/T9059R). The Kepler Science Processing Pipeline is described in Jenkins et al. (2010).

We also use the publicly available NASA Exoplanet Archive database [2] to create a labeled training dataset. We cross-reference the target names with the Kepler ID's in this database to label lightcurves in our training set as "confirmed" positive observations (ie. visual inspection or follow-up observing has confirmed the presence of a transiting exoplanet in the stellar system), or "false positive" negative observations (ie. visual inspection or follow-up observing has shown that the lightcurve does not contain transits). We are careful to construct sets with even class balance, since many more lightcurves in the initial set do not contain transits than do.

5 Pre-processing

Kepler lightcurves can be accessed from the MAST database using the interface provided by the `LightKurve` package in Python. These curves often contain extended intervals of missing data (due to the telescope entering safe mode, rotating towards Earth, or executing a quarterly roll) as well as individual data points flagged for quality issues (due to cosmic ray hits, reaction wheel zero crossings, impulsive outliers, thruster firings, etc.) (Thompson et al., 2016). Pre-processing our data to account for these missing values is one key challenge in this project.

Our first step is to mask out data with quality issues flagged during Kepler data acquisition. In this step, these datapoints are set to NaN, but they will be estimated later. The next pre-processing step must be included to address the fact that Kepler's observing run is divided into quarters, punctuated by rolls of the telescope. To stitch the quarters together, we first fit a linear trend to each of them. Removing this trend allows the quarters to align properly when combined into one continuous lightcurve. Next, we remove medium timescale stellar variability (which can mimic the periodicity of the transit signals or obscure real transit signals). We do this by smoothing our lightcurves with and median filter using a window size of 51 timesteps. This window size was chosen so that variation on the timescale of stellar variability could be removed while allowing transits themselves to persist (see Figure 1). Finally, we deal with NaN values, which most of our models cannot take as input, by interpolation. Because of the timeseries nature of our data and because we are interested in signals which occur on longer timescales than individual timesteps, we can safely interpolate without worrying about adding spurious signals.

[1] As of June 6, 2022, as reported by the NASA Exoplanet Archive (https://exoplanetarchive.ipac.caltech.edu)
[2] https://catcopy.ipac.caltech.edu/dois/doi.php?id=10.26133/NEA4

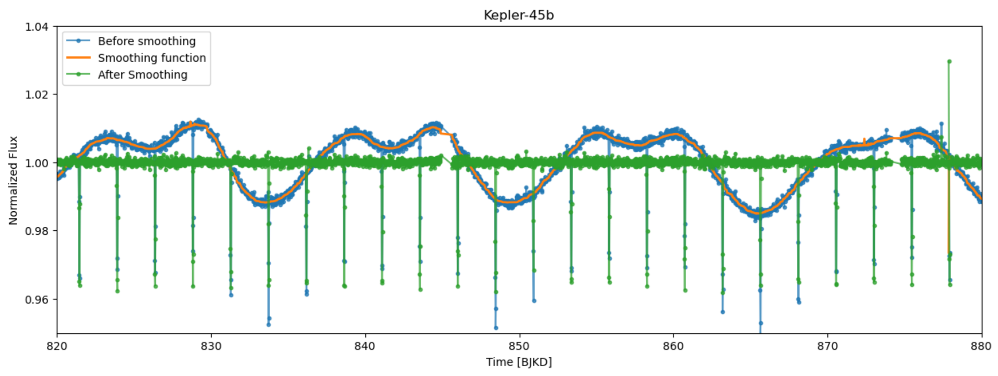

Figure 1: An example of smoothing a stellar lightcurve to remove stellar variability using a median filter (Kepler-45).

6 Feature Selection and Engineering

6.1 Feature Engineering

In an attempt to improve the performance of our models, we generate another dataset with an additional step in pre-processing. Because the timestep information in the original curves is largely meaningless (the change in time between observations is not consistent, and comparison between lightcurves at the same timestep is not meaningful), we construct "phase-folded" curves, which represent flux as a function of phase within some periodic cycle, as opposed to flux as a function of raw time. To conduct a phase-folding operation, we first remove stellar variability by smoothing using a rolling median filter. Then, we construct a boxed-least squares (BLS) periodogram to determine the period of maximum power for each lightcurve (Figure 2), and then phase fold the lightcurve on that period (Figure 3). This is a common method for dramatically increasing the signal-to-noise of any periodic signal that might be present, though with the limitation that only one periodic signal per lightcurve can be analyzed at a time. We will use both the time-domain and phase folded lightcurves in our subsequent analysis.

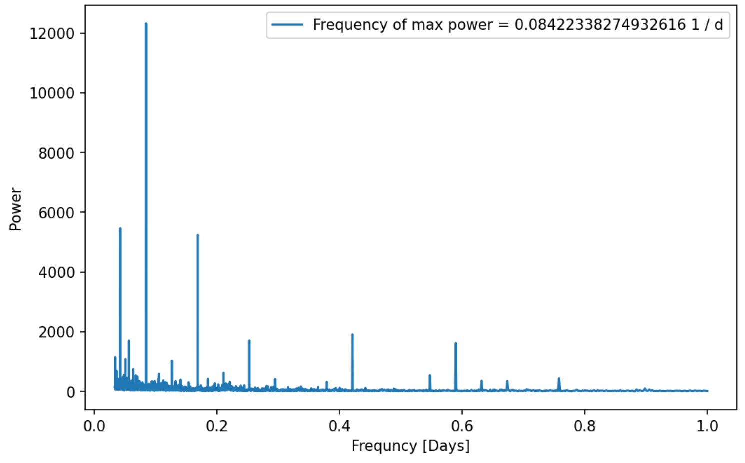

Figure 2: An example of BLS periodogram. The frequency of highest power is used to construct the phase-folded lightcurve.

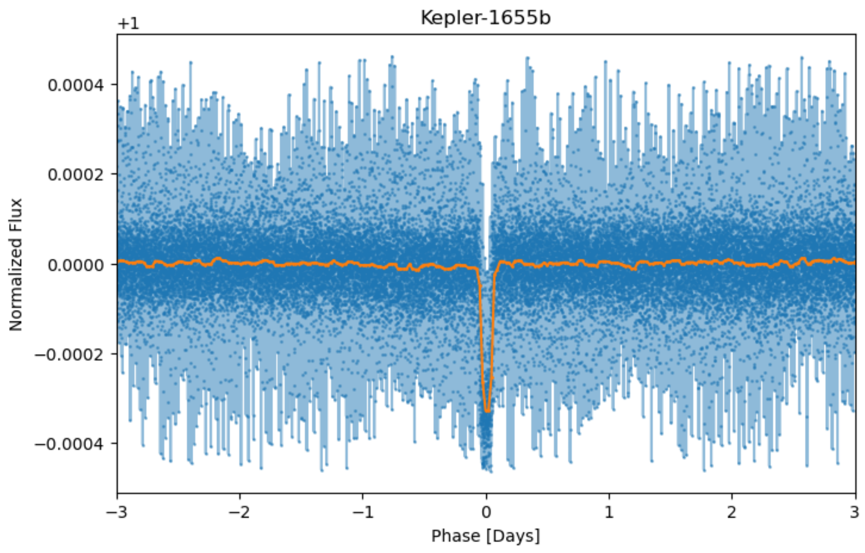

Figure 3: An example of a phase-folded lightcurve. Blue points represent folded flux values, and the orange curve represents the binned folded lightcurve. Features should be more meaningful after phase folding because a dip of a few percent at a phase of zero should appear in all positive observation, though some types of stellar variability could also produce this pattern.

6.2 Feature Selection

We also explore the possibility of using feature selection on our dataset. In general, the goal of feature selection is to improve performance, reduce overfitting, and enhance interpretability by focusing on the most informative features while discarding irrelevant or redundant ones. Feature selection is particularly important when dealing with high-dimensional datasets (like ours). We tried out two feature selection methods: Recursive feature selection (RFE) and Filter Method selection (using `sklearn` selectKBest).

1. **Recursive feature selection (RFE)**: RFE is an iterative feature selection method that starts with all features and repeatedly removes the least important features based on a model's performance until a desired number of features is reached. Since the number of features in our dataset was very large this method was limited by long runtime.

2. **Filter Method:** Filter methods pick up the intrinsic properties of the features measured via univariate statistics instead of cross-validation performance. They evaluate the relevance of features based on their individual characteristics, such as correlation with the target variable or statistical tests like ANOVA or chi-square tests. We experimented with the `SelectKBest` API to reduce the number of features. We used the default scrore_function of f_classif. For each feature, the f_classif method calculates the ANOVA F-statistic, which measures the variation between the class labels (target variable) for that feature.

We attempted to train multiple models (see Section 7) using datasets constructed using these feature selection methods. However, it was immediately apparent that they did not improve model performance for any models. On further thought, the fact that feature selection provides no advantage with our data makes sense. Features themselves (the raw time-step information) are not meaningful. The same timesteps will not contain the same information for each lightcurve. Thus, there is no subset of timesteps for which the flux values at those steps will be particularly significant for all curves.

7 Standard Supervised Learning Methods

7.1 Initial Attempt without Feature Engineering

7.1.1 K-Nearest Neighbors

The first algorithm we experiment with is a simple KNN classifier, in order to have a baseline to compare to other classification methods. The KNN algorithm works by storing training data as points in feature space. When a new observation is introduced, we find the k nearest neighbors of that new data point, using a pre-defined distance metric. Distance metrics include variations of Minkowski distance, for example the Euclidean distance (the square root of the sum of the squared differences between the feature values), and the Manhattan distance (the sum of the absolute value in differences). Since we are using a KNN for classification, we take the mode of the labels of the k neighbors to predict the label of the new observation. KNN is considered a "lazy learner" because training involves merely storing the training data, and only when we want to make a prediction is this training data processed in any way. As such, it does not produce a true "model" that one can examine, and so we look to other models to gain insight into our data.

We implement our KNN using the Python package `sklearn`. We first train on full (not phase-folded) lightcurves, after initially applying an interpolation to account for null data values which are not accepted by the `sklearn` KNN implementation. We split the data into a training, validation, and test set (with a third of the data in each set). We assess the training and validation accuracy using a range of k-values, and find the highest validation accuracy for k=6. Using this value for k, we then train a KNN on the training data and evaluate its performance using a confusion matrix as shown in Figure 4. We find an overall validation accuracy of 62.7% and test accuracy of 56.7%, somewhat better than random guessing but far from perfect. We achieve a validation F1 score of 64.8% and a test F1 score of only 45.3%. For our science purposes, we are most interested in correctly identifying all lightcurves which likely contain transits, and we are willing to accept a relatively high rate of false positives if we are able to achieve a low rate of false negatives. Thus, though the KNN has a low overall accuracy, the fact that the (slight) majority of miss-classifications are false positives is encouraging.

We do not anticipate that fine tuning hyperparameters will be able to significantly improve the performance of our KNN without first implementing more sophisticated feature engineering. This is because the KNN measures the distance between observations in Euclidean space by default, and the Euclidean distance between timeseries observations is not particularly meaningful due to the fact that very similar timeseries may be shifted or scaled relative to each other, producing a large distance values. Instead, we retrain the model on our phase-folded lightcurve data, as discussed in Section 7.2

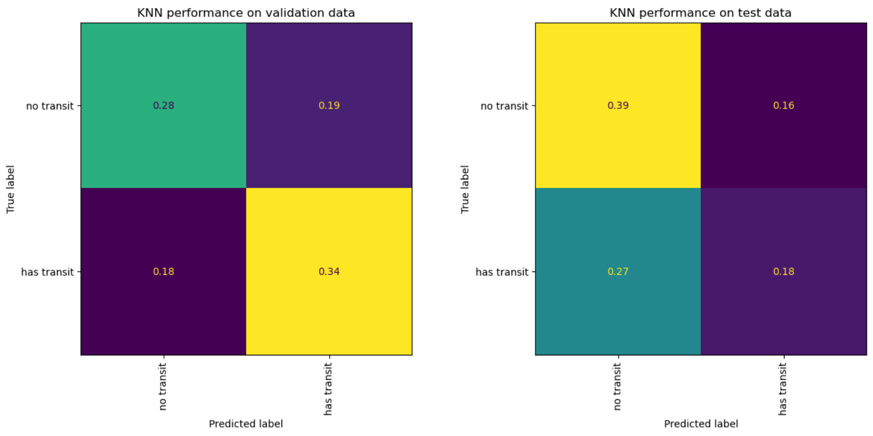

Figure 4: A confusion matrix for our KNN classifier trained using k=6. The classifier is far from perfect, as expected given the application of standard distance metrics.

7.1.2 Random Forest

In addition to a KNN, we implement a Random Forest (RF) classifier on our light curves. A RF is a collection of Decision Tree (DT) classifiers, each trained on subsamples of the data. To construct each DT, we iteratively split the data by determining the feature for which a split based on its value would result in the most information gain. For each potential split, information gain is calculated as the decrease in entropy between the root node and the post-split nodes:

$$\text{entropy}_{\text{parent}} = -\sum_{i=1}^{n_{classes}} p_i \log_2(p_i) \tag{1}$$

$$\text{entropy}_{\text{children}} = -\sum_{j=1}^{n_{values}} p_j \sum_{i=1}^{n_{classes}} p_i \log_2(p_i) \tag{2}$$

$$\text{info}_{\text{split}} = \text{entropy}_{\text{parent}} - \text{entropy}_{\text{children}} \tag{3}$$

The algorithm continues to select splitting features this way as we move down the tree, creating branches that contain observations of increasing homogeneity. It stops once all nodes contain only one class. Since we prune the trees to reduce over-fitting (see below), new observations are labelled with the mode of the training labels in the leaf node they reach. Once each DT in the RF has formed its class prediction, the final prediction is assigned using the majority vote.

We again use the `sklearn` Python package to implement this model. We split the data into a training, validation, and test set (with a third of the data in each set). We assess the training and validation accuracy using a range of values for the hyperparameters n_estimators and max_depth. We find that for any combination of hyperparameters, the model was subject to a high degree of overfitting, which persisted even when we attempted to avoid overfitting through pruning the tree by reducing the allowed maximum depth of each tree (unless we reduced the depth so much that validation accuracy was reduced to random guessing). We find that the validation accuracy is not sensitive to the choice of max_depth, and that the maximum validation accuracy occurs with value of n_estimators = 25. Using this value, we then train a Random Forest on the training data and evaluate its performance using a confusion matrix. The model has an overall validation accuracy of 66.0%, again somewhat better than random guessing but far from perfect. Based on the confusion matrix, we notice that the model has a higher propensity to misclassify observations as negative than as positive. This is the opposite of what we would like, so we add a set of class weights to encourage the model to give more weight to positive classifications. Using these weights, we achieve a similar validation accuracy of 67.2% and test accuracy of 62.7% and a validation F1 score of 71.8% and a test F1 score of 62.7%. We achieve a much more useful bias towards classifying observations as positive rather than negative. This is shown in the new confusion matrix in Figure 5.

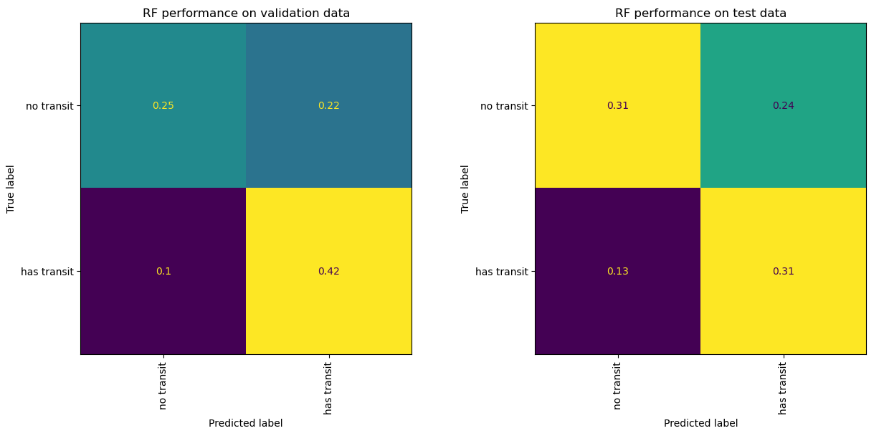

Figure 5: A confusion matrix for our RF classifier trained using n-estimators=25, using class weights to emphasize false positives over false negatives. The classifier does not give high performance.

7.1.3 Weighted Logistic Regression

Next, we attempt to train a Logistic Regression model. Logistic Regression is a simple model that constructs a decision boundary in feature space by finding the logistic function that best splits the data correctly by class label. This best-split is determined by minimizing the log-loss of the predicted and true values.

To construct this model, we again use `sklearn`'s Python implementation on interpolated light curves. We split the data into a training, validation, and test set (with a third of the data in each set). For this model, we are unable to achieve good performance for any set of hyperparameters, because the model either classifies all observations as positive or all observations as negative. We attempt to add a set of class weights to encourage the model to make both positive and negative classifications, but are unable to find a set of weights that allow for both positive and negative classifications.

7.1.4 Support Vector Machine(SVM) Classification

Another popular classification model is the Support Vector Machine(SVM). One of the primary reasons for testing SVM on our dataset is the advantage it has on the classification of high dimensional spaces. For our main dataset, we use 1000 features. Using the SVM kernel, high dimension non-linear relationships can be easily discovered in the data. The main objective of the SVM algorithm is to find the optimal hyperplane in an N-dimensional space that can separate the data points in different classes in the feature space. The model tries to maximize the margin between the closest points of different classes. The points that are closest to the hyperplane are called the support vectors. The prediction of the label is solely based on these support vectors and hence this model is very memory efficient. Moreover, unlike in KNN the prediction step is very quick.

We again use the `sklearn` python package and divide our dataset into a train/test split of 80/20. We train using K-fold cross-validation. In this technique, the data is divided into K folds or subsets. We train the model k times, each time using one of these folds as a validation set, and training the model on the remaining K-1 folds. Finally, the results from each validation step are averaged to produce a more robust estimate of the model's performance. The main purpose of cross-validation is to prevent overfitting, which occurs when a model is trained too well on the training data and performs poorly on new, unseen data. By evaluating the model on multiple validation sets, cross-validation provides a more realistic estimate of the model's generalization performance.

Next, for the identification of the hyperparameters we use the `GridSearchCV` API provided in the `sklearn` library. GridsearchCV provides an option to perform an exhaustive search over specified parameter values for an estimator. It tries out all the permutations of the parameters passed to it and does cross-validation training on each permutation. After trying out all possible permutations it returns the parameters with the highest accuracy score. The hyperparameters tuned are C (The regularization parameter), gamma (The kernel coefficient), and kernel (kernel type - rbf, sigmoid, poly). We tried out 5 different values of C and gamma each, along with the 3 different kernel types. We repeat this step with different values of C and gamma parameters based on the result of the previous attempt. The model had an overall cross-validated accuracy of 76% with the parameter of C as 1000, gamma as 1, and the kernel type as radial basis function(rbf). Even after training the model with the cross-validation technique, there was a small amount of overfitting as the test accuracy was 64%. The confusion matrix for the train and test is shown in Figure 6. Although the recall score with this model is close to 1, the low accuracy means the model is likely predicting far too many transits, and will not be able to generalize well.

7.1.5 AdaBoost Classifier

Since the Random Forest Model does well, we explore other ensemble-based classification models, the first being the AdaBoost algorithm. AdaBoost applies the Boosting ensemble technique. It attempts to build a strong classifier from a number of weak classifiers. It starts with building a model by using a weak estimator. Firstly, a model is built from the training data. Then the second model is built which tries to correct the errors present in the first model. This procedure is continued and models are added until either the complete training data set is predicted correctly or the maximum number of models are added.

AdaBoost operates on a fundamental principle, utilizing a series of weak learners (models that are slightly better than random guessing, like small decision trees) trained on modified versions of the data. The

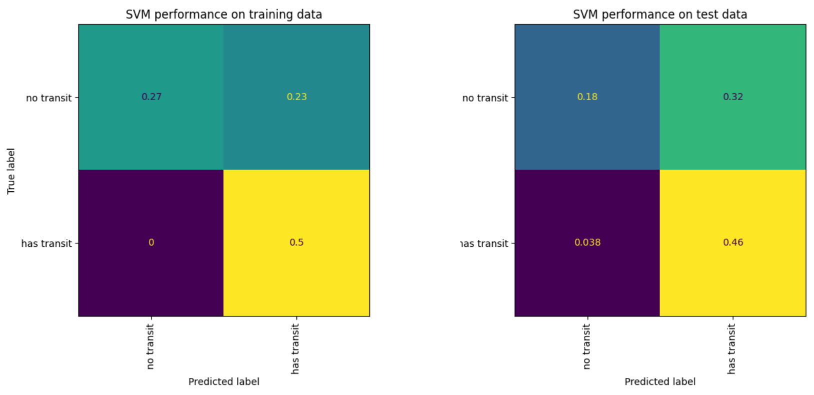

Figure 6: A confusion matrix for our SVM classifier trained using kernel as rbf, C=1000, and gamma=1.

predictions from these learners are then combined using a weighted majority vote (or sum) to arrive at the final prediction. During each boosting iteration, the data undergoes specific modifications by applying individual weights to the training samples. Initially, all weights are set to a default constant (usually 1/number_of_samples), leading to the first step where a weak learner is trained on the original data. In each subsequent iteration, the sample weights are adjusted, and the learning algorithm is applied again to the reweighted data. At each step, the samples that were incorrectly predicted by the boosted model in the previous iteration have their weights increased, while the weights are decreased for those samples that were predicted correctly. As a result, difficult-to-predict examples gain more influence as the iterations progress. Consequently, each subsequent weak learner is compelled to focus on the examples that were missed by the previous ones in the sequence.

Similar to the process described for SVM in Section 7.1.4, we split our dataset into a train/test ratio of 80/20 and use K-fold cross-validation along with `GridsearchCV` APIs of `sklearn` to find the optimal hyperparameter values. The base estimator used is a DecisionTreeClassifier. We assess the training and validation accuracy using a range of values for the hyperparameters n_estimators, learning rate, max_depth for the base estimator, and min_samples_leaf for the base estimator. The max_depth and min_samples_leaf are tuned to make sure that the model is not overfitted and these parameters help prune the tree. n_estimators is the maximum number of estimators at which boosting is terminated and the learning rate indicates the weight applied to each classifier at each boosting iteration. A cross-validation accuracy of 64% is achieved on the training data with the hyperparameters of base_estimator as max-depth=20 and min-samples-leaf=7, for the model learning_rate=0.01 and n_estimators=60. On the test data, the accuracy is 65.5% with a recall score of 0.66. The confusion matrix for the train and test is shown in Figure 7. Overall the model performs similarly to the Random Forest.

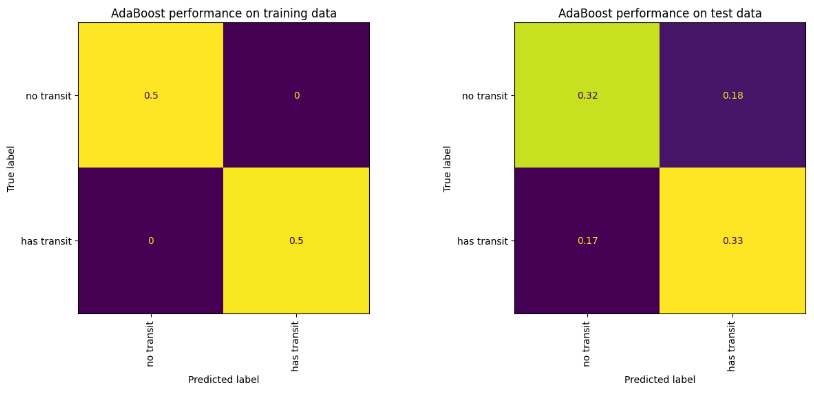

Figure 7: A confusion matrix for our AdaBoost classifier on both the train and test data.

7.1.6 Gradient Boosting Classification

The second type of ensemble boosting algorithm we experiment with is a GradientBoostingClassifier. Similarly to AdaBoost, it utilizes the boosting concept of building weak learners and improving them over each iteration by updating the weights for each model.

The Algorithm works in the following steps:

1. **Initialization:** The process begins by creating a simple model, usually a decision tree with just one node (also called a "stump"). This is the first weak learner. The initial model predicts the target variable but is often inaccurate since it's too simplistic.

2. **Residual Calculation:** The next step involves calculating the residuals, which are the differences between the actual target values and the predictions made by the first weak learner. These residuals represent the errors that the first model couldn't capture.

3. **Building Weak Learners Iteratively:** In each iteration, a new weak learner (decision tree) is built to predict the residuals from the previous step, not the original target values. The new weak learner is trained to minimize the residual error, thus correcting the mistakes of the previous model. The learning rate is applied to the predictions of each weak learner, controlling the contribution of each model to the final ensemble. A lower learning rate makes learning more conservative.

4. **Updating the Ensemble:** The predictions of the new weak learner are combined with the predictions from the previous models using a weighted sum or a weighted majority vote. The combined predictions represent an improved estimate of the target variable compared to the previous iterations.

5. **Iterative Process:** The above steps are repeated for a predefined number of iterations or until a specified performance threshold is reached. With each iteration, the ensemble becomes more accurate, as subsequent weak learners focus on the remaining errors.

6. **Final Prediction:** The final prediction of the Gradient Boosting Classifier is the summation of predictions made by all weak learners, taking into account the learning rate and their individual weights.

Again, we use K-fold cross-validation and `GridsearchCV` APIs of `sklearn` to find the optimal hyperparameter values. The base estimator used is a DecisionTreeClassifier. We assess the training and validation accuracy using a range of values for the hyperparameters n_estimators, learning rate, max_features, max_depth, and min_samples_leaf. The max_depth, max_features, and min_samples_leaf are tuned to make sure that the model is not overfitted and these parameters help prune the tree. n_estimators is the number of boosting stages to perform and the learning rate shrinks the contribution of each tree by the particular value. A cross-validation accuracy of 64% is achieved on the training data with the hyperparameters as max-depth=10, n_estimators=150, max_features=sqrt, learning_rate=0.1, and min-samples-leaf=5. On the test data, the accuracy is 72% with a recall score of 0.73. The confusion matrix for the train and test is shown in Figure 8.

7.2 Standard Supervised Learning Methods: Second Attempt using Feature Engineering

After training the above models, we realized that we could keep fine tuning hyperparameters on these models, or choose new types of classifiers to train, but that the underlying limitation was our data itself. Our data is inherently difficult to classify by methods that test the values of each feature (timestep) of a new observation against the same timestep of the observations in the training set, because transits will not appear with the same period and duration in each lightcurve. Thus, we need to implement some feature engineering. We do this by first smoothing our lightcurves using a rolling median filter with window size of 51 timesteps to remove stellar variability, and then phase folding the lightcurves, as described in 6.1. After phase folding, we bin folded flux values to remove noise and to ensure that each folded lightcurve has the same number of features.

Next, we present the results of each model after training on this feature-engineered dataset.

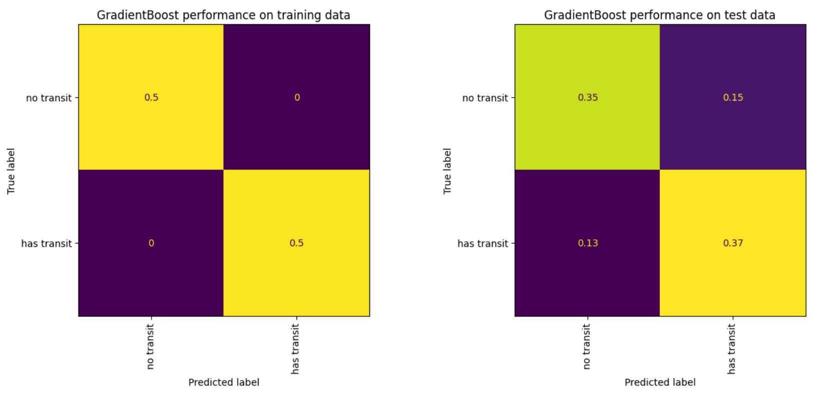

Figure 8: A confusion matrix for our GradientBoost classifier on the training and test data

7.2.1 KNN

The use of phase-folded curves should allow us to capitalize on the fact that after folding on the period of maximum power lightcurves with strong periodic signals will appear closer together as measured by the chosen distance metric.

We assess the training and validation accuracy using a range of k-values, and find the highest validation accuracy for k=6. Using this value for k, we then train a KNN on the training data and evaluate its performance using a confusion matrix as shown in Figure 9. The KNN has an overall validation accuracy of 62% and test accuracy of 68%, somewhat better than random guessing but far from perfect. We achieve a validation F1 score of 65% and a test F1 score of 64.7%.

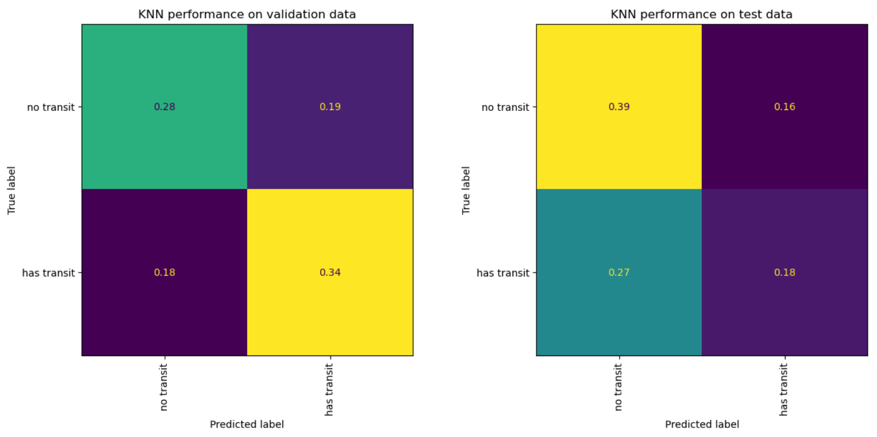

Figure 9: A confusion matrix for our KNN classifier trained using k=6 on engineered data.

7.2.2 Random Forest

We assess the training and validation accuracy using a range of values for the number of estimators, and find the highest accuracy for 36 estimators. Using this value, we then train a Random Forest classifier on the training data and evaluate its performance using a confusion matrix as shown in Figure 10. The model has an overall validation accuracy of 71.3% and test accuracy of 69.8%, somewhat better than the results on non-engineered features. We achieve a validation F1 score of 71.3% and a test F1 score of 65%. .

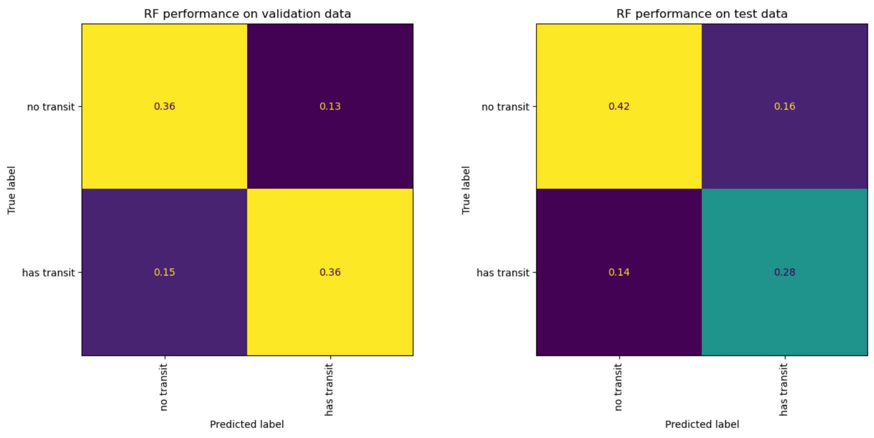

Figure 10: A confusion matrix for our Random Forest classifier trained using 36 estimators on engineered data.

7.2.3 Weighted Logistic Regression

Using the engineered observations, we are able to achieve somewhat better results from Logistic Regression than we were with the non-engineered features.

We assess the training and validation accuracy using a range of values for class weights, and find that the results are extremely sensitive to the chosen class weights. We are able to produce both positive and negative classifications when using the engineered dataset, but only for a very precise value of the class weights. We use a negative class weight of 0.52098196, where the positive class weight is 1 minus the negative weight. Using this value, we then train a Logistic Regression classifier on the training data and evaluate its performance using a confusion matrix as shown in Figure 11. The model has an overall validation accuracy of 61.7% and test accuracy of 58.7%, again only slightly better than random guessing. We achieve a validation F1 score of 68.1% and a test F1 score of 61.0%. The performance is shown in Figure 11.

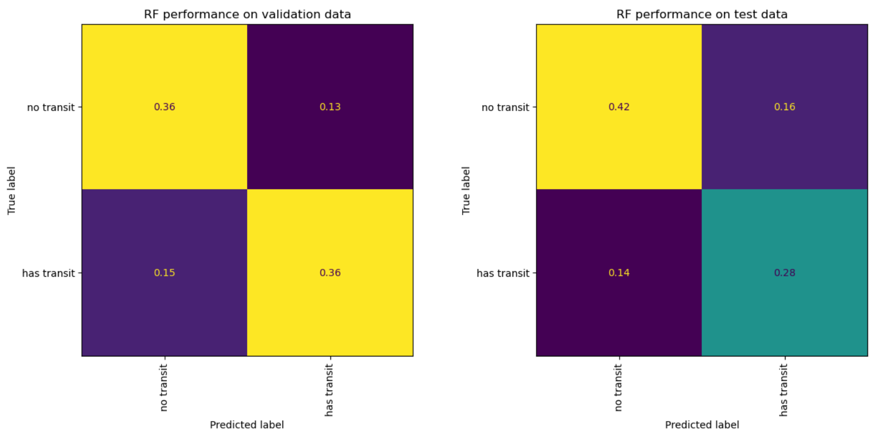

Figure 11: A confusion matrix for our Random Forest classifier trained using 36 estimators on engineered data.

7.2.4 Support Vector Machine (SVM)

We next train another SVM model using the phase-folded data. Surprisingly, the results are worse than when the non-phase-folded data is used. With the best hyperparameter values (which again turn out to be C=1000, gamma=1, and an rbf kernel), the cross-validated accuracy and test accuracy are only 61.75% and

57.5% respectively. The confusion matrix for the train and test data is shown in Figure 12. The recall score is 98% with 90% of all predictions as labeled "true".

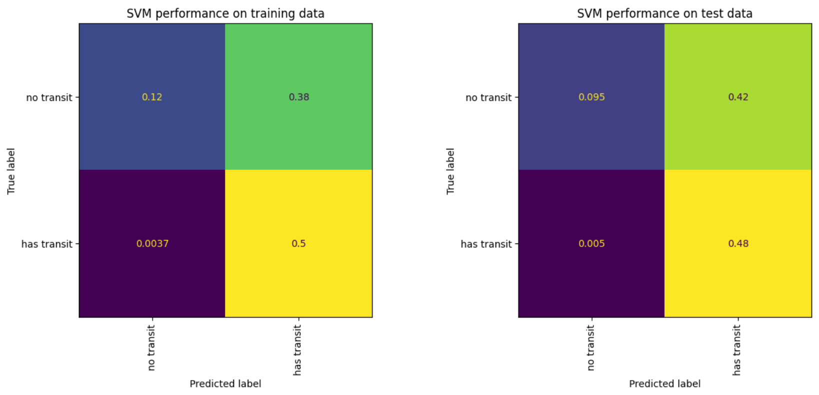

Figure 12: A confusion matrix for our SVM classifier trained on the engineered data.

7.2.5 AdaBoost Classifier

With the AdaBoost algorithm, the use of phase-folded data does allow us to achieve improved results. After finding the optimal hyperparameters (now base_estimator as max-depth=10 and min-samples-leaf=5, learning_rate=0.01 and n_estimators=70), the test accuracy is 75.50% and a recall score is 70%. The confusion matrix for the train and test is shown in Figure 13. Clearly, phase-folding is helpful for at least some models.

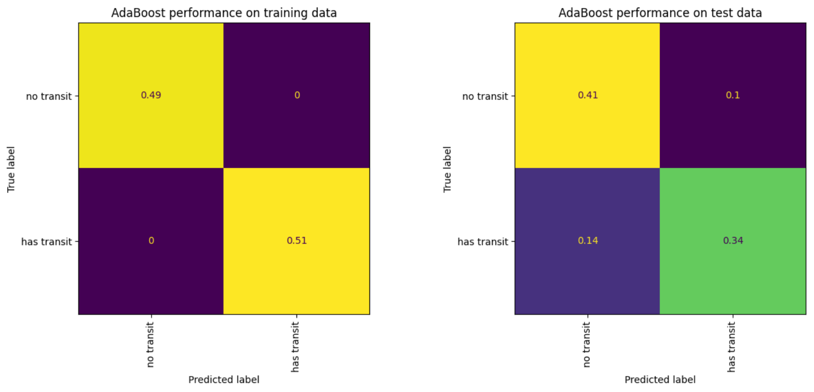

Figure 13: A confusion matrix for our AdaBoost classifier trained on the engineered data.

7.2.6 GradientBoost Classifier

As with AdaBoost, the GradientBoost Classifier achieves better results with the phase-folded dataset. With optimal hyperparameters (base_estimator as max-depth=30, n_estimators=150, max_features=sqrt, learning_rate=0.15, and min-samples-leaf=5), the test accuracy is 77% and recall score is 80%. The confusion matrix for the train and test is shown in Figure 14. This improvement further confirms that phase-folding can significantly improve model accuracy.

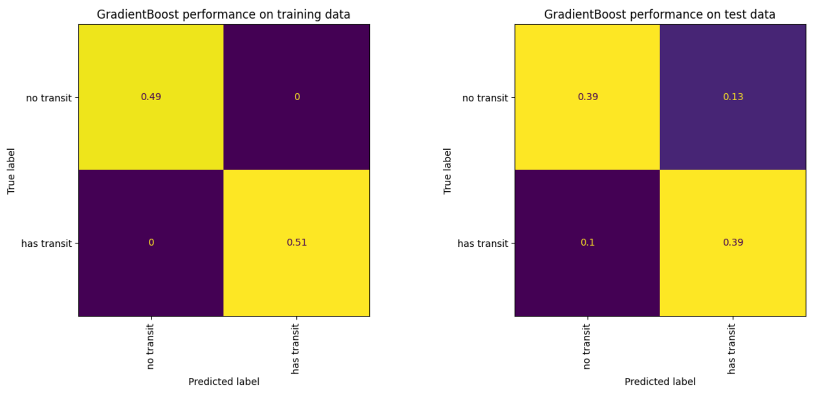

Figure 14: A confusion matrix for our GradientBoost classifier trained on the engineered data.

8 Deep Learning

8.1 General Structure of a Neural Network

In addition to the standard supervised learning methods discussed above, we have also made progress implementing deep learning networks for our classification goal. In general, a Neural Network is a network of nodes, or "neurons," that learn how to weight various learned features of an input dataset to produce the correct output. These nodes are arranged into layers, such that we can think of the nodes of each layer as the number of "features" that layer contains. Each node in a given layer is connected to some or all of the nodes of the previous one through a series of weighting functions. Layers can also contain "bias" nodes that are not connected to the previous layers. Different types of networks "pass" the data through the layers in different ways (see 8.2), but the end result is a set of predicted class probabilities. The predicted label is assigned as the most probable class.

To train a network, we give the model a cost function, C, to quantify the difference between its predictions and the true labels. Common cost functions make use of the Mean Squared Error or Cross Entropy Loss of the predictions and the labels. To train a network, we use a process called back-propagation to adjust the weights and biases to minimize the cost function. This works by calculating the gradient of the loss function with respect to each weight and bias term. Using the chain rule, we write this as:

$$\frac{dC}{dw_i} = \frac{dC}{d\hat{y}} \times \frac{d\hat{y}}{dw_i} \tag{4}$$

$$\frac{dC}{db_i} = \frac{dC}{d\hat{y}} \times \frac{d\hat{y}}{db_i} \tag{5}$$

Then, we update each weight and bias term using gradient descent, with a learning rate, α, added to set how much adjustment is done (e.g. how fast the model learns):

$$w_i = w_i - (\alpha \times \frac{dC}{dw_i}) \tag{6}$$

$$b_i = b_i - (\alpha \times \frac{dC}{db_i}) \tag{7}$$

8.2 Network Architectures

Using the `PyTorch` neural network library, we have attempted four neural network (NN) architectures: a simple linear feed-forward (fully-connected) network, a 1D convolutional network with a single feature channel, a 1D convolutional network with two feature channels, and a model using the LSTM sub-structure. We experiment with training on both full time-series lightcurves and on phase-folded lightcurves. For the 2-channel model, we use phase-folded lightcurves and their corresponding phase-steps.

A linear network is defined by the use of "linear" layers, which connect inputs and outputs via simple matrix multiplications. To transform the outputs of layer i into the inputs of layer $i+1$, we first perform multiplication by the matrix of weights, θ_i, where each weight in θ_i is associated with a connection between a node in layer $i-1$ and a node in layer i. Then, we add a bias term, b, (if desired) and apply an activation function, g. Using the sigmoid function as the activation function (a common choice), this looks like:

$$\vec{x}_{i+1} = g(\theta \cdot \vec{x}_i + b) \tag{8}$$

$$\text{with } g(x) = \frac{1}{1+e^{-x}} \tag{9}$$

A convolutional network is defined by the use of convolutional layers (often along with linear layers). In a convolutional layer, the input is transformed into the output via convolution with a kernel. For discrete vectors of features (as is the case in a NN), this convolution is identical to a cross-correlation, so the full computation can be written as:

$$\vec{x}_{i+1} = g(b + \sum_{k}^{n_{nodes-1}} w_{k,i} * x_{k,i-1}) \tag{10}$$

Here, the weights, w_i, represent the elements of the convolution kernel. It is these weights (along with any bias terms) that are learned as the model is trained.

Many other layer types exist, and a collection of layers called a "Long Short-Term Memory" (LSTM) cell is often used when working with sequential data. An LSTM model is a type of recurrent neural network (RNN). In an RNN, information about previous steps is stored in the "hidden state" associated with each layer. The hidden state for layer i is a function of hidden state of layer $i-1$, as well as the inputs to layer i. Then, the output of layer i is calculated by multiplying the layer's weight matrix by the value of its hidden state. An LSTM adds another input to each layer: a "long-term memory" state. The LSTM cell contains three "gates" that control what information is 1) allowed into the cell, 2) passed into the long-term memory state, and 3) passed into the hidden (short-term memory) state. The "input gate" generally consists of a "filter" vector generated by passing the current input and hidden state through a sigmiod function. When multiplied by the inputs, the filter transforms them to the range 0 to 1, so that those that are now 0 are discarded. The "forget" gate contains a similar filter created with a different sigmiod function. The incoming long-term state is multiplied by this vector. Finally, in the "output" gate, another filter is created with yet another sigmoid function, which controls output into the new hidden sate by multiplying the incoming hidden state. As the model is trained, it learns the weights of the sigmoid functions in the three gates that allow the most important information to be stored in the short and long term memory.

8.3 Network Performance

We use a linear architecture trained on the original light curves as a baseline method, and, as might be expected, this has yielded very poor results. After experimenting with various structure hyperparameters (e.g. modifying the number of hidden layers and the number of nodes per hidden layer), the best model achieved an overall accuracy of only around 45% (worse than random guessing). Even worse, with a recall of $\sim$29%, it is clear that we are missing an intolerably high number of true positives. Assuming these terrible results might be due to excessive noise still being present in the light curves even after pre-processing, we train a second linear model on phase-folded curves. Unfortunately, while the results improve slightly, we are not able to find any combination of structure parameters that achieve more than $\sim$59% accuracy, $\sim$36% recall and an F1-score of $\sim$23%.

The inability to pick out temporal trends on the correct scale, even with phase-folded data, points to a need for a model that learns from the "spatial" (in this case sequential) information stored in the data. To this end, we expect that a convolutional network will perform much better. Surprisingly, we are not able to achieve any significant improvement in accuracy (the best being $\sim$57%), and only a slight increase in recall ($\sim$54%) and F1-score ($\sim$28%) with a 1-channel convlutional network. This may be because a large amount of information in the light curves is contained not just in the sequence of the flux values (value 1, then value 2, then value 3, etc), but also in the phase-step information associated with them (value 1 at time 1, value 2 at time 2, value 3 at time 3, etc.)

To make use the phase-step information, we add a second channel to the convlutional network, so that both flux and phase arrays can be passed in. While overall accuracy and F1-score are still objectively quite poor (around 62% and 36%, respectively), these are significantly better than for previous networks. More importantly, the recall score jumps to around 99%, which is quite encouraging given our desire for a low false negative rate. As a final attempt to improve on overal performance, we look to network structures specifically designed for time-series classification. Specifically, we integrate PyTorch's LSTM module, which performs a Long Short-Term Memory operation on input data. Oddly, while recall rises to almost 100%, total accuracy drops to ∼49%, and F1 to ∼33%.

Overall, while the LSTM model achieves the best recall, the extremely low accuracy likely means it is predicting significantly too many transits. Thus, the 2-channel CNN seems to be the "best" model, as it has much better accuracy and still very high recall. However, while we do care most about avoiding false negatives, we cannot reasonably argue that near perfect recall "outweighs" the fact that the overall accuracy is only around 62%. Thus, we conclude that none of the neural network models do better than some of the simpler algorithms, so in selecting our best model we turn back to those.

9 Final Model Selection

Our best performance was achieved using a Gradient Boost Classifier, with 150 estimators, a maximum depth of 30, a minimum of 5 samples per leaf node, and a maximum number of features considered in a split taken as the square root of the total number of features. Trained on phase-folded light curves, we were able to achieve an accuracy of 77%, recall of 79%, and F1 score of 77%. Even though this is not very high performance, of the models we trained we select this is are best performing classifier.

10 Conclusions and Future Work

The groundbreaking detection of the exoplanetary system PSR1257 + 12 in 1992 marked a significant milestone in understanding planets outside our solar system. Since then, the detection of new exoplanetary systems has become crucial for studying the formation and evolution of planetary systems and investigating conditions for life. Recent advancements in time-domain optical surveys have made detections through transit photometry even more promising. In this project, we aimed to explore machine learning algorithms to automate the identification of exoplanet transits in stellar lightcurves, combining the frontiers of exoplanet science and the power of machine learning.

Our work spanned a wide range of algorithms and involved the application of these methods to various transformations of our lightcurve datasets. While some models did better than others, no model was able to achieve a high accuracy score while maintaining a high recall. Thus, our main takeaway is that classification of lightcurves is a challenging task. Transit signals from small planets are often difficult to extract from noise in flux measurements, and stellar variability can mimic periodic transit signals. What's more, lightcurves by nature make up a very high dimensional dataset, requiring complex methods of feature engineering to extract meaningful information. We believe that if we were to take this project further, the most impactful next step would be to explore new methods of feature engineering beyond those we worked with here. For example, we could fit transit models and extract transit parameters as features, or transform lightcurves using Dynamic Time Warping or Wavelet Analysis. We could also experiment with new classification methods designed specifically for time-series data, for example, the AR(I)MA (Autoregressive (Integrated) Moving Average) algorithm or other time-series neural network architectures.

11 Code Availability

All code for this project can be found at github.com/annazuckerman/transit_classification.

References

S. Aigrain and F. Favata. Bayesian detection of planetary transits. A modified version of the Gregory-Loredo method for Bayesian periodic signal detection. , 395:625–636, Nov. 2002. doi: 10.1051/0004-6361:20021290.

M. Ansdell, Y. Ioannou, H. P. Osborn, M. Sasdelli, J. C. Smith, D. Caldwell, J. M. Jenkins, C. Räissi, D. Angerhausen, and and. Scientific domain knowledge improves exoplanet transit classification with deep learning. *The Astrophysical Journal*, 869(1):L7, dec 2018. doi: 10.3847/2041-8213/aaf23b.

D. J. Armstrong, J. Kirk, K. W. F. Lam, J. McCormac, H. P. Osborn, J. Spake, S. Walker, D. J. A. Brown, M. H. Kristiansen, D. Pollacco, R. West, and P. J. Wheatley. K2 variable catalogue – II. machine learning classification of variable stars and eclipsing binaries in k2 fields 0–4. *Monthly Notices of the Royal Astronomical Society*, 456(2):2260–2272, dec 2015. doi: 10.1093/mnras/stv2836.

N. M. Batalha, W. J. Borucki, D. G. Koch, S. T. Bryson, M. Haas, T. M. Brown, D. A. Caldwell, J. R. Hall, R. L. Gilliland, D. W. Latham, S. Meibom, and D. G. Monet. SELECTION, PRIORITIZATION, AND CHARACTERISTICS OF iKEPLER/i TARGET STARS. *The Astrophysical Journal*, 713(2):L109–L114, mar 2010. doi: 10.1088/2041-8205/713/2/l109.

J. Catanzarite. Autovetter Planet Candidate Catalog for Q1-Q17 Data Release 24. *Astronomy and Astrophysics*, July 2015.

D. Charbonneau, T. M. Brown, D. W. Latham, and M. Mayor. Detection of Planetary Transits Across a Sun-like Star. , 529(1):L45–L48, Jan. 2000. doi: 10.1086/312457.

P. Chintarungruangchai and I.-G. Jiang. Detecting exoplanet transits through machine-learning techniques with convolutional neural networks. *Publications of the Astronomical Society of the Pacific*, 131(1000): 064502, may 2019. doi: 10.1088/1538-3873/ab13d3.

J. L. Coughlin, F. Mullally, S. E. Thompson, J. F. Rowe, C. J. Burke, D. W. Latham, N. M. Batalha, A. Ofir, B. L. Quarles, C. E. Henze, A. Wolfgang, D. A. Caldwell, S. T. Bryson, A. Shporer, J. Catanzarite, R. Akeson, T. Barclay, W. J. Borucki, T. S. Boyajian, J. R. Campbell, J. L. Christiansen, F. R. Girouard, M. R. Haas, S. B. Howell, D. Huber, J. M. Jenkins, J. Li, A. Patil-Sabale, E. V. Quintana, S. Ramirez, S. Seader, J. C. Smith, P. Tenenbaum, J. D. Twicken, and K. A. Zamudio. Planetary Candidates Observed by Kepler. VII. The First Fully Uniform Catalog Based on the Entire 48-month Data Set (Q1-Q17 DR24). , 224(1):12, May 2016. doi: 10.3847/0067-0049/224/1/12.

S. Grziwa, M. Pätzold, and L. Carone. The needle in the haystack: searching for transiting extrasolar planets in CoRoT stellar light curves. , 420(2):1045–1052, Feb. 2012. doi: 10.1111/j.1365-2966.2011.19970.x.

J. M. Jenkins, D. A. Caldwell, H. Chandrasekaran, J. D. Twicken, S. T. Bryson, E. V. Quintana, B. D. Clarke, J. Li, C. Allen, P. Tenenbaum, H. Wu, T. C. Klaus, C. K. Middour, M. T. Cote, S. McCauliff, F. R. Girouard, J. P. Gunter, B. Wohler, J. Sommers, J. R. Hall, A. K. Uddin, M. S. Wu, P. A. Bhavsar, J. Van Cleve, D. L. Pletcher, J. A. Dotson, M. R. Haas, R. L. Gilliland, D. G. Koch, and W. J. Borucki. Overview of the Kepler Science Processing Pipeline. , 713(2):L87–L91, Apr. 2010. doi: 10.1088/2041-8205/713/2/L87.

G. Kovács, S. Zucker, and T. Mazeh. A box-fitting algorithm in the search for periodic transits. , 391: 369–377, Aug. 2002. doi: 10.1051/0004-6361:20020802.

A. Malik, B. P. Moster, and C. Obermeier. Exoplanet detection using machine learning. *Monthly Notices of the Royal Astronomical Society*, dec 2021. doi: 10.1093/mnras/stab3692.

S. D. McCauliff, J. M. Jenkins, J. Catanzarite, C. J. Burke, J. L. Coughlin, J. D. Twicken, P. Tenenbaum, S. Seader, J. Li, and M. Cote. AUTOMATIC CLASSIFICATION OFiKEPLER/iPLANETARY TRANSIT CANDIDATES. *The Astrophysical Journal*, 806(1):6, jun 2015. doi: 10.1088/0004-637x/806/1/6.

K. A. Pearson, L. Palafox, and C. A. Griffith. Searching for exoplanets using artificial intelligence. , 474(1): 478–491, Feb. 2018. doi: 10.1093/mnras/stx2761.

G. R. Ricker, J. N. Winn, R. Vanderspek, D. W. Latham, G. Á. Bakos, J. L. Bean, Z. K. Berta-Thompson, T. M. Brown, L. Buchhave, N. R. Butler, R. P. Butler, W. J. Chaplin, D. Charbonneau, J. Christensen-Dalsgaard, M. Clampin, D. Deming, J. Doty, N. De Lee, C. Dressing, E. W. Dunham, M. Endl, F. Fressin, J. Ge, T. Henning, M. J. Holman, A. W. Howard, S. Ida, J. M. Jenkins, G. Jernigan, J. A. Johnson, L. Kaltenegger, N. Kawai, H. Kjeldsen, G. Laughlin, A. M. Levine, D. Lin, J. J. Lissauer, P. MacQueen, G. Marcy, P. R. McCullough, T. D. Morton, N. Narita, M. Paegert, E. Palle, F. Pepe, J. Pepper, A. Quirrenbach, S. A. Rinehart, D. Sasselov, B. Sato, S. Seager, A. Sozzetti, K. G. Stassun, P. Sullivan, A. Szentgyorgyi, G. Torres, S. Udry, and J. Villasenor. Transiting Exoplanet Survey Satellite (TESS). *Journal of Astronomical Telescopes, Instruments, and Systems*, 1:014003, Jan. 2015. doi: 10.1117/1.JATIS.1.1.014003.

N. Schanche, A. C. Cameron, G. Hé brard, L. Nielsen, A. H. M. J. Triaud, J. M. Almenara, K. A. Alsubai, D. R. Anderson, D. J. Armstrong, S. C. C. Barros, F. Bouchy, P. Boumis, D. J. A. Brown, F. Faedi, K. Hay, L. Hebb, F. Kiefer, L. Mancini, P. F. L. Maxted, E. Palle, D. L. Pollacco, D. Queloz, B. Smalley, S. Udry, R. West, and P. J. Wheatley. Machine-learning approaches to exoplanet transit detection and candidate validation in wide-field ground-based surveys. *Monthly Notices of the Royal Astronomical Society*, 483(4): 5534–5547, nov 2018. doi: 10.1093/mnras/sty3146.

C. J. Shallue and A. Vanderburg. Identifying exoplanets with deep learning: A five-planet resonant chain around kepler-80 and an eighth planet around kepler-90. *The Astronomical Journal*, 155(2):94, jan 2018. doi: 10.3847/1538-3881/aa9e09.

M. C. Stumpe, J. C. Smith, J. H. Catanzarite, J. E. Van Cleve, J. M. Jenkins, J. D. Twicken, and F. R. Girouard. Multiscale Systematic Error Correction via Wavelet-Based Bandsplitting in Kepler Data. , 126 (935):100, Jan. 2014. doi: 10.1086/674989.

S. Thompson, D. Fraquelli, J. E. van Cleve, and D. A. Caldwell. Kepler: A search for terrestrial planets (kepler archive manual). may 2016.

A. Wolszczan and D. A. Frail. A planetary system around the millisecond pulsar PSR1257 + 12. , 355 (6356):145–147, Jan. 1992. doi: 10.1038/355145a0.

L. Yu, A. Vanderburg, C. Huang, C. J. Shallue, I. J. M. Crossfield, B. S. Gaudi, T. Daylan, A. Dattilo, D. J. Armstrong, G. R. Ricker, R. K. Vanderspek, D. W. Latham, S. Seager, J. Dittmann, J. P. Doty, A. Glidden, and S. N. Quinn. Identifying exoplanets with deep learning. III. automated triage and vetting of iTESS/i candidates. *The Astronomical Journal*, 158(1):25, jun 2019. doi: 10.3847/1538-3881/ab21d6.

S. Zucker and R. Giryes. Shallow Transits—Deep Learning. I. Feasibility Study of Deep Learning to Detect Periodic Transits of Exoplanets. , 155(4):147, Apr. 2018. doi: 10.3847/1538-3881/aaae05.