[redacted]

Question 2

The steel type offers the best balance between overall stats and variablility of second types, so it is the best option in a monotype game

Choosing a team of the strongest battlers and defenders for battles and raids is a complex question, the answer to which depends on the nuances of the Pokémon being considered. Some of those nuances include:

- What are their underlying base stats of the Pokémon?
- Which stats are most important for battling? And which are most important for defending?
- What are the moves that the Pokémon has access to?
- What is the type of the Pokémon? Does it have a secondary typing?
- Does the Pokémon have any special abilities that limit or improve its performance?
- What are the answers to all of the above questions for the Pokémon yours is most weak to?
- What are the answers to all of the above questions for the Pokémon yours is strongest against?

Adding the constraint that the team needs to be monotype introduces a significant layer of complexity because the types were made such that each one is both weak to and strong against multiple other types. Because we are not given information on move access, the most essential considerations are overall stats and the variety of secondary typing available within a certain type category. I will show why the Steel type slightly edges out the Dragon type for the best option.

In [253]:
```
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
```

In [3]:
```
data = pd.read_csv('../Downloads/pokemon_data_science.csv')
```

In [11]:
```
data[['Total','HP','Attack','Defense','Sp_Atk','Sp_Def','Speed']].describe
()
```

Out[11]:

	Total	HP	Attack	Defense	Sp_Atk	Sp_Def	Speed
count	721.000000	721.000000	721.000000	721.000000	721.000000	721.000000	721.000000
mean	417.945908	68.380028	75.013870	70.808599	68.737864	69.291262	65.714286
std	109.663671	25.848272	28.984475	29.296558	28.788005	27.015860	27.277920
min	180.000000	1.000000	5.000000	5.000000	10.000000	20.000000	5.000000
25%	320.000000	50.000000	53.000000	50.000000	45.000000	50.000000	45.000000
50%	424.000000	65.000000	74.000000	65.000000	65.000000	65.000000	65.000000
75%	499.000000	80.000000	95.000000	85.000000	90.000000	85.000000	85.000000

[redacted]

[redacted]

```
max    720.000000    255.000000    165.000000    230.000000    154.000000    230.000000    160.000000
```

In [40]:
```
all_types = data['Type_1'].unique().tolist()
```

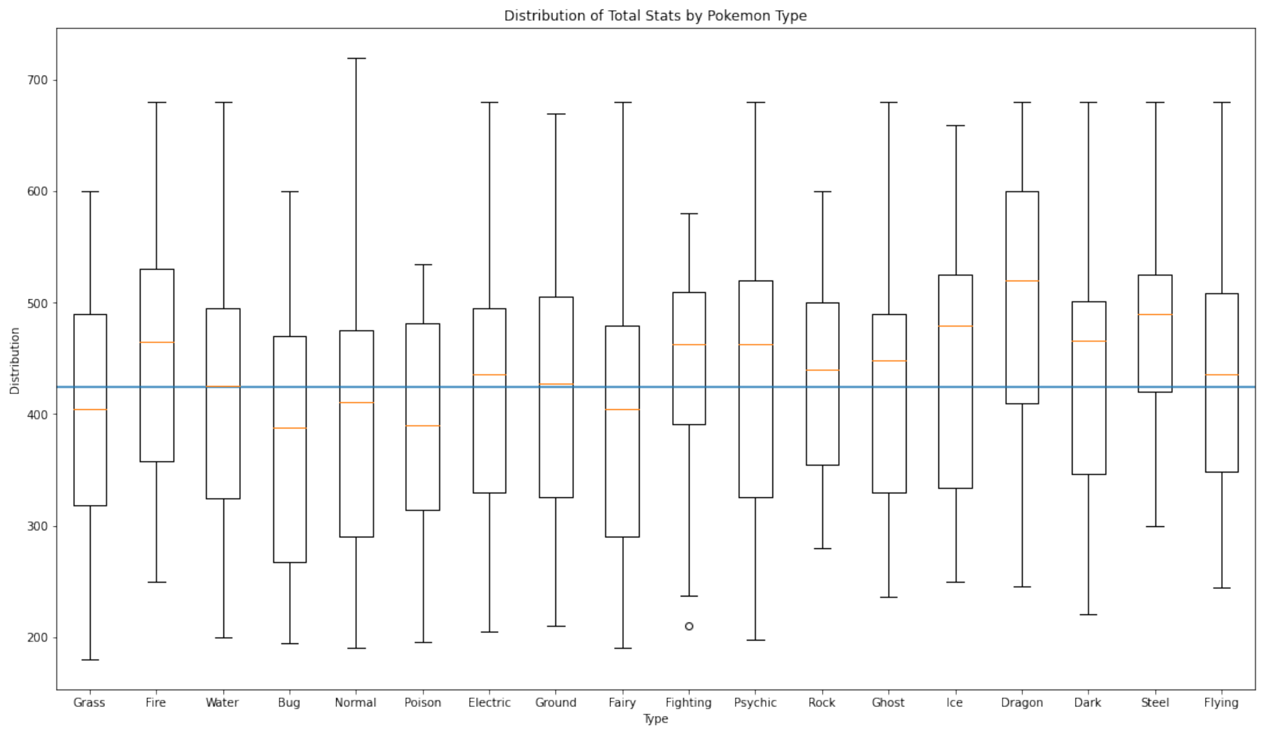
In [329]:
```
# we need to completely identify all Pokemon of a particular type, regardl
ess of whether it is their primary or secondary type
data_enriched = pd.concat([data,pd.get_dummies(data['Type_1'])+pd.get_dumm
ies(data['Type_2'])],axis=1).drop(['Type_1','Type_2'],axis=1)
```

In [330]:
```
# extract all the distributions of Total Stats for each type
type_total_distributions = {i: data_enriched[data_enriched[i]==1]['Total']
.tolist() for i in all_types}
```

In [331]:
```
overall_median = data_enriched['Total'].median()
type_medians = {i: data_enriched[data_enriched[i]==1]['Total'].median() fo
r i in all_types}
```

In [332]:
```
fig, ax = plt.subplots()
fig.set_size_inches(18.5, 10.5)
ax.boxplot(type_total_distributions.values())
ax.set_xticklabels(type_total_distributions.keys())
ax.axhline(overall_median)
plt.xlabel("Type")
plt.ylabel("Distribution")
plt.title("Distribution of Total Stats by Pokemon Type")
```

Out[332]:
```
Text(0.5, 1.0, 'Distribution of Total Stats by Pokemon Type')
```



In [333]:
```
ranking = {i: data_enriched[data_enriched[i]==1]['Total'].median()-overall
_median for i in all_types}
```

[redacted]

[redacted]

```
In [334]: # taking top 8
          sorted(ranking.items(),key=lambda item: item[1])[-8:]

Out[334]: [('Ghost', 24.0),
           ('Fighting', 38.5),
           ('Psychic', 39.0),
           ('Fire', 41.0),
           ('Dark', 42.5),
           ('Ice', 56.0),
           ('Steel', 66.0),
           ('Dragon', 96.5)]
```

From the plot and the median difference calculation, we can see that the median value for certain types is significantly better than the overall median. The following types offer a significant advantage compared to the others in way of Total stats:

1. Dragon
2. Steel
3. Ice
4. Dark
5. Fire
6. Psychic
7. Fighting
8. Ghost

With these types ranked based on Total Stats, we will now examine the number of unique dual type combinations that each of these categories offers

```
In [335]: best_types = ['Dragon','Steel','Ice','Dark','Fire','Psychic','Fighting','G
          host']
          num_additional_types = [sum(data_enriched[data_enriched[i]==1][all_types].
          sum()>0)-1 for i in best_types]

In [336]: sorted(dict(zip(best_types,num_additional_types)).items(),key=lambda item:
           item[1])

Out[336]: [('Ice', 9),
           ('Fighting', 10),
           ('Dragon', 12),
           ('Fire', 12),
           ('Dark', 13),
           ('Ghost', 13),
           ('Steel', 14),
           ('Psychic', 14)]
```

Using the number of additional types available for each single type, we have the following ranking:

1. Psychic / Steel
2. Ghost / Dark
3. Fire / Dragon
4. Fighting

[redacted]

[redacted]

5. Ice

Thus, it looks like it is a **close match between Dragon and Steel types**, but since Steel resists Dragon moves, **I would pick Steel as the best monotype option**

Question 3

If tasked with this problem, I would take the approach of trying multiple models and observing the relative performance. I would want to try models of different types, such as:

- linear models (Logistic Regression, Ridge, Lasso)
  - PROs:
    - Simple and easy to understand/interpret
    - Works well with linear data
  - CONs:
    - Biased if the data is non-linear
- tree-based models (Random Forest)
  - PROs:
    - handle categorical variables very well
    - works well on imbalanced data
    - don't have to worry about overfitting as much
    - can easily extract feature importance
  - CONs:
    - Tough to extrapolate the explanatory relationships of the features
- KNN
  - PROs:
    - Simple
    - Doesn't make any assumptions about the data
    - Only requires one hyperparameter
  - CONs:
    - Doesn't handle class imbalance well
    - Sensitive to outliers

I will implement a Ridge Regression for this task

Question 4

Below is my implementation of the model. Here is a summary of the results:

- in-sample accuracy of the final model: **0.9885**
- out-of-sample accuracy of the final model: **0.9869**

[redacted]

[redacted]

- the **features** that I ended up using to achieve this accuracy were:
    - Undiscovered
    - Weight_kg
    - hasGender
    - Mineral
    - Defense
    - Pr_Female
    - Flying
    - Sp_Atk
    - Ditto

We need to clean and encode the data

In [338]:
```
# Using one hot encoding for the categorical features
generations = pd.get_dummies(data_enriched['Generation'])
colors = pd.get_dummies(data_enriched['Color'])
body = pd.get_dummies(data_enriched['Body_Style'])

# have to create one unified dummy for each egg group even if the pokemon
has two
egg_group_1 = pd.get_dummies(data_enriched['Egg_Group_1'])
egg_group_2 = pd.get_dummies(data_enriched['Egg_Group_2'])
egg_group = egg_group_1.drop(['Ditto','Undiscovered'],axis=1)+egg_group_2
egg_group = pd.concat([egg_group,egg_group_1[['Ditto','Undiscovered']]],ax
is=1)

# these are already encoded as Truth values, so we can pass them through i
nt to encode them
data_enriched['hasGender'] = data_enriched['hasGender'].apply(int)
data_enriched['hasMegaEvolution'] = data_enriched['hasMegaEvolution'].appl
y(int)

# There are some missing values here where pokemon do not have genders, so
 will set these to zeros and also create a
# Pr_Female column
data_enriched['Pr_Female'] = 1- data_enriched['Pr_Male']
data_enriched['Pr_Male'].fillna(0,inplace=True)
data_enriched['Pr_Female'].fillna(0,inplace=True)

data_enriched.drop(['Number','Name','Total','Generation','Egg_Group_1','Eg
g_Group_2','Catch_Rate','Body_Style','Color'],axis=1,inplace=True)
data_enriched = pd.concat([data_enriched,generations,colors,egg_group,body
],axis=1)
isLegendary = data_enriched.pop('isLegendary')
isLegendary = isLegendary.apply(int)
```

In [339]:
```
# Scaling the data because we are using ridge regression
data_enriched[['HP','Attack','Defense','Sp_Atk','Sp_Def','Speed','Height_m
','Weight_kg']] = data_enriched[['HP','Attack','Defense','Sp_Atk','Sp_Def'
,'Speed','Height_m','Weight_kg']] / data_enriched[['HP','Attack','Defense'
,'Sp_Atk','Sp_Def','Speed','Height_m','Weight_kg']].max()
```

In [275]:
```
# there is significant class imbalance; 94% are not legendary, so predicti
```

[redacted]

[redacted]

```
ng only "not legendary" would give 94% accuracy
# thus, the baseline we will be improving upon is 94%
isLegendary.value_counts()
```

Out[275]:
```
0    675
1     46
Name: isLegendary, dtype: int64
```

Time to split the data and implement the model

In [340]:
```
from sklearn.model_selection import LeaveOneOut
from sklearn.model_selection import train_test_split
from sklearn.linear_model import RidgeClassifier
```

In [389]:
```
X_train, X_test, y_train, y_test = train_test_split(data_enriched, isLegen
dary, test_size=0.25)
model = RidgeClassifier()
model.fit(X_train,y_train)

predictions = model.predict(X_test)
print(np.sum(predictions==y_test) / len(predictions)) # accuracy
```

```
0.988950276243094
```

In [390]:
```
from sklearn.metrics import classification_report
from sklearn.metrics import confusion_matrix

print(confusion_matrix(y_test,predictions))
print(classification_report(y_test,predictions))
```

```
[[169   2]
 [  0  10]]
              precision    recall  f1-score   support

           0       1.00      0.99      0.99       171
           1       0.83      1.00      0.91        10

    accuracy                           0.99       181
   macro avg       0.92      0.99      0.95       181
weighted avg       0.99      0.99      0.99       181
```

The model struggles the most with its predicting false negatives, as it predicted two pokemon as not legendaries that were in fact legendaries (evidenced by the confusion matrix and the relatively low precision)

Below is an estimate of the accuracy of the initial model using Leave-One-Out Cross Validation

In [419]:
```
from sklearn.model_selection import LeaveOneOut
loo = LeaveOneOut()

results = []
for train_index, test_index in loo.split(data_enriched.index):
    X_train, X_test = data_enriched.iloc[train_index], data_enriched.iloc[
```

[redacted]

[redacted]

```
test_index]
    y_train, y_test = isLegendary.iloc[train_index], isLegendary.iloc[test
_index]
    model = RidgeClassifier()
    model.fit(X_train,y_train)

    predictions = model.predict(X_test)
    results.append(np.sum(predictions==y_test) / len(predictions)) # accur
acy
print("accuracy estimate: " + str(sum(results)/ len(results)))
```

```
accuracy estimate: 0.986130374479889
```

For this model, we will simplify it by reducing the feature space. To do this, we will take the coefficients from the model and rank the features using them

In [434]:
```
features = data_enriched.columns.tolist()
coefficients = model.coef_.tolist()
feature_importance = pd.DataFrame({"features": features, "coefficients": c
oefficients[0]})
```

In [438]:
```
feature_importance['abs_coef'] = feature_importance['coefficients'].apply(
abs)
feature_importance.sort_values(by='abs_coef', ascending=False,inplace=True
)
```

In [489]:
```
top_10_features = feature_importance.iloc[:10]['features'].tolist()
```

In [491]:
```
print(top_10_features)
```

```
['Undiscovered', 'Weight_kg', 'hasGender', 'Mineral', 'Defense', 'Pr_Femal
e', 'Flying', 'Sp_Atk', 'Ditto', 'Height_m']
```

We will systematically add features in descending order of importance to find the best balance between accuracy and model complexity

In [482]:
```
accuracies = []
for i in range(1,76):
    top_i_features = feature_importance.iloc[:i]['features'].tolist()
    results = []
    for _ in range(100):
        X_train, X_test, y_train, y_test = train_test_split(data_enriched[
top_i_features], isLegendary, test_size=0.25)
        model = RidgeClassifier()
        model.fit(X_train,y_train)
        predictions = model.predict(X_test)
        results.append(np.sum(predictions==y_test) / len(predictions)) # a
ccuracy
    avg_accuracy = sum(results)/ len(results)
    accuracies.append([i,avg_accuracy])
    print(f"accuracy estimate with top {i} features: " + str(avg_accuracy)
)
```

```
accuracy estimate with top 1 features: 0.9607373271889409
```

[redacted]

[redacted]

```
accuracy estimate with top 2 features: 0.9600000000000009
accuracy estimate with top 3 features: 0.9838248847926271
accuracy estimate with top 4 features: 0.9835023041474658
accuracy estimate with top 5 features: 0.984193548387097
accuracy estimate with top 6 features: 0.9848387096774196
accuracy estimate with top 7 features: 0.9852534562211984
accuracy estimate with top 8 features: 0.9859447004608297
accuracy estimate with top 9 features: 0.98778801843318
accuracy estimate with top 10 features: 0.985529953917051
accuracy estimate with top 11 features: 0.986267281105991
accuracy estimate with top 12 features: 0.9842857142857145
accuracy estimate with top 13 features: 0.9860829493087561
accuracy estimate with top 14 features: 0.9835944700460832
accuracy estimate with top 15 features: 0.9854838709677423
accuracy estimate with top 16 features: 0.9859447004608297
accuracy estimate with top 17 features: 0.9854838709677423
accuracy estimate with top 18 features: 0.9852534562211984
accuracy estimate with top 19 features: 0.9857603686635947
accuracy estimate with top 20 features: 0.9866359447004612
accuracy estimate with top 21 features: 0.9859447004608297
accuracy estimate with top 22 features: 0.9862211981566824
accuracy estimate with top 23 features: 0.9861751152073736
accuracy estimate with top 24 features: 0.985898617511521
accuracy estimate with top 25 features: 0.9867741935483874
accuracy estimate with top 26 features: 0.9867281105990786
accuracy estimate with top 27 features: 0.986451612903226
accuracy estimate with top 28 features: 0.9861290322580648
accuracy estimate with top 29 features: 0.9869585253456223
accuracy estimate with top 30 features: 0.9876497695852536
accuracy estimate with top 31 features: 0.9869124423963137
accuracy estimate with top 32 features: 0.9853456221198159
accuracy estimate with top 33 features: 0.9871889400921661
accuracy estimate with top 34 features: 0.9863133640552998
accuracy estimate with top 35 features: 0.9869585253456223
accuracy estimate with top 36 features: 0.985898617511521
accuracy estimate with top 37 features: 0.9863594470046085
accuracy estimate with top 38 features: 0.9868663594470048
accuracy estimate with top 39 features: 0.9860829493087561
accuracy estimate with top 40 features: 0.9863594470046085
accuracy estimate with top 41 features: 0.9875576036866361
accuracy estimate with top 42 features: 0.9872811059907837
accuracy estimate with top 43 features: 0.9868663594470048
accuracy estimate with top 44 features: 0.9870967741935486
accuracy estimate with top 45 features: 0.9882027649769587
accuracy estimate with top 46 features: 0.9872350230414749
accuracy estimate with top 47 features: 0.9872811059907837
accuracy estimate with top 48 features: 0.9889400921658988
accuracy estimate with top 49 features: 0.9884331797235025
accuracy estimate with top 50 features: 0.98815668202765
accuracy estimate with top 51 features: 0.9870046082949311
accuracy estimate with top 52 features: 0.9883410138248849
accuracy estimate with top 53 features: 0.9860368663594472
accuracy estimate with top 54 features: 0.9881105990783412
accuracy estimate with top 55 features: 0.9889861751152076
accuracy estimate with top 56 features: 0.987972350230415
accuracy estimate with top 57 features: 0.9870506912442398
accuracy estimate with top 58 features: 0.9875115207373274
```

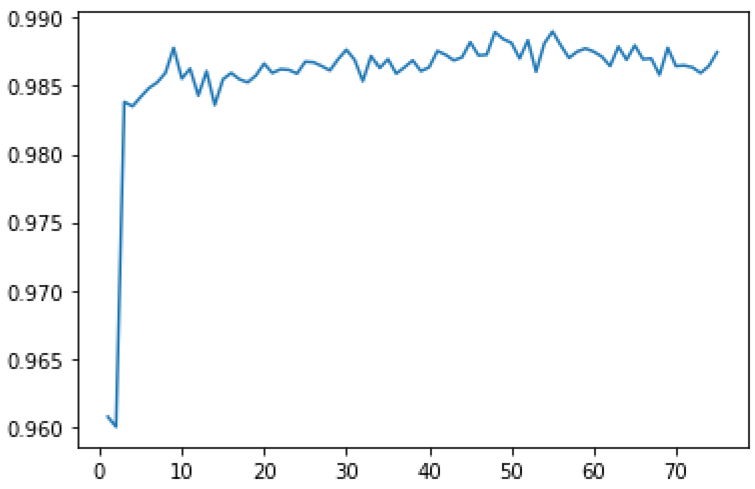
[redacted]

[redacted]

```
accuracy estimate with top 59 features: 0.9877419354838711
accuracy estimate with top 60 features: 0.9875115207373274
accuracy estimate with top 61 features: 0.9871428571428573
accuracy estimate with top 62 features: 0.986451612903226
accuracy estimate with top 63 features: 0.9878801843317975
accuracy estimate with top 64 features: 0.9869124423963136
accuracy estimate with top 65 features: 0.987972350230415
accuracy estimate with top 66 features: 0.9869585253456223
accuracy estimate with top 67 features: 0.9870046082949311
accuracy estimate with top 68 features: 0.9858064516129035
accuracy estimate with top 69 features: 0.98778801843318
accuracy estimate with top 70 features: 0.986451612903226
accuracy estimate with top 71 features: 0.9864976958525348
accuracy estimate with top 72 features: 0.9863594470046085
accuracy estimate with top 73 features: 0.9859447004608297
accuracy estimate with top 74 features: 0.986451612903226
accuracy estimate with top 75 features: 0.9874654377880188
```

In [487]:
```
plt.plot([i[0] for i in accuracies],[i[1] for i in accuracies])
```

Out[487]:
```
[<matplotlib.lines.Line2D at 0x1625c2e8a90>]
```



You can see that the accuracy peaks out at using the top 9 features, so I will remake the model with those features to make it simpler and then re-calculate both the in-sample and out-of-sample accuracy

In [503]:
```
top_9_features = feature_importance.iloc[:9]['features'].tolist()
in_sample_accuracies = []
oos_accuracies = []
for _ in range(100):
    X_train, X_test, y_train, y_test = train_test_split(data_enriched[top_
9_features], isLegendary, test_size=0.25)
    model = RidgeClassifier()
    model.fit(X_train,y_train)
    predictions = model.predict(X_test)
    oos_accuracies.append(np.sum(predictions==y_test) / len(predictions))
# accuracy
    in_sample_predictions = model.predict(X_train)
    in_sample_accuracies.append(np.sum(in_sample_predictions==y_train) / l
en(in_sample_predictions))
```

[redacted]

[redacted]

```
print("in-sample accuracy estimate with top 9 features: " + str(sum(in_sam
ple_accuracies)/len(in_sample_accuracies)))
print("out-of-sample accuracy estimate with top 9 features: " + str(sum(oo
s_accuracies)/len(oos_accuracies)))
```

```
in-sample accuracy estimate with top 9 features: 0.9885000000000001
out-of-sample accuracy estimate with top 9 features: 0.9869060773480662
```

I ended up using the following features:

In [505]:
```
print(top_9_features)
```

```
['Undiscovered', 'Weight_kg', 'hasGender', 'Mineral', 'Defense', 'Pr_Femal
e', 'Flying', 'Sp_Atk', 'Ditto']
```

[redacted]