

OPEN
DATA
SCIENCE
CONFERENCE



San Francisco | Oct. 28 - Nov. 1, 2019

pomegranate

fast and flexible probabilistic modelling in python

Jacob Schreiber

Paul G. Allen School of Computer Science & Engineering
University of Washington



[jmschreiber91](#)



[@jmschrei](#)



[@jmschreiber91](#)



Acknowledgements



UNIVERSITY *of* WASHINGTON

eScience Institute

ADVANCING DATA-INTENSIVE DISCOVERY IN ALL FIELDS

Inria
INVENTORS FOR THE DIGITAL WORLD

 PARIETAL





Overview

pomegranate is **flexible**, **fast**, **intuitive to use**, and can do it all **in parallel**



pomegranate supports many models

Main Models

1. Probability Distributions
2. General Mixture Models
3. Hidden Markov Models
4. Naive Bayes / Bayes' Classifiers
5. Markov Chains
6. Bayesian Networks



Supporting Models

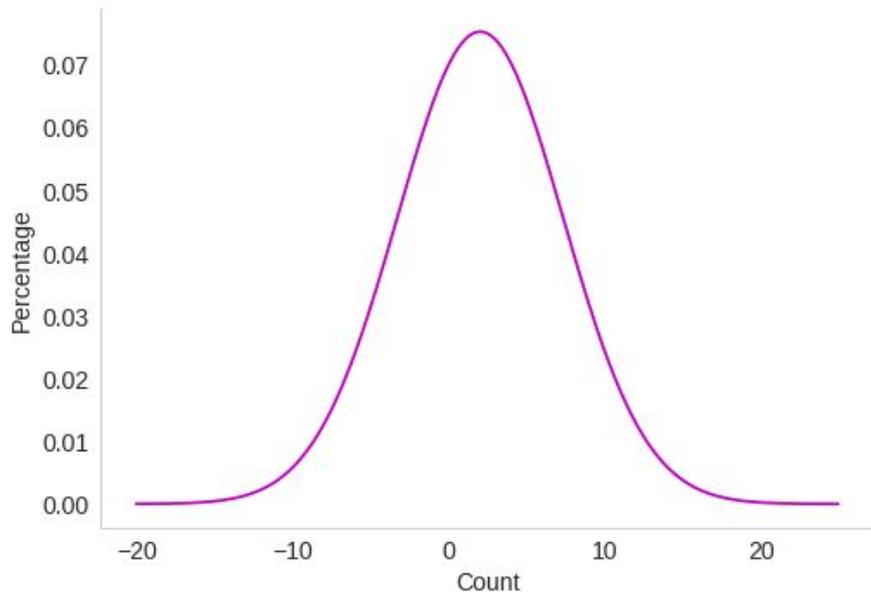
- k-means / kmeans++ / kmeans||
- Factor graphs



Models can be made in two ways...

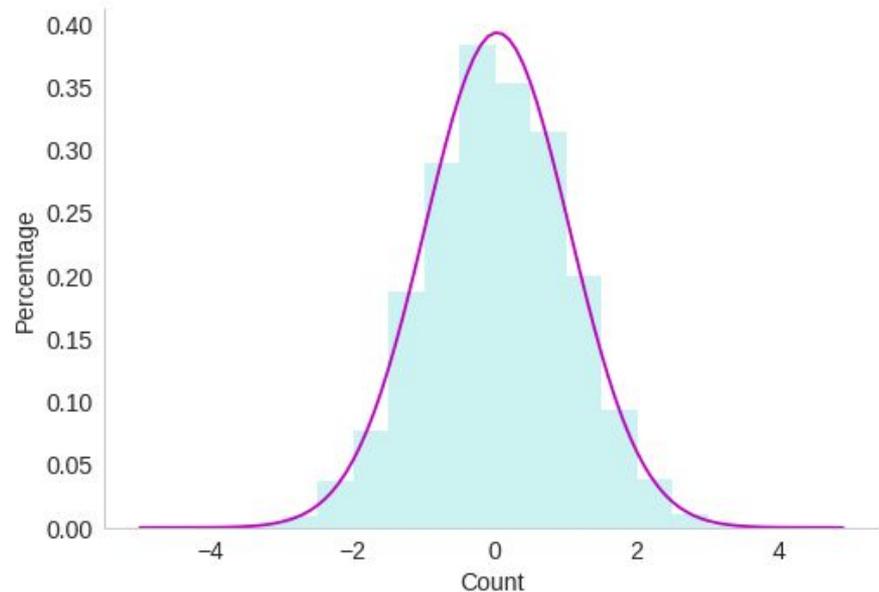
...from known values

```
d = NormalDistribution(5, 2.3)
```



...from data

```
d = NormalDistribution.from_samples(X)
```





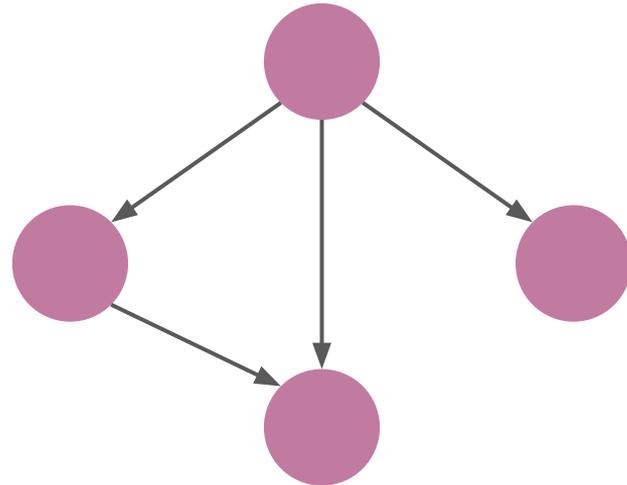
Models can be made in two ways...

...from known values

```
n1 = Node(...)  
n2 = Node(...)  
model = BayesianNetwork()  
model.add_nodes(n1, n2...)  
model.add_edges(...)
```

...from data

```
d = BayesianNetwork.from_samples(X)
```





The API is common to all models

`model.log_probability(X) / model.probability(X)`

`model.sample()`

`model.fit(X, weights, inertia)`

All models have these methods!

`model.summarize(X, weights)`

`model.from_summaries(inertia)`

`Model.from_samples(X, weights)`

`model.predict(X)`

`model.predict_proba(X)`

`model.predict_log_proba(X)`

All models composed of distributions (like GMM, HMM...) have these methods too!



Overview: model stacking in pomegranate

```
GeneralMixtureModel.from_samples(NormalDistribution, n_components=3, X=X)
```

```
GeneralMixtureModel.from_samples(ExponentialDistribution, n_components=3,  
X=X)
```

```
BayesClassifier.from_samples(MultivariateGaussianDistribution, X, y)
```

```
d1 = GeneralMixtureModel.from_samples...
```

```
d2 = GeneralMixtureModel.from_samples...
```

```
model = BayesClassifier([d1, d2])
```



pomegranate is just as fast as numpy

Fitting Multivariate Gaussian to 10,000,000 samples of 10 dimensions

```
data = numpy.random.randn(10000000, 10)

print "numpy time:"
%timeit -n 10 data.mean(axis=0), numpy.cov(data, rowvar=False, bias=True)
print "\n" "pomegranate time:"
%timeit -n 10 MultivariateGaussianDistribution.from_samples(data)
```

numpy time:
10 loops, best of 3: 3.52 s per loop

pomegranate time:
10 loops, best of 3: 2.87 s per loop



pomegranate uses additive summarization

pomegranate reduces data to sufficient statistics for updates and so only has to go datasets once (for all models).



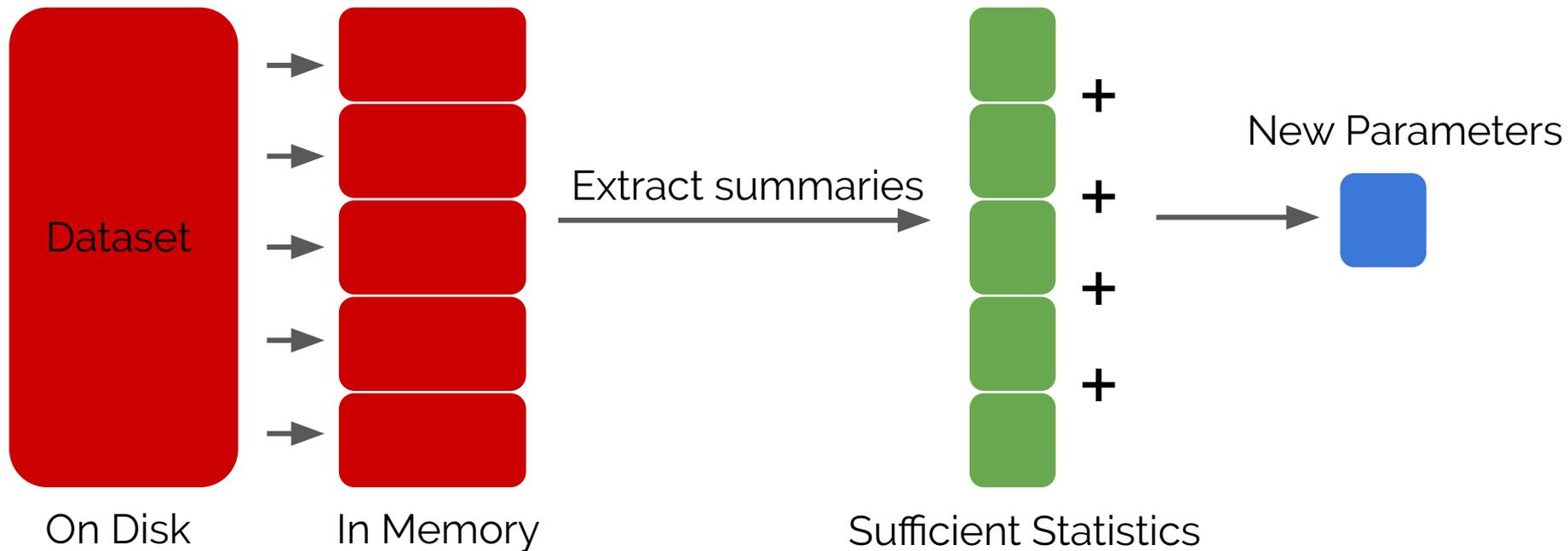
Here is an example of the Normal Distribution sufficient statistics

$$\sum_{i=1}^n w_i \quad \sum_{i=1}^n w_i x_i \quad \sum_{i=1}^n w_i x_i^2 \quad \longrightarrow \quad \begin{aligned} \mu &= \frac{\sum_{i=1}^n w_i x_i}{\sum_{i=1}^n w_i} \\ \sigma^2 &= \frac{\sum_{i=1}^n w_i x_i^2}{\sum_{i=1}^n w_i} - \frac{\left(\sum_{i=1}^n w_i x_i\right)^2}{\left(\sum_{i=1}^n w_i\right)^2} \end{aligned}$$



pomegranate supports out-of-core learning

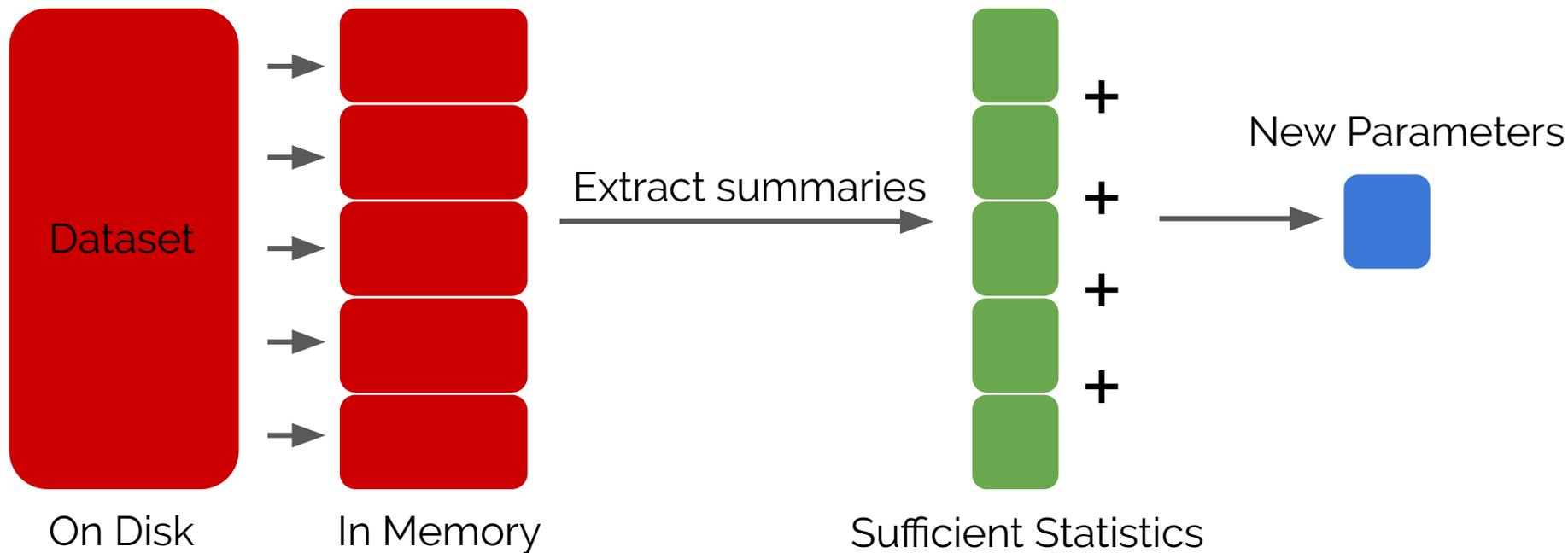
Batches from a dataset can be reduced to additive summary statistics, enabling exact updates from data that can't fit in memory.





pomegranate supports parallelization

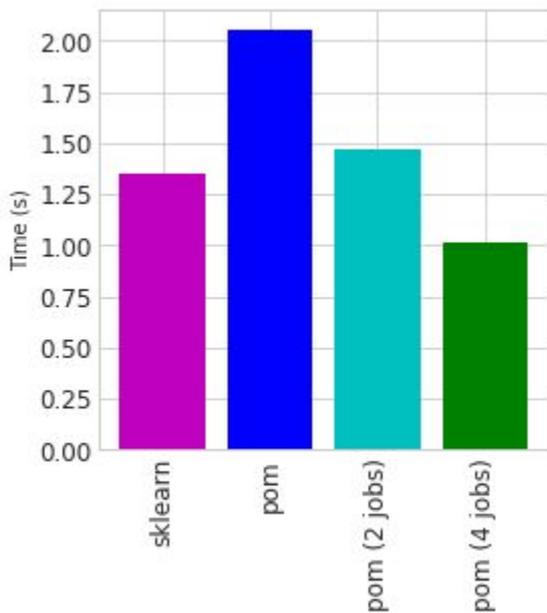
Multiple batches can be loaded at the same time and processed by different threads using `n_jobs` in either fitting or prediction methods



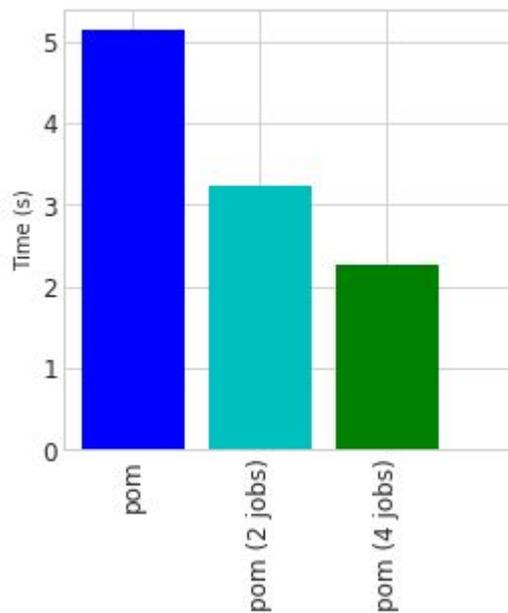


Training models in parallel

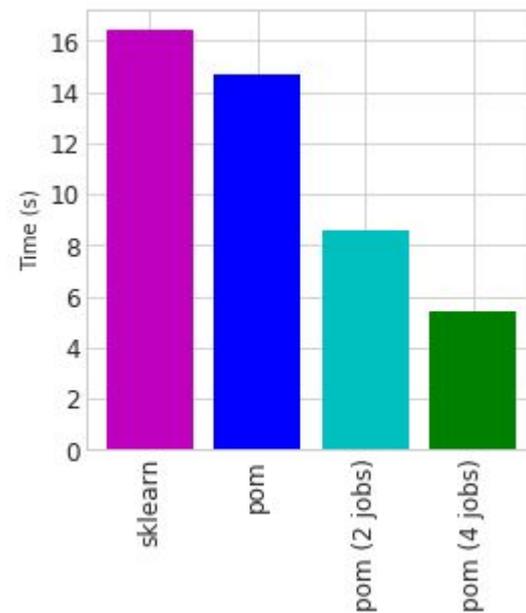
Naive Bayes' (50k, 1k)



Bayes' Classifier (50k, 1k)



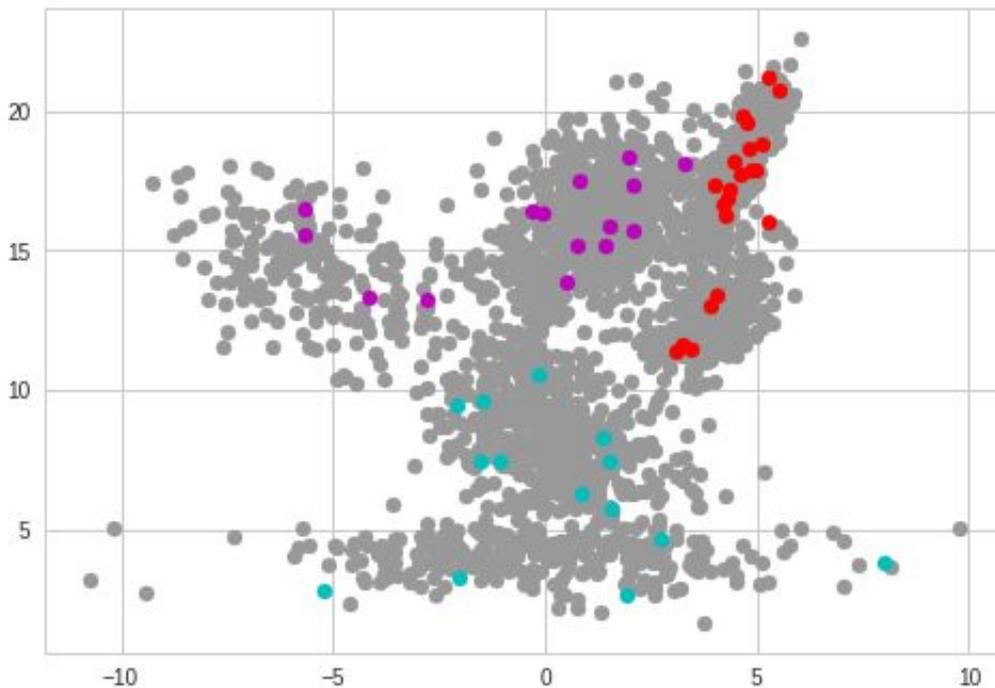
GMM (1M by 5)





pomegranate allows semisupervised learning

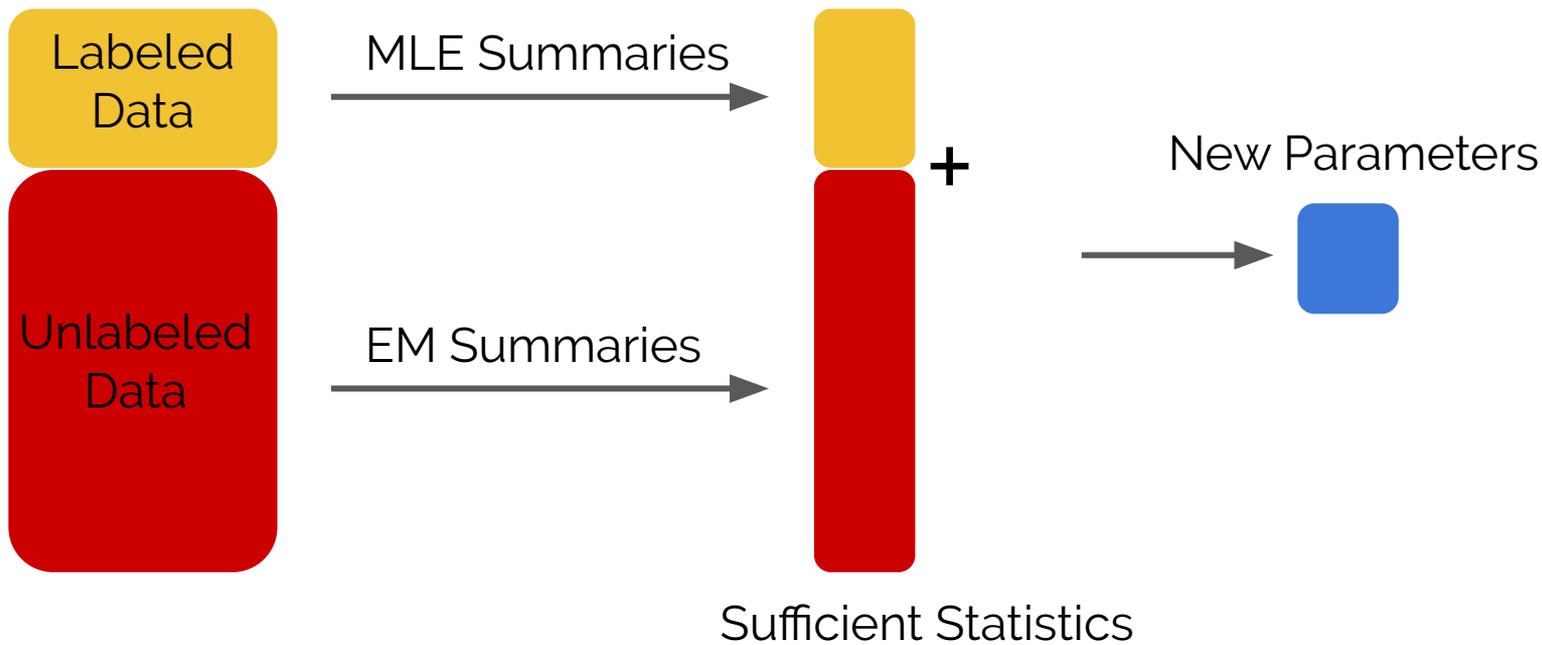
For many tasks, there is limited labeled data but a deluge of unlabeled data, and one wants to utilize both.





Training uses labeled and unlabeled data

Summaries from MLE on the labeled data can be added to summaries from EM on the unlabeled data



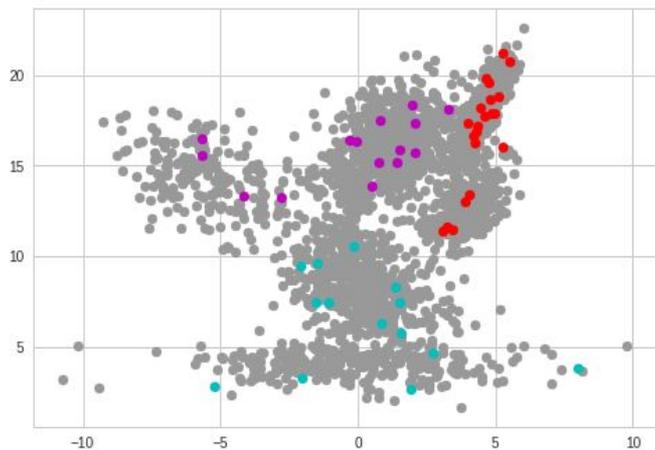


Resulting models can be more accurate

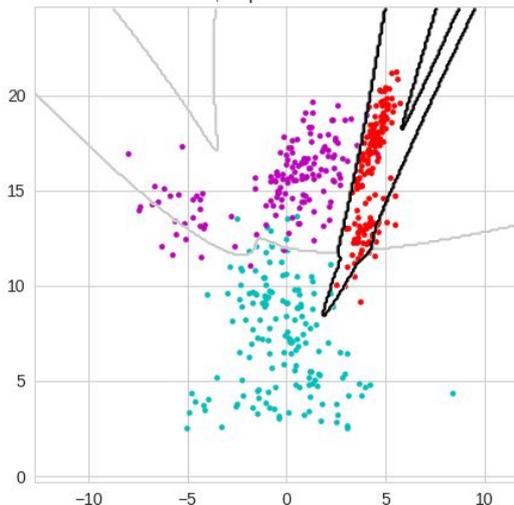
Supervised Acc: 0.93

Semisupervised Acc: 0.96

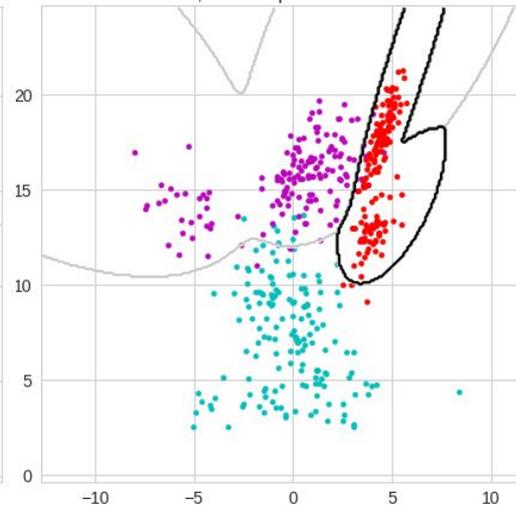
Training Data



Test Data, Supervised Boundaries



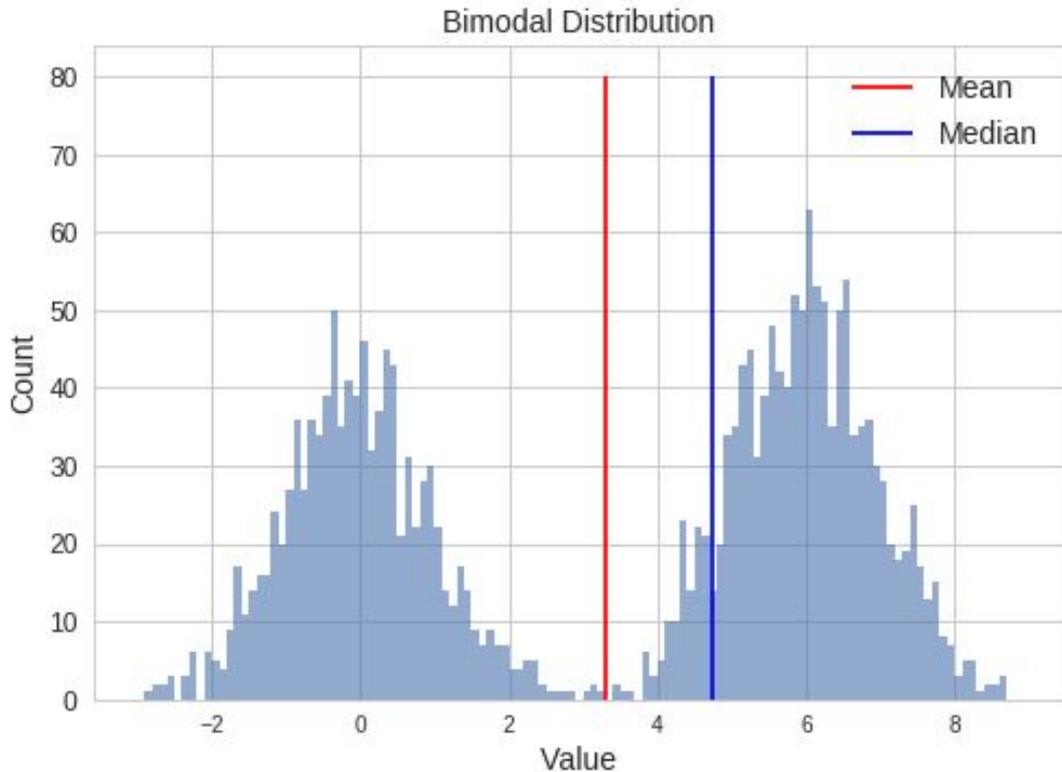
Test Data, Semi-supervised Boundaries





pomegranate supports missing data

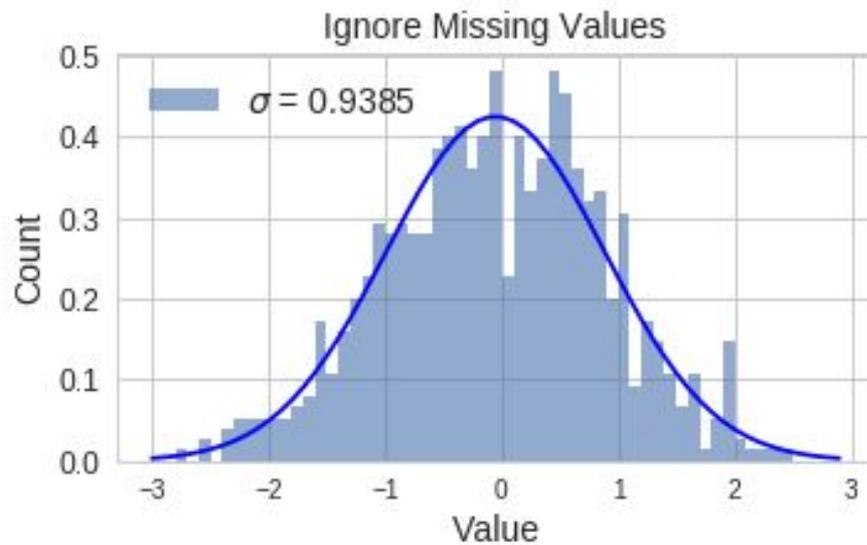
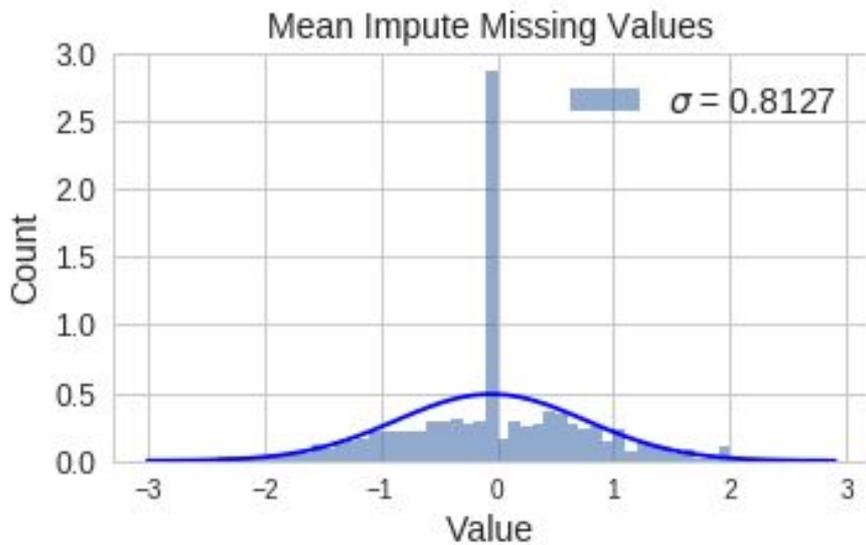
Many real world tasks involve missing data, but common approaches aren't sufficient for tackling the problem.





pomegranate supports missing data

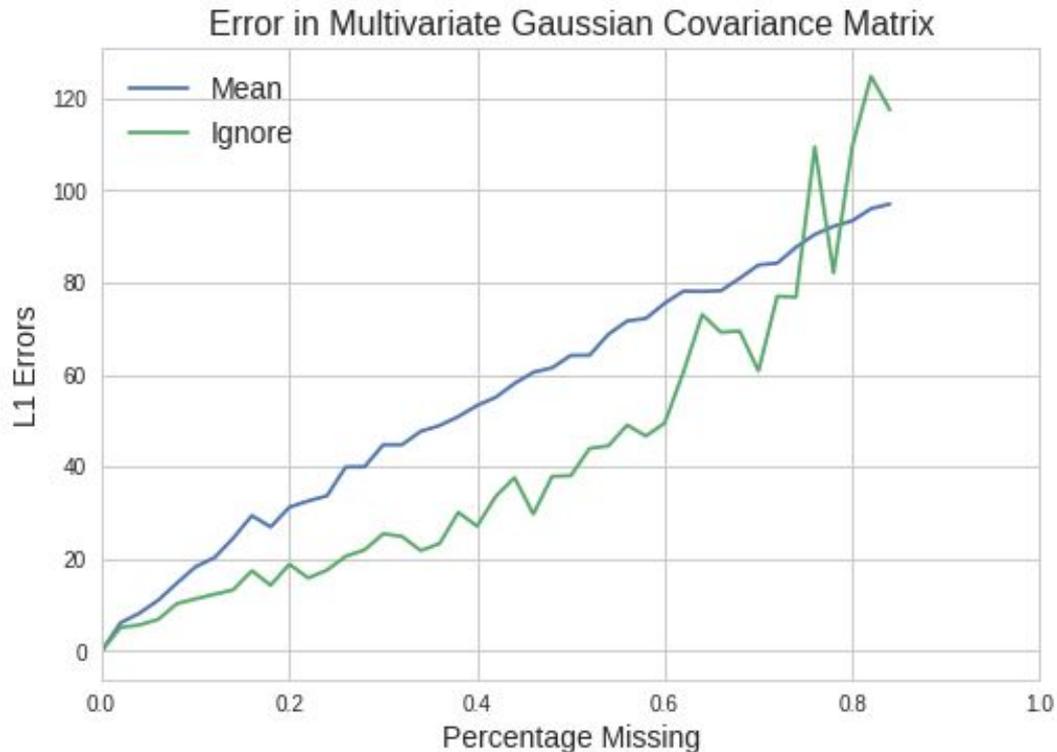
Many real world tasks involve missing data, but common approaches aren't sufficient for tackling the problem.





pomegranate supports missing data

Many real world tasks involve missing data, but common approaches aren't sufficient for tackling the problem.





pomegranate supports missing data

Pomegranate supports **model fitting**, **structure learning**, and **model predictions** on data sets that include missing values, no matter how complicated the model or sparse the data set.

You can **fit a Gaussian mixture model** to incomplete data sets.

You can run the **Viterbi or forward-backward algorithm** using a HMM on incomplete data sets.

You can **learn the structure of a Bayesian network** on incomplete data sets.

All without having to change your command, simply by including `np.nan` in the place of the missing value



pomegranate can be faster than scipy

```
mu, cov = numpy.random.randn(2000), numpy.eye(2000)
d = MultivariateGaussianDistribution(mu, cov)
X = numpy.random.randn(2000, 2000)
print "scipy time: ",
%timeit multivariate_normal.logpdf(X, mu, cov)
print "pomegranate time: ",
%timeit MultivariateGaussianDistribution(mu, cov).log_probability(X)
print "pomegranate time (w/ precreated object): ",
%timeit d.log_probability(X)
```

```
scipy time: 1 loop, best of 3: 1.67 s per loop
pomegranate time: 1 loop, best of 3: 801 ms per loop
pomegranate time (w/ precreated object): 1 loop, best of 3: 216 ms per loop
```



pomegranate caches aggressively

$$P(X|\mu, \sigma) = \frac{1}{\sqrt{2\pi}\sigma} \exp\left(-\frac{(x - \mu)^2}{2\sigma^2}\right)$$

$$\log P(X|\mu, \sigma) = -\log(\sqrt{2\pi}\sigma) - \frac{(x - \mu)^2}{2\sigma^2}$$

$$\log P(X|\mu, \sigma) = \alpha + \beta(x - \mu)^2$$



GOSSIP GIRL



Example 'blast' from Gossip Girl

Spotted: Lonely Boy. Can't believe the love of his life has returned. If only she knew who he was. But everyone knows Serena. And everyone is talking. Wonder what Blair Waldorf thinks. Sure, they're BFF's, but we always thought Blair's boyfriend Nate had a thing for Serena.



Example 'blast' from Gossip Girl

Why'd she leave? Why'd she return? Send me all the deets.
And who am I? That's the secret I'll never tell. The only one.
—XOXO. Gossip Girl.



How do we encode these 'blasts'?

Better lock it down with Nate, B. Clock's ticking.

+1 Nate

-1 Blair



How do we encode these 'blasts'?

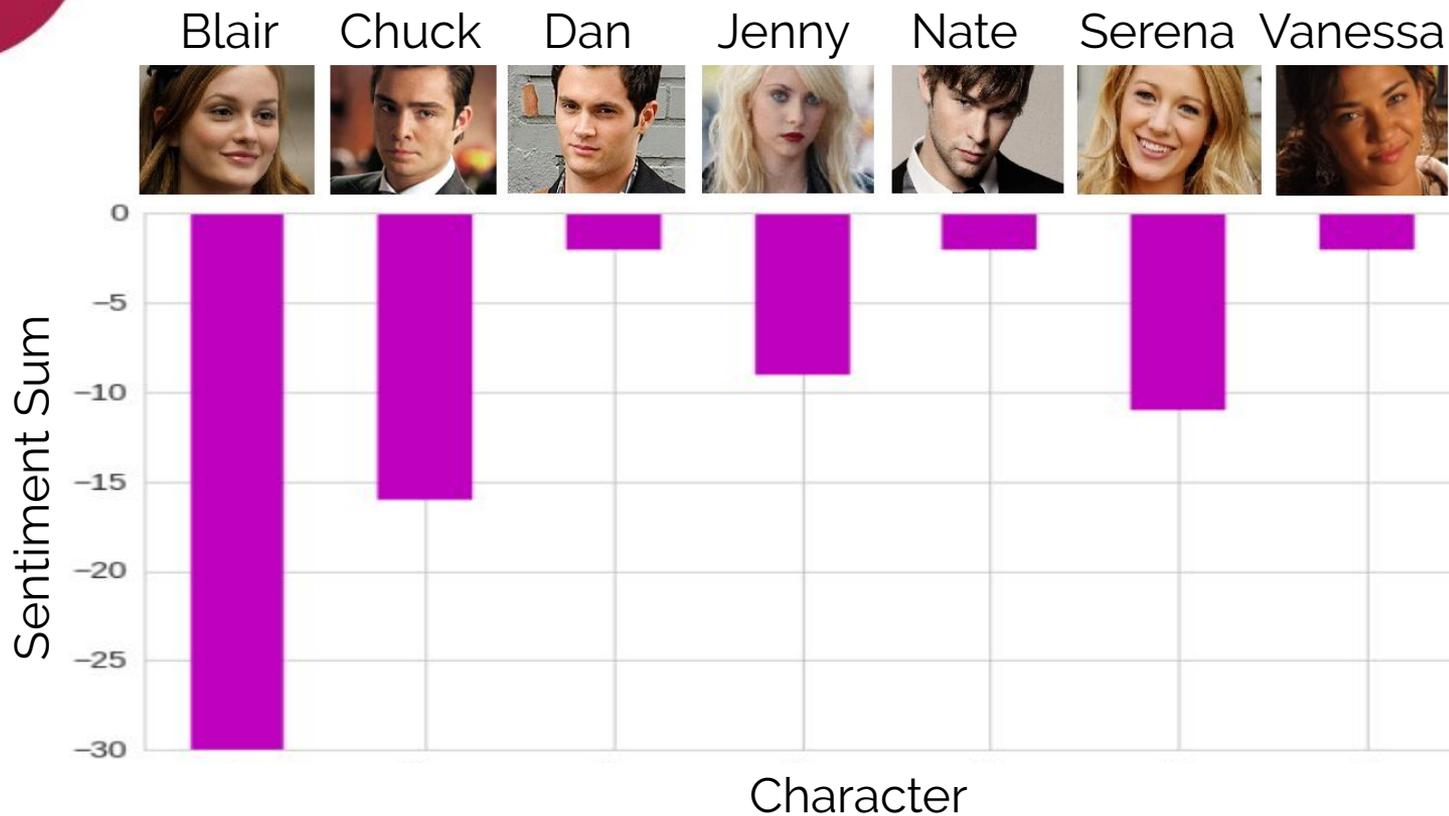
This just in: S and B committing a crime of fashion. Who doesn't love a five-finger discount. Especially if it's the middle one.

-1 Blair

-1 Serena

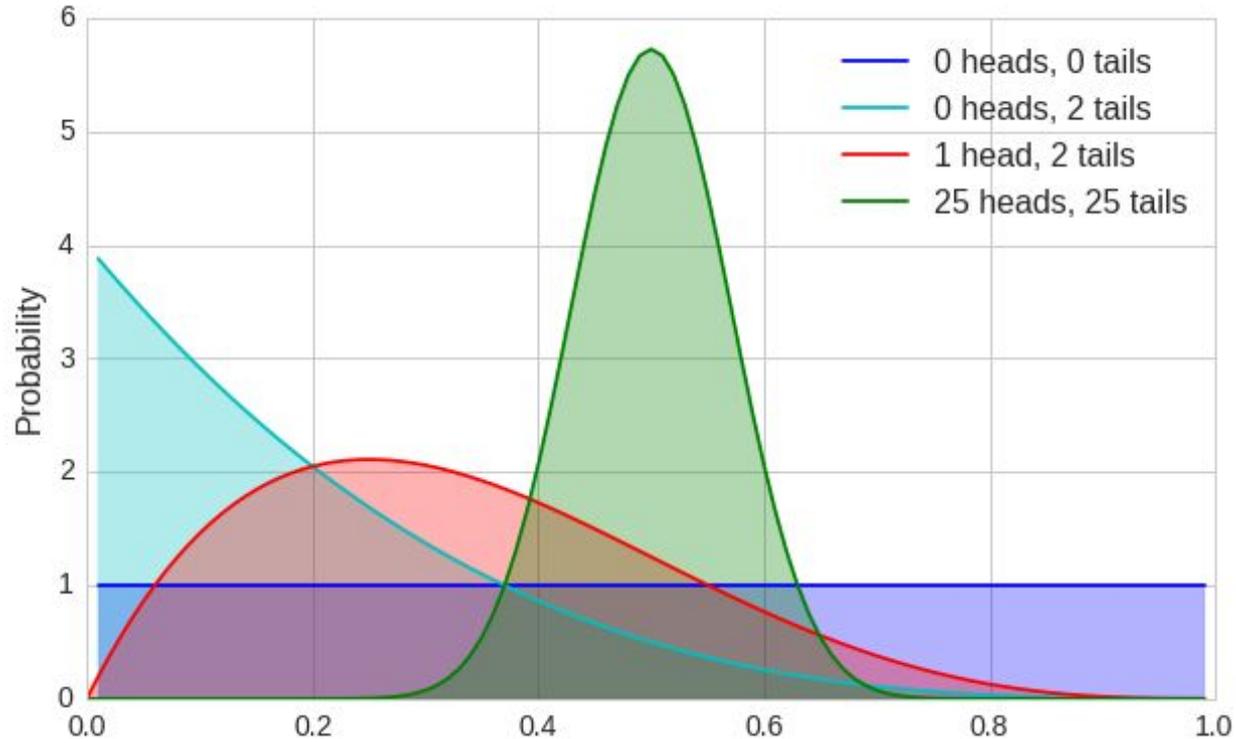


Simple summations don't work well



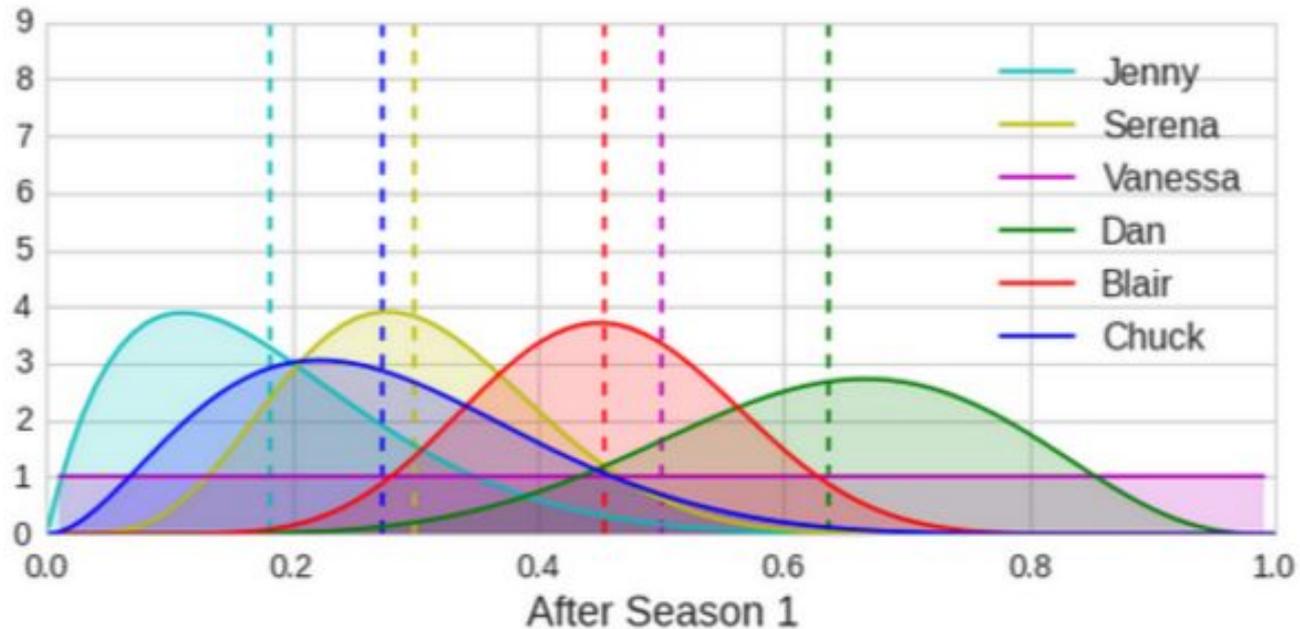


Beta distributions can model uncertainty



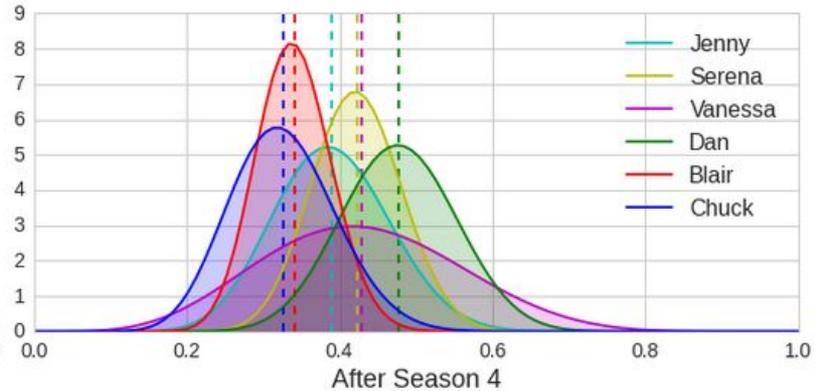
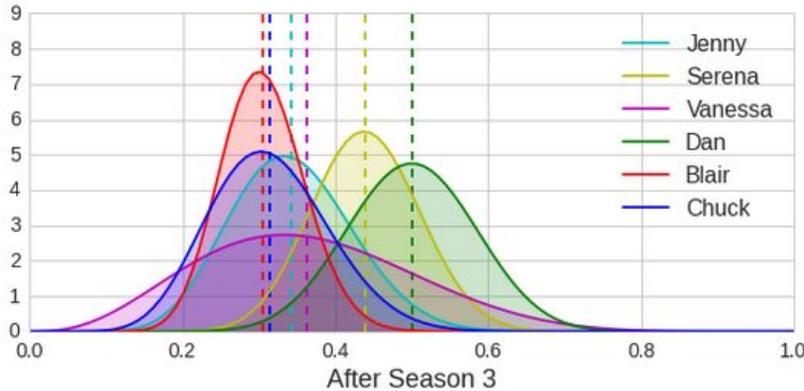
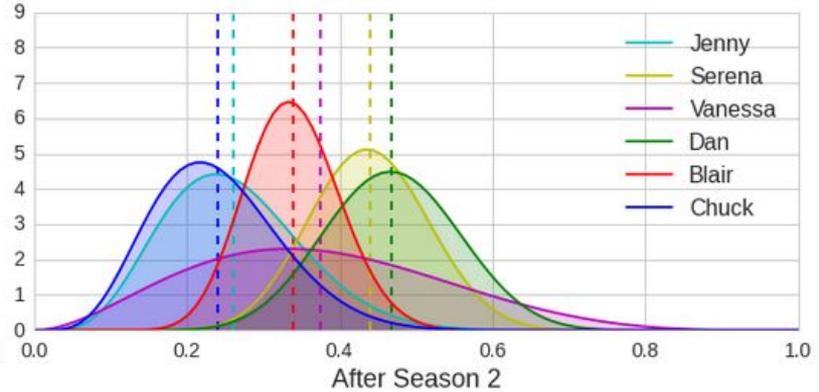
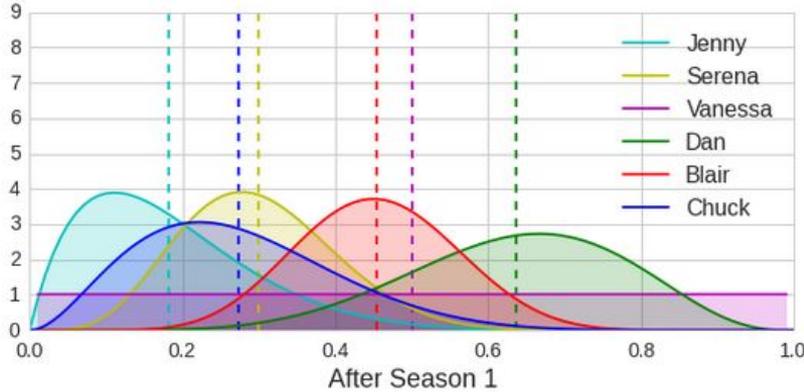


Beta distributions can model uncertainty





Beta distributions can model uncertainty





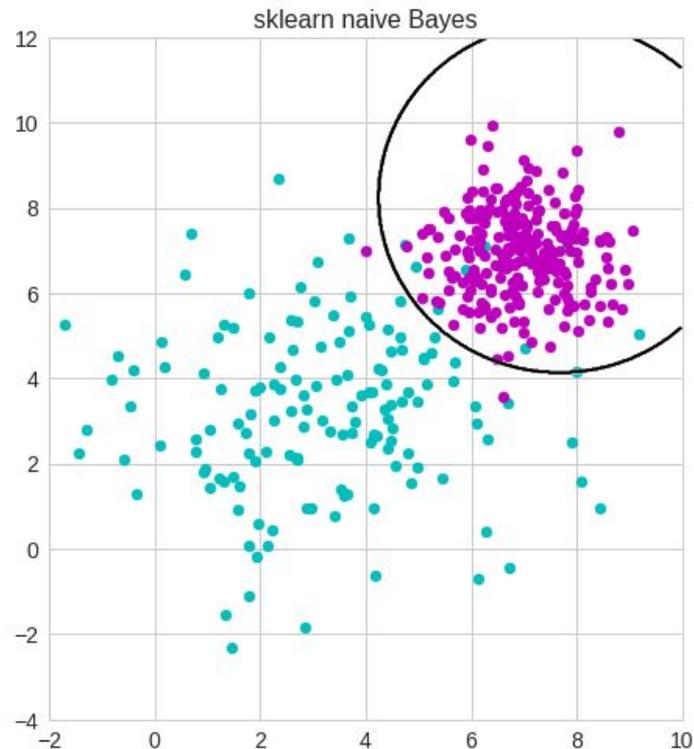
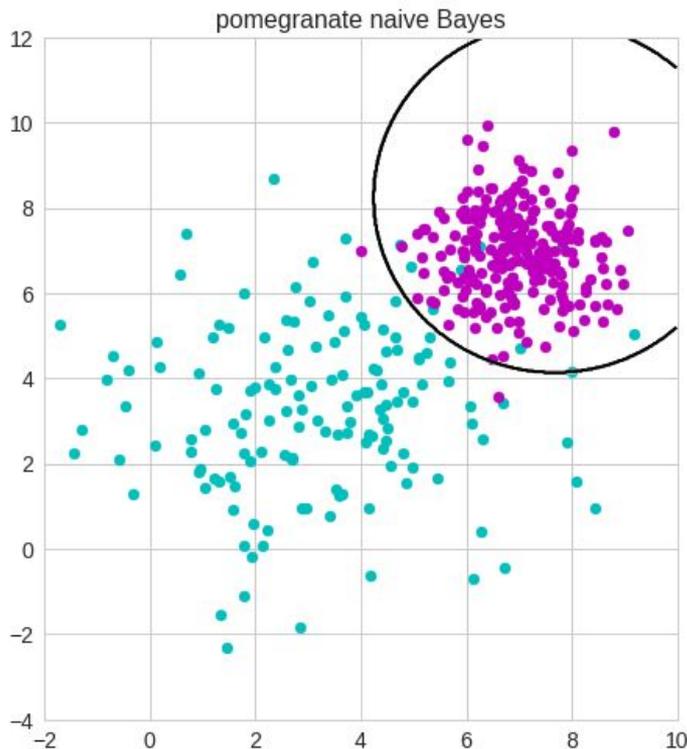
Naive Bayes assumes independent features

$$\textit{Posterior} = \frac{\textit{Likelihood} * \textit{Prior}}{\textit{Normalization}}$$

$$P(M|D) = \frac{\prod_{i=1}^d P(D_i|M)P(M)}{\sum_M \prod_{i=1}^d P(D_i|M)P(M)}$$



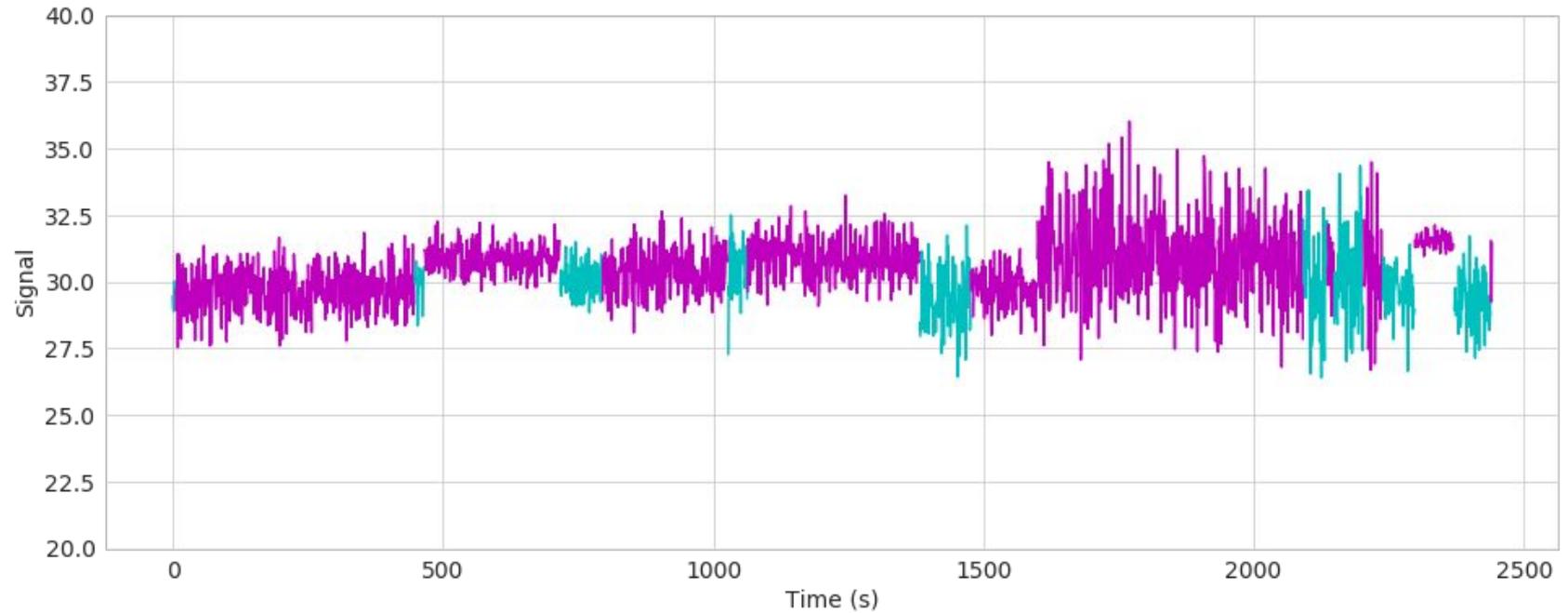
Naive Bayes produces ellipsoid boundaries



```
model = NaiveBayes.from_samples(NormalDistribution, X, y)
```

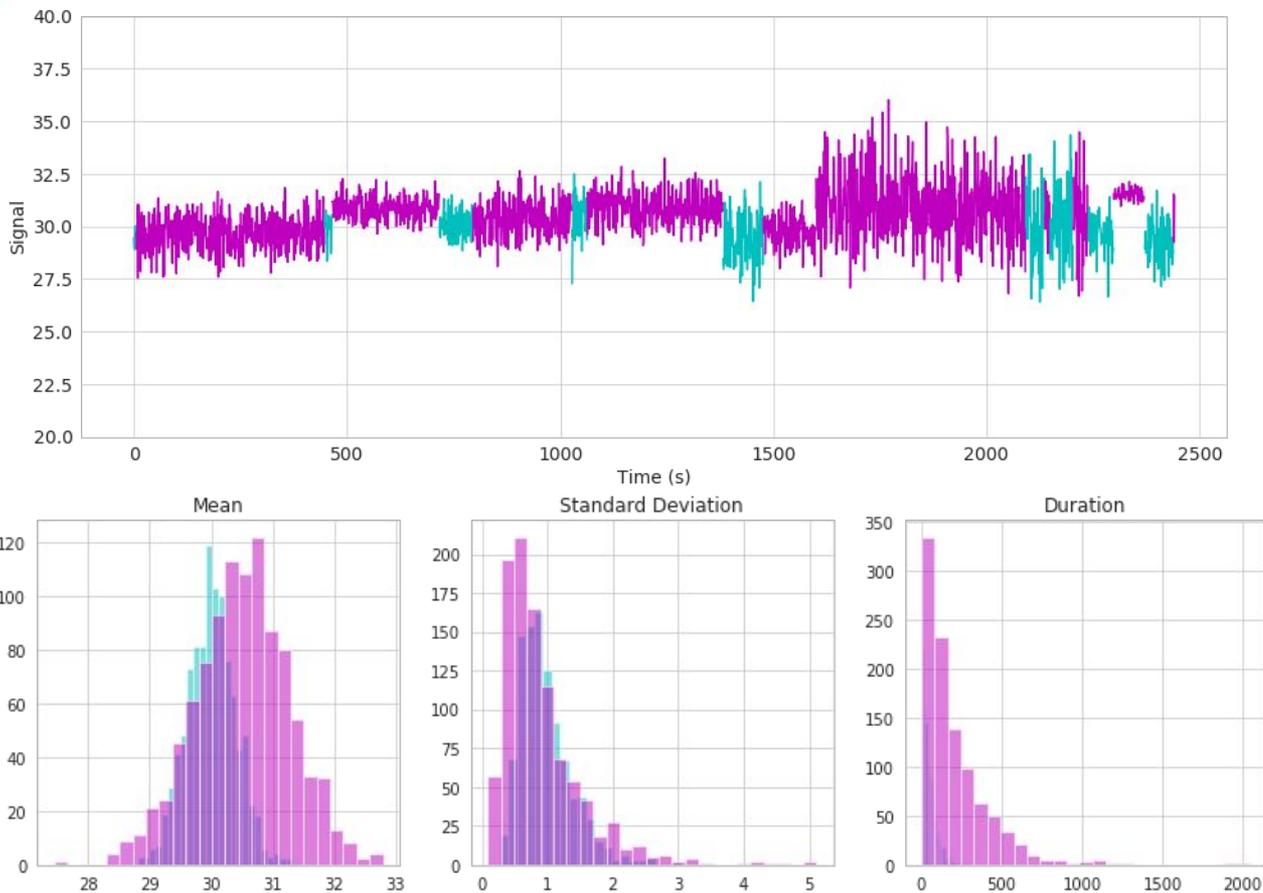


Different features can have different distributions





Different features can have different distributions





Appropriate distributions can improve performance

```
: model = NaiveBayes.from_samples(NormalDistribution, X_train, y_train)
print "Gaussian Naive Bayes: ", (model.predict(X_test) == y_test).mean()

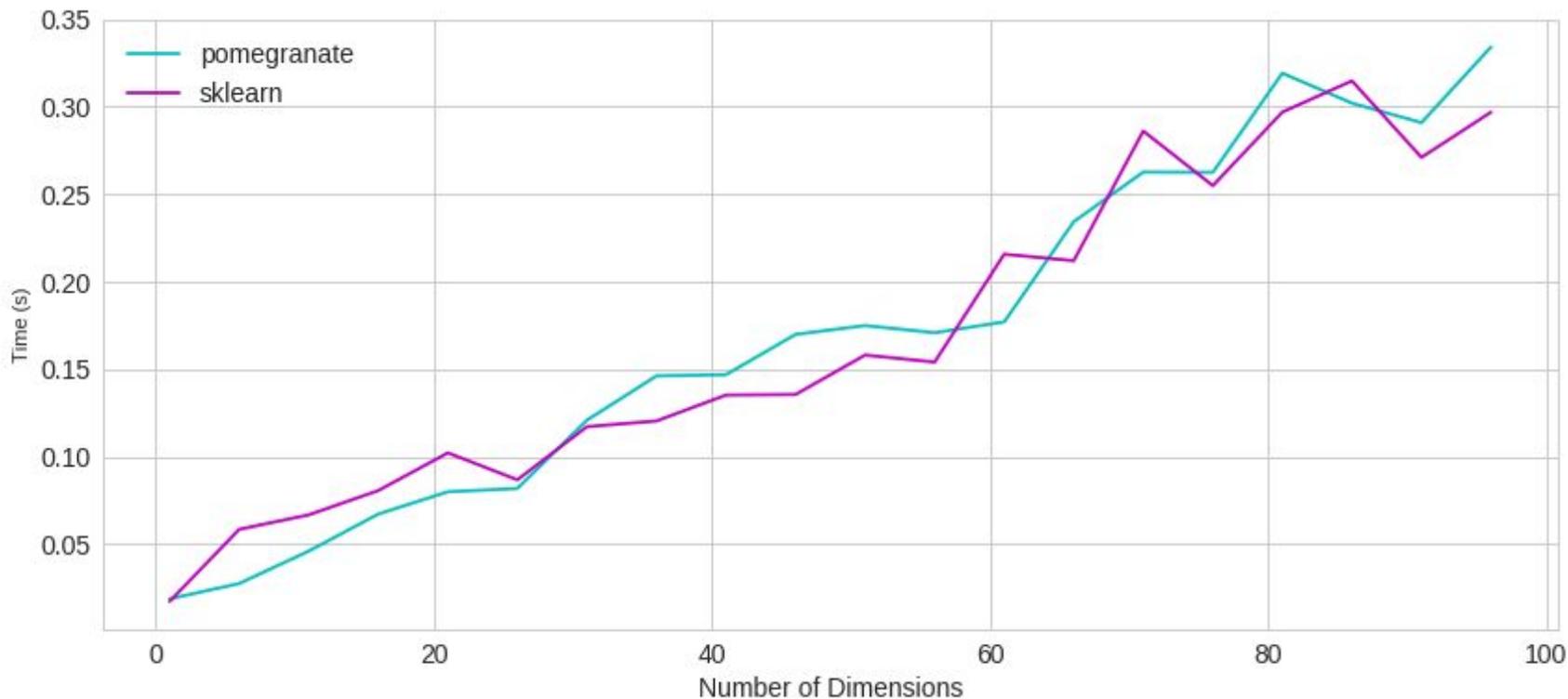
clf = GaussianNB().fit(X_train, y_train)
print "sklearn Gaussian Naive Bayes: ", (clf.predict(X_test) == y_test).mean()

model = NaiveBayes.from_samples([NormalDistribution, LogNormalDistribution, ExponentialDistribution], X_train, y_train)
print "Heterogeneous Naive Bayes: ", (model.predict(X_test) == y_test).mean()
```

```
Gaussian Naive Bayes: 0.794
sklearn Gaussian Naive Bayes: 0.794
Heterogeneous Naive Bayes: 0.822
```



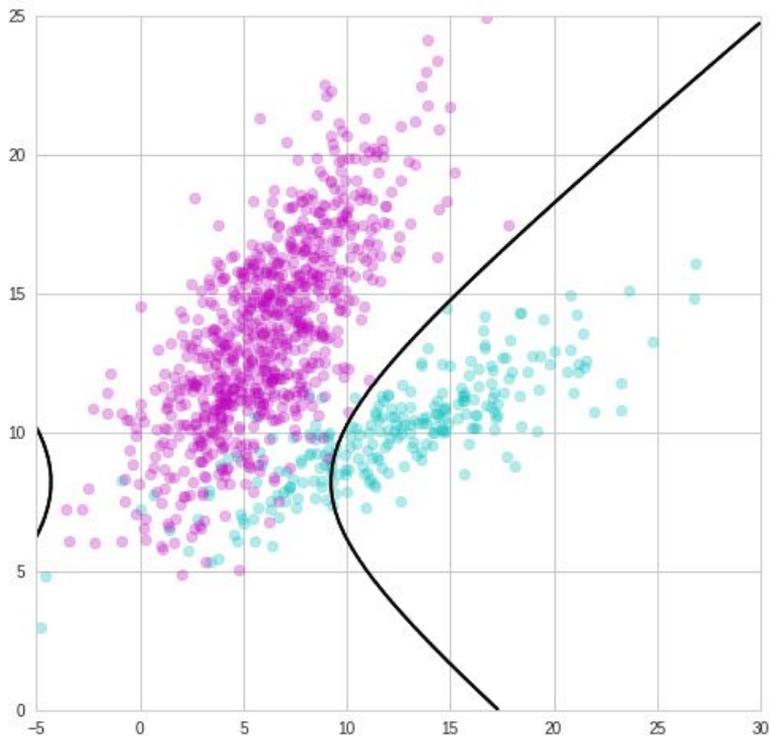
This additional flexibility is just as fast



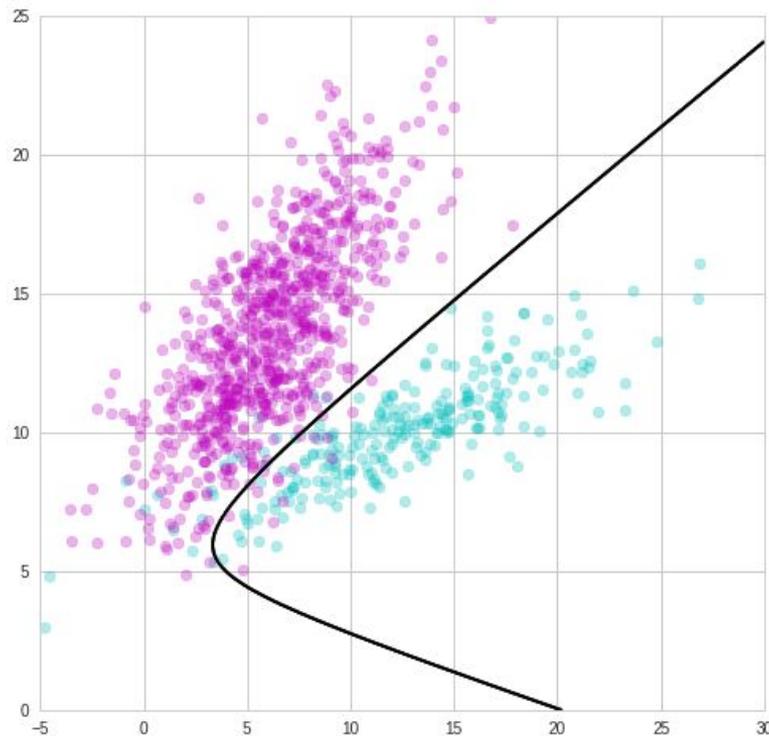


What if you didn't require independence?

naive accuracy: 0.929

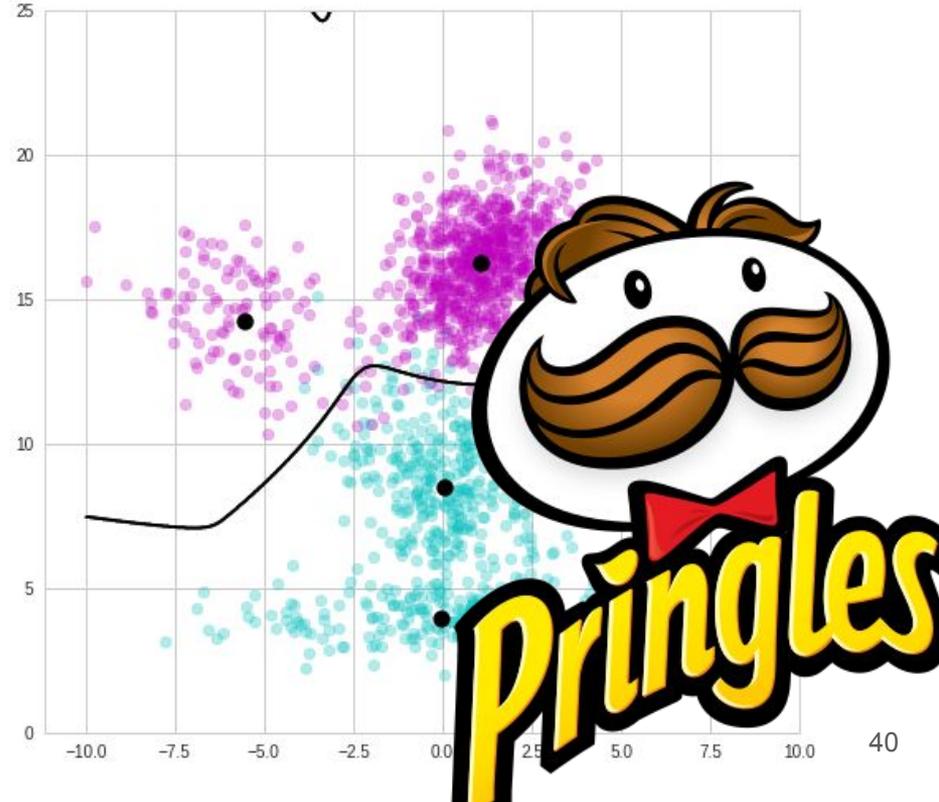
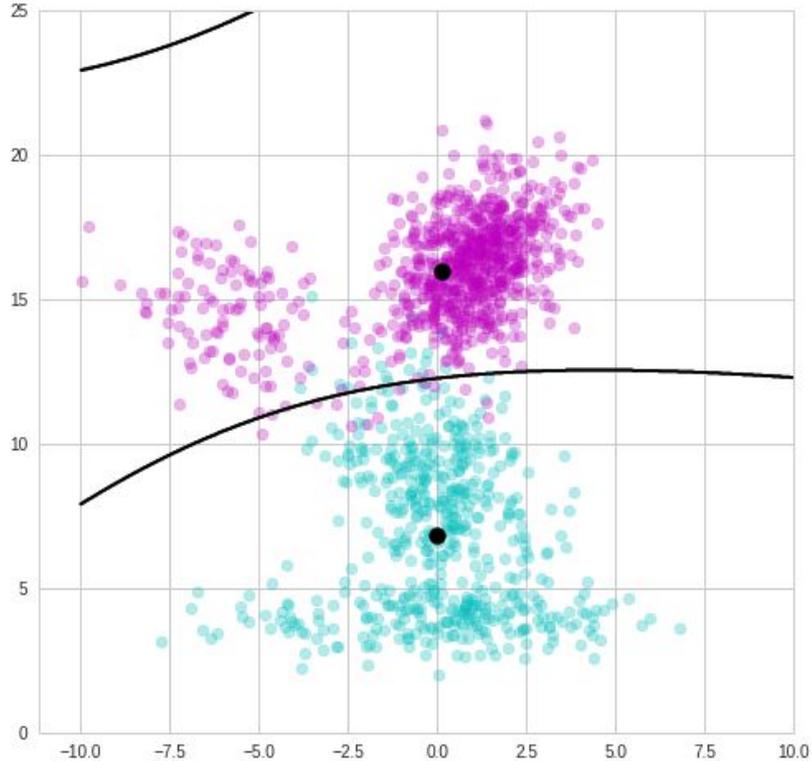


bayes classifier accuracy: 0.966





Gaussian mixture model Bayes classifier





You can now pass in your own distributions

```
class StudentTDistribution():
    def __init__(self, mu, std, df=1.0):
        self.mu = mu
        self.std = std
        self.df = df
        self.parameters = (self.mu, self.std)
        self.d = 1
        self.summaries = numpy.zeros(3)

    def probability(self, X):
        return numpy.exp(self.log_probability(X))

    def log_probability(self, X):
        return scipy.stats.t.logpdf(X, self.df, self.mu, self.std)

    def summarize(self, X, w=None):
        if w is None:
            w = numpy.ones(X.shape[0])

        X = X.reshape(X.shape[0])
        self.summaries[0] += w.sum()
        self.summaries[1] += X.dot(w)
        self.summaries[2] += (X ** 2.).dot(w)

    def from_summaries(self, inertia=0.0):
        self.mu = self.summaries[1] / self.summaries[0]
        self.std = self.summaries[2] / self.summaries[0] - self.summaries[1] ** 2 / (self.summaries[0] **
2)

        self.std = numpy.sqrt(self.std)
        self.parameters = (self.mu, self.std)
        self.clear_summaries()

    def clear_summaries(self, inertia=0.0):
        self.summaries = numpy.zeros(3)

    @classmethod
    def from_samples(cls, X, weights=None, df=1):
        d = StudentTDistribution(0, 0, df)
        d.summarize(X, weights)
        d.from_summaries()
        return d
```



You can now pass in your own distributions

Take in parameters

```
class StudentTDistribution():
    def __init__(self, mu, std, df=1.0):
        self.mu = mu
        self.std = std
        self.df = df
        self.parameters = (self.mu, self.std)
        self.d = 1
        self.summaries = numpy.zeros(3)

    def probability(self, X):
        return numpy.exp(self.log_probability(X))

    def log_probability(self, X):
        return scipy.stats.t.logpdf(X, self.df, self.mu, self.std)

    def summarize(self, X, w=None):
        if w is None:
            w = numpy.ones(X.shape[0])

        X = X.reshape(X.shape[0])
        self.summaries[0] += w.sum()
        self.summaries[1] += X.dot(w)
        self.summaries[2] += (X ** 2.).dot(w)

    def from_summaries(self, inertia=0.0):
        self.mu = self.summaries[1] / self.summaries[0]
        self.std = self.summaries[2] / self.summaries[0] - self.summaries[1] ** 2 / (self.summaries[0] **
2)

        self.std = numpy.sqrt(self.std)
        self.parameters = (self.mu, self.std)
        self.clear_summaries()

    def clear_summaries(self, inertia=0.0):
        self.summaries = numpy.zeros(3)

    @classmethod
    def from_samples(cls, X, weights=None, df=1):
        d = StudentTDistribution(0, 0, df)
        d.summarize(X, weights)
        d.from_summaries()
        return d
```



You can now pass in your own distributions

Calculate probabilities

```
class StudentTDistribution():
    def __init__(self, mu, std, df=1.0):
        self.mu = mu
        self.std = std
        self.df = df
        self.parameters = (self.mu, self.std)
        self.d = 1
        self.summaries = numpy.zeros(3)

    def probability(self, X):
        return numpy.exp(self.log_probability(X))

    def log_probability(self, X):
        return scipy.stats.t.logpdf(X, self.df, self.mu, self.std)

    def summarize(self, X, w=None):
        if w is None:
            w = numpy.ones(X.shape[0])

        X = X.reshape(X.shape[0])
        self.summaries[0] += w.sum()
        self.summaries[1] += X.dot(w)
        self.summaries[2] += (X ** 2.).dot(w)

    def from_summaries(self, inertia=0.0):
        self.mu = self.summaries[1] / self.summaries[0]
        self.std = self.summaries[2] / self.summaries[0] - self.summaries[1] ** 2 / (self.summaries[0] **
2)

        self.std = numpy.sqrt(self.std)
        self.parameters = (self.mu, self.std)
        self.clear_summaries()

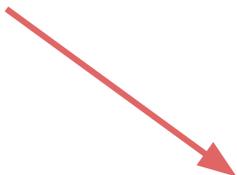
    def clear_summaries(self, inertia=0.0):
        self.summaries = numpy.zeros(3)

    @classmethod
    def from_samples(cls, X, weights=None, df=1):
        d = StudentTDistribution(0, 0, df)
        d.summarize(X, weights)
        d.from_summaries()
        return d
```



You can now pass in your own distributions

Out-of-core update
functions



```
class StudentTDistribution():
    def __init__(self, mu, std, df=1.0):
        self.mu = mu
        self.std = std
        self.df = df
        self.parameters = (self.mu, self.std)
        self.d = 1
        self.summaries = numpy.zeros(3)

    def probability(self, X):
        return numpy.exp(self.log_probability(X))

    def log_probability(self, X):
        return scipy.stats.t.logpdf(X, self.df, self.mu, self.std)

    def summarize(self, X, w=None):
        if w is None:
            w = numpy.ones(X.shape[0])

        X = X.reshape(X.shape[0])
        self.summaries[0] += w.sum()
        self.summaries[1] += X.dot(w)
        self.summaries[2] += (X ** 2.).dot(w)

    def from_summaries(self, inertia=0.0):
        self.mu = self.summaries[1] / self.summaries[0]
        self.std = self.summaries[2] / self.summaries[0] - self.summaries[1] ** 2 / (self.summaries[0] **
2)

        self.std = numpy.sqrt(self.std)
        self.parameters = (self.mu, self.std)
        self.clear_summaries()

    def clear_summaries(self, inertia=0.0):
        self.summaries = numpy.zeros(3)

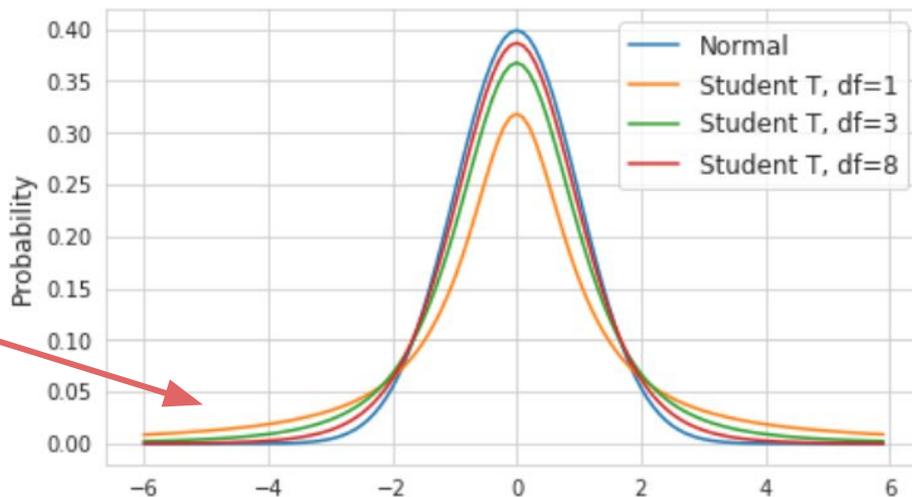
    @classmethod
    def from_samples(cls, X, weights=None, df=1):
        d = StudentTDistribution(0, 0, df)
        d.summarize(X, weights)
        d.from_summaries()
        return d
```



You can now pass in your own distributions

```
dn = NormalDistribution(0, 1)
dt1 = StudentTDistribution(0, 1, 1)
dt3 = StudentTDistribution(0, 1, 3)
dt8 = StudentTDistribution(0, 1, 8)
```

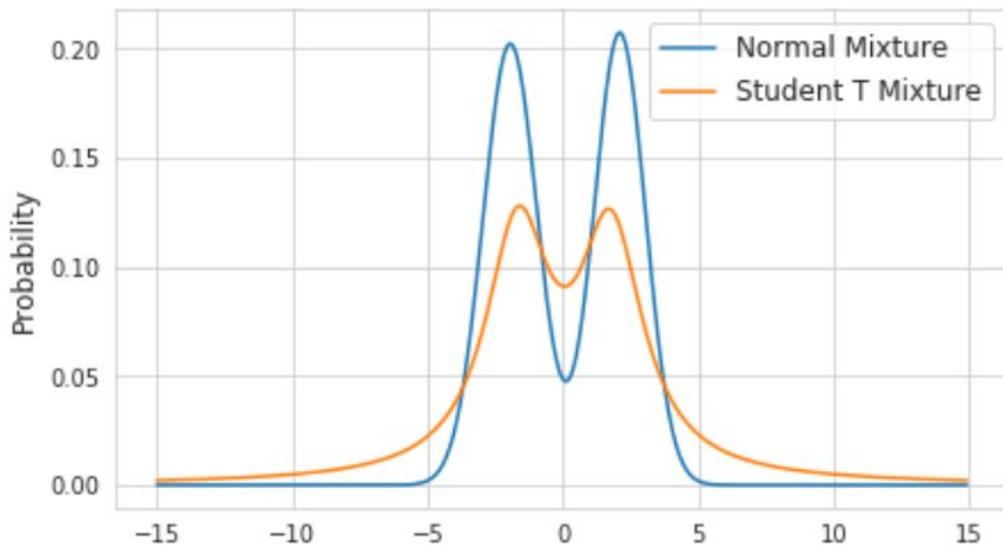
Larger tails to
capture more
uncertainty





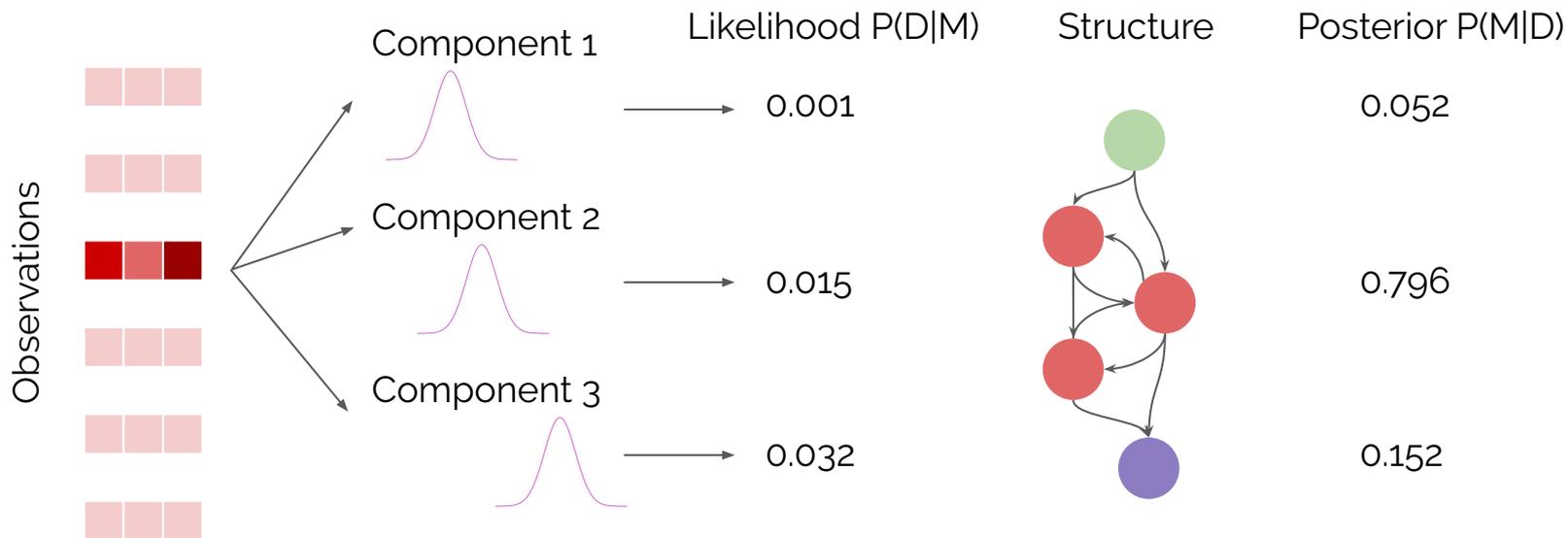
Custom distributions simply compatible

```
modeln = GeneralMixtureModel.from_samples(NormalDistribution, 2, X)  
modelt = GeneralMixtureModel.from_samples(StudentTDistribution, 2, X)
```





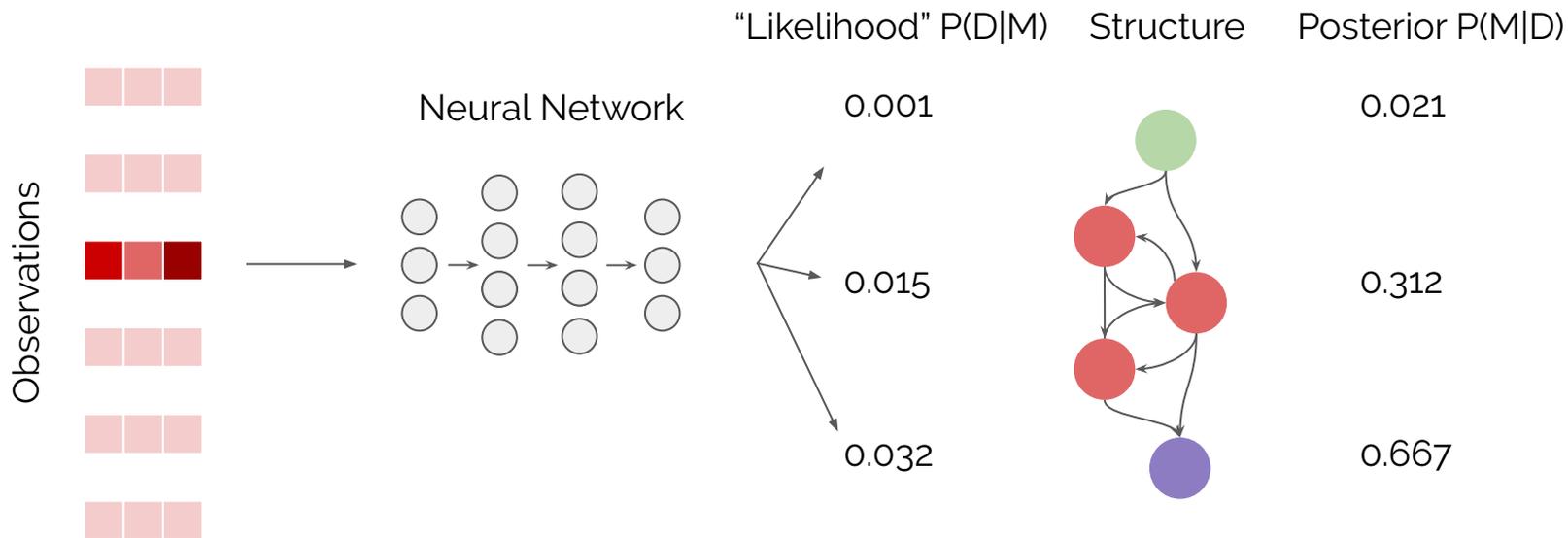
HMMs typically use a set of distributions





Neural HMMs use a single neural network

Can model complex interactions between features, e.g., pixels in an image, much better than individual distributions

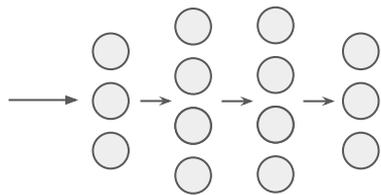




The HMM adds structural regularization to the NN



CNN (VGG,
GoogLeNet...)



"Likelihood" $P(D|M)$

Fish: 0.001

Lion: 0.015

Li'l fluffers: 0.932



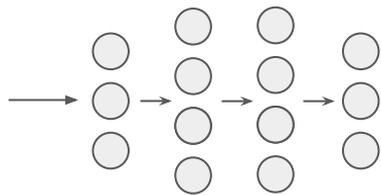


The HMM adds structural regularization to the NN

Video

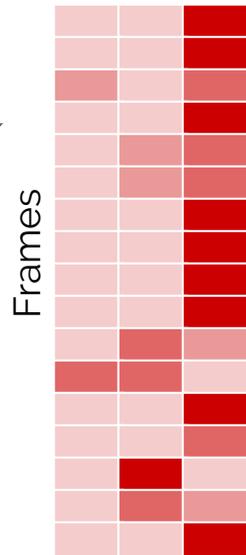


CNN (VGG,
GoogLeNet...)

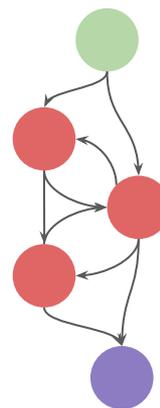


"Likelihood" $P(D|M)$

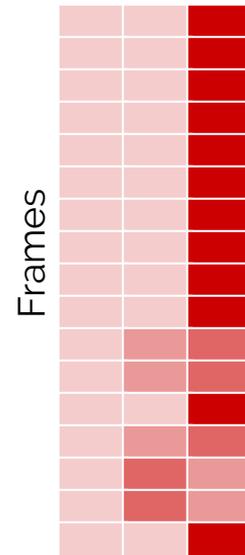
Classes



Structure



Classes



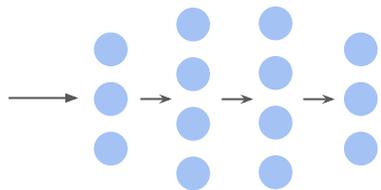


Freezing pre-trained networks can reduce compute

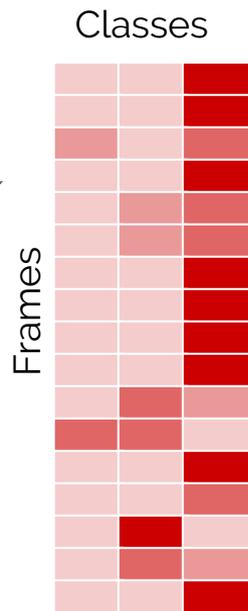
Video



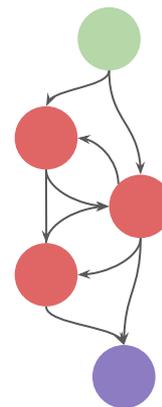
CNN (VGG,
GoogLeNet...)



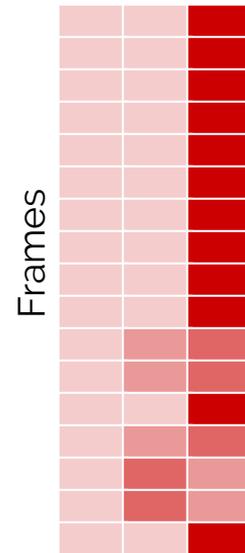
"Likelihood" $P(D|M)$



Structure



Classes





pomegranate paper at JMLR-MLOSS

pomegranate: fast and flexible probabilistic modeling in python

Jacob Schreiber

Paul G. Allen School of Computer Science
University of Washington
Seattle, WA 98195
jmschr@cs.washington.edu

Abstract

We present pomegranate, an open source machine learning package for probabilistic modeling in Python. Probabilistic modeling encompasses a wide range of methods that explicitly describe uncertainty using probability distributions. Three widely used probabilistic models implemented in pomegranate are general mixture models, hidden Markov models, and Bayesian networks. A primary focus of pomegranate is to abstract away the complexities of training models from their definition. This allows users to focus on specifying the correct model for their



pomegranate is NumFOCUS affiliated



[Home](#) [About](#) [Open Source Projects](#) [Community](#) [Programs](#) [Blog](#)



pomegranate

pomegranate is a Python module for fast and flexible probabilistic modeling inspired by the design of scikit-learn. A primary focus of pomegranate is to abstract away the intricacies of a model from its definition, allowing users to easily prototype with complex models and training strategies. Its modular implementation allows for probability distributions to be swapped in or out for each other with ease and for models to be stacked within each other, yielding such delights as a mixture of Bayesian networks or a Gaussian mixture model Bayes classifier.

<https://www.numfocus.org/open-source-projects/affiliated-projects/>



Documentation available at Readthedocs

🏠 pomegranate
latest

GETTING STARTED

Home

- Installation
- FAQ
- Release History

FEATURES

- Out of Core Learning
- Semi-Supervised Learning
- Parallelism
- GPU Usage

MODELS

- Probability Distributions
- General Mixture Models
- Hidden Markov Models
- Bayes Classifiers and Naive Bayes
- Markov Chains

Docs » Home Edit on GitHub

pomegranate

build passing build passing docs passing

Home

pomegranate is a python package which implements fast, efficient, and extremely flexible probabilistic models ranging from probability distributions to Bayesian networks to mixtures of hidden Markov models. The most basic level of probabilistic modeling is the a simple probability distribution. If we're modeling language, this may be a simple distribution over the frequency of all possible words a person can say.

- [Probability Distributions](#)

The next level up are probabilistic models which use the simple distributions in more complex ways. A markov chain can extend a simple probability distribution to say that the probability of a certain word depends on the word(s) which have been said previously. A hidden Markov model may say that the probability of a certain words depends on the latent/hidden state of the previous word,

<https://pomegranate.readthedocs.io/en/latest/>



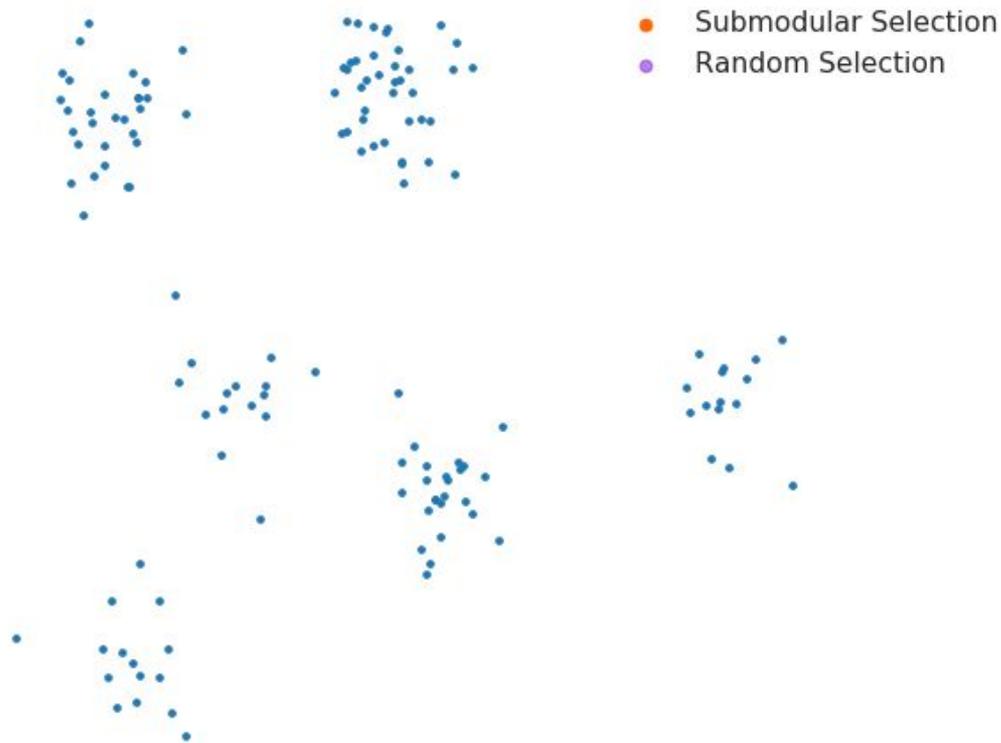
Tutorials available on GitHub

Branch: master ▾ pomegranate / tutorials / Create new file Upload files Find file History

jmschrei ENH NB/BC notebook Latest commit 5cd8d68 5 days ago
..
old ADD new overview tutorial a month ago
A_Overview.ipynb ADD new notebook features 12 days ago
B_Model_Tutorial_1_Distributions.ipynb ENH NB/BC notebook 5 days ago
B_Model_Tutorial_2_General_Mixture_Models.ipynb ADD new notebook features 12 days ago
B_Model_Tutorial_3_Hidden_Markov_Models.ipynb ADD new notebook features 12 days ago
B_Model_Tutorial_4_Bayesian_Networks.ipynb ENH NB/BC notebook 5 days ago
B_Model_Tutorial_4b_Bayesian_Network_Structure_Learning.ip... ADD new notebook features 12 days ago
B_Model_Tutorial_5_Bayes_Classifiers.ipynb ENH NB/BC notebook 5 days ago
B_Model_Tutorial_6_Markov_Chain.ipynb ADD new notebook features 12 days ago
C_Feature_Tutorial_1_Parallelization_and_GPUs.ipynb ADD new notebook features 12 days ago
C_Feature_Tutorial_8_Semisupervised_Learning.ipynb ADD new notebook features 12 days ago
C_Feature_Tutorial_9_Missing_Values.ipynb ADD new notebook features 12 days ago
GGBlasts.xlsx PyData Chicago 2016 2 years ago
README.md Update README.md 3 years ago

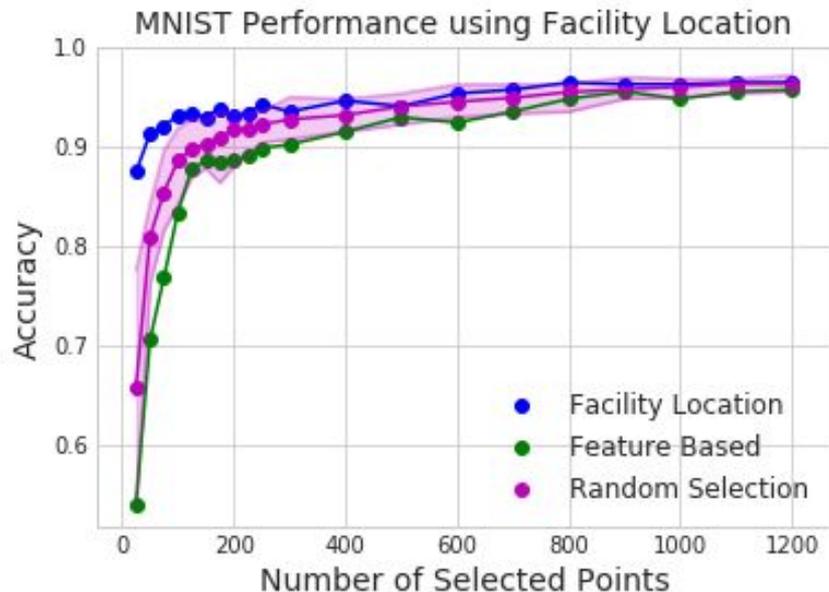
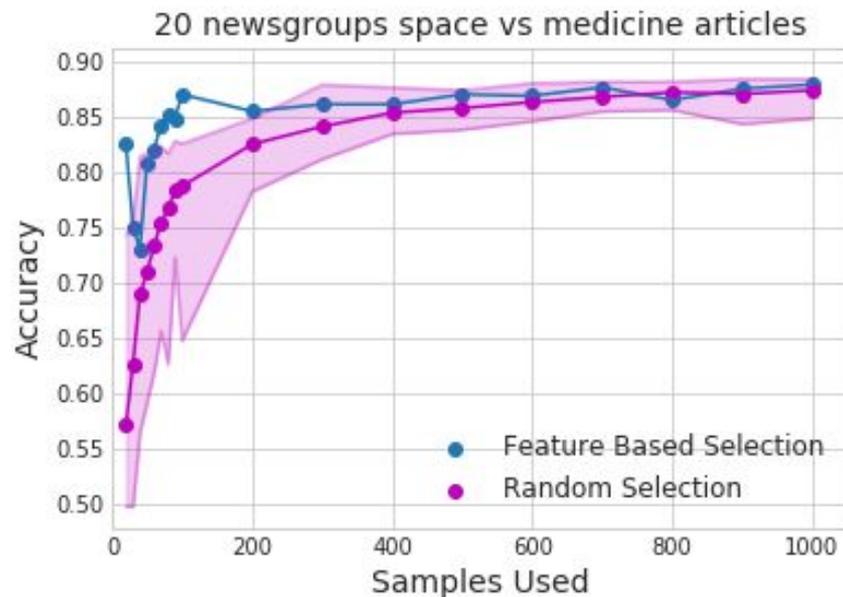
<https://github.com/jmschrei/pomegranate/tree/master/tutorials>

apricot implements submodular selection for training machine learning models faster



<https://github.com/jmschrei/apricot>

apricot implements submodular selection for training machine learning models faster



<https://github.com/jmschrei/apricot>

pomegranate

fast and flexible probabilistic modelling in python

Jacob Schreiber

Paul G. Allen School of Computer Science & Engineering
University of Washington



jmschreiber91



@jmschrei



@jmschreiber91