IMPLEMENTATION DETAILS

Since there are several different metrics for comparing the models, each taking two list of values and then returning a single value, I decided to create an abstract interface containing the method calculate, which took two list parameters. Different metric classes would then extend this interface and implement the methods.

Here is the abstract class MetricTest

```
class MetricTest:

    @abstractmethod
    def calculate(self, listA, listB):
        pass
```

Each metric is implemented using this interface:

```
class MeanDiff(MetricTest):

    def calculate(self, listA, listB):
        mean_la = stats.tmean(listA)
        mean_lb = stats.tmean(listB)
        return mean_la - mean_lb
```

```
class WilcoxonTest(MetricTest):

    def calculate(self, listA, listB):
        value, pvalue = wilcoxon(listA, listB)
        return pvalue
```

```
class TTest(MetricTest):

    def calculate(self, listA, listB):
        value, pvalue = ttest_ind(listA, listB)
        return pvalue
```

```
class KSTest(MetricTest):

    def calculate(self, listA, listB):
        value, pvalue = ks_2samp(listA, listB)
        return pvalue
```

To easily specify the comparison metric and which corresponding columns they represented in the data, I created dictionary of Title -> tuple of column indexes. This allowed me to easily loop over this data and write to file.

```
self.significanceMetrics_rouge2 = {'Baseline & Fusion': (0, 1),
                                   'Baseline & Ordering': (0, 2),
                                   'Fusion & Ordering': (1, 2),
                                   'Baseline & Ordering+Fusion': (0, 3),
                                   'Ordering & Ordering+Fusion': (2, 3),
                                   'Fusion & Ordering+Fusion': (1, 3)}
```

```
self.significanceMetrics_rougesu4 = {'Baseline & Fusion': (4, 5),
                                     'Baseline & Ordering': (4, 6),
                                     'Fusion & Ordering': (5, 6),
                                     'Baseline & Ordering+Fusion': (4, 7),
                                     'Ordering & Ordering+Fusion': (6, 7),
                                     'Fusion & Ordering+Fusion': (5, 7)}
```

Using the two dictionaries created above and the MetricTest implementations, I calculated the metrics:

```
def calculateMetrics(self, resultsData):

    md = MeanDiff()
    tt = TTest()
    wx = WilcoxonTest()
    ks = KSTest()

    for i, key in enumerate(self.significanceMetrics_rouge2):
        v = self.significanceMetrics_rouge2[key]
        resultsData[i+1][1] = key
        resultsData[i+1][2] = md.calculate(self.data[v[0]], self.data[v[1]])
        resultsData[i+1][3] = tt.calculate(self.data[v[0]], self.data[v[1]])
        resultsData[i+1][4] = wx.calculate(self.data[v[0]], self.data[v[1]])
        resultsData[i+1][5] = ks.calculate(self.data[v[0]], self.data[v[1]])

    for i, key in enumerate(self.significanceMetrics_rougesu4):
        v = self.significanceMetrics_rougesu4[key]
        resultsData[i+7][1] = key
        resultsData[i+7][2] = md.calculate(self.data[v[0]], self.data[v[1]])
        resultsData[i+7][3] = tt.calculate(self.data[v[0]], self.data[v[1]])
        resultsData[i+7][4] = wx.calculate(self.data[v[0]], self.data[v[1]])
        resultsData[i+7][5] = ks.calculate(self.data[v[0]], self.data[v[1]])
```