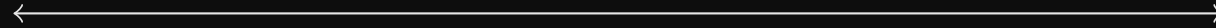


# mlpack: or, How I Learned To Stop Worrying and Love C++

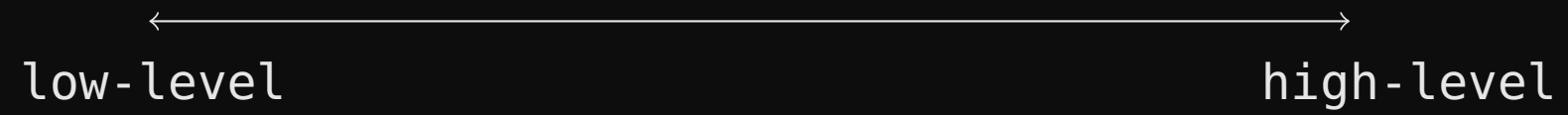
Ryan R. Curtin  
RelationalAI (*Atlanta, GA*)  
ryan@ratml.org  
ODSC East 2019 #ODSC  
May 2, 2019



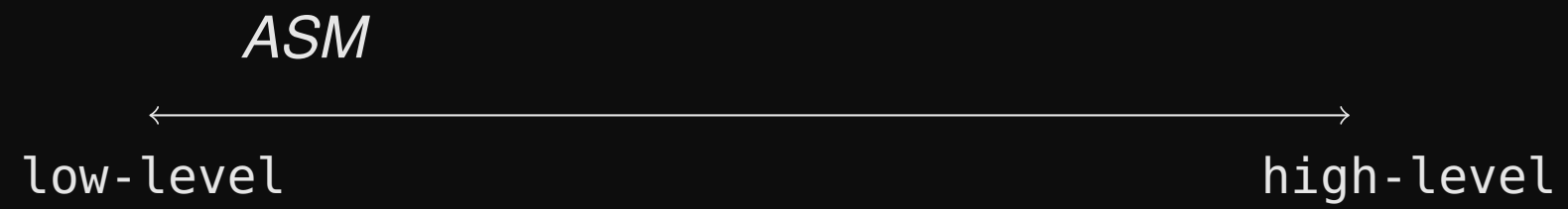
# Graph #1



# Graph #1



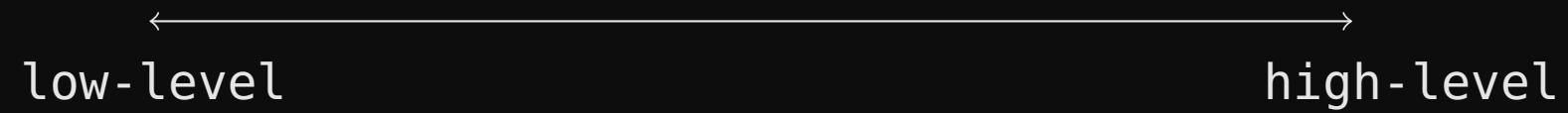
# Graph #1



# Graph #1



*ASM*

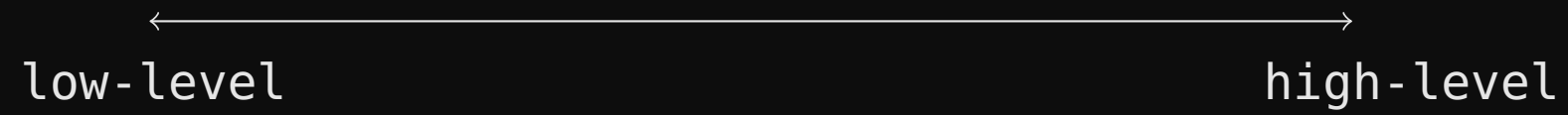


# Graph #1



*ASM*

*VB*



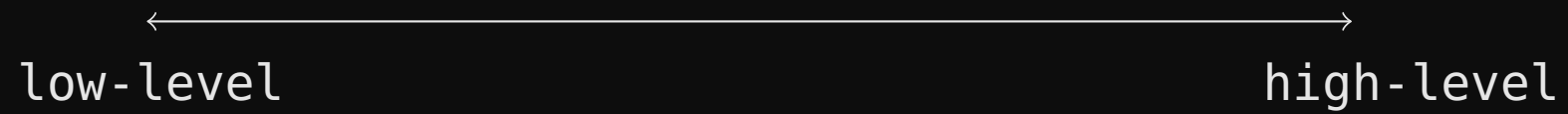
# Graph #1



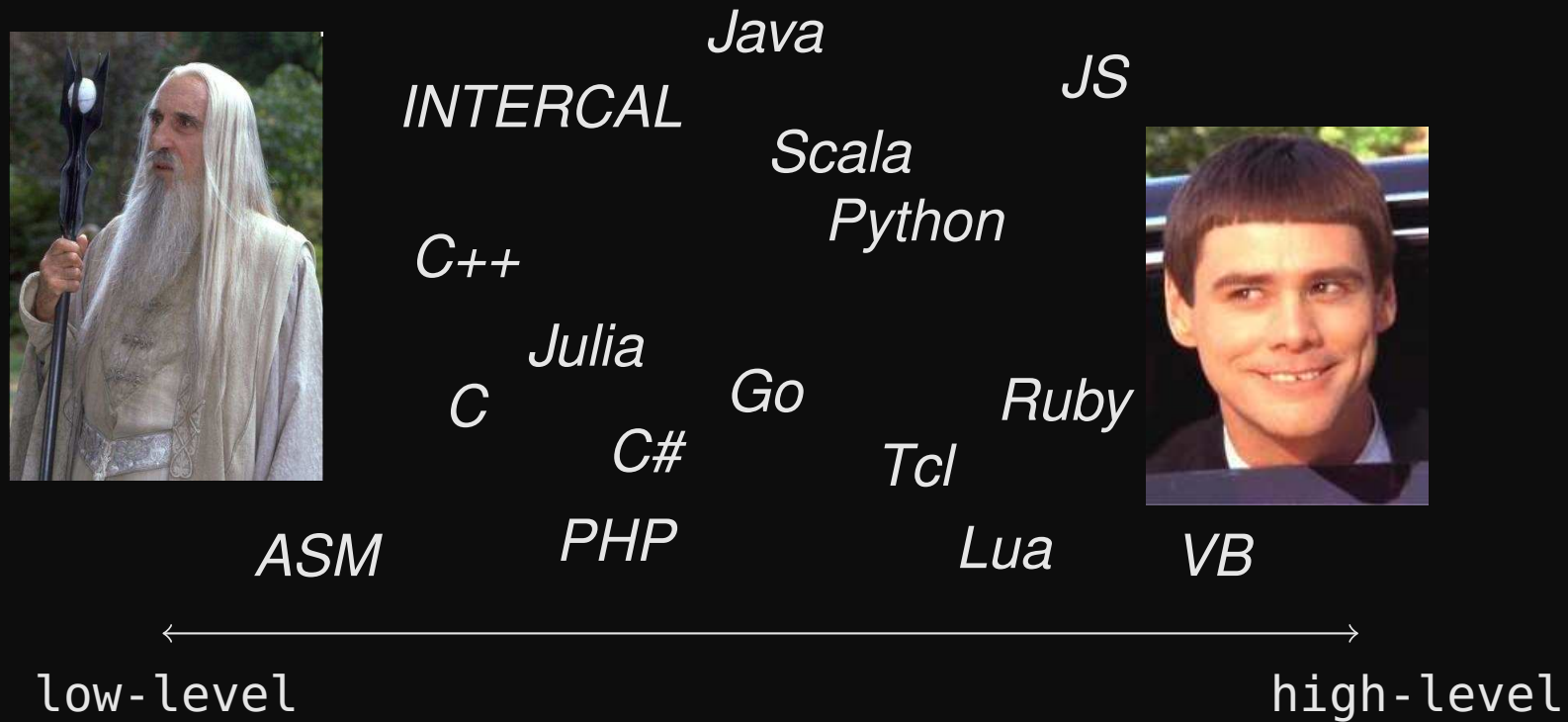
*ASM*



*VB*



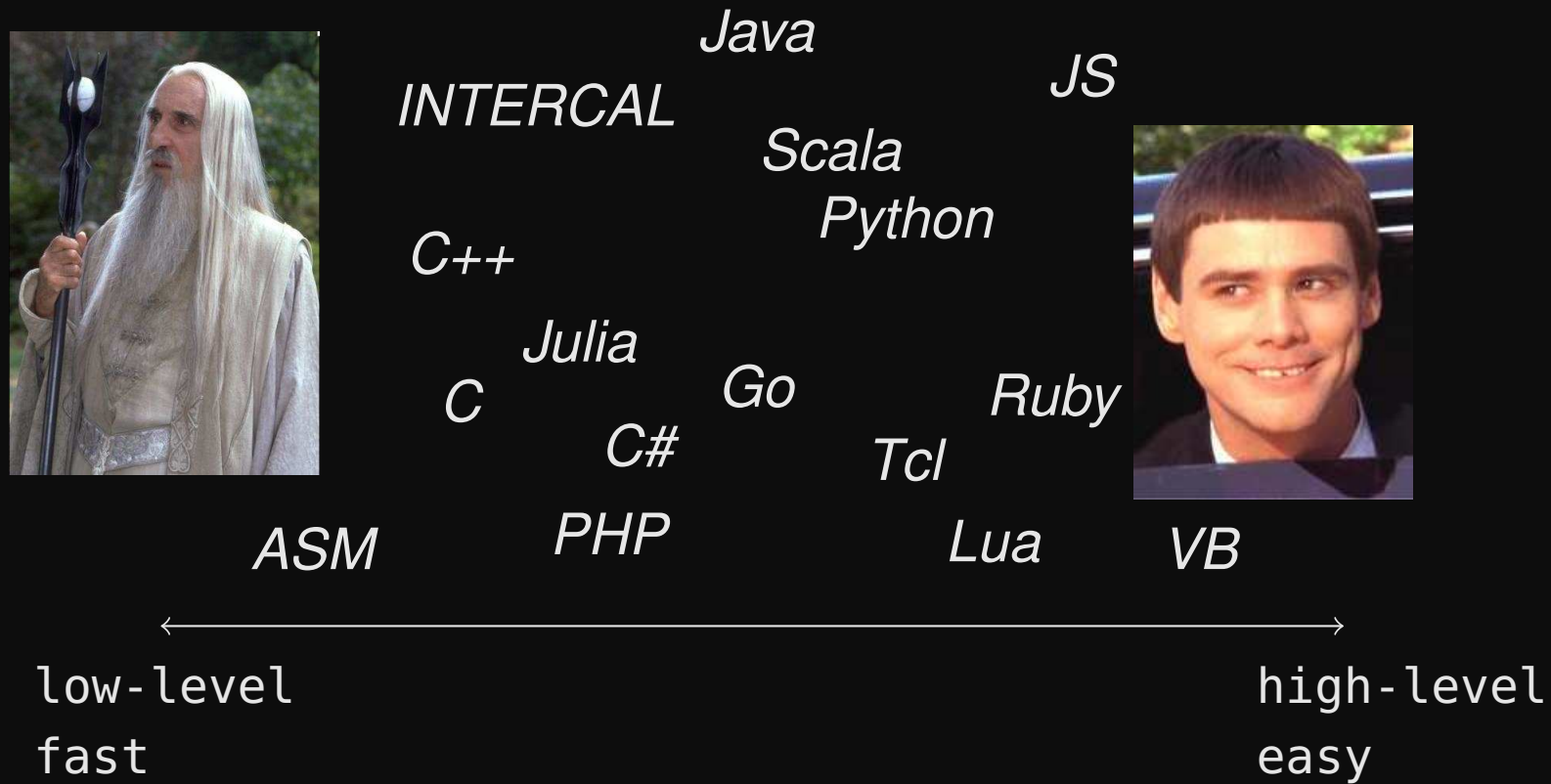
# Graph #1



**Note:** this is not a scientific or particularly accurate representation.

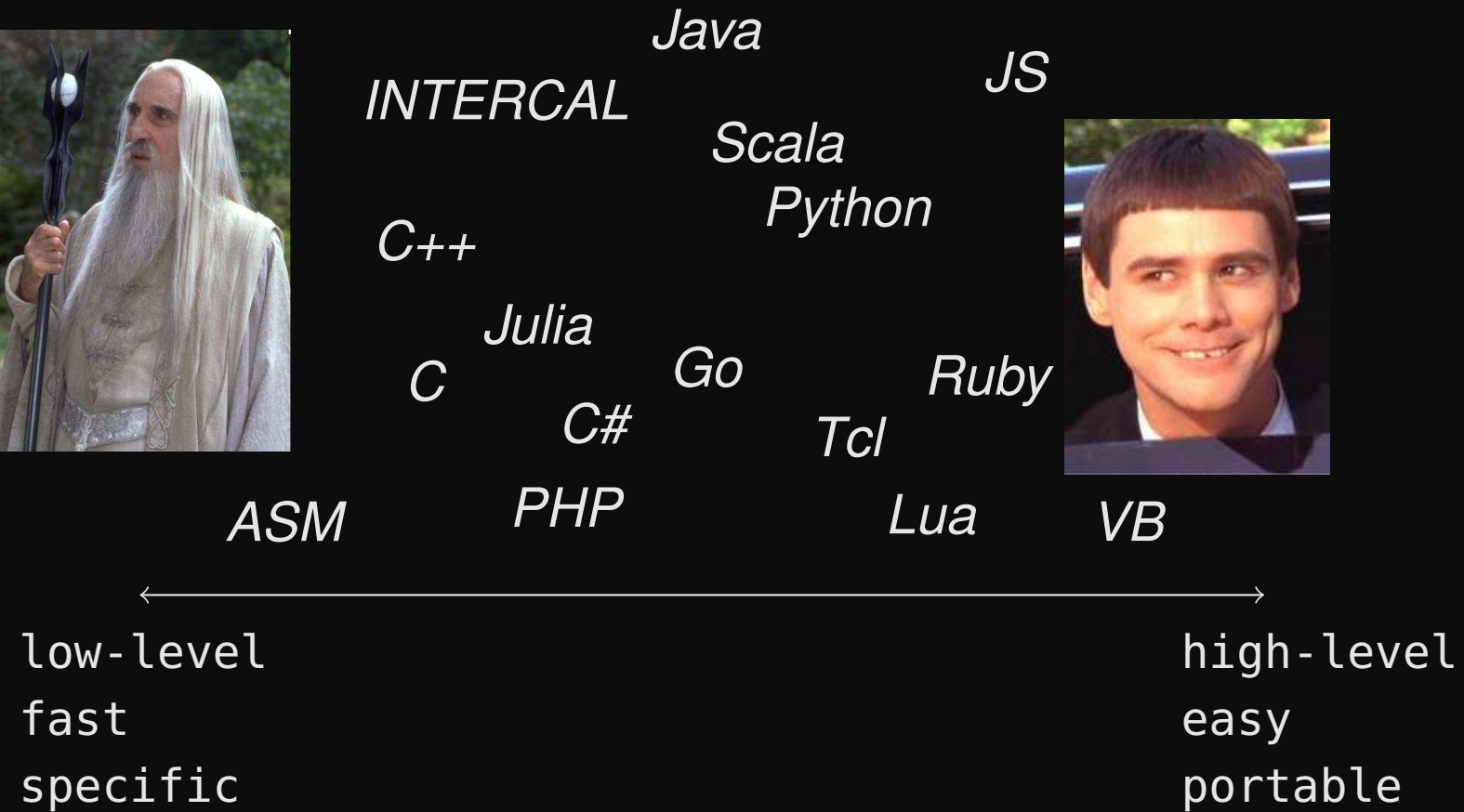


# Graph #1



**Note:** this is not a scientific or particularly accurate representation.

# Graph #1



**Note:** this is not a scientific or particularly accurate representation.

# The Big Tradeoff

**speed vs. portability and readability**


# The Big Tradeoff

**speed vs. portability and readability**




# The Big Tradeoff

**speed vs. portability and readability**



If we're careful, we can get speed, portability, *and* readability by using C++.

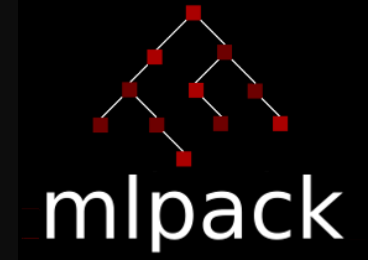


**So, mlpack.**

What is it?



# So, mlpack.



What is it?

- a fast general-purpose C++ machine learning library
- contains flexible implementations of common and cutting-edge machine learning algorithms
- for fast or big runs on single workstations
- bindings are available for R, Python, and the command line, and are coming for other languages (Julia, Go, Java?)
  
- 100+ developers from around the world
- frequent participation in the Google Summer of Code program

# So, mlpack.



What is it?

- a fast general-purpose C++ machine learning library
- contains flexible implementations of common and cutting-edge machine learning algorithms
- for fast or big runs on single workstations
- bindings are available for R, Python, and the command line, and are coming for other languages (Julia, Go, Java?)
  
- 100+ developers from around the world
- frequent participation in the Google Summer of Code program

<http://www.mlpack.org/>

<https://github.com/mlpack/mlpack/>

R.R. Curtin, M. Edel, M. Lozhnikov, Y. Mentekidis, S. Ghaisas, S. Zhang. “**mlpack 3**: a fast, flexible machine learning library”, in *The Journal of Open Source Software*, vol. 3, issue 26, 2018.

R.R. Curtin, J.R. Cline, N.P. Slagle, W.B. March, P. Ram, N.A. Mehta, A.G. Gray, “**mlpack**: a scalable C++ machine learning library”, in *The Journal of Machine Learning Research*, vol. 14, p. 801–805, 2013.



# What does mlpack implement?

mlpack implements a lot of standard machine learning techniques and also new, cutting-edge techniques.

## Classification

Naive Bayes Classifier  
ID3  
Decision Stumps  
Hidden Markov Models  
Perceptrons  
Softmax Regression  
Logistic Regression  
Decision Trees  
Deep Learning  
Q Learning  
Random Forests  
Sparse SVM  
AdaBoost.MH  
Reinforcement Learning  
Hoeffding Trees

## Regression

Collaborative Filtering  
Deep Learning  
Linear Regression  
LARS  
HMM Regression

## Distance-Based Techniques

Kernel PCA  
Rank-Approximate kNN  
Nystroem Method  
Range Search  
EMST  
Sparse Coding  
Locality-Sensitive Hashing  
PCA  
k-Nearest-Neighbor Search  
Density Estimation  
Trees  
NCA  
k-Furthest-Neighbor Search  
Max-Kernel Search  
Local Coordinate Coding  
Approximate KFN  
Sparse Autoencoder

## Other Tools

Randomized SVD  
Matrix Completion  
Hyper-Parameter Tuner  
Preprocessing Utilities  
Non-Negative Matrix Factorization  
QUIC-SVD  
Regularized SVD  
Optimization Toolkit  
Collaborative Filtering  
Incremental SVD

## Clustering

k-means  
DBSCAN  
Gaussian Mixture Models  
Mean Shift

# How do we get mlpack?

Linux (Debian/Ubuntu): `$ sudo apt-get install libmlpack-dev`

Linux (Red Hat/Fedora): `$ sudo dnf install mlpack-devel`

OS X (Homebrew): `$ brew tap brewsci/science &&`  
`brew install mlpack`

Windows (nuget): `> nuget add mlpack-windows`

# How do we get mlpack?

Linux (Debian/Ubuntu): `$ sudo apt-get install libmlpack-dev`

Linux (Red Hat/Fedora): `$ sudo dnf install mlpack-devel`

OS X (Homebrew): `$ brew tap brewsci/science &&`  
`brew install mlpack`

Windows (nuget): `> nuget add mlpack-windows`

Or install from source:

```
$ git clone https://github.com/mlpack/mlpack
$ mkdir mlpack/build && cd mlpack/build
$ cmake ../
$ make -j8 # Probably good to use many cores.
$ sudo make install
```

<https://www.mlpack.org/doc/mlpack-3.1.0/doxygen/build.html>

<https://keon.io/mlpack/mlpack-on-windows/>

# Installing from Python

Use pip:

```
$ pip install mlpack3
```

Or use conda:

```
$ conda install -c mlpack mlpack
```

# Command-line programs

You don't need to be a C++ expert.

```
# Train AdaBoost model.
$ mlpack_adaboost -t training_file.h5 -l training_labels.h5 \
> -M trained_model.bin
# Predict with AdaBoost model.
$ mlpack_adaboost -m trained_model.bin -T test_set.csv \
> -o test_set_predictions.csv
```

# Command-line programs

You don't need to be a C++ expert.

```
# Train AdaBoost model.
```

```
$ mlpack_adaboost -t training_file.h5 -l training_labels.h5 \  
> -M trained_model.bin
```

```
# Predict with AdaBoost model.
```

```
$ mlpack_adaboost -m trained_model.bin -T test_set.csv \  
> -o test_set_predictions.csv
```

```
# Find the 5 nearest neighbors of the data in dataset.txt, storing the  
# indices of the neighbors in 'neighbors.csv'.
```

```
$ mlpack_knn -r dataset.txt -k 5 -n neighbors.csv
```

# Command-line programs

You don't need to be a C++ expert.

```
# Train AdaBoost model.
```

```
$ mlpack_adaboost -t training_file.h5 -l training_labels.h5 \  
> -M trained_model.bin
```

```
# Predict with AdaBoost model.
```

```
$ mlpack_adaboost -m trained_model.bin -T test_set.csv \  
> -o test_set_predictions.csv
```

```
# Find the 5 nearest neighbors of the data in dataset.txt, storing the  
# indices of the neighbors in 'neighbors.csv'.
```

```
$ mlpack_knn -r dataset.txt -k 5 -n neighbors.csv
```

```
# Impute missing values ("NULL") in the input dataset to the  
# mean in that dimension.
```

```
$ mlpack_preprocess_imputer -i dataset.h5 -s mean -o imputed.h5
```

# Python bindings

Can be dropped directly into a Python workflow.

```
>>>
```



# Python bindings

Can be dropped directly into a Python workflow.

```
>>> import numpy as np
```

# Python bindings

Can be dropped directly into a Python workflow.

```
>>> import numpy as np
```

```
>>>
```

# Python bindings

Can be dropped directly into a Python workflow.

```
>>> import numpy as np
```

```
>>> from mlpack import pca
```

# Python bindings

Can be dropped directly into a Python workflow.

```
>>> import numpy as np
```

```
>>> from mlpack import pca
```

```
>>>
```

# Python bindings

Can be dropped directly into a Python workflow.

```
>>> import numpy as np
>>> from mlpack import pca
>>> x = np.genfromtxt('my_data.csv', delimiter=',')
```

# Python bindings

Can be dropped directly into a Python workflow.

```
>>> import numpy as np
>>> from mlpack import pca
>>> x = np.genfromtxt('my_data.csv', delimiter=',')
>>>
```

# Python bindings

Can be dropped directly into a Python workflow.

```
>>> import numpy as np
>>> from mlpack import pca
>>> x = np.genfromtxt('my_data.csv', delimiter=',')
>>> x.shape
```

# Python bindings

Can be dropped directly into a Python workflow.

```
>>> import numpy as np
>>> from mlpack import pca
>>> x = np.genfromtxt('my_data.csv', delimiter=',')
>>> x.shape
(2048, 10)
>>>
```



# Python bindings

Can be dropped directly into a Python workflow.

```
>>> import numpy as np
>>> from mlpack import pca
>>> x = np.genfromtxt('my_data.csv', delimiter=',')
>>> x.shape
(2048, 10)
>>> result = pca(input=x, new_dimensionality=5, verbose=True)
```

# Python bindings

Can be dropped directly into a Python workflow.

```
>>> import numpy as np
>>> from mlpack import pca
>>> x = np.genfromtxt('my_data.csv', delimiter=',')
>>> x.shape
(2048, 10)
>>> result = pca(input=x, new_dimensionality=5, verbose=True)
[INFO ] Performing PCA on dataset...
```

# Python bindings

Can be dropped directly into a Python workflow.

```
>>> import numpy as np
>>> from mlpack import pca
>>> x = np.genfromtxt('my_data.csv', delimiter=',')
>>> x.shape
(2048, 10)
>>> result = pca(input=x, new_dimensionality=5, verbose=True)
[INFO ] Performing PCA on dataset...
[INFO ] 99.9491% of variance retained (5 dimensions).
```

# Python bindings

Can be dropped directly into a Python workflow.

```
>>> import numpy as np
>>> from mlpack import pca
>>> x = np.genfromtxt('my_data.csv', delimiter=',')
>>> x.shape
(2048, 10)
>>> result = pca(input=x, new_dimensionality=5, verbose=True)
[INFO ] Performing PCA on dataset...
[INFO ] 99.9491% of variance retained (5 dimensions).
>>>
```

# Python bindings

Can be dropped directly into a Python workflow.

```
>>> import numpy as np
>>> from mlpack import pca
>>> x = np.genfromtxt('my_data.csv', delimiter=',')
>>> x.shape
(2048, 10)
>>> result = pca(input=x, new_dimensionality=5, verbose=True)
[INFO ] Performing PCA on dataset...
[INFO ] 99.9491% of variance retained (5 dimensions).
>>> result['output'].shape
```

# Python bindings

Can be dropped directly into a Python workflow.

```
>>> import numpy as np
>>> from mlpack import pca
>>> x = np.genfromtxt('my_data.csv', delimiter=',')
>>> x.shape
(2048, 10)
>>> result = pca(input=x, new_dimensionality=5, verbose=True)
[INFO ] Performing PCA on dataset...
[INFO ] 99.9491% of variance retained (5 dimensions).
>>> result['output'].shape
(2048, 5)
>>>
```

# Python bindings

A simple example: collaborative filtering for item recommendations.

```
>>>
```

# Python bindings

A simple example: collaborative filtering for item recommendations.

```
>>> import numpy as np
```



# Python bindings

A simple example: collaborative filtering for item recommendations.

```
>>> import numpy as np
```

```
>>>
```

# Python bindings

A simple example: collaborative filtering for item recommendations.

```
>>> import numpy as np
```

```
>>> from mlpack import cf
```

# Python bindings

A simple example: collaborative filtering for item recommendations.

```
>>> import numpy as np
>>> from mlpack import cf
>>>
```

# Python bindings

A simple example: collaborative filtering for item recommendations.

```
>>> import numpy as np
```

```
>>> from mlpack import cf
```

```
>>> x = np.genfromtxt('GroupLens100k.csv', delimiter=',')
```

# Python bindings

A simple example: collaborative filtering for item recommendations.

```
>>> import numpy as np
>>> from mlpack import cf
>>> x = np.genfromtxt('GroupLens100k.csv', delimiter=',')
>>>
```

# Python bindings

A simple example: collaborative filtering for item recommendations.

```
>>> import numpy as np
>>> from mlpack import cf
>>> x = np.genfromtxt('GroupLens100k.csv', delimiter=',')
>>> x.shape
```

# Python bindings

A simple example: collaborative filtering for item recommendations.

```
>>> import numpy as np
>>> from mlpack import cf
>>> x = np.genfromtxt('GroupLens100k.csv', delimiter=',')
>>> x.shape
(100000, 3)
>>>
```

# Python bindings

A simple example: collaborative filtering for item recommendations.

```
>>> import numpy as np
>>> from mlpack import cf
>>> x = np.genfromtxt('GroupLens100k.csv', delimiter=',')
>>> x.shape
(100000, 3)
>>> help(cf)
```



Help on built-in function cf in module mlpack.cf:

cf(...)

### Collaborative Filtering

This program performs collaborative filtering (CF) on the given dataset. Given a list of user, item and preferences (the 'training' parameter), the program will perform a matrix decomposition and then can perform a series of actions related to collaborative filtering. Alternately, the program can load an existing saved CF model with the 'input\_model' parameter and then use that model to provide recommendations or predict values.

The input matrix should be a 3-dimensional matrix of ratings, where the first dimension is the user, the second dimension is the item, and the third dimension is that user's rating of that item. Both the users and items should be numeric indices, not names. The indices are assumed to start from 0.

A set of query users for which recommendations can be generated may be specified with the 'query' parameter; alternately, recommendations may be generated for every user in the dataset by specifying the 'all\_user\_recommendations' parameter. In addition, the number of recommendations per user to generate can be specified with the 'recommendations' parameter, and the number of similar users (the size of the neighborhood) to be considered when generating recommendations can be specified with the 'neighborhood' parameter.

For performing the matrix decomposition, the following optimization algorithms can be specified via the 'algorithm' parameter:

'RegSVD' -- Regularized SVD using a SGD optimizer

NMF Non-negative matrix factorization with alternating least squares  
update rules  
'BatchSVD' -- SVD batch learning  
'SVDIncompleteIncremental' -- SVD incomplete incremental learning  
'SVDCompleteIncremental' -- SVD complete incremental learning  
A trained model may be saved to with the 'output\_model' output parameter.

To train a CF model on a dataset 'training\_set' using NMF for decomposition and saving the trained model to 'model', one could call:

```
>>> cf(training=training_set, algorithm='NMF')  
>>> model = output['output_model']
```

Then, to use this model to generate recommendations for the list of users in the query set 'users', storing 5 recommendations in 'recommendations', one could call

```
>>> cf(input_model=model, query=users, recommendations=5)  
>>> recommendations = output['output']
```

Input parameters:

- algorithm (string): Algorithm used for matrix factorization. Default value 'NMF'.
- all\_user\_recommendations (bool): Generate recommendations for all users.
- copy\_all\_inputs (bool): If specified, all input parameters will be deep copied before the method is run. This is useful for debugging problems where the input parameters are being modified by the algorithm, but can slow down the code.
- input\_model (CFTType): Trained CF model to load.

```
(100000, 3)
```

```
>>> help(cf)
```

```
>>>
```

```
(100000, 3)
```

```
>>> help(cf)
```

```
>>> output = cf(training=x, algorithm='NMF', verbose=True)
```

```
(100000, 3)
```

```
>>> help(cf)
```

```
>>> output = cf(training=x, algorithm='NMF', verbose=True)
```

```
[INFO ] Performing CF matrix decomposition on dataset...
```

```
(100000, 3)
```

```
>>> help(cf)
```

```
>>> output = cf(training=x, algorithm='NMF', verbose=True)
```

```
[INFO ] Performing CF matrix decomposition on dataset...
```

```
[INFO ] No rank given for decomposition; using rank of 11 calculated by  
density-based heuristic.
```

```
(100000, 3)
```

```
>>> help(cf)
```

```
>>> output = cf(training=x, algorithm='NMF', verbose=True)
```

```
[INFO ] Performing CF matrix decomposition on dataset...
```

```
[INFO ] No rank given for decomposition; using rank of 11 calculated by  
density-based heuristic.
```

```
[INFO ] Initialized W and H.
```

```
(100000, 3)
```

```
>>> help(cf)
```

```
>>> output = cf(training=x, algorithm='NMF', verbose=True)
```

```
[INFO ] Performing CF matrix decomposition on dataset...
```

```
[INFO ] No rank given for decomposition; using rank of 11 calculated by  
density-based heuristic.
```

```
[INFO ] Initialized W and H.
```

```
[INFO ] Iteration 1; residue 0.710812.
```



```
(100000, 3)
```

```
>>> help(cf)
```

```
>>> output = cf(training=x, algorithm='NMF', verbose=True)
```

```
[INFO ] Performing CF matrix decomposition on dataset...
```

```
[INFO ] No rank given for decomposition; using rank of 11 calculated by  
density-based heuristic.
```

```
[INFO ] Initialized W and H.
```

```
[INFO ] Iteration 1; residue 0.710812.
```

```
[INFO ] Iteration 2; residue 0.0627744.
```

```
(100000, 3)
```

```
>>> help(cf)
```

```
>>> output = cf(training=x, algorithm='NMF', verbose=True)
```

```
[INFO ] Performing CF matrix decomposition on dataset...
```

```
[INFO ] No rank given for decomposition; using rank of 11 calculated by  
density-based heuristic.
```

```
[INFO ] Initialized W and H.
```

```
[INFO ] Iteration 1; residue 0.710812.
```

```
[INFO ] Iteration 2; residue 0.0627744.
```

```
[INFO ] Iteration 3; residue 0.156398.
```

```
(100000, 3)
```

```
>>> help(cf)
```

```
>>> output = cf(training=x, algorithm='NMF', verbose=True)
```

```
[INFO ] Performing CF matrix decomposition on dataset...
```

```
[INFO ] No rank given for decomposition; using rank of 11 calculated by  
density-based heuristic.
```

```
[INFO ] Initialized W and H.
```

```
[INFO ] Iteration 1; residue 0.710812.
```

```
[INFO ] Iteration 2; residue 0.0627744.
```

```
[INFO ] Iteration 3; residue 0.156398.
```

```
...
```

```
(100000, 3)
```

```
>>> help(cf)
```

```
>>> output = cf(training=x, algorithm='NMF', verbose=True)
```

```
[INFO ] Performing CF matrix decomposition on dataset...
```

```
[INFO ] No rank given for decomposition; using rank of 11 calculated by  
density-based heuristic.
```

```
[INFO ] Initialized W and H.
```

```
[INFO ] Iteration 1; residue 0.710812.
```

```
[INFO ] Iteration 2; residue 0.0627744.
```

```
[INFO ] Iteration 3; residue 0.156398.
```

```
...
```

```
[INFO ] Iteration 26; residue 5.93531e-06.
```

```
(100000, 3)
>>> help(cf)
>>> output = cf(training=x, algorithm='NMF', verbose=True)
[INFO ] Performing CF matrix decomposition on dataset...
[INFO ] No rank given for decomposition; using rank of 11 calculated by
density-based heuristic.
[INFO ] Initialized W and H.
[INFO ] Iteration 1; residue 0.710812.
[INFO ] Iteration 2; residue 0.0627744.
[INFO ] Iteration 3; residue 0.156398.
...
[INFO ] Iteration 26; residue 5.93531e-06.
[INFO ] AMF converged to residue of 5.93531e-06 in 26 iterations.
```

```
(100000, 3)
>>> help(cf)
>>> output = cf(training=x, algorithm='NMF', verbose=True)
[INFO ] Performing CF matrix decomposition on dataset...
[INFO ] No rank given for decomposition; using rank of 11 calculated by
density-based heuristic.
[INFO ] Initialized W and H.
[INFO ] Iteration 1; residue 0.710812.
[INFO ] Iteration 2; residue 0.0627744.
[INFO ] Iteration 3; residue 0.156398.
...
[INFO ] Iteration 26; residue 5.93531e-06.
[INFO ] AMF converged to residue of 5.93531e-06 in 26 iterations.
>>>
```

```
(100000, 3)
>>> help(cf)
>>> output = cf(training=x, algorithm='NMF', verbose=True)
[INFO ] Performing CF matrix decomposition on dataset...
[INFO ] No rank given for decomposition; using rank of 11 calculated by
density-based heuristic.
[INFO ] Initialized W and H.
[INFO ] Iteration 1; residue 0.710812.
[INFO ] Iteration 2; residue 0.0627744.
[INFO ] Iteration 3; residue 0.156398.
...
[INFO ] Iteration 26; residue 5.93531e-06.
[INFO ] AMF converged to residue of 5.93531e-06 in 26 iterations.
>>> model = output['output_model']
```

```
(100000, 3)
>>> help(cf)
>>> output = cf(training=x, algorithm='NMF', verbose=True)
[INFO ] Performing CF matrix decomposition on dataset...
[INFO ] No rank given for decomposition; using rank of 11 calculated by
density-based heuristic.
[INFO ] Initialized W and H.
[INFO ] Iteration 1; residue 0.710812.
[INFO ] Iteration 2; residue 0.0627744.
[INFO ] Iteration 3; residue 0.156398.
...
[INFO ] Iteration 26; residue 5.93531e-06.
[INFO ] AMF converged to residue of 5.93531e-06 in 26 iterations.
>>> model = output['output_model']
>>>
```



```
(100000, 3)
>>> help(cf)
>>> output = cf(training=x, algorithm='NMF', verbose=True)
[INFO ] Performing CF matrix decomposition on dataset...
[INFO ] No rank given for decomposition; using rank of 11 calculated by
density-based heuristic.
[INFO ] Initialized W and H.
[INFO ] Iteration 1; residue 0.710812.
[INFO ] Iteration 2; residue 0.0627744.
[INFO ] Iteration 3; residue 0.156398.
...
[INFO ] Iteration 26; residue 5.93531e-06.
[INFO ] AMF converged to residue of 5.93531e-06 in 26 iterations.
>>> model = output['output_model']
>>> result = cf(input_model=model, query=[[1]],
               recommendations=3, verbose=True)
```

```
(100000, 3)
>>> help(cf)
>>> output = cf(training=x, algorithm='NMF', verbose=True)
[INFO ] Performing CF matrix decomposition on dataset...
[INFO ] No rank given for decomposition; using rank of 11 calculated by
density-based heuristic.
[INFO ] Initialized W and H.
[INFO ] Iteration 1; residue 0.710812.
[INFO ] Iteration 2; residue 0.0627744.
[INFO ] Iteration 3; residue 0.156398.
...
[INFO ] Iteration 26; residue 5.93531e-06.
[INFO ] AMF converged to residue of 5.93531e-06 in 26 iterations.
>>> model = output['output_model']
>>> result = cf(input_model=model, query=[[1]],
               recommendations=3, verbose=True)
[INFO ] Generating recommendations for 1 user.
[INFO ] 41 node combinations were scored.
[INFO ] 40 base cases were calculated.
```

```
(100000, 3)
>>> help(cf)
>>> output = cf(training=x, algorithm='NMF', verbose=True)
[INFO ] Performing CF matrix decomposition on dataset...
[INFO ] No rank given for decomposition; using rank of 11 calculated by
density-based heuristic.
[INFO ] Initialized W and H.
[INFO ] Iteration 1; residue 0.710812.
[INFO ] Iteration 2; residue 0.0627744.
[INFO ] Iteration 3; residue 0.156398.
...
[INFO ] Iteration 26; residue 5.93531e-06.
[INFO ] AMF converged to residue of 5.93531e-06 in 26 iterations.
>>> model = output['output_model']
>>> result = cf(input_model=model, query=[[1]],
                recommendations=3, verbose=True)
[INFO ] Generating recommendations for 1 user.
[INFO ] 41 node combinations were scored.
[INFO ] 40 base cases were calculated.
>>>
```

```
(100000, 3)
>>> help(cf)
>>> output = cf(training=x, algorithm='NMF', verbose=True)
[INFO ] Performing CF matrix decomposition on dataset...
[INFO ] No rank given for decomposition; using rank of 11 calculated by
density-based heuristic.
[INFO ] Initialized W and H.
[INFO ] Iteration 1; residue 0.710812.
[INFO ] Iteration 2; residue 0.0627744.
[INFO ] Iteration 3; residue 0.156398.
...
[INFO ] Iteration 26; residue 5.93531e-06.
[INFO ] AMF converged to residue of 5.93531e-06 in 26 iterations.
>>> model = output['output_model']
>>> result = cf(input_model=model, query=[[1]],
               recommendations=3, verbose=True)
[INFO ] Generating recommendations for 1 user.
[INFO ] 41 node combinations were scored.
[INFO ] 40 base cases were calculated.
>>> print(result['output'])
```

```
(100000, 3)
>>> help(cf)
>>> output = cf(training=x, algorithm='NMF', verbose=True)
[INFO ] Performing CF matrix decomposition on dataset...
[INFO ] No rank given for decomposition; using rank of 11 calculated by
density-based heuristic.
[INFO ] Initialized W and H.
[INFO ] Iteration 1; residue 0.710812.
[INFO ] Iteration 2; residue 0.0627744.
[INFO ] Iteration 3; residue 0.156398.
...
[INFO ] Iteration 26; residue 5.93531e-06.
[INFO ] AMF converged to residue of 5.93531e-06 in 26 iterations.
>>> model = output['output_model']
>>> result = cf(input_model=model, query=[[1]],
               recommendations=3, verbose=True)
[INFO ] Generating recommendations for 1 user.
[INFO ] 41 node combinations were scored.
[INFO ] 40 base cases were calculated.
>>> print(result['output'])
[[123 8 136]]
```

## From the command line

Actually, we could have done the exact same thing from the command line:

```
$ mlpack_cf -t GroupLens100k.csv -M model.bin -a NMF
```

# From the command line

Actually, we could have done the exact same thing from the command line:

```
$ mlpack_cf -t GroupLens100k.csv -M model.bin -a NMF  
$ mlpack_cf -m model.bin -q query.csv -c 3 -o recs.csv
```

# From the command line

Actually, we could have done the exact same thing from the command line:

```
$ mlpack_cf -t GroupLens100k.csv -M model.bin -a NMF  
$ mlpack_cf -m model.bin -q query.csv -c 3 -o recs.csv  
$ cat recs.csv
```



# From the command line

Actually, we could have done the exact same thing from the command line:

```
$ mlpack_cf -t GroupLens100k.csv -M model.bin -a NMF
$ mlpack_cf -m model.bin -q query.csv -c 3 -o recs.csv
$ cat recs.csv
123, 8, 136
```

# From the command line

Actually, we could have done the exact same thing from the command line:

```
$ mlpack_cf -t GroupLens100k.csv -M model.bin -a NMF
$ mlpack_cf -m model.bin -q query.csv -c 3 -o recs.csv
$ cat recs.csv
123, 8, 136
```

**Basically all mlpack algorithm bindings to the command-line, Python, or other languages operate like this.**

# Pros of C++

C++ is great!

# Pros of C++

C++ is great!

- Generic programming *at compile time* via templates.

# Pros of C++

C++ is great!

- Generic programming *at compile time* via templates.
- Low-level memory management.

# Pros of C++

C++ is great!

- Generic programming *at compile time* via templates.
- Low-level memory management.
- Little to no runtime overhead.

# Pros of C++

C++ is great!

- Generic programming *at compile time* via templates.
- Low-level memory management.
- Little to no runtime overhead.
- Well-known!

# Pros of C++

C++ is great!

- Generic programming *at compile time* via templates.
- Low-level memory management.
- Little to no runtime overhead.
- Well-known!
- The Armadillo library gives us good linear algebra primitives.



# Pros of C++

C++ is great!

- Generic programming *at compile time* via templates.
- Low-level memory management.
- Little to no runtime overhead.
- Well-known!
- The Armadillo library gives us good linear algebra primitives.

```
using namespace arma;  
extern mat x, y;  
mat z = (x + y) * chol(x) + 3 * chol(y.t());
```

# Cons of C++

C++ is not great!

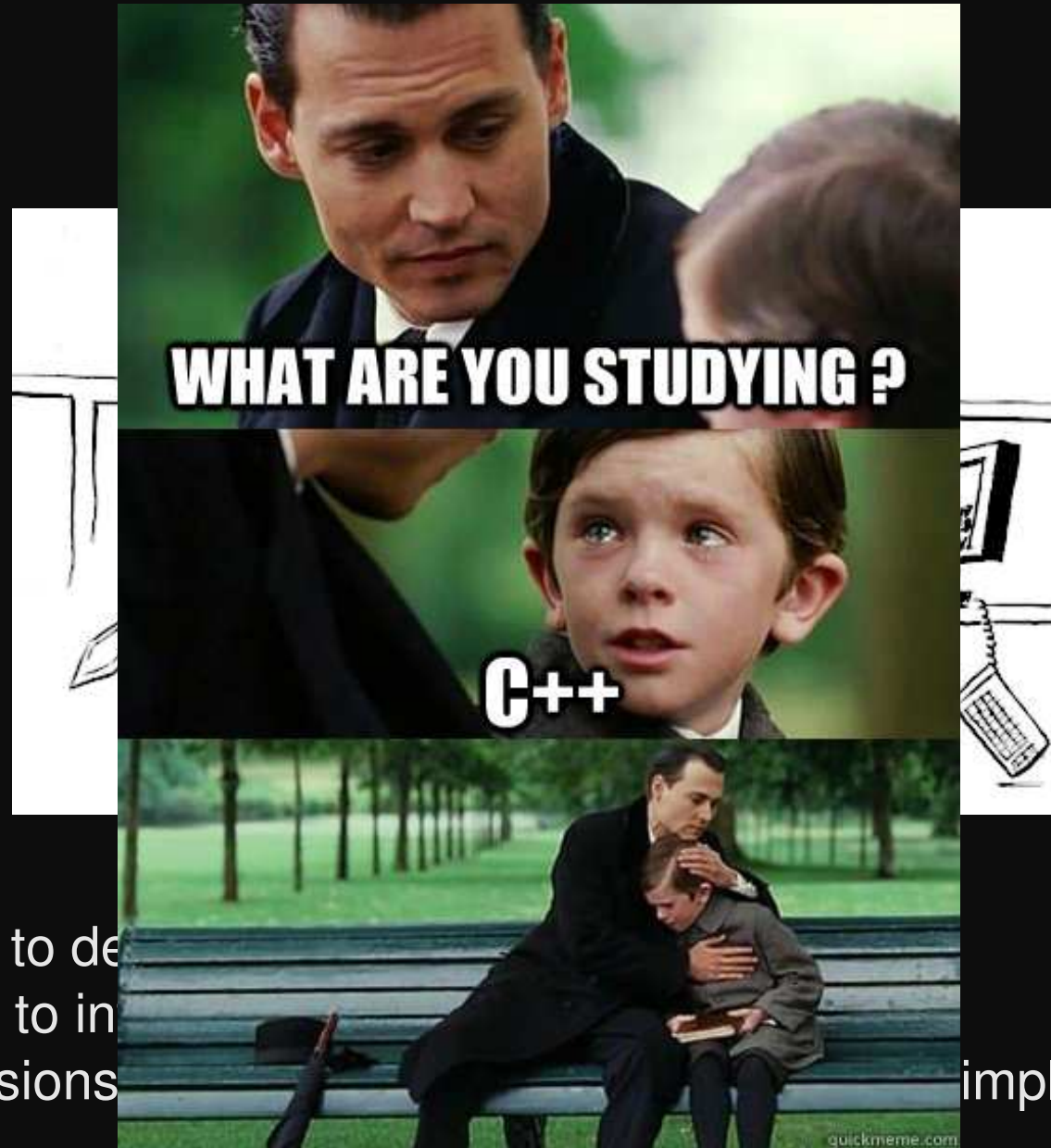






# Cons of C++

C++ is not great!



- Templates can be hard to de
- Memory bugs are easy to in
- The new language revisions

impler...

# Genericity

Why write an algorithm for one specific situation?

# Genericity

Why write an algorithm for one specific situation?

```
NearestNeighborSearch n(dataset);  
n.Search(query_set, 3, neighbors, distances);
```

What if I don't want the Euclidean distance?



# Genericity

Why write an algorithm for one specific situation?

```
// The numeric parameter is the value of p for the p-norm to  
// use. 1 = Manhattan distance, 2 = Euclidean distance, etc.  
NearestNeighborSearch n(dataset, 1);  
n.Search(query_set, 3, neighbors, distances);
```

Ok, this is a little better!

# Genericity

Why write an algorithm for one specific situation?

```
// ManhattanDistance is a class with a method Evaluate().  
NearestNeighborSearch<ManhattanDistance> n(dataset);  
n.Search(query_set, 3, neighbors, distances);
```

This is much better! The user can specify whatever distance metric they want, including one they write themselves.

# Genericity

Why write an algorithm for one specific situation?

```
// This will _definitely_ get me best paper at ICML! I can
// feel it!
class MyStupidDistance
{
    static double Evaluate(const arma::vec& a,
                          const arma::vec& b)
    {
        return 15.0 * std::abs(a[0] - b[0]);
    }
};

// Now we can use it!
NearestNeighborSearch<MyStupidDistance> n(dataset);
n.Search(query_set, 3, neighbors, distances);
```

# Genericity

Why write an algorithm for one specific situation?

```
// We can also use sparse matrices instead!  
NearestNeighborSearch<MyStupidDistance, arma::sp_mat>  
    n(sparse_dataset);  
n.Search(sparse_query_set, 3, neighbors, distances);
```

# Genericity

Why write an algorithm for one specific situation?

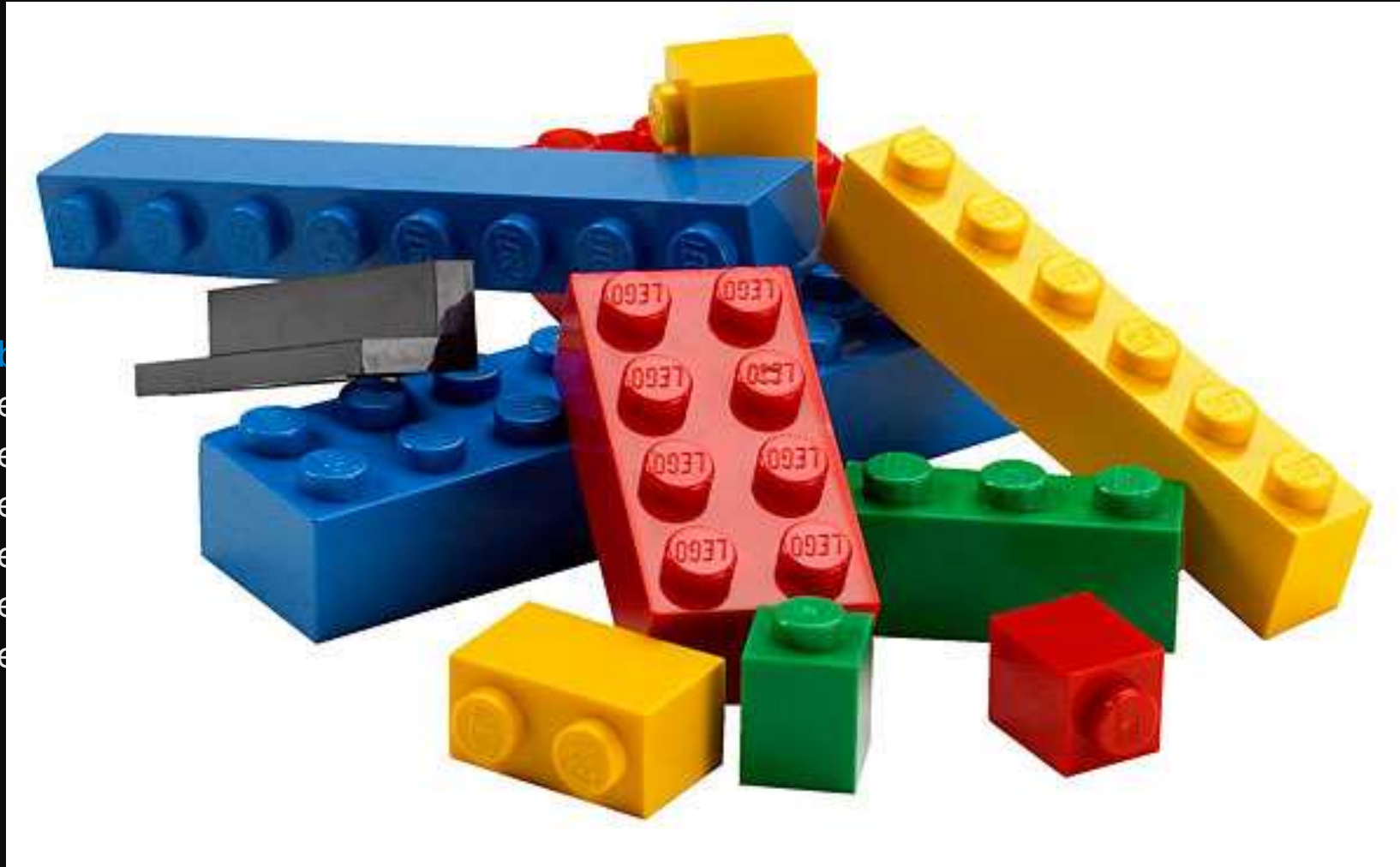
```
// Nearest neighbor search with arbitrary types of trees!  
NearestNeighborSearch<EuclideanDistance, arma::mat, KDTree> kn;  
NearestNeighborSearch<EuclideanDistance, arma::sp_mat, CoverTree> cn;  
NearestNeighborSearch<ManhattanDistance, arma::mat, Octree> on;  
NearestNeighborSearch<ChebyshevDistance, arma::sp_mat, RPlusTree> rn;  
NearestNeighborSearch<MahalanobisDistance, arma::mat, RPTree> rpn;  
NearestNeighborSearch<EuclideanDistance, arma::mat, XTree> xn;
```

R.R. Curtin, "Improving dual-tree algorithms". *PhD thesis, Georgia Institute of Technology, Atlanta, GA, 8/2015.*

# Genericity

Why write an algorithm for one specific situation?

```
// Nearest neighbor  
NearestNeighborSe  
NearestNeighborSe  
NearestNeighborSe  
NearestNeighborSe  
NearestNeighborSe  
NearestNeighborSe
```



R.R. Curtin, "Improving dual-tree algorithms". *PhD thesis, Georgia Institute of Technology, Atlanta, GA, 8/2015.*

# Genericity

Why write an algorithm for one specific situation?

```
// Nearest neighbor  
NearestNeighborSe  
NearestNeighborSe  
NearestNeighborSe  
NearestNeighborSe  
NearestNeighborSe  
NearestNeighborSe
```



Tampa, CA, 8/2013.

# Why templates?

What about virtual inheritance?



# Why templates?

What about virtual inheritance?

```
class MyStupidDistance : public Distance
{
    virtual double Evaluate(const arma::vec& a,
                           const arma::vec& b)
    {
        return 15.0 * std::abs(a[0] - b[0]);
    }
};
```

```
NearestNeighborSearch n(dataset, new MyStupidDistance());
n.Search(3, neighbors, distances);
```

# Why templates?

What about virtual inheritance?

```
class MyStupidDistance : public Distance
{
    virtual double Evaluate(const arma::vec& a,
                           const arma::vec& b)
    {
        return 15.0 * std::abs(a[0] - b[0]);
    }
};
```

```
NearestNeighborSearch n(dataset, new MyStupidDistance());
n.Search(3, neighbors, distances);
```

vtable lookup penalty!

# Why templates?

Using inheritance to call a function costs us instructions:

```
Distance* d =  
    new MyStupidDistance();  
d->Evaluate(a, b);
```

---

```
MyStupidDistance::Evaluate(a, b);
```

# Why templates?

Using inheritance to call a function costs us instructions:

```
Distance* d =  
    new MyStupidDistance();  
d->Evaluate(a, b);
```

```
; push stack pointer  
movq %rsp, %rdi  
; get location of function  
movq $_ZTV1A+16, (%rsp)  
; call Evaluate()  
call _ZN1A1aEd
```

```
MyStupidDistance::Evaluate(a, b);
```

```
; just call Evaluate()!  
call _ZN1B1aEd.isra.0.constprop.1
```

# Why templates?

Using inheritance to call a function costs us instructions:

```
Distance* d =  
    new MyStupidDistance();  
d->Evaluate(a, b);
```

```
; push stack pointer  
movq %rsp, %rdi  
; get location of function  
movq $_ZTV1A+16, (%rsp)  
; call Evaluate()  
call _ZN1A1aEd
```

```
MyStupidDistance::Evaluate(a, b);
```

```
; just call Evaluate()  
call _ZN1B1aEd.isra.0.constprop.1
```

Up to 10%+ performance penalty in some situations!

# Compile-time expressions

What about math? (Armadillo)



# Compile-time expressions

What about math? (Armadillo)

In C:

```
extern double** a, b, c, d, e;  
extern int rows, cols;
```

```
// We want to do  $e = a + b + c + d$ .  
mat_copy(e, a, rows, cols);  
mat_add(e, b, rows, cols);  
mat_add(e, c, rows, cols);  
mat_add(e, d, rows, cols);
```



# Compile-time expressions

What about math? (Armadillo)

In C with a custom function:

```
extern double** a, b, c, d, e;  
extern int rows, cols;
```

```
// We want to do e = a + b + c + d.
```





# Compile-time expressions

What about math? (Armadillo)

In C with a custom function:

```
extern double** a, b, c, d, e;  
extern int rows, cols;  
  
// We want to do  $e = a + b + c + d$ .  
mat_add4_into(e, a, b, c, d, rows, cols);
```

Fastest! (one pass)



# Compile-time expressions

What about math? (Armadillo)

In C with a custom function:

```
extern double** a, b, c, d, e;  
extern int rows, cols;
```

```
// We want to do e = a + b + c + d.  
mat_add4_into(e, a, b, c, d, rows, cols);
```

Fastest! (one pass)

```
void mat_add4_into(double** e, double** a, double** b,  
                  double** c, double** d, int rows, int cols)  
{  
    for (int r = 0; r < rows; ++r)  
        for (int c = 0; c < cols; ++c)  
            e[r][c] = a[r][c] + b[r][c] + c[r][c] + d[r][c];  
}
```



# Compile-time expressions

What about math? (Armadillo)

In MATLAB:

```
e = a + b + c + d
```



# Compile-time expressions

What about math? (Armadillo)

In MATLAB:

```
e = a + b + c + d
```

Beautiful!



# Compile-time expressions

What about math? (Armadillo)



# Compile-time expressions

What about math? (Armadillo)



# Compile-time expressions

What about math? (Armadillo)

In C++ (with Armadillo):

```
using namespace arma;  
extern mat a, b, c, d;  
  
mat e = a + b + c + d;
```

No temporaries, only one pass! Just as fast as the fastest C implementation.



# Compile-time expressions

What about math? (Armadillo)

In C++ (with Armadillo):

```
using namespace arma;  
extern mat a, b, c, d;
```

```
mat e = a + b + c + d;
```

C++ allows us templated operator overloading:

```
template<typename T1, typename T2>  
const op<T1, T2, add> operator+(const T1& x, const T2& y);
```





# Compile-time expressions

What about math? (Armadillo)

In C++ (with Armadillo):

```
using namespace arma;  
extern mat a, b, c, d;
```

```
mat e = a + b + c + d;
```

C++ allows us templated operator overloading:

```
template<typename T1, typename T2>  
const op<T1, T2, add> operator+(const T1& x, const T2& y);
```

- `mat + mat`  
→ `op<mat, mat, add>`



# Compile-time expressions

What about math? (Armadillo)

In C++ (with Armadillo):

```
using namespace arma;  
extern mat a, b, c, d;
```

```
mat e = a + b + c + d;
```

C++ allows us templated operator overloading:

```
template<typename T1, typename T2>  
const op<T1, T2, add> operator+(const T1& x, const T2& y);
```

- `mat + mat`  
→ `op<mat, mat, add>`
- `mat + mat + mat`  
→ `op<mat, mat, add> + mat`



# Compile-time expressions

What about math? (Armadillo)

In C++ (with Armadillo):

```
using namespace arma;  
extern mat a, b, c, d;
```

```
mat e = a + b + c + d;
```

C++ allows us templated operator overloading:

```
template<typename T1, typename T2>  
const op<T1, T2, add> operator+(const T1& x, const T2& y);
```

- `mat + mat`  
→ `op<mat, mat, add>`
- `mat + mat + mat`  
→ `op<op<mat, mat, add>, mat, add>`



# Compile-time expressions

What about math? (Armadillo)

In C++ (with Armadillo):

```
using namespace arma;  
extern mat a, b, c, d;
```

```
mat e = a + b + c + d;
```

C++ allows us templated operator overloading:

```
template<typename T1, typename T2>  
const op<T1, T2, add> operator+(const T1& x, const T2& y);
```

- `mat + mat`  
→ `op<mat, mat, add>`
- `mat + mat + mat`  
→ `op<op<mat, mat, add>, mat, add>`
- `mat + mat + mat + mat`  
→ `op<mat, mat, add> + mat + mat`



# Compile-time expressions

What about math? (Armadillo)

In C++ (with Armadillo):

```
using namespace arma;  
extern mat a, b, c, d;
```

```
mat e = a + b + c + d;
```

C++ allows us templated operator overloading:

```
template<typename T1, typename T2>  
const op<T1, T2, add> operator+(const T1& x, const T2& y);
```

- `mat + mat`  
→ `op<mat, mat, add>`
- `mat + mat + mat`  
→ `op<op<mat, mat, add>, mat, add>`
- `mat + mat + mat + mat`  
→ `op<op<mat, mat, add>, mat, add> + mat`



# Compile-time expressions

What about math? (Armadillo)

In C++ (with Armadillo):

```
using namespace arma;  
extern mat a, b, c, d;
```

```
mat e = a + b + c + d;
```

C++ allows us templated operator overloading:

```
template<typename T1, typename T2>  
const op<T1, T2, add> operator+(const T1& x, const T2& y);
```

- `mat + mat`  
→ `op<mat, mat, add>`
- `mat + mat + mat`  
→ `op<op<mat, mat, add>, mat, add>`
- `mat + mat + mat + mat`  
→ `op<op<op<mat, mat, add>, mat, add>, mat, add>`



# Compile-time expressions

What about math? (Armadillo)

In C++ (with Armadillo):

```
using namespace arma;  
extern mat a, b, c, d;
```

```
mat e = a + b + c + d;
```

C++ allows us templated operator overloading:

```
template<typename T1, typename T2>  
const op<T1, T2, add> operator+(const T1& x, const T2& y);
```

The expression yields type `op<op<op<mat, mat, add>, mat, add>, mat, add>`.

```
// This can accept an op<...> type.  
template<typename T1, typename T2>  
mat::operator=(const op<T1, T2, add>& op);
```



# Compile-time expressions

What about math? (Armadillo)

In C++ (with Armadillo):

```
using namespace arma;  
extern mat a, b, c, d;
```

```
mat e = a + b + c + d;
```

C++ allows us templated operator overloading:

```
template<typename T1, typename T2>  
const op<T1, T2, add> operator+(const T1& x, const T2& y);
```

The expression yields type `op<op<op<mat, mat, add>, mat, add>, mat, add>`.

```
// This can accept an op<...> type.  
template<typename T1, typename T2>  
mat::operator=(const op<T1, T2, add>& op);
```

The assignment operator "unwraps" the operation and generates optimal code.





# Take-home

- Templates give us generic code.
- Templates allow us to generate fast code.

# Optimization in C++ with ensmallen

Optimization is a fundamental machine learning problem:

$$\operatorname{argmin}_x f(x)$$

# Optimization in C++ with ensmallen

Optimization is a fundamental machine learning problem:

$$\operatorname{argmin}_x f(x)$$

mlpack provides some nice facilities to do this, via the new `ensmallen` library:

<https://github.com/mlpack/ensmallen>. In order to optimize a differentiable function we just need a class with two methods:

```
// Return the value of f(x).  
double Evaluate(const arma::mat& x);  
  
// Compute the gradient of f(x) with respect to x.  
void Gradient(const arma::mat& x, arma::mat& gradient);
```

# Optimization in C++ with ensmallen

Let's take linear regression as an example:

- $A$ : data matrix
- $b$ : data responses
- $x$ : parameters for linear regression

# Optimization in C++ with ensmallen

Let's take linear regression as an example:

- $A$ : data matrix
- $b$ : data responses
- $x$ : parameters for linear regression

$$f(x) = (Ax - b)^T (Ax - b).$$

And the gradient:

$$\nabla f(x) = A^T (Ax - b).$$

# Optimization in C++ with ensmallen

Let's take linear regression as an example:

- $A$ : data matrix
- $b$ : data responses
- $x$ : parameters for linear regression

$$f(x) = (Ax - b)^T (Ax - b).$$

And the gradient:

$$\nabla f(x) = A^T (Ax - b).$$

We want to minimize  $f(x)$ .

# Optimization in C++ with ensmallen

Let's take linear regression as an example:

- $A$ : data matrix
- $b$ : data responses
- $x$ : parameters for linear regression

$$f(x) = (Ax - b)^T (Ax - b).$$

And the gradient:

$$\nabla f(x) = A^T (Ax - b).$$

We want to minimize  $f(x)$ .

*(The point of the demo here is to show how easy it is to implement, not to detail the intricacies of linear regression, so don't worry about the math much.)*

# Optimization in C++ with ensmallen

Remember, we just need two functions inside of a class.

```
class LinearRegressionFunction  
{
```



# Optimization in C++ with ensmallen

Remember, we just need two functions inside of a class.

```
class LinearRegressionFunction
{
private:
    const arma::mat& data; // Store a reference to the data.
    const arma::rowvec& responses;

public:
    LinearRegressionFunction(const arma::mat& data, const arma::rowvec& responses) : data(data),
        responses(responses) { }
```

# Optimization in C++ with ensmallen

Remember, we just need two functions inside of a class.

```
class LinearRegressionFunction
{
private:
    const arma::mat& data; // Store a reference to the data.
    const arma::rowvec& responses;

public:
    LinearRegressionFunction(const arma::mat& data, const arma::rowvec& responses) : data(data),
        responses(responses) { }

    double Evaluate(const arma::mat& x)
    {
```

# Optimization in C++ with ensmallen

Remember, we just need two functions inside of a class.

```
class LinearRegressionFunction
{
private:
    const arma::mat& data; // Store a reference to the data.
    const arma::rowvec& responses;

public:
    LinearRegressionFunction(arma::mat& data, arma::rowvec& responses) : data(data),
        responses(responses) { }

    double Evaluate(const arma::mat& x)
    {
```

$$f(x) = (Ax - b)^T (Ax - b).$$

# Optimization in C++ with ensmallen

Remember, we just need two functions inside of a class.

```
class LinearRegressionFunction
{
private:
    const arma::mat& data; // Store a reference to the data.
    const arma::rowvec& responses;

public:
    LinearRegressionFunction(arma::mat& data, arma::rowvec& responses) : data(data),
        responses(responses) { }

    double Evaluate(const arma::mat& x)
    {
        return (data * x - responses).t() * (data * x - responses);
    }
}
```

$$f(x) = (Ax - b)^T (Ax - b).$$

# Optimization in C++ with ensmallen

Remember, we just need two functions inside of a class.

```
class LinearRegressionFunction
{
private:
    const arma::mat& data; // Store a reference to the data.
    const arma::rowvec& responses;

public:
    LinearRegressionFunction(const arma::mat& data, const arma::rowvec& responses) : data(data),
        responses(responses) { }

    double Evaluate(const arma::mat& x)
    {
        return (data * x - responses).t() * (data * x - responses);
    }

    void Gradient(const arma::mat& x, arma::mat& gradient)
    {
```

# Optimization in C++ with ensmallen

Remember, we just need two functions inside of a class.

```
class LinearRegressionFunction
{
private:
    const arma::mat& data; // Store a reference to the data.
    const arma::rowvec& responses;

public:
    LinearRegressionFunction(const arma::mat& data, const arma::rowvec& responses) : data(data),
        responses(responses) { }

    double Evaluate(const arma::mat& x)
    {
        return (data * x - responses).t() * (data * x - responses);
    }

    void Gradient(const arma::mat& x, arma::mat& gradient)
    {
```

$$\nabla f(x) = A^T (Ax - b).$$

# Optimization in C++ with ensmallen

Remember, we just need two functions inside of a class.

```
class LinearRegressionFunction
{
private:
    const arma::mat& data; // Store a reference to the data.
    const arma::rowvec& responses;

public:
    LinearRegressionFunction(arma::mat& data, arma::rowvec& responses) : data(data),
        responses(responses) { }

    double Evaluate(const arma::mat& x)
    {
        return (data * x - responses).t() * (data * x - responses);
    }

    void Gradient(const arma::mat& x, arma::mat& gradient)
    {
        gradient = data.t() * (data * x - responses);
    }
};
```

$$\nabla f(x) = A^T (Ax - b).$$

# Optimization in C++ with ensmallen

Remember, we just need two functions inside of a class.

```
class LinearRegressionFunction
{
private:
    const arma::mat& data; // Store a reference to the data.
    const arma::rowvec& responses;

public:
    LinearRegressionFunction(const arma::mat& data, const arma::rowvec& responses) : data(data),
        responses(responses) { }

    double Evaluate(const arma::mat& x)
    {
        return (data * x - responses).t() * (data * x - responses);
    }

    void Gradient(const arma::mat& x, arma::mat& gradient)
    {
        gradient = data.t() * (data * x - responses);
    }
};
```



# Optimization in C++ with ensmallen

Now we can take our `LinearRegressionFunction` and optimize it!

# Optimization in C++ with ensmallen

Now we can take our `LinearRegressionFunction` and optimize it!

```
using namespace mlpack::optimization;

// Create the function.
LinearRegressionFunction lrf(data, responses);

arma::mat x;
L_BFGS l; // Construct optimizer with default parameters.
l.Optimize(lrf, x); // Find the minimum of lrf and store the parameters in x.
```

# Optimization in C++ with ensmallen

Now we can take our `LinearRegressionFunction` and optimize it!

```
using namespace mlpack::optimization;

// Create the function.
LinearRegressionFunction lrf(data, responses);

arma::mat x;
L_BFGS l; // Construct optimizer with default parameters.
l.Optimize(lrf, x); // Find the minimum of lrf and store the parameters in x.

GradientDescent g;
g.Optimize(lrf, x);
```

# Optimization in C++ with ensmallen

Now we can take our `LinearRegressionFunction` and optimize it!

```
using namespace mlpack::optimization;

// Create the function.
LinearRegressionFunction lrf(data, responses);

arma::mat x;
L_BFGS l; // Construct optimizer with default parameters.
l.Optimize(lrf, x); // Find the minimum of lrf and store the parameters in x.

GradientDescent g;
g.Optimize(lrf, x);

SA s; // Simulated Annealing.
s.Optimize(lrf, x);
```

# Optimization in C++ with ensmallen

Now we can take our `LinearRegressionFunction` and optimize it!

```
using namespace mlpack::optimization;

// Create the function.
LinearRegressionFunction lrf(data, responses);

arma::mat x;
L_BFGS l; // Construct optimizer with default parameters.
l.Optimize(lrf, x); // Find the minimum of lrf and store the parameters in x.

GradientDescent g;
g.Optimize(lrf, x);

SA s; // Simulated Annealing.
s.Optimize(lrf, x);

IQN i;
i.Optimize(lrf, x);
```

# A wide range of optimizers for different problem types

ensmallen has a huge collection of optimizers.

- **Quasi-Newton variants:** Limited-memory BFGS (L-BFGS), incremental Quasi-Newton method (IQN), Augmented Lagrangian Method
- **SGD variants:** Stochastic Gradient Descent (SGD), Stochastic Coordinate Descent (SCD), Parallel Stochastic Gradient Descent (Hogwild!), Stochastic Gradient Descent with Restarts (SGDR), SMORMS3, AdaGrad, AdaDelta, RMSProp, Adam, AdaMax, Padam, Nadam, WNGrad, AMSGrad
- **Genetic variants:** Conventional Neuro-evolution (CNE), Covariance Matrix Adaptation Evolution Strategy (CMA-ES)
- **Other:** Conditional Gradient Descent, Frank-Wolfe algorithm, Simulated Annealing, SPSA

# A wide range of optimizers for different problem types

ensmallen has a huge collection of optimizers.

- **Quasi-Newton variants:** Limited-memory BFGS (L-BFGS), incremental Quasi-Newton method (IQN), Augmented Lagrangian Method
- **SGD variants:** Stochastic Gradient Descent (SGD), Stochastic Coordinate Descent (SCD), Parallel Stochastic Gradient Descent (Hogwild!), Stochastic Gradient Descent with Restarts (SGDR), SMORMS3, AdaGrad, AdaDelta, RMSProp, Adam, AdaMax, Padam, Nadam, WNGrad, AMSGrad
- **Genetic variants:** Conventional Neuro-evolution (CNE), Covariance Matrix Adaptation Evolution Strategy (CMA-ES)
- **Other:** Conditional Gradient Descent, Frank-Wolfe algorithm, Simulated Annealing, SPSA

And it is also easy to implement new optimizers. <https://ensmallen.org/>

S. Bhardwaj, R.R. Curtin, M. Edel, Y. Mentekidis, C. Sanderson, "ensmallen: a flexible C++ library for efficient function optimization", *Systems for ML Workshop at NeurIPS 2018*, 2018.

# Deep Neural Networks with mlpack

With ensmallen, we can do deep learning.



# Deep Neural Networks with mlpack

With ensmallen, we can do deep learning.

```
using namespace mlpack::ann;
extern arma::mat data, responses, testData;

// Create a 3-layer sigmoid neural network with 10 outputs.
FFN<NegativeLogLikelihood<>, RandomInitialization> model;
model.Add<Linear<>>(data.n_rows, 100);
model.Add<SigmoidLayer<>>();
model.Add<Linear<>>(100, 100);
model.Add<SigmoidLayer<>>();
model.Add<Linear<>>(100, 10);
model.Add<LogSoftMax<>>();
```

# Deep Neural Networks with mlpack

With ensmallen, we can do deep learning.

```
using namespace mlpack::ann;
extern arma::mat data, responses, testData;

// Create a 3-layer sigmoid neural network with 10 outputs.
FFN<NegativeLogLikelihood<>, RandomInitialization> model;
model.Add<Linear<>>(data.n_rows, 100);
model.Add<SigmoidLayer<>>();
model.Add<Linear<>>(100, 100);
model.Add<SigmoidLayer<>>();
model.Add<Linear<>>(100, 10);
model.Add<LogSoftMax<>>();

// Train the model.
SGD<> optimizer(0.001 /* step size */, 1024 /* batch size */,
               100000 /* max iterations */);
model.Train(data, responses, optimizer);
```

# Deep Neural Networks with mlpack

With ensmallen, we can do deep learning.

```
using namespace mlpack::ann;
extern arma::mat data, responses, testData;

// Create a 3-layer sigmoid neural network with 10 outputs.
FFN<NegativeLogLikelihood<>, RandomInitialization> model;
model.Add<Linear<>>(data.n_rows, 100);
model.Add<SigmoidLayer<>>>();
model.Add<Linear<>>(100, 100);
model.Add<SigmoidLayer<>>>();
model.Add<Linear<>>(100, 10);
model.Add<LogSoftMax<>>>();

// Train the model.
SGD<> optimizer(0.001 /* step size */, 1024 /* batch size */,
                100000 /* max iterations */);
model.Train(data, responses, optimizer);

// Predict on test points.
arma::mat predictions;
model.Predict(testData, predictions);
```

# Benchmarks

Did C++ get us what we wanted?

# Benchmarks

Task 1:  $z = 2(x' + y) + 2(x + y')$ .

```
extern int n;  
mat x(n, n, fill::randu);  
mat y(n, n, fill::randu);  
mat z = 2 * (x.t() + y) + 2 * (x + y.t()); // only time this line
```

$n$	arma	numpy	octave	R	Julia
1000	0.029s	0.040s	0.036s	0.052s	<b>0.027s</b>
3000	0.047s	0.432s	0.376s	0.344s	<b>0.041s</b>
10000	<b>0.968s</b>	5.948s	3.989s	4.952s	3.683s
30000	<b>19.167s</b>	62.748s	41.356s	<i>fail</i>	36.730s

# Benchmarks

Task 2:  $z = (x + 10 * I)^\dagger - y$ .

```
extern int n;  
mat x(n, n, fill::randu);  
mat y(n, n, fill::randu);  
mat z = pinv(x + 10 * eye(n, n)) - y; // only time this line
```

$n$	arma	numpy	octave	R	Julia
300	<b>0.081s</b>	<b>0.080s</b>	0.324s	0.096s	0.098s
1000	1.321s	1.354s	26.156s	1.444s	<b>1.236s</b>
3000	<b>28.817s</b>	28.955s	648.64s	29.732s	29.069s
10000	<b>777.55s</b>	785.58s	17661.9s	787.201s	778.472s

The computation is dominated by the calculation of the pseudoinverse.

# Benchmarks

Task 3:  $z = abcd$  for decreasing-size matrices.

```
extern int n;  
mat a(n, 0.8 * n, fill::randu);  
mat b(0.8 * n, 0.6 * n, fill::randu);  
mat c(0.6 * n, 0.4 * n, fill::randu);  
mat d(0.4 * n, 0.2 * n, fill::randu);  
mat z = a * b * c * d; // only time this line
```

$n$	arma	numpy	octave	R	Julia
1000	0.042s	0.051s	<b>0.033s</b>	0.056s	0.037s
3000	<b>0.642s</b>	0.812s	0.796s	0.846s	0.844s
10000	<b>16.320s</b>	26.815s	26.478s	26.957s	26.576s
30000	<b>329.87s</b>	708.16s	706.10s	707.12s	704.032s

Armadillo can automatically select the correct ordering for multiplication.

# Benchmarks

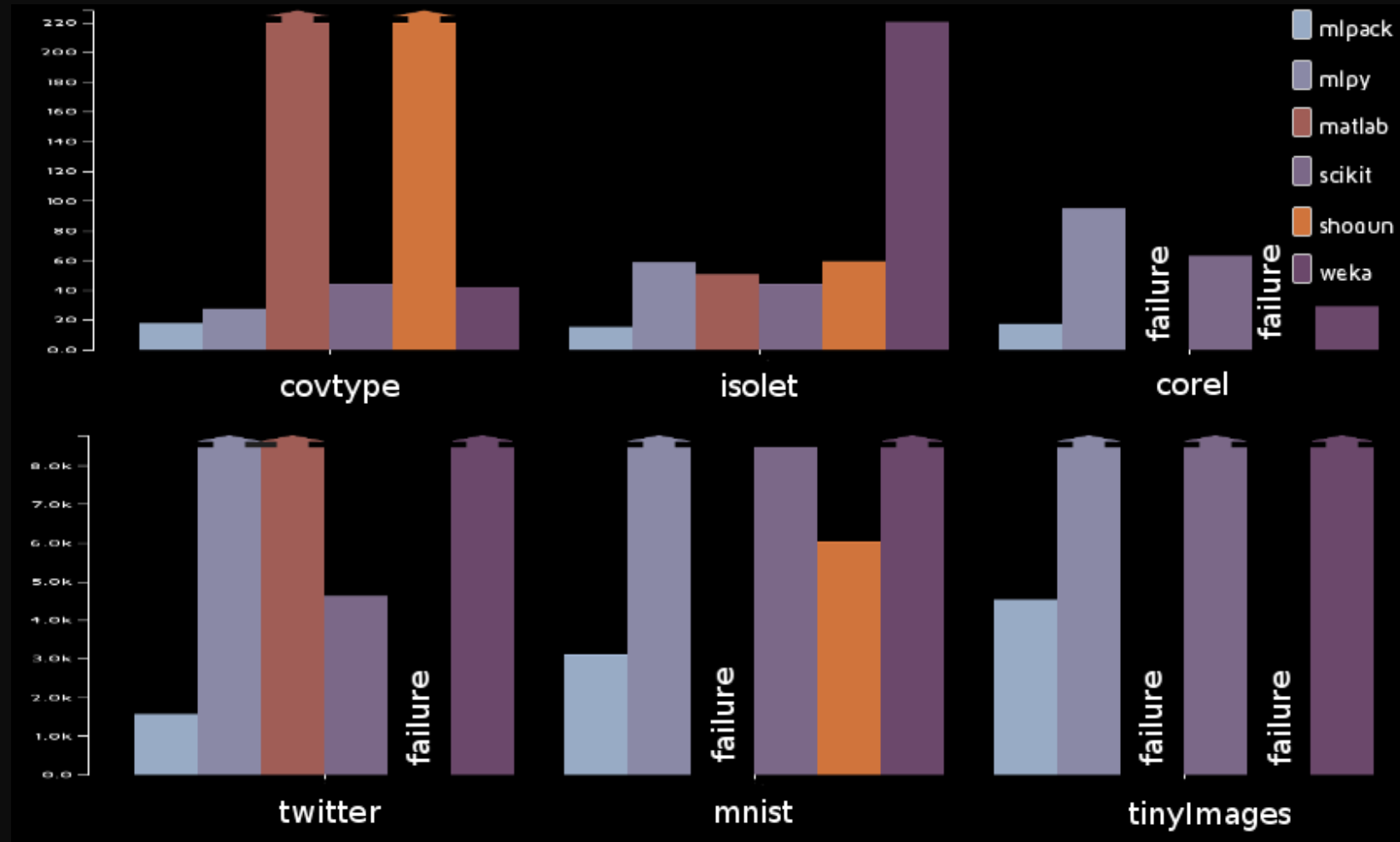
Task 4:  $z = a'(\text{diag}(b)^{-1})c$ .

```
extern int n;  
vec a(n, fill::randu);  
vec b(n, fill::randu);  
vec c(n, fill::randu);  
double z = as_scalar(a.t() * inv(diagmat(b)) * c); // only time this line
```

$n$	arma	numpy	octave	R	Julia
1k	8e-6s	0.100s	2e-4s	0.014s	0.057s
10k	8e-5s	49.399s	4e-4s	0.208s	18.189s
100k	8e-4s	<i>fail</i>	0.002s	<i>fail</i>	<i>fail</i>
1M	0.009s	<i>fail</i>	0.024s	<i>fail</i>	<i>fail</i>
10M	0.088s	<i>fail</i>	0.205s	<i>fail</i>	<i>fail</i>
100M	0.793s	<i>fail</i>	1.972s	<i>fail</i>	<i>fail</i>
1B	8.054s	<i>fail</i>	19.520s	<i>fail</i>	<i>fail</i>



# kNN benchmarks



dataset	$d$	$N$	mlpack	mlpy	matlab	scikit	shogun	Weka
isolet	617	8k	15.65s	59.09s	50.88s	44.59s	59.56s	220.38s
corel	32	68k	17.70s	95.26s	fail	63.32s	fail	29.38s
covertypes	54	581k	18.04s	27.68s	>9000s	44.55s	>9000s	42.34s
twitter	78	583k	1573.92s	>9000s	>9000s	4637.81s	fail	>9000s
mnist	784	70k	3129.46s	>9000s	fail	8494.24s	6040.16s	>9000s
tinyImages	384	100k	4535.38s	9000s	fail	>9000s	fail	>9000s

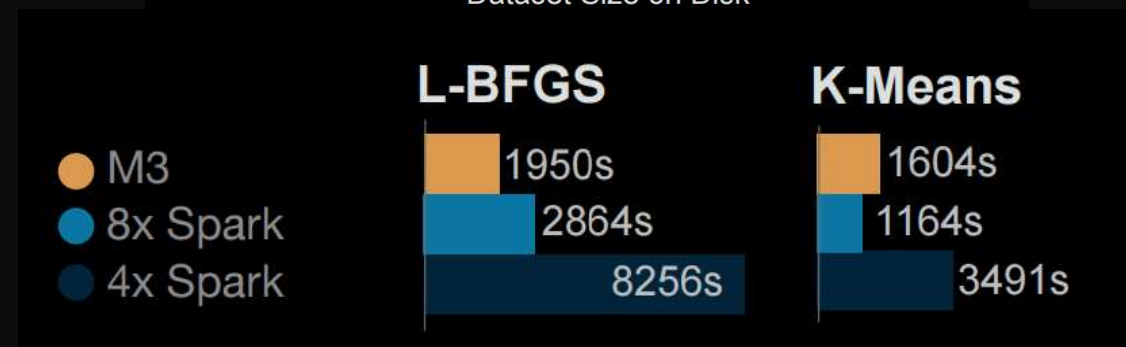
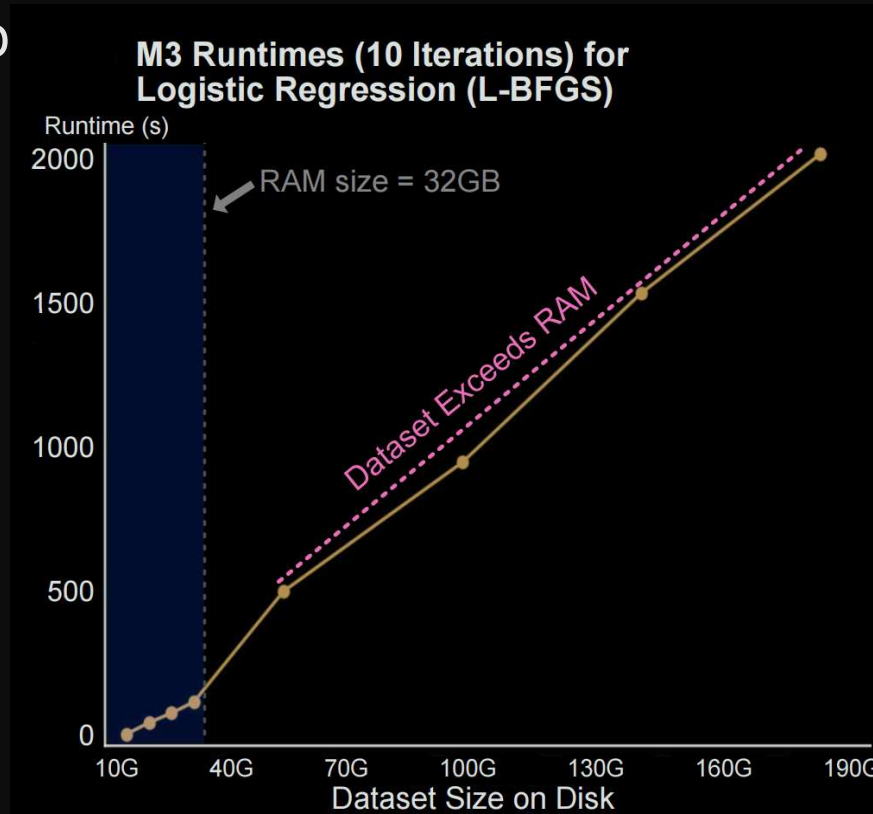
## **vs. Spark**

We can use `mmap()` for out-of-core learning since our algorithms are generic!

# vs. Spark

We can use `mmap()` for out-of-co

e generic!



# ensmallen benchmarks

Runtimes for the linear regression function on various dataset sizes, with  $n$  indicating the number of samples, and  $d$  indicating the dimensionality of each sample. All Julia runs do not count compilation time.

<i>algorithm</i>	<i>d</i> : 100, <i>n</i> : 1k	<i>d</i> : 100, <i>n</i> : 10k	<i>d</i> : 100, <i>n</i> : 100k	<i>d</i> : 1k, <i>n</i> : 100k
<b>ensmallen-2</b>	<b>0.002s</b>	<b>0.016s</b>	<b>0.182s</b>	<b>2.522s</b>
Optim.jl	0.006s	0.030s	0.337s	4.271s
scipy	0.003s	0.017s	0.202s	2.729s
bfgsmin	0.071s	0.859s	23.220s	2859.81s
ForwardDiff.jl	0.497s	1.159s	4.996s	603.106s
autograd	0.007s	0.026s	0.210s	2.673s

S. Bhardwaj, R.R. Curtin, M. Edel, Y. Mentekidis, C. Sanderson, "ensmallen: a flexible C++ library for efficient function optimization", *Systems for ML Workshop at NeurIPS 2018*, 2018.

# ensmallen benchmarks

Runtimes for the linear regression function on various dataset sizes, with  $n$  indicating the number of samples, and  $d$  indicating the dimensionality of each sample. All Julia runs do not count compilation time.

<i>algorithm</i>	<i>d</i> : 100, <i>n</i> : 1k	<i>d</i> : 100, <i>n</i> : 10k	<i>d</i> : 100, <i>n</i> : 100k	<i>d</i> : 1k, <i>n</i> : 100k
<b>ensmallen-1</b>	<b>0.001s</b>	<b>0.009s</b>	<b>0.154s</b>	<b>2.215s</b>
ensmallen-2	0.002s	0.016s	0.182s	2.522s
Optim.jl	0.006s	0.030s	0.337s	4.271s
scipy	0.003s	0.017s	0.202s	2.729s
bfgsmin	0.071s	0.859s	23.220s	2859.81s
ForwardDiff.jl	0.497s	1.159s	4.996s	603.106s
autograd	0.007s	0.026s	0.210s	2.673s

S. Bhardwaj, R.R. Curtin, M. Edel, Y. Mentekidis, C. Sanderson, "ensmallen: a flexible C++ library for efficient function optimization", *Systems for ML Workshop at NeurIPS 2018*, 2018.

# Application: low-latency webserver comment filtering

Let's talk about how we can use mlpack in a deployment environment.  
Here's our (hypothetical) situation:

- We run a news website for some locality or region.
- We get lots of comment spam.
- Our boss has told us we better fix the comment spam issue or else!

# Current spam-filtering workflow

1. User submits comment.

# Current spam-filtering workflow

1. User submits comment.
2. Information about user and comment get logged.



# Current spam-filtering workflow

1. User submits comment.
2. Information about user and comment get logged.
3. Comment gets posted.

# Proposed spam-filtering workflow

1. User submits comment.

# Proposed spam-filtering workflow

1. User submits comment.
2. Information about user and comment get logged.

# Proposed spam-filtering workflow

1. User submits comment.
2. Information about user and comment get logged.
3. **Use kernel density estimation on the IP's geolocation.** If the comment is coming from a place where we don't get lots of valid comments, return a CAPTCHA.

# Proposed spam-filtering workflow

1. User submits comment.
2. Information about user and comment get logged.
3. **Use kernel density estimation on the IP's geolocation.** If the comment is coming from a place where we don't get lots of valid comments, return a CAPTCHA.
4. **Extract more features from the request for spam filtering.**

# Proposed spam-filtering workflow

1. User submits comment.
2. Information about user and comment get logged.
3. **Use kernel density estimation on the IP's geolocation.** If the comment is coming from a place where we don't get lots of valid comments, return a CAPTCHA.
4. **Extract more features from the request for spam filtering.**
5. **Pass extracted features into a fast logistic regression classifier.**

# Proposed spam-filtering workflow

1. User submits comment.
2. Information about user and comment get logged.
3. **Use kernel density estimation on the IP's geolocation.** If the comment is coming from a place where we don't get lots of valid comments, return a CAPTCHA.
4. **Extract more features from the request for spam filtering.**
5. **Pass extracted features into a fast logistic regression classifier.**
6. **If returned prediction is above a threshold, return a CAPTCHA.**

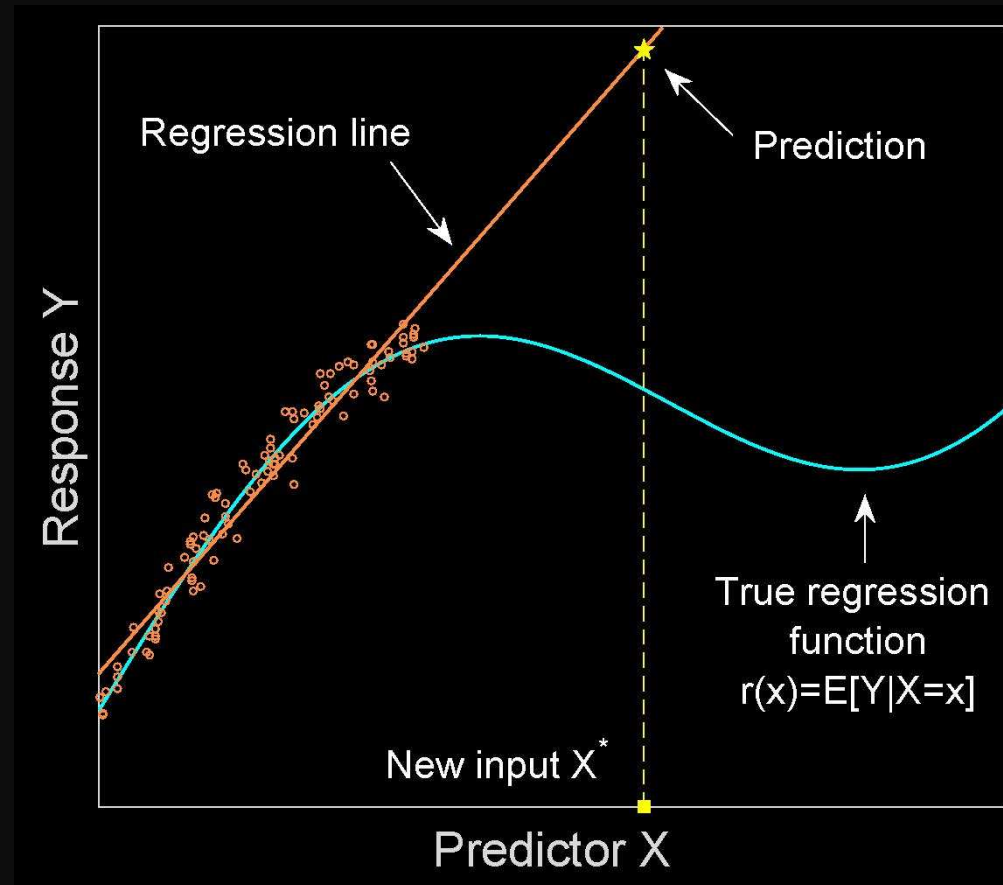
# Proposed spam-filtering workflow

1. User submits comment.
2. Information about user and comment get logged.
3. **Use kernel density estimation on the IP's geolocation.** If the comment is coming from a place where we don't get lots of valid comments, return a CAPTCHA.
4. **Extract more features from the request for spam filtering.**
5. **Pass extracted features into a fast logistic regression classifier.**
6. **If returned prediction is above a threshold, return a CAPTCHA.**
7. Comment gets posted.



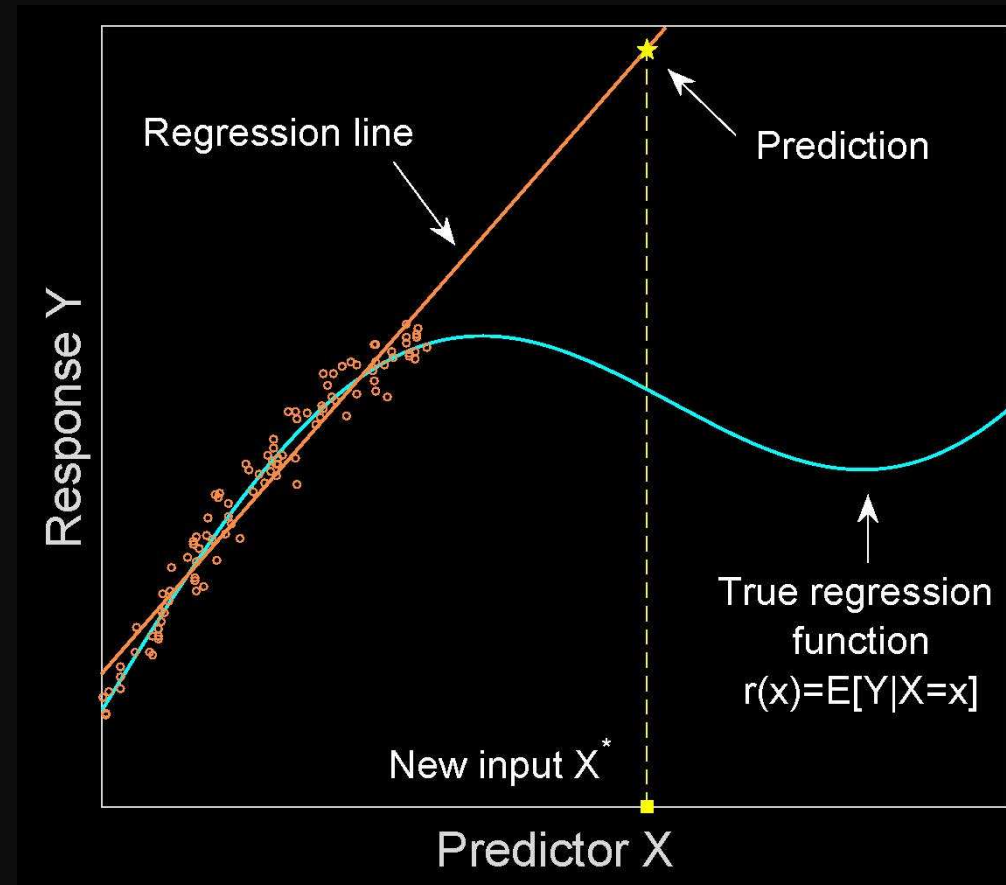
# Why KDE?

**Model extrapolation** is a problem.



# Why KDE?

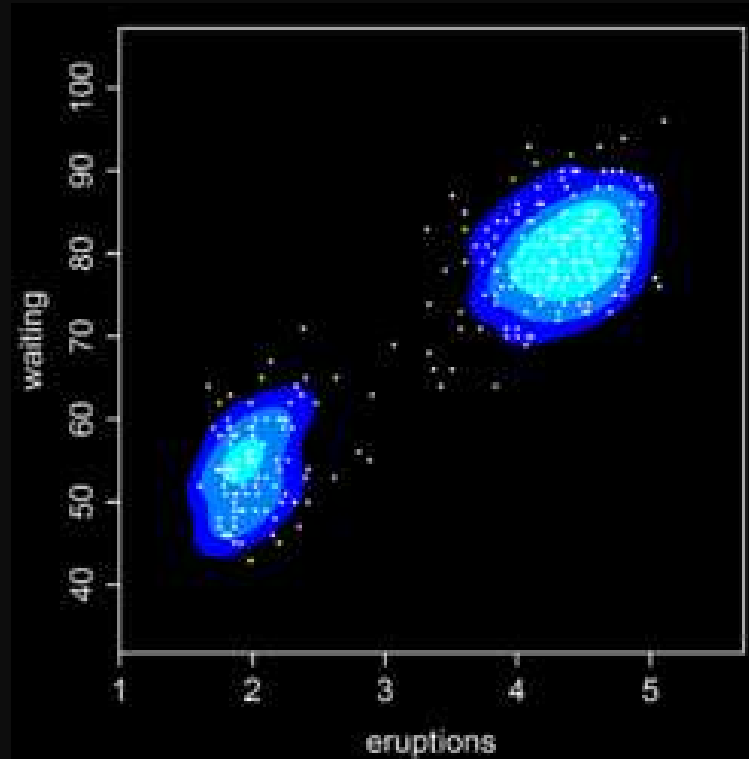
**Model extrapolation** is a problem.



We don't want the model to make predictions for comments coming from locations it wasn't trained on. *We don't have much idea what the model would do in that case!*

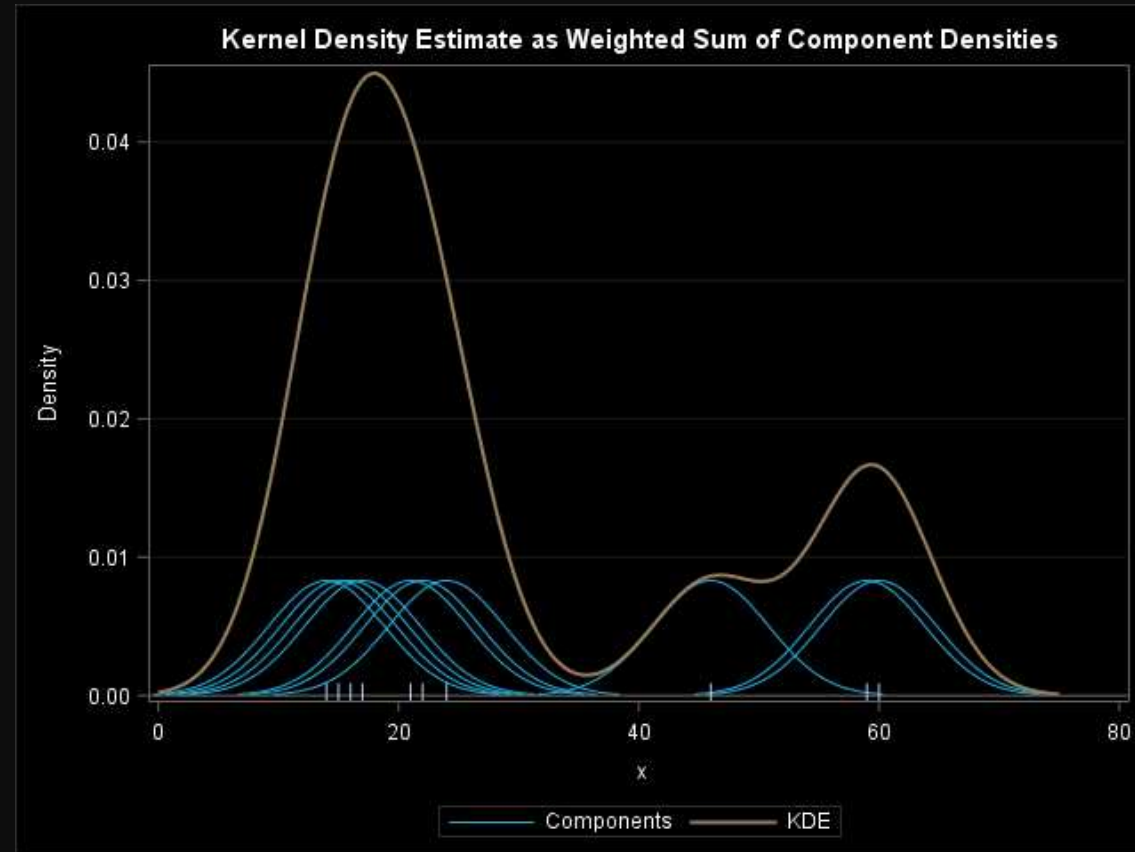
# What is KDE?

**Kernel density estimation** gives us an estimate of the probability density function at a given location based on the training data.



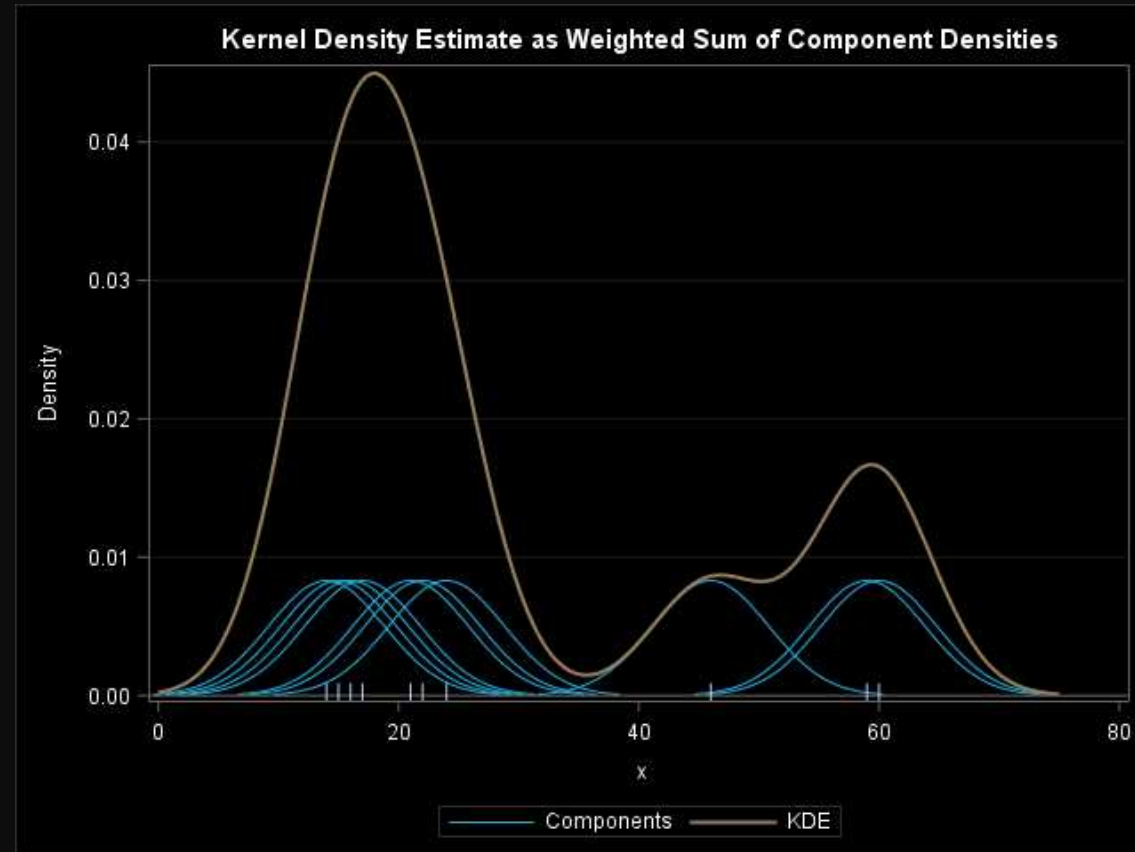
# What is KDE?

To compute a prediction we compute and sum  $K(d(x, y))$  for some distance  $d(\cdot, \cdot)$  and kernel  $K(\cdot)$ .



# What is KDE?

To compute a prediction we compute and sum  $K(d(x, y))$  for some distance  $d(\cdot, \cdot)$  and kernel  $K(\cdot)$ .



mlpack's KDE<> uses dual-tree and single-tree algorithms to provide fast approximate density estimates!

A.G. Gray, A.W. Moore. "Nonparametric density estimation: toward computational tractability." In *Proceedings of the 2003 SIAM International Conference on Data Mining (SIAM Data Mining 2003)*, p. 203–211, 2003.

# Why logistic regression?

We choose logistic regression here because it's *fast* and we are interested in low-latency. To get a prediction, it is just a quick vector-vector dot product, and it won't slow down our pipeline much.

$$y = e^{-(x^T \beta + c)}$$

*Hang on: we'll talk about making it more complex later.*

# Why logistic regression?

We choose logistic regression here because it's *fast* and we are interested in low-latency. To get a prediction, it is just a quick vector-vector dot product, and it won't slow down our pipeline much.

$$y = e^{-(x^T \beta + c)}$$

*Hang on: we'll talk about making it more complex later.*

We can use mpack's `LogisticRegression<>` for this.

# Feature engineering

As input to our logistic regression model, we'll use brutally simple slide-optimized features:

- Number of comments from this IP
- Percentage of spam comments from this IP
- Number of comments from this region
- Percentage of spam comments from this region
- Unigram character counts (number of a's, b's, etc.)



# Feature engineering

As input to our logistic regression model, we'll use brutally simple slide-optimized features:

- Number of comments from this IP
- Percentage of spam comments from this IP
- Number of comments from this region
- Percentage of spam comments from this region
- Unigram character counts (number of a's, b's, etc.)

In a real situation you'd pick more/different features...

# “Before” code

We have some auxiliary functions available to us:

```
// Get the string part of a comment.
std::string getCommentString(const Http::Request& req);

// Post a comment and return the user to the page.
void postComment(const Http::Request& req, Http::ResponseWriter& response);

// Make the user fill out a captcha and post comment if successful.
void captcha(const Http::Request& req, Http::ResponseWriter& response);

// Return radian long/lat coordinates from an IP.
std::pair<double, double> geolocate(const Http::Request& req);

// Log information about a request.
void logRequest(const Http::Request& req);

// Log query functionality.
size_t getNumCommentsFromIP(const Http::Request& req);
size_t getNumCommentsFromRegion(const Http::Request& req);
double getSpamPercentageFromIP(const Http::Request& req);
double getSpamPercentageFromRegion(const Http::Request& req);
size_t getNumComments();
Http::Request& getComment(const size_t id);

// Get whether or not a comment was spam.
bool wasSpam(const Http::Request& req);
```

# “Before” comment handler

Written in Pistache. <http://pistache.io/>

```
struct CommentHandler : public Http::Handler {
    void onRequest(const Http::Request& req, Http::ResponseWriter response) {
        // TODO: GDPR (wait until we get sued)
        logRequest(req);

        // Just post the comment! Don't check for spam, it might reduce our
        // engagement numbers!
        postComment(req, response);
    }
};
```

# Training the KDE model

We need a KDE model ready for use. But the KDE model should work with a distance over the surface of the Earth...

```
class GreatCircleDistance {
public:
    // a and b should be two elements long and have radian long/lat.
    static double Evaluate(const arma::vec& a, const arma::vec& b) {
        // Numerically unstable but slide-ready!
        return std::acos(std::sin(a[0]) * std::sin(b[0]) +
            std::cos(a[0]) * std::cos(b[0]) * std::cos(b[1] - a[1]));

        // A better implementation would use the Haversine formula.
    }
};
```

# Training the KDE model

Before we run our server, let's train our KDE model.

```
// Build the training dataset.  mlpack/Armadillo is column major!
arma::mat dataset(2, getNumComments());
for (size_t currentCol = 0, i = 0; i < dataset.n_cols; ++i) {
    if (!wasSpam(getComment(i))) {
        std::pair<double, double> latLong = geolocate(getComment(i));
        dataset(0, currentCol) = latLong.first;
        dataset(1, currentCol) = latLong.second;
        ++currentCol;
    }
}
dataset.shed_cols(currentCol, dataset.n_cols - 1); // Remove any extra columns.

// Train the model.
KDE<GaussianKernel, GreatCircleDistance> kde(0.05 /* relative error tolerance */,
                                             0.0, /* absolute error tolerance */,
                                             GaussianKernel(BANDWIDTH));

kde.Train(dataset);

// Save the model.
data::Save("kde_model.bin", "model", kde);
```

# Training the logistic regression model: feature extraction

Let's write a utility function to turn a `Http::Request` into an `arma::vec`.

```
// Process a point in the dataset.
arma::vec extractFeatures(const Http::Request& req) {
    arma::vec result(30, arma::fill::zeros);

    result(0) = (double) getNumCommentsFromIP(req);
    result(1) = getSpamPercentageFromIP(req);
    result(2) = (double) getNumCommentsFromRegion(req);
    result(3) = getSpamPercentageFromRegion(req);

    std::string comment = getCommentString(req);
    for (size_t j = 0; j < comment.size(); ++j)
        if (std::tolower(comment[j]) >= 'a' && std::tolower(comment[j]) <= 'z')
            result(4 + size_t(std::tolower(comment[j]) - 'a'))++;

    return result;
}
```

# Training the logistic regression model

Training the logistic regression model is simple: create the dataset, then train.

```
// Build the logistic regression training dataset. One observation per comment.
arma::mat dataset(30, getNumComments());
arma::Row<size_t> labels(getNumComments()); // 0: clean; 1: spam
for (size_t i = 0; i < dataset.n_cols; ++i)
{
    dataset.col(i) = extractFeatures(getComment(i));
    labels(i) = (size_t) wasSpam(getComment(i));
}

LogisticRegression<> lr(dataset, labels, 0.1 /* lambda (penalty) */);
data::Save("lr_model.bin", "model", lr);
```

# Training the logistic regression model

Training the logistic regression model is simple: create the dataset, then train.

```
// Build the logistic regression training dataset. One observation per comment.
arma::mat dataset(2, getNumComments());
arma::Row<size_t> labels(getNumComments()); // 0: clean; 1: spam
for (size_t i = 0; i < dataset.n_cols; ++i)
{
    dataset.col(i) = extractFeatures(getComment(i));
    labels(i) = (size_t) wasSpam(getComment(i));
}

LogisticRegression<> lr(dataset, labels, 0.1 /* lambda (penalty) */);
data::Save("lr_model.bin", "model", lr);
```

But... how do we know we chose the best  $\lambda$  so we didn't overfit?



# Training the logistic regression model

Training the logistic regression model is simple: create the dataset, then train.

```
// Build the logistic regression training dataset. One observation per comment.
arma::mat dataset(2, getNumComments());
arma::Row<size_t> labels(getNumComments()); // 0: clean; 1: spam
for (size_t i = 0; i < dataset.n_cols; ++i)
{
    dataset.col(i) = extractFeatures(getComment(i));
    labels(i) = (size_t) wasSpam(getComment(i));
}

LogisticRegression<> lr(dataset, labels, 0.1 /* lambda (penalty) */);
data::Save("lr_model.bin", "model", lr);
```

But... how do we know we chose the best  $\lambda$  so we didn't overfit? **We can use the hyperparameter tuner.**

# Training the logistic regression model

We can also select the best lambda using the hyperparameter tuner.

```
// Build the logistic regression training dataset. One observation per comment.
arma::mat dataset(2, getNumComments());
arma::Row<size_t> labels(getNumComments()); // 0: clean; 1: spam
for (size_t i = 0; i < dataset.n_cols; ++i)
{
    dataset.col(i) = extractFeatures(getComment(i));
    labels(i) = (size_t) wasSpam(getComment(i));
}

// Create the hyperparameter tuner; it will use 20% of the data as the validation set.
HyperParameterTuner<LogisticRegression, F1<Binary>, SimpleCV> hpt(0.2, dataset, labels);
```

# Training the logistic regression model

We can also select the best lambda using the hyperparameter tuner.

```
// Build the logistic regression training dataset. One observation per comment.
arma::mat dataset(2, getNumComments());
arma::Row<size_t> labels(getNumComments()); // 0: clean; 1: spam
for (size_t i = 0; i < dataset.n_cols; ++i)
{
    dataset.col(i) = extractFeatures(getComment(i));
    labels(i) = (size_t) wasSpam(getComment(i));
}

// Create the hyperparameter tuner; it will use 20% of the data as the validation set.
HyperParameterTuner<LogisticRegression, F1<Binary>, SimpleCV> hpt(0.2, dataset, labels);

// Use grid search on a set of lambdas.
arma::vec lambdas { 0.0, 0.01, 0.05, 0.1, 0.5, 1.0, 5.0, 10.0 };
double bestLambda;
std::tie(bestLambda) = hpt.Optimize(lambdas);
```

# Training the logistic regression model

We can also select the best lambda using the hyperparameter tuner.

```
// Build the logistic regression training dataset. One observation per comment.
arma::mat dataset(2, getNumComments());
arma::Row<size_t> labels(getNumComments()); // 0: clean; 1: spam
for (size_t i = 0; i < dataset.n_cols; ++i)
{
    dataset.col(i) = extractFeatures(getComment(i));
    labels(i) = (size_t) wasSpam(getComment(i));
}

// Create the hyperparameter tuner; it will use 20% of the data as the validation set.
HyperParameterTuner<LogisticRegression, F1<Binary>, SimpleCV> hpt(0.2, dataset, labels);

// Use grid search on a set of lambdas.
arma::vec lambdas { 0.0, 0.01, 0.05, 0.1, 0.5, 1.0, 5.0, 10.0 };
double bestLambda;
std::tie(bestLambda) = hpt.Optimize(lambdas);

// Finally train the model with the best lambda.
LogisticRegression<> lr(dataset, labels, bestLambda);
data::Save("lr_model.bin", "model", lr);
```

# Training the logistic regression model

The hyperparameter tuner supports continuous optimization! So we can use gradient descent too.

```
// Build the logistic regression training dataset. One observation per comment.
arma::mat dataset(2, getNumComments());
arma::Row<size_t> labels(getNumComments()); // 0: clean; 1: spam
for (size_t i = 0; i < dataset.n_cols; ++i)
{
    dataset.col(i) = extractFeatures(getComment(i));
    labels(i) = (size_t) wasSpam(getComment(i));
}

// Create the hyperparameter tuner; it will use 20% of the data as the validation set.
HyperParameterTuner<LogisticRegression, F1<Binary>, SimpleCV, GradientDescent> hpt(0.2, dataset, labels);
hpt.StepSize() = 0.01; // More parameters to tune... :)
hpt.Tolerance() = 1e-5; // Tolerance before terminating search.

// Use gradient descent to find lambda.
double bestLambda;
std::tie(bestLambda) = hpt.Optimize(0.01);

// Finally train the model with the best lambda.
LogisticRegression<> lr(dataset, labels, bestLambda);
data::Save("lr_model.bin", "model", lr);
```

## Back to the CommentHandler...

```
struct CommentHandler : public Http::Handler {
    void onRequest(const Http::Request& req, Http::ResponseWriter response) {
        // TODO: GDPR (wait until we get sued)
        logRequest(req);

        // Just post the comment! Don't check for spam, it might reduce our
        // engagement numbers!
        postComment(req, response);
    }
};
```

# Back to the CommentHandler...

```
struct CommentHandler : public Http::Handler {
    CommentHandler() {
        data::Load("kde_model.bin", "model", kde);
        data::Load("lr_model.bin", "model", lr);
    }

    void onRequest(const Http::Request& req, Http::ResponseWriter response) {
        // TODO: GDPR (wait until we get sued)
        logRequest(req);

        // Just post the comment! Don't check for spam, it might reduce our
        // engagement numbers!
        postComment(req, response);
    }

    KDE<GaussianKernel, GreatCircleDistance> kde;
    LogisticRegression<> lr;
};
```

# Back to the CommentHandler...

```
struct CommentHandler : public Http::Handler {
    CommentHandler() {
        data::Load("kde_model.bin", "model", kde);
        data::Load("lr_model.bin", "model", lr);
    }

    void onRequest(const Http::Request& req, Http::ResponseWriter response) {
        // TODO: GDPR (wait until we get sued)
        logRequest(req);

        std::pair<double, double> latLong = geolocate(req);
        arma::vec result, point { latLong.first, latLong.second };
        kde.Evaluate(point, result);
        if (result[0] < THRESHOLD) {
            captcha(req, response);
            return;
        }

        // TODO: more machine learning...
        postComment(req, response);
    }

    KDE<GaussianKernel, GreatCircleDistance> kde;
    LogisticRegression<> lr;
};
```



# Back to the CommentHandler...

```
struct CommentHandler : public Http::Handler {
    CommentHandler() {
        data::Load("kde_model.bin", "model", kde);
        data::Load("lr_model.bin", "model", lr);
    }

    void onRequest(const Http::Request& req, Http::ResponseWriter response) {
        // TODO: GDPR (wait until we get sued)
        logRequest(req);

        std::pair<double, double> latLong = geolocate(req);
        arma::vec result, point { latLong.first, latLong.second };
        kde.Evaluate(point, result);
        if (result[0] < THRESHOLD) {
            captcha(req, response);
            return;
        }

        // We did not filter, so use the logistic regression model.
        point = extractFeatures(req);
        size_t predClass = lr.Predict(point); // Could use custom threshold.
        (predClass == 1) ? captcha(req, response) : postComment(req, responses);
    }

    KDE<GaussianKernel, GreatCircleDistance> kde;
    LogisticRegression<> lr;
};
```

# We saved our job



What did mlpack get us?

- Fast distance implementation via template type `GreatCircleDistance`
- Flexibility via templates to implement `GreatCircleDistance`
- Fast KDE via dual-tree and single-tree algorithms
- Hyperparameter tuner for easy optimization of `LogisticRegression<>` model.
- Easy integration with performance-optimized C++ code.

# We saved our job



How do we keep our job in the future?

- Add a character-RNN after LogisticRegression<> to filter more points
- Hyperparameter tuning, different/custom kernels for KDE
- Better embeddings or features for the comment text, additional features
- ...

We should be employable for at least the next ten years!

# What didn't I talk about in depth?

- hyper-parameter tuner (*there's a lot more to it!*)
- tree infrastructure for problems like nearest neighbor search
- reinforcement learning code
- matrix decomposition infrastructure
- benchmarking system
- automatic binding generator
- preprocessing utilities
- ...and surely more I am not thinking of...

# What's coming?

mlpack 3.1.0 was just released and is ready for production use!

<http://mlpack.org/blog/mlpack-3-released.html>



<http://www.mlpack.org/>  
<https://github.com/mlpack/mlpack/>

# Further out

Armadillo-like library for GPU matrix operations: **Bandicoot**



<http://coot.sourceforge.io/>

Two separate use case options:

- Bandicoot can be used as a drop-in accelerator to Armadillo, offloading intensive computations to the GPU when possible.
- Bandicoot can be used as its own library for GPU matrix programming.

# Further out

Armadillo-like library for GPU matrix operations: **Bandicoot**



<http://coot.sourceforge.io/>

Two separate use case options:

- Bandicoot can be used as a drop-in accelerator to Armadillo, offloading intensive computations to the GPU when possible.
- Bandicoot can be used as its own library for GPU matrix programming.

```
using namespace coot;  
mat x(n, n, fill::randu); // matrix allocated on GPU  
mat y(n, n, fill::randu);  
mat z = x * y; // computation done on GPU
```

Questions and comments?

