Malware Detection and Prediction: Embedded URLs

Nick Alonso | Math 440: Statistical Learning 1
College of Charleston

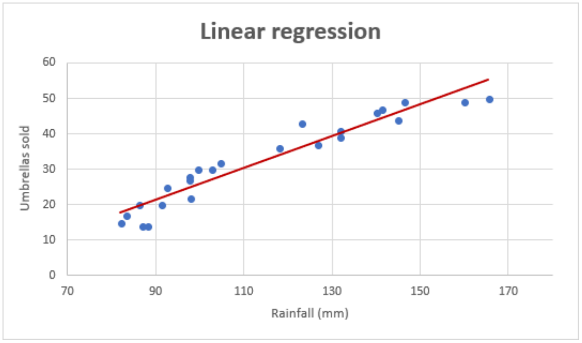
Abstract

Email has become one of the most popular methods of communication, especially in the business world. With the popularity of email, also arises the popularity of using email as a platform to embed malware by a method called *Phishing*. The idea is that a user will receive an email that persuades them to click on an embedded URL, which then directs them to malicious content. The primary goal of this project was to train a model that can predict whether a URL is malicious or legitimate. We accomplished this by implementing three different statistical models comparing their performance when predicting if a URL is safe or malicious.

Statistical Analysis Techniques

Regression

Throughout the semester we were exposed to various regression models that could be used for both regression or classification problems. These algorithms included simple linear regression, multiple linear regression and the regression variation that we gained the most experience with this semester: logistic regression. In general, the goal of a regression algorithm in a supervised learning scenario, is to predict a quantitative response Y from a predictor variable X. Linear regression can be fit to linear and non-linear equations. An easy example of explaining linear regression would be using regression to predict the mpg a car gets. The mpg a car gets (response) we can be predicted based off a set of the features (predictors) such as weight, length, height, engine size etc. For instance, given a data set of 100 cars, with a Y vector for mpg, and a X vector for the features as mentioned earlier, we could map observations and fit a regression line through the observations using the least squares method. The least squares method minimizes the sum of squares of the residuals. In short, this minimizes the mean vertical distance between the observations and the regression line. Another example that can be used to depict this relationship is trying to predict the number of umbrellas sold based off of rainfall measured in millimeters. Here, the response would be the number of umbrellas sold and the predictor would be rainfall in in millimeters. If we took a data set of 24 days that it rained and a y and x vector, we could map the observations and fit a regression model based on the least squares method.



Here, the red line represents the regression line and each blue point represents an observation. In this example there appears to be a strong linear relationship between the two variables.

Classification

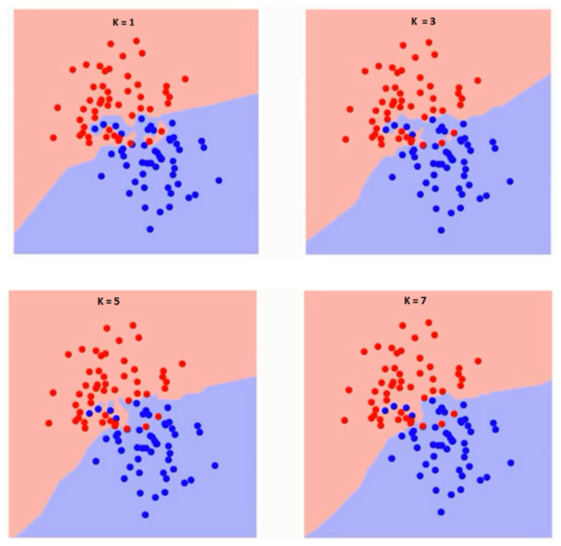
A classification problem differs from a regression problem in that it is used to predict a qualitative variable, labeling the output based off of features that categorize observations. A classification problem can however be casted as a regression problem by assigning integers to each class label. For example, if we wanted to predict the gender of someone, we could assign the integer 1 – male and 0- female. An example of a classification problem would be predicting the probability that a person develops lung cancer based off of categorical, binarized predictors such as “Do you smoke? Yes or no”, “Gender: Male or Female”, “Do you Drink? Yes or no”. The purpose is to cluster groupings of features and classify new observations to the correct cluster based on similar features. Various classification algorithms were discussed this semester including: Logistic Regression, Support Vector Machines, Naïve Bayes Classification, Decision Trees, Boosted Trees, Random Forests, K-Nearest Neighbor, and Neural Networks. As mentioned earlier, regression can be used in classification problems. Logistic Regression is based off of the same family of regression algorithms. The major assumptions of the logistic regression model are that there are no outliers in the data, no multicollinearity among the predictors, and that the dependent variable should be dichotomous.

Naïve Bayes Classifier

The Naïve Bayes Classifier is based on the Bayes’ Theorem with a strong (naïve) assumption that all predictors are independent of one another. In short, the presence of one feature in a class label is independent to the presence of any other predictors. Even in a case where there is a dependency between features, all of the features independently contribute to the overall probability. This algorithm is very useful for large data sets. It is one of the highest performing classification models when implemented on a large set.

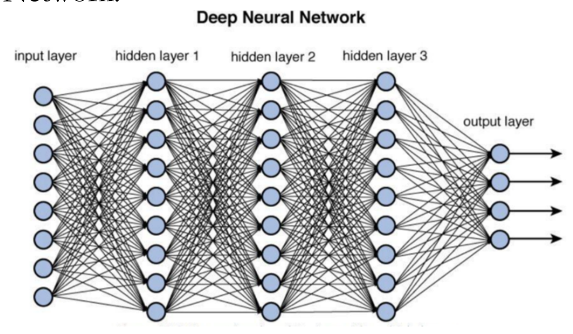
K-Nearest Neighbor

The K-Nearest Neighbor algorithm is typically used in a supervised learning problem. Given a data set with many labeled observations, the model learns how to label new observations based off of the clusters formed. The model looks at the labeled points closest to the new observation and makes the appropriate classification based on the grouping. Here, K represents the number of neighboring points the model will check. As the value of K is increased, the decision boundary visually becomes smoother providing legitimate predictions. When K = 1 the data is being overfitted and any inference gained from this model would be pretty unsubstantial.



Neural Networks

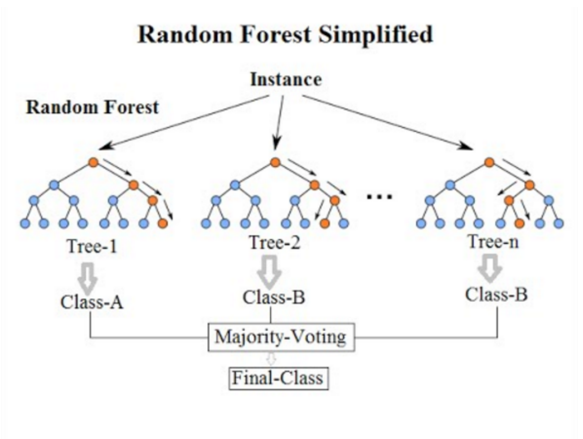
A Neural Network is one of the more advanced classification algorithms. It is modeled after the human brain, containing millions of processing nodes that are intertwined. Data typically moves through layers of these nodes in a process known as “feed forward”. A node will assign a weight to each of the incoming connections. When a node receives data, it multiplies it by its weight then adds the products together. If the sum is greater than the threshold the node will “fire” which simply means that the node sends the data to all of its outgoing connections. The input layer receives data and calculates the sum of weights. If the sum exceeds the threshold the data is passed to the hidden layer through all outgoing paths. The hidden layer then makes calculations and makes the appropriate decisions to generate classifications. This figure shows the typical work flow of a Neural Network.



Neural nets are being used in very exciting ways in the deep learning field such as speech recognition. It is a highly sophisticated form of artificial intelligence that has been used to make very accurate predictions. In an image recognition instance, the model may be trained on thousands of images labeled as car, cup, tool, house, street etc. and learn visual patterns associated with each label. This is done by mapping the sum of weights to a label. A very intriguing question remains: how does a neural net learn and make classifications. In the image recognition example, the model can associate and pick up on certain features of a car or patterns and assign weights and base its classifications off of this technique. The question still remains: which patterns the neural net is actually looking at and how does it piece these features together to make legitimate, substantial predictions. This is still a developing research field.

Random Forest

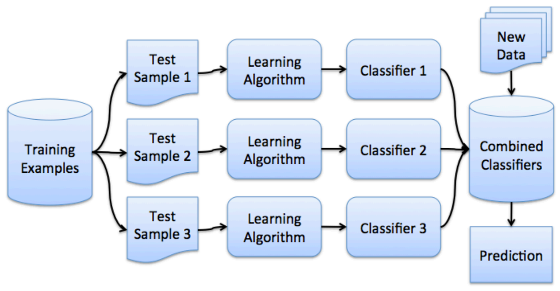
Random forest is a great alternative to the decision tree model. It is a highly flexible algorithm that produces accurate results even in the absence of hyperparameter tuning. The random forest method will typically correct a standard decision trees habit of overfitting the data. A "forest" works by constructing multiple decision trees then merges them together selecting the most common label to output. Typically, a random forest will have the same hyperparameters as a standard decision tree or bagging model. As the name states, it will add a higher degree of randomness to the model to negate a decision tree's habit of overfitting the data. Rather than search for the most useful feature when splitting the node, it looks for the most important feature from a random subset of features, creating a more diverse efficient model. To summarize the work flow of the random forest algorithm, multiple decision trees are constructed by taking a random subset of features into consideration at each node, then merging them back together to provide the highest probable class label of the observation.



A good analogy we found to describe this concept uses a real-life situation and applies the random forest algorithm. If we wanted to decide on a location to take a vacation, we would ask people that know us for advice. We would go to the first person, say, a family member, and ask them where we should go. They would ask us where we have traveled in the past and how much we enjoyed those locations. Based on our answers, the family member would give us advice on where to take our next trip. At this stage, we have used a typical decision tree approach to plan our trip. We can take it a step further by asking more and more family members and friends to guide us on where to take our trip. After several family members and friends have given us an answer, we would then select the most common recommendation. This is a typical random forest approach applied in a real-life scenario.

Resampling Methods: Bagging Boosting and the Bootstrap Technique

Resampling is a set of techniques that utilizes repeated samples from the original data set, to calculate a test statistic. It can be very useful when working with a small data set. The draw back to this approach is computation expense. These methods can be very expensive due to fitting the same model to the training data multiple times using different subsets of the data. Bootstrap is a very common method usually used to calculate the accuracy of a model or parameter estimate. The bootstrap creates a statistic by replacing a lack of data with random samples from the original data set. The primary advantage here is that if a user is working with a smaller data set, they can add as many samples as desired to their training set. The disadvantage is that it does create inherent bias by repeatedly filling the training set with copies of the same data. It is really only reasonable to apply this technique to a data set larger than 30 observations. Any set less than 30 would provide useless results.



Bagging is a procedure used to reduce the variance of a model. This comes from a classical statistic theorem that the averaging of a set of observations will reduce the variance in the set. This is a great concept but not practical to implement alone for the fact that typically a user does not have access to multiple datasets. The answer to this problem is to apply the bootstrap method which is where then name is derived from: bootstrap aggregation (bagging). Bagging can dramatically improve the performance of decision trees. It can also be applied to other models such as Logistic Regression. Boosting is an additional approach used to improve the accuracy of a decision tree. It can also be applied to many other algorithms in a regression or classification instance. We will explain boosting by describing the application of this technique with the random forest algorithm. Boosting works in a very similar way to bagging aside from the fact that each tree is grown sequentially using information from the previous tree. Boosting also differs from bagging in that it does not use the bootstrap method but rather fits each tree on a modified version of the original data set.

Unsupervised Learning Algorithms

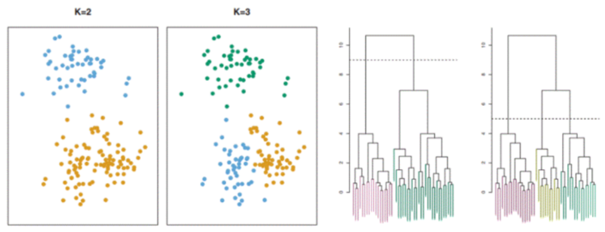
Up until this point, all of the machine learning algorithms we have discussed have been applied in a supervised learning setting, meaning, the goal of the model is to predict a response Y with selected predictors X. Unsupervised learning however is not given a response. Unsupervised learning is a form of exploratory data analysis with the goal of extracting subgroups of the data and relationships between them, providing insight on how to best analyze the information. The most common form of unsupervised learning is clustering. Clustering is a way to represent a large number of observations in easily identifiable sub groups.

K-Means Clustering

K-Means clustering is a common unsupervised learning algorithm used to distinguish sub groups of observations. When applying this algorithm, the user first needs to establish how many clusters K should be generated. The model will then assign each observation to a cluster. The algorithm randomly assigns a number from 1 to K to each observation. These act as the initial cluster assignments for the observations. The algorithm then iterates until the cluster assignments stop changing. During each iteration, the model computes the cluster centroid of each cluster. It then assigns each observation to the cluster whose centroid is the closest.

Hierarchical Clustering

Hierarchical Clustering is another approach for distinguishing subgroups of a data set. Unlike K-Means Clustering, this approach does not require a predetermined K number of groups. The algorithm begins by evaluating n observations and the Euclidean distance of all pairwise dissimilarities. It treats each observation as its own cluster. It then examines all pairwise inter-cluster dissimilarities among clusters and selects the pair of clusters that are the least dissimilar. It then merges the two clusters. The model will then iteratively repeat this process, computing the new pairwise inter-cluster dissimilarities among the remaining clusters.



*A simple illustration showing an example of a K-Means Clustering method on the left, and a Hierarchical Dendrogram on the right

Statistical Analysis Used in Cybersecurity

All of the statistical methods discussed so far also have a wide variety of uses in the cybersecurity field. Common problems include: Network Security Problems, Phishing, Social Network Security, Fake News, Social Engineering, Malware Detection, Beaconing, Botnet Detection, and Real Time Classification.

Phishing

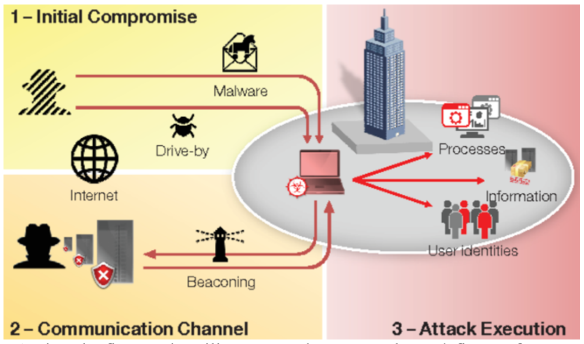
This semester, we became the most familiar with phishing, mainly due to our semester project. Phishing is defined as a fraudulent attempt to obtain user-sensitive information including but not limited to usernames, passwords, social security numbers and financial information. This is typically carried out through electronic communication by disguising the attack to resemble a reputable entity. We specifically focused on phishing attempts that were embedded in emails. For example: the victim may receive an email stating they have been selected to receive an all-expense paid trip to the Cayman Islands, prompting them to click an embedded URL to enter their information and collect their prize. When the user accesses the link, they are then redirected to malicious content. This problem is typically addressed through classification models using text feature extraction and classification to identify dangerous URLs.

Fake News

Fake news has been an increasingly common problem recently, especially with the increased popularity of social networks. Social media has become one of the primary sources of news in the United States. Through this platform, fake news is able to be rapidly distributed and manipulate the social network and its users. Fake news primarily succeeds by targeting bias, specifically, cognitive, social and machine bias. Cognitive bias is created by the unique way a person's brain processes information. If the brain is exposed to too many stimuli it can become overloaded. This is very prevalent with social media due to the highly populated news feeds on the majority of social networks. To negate this overload, the brain relies on cognitive shortcuts to determine where to devote its finite attention. People are easily affected by emotional implications of a news headline, regardless of the article's accuracy. This is a standard technique to target a user with fake news.

People also develop bias from culture and society. On a social media platform, a user selects their friends based off of social biases, which in turn, affects the information they are exposed to. It has been shown that people will engage in content more frequently if it comes from within their social circle. This is another major method of targeting a user with fake news. If the attacker can disseminate information through a user's social circle, they will more likely engage the content and fall susceptible to the attack. The third bias fake news targets are bias of the machine. Social media platforms have algorithms that are used to generate personalized content that each user will see, which in turn, reinforces social and cognitive bias. An example of this is YouTube's built in advertising tools that tailor advertisements to individual users who will be more inclined to believe them. These biases are typically carried out by social bots, disseminating information on platforms such as Twitter or Facebook.

Beaconing
Beaconing is another common method to distribute malware. When a computer becomes infected, it typically beacons to a command server. The malware is able to communicate with the control server and receive further instructions. Beaconing can be especially difficult to detect because it can happen at any time or frequency ranging from seconds to weeks.



*A simple figure that illustrates the general workflow of a malicious attack through beaconing

Implementation: Statistical Methods Used in Cybersecurity

We have been exposed to various algorithmic techniques that can be used to address cybersecurity problems this semester. Classification techniques are most commonly used to combat cybersecurity threats. We gained the most experience from working on our project which implemented a Logistic Regression, Support Vector Machine and Decision Tree algorithm for text classification.

Additional algorithms that deal with various forms of cyber security threats include: Naïve Bayes, Random Forest, Neural Networks, reinforcement learning, real-time classification to combat zero-day threats, K-Nearest Neighbor and unsupervised clustering techniques such as K-Means Clustering. Several examples of these algorithms and the problems they tackle are: Naïve Bayes and Principle Component Analysis for the use of intrusion detection, using Support Vector Machine models to detect DDoS (Distributed Denial of Service) attacks, KNN classification to detect host-based anomalies, and Random Forest models for Botnet detection. The majority of the algorithms are evaluated using true positive rates and false positive rates, precision and accuracy scores, true detection rates, error rates and recall.

Using Statistical Models in a Data Driven Machine Learning Problem

As stated earlier, we chose to work with three statistical models: Logistic Regression, Support Vector Machine, and Decision Trees. The project was done in Python using a variety of imported statistical and graphical packages with the majority of our work using the packages scikit-learn and matplotlib

Dataset Description
We found the data set on GitHub and credit faizann24 for the data. He created a crawler that scanned malicious links off of multiple websites, gathering a total of 400,000 URLs, with 80,000 being flagged malicious. We downloaded it off of GitHub, loaded the data set into python by reading the csv file and converting it to a data frame, then stored it in

a list. It was relatively easy to import data into python using the pandas package and this data set seems to be very clean and easy to work with.

Feature Selection: Term Frequency-Inverse Document Frequency

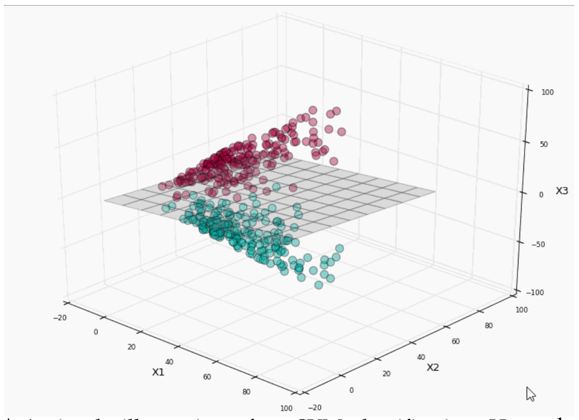
For our feature selection process, we used a function called Term Frequency-Inverse Document Frequency (tf-idf). This is a numerical method that analyzes how important a word is to a document in a corpus. In order to apply this method to a data set of URLs there were several steps that needed to be taken beforehand. The first sub problem we faced was to split each URL up into individual words defined as *tokens,* through a process called *tokenization*. To do this we created a pipeline that split our URLs on three different characters: periods, dashes, and slashes. After we had our tokens, we then removed any duplicates and discarded the word "*com",* as that is a pretty standard segment in typical URLs. Once the URLs were tokenized and cleaned, we were then able to apply the tf-idf function to create our vectors. The tf-idf approach works by scanning the vectors of tokens and identifying the most common tokens associated with each class by assigning a tf-idf score, based on the frequency of the individual token in the data set. In this case, we had two classes: good and bad. Good represented a legitimate URL and bad represented a phishing attempt. We decided to select and analyze the top 15 features from each class and were able to output each feature's tf-idf score and their relevance when compared to other features.

First Model: Logistic Regression

The first model we chose to train uses Logistic Regression. The major assumptions of the logistic regression model are: there are no outliers in the data, no multicollinearity among the predictors, and that the dependent variable should be dichotomous. With all assumptions being met, we created a pipeline that preprocessed the training vectors. We were able to fit the Logistic Regression model on our training features and make very adequate predictions. When applied to the test set, the model produced a 94.6% accuracy score. The model was able to successfully predict whether each URL was legitimate or fake, labeling them "good" or "bad".

Second Model: Support Vector Machine (SVM)

The second model we chose to build used the Support Vector Machine (SVM) algorithm. The SVM model is a supervised learning technique. It is possible to use SVM for both classification and regression problems but is most typically used for classification. The goal of an SVM is to find a hyperplane that best divides a dataset into two classes. The "support vectors" are the observations closet to the hyperplane. If the support vector points were to be removed, the hyperplane's position would move respectively. A hyperplane is most simply defined as a line that linearly separates and labels the observations. In general, the further a data point lies from the hyperplane, the more likely it is that the data point has been correctly labeled. A key focus when using SVMs is finding the right hyperplane that separates the data as accurately as possible. A way to do this is to try and maximize the margin and any data point in the given set. A margin is the distance between the hyperplane and the closest data point from either set. In a perfect situation the observations lie in clear clusters that can be easily classified. This however is typically not the case. Data sets usually will have observations that are scattered in a non-linear fashion making it impossible to determine an accurate hyperplane position. In this case, it is important to move from the $\mathbb{R}^2$ to $\mathbb{R}^3$dimensional space. The method of mapping a set of points from the $\mathbb{R}^2$ to $\mathbb{R}^3$ space is called Kernelling. When working in the $\mathbb{R}^3$ space, the hyperplane will no longer be a line but instead an actual plane.



* A simple illustration of an SVM classification. Here, the observations have been mapped to the $\mathbb{R}^3$dimensional space and separated by the most accurately classifying hyperplane.

In this instance the only difficulty when working with the SVM model was the computing time. We extracted 10% of the data set (40,000 URLs) to train and test this model and it took 124 minutes to produce output. For this reason, it is a much more effective algorithm on smaller data sets.

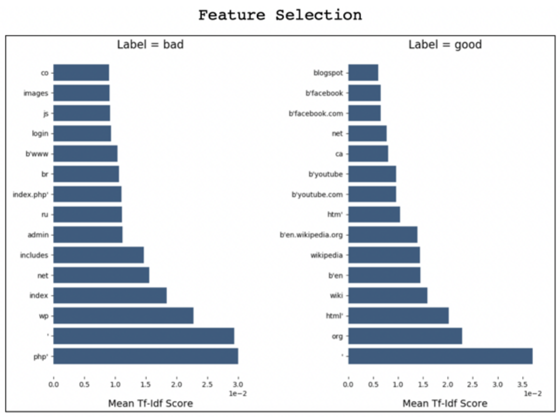
Third Model: Decision Tree

For our final classifier We chose to use a Decision Tree model. As the name sounds, a Decision Tree algorithm makes classifications off of a tree structure. The idea is to break down a data set into smaller subsets until a tree comprised of decision nodes and leaf nodes has been created. A decision node is made up of two or more branches and a leaf node represents the classifying label. These all branch from a root node which is the most useful predictor in the tree. Trees are generally non-robust. A Small change of the training data can lead to a large change in the decision tree and skew the predictions. Discovering an optimal decision tree is currently an NP-complete problem. Decision tree learning algorithms usually are based on the greedy algorithm or some other heuristics where the best decision is made at each node. Cross-Validation can be used to select the best hyperparameters in order to make an approximation of the most optimal instance of a decision tree for the given problem. An example of a set of decision tree parameters would be the max depth of a tree and minimum leaves. Using Grid-Search Cross Validation we can test different combinations of parameters searching for the best pair.

Model Analysis and Comparison

We compared our models with the use of a Receiver Operating Characteristics Curve (ROC) and Grid Search Cross Validation. First, we created a pipeline for each of our models to streamline the preprocessing phase. We then defined a parameter range from [1-10] and a range for our float values of [1.0, .5, 0.1]. Once the parameter range was established, we defined an individual set of parameters for each model to be used in our grid search cross validation. We used 10 folds for each model and selected the best instance of each. For example: The Grid Search Cross-Validation technique was very useful for our decision tree model. We were able to approximate the optimal maximum tree depth and minimum sample leaves that would produce the highest accuracy when predicting. We then displayed the best set of parameters that were selected for each model in the output. In order to implement this, we created a grid search list with each model's defined grid search parameters. This created an easy to access group of variables to iterate through. We then defined a variable to hold the highest accuracy score and tested each model, comparing each model's accuracy against the others. We appended the higher score to the best accuracy score variable and displayed the most efficient model in the output.

We also compared our models using a Receiver Operating Characteristics Curve (ROC). A ROC curve is defined as a graphical plot that illustrates the diagnostic ability of a binary classifier as its discrimination threshold is varied. A ROC curve is created by plotting the true positive rate against the false positive rate at various threshold settings. The area under the curve (AOC) represents the accuracy of each model's predictions. To create this curve, we had to do several things with the data first. The initial classification of each URL fell under the labels "good" or "bad". We had to convert these labels to a binary output in order to manipulate it in our ROC method. We used a built in sklearn function called preprocessing that transformed our target vector and our predictions vector from "good, bad" to "0, 1". Once the labels were converted, we were then able to compare the vectors and analyze each model's accuracy. In addition to the ROC curve, we also imported a function from sklearn.metrics called accuracy_score. As the name sounds, this function produces a score which represents the accuracy with which the model predicted.



We generated a features bar graph in pycharm using the imported package matplolib and a function from matplotlib called pyplot. This graph displays the features associated with each class label. In this case each feature was a token that frequently appeared in the data set of URLs. We defined a separate method that would generate the plot once it received the necessary values. To do this We had to develop three separate methods to specify which features we would be passing in. The first method sorts the features by the top tf-idf scores and selects the top 15 to work with. our second method returns the top 15 features that on average, are the most important among documents identified by indices. our third method returns a list, in which each index holds a feature and the associated tf-idf score for the top 15 features from the data set. Once we obtained the values, we displayed the feature list of good and bad URLs, side by side for comparison purposes.

Conclusion

Overall, this project proved to be very successful. It was a pretty standard supervised learning classification problem with the goal of predicting a response based off of a set of features. The generalized work flow of the project passes a group of URLs into a feature vector, splits the feature vector into a training and test set, picks an optimized instance of a model through Grid Search Cross Validation and fits the model on the training set. It then makes predictions on the test set labeling each URL appropriately and is compared against the other models. In the end, the SVM model outperformed the others producing a 96% accuracy score when predicting whether a URL was malicious or legitimate. The results were satisfactory, but the accuracy of the models could be increased if we were able take advantage of a larger portion of our data set. When we started this project, we initially had to take a .0001% sample of the data set for our models to generate output. With the implementation of several scalable pipelines we were able to increase our capacity including .1% of the dataset which comes out to a little over 40,000 URLs. Ideally, we would like to include the entire dataset. This can be accomplished in the future by using web services to run our script on a Spark API that integrates scikit-learn in order to parallelize our processing. We have already formatted the code to be compatible with a parallel processing, Cross-Validation and model selection platform. We just need to implement our project on a spark cluster in an appropriate environment with the necessary built in integrated packages. While working on this project we were able to gain experience finding and loading a data set into a data frame, as well as cleaning and preprocessing it to be worked with. We were able to transform and manipulate the data by tokenizing each URL, binarizing the class label output, constructing graphs and ROC curve's and formatting predictive output to appear clear and interpretable to the user. We learned how to create saleable pipelines that streamlined our model selection process while at the same time would allow our model selection and cross-validation to be parallel processed for faster computation.

This project also broadened our knowledge of python forcing us to work with multiple imported libraries we have never seen before and forced us to figure out how to make them work for our data on a tight timeline. We received a lot of exposure to the wide variety of statistical learning tools and gained experience using them in a machine learning approach to a very popular cyber security problem. All of these skills are crucial as a data scientist and this project gave us a much better understanding of the fundamentals of the statistical models we researched. Moving forward we would like to implement this project in an Apache Spark cluster on a high-powered AWS EC2 instance. This would not only give us a massive increase in CPU and RAM space, but also allow us to use the spark-scikit learn integration package to parallelize our Grid Search Cross-Validation and model selection. We would also like to construct a neural net model over the next semester and integrate the model into our current comparisons. Finally, we would like to expand on our graphical summary of the project by building a 3-dimensional figure that visually illustrates how our SVM model makes classifications using a hyperplane to separate observations in the $\mathbb{R}^3$ dimension.

References

[1] Ma, Justin, Lawrence K. Saul, Stefan Savage, Geoffrey Voelker, et al. Beyond Blacklists: Learning to Detect Malicious Web Sites from Suspicious *URL*

[2] Hang, Huajun, Liang Qian, Yaojun Wang, et al. "An SVM-Based Technique to Detect Phishing URL's." Information Technology Journal, 2012

[3] "Support Vector Machines for Machine Learning." Machine Learning Mastery, 22 Sept. 2016, machinelearningmastery.com/support-vector-machines-for-machine-learning/.

[4] Donges, N. (2018, February 22). The Random Forest Algorithm – Towards Data Science. Retrieved from https://towardsdatascience.com/the-random-forest-algorithm-d457d499ffcd

[5] These are the three types of bias that explain all the fake news, pseudoscience, and other junk in your News Feed. (n.d.). Retrieved from http://www.niemanlab.org/2018/06/these-are-the-three-types-of-bias-that-explain-all-the-fake-news-pseudoscience-and-other-junk-in-your-news-feed/

[6] Bagging and Random Forest Ensemble Algorithms for Machine Learning. (2016, September 22). Retrieved from https://machinelearningmastery.com/bagging-and-random-forest-ensemble-algorithms-for-machine-learning/

Technical Manual: Phishing Net

PyCharm | Python 3.7 | Conda VirtualEnv

- Data Analysis
- Machine Learning Algorithms
- Performance Analysis

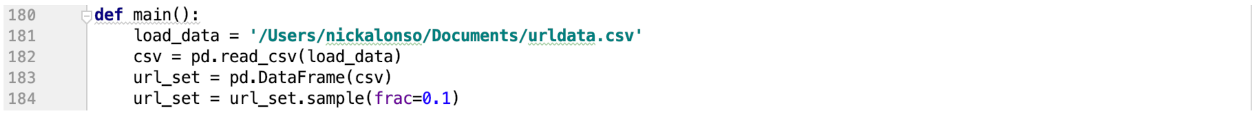
* All packages required for this script to execute have been installed locally. Depending on the environment and machine being used the following packages may need to be installed prior to compilation:

```python
from __future__ import print_function
import warnings
warnings.filterwarnings('ignore')
from sklearn.model_selection import train_test_split
import matplotlib
matplotlib.use("TkAgg")
from sklearn.linear_model import LogisticRegression
import numpy as np
from sklearn.model_selection import GridSearchCV
import pandas as pd
from sklearn.feature_extraction.text import TfidfVectorizer
from sklearn import svm
from sklearn import tree
import matplotlib.pyplot as plt
from sklearn.metrics import accuracy_score
from sklearn import preprocessing
from sklearn.pipeline import Pipeline
from sklearn.metrics import roc_curve, auc
```

Data Analysis

Loading the Data Set

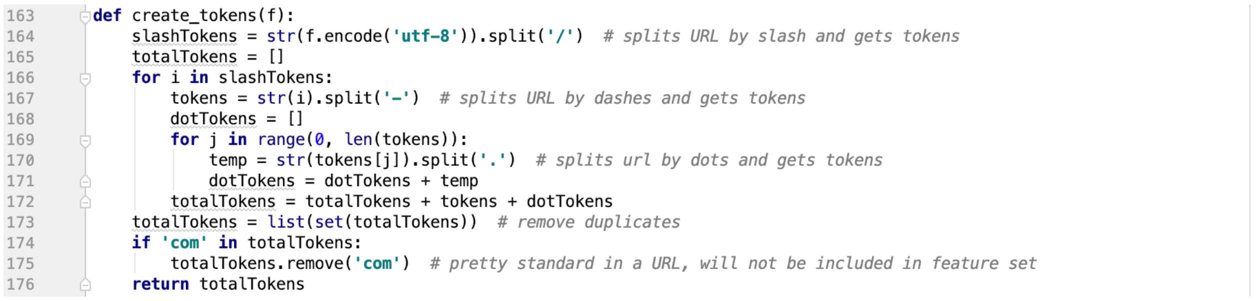
The data set used in this project is a Comma Separated Value (CSV) file pulled off of GitHub and contributed by faizann24. The data was loaded into a dataframe using the pandas package. In order to load this file locally download the included csv file titled "urldata.csv". Once downloaded, copy and paste the directory path to the variable `load_data = "…"`. The next step is converting the data to a dataframe using the `pandas` function that is already imported. Finally set a sample amount of the data to use, in this case 10% of the dataset (40,000 URLs) is manageable on most local computers, anything greater will require higher CPU and RAM storage when implementing the models. Once this is complete the data has been loaded.



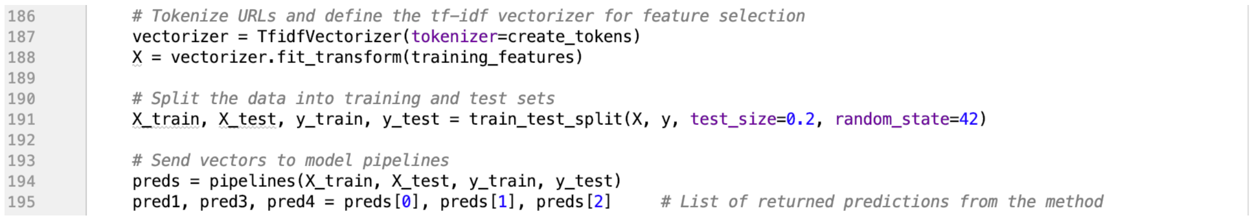
```python
def main():
    load_data = '/Users/nickalonso/Documents/urldata.csv'
    csv = pd.read_csv(load_data)
    url_set = pd.DataFrame(csv)
    url_set = url_set.sample(frac=0.1)
```

Transforming the Data Set

In order for this Data Set to be manipulated several things need to happen first. The URLs need to be broken down into individual words or phrases in order to apply a text classification feature extraction model. This is a process called `Tokenization.` The method titled `create_tokens()` handles this. Once the URLs have been broken down into individual words the entire data set is split into training and test sets using an imported function called `train_test_split.` You can now send the training and test sets to the model selection pipeline which is a method called `pipelines()`.



```python
def create_tokens(f):
    slashTokens = str(f.encode('utf-8')).split('/')  # splits URL by slash and gets tokens
    totalTokens = []
    for i in slashTokens:
        tokens = str(i).split('-')  # splits URL by dashes and gets tokens
        dotTokens = []
        for j in range(0, len(tokens)):
            temp = str(tokens[j]).split('.')  # splits url by dots and gets tokens
            dotTokens = dotTokens + temp
        totalTokens = totalTokens + tokens + dotTokens
    totalTokens = list(set(totalTokens))  # remove duplicates
    if 'com' in totalTokens:
        totalTokens.remove('com')  # pretty standard in a URL, will not be included in feature set
    return totalTokens
```

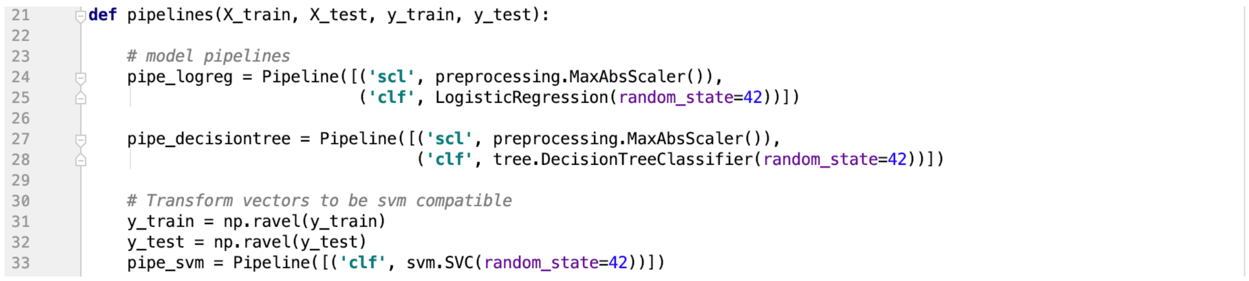


```
    # Tokenize URLs and define the tf-idf vectorizer for feature selection
    vectorizer = TfidfVectorizer(tokenizer=create_tokens)
    X = vectorizer.fit_transform(training_features)

    # Split the data into training and test sets
    X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2, random_state=42)

    # Send vectors to model pipelines
    preds = pipelines(X_train, X_test, y_train, y_test)
    pred1, pred3, pred4 = preds[0], preds[1], preds[2]     # List of returned predictions from the method
```

Machine Learning Algorithms

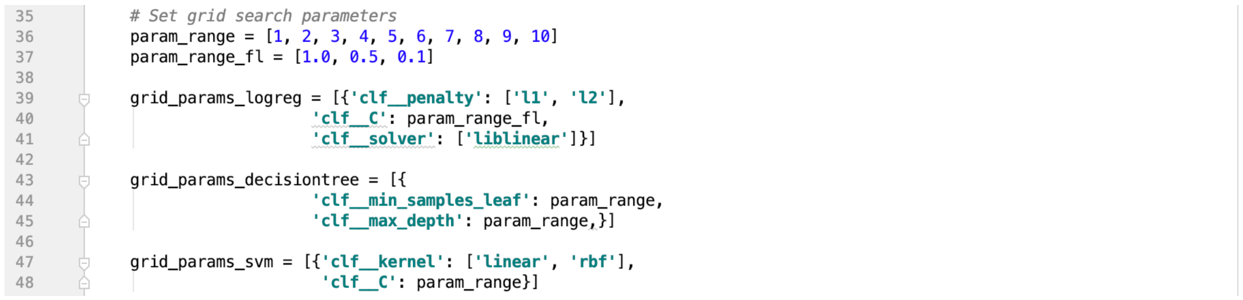
This project compares three different Machine Learning Algorithms: `Logistic Regression`, `Support Vector Machine`, and `Decision Tree`. To implement these models, we created a scalable pipeline for each model to support parallel processing using imported functions called `pipeline` and `preprocessing`. Additionally, we optimized each model using 10-fold Grid Search Cross-Validation by selecting the most useful hyperparameters. This was done using an imported function called `GridSearchCV`. For the parameters, we set a range and a float value range then went on to define the parameters.

Machine learning algorithm pipelines:



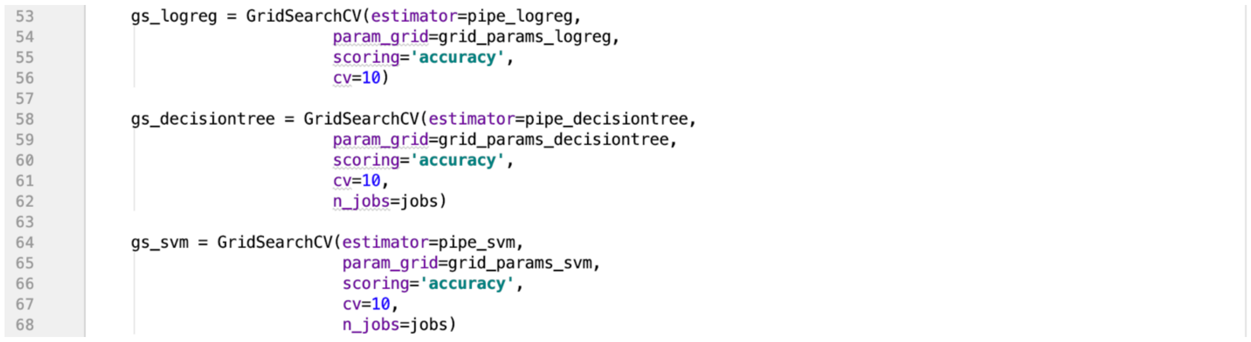
```
def pipelines(X_train, X_test, y_train, y_test):

    # model pipelines
    pipe_logreg = Pipeline([('scl', preprocessing.MaxAbsScaler()),
                            ('clf', LogisticRegression(random_state=42))])

    pipe_decisiontree = Pipeline([('scl', preprocessing.MaxAbsScaler()),
                                  ('clf', tree.DecisionTreeClassifier(random_state=42))])

    # Transform vectors to be svm compatible
    y_train = np.ravel(y_train)
    y_test = np.ravel(y_test)
    pipe_svm = Pipeline([('clf', svm.SVC(random_state=42))])
```

Hyperparameters:



```
    # Set grid search parameters
    param_range = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
    param_range_fl = [1.0, 0.5, 0.1]

    grid_params_logreg = [{'clf__penalty': ['l1', 'l2'],
                           'clf__C': param_range_fl,
                           'clf__solver': ['liblinear']}]

    grid_params_decisiontree = [{
                           'clf__min_samples_leaf': param_range,
                           'clf__max_depth': param_range,}]

    grid_params_svm = [{'clf__kernel': ['linear', 'rbf'],
                        'clf__C': param_range}]
```

Cross-Validation:

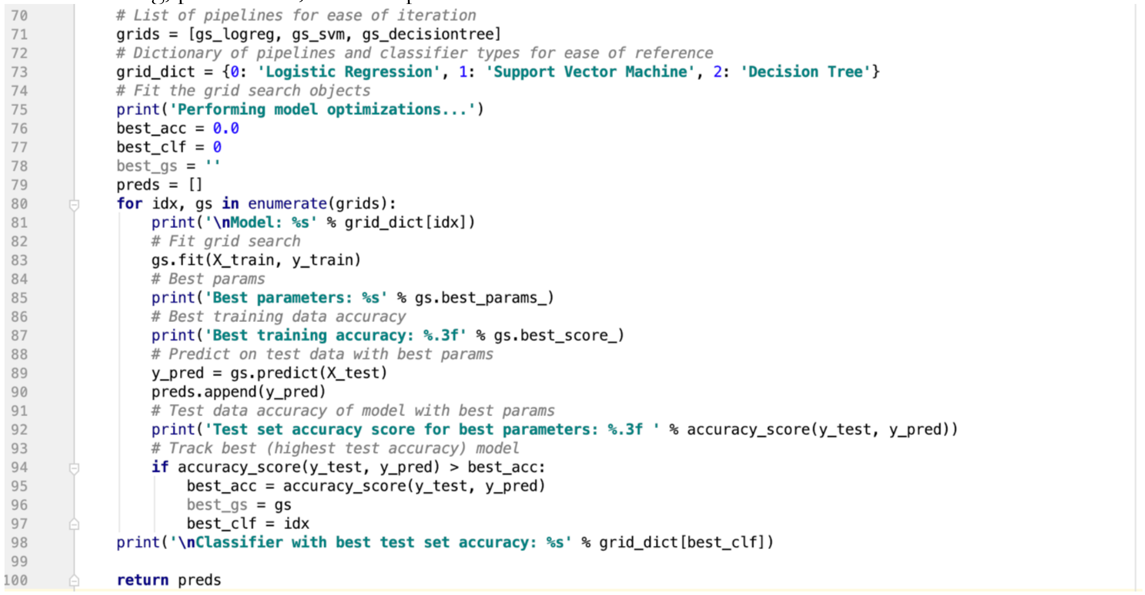


```
    gs_logreg = GridSearchCV(estimator=pipe_logreg,
                             param_grid=grid_params_logreg,
                             scoring='accuracy',
                             cv=10)

    gs_decisiontree = GridSearchCV(estimator=pipe_decisiontree,
                             param_grid=grid_params_decisiontree,
                             scoring='accuracy',
                             cv=10,
                             n_jobs=jobs)

    gs_svm = GridSearchCV(estimator=pipe_svm,
                              param_grid=grid_params_svm,
                              scoring='accuracy',
                              cv=10,
                              n_jobs=jobs)
```

Performance Analysis

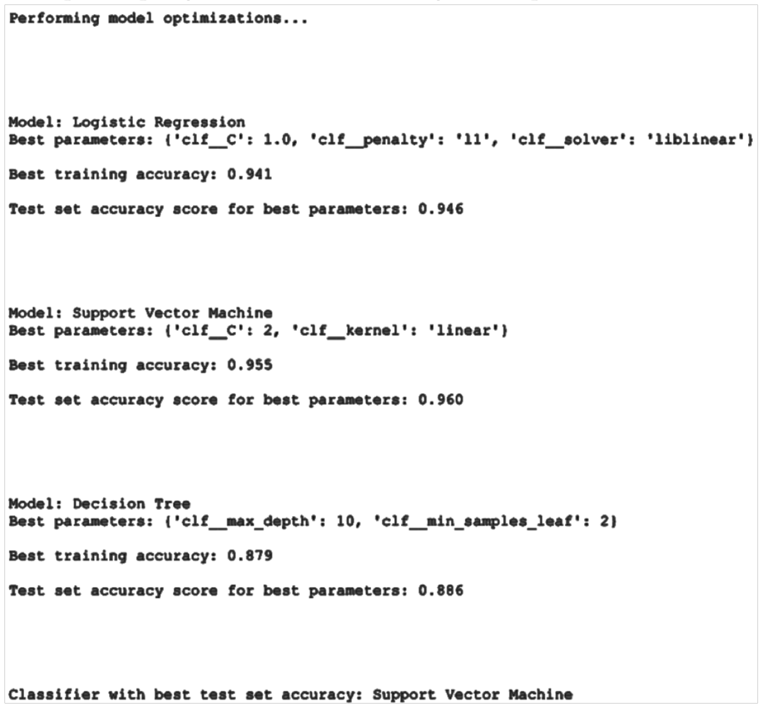
We used two different methods for comparing each model's effectiveness in classifying URLs. The first approach, once the parameters were established, consisted of iterating through our optimized models and recording the accuracy scores of each. We did this using an imported function called `accuracy_score` and by defining variables called `best_acc` to record the highest prediction accuracy out of the three models. We also created a variable `preds` to record the predictions by appending each model's prediction vector to `preds` through each iteration of the loop with the `.append()` command. `Preds` will later be used to create a ROC curve. We then compared performance and selected the most effective model.

Model training, predictions, and comparisons:



```python
    # List of pipelines for ease of iteration
    grids = [gs_logreg, gs_svm, gs_decisiontree]
    # Dictionary of pipelines and classifier types for ease of reference
    grid_dict = {0: 'Logistic Regression', 1: 'Support Vector Machine', 2: 'Decision Tree'}
    # Fit the grid search objects
    print('Performing model optimizations...')
    best_acc = 0.0
    best_clf = 0
    best_gs = ''
    preds = []
    for idx, gs in enumerate(grids):
        print('\nModel: %s' % grid_dict[idx])
        # Fit grid search
        gs.fit(X_train, y_train)
        # Best params
        print('Best parameters: %s' % gs.best_params_)
        # Best training data accuracy
        print('Best training accuracy: %.3f' % gs.best_score_)
        # Predict on test data with best params
        y_pred = gs.predict(X_test)
        preds.append(y_pred)
        # Test data accuracy of model with best params
        print('Test set accuracy score for best parameters: %.3f ' % accuracy_score(y_test, y_pred))
        # Track best (highest test accuracy) model
        if accuracy_score(y_test, y_pred) > best_acc:
            best_acc = accuracy_score(y_test, y_pred)
            best_gs = gs
            best_clf = idx
    print('\nClassifier with best test set accuracy: %s' % grid_dict[best_clf])

    return preds
```

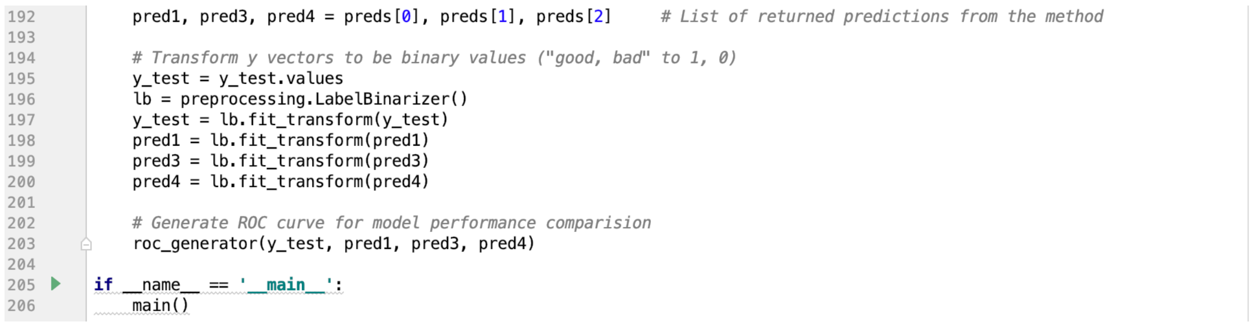
Example output generated when running the script:



```
Performing model optimizations...



Model: Logistic Regression
Best parameters: {'clf__C': 1.0, 'clf__penalty': 'l1', 'clf__solver': 'liblinear'}

Best training accuracy: 0.941

Test set accuracy score for best parameters: 0.946



Model: Support Vector Machine
Best parameters: {'clf__C': 2, 'clf__kernel': 'linear'}

Best training accuracy: 0.955

Test set accuracy score for best parameters: 0.960



Model: Decision Tree
Best parameters: {'clf__max_depth': 10, 'clf__min_samples_leaf': 2}

Best training accuracy: 0.879

Test set accuracy score for best parameters: 0.886



Classifier with best test set accuracy: Support Vector Machine
```

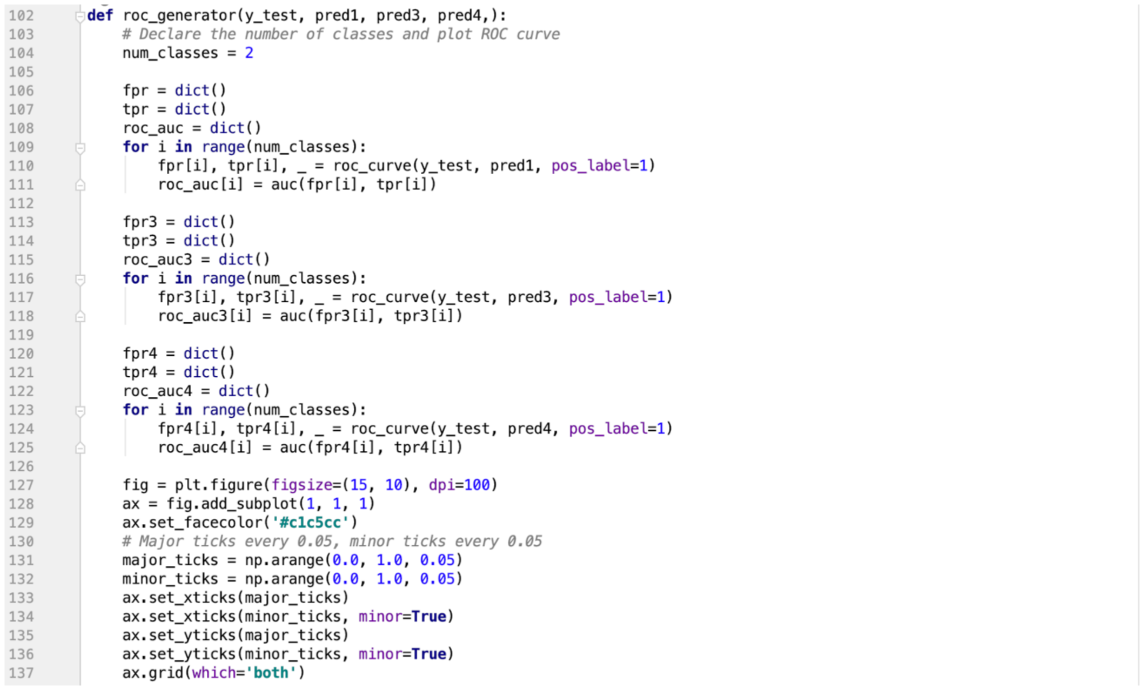
Receiver Operating Characteristics Curve

For the ROC curve, we used the returned `preds` variable as discussed earlier. `Preds` is a list comprised of each models' predictions. In order generate a ROC curve, the prediction vectors must be in binary output. To do this, we first pulled the values of the vector using the `.values` command. Then, using the imported function `preprocessing.LabelBinarizer()` we were able to binarize the output with the variable we defined named `lb.` After this preprocessing stage the data was ready to be sent to our method that generates the ROC curve off of the given information: test set predictions, Logistic Regression classifications, Support Vector Classifications, and the Decision Tree classifications.



```python
    pred1, pred3, pred4 = preds[0], preds[1], preds[2]     # List of returned predictions from the method

    # Transform y vectors to be binary values ("good, bad" to 1, 0)
    y_test = y_test.values
    lb = preprocessing.LabelBinarizer()
    y_test = lb.fit_transform(y_test)
    pred1 = lb.fit_transform(pred1)
    pred3 = lb.fit_transform(pred3)
    pred4 = lb.fit_transform(pred4)

    # Generate ROC curve for model performance comparision
    roc_generator(y_test, pred1, pred3, pred4)

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

To construct the ROC curve, we created a False Positive Rate dictionary and a True Positive Rate dictionary. In the simplest terms, the false positive rate and true positive rate are related by the fact that when the model predicts a true positive, the false positive rate is the probability of being wrong: 1-p. In this instance, we defined the false positive rate as `fpr` and the true positive rate as `tpr.` Before any calculations began, we also needed to define an Area Under the Curve variable named `roc_auc.` The area under the curve measures discrimination of the model in classifying a URL from randomly drawn pairs or in other words, the accuracy of the model when labeling a URL.

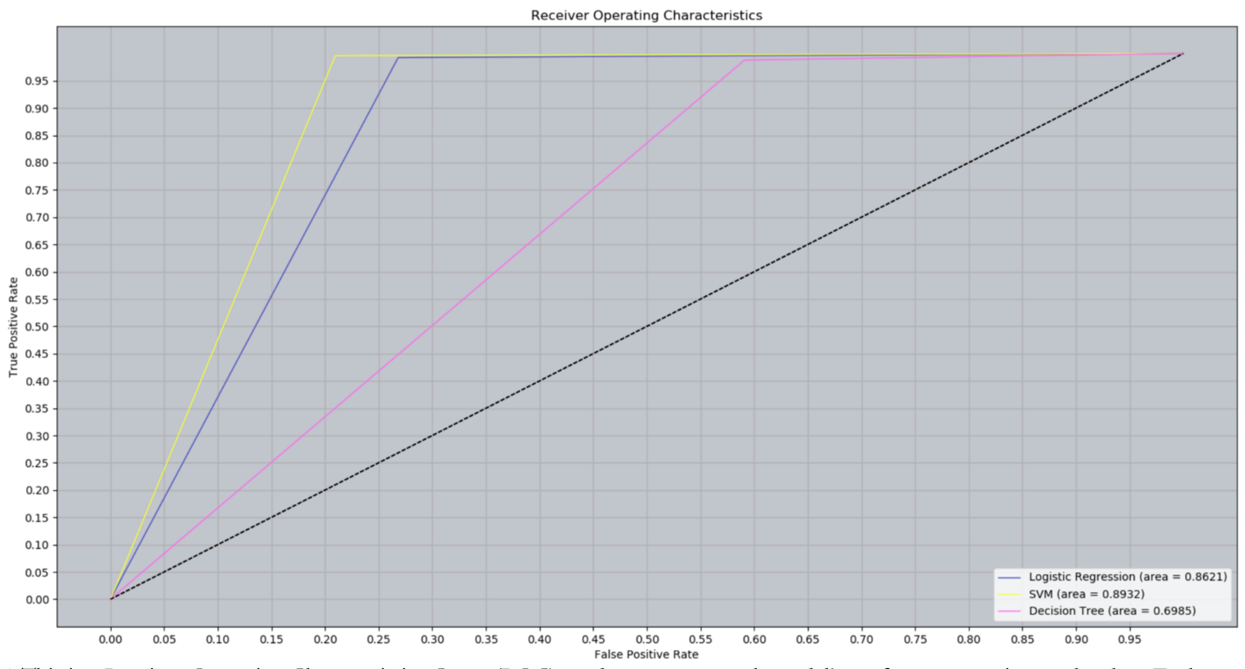


```python
def roc_generator(y_test, pred1, pred3, pred4,):
    # Declare the number of classes and plot ROC curve
    num_classes = 2

    fpr = dict()
    tpr = dict()
    roc_auc = dict()
    for i in range(num_classes):
        fpr[i], tpr[i], _ = roc_curve(y_test, pred1, pos_label=1)
        roc_auc[i] = auc(fpr[i], tpr[i])

    fpr3 = dict()
    tpr3 = dict()
    roc_auc3 = dict()
    for i in range(num_classes):
        fpr3[i], tpr3[i], _ = roc_curve(y_test, pred3, pos_label=1)
        roc_auc3[i] = auc(fpr3[i], tpr3[i])

    fpr4 = dict()
    tpr4 = dict()
    roc_auc4 = dict()
    for i in range(num_classes):
        fpr4[i], tpr4[i], _ = roc_curve(y_test, pred4, pos_label=1)
        roc_auc4[i] = auc(fpr4[i], tpr4[i])

    fig = plt.figure(figsize=(15, 10), dpi=100)
    ax = fig.add_subplot(1, 1, 1)
    ax.set_facecolor('#c1c5cc')
    # Major ticks every 0.05, minor ticks every 0.05
    major_ticks = np.arange(0.0, 1.0, 0.05)
    minor_ticks = np.arange(0.0, 1.0, 0.05)
    ax.set_xticks(major_ticks)
    ax.set_xticks(minor_ticks, minor=True)
    ax.set_yticks(major_ticks)
    ax.set_yticks(minor_ticks, minor=True)
    ax.grid(which='both')
```

We iterated through the predictions of each model and defined `fpr, tpr` and `roc_auc` dictionaries for each instance. The `pos_label` is just stating that a positive class prediction will be labeled as a 1. Following the calculations, we used an imported package called `matplotlib` to construct the actual graph that the curve would be represented on. The `ax.set_ticks()`is the built in command to create the grid structure of the graph. We found this to be distracting from the curve and model comparison with an out of the box white background included in matplotlib. We customized it to display a more transparent grid structure with a light grey background. This is done with the `ax.set_facecolor()` command and we found the exact color match We wanted with a HTML hex color code of `#c1c5cc.`

We then went on to plot each curve corresponding to its respective model using the `pyplot` command from the `matplotlib` package.

```
    # Logistic Regression curve
    plt.plot(fpr[1], tpr[1], color='#4a50bf',
             lw=1, label='Logistic Regression (area = %0.4f)' % roc_auc[1])
    plt.plot([0, 1], [0, 1], color='black', lw=1, linestyle='--')

    # Support Vector Machine
    plt.plot(fpr3[1], tpr3[1], color='#F6FF33',
             lw=1, label='SVM (area = %0.4f)' % roc_auc3[1])
    plt.plot([0, 1], [0, 1], color='black', lw=1, linestyle='--')

    # Decision Tree curve
    plt.plot(fpr4[1], tpr4[1], color='#ff68f0',
             lw=1, label='Decision Tree (area = %0.4f)' % roc_auc4[1])
    plt.plot([0, 1], [0, 1], color='black', lw=1, linestyle='--')

    plt.xlabel('False Positive Rate')
    plt.ylabel('True Positive Rate')
    plt.title('Receiver Operating Characteristics')
    plt.legend(loc="lower right")
    plt.show()
```



* This is a Receiver Operating Characteristics Curve (ROC) used to compare each model's performance against each other. Each model is an optimized version ran through identical pipelines. 10-fold Grid Search Cross-Validation was used to select the most useful hyperparameters.