Introducing the challenge

Introducing the challenge

- Learn from the expert who won DrivenData's challenge
  - Natural language processing
  - Feature engineering
  - Efficiency boosting hashing tricks
- Use data to have a social impact

ERS

Introducing the challenge

- Budgets for schools are huge, complex, and not standardized
  - Hundreds of hours each year are spent manually labelling
- Goal: Build a machine learning algorithm that can automate the process
- Budget data
  - Line-item: “Algebra books for 8th grade students”
  - Labels: “Textbooks”, “Math”, “Middle School”
- This is a supervised learning problem

Over 100 target variables!

- This is a classification problem
  - Pre_K:
    - NO_LABEL
    - Non PreK
    - PreK
  - Reporting:
    - NO_LABEL
    - Non-School
    - School
  - Sharing:
    - Leadership & Management
    - NO_LABEL
    - School Reported
  - Student_Type:
    - Alternative
    - At Risk
    - ...

How we can help

- Predictions will be **probabilities** for each label

	Function__Aides Compensation	Function__Career & Academic Counseling	Function__Communications	...	Use__O&M	Use__Pupil Services & Enrichment	Use__Untracked Budget Set-Aside
180042	0.027027	0.027027	0.027027	...	0.125	0.125	0.125
28872	0.027027	0.027027	0.027027	...	0.125	0.125	0.125
186915	0.027027	0.027027	0.027027	...	0.125	0.125	0.125
412396	0.027027	0.027027	0.027027	...	0.125	0.125	0.125
427740	0.027027	0.027027	0.027027	...	0.125	0.125	0.125

Let's practice!

Exploring the data

A column for each possible value

	Eyes	Hair
Jamal	Brown	Curly
Luisa	Brown	Straight
Jenny	Blue	Wavy
Max	Blue	Straight

	Eyes_Blue	Eyes_Brown	Hair_Curly	Hair_Straight	Hair_Wavy
Jamal	0	1	1	0	0
Luisa	0	1	0	1	0
Jenny	1	0	0	0	1
Max	1	0	0	1	0

Load and preview the data

```
In [1]: import pandas as pd

In [2]: sample_df = pd.read_csv('sample_data.csv')

In [3]: sample_df.head()
Out[3]:
  label    numeric     text  with_missing
0     a  -4.167578      bar     -4.084883
1     a  -0.562668               2.043464
2     a -21.361961             -33.315334
3     b  16.402708  foo bar     30.884604
4     a -17.934356      foo    -27.488405
```

Summarize the data

```
In [4]: sample_df.info()
<class 'pandas.core.frame.DataFrame'>
Int64Index: 100 entries, 0 to 99
Data columns (total 4 columns):
label                100 non-null object
numeric              100 non-null float64
text                 100 non-null object
with_missing         95  non-null float64
dtypes: float64(2), object(2)
memory usage: 3.9+ KB
```

Summarize the data

```
In [5]: sample_df.describe()
Out[5]:
          numeric  with_missing
count  100.000000     95.000000
mean    -1.037411      1.275189
std     10.422602     17.386723
min    -26.594495    -42.210641
25%     -6.952244     -8.312870
50%     -0.653688      1.733997
75%      5.398819     11.777888
max     22.922080     41.967536
```

Let’s practice!

Looking at the datatypes

Objects instead of categories

```
In [1]: sample_df['label'].head()
Out[1]:
0     a
1     a
2     a
3     b
4     a
Name: label, dtype: object
```

Encode labels as categories

- ML algorithms work on numbers, not strings
    - Need a numeric representation of these strings
- Strings can be slow compared to numbers
- In pandas, ‘category’ dtype encodes categorical data numerically
    - Can speed up code

Encode labels as categories (sample data)

```
In [1]: sample_df.label.head(2)
Out[1]:
0     a
1     b
Name: label, dtype: object

In [2]: sample_df.label = sample_df.label.astype('category')

In [3]: sample_df.label.head(2)
Out[3]:
0     a
1     b
Name: label, dtype: category
Categories (2, object): [a, b]
```

Dummy variable encoding

```
In [4]: dummies = pd.get_dummies(sample_df[['label']], prefix_sep='_')

In [5]: dummies.head(2)
Out[5]:
   label_a  label_b
0        1        0
1        0        1
```

- Also called a ‘binary indicator’ representation

Lambda functions

- Alternative to ‘def’ syntax
- Easy way to make simple, one-line functions

```
In [6]: square = lambda x: x*x

In [6]: square(2)
Out[6]: 4
```

Encode labels as categories

- In the sample dataframe, we only have one relevant column
- In the budget data, there are multiple columns that need to be made categorical

Encode labels as categories

```
In [7]: categorize_label = lambda x: x.astype('category')

In [8]: sample_df.label = sample_df[['label']].apply(categorize_label,
   ...:                                axis=0)

In [9]: sample_df.info()
<class 'pandas.core.frame.DataFrame'>
Int64Index: 100 entries, 0 to 99
Data columns (total 4 columns):
label                100 non-null category
numeric              100 non-null float64
text                 100 non-null object
with_missing         95 non-null float64
dtypes: category(1), float64(2), object(1)
memory usage: 3.2+ KB
```

Let's practice!

How do we measure success?

How do we measure success?

- Accuracy can be misleading when classes are imbalanced
    - Legitimate email: 99%, Spam: 1%
    - Model that never predicts spam will be 99% accurate!
- Metric used in this problem: log loss
    - It is a loss function
    - Measure of error
    - Want to minimize the error (unlike accuracy)

Log loss binary classification

- Log loss for **binary** classification
  - Actual value: y = {1=yes, 0=no}
  - Prediction (probability that the value is 1): p

$$logloss = -\frac{1}{N}\sum_{i=1}^{N}(y_i \log(p_i) + (1 - y_i)\log(1 - p_i))$$

Log loss binary classification: example

$$logloss_{(N=1)} = y \log(p) + (1-y)\log(1-p)$$

- True label = 0
- Model confidently predicts 1 (with p = 0.90)
- Log loss = $(1-y)\log(1-p)$

  = log(1 - 0.9)

  = log(0.1)

  = 2.30

Log loss binary classification: example

$$logloss_{(\mathrm{N=1})} = y\log(p) + (1-y)\log(1-p)$$

- True label = 1
- Model predicts 0 (with p = 0.50)
- Log loss = 0.69
- Better to be less confident than confident and wrong

Computing log loss with NumPy

logloss.py

```
import numpy as np

def compute_log_loss(predicted, actual, eps=1e-14):
    """ Computes the logarithmic loss between predicted and
        actual when these are 1D arrays.

        :param predicted: The predicted probabilities as floats between 0-1
        :param actual: The actual binary labels. Either 0 or 1.
        :param eps (optional): log(0) is inf, so we need to offset our
                               predicted values slightly by eps from 0 or 1.
    """
    predicted = np.clip(predicted, eps, 1 - eps)
    loss = -1 * np.mean(actual * np.log(predicted)
              + (1 - actual)
              * np.log(1 - predicted))

    return loss
```

Computing log loss with NumPy

```
In [1]: compute_log_loss(predicted=0.9, actual=0)
Out[1]: 2.3025850929940459

In [2]: compute_log_loss(predicted=0.5, actual=1)
Out[2]: 0.69314718055994529
```

Let’s practice!

It's time to build a model

It's time to build a model

- Always a good approach to start with a very simple model
- Gives a sense of how challenging the problem is
- Many more things can go wrong in complex models
- How much signal can we pull out using basic methods?

It's time to build a model

- Train basic model on numeric data only
    - Want to go from raw data to predictions quickly
- Multi-class logistic regression
    - Train classifier on each label separately and use those to predict
- Format predictions and save to csv
- Compute log loss score

Splitting the multi-class dataset

- Recall: Train-test split
  - Will not work here
  - May end up with labels in test set that never appear in training set
- Solution: StratifiedShuffleSplit
  - Only works with a single target variable
  - We have many target variables
  - multilabel_train_test_split()

Splitting the data

```
In  [1]: data_to_train = df[NUMERIC_COLUMNS].fillna(-1000)

In  [2]: labels_to_use = pd.get_dummies(df[LABELS])

In  [3]: X_train, X_test, y_train, y_test = multilabel_train_test_split(
                                              data_to_train, labels_to_use,
                                              size=0.2, seed=123)
```

Training the model

```
In [4]: from sklearn.linear_model import LogisticRegression

In [5]: from sklearn.multiclass import OneVsRestClassifier

In [6]: clf = OneVsRestClassifier(LogisticRegression())

In [7]: clf.fit(X_train, y_train)
```

- OneVsRestClassifier:
  - Treats each column of y independently
  - Fits a separate classifier for each of the columns

Let’s practice!

Making predictions

Predicting on holdout data

```
In [1]: holdout = pd.read_csv('HoldoutData.csv', index_col=0)

In [2]: holdout = holdout[NUMERIC_COLUMNS].fillna(-1000)

In [3]: predictions = clf.predict_proba(holdout)
```

- If .predict() was used instead:
  - Output would be 0 or 1
  - Log loss penalizes being confident and wrong
  - Worse performance compared to .predict_proba()

Submitting your predictions as a csv

	Function__Aides Compensation	Function__Career & Academic Counseling	Function__Communications	...	Use__O&M	Use__Pupil Services & Enrichment	Use__Untracked Budget Set-Aside
180042	0.027027	0.027027	0.027027	...	0.125	0.125	0.125
28872	0.027027	0.027027	0.027027	...	0.125	0.125	0.125
186915	0.027027	0.027027	0.027027	...	0.125	0.125	0.125
412396	0.027027	0.027027	0.027027	...	0.125	0.125	0.125
427740	0.027027	0.027027	0.027027	...	0.125	0.125	0.125

- All formatting can be done with the pandas to_csv function

Format and submit predictions

```
In [4]: prediction_df = pd.DataFrame(columns=pd.get_dummies(df[LABELS],
   ...:                                      prefix_sep='__').columns,
   ...:                                      index=holdout.index,
   ...:                                      data=predictions)

In [5]: prediction_df.to_csv('predictions.csv')

In [6]: score = score_submission(pred_path='predictions.csv')
```

DrivenData leaderboard

User or team	Public	Private	Timestamp	Trend	# Entries
quocnle	0.3665	0.3650	Jan. 6, 2015, 12:27 a.m.		96
Abhishek	0.4409	0.4388	Jan. 6, 2015, 4:09 p.m.		71
giba	0.4551	0.4534	Jan. 5, 2015, 4:52 p.m.		34
trev	0.5054	0.5001	Jan. 3, 2015, 2 a.m.		23
Kappa	0.5228	0.5195	Jan. 6, 2015, 11:46 p.m.		17
bamine	0.5344	0.5298	Dec. 12, 2014, 12:52 a.m.		39
futuristic reality	0.5512	0.5477	Nov. 24, 2014, 8:54 a.m.		22
JesseBuesking	0.5584	0.5556	Jan. 6, 2015, 4:51 p.m.		15
mkrump	0.5817	0.5769	Jan. 3, 2015, 5:12 p.m.		57
joel314	0.5806	0.5772	Dec. 10, 2014, 4:41 p.m.		63

Let's practice!

A very brief introduction to NLP

A very brief introduction to NLP

- Data for NLP:
  - Text, documents, speech, ...
- Tokenization
  - Splitting a string into segments
  - Store segments as list
- Example: ‘Natural Language Processing’
  - —> [‘Natural’, ‘Language’, ‘Processing’]

Tokens and token patterns

- Tokenize on whitespace

PETRO-VEND FUEL AND FLUIDS

PETRO-VEND | FUEL | AND | FLUIDS

- Tokenize on whitespace *and* punctuation

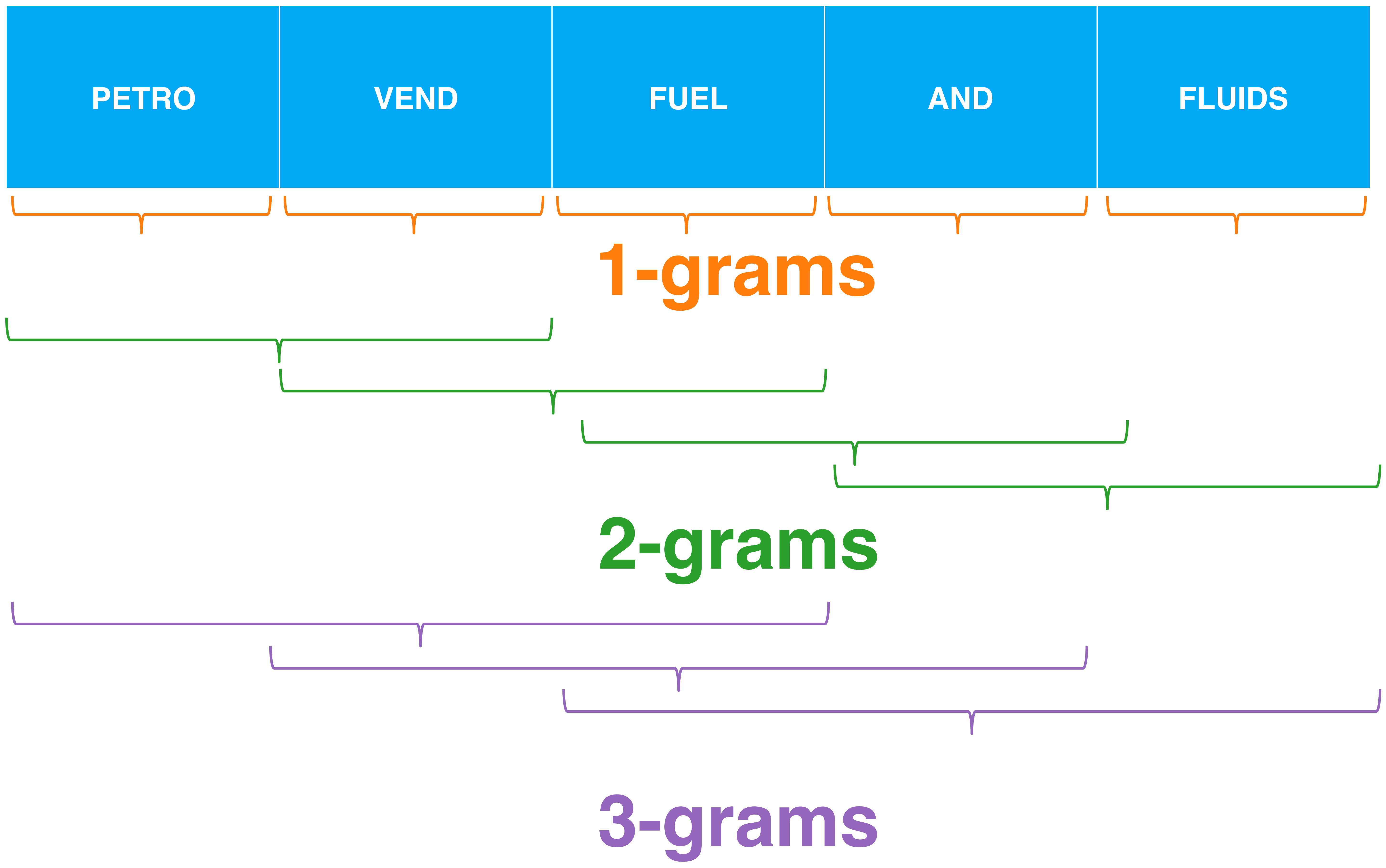
PETRO-VEND FUEL AND FLUIDS

PETRO | VEND | FUEL | AND | FLUIDS

Bag of words representation

- Count the number of times a particular token appears
- “Bag of words”
    - Count the number of times a word was pulled out of the bag
- This approach discards information about word order
    - “Red, not blue” is the same as “blue, not red”

1-gram, 2-gram, ..., n-gram



Let’s practice!

Representing text numerically

Representing text numerically

- Bag-of-words
  - Simple way to represent text in machine learning
  - Discards information about grammar and word order
  - Computes frequency of occurrence

Scikit-learn tools for bag-of-words

- CountVectorizer()
  - Tokenizes all the strings
  - Builds a ‘vocabulary’
  - Counts the occurrences of each token in the vocabulary

Using CountVectorizer() on column of main dataset

```
In [1]: from sklearn.feature_extraction.text import CountVectorizer

In [2]: TOKENS_BASIC = '\\S+(?=\\s+)'

In [3]: df.Program_Description.fillna('', inplace=True)

In [4]: vec_basic = CountVectorizer(token_pattern=TOKENS_BASIC)
```

Using CountVectorizer() on column of main dataset

```
In [5]: vec_basic.fit(df.Program_Description)
Out[5]:
CountVectorizer(analyzer='word', binary=False, decode_error='strict',
        dtype=<class 'numpy.int64'>, encoding='utf-8', input='content',
        lowercase=True, max_df=1.0, max_features=None, min_df=1,
        ngram_range=(1, 1), preprocessor=None, stop_words=None,
        strip_accents=None, token_pattern='\\S+(?=\\s+)',
        tokenizer=None, vocabulary=None)

In [6]: msg = 'There are {} tokens in Program_Description if tokens are
any non-whitespace'

In [7]: print(msg.format(len(vec_basic.get_feature_names())))
There are 157 tokens in Program_Description if tokens are any non-
whitespace
```

Let's practice!

Pipelines, feature & text preprocessing

The pipeline workflow

- Repeatable way to go from raw data to trained model
- Pipeline object takes sequential list of steps
    - Output of one step is input to next step
- Each step is a tuple with two elements
    - Name: string
    - Transform: obj implementing .fit() and .transform()
- Flexible: a step can itself be another pipeline!

Instantiate simple pipeline with one step

```
In [1]: from sklearn.pipeline import Pipeline

In [2]: from sklearn.linear_model import LogisticRegression

In [3]: from sklearn.multiclass import OneVsRestClassifier

In [4]: pl = Pipeline([
   ...:         ('clf', OneVsRestClassifier(LogisticRegression()))
   ...:     ])
```

Train and test with sample numeric data

```
In [5]: sample_df.head()
Out[5]:
  label    numeric     text  with_missing
0     a  -4.167578      bar     -4.084883
1     a  -0.562668               2.043464
2     a -21.361961             -33.315334
3     b  16.402708  foo bar     30.884604
4     a -17.934356      foo    -27.488405
```

Train and test with sample numeric data

```
In [6]: from sklearn.model_selection import train_test_split

In [7]: X_train, X_test, y_train, y_test = train_test_split(
   ...:                                    sample_df[['numeric']],
   ...:                                    pd.get_dummies(sample_df['label']),
   ...:                                    random_state=2)

In [8]: pl.fit(X_train, y_train)
Out[8]:
Pipeline(steps=[('clf', OneVsRestClassifier(estimator=LogisticRegression(C=1.0,
class_weight=None, dual=False, fit_intercept=True,
          intercept_scaling=1, max_iter=100, multi_class='ovr', n_jobs=1,
          penalty='l2', random_state=None, solver='liblinear', tol=0.0001,
          verbose=0, warm_start=False),
          n_jobs=1))])
```

Train and test with sample numeric data

```
In [9]: accuracy = pl.score(X_test, y_test)

In [10]: print('accuracy on numeric data, no nans: ', accuracy)
accuracy on numeric data, no nans:  0.44
```

Adding more steps to the pipeline

```
In [11]: X_train, X_test, y_train, y_test = train_test_split(sample_df[['numeric',
    ...:                                     'with_missing']], pd.get_dummies(
    ...:                                     sample_df['label']), random_state=2)

In [12]: pl.fit(X_train, y_train)
Traceback (most recent call last):
  ...
ValueError: Input contains NaN, infinity or a value too large for
dtype('float64').
```

Preprocessing numeric features with missing data

```
In [13]: from sklearn.preprocessing import Imputer

In [14]: X_train, X_test, y_train, y_test = train_test_split(sample_df[['numeric',
    ...:                                     'with_missing']], pd.get_dummies(
    ...:                                     sample_df['label']), random_state=2)

In [15]: pl = Pipeline([
   ....:         ('imp', Imputer()),
   ....:         ('clf', OneVsRestClassifier(LogisticRegression()))
   ....:     ])
```

Preprocessing numeric features with missing data

```
In [16]: pipeline.fit(X_train, y_train)

In [17]: accuracy = pl.score(X_test, y_test)

In [18]: print('accuracy on all numeric, incl nans: ', accuracy)
accuracy on all numeric, incl nans:  0.48
```

- No errors!

Let's practice!

Text features and feature unions

Preprocessing text features

```
In [1]: from sklearn.feature_extraction.text import CountVectorizer

In [2]: X_train, X_test, y_train, y_test = train_test_split(sample_df['text'],
   ...:                                                     pd.get_dummies(
   ...:                                                     sample_df['label']),
   ...:                                                     random_state=2)

In [3]: pl = Pipeline([
   ....:         ('vec', CountVectorizer()),
   ....:         ('clf', OneVsRestClassifier(LogisticRegression()))
   ....:     ])
```

Preprocessing text features

```
In [4]: pl.fit(X_train, y_train)
Out[4]:
Pipeline(steps=[('vec', CountVectorizer(analyzer='word', binary=False,
decode_error='strict', dtype=<class 'numpy.int64'>, encoding='utf-8',
input='content', lowercase=True, max_df=1.0, max_features=None, min_df=1,
ngram_range=(1, 1), preprocessor=None, stop_words=None, strip_...=None,
solver='liblinear', tol=0.0001, verbose=0, warm_start=False), n_jobs=1))])

In [5]: accuracy = pl.score(X_test, y_test)

In [6]: print('accuracy on sample data: ', accuracy)
accuracy on sample data:  0.64
```

Preprocessing multiple dtypes

- Want to use all available features in one pipeline
- Problem
    - Pipeline steps for numeric and text preprocessing can't follow each other
    - e.g., output of CountVectorizer can't be input to Imputer
- Solution
    - FunctionTransformer() & FeatureUnion()

FunctionTransformer

- Turns a Python function into an object that a scikit-learn pipeline can understand
- Need to write two functions for pipeline preprocessing
  - Take entire DataFrame, return numeric columns
  - Take entire DataFrame, return text columns
- Can then preprocess numeric and text data in separate pipelines

Putting it all together

```
In [7]: X_train, X_test, y_train, y_test = train_test_split(sample_df[['numeric',
   ...:                                           'with_missing', 'text']], pd.get_dummies(
   ...:                                           sample_df['label']), random_state=2)

In [8]: from sklearn.preprocessing import FunctionTransformer

In [9]: from sklearn.pipeline import FeatureUnion
```

Putting it all together

```
In [10]: get_text_data = FunctionTransformer(lambda x: x['text'],
    ...:                                     validate=False)

In [11]: get_numeric_data = FunctionTransformer(lambda x: x[['numeric',
    ...:                                    'with_missing']], validate=False)
```

FeatureUnion Text and Numeric Features

```
In [12]: from sklearn.pipeline import FeatureUnion

In [13]: union = FeatureUnion([
    ...:            ('numeric', numeric_pipeline),
    ...:            ('text', text_pipeline)
    ...:      ])
```

Text Features

Numeric Features

Putting it all together

```
In [14]: numeric_pipeline = Pipeline([
    ...:                         ('selector', get_numeric_data),
    ...:                         ('imputer', Imputer())
    ...:                     ])

In [15]: text_pipeline = Pipeline([
    ...:                         ('selector', get_text_data),
    ...:                         ('vectorizer', CountVectorizer())
    ...:                     ])

In [16]: pl = Pipeline([
    ...:         ('union', FeatureUnion([
    ...:             ('numeric', numeric_pipeline),
    ...:             ('text', text_pipeline)
    ...:         ])),
    ...:         ('clf', OneVsRestClassifier(LogisticRegression()))
    ...:     ])
```

Let's practice!

MACHINE LEARNING WITH THE EXPERTS

Choosing a classification model

Main dataset: lots of text

```
In [1]: LABELS = ['Function', 'Use', 'Sharing', 'Reporting', 'Student_Type',
   ...:           'Position_Type', 'Object_Type',  'Pre_K', 'Operating_Status']

In [2]: NON_LABELS = [c for c in df.columns if c not in LABELS]

In [3]: len(NON_LABELS) - len(NUMERIC_COLUMNS)
Out[3]: 14
```

Using pipeline with the main dataset

```
In [4]: import numpy as np

In [5]: import pandas as pd

In [6]: df = pd.read_csv('TrainingSetSample.csv', index_col=0)

In [7]: dummy_labels = pd.get_dummies(df[LABELS])

In [8]: X_train, X_test, y_train, y_test = multilabel_train_test_split(
   ...:                                    df[NON_LABELS], dummy_labels,
   ...:                                    0.2)
```

Using pipeline with the main dataset

```
In [10]: get_text_data = FunctionTransformer(combine_text_columns,
    ...:                                     validate=False)

In [11]: get_numeric_data = FunctionTransformer(lambda x:
    ...:                        x[NUMERIC_COLUMNS], validate=False)

In [12]: pl = Pipeline([
    ...:         ('union', FeatureUnion([
    ...:                 ('numeric_features', Pipeline([
    ...:                     ('selector', get_numeric_data),
    ...:                     ('imputer', Imputer())
    ...:                 ])),
    ...:                 ('text_features', Pipeline([
    ...:                     ('selector', get_text_data),
    ...:                     ('vectorizer', CountVectorizer())
    ...:                 ]))
    ...:              ])
    ...:         ),
    ...:         ('clf', OneVsRestClassifier(LogisticRegression()))
    ...:     ])
```

Performance using main dataset

```
In [13]: pl.fit(X_train, y_train)
Out[13]:
Pipeline(steps=[('union', FeatureUnion(n_jobs=1,
transformer_list=[('numeric_features', Pipeline(steps=[('selector',
FunctionTransformer(accept_sparse=False, func=<function <lambda> at
0x11415ec80>, pass_y=False, validate=False)), ('imputer', Imputer(axis=0,
copy=True, missing_valu...=None, solver='liblinear', tol=0.0001, verbose=0,
warm_start=False),n_jobs=1))])
```

Flexibility of model step

- Is current model the best?
- Can quickly try different models with pipelines
    - Pipeline preprocessing steps unchanged
    - Edit the model step in your pipeline
    - Random Forest, Naïve Bayes, k-NN

Easily try new models using pipeline

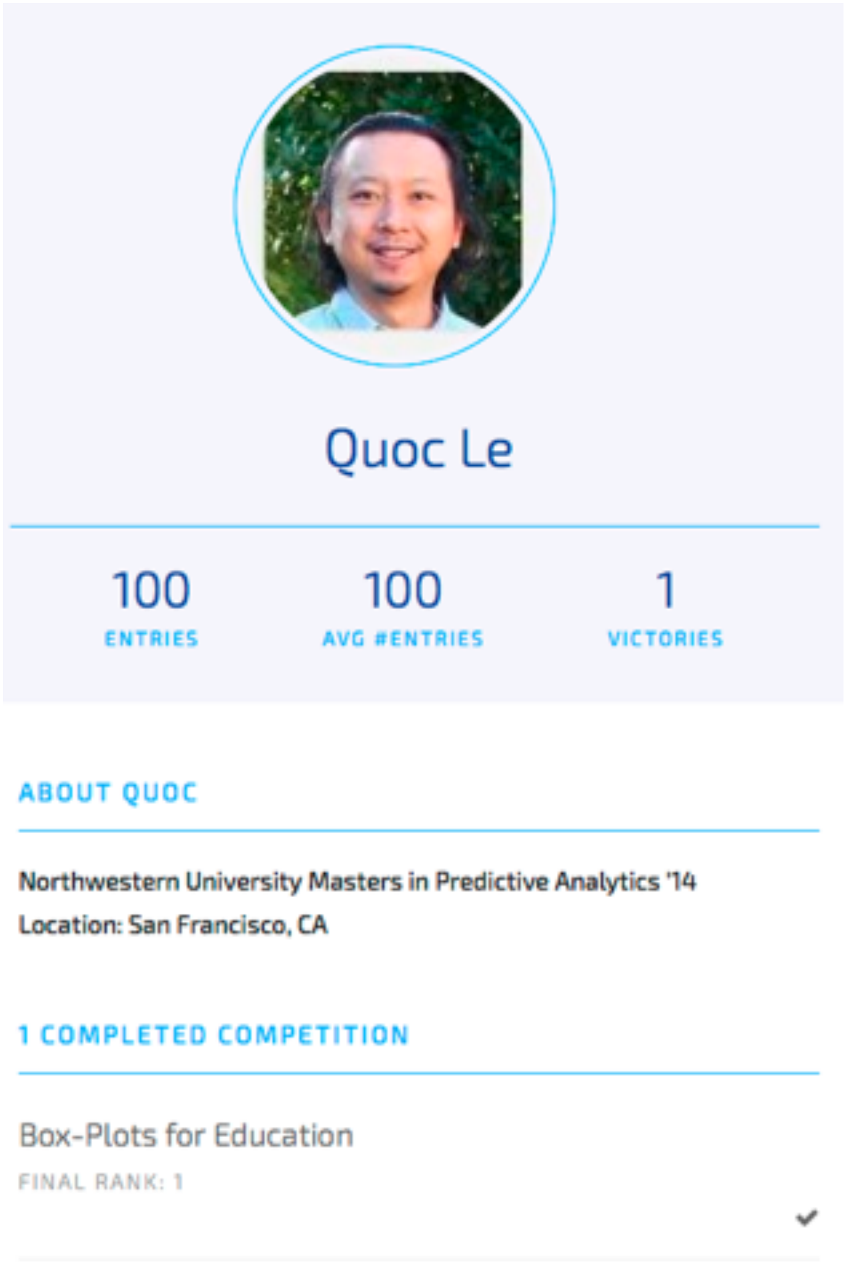
```
In [14]: from sklearn.ensemble import RandomForestClassifier

In [15]: pl = Pipeline([
    ...:         ('union', FeatureUnion(
    ...:             transformer_list = [
    ...:                 ('numeric_features', Pipeline([
    ...:                     ('selector', get_numeric_data),
    ...:                     ('imputer', Imputer())
    ...:                 ])),
    ...:                 ('text_features', Pipeline([
    ...:                     ('selector', get_text_data),
    ...:                     ('vectorizer', CountVectorizer())
    ...:                 ]))
    ...:              ]
    ...:         )),
    ...:         ('clf', OneVsRest(RandomForestClassifier()))
    ...:     ])
```

Let's practice!

Learning from the expert: processing

Learning from the expert

- Text processing
- Statistical methods
- Computational efficiency



Learning from the expert: text preprocessing

- NLP tricks for text data
  - Tokenize on punctuation to avoid hyphens, underscores, etc.
  - Include unigrams **and** bi-grams in the model to capture important information involving multiple tokens - e.g., ‘middle school’

N-grams and tokenization

```
In [1]: vec = CountVectorizer(token_pattern=TOKENS_ALPHANUMERIC,
   ...:                       ngram_range=(1, 2))
```

- Simple changes to CountVectorizer
    - alphanumeric tokenization
    - ngram_range=(1, 2)

Range of n-grams in scikit-learn

```
In [2]: pl.fit(X_train, y_train)
Out[2]:
Pipeline(steps=[('union', FeatureUnion(n_jobs=1,
       transformer_list=[('numeric_features',
Pipeline(steps=[('selector',
FunctionTransformer(accept_sparse=False,
          func=<function <lambda> at 0x11441f7b8>, pass_y=False,
          validate=False)), ('imputer', Imputer(axis=0, copy=True,
missing_valu...=None, solver='liblinear', tol=0.0001,
          verbose=0, warm_start=False),
          n_jobs=1))])
```

Range of n-grams in scikit-learn

```
In [3]: holdout = pd.read_csv('HoldoutData.csv', index_col=0)

In [4]: predictions = pl.predict_proba(holdout)

In [5]: prediction_df = pd.DataFrame(columns=pd.get_dummies(
   ...:                 df[LABELS]).columns, index=holdout.index,
   ...:                 data=predictions)

In [6]: prediction_df.to_csv('predictions.csv')

In [7]: score = score_submission(pred_path='predictions.csv')
```

Let's practice!

Learning from the expert: a stats trick

Learning from the expert: interaction terms

- Statistical tool that the winner used: interaction terms
- Example
  - English teacher for 2nd grade
  - 2nd grade - budget for English teacher
- Interaction terms mathematically describe when tokens appear together

Interaction terms: the math

$$\beta_1 x_1 + \beta_2 x_2 + \beta_3 (x_1 \times x_2)$$

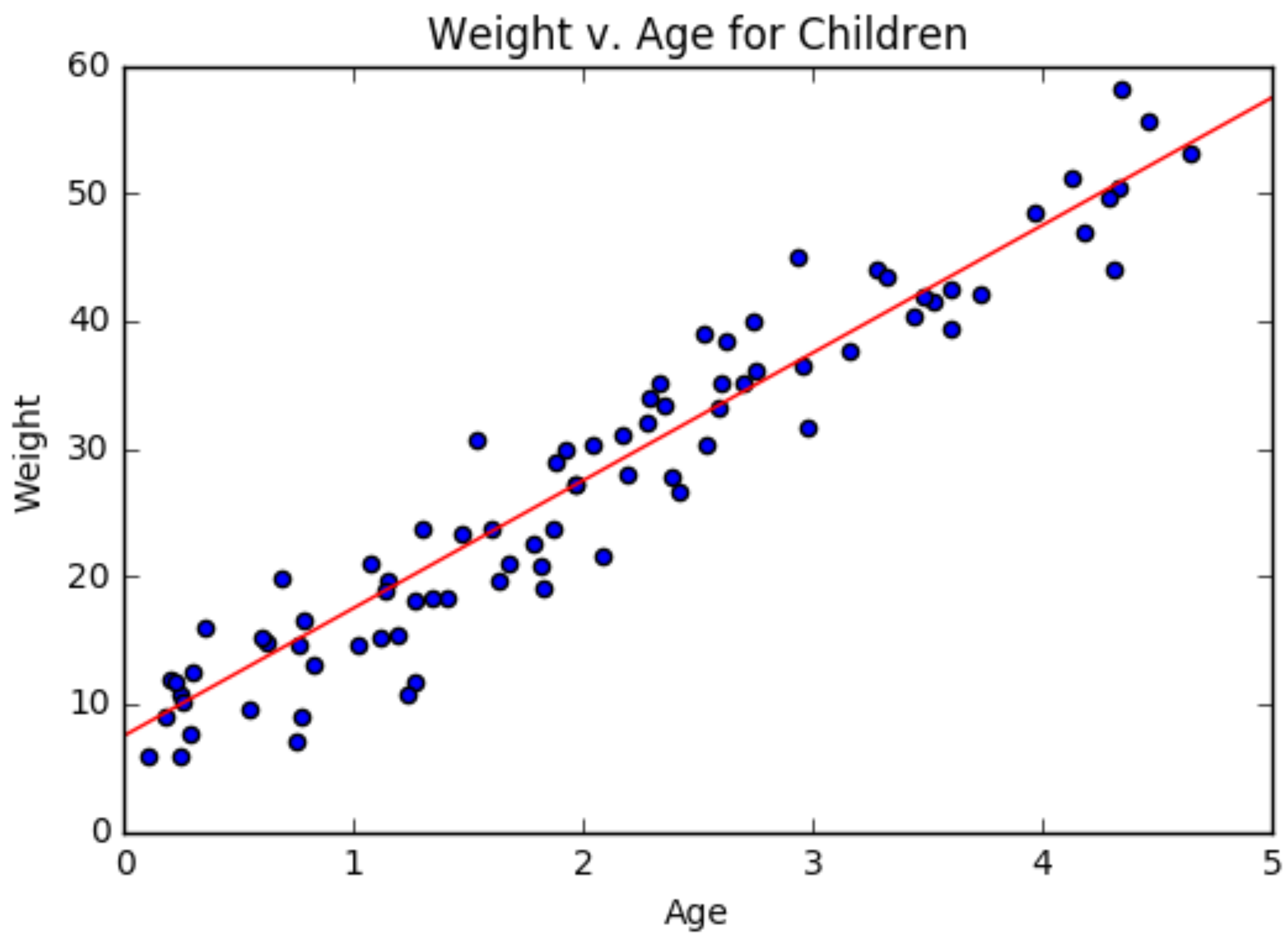
X1	X2
0	1
1	1

X3
X1*X2 = 0*1 = 0
X1*X2 = 1*1 = 1

Adding interaction features with scikit-learn

```
In [1]: from sklearn.preprocessing import PolynomialFeatures

In [2]: x
Out[2]:
   x1  x2
a   0   1
b   1   1

In [3]: interaction = PolynomialFeatures(degree=2,
   ...:                                  interaction_only=True,
   ...:                                  include_bias=False)

In [4]: interaction.fit_transform(x)
Out[4]:
array([[ 0.,  1.,  0.],
       [ 1.,  1.,  1.]])
```

A note about bias terms



- Bias term allows model to have non-zero y value when x value is zero

Sparse interaction features

```
In [5]: SparseInteractions(degree=2).fit_transform(x).toarray()
Out[5]:
array([[ 0.,  1.,  0.],
       [ 1.,  1.,  1.]])
```

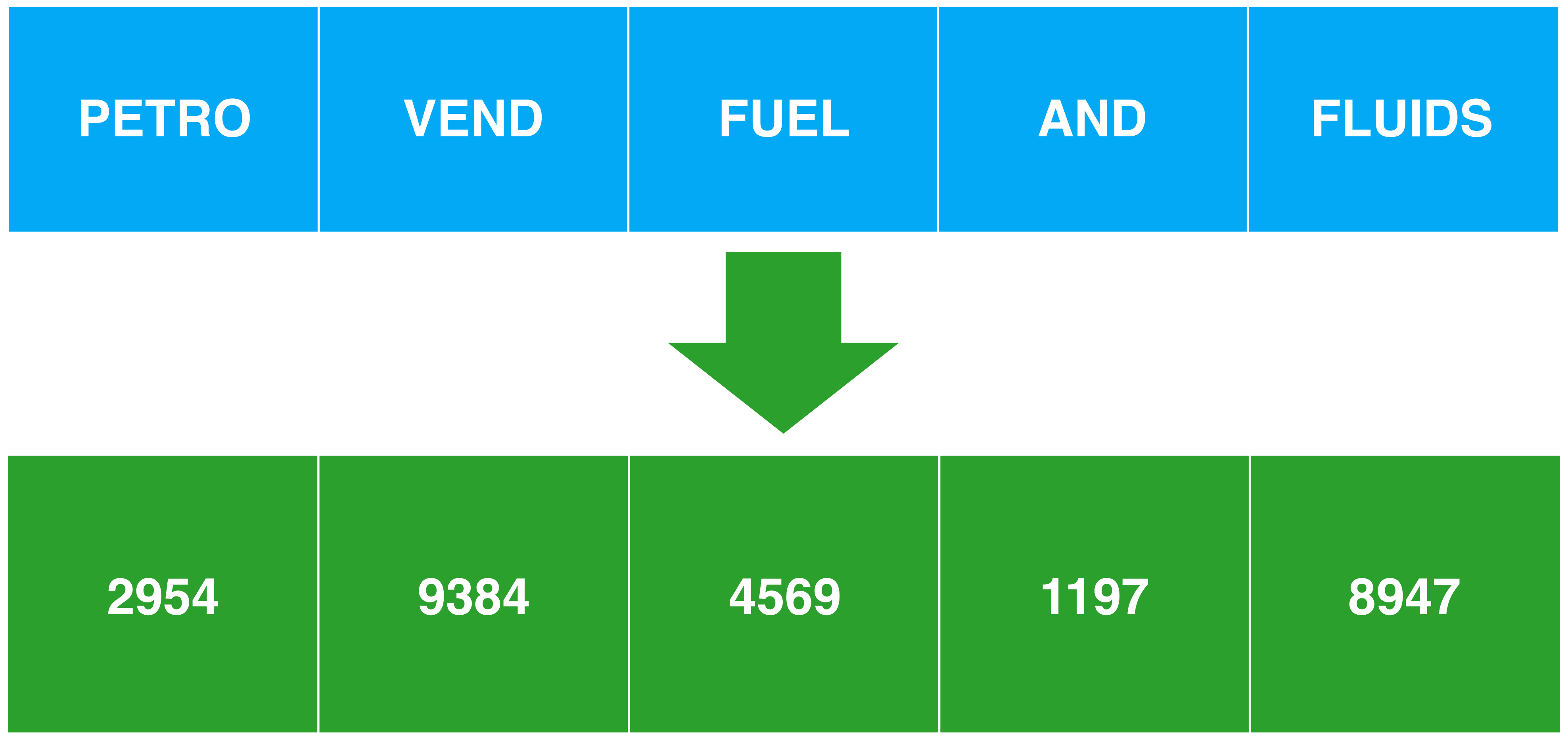
- The number of interaction terms grows exponentially
- Our vectorizer saves memory by using a sparse matrix
- PolynomialFeatures does not support sparse matrices
- We have provided SparseInteractions to work for this problem

Let’s practice!

Learning from the expert: a computational trick and the winning model

Learning from the expert: hashing trick

- Adding new features may cause enormous increase in array size
- Hashing is a way of increasing memory efficiency



- Hash function limits possible outputs, fixing array size

When to use the hashing trick

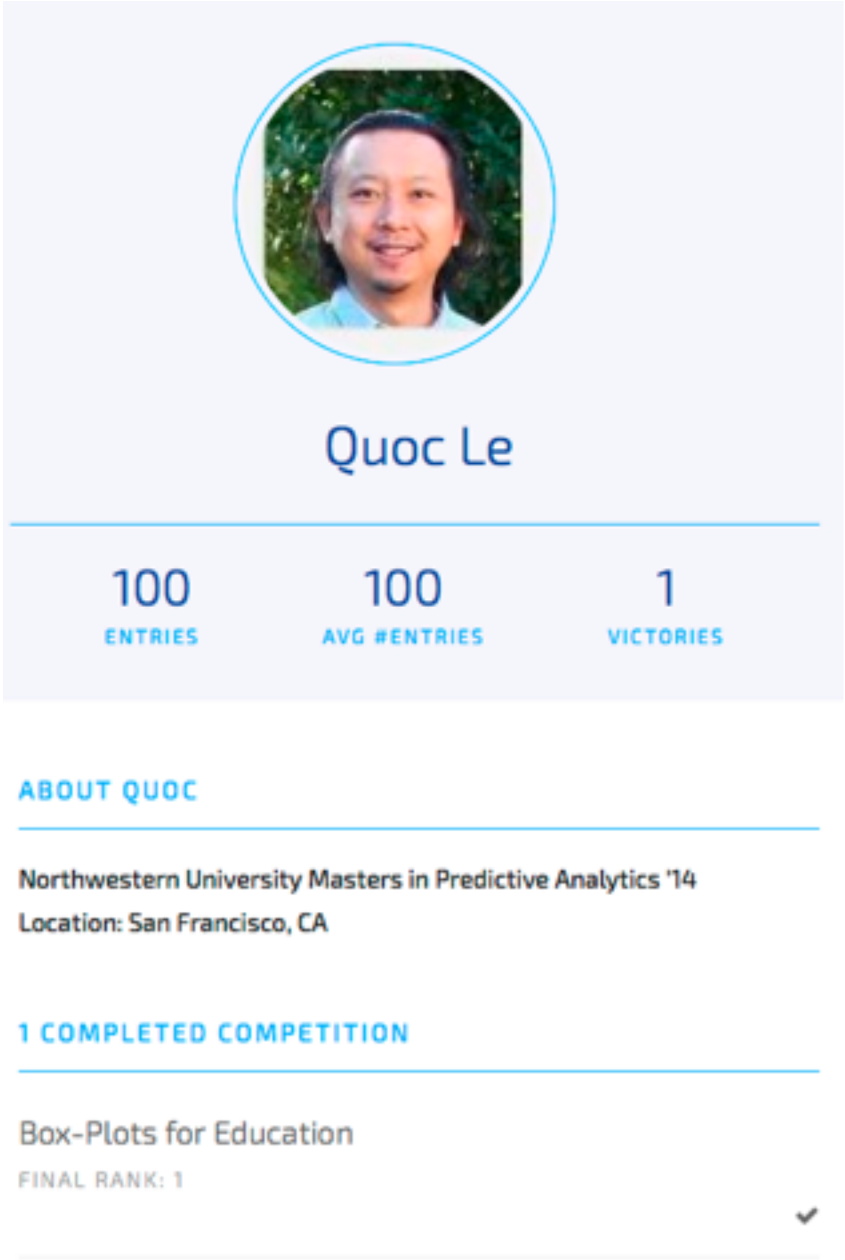
- Want to make array of features as small as possible
  - Dimensionality reduction
- Particularly useful on large datasets
  - e.g., lots of text data!

Implementing the hashing trick in scikit-learn

```
In [5]: from sklearn.feature_extraction.text import HashingVectorizer

In [6]: vec = HashingVectorizer(norm=None,
   ...:                         non_negative=True,
   ...:                         token_pattern=TOKENS_ALPHANUMERIC,
   ...:                         ngram_range=(1, 2))
```

The model that won it all

- You now know all the expert moves to make on this dataset
    - NLP: Range of n-grams, punctuation tokenization
    - Stats: Interaction terms
    - Computation: Hashing trick
- What class of model was used?



The model that won it all

- And the winning model was...
- Logistic regression!
  - Carefully create features
  - Easily implemented tricks
- Favor simplicity over complexity and see how far it takes you!

Let’s practice!

Next steps and the social impact of your work

Can you do better?

- You've seen the flexibility of the pipeline steps
- Quickly test ways of improving your submission
  - NLP: Stemming, stop-word removal
  - Model: RandomForest, k-NN, Naïve Bayes
  - Numeric Preprocessing: Imputation strategies
  - Optimization: Grid search over pipeline objects
  - Experiment with new scikit-learn techniques
- Work with the full dataset at DrivenData!

Hundreds of hours saved

- Make schools more efficient by improving their budgeting decisions
- Saves hundreds of hours each year that humans spent labeling line items
- Can spend more time on the decisions that really matter

DrivenData: Data Science to save the world

- Other ways to use data science to have a social impact at www.drivendata.org
    - Improve your data science skills while helping meaningful organizations thrive
    - Win some cash prizes while you're at it!

Go out and change the world!