Project 7 Report

Team members:

Supratik Chanda and Chris Higgs

Project 7 Introduction:

In this project, we will use Logistic Regression and SVM amonst other techniques to predict two disjoint properties in two disjoint fields: Speaker Gender given voice recording measurements, and Pokémon™ Primary Elemental Type given various statistics.

The rationale for combining these two into a single report has been moderately debated where the only commonalities in these two projects are some of the techniques used and the mere fact that the authors form a group for this assignment. Any complaints or suggestions relating to the combining of these reports should be (kindly) directed to Dr. John Edwards (john.edwards@usu.edu).

In [ ]:

Voice Recognition Introduction:

Gender Recognition by Voice and Speech Analysis:

This database was created to identify a voice as male or female, based upon acoustic properties of the voice and speech. The dataset consists of 3,168 recorded voice samples, collected from male and female speakers. The voice samples are pre-processed by acoustic analysis in R using the seewave and tuneR packages, with an analyzed frequency range of 0hz-280hz (human vocal range).

Dataset:

The following acoustic properties of each voice are measured and included within the CSV:

meanfreq: mean frequency (in kHz)

sd: standard deviation of frequency

median: median frequency (in kHz)

Q25: first quantile (in kHz)

Q75: third quantile (in kHz)

IQR: interquantile range (in kHz)

skew: skewness (see note in specprop description)

kurt: kurtosis (see note in specprop description)

sp.ent: spectral entropy

sfm: spectral flatness

mode: mode frequency

centroid: frequency centroid (see specprop)

peakf: peak frequency (frequency with highest energy)

meanfun: average of fundamental frequency measured across acoustic signal

minfun: minimum fundamental frequency measured across acoustic signal

maxfun: maximum fundamental frequency measured across acoustic signal

meandom: average of dominant frequency measured across acoustic signal

mindom: minimum of dominant frequency measured across acoustic signal

maxdom: maximum of dominant frequency measured across acoustic signal

dfrange: range of dominant frequency measured across acoustic signal

modindx: modulation index. Calculated as the accumulated absolute difference between adjacent measurements of fundamental frequencies divided by the frequency range

label: male or female

Analysis Technique

Steps: ¶

1)At first , each X features column had dollar sign which was cleansed and the data type was changed from object to float64.

2) We then checked for any duplicate values in the dataset and removed any duplicate values

2)Then, we used SelectKBest features from feature_selection package of scikit-learn to select which features should be taken. The SelectKBest features gives an array of score_values in precentage and another array of p_values . Whosoevers' score or p_value is greater , it has a deeper effect on the Y_label. So, we took column name "kurt"(p-0.00 and score-119.87) to "mindom"(p-0.00 and score-9.86). The score_function are chi2 and f_classif. We took chi2 and went ahead with the score func for finding the accuracy,precision and f1_score and confusion_matrix.

Logistic Regression:

We divided the dataset into train_set and test_set and then performed training and testing of the data. At first, we used LabelEncoder for the Y_Label but then we found out that without encoding also, the performance is not decreasing. Hence, we went without encoding for determining the Label.

After that, We used GridSearchCV and RandomizedSearchCV to find out which produces the most accuracy and has the least execution time. We found out that RandomizedSearchCV is the most efficient model_selection technique and the execution time is the quickest. The following analysis is done using RandomizedSearchCV.

We used Multi-Class "OVR" and multi-class "multinomial" to find out which class is the best for generalization . Realized that One over Rest is the best approach.

OVR has the following results:

```
Precision_score:  [0.976109 0.909091]
Recall Score: [0.902208 0.977918]
f_score [0.937705 0.942249]
 The overall f1_score: 0.9399770790771836
 Test_accuracy:  0.9810725552050473
```

Multinomial has the following results:

```
Test_accuracy:  0.5299684542586751
```

We also found out the precision, f_score and confusion_matrix (shown below) to find out the total number of incorrect predictions for OVR approach

```
The test dataset confusion_matrix is :
 [[286  31]
 [  7 310]]
```

7) The next model was SVM Linear:

Taking all the values of C and checking out the accuracy score with kernel as linear.

The C parameter tells the SVM optimization how much you want to avoid misclassifying each training example. For large values of C, the optimization will choose a smaller-margin hyperplane if that hyperplane does a better job of getting all the training points classified correctly. Conversely, a very small value of C will cause the optimizer to look for a larger-margin separating hyperplane, even if that hyperplane misclassifies more points.

Thus for a very large values we can cause overfitting of the model and for a very small value of C we can cause underfitting.Thus the value of C must be chosen in such a manner that it generalised the unseen data well.From the above plot we can see that accuracy has been close to 97% for C=1, C=5 and C=6 and then it drops around 96.8% and remains constant.

```
C_values    Accuracy_values
0   1   0.9694
0   2   0.9688
0   3   0.9688
0   4   0.9688
0   5   0.9691
0   6   0.9694
0   7   0.9691
0   8   0.9688
0   9   0.9685
```

Once done,we used the C values with GridSearchCV and RandomizedSearchCV to find out the accuracy and f1_score.

Surprisingly, we found out that RandomizedCV took less time than GridsearchCV. The time for GridSearchCV was 117ms and that of RandomizedSearchCV was 11ms.

We went ahead with our analysis with RandomizedSearchCV

```
grid best score:  0.968034727703236
grid best parameters:  {'random_state': 45, 'kernel': 'linear', 'class_weight': {0: 1}, 'C': 6}
Execution time: 11.855309963226318 ms
```

We used predict_proba function to find out what are the probabilities of the predicted values. We found out that there are four incorrect predictions. The figure and analysis is given in the results

Next: SVM RBF

Taking kernel as rbf and taking different values gamma

Technically, the gamma parameter is the inverse of the standard deviation of the RBF kernel (Gaussian function), which is used as similarity measure between two points. Intuitively, a small gamma value define a Gaussian function with a large variance. In this case, two points can be considered similar even if are far from each other. In the other hand, a large gamma value means define a Gaussian function with a small variance and in this case, two points are considered similar just if they are close to each other

We saw that for gamma=10 and 100 the kernel is performing poorly.We also saw a slight dip in accuracy score when gamma is 1.

Once done, we used the RandomizedSearchCV as the grid_search model_selectionand passed gamma value=0.01 and a list of class weights inside the param_grids as paramaters. We found out that changing the class weight from {0:1} to {0:6} decreases the accuracy.Hence, stuck with class_weight {0:1}

```
grid best parameters:  {'random_state': 10, 'kernel': 'rbf', 'gamma': 0.01, 'class_weight': {0: 3}, 'C': 1.0}
grid best score:  for class weight 4 0.5189423835832676
grid best parameters:  {'random_state': 5, 'kernel': 'rbf', 'gamma': 0.01, 'class_weight': {0: 4}, 'C': 1.0}
grid best score:  for class weight 5 0.5189423835832676
grid best parameters:  {'random_state': 5, 'kernel': 'rbf', 'gamma': 0.01, 'class_weight': {0: 5}, 'C': 1.0}
grid best score:  for class weight 6 0.5189423835832676
grid best parameters:  {'random_state': 5, 'kernel': 'rbf', 'gamma': 0.01, 'class_weight': {0: 6}, 'C': 1.0}
grid best score:  for class weight 7 0.5189423835832676
```

The execution time was also very less and went on to find the precision,f_score and accuracy for the model.

Also used the predict_proba method to find out the probabilities of all predictions.

Used Confusion_mtarix to find the total number of incorrect predictions and also used boxplots to visually figure out the range of the probabilities for both the correct and the incorrect predictions

```
The confusion matrix is:
```

```
The confusion matrix is:
[[219  98]
 [ 70 247]]
```

NEXT SVM POLY

Taking kernel as poly and different values of polynomial degree

We first used cross-validation to find out which polynomial degree is the best for attaining the best accuracy_score

```
poly_values Accuracy_values
    2   0.8507
    3   0.9458
    4   0.8312
    5   0.866
    6   0.7748
```

Then, we used that degree, gamma and class weights as parameters in RandomizedSearchCV to find out the class weight for the best accuracy score.

```
grid best parameters:  {'random_state': 10, 'kernel': 'rbf', 'gamma': 0.01, 'class_weight': {0: 3}, 'C': 1.0}
grid best score:  for class weight 4 0.5189423835832676
grid best parameters:  {'random_state': 5, 'kernel': 'rbf', 'gamma': 0.01, 'class_weight': {0: 4}, 'C': 1.0}
grid best score:  for class weight 5 0.5189423835832676
grid best parameters:  {'random_state': 5, 'kernel': 'rbf', 'gamma': 0.01, 'class_weight': {0: 5}, 'C': 1.0}
grid best score:  for class weight 6 0.5189423835832676
grid best parameters:  {'random_state': 5, 'kernel': 'rbf', 'gamma': 0.01, 'class_weight': {0: 6}, 'C': 1.0}
grid best score:  for class weight 7 0.5189423835832676
```

We also found the execution time for RandomizedSearchCV.

```
Execution time: 28.46691608428955 ms
```

After that, we analysed the total number of incorrect predictions using confusion_matrix

```
The test dataset confusion_matrix is :
 [[314   3]
 [ 10 307]]
```

Also used the predict_proba method to find out the probabilities of all predictions and applied boxplots to visually figure out the range of the probabilities for both the correct and the incorrect predictions

Results:

The dataset Cleansing:

The Voice Recognition Dataset before cleansing:

Out[51]:

	meanfreq	sd	median	Q25	Q75	IQR	skew	kurt	sp.ent	sfm	...	centroid	meanfun	minfun	maxfun	meandom	mindom	maxdom	dfra
0	$0.06	$0.06	$0.03	$0.02	$0.09	$0.08	$12.86	$274.40	$0.89	$0.49	...	$0.06	$0.08	$0.02	$0.28	$0.01	$0.01	$0.01	$
1	$0.07	$0.07	$0.04	$0.02	$0.09	$0.07	$22.42	$634.61	$0.89	$0.51	...	$0.07	$0.11	$0.02	$0.25	$0.01	$0.01	$0.05	$
2	$0.08	$0.08	$0.04	$0.01	$0.13	$0.12	$30.76	$1,024.93	$0.85	$0.48	...	$0.08	$0.10	$0.02	$0.27	$0.01	$0.01	$0.02	$
3	$0.15	$0.07	$0.16	$0.10	$0.21	$0.11	$1.23	$4.18	$0.96	$0.73	...	$0.15	$0.09	$0.02	$0.25	$0.20	$0.01	$0.56	$
4	$0.14	$0.08	$0.12	$0.08	$0.21	$0.13	$1.10	$4.33	$0.97	$0.78	...	$0.14	$0.11	$0.02	$0.27	$0.71	$0.01	$5.48	$

5 rows × 21 columns

The dataset after cleansing 'S' symbol and converting the object datatype to float datatype

Out[53]:

	meanfreq	sd	median	Q25	Q75	IQR	skew
0	0.059780999999999994	0.0642413	0.032026900000000004	0.0150715	0.09019339999999999	0.07512200000000001	12.8634618
1	0.0660087	0.06731000000000001	0.0402287	0.0194139	0.09266619999999999	0.0732523	22.423285399999997
2	0.0773155	0.0838294	0.0367185	0.008701100000000002	0.131908	0.123207	30.757154600000003
3	0.1512281	0.07211060000000001	0.15801120000000002	0.0965817	0.20795529999999998	0.11137349999999999	1.2328313
4	0.1351204	0.0791461	0.1246562	0.0787202	0.2060449	0.1273247	1.1011737

5 rows × 21 columns

The feature_selection technique used is SelectKBest; chosen among SeleckBest and SelectPrecentile . The function_score tested are chi2 and f_classify and found out the chi2 gives more distinct p_values and score

```
SelectKBest feature_extraction_text using f_classif:Value

pvalues_:   [0.000000 0.000000 0.000000 0.000000 0.000164 0.000000 0.039263 0.000001
 0.000000 0.000000 0.000000 0.000000 0.000000 0.000000 0.000000 0.000000
 0.000000 0.000000 0.000000 0.083031]

scores:     [406.752821 945.461378 277.588158 1121.569223 14.236082 1965.749999
 4.252980 24.255365 1003.308717 463.923194 96.257909 406.752821
 7228.790362 60.282137 90.228036 119.959108 125.110999 126.024161
 121.457858 3.006445]
[ 0  1  2  3  4  5  6  7  8  9 10 11 12 13 14 15 16 17 18 19]

SelectKBest feature_extraction_text using chi2:Value

pvalues_:   [0.181666 0.060081 0.177517 0.000185 0.851080 0.000000 0.000001 0.000000
 0.189369 0.000000 0.066400 0.181666 0.000060 0.440885 0.579711 0.000000
 0.002470 0.000000 0.000000 0.619374]

scores:     [1.783946 3.535149 1.818288 13.977196 0.035247 26.337745 24.329583
 11987.597927 1.722517 31.246850 3.369843 1.783946 16.094352 0.593977
 0.306703 38.460229 9.162780 297.819019 290.272018 0.246748]

SelectPercentile feature_extraction_text:Value

pvalues_:   [0.181666 0.060081 0.177517 0.000185 0.851080 0.000000 0.000001 0.000000
 0.189369 0.000000 0.066400 0.181666 0.000060 0.440885 0.579711 0.000000
 0.002470 0.000000 0.000000 0.619374]

scores:     [1.783946 3.535149 1.818288 13.977196 0.035247 26.337745 24.329583
 11987.597927 1.722517 31.246850 3.369843 1.783946 16.094352 0.593977
 0.306703 38.460229 9.162780 297.819019 290.272018 0.246748]
```

Displayed as a dataframe

Out[56]:

	Columns	P_value	scores
7	kurt	$0.00	$11,987.60
17	maxdom	$0.00	$297.82
18	dfrange	$0.00	$290.27
15	meandom	$0.00	$38.46
9	sfm	$0.00	$31.25
5	IQR	$0.00	$26.34
6	skew	$0.00	$24.33
12	meanfun	$0.00	$16.09
3	Q25	$0.00	$13.98
16	mindom	$0.00	$9.16
1	sd	$0.06	$3.54
10	mode	$0.07	$3.37
2	median	$0.18	$1.82
0	meanfreq	$0.18	$1.78
11	centroid	$0.18	$1.78
8	sp.ent	$0.19	$1.72
13	minfun	$0.44	$0.59
14	maxfun	$0.58	$0.31
19	modindx	$0.62	$0.25
4	Q75	$0.85	$0.04

Took features from kurt to mindom whose p_value was significant

took features from kurt to minkom whose p_value was significant

Logistic Regression

Grid SearchCV results and execution time(multi-class=OVR):

```
For GridSearchCV:
grid best score for train_set:  0.968034727703236
grid best parameters for train_set:  {'C': 1.0, 'multi_class': 'ovr', 'penalty': 'l1', 'random_state': 5}
Execution time: 6.657193660736084 ms
```

Test accuracy:

```
Test_accuracy:  0.9810725552050473
```

Randomized SearchCV results and execution time(multi-class=OVR):

```
For RandomizedSearchCV:
grid best score for train_set:  0.968034727703236
grid best parameters for train_set:  {'random_state': 17, 'penalty': 'l1', 'multi_class': 'ovr', 'C': 1.0}
Execution time: 1.7632555961608887 ms
```

Test accuracy:

```
Test_accuracy:  0.9810725552050473
```

Both accuracy is exactly the same . The only difference is the execution time. Randomized SearchCV has much lesser execution time. So. we went with RandomizedCV and below are the results

The overall scores:

```
Precision_score:  [0.976109 0.909091]
Recall Score: [0.902208 0.977918]
f_score [0.937705 0.942249]
 The overall f1_score: 0.9399770790771836
```

The dataframe containing values with incorrect predictions and their probability.

0 = female,1=male

Out[223]:

	Probability	Actual	Predicted	SameOrNot
10	$0.55	0	1	1
17	$0.73	0	1	1
20	$0.81	0	1	1
35	$0.63	0	1	1
57	$0.74	0	1	1
90	$0.74	0	1	1
136	$0.63	0	1	1
162	$0.51	1	0	1
249	$0.71	0	1	1
261	$0.80	0	1	1
268	$0.59	0	1	1
269	$0.71	0	1	1

Now, Lets see the Score for multi-class=multinomial using RandomizedCV:

```
grid best score:  0.5240726124704025
grid best paramters:  {'solver': 'sag', 'random_state': 10, 'penalty': 'l2', 'multi_class': 'multinomial', 'C': 1.0}
```

```
Test_accuracy:  0.5299684542586751
```

Hence, we see that OVR is the best approach for Logistic Regression

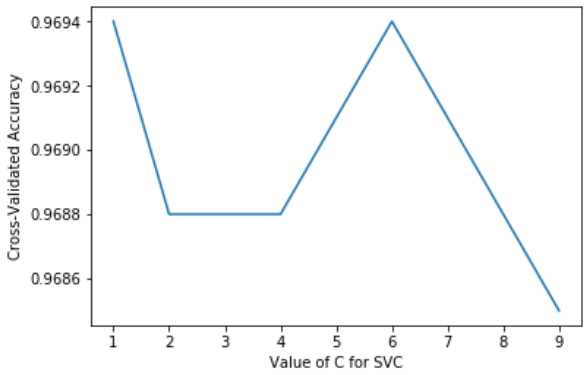
SVM KERNEL=LINEAR

Analysing the accuracy score with different values of C

Out[64]:

	C_values	Accuracy_values
0	1	0.9694
0	2	0.9688
0	3	0.9688
0	4	0.9688
0	5	0.9691
0	6	0.9694
0	7	0.9691
0	8	0.9688
0	9	0.9685

Out[73]: Text(0,0.5,'Cross-Validated Accuracy')



Going ahead with the C-Value which produces the most accuracy.Then feeding the C-value in GridSearchCV and RandomizedSearchCV:

Scores,execution time and test_accuracy for GridSearchCV:

```
grid best score:  0.9696132596685083
grid best paramters:  {'C': 6, 'class_weight': {0: 2}, 'kernel': 'linear', 'random_state': 5}
Execution time: 36.25111722946167 ms
```

```
Test_accuracy:  0.9810725552050473
```

Scores, execution time and test_accuracy for RandomizedSearchCV:

```
grid best score:  0.968034727703236
grid best parameters:  {'random_state': 45, 'kernel': 'linear', 'class_weight': {0: 1}, 'C': 6}
Execution time: 11.855309963226318 ms
```

```
Test_accuracy:  0.9779179810725552
```

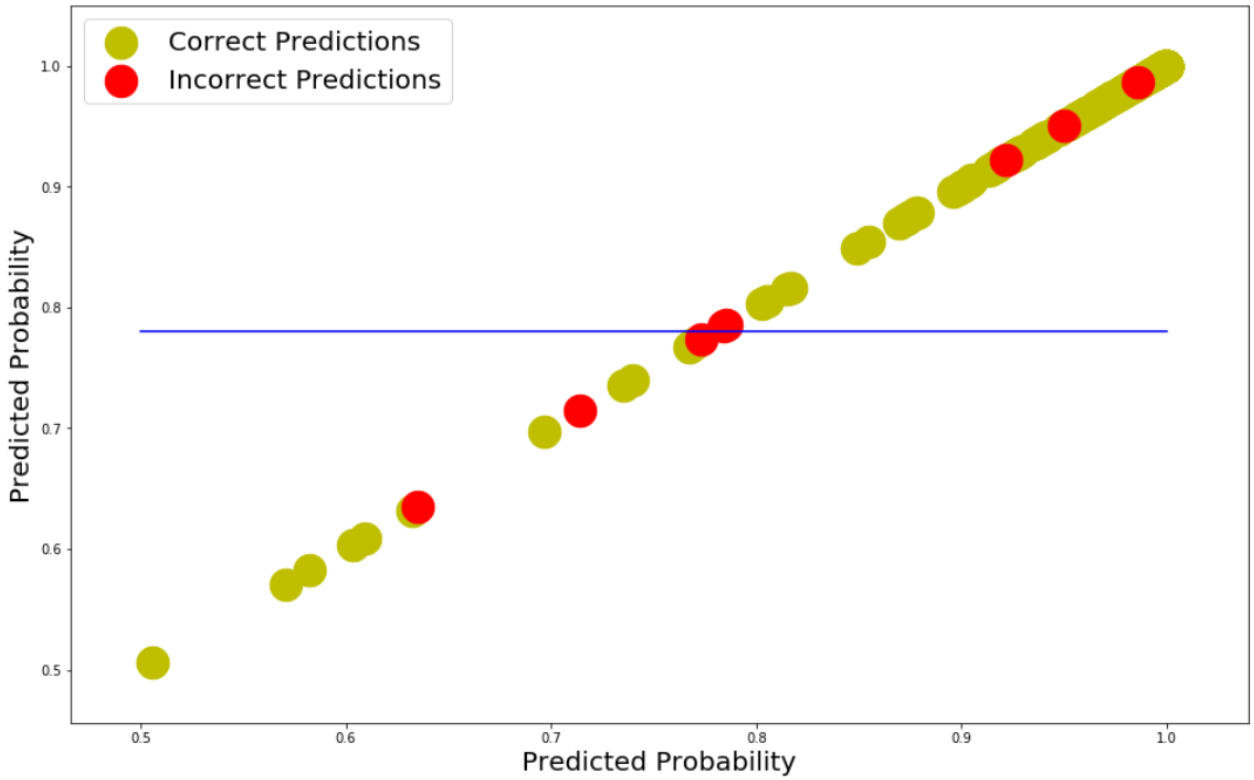
The accuracy is a bit lower but the execution time is much lesser and hence, we are using RandomizedSearchCV.

```
Precision_score:  [0.984026 0.971963]
Recall Score: [0.971609 0.984227]
f_score [0.977778 0.978056]
 The overall f1_score: 0.9780564263322884
```

Then we found out the shape of the dataframe containing the incorrect values and found out the total incorrect values to be 8 .

The linechart of the predict_probabilties is shown below :

X and Y axis are the probabilities of the predicted values



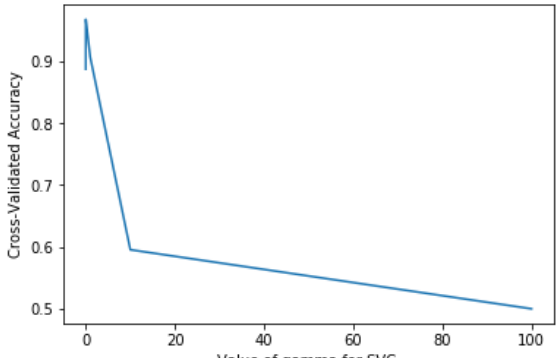
SVM KERNEL=RBF

Analyzing the accuracy score with different values of gamma

Out[154]:

	gamma_values	Accuracy_values
0	$0.00	0.8878
0	$0.00	0.9552
0	$0.01	0.9682
0	$0.10	0.9631
0	$1.00	0.9072
0	$10.00	0.5955
0	$100.00	0.5

Out[137]: Text(0,0.5,'Cross-Validated Accuracy')



Value of gamma for SVC

We found out that gamma value 0.01 gives the highest accuracy.

Hence, we give the gamma value along with other values as parameter to RandomizedSearchCV and see the results:

```
grid best score:  for class weight 0 0.5
grid best parameters:  {'random_state': 5, 'kernel': 'rbf', 'gamma': 0.01, 'class_weight': {0: 0}, 'C': 1.0}
grid best score:  for class weight 1 0.7052091554853985
grid best parameters:  {'random_state': 13, 'kernel': 'rbf', 'gamma': 0.01, 'class_weight': {0: 1}, 'C': 1.0}
grid best score:  for class weight 2 0.6606156274664562
grid best parameters:  {'random_state': 5, 'kernel': 'rbf', 'gamma': 0.01, 'class_weight': {0: 2}, 'C': 1.0}
grid best score:  for class weight 3 0.5868192580899764
grid best parameters:  {'random_state': 10, 'kernel': 'rbf', 'gamma': 0.01, 'class_weight': {0: 3}, 'C': 1.0}
grid best score:  for class weight 4 0.5189423835832676
grid best parameters:  {'random_state': 5, 'kernel': 'rbf', 'gamma': 0.01, 'class_weight': {0: 4}, 'C': 1.0}
grid best score:  for class weight 5 0.5189423835832676
grid best parameters:  {'random_state': 5, 'kernel': 'rbf', 'gamma': 0.01, 'class_weight': {0: 5}, 'C': 1.0}
grid best score:  for class weight 6 0.5189423835832676
grid best parameters:  {'random_state': 5, 'kernel': 'rbf', 'gamma': 0.01, 'class_weight': {0: 6}, 'C': 1.0}
grid best score:  for class weight 7 0.5189423835832676
grid best parameters:  {'random_state': 10, 'kernel': 'rbf', 'gamma': 0.01, 'class_weight': {0: 7}, 'C': 1.0}
grid best score:  for class weight 8 0.5189423835832676
grid best parameters:  {'random_state': 5, 'kernel': 'rbf', 'gamma': 0.01, 'class_weight': {0: 8}, 'C': 1.0}
grid best score:  for class weight 9 0.5189423835832676
grid best parameters:  {'random_state': 5, 'kernel': 'rbf', 'gamma': 0.01, 'class_weight': {0: 9}, 'C': 1.0}
```

We see that calss weight{0:1} gives the best prediction

Then we go ahead with class weight =0:1 and use RandomizedSearchCV:

```
grid best score:  for class weight  0.7154696132596685
grid best parameters:  {'random_state': 5, 'kernel': 'rbf', 'class_weight': {0: 1}, 'C': 1.0}
Execution time: 72.78450512886047 ms
```

Test Accuracy:

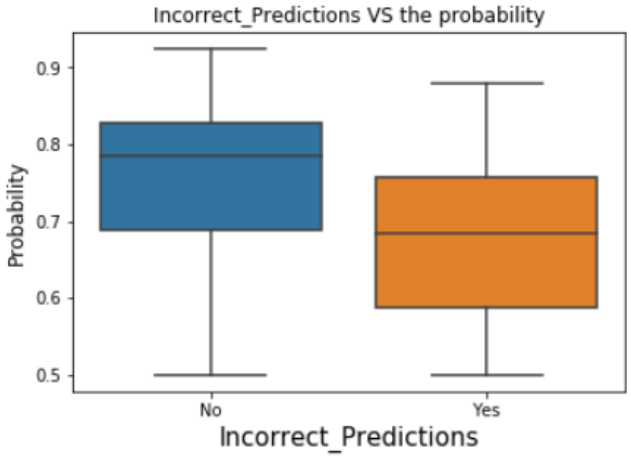
```
Test_accuracy:  0.7350157728706624
```

other results:

```
Precision_score:  [0.757785 0.715942]
Recall Score: [0.690852 0.779180]
f_score [0.722772 0.746224]
 The overall f1_score: 0.7462235649546827
```

Then, we used the predict_proba method of RandomizedSearchCV to find out the probablity of each predicted value.

Boxplot showing the probability range of the incorrect and correct predicted values



1 implies Actual values are not equal to Predicted Values

0 implies Actual values equal to Predicted Values

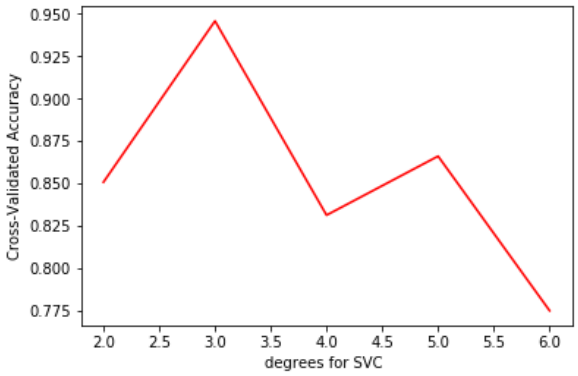
Here we see see that the predicted values which are correct has a greater probability

SVM KERNEL=POLY

Analyzing the accuracy score with different values of polynomial degree

	poly_values	Accuracy_values
0	2	0.8507
0	3	0.9458
0	4	0.8312
0	5	0.866
0	6	0.7748

```
Text(0,0.5,'Cross-Validated Accuracy')
```



We found out that degree value 3 gives the highest accuracy.

Hence, we give the degree value along with other values as parameter to RandomizedSearchCV and see the results:

```
grid best score:  for class weight  0.9494869771112865
grid best parameters:  {'random_state': 5, 'kernel': 'poly', 'degree': 3, 'class_weight': {0: 1}, 'C': 1.0}
grid best score:  for class weight  0.9526440410418311
grid best parameters:  {'random_state': 45, 'kernel': 'poly', 'degree': 3, 'class_weight': {0: 1}, 'C': 1.0}
grid best score:  for class weight  0.9522494080505131
grid best parameters:  {'random_state': 45, 'kernel': 'poly', 'degree': 3, 'class_weight': {0: 1}, 'C': 1.0}
grid best score:  for class weight  0.9526440410418311
grid best parameters:  {'random_state': 45, 'kernel': 'poly', 'degree': 3, 'class_weight': {0: 1}, 'C': 1.0}
grid best score:  for class weight  0.9534333070244673
grid best parameters:  {'random_state': 5, 'kernel': 'poly', 'degree': 3, 'class_weight': {0: 1}, 'C': 1.0}
grid best score:  for class weight  0.9538279400157853
grid best parameters:  {'random_state': 13, 'kernel': 'poly', 'degree': 3, 'class_weight': {0: 1}, 'C': 1.0}
grid best score:  for class weight  0.9538279400157853
grid best parameters:  {'random_state': 45, 'kernel': 'poly', 'degree': 3, 'class_weight': {0: 1}, 'C': 1.0}
```

Since class_weight {0:1} has the greatest grid_score, we go ahead with class weight =0:1 and use RandomizedSearchCV:

```
grid best score:  for class weight  0.9696132596685083
grid best parameters:  {'random_state': 10, 'kernel': 'poly', 'degree': 1, 'class_weight': {0: 1}, 'C': 1.0}
Execution time: 23.117100954055786 ms
```

Test Accuracy:

```
Test_accuracy:  0.9747634069400631
```

The confusion matrix result as shown below:

```
The test dataset confusion_matrix is :
 [[314   3]
 [ 10 307]]
```

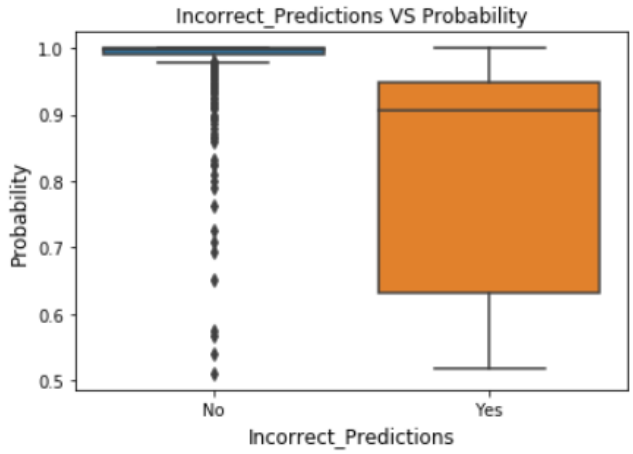
The incorrect values total is 13.

The next dataframe is the dataframe of incorrect predicted values and their predicted proabablities of being the incorrect one

	Probability	Actual	Predicted	Sameornot
10	$0.87	1	0	1
58	$0.95	1	0	1
59	$0.73	1	0	1
185	$0.60	0	1	1
189	$0.65	0	1	1
253	$0.99	1	0	1
256	$0.59	1	0	1
267	$0.54	1	0	1
296	$0.58	0	1	1
487	$0.95	1	0	1
555	$0.75	1	0	1
578	$1.00	1	0	1
588	$0.88	1	0	1

Then, we used the predict_proba method of RandomizedSearchCV to find out the probablity of each predicted value.

Boxplot showing the probability range of the incorrect and correct predicted values



1 implies Actual values are not equal to Predicted Values

0 implies Actual values equal to Predicted Values

Here we see see that the predicted values which are correct has a greater probability

Voice Recognition Code

In [199]:
```
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
import seaborn as sns
from IPython.display import display
from sklearn.model_selection import train_test_split
```

```
from sklearn.model_selection import train_test_split
```

In [200]:
```
df =pd.read_csv('VoiceRecognition.csv')
df.head()
```

Out[200]:

	meanfreq	sd	median	Q25	Q75	IQR	skew	kurt	sp.ent	sfm	...
0	$0.06</td> <td>$0.06	$0.03</td> <td>$0.02	$0.09</td> <td>$0.08	$12.86</td> <td>$274.40	$0.89</td> <td>$0.49	...	$0.06</td> <td>$0.08	$0.02</td> <td>$0.28	$0.01</td> <td>$0.01	$0.01</td> <td>$0.00	$0.00</td> <td>male< </tr> <tr> <th>1</th> <td>$0.07
2	$0.08</td> <td>$0.08	$0.04</td> <td>$0.01	$0.13</td> <td>$0.12	$30.76</td> <td>$1,024.93	$0.85</td> <td>$0.48	...	$0.08</td> <td>$0.10	$0.02</td> <td>$0.27	$0.01</td> <td>$0.01	$0.02</td> <td>$0.01	$0.05</td> <td>male< </tr> <tr> <th>3</th> <td>$0.15
4	$0.14</td> <td>$0.08	$0.12</td> <td>$0.08	$0.21</td> <td>$0.13	$1.10</td> <td>$4.33	$0.97</td> <td>$0.78	...	$0.14</td> <td>$0.11	$0.02</td> <td>$0.27	$0.71</td> <td>$0.01	$5.48</td> <td>$5.48	$0.21

5 rows × 21 columns

In [201]:
```
df.columns.values
```

Out[201]:
```
array(['meanfreq', 'sd', 'median', 'Q25', 'Q75', 'IQR', 'skew', 'kurt',
       'sp.ent', 'sfm', 'mode', 'centroid', 'meanfun', 'minfun', 'maxfun',
       'meandom', 'mindom', 'maxdom', 'dfrange', 'modindx', 'label'],
      dtype=object)
```

In [202]:
```
df[df.duplicated(keep='first')]
```

Out[202]:

	meanfreq	sd	median	Q25	Q75	IQR	skew	kurt	sp.ent	sfm	...
298	$0.21	$0.06</td> <td>$0.24	$0.14</td> <td>$0.26	$0.12</td> <td>$2.11	$7.89</td> <td>$0.86	$0.08</td> <td>... </td> <td>$0.21	$0.13</td> <td>$0.03	$0.25</td> <td>$0.82	$0.13</td> <td>$4.16	$4.03</td> <td>$0.23	ma
2403	$0.21</td> <td>$0.04	$0.22</td> <td>$0.19	$0.25</td> <td>$0.06	$1.86</td> <td>$6.11	$0.88</td> <td>$0.31	...	$0.21</td> <td>$0.14	$0.05</td> <td>$0.28	$1.93</td> <td>$0.02	$15.61</td> <td>$15.59	$0

2 rows × 21 columns

In [203]:
```
df.drop_duplicates(inplace=True)
```

In [204]:
```
df[df.duplicated(keep='first')]
```

Out[204]:

	meanfreq	sd	median	Q25	Q75	IQR	skew	kurt	sp.ent	sfm	...	centroid	meanfun	minfun	maxfun	meandom	mindo

0 rows × 21 columns

In [205]:
```
df.dtypes
```

Out[205]:
```
meanfreq    float64
sd          float64
median      float64
Q25         float64
Q75         float64
IQR         float64
skew        float64
kurt        float64
sp.ent      float64
sfm         float64
mode        float64
centroid    float64
meanfun     float64
minfun      float64
maxfun      float64
meandom     float64
mindom      float64
maxdom      float64
dfrange     float64
```

```
dfrange     float64
modindx     float64
label        object
dtype: object
```

In [259]:
```
# Removing the $ symbol from every column
df1 = df.iloc[:,:-1].astype('str').replace('$','',regex=True)
```

In [260]:
```
df1= df1.join(df['label'])
df1.head()
```

Out[260]:

	meanfreq	sd	median	Q25	Q75	IQ
0	0.059780999999999994	0.0642413	0.032026900000000004	0.0150715	0.09019339999999999	0.0
1	0.0660087	0.06731000000000001	0.0402287	0.0194139	0.09266619999999999	0.0
2	0.0773155	0.0838294	0.0367185	0.008701100000000002	0.131908	0.1
3	0.1512281	0.07211060000000001	0.15801120000000002	0.0965817	0.20795529999999998	0.1
4	0.1351204	0.0791461	0.1246562	0.0787202	0.2060449	0.1

5 rows × 21 columns

In [261]:
```
df1.dtypes
```

Out[261]:
```
meanfreq    object
sd          object
median      object
Q25         object
Q75         object
IQR         object
skew        object
kurt        object
sp.ent      object
sfm         object
mode        object
centroid    object
meanfun     object
minfun      object
maxfun      object
meandom     object
mindom      object
maxdom      object
dfrange     object
modindx     object
label       object
dtype: object
```

In [209]:
```
df.shape
```

Out[209]:
```
(3166, 21)
```

In [210]:
```
np.set_printoptions(formatter={'float_kind':'{:3f}'.format})
from sklearn.feature_selection import SelectKBest,chi2,SelectPercentile,f_classif
selValue1 = SelectKBest(f_classif,k=20).fit(df.iloc[:,:-1],df.iloc[:,-1])
print('\nSelectKBest feature_extraction_text using f_classif:Value\n')
print('pvalues_:  ',selValue1.pvalues_)
print('\nscores:    ',selValue1.scores_)
print(selValue1.get_support(indices=True))
selValue2 = SelectKBest(chi2,k=20).fit(df.iloc[:,:-1],df.iloc[:,-1])
print('\nSelectKBest feature_extraction_text using chi2:Value\n')
print('pvalues_:  ',selValue2.pvalues_)
print('\nscores:    ',selValue2.scores_)
print('\nSelectPercentile feature_extraction_text:Value\n')
# Used SelectPercentile feature_extraction_text
selValue3 = SelectPercentile(chi2,percentile=100).fit(df.iloc[:,:-1],df.iloc[:,-1])
print('pvalues_:  ',selValue3.pvalues_)
print('\nscores:    ',selValue3.scores_)
```

```
SelectKBest feature_extraction_text using f_classif:Value

pvalues_:   [0.000000 0.000000 0.000000 0.000000 0.000169 0.000000 0.039359 0.000001
 0.000000 0.000000 0.000000 0.000000 0.000000 0.000000 0.000000 0.000000
 0.000000 0.000000 0.000000 0.085898]

scores:     [407.163482 944.502049 278.408026 1120.476092 14.178175 1962.973919
 4.248838 24.253878 1004.510839 465.788494 96.583315 407.163482
```

```
 7236.389947 60.013861 89.950060 119.266417 125.918148 125.028709
 120.465472 2.951460]
[ 0  1  2  3  4  5  6  7  8  9 10 11 12 13 14 15 16 17 18 19]

SelectKBest feature_extraction_text using chi2:Value

pvalues_:   [0.181482 0.060157 0.176985 0.000186 0.851390 0.000000 0.000001 0.000000
 0.189120 0.000000 0.065900 0.181482 0.000060 0.441775 0.580188 0.000000
 0.002390 0.000000 0.000000 0.622505]

scores:     [1.785445 3.533062 1.822758 13.970832 0.035098 26.317371 24.314480
 11987.900828 1.724459 31.327900 3.382309 1.785445 16.101725 0.591669
 0.305931 38.232633 9.222764 295.068090 287.506888 0.242363]

SelectPercentile feature_extraction_text:Value

pvalues_:   [0.181482 0.060157 0.176985 0.000186 0.851390 0.000000 0.000001 0.000000
 0.189120 0.000000 0.065900 0.181482 0.000060 0.441775 0.580188 0.000000
 0.002390 0.000000 0.000000 0.622505]

scores:     [1.785445 3.533062 1.822758 13.970832 0.035098 26.317371 24.314480
 11987.900828 1.724459 31.327900 3.382309 1.785445 16.101725 0.591669
 0.305931 38.232633 9.222764 295.068090 287.506888 0.242363]
```

In [211]:
```
dt = pd.DataFrame({'Columns':df.columns.values[:-1],'P_value':pd.Series(selValue2.pvalues_),'scores':pd.Se
ries(selValue2.scores_)})
```

In [212]:
```
pd.options.display.float_format = '${:,.2f}'.format
dt.sort_values(by='scores',ascending=False,inplace=True)
dt
```

Out[212]:

	Columns	P_value	scores
7	kurt	$0.00</td> <td>$11,987.90	
17	maxdom	$0.00</td> <td>$295.07	
18	dfrange	$0.00</td> <td>$287.51	
15	meandom	$0.00</td> <td>$38.23	
9	sfm	$0.00</td> <td>$31.33	
5	IQR	$0.00</td> <td>$26.32	
6	skew	$0.00</td> <td>$24.31	
12	meanfun	$0.00</td> <td>$16.10	
3	Q25	$0.00</td> <td>$13.97	
16	mindom	$0.00</td> <td>$9.22	
1	sd	$0.06</td> <td>$3.53	
10	mode	$0.07</td> <td>$3.38	
2	median	$0.18</td> <td>$1.82	
0	meanfreq	$0.18</td> <td>$1.79	
11	centroid	$0.18</td> <td>$1.79	
8	sp.ent	$0.19</td> <td>$1.72	
13	minfun	$0.44</td> <td>$0.59	
14	maxfun	$0.58</td> <td>$0.31	
19	modindx	$0.62</td> <td>$0.24	
4	Q75	$0.85</td> <td>$0.04	

In [213]:
```
dfnew=pd.DataFrame()
for each in dt.Columns.values:
    dfnew = pd.concat([dfnew,pd.DataFrame(df1[each])],axis=1)
dfnew = dfnew.join(pd.DataFrame(df['label']))
dfnew = dfnew.drop(dfnew.loc[:,'sd':'Q75'].columns,axis=1)
dfnew.head()
```

Out[213]:

	kurt	maxdom	dfrange	meandom	sfm	IQR	skew
0	274.40290550000003	0.0078125	0.0	0.0078125	0.4919177999999996	0.07512200000000001	12.863

1	634.6138545	0.0546875	0.046875	0.009014399999999999	0.5137238000000001	0.0732523	22.423
2	1024.927705	0.015625	0.0078125	0.007990100000000002	0.478905	0.123207	30.757
3	4.1772962	0.5625	0.5546875	0.2014974	0.7272318	0.11137349999999999	1.2328
4	4.333713200000001	5.484375	5.4765625	0.7128125	0.7835681	0.1273247	1.1011

In [214]:
```
dfnew.columns.values
```

Out[214]:
```
array(['kurt', 'maxdom', 'dfrange', 'meandom', 'sfm', 'IQR', 'skew',
       'meanfun', 'Q25', 'mindom', 'label'], dtype=object)
```

Logistics Regression: GridSearchCV

In [215]:
```
from sklearn.model_selection import GridSearchCV,RandomizedSearchCV,train_test_split
from sklearn.linear_model import LogisticRegression
from sklearn.preprocessing import LabelEncoder,StandardScaler
import time
dfnew['label'] = (LabelEncoder().fit_transform(dfnew['label']))
train_set,test_set = train_test_split(dfnew,random_state=9,test_size=0.2,stratify=df['label'])
#display(test_set.loc[:,'kurt':'sd'])
lm=LogisticRegression()
start_time=time.time()
params = dict(penalty=['l1','l2',],C=[0.2,0.5,0.7,1.0],random_state=[5,10,45,24,17],multi_class=['ovr'])
grid = GridSearchCV(lm,params,cv=5).fit(train_set.loc[:,'kurt':'mindom'],train_set['label'])
print('For GridSearchCV:')
print('grid best score for train_set: ',grid.best_score_)
print('grid best parameters for train_set: ',grid.best_params_)
print("Execution time: " + str((time.time() - start_time)) + ' ms')
```

```
For GridSearchCV:
grid best score for train_set:  0.9676145339652449
grid best parameters for train_set:  {'C': 1.0, 'multi_class': 'ovr', 'penalty': 'l1', 'random_state': 5}
Execution time: 13.599502086639404 ms
```

In [216]:
```
print('Test_accuracy: ',grid.score(test_set.loc[:,'kurt':'mindom'],test_set['label']))
```

```
Test_accuracy:  0.9826498422712934
```

Logistic Regression: RandomizedSearchCV

In [217]:
```
from sklearn.model_selection import GridSearchCV,RandomizedSearchCV,train_test_split
from sklearn.linear_model import LogisticRegression
from sklearn.preprocessing import LabelEncoder,StandardScaler
import time
#dfnew['label'] = (LabelEncoder().fit_transform(dfnew['label']))
train_set,test_set = train_test_split(dfnew,random_state=9,test_size=0.2,stratify=df['label'])
#display(test_set.loc[:,'kurt':'sd'])
lm=LogisticRegression()
start_time=time.time()
params = dict(penalty=['l1','l2',],C=[0.2,0.5,0.7,1.0],random_state=[5,10,45,24,17],multi_class=['ovr'])
grid = RandomizedSearchCV(lm,params,cv=5).fit(train_set.loc[:,'kurt':'mindom'],train_set['label'])
print('For RandomizedSearchCV:')
print('grid best score for train_set: ',grid.best_score_)
print('grid best parameters for train_set: ',grid.best_params_)
print("Execution time: " + str((time.time() - start_time)) + ' ms')
```

```
For RandomizedSearchCV:
grid best score for train_set:  0.9640600315955766
grid best parameters for train_set:  {'random_state': 45, 'penalty': 'l1', 'multi_class': 'ovr', 'C': 0.5}
Execution time: 3.2223386764526367 ms
```

In [218]:
```
print('Test_accuracy: ',grid.score(test_set.loc[:,'kurt':'mindom'],test_set['label']))
```

```
Test_accuracy:  0.9810725552050473
```

In [219]:
```
from sklearn.metrics import confusion_matrix,precision_recall_fscore_support,f1_score
from sklearn.preprocessing import LabelEncoder
#dfnew['label'] = (LabelEncoder().fit_transform(dfnew['label']))
#display(test_set.loc[:,'kurt':'sd'])
lm=LogisticRegression()
lm.fit(train_set.loc[:,'kurt':'mindom'],train_set['label'])
y_pred = lm.predict(test_set.loc[:,'kurt':'mindom'])
```

```
print('The test dataset confusion_matrix is :\n',confusion_matrix(test_set['label'],y_pred))
```

```
The test dataset confusion_matrix is :
 [[287  30]
 [  8 309]]
```

In [220]:
```
p,r,f,s = precision_recall_fscore_support(test_set['label'], y_pred)
print('Precision_score: ',p,'\nRecall Score:',r,'\nf_score',f,'\n The overall f1_score: {} '
      .format(f1_score(test_set['label'],y_pred,average="weighted")))
```

```
Precision_score:  [0.972881 0.911504]
Recall Score: [0.905363 0.974763]
f_score [0.937908 0.942073]
 The overall f1_score: 0.9399908337318668
```

In [224]:
```
Label_probabability = pd.Series()
for each in lm.predict_proba((test_set.loc[:,'kurt':'mindom'])):
    Label_probabability = Label_probabability.append(pd.Series(np.amax(each)))
Label_probabability_DTFrame = pd.concat([Label_probabability.reset_index(drop=True),
                                         test_set['label'].reset_index(drop=True),
                                         pd.Series(lm.predict(test_set.loc[:,'kurt':'mindom']))],axis=1,
                                        keys=['Probability','Actual','Predicted'])
Label_probabability_DTFrame['SameOrNot']= abs(Label_probabability_DTFrame.Actual -Label_probabability_DTFr
ame.Predicted)
Label_probabability_DTFrame[Label_probabability_DTFrame.SameOrNot !=0].head()
```

Out[224]:

	Probability	Actual	Predicted	SameOrNot
10	$0.55</td> <td>0</td> <td>1</td> <td>1</td> </tr> <tr> <th>17</th> <td>$0.73	0	1	1
20	$0.81</td> <td>0</td> <td>1</td> <td>1</td> </tr> <tr> <th>35</th> <td>$0.63	0	1	1
57	$0.74	0	1	1

In [ ]:
```

```

In [24]:
```
## Using Multinomial parameter
```

In [25]:
```
#### train_set,test_set = train_test_split(dfnew,random_state=3,test_size=0.2,stratify=df['label'])
#display(test_set.loc[:,'kurt':'sd'])
import warnings
warnings.simplefilter('ignore')
lm=LogisticRegression()
params = dict(penalty=['l2',],C=[0.2,0.5,0.7,1.0],random_state=[5,10,45,24,17],solver=['sag','saga'],
              multi_class=['multinomial'])
grid = RandomizedSearchCV(lm,params,cv=5).fit(train_set.loc[:,'kurt':'mindom'],train_set['label'])
print('grid best score: ',grid.best_score_)
print('grid best paramters: ',grid.best_params_)
```

```
grid best score:  0.5240726124704025
grid best paramters:  {'solver': 'sag', 'random_state': 10, 'penalty': 'l2', 'multi_class': 'multinomial',
'C': 1.0}
```

In [26]:
```
print('Test_accuracy: ',grid.score(test_set.loc[:,'kurt':'mindom'],test_set['label']))
```

```
Test_accuracy:  0.5299684542586751
```

SVM Linear

Using Cross Validation with Different values of C

In [262]:
```
from sklearn.cross_validation import cross_val_score
from sklearn.svm import SVC
from sklearn.preprocessing import StandardScaler
X=StandardScaler().fit_transform(df1.iloc[:,:-1])
C_List=pd.Series()
Acc_List=pd.Series()
DTFrame=pd.DataFrame()
for i in range(1,10):
    svcCrossLinear = SVC(kernel='linear',C=i)
    scores=cross_val_score(svcCrossLinear,X,df1.iloc[:,-1].values,scoring='accuracy',cv=10)
    C_List = C_List.append(pd.Series(i))
    Acc_List = Acc_List.append(pd.Series(round(scores.mean(),4)))
```

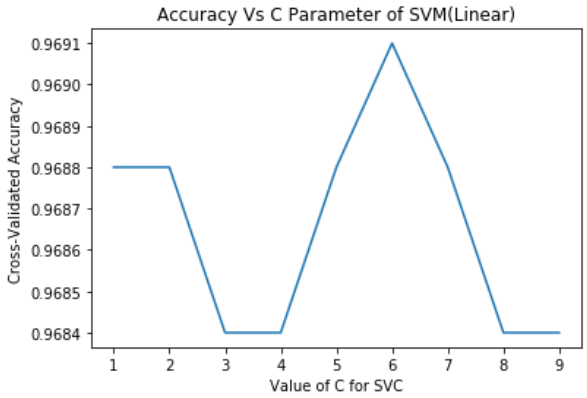
```
    Acc_List = Acc_List.append(pd.Series(round(scores.mean(),4)))
DTFrame= pd.concat([C_List,Acc_List],axis=1,keys=['C_values','Accuracy_values'])
DTFrame['Accuracy_values'] = DTFrame['Accuracy_values'].apply(lambda x : str(x).replace('$',''))
DTFrame
```

Out[262]:

	C_values	Accuracy_values
0	1	0.9688
0	2	0.9688
0	3	0.9684
0	4	0.9684
0	5	0.9688
0	6	0.9691
0	7	0.9688
0	8	0.9684
0	9	0.9684

In [263]:
```
sns.lineplot(x=C_List,y=Acc_List)
plt.xlabel('Value of C for SVC')
plt.ylabel('Cross-Validated Accuracy')
plt.title('Accuracy Vs C Parameter of SVM(Linear)')
```

Out[263]: Text(0.5,1,'Accuracy Vs C Parameter of SVM(Linear)')



Using GridSearchCV

In [269]:
```
from sklearn.model_selection import GridSearchCV,RandomizedSearchCV,train_test_split
from sklearn.svm import SVC
from sklearn.preprocessing import LabelEncoder,StandardScaler
import time
dfnew['label'] = (LabelEncoder().fit_transform(dfnew['label']))
train_set,test_set = train_test_split(dfnew,random_state=9,test_size=0.2,stratify=df['label'])
#display(test_set.loc[:,'kurt':'sd'])
train_set.loc[:,'kurt':'mindom']=StandardScaler().fit_transform(train_set.loc[:,'kurt':'mindom'])
test_set.loc[:,'kurt':'mindom'] =StandardScaler().fit_transform(test_set.loc[:,'kurt':'mindom'])
svcLinearGrid = SVC(probability=True)
start_time=time.time()
params=dict(C=[6],kernel=['linear'],random_state=[5,10,13,45],class_weight=[{0:1},{0:2},{0:6}])
grid=GridSearchCV(svcLinearGrid,params,cv=10).fit(train_set.loc[:,'kurt':'mindom'],train_set['label'])
print('grid best score: ',grid.best_score_)
print('grid best paramters: ',grid.best_params_)
print("Execution time: " + str((time.time() - start_time)) + ' ms')
```

```
C:\Users\supratik chanda\Documents\New folder (2)\lib\site-packages\pandas\core\indexing.py:543: SettingWi
thCopyWarning:
A value is trying to be set on a copy of a slice from a DataFrame.
Try using .loc[row_indexer,col_indexer] = value instead

See the caveats in the documentation: http://pandas.pydata.org/pandas-docs/stable/indexing.html#indexing-v
iew-versus-copy
  self.obj[item] = s
grid best score:  0.9691943127962085
grid best paramters:  {'C': 6, 'class_weight': {0: 1}, 'kernel': 'linear', 'random_state': 5}
Execution time: 72.60279583930969 ms
```

In [270]:
```
print('Test_accuracy: ',grid.score(test_set.loc[:,'kurt':'mindom'],test_set['label']))
```

```
Test_accuracy:  0.9779179810725552
```

Using RandomizedSearchCV

In [316]:
```
from sklearn.model_selection import GridSearchCV,RandomizedSearchCV,train_test_split
from sklearn.svm import SVC
from sklearn.preprocessing import LabelEncoder,StandardScaler
import time
import warnings
warnings.simplefilter('ignore')
dfnew['label'] = (LabelEncoder().fit_transform(dfnew['label']))
sc=StandardScaler()
train_set,test_set = train_test_split(dfnew,random_state=9,test_size=0.2,stratify=df['label'])
train_set.loc[:,'kurt':'mindom'] =sc.fit_transform(train_set.loc[:,'kurt':'mindom'])
test_set.loc[:,'kurt':'mindom']=sc.transform(test_set.loc[:,'kurt':'mindom'])
#display(test_set.loc[:,'kurt':'sd'])
svcLinearRandom = SVC(probability=True)
start_time=time.time()
params=dict(C=[6],kernel=['linear'],random_state=[5,10,13,45],class_weight=[{0:1},{0:2},{0:6}])
grid=RandomizedSearchCV(svcLinearRandom,params,cv=5).fit(train_set.loc[:,'kurt':'mindom'],train_set['labe
l'])
print('grid best score: ',grid.best_score_)
print('grid best parameters: ',grid.best_params_)
print("Execution time: " + str((time.time() - start_time)) + ' ms')
```

```
grid best score:  0.9691943127962085
grid best parameters:  {'random_state': 45, 'kernel': 'linear', 'class_weight': {0: 1}, 'C': 6}
Execution time: 22.810219764709473 ms
```

In [317]:
```
print('Test_accuracy: ',grid.score(test_set.loc[:,'kurt':'mindom'],test_set['label']))
```

```
Test_accuracy:  0.9779179810725552
```

In [318]:
```
y_pred=grid.predict(test_set.loc[:,'kurt':'mindom'])
p,r,f,s = precision_recall_fscore_support(test_set['label'], y_pred)
print('Precision_score: ',p,'\nRecall Score:',r,'\nf_score',f,'\n The overall f1_score: {}  '
      .format(f1_score(test_set['label'],y_pred)))
```

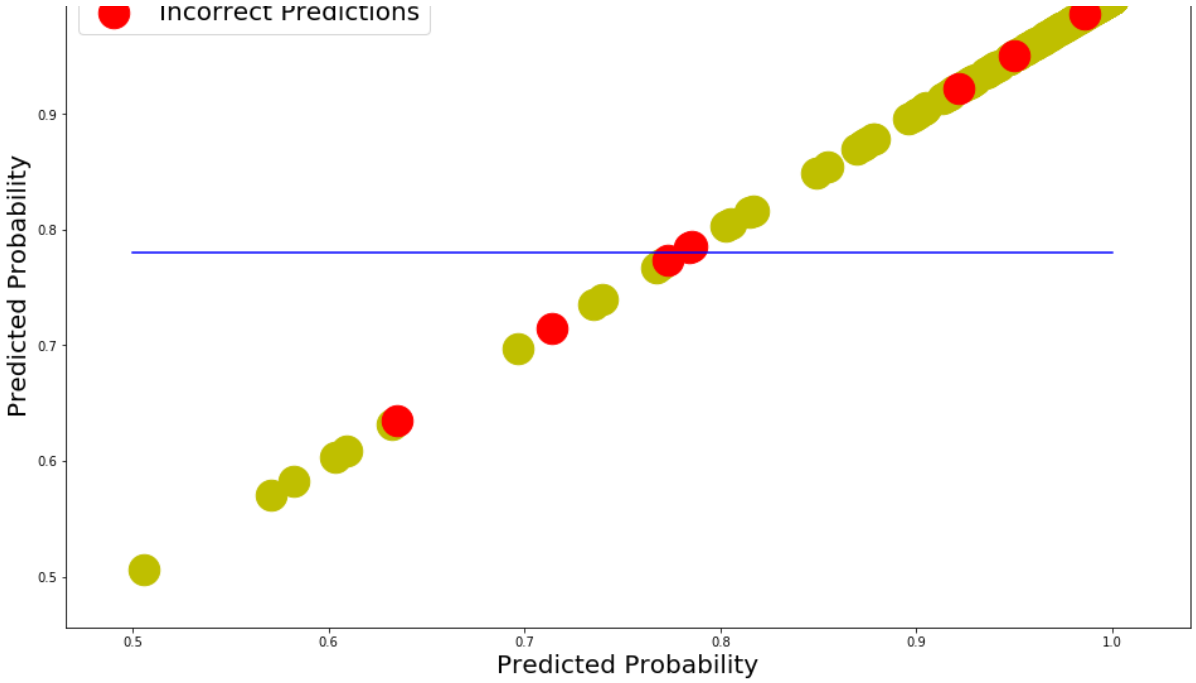
```
Precision_score:  [0.984026 0.971963]
Recall Score: [0.971609 0.984227]
f_score [0.977778 0.978056]
 The overall f1_score: 0.9780564263322884
```

In [319]:
```
Label_probabability = pd.Series()
svcLinearRandom.fit(train_set.loc[:,'kurt':'mindom'],train_set['label'])
for each in svcLinearRandom.predict_proba((test_set.loc[:,'kurt':'mindom'])):
    Label_probabability = Label_probabability.append(pd.Series(np.amax(each)))
Label_probabability_DTFrame = pd.concat([Label_probabability.reset_index(drop=True),
                                         test_set['label'].reset_index(drop=True),
                                         pd.Series(svcLinearRandom.predict(test_set.loc[:,'kurt':'mindom'
]))],axis=1,
                                        keys=['Probability','Actual','Predicted'])
Label_probabability_DTFrame['SameOrNot']=abs(Label_probabability_DTFrame.Actual- Label_probabability_DTFra
me.Predicted)
Label_probabability_DTFrame[Label_probabability_DTFrame.SameOrNot!=0].shape
```

Out[319]:
```
(8, 4)
```

In [320]:
```
plt.figure(figsize=(16,10))
plt.scatter(Label_probabability_DTFrame.query('SameOrNot == 0').Probability,Label_probabability_DTFrame.qu
ery('SameOrNot == 0').Probability,
            color='y',marker='o',s=600,label='Correct Predictions')
plt.scatter(Label_probabability_DTFrame.query('SameOrNot == 1').Probability,Label_probabability_DTFrame.qu
ery('SameOrNot == 1').Probability,
            color='r',marker='o',s=600,label='Incorrect Predictions')
plt.plot(pd.Series([0.5,1]),pd.Series([0.78,0.78]),color='b')
plt.xlabel('Predicted Probability',fontsize=20)
plt.ylabel('Predicted Probability',fontsize=20)
plt.legend(loc='best',fontsize=20)
```

Out[320]:
```
<matplotlib.legend.Legend at 0x282c801c4a8>
```



SVM rbf

Taking kernel as rbf and taking different values gamma

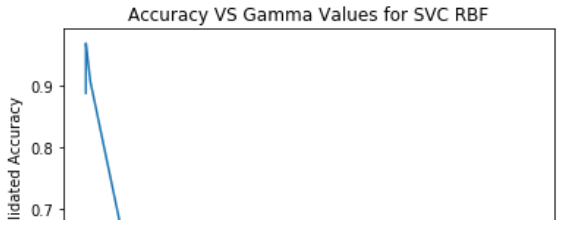
In [280]:
```
from sklearn.cross_validation import cross_val_score
from sklearn.svm import SVC
from sklearn.preprocessing import StandardScaler
X=StandardScaler().fit_transform(df1.iloc[:,:-1])
C_List=pd.Series()
Acc_List=pd.Series()
DTFrame=pd.DataFrame()
gamma_range=[0.0001,0.001,0.01,0.1,1,10,100]
for i in gamma_range:
    svcCrossRBF = SVC(kernel='rbf',gamma=i)
    scores=cross_val_score(svcCrossRBF,X,df1.iloc[:,-1].values,scoring='accuracy',cv=10)
    C_List = C_List.append(pd.Series(i))
    Acc_List = Acc_List.append(pd.Series(round(scores.mean(),4)))
DTFrame= pd.concat([C_List,Acc_List],axis=1,keys=['gamma_values','Accuracy_values'])
DTFrame['Accuracy_values'] = DTFrame['Accuracy_values'].apply(lambda x : str(x).replace('$',''))
DTFrame
```

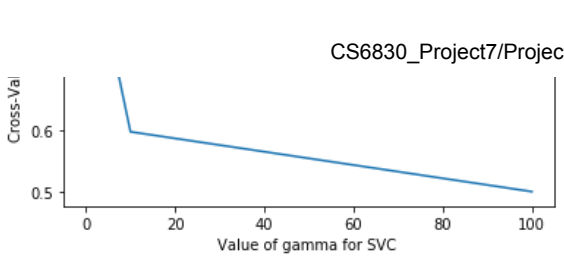
Out[280]:

	gamma_values	Accuracy_values
0	$0.00	0.8878
0	$0.00</td> <td>0.9559</td> </tr> <tr> <th>0</th> <td>$0.01	0.9681
0	$0.10</td> <td>0.9631</td> </tr> <tr> <th>0</th> <td>$1.00	0.9068
0	$10.00</td> <td>0.5968</td> </tr> <tr> <th>0</th> <td>$100.00	0.5

In [289]:
```
sns.lineplot(x=C_List,y=Acc_List)
plt.xlabel('Value of gamma for SVC ')
plt.ylabel('Cross-Validated Accuracy')
plt.title('Accuracy VS Gamma Values for SVC RBF')
```

Out[289]: Text(0.5,1,'Accuracy VS Gamma Values for SVC RBF')





Taking the gamma value for highest accuracy and putting that in RandomizedSearchCV

In [282]:
```
from sklearn.model_selection import GridSearchCV,RandomizedSearchCV,train_test_split
from sklearn.svm import SVC
from sklearn.preprocessing import LabelEncoder,StandardScaler
import time
dfnew['label'] = (LabelEncoder().fit_transform(dfnew['label']))
train_set,test_set = train_test_split(dfnew,random_state=9,test_size=0.2,stratify=df['label'])
#display(test_set.loc[:,'kurt':'sd'])
svc = SVC(probability=True)
start_time=time.time()
for i in range(0,10):
    params=dict(C=[0.2,0.3,0.6,1.0],kernel=['rbf'],random_state=[5,10,13,45],gamma=[0.01],class_weight=[{0
:i}])
    grid=RandomizedSearchCV(svc,params,cv=5).fit(train_set.loc[:,'kurt':'mindom'],train_set['label'])
    print('grid best score:  for class weight '+ str(i),grid.best_score_)
    print('grid best parameters: ',grid.best_params_)
```

```
grid best score:  for class weight 0 0.5
grid best parameters:  {'random_state': 5, 'kernel': 'rbf', 'gamma': 0.01, 'class_weight': {0: 0}, 'C': 0.
6}
grid best score:  for class weight 1 0.6978672985781991
grid best parameters:  {'random_state': 10, 'kernel': 'rbf', 'gamma': 0.01, 'class_weight': {0: 1}, 'C':
1.0}
grid best score:  for class weight 2 0.655608214849921
grid best parameters:  {'random_state': 10, 'kernel': 'rbf', 'gamma': 0.01, 'class_weight': {0: 2}, 'C':
1.0}
grid best score:  for class weight 3 0.5821484992101106
grid best parameters:  {'random_state': 13, 'kernel': 'rbf', 'gamma': 0.01, 'class_weight': {0: 3}, 'C':
1.0}
grid best score:  for class weight 4 0.514612954186414
grid best parameters:  {'random_state': 10, 'kernel': 'rbf', 'gamma': 0.01, 'class_weight': {0: 4}, 'C':
1.0}
grid best score:  for class weight 5 0.514612954186414
grid best parameters:  {'random_state': 45, 'kernel': 'rbf', 'gamma': 0.01, 'class_weight': {0: 5}, 'C':
1.0}
grid best score:  for class weight 6 0.514612954186414
grid best parameters:  {'random_state': 10, 'kernel': 'rbf', 'gamma': 0.01, 'class_weight': {0: 6}, 'C':
1.0}
grid best score:  for class weight 7 0.514612954186414
grid best parameters:  {'random_state': 10, 'kernel': 'rbf', 'gamma': 0.01, 'class_weight': {0: 7}, 'C':
1.0}
grid best score:  for class weight 8 0.514612954186414
grid best parameters:  {'random_state': 5, 'kernel': 'rbf', 'gamma': 0.01, 'class_weight': {0: 8}, 'C': 1.
0}
grid best score:  for class weight 9 0.514612954186414
grid best parameters:  {'random_state': 45, 'kernel': 'rbf', 'gamma': 0.01, 'class_weight': {0: 9}, 'C':
1.0}
```

Since the accuracy for class_weight 1 is the best we are going ahead with class_weight:1

In [283]:
```
from sklearn.model_selection import GridSearchCV,RandomizedSearchCV,train_test_split
from sklearn.svm import SVC
from sklearn.preprocessing import LabelEncoder,StandardScaler
import time
dfnew['label'] = (LabelEncoder().fit_transform(dfnew['label']))
train_set,test_set = train_test_split(dfnew,random_state=9,test_size=0.2,stratify=df['label'])
#display(test_set.loc[:,'kurt':'sd'])
svc = SVC(probability=True)
start_time=time.time()
params=dict(C=[0.2,0.3,0.6,1.0],kernel=['rbf'],random_state=[5,10,13,45],class_weight=[{0:1}])
grid=RandomizedSearchCV(svc,params,cv=5).fit(train_set.loc[:,'kurt':'mindom'],train_set['label'])
print('grid best score:  for class weight ',grid.best_score_)
print('grid best parameters: ',grid.best_params_)
print("Execution time: " + str((time.time() - start_time)) + ' ms')
```

```
grid best score:  for class weight  0.7014218009478673
grid best parameters:  {'random_state': 10, 'kernel': 'rbf', 'class_weight': {0: 1}, 'C': 1.0}
Execution time: 73.6187071800232 ms
```

In [284]:
```
print('Test_accuracy: ',grid.score(test_set.loc[:,'kurt':'mindom'],test_set['label']))
```

```
Test_accuracy:  0.7350157728706624
```

In [285]:
```
y_pred = grid.predict(test_set.loc[:,'kurt':'mindom'])
p,r,f,s = precision_recall_fscore_support(test_set['label'], y_pred)
print('Precision_score: ',p,'\nRecall Score:',r,'\nf_score',f,'\n The overall f1_score: {}'.format(f1_scor
e(test_set['label'],y_pred)))
```

```
Precision_score:  [0.745875 0.725076]
Recall Score: [0.712934 0.757098]
f_score [0.729032 0.740741]
 The overall f1_score: 0.7407407407407407
```

In [292]:
```
Label_probabability = pd.Series()
svc.fit(train_set.loc[:,'kurt':'mindom'],train_set['label'])
for each in svc.predict_proba((test_set.loc[:,'kurt':'mindom'])):
    Label_probabability = Label_probabability.append(pd.Series(np.amax(each)))
Label_probabability_DTFrame = pd.concat([Label_probabability.reset_index(drop=True),
                                         test_set['label'].reset_index(drop=True),
                                         pd.Series(svc.predict(test_set.loc[:,'kurt':'mindom']))],axis=1,
                                        keys=['Probability','Actual','Predicted'])
Label_probabability_DTFrame['SameOrNot']=abs(Label_probabability_DTFrame.Actual- Label_probabability_DTFra
me.Predicted)
Label_probabability_DTFrame['Incorrect_Predictions']=Label_probabability_DTFrame['SameOrNot'].apply(lambda
x: 'No' if x ==0 else 'Yes')
Label_probabability_DTFrame.head()
```

Out[292]:

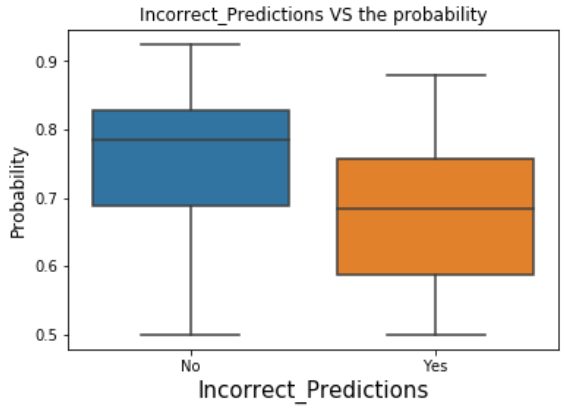
	Probability	Actual	Predicted	SameOrNot	Incorrect_Predictions
0	$0.75</td> <td>1</td> <td>1</td> <td>0</td> <td>No</td> </tr> <tr> <th>1</th> <td>$0.69	1	1	0	No
2	$0.78</td> <td>1</td> <td>1</td> <td>0</td> <td>No</td> </tr> <tr> <th>3</th> <td>$0.64	1	1	0	No
4	$0.57	1	1	0	No

In [293]:
```
from sklearn.metrics import confusion_matrix as cm
svm_pred=grid.predict(test_set.loc[:,'kurt':'mindom'])
matrix = cm(test_set['label'],svm_pred)
print('The confusion matrix is: ')
print(matrix)
```

```
The confusion matrix is:
[[226  91]
 [ 77 240]]
```

In [299]:
```
sns.boxplot(x=Label_probabability_DTFrame.Incorrect_Predictions,y=Label_probabability_DTFrame.Probability,
showfliers=False)
plt.xlabel('Incorrect_Predictions',fontsize=15)
plt.ylabel('Probability',fontsize=12)
plt.title('Incorrect_Predictions VS the probability',fontsize=12)
```

Out[299]: Text(0.5,1,'Incorrect_Predictions VS the probability')



SVM Poly

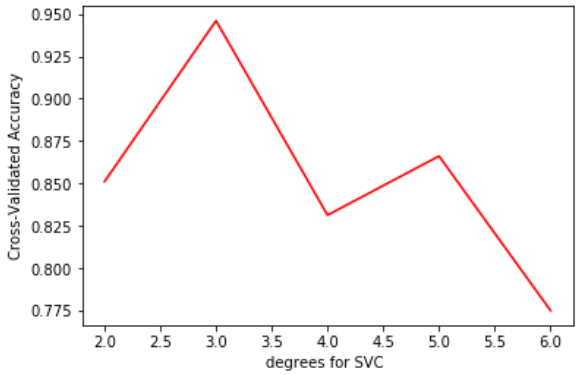
In [300]:
```
from sklearn.cross_validation import cross_val_score
from sklearn.svm import SVC
from sklearn.preprocessing import StandardScaler
X=StandardScaler().fit_transform(df1.iloc[:,:-1])
C_List=pd.Series()
Acc_List=pd.Series()
DTFrame=pd.DataFrame()
polyList=[2,3,4,5,6]
for i in polyList:
    svcCrossRBF = SVC(kernel='poly',degree=i)
    scores=cross_val_score(svcCrossRBF,X,df1.iloc[:,-1].values,scoring='accuracy',cv=10)
    C_List = C_List.append(pd.Series(i))
    Acc_List = Acc_List.append(pd.Series(round(scores.mean(),4)))
DTFrame= pd.concat([C_List,Acc_List],axis=1,keys=['poly_values','Accuracy_values'])
DTFrame['Accuracy_values'] = DTFrame['Accuracy_values'].apply(lambda x : str(x).replace('$',''))
DTFrame
```

Out[300]:

	poly_values	Accuracy_values
0	2	0.8512
0	3	0.946
0	4	0.8314
0	5	0.8662
0	6	0.7752

In [301]:
```
sns.lineplot(x=C_List,y=Acc_List,color='red')
plt.xlabel('degrees for SVC ')
plt.ylabel('Cross-Validated Accuracy')
```

Out[301]: Text(0,0.5,'Cross-Validated Accuracy')



In [302]:
```
from sklearn.model_selection import GridSearchCV,RandomizedSearchCV,train_test_split
from sklearn.svm import SVC
from sklearn.preprocessing import LabelEncoder,StandardScaler
import time
dfnew['label'] = (LabelEncoder().fit_transform(dfnew['label']))
train_set,test_set = train_test_split(dfnew,random_state=9,test_size=0.2,stratify=df['label'])
sc=StandardScaler()
train_set.loc[:,'kurt':'mindom']= sc.fit_transform(train_set.loc[:,'kurt':'mindom'])
#display(test_set.loc[:,'kurt':'sd'])
svc = SVC(probability=True)
start_time=time.time()
params=dict(C=[0.6,1.0],kernel=['poly'],degree=[2,3,4],random_state=[5,10,13,45],class_weight=[{0:1}])
for i in range(3,10):
    grid=RandomizedSearchCV(svc,params,cv=i).fit(train_set.loc[:,'kurt':'mindom'],train_set['label'])
    print('grid best score:  for class weight ',grid.best_score_)
    print('grid best parameters: ',grid.best_params_)
```

```
grid best score:  for class weight  0.9451026856240127
grid best parameters:  {'random_state': 45, 'kernel': 'poly', 'degree': 3, 'class_weight': {0: 1}, 'C': 1.
0}
grid best score:  for class weight  0.9494470774091627
grid best parameters:  {'random_state': 5, 'kernel': 'poly', 'degree': 3, 'class_weight': {0: 1}, 'C': 1.
0}
grid best score:  for class weight  0.9478672985781991
grid best parameters:  {'random_state': 10, 'kernel': 'poly', 'degree': 3, 'class_weight': {0: 1}, 'C': 1.
```

```
0}
grid best score:  for class weight  0.9494470774091627
grid best parameters:  {'random_state': 10, 'kernel': 'poly', 'degree': 3, 'class_weight': {0: 1}, 'C': 1.
0}
grid best score:  for class weight  0.9490521327014217
grid best parameters:  {'random_state': 10, 'kernel': 'poly', 'degree': 3, 'class_weight': {0: 1}, 'C': 1.
0}
grid best score:  for class weight  0.9510268562401264
grid best parameters:  {'random_state': 45, 'kernel': 'poly', 'degree': 3, 'class_weight': {0: 1}, 'C': 1.
0}
grid best score:  for class weight  0.9502369668246445
grid best parameters:  {'random_state': 45, 'kernel': 'poly', 'degree': 3, 'class_weight': {0: 1}, 'C': 1.
0}
```

In [303]:
```
# 5 fold Cross validation produces the best score
```

In [304]:
```
from sklearn.model_selection import GridSearchCV,RandomizedSearchCV,train_test_split
from sklearn.svm import SVC
from sklearn.preprocessing import LabelEncoder,StandardScaler
import time
dfnew['label'] = (LabelEncoder().fit_transform(dfnew['label']))
train_set,test_set = train_test_split(dfnew,random_state=5,test_size=0.2,stratify=df['label'])
sc=StandardScaler()
train_set.loc[:,'kurt':'mindom']= sc.fit_transform(train_set.loc[:,'kurt':'mindom'])
#display(test_set.loc[:,'kurt':'sd'])
svc1 = SVC(probability=True)
start_time=time.time()
params=dict(C=[0.6,1.0],kernel=['poly'],degree=[1,2,3,4],random_state=[5,10,13,45],class_weight=[{0:1}])
grid=RandomizedSearchCV(svc1,params,cv=5).fit(train_set.loc[:,'kurt':'mindom'],train_set['label'])
print('grid best score:  for class weight ',grid.best_score_)
print('grid best parameters: ',grid.best_params_)
print("Execution time: " + str((time.time() - start_time)) + ' ms')
```

```
grid best score:  for class weight  0.9699842022116903
grid best parameters:  {'random_state': 13, 'kernel': 'poly', 'degree': 1, 'class_weight': {0: 1}, 'C': 1.
0}
Execution time: 22.586835384368896 ms
```

In [305]:
```
test_set.loc[:,'kurt':'mindom']=sc.transform(test_set.loc[:,'kurt':'mindom'])
print('Test_accuracy: ',grid.score(test_set.loc[:,'kurt':'mindom'],test_set['label']))
```

```
Test_accuracy:  0.9747634069400631
```

In [306]:
```
from sklearn.metrics import confusion_matrix,precision_recall_fscore_support,f1_score
from sklearn.preprocessing import LabelEncoder
dfnew['label'] = (LabelEncoder().fit_transform(dfnew['label']))
#display(test_set.loc[:,'kurt':'sd'])
svc1.fit(train_set.loc[:,'kurt':'mindom'],train_set['label'])
y_pred = svc1.predict(test_set.loc[:,'kurt':'mindom'])
print('The test dataset confusion_matrix is :\n',confusion_matrix(test_set['label'],y_pred))
```

```
The test dataset confusion_matrix is :
 [[314   3]
 [ 11 306]]
```

In [309]:
```
Label_probabability = pd.Series()
for each in svc1.predict_proba((test_set.loc[:,'kurt':'mindom'])):
    Label_probabability = Label_probabability.append(pd.Series(np.amax(each)))
Label_probabability_DTFrame = pd.concat([Label_probabability.reset_index(drop=True),
                                         test_set['label'].reset_index(drop=True),
                                         pd.Series(svc1.predict(test_set.loc[:,'kurt':'mindom']))],axis=1,
                                        keys=['Probability','Actual','Predicted'])
Label_probabability_DTFrame['SameOrNot']=abs(Label_probabability_DTFrame.Actual - Label_probabability_DTFr
ame.Predicted)
Label_probabability_DTFrame['Incorrect_Predictions']=Label_probabability_DTFrame['SameOrNot'].apply(lambda
x: 'No' if x ==0 else 'Yes')
Label_probabability_DTFrame.head()
```

Out[309]:

	Probability	Actual	Predicted	SameOrNot	Incorrect_Predictions
0	$1.00</td> <td>0</td> <td>0</td> <td>0</td> <td>No</td> </tr> <tr> <th>1</th> <td>$1.00	1	1	0	No
2	$0.99</td> <td>1</td> <td>1</td> <td>0</td> <td>No</td> </tr> <tr> <th>3</th> <td>$1.00	0	0	0	No
4	$0.99	0	0	0	No

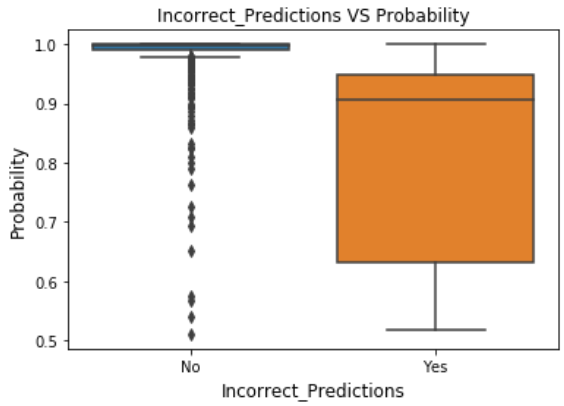
In [312]:
```
Label_probabability_DTFrame[Label_probabability_DTFrame.SameOrNot !=0]
```

Out[312]:

	Probability	Actual	Predicted	SameOrNot	Incorrect_Predictions
73	$1.00	1	0	1	Yes
130	$0.53</td> <td>1</td> <td>0</td> <td>1</td> <td>Yes</td> </tr> <tr> <th>144</th> <td>$0.87	1	0	1	Yes
296	$0.56</td> <td>0</td> <td>1</td> <td>1</td> <td>Yes</td> </tr> <tr> <th>317</th> <td>$1.00	1	0	1	Yes
349	$0.82</td> <td>1</td> <td>0</td> <td>1</td> <td>Yes</td> </tr> <tr> <th>352</th> <td>$0.91	0	1	1	Yes
417	$0.92</td> <td>1</td> <td>0</td> <td>1</td> <td>Yes</td> </tr> <tr> <th>479</th> <td>$0.99	1	0	1	Yes
524	$0.52</td> <td>1</td> <td>0</td> <td>1</td> <td>Yes</td> </tr> <tr> <th>594</th> <td>$0.93	1	0	1	Yes
604	$0.90</td> <td>1</td> <td>0</td> <td>1</td> <td>Yes</td> </tr> <tr> <th>623</th> <td>$0.57	0	1	1	Yes
626	$0.95	1	0	1	Yes

In [314]:
```
sns.boxplot(x=Label_probabability_DTFrame.Incorrect_Predictions,y=Label_probabability_DTFrame.Probability)
plt.xlabel('Incorrect_Predictions',fontsize=12)
plt.ylabel('Probability',fontsize=12)
plt.xlabel('Incorrect_Predictions',fontsize=12)
plt.title('Incorrect_Predictions VS Probability',fontsize=12)
```

Out[314]:
```
Text(0.5,1,'Incorrect_Predictions VS Probability')
```



In [ ]:
```
```

Pokémon™ Introduction

This project is primarily intended for enjoyment and has little use outside of players of the Pokémon™ video games. Perhaps the only exception being individuals seeking to create similar video games or fantasy universes generally.

Pokémon™ is a fantasy universe originally created by Satoshi Tajiri (CEO of Game Freak™) in 1995 and is currently owned by Nintendo®, Game Freak™, and Creatures™. The universe consists of but is not limited to animated films and TV series, playing cards, and video games. In the Pokémon™ universe, humans live side-by-side with creatures called "Pokémon" (being both singular and plural) even though the primary concept in the universe is to collect Pokémon, train them, and force them to fight other Pokémon (not unlike slaves and dog fighting). These Pokémon are classified in a similar manner as our animal/plant/&c kingdoms are. At the very top of the topology, Pokémon are divided into "types," which originally were aimed toward the natural elemental nature of the specific Pokémon (Earth, Water, Air, &c).

Within the Pokémon™ universe, the known types have expanded with continued scientific discovery, and many long-term fans have voiced concern about the seemingly haphazard selection of types, two examples to follow. The Metal type: while the existance of metal-like qualities within a selection of Pokémon could be scientifically meaningful, it does not seem to fit in with the other (more elemental) types, such as Fire and Water. Furthermore, many Pokémon are classified as having a distinct secondary type, which is absurd from a classification standpoint akin to saying an organism belongs to both the Plant and Animal Kingdoms.

In this project, we attempt to predict a Pokémon species' primary type from "scientific" measurements and observations from within the video game version of the Pokémon™ universe. We seek to confirm or refute previous predictiveness claims (and findings) from other members of the fanbase that Pokémon type is nearly impossible to predict accurately from the traditional fighting stats, to improve the accuracy of predictions, and ultimately to make claim for or against the soundness of the current classification topology.

to make claim for or against the soundness of the current classification topology.

Dataset

The data comes from two Pokémon™ datasets available on Kaggle:
"The Complete Pokemon Dataset" (https://www.kaggle.com/rounakbanik/pokemon (https://www.kaggle.com/rounakbanik/pokemon))
"Pokemon for Data Mining and Machine Learning" (https://www.kaggle.com/alopez247/pokemon (https://www.kaggle.com/alopez247/pokemon))
The first of which make a decent attempt at completeness.

These datasets contain baseline measurements and observations about 700+ different Pokémon species; including health, defense, strength, and agility ratings, primary body color, average height and weight, resistance to the effects of other elements, and of course the primary and secondary type classification.

In munging the data, we found that the two datasets were taken from different points in time; one containing data on a set of species that were undiscovered during the time frame of the other dataset as well as having updated information on the known species. For simplicities sake, we truncated the datasets to the least common denominator and used the maximum value for differing measurements. We had significant attribute manipulation for readability and data integrity. We also ended up creating 28 new attributes from categorical data for simplicity of analysis.

Analysis technique

First, we generated multiple custom matrices representing topology breakdowns across various attributes. For example and as reported in the results section, we collected the number of species known within each of the primary types and exhibiting each of the 10 colors as a primary color to help identify correlations, as such generating a matrix of integers. This approach is appropriate (albeit not the most intuitive) as it directly indicates ratios between the color-type subsets to each other and the calculateable totals. For example, as shown in our results section, if a species exhibits primarily a green appearance, it is approximately six times more likely to be a Grass type than the next most plausible, Bug type.

Next, we constructed several Logistic Regression models using various subsets of the available attributes in a OVR-like approach for each type and compared precision, recall, and f1-scores. Logistic Regression is appropriate as the majority of our measurements are quantitative values and by using an OVR-like approach, we are able to address each type class.

Finally, we constructed several SVM models using various subsets of the available attributes. We repeated this for linear kerneling, polynomial kerneling with various degrees, and rbf kerneling with various gamma values. SVN is particularly appropriate because we are attempting to predict an attribute that classifies a record into one of many categories, furthermore as with the applicability of Logistic Regression, most of the available measurements are quantitative values.

Results

For the sake of brevity we only report what we feel to be our most meaningful results.

Collecting the topology breakdown by species primary type and primary color, we obtain the following matrix of species counts:

		Primary Color									
		Green	Red	Blue	White	Brown	Yellow	Purple	Pink	Grey	Black
Primary Type	Grass	41	1	5	3	6	3	0	4	3	0
	Fire	0	27	0	0	12	8	0	0	0	0
	Water	5	9	59	6	4	3	7	8	4	0
	Bug	7	16	4	4	3	12	6	0	8	3
	Normal	1	4	6	9	37	5	5	13	10	3
	Poison	3	0	5	0	2	0	17	0	0	1
	Electric	1	3	6	4	0	16	0	1	3	2
	Ground	4	2	1	0	12	2	2	0	6	1
	Fairy	0	0	1	8	0	0	1	7	0	0
	Fighting	1	2	4	2	6	2	2	0	6	0
	Psychic	6	1	6	6	4	6	7	6	2	3
	Rock	3	2	9	0	11	3	2	1	10	0
	Ghost	0	0	0	1	4	1	8	0	0	9
	Ice	0	2	10	5	3	0	0	1	2	0
	Dragon	4	2	9	2	1	1	3	0	1	1
	Dark	0	4	5	1	2	1	3	0	4	8
	Steel	2	0	4	1	3	1	0	0	10	1
	Flying	1	0	0	0	0	0	2	0	0	0

From these values, we observe that certain primary colors are highly depictive of primary type, such as 41 primarily green Grass type species which is nearly six times greater than the 7 green Bug type species--second greatest type for a green species and the 6 brown Grass type species--second greatest color for a Grass type; while other primary colors and primary types have little to no correlation, 9 black Ghost type species as opposed to the second highest 8 black Dark type species and the second highest 8 purple Ghost type species. We find it specifically telling that species exhibiting black coloring is slightly more likely to be a Ghost type instead of a Dark type, which begs the question of what exactly it means to be a Dark type species...

By training several various logistic regression models, our results support other fanbase members' results, specifically that type prediction is poor

when solely based on fighting statistics:

```
Typical Stats
Is_Grass:       f = 0.000       p = 0.000       r = 0.000
Is_Fire:        f = 0.000       p = 0.000       r = 0.000
Is_Water:       f = 0.000       p = 0.000       r = 0.000
Is_Bug:         f = 0.031       p = 1.000       r = 0.016
Is_Normal:      f = 0.106       p = 0.300       r = 0.065
Is_Poison:      f = 0.000       p = 0.000       r = 0.000
Is_Electric:    f = 0.000       p = 0.000       r = 0.000
Is_Ground:      f = 0.000       p = 0.000       r = 0.000
Is_Fairy:       f = 0.000       p = 0.000       r = 0.000
Is_Fighting:    f = 0.071       p = 0.333       r = 0.040
Is_Psychic:     f = 0.115       p = 0.600       r = 0.064
Is_Rock:        f = 0.000       p = 0.000       r = 0.000
Is_Ghost:       f = 0.000       p = 0.000       r = 0.000
Is_Ice:         f = 0.000       p = 0.000       r = 0.000
Is_Dragon:      f = 0.000       p = 0.000       r = 0.000
Is_Dark:        f = 0.000       p = 0.000       r = 0.000
Is_Steel:       f = 0.214       p = 0.500       r = 0.136
Is_Flying:      f = 0.000       p = 0.000       r = 0.000

Atypical Stats
Is_Grass:       f = 0.000       p = 0.000       r = 0.000
Is_Fire:        f = 0.000       p = 0.000       r = 0.000
Is_Water:       f = 0.000       p = 0.000       r = 0.000
Is_Bug:         f = 0.000       p = 0.000       r = 0.000
Is_Normal:      f = 0.000       p = 0.000       r = 0.000
Is_Poison:      f = 0.000       p = 0.000       r = 0.000
Is_Electric:    f = 0.000       p = 0.000       r = 0.000
Is_Ground:      f = 0.000       p = 0.000       r = 0.000
Is_Fairy:       f = 0.000       p = 0.000       r = 0.000
Is_Fighting:    f = 0.000       p = 0.000       r = 0.000
Is_Psychic:     f = 0.148       p = 0.571       r = 0.085
Is_Rock:        f = 0.000       p = 0.000       r = 0.000
Is_Ghost:       f = 0.000       p = 0.000       r = 0.000
Is_Ice:         f = 0.000       p = 0.000       r = 0.000
Is_Dragon:      f = 0.000       p = 0.000       r = 0.000
Is_Dark:        f = 0.067       p = 0.500       r = 0.036
Is_Steel:       f = 0.000       p = 0.000       r = 0.000
Is_Flying:      f = 0.000       p = 0.000       r = 0.000
```

Notice that between the Typical and Atypical statistics, the best f1 score we were able to obtain was less than 25%. Further notice that the vast majority of f1 scores are 0 (primarily due to no positive predictions); this indicates that there may not be enough data points to predict a type using only these attributes, also that these attributes are not particularly telling of the classified type. We believe the latter because, as a game, each category should be somewhat equivalent in power (much like a game of rock-paper-scissors) at least at the base level. This, however, goes to show that the type system may be a poor classification topology when it comes to these fighting creatures.

By training several more logistic regression models, we find that primary color alone fails to predict any species' type:

```
All Color Boolean Stats
Is_Grass:       f = 0.000       p = 0.000       r = 0.000
Is_Fire:        f = 0.000       p = 0.000       r = 0.000
Is_Water:       f = 0.000       p = 0.000       r = 0.000
Is_Bug:         f = 0.000       p = 0.000       r = 0.000
Is_Normal:      f = 0.000       p = 0.000       r = 0.000
Is_Poison:      f = 0.000       p = 0.000       r = 0.000
Is_Electric:    f = 0.000       p = 0.000       r = 0.000
Is_Ground:      f = 0.000       p = 0.000       r = 0.000
Is_Fairy:       f = 0.000       p = 0.000       r = 0.000
Is_Fighting:    f = 0.000       p = 0.000       r = 0.000
Is_Psychic:     f = 0.000       p = 0.000       r = 0.000
Is_Rock:        f = 0.000       p = 0.000       r = 0.000
Is_Ghost:       f = 0.000       p = 0.000       r = 0.000
Is_Ice:         f = 0.000       p = 0.000       r = 0.000
Is_Dragon:      f = 0.000       p = 0.000       r = 0.000
Is_Dark:        f = 0.000       p = 0.000       r = 0.000
Is_Steel:       f = 0.000       p = 0.000       r = 0.000
Is_Flying:      f = 0.000       p = 0.000       r = 0.000
```

We note that this is likely due to using only the ten boolean color values, where for any given species only one can be true, so there are in reality only ten points in 10-D space. Further, we would be interested to see if this result would differ if RGB or similar color codes were used instead of a strict ten-class field. We leave this for a future study as it would require additional data collection effort.

By training several more logistic regression models, we find that modest predictors can be trained by utilizing strength and weakness values against other types. For clarification, each species has weighted magnitude values for damage given and damage received from a species of another type. For example, an individual Squirtle (a Water type species) has a value of 0.5 when up against any Fire type species, indicating that the Squirtle would take half the normal damage from but deal double the damage to the Fire type species, before considering additional effects. Our reported f1 scores, precision, and accuracy follow:

reported f1 scores, precision, and accuracy follow:

```
All Ag_* attributes
Is_Grass:       f = 0.936       p = 0.880       r = 1.000
Is_Fire:        f = 0.948       p = 0.920       r = 0.979
Is_Water:       f = 0.940       p = 0.918       r = 0.962
Is_Bug:         f = 0.927       p = 0.950       r = 0.905
Is_Normal:      f = 1.000       p = 1.000       r = 1.000
Is_Poison:      f = 0.808       p = 0.875       r = 0.750
Is_Electric:    f = 0.901       p = 0.914       r = 0.889
Is_Ground:      f = 0.825       p = 0.788       r = 0.867
Is_Fairy:       f = 0.944       p = 0.895       r = 1.000
Is_Fighting:    f = 0.815       p = 0.759       r = 0.880
Is_Psychic:     f = 0.939       p = 0.902       r = 0.979
Is_Rock:        f = 0.841       p = 0.787       r = 0.902
Is_Ghost:       f = 0.936       p = 0.917       r = 0.957
Is_Ice:         f = 0.773       p = 0.810       r = 0.739
Is_Dragon:      f = 0.846       p = 0.786       r = 0.917
Is_Dark:        f = 0.947       p = 0.931       r = 0.964
Is_Steel:       f = 0.821       p = 0.941       r = 0.727
Is_Flying:      f = 0.000       p = 0.000       r = 0.000
```

Notice that our models predicted Normal type species with 100% accuracy, but failed to predict any Flying type species. We attribute this disparity to the difference in sample sizes: there are only 3 Flying type species to the 93 Normal type species known in the dataset. Additionally, the type counts are skewed because a species can belong to multiple types, as with nearly any Pokémon belonging to the Dragon type, which is likely to have a secondary type of Flying. We would interested to see how results differ by considering only a subset of types and attributing subset types within the superset (such as with Dragon to Flying), but will leave this for a future project.

Then we find that including primary color and generation (a generalization of when the species was discovered) in addition to the strengths and weakness measurements, the predictiveness of our models generally improve:

```
Ag_* + Colors
Is_Grass:       f = 0.964       p = 0.930       r = 1.000
Is_Fire:        f = 1.000       p = 1.000       r = 1.000
Is_Water:       f = 0.967       p = 0.962       r = 0.971
Is_Bug:         f = 0.927       p = 0.950       r = 0.905
Is_Normal:      f = 1.000       p = 1.000       r = 1.000
Is_Poison:      f = 0.902       p = 1.000       r = 0.821
Is_Electric:    f = 0.901       p = 0.914       r = 0.889
Is_Ground:      f = 0.833       p = 0.833       r = 0.833
Is_Fairy:       f = 0.938       p = 1.000       r = 0.882
Is_Fighting:    f = 0.824       p = 0.808       r = 0.840
Is_Psychic:     f = 0.948       p = 0.920       r = 0.979
Is_Rock:        f = 0.857       p = 0.917       r = 0.805
Is_Ghost:       f = 0.913       p = 0.913       r = 0.913
Is_Ice:         f = 0.773       p = 0.810       r = 0.739
Is_Dragon:      f = 0.868       p = 0.793       r = 0.958
Is_Dark:        f = 0.945       p = 0.963       r = 0.929
Is_Steel:       f = 0.821       p = 0.941       r = 0.727
Is_Flying:      f = 0.000       p = 0.000       r = 0.000

Ag_* + Colors + Generation
Is_Grass:       f = 0.964       p = 0.930       r = 1.000
Is_Fire:        f = 1.000       p = 1.000       r = 1.000
Is_Water:       f = 0.967       p = 0.962       r = 0.971
Is_Bug:         f = 0.934       p = 0.966       r = 0.905
Is_Normal:      f = 1.000       p = 1.000       r = 1.000
Is_Poison:      f = 0.902       p = 1.000       r = 0.821
Is_Electric:    f = 0.901       p = 0.914       r = 0.889
Is_Ground:      f = 0.857       p = 0.818       r = 0.900
Is_Fairy:       f = 0.867       p = 1.000       r = 0.765
Is_Fighting:    f = 0.808       p = 0.778       r = 0.840
Is_Psychic:     f = 0.948       p = 0.920       r = 0.979
Is_Rock:        f = 0.857       p = 0.917       r = 0.805
Is_Ghost:       f = 0.913       p = 0.913       r = 0.913
Is_Ice:         f = 0.773       p = 0.810       r = 0.739
Is_Dragon:      f = 0.868       p = 0.793       r = 0.958
Is_Dark:        f = 0.945       p = 0.963       r = 0.929
Is_Steel:       f = 0.821       p = 0.941       r = 0.727
Is_Flying:      f = 0.800       p = 1.000       r = 0.667
```

Notice that with our previous model the worst f1 score was 0 for Flying types and the average f1 score was 0.842. Notice also that by adding color and generation, our worst f1 score is 0.773 for Ice types, Flying types jump to 0.8, and the average jumps to 0.896. At some point, logistic regression can be prone to overfitting for small datasets, and we wonder if we have started breaching that point. Finally, we admit that by using the intra-type strengths and weaknesses to predict the species type is cheating a little bit because often a species strength or weakness stems directly from their type. In essence, nearly EVERY Water type species is strong against nearly EVERY Fire type species simply because of the natural element. Providing these values to our predictor is not unlike providing to the predictor the target variable directly.

Continuing on, we repeated the same process but training various SVMs and obtained strikingly different results for the fighting stats:

```
Typical Stats
              precision    recall  f1-score   support

         Bug       1.00      1.00      1.00        63
        Dark       0.97      1.00      0.98        28
      Dragon       1.00      1.00      1.00        24
    Electric       1.00      1.00      1.00        36
       Fairy       1.00      0.94      0.97        17
    Fighting       1.00      1.00      1.00        25
        Fire       1.00      0.96      0.98        47
      Flying       1.00      1.00      1.00         3
       Ghost       1.00      1.00      1.00        23
       Grass       1.00      0.97      0.98        66
      Ground       1.00      1.00      1.00        30
         Ice       1.00      1.00      1.00        23
      Normal       1.00      1.00      1.00        93
      Poison       1.00      1.00      1.00        28
     Psychic       0.96      1.00      0.98        47
        Rock       1.00      1.00      1.00        41
       Steel       1.00      0.95      0.98        22
       Water       0.96      0.99      0.98       105

   micro avg       0.99      0.99      0.99       721
   macro avg       0.99      0.99      0.99       721
weighted avg       0.99      0.99      0.99       721

Atypical Stats
              precision    recall  f1-score   support

         Bug       0.92      0.87      0.89        63
        Dark       0.96      0.82      0.88        28
      Dragon       1.00      0.96      0.98        24
    Electric       0.93      0.78      0.85        36
       Fairy       1.00      1.00      1.00        17
    Fighting       0.95      0.84      0.89        25
        Fire       0.88      0.62      0.73        47
      Flying       1.00      0.33      0.50         3
       Ghost       1.00      0.96      0.98        23
       Grass       0.85      0.80      0.83        66
      Ground       0.96      0.80      0.87        30
         Ice       1.00      0.78      0.88        23
      Normal       0.82      0.98      0.89        93
      Poison       0.92      0.79      0.85        28
     Psychic       0.98      0.87      0.92        47
        Rock       1.00      0.98      0.99        41
       Steel       1.00      0.86      0.93        22
       Water       0.69      0.97      0.81       105

   micro avg       0.87      0.87      0.87       721
   macro avg       0.94      0.83      0.87       721
weighted avg       0.89      0.87      0.87       721
```

These results directly contradict results of other fanbase members that type prediction has poor quality when using only fighting stats, specifically with an average f1 score of 0.99. We are astounded at this result and are considering fault in our training of the model, but have been unable to identify any issue other than the small sample size.

We are pleased to report that predicting on color alone yields better results using SVM than it did with our linear regression model, although these averages are nothing to be impressed with:

```
All Colors
              precision    recall  f1-score   support

         Bug       0.19      0.19      0.19        63
        Dark       0.25      0.29      0.27        28
      Dragon       0.00      0.00      0.00        24
    Electric       0.00      0.00      0.00        36
       Fairy       0.00      0.00      0.00        17
    Fighting       0.00      0.00      0.00        25
        Fire       0.36      0.57      0.44        47
      Flying       0.00      0.00      0.00         3
       Ghost       0.00      0.00      0.00        23
       Grass       0.52      0.62      0.57        66
      Ground       0.00      0.00      0.00        30
         Ice       0.00      0.00      0.00        23
      Normal       0.25      0.74      0.38        93
      Poison       0.26      0.61      0.37        28
     Psychic       0.00      0.00      0.00        47
        Rock       0.00      0.00      0.00        41
       Steel       0.00      0.00      0.00        22
```

Water	0.44	0.56	0.49	105
micro avg	0.32	0.32	0.32	721
macro avg	0.13	0.20	0.15	721
weighted avg	0.20	0.32	0.24	721

We note that the difference between our results for SVM and linear regression are likely due to how we implemented only a OVR-like linear regression model instead of actually implementing an OVR linear regression model. In fact, by training a linear SVM with these same attributes, the sklearn package seemed to hang.

Finally, we report that, as with linear regression, training the SVM model on intra-type strengths and weaknesses yield good models (in fact better than linear regression alone) and that by including more attributes improves predictive results, even to the point of overfitting:

All Ag_*

	precision	recall	f1-score	support
Bug	0.98	0.92	0.95	63
Dark	0.96	0.96	0.96	28
Dragon	0.86	1.00	0.92	24
Electric	1.00	0.97	0.99	36
Fairy	1.00	1.00	1.00	17
Fighting	0.96	0.92	0.94	25
Fire	0.98	1.00	0.99	47
Flying	0.00	0.00	0.00	3
Ghost	0.88	1.00	0.94	23
Grass	0.96	1.00	0.98	66
Ground	0.96	0.80	0.87	30
Ice	1.00	0.87	0.93	23
Normal	0.99	0.96	0.97	93
Poison	1.00	0.93	0.96	28
Psychic	1.00	0.94	0.97	47
Rock	0.76	1.00	0.86	41
Steel	0.86	0.86	0.86	22
Water	0.94	0.95	0.95	105
micro avg	0.95	0.95	0.95	721
macro avg	0.89	0.89	0.89	721
weighted avg	0.95	0.95	0.95	721

Ag_* + Colors

	precision	recall	f1-score	support
Bug	0.98	0.95	0.97	63
Dark	0.96	0.96	0.96	28
Dragon	0.86	1.00	0.92	24
Electric	1.00	0.97	0.99	36
Fairy	1.00	1.00	1.00	17
Fighting	0.96	0.92	0.94	25
Fire	0.98	1.00	0.99	47
Flying	0.00	0.00	0.00	3
Ghost	0.88	1.00	0.94	23
Grass	0.96	1.00	0.98	66
Ground	0.96	0.80	0.87	30
Ice	1.00	0.87	0.93	23
Normal	0.99	0.96	0.97	93
Poison	1.00	0.93	0.96	28
Psychic	1.00	0.94	0.97	47
Rock	0.79	1.00	0.88	41
Steel	0.86	0.86	0.86	22
Water	0.94	0.95	0.95	105
micro avg	0.95	0.95	0.95	721
macro avg	0.90	0.90	0.89	721
weighted avg	0.95	0.95	0.95	721

Ag_* + Colors + Generation

	precision	recall	f1-score	support
Bug	0.98	0.97	0.98	63
Dark	0.96	0.96	0.96	28
Dragon	0.92	1.00	0.96	24
Electric	1.00	0.94	0.97	36
Fairy	1.00	1.00	1.00	17
Fighting	0.96	0.92	0.94	25
Fire	1.00	1.00	1.00	47
Flying	1.00	0.67	0.80	3
Ghost	0.92	1.00	0.96	23
Grass	0.97	1.00	0.99	66
Ground	0.96	0.80	0.87	30

```
                Ice       1.00      1.00      1.00        23
             Normal       0.99      0.97      0.98        93
             Poison       1.00      0.96      0.98        28
            Psychic       1.00      0.96      0.98        47
               Rock       0.91      1.00      0.95        41
              Steel       0.85      1.00      0.92        22
              Water       0.98      0.99      0.99       105

          micro avg       0.97      0.97      0.97       721
          macro avg       0.97      0.95      0.96       721
       weighted avg       0.97      0.97      0.97       721

       All Attribs
                     precision    recall  f1-score   support

                Bug       1.00      1.00      1.00        63
               Dark       1.00      1.00      1.00        28
             Dragon       1.00      1.00      1.00        24
           Electric       1.00      1.00      1.00        36
              Fairy       1.00      1.00      1.00        17
           Fighting       1.00      1.00      1.00        25
               Fire       1.00      1.00      1.00        47
             Flying       1.00      1.00      1.00         3
              Ghost       1.00      1.00      1.00        23
              Grass       1.00      1.00      1.00        66
             Ground       1.00      1.00      1.00        30
                Ice       1.00      1.00      1.00        23
             Normal       1.00      1.00      1.00        93
             Poison       1.00      1.00      1.00        28
            Psychic       1.00      1.00      1.00        47
               Rock       1.00      1.00      1.00        41
              Steel       1.00      1.00      1.00        22
              Water       1.00      1.00      1.00       105

          micro avg       1.00      1.00      1.00       721
          macro avg       1.00      1.00      1.00       721
       weighted avg       1.00      1.00      1.00       721
```

In conclusion, we find that predicting Pokémon type from measurements and observations can be accomplished, and can be accomplished well with the correct attributes, but question the legitimacy due to the relative small sample size. We also find that previous work has likely overlooked the use of an SVM (or that we made some error in the training of ours). Finally, while we find that the type classification can be backed by predictiveness, we still dislike the current topology from a zoologic standpoint. Perhaps as more Pokémon species are discovered, the scientists and professors will recognize the limitations of the current system and will revise the topology; until then "Gotta Catch 'em All!"

%%latex \newpage

Pokémon™ Code

Import Data

In [ ]:
```
import matplotlib.pyplot as plt
```

In [2]:
```
import numpy as np
import pandas as pd
pokemon1 = pd.read_csv('data/pokemon.csv')
pokemon2 = pd.read_csv('data/pokemon_alopez247.csv')
print(len(pokemon1.pokedex_number))
print(len(pokemon2.Number))

print(pokemon1.columns)
print(pokemon2.columns)
display(pokemon1.head())
display(pokemon2.head())
```

```
801
721
Index(['abilities', 'against_bug', 'against_dark', 'against_dragon',
       'against_electric', 'against_fairy', 'against_fight', 'against_fire',
       'against_flying', 'against_ghost', 'against_grass', 'against_ground',
       'against_ice', 'against_normal', 'against_poison', 'against_psychic',
       'against_rock', 'against_steel', 'against_water', 'attack',
       'base_egg_steps', 'base_happiness', 'base_total', 'capture_rate',
       'classfication', 'defense', 'experience_growth', 'height_m', 'hp',
```

```
       'japanese_name', 'name', 'percentage_male', 'pokedex_number',
       'sp_attack', 'sp_defense', 'speed', 'type1', 'type2', 'weight_kg',
       'generation', 'is_legendary'],
      dtype='object')
Index(['Number', 'Name', 'Type_1', 'Type_2', 'Total', 'HP', 'Attack',
       'Defense', 'Sp_Atk', 'Sp_Def', 'Speed', 'Generation', 'isLegendary',
       'Color', 'hasGender', 'Pr_Male', 'Egg_Group_1', 'Egg_Group_2',
       'hasMegaEvolution', 'Height_m', 'Weight_kg', 'Catch_Rate',
       'Body_Style'],
      dtype='object')
```

	abilities	against_bug	against_dark	against_dragon	against_electric	against_fairy	against_fight	against_fire	aga
0	['Overgrow', 'Chlorophyll']	1.0	1.0	1.0	0.5	0.5	0.5	2.0	2.0
1	['Overgrow', 'Chlorophyll']	1.0	1.0	1.0	0.5	0.5	0.5	2.0	2.0
2	['Overgrow', 'Chlorophyll']	1.0	1.0	1.0	0.5	0.5	0.5	2.0	2.0
3	['Blaze', 'Solar Power']	0.5	1.0	1.0	1.0	0.5	1.0	0.5	1.0
4	['Blaze', 'Solar Power']	0.5	1.0	1.0	1.0	0.5	1.0	0.5	1.0

5 rows × 41 columns

	Number	Name	Type_1	Type_2	Total	HP	Attack	Defense	Sp_Atk	Sp_Def	...	Color	hasGender	Pr_Male	Egg
0	1	Bulbasaur	Grass	Poison	318	45	49	49	65	65	...	Green	True	0.875	Mon
1	2	Ivysaur	Grass	Poison	405	60	62	63	80	80	...	Green	True	0.875	Mon
2	3	Venusaur	Grass	Poison	525	80	82	83	100	100	...	Green	True	0.875	Mon
3	4	Charmander	Fire	NaN	309	39	52	43	60	50	...	Red	True	0.875	Mon
4	5	Charmeleon	Fire	NaN	405	58	64	58	80	65	...	Red	True	0.875	Mon

5 rows × 23 columns

Data Munging

In [3]:
```
i = 0
while i < 721:
    p1Index = pokemon1.pokedex_number[i]
    p2Index = pokemon2.Number[i]

    if p1Index != i+1 or p2Index != i+1:
        print("Line {}: p1 = {}, p2 = {}".format(i+1,p1Index,p2Index))

    i = i + 1

#No output indicates that pokemon1 and pokemon2 are
#aligned with records in numerical order
```

In [4]:
```
pokemon = pd.merge(pokemon1, pokemon2, left_on='pokedex_number', right_on='Number')

pokemon['Hp'] = pokemon[['hp','HP']].max(axis=1)
pokemon['Atk'] = pokemon[['attack','Attack']].max(axis=1)
pokemon['Def'] = pokemon[['defense','Defense']].max(axis=1)
pokemon['SP_Atk'] = pokemon[['sp_attack','Sp_Atk']].max(axis=1)
pokemon['SP_Def'] = pokemon[['sp_defense','Sp_Def']].max(axis=1)
pokemon['SP'] = pokemon[['speed','Speed']].max(axis=1)
pokemon['Tot'] = pokemon[['base_total','Total']].max(axis=1)

pokemon['Pr_Male'].fillna(-1, inplace=True)

dropCols = ['abilities','classfication','japanese_name','pokedex_number',
            'type1','type2','Type_2','height_m','weight_kg','generation',
            'name','hp','HP','attack','Attack','defense','Defense','hasGender',
```

```
             'sp_attack','Sp_Atk','sp_defense','Sp_Def','speed','Speed',
             'base_total','Total','isLegendary','capture_rate','Body_Style',
             'Egg_Group_1','Egg_Group_2','percentage_male','hasMegaEvolution']
pokemon = pokemon.drop(columns = dropCols)

colDict = {'Hp':'HP', 'SP_Atk':'Sp_Atk', 'SP_Def':'Sp_Def', 'SP':'Speed',
           'against_bug':'Ag_Bug', 'against_dark':'Ag_Dark', 'against_dragon':'Ag_Drag',
           'against_electric':'Ag_Elec', 'against_fairy':'Ag_Fairy', 'against_fight':'Ag_Fight',
           'against_fire':'Ag_Fire', 'against_flying':'Ag_Fly', 'against_ghost':'Ag_Ghost',
           'against_grass':'Ag_Grass', 'against_ground':'Ag_Grou', 'against_ice':'Ag_Ice',
           'against_normal':'Ag_Norm', 'against_poison':'Ag_Pois', 'against_psychic':'Ag_Psy',
           'against_rock':'Ag_Rock', 'against_steel':'Ag_Steel', 'against_water':'Ag_Water',
           'Type_1':'Type','Generation':'Gen','is_legendary':'Is_Legd',
           'base_egg_steps':'Egg_Steps','base_happiness':'Happ',
           'experience_growth':'Exp_Grow'
           }
pokemon = pokemon.rename(index=str, columns=colDict)

pokemon['Is_Green'] = (pokemon.Color == 'Green') * 1
pokemon['Is_Red'] = (pokemon.Color == 'Red') * 1
pokemon['Is_Blue'] = (pokemon.Color == 'Blue') * 1
pokemon['Is_White'] = (pokemon.Color == 'White') * 1
pokemon['Is_Brown'] = (pokemon.Color == 'Brown') * 1
pokemon['Is_Yellow'] = (pokemon.Color == 'Yellow') * 1
pokemon['Is_Purple'] = (pokemon.Color == 'Purple') * 1
pokemon['Is_Pink'] = (pokemon.Color == 'Pink') * 1
pokemon['Is_Grey'] = (pokemon.Color == 'Grey') * 1
pokemon['Is_Black'] = (pokemon.Color == 'Black') * 1

pokemon['Is_Grass'] = (pokemon.Type == 'Grass') * 1
pokemon['Is_Fire'] = (pokemon.Type == 'Fire') * 1
pokemon['Is_Water'] = (pokemon.Type == 'Water') * 1
pokemon['Is_Bug'] = (pokemon.Type == 'Bug') * 1
pokemon['Is_Normal'] = (pokemon.Type == 'Normal') * 1
pokemon['Is_Poison'] = (pokemon.Type == 'Poison') * 1
pokemon['Is_Electric'] = (pokemon.Type == 'Electric') * 1
pokemon['Is_Ground'] = (pokemon.Type == 'Ground') * 1
pokemon['Is_Fairy'] = (pokemon.Type == 'Fairy') * 1
pokemon['Is_Fighting'] = (pokemon.Type == 'Fighting') * 1
pokemon['Is_Psychic'] = (pokemon.Type == 'Psychic') * 1
pokemon['Is_Rock'] = (pokemon.Type == 'Rock') * 1
pokemon['Is_Ghost'] = (pokemon.Type == 'Ghost') * 1
pokemon['Is_Ice'] = (pokemon.Type == 'Ice') * 1
pokemon['Is_Dragon'] = (pokemon.Type == 'Dragon') * 1
pokemon['Is_Dark'] = (pokemon.Type == 'Dark') * 1
pokemon['Is_Steel'] = (pokemon.Type == 'Steel') * 1
pokemon['Is_Flying'] = (pokemon.Type == 'Flying') * 1

#print(len(pokemon))
print(pokemon.columns)
#display(pokemon.head())
```

```
Index(['Ag_Bug', 'Ag_Dark', 'Ag_Drag', 'Ag_Elec', 'Ag_Fairy', 'Ag_Fight',
       'Ag_Fire', 'Ag_Fly', 'Ag_Ghost', 'Ag_Grass', 'Ag_Grou', 'Ag_Ice',
       'Ag_Norm', 'Ag_Pois', 'Ag_Psy', 'Ag_Rock', 'Ag_Steel', 'Ag_Water',
       'Egg_Steps', 'Happ', 'Exp_Grow', 'Is_Legd', 'Number', 'Name', 'Type',
       'Gen', 'Color', 'Pr_Male', 'Height_m', 'Weight_kg', 'Catch_Rate', 'HP',
       'Atk', 'Def', 'Sp_Atk', 'Sp_Def', 'Speed', 'Tot', 'Is_Green', 'Is_Red',
       'Is_Blue', 'Is_White', 'Is_Brown', 'Is_Yellow', 'Is_Purple', 'Is_Pink',
       'Is_Grey', 'Is_Black', 'Is_Grass', 'Is_Fire', 'Is_Water', 'Is_Bug',
       'Is_Normal', 'Is_Poison', 'Is_Electric', 'Is_Ground', 'Is_Fairy',
       'Is_Fighting', 'Is_Psychic', 'Is_Rock', 'Is_Ghost', 'Is_Ice',
       'Is_Dragon', 'Is_Dark', 'Is_Steel', 'Is_Flying'],
      dtype='object')
```

Analysis

In [5]:
```
print(pokemon.Type.unique())
print(pokemon.Color.unique())
```

```
['Grass' 'Fire' 'Water' 'Bug' 'Normal' 'Poison' 'Electric' 'Ground'
 'Fairy' 'Fighting' 'Psychic' 'Rock' 'Ghost' 'Ice' 'Dragon' 'Dark' 'Steel'
 'Flying']
['Green' 'Red' 'Blue' 'White' 'Brown' 'Yellow' 'Purple' 'Pink' 'Grey'
 'Black']
```

In [6]:
```
def getMatchingColor(df, color):
```

```
    return df[df.Color == color]

def getMatchingType(df, expectedType):
    return df[df.Type == expectedType]

def getMatching(df, color, expectedType):
    return df[(df.Color == color) & (df.Type == expectedType)]


matrix = []
allColors = pokemon.Color.unique()
allTypes = pokemon.Type.unique()

for ti in range(len(allTypes)):
    line = []
    for ci in range(len(allColors)):
        line.append(len(getMatching(pokemon, allColors[ci], allTypes[ti])))
    matrix.append(line)
display(allTypes)
display(allColors)
display(matrix)
```

```
array(['Grass', 'Fire', 'Water', 'Bug', 'Normal', 'Poison', 'Electric',
       'Ground', 'Fairy', 'Fighting', 'Psychic', 'Rock', 'Ghost', 'Ice',
       'Dragon', 'Dark', 'Steel', 'Flying'], dtype=object)
```

```
array(['Green', 'Red', 'Blue', 'White', 'Brown', 'Yellow', 'Purple',
       'Pink', 'Grey', 'Black'], dtype=object)
```

```
[[41, 1, 5, 3, 6, 3, 0, 4, 3, 0],
 [0, 27, 0, 0, 12, 8, 0, 0, 0, 0],
 [5, 9, 59, 6, 4, 3, 7, 8, 4, 0],
 [7, 16, 4, 4, 3, 12, 6, 0, 8, 3],
 [1, 4, 6, 9, 37, 5, 5, 13, 10, 3],
 [3, 0, 5, 0, 2, 0, 17, 0, 0, 1],
 [1, 3, 6, 4, 0, 16, 0, 1, 3, 2],
 [4, 2, 1, 0, 12, 2, 2, 0, 6, 1],
 [0, 0, 1, 8, 0, 0, 1, 7, 0, 0],
 [1, 2, 4, 2, 6, 2, 2, 0, 6, 0],
 [6, 1, 6, 6, 4, 6, 7, 6, 2, 3],
 [3, 2, 9, 0, 11, 3, 2, 1, 10, 0],
 [0, 0, 0, 1, 4, 1, 8, 0, 0, 9],
 [0, 2, 10, 5, 3, 0, 0, 1, 2, 0],
 [4, 2, 9, 2, 1, 1, 3, 0, 1, 1],
 [0, 4, 5, 1, 2, 1, 3, 0, 4, 8],
 [2, 0, 4, 1, 3, 1, 0, 0, 10, 1],
 [1, 0, 0, 0, 0, 0, 2, 0, 0, 0]]
```

In [7]:
```
from sklearn.linear_model import LogisticRegression
from sklearn.metrics import precision_recall_fscore_support

def runLogisticRegression(attribList, target):
    import warnings
    from sklearn.exceptions import UndefinedMetricWarning
    warnings.filterwarnings("ignore", category=UndefinedMetricWarning)

    X = pokemon[attribList]
    y = pokemon[target]

    lm = LogisticRegression(solver='lbfgs', max_iter=1000)
    lm.fit(X,y)

    y_pred = lm.predict(X)

    p,r,f,s = precision_recall_fscore_support(y, y_pred, labels=[1])
    print('{}: \tf = {:.3f}\tp = {:.3f}\tr = {:.3f}'.format(target,f[0],p[0],r[0]))

def runLogRegForAllTypes(attribList, name):
    print(name)
    for t in allTypes:
        runLogisticRegression(attribList, 'Is_{}'.format(t))
    print()
```

In [8]:
```
from sklearn import svm
from sklearn.metrics import precision_recall_fscore_support
from sklearn.metrics import classification_report

def runSVM(attribList, target, allLabels=False, name="", kernel='rbf', degree=3, gamma=.1):
    print(name)
```

```
    X = pokemon[attribList]
    y = pokemon[target]

    if kernel == 'linear':
        clf = svm.SVC(kernel='linear')
    elif kernel == 'poly':
        clf = svm.SVC(kernel='poly', degree=degree)
    else:
        clf = svm.SVC(kernel='rbf', gamma=gamma)
    clf.fit(X, y)

    y_pred = clf.predict(X)
    if allLabels:
        print(classification_report(y, y_pred))

    else:
        print(classification_report(y, y_pred, labels=[1]))
```

In [9]:
```
allAgs = ['Ag_Bug', 'Ag_Dark', 'Ag_Drag', 'Ag_Elec', 'Ag_Fairy', 'Ag_Fight',
       'Ag_Fire', 'Ag_Fly', 'Ag_Ghost', 'Ag_Grass', 'Ag_Grou', 'Ag_Ice',
       'Ag_Norm', 'Ag_Pois', 'Ag_Psy', 'Ag_Rock', 'Ag_Steel', 'Ag_Water']

typicalStats = ['HP','Atk','Def','Sp_Atk','Sp_Def','Speed','Tot']

atypicalStats = ['Egg_Steps', 'Happ', 'Exp_Grow', 'Is_Legd',
       'Gen', 'Pr_Male', 'Height_m', 'Weight_kg', 'Catch_Rate']

allCols = ['Is_Green', 'Is_Red', 'Is_Blue', 'Is_White', 'Is_Brown',
           'Is_Yellow', 'Is_Purple', 'Is_Pink','Is_Grey', 'Is_Black']
```

In [10]:
```
runLogRegForAllTypes(allAgs, "All Ag_* attributes")
runLogRegForAllTypes(typicalStats, "Typical Stats")
runLogRegForAllTypes(atypicalStats, "Atypical Stats")
runLogRegForAllTypes(allCols, "All Color Boolean Stats")
runLogRegForAllTypes(allAgs + allCols, "Ag_* + Colors")
runLogRegForAllTypes(allAgs + allCols + ['Gen'], "Ag_* + Colors + Generation")
```

```
All Ag_* attributes
Is_Grass:       f = 0.936       p = 0.880       r = 1.000
Is_Fire:        f = 0.948       p = 0.920       r = 0.979
Is_Water:       f = 0.940       p = 0.918       r = 0.962
Is_Bug:         f = 0.927       p = 0.950       r = 0.905
Is_Normal:      f = 1.000       p = 1.000       r = 1.000
Is_Poison:      f = 0.808       p = 0.875       r = 0.750
Is_Electric:    f = 0.901       p = 0.914       r = 0.889
Is_Ground:      f = 0.825       p = 0.788       r = 0.867
Is_Fairy:       f = 0.944       p = 0.895       r = 1.000
Is_Fighting:    f = 0.815       p = 0.759       r = 0.880
Is_Psychic:     f = 0.939       p = 0.902       r = 0.979
Is_Rock:        f = 0.841       p = 0.787       r = 0.902
Is_Ghost:       f = 0.936       p = 0.917       r = 0.957
Is_Ice:         f = 0.773       p = 0.810       r = 0.739
Is_Dragon:      f = 0.846       p = 0.786       r = 0.917
Is_Dark:        f = 0.947       p = 0.931       r = 0.964
Is_Steel:       f = 0.821       p = 0.941       r = 0.727
Is_Flying:      f = 0.000       p = 0.000       r = 0.000

Typical Stats
Is_Grass:       f = 0.000       p = 0.000       r = 0.000
Is_Fire:        f = 0.000       p = 0.000       r = 0.000
Is_Water:       f = 0.000       p = 0.000       r = 0.000
Is_Bug:         f = 0.031       p = 1.000       r = 0.016
Is_Normal:      f = 0.106       p = 0.300       r = 0.065
Is_Poison:      f = 0.000       p = 0.000       r = 0.000
Is_Electric:    f = 0.000       p = 0.000       r = 0.000
Is_Ground:      f = 0.000       p = 0.000       r = 0.000
Is_Fairy:       f = 0.000       p = 0.000       r = 0.000
Is_Fighting:    f = 0.071       p = 0.333       r = 0.040
Is_Psychic:     f = 0.115       p = 0.600       r = 0.064
Is_Rock:        f = 0.000       p = 0.000       r = 0.000
Is_Ghost:       f = 0.000       p = 0.000       r = 0.000
Is_Ice:         f = 0.000       p = 0.000       r = 0.000
Is_Dragon:      f = 0.000       p = 0.000       r = 0.000
Is_Dark:        f = 0.000       p = 0.000       r = 0.000
Is_Steel:       f = 0.214       p = 0.500       r = 0.136
Is_Flying:      f = 0.000       p = 0.000       r = 0.000

Atypical Stats
```

```
Is_Grass:       f = 0.000       p = 0.000       r = 0.000
Is_Fire:        f = 0.000       p = 0.000       r = 0.000
Is_Water:       f = 0.000       p = 0.000       r = 0.000
Is_Bug:         f = 0.000       p = 0.000       r = 0.000
Is_Normal:      f = 0.000       p = 0.000       r = 0.000
Is_Poison:      f = 0.000       p = 0.000       r = 0.000
Is_Electric:    f = 0.000       p = 0.000       r = 0.000
Is_Ground:      f = 0.000       p = 0.000       r = 0.000
Is_Fairy:       f = 0.000       p = 0.000       r = 0.000
Is_Fighting:    f = 0.000       p = 0.000       r = 0.000
Is_Psychic:     f = 0.148       p = 0.571       r = 0.085
Is_Rock:        f = 0.000       p = 0.000       r = 0.000
Is_Ghost:       f = 0.000       p = 0.000       r = 0.000
Is_Ice:         f = 0.000       p = 0.000       r = 0.000
Is_Dragon:      f = 0.000       p = 0.000       r = 0.000
Is_Dark:        f = 0.067       p = 0.500       r = 0.036
Is_Steel:       f = 0.000       p = 0.000       r = 0.000
Is_Flying:      f = 0.000       p = 0.000       r = 0.000

All Color Boolean Stats
Is_Grass:       f = 0.000       p = 0.000       r = 0.000
Is_Fire:        f = 0.000       p = 0.000       r = 0.000
Is_Water:       f = 0.000       p = 0.000       r = 0.000
Is_Bug:         f = 0.000       p = 0.000       r = 0.000
Is_Normal:      f = 0.000       p = 0.000       r = 0.000
Is_Poison:      f = 0.000       p = 0.000       r = 0.000
Is_Electric:    f = 0.000       p = 0.000       r = 0.000
Is_Ground:      f = 0.000       p = 0.000       r = 0.000
Is_Fairy:       f = 0.000       p = 0.000       r = 0.000
Is_Fighting:    f = 0.000       p = 0.000       r = 0.000
Is_Psychic:     f = 0.000       p = 0.000       r = 0.000
Is_Rock:        f = 0.000       p = 0.000       r = 0.000
Is_Ghost:       f = 0.000       p = 0.000       r = 0.000
Is_Ice:         f = 0.000       p = 0.000       r = 0.000
Is_Dragon:      f = 0.000       p = 0.000       r = 0.000
Is_Dark:        f = 0.000       p = 0.000       r = 0.000
Is_Steel:       f = 0.000       p = 0.000       r = 0.000
Is_Flying:      f = 0.000       p = 0.000       r = 0.000

Ag_* + Colors
Is_Grass:       f = 0.964       p = 0.930       r = 1.000
Is_Fire:        f = 1.000       p = 1.000       r = 1.000
Is_Water:       f = 0.967       p = 0.962       r = 0.971
Is_Bug:         f = 0.927       p = 0.950       r = 0.905
Is_Normal:      f = 1.000       p = 1.000       r = 1.000
Is_Poison:      f = 0.902       p = 1.000       r = 0.821
Is_Electric:    f = 0.901       p = 0.914       r = 0.889
Is_Ground:      f = 0.833       p = 0.833       r = 0.833
Is_Fairy:       f = 0.938       p = 1.000       r = 0.882
Is_Fighting:    f = 0.824       p = 0.808       r = 0.840
Is_Psychic:     f = 0.948       p = 0.920       r = 0.979
Is_Rock:        f = 0.857       p = 0.917       r = 0.805
Is_Ghost:       f = 0.913       p = 0.913       r = 0.913
Is_Ice:         f = 0.773       p = 0.810       r = 0.739
Is_Dragon:      f = 0.868       p = 0.793       r = 0.958
Is_Dark:        f = 0.945       p = 0.963       r = 0.929
Is_Steel:       f = 0.821       p = 0.941       r = 0.727
Is_Flying:      f = 0.000       p = 0.000       r = 0.000

Ag_* + Colors + Generation
Is_Grass:       f = 0.964       p = 0.930       r = 1.000
Is_Fire:        f = 1.000       p = 1.000       r = 1.000
Is_Water:       f = 0.967       p = 0.962       r = 0.971
Is_Bug:         f = 0.934       p = 0.966       r = 0.905
Is_Normal:      f = 1.000       p = 1.000       r = 1.000
Is_Poison:      f = 0.902       p = 1.000       r = 0.821
Is_Electric:    f = 0.901       p = 0.914       r = 0.889
Is_Ground:      f = 0.857       p = 0.818       r = 0.900
Is_Fairy:       f = 0.867       p = 1.000       r = 0.765
Is_Fighting:    f = 0.808       p = 0.778       r = 0.840
Is_Psychic:     f = 0.948       p = 0.920       r = 0.979
Is_Rock:        f = 0.857       p = 0.917       r = 0.805
Is_Ghost:       f = 0.913       p = 0.913       r = 0.913
Is_Ice:         f = 0.773       p = 0.810       r = 0.739
Is_Dragon:      f = 0.868       p = 0.793       r = 0.958
Is_Dark:        f = 0.945       p = 0.963       r = 0.929
Is_Steel:       f = 0.821       p = 0.941       r = 0.727
Is_Flying:      f = 0.800       p = 1.000       r = 0.667
```

In [11]:
```
runSVM(allAgs, 'Type', True, "All Ag_*")
runSVM(typicalStats, 'Type', True, "Typical Stats")
runSVM(atypicalStats, 'Type', True, "Atypical Stats")
runSVM(allCols, 'Type', True, "All Colors")
runSVM(allAgs + allCols, 'Type', True, "Ag_* + Colors")
runSVM(allAgs + allCols + ['Gen'], 'Type', True, "Ag_* + Colors + Generation")

runSVM(allAgs + typicalStats + atypicalStats + allCols + ['Gen'],
       'Type', True, "All Attribs")
```

```
All Ag_*
              precision    recall  f1-score   support

         Bug       0.98      0.92      0.95        63
        Dark       0.96      0.96      0.96        28
      Dragon       0.86      1.00      0.92        24
    Electric       1.00      0.97      0.99        36
       Fairy       1.00      1.00      1.00        17
    Fighting       0.96      0.92      0.94        25
        Fire       0.98      1.00      0.99        47
      Flying       0.00      0.00      0.00         3
       Ghost       0.88      1.00      0.94        23
       Grass       0.96      1.00      0.98        66
      Ground       0.96      0.80      0.87        30
         Ice       1.00      0.87      0.93        23
      Normal       0.99      0.96      0.97        93
      Poison       1.00      0.93      0.96        28
     Psychic       1.00      0.94      0.97        47
        Rock       0.76      1.00      0.86        41
       Steel       0.86      0.86      0.86        22
       Water       0.94      0.95      0.95       105

   micro avg       0.95      0.95      0.95       721
   macro avg       0.89      0.89      0.89       721
weighted avg       0.95      0.95      0.95       721

Typical Stats
              precision    recall  f1-score   support

         Bug       1.00      1.00      1.00        63
        Dark       0.97      1.00      0.98        28
      Dragon       1.00      1.00      1.00        24
    Electric       1.00      1.00      1.00        36
       Fairy       1.00      0.94      0.97        17
    Fighting       1.00      1.00      1.00        25
        Fire       1.00      0.96      0.98        47
      Flying       1.00      1.00      1.00         3
       Ghost       1.00      1.00      1.00        23
       Grass       1.00      0.97      0.98        66
      Ground       1.00      1.00      1.00        30
         Ice       1.00      1.00      1.00        23
      Normal       1.00      1.00      1.00        93
      Poison       1.00      1.00      1.00        28
     Psychic       0.96      1.00      0.98        47
        Rock       1.00      1.00      1.00        41
       Steel       1.00      0.95      0.98        22
       Water       0.96      0.99      0.98       105

   micro avg       0.99      0.99      0.99       721
   macro avg       0.99      0.99      0.99       721
weighted avg       0.99      0.99      0.99       721

Atypical Stats
              precision    recall  f1-score   support

         Bug       0.92      0.87      0.89        63
        Dark       0.96      0.82      0.88        28
      Dragon       1.00      0.96      0.98        24
    Electric       0.93      0.78      0.85        36
       Fairy       1.00      1.00      1.00        17
    Fighting       0.95      0.84      0.89        25
        Fire       0.88      0.62      0.73        47
      Flying       1.00      0.33      0.50         3
       Ghost       1.00      0.96      0.98        23
       Grass       0.85      0.80      0.83        66
      Ground       0.96      0.80      0.87        30
         Ice       1.00      0.78      0.88        23
      Normal       0.82      0.98      0.89        93
```

```
          Poison       0.92      0.79      0.85        28
         Psychic       0.98      0.87      0.92        47
            Rock       1.00      0.98      0.99        41
           Steel       1.00      0.86      0.93        22
           Water       0.69      0.97      0.81       105

       micro avg       0.87      0.87      0.87       721
       macro avg       0.94      0.83      0.87       721
    weighted avg       0.89      0.87      0.87       721

    All Colors
                  precision    recall  f1-score   support

             Bug       0.19      0.19      0.19        63
            Dark       0.25      0.29      0.27        28
          Dragon       0.00      0.00      0.00        24
        Electric       0.00      0.00      0.00        36
           Fairy       0.00      0.00      0.00        17
        Fighting       0.00      0.00      0.00        25
            Fire       0.36      0.57      0.44        47
          Flying       0.00      0.00      0.00         3
           Ghost       0.00      0.00      0.00        23
           Grass       0.52      0.62      0.57        66
          Ground       0.00      0.00      0.00        30
             Ice       0.00      0.00      0.00        23
          Normal       0.25      0.74      0.38        93
          Poison       0.26      0.61      0.37        28
         Psychic       0.00      0.00      0.00        47
            Rock       0.00      0.00      0.00        41
           Steel       0.00      0.00      0.00        22
           Water       0.44      0.56      0.49       105

       micro avg       0.32      0.32      0.32       721
       macro avg       0.13      0.20      0.15       721
    weighted avg       0.20      0.32      0.24       721

    Ag_* + Colors
                  precision    recall  f1-score   support

             Bug       0.98      0.95      0.97        63
            Dark       0.96      0.96      0.96        28
          Dragon       0.86      1.00      0.92        24
        Electric       1.00      0.97      0.99        36
           Fairy       1.00      1.00      1.00        17
        Fighting       0.96      0.92      0.94        25
            Fire       0.98      1.00      0.99        47
          Flying       0.00      0.00      0.00         3
           Ghost       0.88      1.00      0.94        23
           Grass       0.96      1.00      0.98        66
          Ground       0.96      0.80      0.87        30
             Ice       1.00      0.87      0.93        23
          Normal       0.99      0.96      0.97        93
          Poison       1.00      0.93      0.96        28
         Psychic       1.00      0.94      0.97        47
            Rock       0.79      1.00      0.88        41
           Steel       0.86      0.86      0.86        22
           Water       0.94      0.95      0.95       105

       micro avg       0.95      0.95      0.95       721
       macro avg       0.90      0.90      0.89       721
    weighted avg       0.95      0.95      0.95       721

    Ag_* + Colors + Generation
                  precision    recall  f1-score   support

             Bug       0.98      0.97      0.98        63
            Dark       0.96      0.96      0.96        28
          Dragon       0.92      1.00      0.96        24
        Electric       1.00      0.94      0.97        36
           Fairy       1.00      1.00      1.00        17
        Fighting       0.96      0.92      0.94        25
            Fire       1.00      1.00      1.00        47
          Flying       1.00      0.67      0.80         3
           Ghost       0.92      1.00      0.96        23
           Grass       0.97      1.00      0.99        66
          Ground       0.96      0.80      0.87        30
             Ice       1.00      1.00      1.00        23
          Normal       0.99      0.97      0.98        93
          Poison       1.00      0.96      0.98        28
```

```
              Psychic       1.00      0.96      0.98        47
                 Rock       0.91      1.00      0.95        41
                Steel       0.85      1.00      0.92        22
                Water       0.98      0.99      0.99       105

            micro avg       0.97      0.97      0.97       721
            macro avg       0.97      0.95      0.96       721
         weighted avg       0.97      0.97      0.97       721

         All Attribs
                       precision    recall  f1-score   support

                  Bug       1.00      1.00      1.00        63
                 Dark       1.00      1.00      1.00        28
               Dragon       1.00      1.00      1.00        24
             Electric       1.00      1.00      1.00        36
                Fairy       1.00      1.00      1.00        17
             Fighting       1.00      1.00      1.00        25
                 Fire       1.00      1.00      1.00        47
               Flying       1.00      1.00      1.00         3
                Ghost       1.00      1.00      1.00        23
                Grass       1.00      1.00      1.00        66
               Ground       1.00      1.00      1.00        30
                  Ice       1.00      1.00      1.00        23
               Normal       1.00      1.00      1.00        93
               Poison       1.00      1.00      1.00        28
              Psychic       1.00      1.00      1.00        47
                 Rock       1.00      1.00      1.00        41
```