Predict the unknown binary variable

Goal: to predict a 0/1 for each row as accurately as possible. How should we start?

Things to keep in mind:

- What are we even classifying?
- How is that related to the different features?

Data exploration/visualization/cleaning

In [1]:
```
# Imports:
import pandas as pd
import matplotlib

import matplotlib.pyplot as plt
import seaborn as sns
import numpy as np

from scipy import stats
from scipy.stats import norm
from scipy.stats import skew

from sklearn.preprocessing import StandardScaler

# Required Python Packages
import pandas as pd
from sklearn.model_selection import train_test_split
from sklearn.ensemble import RandomForestClassifier
from sklearn.metrics import accuracy_score
from sklearn.metrics import confusion_matrix


import warnings
warnings.filterwarnings('ignore')

%matplotlib inline
```

In [2]:
```
# get data
df_train = pd.read_csv('../input/input/train_final.csv')
df_test = pd.read_csv('../input/input/test_final.csv')
combine = [df_train,df_test]
```

Data exploration/cleaning

In [3]:
```
# what does the data look like?
print(df_train.shape)
print(df_test.shape)
print('_'*40)
print(df_train.columns)
print(df_test.columns)
```

```
(16383, 29)
(16385, 25)
________________________________________
Index(['id', 'Y', 'F1', 'F2', 'F3', 'F4', 'F5', 'F6', 'F7', 'F8', 'F
9', 'F10',
       'F11', 'F12', 'F13', 'F14', 'F15', 'F16', 'F17', 'F18', 'F19'
, 'F20',
       'F21', 'F22', 'F23', 'F24', 'F25', 'F26', 'F27'],
      dtype='object')
Index(['id', 'F1', 'F2', 'F3', 'F4', 'F5', 'F6', 'F7', 'F8', 'F9', '
F10',
       'F11', 'F12', 'F13', 'F14', 'F15', 'F16', 'F17', 'F18', 'F19'
, 'F20',
       'F21', 'F22', 'F23', 'F24'],
      dtype='object')
```

Looks like the training set has 29 features and 16383 rows while the test set has only 25 features and 16385 rows.

Let's take a closer look at the training data:

In [4]:
```
df_train.head()
```

Out[4]:

	id	Y	F1	F2	F3	F4	F5	F6	F7	F8	...	F18	F19	F20	F21	I
0	1	1	38733	61385	0	38	118751	1000	32020	1	...	1	118830	1	1	126
1	2	1	34248	51329	0	41	120800	1000	130630	1	...	1	118832	1	1	130
2	3	1	15830	5522	0	50	118779	1000	303218	2	...	1	118832	1	2	127
3	4	1	19417	6754	0	45	123163	2000	19024	1	...	1	118832	1	1	152
4	5	1	42122	16991	0	41	119193	1000	303218	1	...	1	118832	1	1	133

5 rows × 29 columns

In [5]:
```
df_test.head()
```

Out[5]:

	id	F1	F2	F3	F4	F5	F6	F7	F8	F9	...	F15	F16	
0	16384	27991	135396	0	33	120578	17000	143022	1	124156	...	1	128168	12
1	16385	82444	54655	0	38	120064	18000	315517	1	123643	...	1	121648	12
2	16386	37950	23477	1	27	120267	1000	142929	1	123845	...	1	314350	12
3	16387	75000	92055	0	33	118844	2000	130186	1	183832	...	1	140144	18
4	16388	84243	8156	0	40	136613	2000	132071	1	139841	...	1	121642	13

5 rows × 25 columns

Interesting, looks like for the training set F25-F27 are filled with NaNs

In [6]:
```
df_train.info()
print('_'*40)
df_test.info()
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 16383 entries, 0 to 16382
Data columns (total 29 columns):
id     16383 non-null int64
Y      16383 non-null int64
F1     16383 non-null int64
F2     16383 non-null int64
F3     16383 non-null int64
F4     16383 non-null int64
F5     16383 non-null int64
F6     16383 non-null int64
F7     16383 non-null int64
F8     16383 non-null int64
F9     16383 non-null int64
F10    16383 non-null int64
F11    16383 non-null int64
F12    16383 non-null int64
F13    16383 non-null int64
F14    16383 non-null int64
F15    16383 non-null int64
F16    16383 non-null int64
F17    16383 non-null int64
F18    16383 non-null int64
F19    16383 non-null int64
F20    16383 non-null int64
F21    16383 non-null int64
F22    16383 non-null int64
F23    16383 non-null int64
F24    16383 non-null int64
```

```
F25    0 non-null float64
F26    0 non-null float64
F27    0 non-null float64
dtypes: float64(3), int64(26)
memory usage: 3.6 MB
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 16385 entries, 0 to 16384
Data columns (total 25 columns):
id     16385 non-null int64
F1     16385 non-null int64
F2     16385 non-null int64
F3     16385 non-null int64
F4     16385 non-null int64
F5     16385 non-null int64
F6     16385 non-null int64
F7     16385 non-null int64
F8     16385 non-null int64
F9     16385 non-null int64
F10    16385 non-null int64
F11    16385 non-null int64
F12    16385 non-null int64
F13    16385 non-null int64
F14    16385 non-null int64
F15    16385 non-null int64
F16    16385 non-null int64
F17    16385 non-null int64
F18    16385 non-null int64
F19    16385 non-null int64
F20    16385 non-null int64
F21    16385 non-null int64
F22    16385 non-null int64
F23    16385 non-null int64
F24    16385 non-null int64
dtypes: int64(25)
memory usage: 3.1 MB
```

Normally, we could replace any NaNs with the average of the column. However, it looks like F25-F27 are just empty. Should we just delete them? Let's see if they are really empty:

In [7]:
```
df_train.describe()
```

Out[7]:

	id	Y	F1	F2	F3	
count	16383.000000	16383.000000	16383.000000	16383.000000	16383.000000	163
mean	8192.000000	0.941464	44312.117256	26032.070927	0.048953	40.0
std	4729.509065	0.234762	34815.325971	35742.773305	0.281347	4.99
min	1.000000	0.000000	999.000000	43.000000	0.000000	21.0
25%	4096.500000	1.000000	21896.000000	4603.000000	0.000000	37.0
50%	8192.000000	1.000000	36806.000000	13819.000000	0.000000	40.0
75%	12287.500000	1.000000	75414.000000	41799.500000	0.000000	43.0
max	16383.000000	1.000000	314150.000000	311733.000000	7.000000	59.0

8 rows × 29 columns

Since like F25, F26, F27 don't have any useful data, they are just NaNs, and because the test set doesnt have them, let's remove those:

In [8]:
```
# remove null columns
df_train = df_train.dropna(axis=1,how='all')
df_train.describe()
```

Out[8]:

	id	Y	F1	F2	F3	
count	16383.000000	16383.000000	16383.000000	16383.000000	16383.000000	163
mean	8192.000000	0.941464	44312.117256	26032.070927	0.048953	40.0
std	4729.509065	0.234762	34815.325971	35742.773305	0.281347	4.99
min	1.000000	0.000000	999.000000	43.000000	0.000000	21.0
25%	4096.500000	1.000000	21896.000000	4603.000000	0.000000	37.0
50%	8192.000000	1.000000	36806.000000	13819.000000	0.000000	40.0
75%	12287.500000	1.000000	75414.000000	41799.500000	0.000000	43.0
max	16383.000000	1.000000	314150.000000	311733.000000	7.000000	59.0

8 rows × 26 columns

If we knew more about the features and how they are related (i.e.: if they had more useful feature names), perhaps we could do some more feature engineering now, that is consolidating and adding features.

We could also convert categorical data into numerical format to make it easier to analyze mathematically using techniques like one hot encoding or dummy variables, but since the data is already numerical we don't have to do that.

In [9]:
```
# #pair plots of entire dataset (takes too long)
# pp = sns.pairplot(data1, hue = 'Y', palette = 'deep', size=1.2, diag
_kind = 'kde', diag_kws=dict(shade=True), plot_kws=dict(s=10) )
# pp.set(xticklabels=[])
```

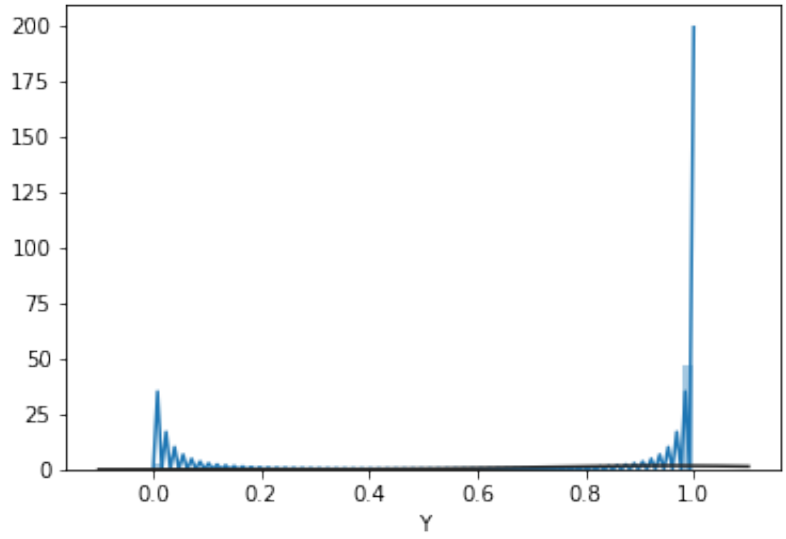
What is the distribution of feature values across the samples?

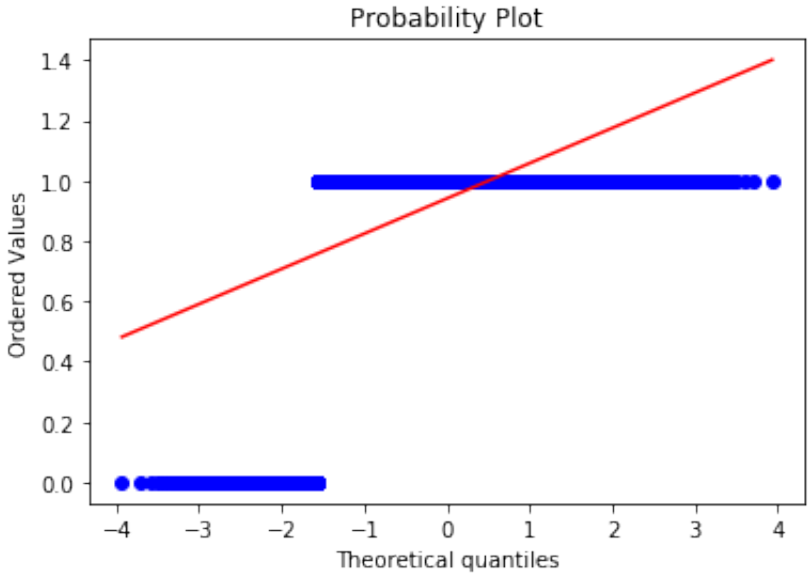
Overall we can see that:

- Y is a categorical feature with 0-1 values.
- Y is mostly 1's for this set

Let's take a closer look at Y:

In [10]:
```
#histogram and normal probability plot
sns.distplot(df_train['Y'], fit=norm);
fig = plt.figure()
res = stats.probplot(df_train['Y'], plot=plt)
```

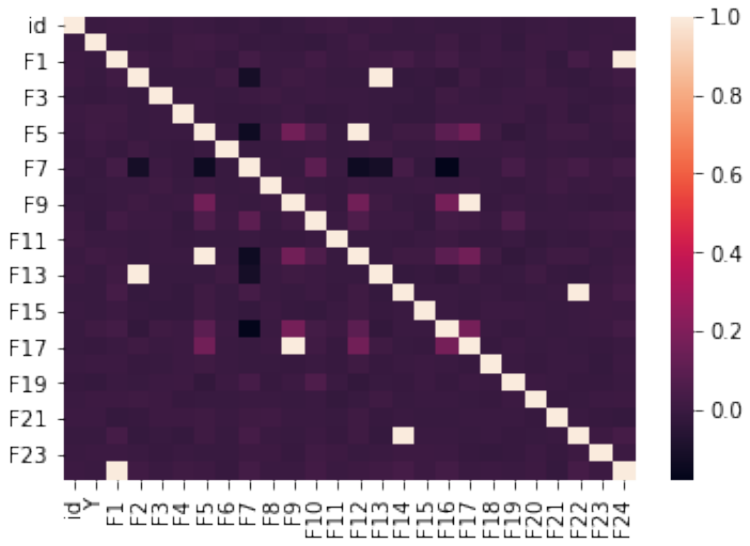




It looks like in this data set, the values tend more towards 1 than 0. Let's explore some of the relationships between the features:

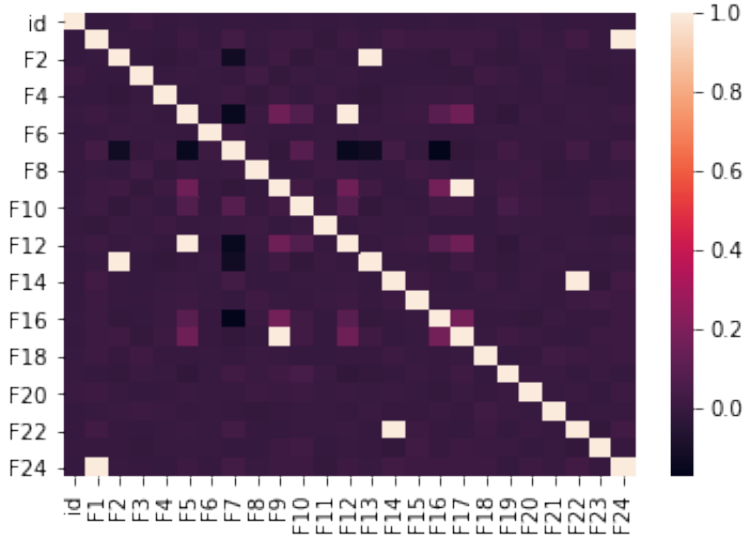
In [11]:
```
sns.heatmap(df_train.corr())
```

Out[11]: <matplotlib.axes._subplots.AxesSubplot at 0x7f41f19a8630>



In [12]:
```
sns.heatmap(df_test.corr())
```

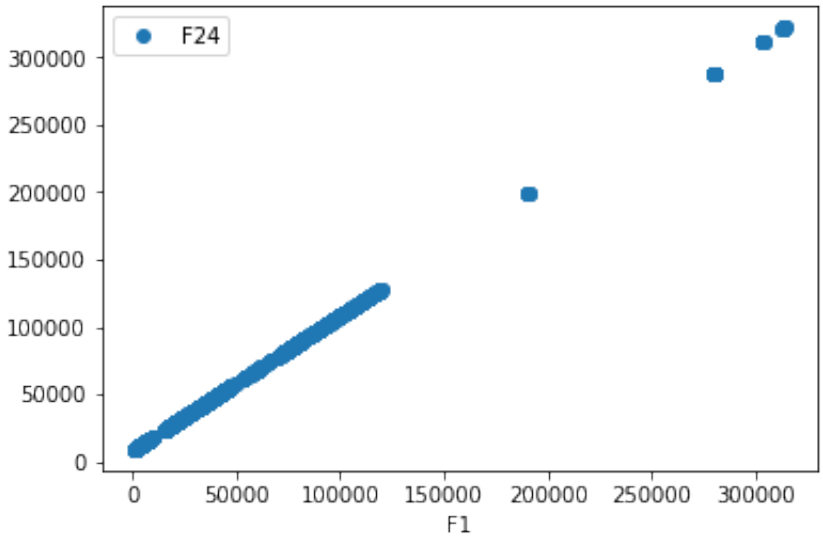
Out[12]: <matplotlib.axes._subplots.AxesSubplot at 0x7f41f06928d0>

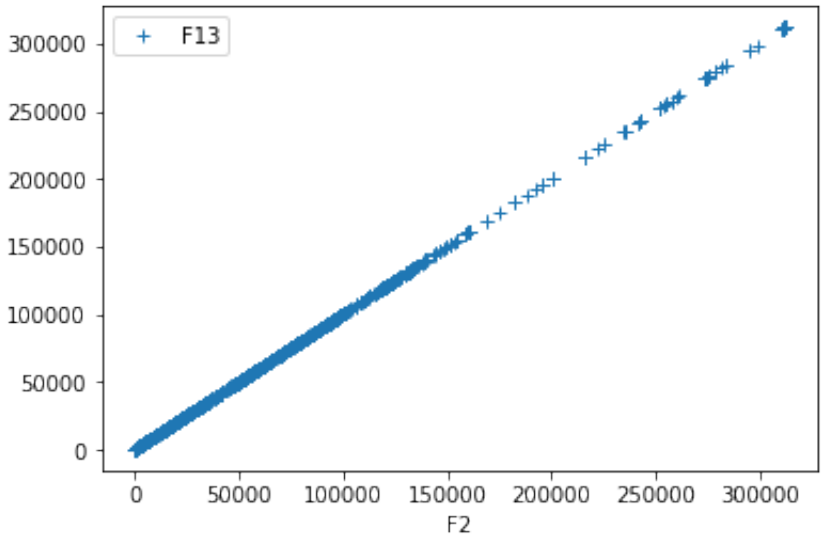


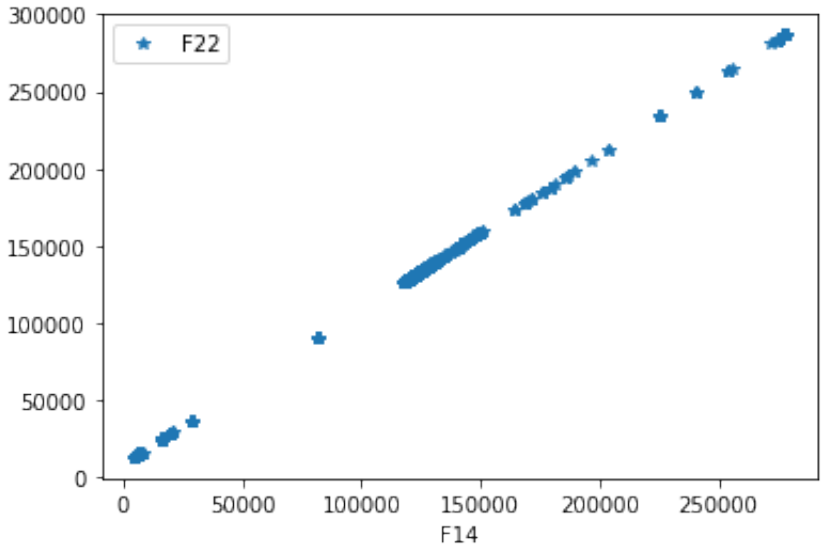
Looks like the pairs (F1, F24), (F2, F13), (F9, F17), and (F14, F22) are closely correlated.

In [13]:
```
df_test.plot(x='F1', y='F24', style='o')
df_test.plot(x='F2', y='F13', style='+')
df_test.plot(x='F14', y='F22', style='*')
```

Out[13]: <matplotlib.axes._subplots.AxesSubplot at 0x7f41f05a4550>







Those are indeed very linear relationships. But what does that mean? Perhaps we can consolidate the data somehow?

I tried to do log transforms on the data for some extra feature engineering/data preprocessing, but it didn't work with the random forest model I chose. Random forest models are invariant to monotone tranformations such as multiplications or other increasing functions, however using dummy variables or modular arithmetic would be valid transformations. However, since I don't know enough about the features to perform any sensible operations I decided not to...

In [14]:
```
# #log transform the target:
# df_train["Y"] = np.log1p(df_train["Y"])

# #log transform skewed numeric features:
# numeric_feats = df_train.dtypes[df_train.dtypes != "object"].index
# # numeric_feats2 = df_test.dtypes[df_test.dtypes != "object"].index

# skewed_feats = df_train[numeric_feats].apply(lambda x: skew(x.dropna
())) #compute skewness
# skewed_feats = skewed_feats[skewed_feats > 0.75]
# skewed_feats = skewed_feats.index

# # skewed_feats2 = df_test[numeric_feats2].apply(lambda x: skew(x.dro
pna())) #compute skewness
# # skewed_feats2 = skewed_feats2[skewed_feats2 > 0.75]
# # skewed_feats2 = skewed_feats2.index

# df_train[skewed_feats] = np.log1p(df_train[skewed_feats])
# # df_test[skewed_feats2] = np.log1p(df_train[skewed_feats2])
```

There are many different algorithms/models we learned about, so I wanted to try using some code I found to see which would perform the best, but it didn't work.

After some research, I saw that these learning algorithms are best suited for classification problems:

- logistic regression
- naive bayes
- neural networks
- decision trees
- random forest
- gradient boosting

I'll explain more why I chose random forest and gradient boosting below.

In [15]:
```
# Tried doing algorithm selection but it didn't work:

# from sklearn import svm, tree, linear_model, neighbors, naive_bayes,
```

```
ensemble, discriminant_analysis, gaussian_process
# from xgboost import XGBClassifier

# #Common Model Helpers
# from sklearn.preprocessing import OneHotEncoder, LabelEncoder
# from sklearn import feature_selection
# from sklearn import model_selection
# from sklearn import metrics

# #Machine Learning Algorithm (MLA) Selection and Initialization
# MLA = [
#     #Ensemble Methods
#     ensemble.AdaBoostClassifier(),
#     ensemble.BaggingClassifier(),
#     ensemble.ExtraTreesClassifier(),
#     ensemble.GradientBoostingClassifier(),
#     ensemble.RandomForestClassifier(),

#     #Gaussian Processes
#     gaussian_process.GaussianProcessClassifier(),

#     #GLM
#     linear_model.LogisticRegressionCV(),
#     linear_model.PassiveAggressiveClassifier(),
#     linear_model.RidgeClassifierCV(),
#     linear_model.SGDClassifier(),
#     linear_model.Perceptron(),

#     #Navies Bayes
#     naive_bayes.BernoulliNB(),
#     naive_bayes.GaussianNB(),

#     #Nearest Neighbor
#     neighbors.KNeighborsClassifier(),

#     #SVM
#     svm.SVC(probability=True),
#     svm.NuSVC(probability=True),
#     svm.LinearSVC(),

#     #Trees
#     tree.DecisionTreeClassifier(),
#     tree.ExtraTreeClassifier(),

#     #Discriminant Analysis
#     discriminant_analysis.LinearDiscriminantAnalysis(),
#     discriminant_analysis.QuadraticDiscriminantAnalysis(),


#     #xgboost: http://xgboost.readthedocs.io/en/latest/model.html
#     XGBClassifier()
#     ]
```

```
# #split dataset in cross-validation with this splitter class: http://scikit-learn.org/stable/modules/generated/sklearn.model_selection.ShuffleSplit.html#sklearn.model_selection.ShuffleSplit
# #note: this is an alternative to train_test_split
# cv_split = model_selection.ShuffleSplit(n_splits = 10, test_size = .3, train_size = .6, random_state = 0 ) # run model 10x with 60/30 split intentionally leaving out 10%

# #create table to compare MLA metrics
# MLA_columns = ['MLA Name', 'MLA Parameters','MLA Train Accuracy Mean', 'MLA Test Accuracy Mean', 'MLA Test Accuracy 3*STD' ,'MLA Time']
# MLA_compare = pd.DataFrame(columns = MLA_columns)

# #create table to compare MLA predictions
# MLA_predict = df_train['Y']
# data1 = df_train.copy(deep = True)
# data1_x_bin = df_train.columns

# #index through MLA and save performance to table
# row_index = 0
# for alg in MLA:

#     #set name and parameters
#     MLA_name = alg.__class__.__name__
#     MLA_compare.loc[row_index, 'MLA Name'] = MLA_name
#     MLA_compare.loc[row_index, 'MLA Parameters'] = str(alg.get_params())

#     #score model with cross validation: http://scikit-learn.org/stable/modules/generated/sklearn.model_selection.cross_validate.html#sklearn.model_selection.cross_validate
#     cv_results = model_selection.cross_validate(alg, data1[data1_x_bin], data1['Y'], cv  = cv_split)

#     MLA_compare.loc[row_index, 'MLA Time'] = cv_results['fit_time'].mean()
#     MLA_compare.loc[row_index, 'MLA Train Accuracy Mean'] = cv_results['train_score'].mean()
#     MLA_compare.loc[row_index, 'MLA Test Accuracy Mean'] = cv_results['test_score'].mean()
#     #if this is a non-bias random sample, then +/-3 standard deviations (std) from the mean, should statistically capture 99.7% of the subsets
#     MLA_compare.loc[row_index, 'MLA Test Accuracy 3*STD'] = cv_results['test_score'].std()*3   #let's know the worst that can happen!


#     #save MLA predictions - see section 6 for usage

#     alg.fit(data1[data1_x_bin], data1['Y'])
#     MLA_predict[MLA_name] = alg.predict(data1[data1_x_bin])
```

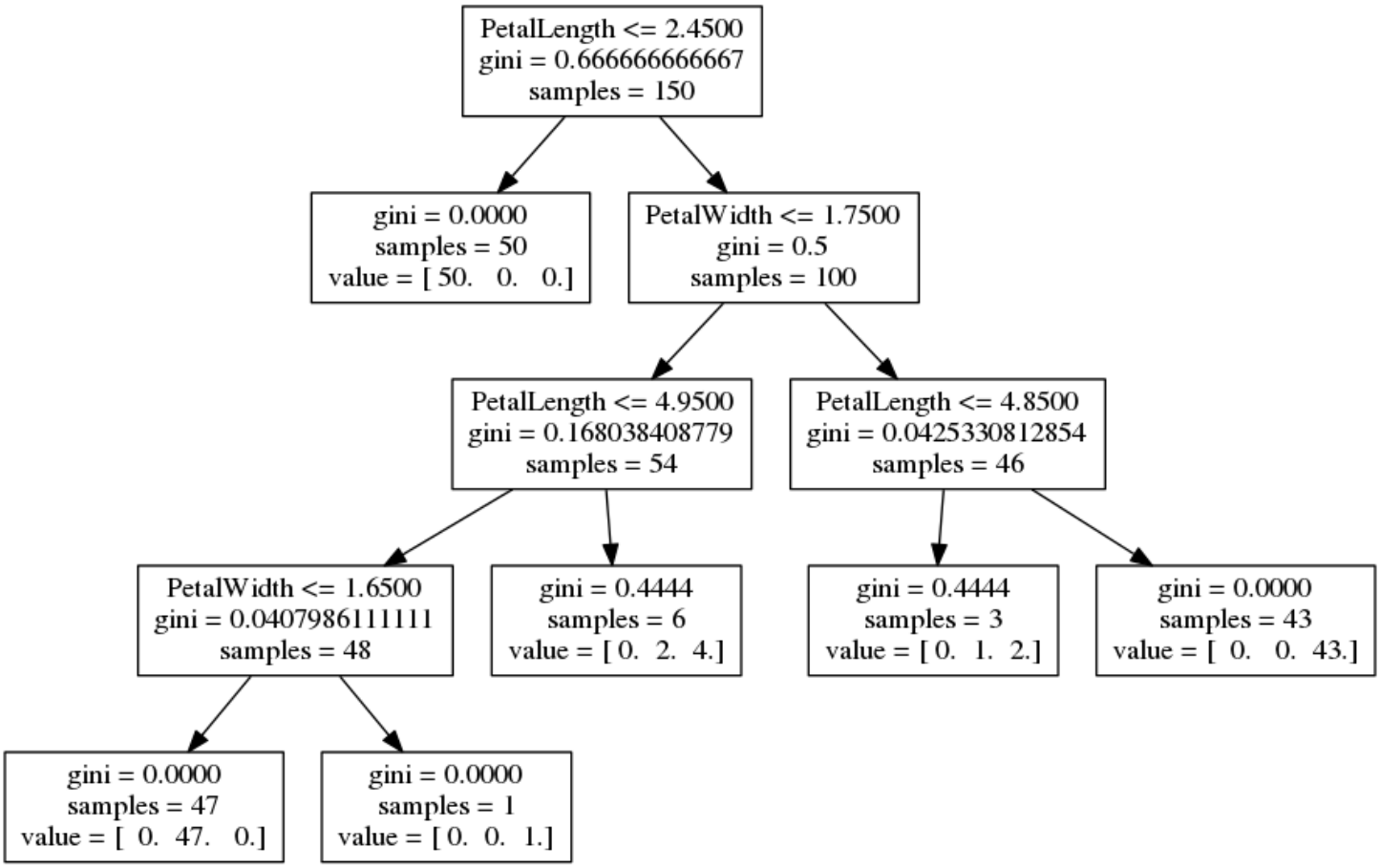
```
#     row_index+=1


# #print and sort table: https://pandas.pydata.org/pandas-docs/stable/generated/pandas.DataFrame.sort_values.html
# MLA_compare.sort_values(by = ['MLA Test Accuracy Mean'], ascending = False, inplace = True)
# MLA_compare
# #MLA_predict
```

Let's try making a model

Before researching, I tried using models we used in the labs like linear regrssion, ridge regression, lasso regression, and principal component regression (PCR), and k-nearest neigbors (KNN). However, after realizing that this is a classification not a regression problem I decided to scrap those methods and do some more reserach into models more suited for classification.

After researching different learning algorithms I decided to use random forests + gradient boosting (xgboost) since random forests are pretty standard for classification problems. I used this tutorial (https://towardsdatascience.com/random-forest-in-python-24d0893d51c0) for my model.

Random forests are built on decision trees, which can be visualized like so:



Ying Dong (http://www.ke.tu-darmstadt.de/lehre/arbeiten/studien/2015/Dong_Ying.pdf) describes random forests in terms of decision trees well, saying "Random forest uses an ensemble method by combining a multitude of decision trees. The main idea behind ensemble methods is to construct a single model by combining a set of base models[14]. It has been proven that using ensemble methods can give better results than using a single model when measured on predictive accuracy. Random forest uses a bagging method, which averages the predictions of multiple models trained on different samples to reduce the variance and achieve higher accuracy."

In other words, random forests uses ensembles + bagging + decision trees to achieve higher accuracy than a single decision tree.

I also considered neural nets but didn't have enough time to experiment. But since neural nets seem more suited to "natural" data like images I decided not to use it for this data set. I also considered using logistic regression, but since I couldn't glean much from the data I thought the random forest would be better.

As for xgboost, every successful kaggler seems to use it in their solutions, so I also decided to use it. It's fast, effective, and good for many different data sets.

In [16]:
```
from sklearn import cross_validation, grid_search, metrics, ensemble
```

In [17]:
```
df_train.head()
```

Out[17]:

	id	Y	F1	F2	F3	F4	F5	F6	F7	F8	...	F15	F16	F17	F18
0	1	1	38733	61385	0	38	118751	1000	32020	1	...	1	119757	119100	1
1	2	1	34248	51329	0	41	120800	1000	130630	1	...	1	138110	121149	1
2	3	1	15830	5522	0	50	118779	1000	303218	2	...	1	119777	119126	1
3	4	1	19417	6754	0	45	123163	2000	19024	1	...	2	270637	123511	1
4	5	1	42122	16991	0	41	119193	1000	303218	1	...	1	119777	119542	1

5 rows × 26 columns

After experimenting with different numbers of estimators and max_depths, I saw that my score would go up if I used more estimators/had deeper trees, so I decided to use 1200 and 25 for my estimators and max_depth

In [18]:
```
# Model with the best estimator
model = ensemble.RandomForestClassifier(n_estimators=1200, max_depth=2
5)
model.fit(df_train[df_train.columns[df_train.columns != 'Y']], df_trai
n['Y'])
```

Out[18]:
```
RandomForestClassifier(bootstrap=True, class_weight=None, criterion=
'gini',
            max_depth=25, max_features='auto', 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, n_estimators=1200, n_jobs=
1,
            oob_score=False, random_state=None, verbose=0,
            warm_start=False)
```

In [19]:
```
# Use numpy to convert to arrays
import numpy as np
# Labels are the values we want to predict
labels = np.array(df_train['Y'])

# Remove the labels from the features
# axis 1 refers to the columns
df = df_train.drop('Y', axis = 1)

# Saving feature names for later use
feature_list = list(df_train.columns)

# Convert to numpy array
features = np.array(df_train)
```

I tried splitting the training data using scikit-learn to validate my model, but I found that training the model with this smaller data set decreased performance. However, when I did validate with the trainig and testing sets from scikit-learn, the results showed that the random forest model's Mean Absolute Error was 0.04 degrees.

In [20]:
```
# # Using Skicit-learn to split data into training and testing sets
# from sklearn.model_selection import train_test_split

# # Split the data into training and testing sets
# train_features, test_features, train_labels, test_labels = train_tes
t_split(df, labels, test_size = 0.25, random_state = 42)
```

In [ ]:
```
# print('Training Features Shape:', train_features.shape)
# print('Training Labels Shape:', train_labels.shape)
# print('Testing Features Shape:', test_features.shape)
# print('Testing Labels Shape:', test_labels.shape)
# print('-'*40)
df_test['Y'] =  0
print('Training Features Shape:', df_train.drop(['Y'],axis = 1).shape)
print('Training Labels Shape:', df_train['Y'].shape)
print('Testing Features Shape:', df_test.drop(['Y'], axis=1).shape)
print('Testing Labels Shape:', df_test['Y'].shape)
```

```
Training Features Shape: (16383, 25)
Training Labels Shape: (16383,)
Testing Features Shape: (16385, 25)
Testing Labels Shape: (16385,)
```

Train model:

In [ ]:
```
# Import the model we are using
from sklearn.ensemble import RandomForestRegressor

# Instantiate model with 1200 decision trees
rf = RandomForestRegressor(n_estimators = 1200, random_state = 42)

# Train the model on training data
#rf.fit(train_features, train_labels);
rf.fit(df_train.drop(['Y'], axis = 1), df_train['Y']);
```

In [ ]:
```
# Use the forest's predict method on the test data
#predictions = rf.predict(test_features)
predictions = rf.predict(df_test.drop(['Y'], axis=1))

# Calculate the absolute errors
# errors = abs(predictions - df_test['Y'])

# # Print out the mean absolute error (mae)
# print('Mean Absolute Error:', round(np.mean(errors), 2), 'degrees.')
```

In [ ]:
```
predictions
```

In [ ]:
```
# Use the forest's predict method on the test data
#predictions = rf.predict(test_features)
# predictions = rf.predict(df_train.drop(['Y'], axis=1))
# predictions
```

I wanted to see which features were most important in an attempt to build a random forest model with only the top important features:

In [ ]:
```
# Get numerical feature importances
importances = list(rf.feature_importances_)

# List of tuples with variable and importance
feature_importances = [(feature, round(importance, 2)) for feature, im
portance in zip(feature_list, importances)]

# Sort the feature importances by most important first
feature_importances = sorted(feature_importances, key = lambda x: x[1]
, reverse = True)

# Print out the feature and importances
[print('Variable: {:20} Importance: {}'.format(*pair)) for pair in fea
ture_importances];
```

In [ ]:
```
# Import matplotlib for plotting and use magic command for Jupyter Not
ebooks
import matplotlib.pyplot as plt
%matplotlib inline
# Set the style
plt.style.use('fivethirtyeight')
# list of x locations for plotting
x_values = list(range(len(importances)))
# Make a bar chart
plt.bar(x_values, importances, orientation = 'vertical')
# Tick labels for x axis
plt.xticks(x_values, feature_list, rotation='vertical')
# Axis labels and title
plt.ylabel('Importance'); plt.xlabel('Variable'); plt.title('Variable
Importances');
```

I tried building a random forest with only the top 7 features, (to make training faster) but this model didn't do as well as the one with all the features so I decided not to use it.

In [ ]:
```
# # New random forest with only the two most important variables
# rf_most_important = RandomForestRegressor(n_estimators= 1000, random
_state=42)

# # Extract the two most important features
# important_indices = [feature_list.index('id'), feature_list.index('Y
'), feature_list.index('F3'), feature_list.index('F15'),
#                       feature_list.index('F23'), feature_list.index('
F1'), feature_list.index('F12')]
# # train_important = train_features[:, important_indices]
# train_important = df_train.iloc[:, important_indices]
# test_important = df_test.iloc[:, important_indices]

# # Train the random forest
# rf_most_important.fit(train_important, df_train['Y'])

# # Make predictions and determine the error
# predictions = rf_most_important.predict(test_important)
# # predictions = rf_most_important.predict(train_important)

# # errors = abs(predictions - df_train['Y'])
# # # Display the performance metrics
# # print('Mean Absolute Error:',round(np.mean(errors), 2), 'degrees.'
)
```

Xgboost:

Now to add xgboost. I tried training the xgb model on the sci-kit split data to validate, but again, training on the smaller dataset produced poorer results. Though in retrospect, I probably could have trained and validated on the smaller set then trained on the whole set again...but then maybe this would result in overfitting? Nevertheless, when I validated the xgb model on the split data set this is what I got:

- Average precision: 0.92
- Average recall: 0.94
- Average f1-score: 0.92

In [ ]:
```
import xgboost as xgb
from xgboost import XGBClassifier, plot_importance
from sklearn.grid_search import GridSearchCV
from matplotlib import pyplot
from numpy import nan
from sklearn.model_selection import StratifiedKFold
%matplotlib inline
```

In [ ]:
```
dtrain = xgb.DMatrix(df_train.drop(['Y'],axis = 1), label = df_train['
Y'])
dtest = xgb.DMatrix(df_test)
```

In [ ]:
```
params = {"max_depth":5, "eta":0.1}
model = xgb.cv(params, dtrain,  num_boost_round=500, early_stopping_ro
unds=100)
```

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

In [ ]:
```
model.loc[30:,["test-rmse-mean", "train-rmse-mean"]].plot()
```

I know that this kaggle competition uses area under the curve to validate, but I wanted to see how this model did with respect to RMSE since that is what I'm familiar with. This plot was pretty concerning...since it indicates that the test rmse doesn't decrease with the training rmse...but I went ahead and tried submitting anyways.

In [ ]:
```
model_xgb = xgb.XGBRegressor(n_estimators=360, max_depth=5, learning_r
ate=0.1) #the params were tuned using xgb.cv
# model_xgb.fit(X_train, y)
model_xgb.fit(df_train.drop(['Y'], axis=1), df_train['Y'])
```

I also tried using an XGBClassifier instead of an XGBRegressor, since this is a classification problem, but that gave me worse results.

In [ ]:
```
# xgb_preds = np.expm1(model_xgb.predict(X_test))
# lasso_preds = np.expm1(model_lasso.predict(X_test))
xgb_preds = np.expm1(model_xgb.predict(df_test.drop(['Y'], axis=1)))
rf_preds = np.expm1(rf.predict(df_test.drop(['Y'],axis = 1)))
```

In [ ]:
```
preds = pd.DataFrame({"xgb":xgb_preds, "rf":rf_preds})
preds.plot(x = "xgb", y = "rf", kind = "scatter")
```

In [ ]:
```
# TO DO: tune xgb ratios
preds = 0.70*rf_preds + 0.30*xgb_preds
```

In [ ]:
```
# solution = pd.DataFrame({"Y":df_train.F8, "id":df_train.id  })
# solution.to_csv("sol.csv", index = False)

# no xgboost
# solution = pd.DataFrame({"Y":predictions, "id":df_train.id  })
# solution.to_csv("sol.csv", index = False)
```

In [ ]:
```
# with xgboost
solution = pd.DataFrame({"id":df_test.id, "Y":preds})
solution.to_csv("rf_sol.csv", index = False)
```