Gabriel Nicholson
Steven Shi
STAT 27850
3/13/2022

Group Project 2: Kepler Objects of Interest

Section 1: Question

Our question is as follows: given a classifier method that maps the properties of the potential planet and its star (call these the "X" variables) to whether or not it is confirmed as an exoplanet (call this "Y"), can we estimate the accuracy of this method in the set of unclassified planets, where the distribution of the X variables is sometimes very different from the distribution of these variables in the classified planets? Even if we assume that Y depends on the X variables in the same way across the observations that are officially classified (training set) and the observations that are not yet classified (test set), we may still have inference issues if the distribution of the X variables differs across these sets. For example, if the objects in our training set tend to be larger than the objects in the test set, and larger objects are more likely to be confirmed as exoplanets, the joint distribution of X and Y will be different in the test and training set. As a result, we need to adjust for this when making inferences about our predictions for Y in the test set by using the classifier fitted on the training set. Our goal is to find a method for creating confidence sets that contain the true value of Y in the testing data with a certain frequency (i.e. $1 - \alpha$) even though the distribution of X in this data can be very different from the distribution of X in the training data.

Section 2: Data

We used the given data with a few modifications. First, we decided to keep koi_period, koi_time0bk, koi_impact, koi_duration, koi_depth, koi_prad, koi_teq, koi_steff, koi_slogg, and koi_srad as predictors. We did not include ra, dec, or koi_insol as predictors, and the reasoning for this is explained in the discussion section. In order to use our methodology, we dropped all observations for which at least one of these variables was missing. This deleted 363 out of 9,564 observations; 300 of them were in the training set, while 63 of them were in the test set. Overall, there were 7,016 observations in the training set and 2,185 observations in the test set.

Section 3: Methods

Our primary methodology was to use logistic regression as our classification method and weighted full conformal prediction for inference. To find the groups of observations we need to weight differently based on differences in the predictor distributions in the two datasets, we first

used PCA to reduce the dimensionality of the predictors and then k-means to cluster the observations based on these principal components.

The first step to obtain valid inference was to figure out which predictors had different distributions across the training and test data. We wanted to quantify this difference using the Kolmogorov-Smirnov (K-S) statistic for each predictor. Before doing this, we noticed that the distributions of many of the predictors had long right tails; this can be seen in figure 1. This may skew the results of the K-S test. Since the training set has many more observations than the test set, the observations that have predictor values in these tails are more likely to be present in the training data even if the underlying distributions of the training and test sets were the same due to a limited sample size. As a result, we had to find a way to account for these tails when looking for differences in distributions of the predictors without simply dropping them.

To do this, we transformed each variable by replacing each value with its percentile rank across observations. For example, if there were 4 observations and a variable took the values 1, 2, 3, and 1000, we would replace them with $\frac{1}{4}, \frac{1}{2}, \frac{3}{4}$, and 1. This reduces the influence of long tails on the K-S statistic, but if the values in the test set were consistently smaller or larger than the values in the training set, we could still detect this by looking at the ranked data.

To get the null distribution of the K-S statistic without making too many assumptions, we permuted the data across the training and test sets 20,000 times and computed the K-S statistic each time. However, after doing this, we found that the K-S statistic in the original dataset was larger than 95% of the permuted K-S statistics for 8 of the 10 predictors. Furthermore, the other two predictors had K-S statistics that were larger than 90% of their permuted K-S statistics, meaning that we could not safely rule out the possibility that the distribution of these variables were the same across the datasets either. This was problematic for several reasons. First, binning would not work because even if we used three bins for each of the significantly different (at the $\alpha = 0.05$ level) predictors, we would need $3^8 = 6,561$ bins. This is an issue given that we only have 9,201 total observations, leaving very few or no observations in each bin. Second, k-means clustering is unreliable in high dimensions. As a result, to obtain a valid set of groups to weight with full conformal prediction, we needed to reduce dimensionality with a different method.

To do this, we used principal component analysis (PCA) on all 10 X variables to reduce dimensionality. We included both the training and testing data in the process, since it is necessary to obtain principal components that are comparable across datasets. If the distribution of the X variables differ across the datasets and the principal components properly reduce the dimensionality of X, the difference in the distribution of the principal components across datasets should capture this difference. We decided to run PCA on the ranked variables rather than the original ones for two reasons. First, PCA places greater importance on the variables with higher variances. However, the predictors are measured in various different units, and some of the variables have a larger variance just because of the units they are measured in rather than a theoretical justification for their importance. Thus, ranking prevents PCA from placing undue importance on some predictors over the others. Second, since we planned to use k-means to detect clusters in the principal components, we wanted to ensure that the principal components

were on relatively similar scales, which is necessary for k-means to perform well. By ensuring that the variables entering the PCA were of the same scale, we knew that the principal components would be of similar scales. This prevented us from having to rescale the principal components after PCA, which is desirable because k-means places more emphasis on the variables with higher variance, and the first principal components have higher variance and are more representative of the original data by definition.

As seen in figure 2, we found that 5 components explained 92 percent of the variation in our standardized variables, so we chose this as the number of components to use (we wanted the PC's to explain at least 90% of the variation). After obtaining these 5 principal components, we used k-means clustering to obtain the groups we would look for differences in. The k-means algorithm included both the training and test data because we are looking for differences across these datasets, and putting them together allows us to detect clusters that may have more or fewer points in each set. Again, we chose k-means over binning because it avoids the curse of dimensionality.

To choose the number of clusters for k-means, we used the elbow method as a starting point. First, we computed the within-cluster sum of squares (WCSS) for each number of clusters between 2 and 20. In figure 3, we can see that the WCSS begins to level off at around 5 clusters. Our decision to use 5 clusters is further justified in section 4b using simulations with semisynthetic data. The number of points in each cluster can be seen in figure 4. The distribution of these clusters across the training and test sets can be seen in figures 5 and 6.

Next, we implemented the weighted full conformal strategy. Since the outcome was binary, we can formalize the process in the following manner:

1. For each cluster $c = 1, 2, \ldots, C$ obtained by k-means, let $\tilde{q}_c$ denote the proportion of the observations in the training data that lies in cluster j and q_c denote the proportion of the observations in the test data that lies in cluster c
2. Let $(X_1, Y_1), \ldots, (X_N, Y_N)$ represent our training data (here X_i is a vector containing the predictors, not the principal components, and Y_i is an indicator that takes value 1 if the object is a confirmed exoplanet)
3. For each point in the test set, label its predictor values with X_{N+1}
4. For $y \in \{0, 1\}$, let $\hat{p}_y(X_i)$ denote the predicted probability of observation i being a confirmed exoplanet based on logistic regression run on $(X_1, Y_1), \ldots, (X_N, Y_N), (X_{N+1}, y)$
5. For each observation, define $S_i^y = \frac{1}{\hat{p}_y(X_i)}$. We have

   $\hat{C}_{N+1}(X_{N+1}) = \left\{ y: S^y_{N+1} \leq Q_{1-\alpha}\left(\sum_{i=1}^{N+1} w_i \delta_{S_i^y} \right) \right\}$ where $w_i = \frac{q_{cluster(i)} / \tilde{q}_{cluster(i)}}{\sum_{j=1}^{N+1} \left(q_{cluster(j)} / \tilde{q}_{cluster(j)} \right)}$ and

   $cluster(i)$ denotes the cluster number that observation i lies in (meaning that

$cluster(i) \in \{1, 2, \ldots, C\}$). Recall that $\delta_{S_i^y}$ is the point mass at S_i^y, making $\sum_{i=1}^{N+1} w_i \delta_{S_i^y}$ a discrete distribution.

We chose full conformal prediction over other methods such as split conformal or Jackknife+ for several reasons. The most important reason was that out of the methods we studied, full conformal has on average the best statistical efficiency. Although our sample size is relatively large, we are also using a large number of predictors for logistic regression. As a result, it is best to use as many of our observations as possible. Furthermore, full conformal is not as computationally expensive in this scenario as it may be in other cases. Since $y \in \{0, 1\}$, we only need to fit the model twice for each point in the test set, as opposed to once for split conformal. By using weighting, we are adjusting for the difference in distributions of the X variables by placing more weight on the scores from observations with X values lying in clusters that are more common in the testing data than the training data. If our PCA-based dimensionality reduction and k-means clustering algorithm correctly accounted for the differences in the distributions of the predictors between the training and test sets, we should have $Y_i \in \hat{C}_i(X_i)$ over all observations in the test set with frequency at least $1 - \alpha$.

Section 4: Checking Assumptions

The most important assumption we are making to have proper inference despite covariate shift is that our clusters properly capture the difference in the distributions of the covariates across the training and test data. There are several conditions that must hold for this to occur.

Section 4a: PCA Assumptions

First, PCA must have properly reduced the dimensionality of our covariates. Since PCA generates linear combinations of our covariates, we must check if the covariates are indeed linearly related so that these combinations do not miss important relationships between the predictors. The scatterplots of a random subset of covariates against each other can be seen in figure 7. As we can see, there is generally either a linear relationship or no clear relationship between these variables. Thus, this assumption appears to be satisfied.

Another assumption related to PCA is that the variables are sufficiently correlated. Statistically, we can test this using Bartlett's test of sphericity, for which we found a p-value of 0. This indicates that the data was suitable for dimensionality reduction. From an intuitive standpoint, this makes sense because some of the variables are clearly related. For example, koi_impact measures the distance between the object and its star, while koi_duration measures the object's transit time, or the time it takes for the object to pass from one side of the star to the other. We would expect that objects that are further from its star would also have a longer transit time because the object would have to travel a greater distance. This relationship is also linear,

since the distance the object would have to travel (part of the circumference of a circle) is linearly related to the distance between the object and the star (the radius of the circle). As a result, this is a real-world reason for why we would believe that the PCA assumptions hold. Another example is that koi_teq measures the temperature of the planet, while koi_steff measures the temperature of the star. All else equal, hotter stars would result in hotter objects, which is another reason to expect correlation in our variables and the potential for dimensionality reduction.

Another condition for PCA to work well is the absence of large outliers, which can skew the results. However, we do not expect this to be an issue because we rescaled each variable by ranking the values of each observation and obtaining the percentile score of that observation. As a result, all predictors are on the same scale, and outliers have been eliminated since the ranking of two points is the same regardless of how much larger one point is than the other.

Even if these assumptions hold, our choice of 5 principal components is somewhat arbitrary. As we explain more in section 6b, it is entirely possible that more principal components are needed to capture a sufficient amount of information from our X variables. This could be problematic, but we did not have time to explore how performance changes depending on the number of components chosen using simulations.

Section 4b: k-means Assumptions

The next set of assumptions has to deal with our k-means algorithm. First, we know that k-means is skewed if the scale of the variables are very different. However, since we scaled the original data through ranking, the principal components created from these variables are already on a similar scale, as we can see in figure 8.

A very important assumption we are making is that 5 k-means clusters is sufficient to capture the difference in distributions of the principal components. In the graph of within-cluster sum of squares with various cluster numbers in figure 3, the inflection point that we chose at 5 clusters is not extremely clear. As a result, 5 clusters may be insufficient to identify differences in the distributions across the training and testing data. To see if this is the case, we used simulations run on semisynthetic data following the procedure outlined below.

First, since we do not know the true classification of the objects in our test set, we generated fictitious classifications for all observations in both datasets following a known procedure. This took the following steps:

1. For each observation, draw a Bernoulli random variable (call it Z_i) with a parameter depending on the values of the principal components. Specifically, we will have $Z_i \sim Bernoulli\left(\left(\frac{<P_i, \bar{P}>}{||P_i|| \cdot ||\bar{P}||} + 1\right)/8\right)$ where P_i represents the principal component values for observation i and $\bar{P} = \bar{P}_{test} - \bar{P}_{train}$ where $\bar{P}_{test}$ is the vector of means of the principal components in the test set and $\bar{P}_{train}$ is the vector of means of the principal

components of the training set. Here, we are fixing the number of principal components at 5, although we consider changing this in the discussion section. Note that $<\cdot, \cdot>$ represents the dot product and $|| \cdot ||$ represents the norm. Also, this is a valid distribution because $\frac{<P_i, \bar{P}>}{||P_i|| \cdot ||\bar{P}||}$ lies between -1 and 1 (it represents the cosine of the angle between P_i and $\bar{P}$), so adding 1 and dividing by 8 will shift the parameter into the range $[0, 1]$. Note that we divide by 8 instead of 2 in order to make the data less noisy.

2. If $Z_i = 0$, then $Y_i \sim Bernoulli(p)$ where $p = \frac{e^{\beta_0 + \beta_1 koi\_period_i + \ldots + \beta_{10} koi\_srad_i}}{1 + e^{\beta_0 + \beta_1 koi\_period_i + \ldots + \beta_{10} koi\_srad_i}}$. For simplicity, we take β_k to be 1 divided by the standard deviation for its associated for all k to prevent the exponent from becoming too large, making p very close to 1.
3. If $Z_i = 1$, then simply draw $Y_i \sim Bernoulli\left(\frac{1}{2}\right)$.

Using this data, we could run our weighted conformal method with various numbers of clusters and check coverage rates in the test set since the fictitious Y values are known. This is helpful because we have the same X variables, leaving the clusters the same as the original data. We designed this procedure so that Y is noisier for observations in the direction of $\bar{P}$, since $\frac{<P_i, \bar{P}>}{||P_i|| \cdot ||\bar{P}||}$ will be larger. Importantly, the distribution of $Y|X$ does not depend on whether the point is in the training or test set. However, the test set should contain more observations in the range of X values that lead to more noise in the distribution of $Y|X$, and we would have to adjust for this when conducting inference. We would expect our prediction sets from our weighted conformal method to contain the true value of Y with a frequency of about $1 - \alpha$ if we have a "good" number of clusters, while the coverage rate would be too low if we chose the wrong number of clusters. Here, we run the simulation using $\alpha = 0.1$.

From figure 9, we see that overall, the coverage rate decreases as the number of clusters increases. However, coverage remains above 0.8 for all clusters between 1 and 20. This decrease is accompanied by an increase in the proportion of intervals that only contain 1 value—either "confirmed" or "false positive". The trend in this proportion is shown in figure 10. This is a form of the trade-off between power and conservativeness. When we have a small number of clusters, there are a large number of observations for which we include both "confirmed" and "false positive" in $\hat{C}_{N+1}(X_{N+1})$. This guarantees coverage, but it does not help us narrow down the classification of the object. As we increase the number of clusters, we are better able to distinguish between when we only need to include one of the two classifications in $\hat{C}_{N+1}(X_{N+1})$. This makes sense, as more clusters allows us to capture more of the heterogeneity in the distributions of the predictors across the training and testing data. However, the cost of greater precision is that we also have (slightly) lower coverage. This may be because with many clusters, we could have abnormally high weights due to limitations in our sample size. Note that these

figures are generated by only using 1 draw of the data for each number of clusters; the coverage rates and proportion of prediction sets with 1 element could be more precise by averaging over many draws of the data. Unfortunately, this process was already very computationally intensive (taking 30 minutes); multiple simulations for each number of clusters would take several hours. Since the trend is very clear and consistent across several runs of our code, we believe that the results should be very similar if we averaged over many simulations.

If our only interest was obtaining a coverage rate of at least $1 - \alpha$, this simulation would suggest that we may consider using 2 clusters or one cluster, which is equivalent to not using k-means at all. However, this is not helpful because only about 60% of the prediction sets contain just one element when using two clusters, while less than 20% of the prediction sets contain only 1 element when we use one cluster. In other words, this does not allow us to predict much about the objects' classifications even if we technically have valid inference. Instead, this simulation provides further support for our decision to use 5 clusters. We still obtain an 85% coverage rate in this simulation, which is very close to $1 - \alpha$, and we also have just one element in approximately 70% of our prediction sets. Since this data simulates what we anticipate the real data to look like, these results should translate to our analysis on the actual data as well.

Section 5: Results

First, we examine the results of logistic regression fitted on the training data. Even though we are primarily concerned about the problem of inference, we do this as a sanity check to make sure that our classification method is yielding intuitive results. Table 1 displays the coefficients on the predictors in the fitted model. Since we only add one additional observation when fitting the model for conformal prediction, we do not expect the coefficients to deviate very far from this for most observations. The coefficient on koi_period is positive and statistically significant, indicating that objects with longer planetary transit times are more likely to be confirmed exoplanets holding the other variables constant. The coefficient on koi_time0bk is negative and significant, meaning that objects discovered more recently are less likely to be confirmed. This makes sense, since better technology over time may lead us to discover objects that were undetectable earlier, and these are probably less likely to be exoplanets since they may just be random objects that we could not see before.

Next, the coefficients on koi_impact and koi_duration are both positive, meaning that the objects that are further away from their stars and have longer transit times are more likely to be confirmed exoplanets, holding the other variables constant. The coefficient on koi_depth is negative, which means that objects blocking out more radiation from the star are more likely to be exoplanets. This is also intuitive because objects that block out a higher fraction of radiation are likely bigger and made of denser materials. We are already controlling for the object's size via its planetary radius koi_prad, which has a large, positive coefficient as expected. Also, we are already controlling for the star's size via its radius koi_srad, which has a significant, positive coefficient. This also makes sense because larger stars are more likely to be orbited by

exoplanets. Fixing the object's size and its star's size, the amount of radiation blocked may depend on the material the object is made of. Exoplanets should intuitively consist of materials that block a lot of radiation, which would explain why koi_depth has a positive coefficient.

Finally, we see that koi_teq, koi_steff, and koi_slogg all have significant, positive coefficients. The positive coefficients on koi_teq and koi_steff indicate that hotter objects and objects near hotter stars are more likely to be exoplanets. The positive coefficient on koi_slogg indicates that after controlling for the other variables, the log acceleration due to gravity at the surface of the star (given by $G\frac{M}{r^2}$ where G is a universal constant, M is the star's mass, and r is the star's radius) is associated with a higher probability that the object is confirmed as an exoplanet.

Having checked that the results of logistic regression are sensible, we can move to our results on inference. We found that 16.8 percent of our prediction sets for the test data only contain "false positive", while the remaining 83.2 percent contain both. This is very different from the results of our simulation, but this makes sense because the simulated data was created for logistic regression, whereas the real dataset is likely much messier, making classification with logistic regression more difficult. As a result, we expect to see a much larger fraction of these prediction sets contain both values since logistic regression should be less accurate, requiring "wider intervals" in the form of larger prediction sets to guarantee coverage. Since the real dataset was also much more likely to contain false positives than confirmed objects, it makes sense that the only prediction sets containing a single element contain "false positive", as these are the classifications logistic regression is most confident in.

Section 6: Discussion, Critiques, and Limitations

The most obvious limitation with this method is the large number of prediction sets that contain both possible values. The fact that 83.2% of these prediction sets contain "confirmed" and "false positive" already guarantees at least an 83.2% coverage rate. However, as discussed earlier, this is not exactly helpful because it does not help us narrow what the object could be. A significant reason for this could be that our logistic regression classifier does not work well on this data. With a better classifier, this method could yield both good coverage and more precise prediction sets. We already have evidence of this with our semisynthetic data. In that dataset, the fictitious Y truly depended on the X variables in a logistic manner, and we got both high coverage and many prediction sets with just one element.

Section 6a: Ignoring Variables

Outside of the classifier, there are several potential problems with our analysis regarding inference. First, we ignored the error estimates for each variable. We did this because it did not make sense to incorporate the errors into PCA, since these uncertainties are linked to our original predictors in a way that is undetectable through PCA. However, this may be problematic if (1)

objects whose measurements are made with more uncertainty also tend to be more/less likely to be confirmed and (2) the objects in our test set have more uncertainty than objects in our training set. We can test both of these potential problems. First, by subtracting the lower bound from the upper bound, we found the width of the confidence interval for each variable. Next, we computed the correlation between this width and the indicator for whether or not the object was confirmed in the training set. By permuting the width across observations 15,000 times, we found how significant this correlation was based on a permutation test. We found that 9 out of 10 confidence interval widths were significantly correlated with whether or not the object was confirmed at the $\alpha = 0.05$ level. To check if the distribution of the confidence intervals were different across the training and testing data, we computed the K-S statistic and used a permutation test, randomly assigning points to the training and test sets 15,000 times. We found that all 10 of the intervals had significantly different distributions at the $\alpha = 0.05$ level.

This indicates that we may still have a covariate shift issue in our data. Since the uncertainty in the measurements are correlated with the outcome, they may help predict the outcome; furthermore, since they have different distributions across the training and test sets, we would have to adjust for this difference when conducting inference after incorporating this uncertainty into our model. However, as we mentioned earlier, we may have to adjust for this difference using a different method because the PCA-based method we used for dimensionality reduction has no indication that each of the uncertainty variables is linked to a specific other variable. This is a challenging issue that merits further exploration.

On the topic of excluding predictors, we chose not to include ra, dec, and koi_insol even though we had data on these variables. We excluded koi_insol because it is just a different measurement of the object's temperature, which is already given by koi_teq. As a result, it offers little new information and just increases the dimensionality of our data. We excluded ra and dec because these variables simply represent where the objects are in the sky. This could be problematic if (1) some areas of the sky are more likely to contain exoplanets and (2) the objects in the test set are more likely to come from some areas of the sky than others. We believe that (2) is likely to hold, as our permutation tests using K-S statistics show that the distribution of ra is different between the two datasets at the $\alpha = 0.05$ level while dec is different at the $\alpha = 0.1$ level. Of course, it is the joint distribution of ra and dec that determines each object's position in the sky. However, since the marginal distributions are very different, their joint distribution is likely to be very different as well. Nonetheless, this is not a problem because (1) is unlikely to hold. There is no theoretical reason why exoplanets should be more likely to form in some areas of the sky compared to others.

However, if some areas of the sky are more likely to have exoplanets than the others, this could be a major issue that is hard to resolve with our current methodology. First, the way we delineate these areas numerically using ra and dec should not affect the odds of an object being a confirmed exoplanet in a linear manner as assumed by logistic regression. As a result, the best way to account for these variables would be to use indicator variables to bin certain parts of the sky. However, it would not make sense to include these indicators in our PCA or k-means

algorithms along with the other variables, which are not discrete. Instead, we would use each bin of the sky and separate the observations in each bin according to the k-means clusters based on the other variables. Depending on the number of bins, this could result in a high dimensionality. Regardless, the biggest issue is that it is unclear exactly how we would bin the sky in a non-data dependent manner, since we have no theoretical basis for dividing ra and dec in the first place.

Section 6b: Critiques of the Method

Another potentially problematic choice we made was to use a uniform ranking to rescale all of our predictors. By using rankings, we lose information since we only make use of the ordinal rank of each variable across observations, ignoring how far these observations are from each other. This compresses our data and could cause us to miss certain patterns later in our analysis. For a toy example, suppose there is just one predictor and 8 observations. If the observations took the values 1, 2, 3, 4, 1001, 1002, 1003, and 1004, it would make sense to use two clusters. However, our method would rescale these to be $\frac{1}{8}, \frac{1}{4}, \frac{3}{8}, \ldots, 1$. Here, it is no longer obvious that there are two main clusters in the predictor space. Although this is certainly an extreme example and only involves one dimension, it is intuitive to see how this problem could extend to multiple dimensions as well.

Our response to this issue is that most of the observations for each variable are already close to uniformly distributed except for the fact that some observations have long right tails as we saw earlier. As a result, the main consequence of ranking is just to bring these tails closer to the bulk of the data. If the observations that are in the right tail of one variable are more likely to be in the right tails of other variables, this will be retained by the ranking because the observations will be consistently ranked highly. Such a structure would still be detected by k-means. Thus, we do not expect a major loss of structure to occur because of this rescaling method. Instead, its benefits outweigh its costs because putting all of the predictors on the same scale is helpful for PCA and k-means.

PCA represents another aspect of our methodology that could cause us to miss structure in our data. Even if rescaling our variables through ranking does not remove any important relationships between the predictors, PCA may certainly do this by definition of dimensionality reduction. Ultimately, we are interested in differences in the distribution of the predictors across the training and test sets, and these predictors exist in 10-dimensional space. By using PCA with 5 principal components, we are confining ourselves to look for differences in 5 dimensions that arise from linear combinations of these original 10 variables. This affects our results in two ways. First, fixing the number of clusters, the k-means clusters that are created from 5 principal components may be very different from the k-means clusters created from the 6 principal components. This would affect our weights and potentially change our results. Furthermore, with a different number of principal components, we could have selected a different number of clusters following the criteria outlined in the rest of our methodology. This would also change the weights and affect our results. In other words, the number of principal components chosen

has the potential to have major effects on our results, and the decision to use 5 principal components was somewhat arbitrary.

Given more time, we would have used simulations to check how coverage rates changed with the number of principal components along with the number of k-means clusters using semisynthetic data. One way to do this would be to use the same method of generating our fictitious outcome as explained in section 4b, except with a different number of principal components each time. By varying this and the number of clusters, we could determine which combination works best on the semisynthetic data, which will likely work best on the real data as well.

Figures

Figure 1: Density of Covariates

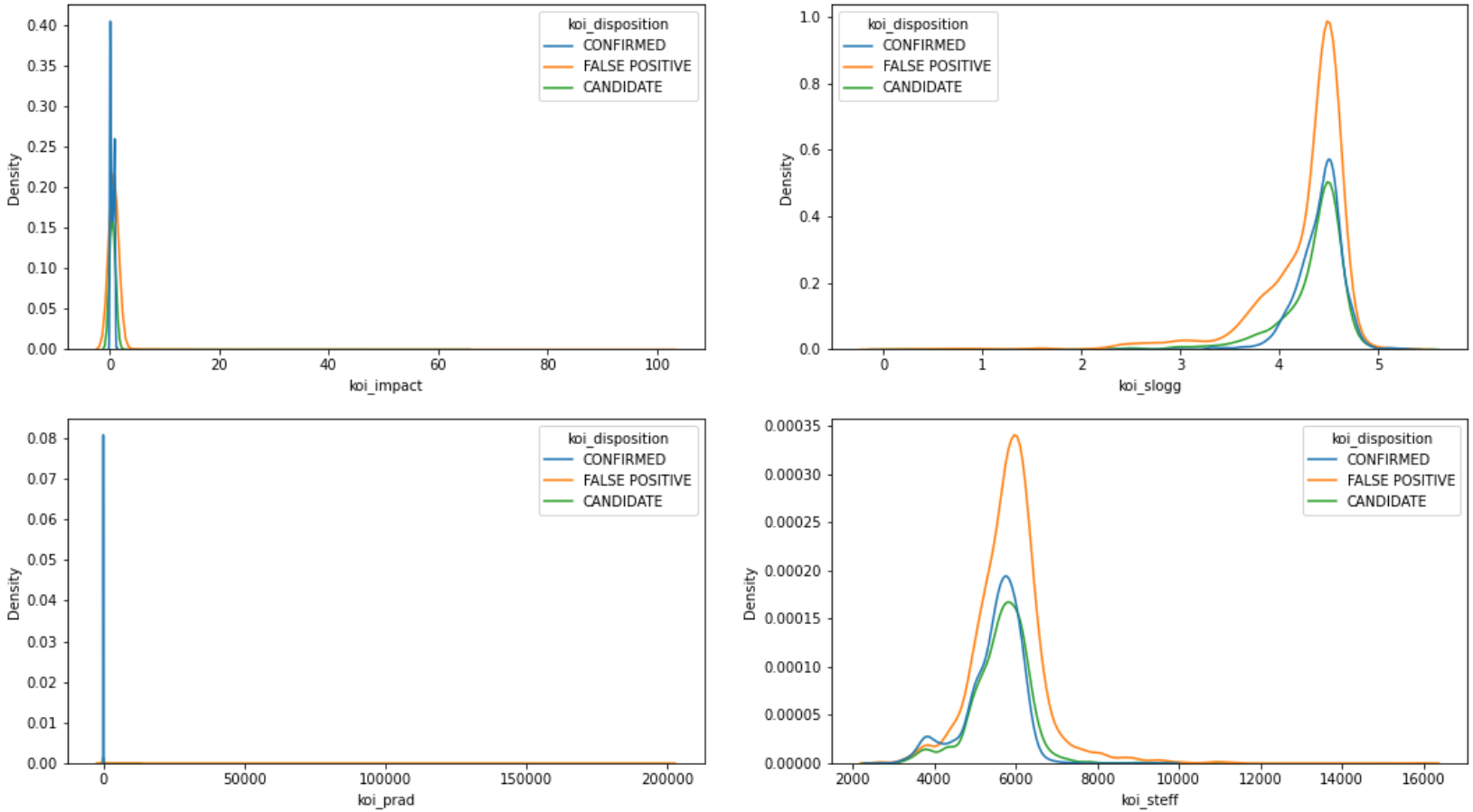


Figure 2: Explained Variance from 5 Principal Components

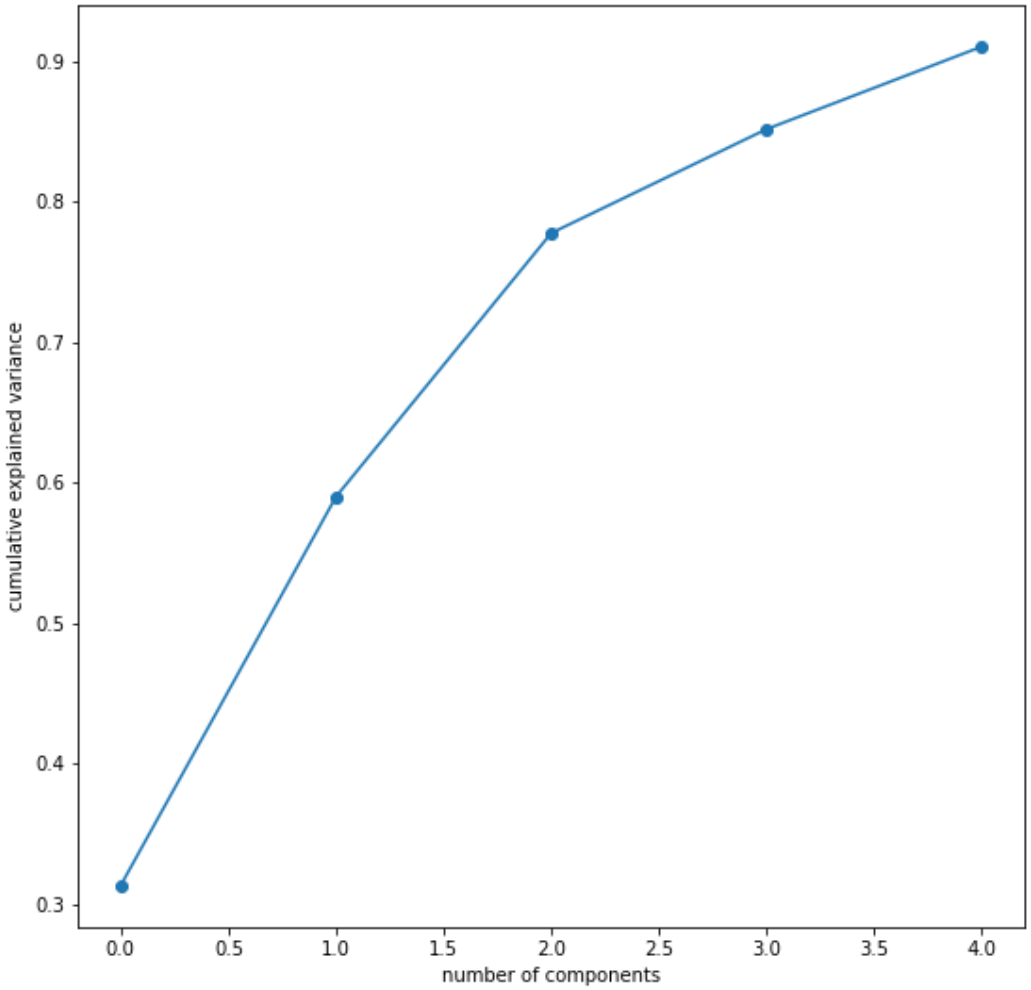


Figure 3: Elbow Method (works best if you squint)

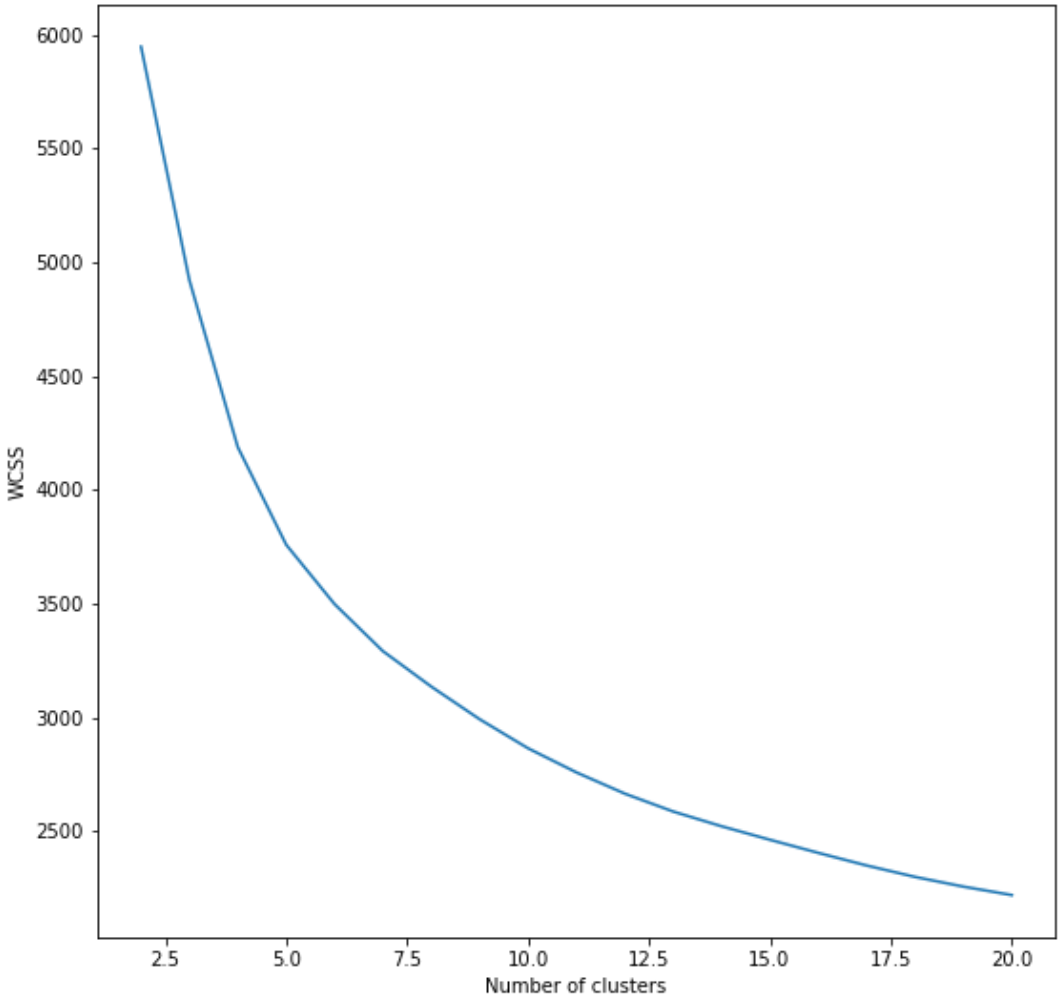


Figure 4: Number of Observations per Cluster

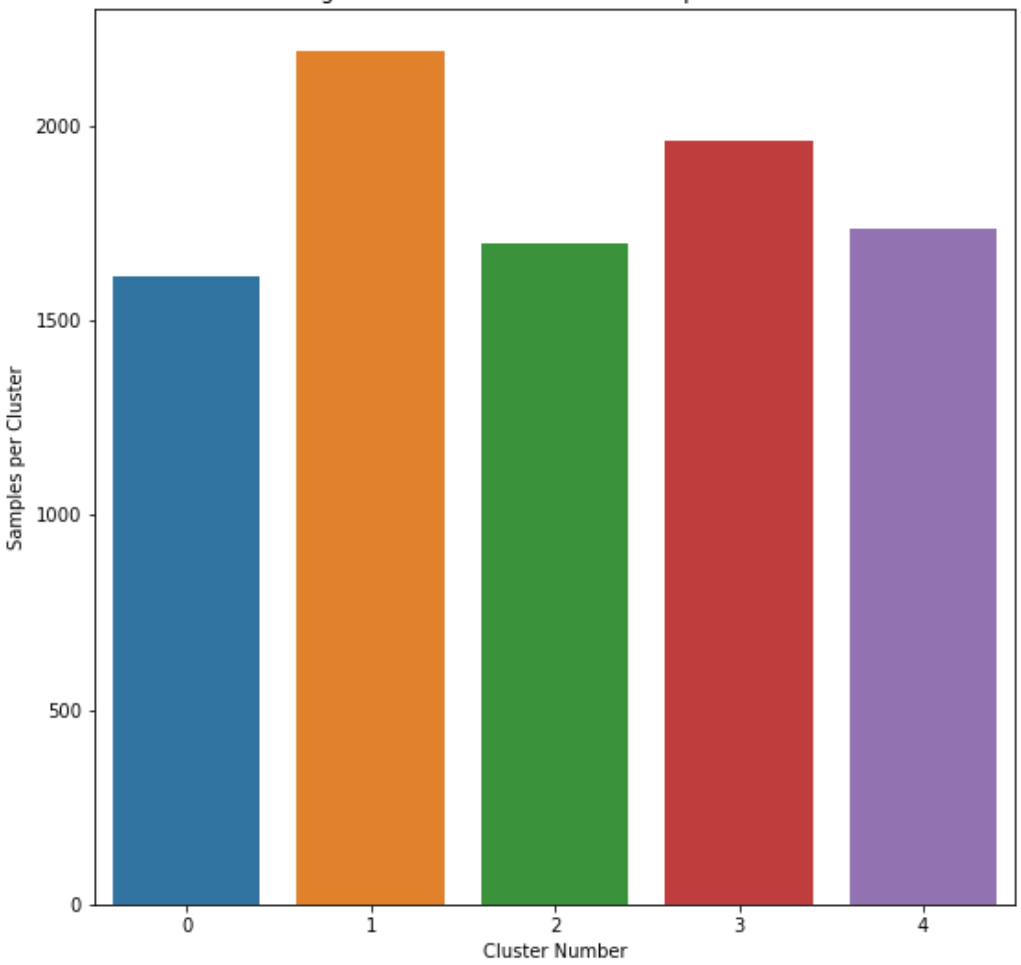


Figure 5: Cluster Proportions in Training Set

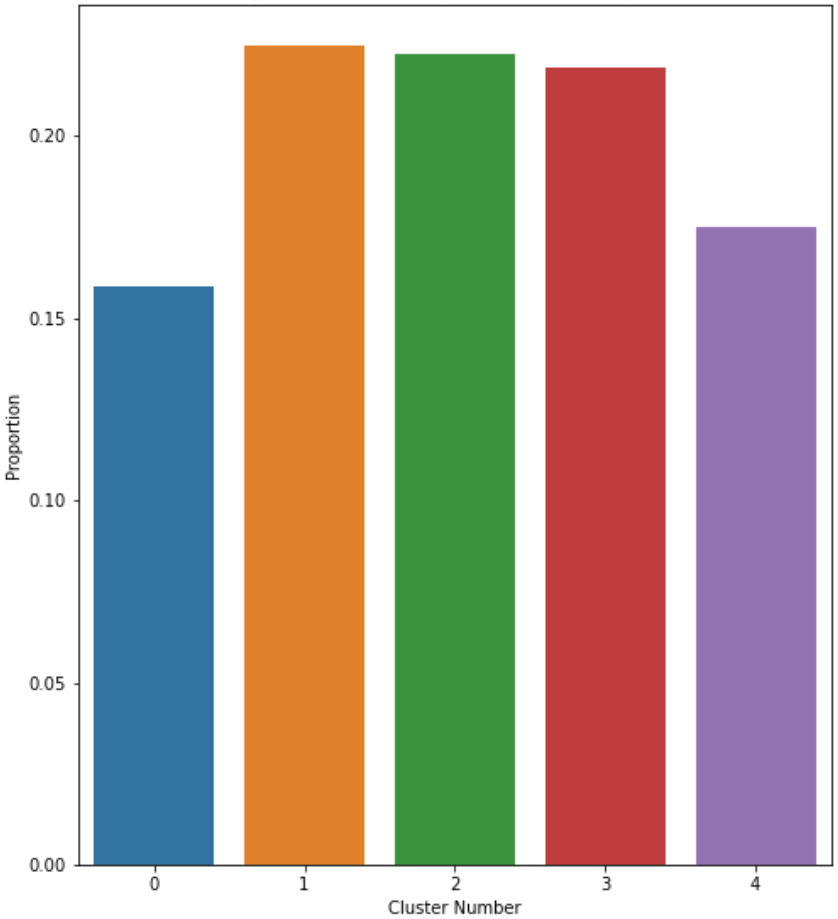


Figure 6: Cluster Proportions in Test Set

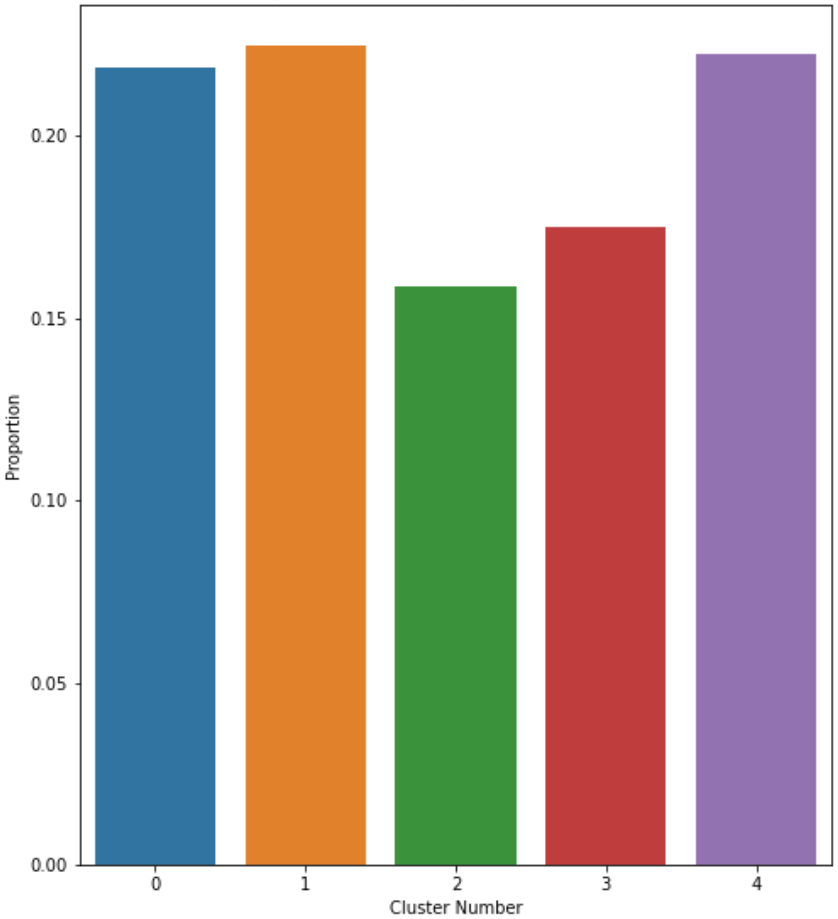


Figure 7: Scatterplots of Covariates

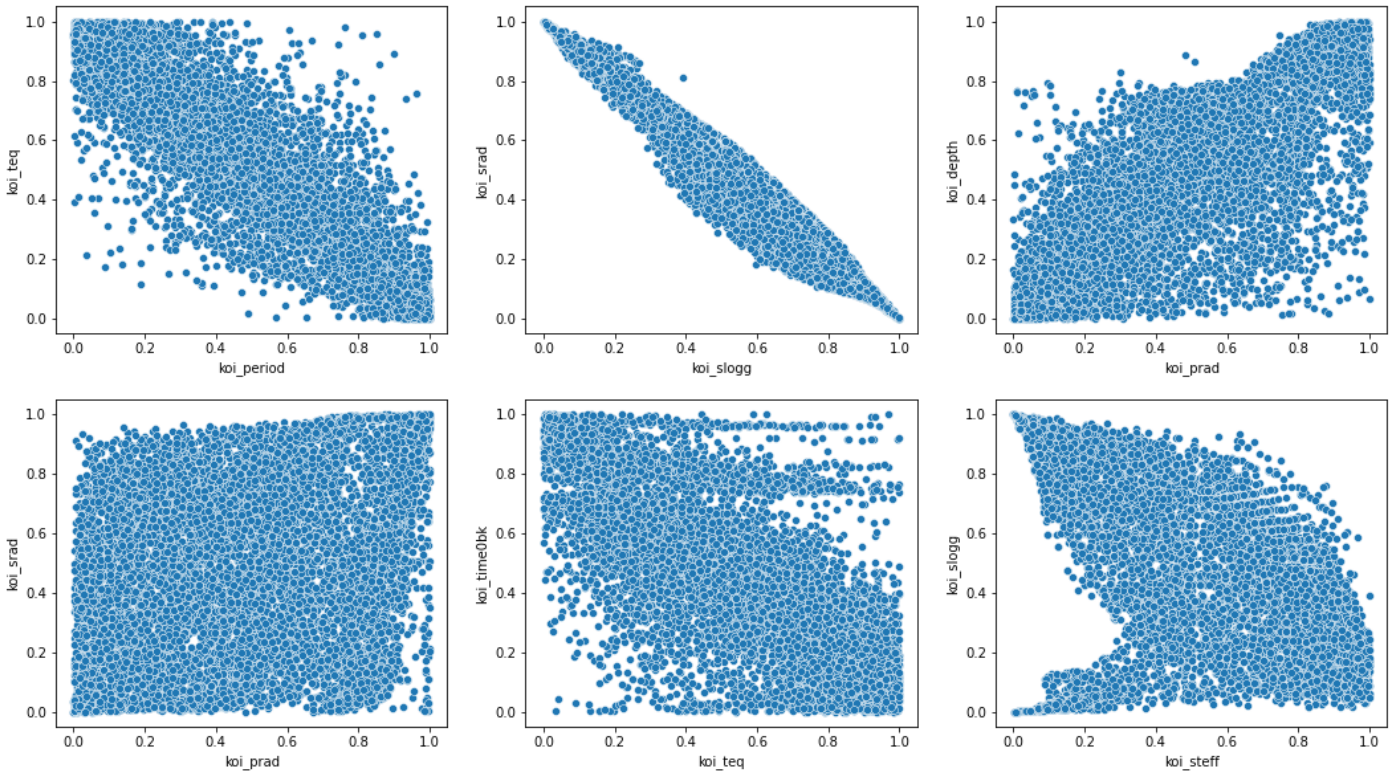


Figure 8: Distribution of Principal Components for Train and Testing Sets

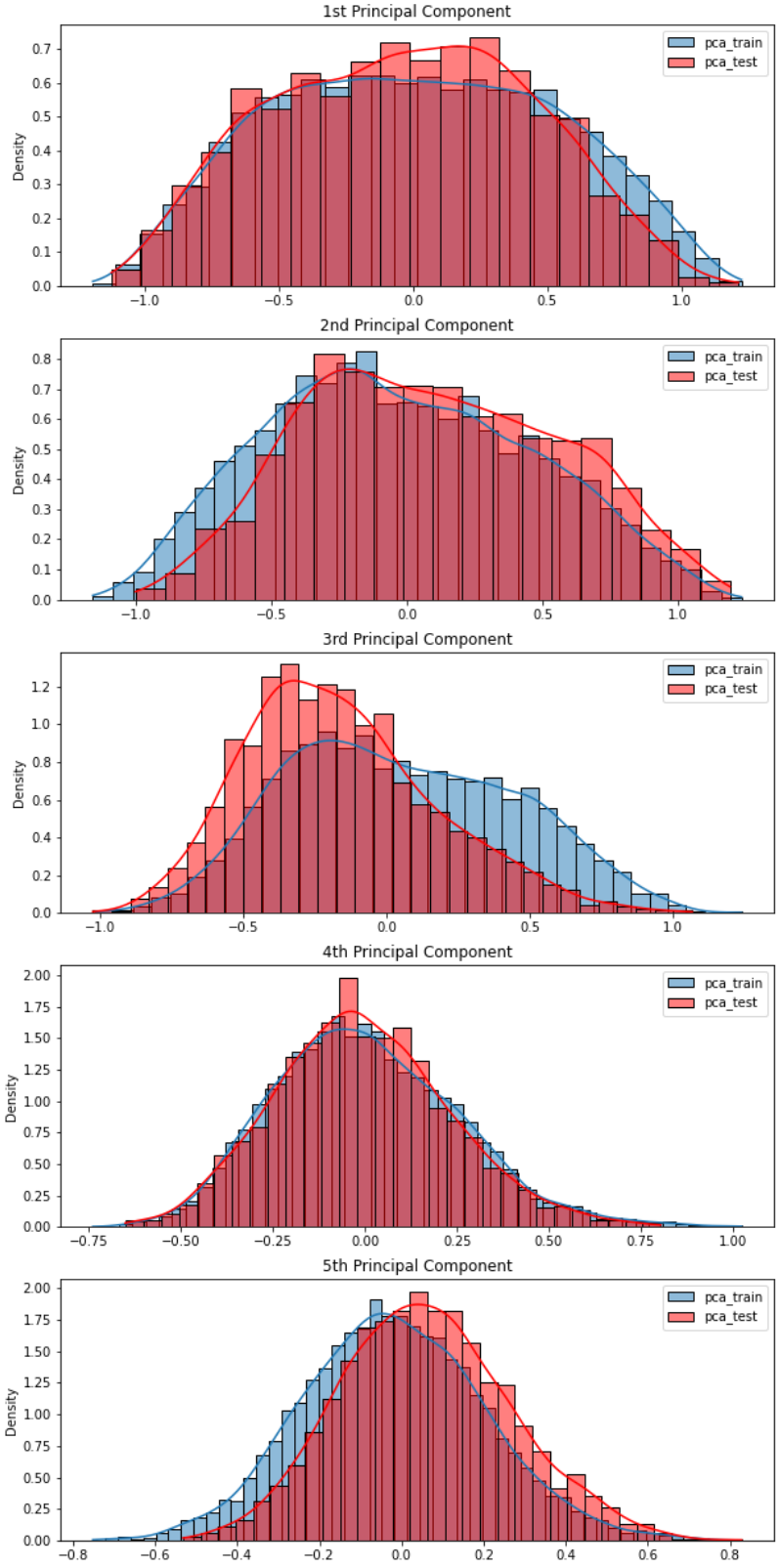


Figure 9: Coverage Rate Over Different Numbers of Clusters

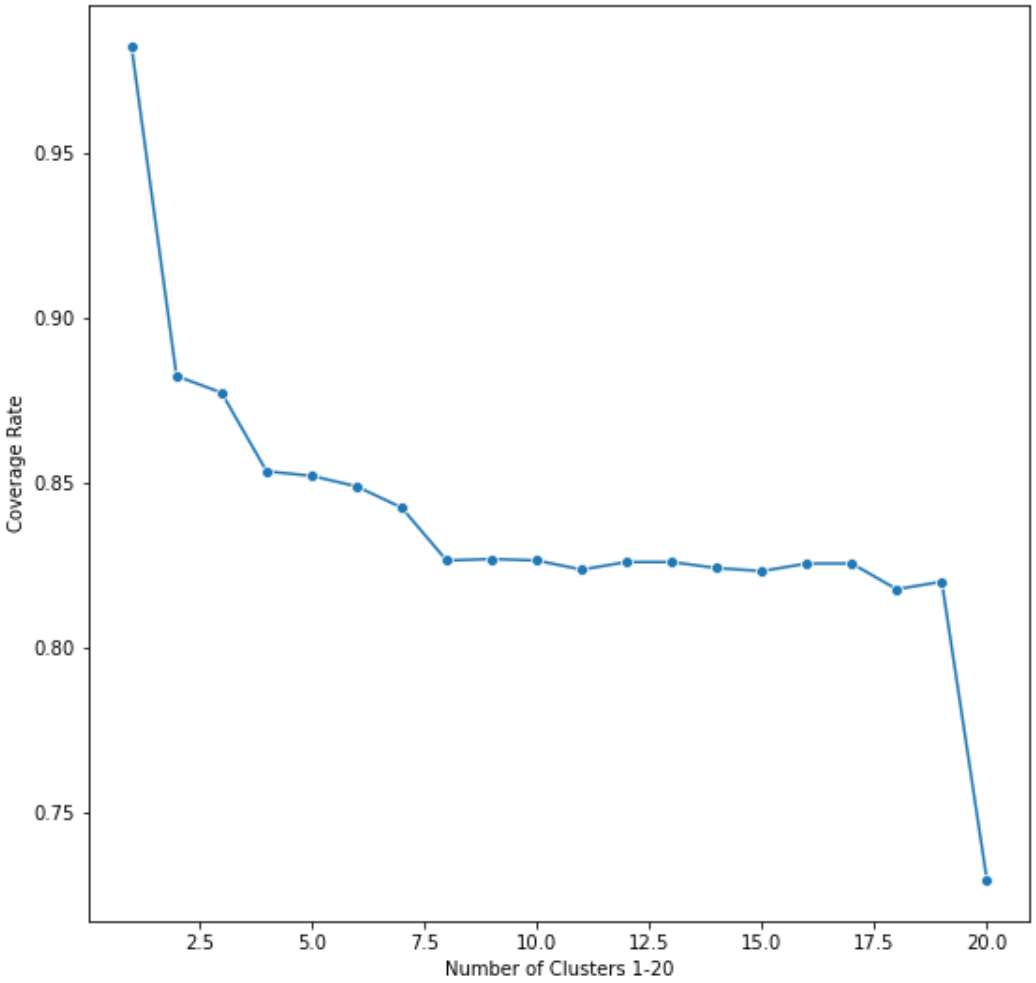
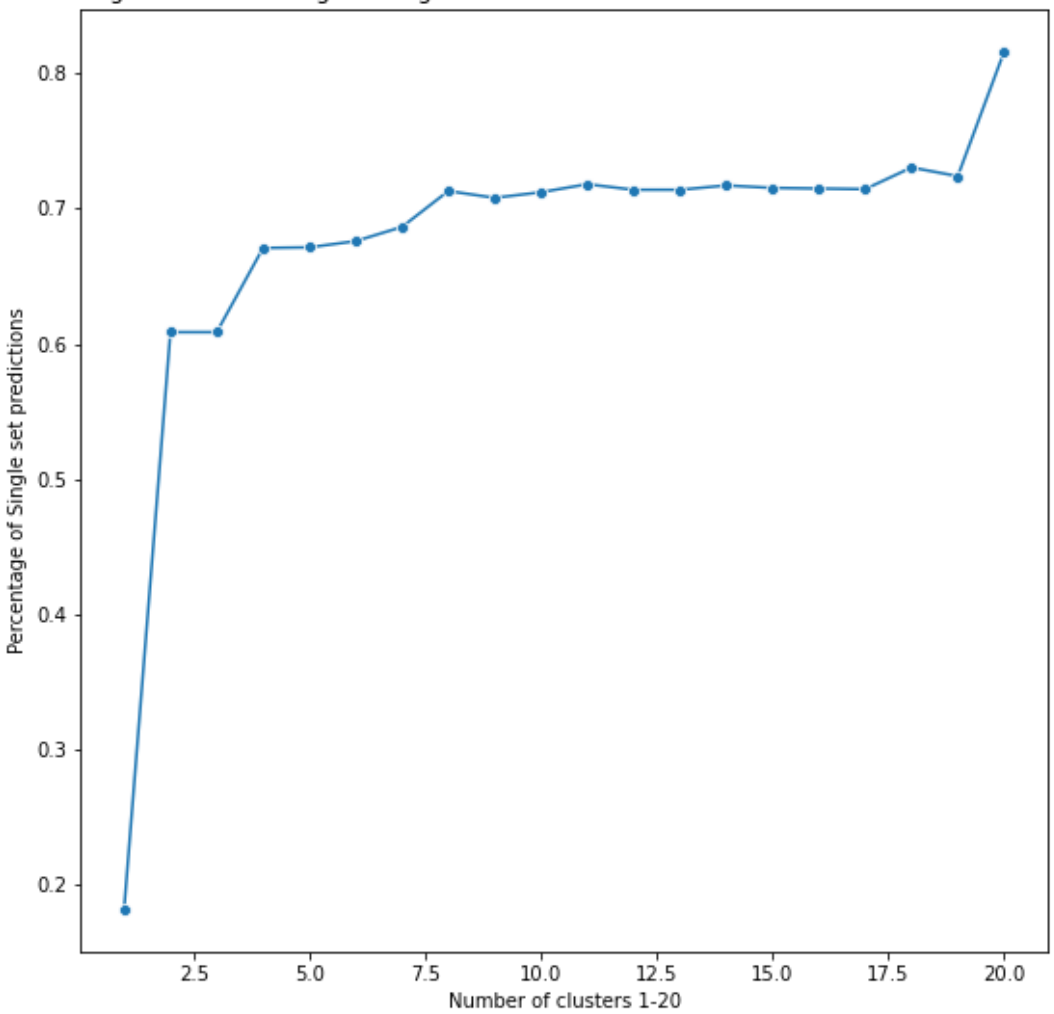


Figure 10: Percentage of Single Set Predictions Over Different Number of Clusters



Tables

Table 1: Logistic Regression Results

Dep. Variable:	y	No. Observations:	7016
Model:	Logit	Df Residuals:	7005
Method:	MLE	Df Model:	10
Date:	Wed, 16 Mar 2022	Pseudo R-squ.:	inf
Time:	10:56:28	Log-Likelihood:	-4.6476e+05
converged:	True	LL-Null:	0.0000
Covariance Type:	nonrobust	LLR p-value:	1.000

	coef	std err	z	P>|z|	[0.025	0.975]
const	-26.7236	2.145	-12.459	0.000	-30.927	-22.520
koi_period	0.0167	0.001	19.836	0.000	0.015	0.018
koi_time0bk	-0.0028	0.001	-3.029	0.002	-0.005	-0.001
koi_impact	0.7487	0.111	6.761	0.000	0.532	0.966
koi_duration	0.0980	0.010	9.365	0.000	0.077	0.118
koi_depth	2.561e-05	6.85e-06	3.741	0.000	1.22e-05	3.9e-05
koi_prad	0.0896	0.008	10.679	0.000	0.073	0.106
koi_teq	0.0028	0.000	27.782	0.000	0.003	0.003
koi_steff	0.0002	6.27e-05	3.107	0.002	7.19e-05	0.000
koi_slogg	4.6442	0.410	11.322	0.000	3.840	5.448
koi_srad	1.1350	0.157	7.221	0.000	0.827	1.443

Project 2 Code

March 17, 2022

0.1 Basic imports

[ ]:
```
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
import seaborn as sns
from math import log2
from tqdm import tqdm
from scipy.stats import kstest
from sklearn.preprocessing import StandardScaler
from sklearn.decomposition import PCA
plt.rcParams["figure.figsize"] = (9,9)
pd.set_option("display.max_columns", None)
```

[ ]:
```
# This only keeps non-error columns
df = pd.read_csv('KOI.csv')
predictor_list = []
error_list = []
for i in df.columns:
    if '2' in i or '1' in i:
        error_list.append(i)
    else:
        predictor_list.append(i)
predictor_list = predictor_list[1:]
```

Error Confidence Permutation Test

[ ]:
```
# This is formatting for doing permutation test.
error_df = df[error_list]
error_df['koi_disposition'] = df.koi_disposition
error_df.drop(['koi_teq_err1','koi_teq_err2' ],axis=1,inplace=True)
error_df.dropna(inplace=True)
new_names = 'koi_period koi_time0bk koi_impact koi_duration koi_depth koi_prad␣
↪koi_insol koi_steff koi_slogg koi_srad'.split(' ')

error_df = pd.concat((error_df,pd.Series(error_df.iloc[:,0] - error_df.iloc[:
↪,1],name=new_names[0]+'_error')),axis=1,)
```

```
error_df = pd.concat((error_df,pd.Series(error_df.iloc[:,2] - error_df.iloc[:
↪,3],name=new_names[1]+'_error')),axis=1,)
error_df = pd.concat((error_df,pd.Series(error_df.iloc[:,4] - error_df.iloc[:
↪,5],name=new_names[2]+'_error')),axis=1,)
error_df = pd.concat((error_df,pd.Series(error_df.iloc[:,6] - error_df.iloc[:
↪,7],name=new_names[3]+'_error')),axis=1,)
error_df = pd.concat((error_df,pd.Series(error_df.iloc[:,8] - error_df.iloc[:
↪,9],name=new_names[4]+'_error')),axis=1)
error_df = pd.concat((error_df,pd.Series(error_df.iloc[:,10] - error_df.iloc[:
↪,11],name=new_names[5]+'_error')), axis=1)
error_df = pd.concat((error_df,pd.Series(error_df.iloc[:,12] - error_df.iloc[:
↪,13],name=new_names[6]+'_error')), axis=1)
error_df = pd.concat((error_df,pd.Series(error_df.iloc[:,14] - error_df.iloc[:
↪,15],name=new_names[7]+'_error')), axis=1)
error_df = pd.concat((error_df,pd.Series(error_df.iloc[:,16] - error_df.iloc[:
↪,17],name=new_names[8]+'_error')), axis=1)
error_df = pd.concat((error_df,pd.Series(error_df.iloc[:,18] - error_df.iloc[:
↪,19],name=new_names[9]+'_error')), axis=1)

error_df_test = error_df[error_df.koi_disposition == 'CANDIDATE']
error_df_train = error_df[error_df.koi_disposition != 'CANDIDATE']
def change(x):
    if x == 'CONFIRMED':
        return 1
    else:
        return 0
error_df_train['koi_disposition'] = error_df_train['koi_disposition'].
↪apply(lambda x: change(x))
```

[ ]:
```
def permut_p_value(column_name, num_perm=15000):
    """ Permute Error Columns.
    """
    permutation_statistics = []
    original_statistic = np.abs(np.corrcoef(error_df_train.iloc[:,column_name],␣
↪error_df_train['koi_disposition'])[1][0])

    for i in range(num_perm):
        shuffled_df = np.random.permutation(error_df_train['koi_disposition'])
        perm_statistic = np.corrcoef(error_df_train.iloc[:,column_name],␣
↪shuffled_df)[1][0]
        permutation_statistics.append(perm_statistic)
    permutation_statistics = np.array(permutation_statistics)
    p_val = (1 + sum(np.abs(permutation_statistics) >= original_statistic))/(1␣
↪+ num_perm)
    return np.around(p_val,4)
```

[ ]:
```
# Checks p value for the error columns interval.
p_vals = {}
for col_name in range(21,31):
    p = permut_p_value(col_name)
    p_vals[error_df.columns[col_name]] = p
table = pd.DataFrame(pd.Series(p_vals), columns=['p-value'])
table.index.names = ['Error Variable Name']

table
```

[ ]:
```
def ks_permut_p_value(column_name, num_perm=15000):
    """
    Permute using the KS test statistic.
    """
    permutation_statistics = []
    original_statistic = kstest(error_df_test[column_name],␣
 ↪error_df_train[column_name])[1]
    for i in range(num_perm):
        shuffled_df = np.random.permutation(error_df[column_name])
        test_permutation = shuffled_df[:2163]
        train_permutation = shuffled_df[2163:]
        ks_statistic = kstest(test_permutation, train_permutation)[1]
        permutation_statistics.append(ks_statistic)

    permutation_statistics = np.array(permutation_statistics)
    p_val = (1 + sum(permutation_statistics <= original_statistic))/(1 +␣
 ↪num_perm)
    return np.around(p_val,4)
```

[ ]:
```
# Checks the KS p value for the error columns interval.
p_vals = {}
for col_name in tqdm(error_df.columns[21:31]):
    p = ks_permut_p_value(col_name)
    p_vals[col_name] = p
    print(p_vals)w
table = pd.DataFrame(pd.Series(p_vals), columns=['p-value'])
table.index.names = ['Error Variable Name']
table
```

[ ]:
```
df = df[predictor_list] # This is when removing the error columns
df.drop(df.iloc[:,3:7].columns,axis=1,inplace=True) # Removes flag variables
```

[ ]:
```
sns.heatmap(df.isna())
```

Lots of NA values.

[ ]:
```
print(df[df.isnull()['koi_srad']]['koi_disposition'].value_counts());
print()
print(f'The proportion of False Positives: {np.around(299/(299+63),3)}')
```

[ ]:
```
df.koi_disposition.value_counts()
proportion = 5023/(2293 + 5023)
print(f'The propotion of False Positives on training dataset {np.
 ↪around(proportion,3)}')
```

This shows that the proportion of False positives among the NA values is around 14% higher.

[ ]:
```
df.dropna(inplace=True) # solution is to just drop them.
```

[ ]:
```
# Bit of an uneven distribution.
df.koi_disposition.value_counts()
```

High correlation values on (koi_Slogg,koi_Srad), (koi_prad, koi_impact).

[ ]:
```
# Plot density of four covariates (figure 1)

fig, axes = plt.subplots(2, 2, figsize=(18, 10))

sns.kdeplot(df['koi_impact'], hue=df['koi_disposition'], ax=axes[0, 0])

# axes[0, 0].set_title(f'Adjusted p-value: {np.
 ↪around(corrected_pvals[significant_routes[0]],3)}')

sns.kdeplot(df['koi_slogg'], hue=df['koi_disposition'], ax=axes[0, 1])
# axes[0, 1].set_title(f'Adjusted p-value: {np.
 ↪around(corrected_pvals[significant_routes[1]],3)}')

sns.kdeplot(df['koi_prad'], hue=df['koi_disposition'], ax=axes[1, 0])
# axes[0, 2].set_title(f'Adjusted p-value: {np.
 ↪around(corrected_pvals[significant_routes[2]],3)}')

sns.kdeplot(df['koi_steff'], hue=df['koi_disposition'], ax=axes[1, 1])

plt.suptitle('Figure 1: Density of Four Covariates', fontsize=15)
plt.savefig('fig1.png')
```

[ ]:
```
from pprint import PrettyPrinter
from scipy.stats import kstest
pp = PrettyPrinter(indent=4)

df_test = df[df.koi_disposition == 'CANDIDATE']
df_train = df[df.koi_disposition != 'CANDIDATE']
ks_tests = {}
```

```
for col in df.columns:
    dist_test = kstest(df_test[col], df_train[col])[1]
    ks_tests[col] = np.around(dist_test,4)
pp.pprint(ks_tests)
```

1 INFERENCE Full Conformal

[ ]:
```
pd.options.mode.chained_assignment = None  # default='warn'
from sklearn.decomposition import PCA
import scipy.stats as stats
from sklearn.cluster import KMeans
from sklearn.linear_model import LogisticRegression
df = pd.read_csv('/Users/gabrielnicholson/Desktop/KOI.csv')

# This only keeps non-error columns
predictor_list = []
for i in df.columns:
    if '2' in i or '1' in i:
        pass
    else:
        predictor_list.append(i)

predictor_list = predictor_list[1:] # removing kepid
df = df[predictor_list] # This is when removing the error columns
df.drop(df.iloc[:,3:7].columns,axis=1,inplace=True) # Removes flag variables
df.dropna(inplace=True) # solution is to just drop them.
df.drop(['ra', 'dec'], axis=1, inplace=True)
df.drop('koi_insol', axis=1, inplace=True)

df_test = df[df.koi_disposition == 'CANDIDATE']
df_train = df[df.koi_disposition != 'CANDIDATE']
regular_df = df.copy()

X_train = df_train.drop('koi_disposition',axis=1).rank(pct=True)
X_test = df_test.drop('koi_disposition',axis=1).rank(pct=True)
all_X = df.drop('koi_disposition',axis=1).rank(pct=True)

all_X.reset_index(inplace=True, drop=True)  # Indexs get messed up after␣
↪dropping koi_disposition.
X_test.reset_index(inplace=True, drop=True)
df.reset_index(inplace=True, drop=True)

df = pd.concat((df.koi_disposition,all_X),axis=1) # Put koi_disposition back␣
↪after not giving it a rank.
```

[ ]:
```
# Creating PCA and K means using only the ranked X data.
pca = PCA(n_components=5)
pca_transformed = pca.fit_transform(all_X)

kmeans = KMeans(n_clusters=5, init='k-means++', max_iter=500, n_init=30,␣
↪random_state=42)
df['cluster'] = kmeans.fit_predict(pca_transformed)
regular_df['cluster'] = kmeans.fit_predict(pca_transformed)
```

[ ]:
```
regular_df_train = regular_df[regular_df['koi_disposition'] != 'CANDIDATE' ]
regular_df_test = regular_df[regular_df['koi_disposition'] == 'CANDIDATE' ]
df_test = df[df.koi_disposition == 'CANDIDATE']
```

[ ]:
```
# Create scatterplots of covariates against each other (figure 7)

fig, axes = plt.subplots(2, 3, figsize=(18, 10))

sns.scatterplot(ax=axes[0, 0], data=df, x = 'koi_period', y = 'koi_teq')

sns.scatterplot(ax=axes[0, 1], data=df, x = 'koi_slogg', y = 'koi_srad')

sns.scatterplot(ax=axes[0, 2], data=df, x = 'koi_prad', y = 'koi_depth')

sns.scatterplot(ax=axes[1, 0], data=df, x = 'koi_prad', y = 'koi_srad')

sns.scatterplot(ax=axes[1, 1], data=df, x = 'koi_teq', y = 'koi_time0bk')

sns.scatterplot(ax=axes[1, 2], data=df, x = 'koi_steff', y = 'koi_slogg')

plt.suptitle('Figure 7: Scatterplots of Covariates', fontsize=15)

plt.savefig('fig7.png')
```

[ ]:
```
X_test = df_test.drop(['koi_disposition', 'cluster'],axis=1).rank(pct=True)
X_test.reset_index(inplace=True, drop=True)
df_test.reset_index(inplace=True, drop=True)

test_df = pd.concat((X_test, df_test[['koi_disposition', 'cluster']]),axis=1)

# Calculates information needed for clustering
num_train_clusters = df[df.koi_disposition != 'CANDIDATE']['cluster'].
↪value_counts()
num_test_clusters = df[df.koi_disposition == 'CANDIDATE']['cluster'].
↪value_counts()

train_cluster_len = [num_train_clusters[i] for i in␣
↪range(len(num_train_clusters))] # The indexes are in order of cluster.
```

```
test_cluster_len = [num_test_clusters[i] for i in␣
 ↪range(len(num_test_clusters))] # The indexes are in order of cluster.

total_train_clus = np.sum(train_cluster_len)
total_test_clus = np.sum(test_cluster_len)
```

[ ]:
```
from factor_analyzer.factor_analyzer import calculate_bartlett_sphericity
chi_square_value, p_value = calculate_bartlett_sphericity(df.
 ↪drop(['koi_disposition', 'cluster'],axis=1))
print(f'chi_square_value: {chi_square_value}')
print(f'p_value: {p_value}')
```

[ ]:
```
# Plot cluster proportions across training and test sets (figures 5 and 6)

plt_train_clusters = df[df.koi_disposition != 'CANDIDATE']['cluster'].
 ↪value_counts(normalize=True)
plt_test_clusters = df[df.koi_disposition == 'CANDIDATE']['cluster'].
 ↪value_counts(normalize=True)

fig, axs = plt.subplots(ncols=2, figsize=(14,8))
fig = sns.barplot(x = plt_train_clusters.index, y = plt_train_clusters,␣
 ↪ax=axs[0])
fig.set_title('Figure 5: Cluster Proportions in Training Set')
fig.set_ylabel('Proportion')
fig.set_xlabel('Cluster Number')

fig = sns.barplot(x = plt_test_clusters.index, y = plt_train_clusters,␣
 ↪ax=axs[1])
fig.set_title('Figure 6: Cluster Proportions in Test Set')
fig.set_ylabel('Proportion')
fig.set_xlabel('Cluster Number')
plt.tight_layout()
plt.savefig('figure 5 & 6')
```

[ ]:
```
# Plots overall cluster proportions (figure 4)

sns.barplot(x=df.cluster.value_counts(sort=False).index, y=df.cluster.
 ↪value_counts(sort=False))
plt.ylabel('Samples per Cluster')
plt.title('Figure 4: Number of Observations per Cluster')
plt.xlabel('Cluster Number')
plt.savefig('fig4')
```

[ ]:
```
def full_conformal(test_row, cp_df_train, alpha):
    """ Our main algorithm.
    """
```

```
    cp_conformal_df = cp_df_train.append(test_row) # appends the test row to␣
↪the bottom  of training data
    X = cp_conformal_df.drop(['koi_disposition', 'cluster'], axis=1)
    y = cp_conformal_df['koi_disposition']

    # Calculates the denominator.
    test_clust = test_row.cluster
    denominator = 7015.999999999839
    q_test =  test_cluster_len[test_clust] / total_test_clus
    q_train = train_cluster_len[test_clust] / total_train_clus
    denominator += (q_test/q_train)

    # Calculates the weights for each cluster.
    weights = []
    for c_num in cp_conformal_df.cluster:
        q_test =  test_cluster_len[c_num] / total_test_clus
        q_train = train_cluster_len[c_num] / total_train_clus
        weight_i = (q_test / q_train) / (denominator)
        weights.append(weight_i)

    y.iloc[-1] = 'CONFIRMED'        # Put in the guessed value for y.
    scaler = StandardScaler()
    X = scaler.fit_transform(X)
    model_confirmed = LogisticRegression(max_iter=1000, C=100)
    model_confirmed.fit(X, y)

    y.iloc[-1] = 'FALSE POSITIVE'   # Put in the guessed value for y.
    model_false_positive = LogisticRegression(max_iter=1000, C=100)
    model_false_positive.fit(X, y)

    scores_confirmed = 1/(model_confirmed.predict_proba(X)[:,0] + 0.00001)
    scores_fp = 1/(model_false_positive.predict_proba(X)[:,1] + 0.00001)

    # Keeps track of score on test sample.
    TEST_SCORE_confirmed = scores_confirmed[-1]
    TEST_SCORE_FP = scores_fp[-1]

    confirmed_df = pd.DataFrame({'Scores of Confirmed':scores_confirmed,␣
↪'Weights':weights})
    false_positive_df =  pd.DataFrame({'false positive score':scores_fp,␣
↪'Weights':weights})

    confirmed_df.sort_values(by='Scores of Confirmed',inplace=True)
    false_positive_df.sort_values(by='false positive score',inplace=True)

    confirmed_df['weight_cdf'] = np.cumsum(confirmed_df['Weights'])
    false_positive_df['weight_cdf'] = np.cumsum(false_positive_df['Weights'])
```

```
    # Checking if the score is within the 1-alpha weight quantile.
    prediction_set = []
    score_Q_confirmed = confirmed_df[confirmed_df['weight_cdf'] <=␣
↪(1-alpha)]['Scores of Confirmed'].iloc[-1]
    score_Q_false_positive = false_positive_df[false_positive_df['weight_cdf']␣
↪<= (1-alpha)]['false positive score'].iloc[-1]

    if TEST_SCORE_confirmed < score_Q_confirmed:
        prediction_set.append('CONFIRMED')

    if TEST_SCORE_FP < score_Q_false_positive:
        prediction_set.append('FALSE POSITIVE')

    return prediction_set, confirmed_df
```

[ ]:
```
regular_df_test.reset_index(drop=True,inplace=True)
regular_df_train.reset_index(drop=True,inplace=True)
```

[ ]:
```
denominator = 0
for c_num in regular_df_train.cluster:
    q_test =  test_cluster_len[c_num] / total_test_clus
    q_train = train_cluster_len[c_num] / total_train_clus
    denominator += (q_test/q_train)
denominator
```

[ ]:
```
predictions = []
cp_df_train = regular_df_train.copy() #9:31

for row in tqdm(range(len(regular_df_test))):
    prediction_set, confirmed_df = full_conformal(test_row=regular_df_test.
↪iloc[row], cp_df_train=cp_df_train, alpha=0.1)
    predictions.append(prediction_set)
```

[ ]:
```
# Check distribution of elements in each set

both = 0
false_pos_single = 0
confirmed_single = 0
nothing_in_set = 0
for lst in predictions:
    if len(lst) == 2:
        both += 1
    elif len(lst) == 1:
        if lst == 'CONFIRMED':
            confirmed_single += 1
        else:
```

```
            false_pos_single += 1
    else:
        nothing_in_set += 1

print(f'both: {both/len(predictions)}')
print(f'false_pos_single: {false_pos_single/len(predictions)}')
print(f'confirmed_single: {confirmed_single/len(predictions)}')
print(f'nothing_in_set: {nothing_in_set/len(predictions)}')
```

1.0.1 Simulating Data

[ ]:
```
from numpy import dot
from numpy.linalg import norm

pca_training = pca_transformed[np.where(df.koi_disposition != 'CANDIDATE')]
pca_test = pca_transformed[np.where(df.koi_disposition == 'CANDIDATE')]

pca_transformed # Find the indicies for training and test
```

[ ]:
```
# Computing Z according to the procedure

Z_list = []
p_bar = np.mean(pca_test,axis=0) - np.mean(pca_training,axis=0)

for i in range(len(pca_transformed)):
    p_i = pca_transformed[i]
    probability_i = ((dot(p_i, p_bar)/(norm(p_i)*norm(p_bar))) + 1) / 8
    Z_list.append(probability_i)
Z_list = np.random.binomial(n=1, p=Z_list)

def sigmoid(x):
  return 1 / (1 + np.exp(-x))
```

[ ]:
```
# Simulating data

all_xs = regular_df.drop(['koi_disposition','cluster'],axis=1)
scaler = StandardScaler()
all_xs = scaler.fit_transform(all_xs)
simulated_y = {} # Using dictionary to keep track of index for concattinating␣
 ↪with main dataframe.
for num, Z in enumerate(Z_list):
    if Z == 0:
        y_i = np.random.binomial(n=1,p=sigmoid(np.sum(all_xs[num])))
        simulated_y[num] = y_i
    else:
        y_i = np.random.binomial(n=1,p=0.5)
```

```
        simulated_y[num] = y_i

regular_df.reset_index(drop=True,inplace=True)
sim_df = pd.concat((pd.Series(simulated_y,␣
↪name='simulated_y'),regular_df),axis=1)
```

1.0.2 Running Simulations

[ ]:
```
def full_conformal(test_row, cp_df_train, alpha):
    cp_conformal_df = cp_df_train.append(test_row) # appends the test row to␣
↪the bottom  of training data
    X = cp_conformal_df.drop(['koi_disposition', 'cluster', 'simulated_y'],␣
↪axis=1)
    y = cp_conformal_df['simulated_y']

    # Calculates the denominator.
    test_clust = test_row.cluster
    denominator = 7015.999999999839
    q_test =  test_cluster_len[test_clust] / total_test_clus
    q_train = train_cluster_len[test_clust] / total_train_clus
    denominator += (q_test/q_train)

    # Calculates the weights for each cluster.
    weights = []
    for c_num in cp_conformal_df.cluster:
        q_test =  test_cluster_len[c_num] / total_test_clus
        q_train = train_cluster_len[c_num] / total_train_clus
        weight_i = (q_test / q_train) / (denominator)
        weights.append(weight_i)

    # weights = np.ones(len(cp_conformal_df.cluster)) * 1/len(cp_conformal_df.
↪cluster)

    y.iloc[-1] = 0     # Put in the guessed value for y.
    scaler = StandardScaler()
    X = scaler.fit_transform(X)
    model_confirmed = LogisticRegression(max_iter=1000, C=100)
    model_confirmed.fit(X, y)

    y.iloc[-1] = 1   # Put in the guessed value for y.
    model_false_positive = LogisticRegression(max_iter=1000, C=100)
    model_false_positive.fit(X, y)

    scores_confirmed = 1/(model_confirmed.predict_proba(X)[:,0] + 0.00001)
    scores_fp = 1/(model_false_positive.predict_proba(X)[:,1] + 0.00001)
```

```
    # Keeps track of score on test sample.
    TEST_SCORE_confirmed = scores_confirmed[-1]
    TEST_SCORE_FP = scores_fp[-1]

    confirmed_df = pd.DataFrame({'Scores of Confirmed':scores_confirmed,␣
↪'Weights':weights})
    false_positive_df =  pd.DataFrame({'false positive score':scores_fp,␣
↪'Weights':weights})

    confirmed_df.sort_values(by='Scores of Confirmed',inplace=True)
    false_positive_df.sort_values(by='false positive score',inplace=True)

    confirmed_df['weight_cdf'] = np.cumsum(confirmed_df['Weights'])
    false_positive_df['weight_cdf'] = np.cumsum(false_positive_df['Weights'])

    # Checking if the score is within the 1-alpha weight quantile.
    prediction_set = []
    score_Q_confirmed = confirmed_df[confirmed_df['weight_cdf'] <=␣
↪(1-alpha)]['Scores of Confirmed'].iloc[-1]
    score_Q_false_positive = false_positive_df[false_positive_df['weight_cdf']␣
↪<= (1-alpha)]['false positive score'].iloc[-1]

    if TEST_SCORE_confirmed < score_Q_confirmed:
        prediction_set.append('CONFIRMED')

    if TEST_SCORE_FP < score_Q_false_positive:
        prediction_set.append('FALSE POSITIVE')

    return prediction_set, confirmed_df
```

[ ]:
```
pca = PCA(n_components=5)
pca_transformed = pca.fit_transform(all_X)

kmeans = KMeans(n_clusters=20, init='k-means++', max_iter=500, n_init=30,␣
 ↪random_state=42)
df['cluster'] = kmeans.fit_predict(pca_transformed)
regular_df['cluster'] = kmeans.fit_predict(pca_transformed)

regular_df.reset_index(drop=True,inplace=True)
sim_df = pd.concat((pd.Series(simulated_y,␣
 ↪name='simulated_y'),regular_df),axis=1)

simulated_df_test = sim_df[sim_df.koi_disposition == 'CANDIDATE']
cp_df_train = sim_df.copy()
```

```
num_train_clusters = df[df.koi_disposition != 'CANDIDATE']['cluster'].
 ↪value_counts()
num_test_clusters = df[df.koi_disposition == 'CANDIDATE']['cluster'].
 ↪value_counts()

train_cluster_len = [num_train_clusters[i] for i in␣
 ↪range(len(num_train_clusters))] # The indexes are in order of cluster.
test_cluster_len = [num_test_clusters[i] for i in␣
 ↪range(len(num_test_clusters))] # The indexes are in order of cluster.

total_train_clus = np.sum(train_cluster_len)
total_test_clus = np.sum(test_cluster_len)

predictions = []
for row in tqdm(range(len(simulated_df_test))):
    prediction_set, confirmed_df = full_conformal(test_row=simulated_df_test.
 ↪iloc[row], cp_df_train=cp_df_train, alpha=0.1)
    predictions.append(prediction_set)
```

[ ]:
```
final_clust = len([i for i in predictions if len(i)<=1])/len(predictions) #␣
 ↪zero clusters
final_clust_cov = coverage/len(predictions)
```

[ ]:
```
zero_clust = len([i for i in predictions if len(i)<=1])/len(predictions) # zero␣
 ↪clusters
zero_clust_cov = coverage/len(predictions)
```

[ ]:
```
plot_coverage = []
plot_power = []
total_predictions = []

for i in range(2,21):
    pca = PCA(n_components=5)
    pca_transformed = pca.fit_transform(all_X)

    kmeans = KMeans(n_clusters=i, init='k-means++', max_iter=500, n_init=30,␣
 ↪random_state=42)
    df['cluster'] = kmeans.fit_predict(pca_transformed)
    regular_df['cluster'] = kmeans.fit_predict(pca_transformed)

    regular_df.reset_index(drop=True,inplace=True)
    sim_df = pd.concat((pd.Series(simulated_y,␣
 ↪name='simulated_y'),regular_df),axis=1)

    simulated_df_test = sim_df[sim_df.koi_disposition == 'CANDIDATE']
    cp_df_train = sim_df.copy()
```

```
    num_train_clusters = df[df.koi_disposition != 'CANDIDATE']['cluster'].
↪value_counts()
    num_test_clusters = df[df.koi_disposition == 'CANDIDATE']['cluster'].
↪value_counts()

    train_cluster_len = [num_train_clusters[i] for i in␣
↪range(len(num_train_clusters))] # The indexes are in order of cluster.
    test_cluster_len = [num_test_clusters[i] for i in␣
↪range(len(num_test_clusters))] # The indexes are in order of cluster.

    total_train_clus = np.sum(train_cluster_len)
    total_test_clus = np.sum(test_cluster_len)

    predictions = []
    for row in tqdm(range(len(simulated_df_test))):
        prediction_set, confirmed_df =␣
↪full_conformal(test_row=simulated_df_test.iloc[row],␣
↪cp_df_train=cp_df_train, alpha=0.1)
        predictions.append(prediction_set)

    def kk(x):
        if x == 0:
            return 'CONFIRMED'
        else:
            return 'FALSE POSITIVE'
    check = simulated_df_test['simulated_y'].apply(lambda x: kk(x))

    coverage = 0

    total_predictions.append(predictions)
    plot_coverage.append(coverage/len(check))
    plot_power.append(len([i for i in predictions if len(i)<=1])/
↪len(predictions))
```

[ ]:
```
# Plotting coverage rate against number of clusters (figure 9)
plot_coverage.append(final_clust)
plot_power.append(final_clust_cov)
plot_coverage.insert(0, zero_clust_cov)
sns.lineplot(x=np.arange(1,21), y=plot_coverage, marker='o')
plt.title('Figure 9: Coverage Rate Over Different Numbers of Clusters')
plt.xlabel('Number of Clusters 1-20')
plt.ylabel('Coverage Rate')
plt.savefig('fig9')
```

[ ]:
```
# Plotting proportion of single set predictions against number of clusters␣
↪(figure 10)
plot_power.insert(0, zero_clust)
sns.lineplot(x=np.arange(1,21), y=plot_power, marker='o')
plt.title('Figure 10: Percentage of Single Set Predictions Over Different␣
↪Number of Clusters')
plt.ylabel('Percentage of Single set predictions')
plt.xlabel('Number of clusters 1-20')
plt.savefig('fig10')
```

1.0.3 Verifying K-Means and PCA

[ ]:
```
all = all_X.copy()
```

[ ]:
```
# Or fit on all the data.
pca = PCA(n_components=5)
all = pca.fit_transform(all)
pca.explained_variance_ratio_.sum() # 5 components can explain 91% variance.
```

[ ]:
```
pca_train = all[np.where(df.koi_disposition != 'CANDIDATE')]
pca_test = all[np.where(df.koi_disposition == 'CANDIDATE')]
```

[ ]:
```
# Plotting distributions of principal components (figure 8)
fig, axs = plt.subplots(nrows=5, figsize=(10,20))

sns.histplot(pca_train[:,0], label='pca_train', ax=axs[0], kde=True,␣
↪stat='density')
sns.histplot(pca_test[:,0], color='r', label='pca_test', ax=axs[0],␣
↪kde=True,stat='density')
axs[0].set_title('1st Principal Component')
sns.histplot(pca_train[:,1], label='pca_train', ax=axs[1],␣
↪kde=True,stat='density')
sns.histplot(pca_test[:,1], color='r', label='pca_test', kde=True,ax=axs[1],␣
↪stat='density')
axs[1].set_title('2nd Principal Component')
sns.histplot(pca_train[:,2], label='pca_train', ax=axs[2],␣
↪kde=True,stat='density')
sns.histplot(pca_test[:,2], color='r', label='pca_test', kde=True,ax=axs[2],␣
↪stat='density')
axs[2].set_title('3rd Principal Component')
sns.histplot(pca_train[:,3], label='pca_train', ax=axs[3],␣
↪kde=True,stat='density')
sns.histplot(pca_test[:,3], color='r', label='pca_test', ax=axs[3],␣
↪kde=True,stat='density')
axs[3].set_title('4th Principal Component')
```

```
sns.histplot(pca_train[:,4], label='pca_train', ax=axs[4], kde=True,␣
 ↪stat='density')
sns.histplot(pca_test[:,4], color='r', label='pca_test', ax=axs[4], kde=True,␣
 ↪stat='density')
axs[4].set_title('5th Principal Component')
axs[0].legend()
axs[1].legend()
axs[2].legend()
axs[3].legend()
axs[4].legend()

fig.subplots_adjust(top=0.95)
plt.suptitle('Figure 8: Distribution of Principal Components for Train and␣
 ↪Testing Sets', fontsize=15)
plt.savefig('fig8')
```

[ ]:
```
# Plotting WCSS against number of clusters (figure 3)

max_clusters = 20

wcss = []
for i in range(2, max_clusters + 1):
    kmeans = KMeans(n_clusters=i, init='k-means++', max_iter=500, n_init=25,␣
 ↪random_state=42)
    kmeans.fit(all_X)
    wcss.append(kmeans.inertia_)

plt.plot(range(2, max_clusters + 1), wcss)
plt.title('Figure 3: Elbow Method (works best if you squint)')
plt.xlabel('Number of clusters')
plt.ylabel('WCSS')
plt.savefig('figure 3')
```

[ ]:
```
# Plotting variance explained by number of components (figure 2)
plt.plot(np.cumsum(pca.explained_variance_ratio_), marker='o')
plt.xlabel('number of components')
plt.ylabel('cumulative explained variance');
plt.title('Figure 2: Explained Variance from 5 Principal Components')
plt.savefig('fig2')
```

1.0.4 Logit Regression

[ ]:
```
from sklearn.linear_model import LogisticRegression
from statsmodels.tools.tools import add_constant
from statsmodels.discrete.discrete_model import Logit
```

[ ]:
```
X = regular_df_train.drop(['koi_disposition', 'cluster'], axis=1)
y = pd.get_dummies(regular_df_train['koi_disposition'],drop_first=True).
 ↪to_numpy()
X = add_constant(X)
model = Logit(y,X).fit()
```

[ ]:
```
# Finding logistic regression results on training data (table 1)
results_summary = model.summary()
print(results_summary)
```

2 Permutation test for distribution check

Permutation tests

[ ]:
```
from scipy.stats import ttest_ind
from scipy.stats import median_test
```

[ ]:
```
t_test_p_vals = {}
for col in df_train.columns[1:]:
    p_val = ttest_ind(df_test[col], df_train[col],permutations=20000,␣
 ↪equal_var=False)[1]
    t_test_p_vals[col] = p_val
```

[ ]:
```
pp.pprint(t_test_p_vals) # Equal variance = False
```

[ ]:
```
pp.pprint(t_test_p_vals) # Equal variance assumption = True
```

[ ]:
```
def permut_p_value(column_name, num_perm=15000, variance=False):
    """
    """
    permutation_statistics = []
    if variance:
        original_statistic = np.var(df_test[column_name])
        for i in range(num_perm):
            permutation = np.random.permutation(df[column_name])[:2185]
            variance_of_permutation = np.var(permutation)
            permutation_statistics.append(variance_of_permutation)
    else:
        stat, p, med, tbl = median_test(df_test[column_name],␣
 ↪df_train[column_name])
        original_statistic = np.abs(stat)
        for i in range(num_perm):
            permutation_set = np.random.permutation(df[column_name])
            test_set = permutation_set[:2185]
            train_set = permutation_set[2185:]
            stat,perm_stat,_,_ = median_test(test_set, train_set)
            permutation_statistics.append(stat)
```

```
    permutation_statistics = np.array(permutation_statistics)

    p_val = 1 + sum(np.abs(permutation_statistics) >= original_statistic)/
↪(num_perm - 1) # Is there a one minus in denominator?
    return p_val
```

[ ]:
```
def permut_p_value(column_name, num_perm=15000):
    """
    Permute using the KS test statistic.
    """
    permutation_statistics = []
    original_statistic = kstest(df_test[column_name], df_train[column_name])[1]
    for i in range(num_perm):
        shuffled_df = np.random.permutation(df[column_name])
        test_permutation = shuffled_df[:2185]
        train_permutation = shuffled_df[2185:]
        ks_statistic = kstest(test_permutation, train_permutation)[1]
        permutation_statistics.append(ks_statistic)
    permutation_statistics = np.array(permutation_statistics)

    p_val = (1 + sum(permutation_statistics <= original_statistic))/(num_perm)
    return np.around(p_val,4)
```

[ ]:
```
p_values_permut = {}
for col in tqdm(df.columns[1:]):
    p_val = permut_p_value(col)
    p_values_permut[col] = p_val
```