In [270]:
```
import json
import numpy as np
#import math
#from sklearn.metrics import accuracy_score
```

In [372]:
```
def checkAcc(prediction, y):
    count = 0
    for y_hat, y_val in zip(prediction, y):
        if y_hat == y_val:
            count += 1
    return count/len(y)
```

In [281]:
```
def get_vocabulary(D):
    """
    Given a list of documents, where each document is represented as
    a list of tokens, return the resulting vocabulary. The vocabulary
    should be a set of tokens which appear more than once in the entire
    document collection plus the "<unk>" token.
    """
    # TODO

    vocabulary = set()
    appeared = set()
    for docs in D:
        for words in docs:
            #only if appeared before, we add it to set
            if words in appeared:
                vocabulary.add(words)
            else:
                appeared.add(words)
        vocabulary.add('<unk>')
    return vocabulary
```

In [6]:
```
class BBoWFeaturizer(object):
    def convert_document_to_feature_dictionary(self, doc, vocab):
        """
        Given a document represented as a list of tokens and the vocabulary
        as a set of tokens, compute the binary bag-of-words feature representati
        This function should return a dictionary which maps from the name of the
        feature to the value of that feature.
        """
        # TODO

        res = {}
        for tokens in doc:
            if tokens in vocab:
                res[tokens] = 1
            else:
                res['<unk>'] = 1
        return res
        #raise NotImplementedError
```

In [304]:

```
class CBoWFeaturizer(object):
    def convert_document_to_feature_dictionary(self, doc, vocab):
        """
        Given a document represented as a list of tokens and the vocabulary
        as a set of tokens, compute the count bag-of-words feature representatio
        This function should return a dictionary which maps from the name of the
        feature to the value of that feature.
        """
        # TODO

        res = {}
        for tokens in doc:
            if tokens in vocab:
                if tokens in res:
                    res[tokens] += 1
                else:
                    res[tokens] = 1
            else:
                if '<unk>' in res:
                    res['<unk>'] += 1
                else:
                    res['<unk>'] = 1
        return res
```

In [346]:
```
def compute_idf(D, vocab):
    """
    Given a list of documents D and the vocabulary as a set of tokens,
    where each document is represented as a list of tokens, return the IDF scores
    for every token in the vocab. The IDFs should be represented as a dictionary
    maps from the token to the IDF value. If a token is not present in the
    vocab, it should be mapped to "<unk>".
    """
    res = {}
    for words in vocab:
        if words != '<unk>':
            res[words] = 0
    for docs in D:
        unk_appeared = False
        for tokens in set(docs): # to deduplicate/avoid double counting
            if tokens in vocab:
                res[tokens] += 1
            else:
                if not unk_appeared:
                    if '<unk>' in res:
                        res['<unk>'] += 1
                    else:
                        res['<unk>'] = 1
                    unk_appeared = True
    for keys in res:
        res[keys] = np.log(len(D)/res[keys])
    return res

class TFIDFFeaturizer(object):
    def __init__(self, idf):
        """The idf scores computed via `compute_idf`."""
        self.idf = idf

    def convert_document_to_feature_dictionary(self, doc, vocab):
        """
        Given a document represented as a list of tokens and
        the vocabulary as a set of tokens, compute
        the TF-IDF feature representation. This function
        should return a dictionary which maps from the name of the
        feature to the value of that feature.
        """
        # TODO
        res = {}
        for tokens in doc:
            if tokens in vocab:
                if tokens in res:
                    res[tokens] += 1
                else:
                    res[tokens] = 1
            else:
                if '<unk>' in res:
                    res['<unk>'] += 1
                else:
                    res['<unk>'] = 1

        for keys in res:
```

```
            res[keys] *= self.idf.get(keys)
        return res
```

In [46]:
```
# You should not need to edit this cell
def load_dataset(file_path):
    D = []
    y = []
    with open(file_path, 'r') as f:
        for line in f:
            instance = json.loads(line)
            D.append(instance['document'])
            y.append(instance['label'])
    return D, y

def convert_to_features(D, featurizer, vocab):
    X = []
    for doc in D:
        X.append(featurizer.convert_document_to_feature_dictionary(doc, vocab))
    return X
```

In [156]:
```
def train_naive_bayes(X, y, k, vocab):
    """
    Computes the statistics for the Naive Bayes classifier.
    X is a list of feature representations, where each representation
    is a dictionary that maps from the feature name to the value.
    y is a list of integers that represent the labels.
    k is a float which is the smoothing parameters.
    vocab is the set of vocabulary tokens.

    Returns two values:
        p_y: A dictionary from the label to the corresponding p(y) score
        p_v_y: A nested dictionary where the outer dictionary's key is
            the label and the inner dictionary maps from a feature
            to the probability p(v|y). For example, `p_v_y[1]["hello"]`
            should be p(v="hello"|y=1).
    """
    # p_y
    p_y  = {}
    size = len(vocab)
    p_y[0], p_y[1] = y.count(0)/len(y), y.count(1)/len(y)

    # p_v_y
    p_v_y = {}
    p_v_y[1] = {}
    p_v_y[0] = {}

    totalCount_1 = 0
    totalCount_0 = 0

    for word in vocab:
        p_v_y[1][word] = 0
        p_v_y[0][word] = 0

    for doc, label in zip(X,y):
        for word in doc.keys():
            p_v_y[label][word] += doc[word]
            if label == 1:
                totalCount_1 += doc[word]
            else:
                totalCount_0 += doc[word]

    p_v_y[1] = {key: ((k + value) / (totalCount_1 + k*size)) for key, value in p_
    p_v_y[0] = {key: ((k + value) / (totalCount_0 + k*size)) for key, value in p_

    return p_y, p_v_y
```

In [348]:
```
def predict_naive_bayes(D, p_y, p_v_y):
    """
    Runs the prediction rule for Naive Bayes. D is a list of documents,
    where each document is a list of tokens.
    p_y and p_v_y are output from `train_naive_bayes`.

    Note that any token which is not in p_v_y should be mapped to
    "<unk>". Further, the input dictionaries are probabilities. You
    should convert them to log-probabilities while you compute
    the Naive Bayes prediction rule to prevent underflow errors.

    Returns two values:
        predictions: A list of integer labels, one for each document,
            that is the predicted label for each instance.
        confidences: A list of floats, one for each document, that is
            p(y|d) for the corresponding label that is returned.
    """

    # TODO

    prediction = []
    confidence = [] # P(y|d)
    p_d = []
    p_d_y= []

    vocab = set(p_v_y[0])

    for docs in D:
        scores = []
        for label, prob in p_y.items():
            score = 0
            for word in docs:
                if word in vocab:
                    score += np.log(p_v_y[label][word])
                else:
                    score += np.log(p_v_y[label]['<unk>'])
            scores.append(score + np.log(prob))
        prediction.append(scores.index(max(scores)))
        p_d.append(np.logaddexp(scores[0],scores[1]))
        p_d_y.append(max(scores)) # p(D|y) * P(y)

    confidence = list(np.exp(np.array(p_d_y) - np.array(p_d)))
    return prediction, confidence
```

In [441]:
```
def train_semi_supervised(X_sup, y_sup, D_unsup, X_unsup, D_valid, y_valid, k, v
    """
    Trains the Naive Bayes classifier using the semi-supervised algorithm.

    X_sup: A list of the featurized supervised documents.
    y_sup: A list of the corresponding supervised labels.
    D_unsup: The unsupervised documents.
    X_unsup: The unsupervised document representations.
    D_valid: The validation documents.
    y_valid: The validation labels.
    k: The smoothing parameter for Naive Bayes.
    vocab: The vocabulary as a set of tokens.
    mode: either "threshold" or "top-k", depending on which selection
        algorithm should be used.

    Returns the final p_y and p_v_y (see `train_naive_bayes`) after the
    algorithm terminates.
    """
    # TODO
    threshold = 0.98
    top_K = 10000
    while True:
        p_y, p_v_y = train_naive_bayes(X_sup, y_sup, k, vocab)
        prediction, confidence = predict_naive_bayes(D_unsup, p_y, p_v_y)
        prediction_acc = checkAcc(predict_naive_bayes(D_valid, p_y, p_v_y)[0], y_
        print("Accuracy: ", prediction_acc)
        if mode == 'threshold':
            pass_index = []
            zip_data = list(zip(prediction, confidence))
            for i in range(len(zip_data)):
                if confidence[i] > 0.98:
                    pass_index.append(i)
            X_sup_new = [X_unsup[j] for j in pass_index]
            y_sup_new = [prediction[k] for k in pass_index]
            if len(X_sup_new) == 0:
                return p_y, p_v_y
            X_sup.extend(X_sup_new)
            y_sup.extend(y_sup_new)
            D_unsup = [D_unsup[n] for n in range(len(D_unsup)) if n not in pass_
        if mode == 'top-k':
            if D_unsup == []:
                return p_y,p_v_y
            zip_data = list(zip(D_unsup, X_unsup, prediction, confidence))
            zip_data.sort(key = lambda x: x[3])
            X_sup.extend([item[1] for item in zip_data[-10000:] ])
            y_sup.extend([item[2] for item in zip_data[-10000:] ])
            D_unsup = [item[0] for item in zip_data if item not in [bad for bad 
```

In [458]:
```
# Variables that are named D_* are lists of documents where each
# document is a list of tokens. y_* is a list of integer class labels.
# X_* is a list of the feature dictionaries for each document.
D_train, y_train = load_dataset('data/train.jsonl')
D_valid, y_valid = load_dataset('data/valid.jsonl')
D_test, y_test = load_dataset('data/test.jsonl')

vocab = get_vocabulary(D_train)
```

In [157]:
```
# Compute the features, for example, using the BBowFeaturizer.
# You actually only need to conver the training instances to their
# feature-based representations.
#
# This is just starter code for the experiment. You need to fill in
# the rest.
```

In [364]:
```
##BBow

k_vals = [0.001,0.01,0.1,1.0,10.0]
accs = []

for k in k_vals:
    featurizer = BBoWFeaturizer()
    X_train = convert_to_features(D_train, featurizer, vocab)
    p_y, p_v_y = train_naive_bayes(X_train, y_train, k, vocab)
    prediction, confidence = predict_naive_bayes(D_valid, p_y, p_v_y)
    acc = checkAcc(prediction, y_valid)
    accs.append(acc)

best_k = k_vals[accs.index(max(accs))]
best_acc = max(accs)
print("best_k: ", best_k, "best_acc: ", best_acc)
```

```
best_k:  0.1 best_acc:  0.8668
```

In [366]:
```
##CBow

k_vals = [0.001,0.01,0.1,1.0,10.0]
accs = []

for k in k_vals:
    featurizer = CBoWFeaturizer()
    X_train = convert_to_features(D_train, featurizer, vocab)
    p_y, p_v_y = train_naive_bayes(X_train, y_train, k, vocab)
    prediction, confidence = predict_naive_bayes(D_valid, p_y, p_v_y)
    acc = checkAcc(prediction, y_valid)
    accs.append(acc)

best_k = k_vals[accs.index(max(accs))]
best_acc = max(accs)
print("best_k: ", best_k, "best_acc: ", best_acc)
```

```
best_k:  0.1 best_acc:  0.8676
```

In [369]:
```
##Tfidf

k_vals = [0.001,0.01,0.1,1.0,10.0]
accs = []

for k in k_vals:
    featurizer = TFIDFFeaturizer(compute_idf(D_train, vocab))
    X_train = convert_to_features(D_train, featurizer, vocab)
    p_y, p_v_y = train_naive_bayes(X_train, y_train, k, vocab)
    prediction, confidence = predict_naive_bayes(D_valid, p_y, p_v_y)
    acc = checkAcc(prediction, y_valid)
    accs.append(acc)

best_k = k_vals[accs.index(max(accs))]
best_acc = max(accs)
print("best_k: ", best_k, "best_acc: ", best_acc)
```

```
best_k:  1.0 best_acc:  0.8364
```

In [459]:
```
#we use CBOW since it performs the best in the previous part:

featurizer = CBoWFeaturizer()
X_train = convert_to_features(D_train, featurizer, vocab)
```

In [462]:
```
#semi-supervised - Threshold

sample_index = np.random.randint(0, 45000, 5000)
X_sup = [X_train[i] for i in sample_index]
y_sup = [y_train[i] for i in sample_index]
D_unsup = [D_train[i] for i in range(len(X_train)) if i not in sample_index]
X_unsup = [X_train[i] for i in range(len(X_train)) if i not in sample_index]
k = 0.1

p_y, p_v_y = train_semi_supervised(X_sup, y_sup, D_unsup, X_unsup, D_valid, y_va
checkAcc(predict_naive_bayes(D_test, p_y, p_v_y)[0], y_test)
```

```
Accuracy:  0.824
Accuracy:  0.7888
Accuracy:  0.7864
Accuracy:  0.7864
Accuracy:  0.7864
```

Out[462]: 0.7832

In [466]:
```
#semi-supervised - Top-K

sample_index = np.random.randint(0, 45000, 5000)
X_sup = [X_train[i] for i in sample_index]
y_sup = [y_train[i] for i in sample_index]
D_unsup = [D_train[i] for i in range(len(X_train)) if i not in sample_index]
X_unsup = [X_train[i] for i in range(len(X_train)) if i not in sample_index]
k = 0.1

p_y, p_v_y = train_semi_supervised(X_sup, y_sup, D_unsup, X_unsup, D_valid, y_va
checkAcc(predict_naive_bayes(D_test, p_y, p_v_y)[0], y_test)
```

```
Accuracy:  0.8216
Accuracy:  0.7792
Accuracy:  0.7664
Accuracy:  0.7536
Accuracy:  0.7504
Accuracy:  0.7496
```

Out[466]: 0.7448