This notebook explores the use of the bootstrap to create confidence intervals for any statistic of interest that is estimated from data.

In [ ]:
```
import sys
from collections import Counter
from sklearn import preprocessing
from sklearn import linear_model
import pandas as pd
from scipy import sparse
import numpy as np
from math import sqrt
from scipy.stats import norm
from random import choices
from nltk import word_tokenize
```

In [ ]:
```
def read_data(filename):
    X=[]
    Y=[]
    with open(filename, encoding="utf-8") as file:
        for line in file:
            cols=line.rstrip().split("\t")
            label=cols[0]
            text=cols[1]
            # assumes text is tokenized
            X.append(text)
            Y.append(label)
    return X, Y
```

In [ ]:
```
# Change this to the directory with your data (from the CheckData_TODO.ipynb exercise).
# The directory should contain train.tsv, dev.tsv and test.tsv
directory="../data/spooky"
```

In [ ]:
```
trainX, trainY=read_data("%s/train.tsv" % directory)
devX, devY=read_data("%s/dev.tsv" % directory)
```

In [ ]:
```
eab_topics=set(["death", "dead", "regret", "heart", "criminal", "crime", "murder", "jail"])
mws_topics=set(["desire","anguish", "birth", "creature", "Frankenstein", "family", "woman"])
hpl_topics=set(["weird","science", "new", "england", "shadow", "call", "america", "ghost"])
logistic_permutation_test=set(["live", "instead", "though", "off", "landscape", "exquisite", "pestilence", "enveloped", "still

def topical_unigrams(tokens):
    feats={}
    for word in tokens:
        if word in eab_topics:
            feats["word_in_eab_topics"]=1
        if word in mws_topics:
            feats["word_in_mws_topics"]=1
        if word in hpl_topics:
            feats["word_in_hpl_topics"]=1
        if word in logistic_permutation_test:
            feats["words_important_in_permutation_test"]=1
    return feats

def unigrams(tokens):
    feats={}
    for word in tokens:
        feats["UNIGRAM_%s" % word]=1
    return feats
```

In [ ]:
```
def build_features(trainX, feature_functions):
    data=[]
    for doc in trainX:
        feats={}

        # sample text data is already tokenized; if yours is not, do so here
        tokens = word_tokenize(doc.lower())

        for function in feature_functions:
            feats.update(function(tokens))

        data.append(feats)
    return data
```

In [ ]:
```
# This helper function converts a dictionary of feature names to unique numerical ids
def create_vocab(data):
    feature_vocab={}
    idx=0
    for doc in data:
        for feat in doc:
            if feat not in feature_vocab:
                feature_vocab[feat]=idx
                idx+=1

    return feature_vocab
```

In [ ]:
```
# This helper function converts a dictionary of feature names to a sparse representation
# that we can fit in a scikit-learn model.  This is important because almost all feature
# values will be 0 for most documents (note: why?), and we don't want to save them all in
# memory.

def features_to_ids(data, feature_vocab):
    new_data=sparse.lil_matrix((len(data), len(feature_vocab)))
    for idx,doc in enumerate(data):
        for f in doc:
            if f in feature_vocab:
                new_data[idx,feature_vocab[f]]=doc[f]
    return new_data
```

In [ ]:
```
# This function trains a model and returns the predicted and true labels for test data
def evaluate(trainX, devX, trainY, devY, feature_functions):
    trainX_feat=build_features(trainX, feature_functions)
    devX_feat=build_features(devX, feature_functions)

    # just create vocabulary from features in *training* data
    feature_vocab=create_vocab(trainX_feat)

    trainX_ids=features_to_ids(trainX_feat, feature_vocab)
    devX_ids=features_to_ids(devX_feat, feature_vocab)

    le=preprocessing.LabelEncoder()
    le.fit(trainY)

    trainY=le.transform(trainY)
    devY=le.transform(devY)

    print ("Class 1 is %s" % le.inverse_transform([1]))

    logreg = linear_model.LogisticRegression(C=1.0, solver='lbfgs', penalty='l2', max_iter=10000)
    logreg.fit(trainX_ids, trainY)
    print ("Accuracy: %.3f"  % logreg.score(devX_ids, devY))
    predictions=logreg.predict(devX_ids)

    return (predictions, devY)
```

In [ ]:
```
def binomial_confidence_intervals(predictions, truth, confidence_level=0.95):
    correct=[]
    for pred, gold in zip(predictions, truth):
        correct.append(int(pred==gold))

    success_rate=np.mean(correct)

    # two-tailed test
    critical_value=(1-confidence_level)/2
    # ppf finds z such that p(X < z) = critical_value
    z_alpha=-1*norm.ppf(critical_value)

    # the standard error is the square root of the variance/sample size
    # the variance for a binomial test is p*(1-p)
    standard_error=sqrt((success_rate*(1-success_rate))/len(correct))

    lower=success_rate-z_alpha*standard_error
    upper=success_rate+z_alpha*standard_error
    print("%.3f, %s%% Confidence interval: [%.3f,%.3f]" % (success_rate, confidence_level*100, lower, upper))
```

In [ ]:
```
def accuracy(truth, predictions):
    correct=0.
    for idx in range(len(truth)):
        g=truth[idx]
        p=predictions[idx]
        if g == p:
            correct+=1
    return correct/len(truth)
```

In [ ]:
```
def F1(truth, predictions):
    correct=0.
    trials=0.
    trues=0.
    for idx in range(len(truth)):
        g=truth[idx]
        p=predictions[idx]
        if g == p and g == 1:
            correct+=1
        if g == 1:
            trues+=1
        if p == 1:
            trials+=1

    precision=correct/trials if trials > 0 else 0
    recall=correct/trues if trues > 0 else 0
    f=(2*precision*recall)/(precision+recall) if (precision+recall) > 0 else 0
    return f
```

Specify features for model and train logistic regression

In [ ]:
```
features=[topical_unigrams] #, unigrams]
predictions, truth=evaluate(trainX, devX, trainY, devY, features)
```

```
Class 1 is ['HPL']
Accuracy: 0.433
```

First, let's just see what parametric confidence intervals are for accuracy (for which the underlying assumptions of normality are justified by the CLT).

In [ ]:
```
binomial_confidence_intervals(predictions, truth, confidence_level=0.95)
```

```
0.433, 95.0% Confidence interval: [0.408,0.457]
```

Q1: Implement the bootstrap to create confidence intervals at a specified confidence level for any function `metric(truth, predictions)` where *truth* is an array of true labels for a set of data points, and *predictions* is an array of predicted labels for those same points. See `accuracy(truth, predictions)` and `F1(truth, predictions)` above for examples of metrics that should be supported. `bootstrap` should return a tuple of (lower, median, upper), where *lower* is the lower confidence bound, *upper* is the upper confidence bound, and *median* is the median value of the metric among the bootstrap resamples. Hint: see np.percentile.

In [ ]:
```
def bootstrap(gold, predictions, metric, B=10000, confidence_level=0.95):
    #inspired by https://machinelearningmastery.com/calculate-bootstrap-confidence-intervals-machine-learning-results-python/
    bs_statistics = []
    df = pd.DataFrame([predictions, gold], index=['predictions', 'truth']).T
    for i in range(0, B):
        sample = df.sample(frac=1, replace=True)
        stat = metric(np.array(sample['truth']), np.array(sample['predictions']))
        bs_statistics.append(stat)
    bs_ordered = sorted(bs_statistics)
    lower = np.percentile(np.array(bs_ordered), (1-confidence_level)/2)
    median = np.percentile(np.array(bs_ordered), 0.5)
    upper = np.percentile(np.array(bs_ordered), confidence_level+((1-confidence_level)/2))

    return lower, median, upper
```

Q2: Use your bootstrap implementation to generate confidence intervals for accuracy and F1 score.

In [ ]:
```
confidence_level=0.95
lower, median,upper=bootstrap(truth, predictions, accuracy, B=10000,confidence_level=confidence_level)
print("%.3f, %s%% Bootstrap confidence interval: [%.3f, %.3f]" % (median, confidence_level*100, lower, upper))
```

```
0.401, 95.0% Bootstrap confidence interval: [0.389, 0.404]
```

In [ ]:
```
confidence_level=0.95
lower, median,upper=bootstrap(truth, predictions, F1, B=10000,confidence_level=confidence_level)
print("%.3f, %s%% Bootstrap confidence interval: [%.3f, %.3f]" % (median, confidence_level*100, lower, upper))
```

```
0.125, 95.0% Bootstrap confidence interval: [0.107, 0.130]
```