HUFFMAN ENCODING:

The first decision our group made about this feature is whether or not to consider each doc number as an atom or each character. We chose characters as atoms for simplicity and because we were unsure about how choosing numbers as atoms would scale.

To facilitate huffman encoding a class was created that stored the information needed to encode and decode for a particular document set. The class also had the methods needed to do the encoding and decoding.

To set up huffman encoding, the index files were stripped of all the terms plus the first separator because the terms were not to be encoded.

A code was then created with createCode.py on the stripped index files to get a proper encoding for the document set.

The result of creating the code was a huffman coder object that had the data and methods needed to encode and decode that document set. The object was pickled and saved into a file called 'huffmanCode'.

The index files were then processed into maps and pickled. Two maps were created for each index, one that had compressed values and the other non-compressed. The maps were actually maps of maps where the first level had a 'type' key saying whether the data was compressed or uncompressed and a 'data' key whose corresponding value was a map containing all the terms with the string that contains their counts information.

The compressed index had to be pickled so that the count string wouldn't create ambiguities when compressed into a binary form. If the terms were not separated from the compressed count string the compressed count string could have been compressed into a newline followed by what looks like a term making one think that the part that is compressed follows the illusionary newline and false-term.

Since the compressed index had to be pickled, the normal one was also pickled for consistency and also so that the pickled map could hold information about whether or not it was compressed.

Example index:
ant:4:(434,1)

ant becomes key with corresponding value: '4:(434,1)' in non-compressed version
ant becomes key with corresponding value: '#$^$' in compressed version

DECODING:

The retrieval model class contains a 'coder' member which is read into the class from the 'huffmanCode' file in the data directory at initialization.

This coder member is stored and sent to the score calculation functions to be used when retrieving term counts. The score calculation functions pass the coder to the getTermCount function which returns the terms based on the index type. The index is also passed to the getTermCount function and its 'type' key is used to identify if it is compressed or not. If it is compressed the compressed count string is decompressed using the coder and the resulting decompressed string is then parsed into a map containing the counts for the term overall and in each document.

If the type of the index is 'uncompressed' the index will have already had its count string parsed into an appropriate map which is returned by getTermCount.

The compression ratio for each count string was generally a little under 2.

For index-0 the size in memory of each map was ~460000 bytes compressed and ~740000 non-compressed (before each non-compressed entry is parsed into a map after which the non-compressed size would be much larger). The original file size was ~570000 bytes.