Solution: Data vs. Wild

Import Libraries and Dataset

In [1]:

```
import pandas as pd
pd.set_option('display.max_columns', 500)

import matplotlib.pyplot as plt
%matplotlib inline

import seaborn as sns
sns.set_style('whitegrid')
```

In [2]:

```
df = pd.read_csv('data/data_vs_wild.csv')
```

Exploratory Analysis

Per the instructions, we won't spend too much time on Exploratory Analysis or data cleaning. However, there are a few things we should still check (our safety is at stake!).

First, we should still get a sense of what's in the dataset:

In [3]:

```
# Example Observations
df.head()
```

Out[3]:

	safe	cap-shape	cap-surface	cap-color	bruises	odor	gill-attachment	gill-spacing	gill-size	gill-color	stalk-shape	stalk-root	stalk-surface-above-ring
0	0	f	f	e	f	s	a	c	n	b	e	?	s
1	0	x	f	y	f	p	a	c	n	u	e	e	k
2	0	b	g	n	t	n	f	c	b	k	e	b	k
3	0	k	y	n	f	f	a	c	n	p	t	?	s
4	0	x	s	b	f	f	f	c	n	b	t	?	s

One thing to note is that `stalk-root` has values that are labeled as `?`. This alone is not a big concern unless it also has other missing values, as "unknown" is still a valid class.

We should also take a glance at summary statistics to get our footing while checking to see if

In [4]:

```
# Summary statistics for numerical features
df.describe()
```

Out[4]:

	safe
count	1000000.000000
mean	0.004159
std	0.064356
min	0.000000
25%	0.000000
50%	0.000000
75%	0.000000
max	1.000000

The only numerical feature we have is `safe` , our target variable. One thing to note is that only 0.4% of observations were labeled as safe. We may run into issues with imbalanced classes. We'll keep this in the back of our mind.

In [5]:

```
# Summary statistics for categorical features
df.describe(include='object')
```

Out[5]:

	cap-shape	cap-surface	cap-color	bruises	odor	gill-attachment	gill-spacing	gill-size	gill-color	stalk-shape
count	1000000	1000000	1000000	1000000	1000000	1000000	1000000	1000000	1000000	1000000
unique	6	4	10	2	9	2	2	2	12	2
top	x	y	n	f	f	f	c	n	b	t
freq	301106	346351	180545	668946	330047	747593	735686	533439	261334	506740

Nothing looks out the ordinary for our categorical features.

Finally, let's check for missing values.

In [6]:

```
# Check for missing values
df.isnull().sum()
```

Out[6]:

```
safe                    0
cap-shape               0
cap-surface             0
```

```
cap-color                   0
bruises                     0
odor                        0
gill-attachment             0
gill-spacing                0
gill-size                   0
gill-color                  0
stalk-shape                 0
stalk-root                  0
stalk-surface-above-ring    0
stalk-surface-below-ring    0
stalk-color-above-ring      0
stalk-color-below-ring      0
veil-type                   0
veil-color                  0
ring-number                 0
ring-type                   0
spore-print-color           0
population                  0
habitat                     0
dtype: int64
```

Machine Learning

Alright, enough dilly-dallying! Let's train that model.

Since we already know that we may run into issues with imbalanced classes, let's do 2 things right off the bat:

1. First, we'll use an **Random Forest** classifier. In general, tree ensembles tend to perform better on datasets with imabalanced classes because their hierarchical structure allows them to learn signals from both classes.
2. Second, we will use **AUROC score** to evaluate our model performance. We should not use traditional accuracy because a model that always predicts "unsafe" will have an accuracy of 99.6%, but it won't be very useful because we'll end up starving.

Tip: In take-home challenges, you typically do not need spend a large amount of time optimizing your model. Just pick a reasonable model (we prefer random forests), explain why (because they work well out-of-the-box, can model non-linear relationships, and are quite robust to outliers, etc.), and make a note of what you would try if given more time (tune hyperparameters, try other algorithms, etc.).

In [7]:

```
from sklearn.ensemble import RandomForestClassifier
from sklearn.metrics import roc_auc_score

from sklearn.model_selection import train_test_split
```

We should create dummy variables for our categorical features using one-hot encoding.

In [8]:

```
# Create
abt = pd.get_dummies( df )
```

Next, we'll split our dataset into training and test sets. One thing to note is that we should **stratify** on our target variable because it's so imbalanced. This will avoid a situation in which we randomly don't have any safe mushrooms in our test set.

In [9]:

```
y = abt.safe
X = abt.drop('safe', axis=1)

X_train, X_test, y_train, y_test = train_test_split(X, y,
                                                    stratify=y,
                                                    random_state=123)
```

Benchmark Model

First, we'll fit a benchmark classifier using the training and test sets. We'll see how well the out-of-box random forest can model this dataset.

In [10]:

```
# Fit benchmark
clf = RandomForestClassifier(random_state=1234)
clf.fit(X_train, y_train)

# Predict
pred = clf.predict_proba(X_test)
pred = [p[1] for p in pred]
```

Let's calculate the AUROC for this benchmark model.

In [11]:

```
# Calculate AUROC
auroc = roc_auc_score(y_test, pred)
print( auroc )
```

```
0.7188476118344622
```

That's actually not a bad start. An AUROC of 0.72 means that if we were randomly give 1 poisonous mushroom and 1 safe mushroom, our model would be able to distinguish them 72% of the time.

Good, but not really good enough. We should look into down-sampling poisonous mushrooms or up-sampling safe mushrooms into the training set.

Tip: We've written about this topic in detail here: How to Handle Imbalanced Classes in Machine Learning

Prepare for Up/Down-Sampling

In this scenario, we should only up/down-sample for the training set, not the test set. The test set should still represent the same proportion of safe/poisonous mushrooms that would be found in the wild.

Let's rebuild the `train` set from the `X_train` and `y_train` objects.

In [12]:

```
# Rebuild train
train = X_train.copy()
train['safe'] = y_train

# Minority and Majority class tables
train_minority = train[train.safe == 1]
train_majority = train[train.safe == 0]
```

We'll also import another module to help us resample.

In [13]:

```
from sklearn.utils import resample
```

Up-Sampling Safe Shrooms

We up-sample the minority class (safe) with replacement to create a balanced training set.

In [14]:

```
# Upsample minority class
minority_upsampled = resample(train_minority,
                              replace=True,
                              n_samples=len(train_majority),
                              random_state=123)

# Combine majority class with upsampled minority class
train_upsampled = pd.concat([train_majority, minority_upsampled])

# Display new class counts
train_upsampled.safe.value_counts()
```

Out[14]:

```
1    746881
0    746881
Name: safe, dtype: int64
```

As you can see, the target variable in our training set is now balanced.

We will create a new `X_train_us` and `y_train_us` from the up-sampled training set.

In [15]:

```
# New X_train and y_train for upsampled (_us)
y_train_us = train_upsampled.safe
X_train_us = train_upsampled.drop('safe', axis=1)
```

Finally, we'll train a classifier using the up-sampled training set and calculate its AUROC.

In [16]:

```
# Train new classifer (upsampled train set)
clf_us = RandomForestClassifier(random_state=1234)
clf_us.fit(X_train_us, y_train_us)

# Predict same test set
pred_us = clf_us.predict_proba(X_test)
pred_us = [p[1] for p in pred_us]
```

In [17]:

```
auroc_us = roc_auc_score(y_test, pred_us)
print( auroc_us )
```

```
0.7550535690008898
```

Up-sampling seemed to bump the AUROC by a little, but the difference is not that big. In fact, it may or may not even be statistically significant.

Let's try down-sampling poisonous mushrooms from the training set instead.

Down-Sampling Poisonous Shrooms

We down-sample the majority class (poisonous) without replacement to create a balanced training set.

In [18]:

```
# Downsample majority class
majority_downsampled= train_majority.sample(n=len(train_minority),
                                            random_state=321)

# Combine minority class with downsampled majority class
train_downsampled = pd.concat([train_minority, majority_downsampled])

# Display new class counts
train_downsampled.safe.value_counts()
```

Out[18]:

```
1    3119
0    3119
```

```
Name: safe, dtype: int64
```

Again, the target variable in our training set is now balanced.

We will create a new `X_train_ds` and `y_train_ds` from the down-sampled training set.

In [19]:
```
# New X_train and y_train for downsampled (_ds)
y_train_ds = train_downsampled.safe
X_train_ds = train_downsampled.drop('safe', axis=1)
```

Finally, we'll train a classifier using the down-sampled training set and calculate its AUROC.

In [20]:
```
# Train new classifer (downsampled train set)
clf_ds = RandomForestClassifier(random_state=1234)
clf_ds.fit(X_train_ds, y_train_ds)

pred_ds = clf_ds.predict_proba(X_test)
pred_ds = [p[1] for p in pred_ds]
```

In [21]:
```
auroc_ds = roc_auc_score(y_test, pred_ds)
print( auroc_ds )
```

```
0.9197458890523038
```

Down-sampling seems to boost the AUROC by much more. It's still not perfect, but we might be able to use this model in a pinch.

Comparing Models

As a last step, let's compare the three models and choose an appropriate decision boundary.

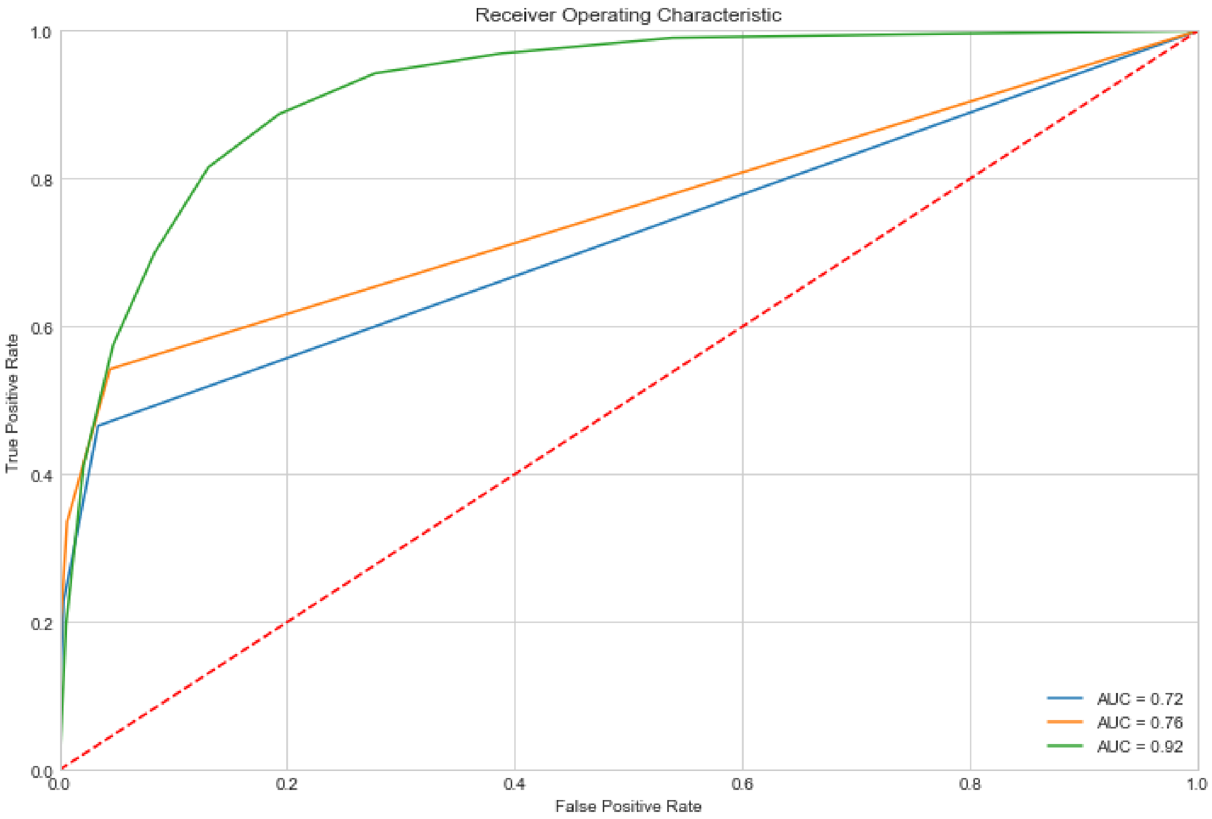
In [22]:
```
from sklearn.metrics import roc_curve
```

First, we need to calculate the false positive and true positive rates.

In [23]:
```
# Benchmark
fpr, tpr, threshold = roc_curve(y_test, pred)

# Up-Sampled Minority Class
fpr_us, tpr_us, threshold = roc_curve(y_test, pred_us)

# Down-Sampled Majority Class
fpr_ds, tpr_ds, threshold = roc_curve(y_test, pred_ds)
```

Next, let's plot the ROC Curve for all three models.

In [24]:
```
plt.figure(figsize=(12,8))
plt.title('Receiver Operating Characteristic')

plt.plot(fpr, tpr, label = 'AUC = %0.2f' % auroc)
plt.plot(fpr_us, tpr_us,label = 'AUC = %0.2f' % auroc_us)
plt.plot(fpr_ds, tpr_ds, label = 'AUC = %0.2f' % auroc_ds)

plt.legend(loc = 'lower right')
plt.plot([0, 1], [0, 1],'r--')
plt.xlim([0, 1])
plt.ylim([0, 1])
plt.ylabel('True Positive Rate')
plt.xlabel('False Positive Rate')
plt.show()
```



As you can, when our threshold for labeling is very high (left side of the chart), all three models perform similary. The false positive rate is low, but so is the true positive rate. In other words, we won't be eating too many poisonous mushrooms, but we also might starve.

On the other hand, if we decrease our threshold a bit, then the classifier trained on the down-sampled dataset really starts to shine. For example, if we can endure a false positive rate of 20%, then we can get our true positive rate up to about 90%. The right threshold will depend on how much poison we can healthily sustain (if we make the simplifying assumption that all poisonous