Realtime Predictive Analytics

Using scikit-learn and RabbitMQ

Michael Becker

PyData Boston 2013

Who is this guy?

Software Engineer @ AWeber
Founder of the DataPhilly Meetup group
@beckerfuffle
beckerfuffle.com

These slides and more @ github.com/mdbecker

Good morning everyone, My name is Michael Becker, I work in the Data Analysis and Management team at AWeber, an email marketing company in Chalfont, PA
I'm also the founder of the DataPhilly Meetup group
You can find me online @beckerfuffle on Twitter. At beckerfuffle.com, and I'm also mdbecker on github. I'll be posting the materials for this talk on my github.

What I'll cover

- Model Distribution

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

In this talk I’ll cover:
How to distribute your model

What I'll cover

- Model Distribution
- Data flow

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

What I'll cover

- Model Distribution
- Data flow
- RabbitMQ

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

What I'll cover

- Model Distribution
- Data flow
- RabbitMQ
- Demo

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

What I'll cover

- Model Distribution
- Data flow
- RabbitMQ
- Demo
- Scalability

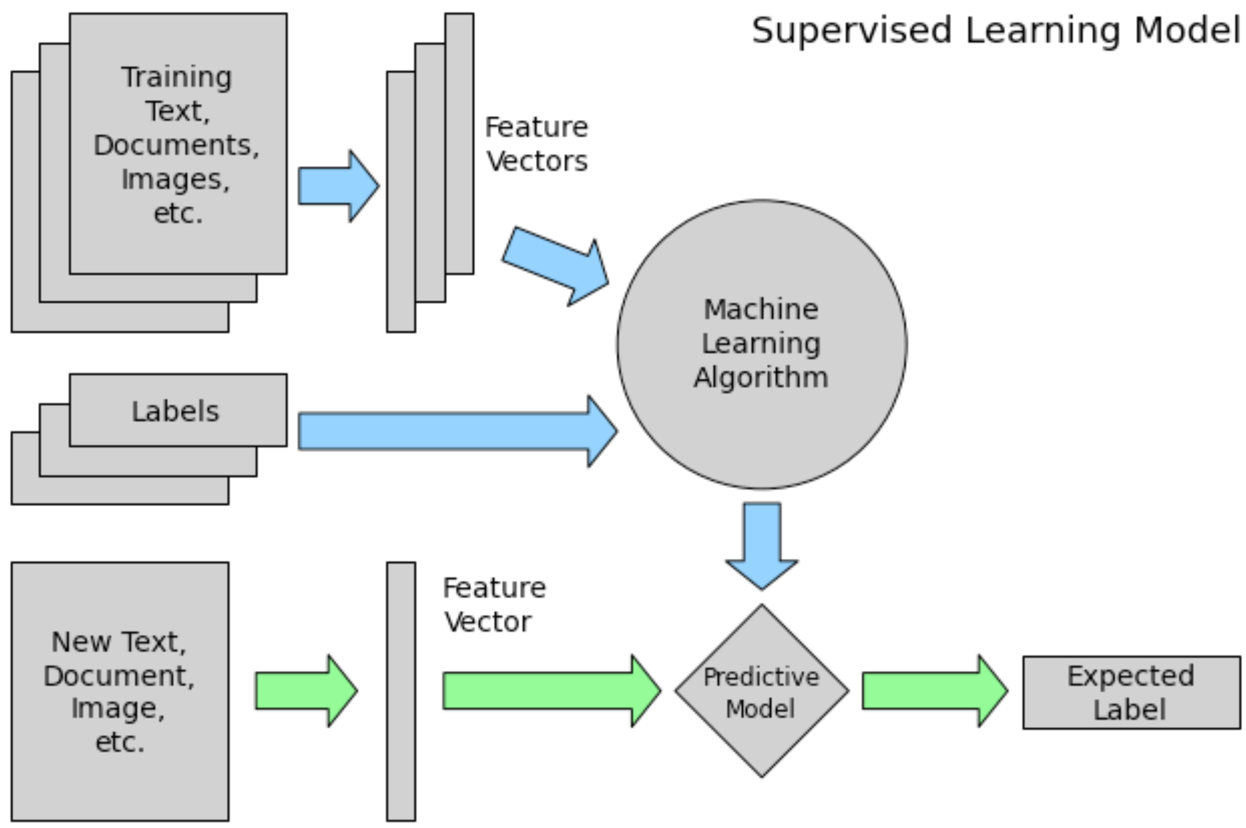
I'll discuss how to scale your model

What I'll cover

- Model Distribution
- Data flow
- RabbitMQ
- Demo
- Scalability
- Other considerations

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

Supervised Learning



To start off, let's recap what the supervised model training process looks like.

1) You have your training data and labels
2) You vectorize your data, you train your machine learning algorithm.
3) ???
4) Make predictions with new data
5) Profit

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 (Bokmål) 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 case I'm going to talk about one of the first models I created. A model that predicts the language of input text. To create this model, I used 38 of the top Wikipedias based on number of articles. I then dumped several of the most popular articles as defined by their number of hits.

The model

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

```
vect = TfidfVectorizer(analyzer='char', ngram_range=(2, 3), norm='l2', use_idf=True)
clf = LinearSVC(loss='l2', C=1, dual=True)
text_clf = Pipeline([
    ('vect', vect),
    ('clf', clf),
])
model = text_clf.fit(X_train, y_train)
```


I converted the wiki markup to plain text. I trained a LinearSVC (Support Vector Classifier) model using a bi/trigram (n-gram) approach I had read worked well for language classification. This approach involves counting all combinations of 2 (bigram) or 3 (trigram) character sequences in your dataset. I tested the model and I was seeing ~99% accuracy.

The model

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

```
vect = TfidfVectorizer(analyzer='char', ngram_range=(2, 3), norm='l2', use_idf=True)
clf = LinearSVC(loss='l2', C=1, dual=True)
text_clf = Pipeline([
    ('vect', vect),
    ('clf', clf),
])
model = text_clf.fit(X_train, y_train)
```

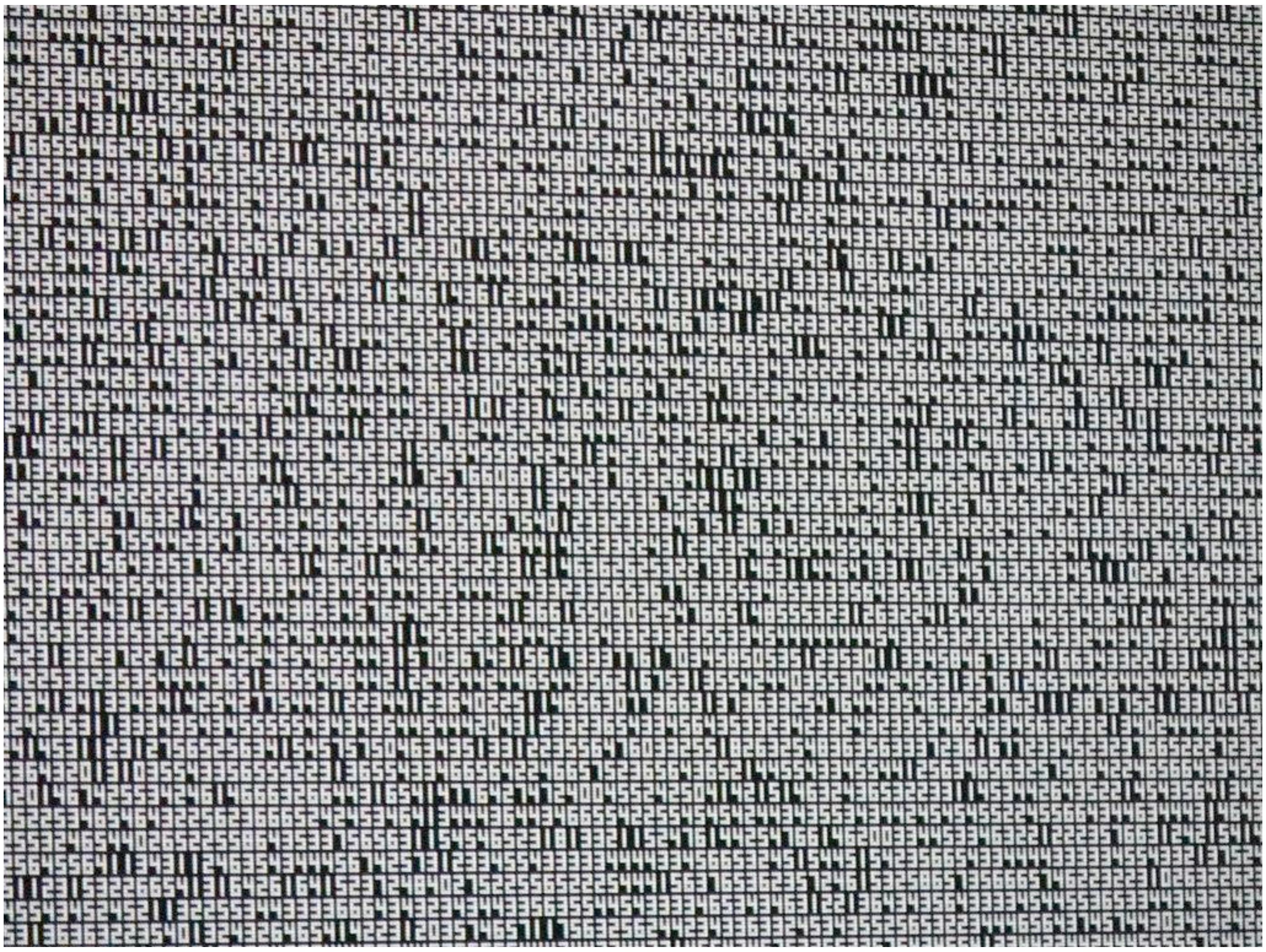

Here I've defined a pipeline combining a text feature extractor with a simple classifier. A pipeline is a utility used to build a composite classifier.
To extract features, I'm using a TfidfVectorizer. The vectorizer first counts the number of occurrences of each n-gram in each document to "vectorize the text." It then applies the TF-IDF (term frequency–inverse document frequency) algorithm. TF-IDF reflects how important a word is to a 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. So for example an common word like "the" would get down weighted compared to a less common word like "automobile."

Distributing the model

So the first thing you might ask yourself after you've trained your awesome model is "now what?"

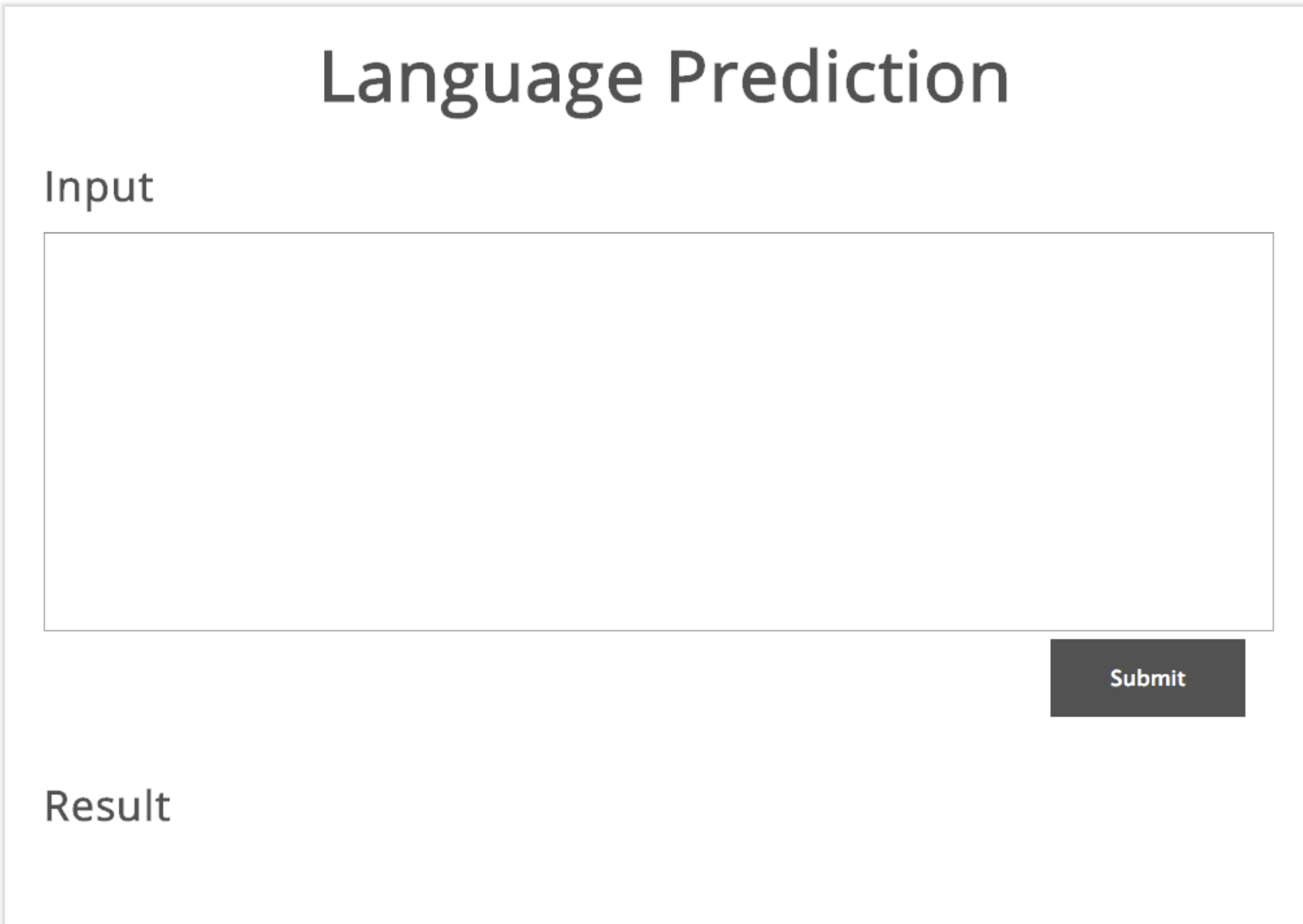
So one of the first problems you'll want to solve is how to distribute your model? The easiest thing to do this is to pickle (serialize) the model to disk and distribute it as part of your application. You can also store it in a database such as GridFS or Amazon S3. In the case of my model, it took up roughly 400MB in memory. This is pretty big, but easily storable on disk (and more importantly in memory).

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.

Message loss

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.

Enter RabbitMQ

Reliability

Flexible Routing

Clustering

RabbitMQ

HA Queues

Many clients

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.

AMQP

Reliability

Flexible Routing

Clustering

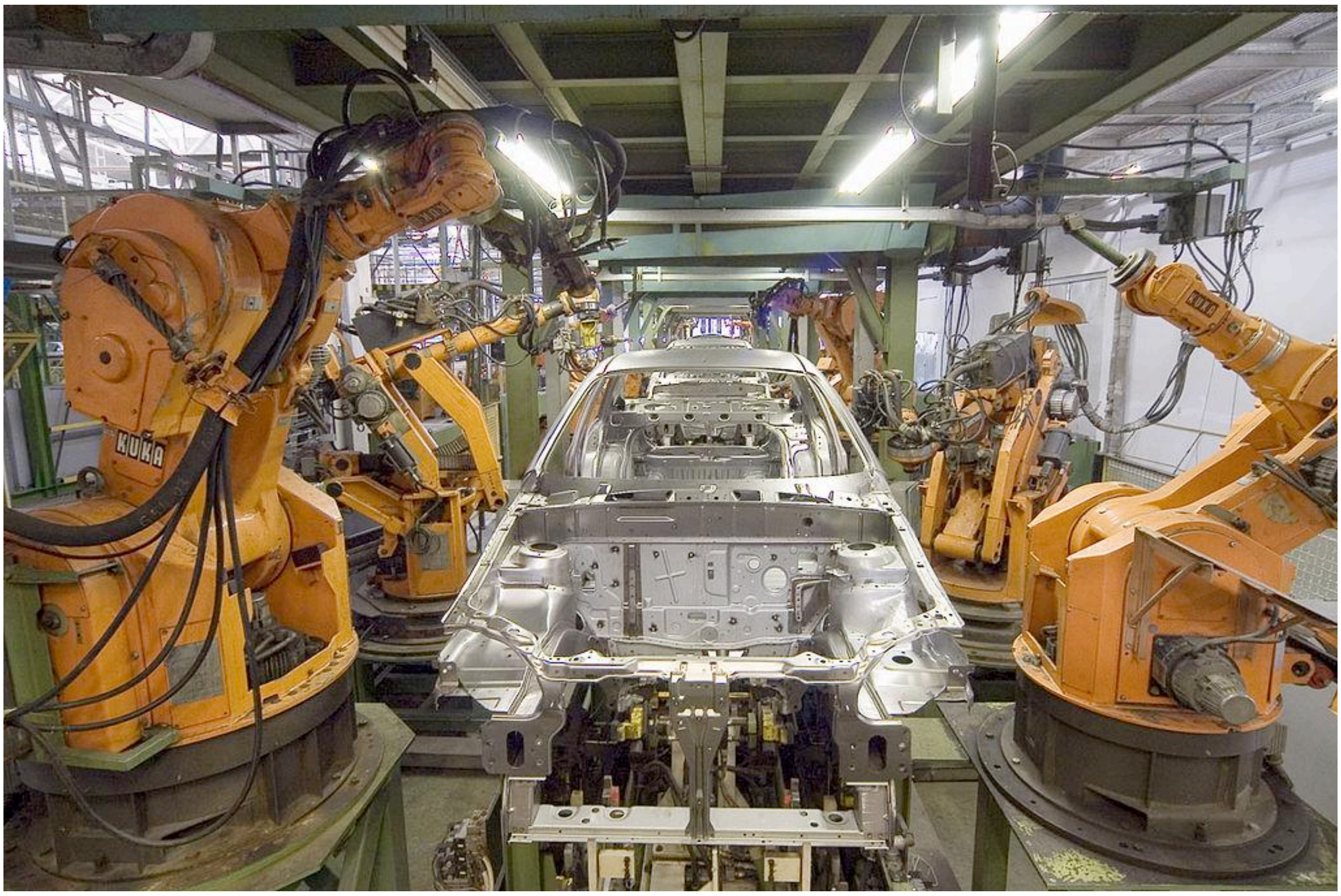
RabbitMQ

HA Queues

Many clients

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



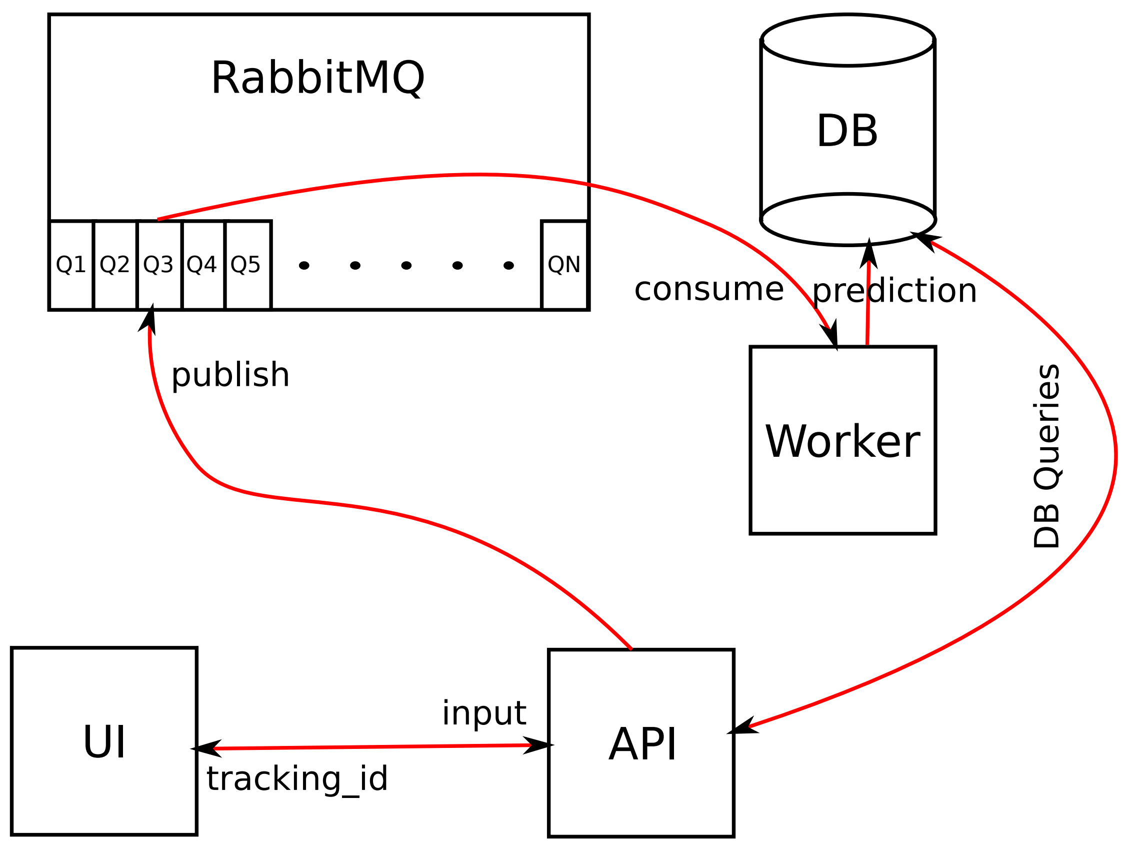
Once you hook your data input source into RabbitMQ and start publishing data, all you need to do is put your model in a persistent worker and start consuming input.

The worker

```python
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 worker that unpickles the classifier and subscribes to an input queue. It then runs an event loop (main) that pulls new messages as they become available and passes them to process_event. 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.

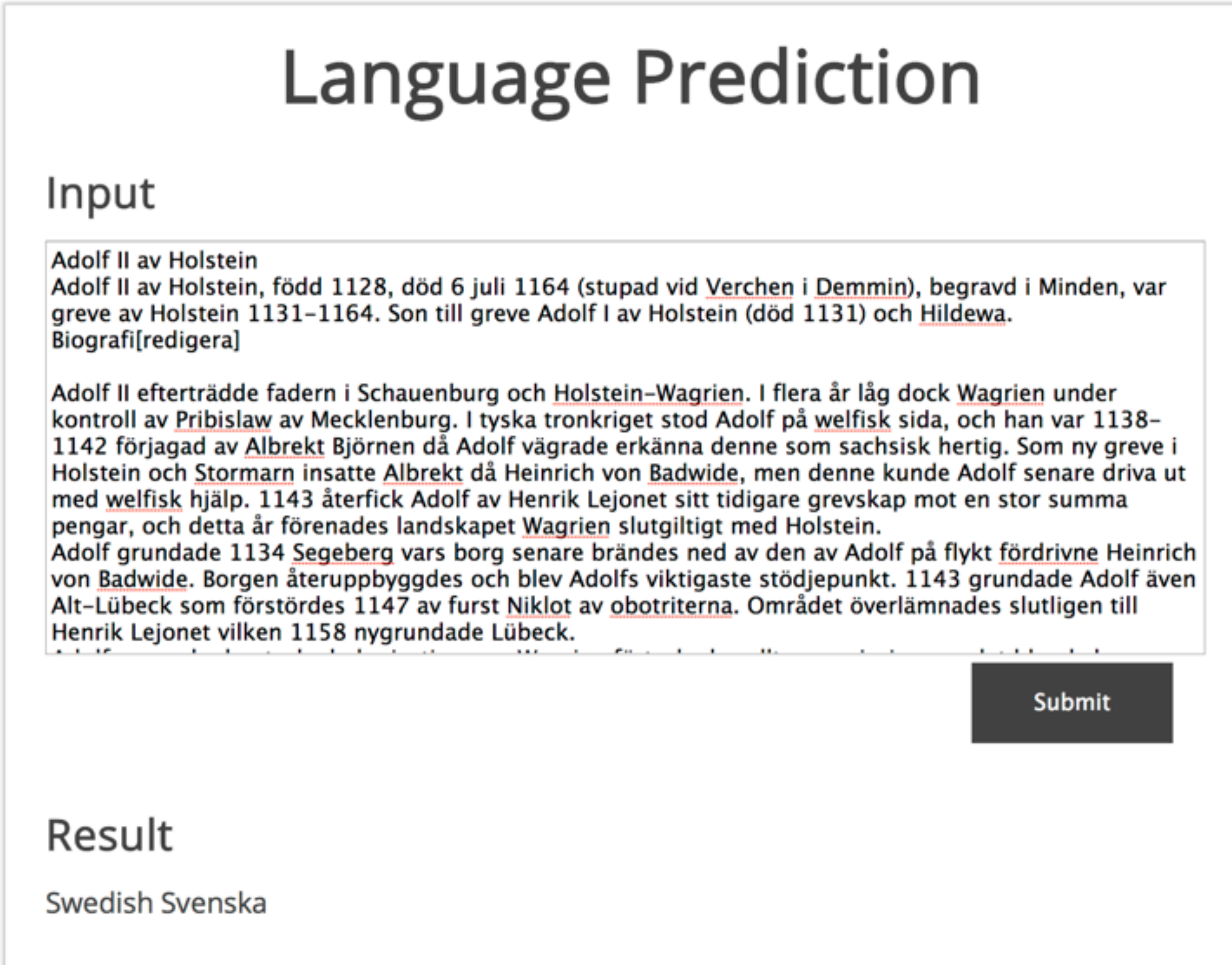
The design



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

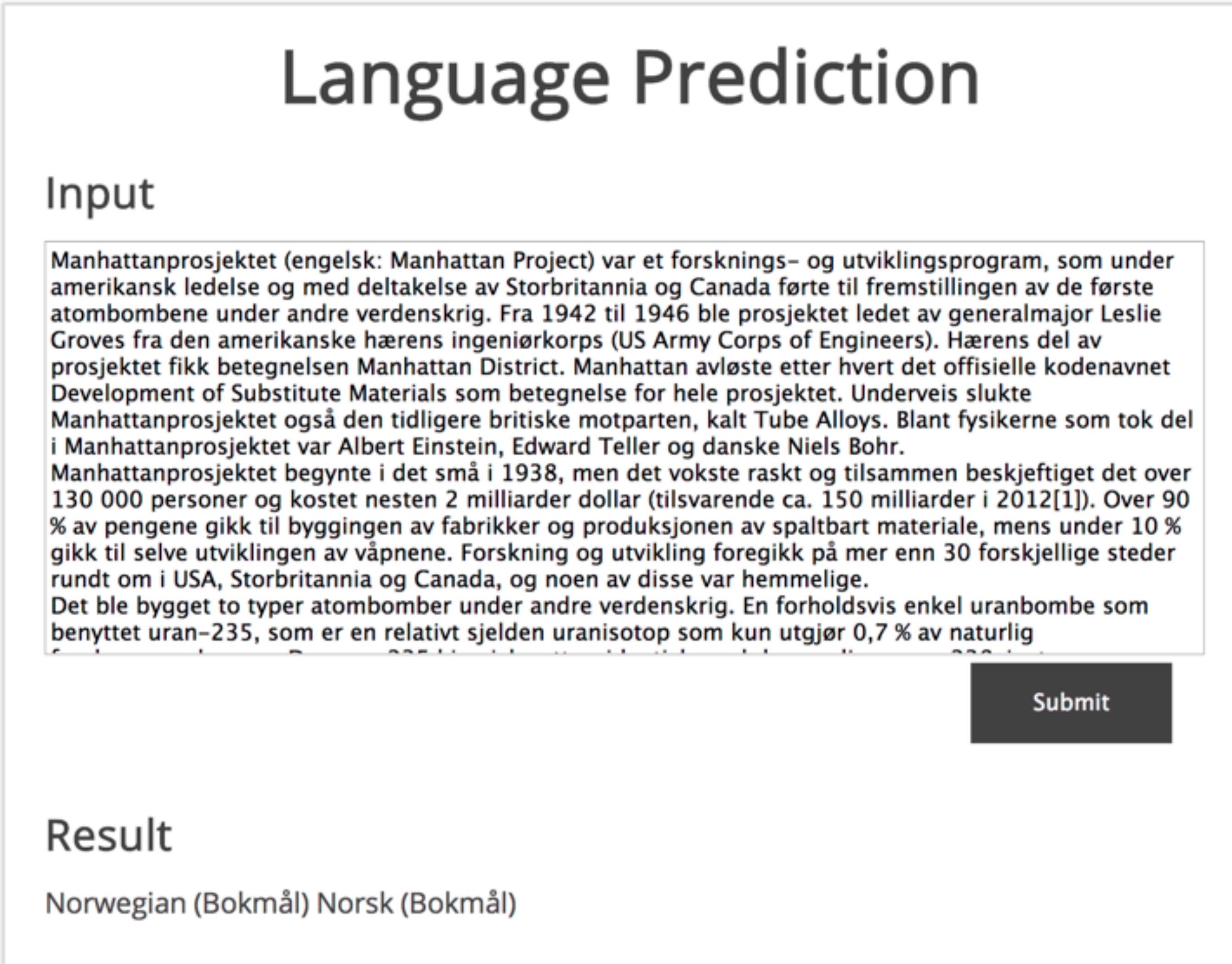
The input comes from the UI where the user enters some text they wish to classify. The UI hits a Flask REST API via a GET request. The API stores the request in the DB. The API sends a message to RabbitMQ with the text to classify and the tracking_id for storing the resulting classification. The API returns a json response to the UI with the tracking_id. The worker pulls the message off the queue in RabbitMQ. The worker calls predict on the classifier with the text as input. The classifier returns a prediction. The worker updates the database with the result. The UI displays the result.

Demo time!



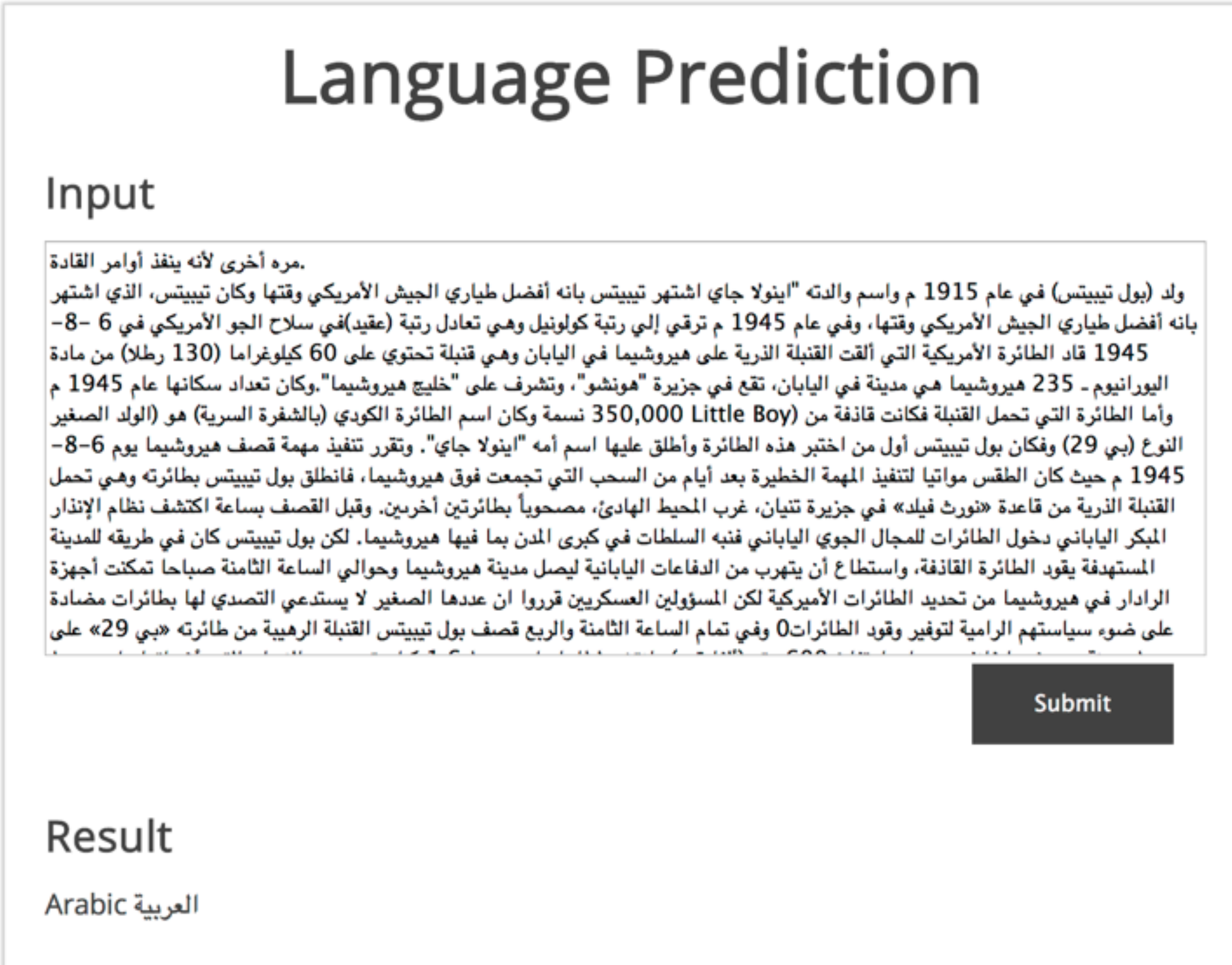
Alright so let's see what this all looks like in action!

Demo time!



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

Demo time!



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

Scaling

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 likely crash RabbitMQ. The easiest way to scale your workers 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.

Realtime vs batch

Another way to scale your worker is to convert it to processing requests in batches. Many of the algorithms 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).

Load

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

Verify

Periodically re-verify your algorithm using new data. Build in a feedback loop so that you can collect new labeled samples to verify the performance

Version control your classifier. Keep detailed changelogs and performance metrics/characteristics.

Thank you

API & Worker: Kelly O'Brien (linkedin.com/in/kellyobie)
UI: Matt Parke (ordinaryrobot.com)
Classifier: Michael Becker (github.com/mdbecker)
Images: Wikipedia

I'd like to thank Kelly O'brien and Matt Parke for helping me with the front-end and back-end for the demo. Without them things would be a lot less exciting!

My info

Tweet me @beckerfuffle
Find me at beckerfuffle.com

These slides and more @ github.com/mdbecker

You can find me online @beckerfuffle on Twitter. At beckerfuffle.com, and I'm also mdbecker on github. I'll be posting the materials for this talk on my github.