Ham/Spam classification for Short Messages

Here is a bare-bones write-up of how I put together a machine-learning model for classifying short messages as ham/spam.

1) I started off by visually inspecting the file for inconsistencies in data. The data seemed pristine, so I did not have to think of strategies to clean the data.
2) Since the data is all text-messages, my first intuition is that spam messages have a higher incidence of certain words like “free”, “offer”, etc. Ham messages may also contain these words, but the incidence may be lesser. By looking at the entire corpus, we may be able to assign weightages to individual words. A bag-of-words approach may suffice. However, certain “groups of words” like “call now”, “exclusive offer” may also have higher incidence in spam messages. Maybe we should do bigram/trigram frequencies too. Let’s see.
3) The next step was to read the data from the file into a DataFrame. The DataFrame has two columns – one for the ham/spam label and one for the content of the message. I transformed “spam” into a 1 and “ham” into a 0.
4) I picked the scikit-learn package because I have used it before.
5) I experimented with a CountVectorizer, TfIdfVectorizer and HashingVectorizer for getting word counts. TfIdfTransform is used to re-weight words by discounting the weights of common words like “a”, “the”, etc. that convey little meaning. I chose HashingVectorizer in order to try the TfIdfTransform. The TfIdfVectorizer is used because I wanted to see if bigram/trigram tokenization would help.
6) I played around with some classifiers like MultinomialNB, BernoulliNB, SGDClassifier and PassiveAggressiveClassifier. I started with the naïve(no pun intended!!) assumption that the features are independent of one another(trigram, bigram?). Hence I tried two NB classifiers. MultinomialNB is famed for its performance on word-count probabilities and is a classic tool in text classification. On the other hand, BernoulliNB is known for performance on smaller datasets (with ~5500 messages, ours is a very small dataset) and predicts based on word occurrence probabilities. We will try both.
7) The SGDClassifier is essentially a linear classifier with the gradient descent error calculated for each sample in the training set, and the curve adjusted accordingly.
8) The passive aggressive classifier does nothing about correctly classified samples, but aggressively updates the model for mis-classified samples (I am worried about over-fitting, but we shall see).

Here is the performance for each combination:

Vectorizer	TfIdfTransform	Classifier	Score
CountVectorizer	No	BernoulliNB	0.9163
CountVectorizer	Yes	BernoulliNB	0.9163
CountVectorizer	No	PassiveAggressive	0.9412
CountVectorizer	Yes	PassiveAggressive	0.9512
CountVectorizer	No	SGDClassifier	0.9314
CountVectorizer	Yes	SGDClassifier	0.9530
CountVectorizer	No	MultinomialNB	0.9502
TfIdfVectorizer	No	BernoulliNB	0.9326
TfIdfVectorizer	Yes	BernoulliNB	0.9326
TfIdfVectorizer	No	PassiveAggressive	0.9402
TfIdfVectorizer	Yes	PassiveAggressive	0.9424
TfIdfVectorizer	No	SGDClassifier	0.9420
TfIdfVectorizer	Yes	SGDClassifier	0.9435
TfIdfVectorizer	No	MutinomialNB	0.9132
HashingVectorizer	No	PassiveAggressive	0.9411
HashingVectorizer	Yes	PassiveAggressive	0.9353
HashingVectorizer	No	SGDClassifier	0.9360
HashingVectorizer	Yes	SGDClassifier	0.9312
HashingVectorizer	No	MultinomialNB	0.9125

For this small dataset, looks like a pipeline with the CountVectorizer as the vectorizer and an SGDClassifier as classifier beats other combinations, with a very healthy 95.3% accuracy!

Very high-level thoughts about an ingestion system for images

1) My initial thought is to write a minimal, per-domain, distributed crawler for the major domains like Instagram, Flickr, Twitter, etc. This is because each domain might have some specific metadata not available on other platforms (for ex., Flickr images may frequently be accompanied by exif information, twitter images may be hash-tagged and so on). I'm envisioning the crawlers to be in per-domain clusters so that machines may be added and removed based on demand. These could be run efficiently on Amazon EC2 instances. We could have these clusters per-geography too, spooling down instances in geographies where the traffic is low and spooling up in geographies where traffic is high (based on the time of the day, possibly).
2) Crawlers fetch URLs and add them into a queue like the Amazon SQS. This helps to de-couple the crawling from the processing,
3) Another cluster of "download servers" would just be feeding off of the URL queue and downloading images and metadata on a continuous basis (Amazon SQS ReceiveMessage API has enough safeguards in place to avoid the same URL being processed by more than one download server).

4) The download servers are per-domain too; they download exif data from Flickr-like sites and hashtag information from Twitter-like sites, for example). They could be hosted on EC2 or similar instances.
5) The download servers add the downloaded data into another "processing queue", which are consumed by image processing servers.
6) Image processing servers may be hosted on a cluster of EC2 instances. These machines strip metadata (engineer new metadata too, possibly) and send the metadata off to the metadata storage system. The metadata storage system then returns a "locator" which acts like a foreign key to the image storage system. During the storage phase, a "thumbnail server" generates and stores thumbnails of the images being stored, in order to avoid processing at query time.
7) The storage could be hosted on Amazon S3 in order to avoid the high acquisition, maintenance and management costs associated with a storage cluster.
8) For simple image metadata storage, I am tending towards Amazon's DynamoDB. Considering our specific application here, given that our data is streaming and our need for building a recommendation engine, Amazon Kinesis-based solutions seem to be a good choice.