Look! It's a Bird! It's a Plane! It's an Exoplanet!

Eva Chow and Benjamin Earnest

Shiley-Marcos School of Engineering, University of San Diego

August 15, 2022

It's an Exoplanet!

Abstract

With new data collected from the Kepler telescope constantly updating, the goal of this project is to construct a model that would take characteristics observed by the Kepler telescope and categorize newly observed space objects as being a planet candidate or not. While identifying exoplanets may seem unrelated to matters concerning Earth, what we can gain from exoplanet identification is knowledge. This knowledge can translate over to our solar system and help us identify and possibly update our perceptions of planetary processes, which may include better understanding planetary formation and the criteria for what constitutes a planet. Using a variety of linear models (i.e. Perceptron, Logistic Regression, Support Vector Machine, and Stochastic Gradient Descent) as well as nonlinear models (i.e. K-Nearest Neighbors, Decision Trees, and Random Forest with the aid of Principal Component Analysis), we found that Support Vector Machine was the most accurate model in classifying exoplanets.

It's an Exoplanet!

Problem Statement

The Kepler telescope collected data from space for 9.6 years. It observed over 530,506 stars, and confirmed the existence of 2,662 exoplanets to date. The Kepler mission generated 638 GB of scientific data, which has led to some key scientific findings. This includes the discovery that planets outnumber stars, small planets are common, planets are very diverse in size and make up, and solar systems are also very diverse (NASA, 2022). The amount of scientific data collected provides tremendous opportunities for data scientists to explore and build predictive models. With more data provided by the Kepler mission as of late, our objective is to use this data to classify whether a space object is a possible candidate for being an exoplanet. Taking into account various aspects of the object such as size and gravity, we will aim to categorize each object as an exoplanet candidate, or not an exoplanet candidate. This categorization will take place by predicting a binary label for each observation (candidate or false positive), and separately predict a "koi_score". This score can also be used to categorize each observation as a candidate or false positive.

While identifying exoplanets may seem unrelated to anything concerning Earth, what we can gain from exoplanet identification is knowledge. This knowledge can then translate over to our solar system and help us identify and possibly change our perceptions of planetary processes, such as better understanding how a planet is formed and what criteria may constitute what is a planet.

The dataset is not completely untouched. Our data comes from observations from the Kepler mission as well as from previous observations made from NASA. These observations have been identified by looking for dips in the brightness of stars when a planet travels in front of it. Information regarding this planet is then inferred based on how deep the planet's travel depth is and how bright the star it traveled in front of is. This is how planet size is calculated.

Exploratory Data Analysis

It's an Exoplanet!

This effort is a secondary data analysis. Our dataset is a combination of a previous Kaggle dataset with updates from the Kepler mission, as provided by Cal Tech Labs (Kaggle, 2020). Features of the dataset include various observation identifiers (i.e. id and names if applicable) as well as variables that indicate characteristics of the potential exoplanet (i.e. radius, temperature, and orbital period). This dataset was updated one month ago and is a collection of observations from NASA and Caltech. There are a total of 49 variables and 9,564 observations. Our target variable is binary, and identifies whether an observation is a "candidate" or a "false positive".

Our first step in exploring the data set was to check for any missing values that may require removal, imputation, or other actions to ensure they don't inappropriately influence or interfere with model development. We found several missing values. Table 1 shows the missing values by variable.

Table 1. Missing values by variable.

Missing Values

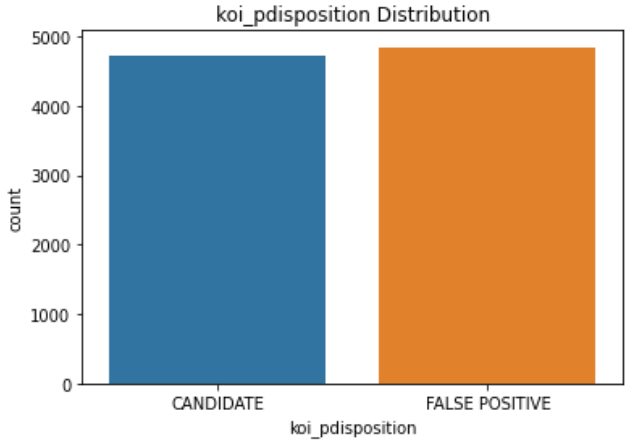
Variable	Missing Data	Variable	Missing Data	Variable	Missing Data	Variable	Missing Data	Variable	Missing Data
koi_disposition	0	**koi_time0bk_err1**	454	**koi_prad**	363	**koi_tce_delivname**	346	**dec**	0
koi_pdisposition	0	**koi_time0bk_err2**	454	**koi_prad_err1**	363	**koi_steff**	363	**koi_kepmag**	1
koi_score	1510	**koi_impact**	363	**koi_prad_err2**	363	**koi_steff_err1**	468		
koi_fpflag_nt	0	**koi_impact_err1**	454	**koi_teq**	363	**koi_steff_err2**	483		
koi_fpflag_ss	0	**koi_impact_err2**	454	**koi_teq_err1**	9564	**koi_slogg**	363		
koi_fpflag_co	0	**koi_duration**	0	**koi_teq_err2**	9564	**koi_slogg_err1**	468		
koi_fpflag_ec	0	**koi_duration_err1**	454	**koi_insol**	321	**koi_slogg_err2**	468		
koi_period	0	**koi_duration_err2**	454	**koi_insol_err1**	321	**koi_srad**	363		
koi_period_err1	454	**koi_depth**	363	**koi_insol_err2**	321	**koi_srad_err1**	468		
koi_period_err2	454	**koi_depth_err1**	454	**koi_model_snr**	363	**koi_srad_err2**	468		
koi_time0bk	0	**koi_depth_err2**	454	**koi_tce_plnt_num**	346	**ra**	0		

Missing values will be dealt with during data preprocessing. As a note, some of the variables, like "koi_teq_err2", do not have any data recorded for any of the observations, and will be removed from the data set. Others will require imputation.

The target variable, koi_pdisposition, is close to evenly distributed. There are 4,717 candidate observations, and 4,847 false positive observations. Figure 1 shows a graphical representation of the target variable, which confirms our observation about the distribution.

It's an Exoplanet!

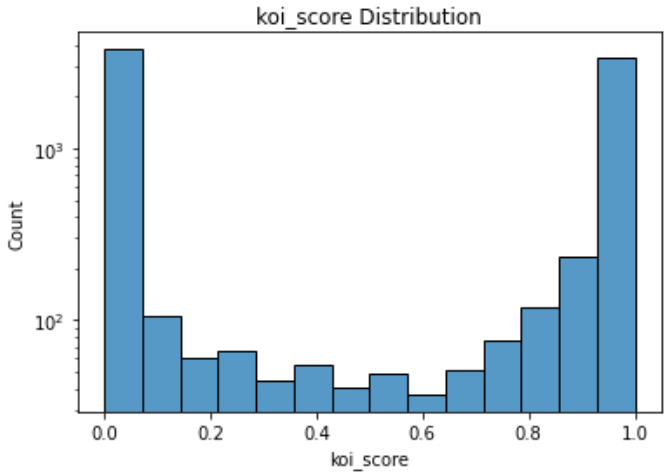
Figure 1 Target variable distribution



This suggests we also have a balanced data set.

Now we want to look at how the koi_score variable is distributed, both for our target variable, and for each value of the target variable. This will help us understand the relationship between koi_score and koi_pdisposition. Figure 2 shows the distribution for all of the koi_scores, scaled to support a better visual understanding of the distribution.

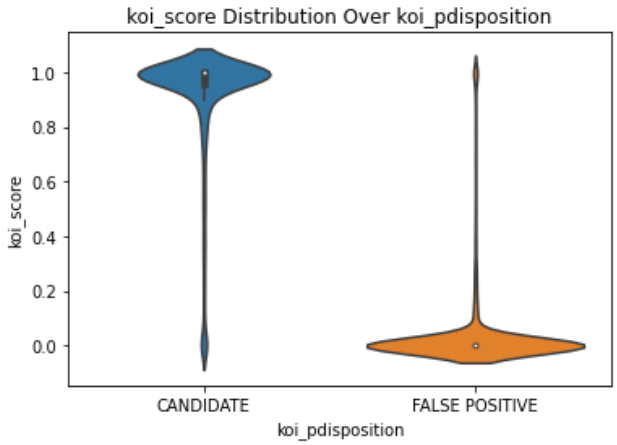
Figure 2 Koi Score Distribution



Next, we want to understand how the koi_score is distributed for the target outcomes. Figure 2 shows this for the "Candidate" and "False Positive" outcomes using violin plots.

It's an Exoplanet!

Figure 3 Koi_score distribution for target outcomes



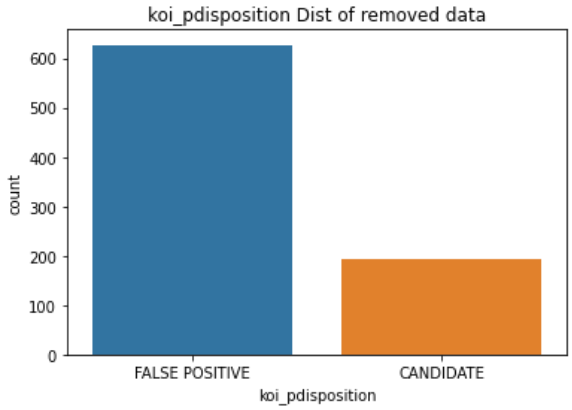
It's clear that koi_score close to 1 classifies observations as candidates, while close to zero is a false positive. The koi_score is something derived from the observed data, used to predict the existence of an exoplanet. We are trying to develop a model that can be trained to predict exoplanet status (candidate vs. false positive) on unseen data, which won't have a koi_score. This value will be removed from the training data set for models predicting koi_pdisposition, and could be used as a target variable for models predicting koi_score.

Data Pre-Processing

Considering our options to deal with the missing data, we first wanted to remove the columns that were completely devoid of input (all NaN). These variables would obviously not provide any value. This included "koi_teq_err1" and "koi_teq_err2". We then wanted to know how many observations included missing data. There are 820 out of the total 9,564, or about 8.5%. Looking at the labels associated with the observations with missing data, we found they heavily favored false positive results, as seen in figure 4.

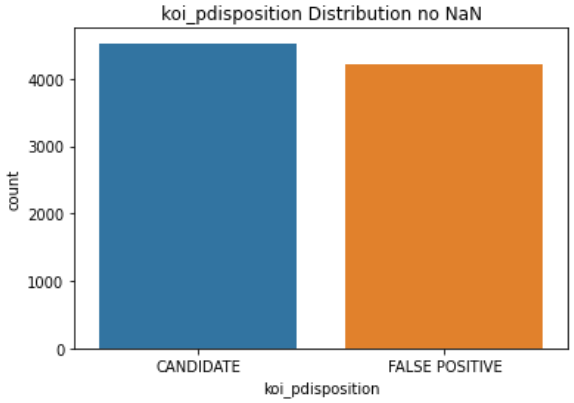
It's an Exoplanet!

Figure 4. Observations with NaN values.



We took this to suggest the missing values were more likely to result in false positives, or that they're biased toward being a false positive. We now removed these observations from the data set, and checked to see if there was still relative balance in the outcomes, which was confirmed in figure 5.

Figure 5. Label distribution after removing missing data.



The remaining preprocessing consisted of centering, scaling, removing near zero variance predictors, and/or removing highly correlated values. This was done on a case by case basis, depending on the type of model being fit. For example, logistic regression requires the near zero variance predictors to be removed, the data to be centered and scaled, and highly correlated predictors be removed. Other models, like decision trees, or support vector machines, don't require the same preprocessing.

Data Splitting

It's an Exoplanet!

The data was split using the "train_test_split" method from the sklearn library in python. We first separated the predictor variables from the target (koi_pdisposition), and assigned them to the "X" variable. We then assigned the target to y, as the variable we're trying to predict. We chose an 80/20 split for training and test, and ended up with 6,995 training observations, and 1,749 test observations. The test observations will be set aside to support model testing and validation. We were also sure to conduct preprocessing like centering, scaling, etc until after the data was split, to avoid data leakage.

Modeling Strategies

Since our target variable is categorical and binary, this is a classification problem. Were we to focus on predicting the koi_score, it may be better suited for regression. Since it's a classification problem, we'll focus on linear and nonlinear models that will predict the koi_pdisposition for each observation. Each model will first be fit on the training data (after preprocessing, where necessary), and then validated using the test data set. For the linear models, we will consider perceptron, logistic regression (LR), support vector machine (SVM), and stochastic gradient descent (SGD). Each model will first be fit using the default hyperparameters in sklearn, then five fold cross validation will be used along with the "best_params" method in sklearn. Another model will then be fit using the best hyperparameters.

For the nonlinear models, hyperparameter tuning for each model was done in order to select for optimal accuracy performance. For K-Nearest Neighbors (KNN), there were three distance metrics that were used for modeling. These distances were Euclidean, Manhattan, and Cosine. For each of these distance metrics, an optimal k value was determined based on test accuracy and fit accordingly. Our KNN models with all three distances were optimized with k=7.

For Decision Trees, a variety of tuning parameters were tested to achieve the most optimal Decision Tree model. The first Decision Tree that was modeled used only default parameters for modeling. The second Decision Tree modeled was optimized over max_depth, which would limit the

number of layers that the Decision Tree would split towards. The max_depth selected was six. The final Decision Tree Model was tuned using Cost Complexity Pruning. This involves using an alpha value to determine how much of the tree would be pruned, with a greater alpha value indicating more pruning. For this model, a ccp_alpha value of 0.000135 was selected.

Our final nonlinear model involved using Principal Component Analysis (PCA) on a Random Forest model. For this model, our predictor variables were normalized using StandardScaler prior to usage within the model.

Validation and Testing

For the linear models, after fitting the model to the best parameters, we will output both the confusion matrix and accuracy score of the best parameter model, so we can compare the performance of each model to the others. The accuracy provides the overall proportion of correct predictions in the model, or the true positives and true negatives, divided by the total number of predictions. This works well for us, as accuracy is generally a good measure when the labels are almost balanced, and correct predictions are equally important (Bali et al, 2018). As shown in the exploratory analysis section, we meet this criteria for use of accuracy.

For the nonlinear models, after fitting the model with the optimal parameters, a confusion matrix was generated in order to evaluate the model predictions. From these predictions, an accuracy score was generated, calculating the total true positives and true negatives predicted over the total number of predictions made. This was evaluated against the target variables of the test data.

Results and Final Model Selection

In total, eleven models were evaluated. This consisted of four linear models, which included Perceptron, Logistic Regression, Support Vector Machine, and Stochastic Gradient Descent. The first of the linear models evaluated was Perceptron. While an initial model with default parameters yielded an accuracy of 68.55%, a model with optimal hyperparameters was able to be evaluated with improved

accuracy. This improved Perceptron model used max_iter=5 when evaluated between various max_iter between 1 and 1000. The final tuned Perceptron model thus resulted in an improved test accuracy score of 71.76%.

The second linear model evaluated was Logistic Regression. For this model, predictors with zero variance and highly correlated predictors were removed. The remaining predictors were then centered and scaled for modeling. While a Logistic Regression model was initially modeled using default parameters, yielding a test accuracy of 81.70%, tuning the model with an optimal C=1000 only made minor accuracy improvements. The final tuned Logistic Regression Model resulted in a test accuracy of 81.76%.

Stochastic Gradient Descent was initially modeled using default parameters, yielding an accuracy score of 81.65%. However, after tuning the model with loss=hinge, alpha=0.001, max_iter=100, and learning_rate=optimal, the final Stochastic Gradient Descent model minorly improved with a test accuracy of 83.30%.

Of the linear models evaluated, the best performing linear model was determined to be Support Vector Machine (SVM) based off of an accuracy score on test data of 85.08%. While SVM with no parameter tuning was not as impressive, finding optimal hyperparameters allowed for SVM to gain more accuracy, most noticeably in more accurately predicting true exoplanet candidates. These optimal parameters were C=1000, gamma=0.001, and kernel=rbf.

The three core nonlinear models evaluated were K-Nearest Neighbors, Decision Trees, and Random Forest utilizing Principal Component Analysis. The first nonlinear model that was evaluated was K-Nearest neighbors. This model was evaluated using three different distance metrics: Euclidean, Manhattan, and Cosine distance. All three distance metrics were evaluated using a range of k values between one and fifteen. Ultimately, all three distance metrics were then paired with k=7. Using this optimized k value, a KNN model using Euclidean distance obtained a test accuracy of 78.16%, Cosine

distance obtained a test accuracy of 78.62%, and Manhattan distance obtained the best test accuracy score amongst the KNN models with an accuracy of 79.53%.

The next nonlinear model evaluated was decision trees. The first decision tree that was modeled used default parameters with no hyperparameter tuning. This resulted in a test accuracy of 80.10%. A decision tree using cost complexity pruning, which "trims" the decision tree down further the greater the cpp_alpha value is, was then modeled with an optimized cpp_alpha value of 0.000135. This resulted in a test accuracy of 80.27%. The final and best performing decision tree model was tuned using max_depth. For this model, a range of max_depth was evaluated from one to fifteen. Ultimately, a max_depth=6 was used to evaluate the model. This resulted in the best performing decision tree model, with a test accuracy of 83.13%.

Of the nonlinear models, Random Forest utilizing Principal Component Analysis was the best performing model based off of an accuracy score on test data of 84.62%. For this model, predictor variables were normalized. After normalization, PCA was performed to fit and transform predictor variables for modeling. Random Forest was decided as the last nonlinear model on which PCA would be performed and evaluated, resulting in the best performing nonlinear model.

Discussion and Conclusions

Of the models evaluated, the best performance was obtained from the Support Vector Machine linear model, with a resulting test accuracy of 85.08%. This high level of accuracy in determining the exoplanet candidacy of an object observed by the Kepler telescope can greatly increase our understanding of objects outside of our solar system. By taking into account the object size and gravity, we can observe dips in star brightness when an object passes in front of it and more accurately determine from this information the exoplanet status of that object. By accurately determining exoplanet status, we may update our understanding of planets and concepts involving planetary formation.

References

Kaggle. (2022). NASA Exoplanet Data Set. https://www.kaggle.com/datasets/arashnic/exoplanets

National Aeronautics and Space Association. (2022, 15 August). Kepler's legacy: discoveries and more. https://exoplanets.nasa.gov/keplerscience/

Bali, R., Sarkar, D., & Sharma, T. (2018). Practical Machine Learning with Python. A Problem-Solver's Guide to Building Real-World Intelligent Systems. Apress.

Look! It's a Bird! It's a Plane! It's an Exoplanet!

Machine Learning to Determine Exoplanet Classification

Data

Source:

NASA exoplanet data set from kaggle. Updated as of June 2022. https://www.kaggle.com/datasets/arashnic/exoplanets (https://www.kaggle.com/datasets/arashnic/exoplanets)

Goal:

Build model to predict if an observation is a planet

EDA

In [ ]:
```
# load necessary packages
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
import seaborn as sns
from seaborn import scatterplot

from sklearn.preprocessing import StandardScaler, Normalizer, LabelEncoder
from sklearn.impute import SimpleImputer
from sklearn.linear_model import Perceptron, LogisticRegression
from sklearn.model_selection import train_test_split, cross_val_score
from sklearn.metrics import confusion_matrix, accuracy_score, roc_curve, ro
c_auc_score, classification_report
from sklearn.feature_selection import VarianceThreshold
from sklearn.svm import SVC
from sklearn.model_selection import GridSearchCV
from sklearn.linear_model import SGDClassifier
from sklearn.neighbors import KNeighborsClassifier
from sklearn import tree
from sklearn.decomposition import PCA
from sklearn.ensemble import RandomForestClassifier
```

In [2]:
```
from google.colab import drive
drive.mount('/content/drive')
```

```
Drive already mounted at /content/drive; to attempt to forcibly remount, ca
ll drive.mount("/content/drive", force_remount=True).
```

In [ ]:
```
# load data
kp = pd.read_csv('/content/drive/MyDrive/ADS-504/Project/exoplanets.csv')
kp.shape
```

Out[ ]:
```
(9564, 49)
```

In [ ]:
```
kp.head()
```

Out[ ]:

	kepid	kepoi_name	kepler_name	koi_disposition	koi_pdisposition	koi_score	koi_fpflag
0	10797460	K00752.01	Kepler-227 b	CONFIRMED	CANDIDATE	1.000	
1	10797460	K00752.02	Kepler-227 c	CONFIRMED	CANDIDATE	0.969	
2	10811496	K00753.01	NaN	CANDIDATE	CANDIDATE	0.000	
3	10848459	K00754.01	NaN	FALSE POSITIVE	FALSE POSITIVE	0.000	
4	10854555	K00755.01	Kepler-664 b	CONFIRMED	CANDIDATE	1.000	

5 rows × 49 columns

We're looking to predict (target variable) disposition, so our first check is to see what the disposition count looks like.

Target Variable Determination

In [ ]:
```
kp['koi_disposition'].value_counts()
```

Out[ ]:
```
FALSE POSITIVE    4840
CANDIDATE         2367
CONFIRMED         2357
Name: koi_disposition, dtype: int64
```

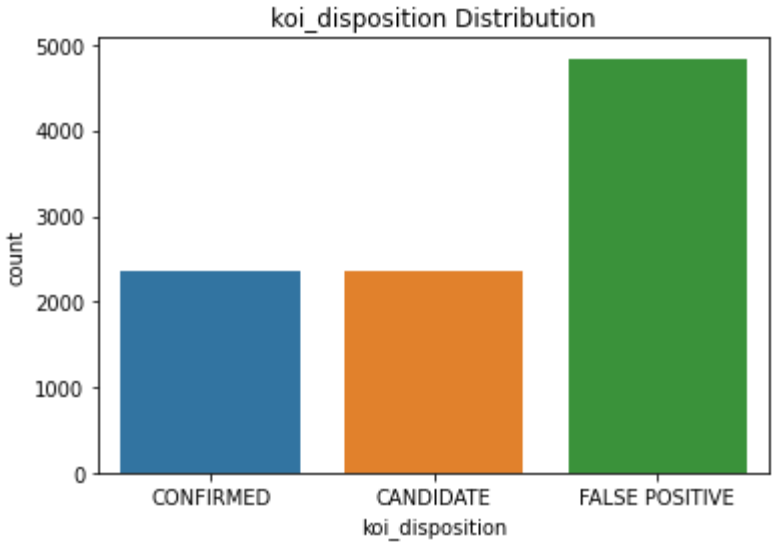
In [ ]:
```
kp['koi_pdisposition'].value_counts()
```

Out[ ]:
```
FALSE POSITIVE    4847
CANDIDATE         4717
Name: koi_pdisposition, dtype: int64
```

We can visualize in a bar chart as well.

In [ ]:
```
sns.countplot(kp['koi_disposition'])
plt.title('koi_disposition Distribution');
```

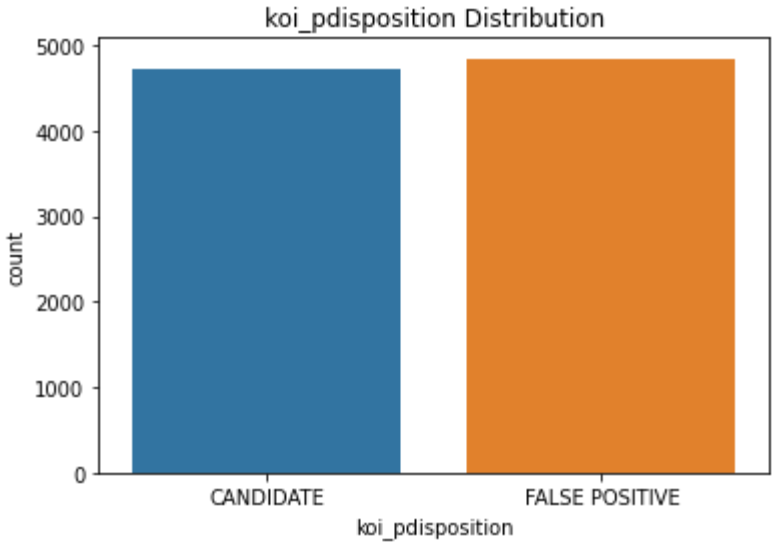
```
/usr/local/lib/python3.7/dist-packages/seaborn/_decorators.py:43: FutureWar
ning: Pass the following variable as a keyword arg: x. From version 0.12, t
he only valid positional argument will be `data`, and passing other argumen
ts without an explicit keyword will result in an error or misinterpretatio
n.
  FutureWarning
```



The large number of false positives may be something to deal with in preprocessing. We can treat this as a binary classification problem, or we can do a multiclass prediction, and include predicting the data will classify certain observations as false positives.

In [ ]:
```
sns.countplot(kp['koi_pdisposition'])
plt.title('koi_pdisposition Distribution');
```

```
/usr/local/lib/python3.7/dist-packages/seaborn/_decorators.py:43: FutureWar
ning: Pass the following variable as a keyword arg: x. From version 0.12, t
he only valid positional argument will be `data`, and passing other argumen
ts without an explicit keyword will result in an error or misinterpretatio
n.
  FutureWarning
```



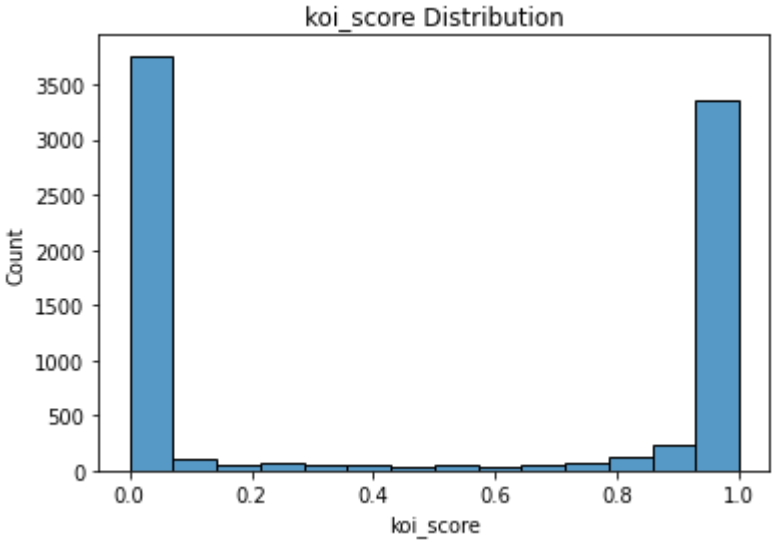
In [ ]:
```
pd.DataFrame(kp['koi_score'].describe())
```

Out[ ]:

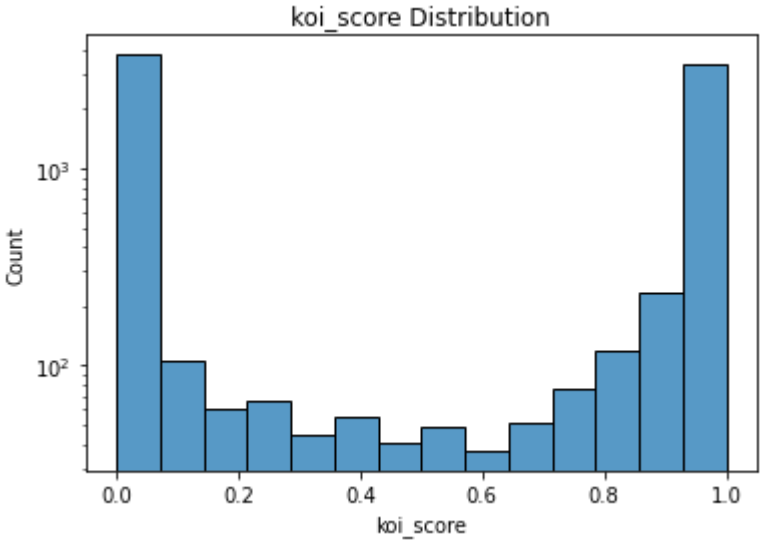
	koi_score
count	8054.000000
mean	0.480829
std	0.476928
min	0.000000
25%	0.000000
50%	0.334000
75%	0.998000
max	1.000000

koi_score looks to be used for assigning candidates and confirmations. This should be left out of the model, as it will probably drive the predictions.

In [ ]:
```
sns.histplot(kp['koi_score'])
plt.title('koi_score Distribution')
plt.show()
```

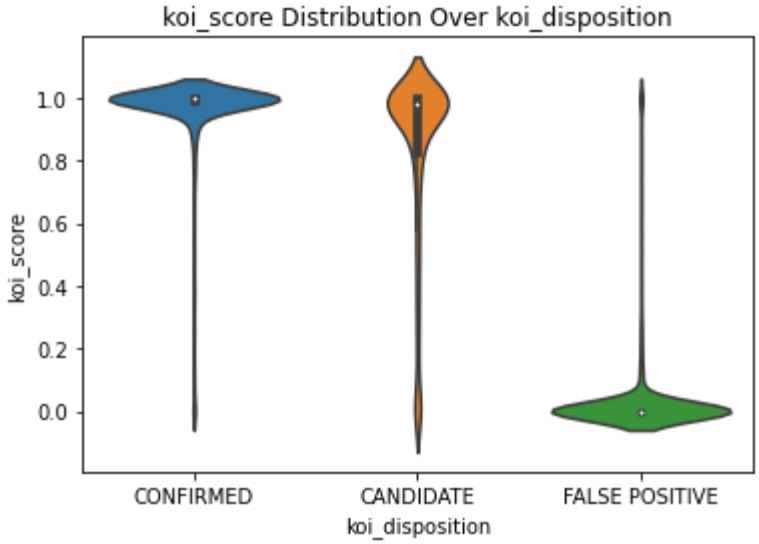


In [ ]:
```
# let's scale the counts to better see the distributions in the center
sns.histplot(kp['koi_score'])
plt.title('koi_score Distribution')
plt.gca().set_yscale('log')
plt.show()
```



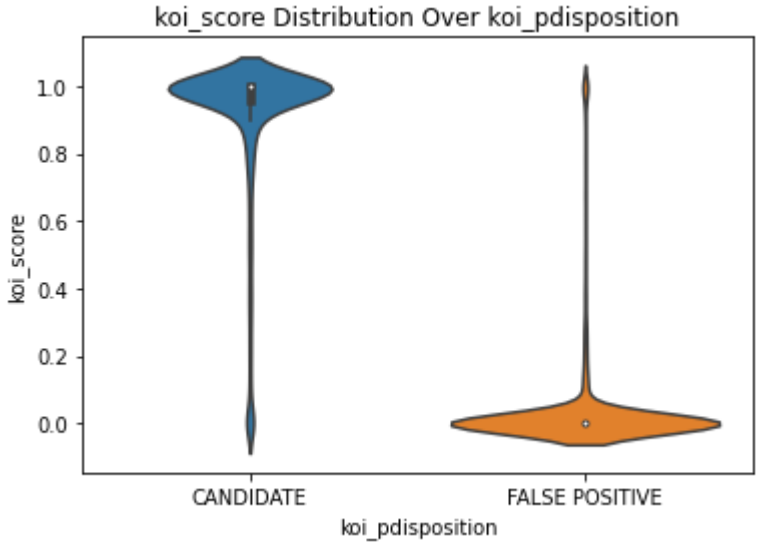
In [ ]:
```
# let's see how our three target variable candidates compare against each o
ther
sns.violinplot(data=kp, x='koi_disposition', y='koi_score')
plt.title('koi_score Distribution Over koi_disposition')
```

Out[ ]:
```
Text(0.5, 1.0, 'koi_score Distribution Over koi_disposition')
```



In [ ]:
```
sns.violinplot(data=kp, x='koi_pdisposition', y='koi_score')
plt.title('koi_score Distribution Over koi_pdisposition')
```

Out[ ]:
```
Text(0.5, 1.0, 'koi_score Distribution Over koi_pdisposition')
```



As suspected, a numerical koi_score of 1 highly correlates to koi_disposition and koi_pdisposition that categorizes an observation as a planet or planet candidate, while a numerical koi_score of 0 highly correlates to a false positive for koi_disposition and koi_pdisposition. koi_score will be excluded from our model as a predictor variable.

koi_disposition was the derived archival categorisation while koi_pdisposition uses the newer Kepler data for categorisation. As we will be working with data collected from the Kepler mission, koi_pdisposition will be used as our target variable.

Feature Reduction

Handling missing Data

In [ ]:
```
#Find rows with missing data.
kp.isnull().sum()
```

Out[ ]:
```
kepid                  0
kepoi_name             0
kepler_name         7205
koi_disposition        0
koi_pdisposition       0
koi_score           1510
koi_fpflag_nt          0
koi_fpflag_ss          0
koi_fpflag_co          0
koi_fpflag_ec          0
koi_period             0
koi_period_err1      454
koi_period_err2      454
koi_time0bk            0
koi_time0bk_err1     454
koi_time0bk_err2     454
koi_impact           363
koi_impact_err1      454
koi_impact_err2      454
koi_duration           0
koi_duration_err1    454
koi_duration_err2    454
koi_depth            363
koi_depth_err1       454
koi_depth_err2       454
koi_prad             363
koi_prad_err1        363
koi_prad_err2        363
koi_teq              363
koi_teq_err1        9564
koi_teq_err2        9564
koi_insol            321
koi_insol_err1       321
koi_insol_err2       321
koi_model_snr        363
koi_tce_plnt_num     346
koi_tce_delivname    346
koi_steff            363
koi_steff_err1       468
koi_steff_err2       483
koi_slogg            363
koi_slogg_err1       468
koi_slogg_err2       468
koi_srad             363
koi_srad_err1        468
koi_srad_err2        468
ra                     0
dec                    0
koi_kepmag             1
dtype: int64
```

Our focus for the predictor variables will be the data measured by Kepler, so we'll remove any derived data from the measurements. This includes nomenclature as well as variables consisting of all missing observations.

In [ ]:
```
kp_df = pd.DataFrame(kp).drop(kp[['kepoi_name','kepid', 'kepler_name','koi_disposition', 'koi_score', 'koi_fpflag_nt',
                                  'koi_fpflag_ss', 'koi_fpflag_co', 'koi_fpflag_ec', 'koi_teq_err1', 'koi_teq_err2', 'koi_tce_delivname']], axis = 1)
```

In [ ]:
```
kp_df.isnull().sum()
```

Out[ ]:
```
koi_pdisposition       0
koi_period             0
koi_period_err1      454
koi_period_err2      454
koi_time0bk            0
koi_time0bk_err1     454
koi_time0bk_err2     454
koi_impact           363
koi_impact_err1      454
koi_impact_err2      454
koi_duration           0
koi_duration_err1    454
koi_duration_err2    454
koi_depth            363
koi_depth_err1       454
koi_depth_err2       454
koi_prad             363
koi_prad_err1        363
koi_prad_err2        363
koi_teq              363
koi_insol            321
koi_insol_err1       321
koi_insol_err2       321
koi_model_snr        363
koi_tce_plnt_num     346
koi_steff            363
koi_steff_err1       468
koi_steff_err2       483
koi_slogg            363
koi_slogg_err1       468
koi_slogg_err2       468
koi_srad             363
koi_srad_err1        468
koi_srad_err2        468
ra                     0
dec                    0
koi_kepmag             1
dtype: int64
```

We'll look at the impact of dropping the rows with missing data on the size of the data set.

In [ ]:
```
kp_drop = kp_df.dropna()
kp_drop.shape
```

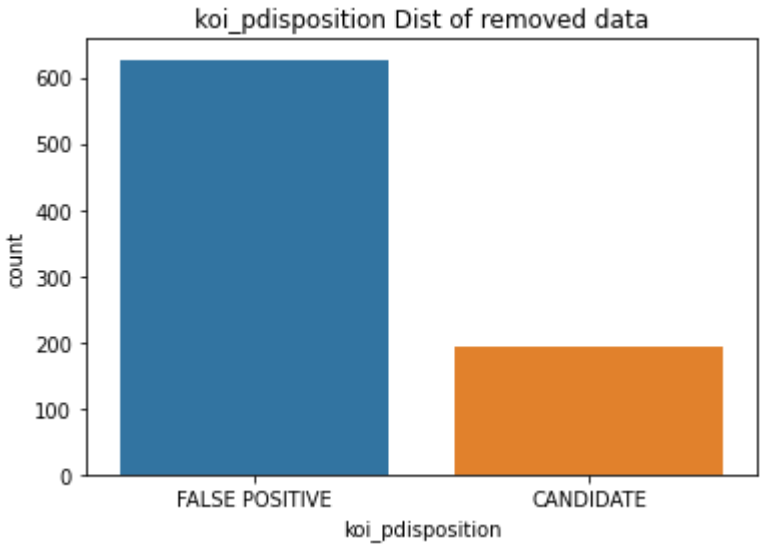
Out[ ]:
```
(8744, 37)
```

In [ ]:
```
null_data = kp_df[kp_df.isnull().any(axis=1)]
print(null_data.shape)
sns.countplot(null_data['koi_pdisposition'])
plt.title('koi_pdisposition Dist of removed data')
plt.show
```

```
(820, 37)

/usr/local/lib/python3.7/dist-packages/seaborn/_decorators.py:43: FutureWar
ning: Pass the following variable as a keyword arg: x. From version 0.12, t
he only valid positional argument will be `data`, and passing other argumen
ts without an explicit keyword will result in an error or misinterpretatio
n.
  FutureWarning
```

Out[ ]:
```
<function matplotlib.pyplot.show>
```

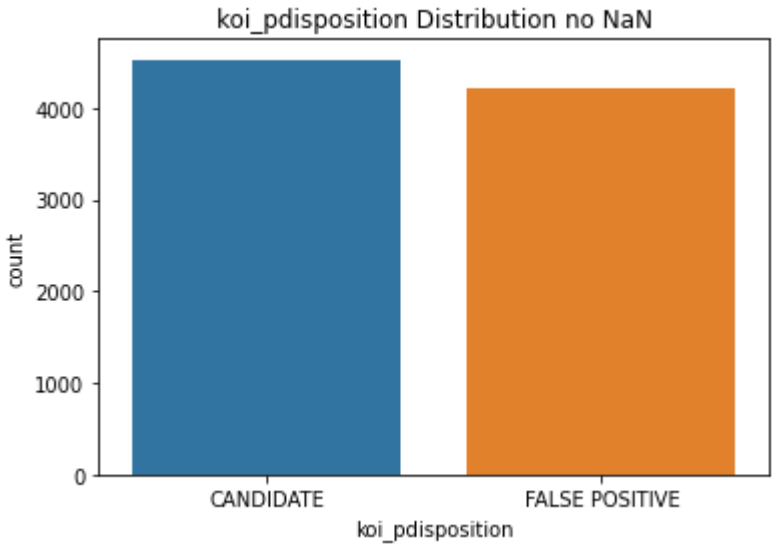


There are 820 rows with missing data, which comprises less than 10% of the total data set. Further, of the rows with missing data, over 600 of the 820 are false positives. This suggests missing data is likely to lead to a false positive. Another way to look at it may be that the missing data is biasing the model to a false positive. Rather than imputing, which may still drive bias into the data set, we decided to remove the rows with missing data.

In [ ]:
```
sns.countplot(kp_drop['koi_pdisposition'])
plt.title('koi_pdisposition Distribution no NaN')
plt.show
```

```
/usr/local/lib/python3.7/dist-packages/seaborn/_decorators.py:43: FutureWar
ning: Pass the following variable as a keyword arg: x. From version 0.12, t
he only valid positional argument will be `data`, and passing other argumen
ts without an explicit keyword will result in an error or misinterpretatio
n.
  FutureWarning
```

Out[ ]:
```
<function matplotlib.pyplot.show>
```

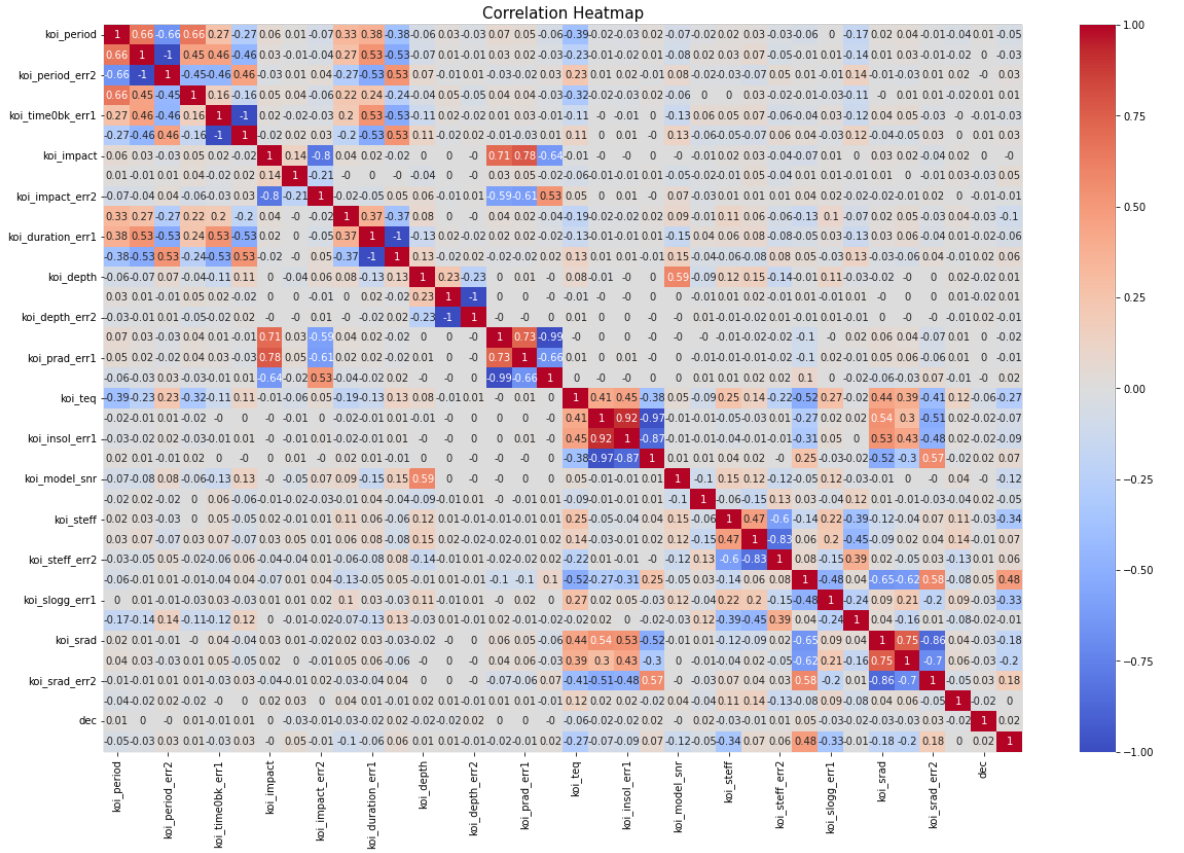


The distribution of false positives to candidates is still close to even after removing rows with missing data. This suggests balance, and better chances of a generalized model.

Find highly correlated variables. For this project, we'll set this at an absolute correlation coefficient greater than or equal to 0.70.

In [ ]:

```
matrix = kp_drop.corr().round(2)
sns.heatmap(matrix, annot=True, cmap='coolwarm')
plt.title('Correlation Heatmap', fontsize=15)
plt.gcf().set_size_inches(20, 13)
plt.show()
```



In [ ]:
```
# let's list these high correlation values
matrix = kp_drop.corr()
matrix = matrix.unstack()
matrix = matrix[abs(matrix) >= 0.7]
matrix = matrix[matrix != 1]
matrix = matrix[matrix != -1]

print(matrix)
```

```
koi_impact       koi_impact_err2   -0.803663
                 koi_prad           0.710456
                 koi_prad_err1      0.780784
koi_impact_err2  koi_impact        -0.803663
koi_prad         koi_impact         0.710456
                 koi_prad_err1      0.725682
                 koi_prad_err2     -0.988068
koi_prad_err1    koi_impact         0.780784
                 koi_prad           0.725682
koi_prad_err2    koi_prad          -0.988068
koi_insol        koi_insol_err1     0.917568
                 koi_insol_err2    -0.965981
koi_insol_err1   koi_insol          0.917568
                 koi_insol_err2    -0.870049
koi_insol_err2   koi_insol         -0.965981
                 koi_insol_err1    -0.870049
koi_steff_err1   koi_steff_err2    -0.825551
koi_steff_err2   koi_steff_err1    -0.825551
koi_srad         koi_srad_err1      0.754008
                 koi_srad_err2     -0.857725
koi_srad_err1    koi_srad           0.754008
koi_srad_err2    koi_srad          -0.857725
dtype: float64
```

Matrix variables may be removed for some of our models to avoid multicollinearity.

In [ ]:
```
# IDing variables that may be removed to avoid multicollinearity
multicol_candidates = kp_drop.corr().where(np.triu(np.ones(kp_drop.corr().s
hape),k=1).astype(bool))
multicol_drop = [column for column in multicol_candidates.columns if any(ab
s(multicol_candidates[column]) >= 0.70)]
print('The multicollinearity candidates to drop are:')
list(multicol_drop)
```

```
The multicollinearity candidates to drop are:
```

Out[ ]:
```
['koi_period_err2',
 'koi_time0bk_err2',
 'koi_impact_err2',
 'koi_duration_err2',
 'koi_depth_err2',
 'koi_prad',
 'koi_prad_err1',
 'koi_prad_err2',
 'koi_insol_err1',
 'koi_insol_err2',
 'koi_steff_err2',
 'koi_srad_err1',
 'koi_srad_err2']
```

In [ ]:
```
len(multicol_drop)
```

Out[ ]:
```
13
```

For some models, we may drop an additional 13 variables to avoid multicollinearity.

Modeling

Train and Test

In [ ]:
```
#define the X and y:
y = kp_drop['koi_pdisposition']
X = pd.DataFrame(kp_drop.drop(kp_drop[['koi_pdisposition']], axis = 1))
```

In [ ]:
```
Xtrain, Xtest, ytrain, ytest = train_test_split(X, y, test_size=.2, random_
state=5)
print(Xtrain.shape)
print(ytrain.shape)
print(Xtest.shape)
print(ytest.shape)
```

```
(6995, 36)
(6995,)
(1749, 36)
(1749,)
```

In [ ]:
```
Xtest.dtypes
```

Out[ ]:
```
koi_period           float64
koi_period_err1      float64
koi_period_err2      float64
koi_time0bk          float64
koi_time0bk_err1     float64
koi_time0bk_err2     float64
koi_impact           float64
koi_impact_err1      float64
koi_impact_err2      float64
koi_duration         float64
koi_duration_err1    float64
koi_duration_err2    float64
koi_depth            float64
koi_depth_err1       float64
koi_depth_err2       float64
koi_prad             float64
koi_prad_err1        float64
koi_prad_err2        float64
koi_teq              float64
koi_insol            float64
koi_insol_err1       float64
koi_insol_err2       float64
koi_model_snr        float64
koi_tce_plnt_num     float64
koi_steff            float64
koi_steff_err1       float64
koi_steff_err2       float64
koi_slogg            float64
koi_slogg_err1       float64
koi_slogg_err2       float64
koi_srad             float64
koi_srad_err1        float64
koi_srad_err2        float64
ra                   float64
dec                  float64
koi_kepmag           float64
dtype: object
```

In [ ]:
```
sns.countplot(ytrain['koi_pdisposition'])
plt.title('koi_pdisposition Distribution Train Data')
plt.show
```

```
---------------------------------------------------------------------------
NameError                                 Traceback (most recent call last)
<ipython-input-1-e16763ca991a> in <module>()
----> 1 sns.countplot(ytrain['koi_pdisposition'])
      2 plt.title('koi_pdisposition Distribution Train Data')
      3 plt.show

NameError: name 'sns' is not defined
```

Models

Linear Models

Perceptron

Modify the labels so they are plus and minus

In [ ]:
```
le = LabelEncoder()
ytrain_perc = le.fit_transform(ytrain)
ytrain_perc[ytrain_perc==0] = -1
ytrain_perc = pd.DataFrame(ytrain_perc)
ytrain_perc
```

Out[ ]:

	0
0	1
1	-1
2	-1
3	-1
4	1
...	...
6990	1
6991	1
6992	1
6993	-1
6994	1

6995 rows × 1 columns

In [ ]:
```
le = LabelEncoder()
ytest_perc = le.fit_transform(ytest)
ytest_perc[ytest_perc==0] = -1
ytest_perc = pd.DataFrame(ytest_perc)
ytest_perc
```

Out[ ]:

	0
0	-1
1	-1
2	1
3	-1
4	-1
...	...
1744	1
1745	1
1746	-1
1747	1
1748	-1

1749 rows × 1 columns

Fit the model:

In [ ]:
```
#model using defaults
perc = Perceptron().fit(Xtrain, ytrain_perc)
```

```
/usr/local/lib/python3.7/dist-packages/sklearn/utils/validation.py:993: Dat
aConversionWarning: A column-vector y was passed when a 1d array was expect
ed. Please change the shape of y to (n_samples, ), for example using ravel
().
  y = column_or_1d(y, warn=True)
```

In [ ]:
```
perc_cm = confusion_matrix(ytest_perc, perc.predict(Xtest))
disposition_labels = {'False Positive', 'Candidate'}
pd.DataFrame(perc_cm, index = disposition_labels, columns= disposition_labe
ls)
```

Out[ ]:

	False Positive	Candidate
False Positive	562	326
Candidate	224	637

In [ ]:
```
pred_labels = perc.predict(Xtest)
actual_labels = np.array(ytest_perc)
#ytest_probs = perc.predict(Xtest)[:,1]
perc_acc = float(accuracy_score(actual_labels, pred_labels))*100

#print("AUC:", float(roc_auc_score(ytest_perc, ytest_probs)))
print('Accuracy:', float(accuracy_score(actual_labels, pred_labels))*100,
'%')
print('Classification Stats:')
print(classification_report(actual_labels, pred_labels))
```

```
Accuracy: 68.55345911949685 %
Classification Stats:
              precision    recall  f1-score   support

          -1       0.72      0.63      0.67       888
           1       0.66      0.74      0.70       861

    accuracy                           0.69      1749
   macro avg       0.69      0.69      0.68      1749
weighted avg       0.69      0.69      0.68      1749
```

In [ ]:
```
#look for the best parameters:
params = {'max_iter': [1, 5, 10, 50, 100, 1000]}

perc1 = GridSearchCV(Perceptron(), params, cv=5, n_jobs=2, scoring='accurac
y')
perc1.fit(Xtrain, ytrain_perc)
print('Perceptron parameters: ', perc1.best_params_)
```

```
Perceptron parameters:  {'max_iter': 5}

/usr/local/lib/python3.7/dist-packages/sklearn/utils/validation.py:993: Dat
aConversionWarning: A column-vector y was passed when a 1d array was expect
ed. Please change the shape of y to (n_samples, ), for example using ravel
().
  y = column_or_1d(y, warn=True)
/usr/local/lib/python3.7/dist-packages/sklearn/linear_model/_stochastic_gra
dient.py:700: ConvergenceWarning: Maximum number of iteration reached befor
e convergence. Consider increasing max_iter to improve the fit.
  ConvergenceWarning,
```

In [ ]:
```
#fit the best model:
perc2 = LogisticRegression(max_iter=5)
perc2.fit(Xtrain, ytrain_perc)
```

```
/usr/local/lib/python3.7/dist-packages/sklearn/utils/validation.py:993: Dat
aConversionWarning: A column-vector y was passed when a 1d array was expect
ed. Please change the shape of y to (n_samples, ), for example using ravel
().
  y = column_or_1d(y, warn=True)
/usr/local/lib/python3.7/dist-packages/sklearn/linear_model/_logistic.py:81
8: ConvergenceWarning: lbfgs failed to converge (status=1):
STOP: TOTAL NO. of ITERATIONS REACHED LIMIT.

Increase the number of iterations (max_iter) or scale the data as shown in:
    https://scikit-learn.org/stable/modules/preprocessing.html
Please also refer to the documentation for alternative solver options:
    https://scikit-learn.org/stable/modules/linear_model.html#logistic-regr
ession
  extra_warning_msg=_LOGISTIC_SOLVER_CONVERGENCE_MSG,
```

Out[ ]:
```
LogisticRegression(max_iter=5)
```

In [ ]:
```
perc2_cm = confusion_matrix(ytest_perc, perc2.predict(Xtest))
disposition_labels = {'Candidate', 'False Positive'}
pd.DataFrame(perc2_cm, index = disposition_labels, columns= disposition_lab
els)
```

Out[ ]:

	False Positive	Candidate
False Positive	854	34
Candidate	460	401

In [ ]:
```
pred_labels = perc2.predict(Xtest)
actual_labels = np.array(ytest_perc)

print('Accuracy:', float(accuracy_score(actual_labels, pred_labels))*100,
'%')
print('Classification Stats:')
print(classification_report(actual_labels, pred_labels))
```

```
Accuracy: 71.75528873642081 %
Classification Stats:
              precision    recall  f1-score   support

          -1       0.65      0.96      0.78       888
           1       0.92      0.47      0.62       861

    accuracy                           0.72      1749
   macro avg       0.79      0.71      0.70      1749
weighted avg       0.78      0.72      0.70      1749
```

Logistic Regression

This model will require more preprocessing, such as centering, scaling, and removing predictors with near zero variance.

In [ ]:
```
#Remove predictors with near zero variance (NZV):
threshold_n=0.95
sel = VarianceThreshold(threshold=(threshold_n* (1 - threshold_n) ))
sel_var=sel.fit_transform(Xtrain)
Xtrain_NZV = Xtrain[Xtrain.columns[sel.get_support(indices=True)]]
print(Xtrain_NZV.shape)

threshold_n=0.95
sel = VarianceThreshold(threshold=(threshold_n* (1 - threshold_n) ))
sel_var=sel.fit_transform(Xtest)
Xtest_NZV = Xtest[Xtest.columns[sel.get_support(indices=True)]]
print(Xtest_NZV.shape)
```

```
(6995, 30)
(1749, 30)
```

In [ ]:
```
#Remove highly correlated predictors:
multicol_candidates = Xtrain_NZV.corr().where(np.triu(np.ones(Xtrain_NZV.co
rr().shape),k=1).astype(bool))
multicol_drop = [column for column in multicol_candidates.columns if any(ab
s(multicol_candidates[column]) >= 0.70)]
Xtrain_NZV_corr = Xtrain_NZV.drop(multicol_drop, axis=1)
Xtest_NZV_corr = Xtest_NZV.drop(multicol_drop, axis=1)
```

In [ ]:
```
#Center the predictor:
norm_X = Normalizer()
norm_X.fit(Xtrain_NZV_corr)
Xtrain_NZV_corr_norm = norm_X.transform(Xtrain_NZV_corr)
Xtest_NZV__corr_norm = norm_X.transform(Xtest_NZV_corr)
```

In [ ]:
```
#Scale the predictor data:
sc_X = StandardScaler()
sc_X.fit(Xtrain_NZV_corr_norm)
Xtrain_NZV_corr_norm_sc = sc_X.transform(Xtrain_NZV_corr_norm)
Xtest_NZV__corr_norm_sc = sc_X.transform(Xtest_NZV__corr_norm)
Xtrain_lr = Xtrain_NZV_corr_norm_sc
Xtest_lr = Xtest_NZV__corr_norm_sc
```

In [ ]:
```
#fit the logistic regression model using the default params on the preproce
ssed training data:
logreg = LogisticRegression(random_state=5).fit(Xtrain_lr, ytrain)
```

In [ ]:
```
#check performance on the test data set:
logreg_cm = confusion_matrix(ytest, logreg.predict(Xtest_lr))
disposition_labels = {'Candidate', 'False Positive'}
pd.DataFrame(logreg_cm, index = disposition_labels, columns= disposition_la
bels)
```

Out[ ]:

	False Positive	Candidate
False Positive	758	130
Candidate	190	671

In [ ]:
```
pred_labels = logreg.predict(Xtest_lr)
actual_labels = np.array(ytest)
ytest_probs = logreg.predict_proba(Xtest_lr)[:,1]

print("AUC:", float(roc_auc_score(ytest, ytest_probs)))
print('Accuracy:', float(accuracy_score(actual_labels, pred_labels))*100,
'%')
print('Classification Stats:')
print(classification_report(actual_labels, pred_labels))
```

```
AUC: 0.9019838653984996
Accuracy: 81.70383076043454 %
Classification Stats:
                precision    recall  f1-score   support

     CANDIDATE       0.80      0.85      0.83       888
FALSE POSITIVE       0.84      0.78      0.81       861

      accuracy                           0.82      1749
     macro avg       0.82      0.82      0.82      1749
  weighted avg       0.82      0.82      0.82      1749
```

In [ ]:
```
#look for the best parameters:
params = {'C':[0.001, 0.01, 0.1, 1, 10, 100, 1000], 'fit_intercept':[True,
False]}

logreg1 = GridSearchCV(LogisticRegression(), params, cv=5, n_jobs=2, scorin
g='roc_auc')
logreg1.fit(Xtrain_lr, ytrain)
print('Logistic Regression parameters: ', logreg1.best_params_)
```

```
Logistic Regression parameters:  {'C': 1000, 'fit_intercept': True}
```

In [ ]:
```
#fit the best model:
logreg2 = LogisticRegression(C = 1000, fit_intercept= True)
logreg2.fit(Xtrain_lr, ytrain)
```

Out[ ]:
```
LogisticRegression(C=1000)
```

In [ ]:
```
logreg2_cm = confusion_matrix(ytest, logreg2.predict(Xtest_lr))
disposition_labels = {'Candidate', 'False Positive'}
pd.DataFrame(logreg2_cm, index = disposition_labels, columns= disposition_l
abels)
```

Out[ ]:

	False Positive	Candidate
False Positive	759	129
Candidate	190	671

In [ ]:
```
pred_labels = logreg2.predict(Xtest_lr)
actual_labels = np.array(ytest)
ytest_probs = logreg.predict_proba(Xtest_lr)[:,1]
logreg_acc = float(accuracy_score(actual_labels, pred_labels))*100

print("AUC:", float(roc_auc_score(ytest, ytest_probs)))
print('Accuracy:', float(accuracy_score(actual_labels, pred_labels))*100,
'%')
print('Classification Stats:')
print(classification_report(actual_labels, pred_labels))
```

```
AUC: 0.9019838653984996
Accuracy: 81.76100628930818 %
Classification Stats:
                precision    recall  f1-score   support

     CANDIDATE       0.80      0.85      0.83       888
FALSE POSITIVE       0.84      0.78      0.81       861

      accuracy                           0.82      1749
     macro avg       0.82      0.82      0.82      1749
  weighted avg       0.82      0.82      0.82      1749
```

Support Vector Machine

Requires Centering and scaling of the test/train data

In [ ]:
```
#Center the predictor:
norm_X = Normalizer()
norm_X.fit(Xtrain)
Xtrain_norm = norm_X.transform(Xtrain)
Xtest_norm = norm_X.transform(Xtest)
```

In [ ]:
```
#Scale the predictor data:
sc_X = StandardScaler()
sc_X.fit(Xtrain_norm)
Xtrain_norm_sc = sc_X.transform(Xtrain_norm)
Xtest_norm_sc = sc_X.transform(Xtest_norm)
Xtrain_svm = Xtrain_norm_sc
Xtest_svm = Xtest_norm_sc
```

In [ ]:
```
#build model based on default parameters and a linear kernel:
svm = SVC(kernel='linear')
svm.fit(Xtrain_svm, ytrain)
```

Out[ ]:
```
SVC(kernel='linear')
```

In [ ]:
```
#check performance on the test data set:
svm_cm = confusion_matrix(ytest, svm.predict(Xtest_svm))
disposition_labels = {'Candidate', 'False Positive'}
pd.DataFrame(svm_cm, index = disposition_labels, columns= disposition_label
s)
```

Out[ ]:

	False Positive	Candidate
False Positive	771	117
Candidate	169	692

In [ ]:
```
accuracy_score(ytest, svm.predict(Xtest_svm))
```

Out[ ]:
```
0.8364779874213837
```

In [ ]:
```
# Finding the best hyperparameters
params = {
    'C': [0.1, 1, 10, 100, 1000],
    'gamma': [1, 0.1, 0.01, 0.001, 0.0001],
    'kernel': ['rbf']
}

svm1 = GridSearchCV(estimator=SVC(), param_grid=params, cv=5, n_jobs=5, ver
bose=1)

svm1.fit(Xtrain_svm, ytrain)
```

```
Fitting 5 folds for each of 25 candidates, totalling 125 fits
```

Out[ ]:
```
GridSearchCV(cv=5, estimator=SVC(), n_jobs=5,
             param_grid={'C': [0.1, 1, 10, 100, 1000],
                         'gamma': [1, 0.1, 0.01, 0.001, 0.0001],
                         'kernel': ['rbf']},
             verbose=1)
```

In [ ]:
```
svm1.best_params_
```

Out[ ]:
```
{'C': 1000, 'gamma': 0.001, 'kernel': 'rbf'}
```

In [ ]:
```
#fit the best model:
svm2 = SVC(kernel='rbf', gamma=0.001, C = 1000)
svm2.fit(Xtrain_svm, ytrain)
```

Out[ ]:
```
SVC(C=1000, gamma=0.001)
```

In [ ]:
```
svm2_cm = confusion_matrix(ytest, svm2.predict(Xtest_svm))
disposition_labels = {'Candidate', 'False Positive'}
pd.DataFrame(svm2_cm, index = disposition_labels, columns= disposition_labe
ls)
```

Out[ ]:

	False Positive	Candidate
False Positive	774	114
Candidate	147	714

In [ ]:
```
pred_labels = svm2.predict(Xtest_svm)
actual_labels = np.array(ytest)
SVM_acc = float(accuracy_score(actual_labels, pred_labels))*100

print('Accuracy:', float(accuracy_score(actual_labels, pred_labels))*100,
'%')
print('Classification Stats:')
print(classification_report(actual_labels, pred_labels))
```

```
Accuracy: 85.07718696397941 %
Classification Stats:
                precision    recall  f1-score   support

     CANDIDATE       0.84      0.87      0.86       888
FALSE POSITIVE       0.86      0.83      0.85       861

      accuracy                           0.85      1749
     macro avg       0.85      0.85      0.85      1749
  weighted avg       0.85      0.85      0.85      1749
```

Stochastic Gradient Descent (SGD)

In [ ]:
```
#We'll use the scaled data for this model
#Scale the predictor data:
sc_X = StandardScaler()
sc_X.fit(Xtrain)
Xtrain_sc = sc_X.transform(Xtrain)
Xtest_sc = sc_X.transform(Xtest)
Xtrain_sgd = Xtrain_sc
Xtest_sgd = Xtest_sc
```

In [ ]:
```
#fit the default model:
sgd = SGDClassifier()
sgd.fit(Xtrain_sc, ytrain)
```

Out[ ]:
```
SGDClassifier()
```

In [ ]:
```
#check performance on the test data set:
sgd_cm = confusion_matrix(ytest, svm.predict(Xtest_sgd))
disposition_labels = {'Candidate', 'False Positive'}
pd.DataFrame(sgd_cm, index = disposition_labels, columns= disposition_label
s)
```

Out[ ]:

	False Positive	Candidate
False Positive	586	302
Candidate	356	505

In [ ]:
```
accuracy_score(ytest, sgd.predict(Xtest_sgd))
```

Out[ ]:
```
0.8164665523156089
```

In [ ]:
```
#now check for the best hyperparameters:
# Finding the best hyperparameters
params = {
    'loss': ['log_loss', 'modified_huber', 'hinge'],
    'alpha': [0.0001, 0.001, 0.01, 0.1, 1.0, 10, 100],
    'max_iter': [1, 5, 10, 50, 100, 1000],
    'learning_rate': ['constant', 'optimal', 'invscaling', 'adaptive']
}

sgd1 = GridSearchCV(estimator=SGDClassifier(), param_grid=params, cv=5, n_j
obs=5, verbose=1)

sgd1.fit(Xtrain_sgd, ytrain)
```

In [ ]:
```
print(sgd1.best_params_)
```

```
{'alpha': 0.001, 'learning_rate': 'optimal', 'loss': 'hinge', 'max_iter': 1
00}
```

In [ ]:
```
#fit the best model:
sgd2 = SGDClassifier(alpha= 0.001, learning_rate= 'optimal', loss='hinge',
max_iter=100)
sgd2.fit(Xtrain_sgd, ytrain)
```

Out[ ]:
```
SGDClassifier(alpha=0.001, max_iter=100)
```

In [ ]:
```
sgd2_cm = confusion_matrix(ytest, sgd2.predict(Xtest_sgd))
disposition_labels = {'Candidate', 'False Positive'}
pd.DataFrame(sgd2_cm, index = disposition_labels, columns= disposition_labe
ls)
```

Out[ ]:

	False Positive	Candidate
False Positive	795	93
Candidate	199	662

In [ ]:
```
pred_labels = sgd2.predict(Xtest_sgd)
actual_labels = np.array(ytest)
SGD_acc = float(accuracy_score(actual_labels, pred_labels))*100

print('Accuracy:', float(accuracy_score(actual_labels, pred_labels))*100,
'%')
print('Classification Stats:')
print(classification_report(actual_labels, pred_labels))
```

```
Accuracy: 83.30474556889651 %
Classification Stats:
                precision    recall  f1-score   support

     CANDIDATE       0.80      0.90      0.84       888
FALSE POSITIVE       0.88      0.77      0.82       861

      accuracy                           0.83      1749
     macro avg       0.84      0.83      0.83      1749
  weighted avg       0.84      0.83      0.83      1749
```

Nonlinear Models

KNN

In [ ]:
```
# let's preprocess our target variables by converting to binary output
le = LabelEncoder()

# target train
ytrain_knn = le.fit_transform(ytrain)
ytrain_knn = pd.DataFrame(ytrain_knn)

# target test
ytest_knn = le.fit_transform(ytest)
ytest_knn = pd.DataFrame(ytest_knn)
```

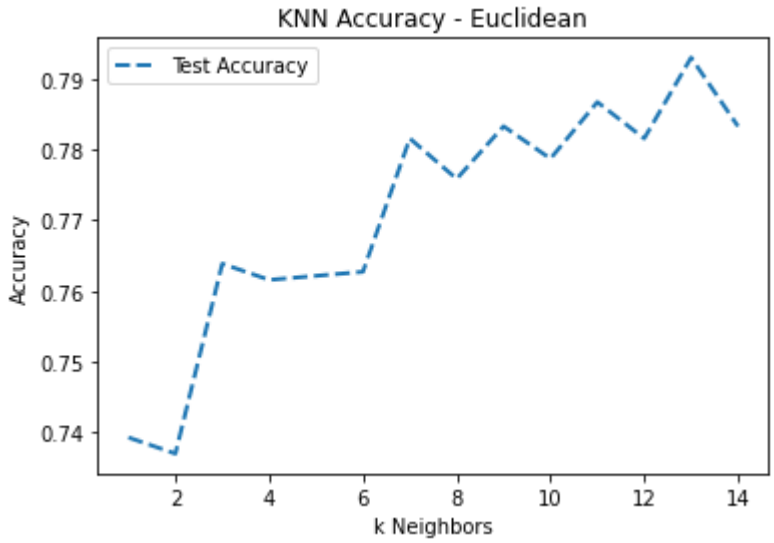
In [ ]:
```
# define k_neighbors
def k_neighbors(Xtrain, ytrain, Xtest, ytest, kvalues, metric):
  knn_accuracy = []
  clfs = []
  for i in kvalues:
    clf = KNeighborsClassifier(metric=metric, n_neighbors=i).fit(Xtrain, yt
rain)
    clf_train_pred = clf.predict(Xtrain)
    clf_test_pred = clf.predict(Xtest)
    clfs.append(clf)
    knn_accuracy.append({'k values': i,
    'Training Accuracy':accuracy_score(clf_train_pred, ytrain),
    'Test Accuracy': accuracy_score(clf_test_pred, ytest)})
  return pd.DataFrame(knn_accuracy), clfs
```

KNN - Euclidean

In [ ]:
```
# let's check performance using euclidean metric
knn_euc_acc, knn_euc_clfs = k_neighbors(Xtrain, ytrain, Xtest, ytest, range
(1,15), metric='euclidean')
display(knn_euc_acc)
```

	k values	Training Accuracy	Test Accuracy
0	1	1.000000	0.739280
1	2	0.864332	0.736993
2	3	0.873052	0.763865
3	4	0.837312	0.761578
4	5	0.843888	0.762150
5	6	0.822731	0.762722
6	7	0.825447	0.781589
7	8	0.816297	0.775872
8	9	0.818156	0.783305
9	10	0.809149	0.778731
10	11	0.813724	0.786735
11	12	0.805861	0.781589
12	13	0.804718	0.793025
13	14	0.799571	0.783305

In [ ]:
```
# let's plot the test accuracy against k values to find optimal k
plt.plot(knn_euc_acc['k values'], knn_euc_acc['Test Accuracy'], '--', linew
idth=2, label='Test Accuracy')
plt.xlabel('k Neighbors')
plt.ylabel('Accuracy')
plt.title('KNN Accuracy - Euclidean')
plt.legend()
plt.show()
```



Looking at the test accuracy, K values after 7 provide minimal increases in test accuracy, so we will choose k=7.

In [ ]:
```
# let's apply best k=7
clf_knn_euc = KNeighborsClassifier(n_neighbors=7, metric='minkowski', p=2)
clf_knn_euc.fit(Xtrain, ytrain)
knn_euc_pred = clf_knn_euc.predict(Xtest)
```

In [ ]:
```
# confusion matrix to see performance
knn_euc_cm = confusion_matrix(ytest, knn_euc_pred)
disposition_labels = {'Candidate', 'False Positive'}
pd.DataFrame(knn_euc_cm, index = disposition_labels, columns= disposition_l
abels)
```

Out[ ]:

	False Positive	Candidate
False Positive	740	148
Candidate	234	627

In [ ]:
```
actual_labels = np.array(ytest)
euc_acc = float(accuracy_score(actual_labels, knn_euc_pred))*100
print('Accuracy:', float(accuracy_score(actual_labels, knn_euc_pred))*100,
'%')
```

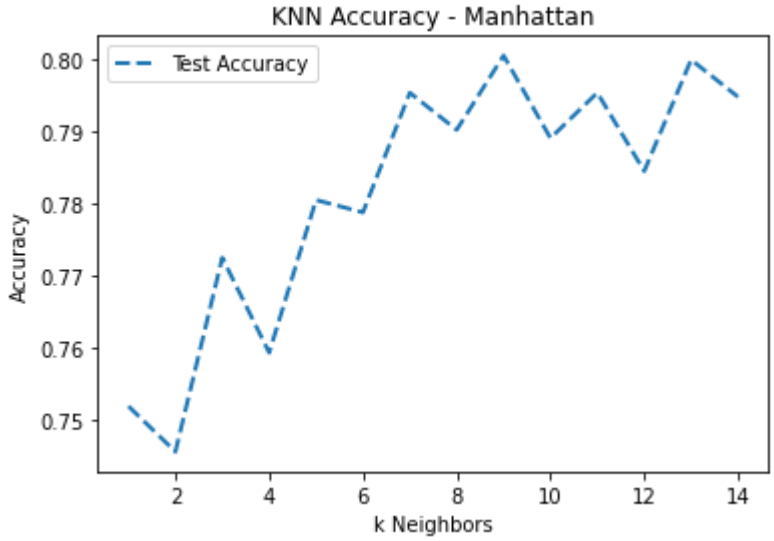
```
Accuracy: 78.15894797026873 %
```

KNN - Manhattan

In [ ]:
```
# let's check performance using manhattan distance
knn_man_acc, knn_man_clfs = k_neighbors(Xtrain, ytrain, Xtest, ytest, range
(1,15), metric='manhattan')
display(knn_man_acc)
```

	k values	Training Accuracy	Test Accuracy
0	1	1.000000	0.751858
1	2	0.869192	0.745569
2	3	0.876340	0.772441
3	4	0.841315	0.759291
4	5	0.848177	0.780446
5	6	0.826162	0.778731
6	7	0.834167	0.795312
7	8	0.821873	0.790166
8	9	0.827162	0.800457
9	10	0.816440	0.789022
10	11	0.820014	0.795312
11	12	0.812294	0.784448
12	13	0.811723	0.799886
13	14	0.806862	0.794740

In [ ]:
```
# same thing, let's plot the test accuracy against k values to find optimal
k
plt.plot(knn_man_acc['k values'], knn_man_acc['Test Accuracy'], '--', linew
idth=2, label='Test Accuracy')
plt.xlabel('k Neighbors')
plt.ylabel('Accuracy')
plt.title('KNN Accuracy - Manhattan')
plt.legend()
plt.show()
```



As with Euclidean distance, K values past 7 provide minimal increase in test accuracy, so k=7 will be used.

In [ ]:
```
# let's apply best k=7
clf_knn_man = KNeighborsClassifier(n_neighbors=7, metric='minkowski', p=1)
clf_knn_man.fit(Xtrain, ytrain)
knn_man_pred = clf_knn_man.predict(Xtest)
```

In [ ]:
```
# confusion matrix to see performance
knn_man_cm = confusion_matrix(ytest, knn_man_pred)
disposition_labels = {'Candidate', 'False Positive'}
pd.DataFrame(knn_man_cm, index = disposition_labels, columns= disposition_l
abels)
```

Out[ ]:

	False Positive	Candidate
False Positive	745	143
Candidate	215	646

In [ ]:
```
man_acc = float(accuracy_score(actual_labels, knn_man_pred))*100
print('Accuracy:', float(accuracy_score(actual_labels, knn_man_pred))*100,
'%')
```

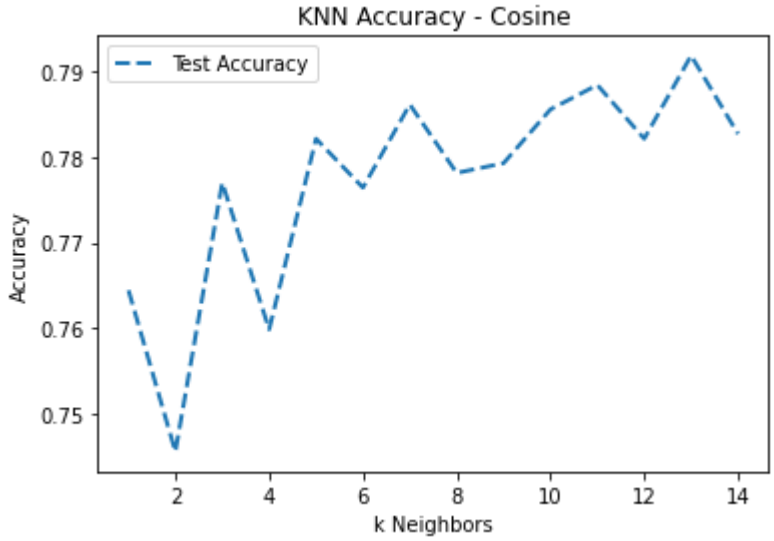
```
Accuracy: 79.53116066323614 %
```

KNN - Cosine

In [ ]:
```
# let's check performance using cosine distance
knn_cos_acc, knn_cos_clfs = k_neighbors(Xtrain, ytrain, Xtest, ytest, range
(1,15), metric='cosine')
display(knn_cos_acc)
```

	k values	Training Accuracy	Test Accuracy
0	1	1.000000	0.764437
1	2	0.862044	0.745569
2	3	0.868192	0.777015
3	4	0.838885	0.759863
4	5	0.839028	0.782161
5	6	0.818442	0.776444
6	7	0.822588	0.786164
7	8	0.815011	0.778159
8	9	0.815297	0.779302
9	10	0.809864	0.785592
10	11	0.811294	0.788451
11	12	0.806004	0.782161
12	13	0.807434	0.791881
13	14	0.803288	0.782733

In [ ]:
```
# same thing, let's plot the test accuracy against k values to find optimal
k
plt.plot(knn_cos_acc['k values'], knn_cos_acc['Test Accuracy'], '--', linew
idth=2, label='Test Accuracy')
plt.xlabel('k Neighbors')
plt.ylabel('Accuracy')
plt.title('KNN Accuracy - Cosine')
plt.legend()
plt.show()
```



As with both euclidean and manhattan distance, cosine distance shows minimal test accuracy improvements after K value of 7, so k=7 will be chosen.

In [ ]:
```
# let's apply best k=7
clf_knn_cos = KNeighborsClassifier(n_neighbors=7, metric='cosine')
clf_knn_cos.fit(Xtrain, ytrain)
knn_cos_pred = clf_knn_cos.predict(Xtest)
```

In [ ]:
```
# confusion matrix to see performance
knn_cos_cm = confusion_matrix(ytest, knn_cos_pred)
disposition_labels = {'Candidate', 'False Positive'}
pd.DataFrame(knn_cos_cm, index = disposition_labels, columns= disposition_l
abels)
```

Out[ ]:

	False Positive	Candidate
False Positive	737	151
Candidate	223	638

In [ ]:
```
cos_acc = float(accuracy_score(actual_labels, knn_cos_pred))*100
print('Accuracy:', float(accuracy_score(actual_labels, knn_cos_pred))*100,
'%')
```

```
Accuracy: 78.61635220125787 %
```

In terms of accuracy, KNN using Manhattan distance performed better and resulted in higher accuracy.

Decision Tree

In [ ]:
```
# let's preprocess our target variables by converting to binary output
# this is the same to what was done for the target in KNN analysis
le = LabelEncoder()

# target train
ytrain_knn = le.fit_transform(ytrain)
ytrain_knn = pd.DataFrame(ytrain_knn)

# target test
ytest_knn = le.fit_transform(ytest)
ytest_knn = pd.DataFrame(ytest_knn)
```

Decision Tree - Default

In [ ]:
```
# fit a default decision tree
clf_dt_default = tree.DecisionTreeClassifier()
clf_dt_default.fit(Xtrain, ytrain)
dt_default_pred = clf_dt_default.predict(Xtest)
```

In [ ]:
```
# confusion matrix to see performance
dt_default_cm = confusion_matrix(ytest, dt_default_pred)
disposition_labels = {'Candidate', 'False Positive'}
pd.DataFrame(dt_default_cm, index = disposition_labels, columns= dispositio
n_labels)
```

Out[ ]:

	False Positive	Candidate
False Positive	728	160
Candidate	188	673

In [ ]:
```
dt_def_acc = float(accuracy_score(actual_labels, dt_default_pred))*100
print('Accuracy:', float(accuracy_score(actual_labels, dt_default_pred))*10
0, '%')
```

```
Accuracy: 80.10291595197255 %
```

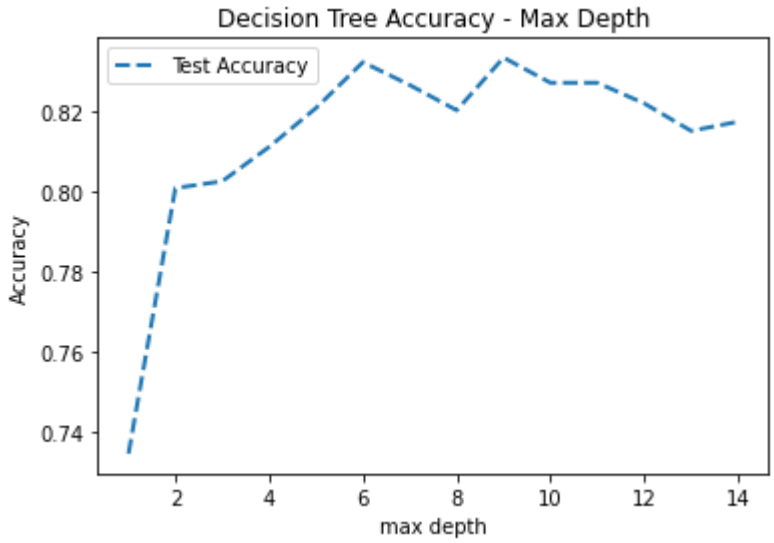
Decision Tree - Max Depth

In [ ]:
```
# let's fit a decision tree with optimal max depth
def dt(Xtrain, ytrain, Xtest, ytest, maxdepth):
  dt_accuracy = []
  clfs_dt = []
  for i in maxdepth:
    clf_dt_depth = tree.DecisionTreeClassifier(max_depth=i).fit(Xtrain, ytr
ain)
    dt_depth_pred = clf_dt_depth.predict(Xtest)
    clfs_dt.append(clf_dt_depth)
    dt_accuracy.append({'max depth': i,
    'Test Accuracy': accuracy_score(dt_depth_pred, ytest)})
  return pd.DataFrame(dt_accuracy), clfs_dt
```

In [ ]:
```
# let's see test accuracy with a range of max depth from 1-15
dt_depth_acc, dt_depth_clfs = dt(Xtrain, ytrain, Xtest, ytest, range(1,15))
display(dt_depth_acc)
```

	max depth	Test Accuracy
0	1	0.734706
1	2	0.801029
2	3	0.802744
3	4	0.811321
4	5	0.821041
5	6	0.832476
6	7	0.826758
7	8	0.820469
8	9	0.833619
9	10	0.827330
10	11	0.827330
11	12	0.822184
12	13	0.815323
13	14	0.817610

In [ ]:
```
# same thing, let's plot the test accuracy of varying max_depth values
plt.plot(dt_depth_acc['max depth'], dt_depth_acc['Test Accuracy'], '--', li
newidth=2, label='Test Accuracy')
plt.xlabel('max depth')
plt.ylabel('Accuracy')
plt.title('Decision Tree Accuracy - Max Depth')
plt.legend()
plt.show()
```



Let's choose a max_depth=6 to fit our decision tree model. After max_depth=6, any increase in layers seems to reduce accuracy, possibly due to overfitting.

In [ ]:
```
# fit a max depth decision tree
clf_dt_depth = tree.DecisionTreeClassifier(max_depth=6)
clf_dt_depth.fit(Xtrain, ytrain)
dt_depth_pred = clf_dt_depth.predict(Xtest)
```

In [ ]:
```
# confusion matrix to see performance
dt_depth_cm = confusion_matrix(ytest, dt_depth_pred)
disposition_labels = {'Candidate', 'False Positive'}
pd.DataFrame(dt_depth_cm, index = disposition_labels, columns= disposition_
labels)
```

Out[ ]:

	False Positive	Candidate
False Positive	811	77
Candidate	218	643

In [ ]:
```
maxdepth_acc = float(accuracy_score(actual_labels, dt_depth_pred))*100
print('Accuracy:', float(accuracy_score(actual_labels, dt_depth_pred))*100,
'%')
```

```
Accuracy: 83.13321898227558 %
```

Compared to our default decision tree, optimizing max_depth allowed our model to more accurately predict False Positives

Decision Tree - Cost Complexity Pruning

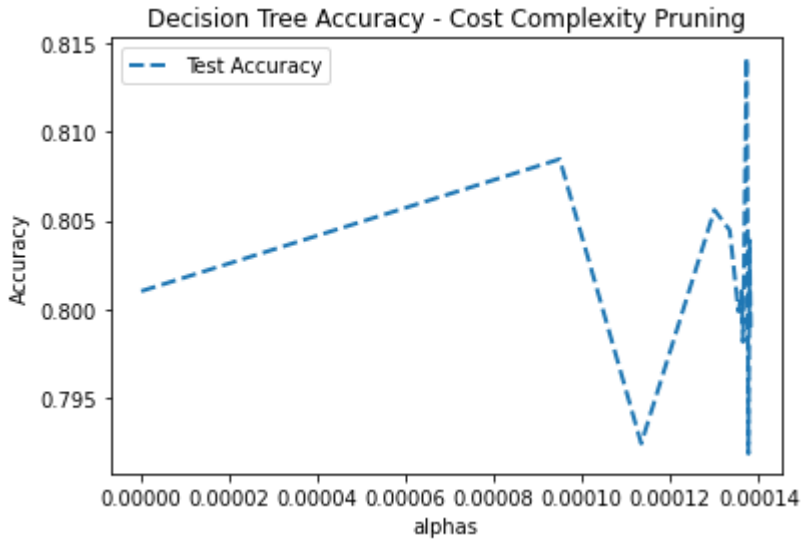
In [ ]:
```
# pruning will help to control the size of the tree w/o us using max_depth
# we'll define this based off of our default decision tree
path = clf_dt_default.cost_complexity_pruning_path(Xtrain, ytrain)
ccp_alphas = path.ccp_alphas
# higher alpha = more of the tree is pruned
```

In [ ]:
```
# let's plot the alphas
def dt_prune(Xtrain, ytrain, Xtest, ytest, ccp_alphas):
  dt_accuracy = []
  clfs_dt_pruned = []
  for i in ccp_alphas:
    clf_dt_pruned_tree = tree.DecisionTreeClassifier(ccp_alpha=i).fit(Xtrai
n, ytrain)
    dt_prune_pred = clf_dt_pruned_tree.predict(Xtest)
    clfs_dt_pruned.append(clf_dt_pruned_tree)
    dt_accuracy.append({'alphas': i,
    'Test Accuracy': accuracy_score(dt_prune_pred, ytest)})
  return pd.DataFrame(dt_accuracy).head(15), clfs_dt_pruned
```

In [ ]:
```
# let's see test accuracy with a range of alpha values
dt_prune_acc, dt_prune_clfs = dt_prune(Xtrain, ytrain, Xtest, ytest, ccp_al
phas)
display(dt_prune_acc)
```

	alphas	Test Accuracy
0	0.000000	0.801029
1	0.000095	0.808462
2	0.000113	0.792453
3	0.000130	0.805603
4	0.000133	0.804460
5	0.000135	0.799886
6	0.000136	0.801029
7	0.000136	0.798170
8	0.000137	0.803888
9	0.000137	0.814180
10	0.000137	0.806175
11	0.000138	0.791881
12	0.000138	0.803888
13	0.000138	0.801601
14	0.000138	0.798742

In [ ]:
```
# same thing, let's plot the test accuracy of varying alpha values
plt.plot(dt_prune_acc['alphas'], dt_prune_acc['Test Accuracy'], '--', linew
idth=2, label='Test Accuracy')
plt.xlabel('alphas')
plt.ylabel('Accuracy')
plt.title('Decision Tree Accuracy - Cost Complexity Pruning')
plt.legend()
plt.show()
```



ccp_alpha = 0.000135 gives the most accuracy before dropping

In [ ]:
```
# fit a ccp decision tree
clf_dt_prune = tree.DecisionTreeClassifier(ccp_alpha=0.000135)
clf_dt_prune.fit(Xtrain, ytrain)
dt_prune_pred = clf_dt_prune.predict(Xtest)
```

In [ ]:
```
# confusion matrix to see performance
dt_prune_cm = confusion_matrix(ytest, dt_prune_pred)
disposition_labels = {'Candidate', 'False Positive'}
pd.DataFrame(dt_prune_cm, index = disposition_labels, columns= disposition_
labels)
```

Out[ ]:

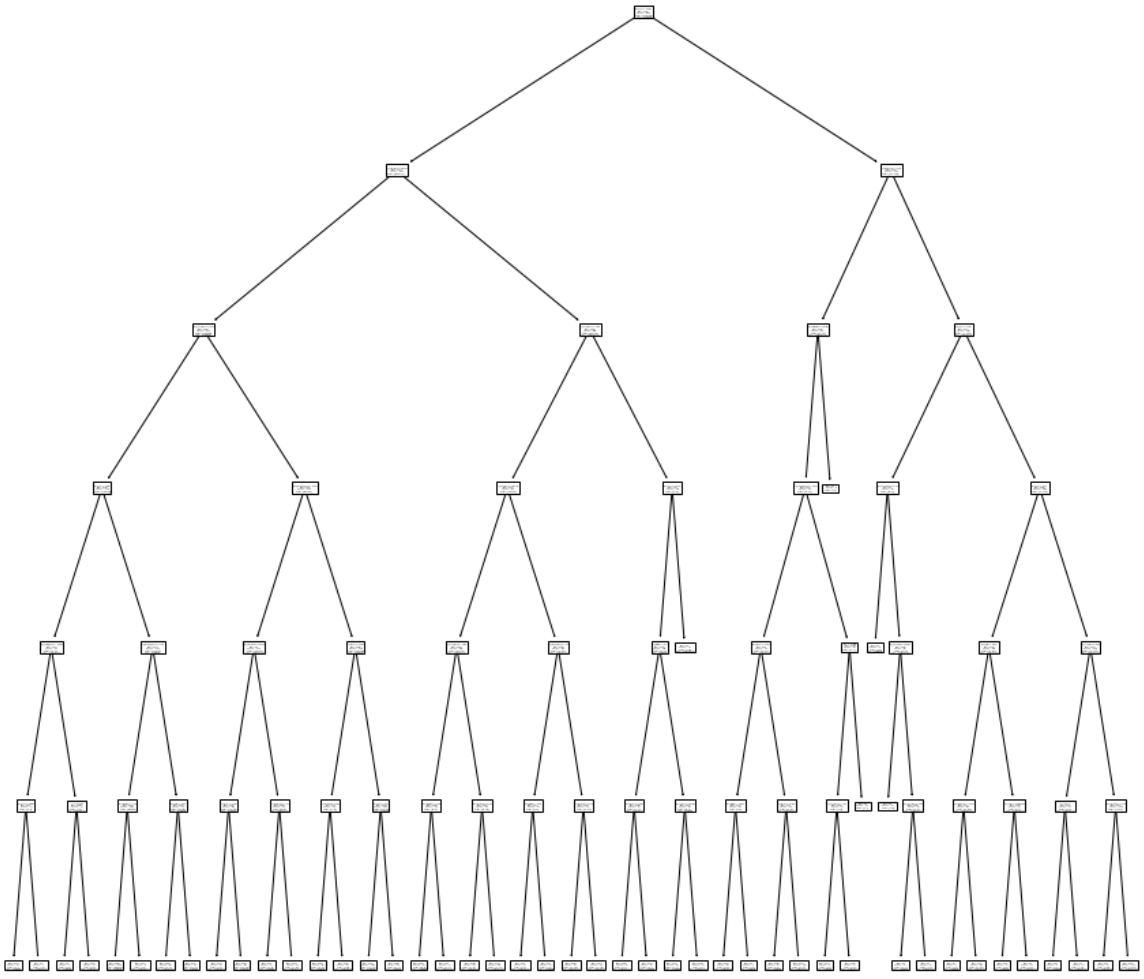
	False Positive	Candidate
False Positive	730	158
Candidate	187	674

In [ ]:
```
ccp_acc = float(accuracy_score(actual_labels, dt_prune_pred))*100
print('Accuracy:', float(accuracy_score(actual_labels, dt_prune_pred))*100,
'%')
```

```
Accuracy: 80.27444253859348 %
```

Amongst the decision trees, the one tuned using max_depth gave the model with the highest test accuracy.

In [ ]:
```
# plot the max_depth decision tree
fig,ax = plt.subplots(figsize = (15,15))
treeplot = tree.plot_tree(clf_dt_depth, feature_names=list(X.columns), clas
s_names = ['Candidate', 'False Positive'], ax=ax)
```



PCA on Random Forest

In [ ]:
```
# first, let's normalize our variables
sc = StandardScaler()
X_train = sc.fit_transform(Xtrain)
X_test = sc.transform(Xtest)
```

In [ ]:
```
# perform PCA
pca = PCA()
X_train = pca.fit_transform(X_train)
X_test = pca.transform(X_test)
```

In [ ]:
```
# let's see how much of the variance is explained
explained_var = pca.explained_variance_ratio_
```

In [ ]:
```
# let's use random forest
clf_PCA_rf = RandomForestClassifier()
clf_PCA_rf.fit(X_train, ytrain)
PCA_rf_pred = clf_PCA_rf.predict(X_test)
```

In [ ]:
```
# confusion matrix to see performance
PCA_rf_cm = confusion_matrix(ytest, PCA_rf_pred)
disposition_labels = {'Candidate', 'False Positive'}
pd.DataFrame(PCA_rf_cm, index = disposition_labels, columns= disposition_la
bels)
```

Out[ ]:

	False Positive	Candidate
False Positive	749	139
Candidate	130	731

In [ ]:
```
pca_acc = float(accuracy_score(actual_labels, PCA_rf_pred))*100
print('Accuracy:', float(accuracy_score(actual_labels, PCA_rf_pred))*100,
'%')
```

```
Accuracy: 84.61978273299027 %
```

Model Analysis

In [ ]:
```
print('Accuracy of Perceptron is:', perc_acc, '%')
print('Accuracy of Logistic Regression is:', logreg_acc, '%')
print('Accuracy of SVM is:', SVM_acc, '%')
print('Accuracy of SGD is:', SGD_acc, '%')
print('Accuracy of KNN Using Euclidean Distance is:', euc_acc, '%')
print('Accuracy of KNN Using Manhattan Distance is:', man_acc, '%')
print('Accuracy of KNN Using Cosine Distance is:', cos_acc, '%')
print('Accuracy of the Default Decision Tree is:', dt_def_acc, '%')
print('Accuracy of Decision Tree with Optimized Max Depth is:', maxdepth_ac
c, '%')
print('Accuracy of CCP Decision Tree is:', ccp_acc, '%')
print('Accuracy of Random Forest Using PCA is:', pca_acc, '%')
```

```
Accuracy of Perceptron is: 68.55345911949685 %
Accuracy of Logistic Regression is: 81.76100628930818 %
Accuracy of SVM is: 85.07718696397941 %
Accuracy of SGD is: 83.30474556889651 %
Accuracy of KNN Using Euclidean Distance is: 78.15894797026873 %
Accuracy of KNN Using Manhattan Distance is: 79.53116066323614 %
Accuracy of KNN Using Cosine Distance is: 78.61635220125787 %
Accuracy of the Default Decision Tree is: 80.10291595197255 %
Accuracy of Decision Tree with Optimized Max Depth is: 83.13321898227558 %
Accuracy of CCP Decision Tree is: 80.27444253859348 %
Accuracy of Random Forest Using PCA is: 84.61978273299027 %
```

In [ ]: