Realtime predictive analytics

using scikit-learn & RabbitMQ

Michael Becker

#pycon

@beckerfuffle

Who Is This Guy?

Data guy @ AWeber

@beckerfuffle

beckerfuffle.com

These slides and more @ github.com/mdbecker

Thanks everybody for coming to my talk! My name is Michael Becker, I work for the Data Analysis and Management Ninjas at AWeber. AWeber is an email service provider with over 120 thousand customers. I'm also the founder of the DataPhilly meetup with over 600 members. If you're not already a member, well shame on you!

You can find me online @beckerfuffle on Twitter. My website is beckerfuffle.com. Materials from this talk can be found on my github.

What My Coworkers Think I Do

$$h_{w,b}(x) = g(w^T x + b)$$

$$\hat{\gamma}^{(i)} = y^{(i)}(w^T x + b)$$

$$\hat{\gamma} = \min_{i=1,\ldots,m} \hat{\gamma}^{(i)}$$

$$w^T \left(x^{(i)} - \gamma^{(i)} \frac{w}{||w||} \right) + b = 0$$

Working on the DAMN team at AWeber, I've introduced several predictive algorithms into our application. My co-workers think I'm some kind of math superhero, reading scholarly papers by day and fighting crime by night! The truth is much more sinister.

What I Actually Do

```
from sklearn.svm import SVC
```


Fortunately for me, I don’t have to be a math genius to look like a superhero, I just have to use scikit-learn! While I’ll cover some math in this talk, I’ll mainly be keeping things high-level. This is because I’m not a math genius. If I get any of the math wrong, feel free to call me out on the interwebs.

What I'll Cover

- Scikit-learn overview

This talk will cover a lot of the logistics behind utilizing a trained scikit-learn model in a real-life production environment.

I'll start off my giving a brief overview of supervised machine learning and text processing with scikit-learn.

What I'll Cover

- Scikit-learn overview
- Model Distribution

I'll cover how to distribute your model

What I'll Cover

- Scikit-learn overview
- Model Distribution
- Data flow

I'll discuss how to get new data to your model for prediction.

What I'll Cover

- Scikit-learn overview
- Model Distribution
- Data flow
- RabbitMQ

I'll introduce RabbitMQ, what it is and why you should care.

What I'll Cover

- Scikit-learn overview
- Model Distribution
- Data flow
- RabbitMQ
- Demo

I'll demonstrate how we can put all this together into a finished product

What I'll Cover

- Scikit-learn overview
- Model Distribution
- Data flow
- RabbitMQ
- Demo
- Scalability

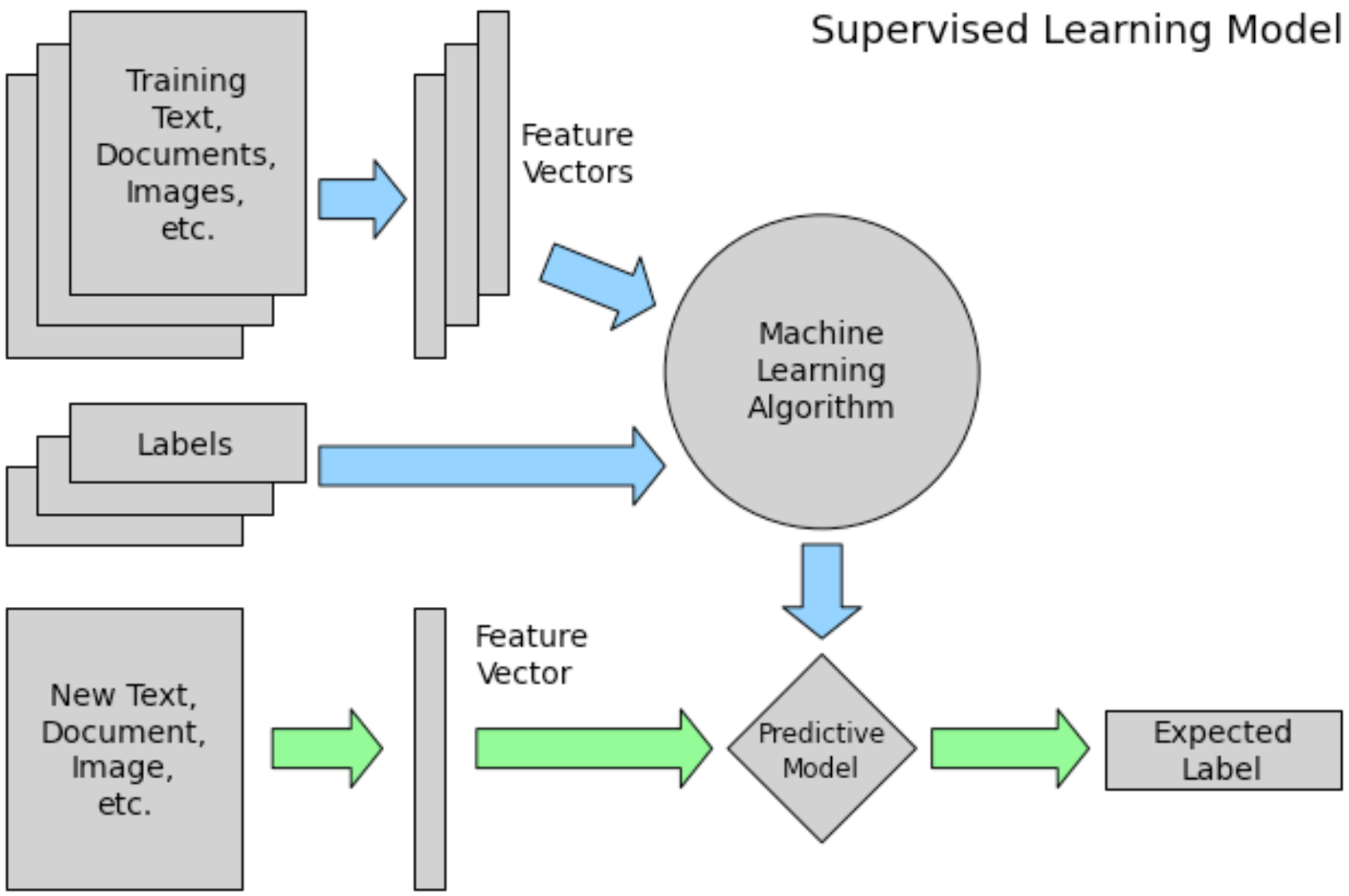
I'll discuss how to scale your model

What I'll Cover

- Scikit-learn overview
- Model Distribution
- Data flow
- RabbitMQ
- Demo
- Scalability
- Other considerations

Finally I'll cover some additional things to consider when using scikit learn models in a realtime production environment.

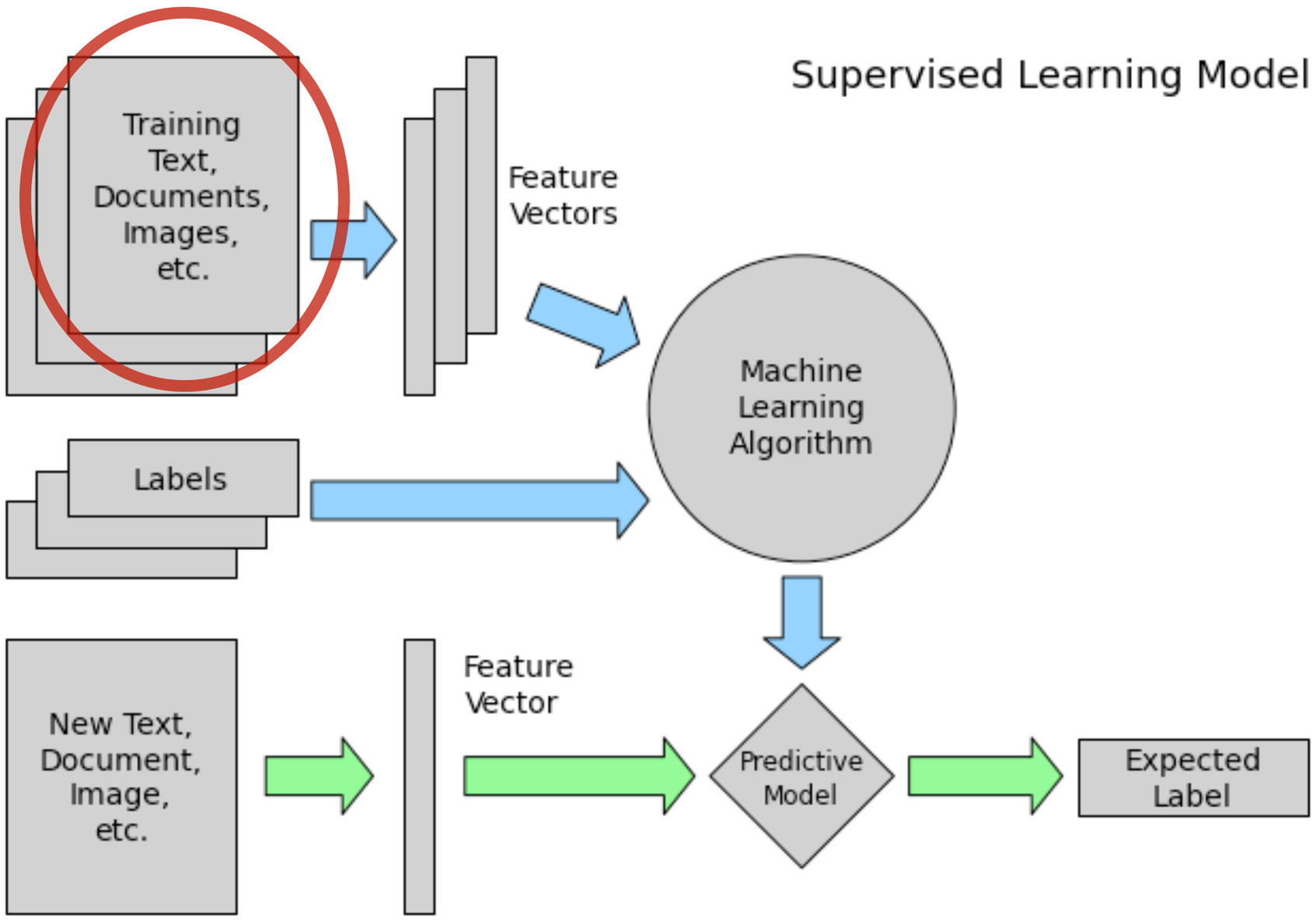
Supervised Learning



The demo in this talk will use a supervised learning algorithm.

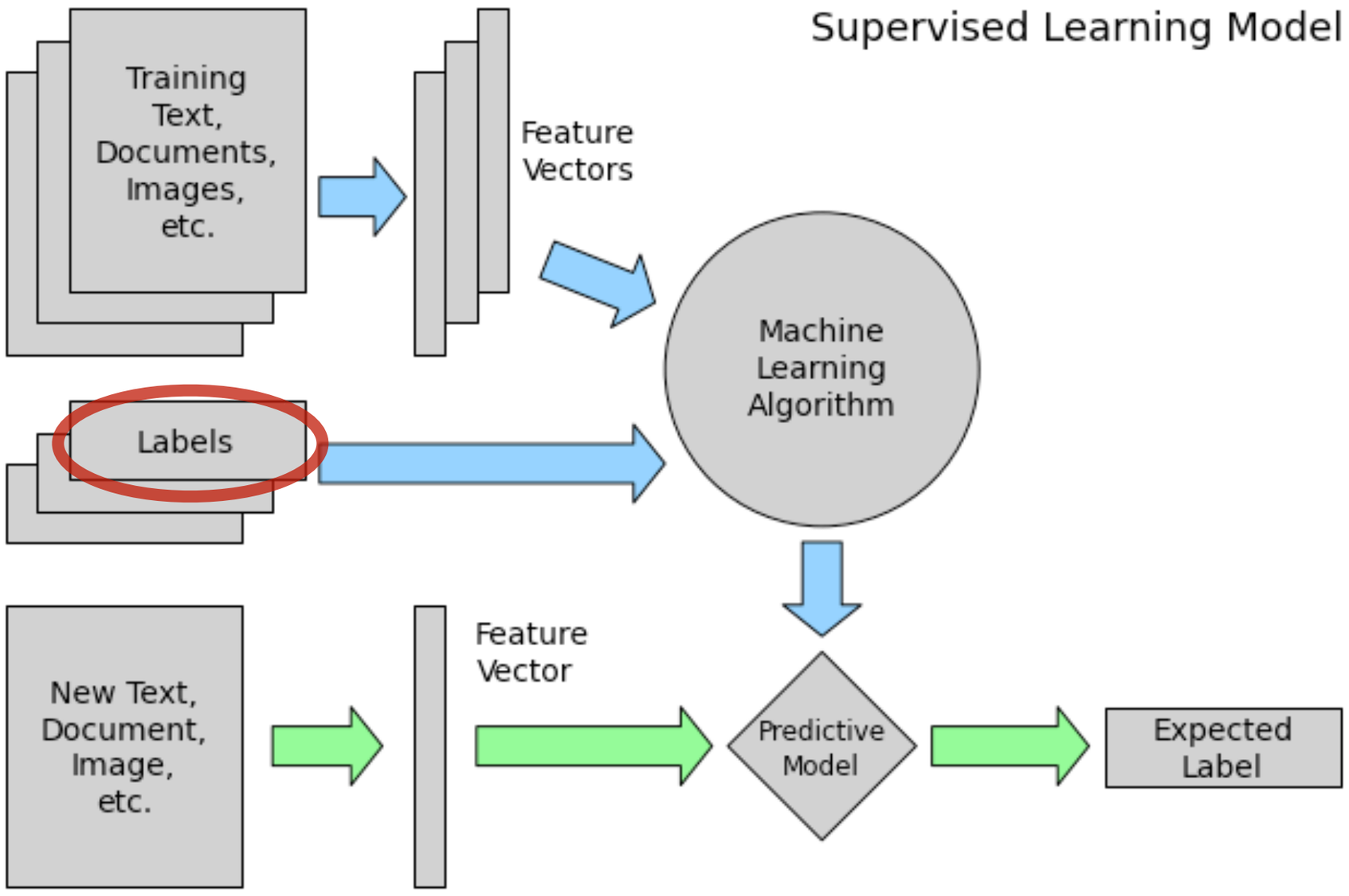
So let’s start off, by defining what the training process looks like for a supervised model.

Supervised Learning



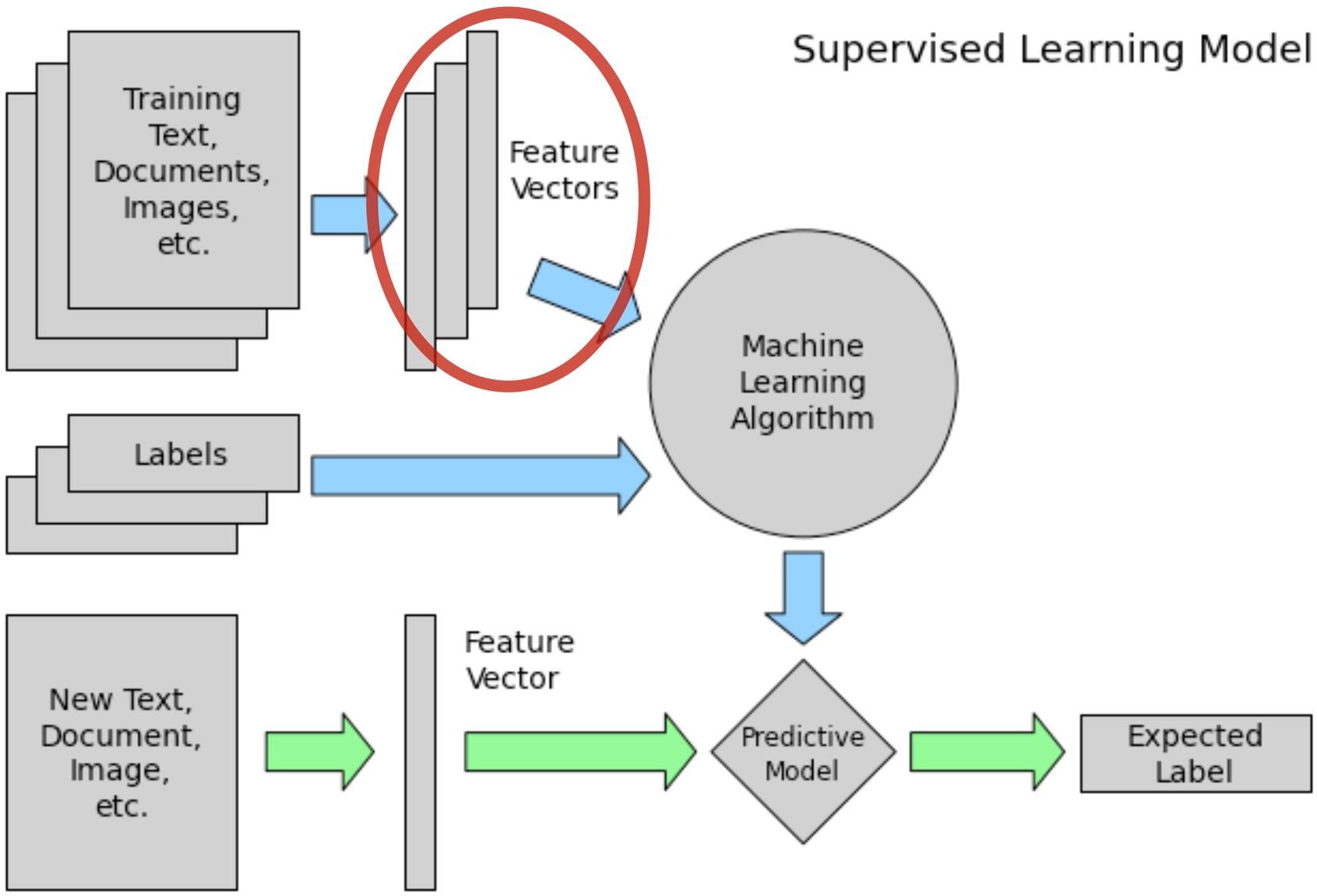
1) You start off with some input that may or may not be numerical. For example you might have text documents as input. This is called your training set.

Supervised Learning



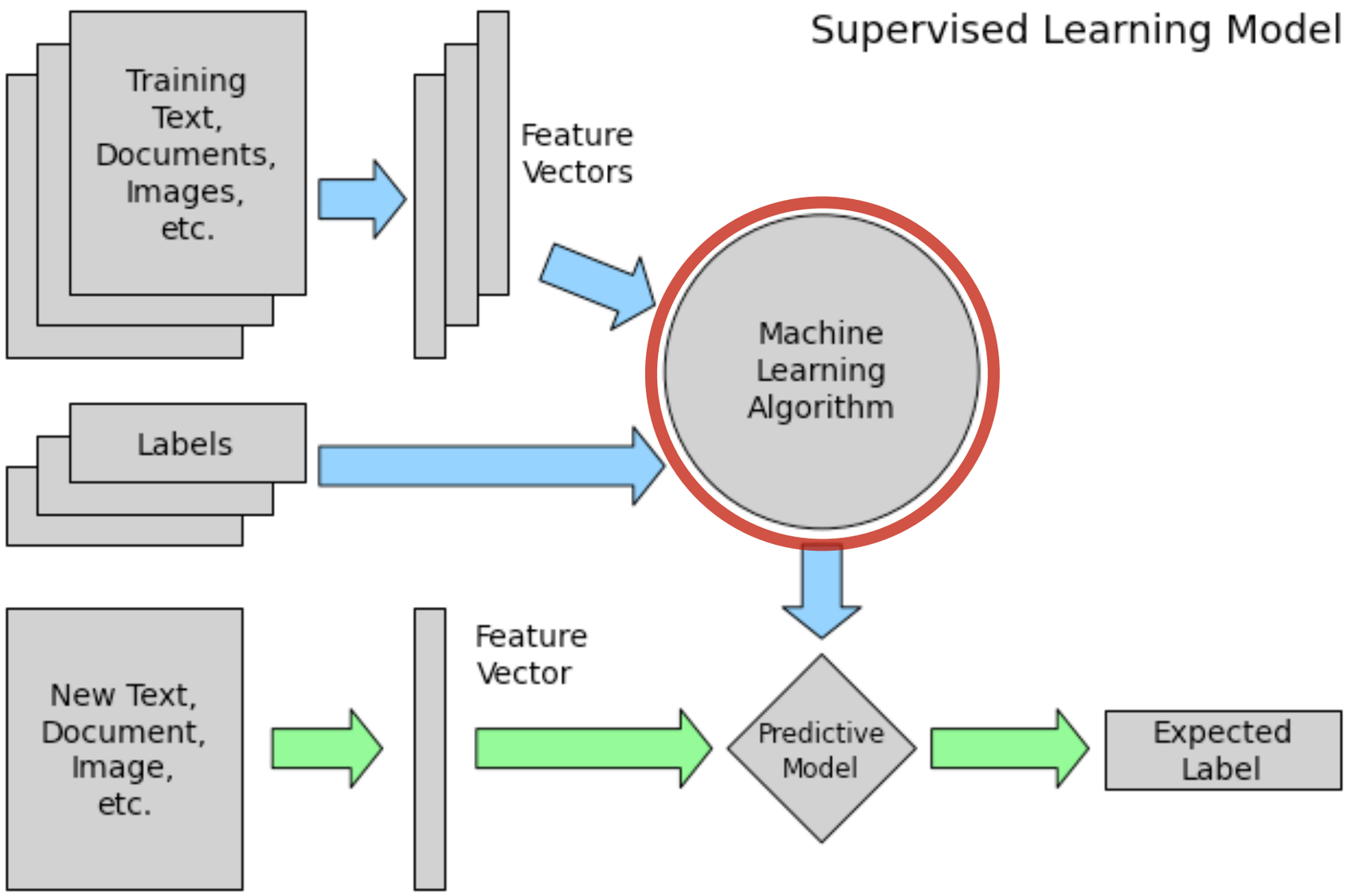
2) You also have labels for each piece of training data.

Supervised Learning



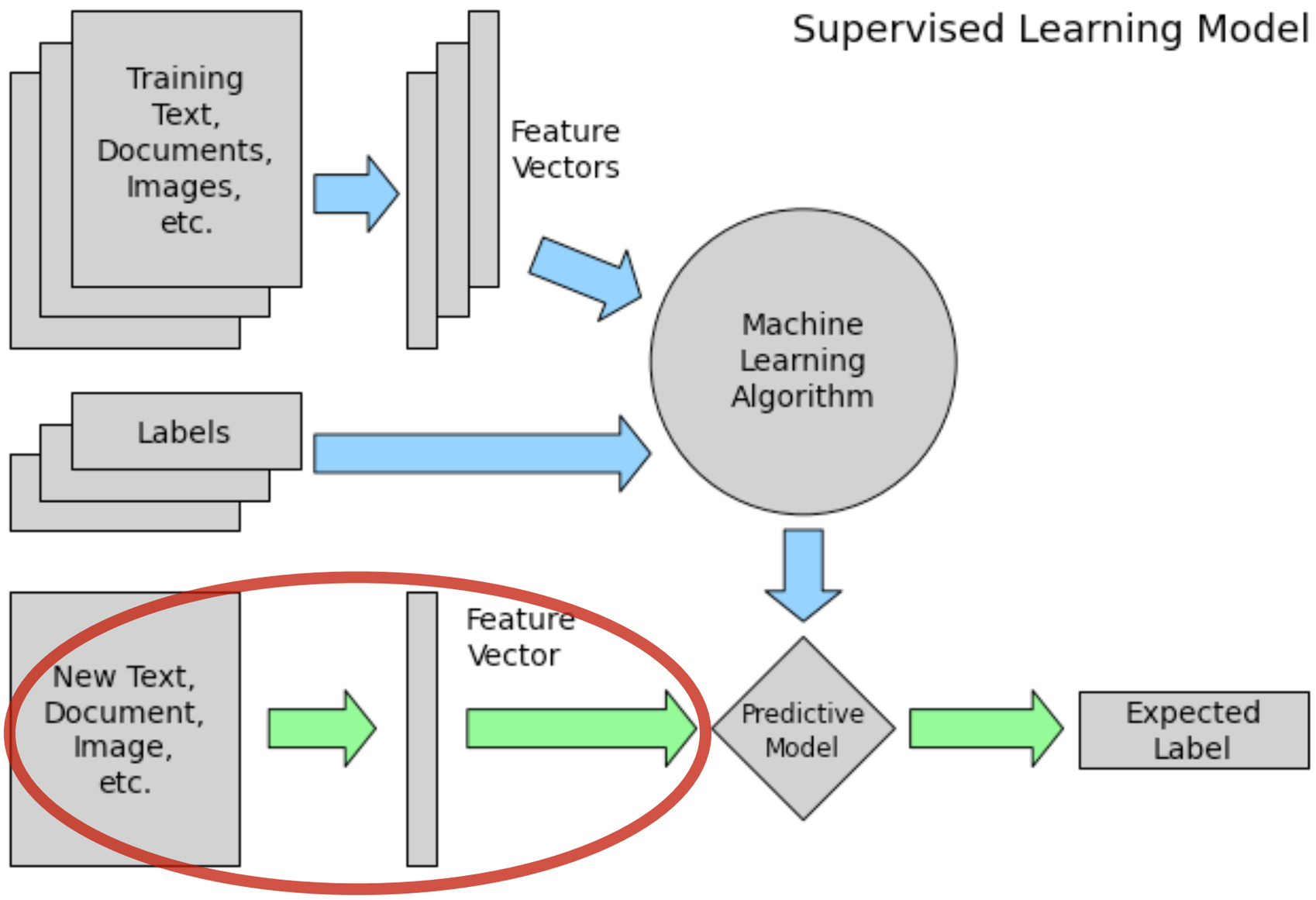
3) You vectorize your training data (which means converting it to numerical values).

Supervised Learning



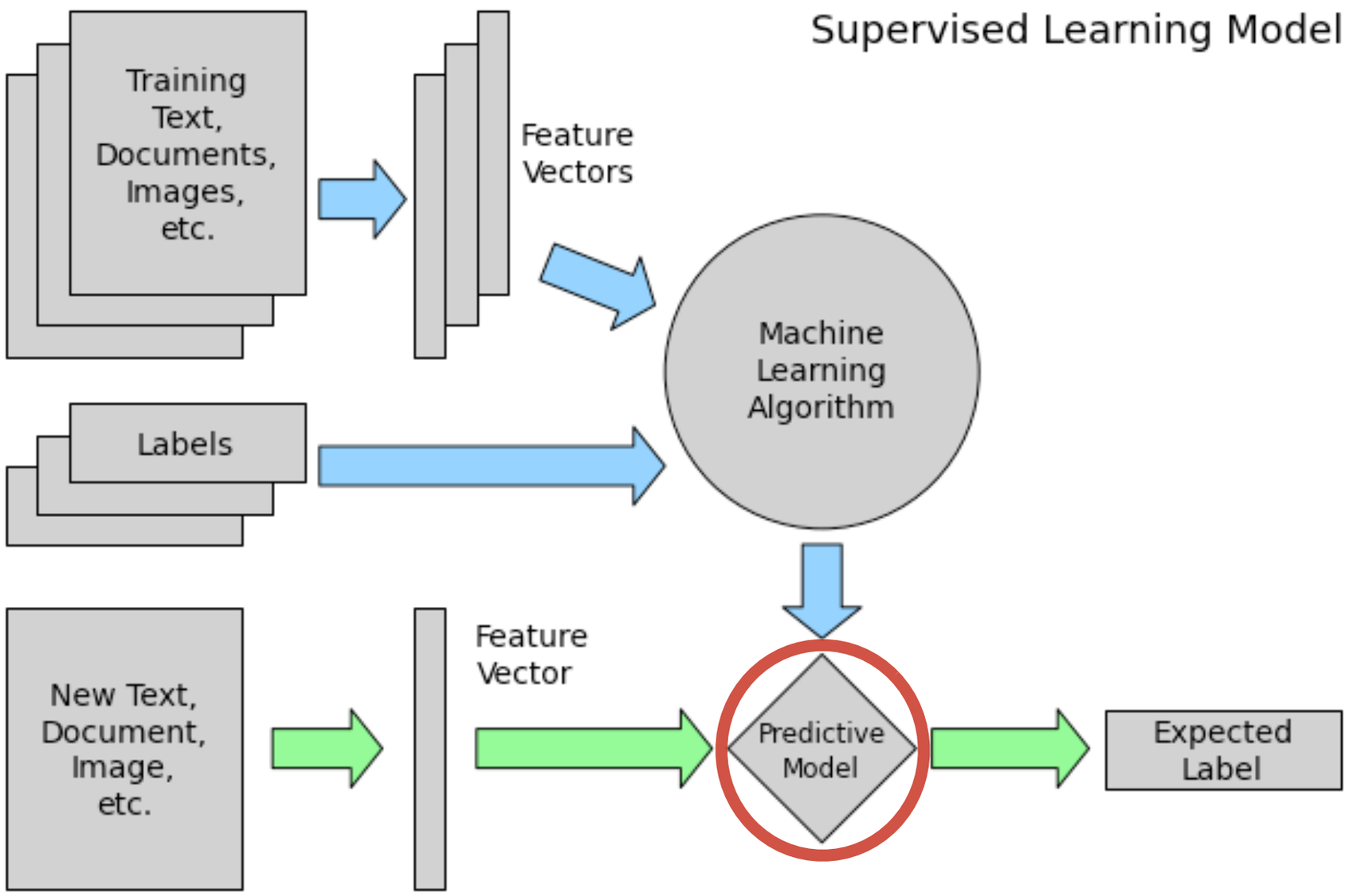
4) Then you train your machine learning algorithm using your vectorized training data, and the labels as input. This is often referred to as fitting your model.
5) At this point you have a model that can take a new piece of unlabeled data, and predict the label.

Supervised Learning



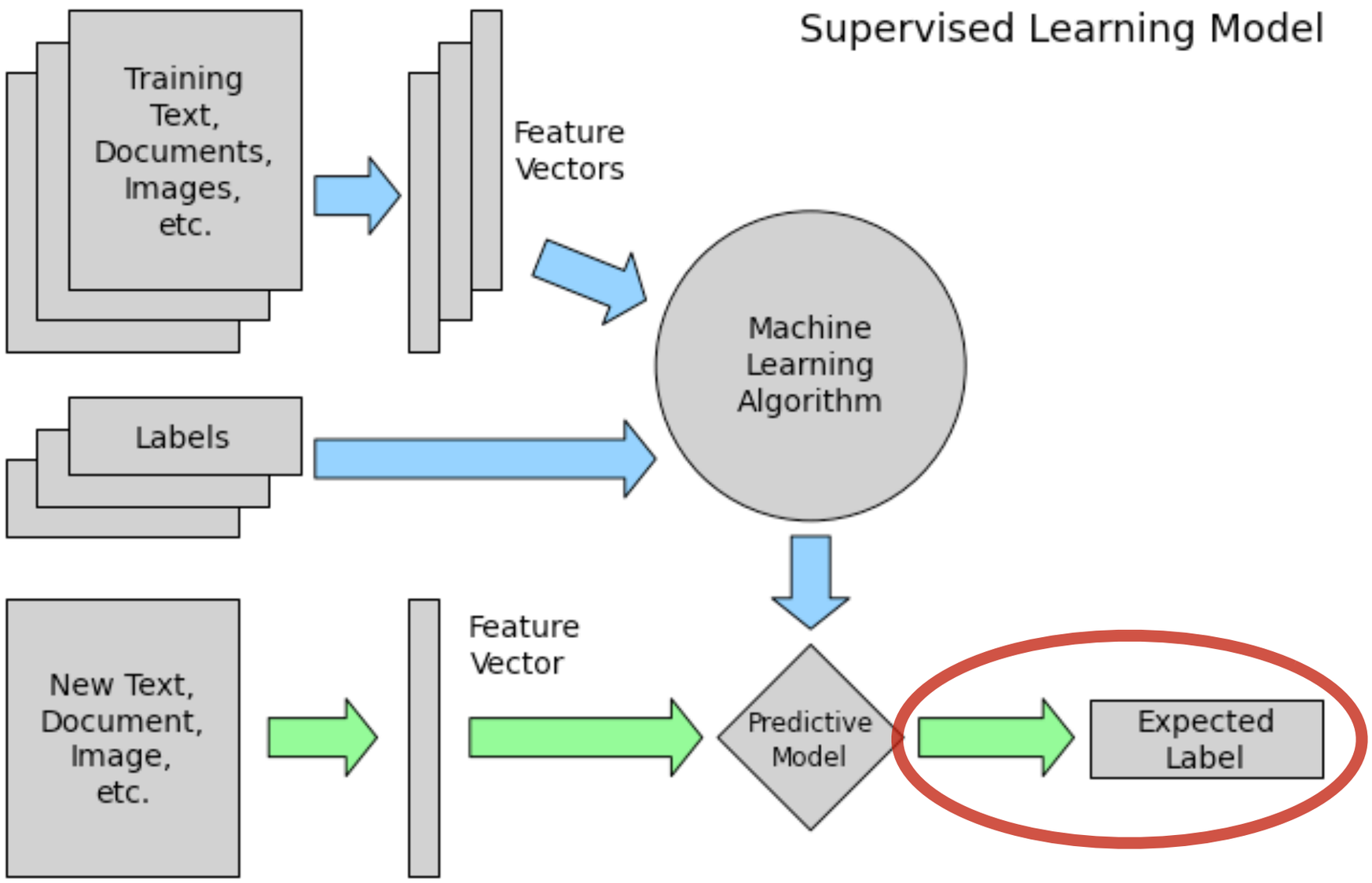
6) So again you need to vectorize your new data point.

Supervised Learning



7) Then you input it into your trained algorithm.

Supervised Learning



8) Your algorithm will spit out a predicted label for this new data point.

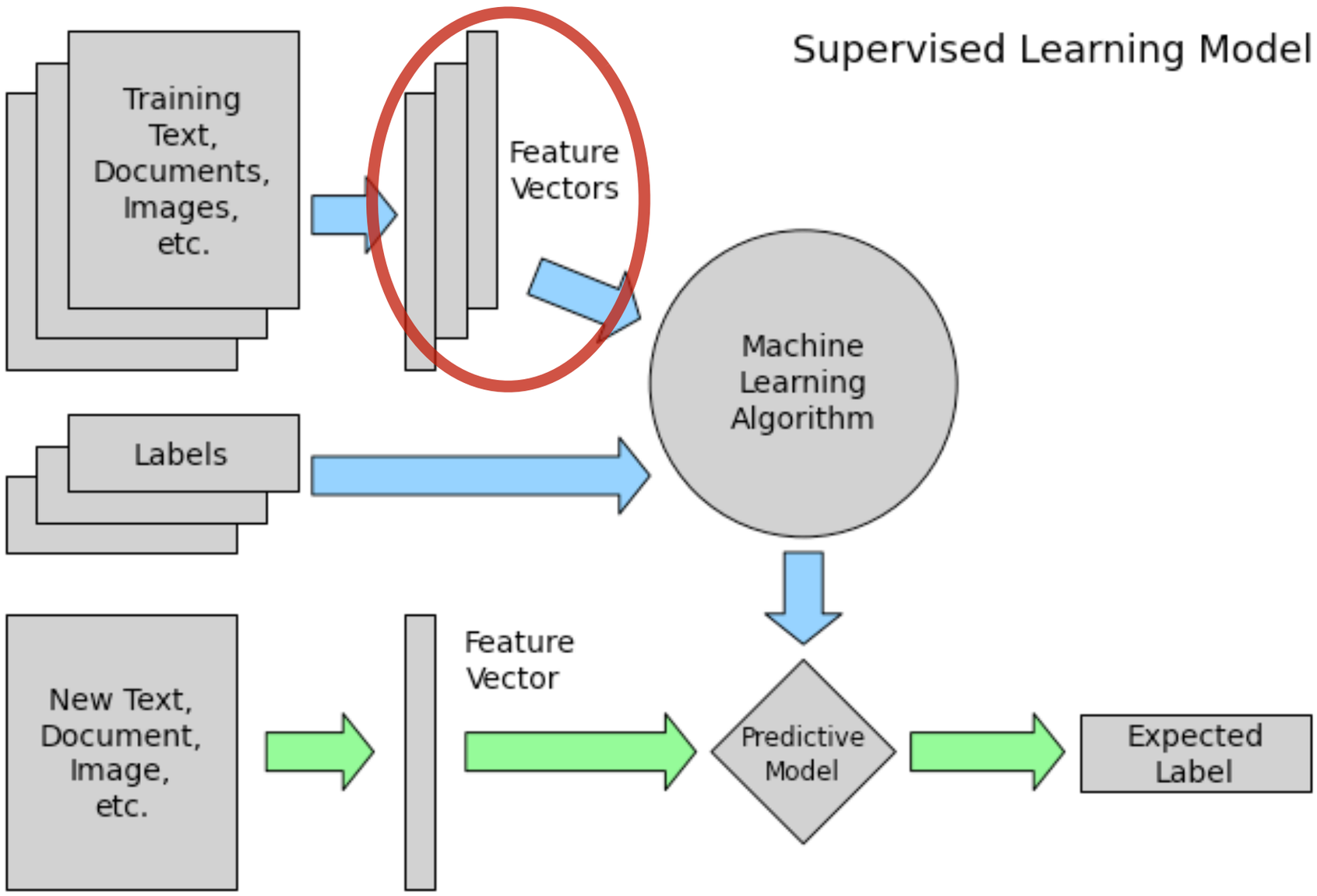
There are other types of machine learning algorithms but today I'll only be talking about supervised learning.

38 Top Wikipedias

Arabic: العربية
Bulgarian: Български
Catalan: Català
Czech: Čeština
Danish: Dansk
German: Deutsch
English: English
Spanish: Español
Estonian: Eesti
Basque: Euskara
Persian: فارسی
Finnish: Suomi
French: Français
Hebrew: עברית
Hindi: हिन्दी
Croatian: Hrvatski
Hungarian: Magyar
Indonesian: Bahasa Indonesia
Italian: Italiano
Japanese: 日本語
Kazakh: Қазақша
Korean: 한국어
Lithuanian: Lietuvių
Malay Bahasa: Melayu
Dutch: Nederlands
Norwegian: Norsk (Bokmål)
Polish: Polski
Portuguese: Português
Romanian: Română
Russian: Русский
Slovak: Slovenčina
Slovenian: Slovenščina
Serbian: Српски / Srpski
Swedish: Svenska
Turkish: Türkçe
Ukrainian: Українська
Vietnamese: Tiếng Việt
Waray-Waray: Winaray

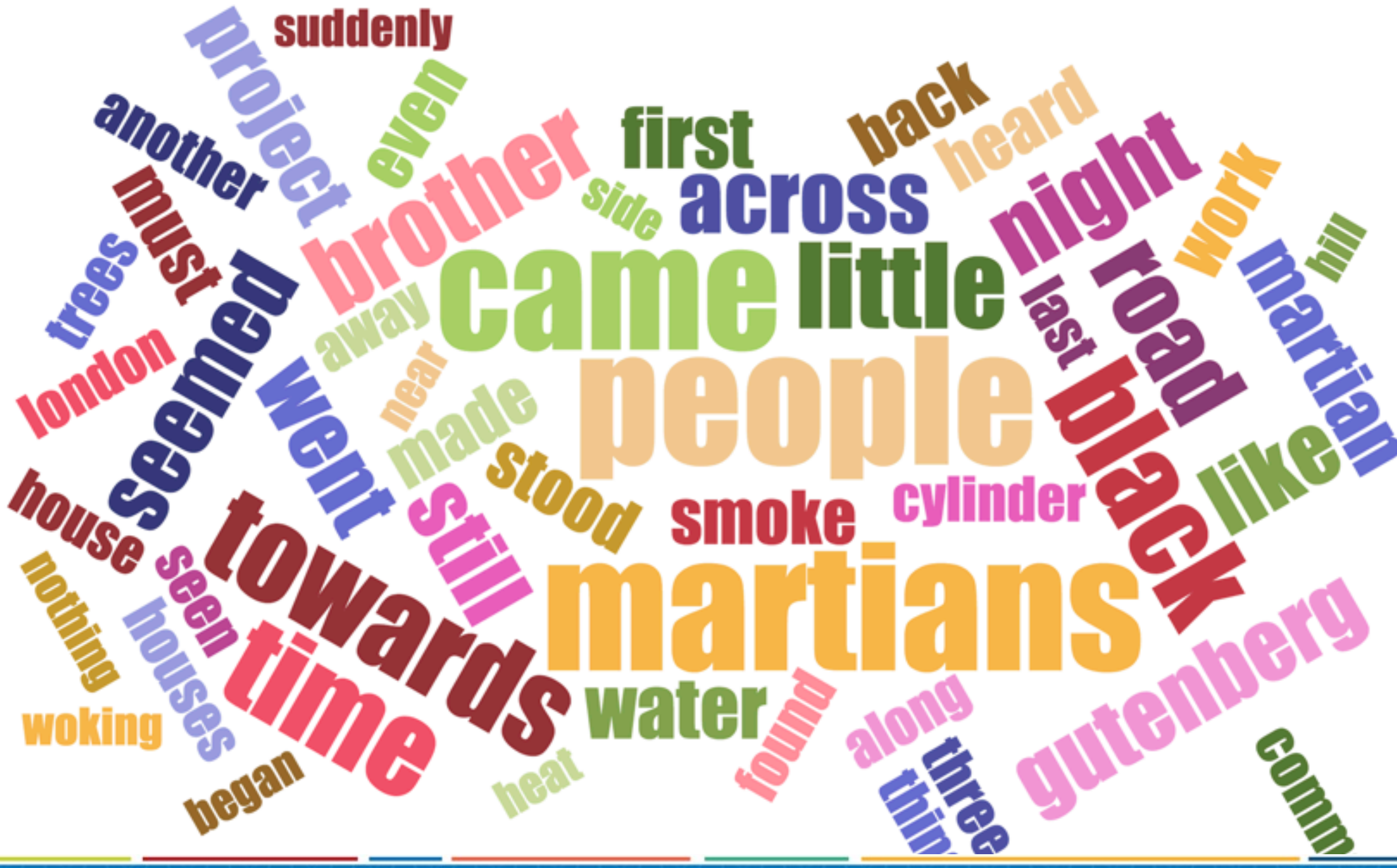
In this talk, I'm going to demonstrate one of the first models I created. A model that predicts the language of input text. **Why did I create it?** To create this model, I used 38 of the top Wikipedias based on number of articles. I dumped several of the most popular articles from each of these wikipedias.

Vectorizing Text



So going back to our diagram, the first thing we need to do is vectorize the text.

To start off I converted the wiki markup to plain text. I had read online about an approach that worked well for language classification. This approach involves counting all the combinations of **n** character sequences in the dataset. These are called n-grams. n-grams are a lot easier to understand if visualize them, so let’s see an example.

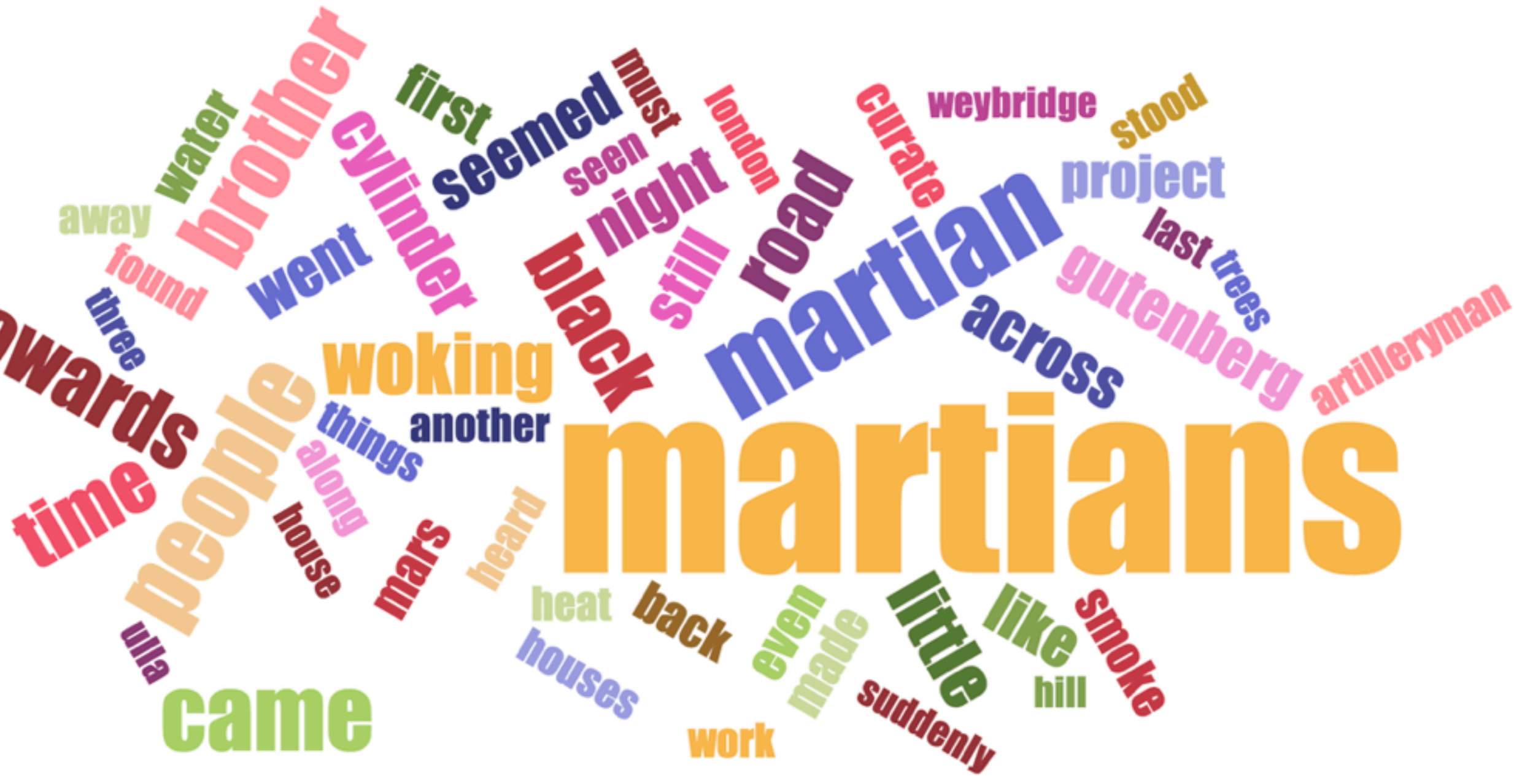


I downloaded 6 H.G Wells books from Project Gutenberg. Here's what War of the Worlds looks like if we visualize the raw word counts for each word. You can accomplish this with one line of code in scikit-learn.

NEXT SLIDE

```
CountVectorizer(analyzer='word', ngram_range=(1, 1))
```

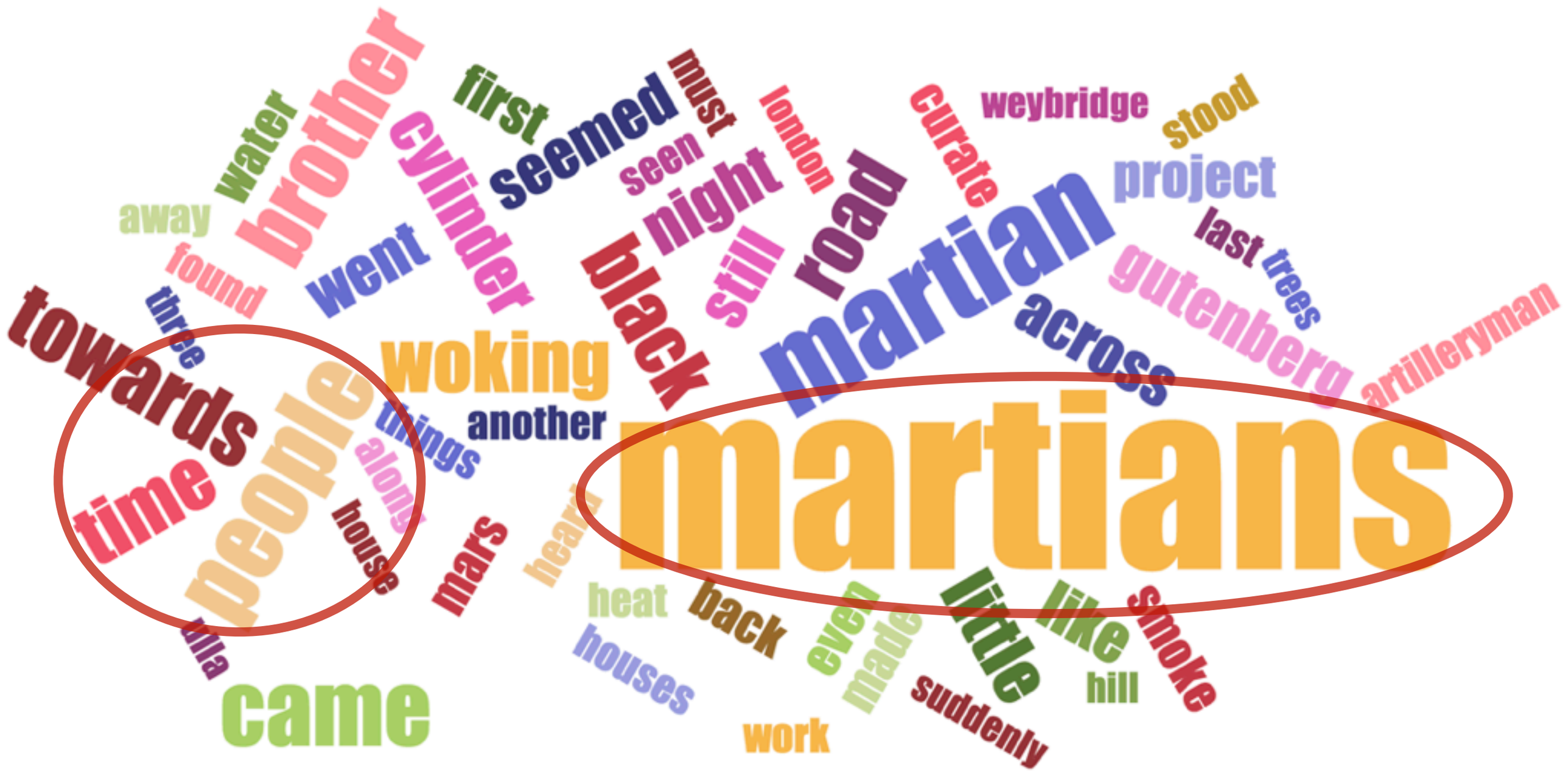
The size of the word is based on the number of times a word shows up in the book. One thing you'll notice looking at this text is that the word "martians" occurs about as frequently as the words "people" and "time." This is counter intuitive. You'd think that martians are more important in this corpus.



Fortunately there is another algorithm, called TF-IDF, that can help solve this problem. TF-IDF which stands for (term frequency–inverse document frequency) reflects how important a word is to a specific document in a collection of documents. The TF-IDF value increases based on the number of times a n-gram appears in the document, but is offset by the frequency of the n-gram in the rest of the documents. In this example a document is 1 of the 6 H.G. Wells books I downloaded.

NEXT SLIDE

```
TfidfVectorizer(analyzer='word', ngram_range=(1, 1))
```



Notice that the word “martians” is comparatively bigger. It was weighted higher than more common words like “people” and “time” because “martians” are important to War of the Worlds. TF-IDF does have trade-offs though. On very large corpuses it might not be computationally practical. For language classifier, I used TFIDF because it increased the accuracy of my model by around 1% and every percentage counts!

The Model

```
from sklearn.feature_extraction.text import TfidfVectorizer
from sklearn.pipeline import Pipeline
from sklearn.svm import SVC
```

```
vect = TfidfVectorizer(analyzer='char', ngram_range=(2, 3))
clf = SVC()
text_clf = Pipeline([
    ('vect', vect),
    ('clf', clf),
])
text_clf = text_clf.fit_transform(X_train, y_train)
```


So let’s see how to put this all together with a simple classifier. We call this type of algorithm a classifier because you provide some input and it classifies is as one of multiple classes. In the case of my algorithm, you’d provide some text as input, and it will classify the language of the text.

To start off I've defined a pipeline combining a TfidfVectorizer with a SVC classifier. A pipeline is a utility used to build a composite classifier.
The TfidfVectorizer converts the wikipedia articles into numerical form. The vectorizer first counts the number of occurrences of each n-gram in each document to "vectorize the text." It then applies the TF-IDF algorithm. The pipeline takes the output of the vectorizer and uses it as input to the classifier. All that’s left to do is fit our data to our model.

NOTE: This isn’t a talk about choosing the right algorithm

X_train

```
for x in X_train[:10]:
    print ' '.join(unicode(x, 'utf8')[:70].split())
```

```
Szöul Szöul (서울 특별시 Sŏul T'ŭkpyŏlsi, szoros átírásban: Szoul thukpjol
Sean Connery Cecil B. DeMille Award (1996) | baftaawards Beste hov
Nemecká demokratická republika Nemecká demokratická republika (NDR, h
Plastic A plastic material is any of a wide range of synthetic or sem
Phaistose ketas Phaistose ketas on põletatud savist ketas, mille leid
Soustava SI Soustava SI (zkratka z francouzského Le Système Internati
Баку Баку () је главни град Азербејџана. Налази се у јужном делу полу
Gottlob Frege nume Friedrich Ludwig Gottlob Frege
Belgrado Belgrado (Београд / Beograd em servo-croata ouça ) é a capi
Cálculo infinitesimal El cálculo infinitesimal o cálculo de infinites
```


X_train is an array of text from the wikipedia articles... besides stripping out the wiki markup, I didn't do anything else to prepare the data. The vectorizor handles converting the text to the numerical form required by the SVC algorithm.

y_train

```
y
```

```
['uk', 'cs', 'ko', 'es', 'war', 'de', 'de', 'pl', 'ar', 'cs']
```

```
from sklearn.preprocessing import LabelEncoder
le = LabelEncoder()
y_train = le.fit_transform(y)
y_train
```

```
array([35,  3, 21, ...,  8, 18, 30])
```


y_train is an array of targets, in this case the language of each article. Unlike X_train, I do need to convert these to numerical form prior to passing them into the fit method.

For each wikipedia page in X_train, there is a corresponding label in y_train.

Dimensionality Reduction

```
from sklearn.decomposition import RandomizedPCA
```

```
vect = TfidfVectorizer(analyzer='char', ngram_range=(2, 3)
pca = RandomizedPCA(n_components=50, whiten=True)
clf = SVC()
text_clf = Pipeline([
    ('vect', vect),
    ('pca', pca),
    ('clf', clf),
])
text_clf = text_clf.fit_transform(X_train, y_train)
```


One simple trick for speeding up training is to use a type of unsupervised learning algorithm called dimensionality reduction. Dimensionality reduction is the task of deriving a set of new abstract features that is smaller than the original feature set while retaining most of the variance of the original data.

One such algorithm for doing this in scikit-learn is RandomizedPCA.

PCA allows you to re-express a set of data points in terms of basic components that explain the most variance in the data. In this case we've specified that the number of new features should be 50.

Let's get rid of the pipeline so that we can see what's going on under the hood.

Dimensionality Reduction

```
vect = TfidfVectorizer(analyzer='char', ngram_range=(2, 3)
X1_train = vect.fit_transform(X_train)
X1_train.shape
```

```
(47221, 1048576)
```


So if we just run the vectorizer by itself, it produces a a dataset with over 1 million features! Because this is text data, the dataset is sparse. This is because each column in our dataset represents an n-gram that was seen in one of the documents. For any given document, most of these n-grams will have a count of 0. So let's see what happens when we run PCA on this dataset.

Dimensionality Reduction

```
vect = TfidfVectorizer(analyzer='char', ngram_range=(2, 3)
X1_train = vect.fit_transform(X_train)
X1_train.shape
```

```
(47221, 1048576)
```

```
pca = RandomizedPCA(n_components=50, whiten=True)
X2_train = pca.fit_transform(X1_train)
X2_train.shape
```

```
(47221, 50)
```


Okay, so now we've decreased the number of features from over a million to 50. But how can we be certain that this won't negatively impact the accuracy of our final classifier? The PCA algorithm has a parameter called "explained variance ratio" that's calculated during the fitting process. You can use this to see if you've picked a good value for n_components.

Dimensionality Reduction

```
pca.explained_variance_ratio_
```

```
array([ 0.36366957,  0.0984124 ,  0.04520123,  0.04459129,  0.03954548,
        0.03202782,  0.02943691,  0.02429984,  0.01907511,  0.01890865,
        0.01858178,  0.0171858 ,  0.016482  ,  0.01596849,  0.01584665,
        0.01499711,  0.01378617,  0.01368058,  0.01190577,  0.01173788,
        0.01131731,  0.01045798,  0.01029921,  0.01004662,  0.00951628,
        0.00803825,  0.00728402,  0.00700877,  0.00667922,  0.00664124,
        0.00574287,  0.00548161,  0.00500291,  0.00433605,  0.003948  ,
        0.00283822,  0.00241628,  0.00189858,  0.00168776,  0.00160769,
        0.001431  ,  0.00137901,  0.00132682,  0.00129177,  0.00126926,
        0.0012235 ,  0.00117239,  0.00114645,  0.00110811,  0.00106229])
```


Okay, so now we've decreased the number of features from over a million to 50. But how can we be certain that this won't negatively impact the accuracy of our final classifier? The PCA algorithm has a parameter called "explained variance ratio" that's calculated during the fitting process. You can use this to see if you've picked a good value for n_components.

Dimensionality Reduction

```
pca.explained_variance_ratio_
```

```
array([ 0.36366957,  0.0984124 ,  0.04520123,  0.04459129,  0.03954548,
        0.03202782,  0.02943691,  0.02429984,  0.01907511,  0.01890865,
        0.01858178,  0.0171858 ,  0.016482  ,  0.01596849,  0.01584665,
        0.01499711,  0.01378617,  0.01368058,  0.01190577,  0.01173788,
        0.01131731,  0.01045798,  0.01029921,  0.01004662,  0.00951628,
        0.00803825,  0.00728402,  0.00700877,  0.00667922,  0.00664124,
        0.00574287,  0.00548161,  0.00500291,  0.00433605,  0.003948  ,
        0.00283822,  0.00241628,  0.00189858,  0.00168776,  0.00160769,
        0.001431  ,  0.00137901,  0.00132682,  0.00129177,  0.00126926,
        0.0012235 ,  0.00117239,  0.00114645,  0.00110811,  0.00106229])
```

```
pca.explained_variance_ratio_.sum()
```

```
0.999999999999999
```


And we can see by adding all of these together that we've retained almost all of the variance from the original dataset; all while significantly decreasing the size of the input to our classifier! This is really awesome, but one last thing!

Don’t Do What I Just Did!

sklearn.decomposition.TruncatedSVD

class `sklearn.decomposition.`**`TruncatedSVD`**(*n_components=2, algorithm='randomized', n_iterations=5, random_state=None, tol=0.0*)

Dimensionality reduction using truncated SVD (aka LSA).

This transformer performs linear dimensionality reduction by means of truncated singular value decomposition (SVD). It is very similar to PCA, but operates on sample vectors directly, instead of on a covariance matrix. This means it can work with scipy.sparse matrices efficiently.

In particular, truncated SVD works on term count/tf–idf matrices as returned by the vectorizers in sklearn.feature_extraction.text. In that context, it is known as latent semantic analysis (LSA).

Don’t use RandomizedPCA on scipy.sparse matrices, such as those output by TfidfVectorizer. TruncatedSVD was designed to work efficiently on sparse matrices, and support for them will be drop from the RandomizedPCA in a future release of scikit-learn. I used RandomizedPCA here because TruncatedSVD doesn’t currently calculate the “explained variance ratio.” It will probably be added in a future release.

Distributing The Model

So now that you have this awesome model, "now what?"

One of the first problems you'll want to solve is how to distribute your model? The recommended method for distributing your model is to use the built-in pickle module to serialize the model to disk and distribute it as part of your application. You can also store it in a database such as GridFS or Amazon S3. In the case of my model, it took up roughly 200MB in memory. This is pretty big, but easily storable on disk (and more importantly in memory).

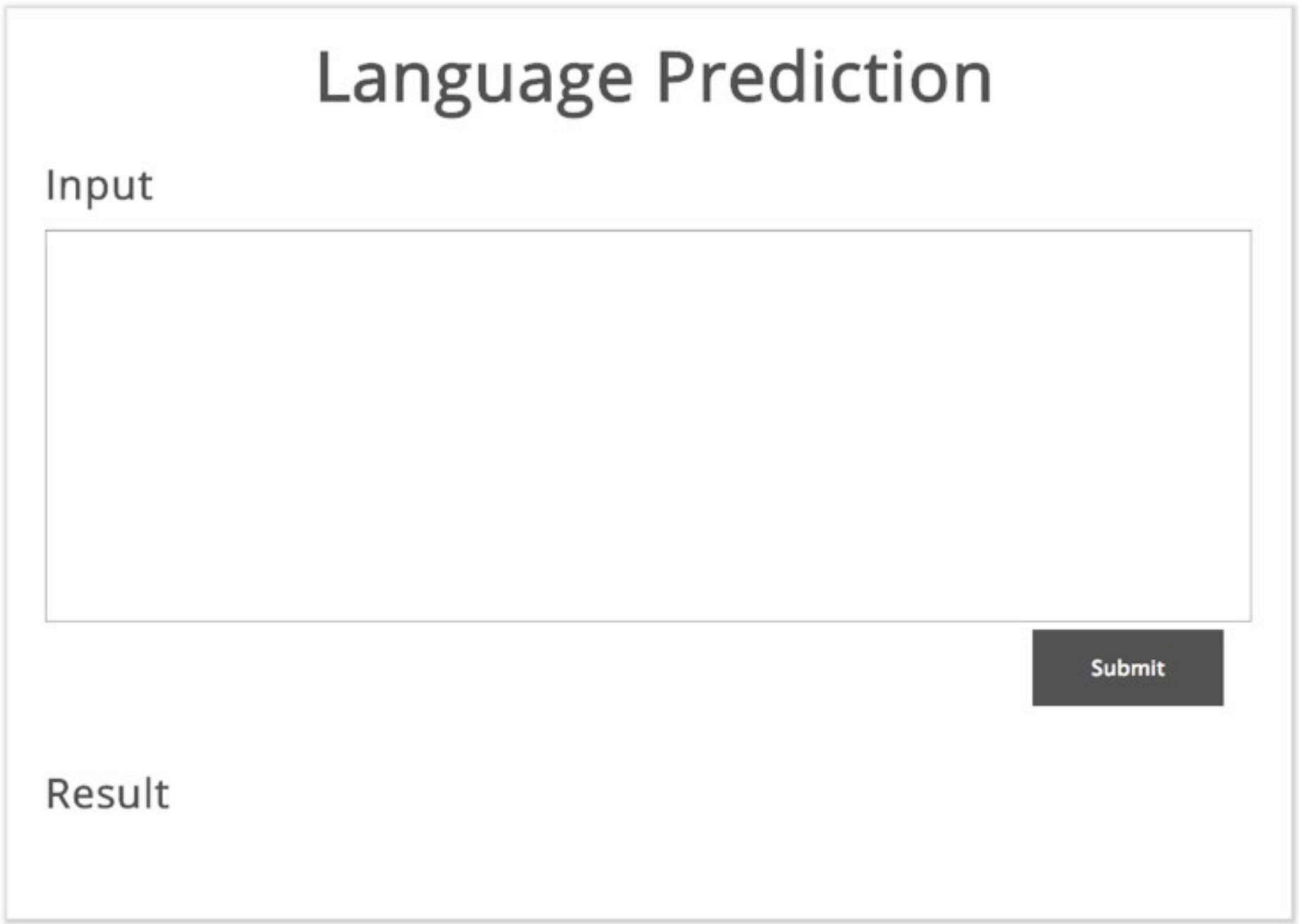
One caveat to this approach is if you upgrade scikit-learn, your pickled model is not guaranteed to work. It’s important that you keep detailed records of you training set, and your models training parameters if you want to be able to upgrade scikit-learn in the future. Take care when upgrading scikit-learn in production to make sure you’re not going to break your application.

Data Input

Next let's discuss how we're going to get data into our model.

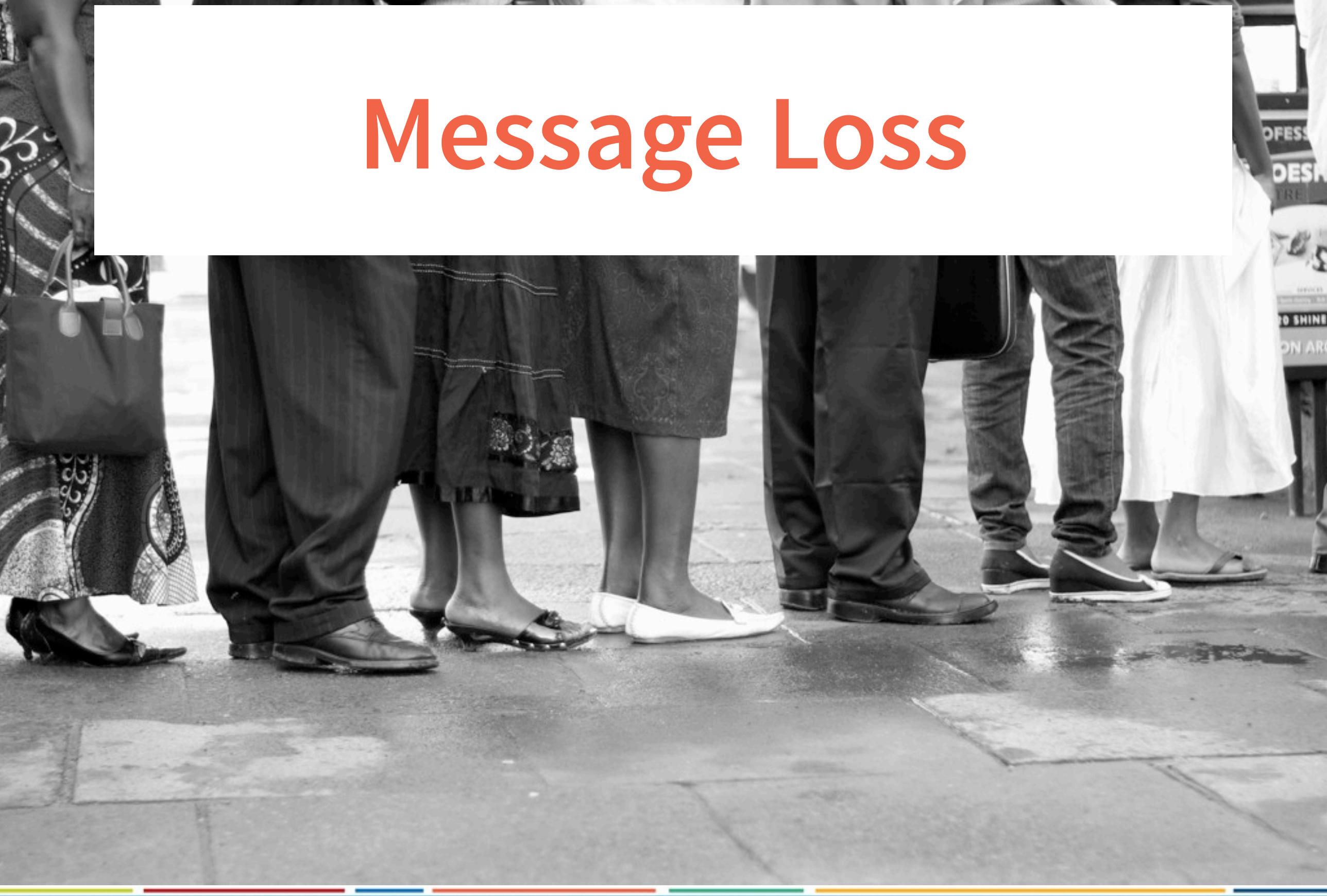
You're data could be coming from many types of sources, a web front-end, a DB trigger, etc.. In many cases, you can't easily control the rate of incoming data and you don't want to hold up the front-end or the database while you wait for a prediction to be made. In these cases, it's useful to be able to process your data asynchronously.

The Client



In the example I'm giving today, we created a simple web front-end (similar to google translate) where a user can enter some text to be classified, and get a classification back. We don't want to hold up a thread or process in the client waiting on our classifier to do its thing. Rather the front-end sends the input to a REST API which will record the text input and return a tracking_id that the client can then use to get the result. There are several other ways we could design this, but for my application, this design works fine.

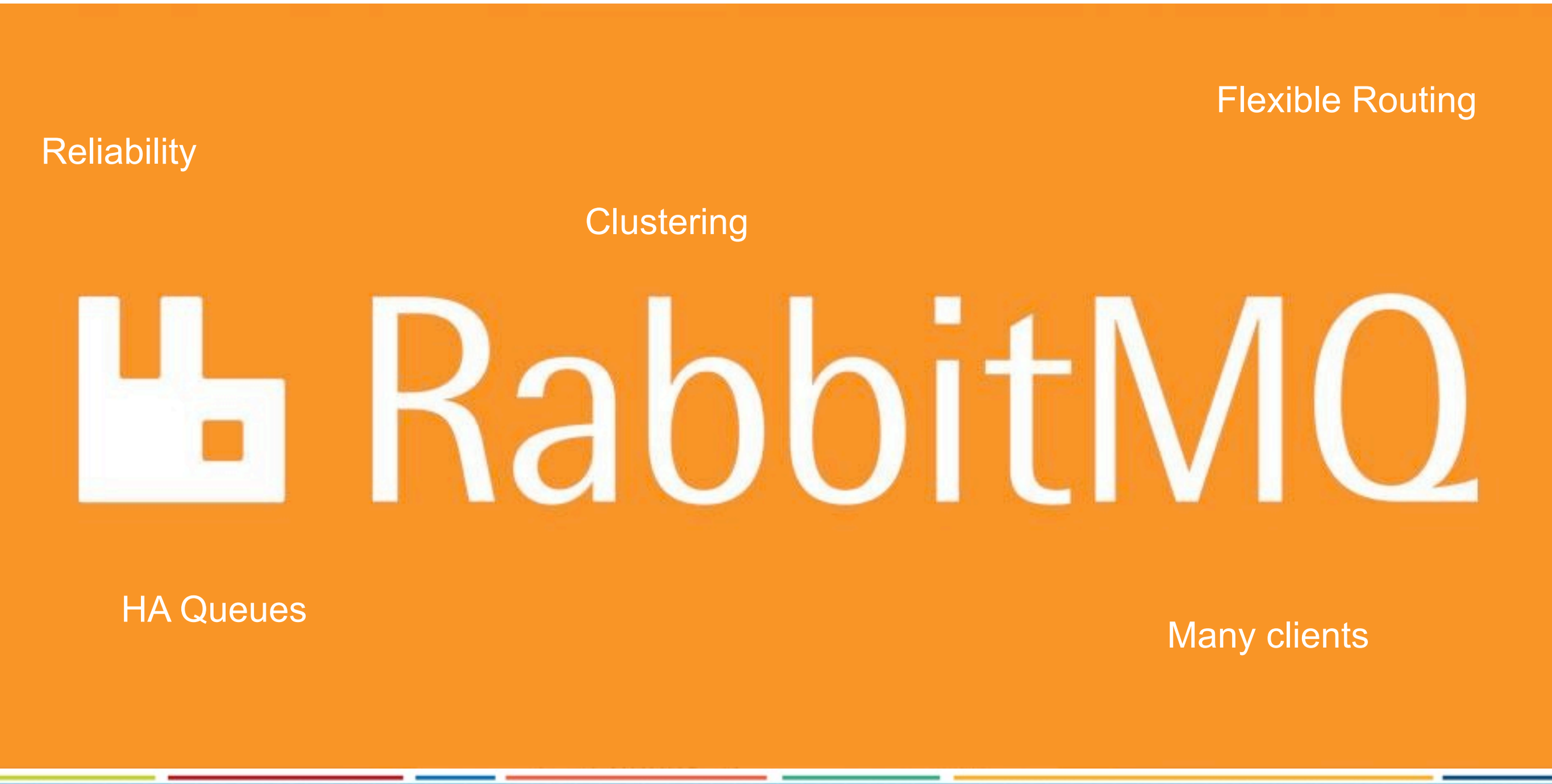
TODO: Break this into multiple slides and highlight what I'm talking about



Decoupling the UI from the backend in this way solves one design issue. However another thing to consider is weather you can afford to lose messages. If all of your data needs to be processed you have 2 options. You either need to have a built in retry mechanism in the front end, or you need a persistent and durable queue to hold your messages in the backend.

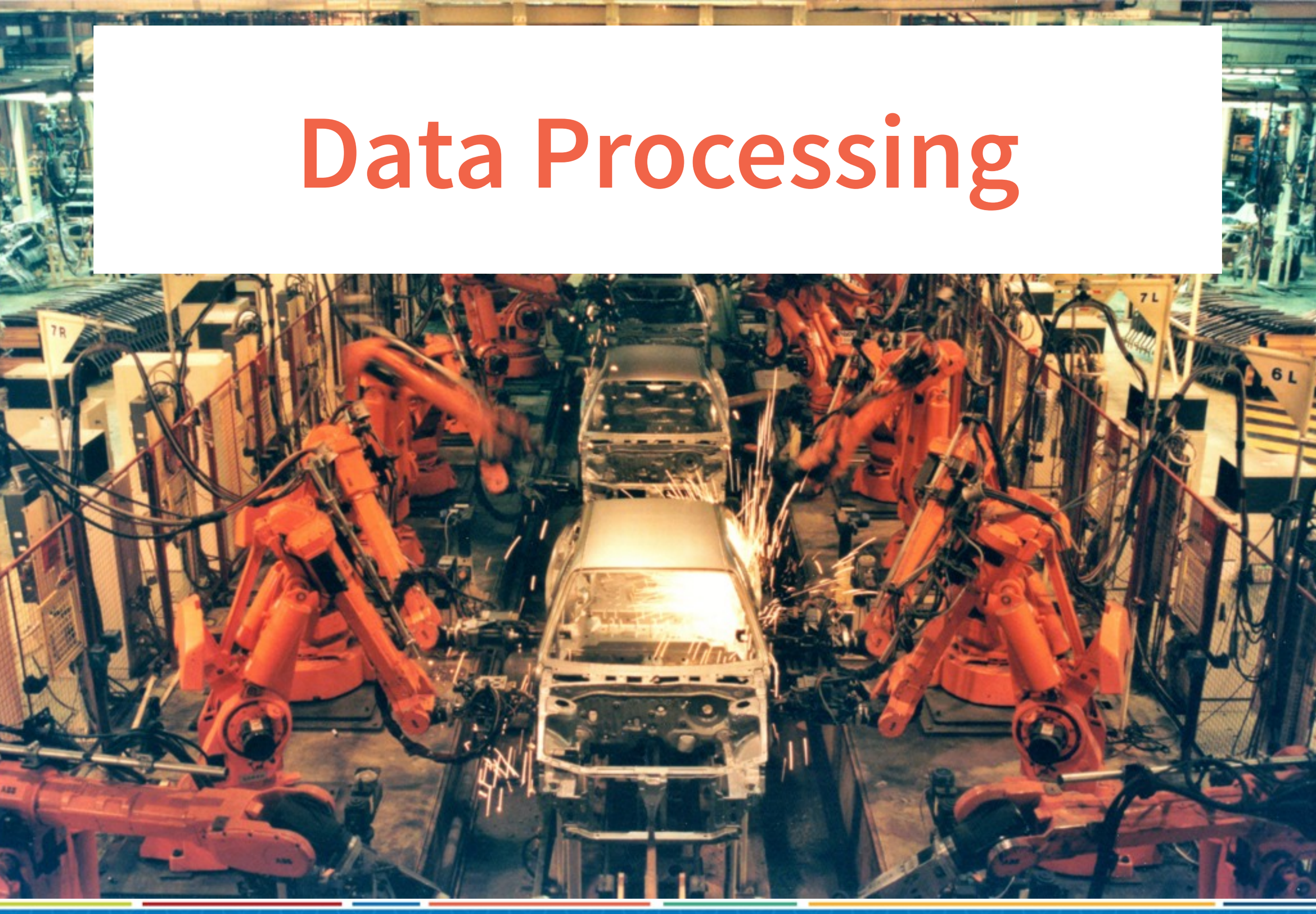
Enter RabbitMQ. One of the many features provided by RabbitMQ is Highly Available Queues. By using RabbitMQ, you can ensure that every message is processed without needing to implement a fancy (and likely error prone) retry mechanism in your front-end.

Enter RabbitMQ



RabbitMQ uses AMQP (Advanced Message Queuing Protocol) for all client communication. Using AMQP allows clients running on different platforms or written in different languages, to easily send messages to each other. From a high level, AMQP enables clients to publish messages, and other clients to consume those messages. It does all this without requiring you to roll your own protocol or library.

Data Processing



Once you hook your data input source into RabbitMQ and start publishing data, all you need to do is put your model in a persistent worker and start consuming input.

The Consumer

```
class LanguagePredictorWorker(object):

    classifier, label_encoder = load_pickled_files(clf)
    def __init__(self):
        subscribe_to_queue()
        self.language_coll = get_db_collection()

    def process_event(self, body, message):
        text = body['text']
        _id = body['id']
        language = self.predict_language_for_text(text)
        result = self.language_coll.update(
            {'_id': ObjectId(_id), 'text_input': text},
            {'$set': {'language': language}},
        )
        message.ack()

    def predict_language_for_text(self, text):
        lang_vector = self.classifier.predict([text])
        lang_labels = self.label_encoder.inverse_transform(lang_vector)
        return lang_labels[0]

worker = LanguagePredictorWorker()
worker.main()
```


In the case of my language classification model, we implemented a simple consumer (**NEXT**) that unpickles the classifier and subscribes to an input queue.

The Consumer

```
class LanguagePredictorWorker(object):

    classifier, label_encoder = load_pickled_files(clf)
    def __init__(self):
        subscribe_to_queue()
        self.language_coll = get_db_collection()

    def process_event(self, body, message):
        text = body['text']
        _id = body['id']
        language = self.predict_language_for_text(text)
        result = self.language_coll.update(
            {'_id': ObjectId(_id), 'text_input': text},
            {'$set': {'language': language}},
        )
        message.ack()

    def predict_language_for_text(self, text):
        lang_vector = self.classifier.predict([text])
        lang_labels = self.label_encoder.inverse_transform(lang_vector)
        return lang_labels[0]

worker = LanguagePredictorWorker()
worker.main()
```


that unpickles the classifier and subscribes to an input queue.

It then runs an event loop (**NEXT**)

The Consumer

```
class LanguagePredictorWorker(object):

    classifier, label_encoder = load_pickled_files(clf)
    def __init__(self):
        subscribe_to_queue()
        self.language_coll = get_db_collection()

    def process_event(self, body, message):
        text = body['text']
        _id = body['id']
        language = self.predict_language_for_text(text)
        result = self.language_coll.update(
            {'_id': ObjectId(_id), 'text_input': text},
            {'$set': {'language': language}},
        )
        message.ack()

    def predict_language_for_text(self, text):
        lang_vector = self.classifier.predict([text])
        lang_labels = self.label_encoder.inverse_transform(lang_vector)
        return lang_labels[0]

worker = LanguagePredictorWorker()
worker.main()
```

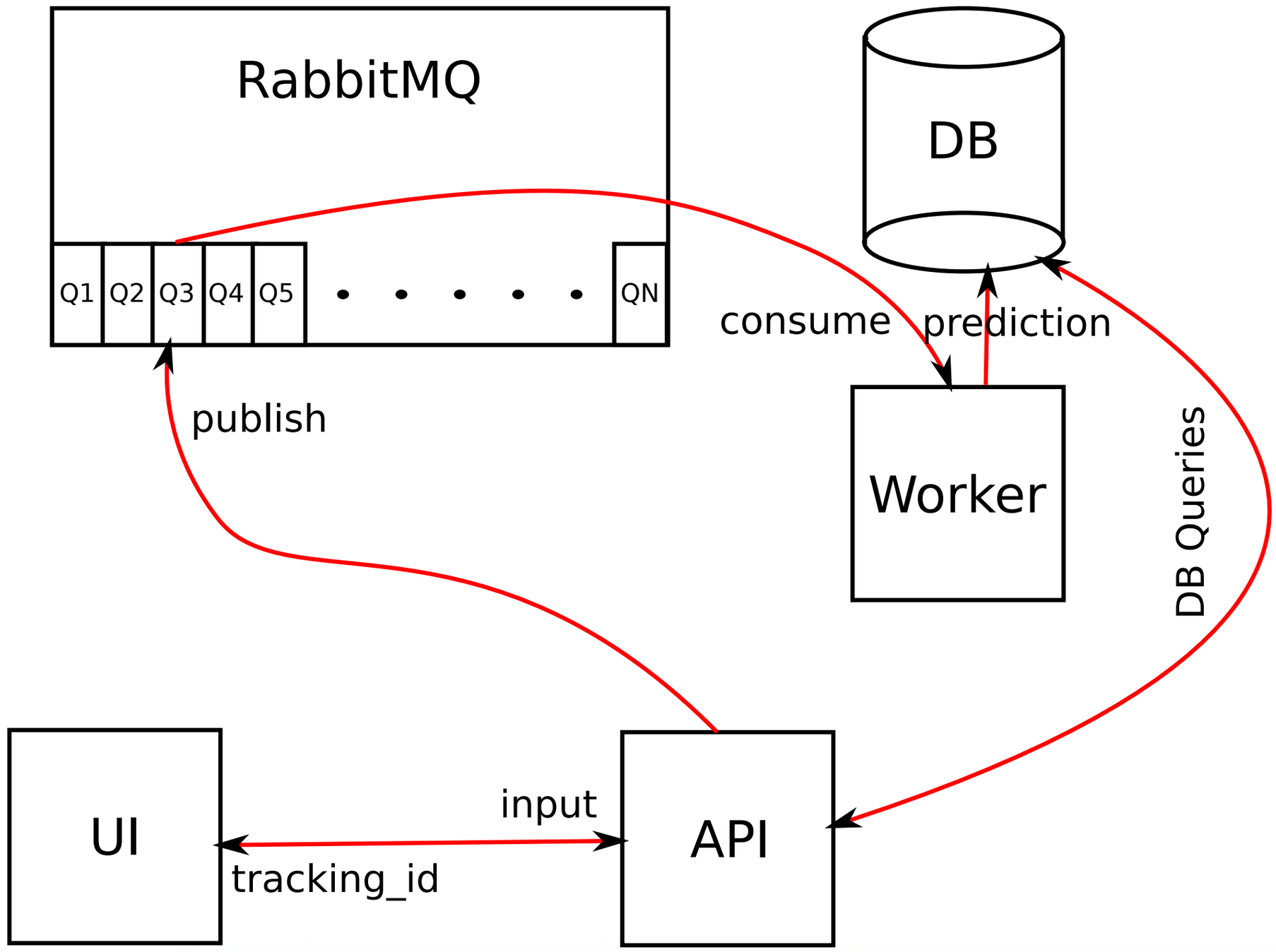

that pulls new messages as they become available and passes them to process_event **(NEXT)**

The Consumer

```
class LanguagePredictorWorker(object):

    classifier, label_encoder = load_pickled_files(clf)
    def __init__(self):
        subscribe_to_queue()
        self.language_coll = get_db_collection()

    def process_event(self, body, message):
        text = body['text']
        _id = body['id']
        language = self.predict_language_for_text(text)
        result = self.language_coll.update(
            {'_id': ObjectId(_id), 'text_input': text},
            {'$set': {'language': language}},
        )
        message.ack()

    def predict_language_for_text(self, text):
        lang_vector = self.classifier.predict([text])
        lang_labels = self.label_encoder.inverse_transform(lang_vector)
        return lang_labels[0]

worker = LanguagePredictorWorker()
worker.main()
```

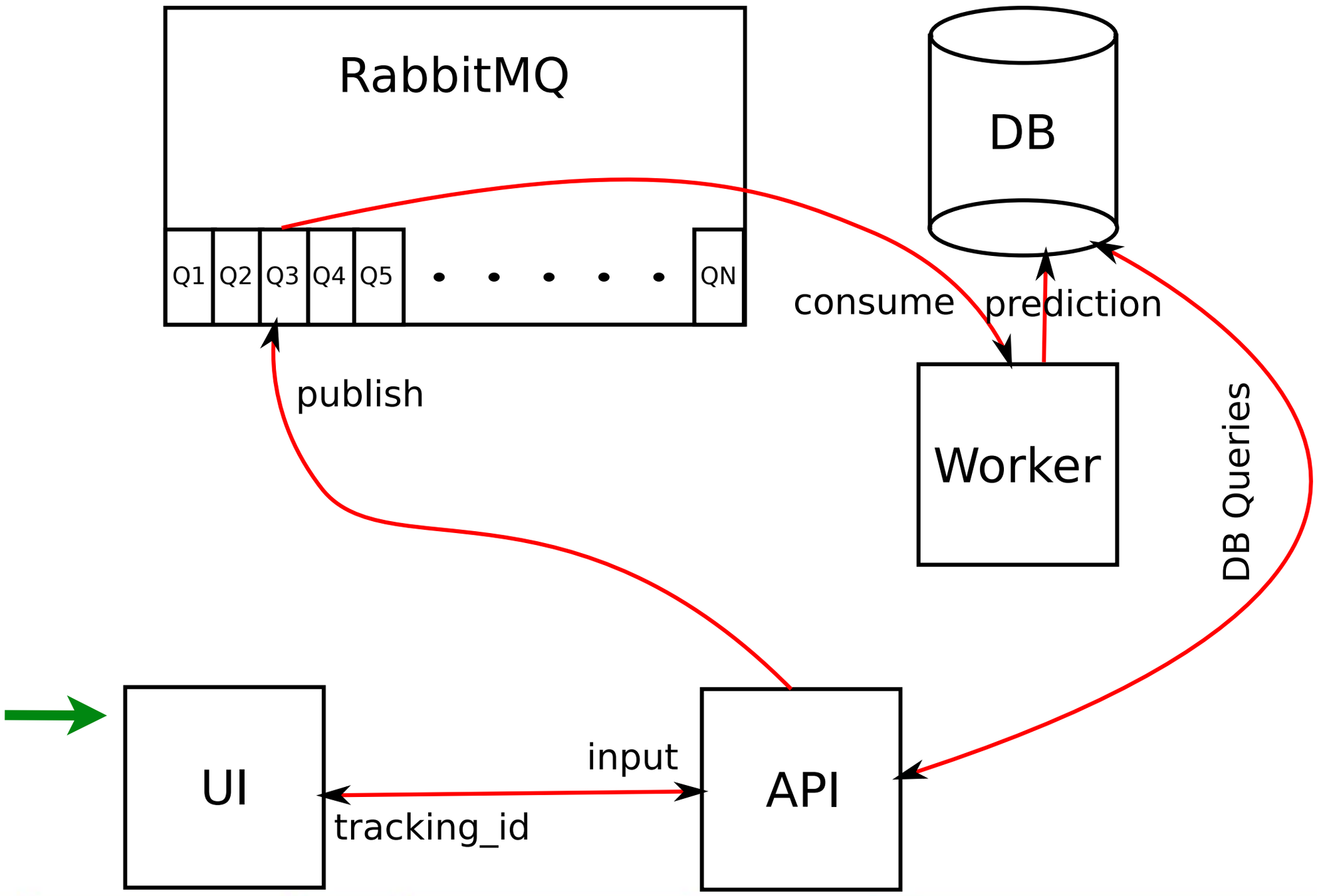

Process event calls predict on our model and converts the numerical prediction to a human readable format. This prediction is then stored in our DB for the front-end to retrieve.

The Design



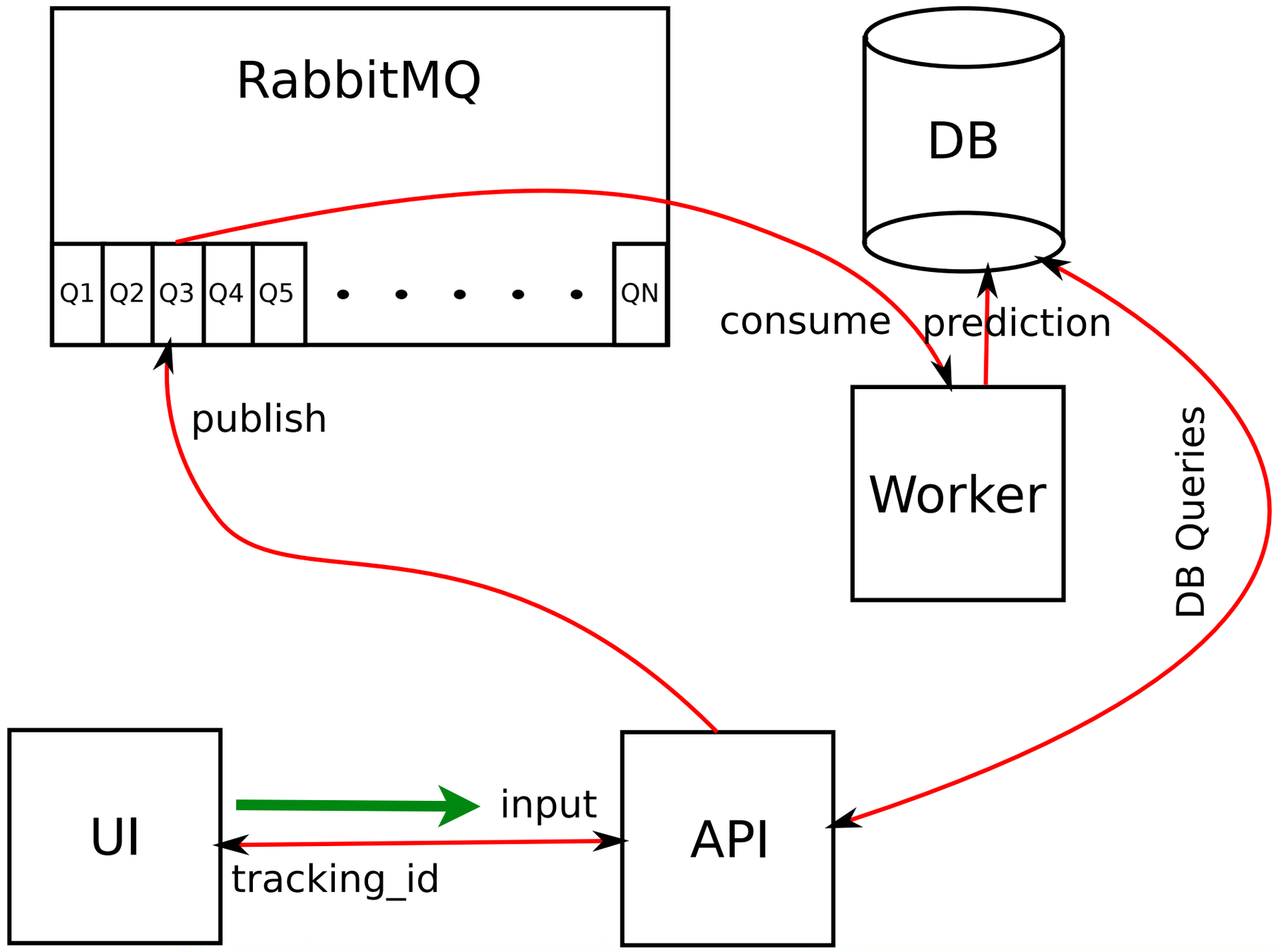
So that's basically it. Our design looks a little something like this:

The Design



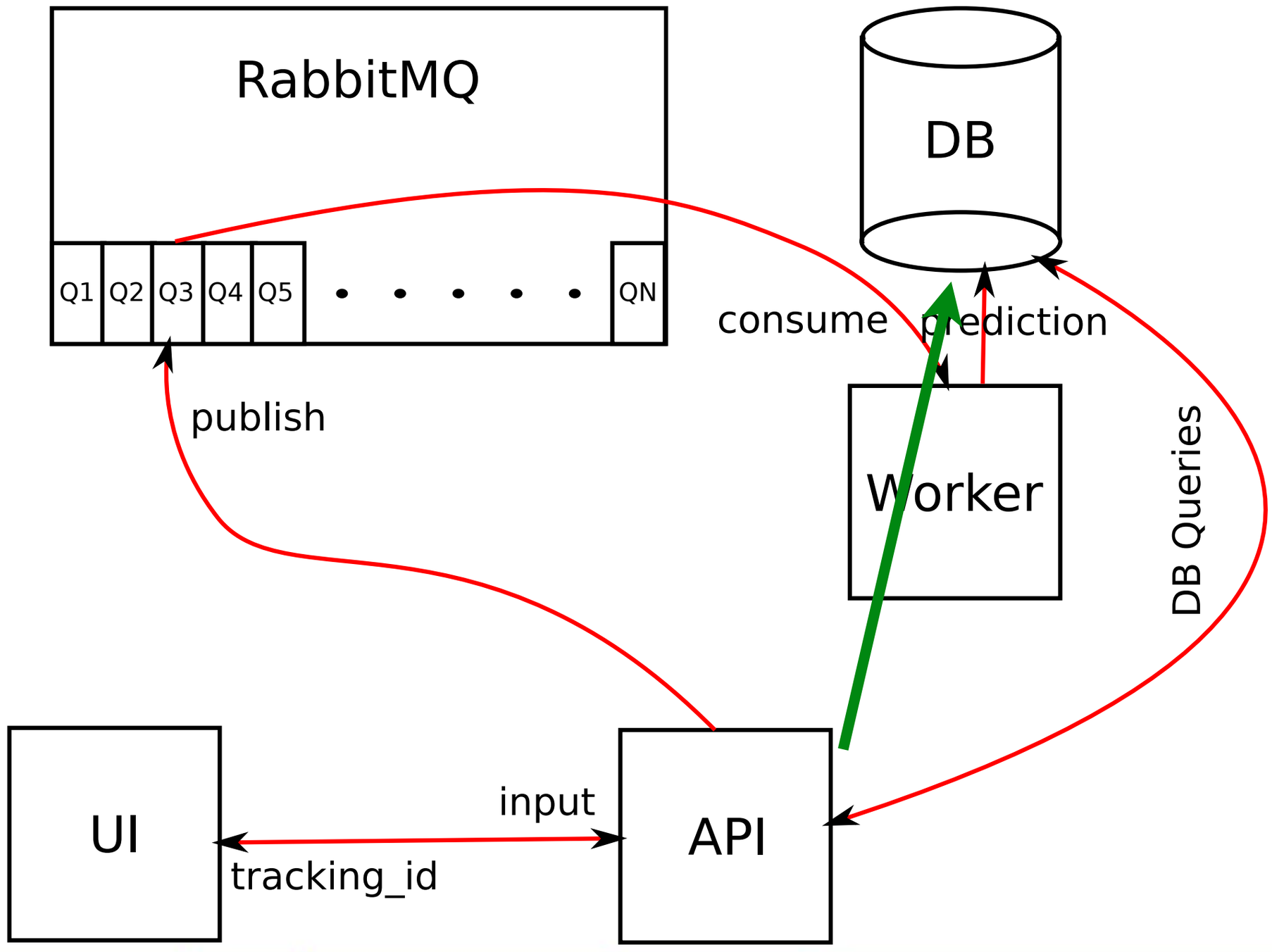
The input comes from the UI where the user enters some text they wish to classify.

The Design



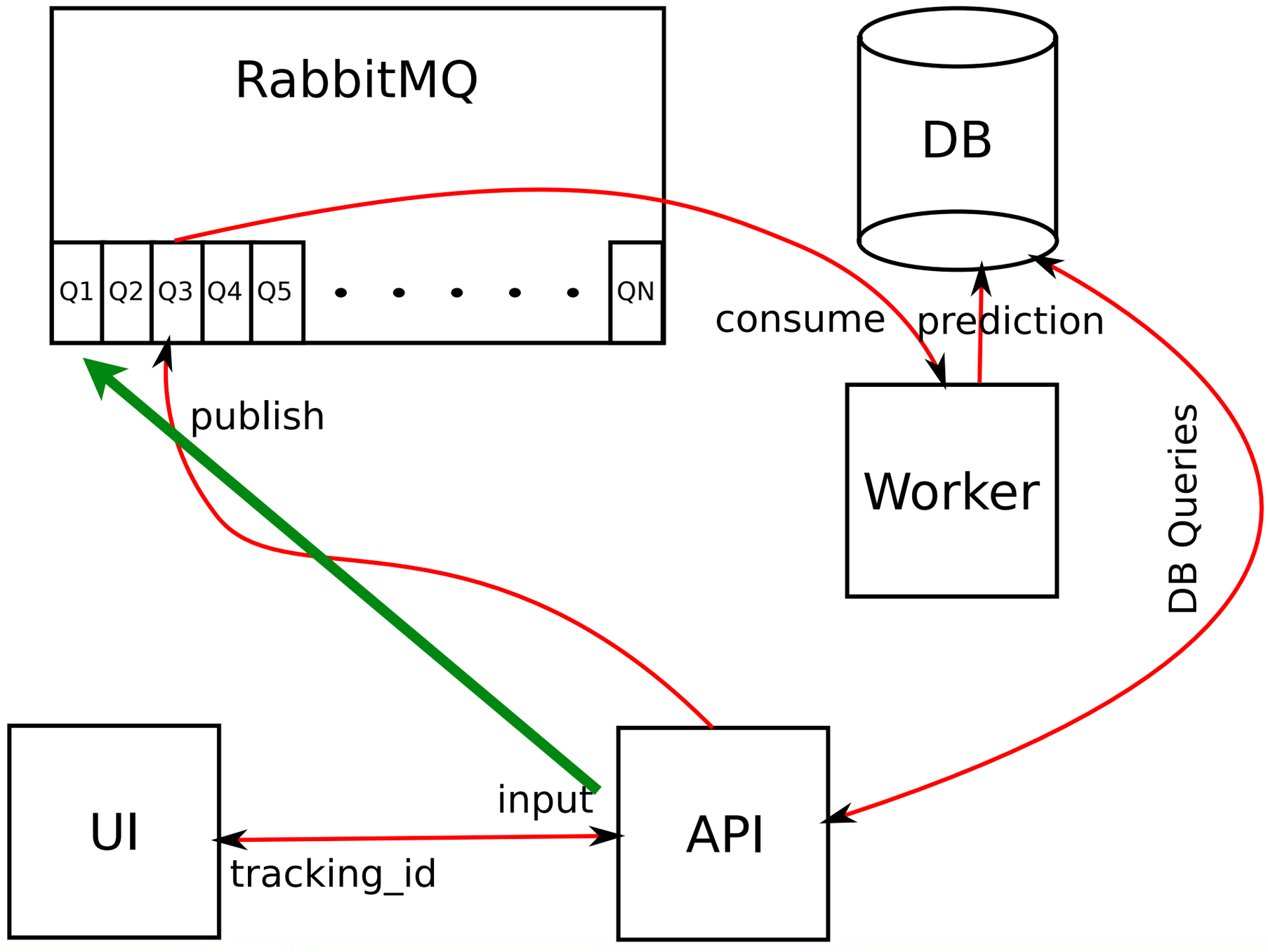
The UI hits a Flask REST API via a GET request.

The Design



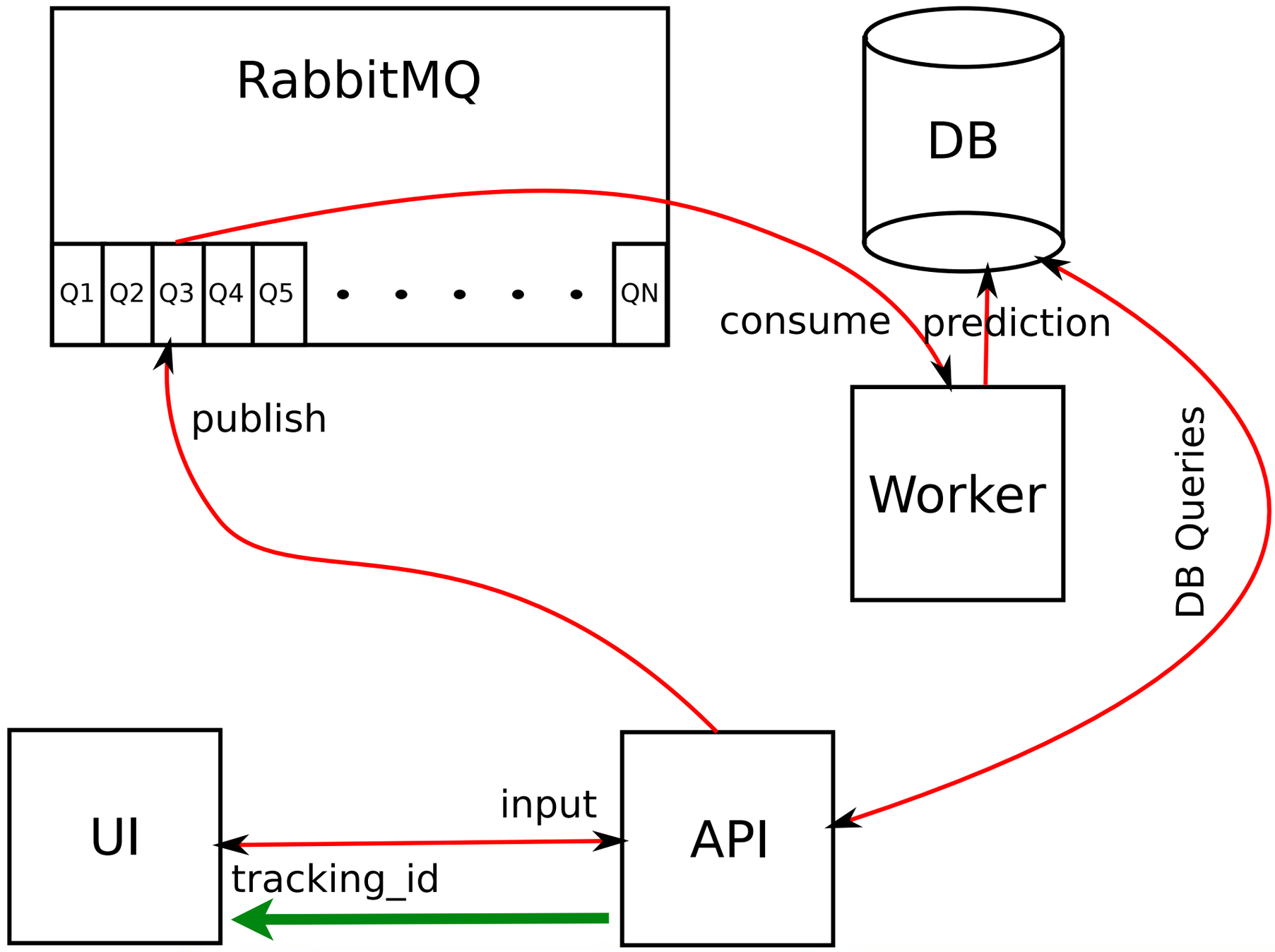
The API stores the request in the DB.

The Design



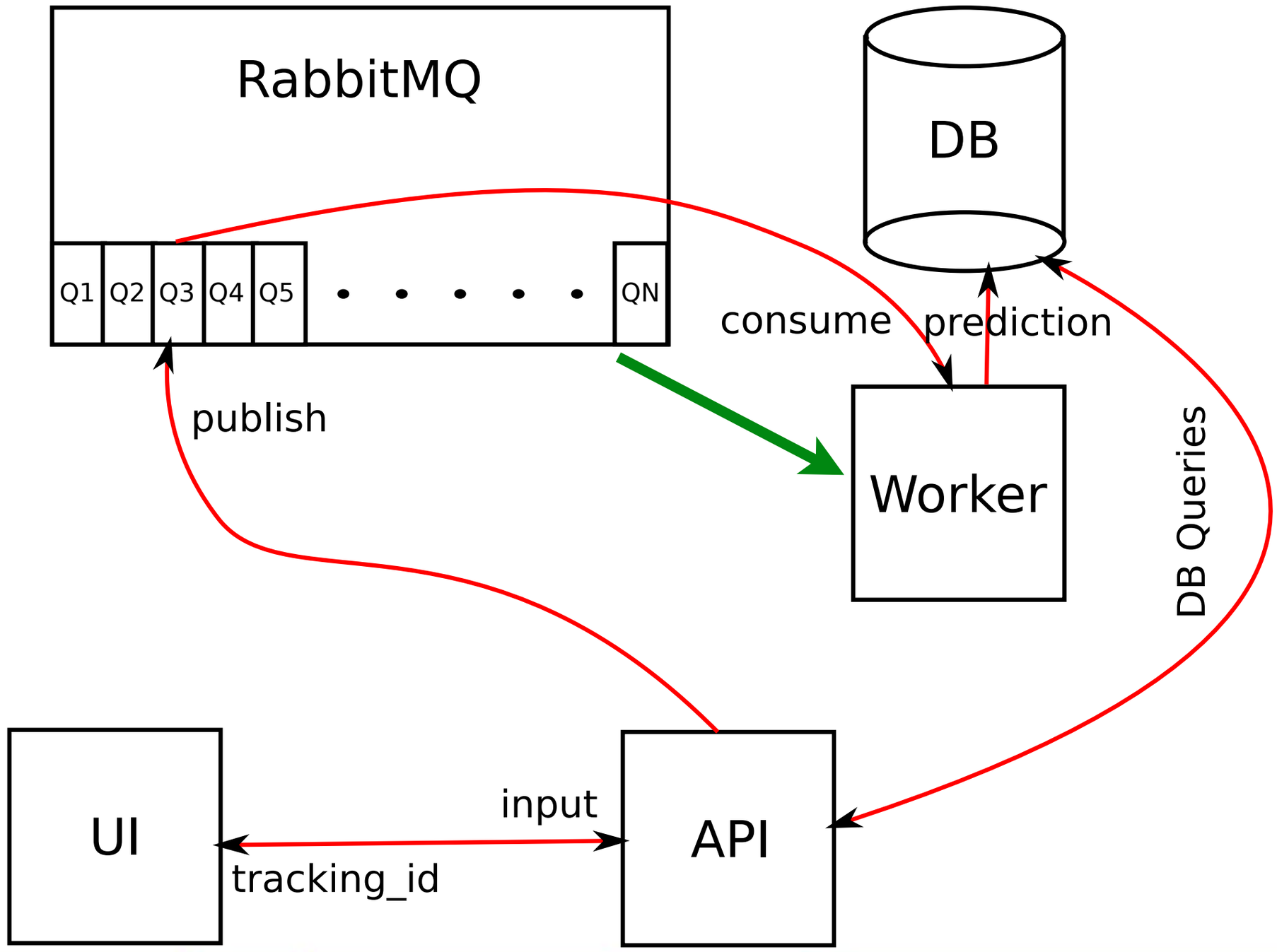
The API sends a message to RabbitMQ with the text to classify and the tracking_id for storing the resulting classification.

The Design



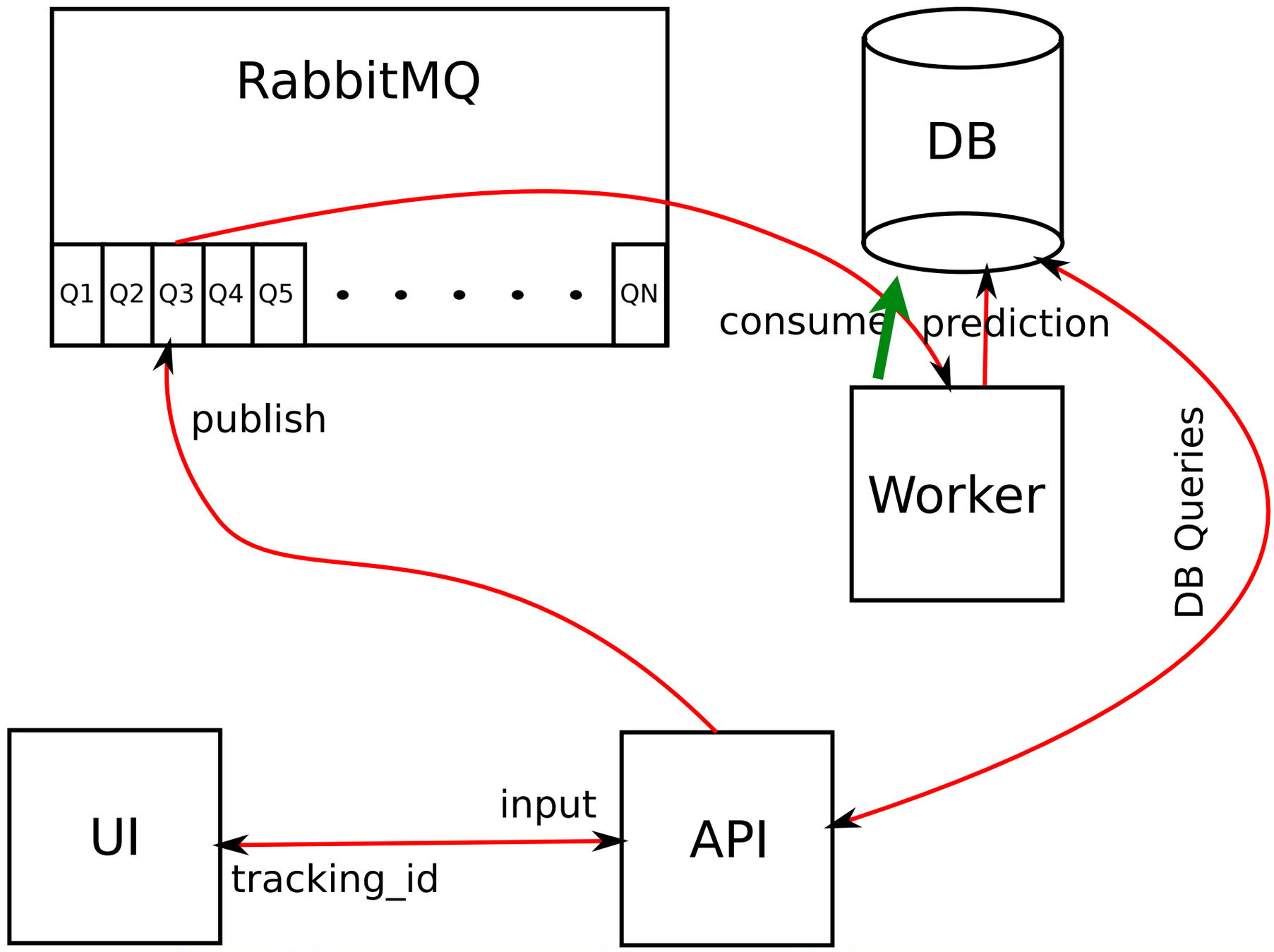
The API returns a json response to the UI with the tracking_id.

The Design



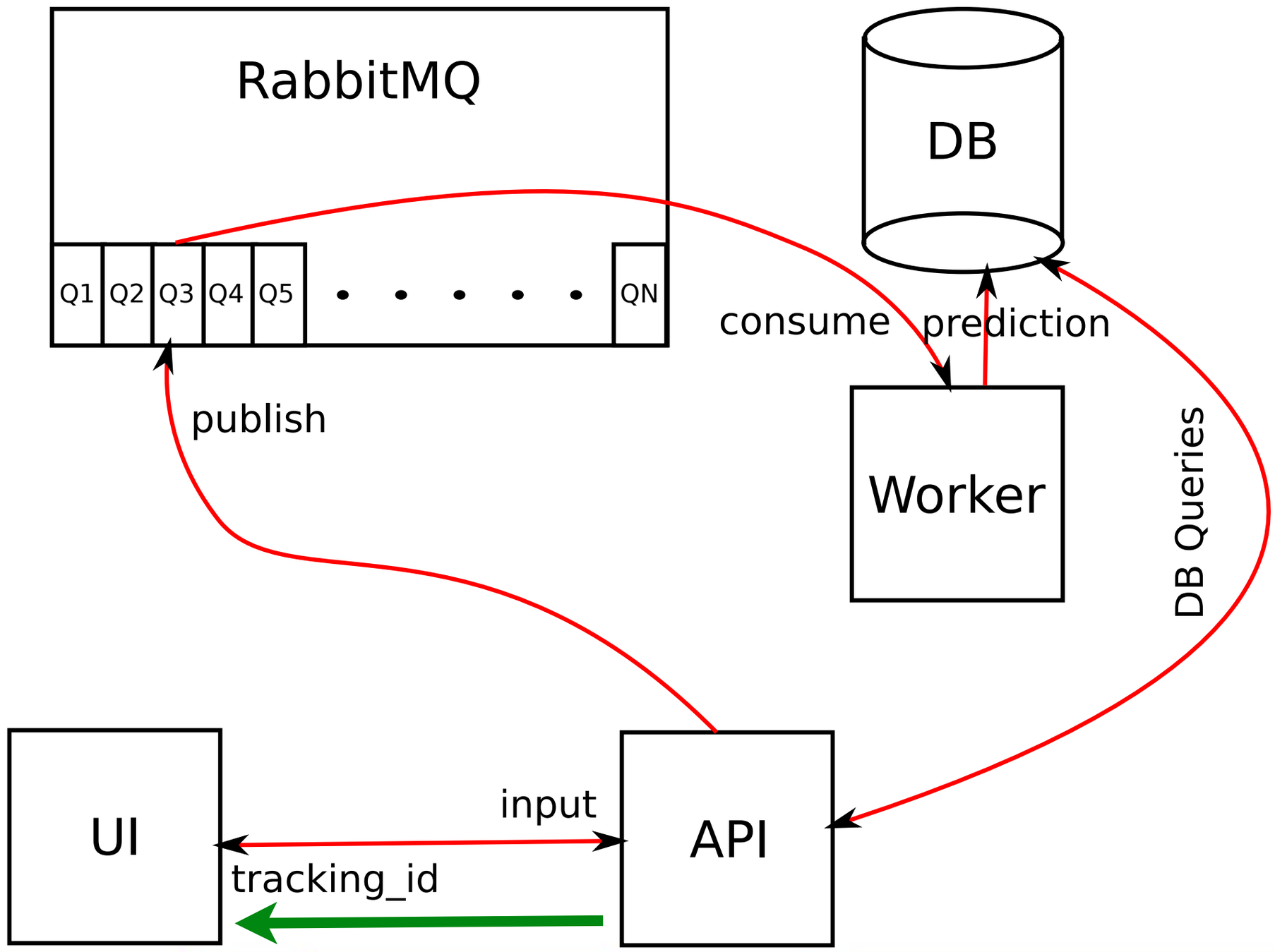
The consumer pulls the message off the queue in RabbitMQ.

The Design



The consumer calls predict on the classifier with the text as input. The classifier returns a prediction. The consumer updates the database with the result.

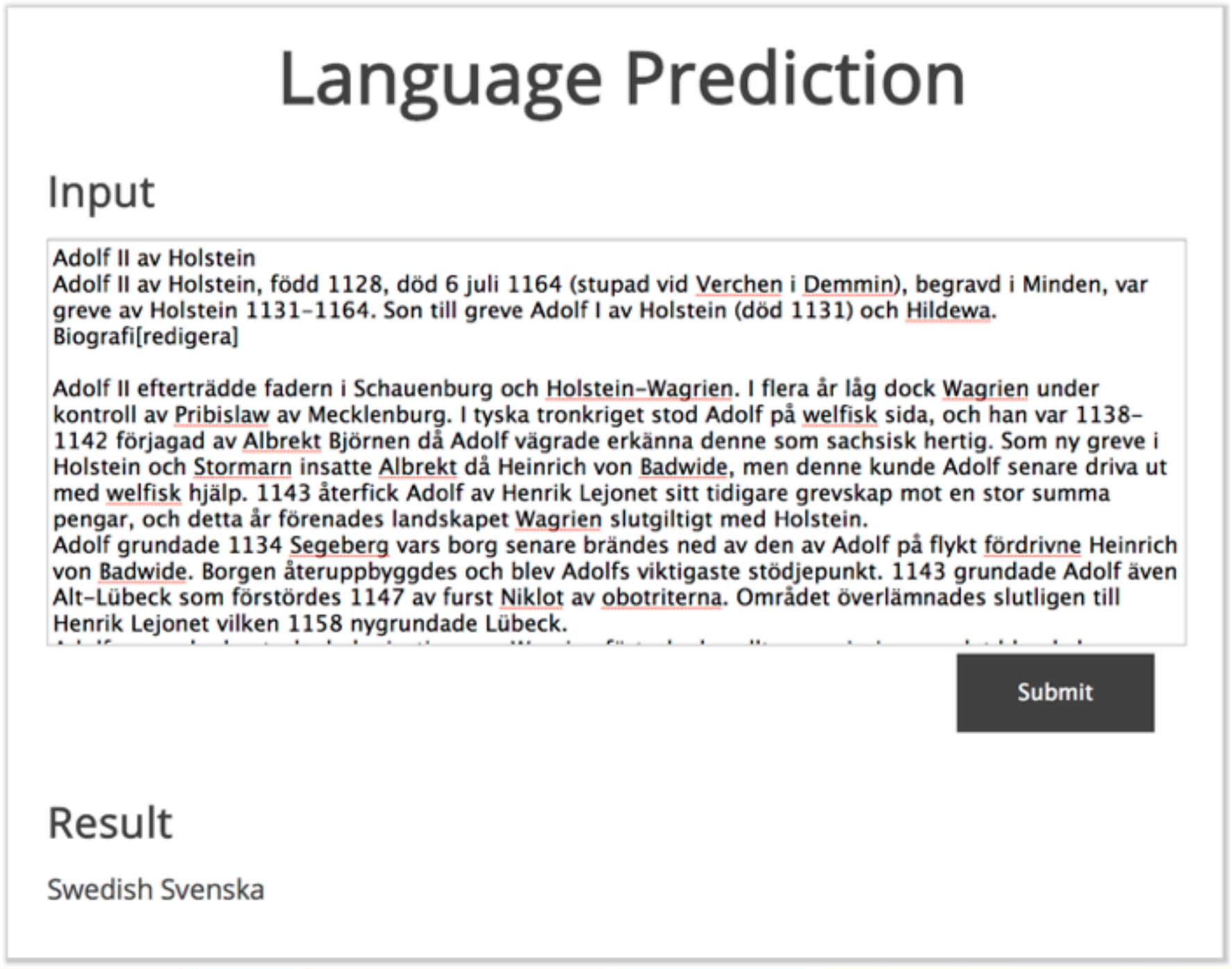
The Design



The UI displays the result.

There isn’t one right way to design a solution like this. For example instead of polling you could utilize websockets. Then you could have the consumer respond directly back to the client, possibly via RabbitMQ.

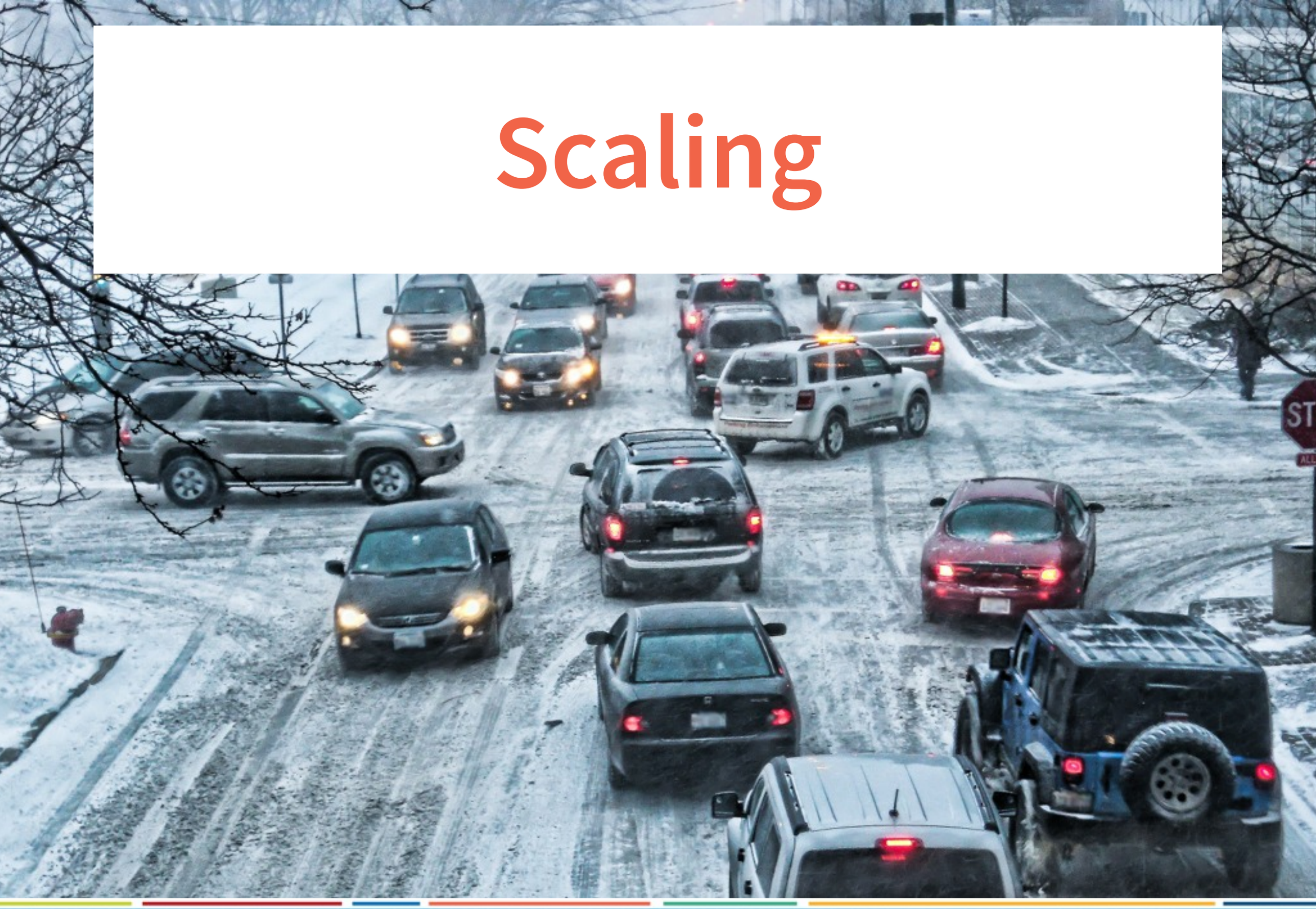
Demo Time!



TODO: Save samples offline in case no internet

Alright so let's see what this all looks like in action!

~~Danish & Norwegian~~ use the same character set, so I'll use them for my demo.
Scots & English



Besides the basic design concerns I’ve already covered, there’s a few more things worth mentioning.

The worst thing that can happen when you're processing data asynchronously is for your queue to backup. Backups will result in longer processing times, and if unbounded, you'll could crash RabbitMQ. The easiest way to scale your consumers is to start another instance. Using this strategy, processing should scale roughly linearly. In my experience, you can easily handle thousands of messages a second this way.

Also if you want to prevent backups, consider making use of TTL queues to detect and prevent backups.

Realtime vs Batch

Another way to scale your consumer is to convert it to processing requests in batches. Many of the algorithms in scikit-learn scale super-linearly when you pass multiple samples to the predict method. The downside of this is that you will no longer be able to process results in realtime. However, if you're restricted on resources (memory & cpu), this might be a worthwhile alternative.

Monitoring

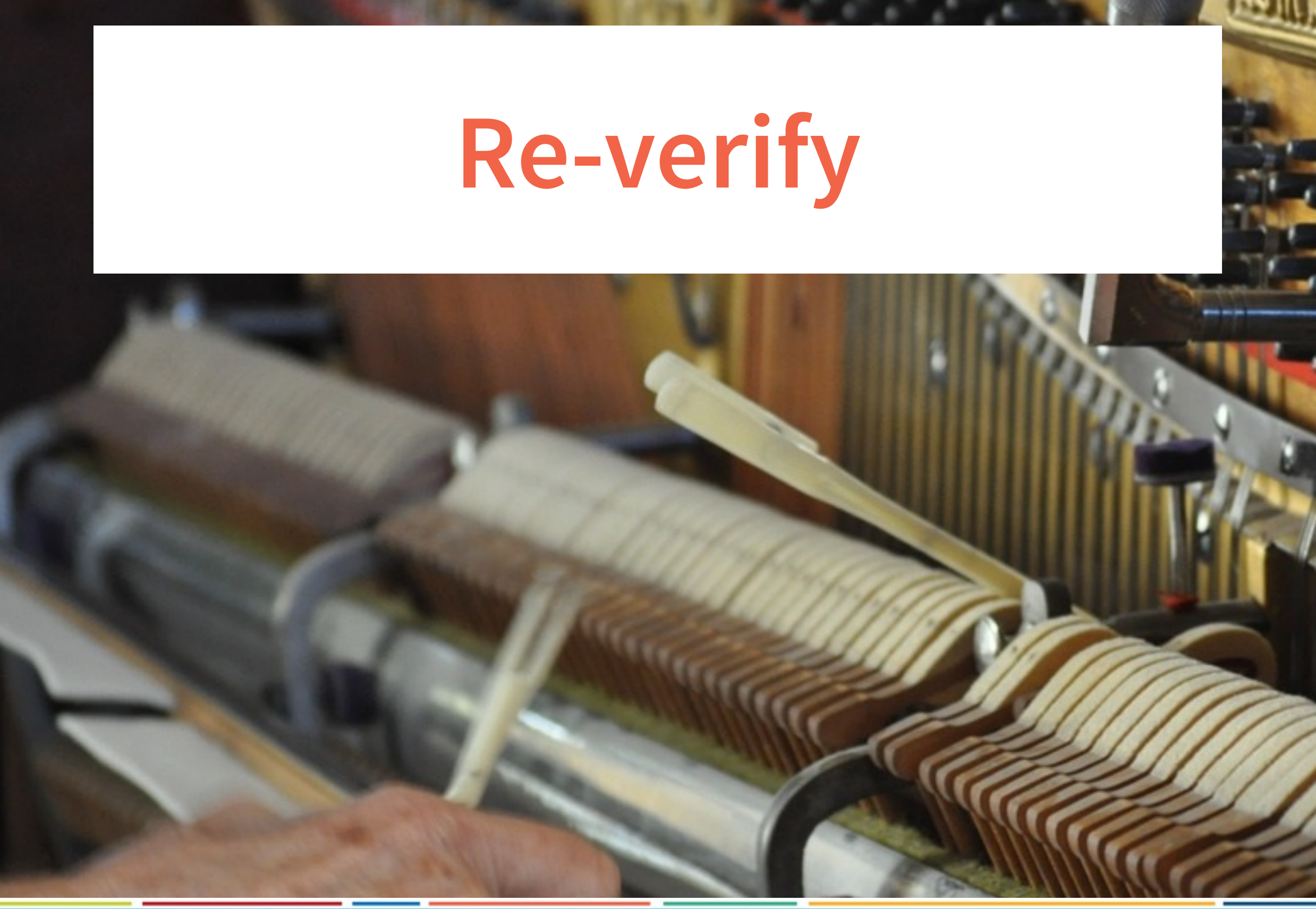
Keep an eye on your queue sizes, alert when they backup. Scale as needed (possibly automatically).

TODO: Talk about TTL queues?

TODO: Talk about other things you can monitor to detect problems (such as # of consumers...)

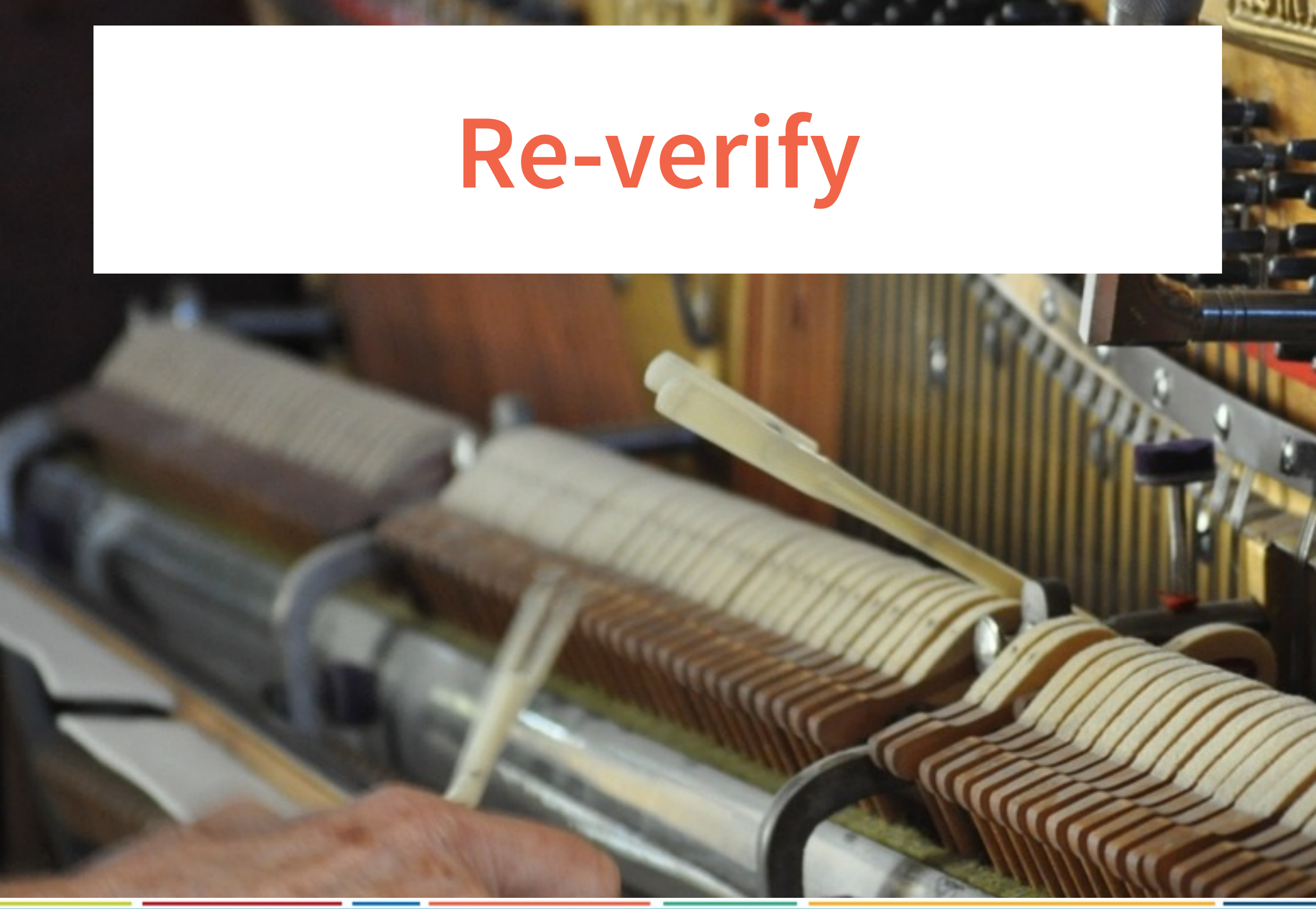
Load

Understand your load requirements. Load test end-to-end to verify you can handle the expected load.



A predictive algorithm is like a piano, if it’s not in tune, it’s going to suck. So how do you make sure your predictive algorithm is performing well with new data?

One solution is to periodically re-verify your algorithm using new data. Depending on your usecase, this might not be easy to accomplish. In the case of the language classifier, how would we collect new labeled data? Maybe we could add a button that allows users to report problems? Then we could go through some of these results, label them manually, and then retrain the algorithm against them.



Another technique, I learned from Oliver Grisel, is to train a new classifier to predict “new” vs “old” data. Then if the accuracy of this classifier is high (more than 55-60%), the distribution of your new data has drifted from your training set. This means it’s probably time to retrain your classifier with new data. You’ll still need to find a way to label your new data though.

No matter how you handle tuning your algorithm, make sure you version control your classifier. Keep detailed changelogs with performance metrics/characteristics.

Thank You!

API & Consumer: Kelly O’Brien (linkedin.com/in/kellyobie)

UI: Matt Parke (ordinaryrobot.com)

Classifier: Michael Becker (github.com/mdbecker)

Images: Wikipedia; flickr.com users: kohsa, spenceyc, statefarm, klara, rh2ox, nasahqphoto, frickfrack

I’d like to thank Kelly O’Brien and Matt Parke for helping me with the back-end and front-end for the demo.

My Info

Twitter: @beckerfuffle

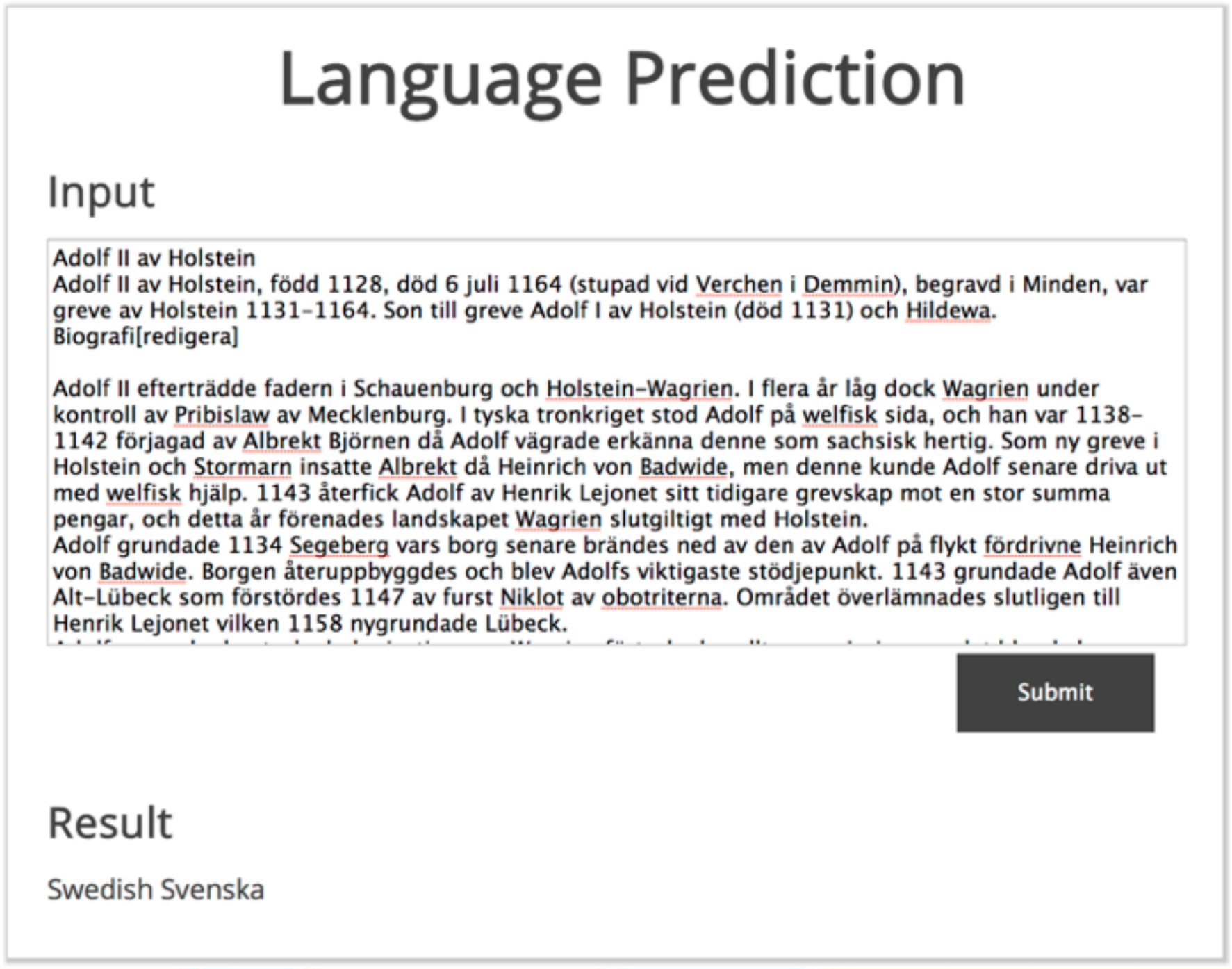
Blog: beckerfuffle.com

These slides and more @ github.com/mdbecker

Thanks everybody for coming to my talk! My name is Michael Becker, I work for the Data Analysis and Management Ninjas at AWeber. AWeber is an email service provider with over 120 thousand customers. I'm also the founder of the DataPhilly meetup with over 600 members. If you're not already a member, well shame on you!

You can find me online @beckerfuffle on Twitter. My website is beckerfuffle.com. Materials from this talk can be found on my github.

Demo time!

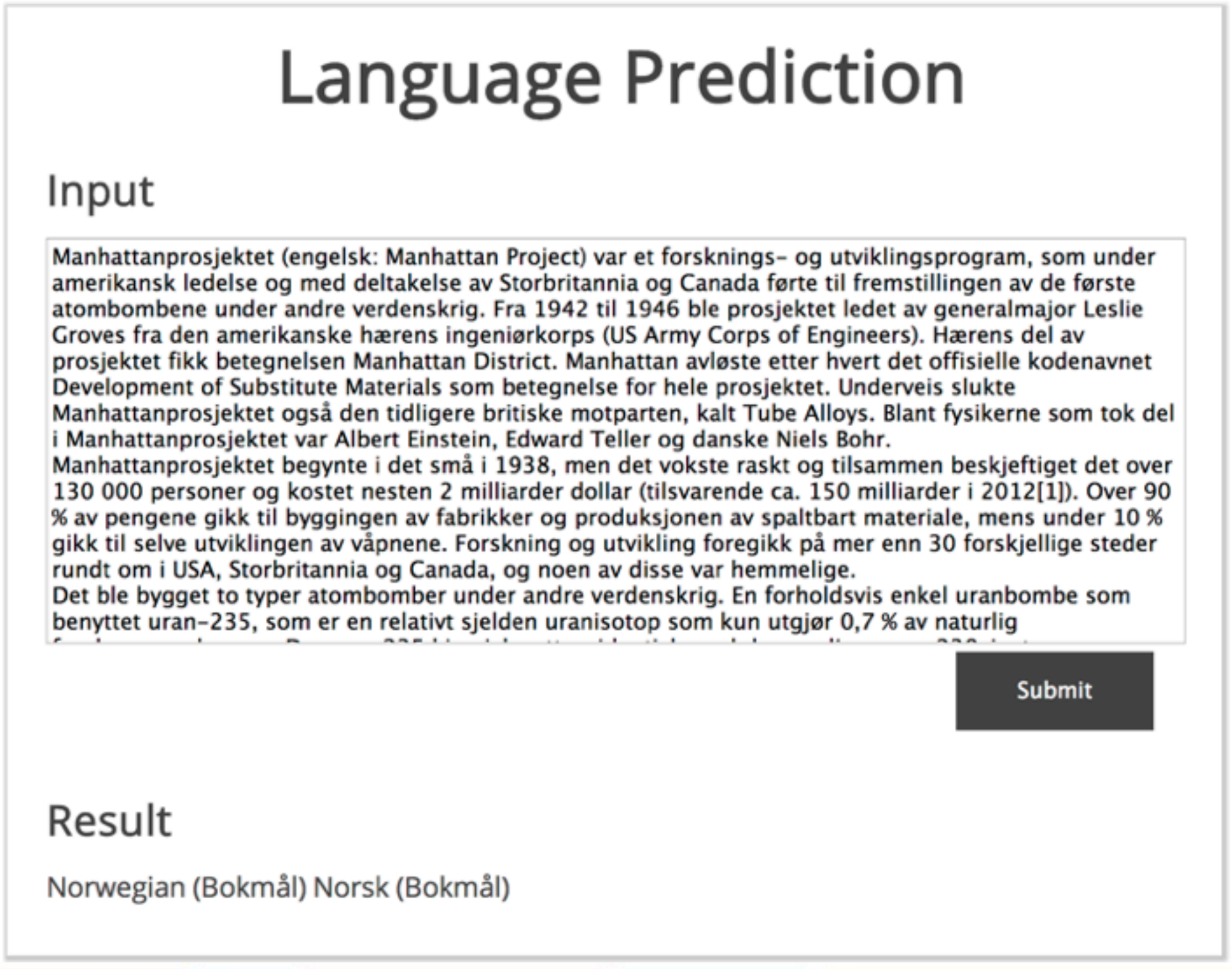


TODO: Save samples offline in case no internet

Alright so let's see what this all looks like in action!

~~Danish & Norwegian~~ use the same character set, so I'll use them for my demo.
Scots & English

Demo time!

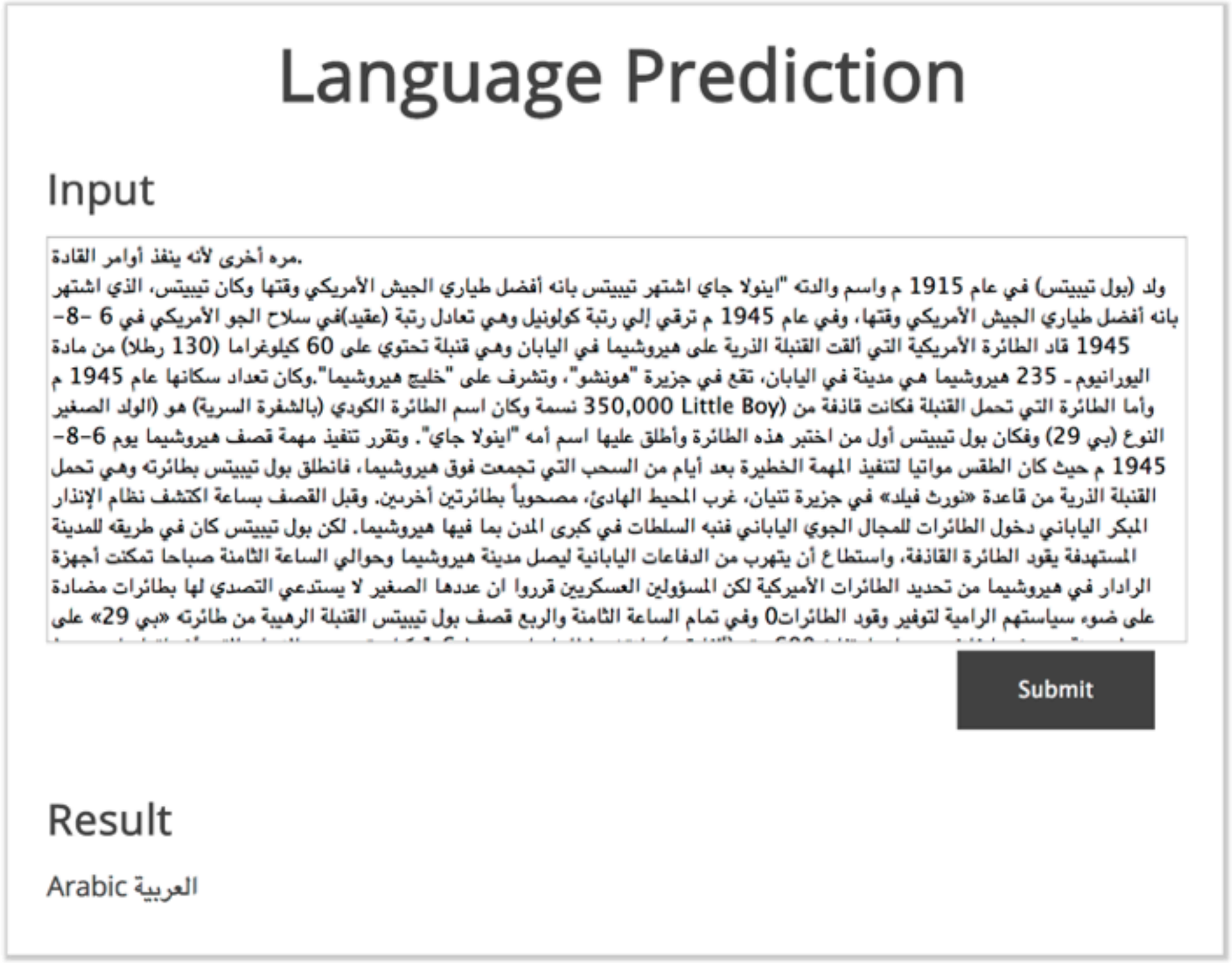


TODO: Save samples offline in case no internet

Alright so let's see what this all looks like in action!

~~Danish & Norwegian~~ use the same character set, so I'll use them for my demo.
Scots & English

Demo time!



TODO: Save samples offline in case no internet

Alright so let's see what this all looks like in action!

~~Danish & Norwegian~~ use the same character set, so I'll use them for my demo.
Scots & English