Machine Learning Systems Design

Lecture 7:

Model Evaluation by [Goku Mohandas]

Evaluation for RecSys by [Chloe He]

Stanford University

Goku Mohandas

Founder @ MadeWithML
AI Researcher @ Apple
ML & Product Lead @ an oncology informatics startup (acq. by Invitae)

→ Find Goku on Twitter and LinkedIn

MLOps

> Learn how to apply ML to build a production grade product and deliver value. → GokuMohandas/MadeWithML

📦 **Purpose**
- Product
- System design
- Project

🔢 **Data**
- Labeling
- Preprocessing
- Exploration
- Splitting
- Augmentation

📈 **Modeling**
- Baselines
- Evaluation
- Experiment tracking
- Optimization

📝 **Scripting**
- Packaging
- Organization
- Logging
- Documentation
- Styling
- Makefile

📦 **Interfaces**
- Command-line
- RESTful API

✅ **Testing**
- Code
- Data
- Models

♻️ **Reproducibility**
- Git
- Pre-commit
- Versioning
- Docker

🚀 **Production**
- Dashboard
- CI/CD workflows
- Infrastructure
- Monitoring
- Feature store
- Pipelines
- Continual learning

Background

- Background in health (informatics, materials, genomics) from Johns Hopkins
- ML + health (informatics, time-series, etc.) at Georgia Tech
- Co-founded a rideshare analytics app to predict surge locations (HotSpot)
- Worked on applied NLP (+ product) at Apple
- ML + Product lead at an oncology informatics startup (acquired by Invitae)
- Teaching, advising + developing at MadeWithML
- Future: back to health as foundations are currently established

Always happy to chat if you need help or working on something cool!
➡ Connect with me on Twitter and LinkedIn

Agenda

1. Task, splitting & baselines set up
2. Metrics (coarse/fine-grained)
3. Confusion matrix
4. Confidence learning (+ calibration)
5. Manual slices
6. Generated slices (explicit & hidden stratification)
7. Evaluating evaluations (CI/CD suite)
8. Testing ML
9. Evaluation reports (dashboards / cards)
10. Monitoring ML
11. Evaluation startup ideas 💡

👉 Follow along using **MadeWithML**'s lesson and interactive notebook.

My take: *Evaluation is one of the most underserved yet critical data-centric aspects of ML systems design that can enable true reliability and programmatic iteration.*

Set up

- Full set up available at [MadeWithML]
- Skipping most of product and data topics so we can focus on evaluation
- Quickly cover task, splitting and baselines since they have strong ties to evaluation
- Also cover aspects of testing, dashboards, CI/CD & monitoring
- Jumping b/w code blocks and content found → here

focus for today →

Task

- [Simplified] Predict topic tags (from a specified set of of tags) for a given project (text).

```python
# Load projects
url = "https://raw.githubusercontent.com/GokuMohandas/MadeWithML/main/datasets
projects = json.loads(urlopen(url).read())
print (json.dumps(projects[-305], indent=2))
```


```
{
  "id": 324,
  "title": "AdverTorch",
  "description": "A Toolbox for Adversarial Robustness Research",
  "tags": [
    "code",
    "library",
    "security",
    "adversarial-learning",
    "adversarial-attacks",
    "adversarial-perturbations"
  ]
}
```

Task

- Using a project’s text + description to predict relevant tags (>30 occurrences).

```
@widgets.interact(min_tag_freq=(0, tags.most_common()[0][1]))
def separate_tags_by_freq(min_tag_freq=30):
    tags_above_freq = Counter(tag for tag in tags.elements()
                              if tags[tag] >= min_tag_freq)
    tags_below_freq = Counter(tag for tag in tags.elements()
                              if tags[tag] < min_tag_freq)
    print ("Most popular tags:\n", tags_above_freq.most_common(5))
    print ("\nTags that just made the cut:\n", tags_above_freq.most_common()[-5:])
    print ("\nTags that just missed the cut:\n", tags_below_freq.most_common(5))
```

```
Most popular tags:
 [('natural-language-processing', 429),
  ('computer-vision', 388),
  ('pytorch', 258),
  ('tensorflow', 213),
  ('transformers', 196)]

Tags that just made the cut:
 [('time-series', 34),
  ('flask', 34),
  ('node-classification', 33),
  ('question-answering', 32),
  ('pretraining', 30)]

Tags that just missed the cut:
 [('model-compression', 29),
  ('fastai', 29),
  ('graph-classification', 29),
  ('recurrent-neural-networks', 28),
  ('adversarial-learning', 28)]
```

```
# Filter tags that have fewer than <min_tag_freq> occurrences
min_tag_freq = 30
tags_above_freq = Counter(tag for tag in tags.elements()
                          if tags[tag] >= min_tag_freq)
df.tags = df.tags.apply(filter, include=list(tags_above_freq.keys()))
```

Splitting

- Offline evaluation can’t be trusted if we don’t properly compose our data splits.

Creating proper data splits

What are the criteria we should focus on to ensure proper data splits?

Show answer

- the dataset (and each data split) should be representative of data we will encounter
- equal distributions of output values across all splits
- shuffle your data if it's organized in a way that prevents input variance
- avoid random shuffles if your task can suffer from data leaks (ex. `time-series`)

→ let’s go to the code!

Baselines

Fix the data. *Iterate* on models.

1. Start with the simplest possible baseline to compare subsequent development with. This is often a random (chance) model.
2. Develop a rule-based approach (when possible) using IFTTT, auxiliary data, etc.
3. Slowly add complexity by *addressing* limitations and *motivating* representations and model architectures.
4. Weigh *tradeoffs* (performance, latency, size, etc.) between performant baselines.
5. Revisit and iterate on baselines as your dataset grows.

Fix the models. *Iterate* on data.

- remove or fix data samples (FP, FN)
- prepare and transform features
- expand or consolidate classes
- incorporate auxiliary datasets
- identify unique slices to augment/boost

Many are discovered post offline evaluation!

zooming in on performance today →
but there are many aspects to
model evaluation!

Tradeoffs to consider

When choosing what model architecture(s) to proceed with, what are important tradeoffs to consider? And how can we prioritize them?

Show answer

Prioritization of these tradeoffs depends on your context.

- `performance` : consider coarse-grained and fine-grained (ex. per-class) performance.
- `latency` : how quickly does your model respond for inference.
- `size` : how large is your model and can you support it's storage.
- `compute` : how much will it cost ($, carbon footprint, etc.) to train your model?
- `interpretability` : does your model need to explain its predictions?
- `bias checks` : does your model pass key bias checks?
- `time to develop` : how long do you have to develop the first version?
- `time to retrain` : how long does it take to retrain your model? This is very important to consider if you need to retrain often.
- `maintenance overhead` : who and what will be required to maintain your model versions because the real work with ML begins after deploying v1. You can't just hand it off to your site reliability team to maintain it like many teams do with traditional software.

Baselines

- Set your reproducibility components!
  - `set_seeds()`
  - `get_data_splits(df, train_size=0.7)`
  - `Trainer(object)`
    - `train_step(self, dataloader)`
    - `eval_step(self, dataloader)`
    - `predict_step(self, dataloader)`
    - `train(self, num_epochs, patience, train_dataloader, val_dataloader)`

- Subset for quick initial runs

```
# Shuffling since projects are chronologically organized
if shuffle:
    df = df.sample(frac=1).reset_index(drop=True)

# Subset
if num_samples:
    df = df[:num_samples]
```

➜ let's go to the code to see the baseline implementations

Labels and predictions

```python
# Data to evaluate
device = torch.device("cuda")
loss_fn = nn.BCEWithLogitsLoss(weight=class_weights_tensor)
trainer = Trainer(model=model.to(device), device=device, loss_fn=loss_fn)
test_loss, y_true, y_prob = trainer.eval_step(dataloader=test_dataloader)
y_pred = np.array([np.where(prob >= threshold, 1, 0) for prob in y_prob])
```

```
array([[0., 0., 0., ..., 0., 0., 0.],
       [0., 0., 1., ..., 0., 0., 0.],
       [0., 0., 0., ..., 0., 0., 0.],
       ...,
       [0., 0., 0., ..., 0., 0., 1.]])
```

```
array([[1.86e-03, 4.90e-03, ..., 3.65e-02],
       [9.99e-03, 2.12e-03, ..., 5.34e-03],
       [5.11e-02, 7.21e-03, ..., 3.85e-02],
       ...,
       [4.84e-02, 9.68e-03, ..., 1.63e-01]])
```

```
array([[0, 0, 0, ..., 0, 0, 0],
       [0, 0, 1, ..., 0, 0, 0],
       [0, 0, 1, ..., 0, 0, 0],
       ...,
       [0, 1, 0, ..., 0, 0, 0],
       [0, 0, 0, ..., 1, 0, 0],
       [0, 0, 0, ..., 0, 0, 0]])
```

Coarse-grained metrics

```python
# Metrics
metrics = {"overall": {}, "class": {}}

# Overall metrics
overall_metrics = precision_recall_fscore_support(y_test, y_pred, average="weighted")
metrics["overall"]["precision"] = overall_metrics[0]
metrics["overall"]["recall"] = overall_metrics[1]
metrics["overall"]["f1"] = overall_metrics[2]
metrics["overall"]["num_samples"] = np.float64(len(y_true))
print (json.dumps(metrics["overall"], indent=4))
```

average metrics with class imbalances factored it

```
{
    "precision": 0.7896647806486397,
    "recall": 0.5965665236051502,
    "f1": 0.6612830799421741,
    "num_samples": 218.0

}
```

Fine-grained metrics

```python
# Per-class metrics
class_metrics = precision_recall_fscore_support(y_test, y_pred, average=None)
for i, _class in enumerate(label_encoder.classes):
    metrics["class"][_class] = {
        "precision": class_metrics[0][i],
        "recall": class_metrics[1][i],
        "f1": class_metrics[2][i],
        "num_samples": np.float64(class_metrics[3][i]),
    }

# Metrics for a specific class
tag = "transformers"
print (json.dumps(metrics["class"][tag], indent=2))
```

```
{
  "precision": 0.6428571428571429,
  "recall": 0.6428571428571429,
  "f1": 0.6428571428571429,
  "num_samples": 28.0

}
```

metrics calculated for each unique class

Fine-grained metrics

- Be sure to especially inspect test metrics of classes with low # of samples

```
# Number of samples vs. performance (per class)
f1s = [metrics["class"][_class]["f1"]*100. for _class in label_encoder.classes]
num_samples = np.sum(y_train, axis=0).tolist()
sorted_lists = sorted(zip(*[num_samples, f1s]))
num_samples, f1s = list(zip(*sorted_lists))
```

Confusion matrix

- **True positives (TP)**: prediction = ground-truth
  → learn about where our model performs well.
- **False positives (FP)**: falsely predict sample belongs to class
  → identify *potentially* mislabeled samples.
- **False negatives (FN)**: falsely predict sample does not belong to class
  → identify the model's less performant areas to boost later.

Tip: we should have a scaled version that's tied to labeling and sampling workflows so we can act on our findings from this view.

False positives

topic modeling bert leveraging transformers class based tf idf create easily interpretable topics

True

```
[
  0 : "attention"
  1 : "huggingface"
  2 : "natural-language-processing"
  3 : "transformers"
]
```

Predicted

```
[
  0 : "attention"
  1 : "interpretability"
  2 : "natural-language-processing"
  3 : "transformers"
]
```

→ let's go to the code to identify these subsets!

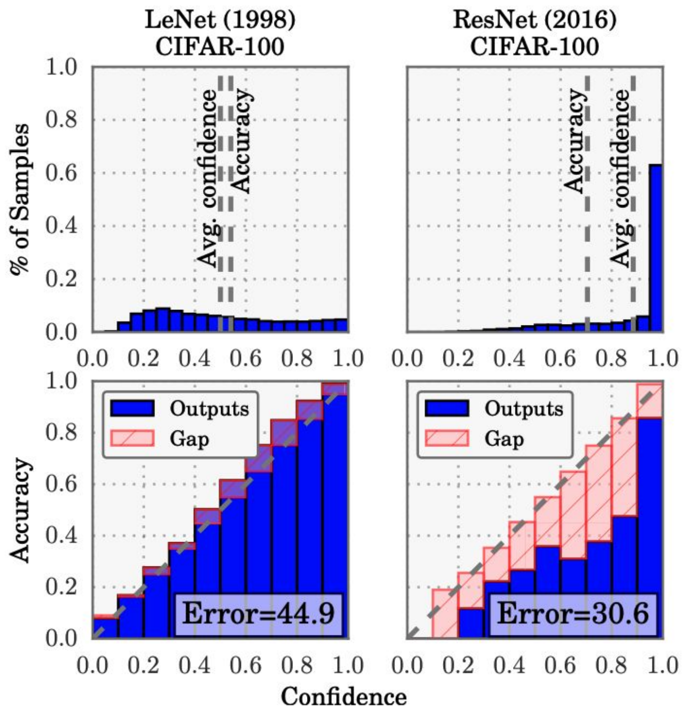
Confidence learning

- Inspect probabilities instead of predicted labels
- Categorical
    - prediction is incorrect (also indicate TN, FP, FN)
    - confidence score for the correct class is below a threshold
    - confidence score for an incorrect class is above a threshold
    - standard deviation of confidence scores over top N samples is low
    - different predictions from same model using different/previous parameters
- Continuous
    - difference between predicted and ground-truth values is above some %

```
# Confidence score for the incorrect class is
above a threshold
high_confidence = []
max_threshold = 0.2
for i in range(len(y_test)):
    indices = np.where(y_test[i]==0)[0]
    probs = y_prob[i][indices]
    classes = []
    for index in
np.where(probs>=max_threshold)[0]:

classes.append(label_encoder.index_to_class[i
ndices[index]])
    if len(classes):
        high_confidence.append({"text":
test_df.text[i], "classes": classes})
```

Calibration

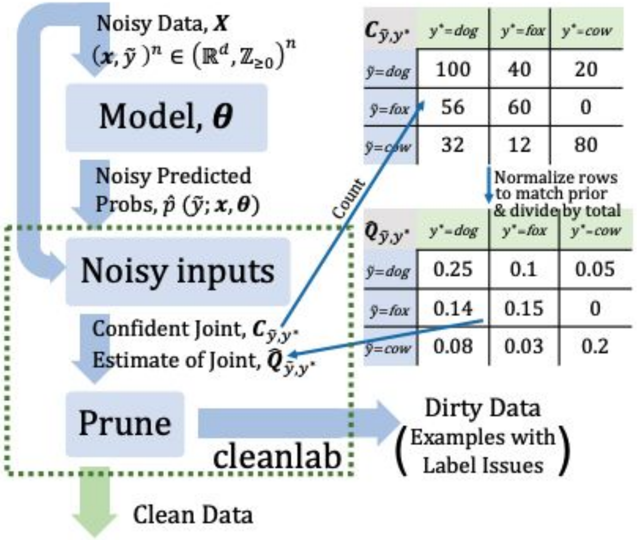
- **Assumption**: *"the probability associated with the predicted class label should reflect its ground truth correctness likelihood."*
- **Reality**: *"modern [large] neural networks are no longer well-calibrated"*
- **Solution**: apply temperature scaling (extension of [Platt scaling]) on model outputs



[On Calibration of Modern Neural Networks] (Weinberger et al., 2017)

Confident learning (CL)

- Learn calibrated joint distribution (cleanlab) between noisy & true labels to identify mislabeled samples



I use specific functions from the package since I already have my noisy labels and their predicted probabilities (view code).

```
import cleanlab
from cleanlab.util import onehot2int
from cleanlab.pruning import get_noise_indices

# Format our noisy labels `s` (cleanlab expects list of
integers for multilabel tasks)
correctly_formatted_labels = onehot2int(y_test)

# Determine potential labeling errors
label_error_indices = get_noise_indices(
            s=correctly_formatted_labels,
            psx=y_prob,
            multi_label=True,
            sorted_index_method="self_confidence",
            verbose=0)
```


Confident Learning: Estimating Uncertainty in Dataset Labels (Chuang et al., 2019)

Manual slices

- Besides fine-grained class metrics, there may be key slices (subsets) of our data that we'll want to evaluate.
  - Target / predicted classes (+ combinations)
  - Features (explicit and implicit)
  - Metadata (timestamps, sources, etc.)
  - Priority slices / experience (minority groups, large customers, etc.)

```python
from snorkel.slicing import PandasSFApplier
from snorkel.slicing import slice_dataframe
from snorkel.slicing import slicing_function

@slicing_function()
def cv_transformers(x):
    """Projects with the `computer-vision` & `transformers` tags."""
    return all(tag in x.tags for tag in ["computer-vision", "transformers"])

@slicing_function()
def short_text(x):
    """Projects with short titles and descriptions."""
    return len(x.text.split()) < 7  # less than 7 words
```

➜ let's go to the code to programmatically create and evaluate these slices!

Generated slices

- Can we auto identify relevant slices of data that are problematic?

Identify top-K slices that have at least T samples in each slice

```
def find_slice(self, k=50, epsilon=0.2, alpha=0.05,
degree=3, risk_control=True, max_workers=1)
```

Bin features with high cardinality

```
def binning(self, col, n_bin=20)
```

Generate slices that are not too big (low comparative loss) but also not too small (high loss, low interpretability)

```
slices = []
for col in X.columns:
    for v in np.unique(X[col]:
        data_idx = X[X[col] == v].index
        s = Slice({col:[[v]]}, data_idx)
        slices.append(s)
```

Merge smaller, insignificant slices together to create more meaningful slices.

```
def merge_slices(self, slices, reference, epsilon)
```

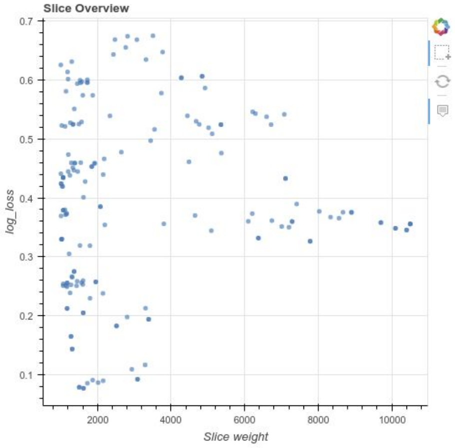
Using hypothesis testing for slice finding and reducing false discovery

```
def filter_by_significance(self, slices, reference,
alpha, max_workers=10)
```

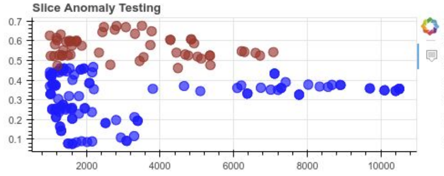
Automated Data Slicing for Model Validation (Whang et al., 2018)
https://github.com/yeounoh/slicefinder (Chung et al.)

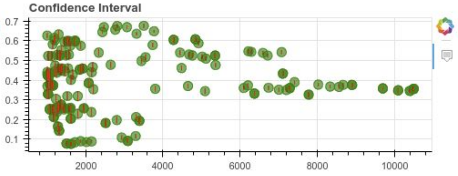
Generated slices

Received 175 rows from dremel.



#	Slice	Weight	Log Loss	Calibration	Avg. Label	Avg. Prediction
0	education_num_x_relationship...	3070	0.6686333558600267	0.9165924278926173	0.43680781758957654	0.4003747380469055
1	education_x_marital_status:B...	2768	0.6555037841686089	0.8267763341533528	0.6701589595375722	0.554071567866499
2	education_x_relationship:Bach...	2433	0.6436002667909259	0.8426198541828379	0.676531031648171	0.5700584792375467
3	age_x_education:[38.900000,...	1201	0.6017001429598788	0.8409174123580909	0.5403830141548709	0.4544174859453796
4	age_x_education:[31.600000,...	1145	0.5810290678740502	0.8482403478625676	0.4820960698689956	0.40893333800885356
5	education_num_x_occupation:...	1870	0.5742113018011095	0.7820396842281266	0.6155080213903743	0.4813516986880073
6	education_x_occupation:Bach...	1369	0.5511045693397273	0.8464060579265189	0.5690284879474069	0.48162915933145234
7	education_x_native_country:B...	4766	0.5249141257193968	0.872142941332496	0.42299622324800673	0.36891317031605375
8	education_x_education_num:...	5355	0.524386525034093	0.903732774192983	0.4147525676937442	0.37482548860553044
9	education:Bachelors	5355	0.524386525034093	0.903732774192983	0.4147525676937442	0.37482548860553044
10	education_num_x_occupation:...	1029	0.5229504503654301	0.9275708613679448	0.3654033041788144	0.338937457603836
11	capital_loss_x_education:[0.0...	5019	0.5191179623195775	0.9306165070928877	0.39748953974895396	0.36991032708713106
12	capital_gain_x_education:[0.0...	5117	0.5087792044857011	0.9548068873644355	0.38772718389681454	0.3702045856030956
13	education_x_hours_per_week:...	2648	0.4776779370519788	0.9959152150783132	0.35158610271903323	0.35014994910797187
14	education_x_marital_status:M...	1003	0.6258045912551992	0.7321253631466119	0.7716849451645065	0.5649701207133376
15	age_x_education_num:[38.90...	1724	0.6006635889958015	0.8044445826594409	0.5568445475638051	0.4479505796711503
16	education_x_native_country:...	1527	0.5997813410598055	0.7199830172054349	0.5671250818598559	0.4083204275703383
17	capital_loss_x_education:[0.0...	1577	0.5971973597450738	0.767394003346034	0.5364616360177552	0.41167744250522814
18	education:Masters	1723	0.5959963434245633	0.7493849892918957	0.5565873476494486	0.4170982035582867





Significant Slice Exploration Design Docs (Whang et. al. 2017)

Generated slices

- Using decision trees and lattice searching is +1 on top of clustering but still many limitations exist:
  - Sampling to find *any* k slices that satisfy significance reqs.
  - Can obscure slices with large errors
- SliceLine: pruning + enumeration + lin alg to find the **exact** top-K slices

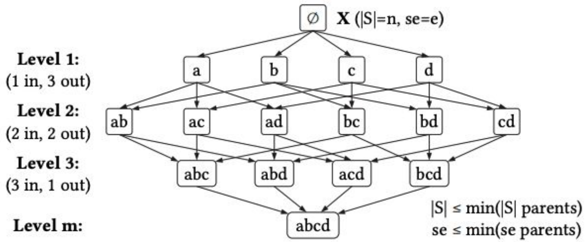


Figure 1: Example Lattice and Slice Properties.

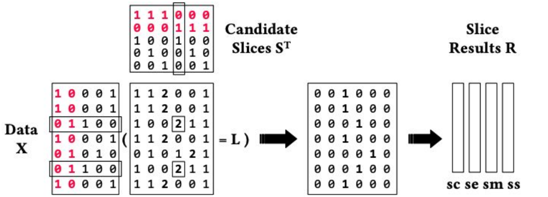


Figure 2: Example Vectorized Slice Evaluation (The matrix **X** has two red/black features, with 2/3 distinct values. The matrix multiplication ($(\mathbf{X} \odot \mathbf{S}^\top)$) evaluates predicates by multiplying the one-hot vectors and counting matching predicates. By checking for ($(\mathbf{X} \odot \mathbf{S}^\top) = L$), we get rows that match all L slice predicates).

SliceLine: Fast, Linear-Algebra-based Slice Finding for ML Model Debugging (Boehm et. al. 2021)

Generated slices

What if the features to generate slices on are implicit/hidden?

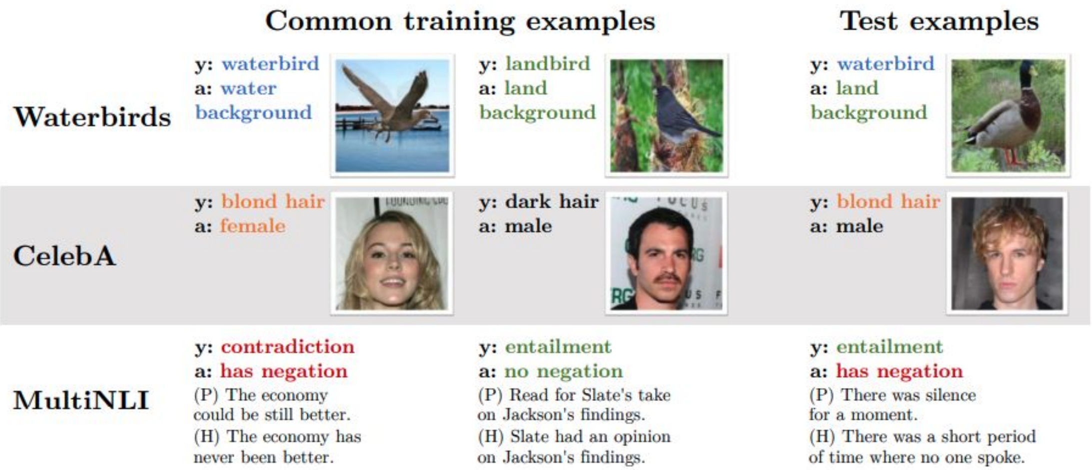


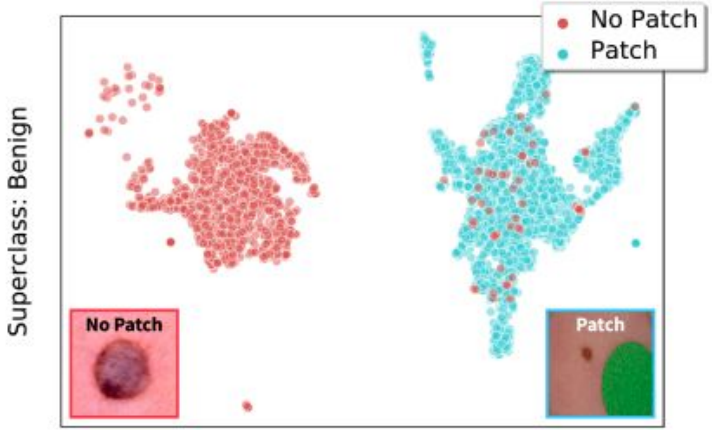
Figure 1: Representative training and test examples for the datasets we consider. The correlation between the label y and the spurious attribute a at training time does not hold at test time.

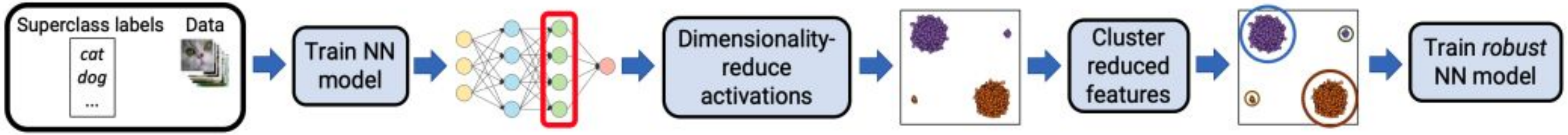
Distributionally Robust Neural Networks for Group Shifts (Liang et. al. 2019)

Generated slices

What if the features to generate slices on are implicit/hidden?

1. Estimate implicit subclass labels via unsupervised clustering
2. Train new more robust model using these clusters





[No Subclass Left Behind: Fine-Grained Robustness in Coarse-Grained Classification Problems]() (Re et. al. 2020)

Generated slices

Can we do better?

1. Learn subgroups
2. Learn transformations (ex. CycleGAN) needed to go from one subgroup to another under the same superclass (label)
3. Augment data with artificially introduced subgroup features
4. Train new robust model on augmented data

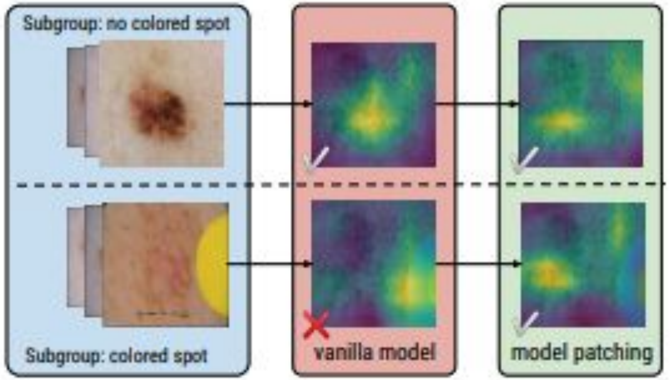


Figure 1: A vanilla model trained on a skin cancer dataset exhibits a subgroup performance gap between images of malignant cancers with and without colored bandages. GradCAM [70] illustrates that the vanilla model spuriously associates the colored spot with benign skin lesions. With model patching, the malignancy is predicted correctly for both subgroups.

Model Patching: Closing the Subgroup Performance Gap with Data Augmentation (Re et. al. 2020)

Evaluating evaluations (CI/CD suites)

- What criteria are most important?
- What criteria cannot regress?
- How much of a regression can be tolerated?
- Add criteria and programmatically enforce via [CI/CD workflows]()

```python
assert precision > prev_precision  # most important, cannot regress
assert recall >= best_prev_recall - 0.03  # recall cannot regress > 3%
assert metrics["class"]["data_augmentation"]["f1"] > prev_data_augmentation_f1  # class
assert metrics["slices"]["class"]["cv_transformers"]["f1"] > prev_cv_transformers_f1  # slice
```

Testing

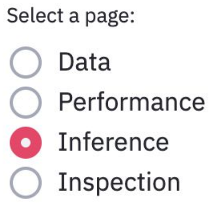
- Evaluation techniques may be model-specific but functional testing is model-agnostic. They should work regardless of model architectures or output attributes, etc.

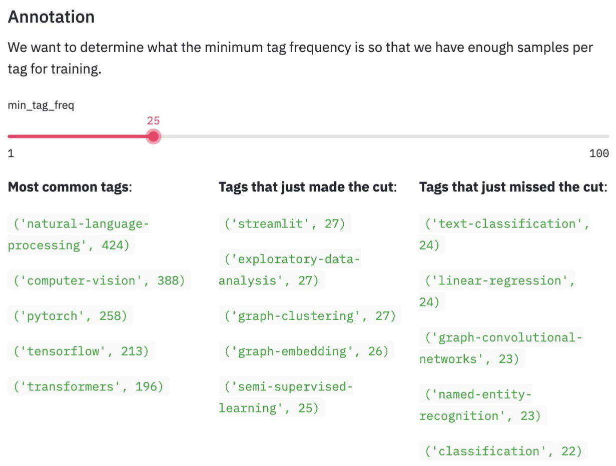
```python
# INVariance via verb injection (changes should not affect outputs)
tokens = ["revolutionized", "disrupted"]
tags = [["transformers"], ["transformers"]]
texts = [f"Transformers have {token} the ML field." for token in tokens]

# DIRectional expectations (changes with known outputs)
tokens = ["PyTorch", "Huggingface"]
tags = [["pytorch", "transformers"],["huggingface", "transformers"]]
texts = [f"A {token} implementation of transformers." for token in tokens]

# Minimum Functionality Tests (simple input/output pairs)
tokens = ["transformers", "graph neural networks"]
tags = [["transformers"], ["graph-neural-networks"]]
texts = [f"{token} have revolutionized machine learning." for token in tokens]
```

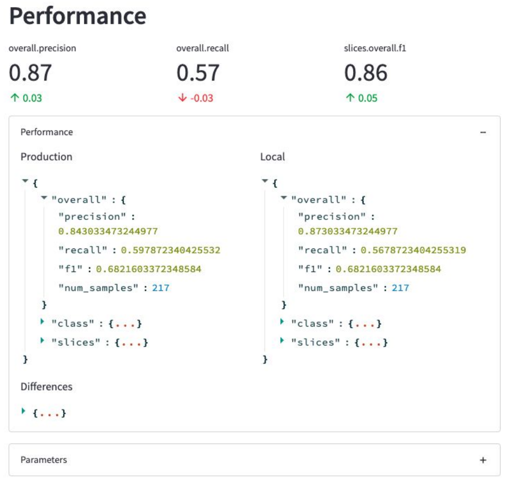
➔ view the [testing lesson](#) for more!

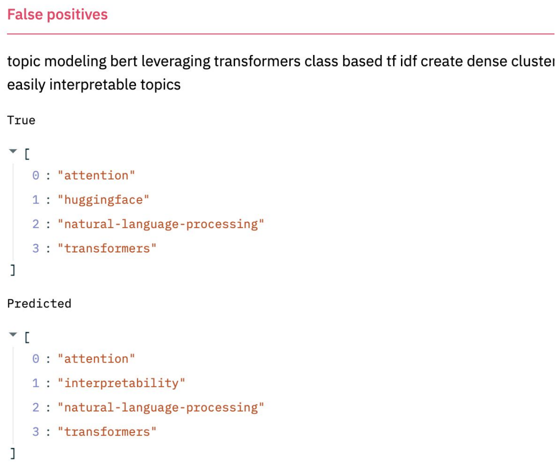
Dashboards / documentation

- Need to communicate evaluation findings with the broader team
  - Expose relevant views (ex. dashboard, model cards) for different personas
  - Should reflect reports respective to the currently deployed systems
  - Auto-generated (w/ templates) and deployed with CI/CD workflows









Model Cards: The value of a shared understanding of AI models (Google)
Metaflow Cards: Integrating Pythonic visual reports into ML pipelines (Outerbounds)

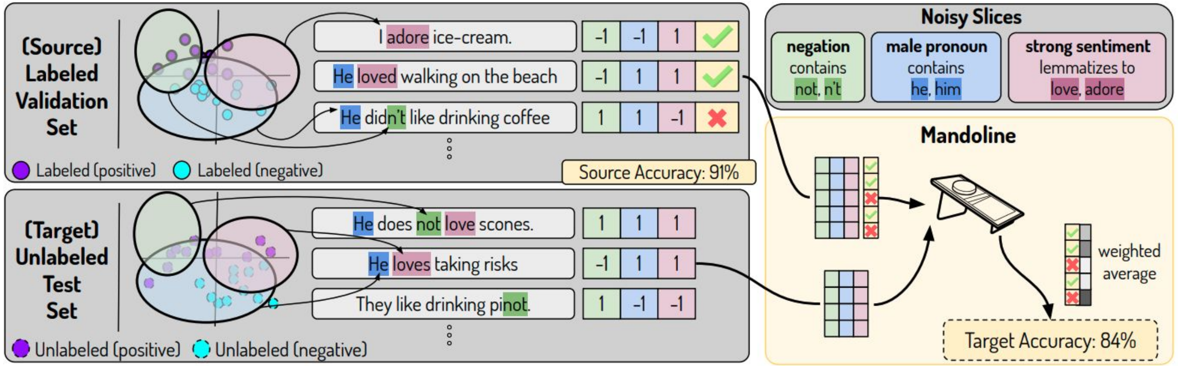
Monitoring

- Components from offline evaluation can be used for online setting but be wary of:
  - cumulative vs. sliding metrics
  - false positives due to data imbalances
- Check out the monitoring lesson for more info!
  - Performance measurements (w/ label lag)
  - Drift (data, target, etc.) location, measurement and mitigation

Monitoring

- What can we do if we want to monitor performance in the event of delayed outcomes?
  - Use approximate metrics as an estimate of performance
  - No reliable approximate metrics? → back to slicing!

1. Design slicing functions that may capture how our data may experience distribution shift (don't need complete coverage)
2. Develop slice matrices for source and target data
3. Compare matrices to approximate performance



Startup ideas

- Horizontal, generalized, low SME, lots of competition
  - Slice generator based on features, data modality, etc. (no code/low code)
  - Calibrated confidences to discover labeling errors (cleanlab)
  - Evaluation template for various tasks and data modalities given inputs, model, logits, labels, predictions, etc.
  - Caution: MANY platforms are working on baking this into larger product
- Specialized, moderate/high SME, industry/task-specific
  - Evaluation suites for products in highly regulated spaces (ex. health, fintech, etc.). Work with regulation entities and incumbents to devise fair criteria and thresholds.
  - Controlled and interpretable data augmentation via automatic identification of subgroups and patching into data (the more specific the space, the better).

Just a few specific ideas around evaluation but there are many other aspects of the ML development lifecycle!

➔ Connect with me on Twitter and LinkedIn

Machine Learning Systems Design

Next lecture: Deployment