Shelter Animal Outcomes

04.26.2019

Project Team

Imelda Flores

Kristen Marenco

Angelo Onato

Andy Xie

Table of Contents

Introduction

“Every year, approximately 7.6 million companion animals end up in US shelters. Many animals are given up as unwanted by their owners, while others are picked up after getting lost or taken out of cruelty situations. Many of these animals find forever families to take them home, but just as many are not so lucky. 2.7 million dogs and cats are euthanized in the US every year. Using a dataset of intake information including breed, color, sex, and age from the Austin Animal Center, we're asking Kagglers to predict the outcome for each animal. “ (Shelter Animal Outcomes)

Goals

The goal of this project is to predict shelter animal outcomes such as, adoption, transfer, euthanasia, died, and return to owner, using the provided data. The features used include, breed, color, sex, and age to predict these shelter animal outcomes. The provided dataset is courtesy of the Austin Animal Center from October 2013 to March 2016.

Dataset Description

Initial Dataset

The initial dataset provided by the Austin Animal Center, included 10 columns and 26,730 rows.

1. AnimalID
    a. All animals receive a unique Animal ID during intake.
2. Name
    a. The animal’s name during intake. (May be empty)
3. DateTime
    a. The time upon Outcome.
4. OutcomeType
    a. Adoption, Died, Euthanasia, Return to owner, Transfer.
5. OutcomeSubtype
    a. Reason for Outcome.
6. AnimalType
    a. Dog, or Cat
7. SexuponOutcome
    a. Intact Female, Intact Male, Spayed Female, Neutered Male, Unknown

8. AgeuponOutcome
    a. The age of the animal upon Outcome.
9. Breed
    a. The breed of the animal.
10. Color
    a. The color of the animal.

Dataset After Changes

The following columns were removed:

AnimalID was removed because it was unique for every animal and it provided no meaningful input for the dataset.

OutcomeSubtype was removed from the dataset because it was considered to be data leakage. This is due to it being too closely related to the OutcomeType (our target), this information was part of future data therefore it was removed.

Name was also removed from the dataset because it provided no meaningful input for the dataset.

1. Name
    a. Boolean, derived from the Name column, except now the datatype is a boolean. True if the animal has a name.
2. AgeinDaysUponOutcome
    a. This column was derived from the AgeuponOutcome column, except now all ages are measured in the same unit, days.
3. Day
    a. The day on which the Outcome occurred.
4. Month
    a. The month on which the Outcome occurred.
5. Year
    a. The year on which the Outcome occurred.
6. Hour0
    a. Boolean, the hour on which the Outcome occurred was between 00 - 05.
7. Hour1
    a. Boolean, the hour on which the Outcome occurred was between 06 - 11.
8. Hour2
    a. Boolean, the hour on which the Outcome occurred was between 12 - 17.
9. Hour3
    a. Boolean, the hour on which the Outcome occurred was between 17 - 24.
10. Cat

    a. Boolean, derived from AnimalType column.
11. Dog
    a. Boolean, derived from AnimalType column.
12. IntactFemale
    a. Boolean, derived from SexuponOutcome column.
13. IntactMale
    a. Boolean, derived from SexuponOutcome column.
14. NeuteredMale
    a. Boolean, derived from SexuponOutcome column.
15. SpayedFemale
    a. Boolean, derived from SexuponOutcome column.
16. UnkownSex
    a. Boolean, derived from SexuponOutcome column.
17. Pit Bull
    a. Boolean, derived from the Breed column. Accounts for 8% of the Breed column.
18. Chihuahua
    a. Boolean, derived from the Breed column. Accounts for 9% of the Breed column.
19. Shepherd
    a. Boolean, derived from the Breed column. Accounts for 5% of the Breed column.
20. Retriever
    a. Boolean, derived from the Breed column. Accounts for 9% of the Breed column.
21. Terrier
    a. Boolean, derived from the Breed column. Accounts for 6% of the Breed column.
22. DomesticShorthair
    a. Boolean, derived from the Breed column. Accounts for 33% of the Breed column.
23. DomesticMediumHair
    a. Boolean, derived from the Breed column. Accounts for 3% of the Breed column.
24. DomesticLonghair
    a. Boolean, derived from the Breed column. Accounts for 2% of the Breed column.
25. Siamese
    a. Boolean, derived from the Breed column. Accounts for 1% of the Breed column.
26. Other Breed

    a. Boolean, derived from the Breed column. Accounts for 24% of the Breed column.
27. Black
    a. Boolean, derived from the Color column. Accounts for 30% of the Color column.
28. Brown
    a. Boolean, derived from the Color column. Accounts for 22% of the Color column.
29. White
    a. Boolean, derived from the Color column. Accounts for 21% of the Color column.
30. Tan
    a. Boolean, derived from the Color column. Accounts for 3% of the Color column.
31. Blue
    a. Boolean, derived from the Color column. Accounts for 4% of the Color column.
32. Tabby
    a. Boolean, derived from the Color column. Accounts for 4% of the Color column.
33. Other Color
    a. Boolean, derived from the Color column. Accounts for 16% of the Color column.
34. OutcomeType
    a. There are 5 different Outcomes: Adoption, Died, Euthanasia, Return to owner, Transfer. This is our target.

Team Member Responsibilities

I. Imelda Flores

Responsible for implementing GridSearchCV, Support Vector Machine (SVM), and Gradient Boosting Classifier. Responsible for writing the "Team Results" section of the report.

II. Kristen Marenco

Responsible for implementing and reporting XGBoost and Random Forest. Responsible for functions listed in "Various Changes to Dataset." Responsible for

writing the “Various Changes to Dataset”, ‘Normalizing the Dataset”, “Splitting the Dataset into Training and Testing sets” and “Team Results” sections of the report.

III. Angelo Onato

Responsible for implementing Adaboost and Decision Tree. Responsible for writing the “Conclusion” section of the report.

IV. Andy Xie

Responsible for implementing KNN, Logistic Regression, and ANN. Responsible for writing the “Conclusion” section of the report.

All team members are responsible for regulating, normalizing, and OneHotEncoding categorical columns, as well as listing their references in the “References” section of the project report, and project presentation slides.

Methods

Various Changes to Dataset

There were many different changes made to the initial dataset.

	AnimalID	Name	DateTime	OutcomeType	OutcomeSubtype	AnimalType	SexuponOutcome	AgeuponOutcome	Breed	Color
0	A671945	Hambone	2014-02-12 18:22:00	Return_to_owner	NaN	Dog	Neutered Male	1 year	Shetland Sheepdog Mix	Brown/White
1	A656520	Emily	2013-10-13 12:44:00	Euthanasia	Suffering	Cat	Spayed Female	1 year	Domestic Shorthair Mix	Cream Tabby
2	A686464	Pearce	2015-01-31 12:28:00	Adoption	Foster	Dog	Neutered Male	2 years	Pit Bull Mix	Blue/White
3	A683430	NaN	2014-07-11 19:09:00	Transfer	Partner	Cat	Intact Male	3 weeks	Domestic Shorthair Mix	Blue Cream
4	A667013	NaN	2013-11-15 12:52:00	Transfer	Partner	Dog	Neutered Male	2 years	Lhasa Apso/Miniature Poodle	Tan

Initial Dataset

The datetime column was converted into Hour Day Month and Year columns.

```
from datetime import datetime, time, date

raw_df['Hour'] = raw_df.DateTime.apply(lambda x : datetime.strptime(x, '%Y-%m-%d %H:%M:%S').strftime('%H'))

raw_df['Day'] = raw_df.DateTime.apply(lambda x : datetime.strptime(x, '%Y-%m-%d %H:%M:%S').strftime('%d'))

raw_df['Month']= raw_df.DateTime.apply(lambda x : datetime.strptime(x,  '%Y-%m-%d %H:%M:%S').strftime('%m'))

raw_df['Year']= raw_df.DateTime.apply(lambda x : datetime.strptime(x,  '%Y-%m-%d %H:%M:%S').strftime('%Y'))
```

The Hour column was converted into Hour0 Hour1 Hour2 and Hour3 columns by using the hourBins function

```
def hourBins(hour):
    hour = int(hour)
    if 0<= hour <= 5:
        return 1
    if 6<= hour <= 11:
        return 2
    if 12<= hour <= 17:
        return 3
    else:
        return 4

raw_df['Hour'] = raw_df['Hour'].apply(lambda x: hourBins(x))

#check if all bin thresholds are in df
raw_df.Hour.unique()
```

```
array([4, 3, 2, 1], dtype=int64)
```

and by OneHotEncoding.

```
hr = LabelEncoder()
hr_ohe = OneHotEncoder()

raw_df['Hour'] = hr.fit_transform(raw_df.Hour)

Xh = hr_ohe.fit_transform(raw_df.Hour.values.reshape(-1,1)).toarray()

Xh_OneHot = pd.DataFrame(Xh, columns = ["Hour"+str(int(i)) for i in range(Xh.shape[1])])
raw_df = pd.concat([raw_df, Xh_OneHot], axis=1)
```

The AnimalType column was converted into Dog and Cat columns by OneHotEncoding.

```
animal = LabelEncoder()
animal_ohe = OneHotEncoder()

raw_df['AnimalType'] = animal.fit_transform(raw_df.AnimalType)

Xa = animal_ohe.fit_transform(raw_df.AnimalType.values.reshape(-1,1)).toarray()

Xa_OneHot = pd.DataFrame(Xa, columns = ["Animal" + str(int(i)) for i in range(Xa.shape[1])])

raw_df = pd.concat([raw_df, Xa_OneHot], axis=1)

raw_df = raw_df.rename(columns = {'Animal0': 'Cat'})
raw_df = raw_df.rename(columns = {'Animal1': 'Dog'})
```

The SexuponOutcome column was converted into NeuteredMale SpayedFemale IntactMale IntactFemale and UnknownSex columns by using the sexUnkown function

```
def sexUnknown(sex):
    sex = int(sex)
    if sex > 4:
        return 4
    else:
        return sex
#This function is merging the Unknown and Blank values for SexuponOutcome
```

And by OneHotEncoding.

```
sex = LabelEncoder()
sex_ohe = OneHotEncoder()

raw_df['SexuponOutcome'] = sex.fit_transform(raw_df['SexuponOutcome'].astype(str))

print(raw_df.SexuponOutcome.unique())

raw_df['SexuponOutcome'] = raw_df['SexuponOutcome'].apply(lambda x: sexUnknown(x))

Xs = sex_ohe.fit_transform(raw_df.SexuponOutcome.values.reshape(-1,1)).toarray()

Xs_OneHot = pd.DataFrame(Xs, columns = ["Sex" + str(int(i)) for i in range(Xs.shape[1])])

raw_df = pd.concat([raw_df, Xs_OneHot], axis=1)

raw_df = raw_df.rename(columns = {'Sex0': 'IntactFemale'})
raw_df = raw_df.rename(columns = {'Sex1': 'IntactMale'})
raw_df = raw_df.rename(columns = {'Sex2': 'NeuteredMale'})
raw_df = raw_df.rename(columns = {'Sex3': 'SpayedFemale'})
raw_df = raw_df.rename(columns = {'Sex4': 'Unknown Sex'})
```

The AgeuponOucome column was converted into AgeinDaysUponOutcome column by using the age function.

```
def age(age):
    age = str(age) #explicit string
    if not age: #check if blank before casting
        return -1 #Note: all blank values will be changed to -1

    strToarr = age.split(" ")
    if len(strToarr) > 1:
        num = int(strToarr[0])
        date = strToarr[1]

        if "year" in date:
            return num * 365
        elif "month" in date:
            return num * 30
        elif "week" in date:
            return num * 7
        elif "day" in date:
            return num
    else:
        return -1
#This function is converting Age into Age in days
```

```
raw_df['AgeuponOutcome'] = raw_df['AgeuponOutcome'].apply(lambda x : age(x))

raw_df = raw_df.rename(columns = {'AgeuponOutcome': 'AgeinDaysUponOutcome'})
```

The Breed column was converted into PitBull Chihuahua Shepherd Retriever Terrier DomesticShorthair DomesticMediumHair DomesticLonghair Siamese and OtherBreed

columns using the breedOrganizer function. The breeds were chosen based on the amount of cats/dogs for a specific breed.

```
def breedOrganizer(breed):
    breed = str(breed) #explicit string

    if "Pit Bull" in breed:
        return 0
    elif "Chihuahua" in breed:
        return 1
    elif "Shepherd" in breed:
        return 2
    elif "Retriever" in breed:
        return 3
    elif "Terrier" in breed:
        return 4
    elif "Domestic Shorthair" in breed:
        return 5
    elif "Domestic Medium Hair" in breed:
        return 6
    elif "Domestic Longhair" in breed:
        return 7
    elif "Siamese" in breed:
        return 8
    else:
        return 9

#This function is separating the Breed Category into the top breeds for Cat/Dog
```

And OneHotEncoding.

```
breed = LabelEncoder()
Xb_ohe = OneHotEncoder()

raw_df['Breed'] = raw_df['Breed'].apply(lambda x : breedOrganizer(x))

Xb = Xb_ohe.fit_transform(raw_df.Breed.values.reshape(-1,1)).toarray()

Xb_OneHot = pd.DataFrame(Xb, columns = ["Pit Bull", "Chihuahua", "Shepherd", "Retriever", "Terrier", "DomesticShorthair", "Do

raw_df = pd.concat([raw_df, Xb_OneHot], axis=1)
```

The Color column was converted into Black Brown White Tan Blue Tabby and OtherColor columns using the colorOrganizer function. The colors chosen based on the higher amount of cats and dogs color.

```
def colorOrganizer(color):
    color = str(color) #explicit string

    if "Black" in color:
        return 0
    elif "Brown" in color:
        return 1
    elif "White" in color:
        return 2
    elif "Tan" in color:
        return 3
    elif "Blue" in color:
        return 4
    elif "Tabby" in color:
        return 5
    else:
        return 6

#This function is separating the Color column into the top colors/other
```

And OneHotEncoding.

```
color = LabelEncoder()
Xc_ohe = OneHotEncoder()

raw_df['Color'] = raw_df['Color'].apply(lambda x : colorOrganizer(x))

Xc = Xc_ohe.fit_transform(raw_df.Color.values.reshape(-1,1)).toarray()

Xc_OneHot = pd.DataFrame(Xc, columns = ["Black", "Brown", "White", "Tan", "Blue", "Tabby", "Other Color"])

raw_df = pd.concat([raw_df, Xc_OneHot], axis=1)
```

The Name column was converted from string to boolean type using the nameToBool function.

```
def nameToBool(name):
    name = str(name) #explicit string

    if len(name) > 3: #animal has a name
            return 1
    else:
        return 0 #animal does not have name
#This function converts the Name column into a boolean column. If true, the animal has a name.
```

All unnecessary columns were removed and after all changes were made the dataset looked like the image below:

	Name	AgeinDaysUponOutcome	Day	Month	Year	Hour0	Hour1	Hour2	Hour3	Cat	...	DomesticLonghair	Siamese	Other Breed	Black	Brown	White	Tan	I
0	1	365	12	02	2014	0.0	0.0	0.0	1.0	0.0	...	0.0	0.0	1.0	0.0	1.0	0.0	0.0	
1	1	365	13	10	2013	0.0	0.0	1.0	0.0	1.0	...	0.0	0.0	0.0	0.0	0.0	0.0	0.0	
2	1	730	31	01	2015	0.0	0.0	1.0	0.0	0.0	...	0.0	0.0	0.0	0.0	0.0	1.0	0.0	
3	0	21	11	07	2014	0.0	0.0	0.0	1.0	1.0	...	0.0	0.0	0.0	0.0	0.0	0.0	0.0	
4	0	730	15	11	2013	0.0	0.0	1.0	0.0	0.0	...	0.0	0.0	1.0	0.0	0.0	0.0	1.0	

5 rows × 33 columns

Dataset after Changes

Normalizing the Dataset

The dataset was normalized using the Sklearn preprocessing function.

```
1 from sklearn import preprocessing
2
3 feat_nparr = preprocessing.scale(animal_df)
4
5 animal_df = pd.DataFrame(feat_nparr, columns = animal_df.columns)
6
7 animal_df.head()
```

Dataset before normalization:

	Name	AgeinDaysUponOutcome	Day	Month	Year	Hour0	Hour1	Hour2	Hour3	Cat
0	1	365	12	02	2014	0.0	0.0	0.0	1.0	0.0
1	1	365	13	10	2013	0.0	0.0	1.0	0.0	1.0
2	1	730	31	01	2015	0.0	0.0	1.0	0.0	0.0
3	0	21	11	07	2014	0.0	0.0	0.0	1.0	1.0
4	0	730	15	11	2013	0.0	0.0	1.0	0.0	0.0

5 rows × 33 columns

Dataset after normalization:

	Name	AgeinDaysUponOutcome	Day	Month	Year	Hour0	Hour1
0	0.698276	-0.395879	-0.421688	-1.409043	-0.580974	-0.120091	-0.428535
1	0.698276	-0.395879	-0.307667	0.879415	-1.929785	-0.120091	-0.428535
2	0.698276	-0.058741	1.744707	-1.695100	0.767836	-0.120091	-0.428535
3	-1.432099	-0.713620	-0.535709	0.021244	-0.580974	-0.120091	-0.428535
4	-1.432099	-0.058741	-0.079626	1.165473	-1.929785	-0.120091	-0.428535

5 rows × 33 columns

Splitting the Dataset into Training and Testing sets

For the following algorithms (unless otherwise stated) the team split the dataset into training and testing sets. The testing set was 25% of the dataset, the other 75% was the training set.

```
from sklearn.model_selection import train_test_split

X_train, X_test, y_train, y_test = train_test_split(animal_df, target, test_size = 0.25, random_state = 3)
```

K Cross Validation

10 Fold Cross Validation was used to find the average of 10 samples. Each time a different sample was used as testing set and remaining 9 as training set. The procedure consists of a parameter called *k*. *K* refers to the number of groups the data is split into. In this case, the date will consist of ten groups. First, the data will be shuffled. Second, the data will be split into 10 groups. Third, for each set of groups cross validation will take a group that will be used as test data. The remaining nine groups will be used as training data. It will continue to repeat until all ten groups have been used as test data. The results are summarized with the mean of the scores achieved with the algorithm.

```
from sklearn.model_selection import cross_val_score

accuracy_list_clf_linear = cross_val_score(clf_linear, X_train, y_train, cv = 10, scoring = 'accuracy')
print(accuracy_list_clf_linear)
```

Principal Component Analysis (PCA)

Principal Component Analysis (PCA) is used for feature extraction. PCA will keep the most important features, the features that have the biggest impact on the accuracy, and drop the 'least important' features, the features that have little to no impact on the accuracy. A benefit of PCA is that all variables are independent from one another. PCA's reduction of features will retain the variation in the data set. PCA variation will be emphasized and bring out strong patterns in the dataset.

Tools

Programs: Python, Jupyter Notebook, Excel, Google

Libraries: Pandas, Numpy, datetime, and sklearn

Algorithms

XGBoost - Kristen Marenco

XGBoost is an implementation of gradient boosted decision trees designed for speed and performance. XGBoost is an approach that converts a set of weak learners into strong learners. To do this, XGBoost takes a set of weak learners and combines them, which is a concept known as continued training, XGBoost does this to further boost an already fitted model on new data. After this, XGBoost then incorporates voting to produce the most accurate model. Because XGBoost features continued training, this makes it extremely useful for models that are under fitted. However, XGBoost will result in overfitting if a model is already well-fitted.

```
from xgboost import XGBClassifier

xg = XGBClassifier()

xg.fit(X_train, y_train)

xg_y_predict = xg.predict(X_test)

xg_proba = xg.predict_proba(X_test)
```

Since XGBoost incorporates decision trees, it supports predicting the probability of an event.

For this project the performance of XGBoost was measured by accuracy and its ROC curve.

Random Forest - Kristen Marenco

Random Forest is an algorithm that builds multiple decision trees and merges them together to get a more accurate prediction model. Random Forest uses bagging in order to produce the best predictive model. Random Forest also prevents overfitting in most cases due to its diversity of decision trees.

```
rf = RandomForestClassifier(n_estimators=31, bootstrap = True, random_state=2)

rf.fit(X_train, y_train)

rf_y_predict = rf.predict(X_test)

print(rf_y_predict)

rf_score = accuracy_score(y_test, rf_y_predict)
```

Random Forest supports predicting the probability of an event.

For this project the performance of Random Forest was measured by accuracy and its ROC curve.

AdaBoost - Angelo Onato

Adaptive Boosting is a machine learning algorithm which helps improve performance of other classifiers. It does this be making weaker classifiers stronger. This was selected to make boost decision tree classifier as it is one of the weaker classifiers and it is non-linear.

```
In [177]: # adaBoost
          from sklearn.ensemble import AdaBoostClassifier

          my_AdaBoost = AdaBoostClassifier(n_estimators = 100, random_state = 4)
          my_AdaBoost.fit(X_train, y_train)

Out[177]: AdaBoostClassifier(algorithm='SAMME.R', base_estimator=None,
                    learning_rate=1.0, n_estimators=100, random_state=4)
```

Decision Tree - Angelo Onato

Decision tree is a supervised machine learning method where data is split due to some by certain choices. In a decision tree, a node is where the data splits as a decision will be made. Connected to these nodes are the branches which is the possible outcome. Once all is done, the data will be classified in the end nodes known as the leafs.

```
In [174]: #Decision Tree
          from sklearn.tree import DecisionTreeClassifier
          my_decisiontree = DecisionTreeClassifier(random_state=4)
          my_decisiontree.fit(X_train, y_train)

Out[174]: DecisionTreeClassifier(class_weight=None, criterion='gini', max_depth=None,
                      max_features=None, max_leaf_nodes=None,
                      min_impurity_decrease=0.0, min_impurity_split=None,
                      min_samples_leaf=1, min_samples_split=2,
                      min_weight_fraction_leaf=0.0, presort=False, random_state=4,
                      splitter='best')
```

With the animal shelter data, the nodes will be filled with the features we'll be using and the leaves will be the outcome type.

KNN - Andy Xie

KNN is short for k-nearest neighbors and it is a algorithm for classification and regression that works by forming a "group" around the object you are classifying. KNN will compare the objects around this "group" (neighbors) and use that information to classify the object.

Logistic Regression - Andy Xie

Logistic regression uses a logistic sigmoid function to output a probability value which can then be used to classify objects. We use a multiclass logistic regression in our project to classify the animal shelter outcomes.

ANN - Andy Xie

Artificial Neural networks (ANN) are modeled after the human brain to help recognize patterns. Like the human brain which uses millions of neurons connected through synapses that send and process signals, ANN uses a large number of connected nodes to process information.

Support Vector Machine (SVM) - Imelda Flores

Support Vector Machine (SVM) has two types of classifiers: linear classifier and non-linear model using Kernel Trick. SVM is used to find the best hyperplane (or set of hyperplanes) used to separate data samples. A hyperplane is a line where if there is 2 features than a line in 2D. If it's 3 features it will be a plane in 3D and so on. To be considered a good classifier, the hyperplane needs to have the largest separation. This means the largest distance to the closest data sample. The margin is found by taking the output of the point using the formula $\boldsymbol{w}^T \boldsymbol{x} + b$. If the value is greater than 1 it is above the arbitrary line and

placed on a class. If the value is below -1 it is below the arbitrary line and placed on another class. If it is is equal to 0 then it is part of the arbitrary line.

```
from sklearn.svm import SVC

svm_linear = SVC(kernel ='linear', gamma ='auto', probability = True)
svm_linear.fit(X_train, y_train)
y_pred_linear = svm_linear.predict(X_test)
score_linear = accuracy_score(y_test, y_pred_linear)

print(score_linear)
```

Kernel Trick is used when the data is non-linear (cannot be separated linearly). This changes the original data by mapping it to higher-dimensional feature space for the data to be linearly separable. Gaussian Radial-Basis Function (RBF) was used compared to other available kernel functions.

```
from sklearn.svm import SVC

rbf = SVC(kernel ='rbf', gamma ='auto', probability = True)
rbf.fit(X_train, y_train)
y_pred_rbf = rbf.predict(X_test)
score_rbf = accuracy_score(y_test, y_pred_rbf)

print(score_rbf)
```

GridSearchCV - Imelda Flores

GridSearchCV is a combination of cross validation and (hyper)parameter tuning. The default value for K-Fold validation is a 3-Fold, unless otherwise specified. The value set for this project was 10. The parameter tuning selects values to maximize the accuracy of the model. GridSearchCV needs another algorithm as the 'estimator' parameter. GridSearch will try the possible combinations with the given parameters and return the highest accuracy found. GridSearch can be slow since it has to execute a large amount of combinations.

```
from sklearn.model_selection import GridSearchCV

param_grid = {'C':[0.001, 0.01, 0.1, 1, 10], 'gamma' : [0.0005]}

grid_cv = GridSearchCV(svm.SVC(kernel = 'rbf'), param_grid, cv = 10)
grid_cv.fit(X_train, y_train)

print(grid_cv.best_score_)
print(grid_cv.best_params_)
```

Gradient Boosting Classifier - Imelda Flores

Gradient Boosting needs the following elements: loss function, weak learner, and additive model. Loss function means that any differentiable loss function can be used with this classifier. It is a weak learner since the trees are constructed in a 'greedy' manner to minimize the loss. Additive model means that the gradient descent minimized the loss when adding trees. When trees are added, they will be added one at time and existing trees will not be modified or changed. The classifier calculates error and update the weights to minimize the error. A tree is added to reduce the loss and recalculated after it's added. When the loss is at a level that no longer improves with the dataset then a fixed number of trees is added or the training stops.

```
from sklearn.ensemble import GradientBoostingClassifier

gb = GradientBoostingClassifier(random_state = 1, n_estimators = 25, max_depth = 3)

gb.fit(X_train, y_train)
y_predict_gb = gb.predict(X_test)

score_gb = accuracy_score(y_test, y_predict_gb)

print(score_gb)
```

Results & Analysis

XGBoost - Kristen Marenco

XGBoost was chosen to use because it has become a very popular algorithm for Kaggle competitions due to it yielding the highest accuracy in almost every competition. XGBoost has great results because of its continued training approach making it very useful when most models have been underfitting the data.

For this project, **XGBoost had initially yielded an accuracy of 64%** when the dataset had 32 features. These 32 features included:

```
features = ['AgeinDaysUponOutcome', 'Day', 'Month', 'Year',
       'Hour0', 'Hour1', 'Hour2', 'Hour3', 'Cat', 'Dog', 'IntactFemale',
       'IntactMale', 'NeuteredMale', 'SpayedFemale', 'Unknown Sex', 'Pit Bull',
       'Chihuahua', 'Shepherd', 'Retriever', 'Terrier', 'DomesticShorthair',
       'DomesticMediumHair', 'DomesticLonghair', 'Siamese', 'Other Breed',
       'Black', 'Brown', 'White', 'Tan', 'Blue', 'Tabby', 'Other Color']
```

Looking at the image above one can see that the 'Name' column is not included. This was because the team did not see the value in OneHotEncoding the Name column because

there were not many similarities between names and this would not benefit the predicive model. However, it was decided to convert the Name column to a boolean column determining whether or not the animal had a name. This was beneficial and is how **the accuracy of XGBoost went up to 65.67%**:

```
from sklearn.metrics import accuracy_score

accuracy_xg = accuracy_score(y_test, xg_y_predict)

print("The accuracy score of XGBoost is: ", accuracy_xg *100, "%")
```

```
The accuracy score of XGBoost is:  65.674098458776 %
```

Since the testing set is 25% of the dataset, 10-fold cross validation was used to yield a more precise reading of the accuracy score of XGBoost.

```
cv_xgb_ac = cross_val_score(xg, X_train, y_train, cv = 10, scoring = 'accuracy')

print("The accuracy score of XGBoost is: ", cv_xgb_ac*100, "%")

cv_xgb_ac_mean = cv_xgb_ac.mean()



print("\n")
print("The mean accuracy score of XGBoost is: ", cv_xgb_ac_mean*100, "%")
```

```
The accuracy score of XGBoost is:  [64.70588235 65.75274177 64.8554337  66.05184447 64.7557328  65.78553616
 65.46906188 64.62075848 64.58541459 64.36781609] %


The mean accuracy score of XGBoost is:  65.09502222909668 %
```

After cross validation was performed, it was precise to say that **the accuracy score of XGBoost was 65.09%.**

XGBoost has a function called plot_importance which plots the features and their importance by determining their F-score. The F-score is a measure of the harmonic mean of precision and recall.

Precision, which we'll denote p for convenience, is defined as

$$p = \frac{tp}{tp+fp}$$

where tp and fp are true positives and false positives respectively.

Recall, which we'll denote as r, is defined as

$$r = \frac{tp}{tp+fn}$$

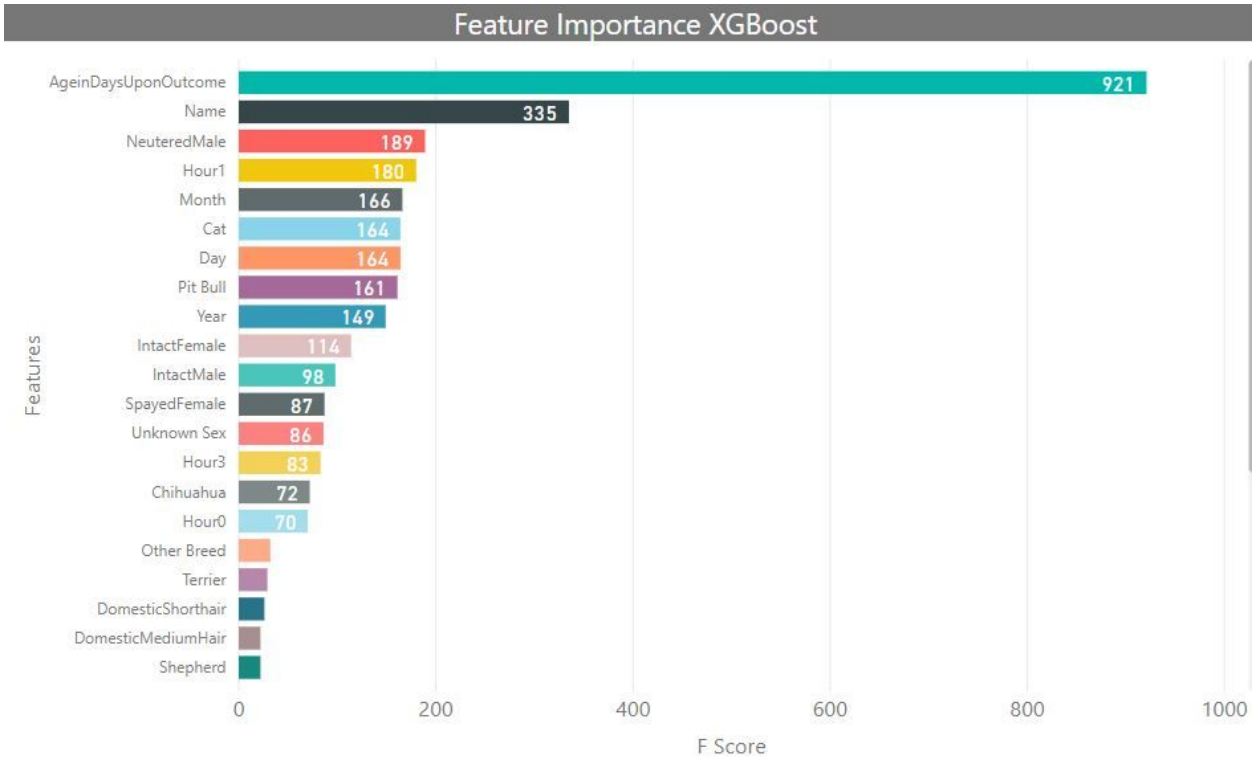
where fn stands for false negatives.

So precision is the ratio of true positives to those predicted positive, while recall is the ratio of true positives to all positives.

The F_1 score is defined as the harmonic mean of p and r:

$$F_1 = 2\frac{pr}{p+r} = \cdots = \frac{2tp}{2tp+fp+fn}$$

What this means is that, the F score is a good measure of how much a feature is contributing to the overall accuracy of a predictive model.



From the Feature Importance XGBoost graph, one can see that the age of animal is highly important when the XGBoost model predicts the animal's *outcome type*. **The top two performing features in the XGBoost model were *AgeinDaysUponOutcome* and *Name*.**

Once deducing the top performing features, to put this knowledge to use, the next step was to incorporate feature selection. Feature selection is useful for training a model faster, reducing the complexity of a model (this avoids overfitting), and improves the accuracy of a model if the right subset is chosen.

The goal when applying feature selection to the XGBoost model was to improve its accuracy score.

```
thresholds = sort(xg.feature_importances_)

for thresh in thresholds:

    selection = SelectFromModel(xg, threshold = thresh, prefit=True)
    select_X_train = selection.transform(X_train)

    selection_model = XGBClassifier()
    selection_model.fit(select_X_train, y_train)

    select_X_test = selection.transform(X_test)
    pred = selection_model.predict(select_X_test)
    predictions = [round(value) for value in pred]
    accuracy = accuracy_score(y_test, predictions)
    print("Thresh=%.3f, n=%d, Accuracy: %.2f%%" % (thresh, select_X_train.shape[1], accuracy*100.0))
```

This code shows the threshold and the amount of features used and their accuracy score.

```
Thresh=0.000, n=33, Accuracy: 65.67%
Thresh=0.001, n=32, Accuracy: 65.67%
Thresh=0.001, n=32, Accuracy: 65.67%
Thresh=0.001, n=30, Accuracy: 65.55%
Thresh=0.001, n=30, Accuracy: 65.55%
Thresh=0.003, n=28, Accuracy: 65.66%
Thresh=0.004, n=27, Accuracy: 65.75%
Thresh=0.004, n=26, Accuracy: 65.70%
Thresh=0.004, n=26, Accuracy: 65.70%
Thresh=0.005, n=24, Accuracy: 65.61%
Thresh=0.005, n=23, Accuracy: 65.75%
Thresh=0.006, n=22, Accuracy: 65.52%
Thresh=0.007, n=21, Accuracy: 65.69%
Thresh=0.007, n=21, Accuracy: 65.69%
Thresh=0.008, n=19, Accuracy: 65.61%
Thresh=0.009, n=18, Accuracy: 65.57%
Thresh=0.010, n=17, Accuracy: 65.63%
Thresh=0.021, n=16, Accuracy: 65.61%
Thresh=0.022, n=15, Accuracy: 64.85%
Thresh=0.025, n=14, Accuracy: 64.52%
Thresh=0.026, n=13, Accuracy: 64.33%
Thresh=0.026, n=12, Accuracy: 64.48%
Thresh=0.030, n=11, Accuracy: 64.15%
Thresh=0.035, n=10, Accuracy: 61.98%
Thresh=0.045, n=9, Accuracy: 59.58%
Thresh=0.049, n=8, Accuracy: 59.57%
Thresh=0.050, n=7, Accuracy: 59.66%
Thresh=0.050, n=7, Accuracy: 59.66%
Thresh=0.051, n=5, Accuracy: 58.73%
Thresh=0.055, n=4, Accuracy: 58.45%
Thresh=0.058, n=3, Accuracy: 57.82%
Thresh=0.102, n=2, Accuracy: 56.19%
Thresh=0.280, n=1, Accuracy: 51.13%
```

From the results, the highest accuracy score occurred twice. The first occurrence at threshold 0.004 with 27 features, and an accuracy score of 65.75%. The second occurrence at threshold 0.005, with 23 features, and an accuracy score of 65.75%. The previous accuracy score without performing feature selection was 65.09% **after performing feature selection the accuracy score went up to 65.75%.**

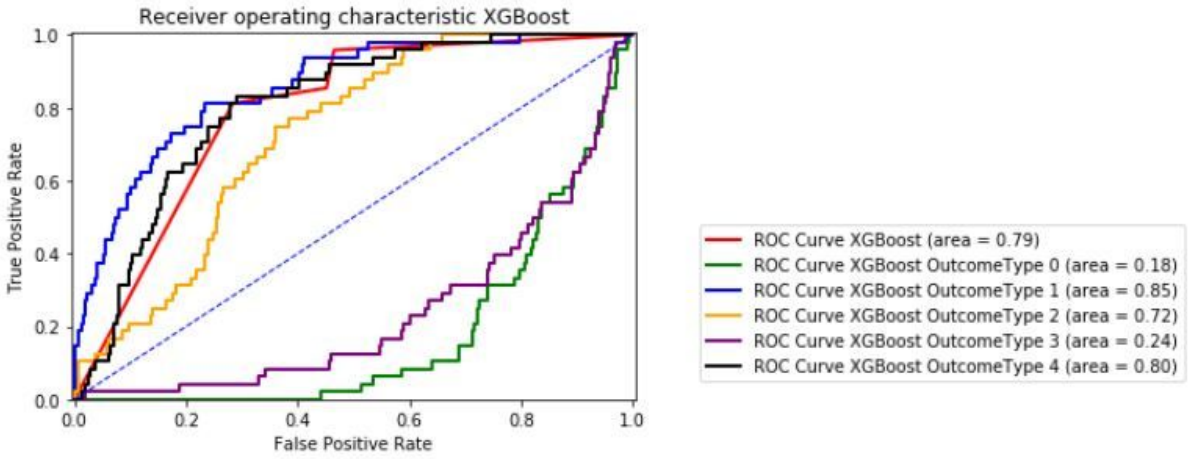
This showed that when there were fewer features the accuracy of the XGBoost predictive model was higher. So, in efforts to reduce the complexity of the model PCA was performed.

```
1  xg_pca = XGBClassifier()
2
3  xg_pca.fit(X_train_p, y_train_p)
4
5  xg_pca.predict(X_test_p)
6
7  xg_pca_score = xg_pca.score(X_test_p, y_test_p)
8
9  print("The accuracy of XGBoost after PCA is: ", xg_pca_score*100 ,"%")
```

```
The accuracy of XGBoost after PCA is:  64.29747119557085 %
```

Unfortunately, **PCA did not result in a higher accuracy score.** This was most likely due to the dataset not being complex enough or having enough overlapping features to benefit from the effect of PCA.

It was decided to keep the XGBoost model that yielded the highest accuracy score and its ROC curve is below:



The ROC curve line graph shows that the XGBoost model, **with an area under the curve of 0.79** was performing good, but not excellent. Based on the ROC curve the XGBoost model was good at predicting Outcome Types 1 and 4. However, the XGBoost model was absolutely horrible at predicting Outcome Types 0 and 3.

Probability analysis

From the ROC curve, it was deduced that the model was relatively effective at predicting Outcome Types 1, 2, and 4. For further inspection the following table shows how the XGBoost model performed:

	Y_test	XG_Model Prediction	Probability Outcome 0	Probability Outcome 1	Probability Outcome 2	Probability Outcome 3	Probability Outcome 4
21783	1	4	0%	1%	3%	0%	96%
11411	4	4	3%	2%	13%	1%	80%
15301	4	4	5%	8%	21%	1%	65%
17486	2	4	8%	0%	11%	24%	56%
14163	4	4	7%	0%	16%	33%	44%
25469	4	4	0%	1%	2%	0%	97%
991	4	0	75%	1%	2%	3%	20%
2123	2	0	31%	1%	25%	24%	20%
6961	4	4	1%	1%	9%	1%	88%
3133	4	3	33%	0%	3%	46%	17%
4892	2	3	27%	1%	23%	29%	21%
18228	4	0	91%	0%	0%	1%	8%
7049	4	4	5%	1%	10%	3%	80%
11003	4	0	43%	0%	4%	16%	37%
22365	4	4	4%	2%	15%	3%	77%
12629	4	0	41%	1%	9%	24%	25%
6359	4	4	2%	2%	20%	10%	66%
11352	2	4	3%	1%	11%	4%	82%
13337	4	4	4%	1%	7%	11%	77%
2241	4	4	0%	1%	7%	0%	92%
13423	4	3	5%	1%	18%	47%	30%
20237	4	0	50%	0%	2%	10%	37%
5	4	4	19%	1%	4%	6%	70%
24932	4	4	2%	1%	10%	2%	86%
2345	4	4	2%	2%	13%	2%	81%
18745	4	0	45%	0%	6%	31%	18%
14677	4	4	0%	1%	3%	0%	96%
16774	4	4	7%	0%	4%	6%	84%
9843	4	4	0%	1%	3%	0%	96%
10082	4	4	0%	1%	3%	0%	96%
24382	4	4	1%	1%	13%	1%	84%
3503	4	4	8%	1%	8%	28%	55%
11977	4	3	30%	1%	18%	30%	21%

This table shows that when the actual outcome of Y_test was 1, the model did not predict it was 1 (refer to sample 21783). However the model never misdiagnosed an outcome as type 1, which is why it had high accuracy for predicting outcome type 1. Out of over 26,000 samples in the dataset, only about 180 samples were outcome type 1. Outcome type 1 was the type that occurred the least in the entire dataset, which explains why the model almost never considered it as an outcome. Instead, the model mostly predicts it to be outcome type 4, the outcome type that occurred the most in the dataset.

The table also shows that the model was good at predicting when an outcome type was 4. However, this also reveals that an outcome of type 4 may also be confused for outcome of type 0. Which brings us to why the model performed so poorly when predicting outcome type 0 in the next table.

	Y_test	XG_Model Prediction	Probability Outcome 0	Probability Outcome 1	Probability Outcome 2	Probability Outcome 3	Probability Outcome 4
25553	3	0	39%	1%	5%	21%	35%
19858	0	0	91%	0%	0%	1%	8%
15722	0	0	87%	0%	1%	1%	10%
22987	3	0	46%	0%	2%	32%	20%
13930	0	0	87%	0%	1%	1%	11%
14332	0	0	39%	0%	3%	38%	20%
15858	0	0	56%	0%	1%	19%	24%
20026	3	3	29%	0%	15%	37%	19%
1435	0	0	41%	0%	13%	28%	18%
13320	0	3	30%	0%	16%	32%	22%
5909	0	0	58%	1%	5%	18%	19%
8656	3	0	48%	0%	2%	32%	18%
12903	0	0	60%	0%	1%	15%	24%
25647	0	4	19%	1%	4%	8%	69%
19877	0	3	31%	0%	9%	45%	15%
11816	3	0	53%	0%	1%	27%	19%
2928	0	0	88%	0%	0%	1%	11%
21461	3	0	49%	0%	2%	24%	25%
973	0	0	87%	0%	1%	1%	11%
20718	0	0	97%	0%	0%	1%	2%
5234	0	0	59%	0%	4%	27%	10%
3155	0	0	63%	0%	1%	9%	28%
13300	0	0	88%	0%	0%	1%	11%
10307	0	0	52%	0%	2%	40%	6%
17301	3	3	33%	1%	4%	35%	28%
1201	3	3	7%	0%	11%	41%	41%
17345	3	3	4%	0%	15%	47%	34%
22762	0	0	93%	0%	1%	1%	6%
3806	0	0	51%	0%	2%	24%	22%
13101	3	0	65%	0%	1%	14%	20%
2317	0	3	38%	0%	3%	42%	16%
14602	0	0	49%	1%	5%	13%	32%
3637	3	0	64%	0%	1%	29%	5%

From the table above, one can see that the XGBoost model was not performing well at differentiating between Outcome Type 0 and Outcome Type 3. To see this closely refer to samples 25553, 22987, and 3637. One can also see in sample 25647 that the model would confuse Outcome Type 0 with Outcome Type 4 as well. This is understandable because type 4 is the most popular type in the dataset. This problem is most likely due to there not being enough features to distinguish between types 0 and 3, they have too many overlapping features in the dataset.

Random Forest - Kristen Marenco

Random Forest was chosen to use because it generally produces a good predictive model. Random Forest usually generates an accurate predictive model because of the way that it builds its model. It uses a diverse set of decision trees, then with bagging and voting, generates the best possible model by morphing all trees together to catch any errors. Random Forest works relatively well when other models are underfitting the dataset.

It was decided to use the parameter of **n_estimators set to be 31**, after testing the value on a range of numbers from 0 to 50. The best accuracy value was found to be at 31.

```
Accuracy value for n_estimators =  21  is:  63.354780787071675 %
Accuracy value for n_estimators =  22  is:  63.41463414634146 %
Accuracy value for n_estimators =  23  is:  63.63908424360317 %
Accuracy value for n_estimators =  24  is:  63.758790962142754 %
Accuracy value for n_estimators =  25  is:  63.63908424360317 %
Accuracy value for n_estimators =  26  is:  63.69893760287296 %
Accuracy value for n_estimators =  27  is:  63.609157563968274 %
Accuracy value for n_estimators =  28  is:  63.93835103995211 %
Accuracy value for n_estimators =  29  is:  63.90842436031722 %
Accuracy value for n_estimators =  30  is:  63.953314379769566 %
Accuracy value for n_estimators =  31  is:  64.04309441867426 %
Accuracy value for n_estimators =  32  is:  63.87849768068234 %
Accuracy value for n_estimators =  33  is:  64.08798443812658 %
Accuracy value for n_estimators =  34  is:  63.80368098159509 %
Accuracy value for n_estimators =  35  is:  63.7139009426904 %
```

The above image shows the results for the different values of the n_estimators parameter.

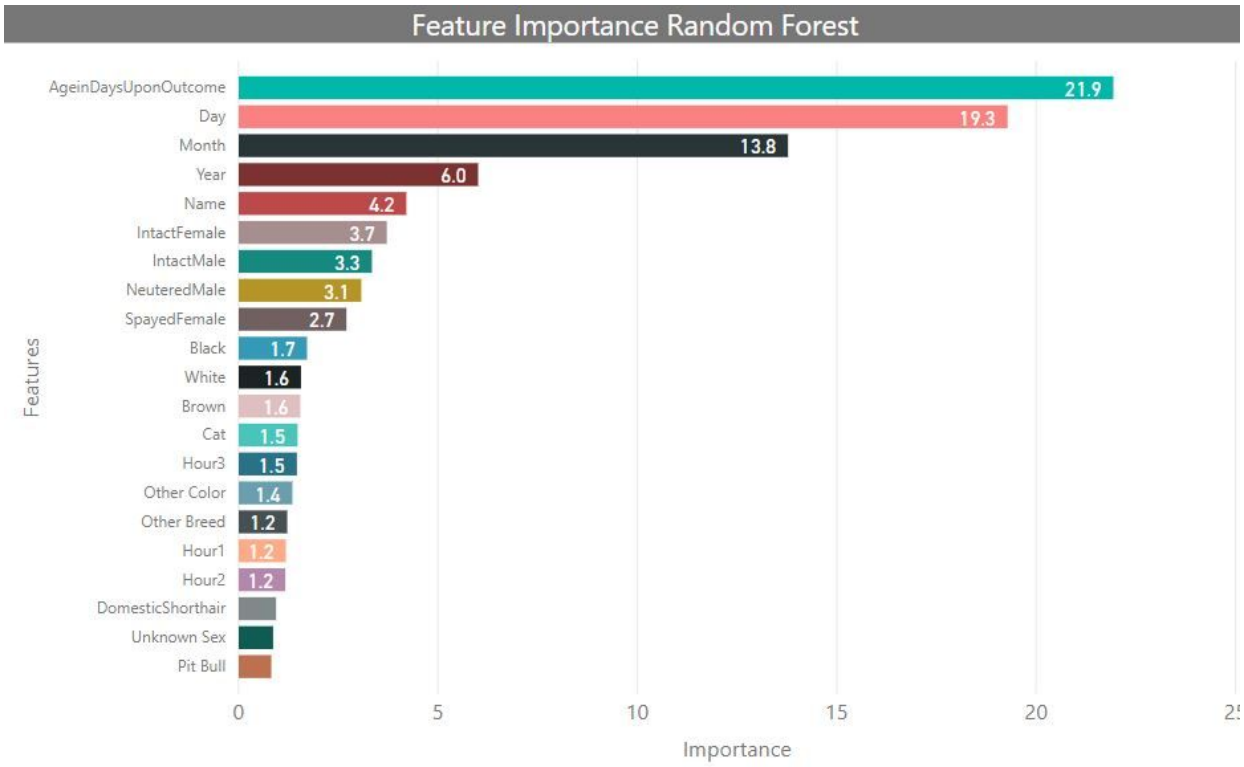
For this project, **Random Forest had initially yielded an accuracy of 64.04%**. To get a more precise reading of accuracy, 10-fold cross validation was performed.

Cross Validation for Random Forest

```
1  cv_rf = cross_val_score(rf, X_train, y_train, cv = 10, scoring='accuracy')
2
3  print(cv_rf)
4
5  cv_rf_mean = cv_rf.mean()
6
7  print("\n The mean accuracy score: ", cv_rf_mean*100, "%")
```

```
[0.62961117 0.61665005 0.62163509 0.63609172 0.63110668 0.62543641
 0.63023952 0.60578842 0.62437562 0.63618191]

 The mean accuracy score:  62.57116602362878 %
```

Random Forest supports the feature importance function. The following image shows the results:



From the Feature Importance Random Forest chart, one can see that **the top three performing features are, *AgeinDaysUponOutcome*, *Day,* and *Month***.

Once the top performing features were revealed, feature selection was incorporated in efforts to improve the accuracy of the model. The initial accuracy score was roughly 64%, which is not ideal.

The feature selection function suggested using the following features:

```
for feature_list_index in sfm.get_support(indices=True):
    print(features[feature_list_index])
```

```
Name
AgeinDaysUponOutcome
Day
Month
Year
```

The results from feature selection are as follows:

```
important_pred = rf_important.predict(X_important_test)

important_accuracy_score = accuracy_score(y_test, important_pred)

print("The accuracy score of feature selection ", important_accuracy_score*100, "%")
```

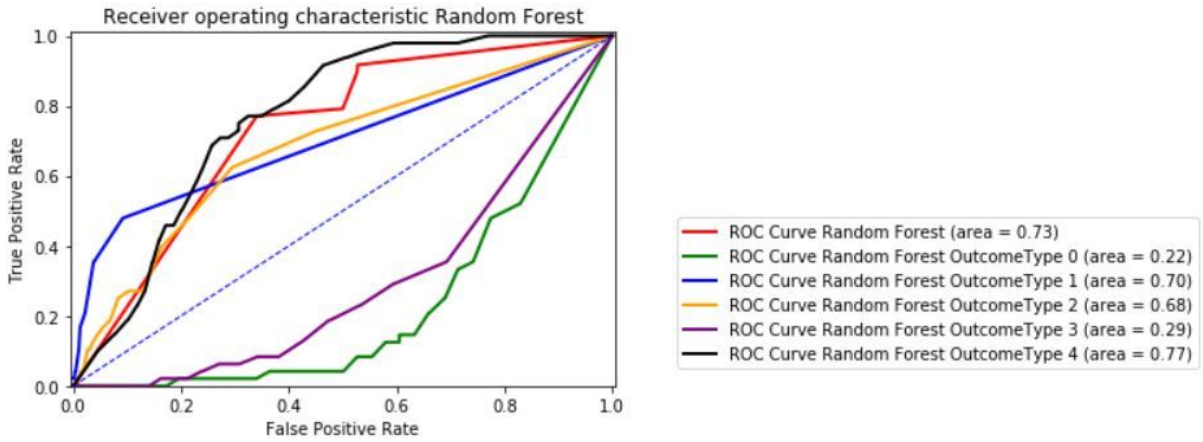
```
The accuracy score of feature selection  53.47897650755649 %
```

The **accuracy score decreased after feature selection**. This showed that the random forest predictive model was underfitting the data. However, to make sure PCA was performed.

```
rf_pca = RandomForestClassifier(n_estimators=31, bootstrap = True, random_state=2)

rf_pca.fit(X_train_p, y_train_p)

rf_pca_y_predict = rf_pca.predict(X_test_p)

print(rf_pca_y_predict)

rf_pca_score = accuracy_score(y_test_p, rf_pca_y_predict)

print("Random Forest PCA Accuracy: ", rf_pca_score*100, "%")
```

```
[4 0 0 ... 0 0 0]
Random Forest PCA Accuracy:  62.0978602424061 %
```

The results from PCA also showed that the Random Forest model was underfitting the data. Unfortunately, it was not possible to increase the accuracy by much for the Random Forest predictive model.



The ROC curve line graph shows that the Random Forest model, **with an area under the curve of 0.73** was performing fair, but not good. The ROC curve shows the Random Forest model was fair at predicting Outcome Types 1, 2, and 4. The Random Forest model was worthless at predicting Outcome Types 0 and 3.

Probability Analysis

For further inspection as to why the model was fair at predicting Outcome Types 1, 2, and 4 the following table shows a more detailed view of how the model performed:

	Y_test	Random Forest Prediction	Probabilty Outcome 0	Probability Outcome 1	Probability Outcome 2	Probabilty Outcome 3	Probability Outcome 4
19426	4	4	10%	3%	29%	0%	58%
11609	1	4	0%	3%	35%	3%	58%
21290	4	4	0%	0%	3%	0%	97%
10517	4	4	6%	0%	10%	0%	84%
12963	4	4	0%	0%	0%	0%	100%
21012	4	4	35%	0%	0%	0%	65%
127	4	4	0%	0%	6%	0%	94%
16935	4	4	3%	0%	26%	23%	48%
17290	4	4	0%	0%	23%	0%	77%
17582	4	4	0%	0%	39%	0%	61%
22919	4	4	19%	0%	6%	0%	74%
16660	2	4	0%	0%	13%	0%	87%
9589	2	4	0%	0%	26%	3%	71%
26711	4	4	39%	0%	3%	13%	45%
19453	4	4	0%	0%	0%	0%	100%
23386	4	4	35%	0%	0%	16%	48%
20763	1	1	0%	81%	0%	0%	19%
2109	4	4	3%	0%	16%	6%	74%
24819	4	4	0%	0%	13%	10%	77%
23852	4	4	0%	0%	0%	0%	100%
26603	4	4	6%	3%	13%	3%	74%
8318	2	4	0%	0%	0%	0%	100%
19875	2	2	3%	3%	61%	3%	29%
3760	4	4	6%	3%	23%	13%	55%
15344	4	4	0%	0%	10%	0%	90%
11577	4	4	3%	0%	0%	6%	90%
19963	4	4	0%	0%	3%	0%	97%
22137	4	4	0%	0%	0%	6%	94%
25466	4	4	0%	0%	23%	0%	77%
9670	4	4	19%	0%	0%	32%	48%
2565	4	4	6%	3%	42%	3%	45%
18339	4	4	0%	6%	0%	0%	94%
23837	4	4	0%	0%	0%	0%	100%

The table above shows that the Random Forest model was able to predict the probability of Outcome Type 4 to be 100%, which is impressive. It was also able to correctly predict the Outcome Type 1 when it occurred. Outcome Type 1 is difficult to catch because it occurs infrequently in the training dataset, it makes up 0.7% of the training dataset. This is also quite impressive. The table also shows that the model was able to correctly predict Outcome Type 2. Although it did confuse type 4 for 2 for sample 9589, with a probability of 26% for type 2 and 71% for type 4. This was most likely due to type 4 being the most popular Outcome Type, and sharing the most characteristics with the other Outcome Types.

	Y_test	Random Forest Prediction	Probabilty Outcome 0	Probability Outcome 1	Probability Outcome 2	Probabilty Outcome 3	Probability Outcome 4
11074	0	3	35%	0%	10%	39%	16%
7121	3	0	58%	0%	3%	16%	23%
2552	3	0	52%	0%	0%	23%	26%
14939	0	0	81%	0%	6%	6%	6%
6027	0	0	71%	0%	0%	23%	6%
2375	0	0	74%	0%	0%	25%	1%
16565	0	3	13%	0%	6%	55%	26%
9317	3	3	10%	0%	29%	55%	6%
6916	3	0	71%	0%	0%	29%	0%
26384	0	3	45%	0%	3%	52%	0%
11899	0	0	97%	0%	0%	0%	3%
11103	0	0	42%	0%	3%	23%	32%
26393	0	0	48%	0%	10%	19%	23%
8261	0	4	35%	0%	0%	19%	45%
6719	0	0	100%	0%	0%	0%	0%
23924	3	0	68%	0%	0%	19%	13%
8102	3	0	39%	0%	13%	35%	13%
17330	0	4	29%	0%	16%	19%	35%
531	3	0	58%	0%	0%	23%	19%
25372	0	0	45%	0%	0%	16%	39%
21552	0	0	97%	0%	0%	0%	3%
22825	0	0	39%	0%	3%	29%	29%
2085	0	0	52%	0%	0%	26%	23%
10462	0	0	39%	0%	3%	19%	39%
23586	0	4	6%	0%	0%	0%	94%
19774	0	0	94%	3%	0%	3%	0%
12789	0	0	100%	0%	0%	0%	0%
20873	0	0	68%	0%	0%	6%	26%

The table above shows how the Random Forest model predicted Outcome Types 0 and 3, the two types it was horrible at predicting. The table shows that the algorithm would get the two types confused with each other. Sample 11074 with actual Outcome Type 0, diagnosed as Outcome Type 3, having probability of 35% for type 0, a probability of 38% for type 3, and a probability of 16% for type 4. Based on the split probability for this outcome, one can see that the algorithm was not a solid resource for predicting either Outcome Type 0 or 3. This is further backed with evidence by referring to sample 17330 whose actual Outcome Type was 0, but diagnosed as Outcome Type 4. This sample had a probability of 29% for type 0, 16% for type 2, 19% for type 3 and, 35% for type 4. This shows that the model as unsure how to diagnose the Outcome Types 0 and 3.

Decision Tree - Angelo Onato

When using the decision tree classifier, the default setting for information gain is set to gini.

In [183]:
```
y_predict_dt = my_decisiontree.predict(X_test)
decTreeProb = my_decisiontree.predict_proba(X_test)
print('Decision Tree: ',y_predict_dt)
print('Probability: ',decTreeProb)
```

```
Decision Tree:  [0 0 0 ... 4 0 4]
Probability:  [[1. 0. 0. 0. 0.]
 [1. 0. 0. 0. 0.]
 [1. 0. 0. 0. 0.]
 ...
 [0. 0. 0. 0. 1.]
 [1. 0. 0. 0. 0.]
 [0. 0. 0. 0. 1.]]
```

In [185]:
```
from sklearn.metrics import accuracy_score

accScoreTree = accuracy_score(y_test, y_predict_dt)
print('Tree:', accScoreTree)
```

```
Tree: 0.5651653449049828
```

The resulting accuracy estimates to 56%. Knowing that the default setting for information gain is set to gini, we can change the value into entropy in order to see if a change can occur.

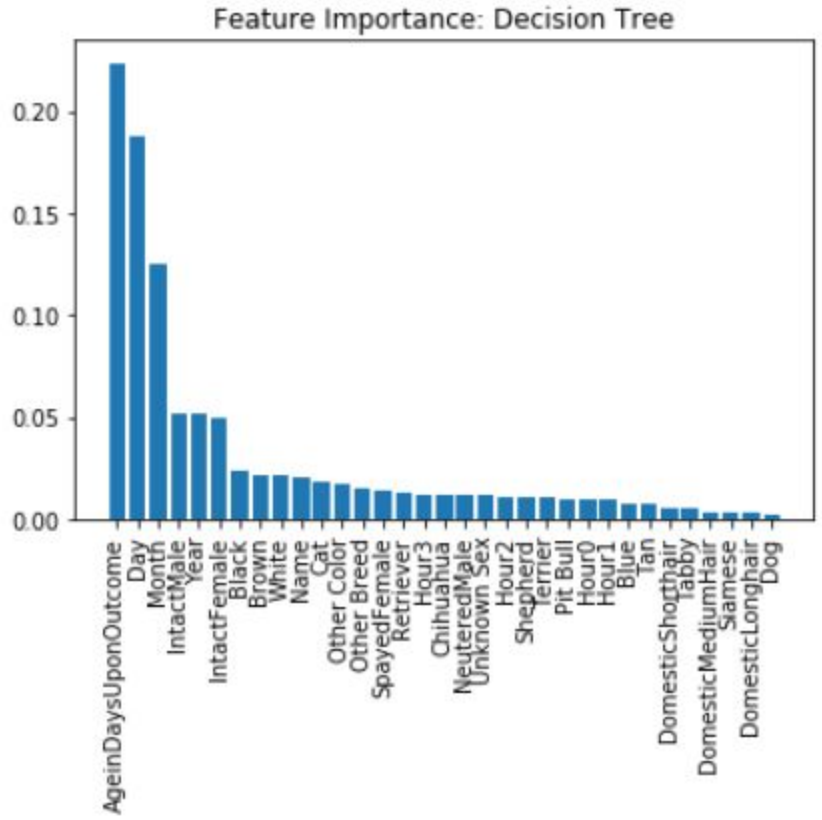
```
In [152]: #Decision Tree
          #Criterion is now entropy
          from sklearn.tree import DecisionTreeClassifier
          en_decisiontree = DecisionTreeClassifier(criterion = "entropy",random_state=4)
          en_decisiontree.fit(X_train, y_train)

Out[152]: DecisionTreeClassifier(class_weight=None, criterion='entropy', max_depth=None,
                      max_features=None, max_leaf_nodes=None,
                      min_impurity_decrease=0.0, min_impurity_split=None,
                      min_samples_leaf=1, min_samples_split=2,
                      min_weight_fraction_leaf=0.0, presort=False, random_state=4,
                      splitter='best')

In [153]: y_predict_dt_ent = en_decisiontree.predict(X_test)
          print('Decision Tree: ',y_predict_dt_ent)

          Decision Tree:  [0 0 0 ... 3 0 4]

In [154]: accScoreTreeEnt = accuracy_score(y_test, y_predict_dt_ent)
          print('Decision Tree with entropy information gain:', accScoreTreeEnt)

          Decision Tree with entropy information gain: 0.5726470148137064
```

By changing the information gain into entropy, the accuracy of the decision tree went up by 1% resulting into 57%. Though it wasn't much of an increase it shows there seems to be a difference with using gini or entropy for information gain.

```
from sklearn.metrics import accuracy_score

accScoreTree = accuracy_score(y_test, y_predict_dt)
print('Tree:', accScoreTree)

from sklearn import cross_validation
from sklearn.cross_validation import train_test_split
cross_tree = cross_validation.cross_val_score(my_decisiontree, X_test, y_test, cv = 10, scoring="accura
print(cross_tree)
mean_decTreecv = cross_tree.mean()
print("Mean accuracy: ",mean_decTreecv *100)

from sklearn.metrics import classification_report
print (classification_report(y_test,y_predict_dt))
```

```
Tree: 0.5620230435433189
[0.53800298 0.55074627 0.54857997 0.52167414 0.54341317 0.53293413
 0.54872564 0.56221889 0.57271364 0.56371814]
Mean accuracy:  54.82726977128804
             precision    recall  f1-score   support

          0       0.66      0.65      0.65      2666
          1       0.13      0.17      0.15        41
          2       0.24      0.24      0.24       410
          3       0.34      0.35      0.34      1162
          4       0.64      0.63      0.64      2404

avg / total       0.57      0.56      0.56      6683
```

By further analyzing what went behind the scenes of the decision tree, we check the classification report. In the report we our given four scores for out outcomes. By observing the support column, outcomes 1 and 2 is significantly less compared to the others. In this report, support is the number of occurrences each outcome has in the dataset. By taking this to account, there seems to be some unbalance in the data. Another observation to take in is the f1-score. The f1-score is the mean of both the precision and recall scores. For outcomes 0 and 4, the f1-scores stay above 50% which is okay. But outcomes 1, 2, and 3 f1-scores resulted below 50%. Knowing how outcomes 1, 2, and 3 have low f1-score, they seem to have problems with their precision and recall scores. This means that those three outcomes had some sort of error with there true positive labels. For example, when looking at outcome 1 recall score only 17% of true positives were classified correctly. This shows that the decision tree classifier had some problems when handling the data as there seem to be problems when labeling some data as true positive.



After implementing the decision tree and recording it's accuracy, we wanted to see what feature was important compared to the other classifiers. When analyzing the important features in the decision tree, it seems to be that some features were important in than orders when placed onto a bar graph. The top three features in decision tree were AgeinDaysUponOutcome, Day, and Month.

AdaBoost - Angelo Onato

Knowing how decision tree resulted on a low accuracy, the next step in the process was to increase it's score. With decision tree being classified as a weak learner, boosting it to make it a strong learner will require adaboost. By using adaboost, we hope to obtain a more higher accuracy score when using decision tree by itself.

```
# adaBoost
from sklearn.ensemble import AdaBoostClassifier

my_AdaBoost = AdaBoostClassifier(DecisionTreeClassifier())
my_AdaBoost.fit(X_train, y_train)
```

```
AdaBoostClassifier(algorithm='SAMME.R',
          base_estimator=DecisionTreeClassifier(class_weight=None, criterion='gini', max_depth=None,
            max_features=None, max_leaf_nodes=None,
            min_impurity_decrease=0.0, min_impurity_split=None,
            min_samples_leaf=1, min_samples_split=2,
            min_weight_fraction_leaf=0.0, presort=False, random_state=None,
            splitter='best'),
          learning_rate=1.0, n_estimators=50, random_state=None)
```

```
y_predict_ada = my_AdaBoost.predict(X_test)
y_predict_ada
```

```
array([0, 0, 0, ..., 3, 0, 4], dtype=int64)
```

```
ada_score = accuracy_score(y_test, y_predict_ada)
print('AdaBoost Score: ',ada_score)
```

```
AdaBoost Score:  0.6301062397127039
```

After using adaboost on the decision tree, the accuracy score is now set at an estimate of 63%. Previously, decision tree accuracy was at an estimate of 57% and after implementing adaboost, the score has increased by 6%. The purpose of using adaboost was to increase the score of the decision tree and it fulfilled that purpose.

KNN - Andy Xie

When using KNN it is important to figure out what parameters we should use for k. To do this, created a grid using GridSearchCV to find the to find the best parameters for “n_neighbors”. We also used 10 fold cross validation on the grid to ensure that the accuracy is correct. We limited the range of k to be within 1 - 30 and increment 3 steps each time since it takes a long time to test many parameters especially with 10 fold cross validation on each of them.

In [204]:
```
from sklearn.neighbors import KNeighborsClassifier
from sklearn import metrics
from sklearn.model_selection import GridSearchCV

knn_accuracies = []

k = list(range(1, 30, 3))
param_grid = dict(n_neighbors=k)
grid = GridSearchCV(knn_model, param_grid, cv=10, scoring='accuracy')
grid.fit(X_train_new, y_train)
grid.cv_results_['mean_test_score']
```

Out[204]:
```
array([0.57470888, 0.60332975, 0.62610485, 0.62750783, 0.62951878,
       0.63288594, 0.63714165, 0.63447599, 0.63550484, 0.63377449])
```

The highest accuracy we obtained was an accuracy of 0.637142 using n_neighbors=19

```
In [205]: print(grid.best_score_)
          print(grid.best_params_)

          0.6371416545854183
          {'n_neighbors': 19}
```

Logistic Regression - Andy Xie

Logistic regression achieved a little higher accuracy than KNN. Like KNN, we also used 10 fold cross validation to find the true accuracy.

```
In [393]: from sklearn.linear_model import LogisticRegression
          from sklearn.model_selection import cross_val_score

          lr_model = LogisticRegression()

          cv_results = cross_val_score(lr_model, X_train_new, y_train, cv=10)
          print("Logistic Regression Accuracy: ", np.mean(cv_results))
```

For logistic regression, we got an accuracy of 0.65276

```
Logistic Regression Accuracy:  0.6527637554366572
```

ANN - Andy Xie

For the ANN we used SciKitLearn's MLPClassifier. We used 3 layers with the first layer having 6 neurons, second layer having four neurons, and third layer having 3 layers. We set the batch size to be auto and max_iterations=200 and performed 10 fold cross validation.

```
from sklearn.neural_network import MLPClassifier

ann_model = MLPClassifier(hidden_layer_sizes=(6, 4, 3), activation= 'logistic',
                      solver='adam', alpha=1e-5, random_state=1,
                      learning_rate_init = 0.1, batch_size='auto', max_iter=200)

accuracy_list = cross_val_score(ann_model, X, y, cv=10, scoring='accuracy')

print("ANN Accuracy: ", np.mean(accuracy_list))
```

Surprisingly, ANN had the lowest accuracy with an accuracy of 0.462302

```
ANN Accuracy:  0.46230217332091106
```

One of the problems could be having incorrect parameters for the number of hidden layers and number of neurons.

Support Vector Machine (SVM) - Imelda Flores

Support Vector Machine (SVM) was chosen because it is 'one of the most accurate supervised learning method.' In some cases it performs better than ANN. Since the dataset is small, it could perform better with better results. The two types of SVM were used on the dataset to find the best type to use on the dataset.

The data was separated into a test size of 25% (0.25) and 75% (0.75) training size. The random state used was 4. After One-Hot Encoding, 32 features were used on the algorithm. The features include: 'AgeinDaysUponOutcome', 'Day', 'Month', 'Year', 'Hour0', 'Hour1', 'Hour2', 'Hour3', 'Cat', 'Dog', 'IntactFemale', 'IntactMale', 'NeuteredMale','SpayedFemale', 'Unknown Sex', 'Pit Bull', 'Chihuahua', 'Shepherd','Retriever', 'Terrier', 'DomesticShorthair', 'DomesticMediumHair', 'DomesticLonghair', 'Siamese', 'Other Breed', 'Black', 'Brown', 'White','Tan', 'Blue', 'Tabby', 'Other Color'.

Linear SVM was used first to calculate the accuracy. Linear SVM finds the Maximum Margin. The margin is the widest area before hitting a data point. To find the margin the following formula is used:

$$\mathbf{M} = \frac{2}{||\boldsymbol{w}||}$$

To find the parallel lines (the separation between the two classes of data) the following formulas were used:

Above the margin: $\boldsymbol{w}^T \boldsymbol{x} + b = 1$

Below the margin: $\boldsymbol{w}^T \boldsymbol{x} + b = -1$

SVM with a linear model and 32 features had an accuracy score of 0.6307047733054018.

In the features listed above the 'Name' column is not listed because it was determined that it will not affect the accuracy of the algorithm. The 'Name' column would not change the accuracy since it wasn't important if the animal had a name or not. Since every name is unique and there are many names one-hot encoding will create multiple columns, making the dataset bigger. It was decided to change the column as boolean. To not oversize the data and to include the column it was easier to incorporate. Adding the column, it would

now include 33 columns.

```
from sklearn.svm import SVC

svm_linear = svm.SVC(kernel ='linear', gamma ='auto', probability = True)
svm_linear.fit(X_train, y_train)
y_pred_linear = clf_linear.predict(X_test)
score_linear = metrics.accuracy_score(y_test, y_pred_linear)
print(score_linear)
```

```
0.6381864432141254
```

The accuracy score of a linear SVM increased by a value of 0.008 to 0.6381864432141254.

To improve the accuracy of the linear SVM 10 fold cross validation was used. Using cross validation will give a more precise accuracy score since it will use a different testing set each time.

```
from sklearn.model_selection import cross_val_score

accuracy_list_clf_linear = cross_val_score(clf_linear, X_train, y_train, cv = 10, scoring = 'accuracy')
print(accuracy_list_clf_linear)
```

```
[0.6492277  0.6492277  0.63926258 0.63559322 0.63441397 0.63023952
 0.63205192 0.64702946 0.62637363 0.62387612]
```

```
accuracy_cv_clf_linear = accuracy_list_clf_linear.mean()

print(accuracy_cv_clf_linear)
```

```
0.6367295821612541
```

The score using cross validation the accuracy score was 0.6367295821612541. The score using cross validation did not improve the accuracy. Cross validation decreased the score from 33 features and no cross validation.

In order to improve the accuracy score of linear SVM feature reduction was applied. Features were chosen by the effect it had on the accuracy. The top seven features that made the most impact on the accuracy were used: 'Name', 'Unknown Sex', 'Neutered Male', 'Terrier', 'Spayed Female', 'Intact Male', and 'Intact Female' with 10 fold cross validation, test size of 0.25 and random state of 4.

The best accuracy was 0.6078429006279485 features for the top five features: 'Name', 'Unknown Sex', 'Neutered Male', 'Terrier', and 'Spayed Female'. After adding the 'Intact Male' feature the accuracy did not change. Using all seven features the accuracy did not change.

```
features:  ['Name', 'Unknown Sex', 'NeuteredMale', 'Terrier', 'SpayedFemale']

average is  0.6078429006279485


features:  ['Name', 'Unknown Sex', 'NeuteredMale', 'Terrier', 'SpayedFemale', 'IntactMale']

average is  0.6078429006279485
```

```
features:  ['Name', 'Unknown Sex', 'NeuteredMale', 'Terrier', 'SpayedFemale', 'IntactMale', 'IntactFemale']

average is  0.6078429006279485
```

The best accuracy score for linear SVM saw with 33 features and no cross validation.

The second type of SVM the kernel was changed to 'rbf'. This method is used to map the data where it can be linearly separable. The following equation is used to find the Gaussian Radial Function (RBF):

$$K(\boldsymbol{x}_i, \boldsymbol{x}_j) = \exp(-\frac{\| \boldsymbol{x}_i - \boldsymbol{x}_j \|^2}{2\sigma^2})$$

The 32 features listed above were used with the same test size and random state as before. The accuracy score was 0.6372886428250786. The 'Name' column was added and the accuracy score was 0.640281310788568. The 'Name' column increased the accuracy score.

```
from sklearn.svm import SVC

rbf = SVC(kernel ='rbf', gamma ='auto', probability = True)
rbf.fit(X_train, y_train)
y_pred_rbf = rbf.predict(X_test)
score_rbf = accuracy_score(y_test, y_pred_rbf)

print(score_rbf)
```

```
0.640281310788568
```

To improve the accuracy score, 10 fold cross validation was used and resulted in an accuracy score of 0.6421176406189908. Cross validation increased the accuracy score of SVM.

```
from sklearn.model_selection import cross_val_score

accuracy_list_rbf= cross_val_score(rbf, X_train, y_train, cv = 10, scoring = 'accuracy')
print(accuracy_list_clf_rbf)
```

```
[0.65321375 0.65470852 0.64673642 0.6445663  0.63790524 0.63023952
 0.63055417 0.64852721 0.63786214 0.63686314]
```

```
accuracy_cv_rbf = accuracy_list_rbf.mean()

print(accuracy_cv_rbf)
```

```
0.6421176406189908
```

Feature reduction was applied with the most important seven features that were used in linear SVM. The features that calculated the highest accuracy score were 'Name', 'Unknown Sex', 'Neutered Male', 'Terrier' and 'Spayed Female' and the score was 0.6081790132500184 using cross validation. These were the same features that calculated the highest score in linear SVM. However, this SVM had a higher accuracy score.

In linear SVM, the accuracy score did not change when adding the features 'Intact Male', 'Intact Female'. In SVM using RBF, the accuracy score decreased after adding 'Intact Male' and did not change after adding 'Intact Female'.

```
features:  ['Name', 'Unknown Sex', 'NeuteredMale', 'Terrier']

average is  0.5404972761815047


features:  ['Name', 'Unknown Sex', 'NeuteredMale', 'Terrier', 'SpayedFemale']

average is  0.6081790132500184


features:  ['Name', 'Unknown Sex', 'NeuteredMale', 'Terrier', 'SpayedFemale', 'IntactMale']

average is  0.6080671016642001
```

The best accuracy score of SVM with a kernel of 'rbf' was using 33 features and 10 fold cross validation.

Probability

The probability of each outcome was calculated for SVM linear and non-linear. The features used were the 33 features listed above.

The probability of linear SVM is 0.9119576535365007. To improve the probability cross validation was used resulting in the probability being 0.920115873057288.

```
from sklearn import svm

SVM_linear = SVC(probability = True, gamma ='auto', kernel ='linear')
SVM_linear.fit(X_train, y_train)
y_predict_linear = SVM_linear.predict_proba(X_test)
score_svm = metrics.log_loss(y_test, y_predict_linear)

print(score_svm)
```

```
0.9119576535365007
```

The probability of non-linear SVM is 0.9119576535365007. Cross validation was used to improve the results 0.9102181729572573.

```
from sklearn import svm

SVM_rbf = SVC(probability = True, gamma ='auto', kernel ='rbf')
SVM_rbf.fit(X_train, y_train)
y_predict_rbf = SVM_rbf.predict_proba(X_test)
score_rbf = metrics.log_loss(y_test, y_predict_rbf)

print(score_rbf)
```

```
0.9119576535365007
```

The best probability was Linear SVM with cross validation. The worst probability was non-linear SVM with cross validation.

Probability vs accuracy

The following table shows the results of Linear SVM prediction of outcome and the percentage probability of an outcome. For example, outcome 0 has 69% probability of happening. Shows the comparison of accuracy to probability for the test data.

	y_test	linear_pred	Probability outcome 0	Probability outcome 1	Probability outcome 2	Probability outcome 3	Probability outcome 4
12015	Adoption	Adoption	0.699623849	0.002900704	0.009707356	0.062823872	0.224944219
15273	Adoption	Adoption	0.708450336	0.003387005	0.00893011	0.069757964	0.209474584
21964	Adoption	Adoption	0.621004624	0.009756046	0.033604853	0.119075012	0.216559466
12191	Adoption	Adoption	0.467325005	0.001690239	0.017543148	0.369058099	0.144383509
12615	Adoption	Adoption	0.717825263	0.003232484	0.012053756	0.051662429	0.215226068
6079	Transfer	Adoption	0.714347586	0.002587069	0.008544435	0.045066644	0.229454266
6521	Transfer	Transfer	0.040985966	0.020383355	0.08025789	0.010722733	0.847650056
21862	Transfer	Adoption	0.717441799	0.00265573	0.008366085	0.041381953	0.230154433
15582	Transfer	Return_to_owner	0.170428127	0.012496189	0.162217912	0.483338941	0.171518832
11268	Adoption	Adoption	0.520742336	0.001688177	0.011982983	0.316010132	0.149576372
17781	Adoption	Adoption	0.65062409	0.003137795	0.009974304	0.197273023	0.138990789
1968	Adoption	Adoption	0.448736888	0.001539424	0.016510673	0.382547655	0.15066536
13505	Return_to_owner	Adoption	0.595742678	0.005359751	0.039281921	0.168277765	0.191337886
20600	Transfer	Transfer	0.029825312	0.021680508	0.12189427	0.016472215	0.810127694
22634	Transfer	Adoption	0.452657821	0.003449329	0.061941569	0.335394308	0.146556974
16355	Adoption	Adoption	0.646489767	0.003840576	0.011641426	0.206046091	0.13198214
20698	Adoption	Adoption	0.583775319	0.001683732	0.009564822	0.214844503	0.190131624
20264	Adoption	Adoption	0.633136546	0.001656646	0.008116285	0.163479443	0.19361108
19589	Return_to_owner	Transfer	0.024455238	0.016986181	0.207514285	0.04419039	0.706853905
8707	Adoption	Adoption	0.715318468	0.002575611	0.007875708	0.045199919	0.229030294

The following table shows the accuracy to probability comparison of non-linear SVM of the first 20 results of the test data:

	y_test	rbf_pred	Probability outcome 0	Probability outcome 1	Probability outcome 2	Probability outcome 3	Probability outcome 4
12015	Adoption	Adoption	0.699623849	0.002900704	0.009707356	0.062823872	0.224944219
15273	Adoption	Adoption	0.708450336	0.003387005	0.00893011	0.069757964	0.209474584
21964	Adoption	Adoption	0.621004624	0.009756046	0.033604853	0.119075012	0.216559466
12191	Adoption	Adoption	0.467325005	0.001690239	0.017543148	0.369058099	0.144383509
12615	Adoption	Adoption	0.717825263	0.003232484	0.012053756	0.051662429	0.215226068
6079	Transfer	Adoption	0.714347586	0.002587069	0.008544435	0.045066644	0.229454266
6521	Transfer	Transfer	0.040985966	0.020383355	0.08025789	0.010722733	0.847650056
21862	Transfer	Adoption	0.717441799	0.00265573	0.008366085	0.041381953	0.230154433
15582	Transfer	Return_to_owner	0.170428127	0.012496189	0.162217912	0.483338941	0.171518832
11268	Adoption	Adoption	0.520742336	0.001688177	0.011982983	0.316010132	0.149576372
17781	Adoption	Adoption	0.65062409	0.003137795	0.009974304	0.197273023	0.138990789
1968	Adoption	Adoption	0.448736888	0.001539424	0.016510673	0.382547655	0.15066536
13505	Return_to_owner	Adoption	0.595742678	0.005359751	0.039281921	0.168277765	0.191337886
20600	Transfer	Transfer	0.029825312	0.021680508	0.12189427	0.016472215	0.810127694
22634	Transfer	Adoption	0.452657821	0.003449329	0.061941569	0.335394308	0.146556974
16355	Adoption	Adoption	0.646489767	0.003840576	0.011641426	0.206046091	0.13198214
20698	Adoption	Adoption	0.583775319	0.001683732	0.009564822	0.214844503	0.190131624
20264	Adoption	Adoption	0.633136546	0.001656646	0.008116285	0.163479443	0.19361108
19589	Return_to_owner	Transfer	0.024455238	0.016986181	0.207514285	0.04419039	0.706853905
8707	Adoption	Adoption	0.715318468	0.002575611	0.007875708	0.045199919	0.229030294

GridSearchCV - Imelda Flores

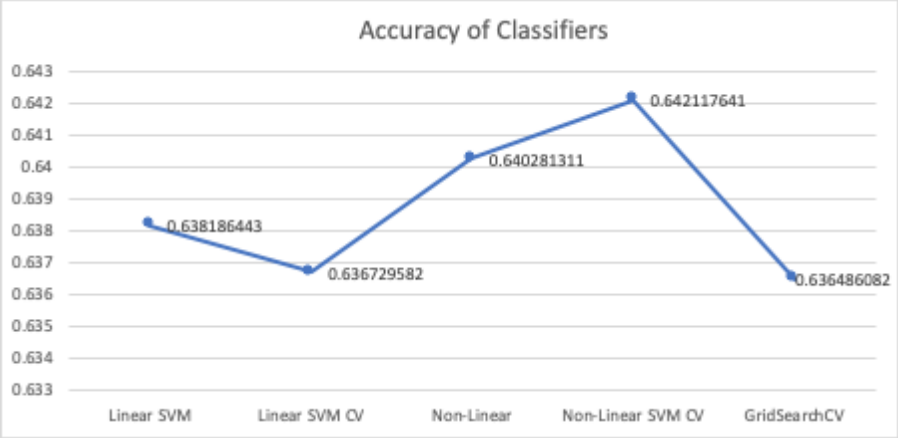
GridSearchCV was used with SVM in order to attain better results. GridSearchCV will loop through all combinations of parameters while using cross validation. Then it will find the best average of cross validation from all the possible combinations using the parameters.

GridSearchCV with a kernel set to 'rbf' had 5 types for parameters for 'C' with a 'gamma' of 0.005. The accuracy score of GridSearchCV was 0.6364860820113738 with 10 being the best value for 'C'.

```
from sklearn.model_selection import GridSearchCV

param_grid = {'C':[0.001, 0.01, 0.1, 1, 10], 'gamma' : [0.0005]}

grid_cv = GridSearchCV(svm.SVC(kernel = 'rbf'), param_grid, cv = 10)
grid_cv.fit(X_train, y_train)

print(grid_cv.best_score_)
print(grid_cv.best_params_)
```

```
0.6364860820113738
{'C': 10, 'gamma': 0.0005}
```

GridSearchCV did not have better results than linear SVM or non-linear SVM with or without cross validation.



Gradient Boosting Classifier - Imelda Flores

Gradient Boosting Classifier was chosen because it has been increasing in use in Kaggle competitions. Most Kaggle competitors use ensemble models such as Gradient Boosting. It is most commonly used with pre-built libraries. Gradient Boosting Classifier is an ensemble algorithm that is a boosting classifier.

Boosting classifiers do not make predictions independently. They make predictions sequentially and learn from previous prediction mistakes. The new predictions will be learning from mistakes made by previous predictions. It will always try to minimize the loss and define it. First the loss function will be updated based on the learning rate until it is the minimum. Gradient Boosting can overfit, reduce bias and variance, and are sequential classifiers.

The data was split with a test size set to 0.25 and random state set to 4. The training set will be 0.75.

There were 32 features used to find the accuracy which include: 'AgeinDaysUponOutcome', 'Day', 'Month', 'Year', 'Hour0', 'Hour1', 'Hour2', 'Hour3', 'Cat', 'Dog', 'IntactFemale', 'IntactMale', 'NeuteredMale','SpayedFemale', 'Unknown Sex', 'Pit Bull', 'Chihuahua', 'Shepherd','Retriever', 'Terrier', 'DomesticShorthair', 'DomesticMediumHair', 'DomesticLonghair', 'Siamese', 'Other Breed', 'Black', 'Brown', 'White','Tan', 'Blue', 'Tabby', 'Other Color'.

Gradient Boosting was used with a random state set to 1, 25 estimators and a max depth of 3. The accuracy score was 0.6363908424360317.

The column ‘Name’ was added. At first it was believed to not have an impact on the accuracy score. After it was added the accuracy score was 0.6392338770013467. The accuracy score increased with the new column.

```
from sklearn.ensemble import GradientBoostingClassifier

gradient_Boost = GradientBoostingClassifier(random_state = 1, n_estimators = 25, max_depth = 3)

gradient_Boost.fit(X_train, y_train)
prob_gradient_Boost = gradient_Boost.predict(X_test)

score_gb = accuracy_score(y_test, prob_gradient_Boost)
print(score_gb)
```

```
0.6392338770013467
```

10-fold cross validation was used to get a better sense of the accuracy score. The accuracy score using cross validation was 0.6494986699139643. The accuracy score was improved from before without cross validation. Surprisingly the highest accuracy score was on the eight iteration of cross validation. It was higher than the mean of cross validation.

```
from sklearn.model_selection import cross_val_score

accuracy_list_gb = cross_val_score(gradient_Boost, X_train, y_train, cv = 10, scoring = 'accuracy')
print(accuracy_list_gb)
```

```
[0.66716492 0.66317887 0.65520678 0.64705882 0.64389027 0.64720559
 0.63754368 0.65701448 0.63936064 0.63736264]
```

```
accuracy_cv_gb = accuracy_list_gb.mean()

print(accuracy_cv_gb)
```

```
0.6494986699139643
```

Using a built-in function to find feature importance, the seven most important features were chosen. The seven that made the most impact were 'Age in Days Upon Outcome', 'Neutered Male', 'Intact Female', 'Spayed Female', 'Name', 'Intact Male', and 'Hour 3'.

```
features = pd.Series(clf_gb.feature_importances_, index = column_name).sort_values(ascending = False)

print(features)
```

```
AgeinDaysUponOutcome    0.373618
NeuteredMale            0.151872
IntactFemale            0.097938
SpayedFemale            0.085241
Name                    0.083131
IntactMale              0.073849
Hour3                   0.031320
```

The accuracy score was calculated using cross validation and the same test size and random stats. The highest accuracy score calculated was 0.6261767093532307 when the seven features were used. The accuracy increases after each feature is added. The largest increase was when four features was used, when ‘Spayed Female’ was added.

```
features:  ['AgeinDaysUponOutcome', 'NeuteredMale', 'IntactFemale', 'SpayedFemale']

average is  0.6190666138976515

features:  ['AgeinDaysUponOutcome', 'NeuteredMale', 'IntactFemale', 'SpayedFemale', 'Name']

average is  0.6242308109998849

features:  ['AgeinDaysUponOutcome', 'NeuteredMale', 'IntactFemale', 'SpayedFemale', 'Name', 'IntactMale', 'Hour3']

average is  0.6261767093532307
```

The best accuracy using gradient boosting classifier was using 33 features and cross validation. The best accuracy found was 0.6494986699139643.

Probability

The probability was calculated using Gradient Boosting Classifier.

The probability with the 32 features listed above was 0.889350712540182. The 'Name' column was added again. The probability with 33 features was 0.8684581373234003. The probability decreased when the 'Name' column was added.

```
from sklearn.ensemble import GradientBoostingClassifier

gradient_Boost = GradientBoostingClassifier(random_state = 1, n_estimators = 25, max_depth = 3)

gradient_Boost.fit(X_train, y_train)
prob_gradient_Boost = gradient_Boost.predict_proba(X_test)

score_gb_prob = metrics.log_loss(y_test, prob_gradient_Boost)
print(score_gb_prob)
```

```
0.8684581373234003
```

To have a better probability with the 33 features listed above, 10-fold cross validation was applied to the dataset. The probability was 0.872748528796844. In the 10 cross validation iterations, the highest probability was on the seven iteration with a probability of 0.89181289.

```
from sklearn.model_selection import cross_val_score

accuracy_list_gb = cross_val_score(gradient_Boost, X_train, y_train, cv = 10, scoring = 'neg_log_loss')
print(accuracy_list_gb)
```

```
[-0.86373847 -0.8645969  -0.85898778 -0.86210023 -0.88478441 -0.88176792
 -0.89181289 -0.86261448 -0.87657595 -0.88050626]
```

```
accuracy_cv_gb = accuracy_list_gb.mean()

accuracy_gb = - accuracy_cv_gb

print(accuracy_gb)
```

```
0.872748528796844
```

The best probability was using 32 features and no cross validation. The predicted probability was 0.889350712540182.

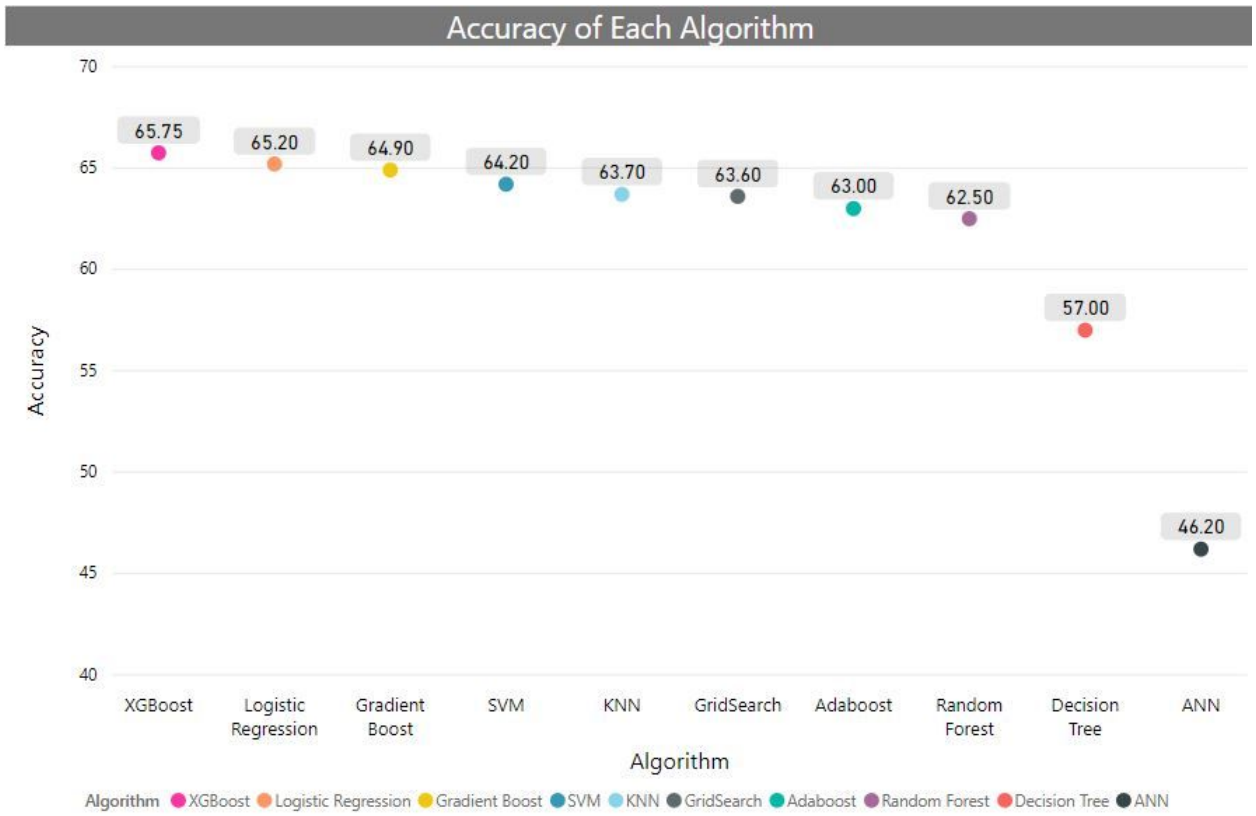
Probability vs accuracy

The following table shows the comparisons between the calculated accuracy of the outcomes and the calculated probability of each outcome. The first 20 calculations are shown with the correct outcome being under y_test and the prediction being gb_pred. The probability of each outcome is also shown:

	y_test	gb_pred	Probability outcome 0	Probability outcome 1	Probability outcome 2	Probability outcome 3	Probability outcome 4
12015	Adoption	Adoption	0.637530346	0.020465234	0.037244679	0.056673026	0.248086714
15273	Adoption	Adoption	0.903308118	0.011092657	0.017623222	0.023859144	0.04411686
21964	Adoption	Adoption	0.707741913	0.016209736	0.031147532	0.047755968	0.197144851
12191	Adoption	Return_to_owner	0.315587039	0.017331073	0.041191372	0.431634305	0.19425621
12615	Adoption	Adoption	0.906966733	0.010798918	0.017735615	0.021574172	0.042924562
6079	Transfer	Adoption	0.803042517	0.012783407	0.020309366	0.026877234	0.136987475
6521	Transfer	Transfer	0.033129606	0.017864579	0.055008168	0.017450365	0.876547282
21862	Transfer	Adoption	0.850864679	0.009515465	0.015627735	0.018582433	0.105409689
15582	Transfer	Return_to_owner	0.281495586	0.021707009	0.166630825	0.315885365	0.214281214
11268	Adoption	Return_to_owner	0.346255827	0.017779587	0.04225737	0.39442382	0.199283396
17781	Adoption	Adoption	0.745576585	0.015854059	0.02796827	0.130963726	0.07963736
1968	Adoption	Return_to_owner	0.343891128	0.016715843	0.039729133	0.40922763	0.190436266
13505	Return_to_o	Adoption	0.422560671	0.023318574	0.055153894	0.228863657	0.270103204
20600	Transfer	Transfer	0.026133021	0.016637026	0.114251221	0.032970884	0.810007848
22634	Transfer	Return_to_owner	0.314080827	0.016127461	0.123356789	0.362701838	0.183733085
16355	Adoption	Adoption	0.876260874	0.010760516	0.01709554	0.050725478	0.045157593
20698	Adoption	Adoption	0.498602247	0.019227867	0.033789437	0.229325795	0.219054654
20264	Adoption	Adoption	0.534857652	0.018761055	0.032969099	0.199675731	0.213736462
19589	Return_to_o	Return_to_owner	0.029073791	0.020168494	0.046570541	0.466903045	0.437284129
8707	Adoption	Adoption	0.817165025	0.011800361	0.019380336	0.02304453	0.128609748

Team Results

The following diagram shows the comparisons of the accuracy scores calculated by all algorithms implemented for this project:



Based on the results from all algorithms, the highest accuracy calculated was from the XGBoost algorithm. XGBoost with feature selection resulted in a 65.75% accuracy. Feature selection improved XGBoost since there were less features used the algorithm performed better. The second highest accuracy calculated was Logistic Regression with cross validation with an accuracy of 65.27%. Cross validation increased the accuracy of Logistic Regression since it provided multiple iterations with a variety of sets.

The algorithm that calculated the worst accuracy was ANN using cross validation resulted with a 46.23% accuracy. Compared to the second lowest accuracy score calculated by Decision Tree of a 57% accuracy. ANN had a very low accuracy score affected by the data not being big enough or no pattern between the dataset is made.

Overall, not including the worst accuracy calculated by the two algorithms, ANN and Decision Tree, the accuracy scores were close in accuracy. The accuracy slowly decreases with each algorithm.

After analyzing each of the algorithms, it was determined that the reason for such low accuracy overall was due to the dataset itself not being the best to work with. There were not enough samples of Outcome Type 1 to correctly predict that outcome.

Another reason the dataset was not the best, was because the column *Datetime* which was changed into *Day, Month, Year,* and *Hour*, were all considered data leakage. The only reason that the column was used was because it was listed to be used on the Kaggle website. In each feature importance analysis section of each algorithm, one could see that *Datetime* was always contributing substantially to each predictive model. However, when removing the column the accuracy of an algorithm would decrease spectacularly. It was unrealistic to be provided with *Datetime* because in the real world one would never know when an animal's Outcome Type would occur upon registering the animal.

In the dataset. column *Color* was broad. It had many colors including colors set together. An animal can be four colors together. The set of colors made for different possibilities for an animal and difficult to calculate the accuracy using all mixtures of color. There were over 1,000 unique possibilities, including unique sets of colors.

Lastly, there were not enough differences among the Outcome Types to differentiate between them using the features provided. In order to increase accuracy performance of other predictive models the dataset should include more distinguishable features.

Conclusion

For this project, we had to experiment with many different solutions to improve accuracy as well as learning how to use different models for machine learning. The biggest problem for our project was figuring out how to modify the dataset to something we could work with. For example, two features of our dataset were "Color" and "Breed" and performing OneHotEncoding on these datasets would result in multiple columns since there are many different combinations of colors and breeds. One of the things we realized during our project was how hard it is to generate a high accuracy despite our efforts to find the best parameters and modification of the dataset. In class labs and homeworks, when working with the given datasets we would usually get an accuracy about ~95%, but for this project most of our models only yielded about ~65% accuracy when we expected accuracy scores similar to the class labs/homework accuracies. Overall, this project proved to be a valuable resource in learning about machine learning and how to work with real world data like a true data scientist.

References

Montclair Dispatch Article:

https://montclairdispatch.com/animal-shelter-montclair/

Kaggle Project Description:

https://www.kaggle.com/c/shelter-animal-outcomes

Towards Data Science Articles:

https://towardsdatascience.com/support-vector-machine-introduction-to-machine-learning-algorithms-934a444fca47

https://towardsdatascience.com/a-one-stop-shop-for-principal-component-analysis-5582fb7e0a9c

Machine Learning Mastery Articles:

https://machinelearningmastery.com/feature-importance-and-feature-selection-with-xgboost-in-python/

https://machinelearningmastery.com/gentle-introduction-gradient-boosting-algorithm-machine-learning/

Stack Abuse Cross Validation Article:

https://stackabuse.com/cross-validation-and-grid-search-for-model-selection-in-python/

Machine Learning Article - Chris Albon:

https://chrisalbon.com/machine_learning/model_evaluation/cross_validation_parameter_tuning_grid_search/

UNMC Article:

http://gim.unmc.edu/dxtests/roc3.htm

Gradient Boosting article:

https://medium.com/mlreview/gradient-boosting-from-scratch-1e317ae4587d

Classification Report:

https://www.scikit-yb.org/en/latest/api/classifier/classification_report.html