Data Science 101

Katie Malone

Director of Data Science, Civis Analytics

#ODSC West, November 2017

CIVIS™ ANALYTICS

Building a Data-Driven World™

Introducing Myself

Currently: Director of Data Science at Civis Analytics

- Chicago-based data science startup, founded 2013
- data science software and services company
- R+D data science role

Previously:

- PhD in physics (Stanford), 2015
- thesis on new particle searches at CERN/LHC
- Udacity's Intro to Machine Learning
- Linear Digressions (podcast)

CIVIS ANALYTICS

Agenda for this Workshop

Advanced beginner level

(approx. 1.5 hours) Sprint through an end-to-end pipeline

- predict functionality of African wells
- one-hot encoding, feature selection, random forest

(approx. 2 hours) Make it maintainable as we add new data

- modeling workflow as a living, breathing thing
- things that are easy once, and exceedingly difficult 10 times

(approx. 0.5 hours) Model framework usability, understanding the model, testing

- how does the model change as new data comes in?
- how can we continue to build sustainably?

Building a model is one thing.

Maintaining and deploying a model is something else entirely.

The latter is arguably more important, and more challenging, but gets less attention.

THAT CHANGES TODAY.

Competition data science vs. real-life data science

Problem	Competition	Real life
what is the problem we want to solve?	classify the “labels” column	who knows?
what does the data look like? where does it come from?	tidy format; the internet	messy; gotta go collect and/or merge it myself
what am I optimizing for?	out-of-sample accuracy	smoothness and clarity of delivery
what’s my biggest problem?	model selection and hyperparameter tuning	stuff not breaking

Our twist today

Scenario: you are the data scientist in serving a team of well inspectors in Africa. They use your model to prioritize future well inspections.

Every day, they go out and measure well functionality, along with many attributes of the wells they've seen. Their goal is to spend their valuable time inspecting the highest-risk wells.

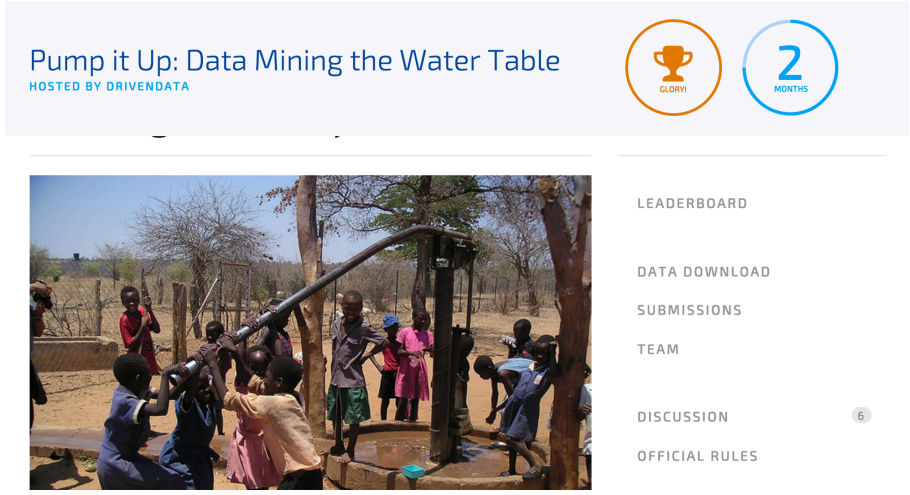
They routinely package up this data and send it back to you.

Your job:

- **incorporate the new data into the model QUICKLY**
- **tell the team if/where the model has changed**
- **say whether the model is getting better, worse, or staying the same**

Preliminaries

- **Clone the workshop repo**
  - https://github.com/cmmalone/ODSC2017_datascience101
  - If you don't have the data already: http://www.drivendata.org/competitions/7/
- **Other tools that I assume you have working locally**
  - python 3 (2.x may work; no guarantees)
  - scikit-learn
  - pandas
  - numpy/scipy
  - ipython
  - pytest



First draft of an analysis

- Data loading and formatting for machine learning
- Deciding on a modeling strategy
- Cross-validation and evaluation of the model

General Framework for Getting Started

1. Read in data
    - in starter notebook (pandas dataframe)
2. Clean
    - already done (no missing data, typos, etc.)
3. Transform into a machine-learning-friendly format
4. cross-validation
5. model
6. Test/evaluate
7. Repeat steps 2-7

Progress Bar

- **Data loading and formatting for machine learning**
- Deciding on a modeling strategy
- Cross-validation and evaluation of the model

Transforming string labels into integers

goal: transform string labels (functional, functional needs repair, non functional) into integers (0, 1, 2)

why: scikit-learn generally assumes numeric inputs

```
id,status_group
61848,functional
48451,non functional
58155,non functional
34169,functional needs repair
18274,functional
48375,functional
6091,functional
58500,functional needs repair
```

```
id,status_group
61848,2
48451,0
58155,0
34169,1
18274,2
48375,2
6091,2
58500,1
```

how: pandas applymap(), or pandas replace() [or another method of your choice]

On to the Features

labels

```
id,status_group
61848,functional
48451,non functional
58155,non functional
34169,functional needs repair
18274,functional
48375,functional
6091,functional
58500,functional needs repair
```

goal: transform string features into something numeric

Many features

Not all features are strings

String features look generally categorical (NOT ordinal)

features

```
54551,0,10/9/12,Rwssp,0,DWE,32.62061707,-4.22619802,Tushirikiane,0,Lake Tanganyika,Nyawishi
Center,Shinyanga,17,3,Kahama,Chambo,0,
      TRUE,GeoData Consultants Ltd,,,TRUE,0,nira/tanira,nira/tanira,handpump,wug,user-group,unknown,unknown,milky,milky,enough,
      enough,shallow well,shallow well,groundwater,hand pump,hand pump
53934,0,11/3/12,Wateraid,0,Water Aid,32.71110001,-5.14671181,Kwa Ramadhan Musa,0,Lake Tanganyika,Imalauduki,Tabora,14,6,Tabora
Urban,Itetemia,0,
      TRUE,GeoData Consultants Ltd,VWC,,TRUE,0,india mark ii,india mark ii,handpump,vwc,user-group,never pay,never pay,salty,
      salty,seasonal,seasonal,machine dbh,borehole,groundwater,hand pump,hand pump
46144,0,8/3/11,Isingiro Ho,0,Artisan,30.62699053,-1.25705061,Kwapeto,0,Lake Victoria,Mkonomre,Kagera,18,1,Karagwe,Kaisho,0,
      TRUE,GeoData Consultants Ltd,,,TRUE,0,nira/tanira,nira/tanira,handpump,vwc,user-group,never pay,never pay,soft,good,enough,
      enough,shallow well,shallow well,groundwater,hand pump,hand pump
49056,0,2/20/11,Private,62,Private,39.20951812,-7.03413939,Mzee Hokororo,0,Wami / Ruvu,Mizugo,Pwani,60,43,Mkuranga,Tambani,345,
      TRUE,GeoData Consultants Ltd,Private operator,,FALSE,2011,submersible,submersible,submersible,private operator,commercial,
      never pay,never pay,salty,salty,enough,enough,machine dbh,borehole,groundwater,other,other
```

Feature Transformations

For each feature (dataframe column) that you want to transform:

1. **Make a list of the unique values to be found in that column**
2. **Create a mapping from each string value to an integer**
3. **Use apply() function to execute the remapping**

id	source	source_type	source_class
69572	spring	spring	groundwater
8776	rainwater harvesting	rainwater harvesting	surface
34310	dam	dam	surface
67743	machine dbh	borehole	groundwater
19728	rainwater harvesting	rainwater harvesting	surface

id	source	source_type	source_class
69572	1	2	0
8776	2	1	1
34310	3	3	1
67743	4	5	0
19728	2	1	1

example ONLY

Progress Bar

- **Data loading and formatting for machine learning**
- Deciding on a modeling strategy
- Cross-validation and evaluation of the model

Suggested Features to be Transformed

```
to_transform = ["funder", "installer", "wpt_name", "basin",
                "subvillage”, "region", "lga", "ward",
                "public_meeting", "recorded_by",
                "scheme_management", "scheme_name", "permit",
                "extraction_type", "extraction_type_group",
                "extraction_type_class",
                "management", "management_group",
                "payment", "payment_type",
                "water_quality", "quality_group", "quantity",
                "quantity_group",
                "source", "source_type", "source_class",
                "waterpoint_type", "waterpoint_type_group"]
```

suggest dropping date_recorded entirely

Classification vs. Regression

Classification

- Discrete output (chocolate, strawberry, vanilla)
- (Often) non-ordinal
- There is no natural order to functional/non-functional wells

Regression

- Continuous output (incomes, ages)
- Ordered dependent variable
- There's a natural order to well functionality
- Regression output needs bucketing before evaluation

Progress Bar

- Data loading and formatting for machine learning
- **Deciding on a modeling strategy**
- Cross-validation and evaluation of the model

Ordinal Modeling

ordered categorial output

- functional→2
- functional needs repair→1
- non functional→0
- ordinal model!
- Not supported out-of-the-box using scikit-learn
- 2 choices:
  - classifier (ignores order information)
  - regressor + bucketing

scikit-learn makes it **straightforward to try both approaches**, so no real disadvantage to **trying both** and picking whichever is better

Progress Bar

- Data loading and formatting for machine learning
- **Deciding on a modeling strategy**
- Cross-validation and evaluation of the model

Predicting well failures with logistic regression

sklearn: machine learning model in **4 lines of code**

cheapest, easiest way to evaluate model performance: sklearn.metrics.cross_val_score()

```
import sklearn.linear_model
import sklearn.cross_validation
clf = sklearn.linear_model.LogisticRegression()
score = sklearn.cross_validation.cross_val_score( clf, features, labels)
print( score )
```

despite the name (Logistic Regression),

we are taking a **classifier approach** here

Progress Bar

- Data loading and formatting for machine learning
- Deciding on a modeling strategy
- **Cross-validation and evaluation of the model**

Comparing logistic regression to tree-based methods

Decision Tree Classifier

```
import sklearn.tree

import sklearn.cross_validation

clf = sklearn.tree.DecisionTreeClassifier()

score = sklearn.cross_validation.cross_val_score( clf,
features, labels)

print( score )
```

Random Forest Classifier

```
import sklearn.tree

import sklearn.cross_validation

clf = sklearn.tree.DecisionTreeClassifier()

score = sklearn.cross_validation.cross_val_score( clf,
features, labels)

print( score )
```

Progress Bar

- Data loading and formatting for machine learning
- Deciding on a modeling strategy
- **Cross-validation and evaluation of the model**

Paying down technical debt and tuning the models

- one-hot encoding of categorical features
- transformers and predictors
- understanding and tuning algorithms

Implicit ordering introduced when making numerical features

id	source	source_type	source_class
69572	spring	spring	groundwater
8776	rainwater harvesting	rainwater harvesting	surface
34310	dam	dam	surface
67743	machine dbh	borehole	groundwater
19728	rainwater harvesting	rainwater harvesting	surface

id	source	source_type	source_class
69572	1	2	0
8776	2	1	1
34310	3	3	1
67743	4	5	0
19728	2	1	1

new feature representation:

machine dbh > dam > rainwater harvesting > spring

is this really what I want?

probably not

Progress Bar

- **One-hot encoding of categorical features**
- Transformers and estimators
- Understanding and tuning algorithms

One-hot encoding to make dummy features

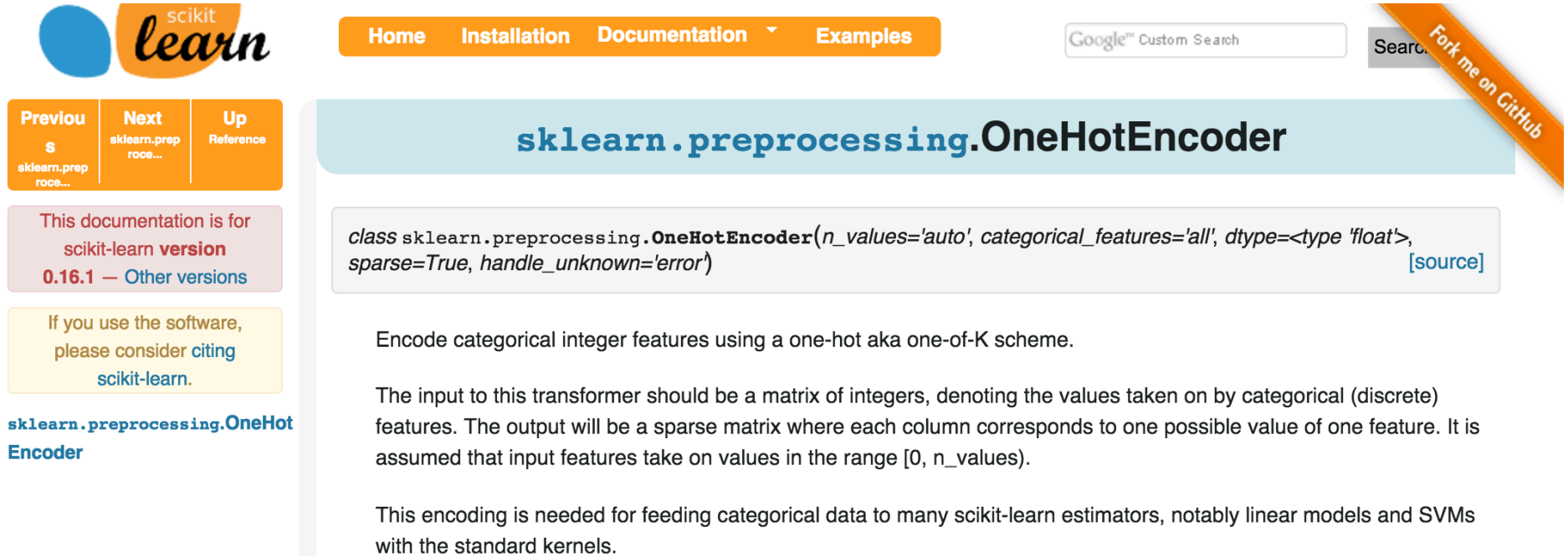
index	country
1	"United States"
2	"Mexico"
3	"Mexico"
4	"Canada"
5	"United States"
6	"Canada"

index	country_UnitedStates	country_Mexico	country_Canada
1	1	0	0
2	0	1	0
3	0	1	0
4	0	0	1
5	1	0	0
6	0	0	1

Preogress Bar

- **One-hot encoding of categorical features**
- Transformers and estimators
- Understanding and tuning algorithms

Coding up a one-hot encoder



sklearn Transformers and Estimators

Transformers

- technical term in sklearn
- .transform() method
- OneHotEncoder, PCA (principal components analysis), SelectKBest (feature selector), etc…

Estimators

- also a technical term (refers to base class of object)
- .predict() method
- DecisionTreeClassifier, LogisticRegression, LinearRegression, RandomForestClassifier, etc…

Progress Bar

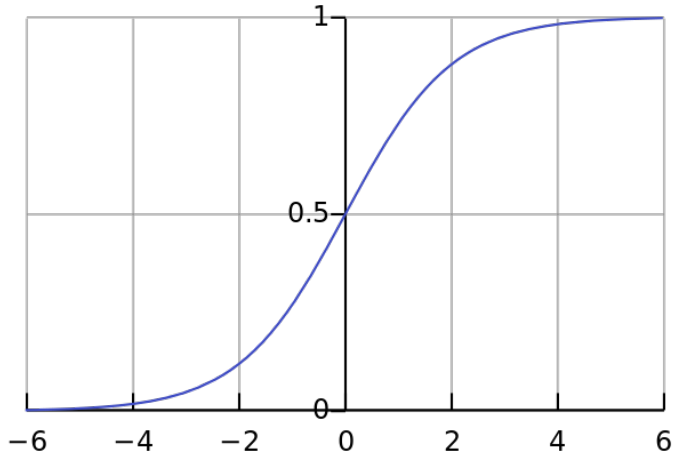
- One-hot encoding of categorical features
- **Transformers and estimators**
- Understanding and tuning algorithms

Logistic ~~Regression Classification~~ Regression

$$F(x) = \frac{1}{1 + e^{-(\beta_0 + \beta_1 x)}}$$

input variables (there will usually be many of these)

outcome variable



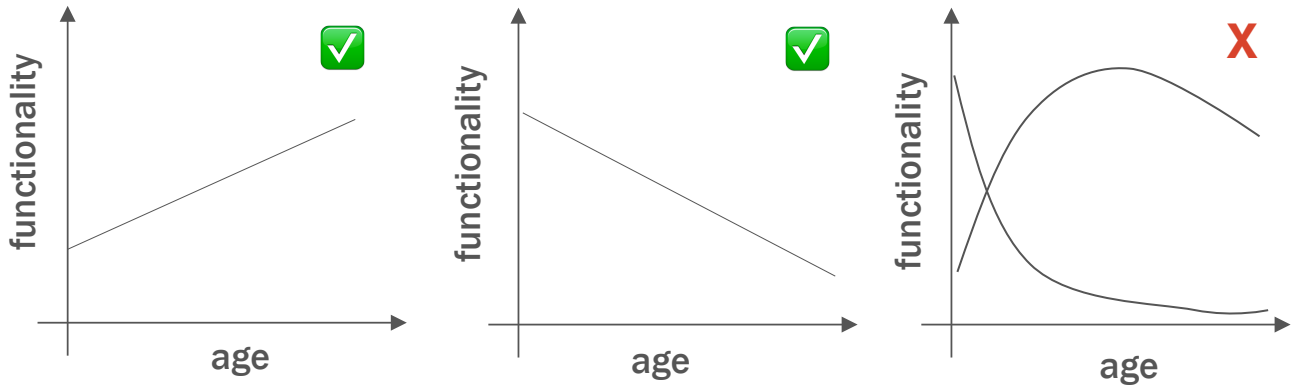
Progress Bar

- One-hot encoding of categorical features
- Transformers and estimators
- **Understanding and tuning algorithms**

Linear Models

$$F(x) = \frac{1}{1 + e^{-(\beta_0 + \beta_1 x)}}$$

example: how does functionality depend on age of the well?



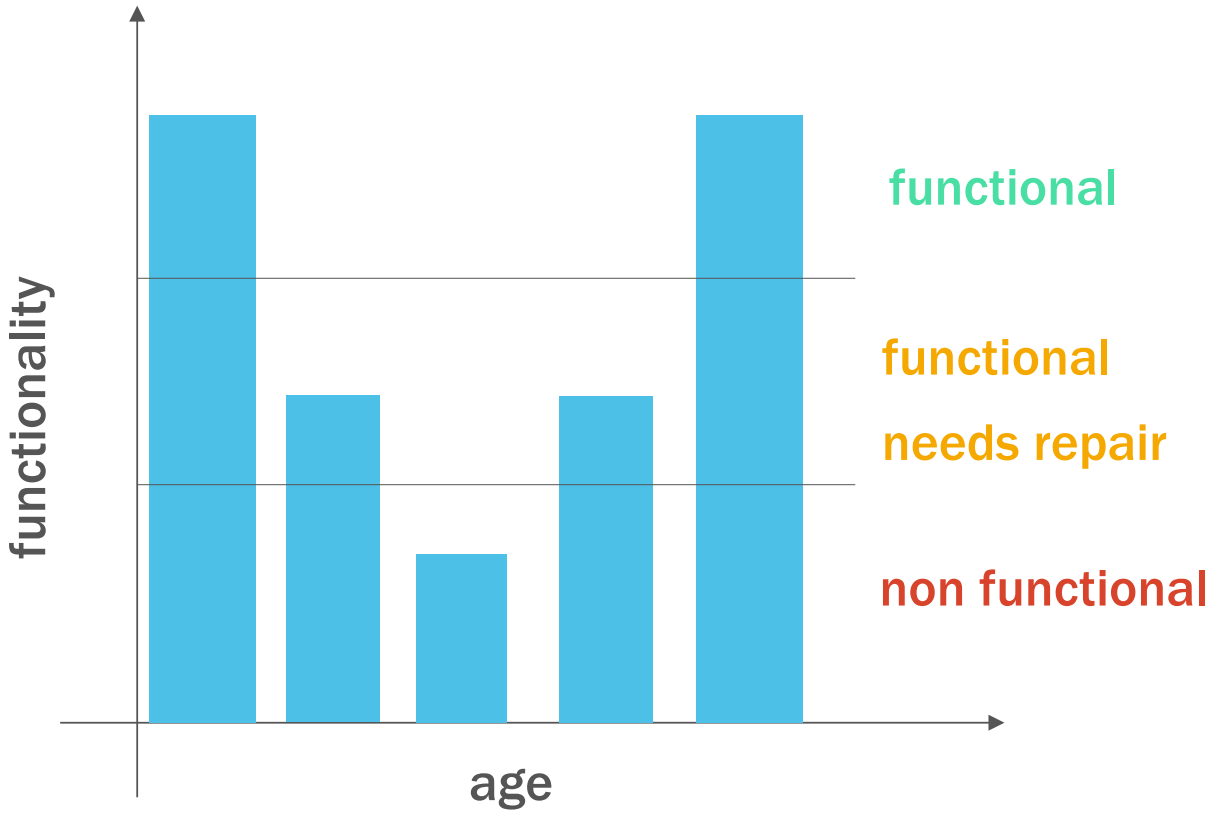
(feature^2, log(feature)) can help

capture non-linearities, but more manual intervention

Progress Bar

- One-hot encoding of categorical features
- Transformers and estimators
- **Understanding and tuning algorithms**

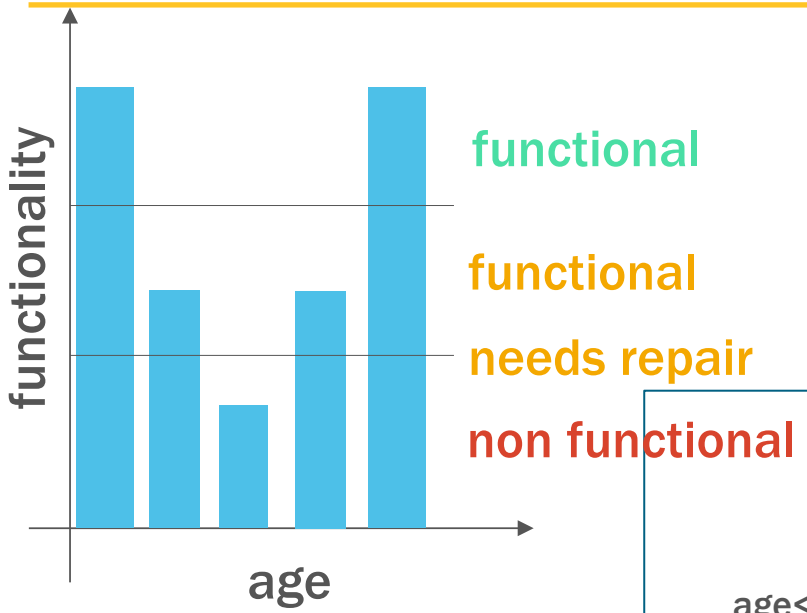
Example of Nonlinearity

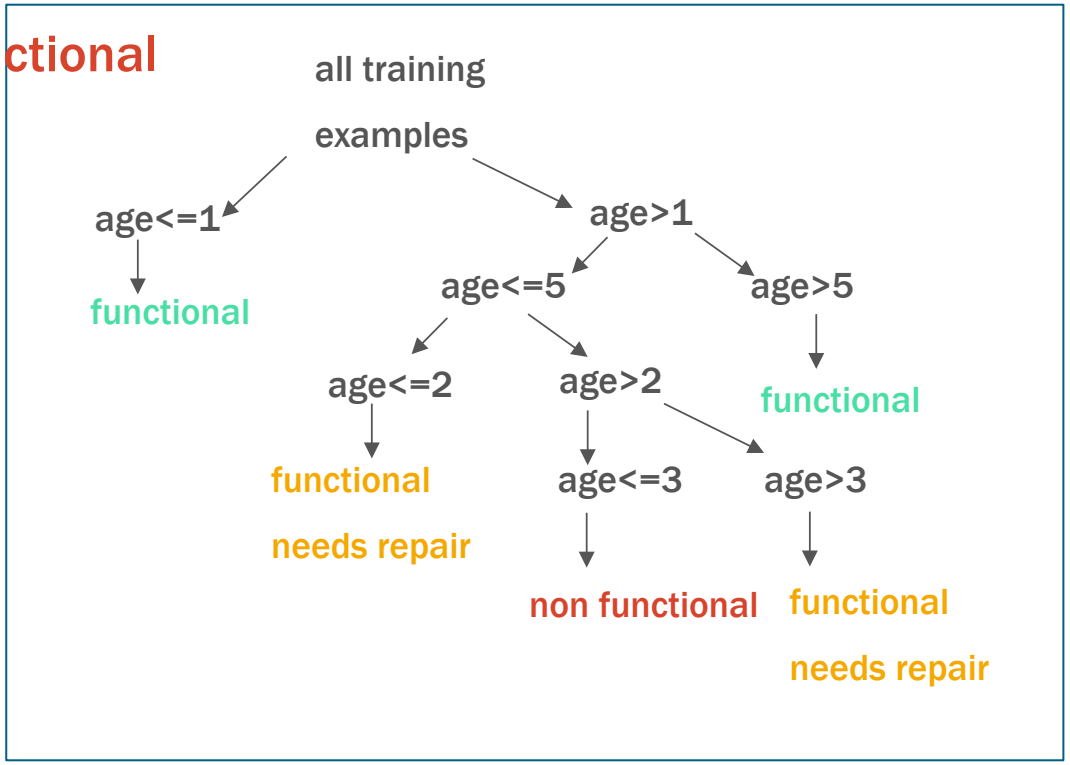


Progress Bar

- One-hot encoding of categorical features
- Transformers and estimators
- **Understanding and tuning algorithms**

Decision Tree Classifier

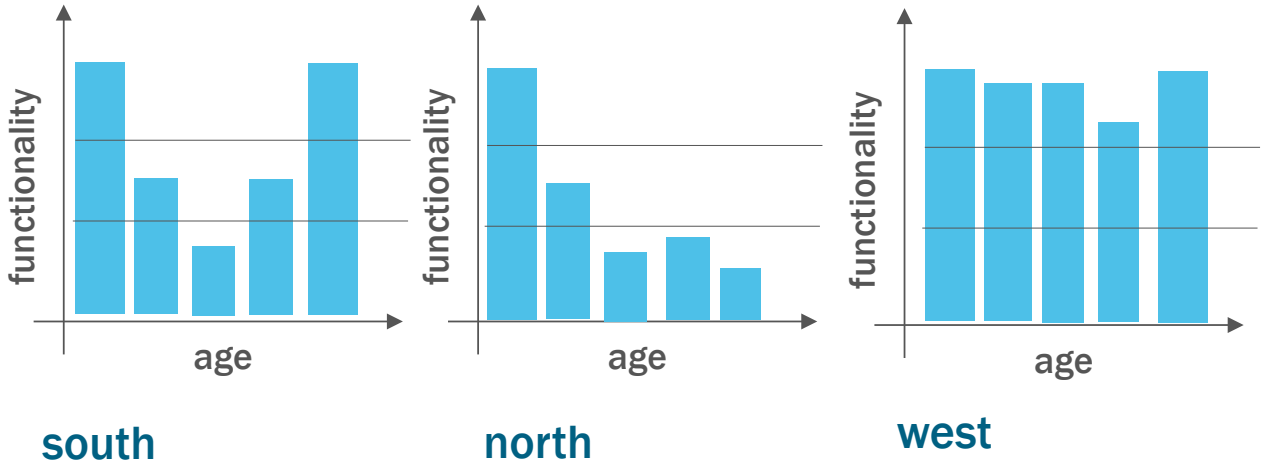


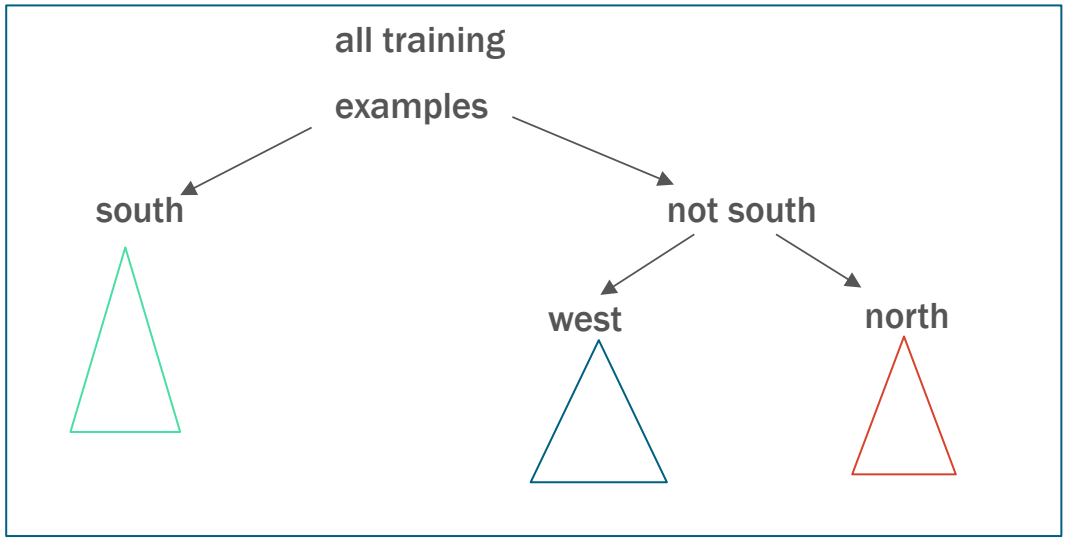


Progress Bar

- One-hot encoding of categorical features
- Transformers and estimators
- **Understanding and tuning algorithms**

Decision Trees gracefully handle relationships between variables

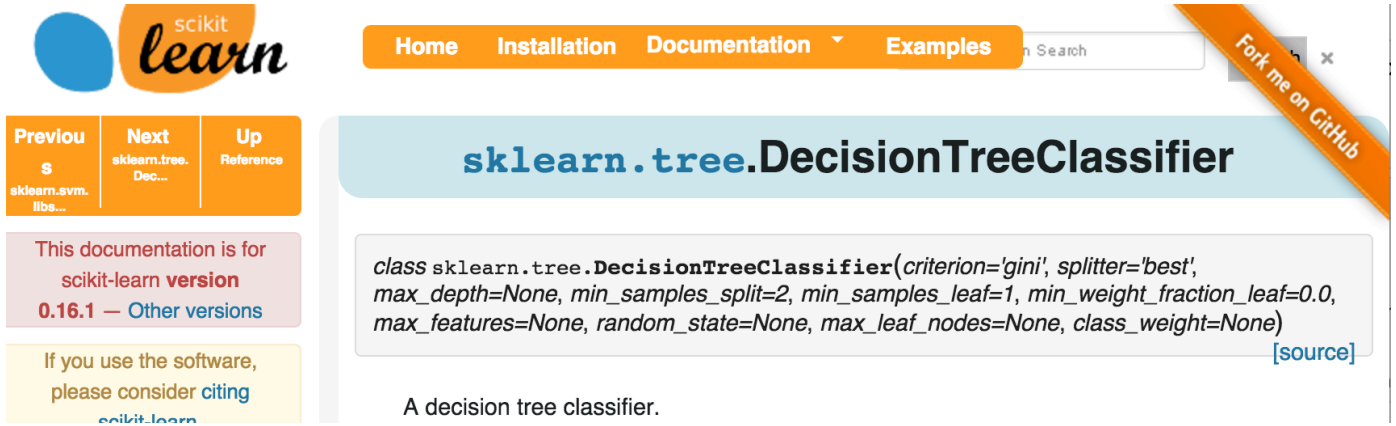




Progress Bar

- One-hot encoding of categorical features
- Transformers and estimators
- **Understanding and tuning algorithms**

Decision Trees take some tuning, though



- many tunable parameters available
- set parameters when instantiating the classifier
- important handle for avoiding overfitting
- how to pick best parameter values?

stay tuned for last part of this workshop

in a word: GridSearchCV

Progress Bar

- One-hot encoding of categorical features
- Transformers and estimators
- **Understanding and tuning algorithms**

Random Forest Classifier

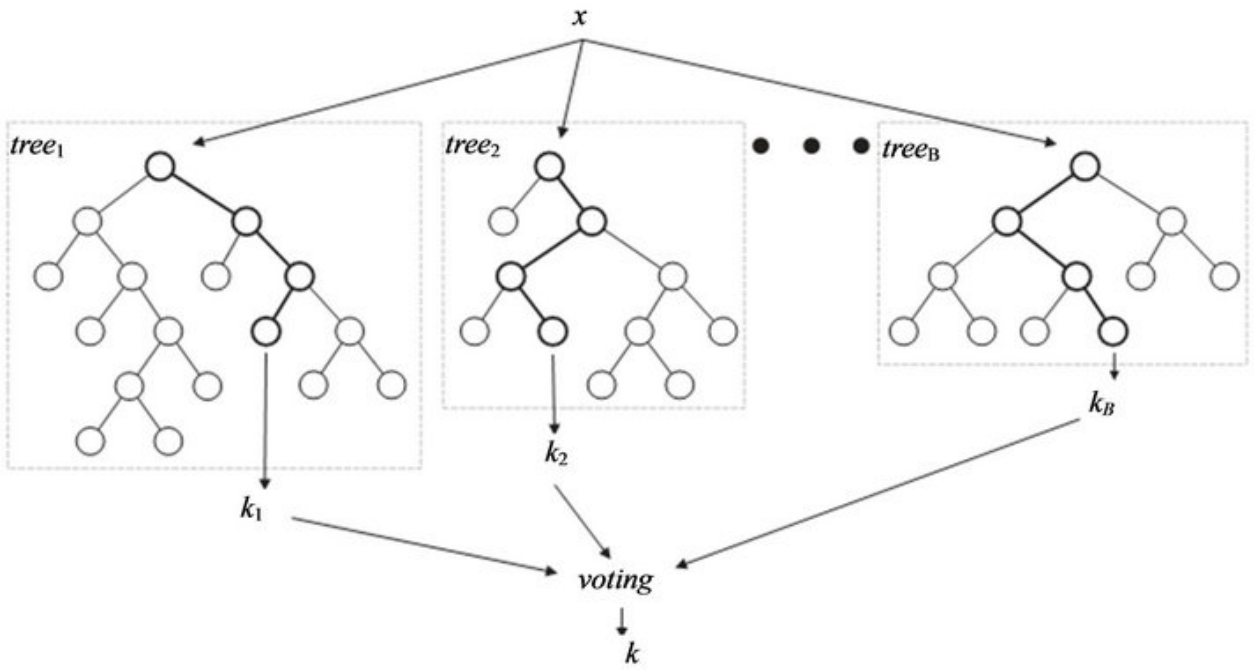


image source: http://file.scirp.org/Html/6-9101686_31887.htm

Progress Bar

- One-hot encoding of categorical features
- Transformers and estimators
- **Understanding and tuning algorithms**

Practicalities of working with random forests

They are awesome

Robust in the face of heterogeneous data

- numerical, boolean, etc.
- need not normalize input features

Good performance

- in meta-studies of algorithm performance for a variety of tasks, RF consistently at/near top

Gracefully handles nonlinearities and correlations

- don't need explicit interaction/nonlinear terms, like you do in regressions

Usually low-bias

- Trees are low-bias, high-variance
- Ensembling mitigates the high variance of any one tree

They are difficult

LOTS of parameters to potentially tune

- like decision tree, but more so

Black-boxy

- how does a new point get classified? Really difficult to reconstruct decision-making process

More difficult to visualize their results

- Different trees can give different predictions for same data point

Status Report

- **random forest classifier seems to give best score**
  - how is score defined/calculated?
  - what parameters are best?
- **features cleaned up and categories handled properly**
  - is there a way to pick out the best ones, and just use those?
- **in general, is there a smarter/cleaner way to proceed?**

YES.

End-to-end workflows using Pipeline and GridSearchCV

- Feature selection, and better model evaluation
- Pipelines
- GridSearchCV

SelectKBest feature selection

before dummying: 39 features

after: 3k+, even with aggressive filtering on highest-multiplicity fields

problems with having tons of features

- slows down computation
- likely do not need all the features to capture trends
- overfitting

code up sklearn.feature_selection.SelectKBest

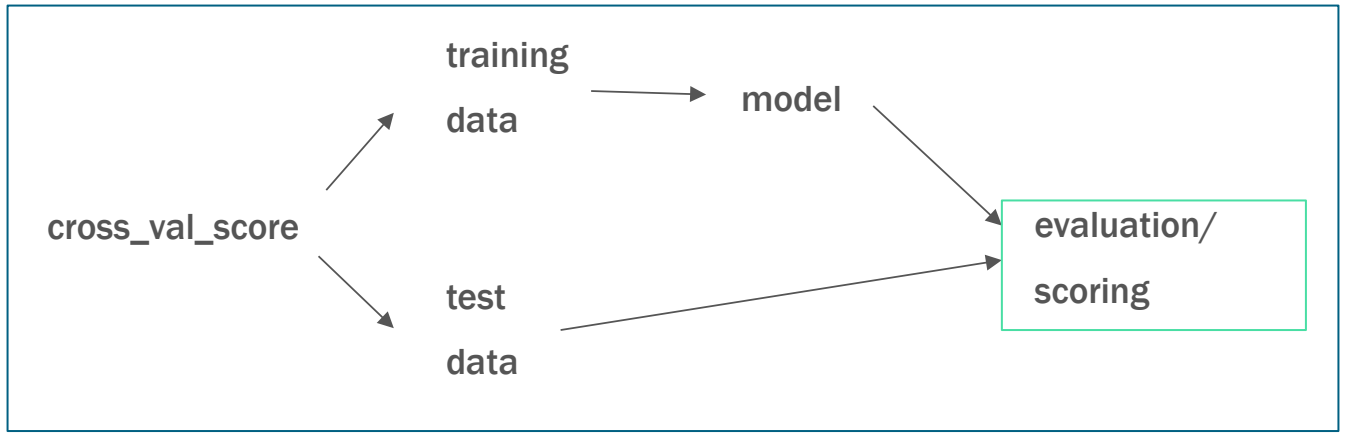
to select the 200 best features for using in our future models

Progress Bar

- **feature selection, and better model evaluation**
- Pipeline
- GridSearchCV

Training/testing split

id	source	source_type	source_class	functionality	
69572	spring	spring	groundwater	1	training data
8776	rainwater harvesting	rainwater harvesting	surface	1	
34310	dam	dam	surface	1	
67743	machine dbh	borehole	groundwater	0	
19728	rainwater harvesting	rainwater harvesting	surface	1	
9944	other	other	unknown	1	
19816	machine dbh	borehole	groundwater	0	
54551	shallow well	shallow well	groundwater	0	test data
53934	machine dbh	borehole	groundwater	0	
46144	shallow well	shallow well	groundwater	1	



Progress Bar

- **feature selection, and better model evaluation**
- Pipeline
- GridSearchCV

cross_val_score → classification_report

	precision	recall	f1-score	support
0	0.80	0.56	0.66	5007
1	0.12	0.00	0.01	942
2	0.69	0.92	0.79	7119
avg / total	0.69	0.72	0.68	13068

$$\text{precision} = \frac{true\ positive}{true\ positive + false\ positive}$$

$$\text{recall} = \frac{true\ positive}{true\ positive + false\ negative}$$

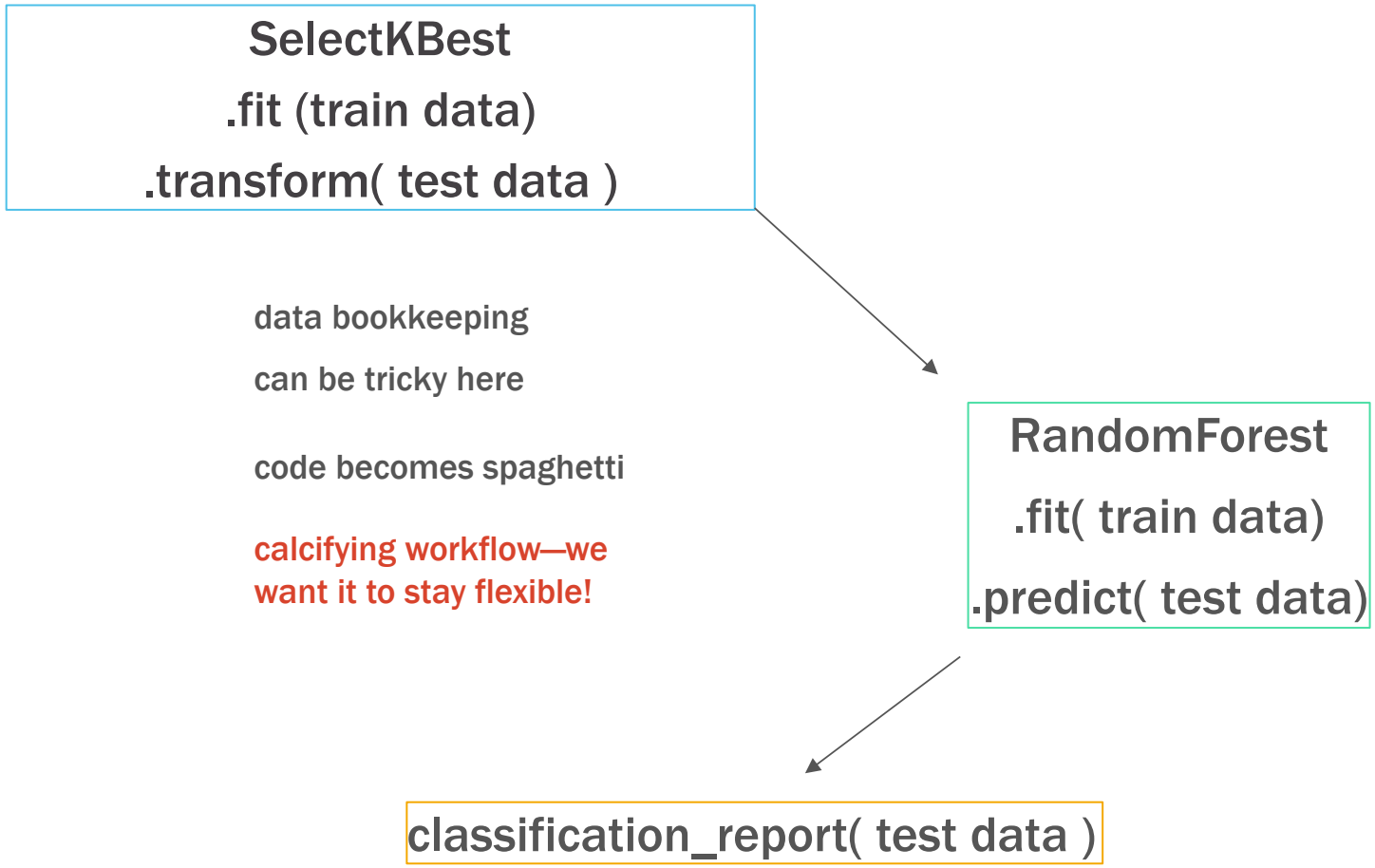
$$\text{F1 score} = 2 * \frac{precision * recall}{precision + recall}$$

support = no. of true occurences of a class

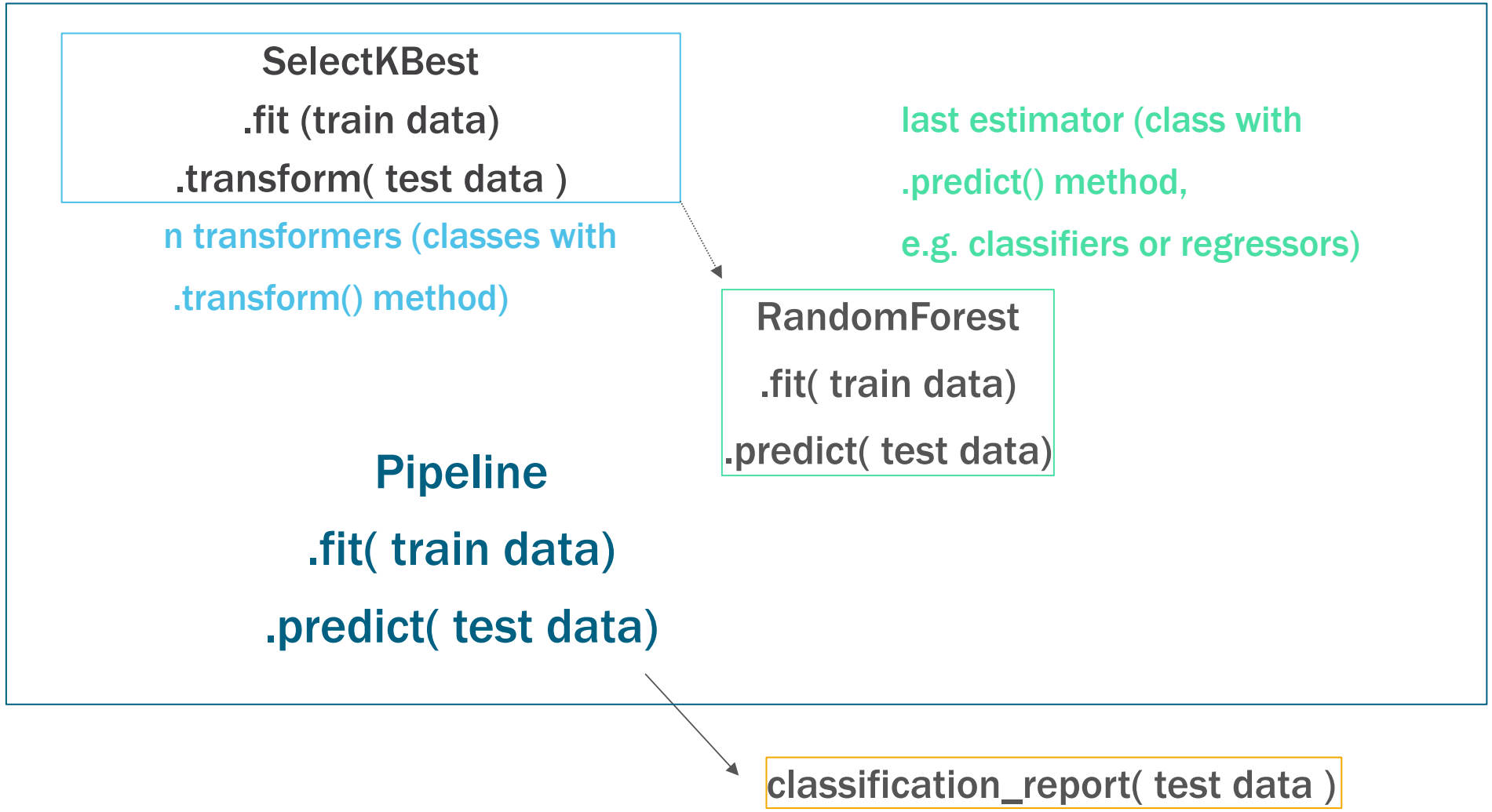
Progress Bar

- **feature selection, and better model evaluation**
- Pipeline
- GridSearchCV

Pipeline theory and uses (I)



Pipeline theory and uses (II)



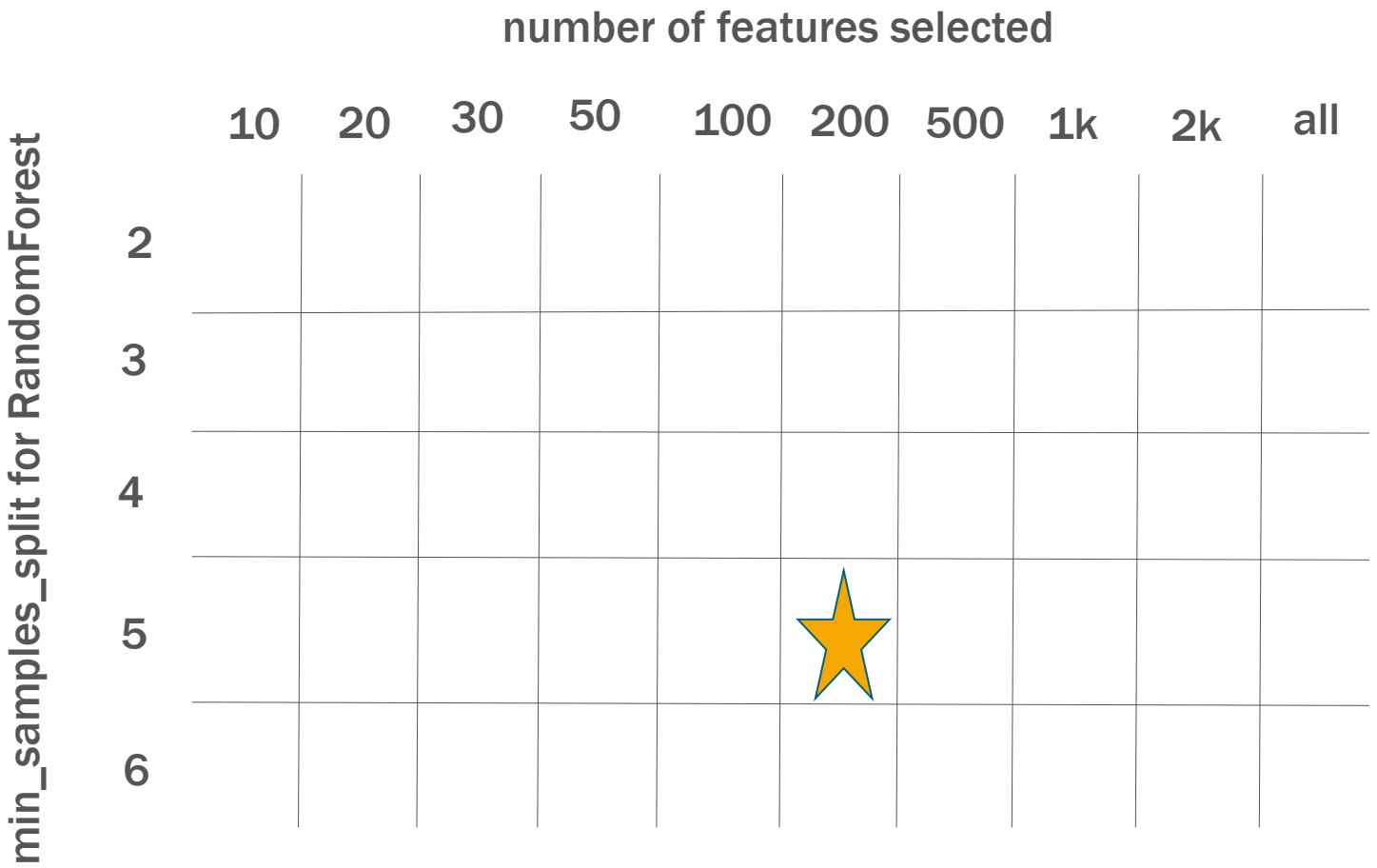
Pipeline Code

```
kbest = SelectKBest(k=100)
clf = RandomForestClassifier()
steps = [("feature_selection", kbest),
         ("random_forest", clf)]

pipeline = sklearn.pipeline.Pipeline(steps)

pipeline.train( X_train, y_train )
predictions = pipeline.predict( X_test )
```

Progress Bar

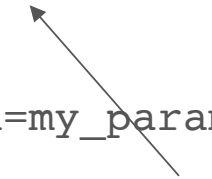
- feature selection, and better model evaluation
- **Pipeline**
- GridSearchCV

Tuning analysis parameters



GridSearchCV

```
steps = [('step_name_1', feature_selector),
         ('step_name_2', classifier)]
pipeline = sklearn.pipeline.Pipeline( steps )

my_parameter_grid = dict(step_1_name__k=[v1, v2, v3],
                         step_2_name__min_samples_split=[u1, u2, u3])

cv = sklearn.grid_search.GridSearchCV(pipeline, param_grid=my_parameter_grid)

cv.fit(X_train, y_train)
cv.predict( X_test )

etc…
```



need to follow this (kinda funny, but you'll get used to it) naming convention for your parameters !!!

Conclusions

- **Pandas and sklearn can (often) get you pretty far**
  - pandas for data input/output and manipulation
  - sklearn for transforms and model building/evaluation
  - the docs are your friend
- **Even simple problems take work to get model-building-ready**
  - feature transforms and selection
  - defining problem scope
  - you never truly know if something will work until you try it
- **Constant tension between theoretical best practices and quickest/easiest way to proceed**
  - “Easy reading is damn hard writing, and vice versa” -- Nathaniel Hawthorne
  - iterate, iterate, iterate

Make it maintainable

- without it crashing

Our twist today

Scenario: you are the data scientist in serving a team of well inspectors in Africa. They use your model to prioritize future well inspections.

Every day, they go out and measure well functionality, along with many attributes of the wells they've seen. Their goal is to spend their valuable time inspecting the highest-risk wells.

They routinely package up this data and send it back to you.

Your job:

- **incorporate the new data into the model QUICKLY**
- **tell the team if/where the model has changed**
- **say whether the model is getting better, worse, or staying the same**

Modularizing a machine learning pipeline

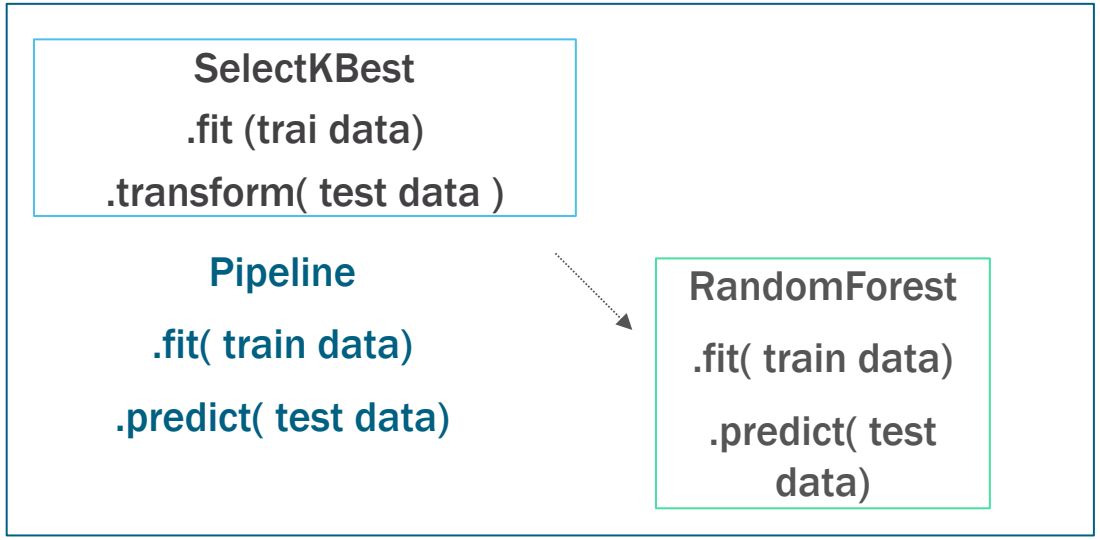
Our code currently exists as one big jupyter notebook

- **hard to version control**
- **hard to share**

Goal

- **analyst specifies updated dataset and the model builds on that data**
- **model builds in a way that's consistent with previous builds**
    - same data preprocessing steps
    - same feature selection and algorithm/parameters
    - same metrics
    - comparable test set
- **easy model comparison as we add in more data**
    - has the model improved, gotten worse, stayed the same?
    - which predictions have changed?

Sklearn pipeline reminder...



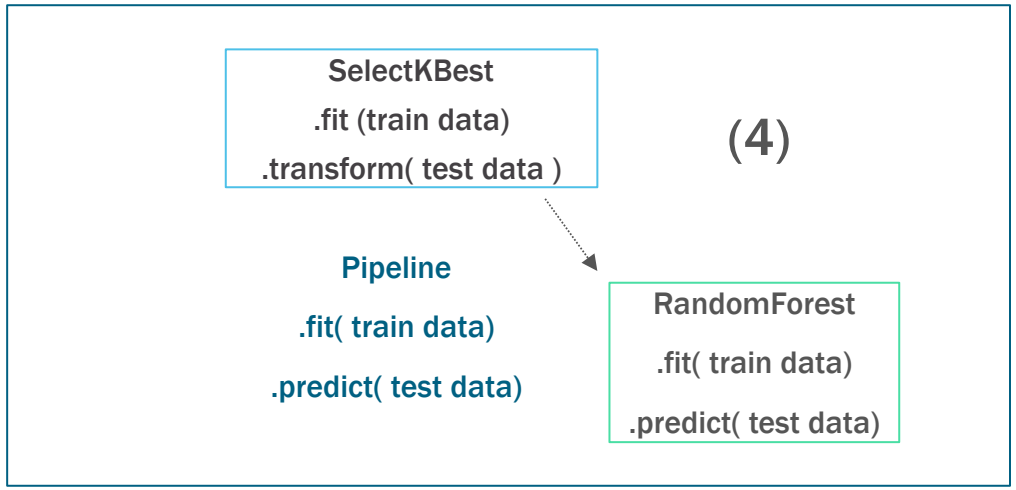
… Now as a piece in a larger workflow

(1) Data ingestion

(2) Define train/test sets

(3) Dummy features

Transform labels



(5) Evaluate/use predictions

Notebook→Modularized Code

All code from this point forward is going into python scripts.

Time to get git involved.

We will proceed in steps. Each one builds upon the last.
Solutions are in the solutions/ folder in the git repo.

Expectations for this part of the workshop

- **Data will change:**
  - In the git repo there is a script data_setup.py
  - Assumes data files are called `well_data.csv` and `well_labels.csv`
  - Breaks datasets into 6 approximately equal-sized chunks
    - “dataset_1.csv”
    - “dataset_2.csv”
    - etc…
  - Mocks up the “new shipments of data” scenario
  - I have some backup copies on thumb drives
- **Pace will change:**
  - brief lecture, then 3-20 minutes of hands-on-keyboard time
    - 16 steps total
  - I’ll post a timer so we stay on track
  - If you don’t complete a step, DON’T PANIC. There are solutions in the git repo.

Step 1: Start a script

- **Create a python script**
- **Create a pandas dataframe that reads in `dataset1.csv`**
- **Use the if `__name__=="__main__"` convention**
- **Name your script `model.py`**

model.py

```
import pandas as pd


def main():
    file_name =
“datasets/dataset_1.csv”
    df = pd.read_csv(file_name)
    print(df.head())


if __name__==“__main__”:
    main()
```

Data ingestion

- **What the user expects**
  - Indicate which dataset to use (e.g. dataset_1.csv)
  - Each dataset should have a REPRODUCIBLE train/test split
    - Old data split between training/testing in a way that's consistent with before
    - New data split between training/testing in a way that's reproducible in the future (because today's new data is tomorrow's old data)
- **What you need to make to accommodate this expectation**
  - My approach: when we see a new dataset
    - recognize from the dataset name that it's new
    - create train/test datasets
    - WRITE the train/test datasets out (and use them for that dataset going forward)

Step 2: Handling data in a function

- **Write a function to handle the datasets**
  - **name:** `handle_data_files`
  - **inputs: name of data file (e.g.** “`dataset_1.csv`”)
  - **outputs:** `train_df, test_df`
  - **other functionality**
    - **write train_df and test_df to the** `datasets/` **directory**
    - **name them** `train_X.csv` **and** `test_X.csv`
    - **where “X” is the dataset number, e.g.** `1` **for** `dataset_1.csv`)

```
import pandas as pd
def handle_data_files(file_name):
    # you fill this in
    return test_df, train_df

def main():
    file_name = “datasets/dataset_1.csv”
    test_df, train_df = handle_data_files(file_name)

if __name__==“__main__”:
    main()
```

Step 3: Retrieve training/testing data if it already exists

- In Step 2, we added code to write training/testing files when we handle the data
- We want to simply retrieve these datasets in the future when they exist
- Add checking to `handle_data_files()`:
  - if a file named `datasets/train_X.csv` already exists, open it (and `test_X.csv`) into pd.DataFrame and return dataframe(s)
  - Hint: using `Path` (in the `pathlib` module) might help

Step 4: Add (and use!) a logger

- Use the `logging` module to set up a logger
  - Like print, but better
    - Set the log level
    - Turn on/off messages together
- Write a few logging messages
- NO PRINT STATEMENTS FOR THE REST OF THE DAY
  - `logger.info()`
  - `logger.debug()`

Scripted vs. Repeatable Data I/O

The easy way

```
import pandas as pd
from sklearn.model_selection import train_test_split
df = pd.read_csv("dataset_1.csv")
train_df, test_df = train_test_split(df, test_size=0.2,
random_state=42)
```

The right way

```
from pathlib import Path
import logging
import pandas as pd

from sklearn.model_selection import train_test_split

logging.basicConfig(level=logging.INFO)
logger = logging.getLogger(__name__)


def handle_data_files(infile_name):
    """
    For a given infile name, open the file into a
    pandas Dataframe and determinately split into training
    and testing subsets. Save those as
    `train_X.csv` and `test_X.csv` in the `datasets`
    directory (where `X` is the integer index of the
    dataset and should be present in infile_name)


    Params
    ------
    infile_name : str
        name of the input dataset, standard format
        is dataset_X.csv, where "X" is an integer index


    Returns
    -------
    train_df, pd.DataFrame
        80% of input data, for training
    test_df, pd.DataFrame
        20% of input data, for testing
    """
    # assume filename format: dataset_X.csv where
    # X is an integer that indexes the file
    df = pd.read_csv(infile_name)


    dataset_id = infile_name.split(".csv")[0].split("_")[1]
    train_name =
"../../datasets/train_{}.csv".format(dataset_id)
    test_name = "../../datasets/test_{}.csv".format(dataset_id)


    train_file = Path(train_name)
    if train_file.is_file():
        logger.info("found {}, retrieving".format(train_name))
        train_df = pd.read_csv(train_name)
        test_df = pd.read_csv(test_name)
    else:
        logger.info("{} not found, creating".format(train_name))
        df = pd.read_csv(infile_name)
        train_df, test_df = train_test_split(df, test_size=0.2,
random_state=42)
        # index=False prevents an unwanted column named
"Unnamed: 0"
        # from being appended
        #
https://stackoverflow.com/questions/36519086/pandas-how-
to-get-rid-of-unnamed-column-in-a-dataframe
        train_df.to_csv(train_name, index=False)
        test_df.to_csv(test_name, index=False)


    return train_df, test_df


def main():
    file_name = "../../datasets/dataset_1.csv"
    train_df, test_df = handle_data_files(file_name)
    logger.info("All done!")


if __name__=="__main__":
    main()
```

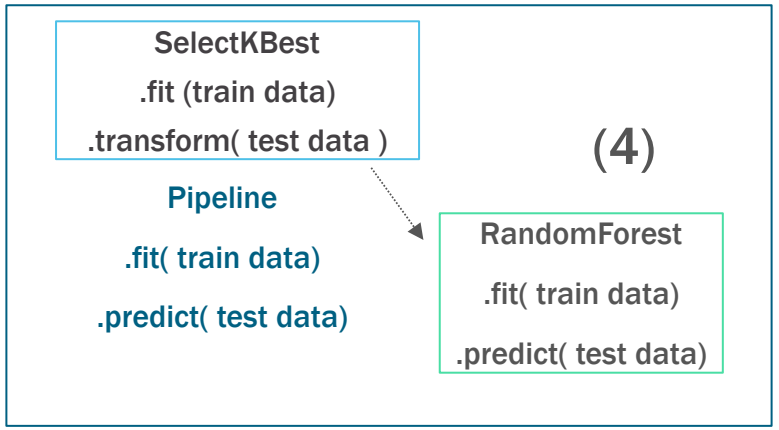
Progress Report

✅ (1) Data ingestion

✅ (2) Define train/test sets

(3) Dummy features

Transform labels



(5) Evaluate/use predictions

Up next: set up framework for steps 3 and 4

Step 5: Structuring the modeling flow

- Goal: functions for building and scoring models
- DO NOT actually fill in code yet! Just function calls/docstrings
- `build_model()`
  - input: train_df
  - output: trained model
- `run_predictions()`
  - inputs: trained model, test_df
  - outputs: predictions of the model on test_df
- call these functions in `main()`

```
import pandas as pd
def handle_data_files(file_name):
    #stuff
    return train_df, test_df
def build_model(train_df):
    # stuff
    return model
def run_predictions(model, test_df):
    # stuff
    return predictions
def main():
    file_name = "datasets/dataset_1.csv"
    test_df, train_df = handle_data_files(file_name)
    model = build_model(train_df)
    predictions = run_predictions(model, test_df)
if __name__=="__main__":
    main()
```

Step 6: YOLO, let's try to build the model

- Integer encode the labels
  - create a function based on `label_map` (in `scripted_model.ipynb`)
    - name: `encode_labels()`
    - inputs: dataframe (training or testing), name of column to encode (default: `"status_group"`)
    - outputs: same dataframe, but indicated column is now integer encoded
  - Use `sklearn.preprocessing.LabelEncoder`
  - split off encoded labels so we have `X` (features) and `y` (labels) for training
- In `build_model()`, Create `sklearn.ModelPipeline`
  - `SelectKBest(k=100)`
  - `RandomForestClassifier()`
- Sure, why not: try to fit your model
  - (it's not going to work)

```
def encode_labels(df, column_name="status_group"):
    # you fill this in
    return df
def build_model(train_df):
    train_df = encode_labels(train_df)
    # more stuff (e.g. ModelPipeline) here
    return model
def run_predictions(model, test_df):
    # stuff
    return predictions
def main():
    file_name = "datasets/dataset_1.csv"
    test_df, train_df = handle_data_files(file_name)
    model = build_model(train_df)
    predictions = run_predictions(model, test_df)
if __name__=="__main__":
    main()
```

Progress Report

✅ (1) Data ingestion

✅ (2) Define train/test sets

(3) Dummy features

Transform labels ✅

SelectKBest

.fit (train data)

.transform(test data)

Pipeline

.fit(train data)

.predict(test data)

(4)

RandomForest

.fit(train data)

.predict(test data)

(5) Evaluate/use predictions

Up next: fill in framework for steps 3 and 4

Step 7: Think about features

- Each column has one fate:
  - Drop it
  - Leave it in, do not transform it
  - Dummy it
- Create 2 functions
  - function 1: named `get_cols_to_dummy`, no inputs, output is a list of columns
    - `basin, region, lga, recorded_by, extraction_type, extraction_type_group, extraction_type_class, management, management_group, payment, payment_type, water_quality, quality_group, quantity, quantity_group, source, source_type, source_class, waterpoint_type, waterpoint_type_group`
  - function 2: named `get_cols_to_drop`, no inputs, output is a list of columns
    - `date_recorded, funder, installer, wpt_name, subvillage, ward, public_meeting, permit, scheme_name, scheme_management`
- Call these functions in `build_model()`
- Drop columns from `train_df`

Step 7 code hints

```
def get_cols_to_dummy():
    # return list of columns to dummy
def get_cols_to_drop():
    # return list of columns to drop
def encode_labels(df, column_name="status_group"):
    # you fill this in
    return df
def build_model(train_df):
    cols_to_dummy = get_cols_to_dummy()
    cols_to_drop = get_cols_to_drop()
    # use cols_to_drop to get rid of unwanted columns from train_df
    # we will dummy here in Step 8
    train_df = encode_labels(train_df)
    # more stuff (e.g. ModelPipeline) here
    return model
def run_predictions(model, test_df):
    # stuff
    return predictions
def main():
    file_name = "datasets/dataset_1.csv"
    test_df, train_df = handle_data_files(file_name)
    model = build_model(train_df)
    predictions = run_predictions(model, test_df)
if __name__=="__main__":
    main()
```

Step 8: It's dummy time

- Problematic option #1: `pd.get_dummies()`
  - We will need to run future datasets through this code
  - We want the dummying to be consistent
    - categories today: ['chocolate', 'vanilla']
    - categories tomorrow: ['chocolate', 'vanilla', 'strawberry']
  - in scikit-learn terms, we'd say that `pd.get_dummies()` has `.fit()` functionality, but not `.transform()`
    - `.fit()`: create a dummying scheme from this dataset, and apply it to this dataset
    - `.transform()`: create dummying scheme on dataset A, apply to dataset B
- Problematic option #2: sklearn.preprocessing.OneHotEncoder()
  - has `.fit()` AND `.transform()`
  - can ignore new classes (`handle_unknown="ignore"` in constructor)
  - BUT only works if the column is ALREADY INTEGERS

Step 8: It's dummy time

- **Problematic option #2:** `sklearn.preprocessing.OneHotEncoder()`
  - **has** `.fit()` **AND** `.transform()`
  - **can ignore new classes (**`handle_unknown="ignore"` **in constructor)**
  - **BUT only works if the column is ALREADY INTEGERS**
- **Solution: encode strings as integers (reproducibly), then use** `OneHotEncoder()`
- `LabelEncoder` **as a shortcut for the integer encoding**
  - **only really meant for y values, not X ones**
  - **has** `.fit()` **and** `.transform()` **methods (that's good!)**

Step 8: It's dummy time

- Put this code in `build_model()`
- For each column in list of columns to dummy:
  - use `LabelEncoder` to integer encode the strings
  - use `OneHotEncoder()` to one-hot encode the integers
  - use `LabelEncoder.classes_` to get the original string values, and use those to label the one-hot encoded columns
  - attach labeled, one-hot encoded columns to `train_df`
  - drop original column

city	city Chicago	city San Francisco	city New York
Chicago	1	0	0
San Francisco	0	1	0
Chicago	1	0	0
Chicago	1	0	0
New York	0	0	1
San Francisco	0	1	0

Step 9: Clean it up

- Your `build_model()` function is getting long
  - Break out the feature dummying/dropping into its own function
    - name: `feature_preprocessing`
    - inputs: `df_train`
    - outputs: `df_train`, but with columns dropped/dummied

```
def encode_labels(df, column_name="status_group"):
    # you fill this in
    return df
def feature_preprocessing(df):
    df = encode_labels(df)
    cols_to_drop = get_cols_to_drop()
    cols_to_dummy = get_cols_to_dummy()
    # all your code dropping/dummying columns
    return df
def build_model(train_df):
    train_df = feature_preprocessing(train_df)
    # more stuff (e.g. ModelPipeline) here
    return model
def run_predictions(model, test_df):
    # stuff
    return predictions
.... etc ...
```

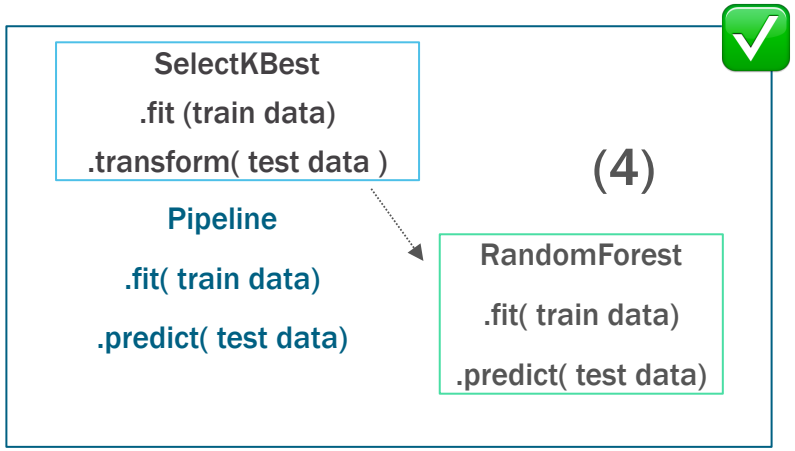
Progress Report

(1) Data ingestion ✅

(2) Define train/test sets ✅

(3) Dummy features ✅

Transform labels ✅



(5) Evaluate/use predictions

Up next: take a stab at step 5;

actually spend a ton of time on step 3

Step 10: Predictions!

- In step 5, we started a function for making predictions. Time to fill it in.
  - name: `run_predictions`
  - inputs: `model` (created/returned in `build_model`), `test_df`
  - outputs: list of predictions
- Welp, `feature_preprocessing()` currently calls `.fit_transform()` for `LabelEncoder` and `OneHotEncoder`, but what we really want is to call `.transform()` using the `LabelEncoder` and `OneHotEncoder` we fit on the training data.
  - Ideally, we would just add the `LabelEncoder` and `OneHotEncoder` to our `Pipeline`, but a few things make this not work in practice:
    - we don't want to dummy all the columns, and the `column_mask` feature in `OneHotEncoder` seems to be broken
    - `LabelEncoder` is meant for y values, not X, and throws an error when you try to use it on X values in a `Pipeline`

Step 10: ~~Predictions!~~ Propagate dummying logic from training to testing data

- **Welp, `feature_preprocessing()` uses `.fit_transform()` for `LabelEncoder` and `OneHotEncoder`, but we really want `.transform()`**
- **Ideally, we would just add the `LabelEncoder` and `OneHotEncoder` to our `Pipeline`, but a few things make this not work in practice:**
  - **we don't want to dummy all the columns, and the `column_mask` feature in `OneHotEncoder` seems to be broken**
  - **`LabelEncoder` is meant for y values, not X, and throws an error when you try to use it on X values in a `Pipeline`**

```
def encode_labels(df, column_name="status_group"):
    # you fill this in
    return df
def feature_preprocessing(df, optional trained LabelEncoder objects):
    df = encode_labels(df)
    cols_to_drop = get_cols_to_drop()
    cols_to_dummy = get_cols_to_dummy()
    if df is the train_df: #rewrite to be real python
        LabelEncoder.fit_transform()
        #store LabelEncoder object for later
    else:
        LabelEncoder.transform()
    return df, LabelEncoders
def build_model(train_df):
    train_df, LabelEncoder objects = feature_preprocessing(train_df)
    # more stuff (e.g. ModelPipeline) here
    return model, LabelEncoders
def run_predictions(model, test_df, LabelEncoders):
    test_df = feature_preprocessing(test_df, LabelEncoders)
    return predictions
```

Propagating dummying logic from training to testing data (one solution)

- In `feature_preprocessing`
  - **Add a dictionary (called** `trained_transformers`**) for storing and returning the trained** `LabelEncoder` **and** `OneHotEncoder` **objects**
  - **Since we want to use** `feature_preprocessing` **for both training AND testing data, make** `trained_transformers` **an optional input parameter to** `feature_preprocessing`
  - **Set default for** `trained_transformers` **to** `None`
  - **If** `trained_transformers` **is** `None`**:**
    - **Call** `.fit_transform()` **on the** `LabelEncoder` **and** `OneHotEncoder`
    - **Fill** `trained_transformers` **with the** `LabelEncoder` **and** `OneHotEncoder`
  - **Else:**
    - **Call** `.transform()` **on the** `LabelEncoder` **and** `OneHotEncoder`
  - **Return** `trained_transformers` **from** `feature_preprocessing` **and build_model, take as parameter into** `run_predictions`

Step 11: Handle new categories

- But wait, there's more!
- `LabelEncoder` has some more fun in store for us
  - say training data has the following columns: ["chocolate", "vanilla"]
  - and test data has ["chocolate", "vanilla", "strawberry"]
  - `LabelEncoder` fit to training data won't know how to transform testing data
    - We saw something similar with `OneHotEncoder`, which has a handy "ignore" flag for this case
    - but not `LabelEncoder`!
- Solution: hack `LabelEncoder` to accommodate new categories
  - add logic to recognize new categories that have shown up in the testing data
  - re-label them as "Other – new"
  - add "Other – new" to `LabelEncoder.classes_`
  - Cry a little

Dummying to accommodate new categories in test data

city	city Chicago	city San Francisco	city New York	city Other
Chicago	1	0	0	0
San Francisco	0	1	0	0
Chicago	1	0	0	0
Chicago	1	0	0	0
New York	0	0	1	0
San Francisco	0	1	0	0

This is a dummying scheme that can accommodate Columbus, Milwaukee, San Jose showing up for the first time in the testing data

However, be sure to add logic that detects new cities (e.g. not NY, SF, Chicago) and change the names to ”Other” so the LabelEncoder knows what to do with them

Oof, that was a ton of work just for dummying

- Yes, yes it was
- What did we learn?
  - Dummying: not as simple as it looks
  - Hard-coding the label categories (in `encode_labels`) was fairly low-drama
  - Once we got the features transformed, the Pipeline was very easy to do `.fit()` and `.predict()`
    - IF we wanted to make this more maintainable, we try another strategy:
      - Hand-write something like `encode_labels` but for arbitrary columns
      - Give it a `.fit()` and a `.transform()` method
      - Then the Pipeline could be
        - `OurCustomLabelEncoder`
        - `OneHotEncoder`
        - `SelectKBest`
        - `RandomForestClassifier`
      - And everything would be much nicer

Step 12: Evaluate models!

- **Add a `classification_report` to your logger, run it on `dataset_1.csv`**
- **Where is the model strong? Where is it weak?**

Step 13: Add data! Compare models!

- Extend the code so it builds 2 models
  - first model: `dataset_1.csv`
  - second model: `dataset_1.csv` AND `dataset_2.csv`
- The second model will have twice as much data—does this improve the weak spots in the first model?

Step 14: Compare models row-by-row

- ”Functional needs repair” (category 1) is toughest to identify
- It gets better as we add more training data
- However, test set also has more data
- Where are the improvements coming from?
  - better predictions on dataset_1?
  - great predictions on dataset_2?
- To answer the question:
  - where model_1 and model_2 are making predictions about THE SAME test cases, does model_2 do better?

```
INFO:__main__:              precision    recall  f1-score   support

          0       0.82      0.91      0.87      1259
          1       0.24      0.09      0.13        65
          2       0.81      0.70      0.75       676

avg / total       0.80      0.81      0.80      2000
```

```
INFO:__main__:              precision    recall  f1-score   support

          0       0.79      0.86      0.83      3350
          1       0.41      0.29      0.34       370
          2       0.80      0.74      0.77      2280

avg / total       0.77      0.78      0.77      6000
```

Step 15: Add a CLI

- **Now: $>** `python model.py`
- **Next: $>** `python model.py –file “dataset_1.csv,dataset_2.csv”`
- **(remove any references inside the code to specific data files)**

Step 16: Unit testing!

- Need pytest installed for this to work
- Usage: $> `py.test`
- This test checks if `feature_preprocessing` actually removes the column it's supposed to remove
  - Note that I've hijacked `get_cols_to_dummy()` and `get_cols_to_drop()` via mocking
- Software testing for data science could be a workshop in its own right
- For adventurous souls: add a test of the dummying capability of `feature_preprocessing`

Conclusions

- **THANK YOU for coming and participating!**
- **Part 1 conclusions**
  - data science isn't so bad!
  - everything is easy with pandas and sklearn
  - we can get something up and running in ~an hour (from a clean dataset)
- **Part 2 conclusions**
  - maintainable data science code is really hard to write!
  - Why? A lot of reasons, but in my opinion, it's largely because data science entangles ALGORITHMS with DATA
  - really effective data scientists put as much (more?) thought into software best practices as they do into modeling
  - **hopefully next time you have to do this, it'll be a little easier because of our time here today**

Thank You