Case Study

Picwell Model Harness

Clara Bennett

May 29, 2019

Setting

Picwell, ca. 2016

- Picwell provides recommendations for Medicare and employer-provided health insurance
- Product incorporates several predictive models: primary model predicts healthcare costs per plan, per household
- I led a product (eng) team, working on all parts of the stack, including data-pipeline and data-infrastructure work

The Problem

Developing our predictive models for production is hard

- Feature engineering and data prep is all DIY
- Model artifacts don't know feature context
- Models are trained offline; prediction is online $\rightarrow$ model updates require parallel changes to both
- Raw training and prediction data formatted differently
- If featurizations for training/prediction don't match, predictions are invalid $\rightarrow$ **silent failure**

Iterating on our models collaboratively is also hard

- Feature experimentation is messy
- Even with shared training scripts, isolating individual features for reuse may be nontrivial
- Lack of structure encourages reimplementation or copy-paste → both error-prone

The Solution

"Model Harness"

- Build a tool that bundles context and featurization machinery together with the model artifact
- Dual purpose:
  - Allow models to be deployed without code changes to model server (in most cases)
  - Make it easier for data scientists to experiment with new features or new combinations of existing features

Initial efforts

- The model harness was seeded with:
  - A collection of featurization utilities
  - An `sklearn` model subclass that supported named features
- First implementation was an abstract class that required subclasses to implement two featurization methods
- Built-in test battery ensured API compatibility

```
# Household (API format)
{
    members: [
        {age: int, gender: choice('male', 'female'), drugs: [int]}
    ],
    pregnancy: bool,     # if anyone in the household is or plans to be
    ...                  # other household-level info not used in cost prediction
}

# Raw training data; possibly multiple records per household_id
{
    household_id: int,
    age: int,
    gender: choice('male', 'female'),
    pregnant: bool,
    <drug_id>: bool,
    ...
}
```

```python
class HarnessModel(object):
    def __init__(self, **model_params):
        self.sklearn_model = NamedFeatureModel(**model_params)

    @staticmethod
    def featurize_household(household):
        raise NotImplementedError

    @staticmethod
    def featurize_raw_data(record):
        raise NotImplementedError

    def train(self, records, labels):
        features = records.apply(self.featurize_raw_data)
        self.sklearn_model.fit(features, labels)

    def predict(self, household):
        return self.sklearn_model.predict(self.featurize_household(household))
```

Better, but still problematic

- Training and prediction featurization code now in one place, but still implemented twice
- Experimentation still promotes copy-paste or reimplementation of features
- Attemps to DRY up model code led to multiple inheritance $\rightarrow$ unfriendly to data scientists (science/econ/math trained)
- Package version hell on shared compute clusters

Model Harness 2.0

- I wanted the harness tool to be composable, rather than inheritance-based, but wasn't sure how to achieve that
- Coworker had idea to eliminate double-implementation of features by converting both raw training data and API prediction data into a common format
- Canonical data format idea provided the missing piece for my composability idea

Model object accepts feature specs on instantiation

```
class HarnessModel(object):
    def __init__(self, feature_specs, **model_params):
        self.feature_specs = feature_specs      # see next slide
        self.sklearn_model = NamedFeatureModel(**model_params)

    def canonicalize_training_data(self, raw_training_records):
        ...
    def canonicalize_api_data(self, household):
        ...
    def _featurize(self, canonical_record):
        return [fs.featurize(canonical_record) for fs in feature_spec]

    def train(self, records, labels):
        canonical_records = self.canonicalize_training_data(records)
        features = canonical_records.transform(self._featurize)
        self.sklearn_model.fit(features, labels)

    def predict(self, household):
        canonical_record = self.canonicalize_api_data(household)
        return self.sklearn_model.predict(self._featurize(canonical_record))
```

Separately defined feature classes only implement featurization once

```
class Feature(object):
    def featurize(self, canonical_record):
        raise NotImplementedError

class CancerTreatment(Feature):
    __drug_ids = (1, 2, 3, 4, 5)

    def featurize(self, canonical_record):
        return any(drug_id in self.__drug_ids
                   for drug_id in canonical_record.drugs)
```

Model features are easily changed without modifying library code

```
model1 = HarnessModel([Feature1(), Feature2()])
model2 = HarnessModel([Feature1(), Feature3()])
```

2.0 Benefits

- Implementations of features* are isolated and independently testable
- Feature composability supports easy experimentation
- Add-only feature library ameliorates dependency hell
- Positive feedback from engineers **and** data scientists 🎉

---

* Almost all

Demographic feature exception

- Harness 2.0 only supports 1:1 relationship between input canonical records and featurized records
- Household aggregation in the family model happens during canonicalization, simply summing for most fields
- Age and gender are more complex → featurization for those fields baked into the model → subject to the same problems as before
- Okay choice practically, because age and gender family features were fairly mature at this point, but limiting for future

Age and gender features baked into canonicalization step

```
class FamilyHarnessModel(object):
    def canonicalize_training_data(self, raw_training_records):
        household_groups = raw_training_records.groupby('household_id')
        demo_features = household_groups.transform(
            self._extract_age_gender_buckets)
        other_features = household_groups.transform(self._canonicalize).sum()
        return demo_features.merge(other_features)

    def canonicalize_api_data(self, household):
        ...

class IndividualHarnessModel(object):
    def canonicalize_training_data(self, raw_training_records):
        demo_features = raw_training_records[['age', 'gender']]
        other_features = raw_training_records.transform(self._canonicalize)
        return demo_features.merge(other_features)

    def canonicalize_api_data(self, household):
        ...
```

```
# Individual canonical format
{
    age: int,
    gender: choice('male', 'female'),
    pregnant: bool,
    <drug_id>: bool,
    ...
}

# Family canonical format
{
    male_0_18: int,
    male_19_over: int,
    female_0_18: int,
    female_19_45: int,
    female_46_over: int,
    pregnant: bool,
    <drug_id>: bool,
    ...
}
```

Hypothetical future work

- I'd like to have made the harness models 100% composable, but I wasn't around to work on the next iteration of the tool
- Passing multiple canonical records to the family featurizer would allow aggregation to occur in the featurization step
- The age/gender buckets could then be written as Feature subclasses, and other features could also have custom aggregations per family

```python
class Feature(object):
    def featurize_individual(self, canonical_record):
        raise NotImplementedError

    def featurize_household(self, canonical_records):
        try:
            return sum(self.featurize_individual(cr) for cr in canonical_records)
        else:
            raise NotImplementedError

class AgeGenderBucket(Feature):
    def __init__(self, age_min, age_max, gender):
        self._age_min = age_min
        self._age_max = age_max
        self._gender = gender

    def featurize_individual(self, canonical_record):
        return (self._age_min <= canonical_record.age < self._age_max
            and canonical_record.gender == self._gender)
```

```python
class HarnessModel(object):
    def __init__(self, feature_specs, **model_params):
        ...
    @staticmethod
    def canonicalize_training_data(raw_training_record):
        ...
    @staticmethod
    def canonicalize_api_data(household):
        ...
    def _featurize(self, canonical_records):
        raise NotImplementedError

    def train(self, records, labels):
        ...     # use _featurize agnostically
    def predict(self, household):
        ...     # use _featurize agnostically
```

```python
class IndividualModel(HarnessModel):
    def _featurize(self, canonical_records):
        return canonical_records.apply(
            [fs.featurize_individual for fs in feature_specs])

class FamilyModel(HarnessModel):
    def _featurize(self, canonical_records):
        return canonical_records.groupby('household_id').apply(
            [fs.featurize_household for fs in feature_specs])


...

model = FamilyModel([AgeGenderBucket(0, 18, 'male'),
                     AgeGenderBucket(0, 18, 'female'),
                     AgeGenderBucket(19, 200, 'male'),
                     AgeGenderBucket(19, 200, 'female'),
                     ...])
```

Questions?