MANNING

Human-in-the-Loop Machine Learning

Active learning and annotation for human-centered AI

ROBERT (MUNRO) MONARCH

Quick reference guide for this book

Understand a model's uncertainty

Softmax Base/Temperature (3.1.2, A.1-A.2)

Least Confidence (3.2.1)

Margin of Confidence (3.2.2)

Ratio of Confidence (3.2.3)

Entropy (3.2.4)

Ensembles of Models (3.4.1)

Query by Committee & Dropouts (3.4.2)

Aleatoric & Epistemic Uncertainty (3.4.3)

Active Transfer Learning for Uncertainty (5.2)

Identify gaps in a model's knowledge

Model-based Outliers (4.2, 4.6.1)

Cluster-based Sampling (4.3, 4.6.2)

Representative Sampling (4.4, 4.6.3)

Real-world Diversity (4.5, 4.6.4)

Active Transfer Learning for Representative Sampling (5.3)

Create a complete active learning strategy

Combining Uncertainty Sampling and Diversity Sampling (5.1.1-6)

Expected Error Reduction (5.1.8)

Active Transfer Learning for Adaptive Sampling (5.4)

Active Learning for already-labeled data (6.6.1)

Data-filtering with rules (9.5.1)

Training data search (9.5.2)

Implement active learning with different machine learning architectures

Logistic Regression & MaxEnt (3.3.1)

Support Vector Machines (3.3.2)

Bayesian Models (3.3.3)

Decision Trees & Random Forests (3.3.4)

Diversity Sampling (4.6.1-4)

Recruit annotators

In-house experts (7.2)

Outsourced workers (7.3)

Crowdsourced workers (7.4)

End users (7.5.1)

Volunteers (7.5.2)

People playing games (7.5.3)

Manage annotation quality

Ground truth data (8.1.1)

Expected accuracy/agreement & adjusting for random chance (8.1.2, A.3.3)

Dataset reliability with Krippendorff's alpha (8.2.3)

Individual annotator agreement (8.2.5)

Per-label & per-demographic agreement (8.2.6)

Extending accuracy with agreement for real-world diversity (8.2.7)

Aggregating annotations (8.3.1-3, 9.2.1-2)

Eliciting annotator-reported confidences (8.3.4)

Calculating annotation uncertainty (8.3.5)

Quality control by expert review (8.4)

Multistep workflows and adjudication/review tasks (8.5)

Creating models to predict whether a single annotation is:

… correct (9.2.3)

… in agreement (9.2.4)

… from a bot (9.2.5)

Trust model predictions as labels (9.3.1)

Use a model prediction as an annotation (9.3.2)

Cross-validating to find mislabeled data (9.3.3)

(Continued on inside back cover)

Human-in-the-Loop Machine Learning

ACTIVE LEARNING AND ANNOTATION FOR HUMAN-CENTERED AI

ROBERT (MUNRO) MONARCH
FOREWORD BY CHRISTOPHER D. MANNING

MANNING
SHELTER ISLAND

For online information and ordering of this and other Manning books, please visit www.manning.com. The publisher offers discounts on this book when ordered in quantity. For more information, please contact

Special Sales Department
Manning Publications Co.
20 Baldwin Road
PO Box 761
Shelter Island, NY 11964
Email: orders@manning.com

The figure on the cover is "Cephaloniene" (Woman from Cephalonia), an illustration by Jacques Grasset de Saint-Sauveur from his 1797 book, *Costumes de Différents Pays*.

Manning Publications Co.
20 Baldwin Road
PO Box 761
Shelter Island, NY 11964

Development editor: Susan Ethridge
Technical development editor: Frances Buontempo
Review editor: Ivan Martinović
Production editor: Deirdre S. Hiam
Copy editor: Keir Simpson
Proofreader: Keri Hales
Technical proofreader: Al Krinker
Typesetter: Gordan Salinovic
Cover designer: Marija Tudor

ISBN 9781617296741
Printed in the United States of America

brief contents

PART 1 FIRST STEPS 1

1 ■ Introduction to human-in-the-loop machine learning 3
2 ■ Getting started with human-in-the-loop machine learning 23

PART 2 ACTIVE LEARNING 49

3 ■ Uncertainty sampling 51
4 ■ Diversity sampling 84
5 ■ Advanced active learning 124
6 ■ Applying active learning to different machine learning tasks 156

PART 3 ANNOTATION 189

7 ■ Working with the people annotating your data 191
8 ■ Quality control for data annotation 219
9 ■ Advanced data annotation and augmentation 252
10 ■ Annotation quality for different machine learning tasks 290

PART 4 HUMAN–COMPUTER INTERACTION FOR MACHINE LEARNING ..323

11 ■ Interfaces for data annotation 325

12 ■ Human-in-the-loop machine learning products 354

contents

foreword xv
preface xvii
acknowledgments xix
about this book xxi
about the author xxiv

PART 1 FIRST STEPS 1

1 *Introduction to human-in-the-loop machine learning* 3

1.1 The basic principles of human-in-the-loop machine learning 4

1.2 Introducing annotation 5

Simple and more complicated annotation strategies 5 ▪ *Plugging the gap in data science knowledge* 6 ▪ *Quality human annotation: Why is it hard?* 6

1.3 Introducing active learning: Improving the speed and reducing the cost of training data 8

Three broad active learning sampling strategies: Uncertainty, diversity, and random 8 ▪ *What is a random selection of evaluation data?* 11 ▪ *When to use active learning* 12

1.4 Machine learning and human–computer interaction 14

User interfaces: How do you create training data? 14 ▪ Priming: What can influence human perception? 15 ▪ The pros and cons of creating labels by evaluating machine learning predictions 16 Basic principles for designing annotation interfaces 16

1.5 Machine-learning-assisted humans vs. human-assisted machine learning 16

1.6 Transfer learning to kick-start your models 17

Transfer learning in computer vision 18 ▪ Transfer learning in NLP 19

1.7 What to expect in this text 21

2 *Getting started with human-in-the-loop machine learning 23*

2.1 Beyond hacktive learning: Your first active learning algorithm 24

2.2 The architecture of your first system 25

2.3 Interpreting model predictions and data to support active learning 29

Confidence ranking 30 ▪ Identifying outliers 30 ▪ What to expect as you iterate 33

2.4 Building an interface to get human labels 34

A simple interface for labeling text 35 ▪ Managing machine learning data 37

2.5 Deploying your first human-in-the-loop machine learning system 38

Always get your evaluation data first 40 ▪ Every data point gets a chance 43 ▪ Select the right strategies for your data 44 ▪ Retrain the model and iterate 46

PART 2 ACTIVE LEARNING 49

3 *Uncertainty sampling 51*

3.1 Interpreting uncertainty in a machine learning model 52

Why look for uncertainty in your model? 53 ▪ Softmax and probability distributions 54 ▪ Interpreting the success of active learning 56

3.2 Algorithms for uncertainty sampling 56

Least confidence sampling 57 ▪ Margin of confidence sampling 59 ▪ Ratio sampling 61 ▪ Entropy (classification entropy) 62 ▪ A deep dive on entropy 65

3.3 Identifying when different types of models are confused 65

Uncertainty sampling with logistic regression and MaxEnt models 67 ▪ Uncertainty sampling with SVMs 67 Uncertainty sampling with Bayesian models 68 ▪ Uncertainty sampling with decision trees and random forests 69

3.4 Measuring uncertainty across multiple predictions 69

Uncertainty sampling with ensemble models 70 ▪ Query by Committee and dropouts 71 ▪ The difference between aleatoric and epistemic uncertainty 73 ▪ Multilabeled and continuous value classification 74

3.5 Selecting the right number of items for human review 74

Budget-constrained uncertainty sampling 75 ▪ Time-constrained uncertainty sampling 76 ▪ When do I stop if I'm not time- or budget-constrained? 77

3.6 Evaluating the success of active learning 77

Do I need new test data? 77 ▪ Do I need new validation data? 78

3.7 Uncertainty sampling cheat sheet 79

3.8 Further reading 82

Further reading for least confidence sampling 82 ▪ Further reading for margin of confidence sampling 82 ▪ Further reading for ratio of confidence sampling 82 ▪ Further reading for entropy-based sampling 82 ▪ Further reading for other machine learning models 82 Further reading for ensemble-based uncertainty sampling 82

4 *Diversity sampling 84*

4.1 Knowing what you don't know: Identifying gaps in your model's knowledge 85

Example data for diversity sampling 88 ▪ Interpreting neural models for diversity sampling 88 ▪ Getting information from hidden layers in PyTorch 90

4.2 Model-based outlier sampling 93

Use validation data to rank activations 93 ▪ Which layers should I use to calculate model-based outliers? 98 ▪ The limitations of model-based outliers 99

4.3 Cluster-based sampling 99

Cluster members, centroids, and outliers 100 ▪ Any clustering algorithm in the universe 101 ▪ K-means clustering with cosine similarity 103 ▪ Reduced feature dimensions via embeddings or PCA 105 ▪ Other clustering algorithms 107

4.4 Representative sampling 108

Representative sampling is rarely used in isolation 109 ▪ Simple representative sampling 110 ▪ Adaptive representative sampling 112

4.5 Sampling for real-world diversity 113

Common problems in training data diversity 114 ▪ Stratified sampling to ensure diversity of demographics 115 ▪ Represented and representative: Which matters? 116 ▪ Per-demographic accuracy 117 ▪ Limitations of sampling for real-world diversity 118

4.6 Diversity sampling with different types of models 118

Model-based outliers with different types of models 118 Clustering with different types of models 119 ▪ Representative sampling with different types of models 119 ▪ Sampling for real-world diversity with different types of models 119

4.7 Diversity sampling cheat sheet 119

4.8 Further reading 121

Further reading for model-based outliers 121 ▪ Further reading for cluster-based sampling 121 ▪ Further reading for representative sampling 121 ▪ Further reading for sampling for real-world diversity 122

5 *Advanced active learning 124*

5.1 Combining uncertainty sampling and diversity sampling 124

Least confidence sampling with cluster-based sampling 125 Uncertainty sampling with model-based outliers 128 ▪ Uncertainty sampling with model-based outliers and clustering 129 Representative sampling cluster-based sampling 129 ▪ Sampling from the highest-entropy cluster 132 ▪ Other combinations of active learning strategies 135 ▪ Combining active learning scores 136 Expected error reduction sampling 136

5.2 Active transfer learning for uncertainty sampling 138

Making your model predict its own errors 139 ▪ Implementing active transfer learning 140 ▪ Active transfer learning with more layers 142 ▪ The pros and cons of active transfer learning 143

5.3 Applying active transfer learning to representative sampling 144

Making your model predict what it doesn't know 145 ▪ Active transfer learning for adaptive representative sampling 146 ▪ The pros and cons of active transfer learning for representative sampling 147

5.4 Active transfer learning for adaptive sampling 147

Making uncertainty sampling adaptive by predicting uncertainty 148 ▪ The pros and cons of ATLAS 151

5.5 Advanced active learning cheat sheets 151

5.6 Further reading for active transfer learning 154

6 *Applying active learning to different machine learning tasks 156*

6.1 Applying active learning to object detection 157

Accuracy for object detection: Label confidence and localization 158 Uncertainty sampling for label confidence and localization in object detection 160 ▪ Diversity sampling for label confidence and localization in object detection 162 ▪ Active transfer learning for object detection 165 ▪ Setting a low object detection threshold to avoid perpetuating bias 166 ▪ Creating training data samples for representative sampling that are similar to your predictions 167 Sampling for image-level diversity in object detection 168 Considering tighter masks when using polygons 168

6.2 Applying active learning to semantic segmentation 169

Accuracy for semantic segmentation 169 ▪ Uncertainty sampling for semantic segmentation 171 ▪ Diversity sampling for semantic segmentation 172 ▪ Active transfer learning for semantic segmentation 172 ▪ Sampling for image-level diversity in semantic segmentation 173

6.3 Applying active learning to sequence labeling 173

Accuracy for sequence labeling 174 ▪ Uncertainty sampling for sequence labeling 175 ▪ Diversity sampling for sequence labeling 176 ▪ Active transfer learning for sequence labeling 178 ▪ Stratified sampling by confidence and tokens 179 Create training data samples for representative sampling that are similar to your predictions 179 ▪ Full-sequence labeling 179 Sampling for document-level diversity in sequence labeling 180

6.4 Applying active learning to language generation 180

Calculating accuracy for language generation systems 181 Uncertainty sampling for language generation 181 ▪ Diversity sampling for language generation 182 ▪ Active transfer learning for language generation 183

6.5 Applying active learning to other machine learning tasks 183

Active learning for information retrieval 184 ▪ Active learning for video 185 ▪ Active learning for speech 186

6.6 Choosing the right number of items for human review 186

Active labeling for fully or partially annotated data 187 Combining machine learning with annotation 187

6.7 Further reading 188

PART 3 ANNOTATION 189

7 *Working with the people annotating your data* 191

7.1 Introduction to annotation 193

Three principles of good data annotation 193 ▪ *Annotating data and reviewing model predictions* 195 ▪ *Annotations from machine learning-assisted humans* 195

7.2 In-house experts 195

Salary for in-house workers 197 ▪ *Security for in-house workers* 197 ▪ *Ownership for in-house workers* 197 ▪ *Tip: Always run in-house annotation sessions* 198

7.3 Outsourced workers 201

Salary for outsourced workers 202 ▪ *Security for outsourced workers* 203 ▪ *Ownership for outsourced workers* 203 ▪ *Tip: Talk to your outsourced workers* 204

7.4 Crowdsourced workers 205

Salary for crowdsourced workers 206 ▪ *Security for crowdsourced workers* 207 ▪ *Ownership for crowdsourced workers* 208 *Tip: Create a path to secure work and career advancement* 209

7.5 Other workforces 209

End users 209 ▪ *Volunteers* 211 ▪ *People playing games* 212 *Model predictions as annotations* 212

7.6 Estimating the volume of annotation needed 214

The orders-of-magnitude equation for number of annotations needed 215 ▪ *Anticipate one to four weeks of annotation training and task refinement* 216 ▪ *Use your pilot annotations and accuracy goal to estimate cost* 217 ▪ *Combining types of workforces* 217

8 *Quality control for data annotation* 219

8.1 Comparing annotations with ground truth answers 220

Annotator agreement with ground truth data 223 ▪ *Which baseline should you use for expected accuracy?* 225

8.2 Interannotator agreement 226

Introduction to interannotator agreement 226 ▪ *Benefits from calculating interannotator agreement* 228 ▪ *Dataset-level agreement with Krippendorff's alpha* 230 ▪ *Calculating Krippendorff's alpha beyond labeling* 234 ▪ *Individual annotator agreement* 234 ▪ *Per-label and per-demographic agreement* 238 *Extending accuracy with agreement for real-world diversity* 239

8.3 Aggregating multiple annotations to create training data 239

Aggregating annotations when everyone agrees 239 ▪ The mathematical case for diverse annotators and low agreement 241 Aggregating annotations when annotators disagree 241 Annotator-reported confidences 243 ▪ Deciding which labels to trust: Annotation uncertainty 244

8.4 Quality control by expert review 246

Recruiting and training qualified people 247 ▪ Training people to become experts 248 ▪ Machine-learning-assisted experts 248

8.5 Multistep workflows and review tasks 249

8.6 Further reading 250

9 *Advanced data annotation and augmentation 252*

9.1 Annotation quality for subjective tasks 253

Requesting annotator expectations 255 ▪ Assessing viable labels for subjective tasks 256 ▪ Trusting an annotator to understand diverse responses 258 ▪ Bayesian Truth Serum for subjective judgments 260 ▪ Embedding simple tasks in more complicated ones 262

9.2 Machine learning for annotation quality control 263

Calculating annotation confidence as an optimization task 263 Converging on label confidence when annotators disagree 264 Predicting whether a single annotation is correct 267 ▪ Predicting whether a single annotation is in agreement 268 ▪ Predicting whether an annotator is a bot 268

9.3 Model predictions as annotations 268

Trusting annotations from confident model predictions 269 Treating model predictions as a single annotator 271 ▪ Cross-validating to find mislabeled data 272

9.4 Embeddings and contextual representations 273

Transfer learning from an existing model 275 ▪ Representations from adjacent easy-to-annotate tasks 276 ▪ Self-supervision: Using inherent labels in the data 276

9.5 Search-based and rule-based systems 278

Data filtering with rules 279 ▪ Training data search 279 Masked feature filtering 281

9.6 Light supervision on unsupervised models 281

Adapting an unsupervised model to a supervised model 282 Human-guided exploratory data analysis 282

9.7 Synthetic data, data creation, and data augmentation 282

Synthetic data 282 ▪ Data creation 283 ▪ Data augmentation 284

9.8 Incorporating annotation information into machine learning models 285

Filtering or weighting items by confidence in their labels 285 Including the annotator identity in inputs 285 ▪ Incorporating uncertainty into the loss function 286

9.9 Further reading for advanced annotation 287

Further reading for subjective data 287 ▪ Further reading for machine learning for annotation quality control 287 ▪ Further reading for embeddings/contextual representations 288 ▪ Further reading for rule-based systems 288 ▪ Further reading for incorporating uncertainty in annotations into the downstream models 288

10 *Annotation quality for different machine learning tasks 290*

10.1 Annotation quality for continuous tasks 291

Ground truth for continuous tasks 291 ▪ Agreement for continuous tasks 292 ▪ Subjectivity in continuous tasks 292 Aggregating continuous judgments to create training data 293 Machine learning for aggregating continuous tasks to create training data 295

10.2 Annotation quality for object detection 296

Ground truth for object detection 297 ▪ Agreement for object detection 299 ▪ Dimensionality and accuracy in object detection 300 ▪ Subjectivity for object detection 301 Aggregating object annotations to create training data 301 Machine learning for object annotations 302

10.3 Annotation quality for semantic segmentation 304

Ground truth for semantic segmentation annotation 304 Agreement for semantic segmentation 305 ▪ Subjectivity for semantic segmentation annotations 305 ▪ Aggregating semantic segmentation to create training data 305 ▪ Machine learning for aggregating semantic segmentation tasks to create training data 307

10.4 Annotation quality for sequence labeling 307

Ground truth for sequence labeling 308 ▪ Ground truth for sequence labeling in truly continuous data 309 ▪ Agreement for sequence labeling 310 ▪ Machine learning and transfer learning for sequence labeling 310 ▪ Rule-based, search-based, and synthetic data for sequence labeling 312

10.5 Annotation quality for language generation 312

Ground truth for language generation 313 ▪ Agreement and aggregation for language generation 314 ▪ Machine learning and transfer learning for language generation 314 ▪ Synthetic data for language generation 314

10.6 Annotation quality for other machine learning tasks 315

Annotation for information retrieval 316 ▪ Annotation for multifield tasks 318 ▪ Annotation for video 318 ▪ Annotation for audio data 319

10.7 Further reading for annotation quality for different machine learning tasks 320

Further reading for computer vision 320 ▪ Further reading for annotation for natural language processing 321 ▪ Further reading for annotation for information retrieval 321

PART 4 HUMAN–COMPUTER INTERACTION FOR MACHINE LEARNING 323

11 *Interfaces for data annotation 325*

11.1 Basic principles of human–computer interaction 326

Introducing affordance, feedback, and agency 326 ▪ Designing interfaces for annotation 328 ▪ Minimizing eye movement and scrolling 329 ▪ Keyboard shortcuts and input devices 331

11.2 Breaking the rules effectively 333

Scrolling for batch annotation 333 ▪ Foot pedals 333 ▪ Audio inputs 334

11.3 Priming in annotation interfaces 334

Repetition priming 334 ▪ Where priming hurts 335 ▪ Where priming helps 336

11.4 Combining human and machine intelligence 336

Annotator feedback 336 ▪ Maximizing objectivity by asking what other people would annotate 337 ▪ Recasting continuous problems as ranking problems 338

11.5 Smart interfaces for maximizing human intelligence 340

Smart interfaces for semantic segmentation 341 ▪ Smart interfaces for object detection 343 ▪ Smart interfaces for language generation 345 ▪ Smart interfaces for sequence labeling 347

11.6 Machine learning to assist human processes 349

The perception of increased efficiency 349 ▪ Active learning for increased efficiency 350 ▪ Errors can be better than absence to maximize completeness 350 ▪ Keep annotation interfaces separate from daily work interfaces 351

11.7 Further reading 352

12 *Human-in-the-loop machine learning products 354*

12.1 Defining products for human-in-the-loop machine learning applications 355

Start with the problem you are solving 355 ▪ Design systems to solve the problem 356 ▪ Connecting Python and HTML 357

12.2 Example 1: Exploratory data analysis for news headlines 359

Assumptions 359 ▪ Design and implementation 360 Potential extensions 361

12.3 Example 2: Collecting data about food safety events 362

Assumptions 363 ▪ Design and implementation 363 Potential extensions 364

12.4 Example 3: Identifying bicycles in images 366

Assumptions 366 ▪ Design and implementation 367 Potential extensions 367

12.5 Further reading for building human-in-the-loop machine learning products 369

appendix Machine learning refresher 371

index 387

foreword

With machine learning now deployed widely in many industry sectors, artificial intelligence systems are in daily contact with human systems and human beings. Most people have noticed some of the user-facing consequences. Machine learning can either improve people's lives, such as with the speech recognition and natural language understanding of a helpful voice assistant, or it can annoy or even actively harm humans, with examples ranging from annoyingly lingering product recommendations to résumé review systems that are systematically biased against women or under-represented ethnic groups. Rather than thinking about artificial intelligence operating in isolation, the pressing need this century is for the exploration of human-centered artificial intelligence—that is, building AI technology that effectively cooperates and collaborates with people, and augments their abilities.

This book focuses not on end users but on how people and machine learning come together in the production and running of machine learning systems. It is an open secret of machine learning practitioners in industry that obtaining the right data with the right annotations is many times more valuable than adopting a more advanced machine learning algorithm. The production, selection, and annotation of data is a very human endeavor. Hand-labeling data can be expensive and unreliable, and this book spends much time on this problem. One direction is to reduce the amount of data that needs to be labeled while still allowing the training of high-quality systems through active learning approaches. Another direction is to exploit machine learning and human–computer interaction techniques to improve the speed and accuracy of human annotation. Things do not stop there: most large, deployed systems also involve

various kinds of human review and updating. Again, the machine learning can either be designed to leverage the work of people, or it can be something that humans need to fight against.

Robert Monarch is a highly qualified guide on this journey. In his work both before and during his PhD, Robert's focus was practical and attentive to people. He pioneered the application of natural language processing (NLP) to disaster-response-related messages based on his own efforts helping in several crisis scenarios. He started with human approaches to processing critical data and then looked for the best ways to leverage NLP to automate some of the process. I am delighted that many of these methods are now being used by disaster response organizations and can be shared with a broader audience in this book.

While the data side of machine learning is often perceived as mainly work managing people, this book shows that this side is also very technical. The algorithms for sampling data and quality control for annotation often approach the complexity of those in the downstream model consuming the training data, in some cases implementing machine learning and transfer learning techniques within the annotation process. There is a real need for more resources on the annotation process, and this book was already having an impact even as it was being written. As individual chapters were published, they were being read by data scientists in large organizations in fields like agriculture, entertainment, and travel. This highlights both the now-widespread use of machine learning and the thirst for data-focused books. This book codifies many of the best current practices and algorithms, but because the data side of the house was long neglected, I expect that there are still more scientific discoveries about data-focused machine learning to be made, and I hope that having an initial guidebook will encourage further progress.

—Christopher D. Manning

Christopher D. Manning is a professor of computer science and linguistics at Stanford University, director of the Stanford Artificial Intelligence Laboratory, and co-director of the Stanford Human-Centered Artificial Intelligence Institute.

preface

I am donating all author proceeds from this book to initiatives for better datasets, especially for low-resource languages and for health and disaster response. When I started writing this book, the example dataset about disaster response was uncommon and specific to my dual background as a machine learning scientist and disaster responder. With COVID-19, the global landscape has changed, and many people now understand why disaster response use cases are so important. The pandemic has exposed many gaps in our machine learning capabilities, especially with regard to access to relevant health care information and to fight misinformation campaigns. When search engines failed to surface the most up-to-date public health information and social media platforms failed to identify widespread misinformation, we all experienced the downside of applications that were not able to adapt fast enough to changing data.

This book is not specific to disaster response. The observations and methods that I share here also come from my experience building datasets for autonomous vehicles, music recommendations, online commerce, voice-enabled devices, translation, and a wide range of other practical use cases. It was a delight to learn about many new applications while writing the book. From data scientists who read draft chapters, I learned about use cases in organizations that weren't historically associated with machine learning: an agriculture company installing smart cameras on tractors, an entertainment company adapting face recognition to cartoon characters, an environmental company predicting carbon footprints, and a clothing company personalizing fashion

recommendations. When I gave invited talks about the book in these data science labs, I'm certain that I learned more than I taught!

All these use cases had two things in common: the data scientists needed to create better training and evaluation data for their machine learning models, and almost nothing was published about how to create that data. I'm excited to share strategies and techniques to help systems that combine human and machine intelligence for almost any application of machine learning.

acknowledgments

I owe the most gratitude to my wife, Victoria Monarch, for supporting my decision to write a book in the first place. I hope that this book helps make the world better for our own little human who was born while I was writing the book.

Most people who have written technical books told me that they stopped enjoying the process by the end. That didn't happen to me. I enjoyed writing this book right up until the final revisions because of all the people who had provided feedback on draft chapters since 2019. I appreciate how intrinsic early feedback is to the Manning Publications process, and within Manning Publications, I am most grateful to my editor, Susan Ethridge. I looked forward to our weekly calls, and I am especially fortunate to have had an editor who previously worked as a "human-in-the-loop" in e-discovery. Not every writer is fortunate to have an editor with domain experience! I am also grateful for the detailed chapter reviews by Frances Buontempo; the technical review by Al Krinker; project editor, Deirdre Hiam; copyeditor, Keir Simpson; proofreader, Keri Hales; review editor, Ivan Martinović; and everyone else within Manning who provided feedback on the book's content, images, and code.

Thank you to all the reviewers: Alain Couniot, Alessandro Puzielli, Arnaldo Gabriel Ayala Meyer, Clemens Baader, Dana Robinson, Danny Scott, Des Horsley, Diego Poggioli, Emily Ricotta, Ewelina Sowka, Imaculate Mosha, Michal Rutka, Michiel Trimpe, Rajesh Kumar R S, Ruslan Shevchenko, Sayak Paul, Sebastián Palma Mardones, Tobias Bürger, Torje Lucian, V. V. Phansalkar, and Vidhya Vinay. Your suggestions helped make this book better.

Thank you to everyone in my network who gave me direct feedback on early drafts: Abhay Agarwa, Abraham Starosta, Aditya Arun, Brad Klingerberg, David Evans, Debajyoti Datta, Divya Kulkarni, Drazen Prelec, Elijah Rippeth, Emma Bassein, Frankie Li, Jim Ostrowski, Katerina Margatina, Miquel Àngel Farré, Rob Morris, Scott Cambo, Tivadar Danka, Yada Pruksachatkun, and everyone who commented via Manning's online forum. Adrian Calma was especially diligent, and I am lucky that a recent PhD in active learning read the draft chapters so closely!

I am indebted to many people I have worked with over the course of my career. In addition to my colleagues at Apple today, I am especially grateful to past colleagues at Idibon, Figure Eight, AWS, and Stanford. I am delighted that my PhD advisor at Stanford, Christopher Manning, provided the foreword for this book.

Finally, I am especially grateful to the 11 experts who shared anecdotes in this book: Ayanna Howard, Daniela Braga, Elena Grewal, Ines Montani, Jennifer Prendki, Jia Li, Kieran Snyder, Lisa Braden-Harder, Matthew Honnibal, Peter Skomoroch, and Radha Basu. All of them have founded successful machine learning companies, and all worked directly on the data side of machine learning at some point in their careers. If you are like most intended readers of this book—someone early in their career who is struggling to create good training data—consider them to be role models for your own future!

about this book

This is the book that I *wish* existed when I was introduced to machine learning, because it addresses the most important problem in artificial intelligence: how should humans and machines work together to solve problems? Most machine learning models are guided by human examples, but most machine learning texts and courses focus only on the algorithms. You can often get state-of-the-art results with good data and simple algorithms, but you rarely get state-of-the-art results with the best algorithm built on bad data. So if you need to go deep in one area of machine learning first, you could argue that the data side is more important.

Who should read this book

This book is primarily for data scientists, software developers, and students who have only recently started working with machine learning (or only recently started working on the data side). You should have some experience with concepts such as supervised and unsupervised machine learning, training and testing machine learning models, and libraries such as PyTorch and TensorFlow. But you don't have be an expert in any of these areas to start reading this book.

When you become more experienced, this book should remain a useful quick reference for the different techniques. This book is the first to contain the most common strategies for annotation, active learning, and adjacent tasks such as interface design for annotation.

How this book is organized: A road map

This book is divided into four parts: an introduction; a deep dive on active learning; a deep dive on annotation; and the final part, which brings everything together with design strategies for human interfaces and three implementation examples.

The first part of this book introduces the building blocks for creating training and evaluation data: annotation, active learning, and the human–computer interaction concepts that help humans and machines combine their intelligence most effectively. By the end of chapter 2, you will have built a human-in-the-loop machine learning application for labeling news headlines, completing the cycle from annotating new data to retraining a model and then using the new model to help decide which data should be annotated next.

Part 2 covers active learning—the set of techniques for sampling the most important data for humans to review. Chapter 3 covers the most widely used techniques for understanding a model's uncertainty, and chapter 4 tackles the complicated problem of identifying where your model might be confident but wrong due to undersampled or nonrepresentative data. Chapter 5 introduces ways to combine different strategies into a comprehensive active learning system, and chapter 6 covers how the active learning techniques can be applied to different kinds of machine learning tasks.

Part 3 covers annotation—the often-underestimated problem of obtaining accurate and representative labels for training and evaluation data. Chapter 7 covers how to find and manage the right people to annotate data. Chapter 8 covers the basics of quality control for annotation, introducing the most common ways to calculate accuracy and agreement. Chapter 9 covers advanced strategies for annotation quality control, including annotations for subjective tasks and a wide range of methods to semi-automate annotation with rule-based systems, search-based systems, transfer learning, semi-supervised learning, self-supervised learning, and synthetic data creation. Chapter 10 covers how annotation can be managed for different kinds of machine learning tasks.

Part 4 completes the "loop" with a deep dive on interfaces for effective annotation in chapter 11 and three examples of human-in-the-loop machine learning applications in chapter 12.

Throughout the book, we continually return to examples from different kinds of machine learning tasks: image- and document-level labeling, continuous data, object detection, semantic segmentation, sequence labeling, language generation, and information retrieval. The inside covers contain quick references that show where you can find these tasks throughout the book.

About the code

All the code used in this book is open source and available from my GitHub account. The code used in the first six chapters of this book is at https://github.com/rmunro/pytorch_active_learning.

Some chapters also use spreadsheets for analysis, and the three examples in the final chapter are in their own repositories. See the respective chapters for more details.

liveBook discussion forum

Purchase of *Human-in-the-Loop Machine Learning* includes free access to a private web forum run by Manning Publications where you can make comments about the book, ask technical questions, and receive help from the author and from other users. To access the forum, go to https://livebook.manning.com/book/human-in-the-loop-machine-learning/welcome/v-11. You can learn more about Manning's forums and the rules of conduct at https://livebook.manning.com/#!/discussion.

Manning's commitment to our readers is to provide a venue where a meaningful dialogue between individual readers and between readers and the author can take place. It is not a commitment to any specific amount of participation on the part of the author, whose contribution to the forum remains voluntary (and unpaid). We suggest that you try asking the author some challenging questions lest his interest stray! The forum and the archives of previous discussions will be accessible from the publisher's website as long as the book is in print.

Other online resources

Each chapter has a "Further reading" section, and with only a handful of exceptions, all the resources listed are free and available online. As I say in a few places, look for highly cited work that cites the papers I referenced. It didn't make sense to include some influential papers, and many other relevant papers will be published after this book.

about the author

Robert Monarch, PhD (formerly Robert Munro), is an expert in combining human and machine intelligence who currently lives in San Francisco and works at Apple. Robert has worked in Sierra Leone, Haiti, the Amazon, London, and Sydney, in organizations ranging from startups to the United Nations. He was the CEO and founder of Idibon, the CTO of Figure Eight, and he led Amazon Web Services's first natural language processing and machine translation services.

Part 1

First steps

Most data scientists spend more time working on the data than on the algorithms. Most books and courses on machine learning, however, focus on the algorithms. This book addresses this gap in material about the data side of machine learning.

The first part of this book introduces the building blocks for creating training and evaluation data: annotation, active learning, and the human–computer interaction concepts that help humans and machines combine their intelligence most effectively. By the end of chapter 2, you will have built a human-in-the-loop machine learning application for labeling news headlines, completing the cycle from annotating new data to retraining a model and then using the new model to decide which data should be annotated next.

In the remaining chapters, you will learn how you might extend your first application with more sophisticated techniques for data sampling, annotation, and combining human and machine intelligence. The book also covers how to apply the techniques you will learn to different types of machine learning tasks, including object detection, semantic segmentation, sequence labeling, and language generation.

1 Introduction to human-in-the-loop machine learning

This chapter covers

- Annotating unlabeled data to create training, validation, and evaluation data
- Sampling the most important unlabeled data items (active learning)
- Incorporating human–computer interaction principles into annotation
- Implementing transfer learning to take advantage of information in existing models

Unlike robots in the movies, most of today's artificial intelligence (AI) cannot learn by itself; instead, it relies on intensive human feedback. Probably 90% of machine learning applications today are powered by supervised machine learning. This figure covers a wide range of use cases. An autonomous vehicle can drive you safely down the street because humans have spent thousands of hours telling it when its

sensors are seeing a pedestrian, moving vehicle, lane marking, or other relevant object. Your in-home device knows what to do when you say "Turn up the volume" because humans have spent thousands of hours telling it how to interpret different commands. And your machine translation service can translate between languages because it has been trained on thousands (or maybe millions) of human-translated texts.

Compared with the past, our intelligent devices are learning less from programmers who are hardcoding rules and more from examples and feedback given by humans who do not need to code. These human-encoded examples—the training data—are used to train machine learning models and make them more accurate for their given tasks. But programmers still need to create the software that allows the feedback from nontechnical humans, which raises one of the most important questions in technology today: *What are the right ways for humans and machine learning algorithms to interact to solve problems?* After reading this book, you will be able to answer this question for many uses that you might face in machine learning.

Annotation and active learning are the cornerstones of human-in-the-loop machine learning. They specify how you elicit training data from people and determine the right data to put in front of people when you don't have the budget or time for human feedback on all your data. Transfer learning allows us to avoid a cold start, adapting existing machine learning models to our new task rather than starting at square one. We will introduce each of these concepts in this chapter.

1.1 The basic principles of human-in-the-loop machine learning

Human-in-the-loop machine learning is a set of strategies for combining human and machine intelligence in applications that use AI. The goal typically is to do one or more of the following:

- Increase the accuracy of a machine learning model.
- Reach the target accuracy for a machine learning model faster.
- Combine human and machine intelligence to maximize accuracy.
- Assist human tasks with machine learning to increase efficiency.

This book covers the most common active learning and annotation strategies and how to design the best interface for your data, task, and annotation workforce. The book gradually builds from simpler to more complicated examples and is written to be read in sequence. You are unlikely to apply all these techniques at the same time, however, so the book is also designed to be a reference for each specific technique.

Figure 1.1 shows the human-in-the-loop machine learning process for adding labels to data. This process could be any labeling process: adding the topic to news stories, classifying sports photos according to the sport being played, identifying the sentiment of a social media comment, rating a video on how explicit the content is, and so on. In all cases, you could use machine learning to automate some of the process of labeling or to speed up the human process. In all cases, using best practices means implementing the cycle shown in figure 1.1: sampling the right data to label, using that data to train a model, and using that model to sample more data to annotate.

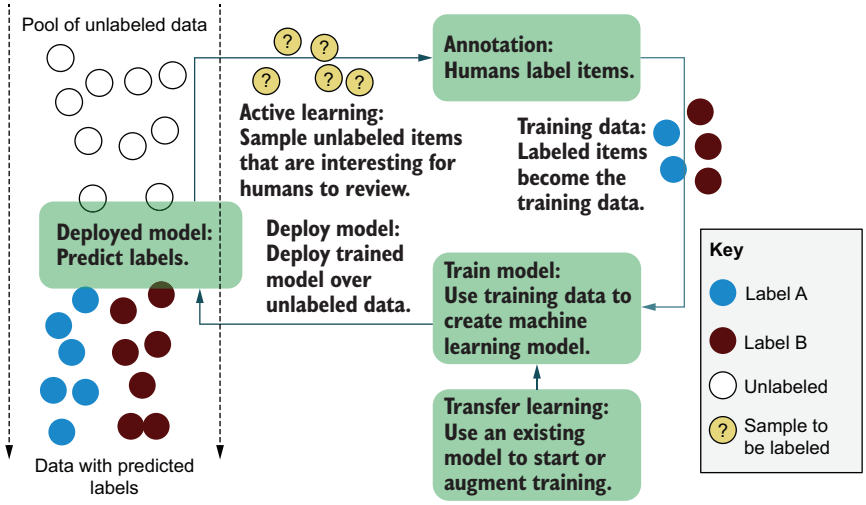


Figure 1.1 A mental model of the human-in-the-loop process for predicting labels on data

In some cases, you may want only some of the techniques. If you have a system that backs off to a human when the machine learning model is uncertain, for example, you would look at the relevant chapters and sections on uncertainty sampling, annotation quality, and interface design. Those topics still represent the majority of this book even if you aren't completing the "loop."

This book assumes that you have some familiarity with machine learning. Some concepts are especially important for human-in-the-loop systems, including deep understanding of softmax and its limitations. You also need to know how to calculate accuracy with metrics that take model confidence into consideration, calculate chance-adjusted accuracy, and measure the performance of machine learning from a human perspective. (The appendix contains a summary of this knowledge.)

1.2 Introducing annotation

Annotation is the process of labeling raw data so that it becomes training data for machine learning. Most data scientists will tell you that they spend much more time curating and annotating datasets than they spend building the machine learning models. Quality control for human annotation relies on more complicated statistics than most machine learning models do, so it is important to take the necessary time to learn how to create quality training data.

1.2.1 Simple and more complicated annotation strategies

An annotation process can be simple. If you want to label social media posts about a product as positive, negative, or neutral to analyze broad trends in sentiment about that product, for example, you could build and deploy an HTML form in a few hours. A simple HTML form could allow someone to rate each social media post according

to the sentiment option, and each rating would become the label on the social media post for your training data.

An annotation process can also be complicated. If you want to label every object in a video with a bounding box, for example, a simple HTML form is not enough; you need a graphical interface that allows annotators to draw those boxes, and a good user experience might take months of engineering hours to build.

1.2.2 Plugging the gap in data science knowledge

Your machine learning algorithm strategy and your data annotation strategy can be optimized at the same time. The two strategies are closely intertwined, and you often get better accuracy from your models faster if you have a combined approach. Algorithms and annotation are equally important components of good machine learning.

All computer science departments offer machine learning courses, but few offer courses on creating training data. At most, you might find one or two lectures about creating training data among hundreds of machine learning lectures across half a dozen courses. This situation is changing, but slowly. For historical reasons, academic machine learning researchers have tended to keep the datasets constant and evaluated their research only in terms of different algorithms.

By contrast with academic machine learning, it is more common in industry to improve model performance by annotating more training data. Especially when the nature of the data is changing over time (which is also common), using a handful of new annotations can be far more effective than trying to adapt an existing model to a new domain of data. But far more academic papers focus on how to adapt algorithms to new domains *without* new training data than on how to annotate the right new training data efficiently.

Because of this imbalance in academia, I've often seen people in industry make the same mistake. They hire a dozen smart PhDs who know how to build state-of-the-art algorithms but don't have experience creating training data or thinking about the right interfaces for annotation. I saw exactly this situation recently at one of the world's largest auto manufacturers. The company had hired a large number of recent machine learning graduates, but it couldn't operationalize its autonomous vehicle technology because the new employees couldn't scale their data annotation strategy. The company ended up letting that entire team go. During the aftermath, I advised the company how to rebuild its strategy by using algorithms and annotation as equally-important, intertwined components of good machine learning.

1.2.3 Quality human annotation: Why is it hard?

To those who study it, annotation is a science that's tied closely to machine learning. The most obvious example is that the humans who provide the labels can make errors, and overcoming these errors requires surprisingly sophisticated statistics.

Human errors in training data can be more or less important, depending on the use case. If a machine learning model is being used only to identify broad trends in

consumer sentiment, it probably won't matter whether errors propagate from 1% bad training data. But if an algorithm that powers an autonomous vehicle doesn't see 1% of pedestrians due to errors propagated from bad training data, the result will be disastrous. Some algorithms can handle a little noise in the training data, and random noise even helps some algorithms become more accurate by avoiding overfitting. But human errors tend not to be random noise; therefore, they tend to introduce irrecoverable bias into training data. No algorithm can survive truly bad training data.

For simple tasks, such as binary labels on objective tasks, the statistics are fairly straightforward for deciding which label is correct when different annotators disagree. But for subjective tasks, or even objective tasks with continuous data, no simple heuristics exist for deciding the correct label. Think about the critical task of creating training data by putting a bounding box around every pedestrian recognized by a self-driving car. What if two annotators have slightly different boxes? Which box is the correct one? The answer is not necessarily either box or the average of the two boxes. In fact, the best way to aggregate the two boxes is to use machine learning.

One of the best ways to ensure quality annotations is to ensure you have the right people making those annotations. Chapter 7 of this book is devoted to finding, teaching, and managing the best annotators. For an example of the importance of the right workforce combined with the right technology, see the following sidebar.

Human insights and scalable machine learning equal production AI

Expert anecdote by Radha Ramaswami Basu

The outcome of AI is heavily dependent on the quality of the training data that goes into it. A small UI improvement like a magic wand to select regions in an image can realize large efficiencies when applied across millions of data points in conjunction with well-defined processes for quality control. An advanced workforce is the key factor: training and specialization increase quality, and insights from an expert workforce can inform model design in conjunction with domain experts. The best models are created by a constructive, ongoing partnership between machine and human intelligence.

We recently took on a project that required pixel-level annotation of the various anatomic structures within a robotic coronary artery bypass graft (CABG) video. Our annotation teams are not experts in anatomy or physiology, so we implemented teaching sessions in clinical knowledge to augment the existing core skills in 3D spatial reasoning and precision annotation, led by a solutions architect who is a trained surgeon. The outcome for our customer was successful training and evaluation data. The outcome for us was to see people from under-resourced backgrounds in animated discussion about some of the most advanced uses of AI as they quickly became experts in one of the most important steps in medical image analysis.

Radha Basu is founder and CEO of iMerit. iMerit uses technology and an AI workforce consisting of 50% women and youth from underserved communities to create advanced technology workers for global clients. Radha previously worked at HP, took Supportsoft public as CEO, and founded the Frugal Innovation Lab at Santa Clara University.

1.3 Introducing active learning: Improving the speed and reducing the cost of training data

Supervised learning models almost always get more accurate with more labeled data. *Active learning* is the process of deciding which data to sample for human annotation. No one algorithm, architecture, or set of parameters makes one machine learning model more accurate in all cases, and no one strategy for active learning is optimal across all use cases and datasets. You should try certain approaches first, however, because they are more likely to be successful for your data and task.

Most research papers on active learning focus on the number of training items, but speed can be an even more important factor in many cases. In disaster response, for example, I have often deployed machine learning models to filter and extract information from emerging disasters. Any delay in disaster response is potentially critical, so getting a usable model out quickly is more important than the number of labels that need to go into that model.

1.3.1 Three broad active learning sampling strategies: Uncertainty, diversity, and random

Many active learning strategies exist, but three basic approaches work well in most contexts: uncertainty, diversity, and random sampling. A combination of the three should almost always be the starting point.

Random sampling sounds the simplest but can be the trickiest. What is random if your data is prefiltered, when your data is changing over time, or if you know for some other reason that a random sample will not be representative of the problem you are addressing? These questions are addressed in more detail in the following sections. Regardless of the strategy, you should always annotate some amount of random data to gauge the accuracy of your model and compare your active learning strategies with a baseline of randomly selected items.

Uncertainty and diversity sampling go by various names in the literature. They are often referred to as *exploitation* and *exploration*, which are clever names that alliterate and rhyme, but are not otherwise very transparent.

Uncertainty sampling is the set of strategies for identifying unlabeled items that are near a decision boundary in your current machine learning model. If you have a binary classification task, these items will have close to a 50% probability of belonging to either label; therefore, the model is called uncertain or confused. These items are most likely to be wrongly classified, so they are the most likely to result in a label that differs from the predicted label, moving the decision boundary after they have been added to the training data and the model has been retrained.

Diversity sampling is the set of strategies for identifying unlabeled items that are underrepresented or unknown to the machine learning model in its current state. The items may have features that are rare in the training data, or they might represent real-world demographics that are currently under-represented in the model. In either case, the result can be poor or uneven performance when the model is applied, especially when the data is changing over time. The goal of diversity sampling is to target new,

unusual, or underrepresented items for annotation to give the machine learning algorithm a more complete picture of the problem space.

Although the term *uncertainty sampling* is widely used, *diversity sampling* goes by different names in different fields, such as representative sampling, stratified sampling, outlier detection, and anomaly detection. For some use cases, such as identifying new phenomena in astronomical databases or detecting strange network activity for security, the goal of the task is to identify the outlier or anomaly, but we can adapt them here as a sampling strategy for active learning.

Uncertainty sampling and diversity sampling have shortcomings in isolation (figure 1.2). Uncertainty sampling might focus on one part of the decision boundary, for example, and diversity sampling might focus on outliers that are a long distance from the boundary. So the strategies are often used together to find a selection of unlabeled items that will maximize both uncertainty and diversity.

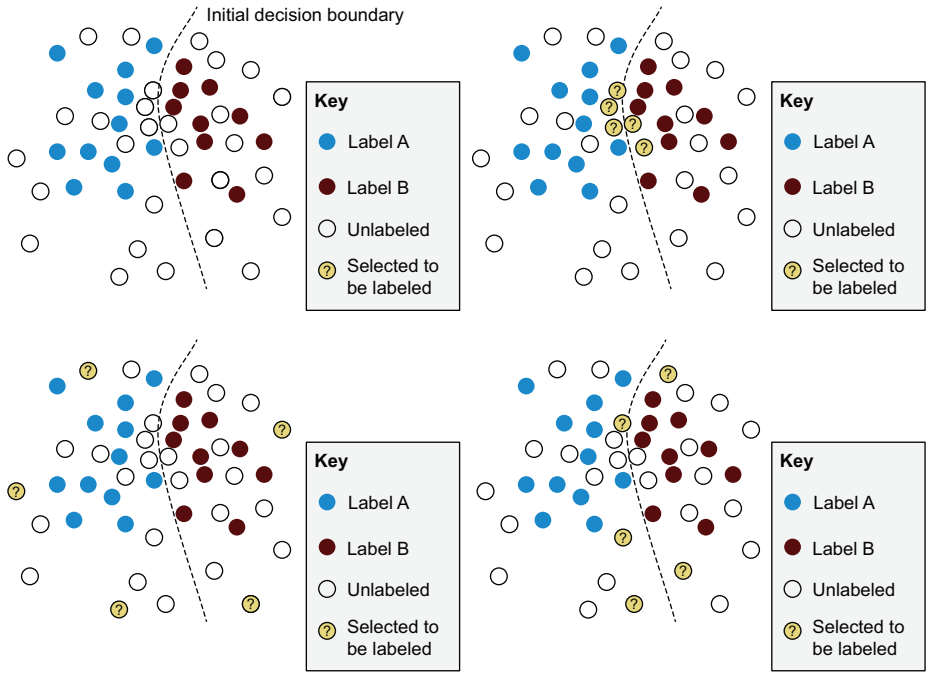


Figure 1.2 Pros and cons of different active learning strategies. *Top left:* The decision boundary from a machine learning algorithm between items, with some items labeled A and some labeled B. *Top right:* One possible result from uncertainty sampling. This active learning strategy is effective for selecting unlabeled items near the decision boundary. These items are the most likely to be wrongly predicted, and therefore, the most likely to get a label that moves the decision boundary. If all the uncertainty is in one part of the problem space, however, giving these items labels will not have a broad effect on the model. *Bottom left:* One possible result of diversity sampling. This active learning strategy is effective for selecting unlabeled items in different parts of the problem space. If the diversity is away from the decision boundary, however, these items are unlikely to be wrongly predicted, so they will not have a large effect on the model when a human gives them a label that is the same as the model predicted. *Bottom right:* One possible result from combining uncertainty sampling and diversity sampling. When the strategies are combined, items are selected that are near diverse sections of the decision boundary. Therefore, we are optimizing the chance of finding items that are likely to result in a changed decision boundary.

It is important to note that the active learning process is iterative. In each iteration of active learning, a selection of items is identified and receives a new human-generated label. Then the model is retrained with the new items, and the process is repeated. Figure 1.3 shows two iterations for selecting and annotating new items, resulting in a changing boundary.

Figure 1.3 The iterative active learning process. *Top left to bottom right:* Two iterations of active learning. In each iteration, items are selected along a diverse selection of the boundary, which in turn causes the boundary to move after retraining, resulting in a more accurate machine learning model. Ideally, we requested human labels for the minimum number of items as part of our active learning strategy. This request speeds the time to get an accurate model and reduces the overall cost of human annotation.

Iteration cycles can be a form of diversity sampling in themselves. Imagine that you used only uncertainty sampling, and sampled from only one part of the problem space in an iteration. You might solve all uncertainty in that part of the problem space; therefore, the next iteration would concentrate somewhere else. With enough iterations, you might not need diversity sampling at all. Each iteration from uncertainty sampling would focus on a different part of the problem space, and together, the iterations are enough to get a diverse sample of items for training.

Implemented properly, active learning has this self-correcting function: each iteration finds new aspects of the data that are best for human annotation. If some part of your data space is inherently ambiguous, however, each iteration could keep bringing you back to the same part of the problem space with those ambiguous items. So it is generally wise to consider both uncertainty and diversity sampling strategies to ensure that you are not focusing all your labeling efforts on a part of the problem space that your model might not be able to solve.

Figures 1.2 and 1.3 give you good intuition about the process for active learning. As anyone who has worked with high-dimensional or sequence data knows, it is not always straightforward to identify distance from a boundary or diversity. At least, the process is more complicated than the simple Euclidean distance in figures 1.2 and 1.3. But the same idea still applies: we are trying to reach an accurate model as quickly as possible with as few human labels as possible.

The number of iterations and the number of items that need to be labeled within each iteration depend on the task. When you're working in adaptive machine+human translation, a single translated sentence is enough training data to require the model to update, ideally within a few seconds. It is easy to see why from a user-experience perspective. If a human translator corrects the machine prediction for some word, but the machine doesn't adapt quickly, the human may need to (re)correct that machine output hundreds of times. This problem is common when you're translating words that are highly context-specific. You may want to translate a person's name literally in a news article, for example, but translate it into a localized name in a work of fiction. The user experience will be bad if the software keeps making the same mistake so soon after a human has corrected it, because we expect recency to help with adaptation.

On the technical side, of course, it is much more difficult to adapt a model quickly. Consider large machine translation models. Currently, it takes a week or more to train these models. From the experience of the translator, a software system that can adapt quickly is employing continuous learning. In most use cases I've worked on, such as identifying the sentiment in social media comments, I needed to iterate only every month or so to adapt to new data. Although few applications have real-time adaptive machine learning today, more are moving this way.

1.3.2 What is a random selection of evaluation data?

It is easy to *say* that you should always evaluate on a random sample of held-out data, but in practical terms, it is rarely easy to ensure that you have a truly random sample of data. If you prefiltered the data that you are working with by keyword, time, or some other factor, you already have a nonrepresentative sample. The accuracy of that sample is not necessarily indicative of the accuracy on the data where your model will be deployed.

I've seen people use the well-known ImageNet dataset and apply machine learning models to a broad selection of data. The canonical ImageNet dataset has 1,000 labels, each of which describes the category of that image, such as "Basketball," "Taxi," or

"Swimming." The ImageNet challenges evaluated held-out data from that dataset, and systems achieved near-human-level accuracy within that dataset. If you apply those same models to a random selection of images posted on a social media platform, however, accuracy immediately drops to something like 10%.

In most applications of machine learning, the data will change over time as well. If you're working with language data, the topics that people talk about will change over time, and the languages themselves will innovate and evolve. If you're working with computer vision data, the types of objects that you encounter will change over time. Equally important, the images themselves will change based on advances and changes in camera technology.

If you can't define a meaningful random set of evaluation data, you should try to define a *representative* evaluation dataset. If you define a representative dataset, you are admitting that a truly random sample isn't possible or meaningful for your dataset. It is up to you to define what is representative for your use case, based on how you are applying the data. You may want to select data points for every label that you care about, a certain number from every time period or a certain number from the output of a clustering algorithm to ensure diversity. (I discuss this topic more in chapter 4.)

You may also want to have multiple evaluation datasets that are compiled through different criteria. One common strategy is to have one dataset drawn from the same data as the training data and at least one out-of-domain evaluation dataset drawn from a different source. Out-of-domain datasets are often drawn from different types of media or different time periods. If all the training data for a natural language processing (NLP) task comes from historical news articles, for example, an out-of-domain dataset might come from recent social media data. For most real-world applications, you should use an out-of-domain evaluation dataset, which is the best indicator of how well your model is truly generalizing to the problem and not simply overfitting quirks of that particular dataset. This practice can be tricky with active learning, however, because as soon as you start labeling that data, it is no longer out-of-domain. If doing so is practical, I recommend that you keep an out-of-domain dataset to which you *don't* apply active learning. Then you can see how well your active learning strategy is generalizing the problem, not simply adapting and overfitting to the domains that it encounters.

1.3.3 When to use active learning

You should use active learning when you can annotate only a small fraction of your data and when random sampling will not cover the diversity of data. This recommendation covers most real-world scenarios, as the scale of the data becomes an important factor in many use cases.

A good example is the amount of data present in videos. Putting a bounding box around every object in every frame of a video, for example, would be time-consuming. Suppose that this video is of a self-driving car on a street with about 20 objects you care about (cars, pedestrians, signs, and so on). At 30 frames a second, that's 30

frames * 60 seconds * 20 objects, so you would need to create *36,000* boxes for one minute of data! Even the fastest human annotator would need at least 12 hours to annotate one minute's worth of data.

If we run the numbers, we see how intractable this problem is. In the United States, people drive an average of 1 hour per day, which means that people in the United States drive 95,104,400,000 hours per year. Soon, every car will have a video camera on the front to assist with driving. So 1 year's worth of driving in the United States alone would take 60,000,000,000 (60 trillion) hours to annotate. There are not enough people on Earth to annotate the videos of drivers in the United States today, even if the rest of the world did nothing but annotate data all day to make U.S. drivers safer.

So any data scientists at an autonomous-vehicle company needs to answer a variety of questions about the annotation process. Is every *n*th frame in a video OK? Can we sample the videos so that we don't have to annotate them all? Are there ways to design an interface for annotation to speed the process?

The intractability of annotation is true in most situations. There will be more data to annotate than there is budget or time to put each data point in front of a human. That's probably why the task is using machine learning in the first place. If you have the budget and time to annotate all the data points manually, you probably don't need to automate the task.

You don't need active learning in every situation, although human-in-the-loop learning strategies might still be relevant. In some cases, humans are required by law to annotate every data point, such as a court-ordered audit that requires a human to look at every communication within a company for potential fraud. Although humans will ultimately need to look at every data point, active learning can help them find the fraud examples faster and determine the best user interface to use. It can also identify potential errors with human annotations. In fact, this process is how many audits are conducted today.

There are also some narrow use cases in which you almost certainly don't need active learning. If you are monitoring equipment in a factory with consistent lighting, for example, it should be easy to implement a computer vision model to determine whether a given piece of machinery is on or off from a light or switch on that machine. As the machinery, lighting, camera, and the like are not changing over time, you probably don't need to use active learning to keep getting training data after your model has been built. These use cases are rare, however. Fewer than 1% of the use cases that I have encountered in industry have no use for more training data.

Similarly, there might be use cases in which your baseline model is accurate enough for your business use case or the cost of more training data exceeds any value that a more accurate model might provide. This criterion could also be the stopping point for active learning iterations.

1.4 Machine learning and human–computer interaction

For decades, a lot of smart people failed to make human translation faster and more accurate with the help of machine translation. It seems obvious that it should be possible to combine human translation and machine translation. As soon as a human translator needs to correct one or two errors in a sentence from machine translation output, however, it would be quicker for the translator to retype the whole sentence from scratch. Using the machine translation sentence as a reference when translating makes little difference in speed, and unless the human translator takes extra care, they will end up perpetuating errors in the machine translation, making their translation less accurate.

The eventual solution to this problem was not in the accuracy of the machine translation algorithms, but in the user interface. Instead of requiring human translators to retype whole sentences, modern translation systems let them use the same kind of predictive text that has become common in phones and (increasingly) in email and document composition tools. Human translators type translations as they always have, pressing Enter or Tab to accept the next word in the predicted translation, increasing their overall speed every time the machine translation prediction is correct. So the biggest breakthrough was in human–computer interaction, not the underlying machine learning algorithm.

Human–computer interaction is an established field in computer science that has recently become especially important for machine learning. When you are building interfaces for humans to create training data, you are drawing on a field that is at the intersection of cognitive science, social sciences, psychology, user-experience design, and several other fields.

1.4.1 User interfaces: How do you create training data?

Often, a simple web form is enough to collect training data. The human–computer interaction principles that underlie interaction with web forms are equally simple: people are accustomed to web forms because they see them all day. The forms are intuitive because a lot of smart people worked on and refined HTML forms. You are building on these conventions: people know how a simple HTML form works, so you don't need to educate them. On the other hand, breaking these conventions would confuse people, so you are constrained to expected behavior. You might have some idea that dynamic text could speed some task, but that convention could confuse more people than it helps.

The simplest interface—binary responses—is also the best for quality control. If you can simplify or break your annotation project into binary tasks, it is a lot easier to design an intuitive interface and to implement the annotation quality control features covered in chapters 8–11.

When you are dealing with more complicated interfaces, the conventions also become more complicated. Imagine that you are asking people to put polygons around certain objects in an image, which is a common use case for autonomous-vehicle

companies. What modalities would an annotator expect? Would they expect freehand, lines, paintbrushes, smart selection by color/region, or other selection tools? If people are accustomed to working on images in programs such as Adobe Photoshop, they might expect the same functionality when annotating images. In the same way that you are building on and constrained by people's expectations for web forms, you are also constrained by their expectations for selecting and editing images. Unfortunately, those expectations might require hundreds of hours of coding to build if you are offering full-featured interfaces.

For anyone who is undertaking a repetitive task such as creating training data, moving a mouse is inefficient and should be avoided if possible. If the entire annotation process can happen on a keyboard, including the annotation itself and any form submissions or navigations, the rhythm of the annotators will be greatly improved. If you have to include a mouse, you should be getting rich annotations to make up for the slower inputs.

Some annotation tasks have specialized input devices. People who transcribe speech to text often use foot pedals to navigate backward and forward in time in the audio recording. The process allows them to leave their hands on the keyboard. Navigating a recording with their feet is much more efficient than navigating the recording with a mouse.

Exceptions such as transcription aside, the keyboard is still king. Most annotation tasks haven't been popular for as long as transcription and therefore haven't developed specialized input devices. For most tasks, using a keyboard on a laptop or PC is faster than using the screen of a tablet or phone. It's not easy to type on a flat surface while keeping your eyes on inputs, so unless a task is a simple binary selection task or something similar, phones and tablets are not suited to high-volume data annotation.

1.4.2 Priming: What can influence human perception?

To get accurate training data, you have to take into account the focus of the human annotator, their attention span, and contextual effects that might cause them to make errors or to otherwise change their behavior. Consider a great example from linguistics research. In a study called "Stuffed toys and speech perception" (https://doi.org/10.1515/ling.2010.027), people were asked to distinguish between Australian and New Zealand accents. Researchers placed a stuffed toy kiwi bird or kangaroo (iconic animals for those countries) on a shelf in the room where participants undertook the study. The people who ran the study did not mention the stuffed toy to the participants; the toy was simply in the background. Incredibly, people interpreted an accent as sounding more New Zealand-like when a kiwi bird was present and more Australia-like when a kangaroo was present. Given this fact, it is easy to imagine that if you are building a machine learning model to detect accents (perhaps you are working on a smart home device that you want to work in as many accents as possible), you need to take context into account when collecting training data.

When the context or sequence of events can influence human perception, this phenomenon is known as *priming*. The most important type in creating training data

is *repetition priming*, which occurs when the sequence of tasks can influence someone's perception. If an annotator is labeling social media posts for sentiment, for example, and they encounter 99 negative sentiment posts in a row, they are more likely to make an error by labeling the hundredth post as negative when it is positive. The post may be inherently ambiguous (such as sarcasm) or a simple error caused by an annotator's fading attention during repetitive work. In chapter 11, I talk about the types of priming you need to control for.

1.4.3 The pros and cons of creating labels by evaluating machine learning predictions

One way to combine machine learning and ensure quality annotations is to use a simple binary-input form to have people evaluate a model prediction and confirm or reject that prediction. This technique can be a nice way to turn a more complicated task into a binary annotation task. You could ask someone whether a bounding box around an object is correct as a simple binary question that doesn't involve a complicated editing/selection interface. Similarly, it is easier to ask an annotator whether some word is a location in a piece of text than it is to provide an interface to efficiently annotate phrases that are locations in free text.

When you do so, however, you run the risk of focusing on localized model uncertainty and missing important parts of the problem space. Although you can simplify the interface and annotation accuracy evaluation by having humans evaluate the predictions of machine learning models, you still need a diversity strategy for sampling, even if that strategy is merely ensuring that a random selection of items is also available.

1.4.4 Basic principles for designing annotation interfaces

Based on what I've covered so far, here are some basic principles for designing annotation interfaces. I'll go into more detail on these principles throughout the book:

- Cast your problems as binary choices wherever possible.
- Ensure that expected responses are diverse to avoid priming.
- Use existing interaction conventions.
- Allow keyboard-driven responses.

1.5 Machine-learning-assisted humans vs. human-assisted machine learning

Human-in-the-loop machine learning can have two distinct goals: making a machine learning application more accurate with human input and improving a human task with the aid of machine learning. The two goals are sometimes combined, and machine translation is a good example. Human translation can be made faster by using machine translation to suggest words or phrases that a human can choose to accept or reject, much as your smartphone predicts the next word as you are typing. This task is a machine-learning-assisted human processing task. I've also worked with customers who use machine translation when human translation would be too expen-

sive. Because the content is similar across the human- and machine-translated data, the machine translation system gets more accurate over time from the data that is human-translated. These systems are hitting both goals, making the humans more efficient and making the machines more accurate.

Search engines are another great example of human-in-the-loop machine learning. People often forget that search engines are a form of AI despite being so ubiquitous for general search and for specific use cases such as e-commerce and navigation (online maps). When you search for a page online and click the fourth link that comes up instead of the first link, for example, you are probably training that search engine (information retrieval system) that the fourth link might be a better top response for your search query. There is a common misconception that search engines are trained only on feedback from end users. In fact, all the major search engines employ thousands of annotators to evaluate and tune their search engines. Evaluating search relevance is the single largest use case for human annotation in machine learning. Although there has been a recent rise in popularity of computer vision use cases, such as autonomous vehicles, and speech use cases, such as in-home devices and smartphones, search relevance is still the largest use case for professional human annotation.

However they appear at first glance, most human-in-the-loop machine learning tasks have some element of both machine-learning-assisted humans and human-assisted machine learning, so you need to design for both.

1.6 Transfer learning to kick-start your models

You don't need to start building your training data from scratch in most cases. Often, existing datasets are close to what you need. If you are creating a sentiment analysis model for movie reviews, for example, you might have a sentiment analysis dataset from product reviews that you can start with and then adapt to your use cases. This process—taking a model from one use case and adapting it to another—is known as transfer learning.

Recently, there has been a large increase in the popularity of adapting general pretrained models to new, specific use cases. In other words, people are building models *specifically* to be used in transfer learning for many use cases. These models are often referred to as *pretrained* models.

Historically, transfer learning has involved feeding the outputs of one process into another. An example in NLP might be

> General part-of-speech tagger > Syntactic parser > Sentiment analysis tagger

Today, transfer learning typically means

> Retraining part of a neural model to adapt to a new task (pretrained models) or using the parameters of one neural model as inputs to another

Figure 1.4 shows an example of transfer learning. A model can be trained on one set of labels and then retrained on another set of labels by keeping the architecture the same and freezing part of the model, retraining only the last layer in this case.

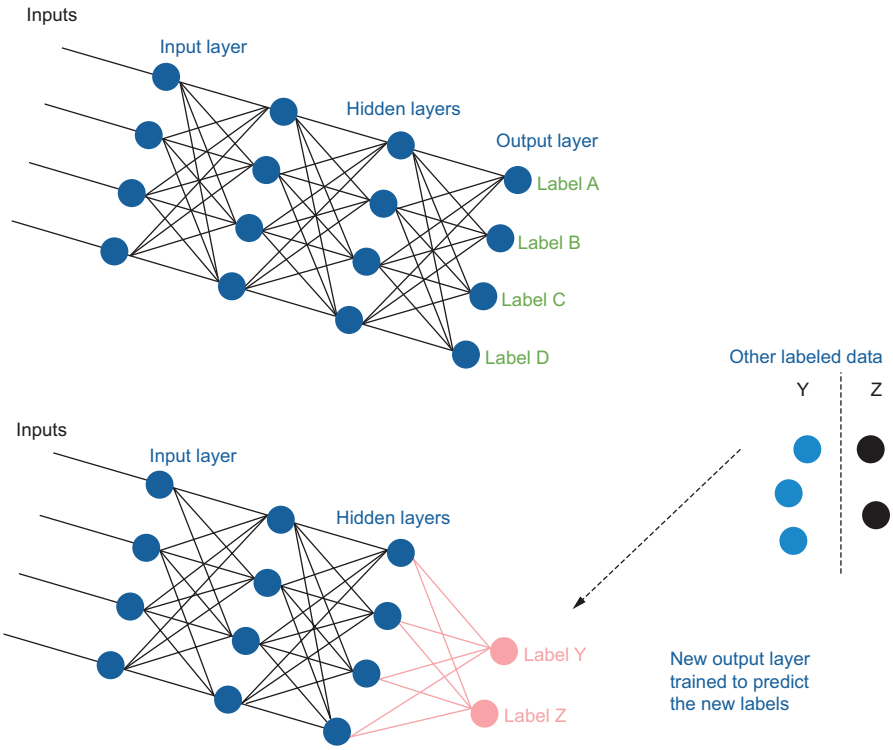


Figure 1.4 An example of transfer learning. A model was built to predict a label as "A," "B," "C," or "D." Retraining the last layer of the model and using far fewer human-labeled items than if we were training a model from scratch, the model is able to predict labels "Y" and "Z."

1.6.1 Transfer learning in computer vision

Transfer learning has seen the most progress recently in computer vision. A popular strategy is to start with the ImageNet dataset and build a model from the millions of examples to classify the 1,000 labels: sports, birds, human-made objects, and so on.

To learn to classify different types of sports, animals, and objects, the machine learning model is learning about the types of textures and edges that are needed to distinguish 1,000 types of items in images. Many of these textures and edges are more general than the 1,000 labels and can be used elsewhere. Because all the textures and edges are learned in the intermediate layers of the network, you can retrain only the last layer on a new set of labels. You may need only a few hundred or a few thousand examples for each new label, instead of millions, because you are already drawing on millions of images for the textures and edges. ImageNet has seen high success when people have retrained the final layer to new labels with little data, including objects such as cells in biology and geographic features from satellite views.

It is also possible to retrain several layers instead of the last one and to add more layers to the model from which you are transferring. Transfer learning can be used

with many architectures and parameters to adapt one model to a new use case, but with the same goal of limiting the number of human labels needed to build an accurate model on new data.

Computer vision has been less successful to date for moving beyond image labeling. For tasks such as detecting objects within an image, it is difficult to create transfer learning systems that can adapt from one type of object to another. The problem is that objects are being detected as collections of edges and textures rather than as whole objects. Many people are working on the problem, however, so there is no doubt that breakthroughs will occur.

1.6.2 Transfer learning in NLP

The big push for pretrained models for NLP is even more recent than for computer vision. transfer learning of this form has become popular for NLP only in the past two or three years, so it is one of the most cutting-edge technologies covered in this text, but it also might become out of date quickly.

ImageNet-like adaptation does not work for language data. Transfer learning for one sentiment analysis dataset to another sentiment analysis dataset provides an accuracy increase of only ~2–3%. Models that predict document-level labels don't capture the breadth of human language to the extent that equivalent computer vision models capture textures and edges. But you can learn interesting properties of words by looking at the contexts in which they occur regularly. Words such as *doctor* and *surgeon* might occur in similar contexts, for example. Suppose that you found 10,000 contexts in which any English word occurs, looking at the set of words before and after. You can see how likely the word *doctor* is to occur in each of these 10,000 contexts. Some of these contexts will be medical-related, so *doctor* will have a high score in those contexts. But most of the 10,000 contexts will not be medical-related, so *doctor* will have a low score in those contexts. You can treat these 10,000 scores like a 10,000-long vector. The word *surgeon* is likely to have a vector similar to that of *doctor* because it often occurs in the same context.

The concept of understanding a word by its context is old and forms the basis of functional theories of linguistics:

> *You shall know a word by the company it keeps (Firth, J. R. 1957:11).*

Strictly, we need to go below the word to get to the most important information. English is an outlier in that words tend to make good atomic units for machine learning. English allows for complex words such as *un-do-ing*; it is obvious why we would want to interpret the separate parts (morphemes), but English does this much more rarely than a typical language. What English expresses with word order, such as subject-verb-object, is more frequently expressed with affixes that English limits to things such as present and past tense and singular/plural distinctions. So for machine learning tasks that are not biased toward a privileged language such as English, which is an outlier, we need to model subwords.

Firth would appreciate this fact. He founded England's first linguistics department at SOAS, where I worked for two years helping record and preserve endangered languages. It was clear from my time there that the full breadth of linguistic diversity means that we need more fine-grained features than words alone. Human-in-the-loop machine learning methods are necessary if we are going to adapt the world's machine-learning capabilities to as many of the 7,000 world languages as possible.

When transfer learning had its recent breakthrough moment, it followed the principle of understanding words (or word segments) in context. We can get millions of labels for our models for free if we predict the word from its context:

My ___ is cute. He ___ play-ing

No human labeling is required. We can remove some percentage of the words in raw text and then turn the remaining text into a predictive machine-learning task. As you can guess, the first blank word might be *dog*, *puppy*, or *kitten*, and the second blank word is likely to be *is* or *was*. As with *surgeon* and *doctor*, we can predict words from context.

Unlike the early example in which transfer learning from one type of sentiment to another failed, these kinds of pretrained models have been widely successful. With only minor tuning from a model that predicts a word in context, it is possible to build state-of-the-art systems with small amounts of human labeling for language tasks such as question answering, sentiment analysis, and textual entailment. Unlike computer vision, transfer learning is quickly becoming ubiquitous for complicated NLP tasks such as summarization and translation.

The pretrained models are not complicated. The most sophisticated ones today are trained to predict a word in context, the order of words in a sentence, and the order of sentences. From that baseline model of three types of predictions that are inherent in the data, we can build almost any NLP use case with a head start. Because word order and sentence order are inherent properties of the documents, the pretrained models don't need human labels. They are still built like supervised machine learning tasks, but the training data is generated for free. The models might be asked to predict one in every ten words that have been removed from the data and to predict when certain sentences follow each other in the source documents, providing a powerful head start before any human labels are required for your task.

Pretrained models, however, are limited by how much unlabeled text is available. Much more unlabeled text is available in English than in other languages, even when you take the overall frequency of different languages into account. There will be cultural biases, too. The example *My dog is cute* might appear frequently in online text, which is the main source of data for pretrained models. But not everyone has a dog as a pet. When I briefly lived in the Amazon to study the Matsés language, monkeys were popular pets. The English phrase *My monkey is cute* rarely appears online, and the Matsés equivalent *Chuna bëdambo ikek* doesn't occur at all. Word vectors and the contextual models in pretrained systems do allow multiple meanings to be expressed by one word, so they could capture both *dog* and *monkey* in this context, but they are still biased toward the data on which they are trained, and the *monkey* context is unlikely to occur

in large volumes in any language. We need to be aware that pretrained systems will tend to amplify cultural biases.

Pretrained models still require additional human labels to achieve accurate results in their tasks, so transfer learning does not change our general architecture for human-in-the-loop machine learning. It can give us a substantial head start in labeling, however, which can influence the choice of active learning strategy that we use to sample additional data items for human annotation and even the interface by which humans provide that annotation.

Transfer learning also forms the basis of some of the advanced active learning strategies discussed in chapter 5 and the advanced data annotation and augmentation strategies in chapter 9.

1.7 What to expect in this text

To think about how the pieces of this text fit together, it can be useful to think of the topics in terms of a knowledge quadrant (figure 1.5).

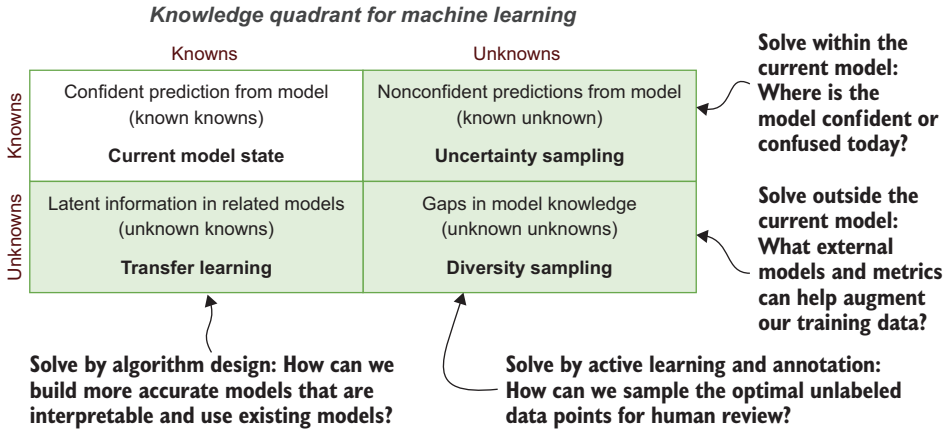


Figure 1.5 A machine learning knowledge quadrant, covering the topics in this book and expressing them in terms of what is known and unknown for your machine learning models

The four quadrants are

- *Known knowns*—What your machine learning model can confidently and accurately do today. This quadrant is your model in its current state.
- *Known unknowns*—What your machine learning model cannot confidently do today. You can apply uncertainty sampling to these items.
- *Unknown knowns*—Knowledge within pretrained models that can be adapted to your task. Transfer learning allows you to use this knowledge.
- *Unknown unknowns*—Gaps in your machine learning model. You can apply diversity sampling to these items.

The columns and rows are meaningful too, with the rows capturing knowledge of your model in its current state and the columns capturing the type of solutions needed:

- The top row captures your model's knowledge.
- The bottom row captures knowledge outside your model.
- The left column can be addressed by the right algorithms.
- The right column can be addressed by human interaction.

This text covers a wide range of technologies, so it might help to keep this figure handy to know where everything fits in.

The book has cheat sheets at the end of the first few chapters as a quick reference for the major concepts that were covered. You can keep these cheat sheets handy while reading later chapters.

Summary

- The broader human-in-the-loop machine learning architecture is an iterative process combining human and machine components. Understanding these components explains how the parts of this book come together.
- You can use some basic annotation techniques to start creating training data. Understanding these techniques ensures that you are getting annotations accurately and efficiently.
- The two most common active learning strategies are uncertainty sampling and diversity sampling. Understanding the basic principles of each type helps you strategize about the right combination of approaches for your particular problems.
- Human–computer interaction gives you a framework for designing the user-experience components of human-in-the-loop machine learning systems.
- Transfer learning allows us to adapt models trained from one task to another and build more accurate models with fewer annotations.

2 Getting started with human-in-the-loop machine learning

This chapter covers

- Ranking predictions by model confidence to identify confusing items
- Finding unlabeled items with novel information
- Building a simple interface to annotate training data
- Evaluating changes in model accuracy as you add more training data

For any machine learning task, you should start with a simple but functional system and build out more sophisticated components as you go. This guideline applies to most technology: ship the minimum viable product (MVP) and then iterate on that product. The feedback you get from what you ship first will tell you which pieces are the most important to build out next.

This chapter is dedicated to building your first human-in-the-loop machine learning MVP. We will build on this system as this book progresses, allowing you to learn about the different components that are needed to build more sophisticated data annotation interfaces, active learning algorithms, and evaluation strategies.

Sometimes, a simple system is enough. Suppose that you work at a media company, and your job is to tag news articles according to their topic. You already have topics such as sports, politics, and entertainment. Natural disasters have been in the news lately, and your boss has asked you to annotate the relevant past news articles as disaster-related to allow better search for this new tag. You don't have months to build out an optimal system; you want to get an MVP out as quickly as possible.

2.1 Beyond hacktive learning: Your first active learning algorithm

You may not realize it, but you've probably used active learning before. As you learned in chapter 1, active learning is the process of selecting the right data for human review. Filtering your data by keyword or some other preprocessing step is a form of active learning, although not an especially principled one.

If you have only recently started experimenting with machine learning, you have probably used common academic datasets such as ImageNet, the MNIST optical character recognition (OCR) dataset, and the CoNLL named entity recognition (NER) datasets. These datasets were heavily filtered with various sampling techniques before the actual training data was created. So if you randomly sample from any of these popular datasets, your sample is not truly random: it is a selection of data that conforms to whatever sampling strategies were used when these datasets were created. In other words, you unknowingly used a sampling strategy that was probably some handcrafted heuristic from more than a decade ago. You will learn more sophisticated methods in this text.

There is a good chance that you have used ImageNet, MNIST OCR, or the CoNLL NER datasets without realizing how filtered they are. Little formal documentation is available, and not mentioned in most that use these datasets, as I know by chance. ImageNet was created by colleagues when I was at Stanford; I ran one of the 15 research teams in the original CoNLL NER task; and I learned about the limitations of MNIST when it was mentioned in a now-famous foundational Deep Learning paper. It is obviously not ideal that piecing together how an existing dataset was created is so difficult and arbitrary, but until this book, there is no place telling you: *don't trust any existing dataset to be representative of data that you encounter in the real world.*

Because you are probably using filtered data by the time you build a machine learning model, it can be helpful to think of most machine learning problems as already being in the middle of the iteration process for active learning. Some decisions about data sampling have already been made; they led you to the current state of what data is annotated, and they probably weren't entirely optimal. So one of the first things you need to worry about is how to start sampling the right data as you move forward.

If you aren't explicitly implementing a good active learning strategy, instead employing ad hoc methods to sample your data, you are implementing *hacktive learning*.[1] It's fine to hack something together, but it is better to get the fundamentals right even if you are doing something quickly.

Your first human-in-the-loop machine learning system is going to look something like figure 2.1. For the remainder of this chapter, you will be implementing this architecture. This chapter assumes that you will be using the dataset introduced in section 2.2, but you can easily use your own data instead. Alternatively, you can build the system described here; then, by making changes in the data and annotation instructions, you should be able to drop in your own text annotation task.

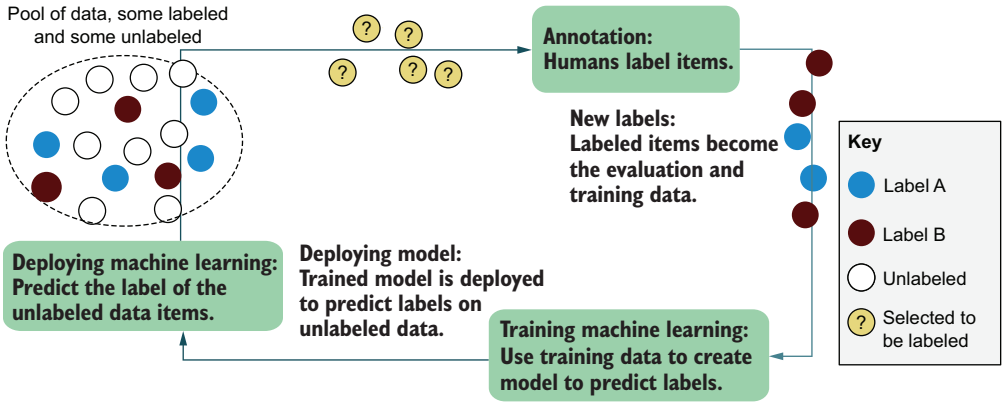


Figure 2.1 The architecture of your first human-in-the-loop machine learning system

2.2 The architecture of your first system

The first human-in-the-loop machine learning system that you will build in this text will label a set of news headlines as "disaster-related" or "not disaster-related." This real-world task could have many application areas:

- Using this dataset to build a machine learning model to help identify disaster-related news articles in real time to help with the response
- Adding a new "disaster-related" tag to news articles to improve the searchability and indexability of a database
- Supporting a social study about how disasters are reported in the media by allowing someone to analyze the relevant headlines

In global epidemic tracking, identifying news articles about outbreaks is an important task. H5N1 (bird flu) was reported openly weeks before it was identified as a new

[1] Thanks to Jennifer Prendki (one of the authors of an anecdote in this text) for the term *hacktive learning*. While working together, we misheard each other due to our different accents, and both of us understood *active learning* to be *hacktive learning*, accidentally inventing this useful phrase.

strain of the flu, and H1N1 (swine flu) was reported openly months in advance. If these reports had been put in front of virologists and epidemiologists sooner, they would have recognized the patterns of new strains of the flu and could have reacted sooner. Although this use case for your first human-in-the-loop machine learning system is simple, it is a real-world use case that could save lives.[2]

For data that you will be using throughout the book, you will use messages from several past disasters on which I worked as a professional disaster responder. In many of these cases, I ran the human-in-the-loop machine learning systems to process the data, so the examples are relevant to this text. The data includes messages sent following earthquakes in Haiti and Chile in 2010, floods in Pakistan in 2010, Hurricane Sandy in the United States in 2012, and a large collection of news headlines focused on disease outbreaks.

You will be joining students in NLP at Stanford, data science students at Udacity, and high-school students enrolled in AI for All (https://ai-4-all.org), who are also using this dataset as part of their courses today. You will be doing the task introduced at the start of the chapter: classifying news headlines. You can download the code and data at https://github.com/rmunro/pytorch_active_learning.

See the readme file for instructions on installing Python 3.6 or later and PyTorch on your machine. Versions of Python and PyTorch change rapidly, so I will keep the readme file updated with instructions for installation rather than try to include that information here.

If you are not familiar with PyTorch, start with the examples in this PyTorch tutorial: http://mng.bz/6gy5. The example in this chapter was adapted from a combination of this PyTorch example and the one in the PyTorch tutorial. If you become familiar with those two tutorials, all the code in this chapter should be clear to you. The data in the CSV files comprises two to five fields, depending on how processed it is, and looks something like the example in table 2.1.

Table 2.1 An example data file, with the ID, actual text, active learning sampling strategy chosen, and score for that sampling strategy

Text ID	Text	Label	Sampling strategy	Score
596124	Flood warning for Dolores Lake residents	1	Low confidence	0.5872
58503	First-aid workers arrive for earthquake relief	1	Random	0.6234
23173	Cyclists are lost trying to navigate new bike lanes	0	Random	0.0937

The data that you will be using in this chapter is from a large collection of news headlines. The articles span many years and hundreds of disasters, but most headlines *are not* disaster-related.

[2] For more on how we were tracking epidemics, see https://nlp.stanford.edu/pubs/Munro2012epidemics.pdf. Since I wrote this note in early 2019, COVID-19 has made the importance of this use case more obvious.

There are four locations for data in the repo:

- */training_data*—The data that your models will be trained on
- */validation_data*—The data that your models will be tuned with
- */evaluation_data*—The data that your models will be evaluated on for accuracy
- */unlabeled_data*—The large pool of data that you want to label

You will see the data in the CSV files in this repo, and they will have this format:

- 0. Text ID (a unique ID for this item)
- 1. Text (the text itself)
- 2. Label (the label: 1 = "disaster-related"; 0 = "not disaster-related")
- 3. Sampling strategy (the active learning strategy that we used to sample this item)
- 4. Confidence (the machine learning confidence that this item is "disaster-related")

(This list counts from 0 instead of 1 so that it will match the index of each field in the items/rows in the code).

These fields are enough information for you to build your first model. You will see that the unlabeled data in the example does not yet have a label, sampling strategy, or confidence, for obvious reasons.

If you want to jump in right away, you can run this script:

```
> python active_learning_basics.py
```

You will initially be prompted to annotate messages as "disaster-related" or "not disaster-related" to create the evaluation data. Then you will be prompted to do the same again for the initial training data. Only then will you see models start being built on your data and the active learning process beginning. We will return to the code later in this chapter and introduce the strategy behind it.

In an actual disaster, you would be classifying data into a large number of fine-grained categories. You might separate requests for food and water, for example, because people can go for much longer without food than without water, so requests for drinking water need to be responded to with more urgency than requests for food. On the other hand, you might be able to provide water locally with filtration, but food still needs to be shipped to the disaster-affected region for a longer period. As a result, different disaster relief organizations often focus on either food or water. The same is true for distinctions between types of medical aid, security, housing, and so on, all of which need fine-grained categories to be actionable. But in any of these situations, filtering between "relevant" and "not relevant" can be an important first step. If the volume of data is low enough, you might need machine learning assistance only to separate related from unrelated information; humans can take care of the rest of the categories. I have run disaster response efforts in which this was the case.

Also, in most disasters, you wouldn't be working in English. English makes up only about 5% of the world's conversations daily, so around 95% of communications about disasters are not in English. The broader architecture could be applied to any language,

however. The biggest difference is that English uses whitespace to break sentences into words. Most languages have more sophisticated prefixes, suffixes, and compounds that make individual words more complicated. Some languages, such as Chinese, don't use whitespace between most words. Breaking words into their constituent parts (*morphemes*) is an important task in itself. In fact, this was part of my PhD thesis: automatically discovering word-internal boundaries for any language in disaster response communications. An interesting and important research area would be to make machine learning truly equal across the world, and I encourage people to pursue it!

It helps to make your data assumptions explicit so that you can build and optimize the architecture that is best for your use case. It is good practice to include the assumptions in any machine learning system, so here are ours:

- The data is only in English.
- The data is in different varieties of English (United Kingdom, United States, English as a second language).
- We can use whitespace-delimited words as our features.
- A binary classification task is sufficient for the use case.

It should be easy to see how the broader framework for human-in-the-loop machine learning will work for any similar use case. The framework in this chapter could be adapted to image classification almost as easily as to another text classification task, for example.

If you have already jumped in, you will see that you are asked to annotate some additional data before you can build a model. This is good practice in general: looking at your data will give you better intuitions for every part of your model. See the following sidebar to learn why you should look at your data.

Sunlight is the best disinfectant

Expert anecdote by Peter Skomoroch

You need to look at real data in depth to know exactly what models to build. In addition to high-level charts and aggregate statistics, I recommend that data scientists go through a large selection of randomly selected, granular data regularly to let these examples wash over them. As executives look at company-level charts every week, and network engineers look over stats from system logs, data scientists should have intuition about their data and how it is changing.

When I was building LinkedIn's Skill Recommendations feature, I built a simple web interface with a Random button that showed recommendation examples alongside the corresponding model inputs so that I could quickly view the data and get an intuition for the kinds of algorithms and annotation strategies that might be most successful. This approach is the best way to ensure that you have uncovered potential issues and obtained vital high-quality input data. You're shining a light on your data, and sunlight is the best disinfectant.

Peter Skomoroch, the former CEO of SkipFlag (acquired by WorkDay), worked as a principal data scientist at LinkedIn on the team that invented the title "data scientist."

2.3 Interpreting model predictions and data to support active learning

Almost all supervised machine learning models will give you two things:

- A predicted label (or set of predictions)
- A number (or set of numbers) associated with each predicted label

The numbers are generally interpreted as confidences in the prediction, although this can be more or less true depending on how the numbers are generated. If there are mutually exclusive categories with similar confidence, you have good evidence that the model is confused about its prediction and that human judgment would be valuable. Therefore, the model will benefit most when it learns to correctly predict the label of an item with an uncertain prediction.

Suppose that we have a message that might be disaster-related, and the prediction looks like this:

```
{
    "Object": {
        "Label": "Not Disaster-Related",
        "Scores": {
            "Disaster-Related": 0.475524352,
            "Not Disaster-Related": 0.524475648
        }
    }
}
```

In this prediction, the message is predicted to be "Not Disaster-Related." In the rest of supervised machine learning, this label is what people care about most: was the label prediction correct, and what is the overall accuracy of the model when predicting across a large held-out dataset?

In active learning, however, the numbers associated with the prediction typically are what we care about most. You can see in the example that "Not Disaster-Related" is predicted with a 0.524 score. This score means that the system is 52.4% confident that the prediction was correct.

From the perspective of the task here, you can see why you might want a human to review the result anyway: there is still a relatively high chance that this is disaster-related. If it *is* disaster-related, your model is getting this example wrong for some reason, so it is likely that you want to add it to your training data so that you don't miss similar examples.

In chapter 3, we will turn to the problem of how reliable a score of 0.524 is. Especially for neural models, these confidences can be widely off. For the sake of this chapter, we can assume that although the exact number may not be accurate, we can generally trust the relative differences in confidence across multiple predictions.

2.3.1 Confidence ranking

Suppose that we had another message with this prediction:

```
{
    "Object": {
        "Label": "Not Disaster-Related",
        "Scores": {
            "Disaster-Related": 0.015524352,
            "Not Disaster-Related": 0.984475648
        }
    }
}
```

This item is also predicted as "Not Disaster-Related" but with 98.4% confidence, compared with 52.4% confidence for the first item. So the model is more confident about the second item than about the first. Therefore, it is reasonable to assume that the first item is more likely to be wrongly labeled and would benefit from human review. Even if we don't trust the 52.4% and 98.4% numbers (and we probably shouldn't, as you will learn in later chapters), it *is* reasonable to assume that the rank order of confidence will correlate with accuracy. This will generally be true of almost all machine learning algorithms and almost all ways of calculating accuracy: you can rank-order the items by the predicted confidence and sample the lowest-confidence items. For a probability distribution over a set of labels *y* for the item *x*, the confidence is given by the equation, where y* is the most confident (c) label:

$$\phi_c(x) = P_\theta(y^* | x)$$

For a binary prediction task like this example, you can simply rank by confidence and sample the items closest to 50% confidence. If you are attempting anything more complicated, however, such as predicting three or more mutually exclusive labels, labeling sequences of data, generating entire sentences (including translation and speech transcription), or identifying objects within images and videos, you have multiple ways to calculate confidence. We will return to other ways of calculating confidence in later chapters. The intuition about low confidence remains the same, and a binary task is easier for your first human-in-the-loop system.

2.3.2 Identifying outliers

As discussed in chapter 1, you often want to make sure that you are getting a diverse set of items for humans to label so that the newly sampled items aren't all like each other. This task can include making sure that you are not missing any important outliers. Some disasters are rare, such as a large asteroid crashing into the Earth. If a news headline says "Asteroid flattens Walnut Creek," and your machine learning model hasn't learned what an asteroid is or that Walnut Creek is a city, it is easy to see why your machine learning model might not have predicted this headline as being disaster-related. You could call this sentence an outlier in this regard: it lies farthest from anything you've seen before.

As with confidence ranking, we have many ways to ensure that we are maximizing the diversity of the content that is selected for human review. You will learn more about such approaches in later chapters. For now, we will focus on a simple metric: the average training data frequency of words in each unlabeled item. Here is the strategy that we will implement in this chapter:

1. For each item in the unlabeled data, count the average number of word matches it has with items already in the training data.
2. Rank the items by their average match.
3. Sample the item with the lowest average number of matches.
4. Add that item to the labeled data.
5. Repeat these steps until you have sampled enough for one iteration of human review.

Note that in step 4, when you have sampled the first item, you can treat that item as being labeled, because you know you are going to get a label for it later.

This method of determining outliers tends to favor small and novel headlines, so you will see that code adds 1 to the count as a smoothing factor. It also disfavors sentences with a lot of common words such as *the*, even if the other words are uncommon. So instead of using average matching, you could track the raw number of novel words to model the total amount of novel information in a headline instead of the overall average.

You could also divide the number of matches in the training data by the total number of times that word occurs across all the data and multiply each of these fractions, which would more or less give you the Bayesian probability of the element's being an outlier. Instead of using word matching, you could use more sophisticated edit-distance-based metrics that take the order of words within the sentence into account. Or you can use many other string-matching and other algorithms to determine outliers.

As with everything else, you can start by implementing the simple example in this chapter and experiment with others later. The main goal is an insurance policy: is there something completely different that we haven't seen yet? Probably not, but if there were, it would be the highest-value item to annotate correctly. We will look at ways of combining sampling by confidence and sampling by diversity in chapter 5.

We will also look at ways to combine your machine learning strategy with your annotation strategy. If you have worked in machine learning for a while but never in annotation or active learning, you have probably optimized models only for accuracy. For a complete architecture, you may want to take a more holistic approach in which your annotation, active learning, and machine learning strategies inform one another. You could decide to implement machine learning algorithms that can give more accurate estimates of their confidence at the expense of accuracy in label prediction. Or you might augment your machine learning models to have two types of inference: one to predict the labels and one to more accurately estimate the confidence of each prediction. If you are building models for more complicated tasks such as generating sequences of text (as in machine translation) or regions within images (as in object detection), the most common approach today is building separate inference capabilities

for the task itself and interpreting the confidence. We will look at these architectures in chapters 9–11 of this book.

The process for building your first human-in-the-loop machine learning model is summarized in figure 2.2.

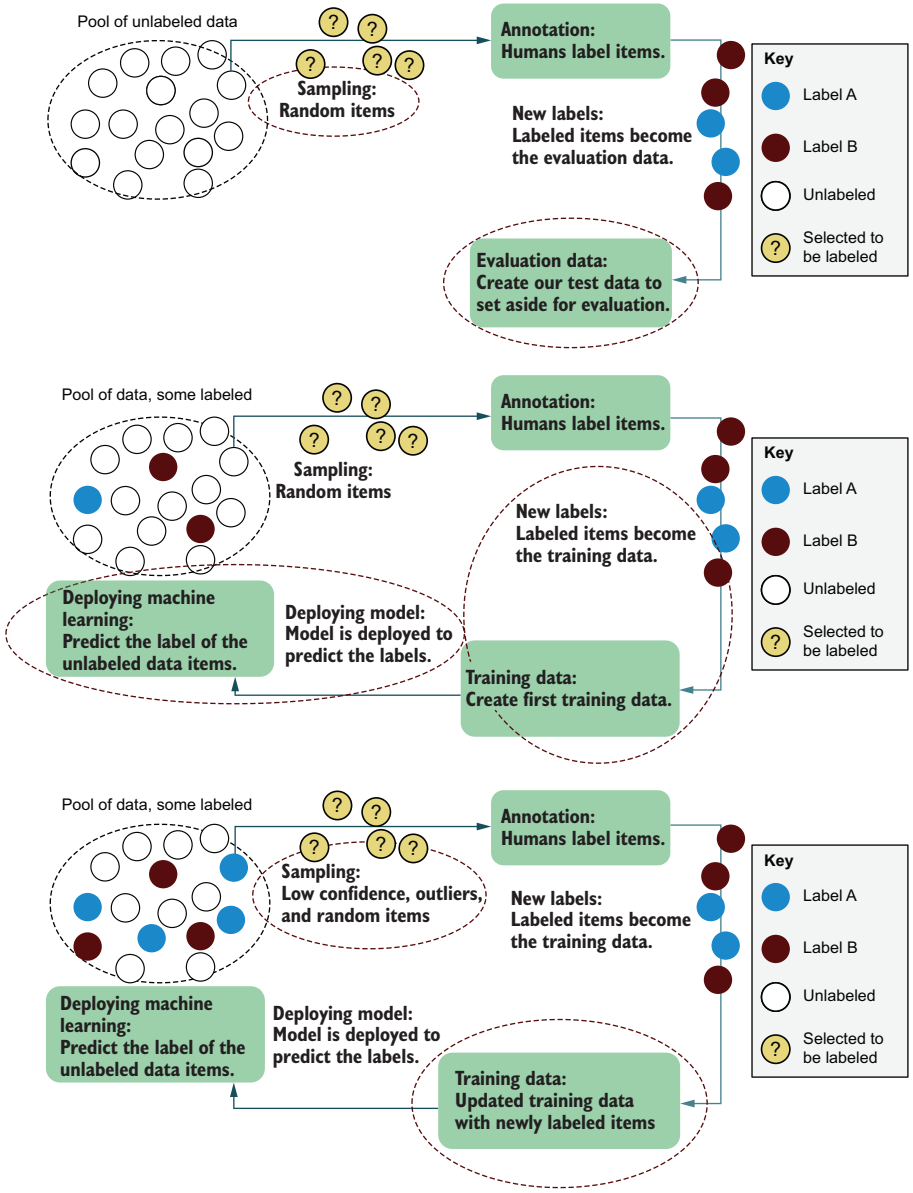


Figure 2.2 The iterative process in your first human-in-the-loop machine learning system. Initially (top), you are annotating a random sample of unlabeled items to set aside as your evaluation data. Then you are labeling the first items to be used for training data (middle), also starting with a random selection. After this point, you start using active learning (bottom) to sample items that are low-confidence or outliers.

2.3.3 What to expect as you iterate

In our example code, after we have enough evaluation and initial training data, we will iterate on active learning every 100 items. This number is probably a little small in terms of the number of items per iteration, as you'll be spending a lot of time waiting for the model to retrain for a relatively small number of new labeled items, but 100 is about right to get a feel for how much the sampled data changes in each iteration.

Here are some things you may notice as you iterate through the active learning process:

- *First iteration*—You are annotating mostly "not disaster-related" headlines, which can feel tedious. The balance will improve when active learning kicks in, but for now, it is necessary to get the randomly sampled evaluation data. You should also notice that this problem is not trivial because journalists often use disaster metaphors for nondisasters, especially sports teams (declaring war, a scoring drought, and so on). You will also be challenged by edge cases. Is a plane crash a disaster, for example, or does its status depend on the size of the plane and/or the cause? These edge cases will help you refine the definition of your task and create the right instructions for engaging a larger workforce to annotate your data at scale.
- *Second iteration*—You have created your first model! Your F-score is probably terrible, maybe only 0.20. Your area under the curve (AUC), however, might be around 0.75. (See the appendix for more on F-score and AUC.) So despite the bad accuracy, you can find disaster-related messages better than chance. You could fix the F-score by playing with the model parameters and architecture, but more data is more important than model architecture right now, as will become clear when you start annotating: you will immediately notice on your second iteration that a large number of items is disaster-related. In fact, most of the items may be. Early on, your model will still try to predict most things as "not disaster-related," so anything close to 50% confidence is at the "disaster-related" end of the scale. This example shows that active learning can be self-correcting: it is oversampling a lower-frequency label without requiring you to explicitly implement a targeted strategy for sampling important labels. You will also see evidence of overfitting. If your randomly selected items in the first iteration happened to have many headlines about floods, for example, you probably have *too* many headlines about floods and not enough about other types of disasters.
- *Third and fourth iterations*—You should start to see model accuracy improve, as you are now labeling many more "disaster-related" headlines, bringing the proposed annotation data closer to 50:50 for each label. If your model had overfitted some terms, such as the floods example, you should have seen some counterexamples, such as "New investment floods the marketplace," These counterexamples help push your models back to more accurate predictions for headlines with

these terms. If the data was genuinely disaster-related for everything with *flood* in it, these items are now predicted with high confidence and are no longer near 50%. Either way, the problem self-corrects, and the diversity of the headlines you are seeing should increase.

- *Fifth to tenth iterations*—Your models start to reach reasonable levels of accuracy, and you should see more diversity in the headlines. As long as either the F-score or AUC goes up by a few percentage points for every 100 annotations, you are getting good gains in accuracy. You are probably wishing that you had annotated more evaluation data so that you could be calculating accuracy on a bigger variety of held-out data. Unfortunately, you can't. It's almost impossible to go back to truly random sampling unless you are prepared to give up a lot of your existing labels.

Although it feels simple, the system that you are building in this chapter follows the same strategy as the initial release of Amazon Web Services's (AWS) SageMaker Ground Truth in 2018 (less than a year before this chapter was written). In fact, in the first version, SageMaker sampled only by confidence and didn't look for outliers in that release. Although the system you are building is simple, it is beyond the level of algorithmic sophistication of an active learning tool that is currently offered by a major cloud provider. I worked briefly on SageMaker Ground Truth when I was at AWS, so this is not a criticism of that product or my colleagues who put much more work into it than I did. Although active learning is becoming part of large-scale commercial offerings for the first time, it is still in an early stage.

We will cover more sophisticated methods for sampling in part 2 of this book. For now, it is more important to focus on establishing the iterative process for active learning, along with the best practices for annotation and retraining and evaluating your models. If you don't get your iteration and evaluation strategies correct, you can easily make your model worse instead of better and not even realize it.

2.4 Building an interface to get human labels

To label your data, you need to start with the right interface. We'll cover what that looks like for our example data in this section.

The right interface for human labeling is as important as the right sampling strategy. If you can make your interface 50% more efficient, that's as good as improving your active learning sampling strategy by 50%. Out of respect for the people who are doing the labeling, you should do as much as you can to ensure that they feel they are as effective as possible. If you genuinely don't know whether an interface or algorithm improvement is the best thing to focus on next, start with the interface to improve the work of the humans, and worry about your CPU's feelings later.

Part 3 of this book is dedicated to data annotation, so we will make a few assumptions to keep the discussion in this chapter simple:

- Annotators aren't making a significant number of errors in the labels, so we don't have to implement quality control for annotations.
- Annotators understand the task and labels perfectly, so they aren't accidentally choosing the wrong labels.
- Only one annotator is working at a time, so we don't have to keep track of any labeling in progress.

These assumptions are big ones. In most deployed systems, you need to implement quality control to ensure that annotators are not making mistakes; you will most likely need several iterations of annotation to refine the definitions of the labels and instructions; and you will need a system to track work assigned to multiple people in parallel. A simple annotation interface like the one discussed here is enough if you want to annotate some data quickly for exploratory purposes, as you are doing here.

2.4.1 A simple interface for labeling text

The interface that you build is determined by your task and the distribution of your data. For a binary labeling task like the one we are implementing here, a simple command-line interface is enough (figure 2.3). You will see it immediately if you run the script that we introduced in this chapter:

```
> python active_learning_basics.py
```

```
Please type 1 if this message is disaster-related, or hit Enter if not.
Type 2 to go back to the last message, type d to see detailed
definitions, or type s to save your annotations.

Firefighting continues in Blue Mountains

> 1
```

Figure 2.3 The command-line interface annotation tool for the example in this chapter

As discussed in the introduction, many human–computer interaction factors go into making a good interface for annotation. But if you have to build something quickly, do the following:

1. Build an interface that allows annotators to focus on one part of the screen.
2. Allow hot keys for all actions.
3. Include a back/undo option.

Get those three things right first, and graphic design can come later.

To see exactly what the code is doing, look at the repo at https://github.com/rmunro/pytorch_active_learning, or clone it locally and experiment with it. Excerpts from that code will be shared in this book for illustrative purposes.

You can see the code to elicit annotations in the first 20 lines of the `get_annotations()` function in the following listing.

Listing 2.1 Sampling the unlabeled items that we want to annotate

```
def get_annotations(data, default_sampling_strategy="random"):
    """Prompts annotator for label from command line and adds annotations to
    ➥ data

    Keyword arguments:
        data -- an list of unlabeled items where each item is
                [ID, TEXT, LABEL, SAMPLING_STRATEGY, CONFIDENCE]
        default_sampling_strategy -- strategy to use for each item if not
        ➥ already specified
    """

    ind = 0
    while ind <= len(data):
        if ind < 0:
            ind = 0 # in case you've gone back before the first
        if ind < len(data):
            textid = data[ind][0]
            text = data[ind][1]
            label = data[ind][2]
            strategy =  data[ind][3]

            if textid in already_labeled:
                print("Skipping seen "+label)
                ind+=1
            else:
                print(annotation_instructions)
                label = str(input(text+"\n\n> "))
                ...
                ...
```

The input() function prompts the user for input.

For our data, the labels are a little unbalanced because most headlines are not related to disasters. This fact has interface-design implications. It would be inefficient and boring for someone to continually select "Not Disaster-Related." You can make "Not Disaster-Related" the default option to improve efficiency so long as you have a back option when annotators inevitably get primed to select the default. You probably did this yourself: annotated quickly and then had to go back when you pressed the wrong answer. You should see this functionality in the next and final 20 lines of code of the `get_annotations()` function.

Listing 2.2 Allowing the annotator to go back to avoid errors through repetition

```
def get_annotations(data, default_sampling_strategy="random"):
                ...
                ...

                if label == "2":
```

```
                ind-=1  # go back
            elif label == "d":
                print(detailed_instructions) # print detailed
                ➥ instructions
            elif label == "s":
                break  # save and exit
            else:
                if not label == "1":
                    label = "0" # treat everything other than 1 as 0

                data[ind][2] = label # add label to our data

                if data[ind][3] is None or data[ind][3] == "":
                    data[ind][3] = default_sampling_strategy # default if
                    ➥ none given
                ind+=1

    else:
        #last one - give annotator a chance to go back
        print(last_instruction)
        label = str(input("\n\n> "))
        if label == "2":
            ind-=1
        else:
            ind+=1

return data
```

2.4.2 Managing machine learning data

For a deployed system, it is best to store your annotations in a database that takes care of backups, availability, and scalability. But you cannot always browse a database as easily as you can files on a local machine. In addition to adding training items to your database, or if you are building a simple system, it can help to have locally stored data and annotations that you can quickly spot-check.

In our example, we will separate the data into separate files according to the label, for additional redundancy. Unless you are working in an organization that already has good data-management processes in place for annotation and machine learning, you probably don't have the same kind of quality control for your data as you do for your code, such as unit tests and good versioning. So it is wise to be redundant in how you store your data. Similarly, you will see that the code appends files but never writes over files. It also keeps the unlabeled_data.csv file untouched, checking for duplicates in the other datasets instead of deleting headlines from that file when the item has been labeled.

Redundancy in how you store labels and enforcing nondeletion of data will save you a lot of headaches when you start experimenting. I've never met a machine learning professional who hasn't accidentally deleted labeled data at some point, so follow this advice! Also remember that if you are storing data on your local machine, that data may belong to someone else or have sensitive content. Make sure that you have permission to store the data, and delete the data when you no longer need it.

Although the topic isn't covered in this book, version control for your data is also important, especially if you are updating your instructions as you go. Some older labels may be incorrect, and you want to be able to reproduce them if you want to re-create your active learning iterations later.

2.5 Deploying your first human-in-the-loop machine learning system

Now let's put all the pieces of your first human-in-the-loop system together! If you didn't do so earlier in the chapter, download the code and data from https://github.com/rmunro/pytorch_active_learning, and see the readme file for installation instructions.

You can run this code immediately, and it will start prompting you to annotate data and automatically train after each iteration. You should experience the changes in data at each iteration that you learned in section 2.3.3.

To see what is happening under the hood, let's go through the main components of this code and the strategies behind it. We use a simple PyTorch machine learning model for text classification. We will use a shallow model that can be retrained quickly to make our iterations fast. In PyTorch, this entire model definition is a dozen lines of code.

Listing 2.3 Simple PyTorch text classification model with one hidden layer

```
class SimpleTextClassifier(nn.Module):  # inherit pytorch's nn.Module
    """Text Classifier with 1 hidden layer

    """

    def __init__(self, num_labels, vocab_size):
        super(SimpleTextClassifier, self).__init__() # call parent init

        # Define model with one hidden layer with 128 neurons
        self.linear1 = nn.Linear(vocab_size, 128)
        self.linear2 = nn.Linear(128, num_labels)

    def forward(self, feature_vec):
        # Define how data is passed through the model

        hidden1 = self.linear1(feature_vec).clamp(min=0) # ReLU
        output = self.linear2(hidden1)
        return F.log_softmax(output, dim=1)
```

Hidden layer with 128 neurons/nodes

Output layer predicting each label

Using linear activation function for our output layer

Optimizing our hidden layer with a ReLU activation function

Returning the log softmax of our linear output to optimize our model in training and to return as a probability distribution for prediction

Our input layer contains the one-hot encoding for every word in our feature set (thousands), our output layer is the two labels, and our hidden layer is 128 nodes.

For training, we know that the data is imbalanced between the labels initially, so we want to ensure that we select something closer to an even number of items for each label. This specification is set in these variables at the start of the code:

```
epochs = 10 # number of epochs per training session
select_per_epoch = 200  # number to sample per epoch per label
```

We are going to train our models for 10 epochs, and for each epoch, we are going to randomly select 200 items from each label. This approach won't make our model completely even, because we are still selecting from a bigger variety of not-disaster-related text across all the epochs, but it will be enough that we get some signal from our data even when we have only 100 or so disaster-related examples.

(The hidden neurons, epochs, and items selected per epoch are sensible but otherwise arbitrary starting points. You can experiment with different hyperparameters, but at the start of the annotation process, you should be concentrating on the data.)

The code to train our model is the `train_model()` function shown next.

Listing 2.4 Training the text classification model

```
def train_model(training_data, validation_data = "", evaluation_data = "",
➥ num_labels=2, vocab_size=0):
    """Train model on the given training_data

    Tune with the validation_data
    Evaluate accuracy with the evaluation_data
    """

    model = SimpleTextClassifier(num_labels, vocab_size)
    # let's hard-code our labels for this example code
    # and map to the same meaningful booleans in our data,
    # so we don't mix anything up when inspecting our data
    label_to_ix = {"not_disaster_related": 0, "disaster_related": 1}

    loss_function = nn.NLLLoss()
    optimizer = optim.SGD(model.parameters(), lr=0.01)

    # epochs training
    for epoch in range(epochs):
        print("Epoch: "+str(epoch))
        current = 0

        # make a subset of data to use in this epoch
        # with an equal number of items from each label

        shuffle(training_data) #randomize the order of the training data
        related = [row for row in training_data if '1' in row[2]]
        not_related = [row for row in training_data if '0' in row[2]]

        epoch_data = related[:select_per_epoch]
        epoch_data += not_related[:select_per_epoch]
        shuffle(epoch_data)
```

Select an equal amount of items with each label to effectively oversample the smaller label, especially in early iterations of labeling.

```
# train our model
for item in epoch_data:
    features = item[1].split()
    label = int(item[2])

    model.zero_grad()

    feature_vec = make_feature_vector(features, feature_index)
    target = torch.LongTensor([int(label)])

    log_probs = model(feature_vec)

    # compute loss function, do backward pass, and update the
    ➥ gradient
    loss = loss_function(log_probs, target)
    loss.backward()
    optimizer.step()
```

You can see that we are keeping our training hyperparameters constant, such as the learning rate and type of activation functions. For an actual system, you would probably want to experiment with training hyperparameters and also with architectures that better model the sequence of words or could better model clusters of pixels if you are doing image classification.

If you are doing any hyperparameter tuning at all, you should create validation data and use that data to tune your model, as you are already accustomed to doing in machine learning. In fact, you may want multiple kinds of validation datasets, including one drawn from your training data at each iteration, one drawn from your unlabeled data before you use active learning, and one drawn from the remaining unlabeled items at each iteration. We will return to validation data for active learning in chapter 3. For now, we save you the additional annotations. If you want to tune your model in the example in this chapter, pull a random selection of data from your training data set at each iteration.

The remainder of the `train_model()` function evaluates the accuracy of the new model and saves it to file in models/. I cover evaluation in the next section.

As stated earlier, you should become familiar with your data before you start building any machine learning system. Fortunately, this best practice applies to active learning too. You should select your evaluation data first, and you should be one of the people who labels it.

2.5.1 Always get your evaluation data first

Evaluation data is often called a test set or held-out data, and for this task, it should be a random sample of headlines that we annotate. We will always hold out these headlines from our training data so that we can track the accuracy of our model after each iteration of active learning.

It is important to get the evaluation data first, as there are many ways to inadvertently bias your evaluation data after you have started other sampling techniques. Here are some of the things that can go wrong if you don't pull out your evaluation data first:

- If you forget to sample evaluation data from your unlabeled items until after you have sampled by low confidence, your evaluation data will be biased toward the remaining high-confidence items, and your model will appear to be more accurate than it is.
- If you forget to sample evaluation data and you pull evaluation data from your training data after you have sampled by confidence, your evaluation data will be biased toward low-confidence items, and your model will appear to be less accurate than it is.
- If you have implemented outlier detection and later try to pull out evaluation data, it is almost impossible to avoid bias, as the items you pulled out have already contributed to the sampling of additional outliers.

What happens if you don't make evaluation data first?

It's difficult to know how accurate your model is if you don't remember to get evaluation data first. This mistake is one of the biggest ones I've seen people make. As soon as data scientists get any new human labels, they naturally want to add those labels to their training data to see how much more accurate their models get. But if your evaluation data is an afterthought, and you aren't careful about making it truly random, you won't know how accurate your model is. I have seen companies building self-driving cars, social media feeds, and dating apps get evaluation data wrong. Know that the car that swerved past you today, the news article that was recommended to you, and the person you might one day marry may all have been determined by machine learning models of uncertain accuracy.

If you want to start training right away, at least set the evaluation data aside first so that it doesn't factor into your analysis. You can return to annotate that data later or annotate in parallel with your training and validation data.

Finally, it may not be possible to select truly random data if you are applying your model to a continuously changing feed of information. In ongoing disaster-response situations, this will absolutely be the case, as new information is reported about the changing conditions and needs over time. For the example we are working on here, we are tasked with labeling a finite set of news headlines, so it is meaningful to select a random sample of the headlines to be in our training data. We will return to sampling strategies for evaluation data in more complicated contexts in chapter 3.

The code to evaluate the accuracy of your model at each iteration is the `evaluate_model()` function.

Listing 2.5 Evaluating the model on held-out data

```
def evaluate_model(model, evaluation_data):
    """Evaluate the model on the held-out evaluation data

    Return the f-value for disaster-related and the AUC
    """

    related_confs = [] # related items and their confidence of being related
    not_related_confs = [] # not related items and their confidence of
    ➥ being _related_

    true_pos = 0.0 # true positives, etc
    false_pos = 0.0
    false_neg = 0.0

    with torch.no_grad():
        for item in evaluation_data:
            _, text, label, _, _, = item

            feature_vector = make_feature_vector(text.split(), feature_index)
            log_probs = model(feature_vector)

            # get confidence that item is disaster-related
            prob_related = math.exp(log_probs.data.tolist()[0][1])

            if(label == "1"):
                # true label is disaster related
                related_confs.append(prob_related)
                if prob_related > 0.5:
                    true_pos += 1.0
                else:
                    false_neg += 1.0
            else:
                # not disaster-related
                not_related_confs.append(prob_related)
                if prob_related > 0.5:
                    false_pos += 1.0
                    ...
                    ...
```

The PyTorch tensors are 2D, so we need to pull out only the predictive confidence.

This code gets the predicted confidence that each item is "disaster-related" and tracks whether each prediction was correct or incorrect. Raw accuracy would not be a good metric to use here. Because the frequency of the two labels is unbalanced, you will get almost 95% accuracy from predicting "not disaster-related" each time. This result is not informative, and our task is specifically to find the disaster-related headlines, so we will calculate accuracy as the F-score of the disaster-related predictions.

In addition to caring about the F-score, we care whether confidence correlates with accuracy, so we calculate the area under the ROC curve. A ROC (receiver operating characteristic) curve rank-orders a dataset by confidence and calculates the rate of true positives versus false positives.

See the appendix for definitions and discussions of precision, recall, F-score, and AUC, all of which are implemented in the `evaluate_model()` function of our code.

Listing 2.6 Calculating precision, recall, F-score, and AUC

```
def evaluate_model(model, evaluation_data):
                    ...
                    ...
    # Get FScore
    if true_pos == 0.0:
        fscore = 0.0
    else:
        precision = true_pos / (true_pos + false_pos)
        recall = true_pos / (true_pos + false_neg)
        fscore = (2 * precision * recall) / (precision + recall)

    # GET AUC
    not_related_confs.sort()
    total_greater = 0 # count of how many total have higher confidence
    for conf in related_confs:
        for conf2 in not_related_confs:
            if conf < conf2:
                break
            else:
                total_greater += 1

    denom = len(not_related_confs) * len(related_confs)
    auc = total_greater / denom

    return[fscore, auc]
```

Harmonic mean of precision and recall

For items with the label we care about ("related," in this case), we want to know how many are predicted to have that label with greater confidence than the items without that label.

If you look at filenames for any models that you have built in the models directory, you will see that the filename includes a timestamp, the accuracy of the model by F-score and AUC, and the number of training items. It is good data-management practice to give your models verbose and transparent names, which will let you track accuracy over time with each iteration simply by looking at the directory listing.

2.5.2 Every data point gets a chance

By including new randomly sampled items in each iteration of active learning, you get a baseline in that iteration. You can compare the accuracy from training on the random items with your other sampling strategies, which can tell you how effective your sampling strategies are compared with random sampling. You will already know how many newly annotated items are different from your model's predicted label, but you won't know how much they will change the model for future predictions after they have been added to the training data.

Even if your other active learning strategies fail in the iteration, you will still get incremental improvement from the random sample, so random sampling is a nice fallback.

There is an ethical choice here too. We are acknowledging that all strategies are imperfect, so every data item still has some chance of being selected randomly and being reviewed by a human, even if none of the sampling strategies would have selected it. In an actual disaster scenario, would you want to eliminate the chance that someone would see an important headline because your sampling strategies would never select it? The ethical question is one you should ask yourself depending on the data and use case you are addressing.

2.5.3 Select the right strategies for your data

We know that disaster-related headlines are rare in our data, so the strategy of selecting outliers is not likely to select many disaster-related items. Therefore, the example code focuses on selecting by confidence and sampling data for each iteration according to the following strategy:

- 10% randomly selected from unlabeled items
- 80% selected from the lowest confidence items
- 10% selected as outliers

Assuming that the low-confidence items are truly 50:50 disaster-related and not disaster-related, the annotators should see a little more than 4/10 disaster-related messages when a large number of items have been annotated and our models are stable. This result is close enough to equal that we don't have to worry that ordering effects will prime the annotators in later iterations.

The following three listings contain the code for the three strategies. First, we get the low confidence predictions.

Listing 2.7 Sampling items with low confidence

```
def get_low_conf_unlabeled(model, unlabeled_data, number=80, limit=10000):
    confidences = []
    if limit == -1:
    print("Get confidences for unlabeled data (this might take a while)")
    else:
    # only apply the model to a limited number of items
    shuffle(unlabeled_data)
    unlabeled_data = unlabeled_data[:limit]

    with torch.no_grad():
        for item in unlabeled_data:
            textid = item[0]
            if textid in already_labeled:
                continue

            text = item[1]

            feature_vector = make_feature_vector(text.split(), feature_index)
            log_probs = model(feature_vector)
```

```
            prob_related = math.exp(log_probs.data.tolist()[0][1])

            if prob_related < 0.5:
                confidence = 1 - prob_related
            else:
                confidence = prob_related

            item[3] = "low confidence"
            item[4] = confidence
            confidences.append(item)

    confidences.sort(key=lambda x: x[4])
    return confidences[:number:]
```

Get the probabilities for each label for the item.

Order the items by confidence.

Next, we get the random items.

Listing 2.8 Sample random items

```
def get_random_items(unlabeled_data, number = 10):
    shuffle(unlabeled_data)

    random_items = []
    for item in unlabeled_data:
        textid = item[0]
        if textid in already_labeled:
            continue
        random_items.append(item)
        if len(random_items) >= number:
            break

    return random_items
```

Finally, we get the outliers.

Listing 2.9 Sample outliers

```
def get_outliers(training_data, unlabeled_data, number=10):
    """Get outliers from unlabeled data in training data
    Returns number outliers

    An outlier is defined as the percent of words in an item in
    unlabeled_data that do not exist in training_data
    """
    outliers = []

    total_feature_counts = defaultdict(lambda: 0)

    for item in training_data:
        text = item[1]
        features = text.split()

        for feature in features:
            total_feature_counts[feature] += 1
```

Count all features in the training data.

```
while(len(outliers) < number):
    top_outlier = []
    top_match = float("inf")

    for item in unlabeled_data:
        textid = item[0]
        if textid in already_labeled:
            continue

        text = item[1]
        features = text.split()
        total_matches = 1 # start at 1 for slight smoothing
        for feature in features:
            if feature in total_feature_counts:
                total_matches += total_feature_counts[feature]

        ave_matches = total_matches / len(features)
        if ave_matches < top_match:
            top_match = ave_matches
            top_outlier = item

    # add this outlier to list and update what is 'labeled',
    # assuming this new outlier will get a label
    top_outlier[3] = "outlier"
    outliers.append(top_outlier)
    text = top_outlier[1]
    features = text.split()
    for feature in features:
        total_feature_counts[feature] += 1

return outliers
```

Add the number of times this feature in the unlabeled data item occurred in the training data.

Update the training data counts for this item to help with diversity for the next outlier that is sampled.

You can see that by default in the `get_low_conf_unlabeled()` function, we are predicting the confidence for only 10,000 unlabeled items, rather than from across the entire dataset. This example makes the time between iterations more manageable, as you would be waiting for many minutes or even hours for all predictions, depending on your machine. This example increases the diversity of the data too, as we are selecting low-confidence items from a different subset of unlabeled items each time.

2.5.4 Retrain the model and iterate

Now that you have your newly annotated items, you can add them to your training data and see the change in accuracy from your model. If you run the script that you downloaded at the start of this chapter, you will see that retraining happens automatically after you finish annotating each iteration.

If you look at that code, you also see the controls that combine all the code that we went through in this chapter. This additional code is the hyperparameters, such as the number of annotations per iteration, and the code at the end of the file to make sure that you get the evaluation data first, train the models, and start iterating with active learning when you have enough evaluation data. The example in this chapter has

fewer than 500 lines of unique code, so it is worth taking the time to understand what is going on in each step and thinking about how you might extend any part of the code.

If you come from a machine learning background, the number of features will probably jump out at you. You probably have more than 10,000 features for only 1,000 labeled training items. That is not what your model should look like if you are not labeling any more data: you would almost certainly get better accuracy if you reduced the number of features. But somewhat counterintuitively, you want a large number of features, especially in the early iterations of active learning, when you want to make every feature count for the rare disaster-related headlines. Otherwise, your early model would be even more biased toward the type of headlines that you happened to sample first randomly. There are many ways that you might want to combine your machine learning architecture and active learning strategies, and I will cover the major ones in chapters 9–11.

After you complete 10 or so iterations of annotation, look at your training data. You will notice that most of the items were selected through low confidence, which is not a surprise. Look for ones that are listed as selected by outlier, and you might be surprised. There will probably be a few examples with words that are obvious (to you) as being disaster-related, which means that these examples increased the diversity of your dataset in a way that might otherwise have been missed.

Although active learning can be self-correcting, can you see any evidence that it didn't self-correct some bias? Common examples include oversampling extra-long or extra-short sentences. The computer vision equivalent would be oversampling images that are extra-large or extra-small, or high- or low-resolution. Your choice of outlier strategy and machine learning model might oversample based on features like these, which are not core to your goal. You might consider applying the methods in this chapter to different buckets of data in that case: lowest-confidence short sentences, lowest-confidence medium sentences, and lowest-confidence long sentences.

If you like, you can also experiment with variations on your sampling strategies within this code. Try retraining on only the randomly selected items, and compare the resulting accuracy with another system retrained on the same number of items selected by low confidence and using outlier sampling. Which strategy has the greatest impact, and by how much?

You can also think about what you should develop next:

- A more efficient interface for annotation
- Quality controls to help stop errors in annotation
- Better active learning sampling strategies
- More sophisticated neural architectures for the classification algorithm

Your subjective experience might be different from mine, and trying this example on your own data instead of the example dataset provided here might have changed things, too. But chances are good that you identified one of the first three options as

the most important component to build out next. If you come from a machine learning background, your first instinct may be to keep the data constant and start experimenting with more sophisticated neural architectures. That task can be the best next step, but it's rarely the most important one early on. Generally, you should get your data right first; tuning the machine learning architecture becomes more important later in the iterations.

The rest of this book helps you learn how to design better interfaces for annotation, implement better quality control for annotation, devise better active learning strategies, and arrive at better ways to combine these components.

Summary

- A simple human-in-the-loop machine learning system can cover the entire cycle, from sampling unlabeled data to updating the model. This approach lets you get started quickly with a complete MVP system that you can build out as needed.
- Two simple active learning strategies are easy to implement: sampling the least most confident items from predictions and sampling outliers. Understanding the basic goals of each of these strategies will help you dive deeper into uncertainty and diversity sampling later in this book.
- A simple command-line interface can allow humans to annotate data efficiently. Even a simple text-only interface can be efficient if it is built according to general human–computer interaction principles.
- Good data management, such as creating evaluation data as the first task, is important to get right. If you don't get your evaluation data right, you may never know how accurate your model is.
- Retraining a machine learning model with newly annotated data at regular iterations shows that your model gets more accurate over time. If designed correctly, the active learning iterations are naturally self-correcting, with overfitting in one iteration corrected by the sampling strategy in the following iterations.

Part 2

Active learning

Now that you have learned about human-in-the-loop architectures in the first two chapters, we will spend four chapters on active learning: the set of techniques for sampling the most important data for humans to review.

Chapter 3 covers *uncertainty sampling*, introducing the most widely used techniques for understanding a model's uncertainty. The chapter starts by introducing different ways to interpret uncertainty from a single neural model and then looks at uncertainty from different types of machine learning architectures. The chapter also covers how to calculate uncertainty when you have multiple predictions for each data item, such as when you are using an ensemble of models.

Chapter 4 tackles the complicated problem of identifying where your model might be confident but *wrong* due to undersampled or nonrepresentative data. It introduces a variety of data sampling approaches that are useful for identifying gaps in your model's knowledge, such as clustering, representative sampling, and methods that identify and reduce real-world bias in your models. Collectively, these techniques are known as *diversity sampling*.

Uncertainty sampling and diversity sampling are most effective when combined, so chapter 5 introduces ways to combine different strategies into a comprehensive active learning system. Chapter 5 also covers some advantage transfer learning techniques that allow you to adapt machine learning models to predict which items to sample.

Chapter 6 covers how the active learning techniques can be applied to different kinds of machine learning tasks, including object detection, semantic segmentation, sequence labeling, and language generation. This information, including the strengths and weaknesses of each technique, will allow you to apply active learning to any machine learning problem.

3 Uncertainty sampling

This chapter covers

- Understanding the scores of a model prediction
- Combining predictions over multiple labels into a single uncertainty score
- Combining predictions from multiple models into a single uncertainty score
- Calculating uncertainty with different kinds of machine learning algorithms
- Deciding how many items to put in front of humans per iteration cycle
- Evaluating the success of uncertainty sampling

The most common strategy that people use to make AI smarter is for the machine learning models to tell humans when they are uncertain about a task and then ask the humans for the correct feedback. In general, unlabeled data that confuses an algorithm is most valuable when it is labeled and added to the training data. If the algorithm can already label an item with high confidence, it is probably correct.

This chapter is dedicated to the problem of interpreting when our model is trying to tell us when it is uncertain about its task. But it is not always easy to know when a model is uncertain and how to calculate that uncertainty. Beyond simple

binary labeling tasks, the different ways of measuring uncertainty can produce vastly different results. You need to understand and consider all methods for determining uncertainty to select the right one for your data and objectives.

For example, imagine that you are building a self-driving car. You want to help the car understand the new types of objects (pedestrians, cyclists, street signs, animals, and so on) that it is encountering as it drives along. To do that, however, you need to understand when your car is uncertain about what object it is seeing and how to best interpret and address that uncertainty.

3.1 Interpreting uncertainty in a machine learning model

Uncertainty sampling is a set of techniques for identifying unlabeled items that are near a decision boundary in your current machine learning model. Although it is easy to identify when a model is confident—there is one result with very high confidence—you have many ways to calculate uncertainty, and your choice will depend on your use case and what is the most effective for your particular data.

We explore four approaches to uncertainty sampling in this chapter:

- *Least confidence sampling*—Difference between the most confident prediction and 100% confidence. In our example, if the model was most confident that a pedestrian was in the image, least confidence captures how confident (or uncertain) that prediction was.
- *Margin of confidence sampling*—Difference between the two most confident predictions. In our example, if the model is most confident that a pedestrian was in the image and second most confident that the image contained an animal, margin of confidence captures the difference between the two confidences.
- *Ratio of confidence*—Ratio between the two most confident predictions. In our example, if the model is most confident that a pedestrian was in the image and the second most confident that the image contained an animal, ratio captures the *ratio* (not difference) between the two confidences.
- *Entropy-based sampling*—Difference between all predictions, as defined by information theory. In our example, entropy-based sampling would capture how much *every* confidence differed from every other.

We'll also look at how to determine uncertainty from different types of machine learning algorithms and how to calculate uncertainty when you have multiple predictions for each data item, such as when you are using an ensemble of models.

Understanding the strengths and weaknesses of each method requires going deeper into exactly what each strategy is doing, so this chapter provides detailed examples along with the equations and code. You also need to know how the confidences are generated before you can start interpreting them correctly, so this chapter starts with how to interpret your model's probability distributions, especially if they are generated by softmax, the most popular algorithm for generating confidences from neural models.

3.1.1 Why look for uncertainty in your model?

Let's return to our self-driving-car example. Suppose that your car spends most of its time on highways, which it is already good at navigating and which have a limited number of objects. You don't see many cyclists or pedestrians on major highways, for example. If you randomly selected video clips from the car's video cameras, your selections will mostly be from highways, where the car is already confident and driving well. There will be little that a human can do to improve the driving skills of the car if humans are mostly giving the car feedback about highway driving, on which the car is already confident.

Therefore, you want to know when your self-driving car is most confused as it is driving. So you decide to take video clips from where the car is most uncertain about the objects it is detecting and then have the human provide the *ground truth* (training data) for the objects in those video clips. The human can identify whether a moving object is a pedestrian, another car, a cyclist, or some other important object that the car's object detection system might have missed. Different objects can be expected to move at different speeds and to be more or less predictable, which will help the car anticipate the movements of those objects.

It might be the case, for example, that the car was most confused when driving through snowstorms. If you show video clips only from a snowstorm, that data doesn't help the car in the 99% of situations when it is not in a snowstorm. In fact, that data could make the car worse. The snowstorm will limit the visible range, and you could unintentionally bias the data so that the car's behavior makes sense only in a snowstorm and is dangerous elsewhere. You might teach the car to ignore all distant objects, as they simply cannot be seen when it is snowing; thus, you would limit the car's ability to anticipate objects at a distance in nonsnowing conditions. So you need different kinds of conditions in which your car is experiencing uncertainty.

Furthermore, it's not clear how to define uncertainty in the context of multiple objects. Is the uncertainty about the most likely object that was predicted? Was it between the two most likely predictions? Or should you take into account every possible object when coming up with an overall uncertainty score for some object that the car detected? When you drill down, deciding what objects from self-driving-car videos you should put in front of a human for review is difficult.

Finally, your model is not telling you in plain language when it is uncertain: even for a single object, the machine learning model gives you a number that might *correspond* to the confidence of the prediction but might not be a reliable measure of accuracy. Our starting point in this chapter is knowing when your model is uncertain. From that base, you will be able to build your broader uncertainty sampling strategies.

The underlying assumption of all active learning techniques is that some data points are more valuable to your model than others. (For a specific example, see the following sidebar.) In this chapter, we'll start with interpreting your model's outputs by taking a look at softmax.

Not all data is equal

Expert anecdote by Jennifer Prendki

If you care about your nutrition, you don't go to the supermarket and randomly select items from the shelves. You might eventually get the nutrients you need by eating random items from the supermarket shelves, but you will eat a lot of junk food in the process. I think it is weird that in machine learning, people still think it's better to sample the supermarket randomly than figure out what they need and focusing their efforts there.

The first active learning system I built was by necessity. I was building machine learning systems to help a large retail store make sure that when someone searched on the website, the right combination of products came up. Almost overnight, a company reorg meant that my human labeling budget was cut in half, and we had a 10x increase in inventory that we had to label. So my labeling team had only 5% the budget per item that we previously did. I created my first active learning framework to discover which was the most important 5%. The results were better than random sampling with a bigger budget. I have used active learning in most of my projects ever since, because not all data is equal!

Jennifer Prendki is the CEO of Alectio, a company that specializes in finding data for machine learning. She previously led data science teams at Atlassian, Figure Eight, and Walmart.

3.1.2 Softmax and probability distributions

As you discovered in chapter 2, almost all machine learning models give you two things:

- A predicted label (or set of predictions)
- A number (or set of numbers) associated with each predicted label

Let's assume that we have a simple object detection model for our self-driving car, one that tries to distinguish among only four types of objects. The model might give us a prediction like the following.

Listing 3.1 JSON-encoded example of a prediction

```
{
    "Object": {
        "Label": "Cyclist",
        "Scores": {
            "Cyclist": 0.9192784428596497,
            "Pedestrian": 0.01409964170306921,
            "Sign": 0.049725741147994995,
            "Animal": 0.016896208748221397
        }
    }
}
```

In this prediction, the object is predicted to be "Cyclist" with 91.9% accuracy. The scores will add to 100%, giving us the probability distribution for this item.

This output is most likely from *softmax*, which converts the logits to a 0–1 range of scores using the exponents. Softmax is defined as follows

$$\sigma(z_i) = \frac{e^{z_i}}{\sum_j e^{z_j}}$$

and as shown in figure 3.1.

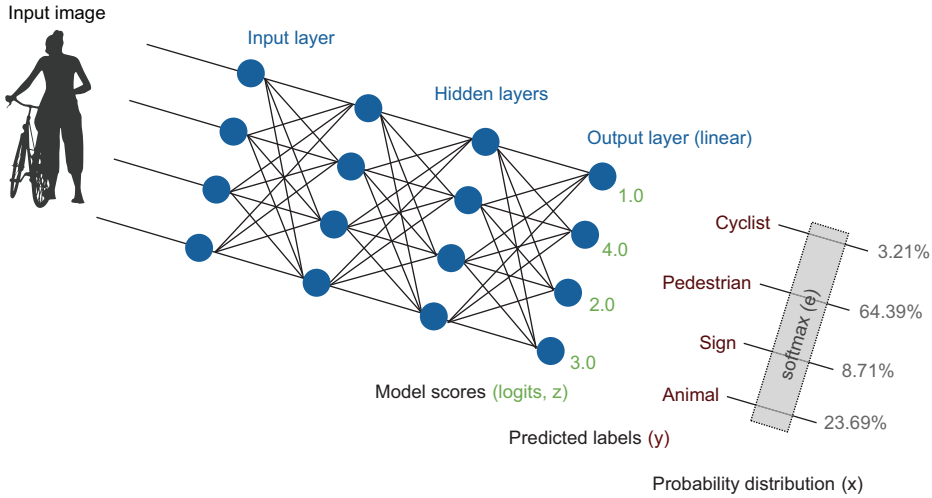


Figure 3.1 How softmax creates probability distributions. A linear activation function is used on the output layer, creating model scores (logits) that are then converted to probability distributions via softmax.

Because softmax divides by exponentials, it loses the scale of the logits. The logits in figure 3.1, for example, are [1, 4, 2, 1]. If the logits were [101, 104, 102, 101], softmax would produce the same probability distribution, so the level of activation in our model is lost in the output. We'll look at how to take activation into account in chapter 4. In this chapter, it's important to understand how some information is lost when only the probability distribution is used.

If you have only used the outputs of softmax in the past, I strongly recommend reading the appendix. As explained there, the softmax base (e) is arbitrary, and by changing the base, you can change the ranked order of confidence for predictions on different items. This fact isn't widely known and wasn't reported at all before this book. Rank order is important for uncertainty sampling, as you will see in this chapter, so for your own experiments, you might want to try changing the softmax base (or, equivalently, the temperature) in addition to employing the techniques described later in this chapter.

One common way to get more accurate confidences from your model is to adjust the base/temperature of softmax by using a validation dataset so that the probability distribution matches the actual accuracy as closely as possible. You might adjust the base/temperature of softmax so that a confidence score of 0.7 is correct 70% of the time, for example. A more powerful alternative to adjusting the base/temperature is using a local regression method such as LOESS to map your probability distributions to the actual accuracy on your validation data. Every stats package will have one or more local regression methods that you can experiment with.

If you are modeling uncertainty only so that you can sample the most uncertain items for active learning, however, it might not matter if the probability distributions are not accurate reflections of the accuracy. Your choice will depend on what you are trying to achieve, and it helps to know all the techniques that are available.

3.1.3 Interpreting the success of active learning

You can calculate the success of active learning with accuracy metrics such as F-score and AUC, as you did in chapter 2. If you come from an algorithms background, this technique will be familiar to you.

Sometimes, however, it makes more sense to look at the human cost. You could compare two active learning strategies in terms of the number of human labels that are required to get to a certain accuracy target, for example. This can be substantially bigger or smaller than comparing the accuracy with the same number of labels, so it can be useful to calculate both.

If you are not putting the items back into the training data, and therefore not implementing the full active learning cycle, it makes more sense to evaluate purely in terms of how many *incorrect* predictions were surfaced by uncertainty sampling. That is, when you sample the N most uncertain items, what percentage was incorrectly predicted by the model?

For more on human-centric approaches to evaluating quality, such as the amount of time needed to annotate data, see the appendix, which goes into more detail about ways to measure model performance.

3.2 Algorithms for uncertainty sampling

Now that you understand where the confidences in the model predictions come from, you can think about how to interpret the probability distributions to find out where your machine learning models are most uncertain.

Uncertainty sampling is a strategy for identifying unlabeled items that are near a decision boundary in your current machine learning model. If you have a binary classification task, like the one you saw in chapter 2, these items are predicted as being close to 50% probability of belonging to either label; therefore, the model is uncertain. These items are most likely to be classified wrongly; therefore, they are the most likely to result in a human label that is different from the predicted label. Figure 3.2 shows how uncertainty sampling should find items close to the decision boundary.

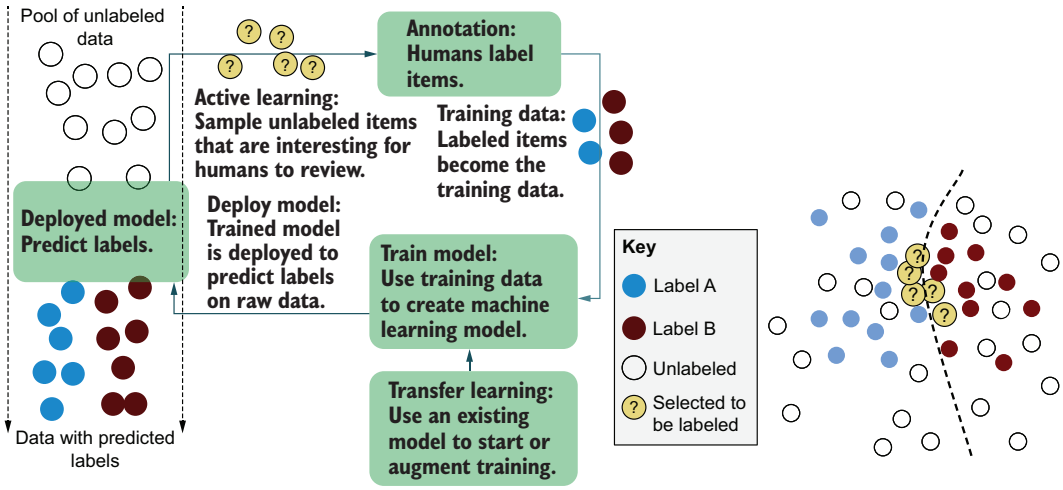


Figure 3.2 Uncertainty sampling is an active learning strategy that oversamples unlabeled items that are closer to the decision boundary (and sometimes to one another), and are therefore more likely to get a human label that results in a change in that decision boundary.

There are many algorithms for calculating uncertainty, some of which we will visit here. They all follow the same principles:

- Apply the uncertainty sampling algorithm to a large pool of predictions to generate a single uncertainty score for each item.
- Rank the predictions by the uncertainty score.
- Select the top *N* most uncertain items for human review.
- Obtain human labels for the top *N* items, retrain the model with those items, and iterate on the processes.

The three methods covered in this chapter are invariant of the data being predicted: a given item will get the same uncertainty score independent of the scores given to other items being predicted. This invariance helps with the simplicity and predictability of the approaches in this chapter: the rank order of uncertainty scores is enough to find the most uncertain across a set of predictions. Other techniques, however, can take the distribution of predictions to change the individual scores. We will return to this topic in chapters 5 and 6.

> **NOTE** For binary classification tasks, the strategies in this chapter are identical, but for three or more labels, the strategies diverge quickly.

3.2.1 Least confidence sampling

The simplest and most common method for uncertainty sampling takes the difference between 100% confidence and the most confidently predicted label for each item. You saw this implementation of active learning in chapter 2. Let's refer to the softmax

result as the probability of the label given the prediction. We know that softmax isn't strictly giving us probabilities, but these equations are general equations that apply to probability distributions from any sources, not only from softmax. The basic equation is simply the probability of the highest confidence for the label, which you implemented in chapter 2:

$$\phi_{LC}(x) = P_\theta(y^* | x)$$

Although you can rank order by confidence alone, it can be useful to convert the uncertainty scores to a 0–1 range, where 1 is the most uncertain score. In that case, we have to normalize the score. We subtract the value from 1, multiply the result by the number of labels, and divide by the number of labels –1. We do this because the minimum confidence can never be less than the one divided by the number of labels, which is when all labels have the same predicted confidence. So least confidence sampling with a 0-1 range is

$$\phi_{LC}(x) = (1 - P_\theta(y^* | x)) \times \frac{n}{n-1}$$

The following listing has an implementation of least confidence sampling in PyTorch.

Listing 3.2 Least confidence sampling in PyTorch

```
def least_confidence(self, prob_dist, sorted=False):
    """
    Returns the uncertainty score of an array using
    least confidence sampling in a 0-1 range where 1 is most uncertain

    Assumes probability distribution is a pytorch tensor, like:
      tensor([0.0321, 0.6439, 0.0871, 0.2369])

    Keyword arguments:
      prob_dist -- a pytorch tensor of real numbers between 0 and 1 that
      ➥ total to 1.0
      sorted - if the probability distribution is pre-sorted from largest to
      ➥ smallest
    """
    if sorted:
      simple_least_conf = prob_dist.data[0]
    else:
      simple_least_conf = torch.max(prob_dist)

    num_labels = prob_dist.numel() # number of labels

    normalized_least_conf = (1 - simple_least_conf) *
    ➥ (num_labels / (num_labels - 1))

    return normalized_least_conf.item()
```

Let's apply least confidence to get an uncertainty score for our self-driving-car prediction. The confidence for "Pedestrian" is all that counts here. Using our example, this uncertainty score would be (1 – 0.6439) * (4 / 3) = 0.4748. Least confidence sampling, therefore, gives you ranked order of predictions where you will sample items with the lowest confidence for their predicted label. This method is sensitive to the values of the second, third, and so on only in that the sum of the other predictions will be the score itself: the amount of confidence that will go to labels other than the most confident.

Predicted label	Cyclist	Pedestrian	Sign	Animal
softmax	0.0321	0.6439	0.0871	0.2369

This method will not be sensitive to uncertainty between any of the other predictions: with the same confidence for the most confident, the second to *n*th confidences can take any values without changing the uncertainty score. If you care only about the most confident prediction for your particular use case, this method is a good starting point. Otherwise, you will want to use one of methods discussed in the following sections.

Least confidence is sensitive to the base used for the softmax algorithm. This example is a little counterintuitive, but recall the example in which softmax(base=10) gives ~0.9 confidence, which would result in an uncertainty score of 0.1—much less than 0.35 on the same data. For different bases, this score will change the overall ranking. Higher bases for softmax will stretch out the differences between the most confident label and the other labels; therefore, at higher bases, the difference between the label confidences will come to weigh more than the absolute difference between the most-confident label and 1.0.

3.2.2 Margin of confidence sampling

The most intuitive form of uncertainty sampling is the difference between the two most confident predictions. That is, for the label that the model predicted, how much more confident was it than for the next-most-confident label? This is defined as

$$\phi_{MC}(x) = P_\theta(y_1^* | x) - P_\theta(y_2^* | x)$$

Again, we can convert this to a 0–1 range. We have to subtract from 1.0 again, but the maximum possible score is already 1, so there is no need to multiply by any factor:

$$\phi_{MC}(x) = 1 - (P_\theta(y_1^* | x) - P_\theta(y_2^* | x))$$

Following is an implementation of margin of confidence sampling with PyTorch.

Listing 3.3 Margin of confidence sampling in PyTorch

```
def margin_confidence(self, prob_dist, sorted=False):
    """
    Returns the uncertainty score of a probability distribution using
```

```
    margin of confidence sampling in 0-1 range where 1 is most uncertain

    Assumes probability distribution is a pytorch tensor, like:
      tensor([0.0321, 0.6439, 0.0871, 0.2369])

    Keyword arguments:
      prob_dist -- a pytorch tensor of real numbers between 0 and 1 that
      ➥ total to 1.0
      sorted -- if the probability distribution is pre-sorted from largest to
      ➥ smallest
        """
    if not sorted:
      prob_dist, _ = torch.sort(prob_dist, descending=True)

    difference = (prob_dist.data[0] - prob_dist.data[1])
    margin_conf = 1 - difference

    return margin_conf.item()
```

Let's apply margin of confidence sampling to our example data. "Pedestrian" and "Animal" are the most-confident and second-most-confident prediction. Using our example, this uncertainty score would be 1.0 - (0.6439 - 0.2369) = 0.5930.

Predicted label	Cyclist	Pedestrian	Sign	Animal
softmax	0.0321	0.6439	0.0871	0.2369

This method will not be sensitive to uncertainty for any but the two most confident predictions: with the same difference in confidence for the most and second-most confident, the third to *nth* confidences can take any values without changing the uncertainty score.

If you care only about the uncertainty between the predicted label and the next-most-confident prediction for your particular use case, this method is a good starting point. This type of uncertainty sampling is the most common type that I've seen people use in industry.

Margin of confidence is less sensitive than least confidence sampling to the base used for the softmax algorithm, but it is still sensitive. Although softmax(base=10) would give a margin of confidence score of 0.1899 for our dataset, compared with 0.5930 with base *e,* all of the two most probable scores will move. Those scores will move at slightly different rates, depending on the total relative difference of all raw scores, but recall that we are sampling from when the model is most uncertain—that is, when the most-confident scores tend to be as low as possible and therefore most similar. For this reason, you might get a difference of only a few percentage points when you sample the most uncertain items by margin of confidence sampling under different bases of softmax.

3.2.3 Ratio sampling

Ratio of confidence is a slight variation on margin of confidence, looking at the ratio between the top two scores instead of the difference. It is the best uncertainty sampling method for improving your understanding of the relationship between confidence and softmax. To make the technique a little more intuitive, think of the ratio as capturing how many times more likely the first label was than the second-most-confident:

$$\phi_{RC}(x) = P_\theta(y_1^* | x) / P_\theta(y_2^* | x)$$

Now let's plug in our numbers again:

0.6439 / 0.2369 = 2.71828

We get back the natural log, $e = 2.71828$! Similarly, if we use base 10, we get

90.01% / 9.001% = 10

We get back 10—the base we used! This example is a good illustration of why e is an arbitrary base for generating confidences. (See the appendix for more on this topic.). Is "Pedestrian" really 2.71828 more likely as a prediction than "Animal" in this context? Probably not. It's doubtful that it's exactly 10 times more likely, either. The only thing that ratio of confidence is telling us is that the raw score from our models was "1" different between "Pedestrian" and "Animal"—nothing more. Ratio of confidence with a division can be defined in terms of the raw scores, in this case with softmax(base=), where is the base used for softmax (if not e):

$$\beta^{(z_1^* - z_2^*)}$$

Ratio of confidence is invariant across any base used in softmax. The score is determined wholly by the distance between the top two raw scores from your model; therefore, scaling by the base or temperature will not change the rank order. To give ratio of confidence a 0-1 normalized range, you can simply take the inverse of the preceding equation:

$$\phi_{RC}(x) = P_\theta(y_2^* | x) / P_\theta(y_1^* | x)$$

We used the noninverted version above so that it directly outputs their softmax base for illustrative purposes. The following listing has an implementation of ratio of confidence sampling using PyTorch.

Listing 3.4 Ratio of confidence sampling in PyTorch

```
def ratio_confidence(self, prob_dist, sorted=False):
    """
    Returns the uncertainty score of a probability distribution using
    ratio of confidence sampling in 0-1 range where 1 is most uncertain
```

```
    Assumes probability distribution is a pytorch tensor, like:
      tensor([0.0321, 0.6439, 0.0871, 0.2369])

    Keyword arguments:
      prob_dist --  pytorch tensor of real numbers between 0 and 1 that total
      ➥ to 1.0
      sorted -- if the probability distribution is pre-sorted from largest to
      ➥ smallest
        """
    if not sorted:
      prob_dist, _ = torch.sort(prob_dist, descending=True)

    ratio_conf = prob_dist.data[1] / prob_dist.data[0]

    return ratio_conf.item()
```

I hope that this example gives you another good way to intuit why margin of confidence sampling is relatively invariant: there's no big difference between subtracting your two highest values and dividing your two highest values when your goal is to rank them.

Happily, where margin of confidence with subtraction *does* differ from ratio of confidence, it does what we want by favoring the most uncertain. Although margin of confidence and ratio of confidence don't explicitly look at the confidences beyond the two most confident, they influence the possible values. If the third-most-confident value is 0.25, the first and second can differ by 0.5 at most. So if the third-most-confident prediction is relatively close to the first and second, the uncertainty score for margin of confidence increases. This variation is small and doesn't occur directly as a result of margin of confidence; it is a byproduct of the denominator in the softmax equation being larger as a result of the larger score for the third-most-confident value, which becomes disproportionately larger as an exponential. Nonetheless, this behavior is right; all else being equal, margin of confidence looks for uncertainty beyond the two most confident predictions in what would otherwise be a tie.

Unlike margin of confidence, in which the variation from the third to *n*th predictions is a lucky byproduct of softmax, our next-most-popular uncertainty sampling strategy explicitly models all the predictions.

3.2.4 Entropy (classification entropy)

One way to look at uncertainty in a set of predictions is by whether you expect to be surprised by the outcome. This concept underlies the entropy technique. How surprised would you be by each of the possible outcomes, relative to their probability?

It is intuitive to think about entropy and surprise in terms of a sporting team you supported for a long time even though it was on a losing streak. For me, that team is the Detroit Lions American football team. In recent years, even when the Lions are ahead early in a game, they still have only a 50% chance of winning that game. So even if the Lions are up early in the game, I don't know what the result will be, and

there is an equal amount of surprise either way in every game. Entropy does not measure the emotional toll of losing—only the surprise. The entropy equation gives us a mathematically well-motivated way to calculate the surprise for outcomes, as shown in figure 3.3.

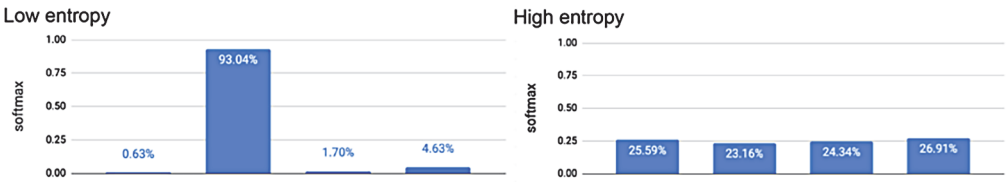


Figure 3.3 Example of low entropy (left) and high entropy (right). High entropy occurs when the probabilities are most like one another and there is the most surprise in any one prediction from the distribution. Entropy is sometimes a little counterintuitive, because the left graph has the most variability and three highly unlikely events. Those three unlikely events, however, are more than canceled by the one highly likely event. Four events at around equal likelihood will have greater total entropy, even if the three rarer events would have greater information on the rare times that they occur.

Entropy applied to a probability distribution involves multiplying each probability by its own log and taking the negative sum:

$$\phi_{ENT}(x) = -\sum_{y} P_{\theta}(y|x) \log_2 P_{\theta}(y|x)$$

We can convert the entropy to a 0–1 range by dividing by the log of the number of predictions (labels):

$$\phi_{ENT}(x) = \frac{-\sum_{y} P_{\theta}(y|x) \log_2 P_{\theta}(y|x)}{\log_2(n)}$$

The following listing shows an implementation of ratio of entropy score using Python and the PyTorch library.

Listing 3.5 Entropy-based sampling in PyTorch

```
def entropy_based(self, prob_dist):
    """
    Returns uncertainty score of a probability distribution using entropy

    Assumes probability distribution is a pytorch tensor, like:
      tensor([0.0321, 0.6439, 0.0871, 0.2369])

    Keyword arguments:
      prob_dist -- a pytorch tensor of real numbers between 0 and 1 that
      ➥ total to 1.0
      sorted -- if the probability distribution is pre-sorted from largest to
      ➥ smallest
    """
```

```
    log_probs = prob_dist * torch.log2(prob_dist)
    raw_entropy = 0 - torch.sum(log_probs)

    normalized_entropy = raw_entropy / math.log2(prob_dist.numel())

    return normalized_entropy.item()
```

Multiply each probability by its base 2 log.

First, don't be scared by another arbitrary base, log(base=2), which is used for historical reasons: the choice of base for entropy does not change the uncertainty sampling rank order. Unlike with softmax, calculating the entropy with different bases for uncertainty sampling does *not* change the rank order of scores across a dataset. You will get different entropy scores depending on the base, but the entropy scores will change monotonically for every probability distribution and therefore will not change the rank order for uncertainty sampling. Base 2 is used in entropy for historical reasons, as entropy comes from information theory, which deals with compressing data streams in binary bits. Let's calculate the entropy on our example data:

Predicted label	Cyclist	Pedestrian	Sign	Animal
P(y|x) aka softmax	0.0321	0.6439	0.0871	0.2369
log2(P(y|x))	–4.963	–0.635	–3.520	–2.078
P(y|x) log2(P(y|x))	–0.159	–0.409	–0.307	–0.492

Summing the numbers and negating them returns

$0 - \text{SUM}(-0.159, -0.409, -0.307, -0.492) = 1.367$

Dividing by the log of the number of labels returns

$1.367 / \log_2(4) = 0.684$

Note that the P(y|x) log(P(y|x)) step is not monotonic with respect to the probability distribution given by softmax. "Pedestrian" returns –0.409, but "Animal" returns –0.492. So "Animal" contributes most to the final entropy score even though it is neither the most-confident or least-confident prediction.

Data ranked for uncertainty by entropy is sensitive to the base used by the softmax algorithm and about equally sensitive as least confidence. It is intuitive why this is the case: entropy *explicitly* uses every number in the probability distribution, so the further these numbers are spread out via a higher base, the more divergent the result will be.

Recall our example in which softmax(base=10) gives ~0.9% confidence, which would result in an uncertainty score of 0.1—much less than 0.35 on the same data. For different bases, this score will change the overall ranking. Higher bases for softmax will stretch out the differences between the most-confident label and the other labels.

3.2.5 A deep dive on entropy

If you want to get deeper into entropy, you can try plugging different confidences into the inner part of the equation where each confidence is multiplied by its own log, such as 0.3 * log(0.3). For this measure of entropy, the per-prediction score of P(y|x) log(P(y|x)) will return the largest (negative) numbers for confidences of around 0.3679. Unlike in softmax, Euler's number is special, as $e^{-1} = 0.3679$. The formula used to derive this result is known as *Euler's Rule*, itself a derivation of the *Thâbit ibn Kurrah Rule*, created sometime in the ninth century to generate amicable numbers. The largest (negative) numbers for each prediction will be around 0.3679 no matter which base you use for entropy, which should help you understand why the base doesn't matter in this case.

You will encounter entropy in a few places in machine learning and signal processing, so this equation is a good one to get your head around. Fortunately, you don't need to derive Euler's Rule or the Thâbit ibn Kurrah Rule to use entropy for uncertainty sampling. The intuition that 0.3679 (or a number near it) contributes most to entropy is fairly simple:

- If the probability is 1.0, the model is completely predictable and has no entropy.
- If the probability is 0.0, that data point provides no contribution to entropy, as it is never going to happen.
- Therefore, some number between 0.0 and 1.0 is optimal for entropy on a per-prediction basis.

But 0.3679 is optimal only for individual probabilities. By using 0.3679 of the probability for one label, you are leaving only 0.6431 for every other label. So the highest entropy for the entire probability distribution, not individual values alone, will always occur when each probability is identical and equal to one divided by the number of labels.

3.3 Identifying when different types of models are confused

You are most likely using neural models in machine learning, but there are many different architectures for neural models and many other popular types of supervised machine learning algorithms. Almost every machine learning library or service will return some form of scores for the algorithms in them, and these scores can be used for uncertainty sampling. In some cases, you will be able to use the scores directly; in other cases, you will have to convert the scores to probability distributions using something like softmax.

Even if you are using only predictive models from neural networks or the default settings on common machine learning libraries and services, it is useful to understand the full range of algorithms and how uncertainty is defined in different kinds of machine learning models. Some are much different from the interpretations that we make from neural network models, but not necessarily any better or worse, so it will help you appreciate the strengths and weaknesses of different common approaches. The strategies for

determining uncertainty for different types of machine learning algorithms are summarized in figure 3.4 and expanded on in more detail in this section.

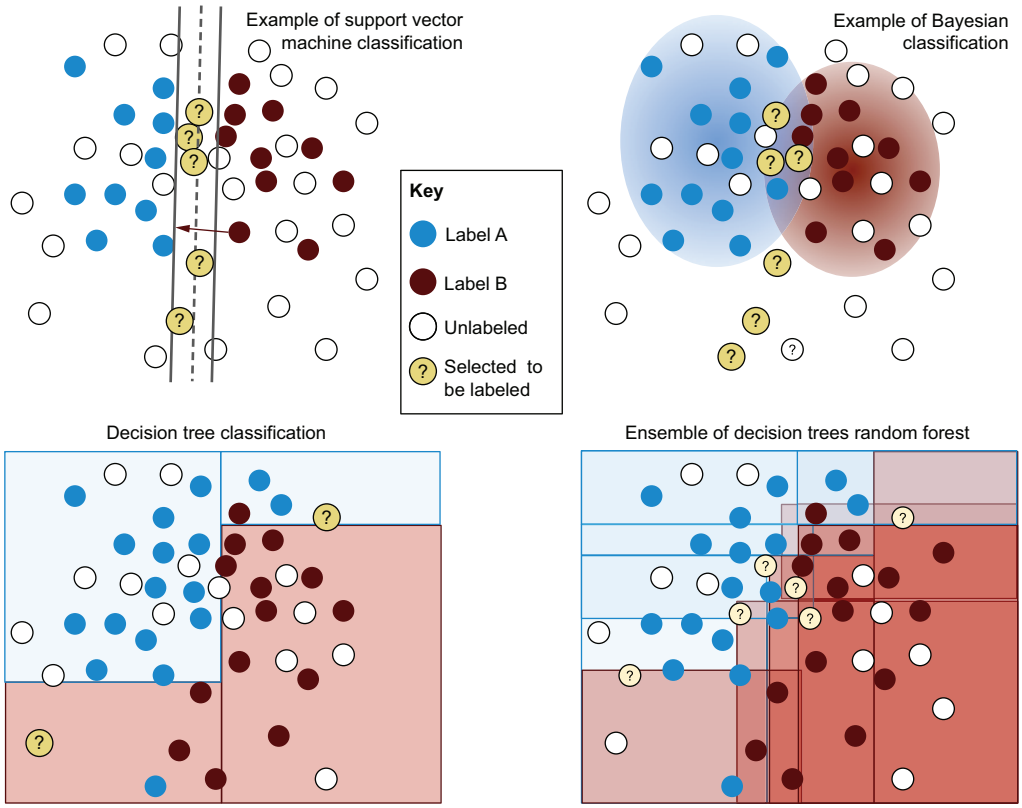


Figure 3.4 Uncertainty sampling from different supervised machine learning algorithms.

Top left: **The decision boundary from a support vector machine (SVM). A discriminative learner, like a neural model, attempts to find a way to divide the data optimally. Unlike neural classifiers, SVMs are also trying to maximize the width of the boundary. This is how an SVM decides which of multiple possible central lines is the best division: it has the widest boundary. Note that the distance from the divider (the *hyperplane* for SVMs) is from the far side of the divider, not the middle line.**

Top right: **A potential Bayesian model. This model is a *generative* supervised learning model, trying to model the distribution of each label rather than model the boundary between them. The confidence on a per-label basis can be read directly as a probability of being that label.**

Bottom left: **The division that a decision tree might provide, dividing and recursively subdividing the data one feature at time. The confidence is defined by the percentage of a label in the final bucket (leaf), The bottom-left leaf, for example, has one Label A and three Label Bs, so a prediction in that leaf would be 25% confidence in Label A and 75% confidence in Label B. Decision trees are sensitive to how far you let them divide—they could keep dividing to leaves of one item—so probabilities tend not to be reliable.**

Bottom right: **An ensemble of decision trees, of which the most well-known variant is a random forest. Multiple decision trees are trained. The different trees are usually achieved by training on different subsets of the data and/or features. The confidence in a label can be the percentage of times an item was predicted across all models or the average confidence across all predictions.**

3.3.1 Uncertainty sampling with logistic regression and MaxEnt models

For interpreting model confidence, you can treat logistic regression and MaxEnt (maximum entropy) models the same as neural models. There is little difference (sometimes none) between a logistic regression model, a MaxEnt model, and a single-layer neural model. Therefore, you can apply uncertainty sampling in the same way that you do for neural models: you might get softmax outputs, or you might get scores to which you can apply softmax. The same caveats apply: it is not the job of a logistic regression or MaxEnt model to calculate the confidence of a model accurately, as the model is trying to distinguish optimally between the labels, so you may want to experiment with different bases/temperatures for softmax if that is how you are generating your probability distribution.

3.3.2 Uncertainty sampling with SVMs

Support vector machines (SVMs) represent another type of discriminative learning. Like neural models, they are attempting to find a way to divide the data optimally. Unlike neural classifiers, SVMs are also trying to maximize the width of the boundary and decide which of the multiple possible divisions is the right one. The optimal boundary is defined as the widest one—more specifically, the one that optimally models the greatest distance between a label and the far side of the dividing boundary. You can see an example of SVMs in figure 3.5. The support vectors themselves are the data points that define the boundaries.

SVMs also differ in how they model more complicated distributions. Neural networks use hidden layers to discover boundaries between labels that are more complicated than simple linear divisions. Two hidden layers are enough to define any

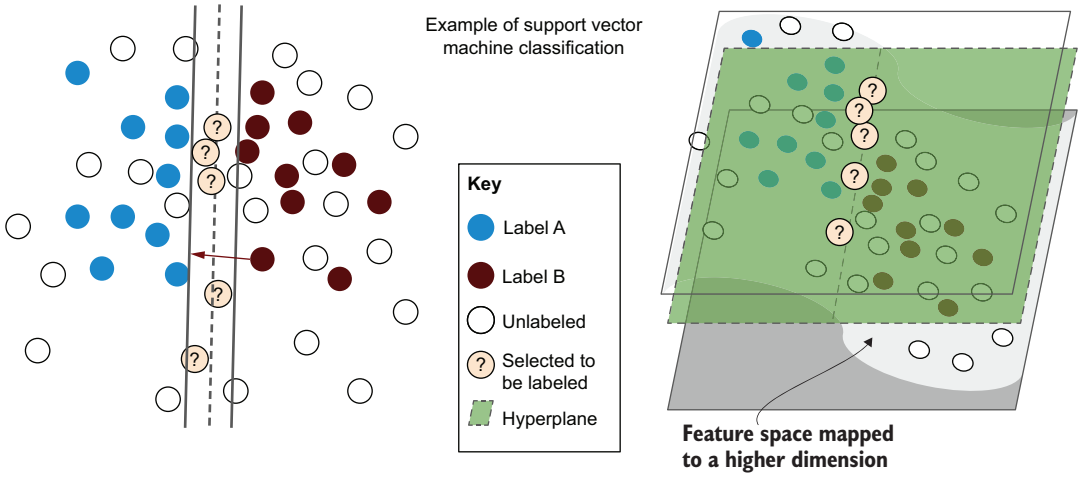


Figure 3.5 SVM projecting our example 2D dataset (top) into 3D (bottom) so that a linear plane can separate the two sets of labels: Label A is above the plane, and Label B is below the plane. The sampled items are the least distance from the plane. If you want to learn from some of the important early active learning literature, you need to understand how SVMs work at this high level.

function. SVMs more or less do the same thing, but with predefined functions that map the data into higher dimensions. In figure 3.5, our 2D example data is projected into a third dimension that raises items on one side of that function and lowers them on the other. With the projection into a higher dimension, the data is linearly separable, and a plane divides the two labels.

It is many orders of magnitude more efficient to train a model when you've predefined the type of function (as in SVMs) rather than let your model find the function itself among all possible alternatives (as in neural models). The chance of predefining the correct function type is low, however, and the cost of hardware is coming down while speed is going up, so SVMs are rarely used today, compared with their earlier popularity.

3.3.3 Uncertainty sampling with Bayesian models

Bayesian models are generative supervised learning models, which means that they are trying to model the distribution of each label and the underlying samples rather than model the boundary between the labels. The advantage of Bayesian models is that you can read the probabilities straight off the model:

$$P_\theta(x|y) = \frac{P_\theta(y|x)P_\theta(x)}{P_\theta(y)}$$

You don't need a separate step or specific activation function to convert arbitrary scores to a probability distribution; the model is explicitly calculating the probability of an item's having a label. Therefore, confidence on a per-label basis can be read directly as a probability of that label.

Because they are not trying to model the differences between labels, Bayesian models tend not to be able to capture more complicated decision boundaries without a lot more fine-tuning. The Naive Bayes algorithm gets the *Naive* part of its name from not being able to model linear relationships between features, let alone more complicated ones, although it can be retrained almost instantly with new training data, which is appealing for human-in-the-loop systems.

Bayesian models also have to make assumptions about data distributions, such as real values falling within a normal distribution, which may not necessarily hold up in your actual data. These assumptions can skew the probabilities away from the true values if you are not careful. They will still tend to be better than probabilities from discriminative models, but you can trust them blindly without understanding their assumptions about the data.

Therefore, although Bayesian models aren't always as likely to get the same accuracy as discriminative models, they typically produce more reliable confidence scores, so they can be used directly in active learning. If you trust your confidence score, for example, you can sample based on that score: sample 90% items with 0.9 uncertainty, sample 10% of items with 0.1 uncertainty, and so on. Beyond simple labeling tasks,

however, when people talk about Bayesian methods for active learning, they typically mean predictions over ensembles of discriminative models, which is covered in section 3.4 later in this chapter.

3.3.4 Uncertainty sampling with decision trees and random forests

Decision trees are discriminative learners that divide the data one feature at a time, recursively subdividing the data into buckets until the final bucket—the leaves—has only one set of labels. The trees are often stopped early (*pruned*) so that the leaves ultimately have some diversity of labels and the models don't overfit the data. Figure 3.4, earlier in this chapter, shows an example.

The confidence is defined by the percentage of a label in the leaf for that prediction. The bottom-left leaf in figure 3.4, for example, has one Label A and three Label Bs, so a prediction in that leaf would be 25% confidence in Label A and 75% confidence in Label B.

Decision trees are sensitive to how far you let them divide; they could keep dividing to leaves of one item. By contrast, if they are not deep enough, each prediction will contain a lot of noise, and the bucket will be large, with relatively distant training items in the same bucket erroneously contributing the confidence. So probabilities tend not to be reliable.

The confidence of single decision trees are rarely trusted for this reason, and they are not recommended for uncertainty sampling. They can be useful for other active learning strategies, as we will cover later, but for any active learning involving decision trees, I recommended that you use multiple trees and combine the results.

Random forests are the best-known ensemble of decision trees. In machine learning, an *ensemble* means a collection of machine learning models that are combined to make a prediction, which we cover more in section 3.4.

For a random forest, multiple different decision trees are trained, with the goal of getting slightly different predictions from each one. The different trees are usually achieved by training on different subsets of the data and/or features. The confidence in a label can be the percentage of times an item was predicted across all models or the average confidence across all predictions.

As figure 3.4 shows with the combination of four decision trees in the bottom-right diagram, the decision boundary between the two labels is starting to become more gradual as you average across multiple predictions. Therefore, random forests make a good, useful approximation confidence along the boundary between two labels. Decision trees are fast to train, so there is little reason not to train many trees in a random forest if they are your algorithm of choice for active learning.

3.4 Measuring uncertainty across multiple predictions

Sometimes, you have multiple models built from your data. You may already be experimenting with different types of models or hyperparameters and want to combine the predictions into a single uncertainty score. If not, you may want to experiment with a

few different models on your data to look at the variance. Even if you are not using multiple models for your data, looking at the variation in predictions from different models will give you an intuition for how stable your model is today.

3.4.1 Uncertainty sampling with ensemble models

Similar to how a random forest is an ensemble of one type of supervised learning algorithm, you can use multiple types of algorithms to determine uncertainty and aggregate across them. Figure 3.6 shows an example. Different classifiers have confidence scores that are unlikely to be directly compatible because of the different types of statistics used.

The simplest way to combine multiple classifiers is to rank-order the items by their uncertainty score for each classifier, give each item a new score based on its rank order, and then combine those rank scores into one master rank of uncertainty.

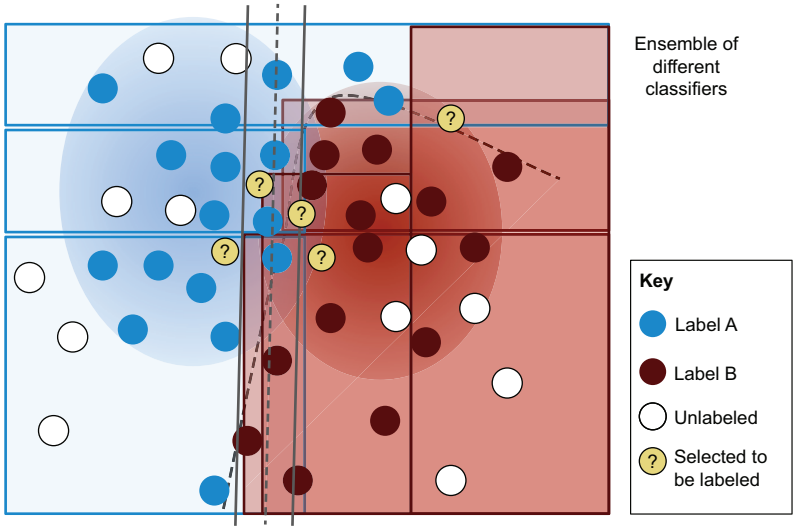


Figure 3.6 An ensemble model that combines predictions from different types of machine learning algorithms: neural models, SVMs, Bayesian models, and decision trees (decision forest). The predictions can be combined in various ways (max, average, and so on) to find the joint uncertainty of each unlabeled item.

You can calculate uncertainty by how often different models agree on the label of an item. The items with the most disagreement are the ones to sample. You can also take the probability distributions of the predictions into account. You can combine the predictions from different models in multiple ways:

- Lowest maximum confidence across all models
- Difference between minimum and maximum confidence across models
- Ratio between minimum and maximum confidence across models
- Entropy across all confidences in all models
- Average confidence across all models

You probably noticed that the first four methods are the same algorithms we used for uncertainty sampling within a single prediction, but in this case across multiple predictions. So you should already be able to implement these methods.

3.4.2 Query by Committee and dropouts

Within active learning, the ensemble-based approach is sometimes known as *Query by Committee*, especially when only one type of machine learning algorithm is used for the ensemble. You could try the ensemble approach with neural models: train a model multiple times and look at the agreement on the unlabeled data across the predictions from each neural model. If you're already retraining your model multiple times to tune hyperparameters, you might as well take advantage of the different predictions to help with active learning.

Following the random forest method, you could try retraining your models with different subsets of items or features to force diversity in the types of models that are built. This approach will prevent one feature (or a small number of features) from dominating the final uncertainty score.

One recently popular method for neural models uses dropouts. You are probably familiar with using dropouts when training a model: you remove/ignore some random percentage of neurons/connections while training the model to avoid overfitting your model to any specific neuron.

You can apply the dropout strategy to predictions: get a prediction for an item multiple times, dropping out a different random selection of neurons/connections each time. This approach results in multiple confidences for an item, and you can use these confidences with the ensemble evaluation methods to sample the right items, as shown in figure 3.7.

You will see more examples throughout the book of using the neural architecture itself to help with active learning. Chapter 4, which covers diversity sampling, begins with a similar example that uses model activation to detect outliers, and many of the advanced techniques later in the book do the same thing.

It is an exciting time to be working in human-in-the-loop machine learning. You get to work with the newest architectures for machine learning algorithms *and* think about how they relate to human–computer interaction.

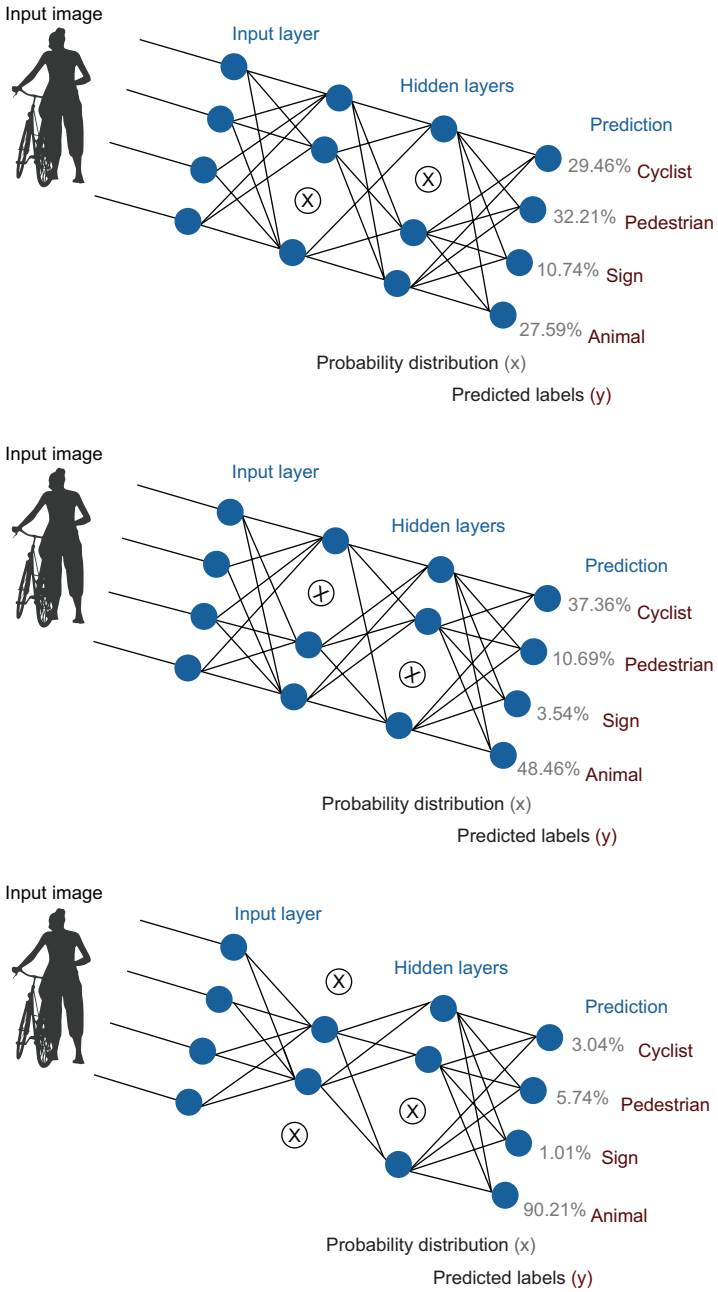


Figure 3.7 Applying dropout to a model to get multiple predictions for a single item. In each prediction, a random set of neurons is dropped (ignored), resulting in different confidences and (possibly) different predicted labels. Then the uncertainty can be calculated as the variation across all predictions: the higher the disagreement, the more uncertainty. This approach to getting multiple predictions from a single model is known as *Monte Carlo dropouts*.

3.4.3 The difference between aleatoric and epistemic uncertainty

The terms *aleatoric uncertainty* and *epistemic uncertainty*, from the philosophy literature, are popular among machine learning scientists who have never read the philosophy literature. In the machine learning literature, the terms typically refer to the methods used. *Epistemic uncertainty* is uncertainty within a single model's predictions, and *aleatoric uncertainty* is uncertainty across multiple predictions (especially Monte Carlo dropouts in the recent literature). *Aleatoric* historically meant inherent randomness, and *epistemic* meant lack of knowledge, but these definitions are meaningful only in machine learning contexts in which no new data can be annotated, which is rare outside academic research.

Therefore, when reading machine learning literature, assume that the researchers are talking only about the methods used to calculate uncertainty, not the deeper philosophical meanings. Figure 3.8 illustrates the differences.

Figure 3.8 shows how multiple predictions allows you to predict uncertainty in terms of variance from multiple decision boundaries in addition to distance from a single decision boundary. For a neural model, the variation in distance from the decision boundary can be calculated as the variation in the labels predicted, the variation in any of the uncertainty sampling metrics covered in section 3.2, or variation across the entire probability distribution for each prediction.

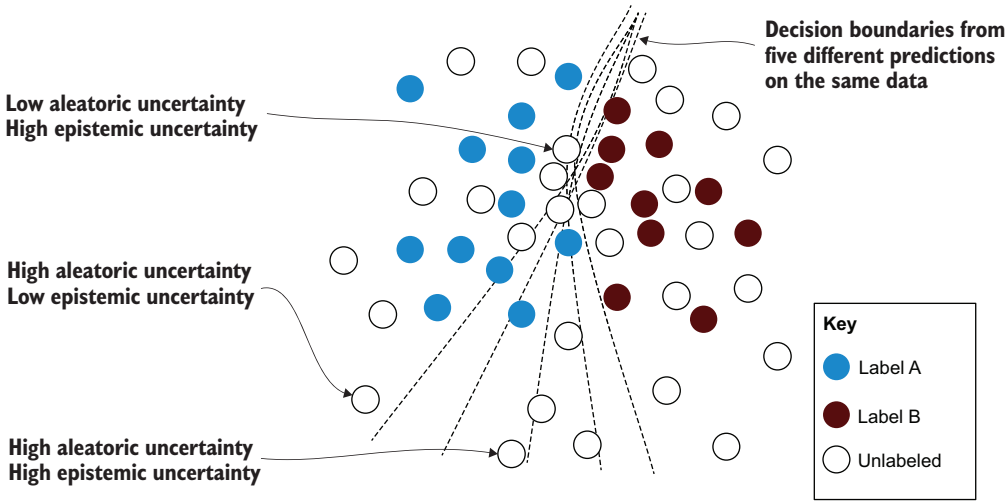


Figure 3.8 Differences between aleatoric and epistemic uncertainty, according to the definitions that are most widely used in machine learning literature. The first highlighted item is near the decision boundary of all five predictions, so it has high epistemic uncertainty, but the decision boundaries are clustered together, so it has low aleatoric uncertainty. The second highlighted item has low epistemic uncertainty because it is not near most decision boundaries, but its distance from the decision boundaries has a large amount of variation, so it has high aleatoric uncertainty. The final item is near the average decision boundary and has great variance in the distance between all boundaries, so it has high uncertainty for both types.

See section 3.8 for further reading on starting places, as this area of research is an active one. The literature on aleatoric uncertainty tends to focus on the optimal types of ensembles or dropouts, and the literature on epistemic uncertainty tends to focus on getting more accurate probability distributions from within a single model.

3.4.4 Multilabeled and continuous value classification

If your task is multilabeled, allowing multiple correct labels for each item, you can calculate uncertainty by using the same aggregation methods as for ensembles. You can treat each label as though it is a binary classifier. Then you can decide whether you want to average the uncertainty, take the maximum uncertainty, or use one of the other aggregation techniques covered earlier in this chapter.

When you treat each label as a binary classifier, there is no difference among the types of uncertainty sampling algorithms (least confidence, margin of confidence, and so on), but you might try the ensemble methods in this section *in addition* to aggregating across the different labels. You can train multiple models on your data and then aggregate the prediction for each label for each item, for example. This approach will give you different uncertainty values for each label for an item, and you can experiment with the right methods to aggregate the uncertainty per label in addition to across labels for each item.

For continuous values, such as a regression model that predicts a real value instead of a label, your model might not give you confidence scores in the prediction. You can apply ensemble methods and look at the variation to calculate the uncertainty in these cases. In fact, Monte Carlo dropouts were first used to estimate uncertainty in regression models in which no new data needed to be labeled. In that controlled environment, you could argue that *epistemic uncertainty* is the right term.

Chapter 6 covers the application of active learning to many use cases, and the section on object detection goes into more detail about uncertainty in regression. Chapter 10 has a section devoted to evaluating human accuracy for continuous tasks that may also be relevant to your task. I recommend that you read those two chapters for more details about working with models that predict continuous values.

3.5 Selecting the right number of items for human review

Uncertainty sampling is an iterative process. You select some number of items for human review, retrain your model, and then repeat the process. Recall from chapter 1 the potential downside of sampling for uncertainty without also sampling for diversity, as shown in figure 3.9.

The most uncertain items here are all near one another. In a real example, thousands of examples might be clustered together, and it would not be necessary to sample them all. No matter where the item is sampled from, you can't be entirely sure what the influence on the model will be until a human has provided a label and the model is retrained.

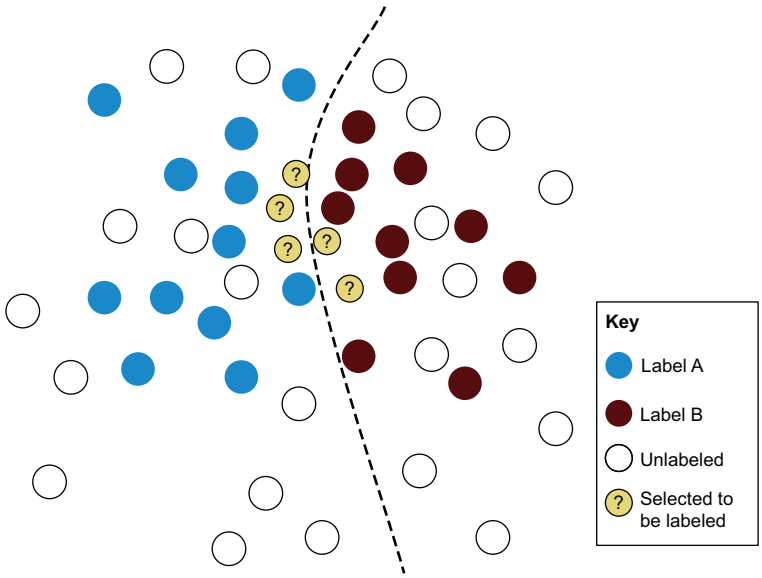


Figure 3.9 A selection of uncertain items that are all from the same region of the feature space, and therefore lack diversity

Retraining a model can take a long time, however, and it can be a waste of time to ask the human annotators to wait during that period. Two competing forces are at work:

- Minimizing the sample size will ensure that the most benefit is gained from each data point at each iteration.
- Maximizing the sample size will ensure that more items get labeled sooner and the model needs to be retrained less often.

As you saw in chapter 2, there was low diversity in the early iterations of your model, but this situation self-corrected in later iterations as the model was retrained. The decision ultimately comes down to a business process. In recent work in translation, we wanted our models to adapt in a few seconds so that they seemed to be responsive in real time to our translators while they worked. I've also seen companies being happy with about one iteration per year to adapt to new data.

3.5.1 Budget-constrained uncertainty sampling

If you have a fixed budget for labels, you should try to get as many iterations as possible. The number of possible iterations will depend on whether you are compensating annotators per label (as with many crowdsourced worker models) or per hour (as with many expert human models).

If your budget is per label, meaning that you are paying a fixed price per label no matter how long the gap is between getting those labels, it is best to optimize for the maximum number of iterations possible. People do tend to get bored waiting for their models to train. When retraining a model takes more than a few days, I've seen people

max out at about 10 iterations and plan accordingly. There's no particular reason to choose 10: it's an intuitive number of iterations to monitor for changes in accuracy.

If your budget is per-hour, meaning that you have a set number of people labeling a set number of hours per day, it is best to optimize for always having data available to label. Have annotators gradually work through the rank order of unlabeled items by uncertainty, and retrain a model at regular intervals, subbing out an old uncertainty ranking for a new one whenever a new model is ready. If you are using uncertainty sampling and want to avoid oversampling from only one part of the problem space, you should replace models regularly. Realistically, if people are working full time to label data for you, you owe them the respect of implementing multiple active learning sampling strategies from this book and sampling from all those strategies, so that those people feel that they are contributing the greatest value possible. You are also less likely to introduce bias that could result from implementing only one of the algorithms, so both humans and machines score a win. We will return to strategies for different kinds of annotation workforces in chapter 7.

3.5.2 Time-constrained uncertainty sampling

If you are time-constrained and need to get an updated model out quickly, you should consider strategies to retrain the models as quickly as possible, as in chapter 2. The quickest way is to use simple models. A model with only one or two layers (or, better yet, a Naive Bayes model) can be retrained incredibly quickly, allowing you to iterate quickly. Further, there is some evidence that uncertainty sampling from a simpler model can be as effective as sampling from a more complicated model. Remember that we're looking for the most confusion, not the most accuracy. Provided that a simple model is the most confused about the same items as a more complicated model, both models will sample the same items.

A more advanced way is to retrain only the final layer(s) of a much larger model. You can retrain your model quickly by retraining only the last layer with new data, compared with retraining the whole model. This process can take a matter of seconds instead of weeks. The retrained model will not necessarily be as accurate, but it may be close. As with choosing a simpler model, this small loss in accuracy may not matter if the goal is to look for more uncertainty. The faster iteration may even result in a more accurate model than if you'd waited a long time to retrain the entire model with fewer iterations.

One advanced method is to have the best of both worlds: use methods to discover which parameters are the most important to retrain across your entire model, and retrain only them. This approach can give you the same accuracy as retraining the entire model, but in a fraction of the time.

Another advanced method that can be easier to implement is to have two models: an incremental model that is updated immediately with every new training item and a second model that is retrained from scratch at regular intervals. One of the example implementations in chapter 12 uses this architecture.

3.5.3 When do I stop if I'm not time- or budget-constrained?

Lucky you! You should stop when your model stops getting more accurate. If you have tried many strategies for uncertainty sampling and are not getting any more gains after a certain accuracy is reached, this condition is a good signal to stop and think about other active learning and/or algorithmic strategies if your desired accuracy goal hasn't been met.

You will ultimately see diminishing returns as you label more data; no matter what strategy you use, the learning rate will decrease as you add more data. Even if the rate hasn't plateaued, you should be able to run a cost-benefit analysis of the accuracy you are getting per label versus the cost of those labels.

3.6 Evaluating the success of active learning

Always evaluate uncertainty sampling on a randomly selected, held-out test set. If the test data is selected randomly from your training data after each iteration, you won't know what your actual accuracy is. In fact, your accuracy is likely to appear to be lower than it is. By choosing items that are hard to classify, you are probably oversampling inherently ambiguous items. If you are testing more on inherently ambiguous items, you are more likely to see errors. (We covered this topic in chapter 2, but it is worth repeating here.) Therefore, don't fall into the trap of forgetting to sample randomly in addition to using uncertainty sampling: you won't know whether your model is improving!

3.6.1 Do I need new test data?

If you already have test data set aside, and you know that the unlabeled data is from more or less the same distribution as your training data, you do not need additional test data. You can keep testing on the same data.

If you know that the test data has a different distribution from your original training data, or if you are unsure, you should collect additional labels through random selection of unlabeled items and add them to your test set or create a second, separate test set.

TIP Create your new test set before your first iteration of uncertainty sampling.

As soon as you have removed some unlabeled items from the pool via uncertainty sampling, that pool is no longer a random selection. That pool is now biased toward *confidently* predicted items, so a random selection from this pool is likely to return erroneously high accuracy if it is used as a test set.

Keep your test set separate throughout all iterations, and do not allow its items to be part of any sampling strategy. If you forget to do this until several iterations in, and your random sample includes items that were selected by uncertainty sampling, you will need to go back to the first iteration. You can't simply remove those test items from the training data going forward, as they were trained on and contributed to selections in the interim uncertainty sampling strategies.

It is also a good idea to see how well your uncertainty sampling technique is performing next to a baseline of random sampling. If you aren't more accurate than random sampling, you should reconsider your strategy! Choose randomly selected items for which you know the comparison will be statistically significant: often, a few hundred items are sufficient. Unlike the evaluation data for your entire model, these items can be added to your training data in the next iteration, as you are comparing the sampling strategy at each step, given what is remaining to be labeled.

Finally, you may want to include a random sample of items along with the ones chosen by uncertainty sampling. If you are not going to implement some of the diversity sampling methods in chapter 4, random sampling will give you the most basic form of diversity sampling and ensure that every data point has a chance of getting human review.

3.6.2 Do I need new validation data?

You should also consider up to four validation sets at each iteration, with data drawn from

- The same distribution as the test set
- The remaining unlabeled items in each iteration
- The same distribution as the newly sampled items in each iteration
- The same distribution as the total training set in each iteration

If you are tuning the parameters of your model after each addition of data, you will use a validation set to evaluate the accuracy. If you tune a model on the test set, you won't know whether your model has truly generalized or you have simply found a set of parameters that happen to work well with that specific evaluation data.

A validation set will let you tune the accuracy of the model without looking at the test set. Typically, you will have a validation set from the outset. As with your test set, you don't need to update/replace it if you think that the unlabeled items come from the same distribution as your initial training data. Otherwise, you should update your validation data before the first iteration of your uncertainty sampling, as with your test data.

You may want to use a second validation set to test how well your active learning strategy is doing within each iteration. After you start active learning iterations, the remaining unlabeled items will no longer be a random sample, so this distribution will not be the same as your existing test set and validation set. This dataset acts as a baseline for each iteration. Is uncertainty sampling still giving you better results than selecting from random among the remaining items? Because this dataset set is useful for only one iteration, it is fine to add these items to the training data at the end of each iteration; these labels aren't human labels that get discarded.

If you want to evaluate the accuracy of the human-labels created in each iteration, you should do this on a third validation data set drawn from the same distribution as the newly sampled data. Your newly sampled data may be inherently easier or harder for humans to label, so you need to evaluate human accuracy on that same distribution.

Finally, you should consider a fourth validation set drawn randomly from the training data at each iteration. This validation data can be used to ensure that the model is not overfitting the training data, which a lot of machine learning libraries will do by default. If your validation data and training data are not from the same distribution, it will be hard to estimate how much you are overfitting, so having a separate validation set to check for overfitting is a good idea.

The downside is the human-labeling cost for up to four validation data sets. In industry, I see people using the wrong validation dataset more often than not, typically letting one validation set be used in all cases. The most common reason is that people want to put as many labeled items as possible into their training data to make that model more accurate sooner. That's also the goal of active learning, of course, but without the right validation data, you won't know what strategic direction to take next to get to greater accuracy.

3.7 Uncertainty sampling cheat sheet

Our example data in this text has only two labels. Uncertainty sampling algorithms will return the same samples with two labels. Figure 3.10 shows an example of target areas for the different algorithms when there are three labels. The figure shows that margin of confidence and ratio sample some items that have only pairwise confusion, which reflects the fact that the algorithms target only the two most likely labels. By contrast, entropy maximizes for confusion among all labels, which is why the highest concentration is between all three labels.

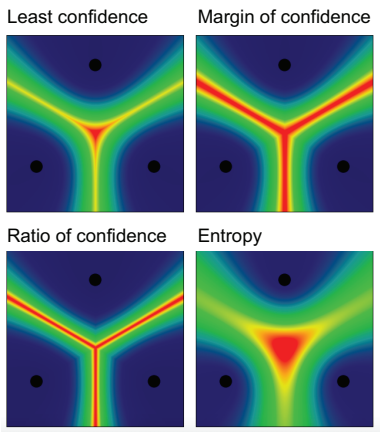


Figure 3.10 A heat map of the four main uncertainty sampling algorithms and the areas that they sample for a three-label problem. In this example, each dot is an item with a different label, and the heat of each pixel is the uncertainty. The hottest (most uncertain) pixels are the lightest pixels (the red pixels, if you're viewing in color). Top left is least confidence sampling, top right is margin of confidence sampling, bottom left is ratio sampling, and bottom right is entropy-based sampling. The main takeaway is that margin of confidence and ratio sample some items that have only pairwise confusion and entropy maximizes for confusion among all labels.

Notice that the difference between the methods becomes even more extreme with more labels. Figure 3.11 compares configurations to highlight the differences among the methods.

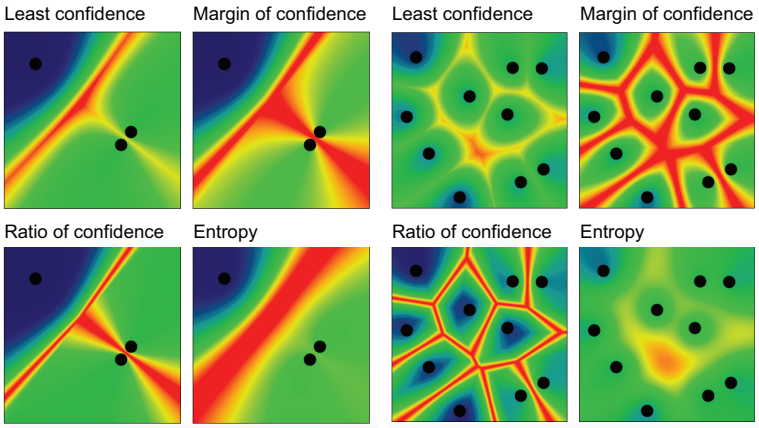


Figure 3.11 A comparison of methods. The four left images show that a lot of the uncertainty space for margin of confidence and ratio is between two of the labels, which is ignored entirely by entropy because it is not ambiguous for the third label. The four right images show that especially in more complicated tasks, the items that will be sampled by different uncertainty sampling algorithms will be different.[1]

TIP You can play around with interactive versions of figure 3.10 and figure 3.11 at http://robertmunro.com/uncertainty_sampling_example.html. The source code for the interactive example has implementations of the uncertainty sampling algorithms in JavaScript, but you're more likely to want the Python examples in the code repository associated with this chapter in PyTorch and in NumPy.

Figure 3.12 summarizes the four uncertainty sampling algorithms that you've implemented in this chapter.

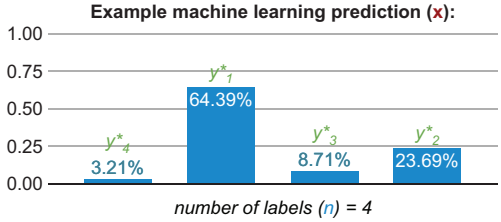
[1] Thank you, Adrian Calma, for suggesting the left images as a great way to highlight the differences.

Uncertainty sampling cheat sheet

When a supervised machine learning model makes a prediction, it often gives a confidence in that prediction. If the model is uncertain (low confidence), human feedback can help. Getting human feedback when a model is uncertain is a type of *active learning* known as *uncertainty sampling*.

This cheat sheet has four common ways to calculate uncertainty, with examples, equations, and Python code.

Example machine learning prediction (x):

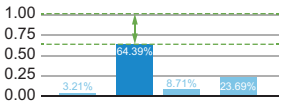


The predictions are a probability distribution (x), meaning that every prediction is between 0 and 1 and the predictions add to 1. y^*_1 is the most confident, y^*_2 is the second-most-confident, and so on for n predicted labels.

This example can be expressed as a PyTorch tensor:

```
prob = torch.tensor ([0.0321, 0.6439, 0.0871,
0.2369]).
```

Least confidence: Difference between the most confident prediction and 100% confidence

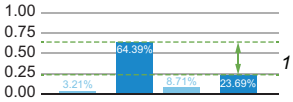


$$\frac{n\,(1-P_\theta(y^*_1 \mid x))}{n-1}$$

```
most_conf = torch.max(prob)
num_labels = prob.numel ()
numerator = (num_labels * (1 - most_conf))
denominator = (num_labels - 1)

least_conf = numerator/denominator
```

Margin of confidence: Difference between the two most confident predictions

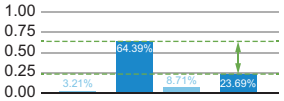


$$1 - (P_\theta(y^*_1 \mid x) - P_\theta(y^*_2 \mid x))$$

```
prob, _  = torch.sort (prob, descending=True)
difference = (prob.data [0] - prob.data[1])

margin_conf = 1 - difference
```

Ratio of confidence: Ratio between the two most confident predictions

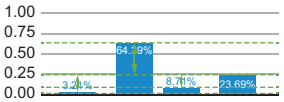


$$\frac{P_\theta(y^*_2 \mid x)}{P_\theta(y^*_1 \mid x)}$$

```
prob, _  = torch.sort (prob, descending=True)

ratio_conf = (prob.data [1] / prob.data [0])
```

Entropy: Difference between all predictions, as defined by information theory



$$\frac{-\sum_y P_\theta(y \mid x)\,\log_2 P_\theta(y \mid x)}{\log_2(n)}$$

```
prbslogs  = prob * torch.log2(prob)
numerator = 0 - torch.sum(prbslogs)
denominator = torch.log2(prob.numel ())

entropy = numerator / denominator
```

Robert (Munro) Monarch. *Human-in-the-Loop Machine Learning*, Manning Publications. http://bit.ly/huml_book
See the book for more details on each method and for more sophisticated problems like sequence models and semantic segmentation, plus other sampling strategies like diversity sampling. robertmunro.com | @WWRob

Figure 3.12 Uncertainty sampling cheat sheet

3.8 Further reading

Uncertainty sampling has been around for a long time, and a lot of good literature has been written about it. For the most cutting-edge research on uncertainty sampling, look for recent papers that are frequently cited themselves.

Note that most of the papers do not normalize the scores to a [0, 1] range. If you're going to deploy your models for real-world situations, I highly recommend that you normalize the outputs. Even if normalizing the outputs won't change the accuracy, it will make spot checks easier and prevent problems with downstream processing, especially for advanced methods that you will learn in later chapters.

3.8.1 Further reading for least confidence sampling

A good early paper on least confidence is "Reducing labeling effort for structured prediction tasks," by Aron Culotta and Andrew McCallum (http://mng.bz/opYj).

3.8.2 Further reading for margin of confidence sampling

A good early paper on margin of confidence is "Active Hidden Markov Models for Information Extraction," by Tobias Scheffer, Christian Decomain, and Stefan Wrobel (http://mng.bz/nMO8).

3.8.3 Further reading for ratio of confidence sampling

I'm not aware of papers on ratio of confidence, although I've taught the subject in classes on active learning. The relationship between ratio and softmax base/temperature was new when it was presented in this book. As ratio of confidence is similar to margin of confidence, in that both look at the relationship between the two most confident predictions, the literature for margin of confidence should be mostly relevant.

3.8.4 Further reading for entropy-based sampling

A good early paper on entropy-based sampling is "Committee-Based Sampling For Training Probabilistic Classifiers," by Ido Dagan and Sean P. Engelson (http://mng.bz/vzWq).

3.8.5 Further reading for other machine learning models

A foundational paper for uncertainty sampling more generally is "A Sequential Algorithm for Training Text Classifiers," by David D. Lewis and William A. Gale (http://mng.bz/4ZQg). This paper uses a Bayesian classifier. If you look at highly cited texts from the following decade, you will find that SVMs and linear models are common. For the reasons mentioned in this chapter, I do not recommend that you try to implement uncertainty sampling with decision trees.

3.8.6 Further reading for ensemble-based uncertainty sampling

The Dagan and Engelson paper (section 3.8.4) covers the use case of multiple classifiers (Query by Committee), so it is a good starting point for ensemble models. For

more recent work focused on neural models, including dropouts and Bayesian approaches to better uncertainty estimates, a good entry point is "Deep Bayesian Active Learning for Natural Language Processing: Results of a Large-Scale Empirical Study," by Zachary C. Lipton and Aditya Siddhant (http://mng.bz/Qmae).

You will see random dropouts called Monte Carlo dropouts and Bayesian (deep) active learning in academic literature. Regardless of the name, the strategy is still randomly selecting neurons/connections to ignore during prediction. The term *Monte Carlo* was a joke made by the physicist who invented the term. The term *Bayesian* comes from the fact that if you squint at the variation, it looks like a Gaussian distribution; it is not an actual Bayesian classifier. On the positive side of understanding the terminology, by passing one extra parameter to your model during prediction, you can impress your friends by telling them that you just implemented *Monte Carlo dropouts for Bayesian deep active learning*.

Summary

- Four common algorithms are used for uncertainty sampling: least confidence, margin of confidence, ratio of confidence, and entropy. These algorithms can help you understand the different kinds of "known unknowns" in your models.
- You can get different samples from each type of uncertainty sampling algorithm. Understanding why will help you decide which one is the best way to measure uncertainty in your models.
- Different types of scores are output by different supervised machine learning algorithms, including neural models, Bayesian models, SVMs, and decision trees. Understanding each score will help you interpret them for uncertainty.
- Ensemble methods and dropouts can be used to create multiple predictions for the same item. You can calculate uncertainty by looking at variation across the predictions from different models.
- There is a trade-off between getting more annotations within each active learning cycle and getting fewer annotations with more cycles. Understanding the trade-offs will let you pick the right number of cycles and size of each cycle when using uncertainty sampling.
- You may want to create different kinds of validation data to evaluate different parts of your system. Understanding the different types of validation data will let you choose the right one to tune each component.
- The right testing framework will help you calculate the accuracy of your system, ensuring that you are measuring performance increases correctly and not inadvertently biasing your data.

4 Diversity sampling

This chapter covers

- Using outlier detection to sample data that is unknown to your current model
- Using clustering to sample more diverse data before annotation starts
- Using representative sampling to target data most like where your model is deployed
- Improving real-world diversity with stratified sampling and active learning
- Using diversity sampling with different types of machine learning architectures
- Evaluating the success of diversity sampling

In chapter 3, you learned how to identify where your model is uncertain: what your model "knows it doesn't know." In this chapter, you will learn how to identify what's missing from your model: what your model "doesn't know that it doesn't know" or the "unknown unknowns." This problem is a hard one, made even harder because what your model needs to know is often a moving target in a constantly changing world. Just like humans are learning new words, new objects, and new behaviors

every day in response to a changing environment, most machine learning algorithms are deployed in a changing environment.

For example, if we are using machine learning to classify or process human language, we typically expect the applications to adapt to new words and meanings, rather than remain stale and understand the language only up to one historical point in time. We'll explore a couple of use cases in speech recognition and computer vision in the upcoming chapters to illustrate the value of diversity sampling for different kinds of machine learning problems.

Imagine that your job is to build a voice assistant that can be successful for as many users as possible. Your company's leaders expect your machine learning algorithms to have much broader knowledge than any one human. A typical English speaker knows about 40,000 words of English's 200,000-word vocabulary, which is only 20% of the language, but your model should have closer to 100% coverage. You have a lot of unlabeled recordings that you can label, but some of the words that people use are rare. If you randomly sampled the recordings, you would miss the rare words. So you need to explicitly try to get training data covering as many different words as possible. You might also want to see what words are most commonly used when people speak to their voice assistants and sample more of those words.

You are also worried about demographic diversity. The recordings are predominantly from one gender and from people living in a small number of locations, so resulting models are likely to be more accurate for that gender and for only some accents. You want to sample as fairly as possible from different demographics to make the model equally accurate for all demographics.

Finally, many people don't speak English and would like a voice assistant, but you have little non-English data. You may have to be open and honest about this limitation in diversity.

This problem is harder than simply knowing when your model is confused, so the solutions for diversity sampling are themselves more algorithmically diverse than those for uncertainty sampling.

4.1 Knowing what you don't know: Identifying gaps in your model's knowledge

We explore four approaches to diversity sampling in this chapter:

- *Model-based outlier sampling*—Determining which items are unknown to the model in its current state (compared with uncertain, as in chapter 3). In our voice assistant example, model-based outlier sampling would help identify words that our voice assistant hasn't encountered before.
- *Cluster-based sampling*—Using statistical methods independent of your model to find a diverse selection of items to label. In our example, cluster-based sampling would help identify natural trends in the data so we can make sure that we don't miss any rare but meaningful trends.
- *Representative sampling*—Finding a sample of unlabeled items that look most like your target domain, compared with your training data. In our example, let's

imagine that people used your voice assistant mostly to request songs. Representative sampling, therefore, would target examples of song requests.

- *Sampling for real-world diversity*—Ensuring that a diverse range of real-world entities are in our training data to reduce real-world bias. In our example, this approach could include targeting recordings for as many accents, ages, and genders as possible.

As you learned in the book's introduction, the phrase *uncertainty sampling* is widely used in active learning, but *diversity sampling* goes by different names in different fields, often tackling only part of the problem. You may have seen diversity sampling referred to as stratified sampling, representative sampling, outlier detection, or anomaly detection. A lot of the time, the algorithms that we use for diversity sampling are borrowed from other use cases. Anomaly detection, for example, is primarily used for tasks such as identifying new phenomena in astronomical databases or detecting strange network activity for security.

So as not to confuse the non-active learning use cases and to provide consistency, we'll use the phrase *diversity sampling* in this text. This phrase intentionally invokes diversity in the sense of the demographics of people represented in the data. Although only the fourth kind of diversity sampling that we look at explicitly targets demographic diversity, the other three types correlate with real-world diversity. Chances are that your unlabeled data is biased toward the most privileged demographics: languages from the wealthiest nations, images from the wealthiest economies, videos created by the wealthiest individuals, and other biases that result from power imbalances. If you build models only on randomly sampled raw data, you could amplify that bias. Any method that increases the diversity of the items that you sample for active learning will likely increase the diversity of the people who can benefit from models built from that data.

Even if you are not worried about biases in demographics of people, you probably still want to overcome sample bias in your data. If you are processing images from agriculture and happen to have one type of crop overrepresented in your raw data, you probably want a sampling strategy that will rebalance the data to represent many types of crops. Also, there may be deeper biases related to people. If you have more examples of one type of crop, is that crop more common in wealthier countries, and do you have more photographs because tractors in wealthier countries are more likely to have cameras? Data bias and real-world bias tend to be closely related when we dig deep. Figure 4.1 repeats the example of diversity sampling that you saw in chapter 1.

For uncertainty sampling, you want to see only what is near your current decision boundary or what varies most across multiple predictions—a relatively small and well-defined feature space. For diversity sampling, you want to explore the much larger problem of every corner of the feature space and expand the decision boundary into new parts of that space. Needless to say, the set of algorithms that you can use are more diverse and sometimes more complicated than those used for uncertainty sampling.

You may not need to worry about every data point if you are looking only at academic datasets, but the problem of diversity is much more common in real-world

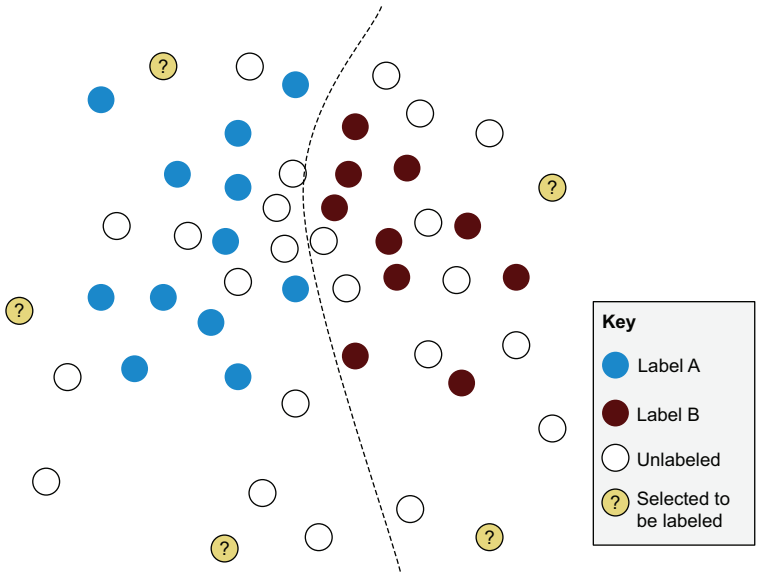


Figure 4.1 Diversity sampling, showing items selected to be labeled that are maximally different from the existing training items and from one another. You want to sample items that are not like the items that are currently in your training data and are also not like one another.

datasets. See the following sidebar for more information about the difference between real-world and academic datasets.

The difference between academic and real-world data labeling

Expert anecdote by Jia Li

It is much harder to deploy machine learning in the real world than for academic research, and the main difference is the data. Real-world data is messy and often hard to access due to institutional hurdles. It is fine to conduct research on clean, unchanging datasets, but when you take those models into the real world, it can be hard to predict how they will perform.

When I was helping to build ImageNet, we didn't have to worry about every possible image class that we might encounter in the real world. We could limit the data to images that were a subset of concepts in the WordNet hierarchy. In the real world, we don't have that luxury. For example, we can't collect large amounts of medical images related to rare diseases. Labeling of such images further requires domain expertise, which poses even more challenges. Real-world systems need both AI technologists and domain experts collaborating closely to inspire research, provide data and analysis, and develop algorithms to solve the problem.

Jia Li is CEO and co-founder of Dawnlight, a health-care company that uses machine learning. She previously led research divisions at Google, Snap, and Yahoo!, and has a PhD from Stanford.

4.1.1 Example data for diversity sampling

In this chapter, we will build on our example from chapter 2 with disaster response messages. Recall from chapter 2 that we wanted to label news headlines as disaster-related or not disaster-related. We implemented a basic outlier detection algorithm in that chapter, which we will now expand on with more sophisticated diversity sampling algorithms. The code is in the same library that you used for chapter 2: https://github.com/rmunro/pytorch_active_learning. The code that we will use in this chapter is in these two files: diversity_sampling.py and active_learning.py.

We will cover many types of diversity sampling strategies in this chapter. For our example data, you can imagine that a machine learning model could be useful for tracking disasters as they are being reported and to distinguish eyewitness reports from secondhand (or thirdhand) information. If you want to deploy this kind of system to track disasters in real time, you want as diverse a set of past training data items as possible. Only one or two news articles about floods might have been reported in your past training data, for example, which could easily have been missed if you chose items at random for humans to label.

You can also imagine new types of disasters, such as disease outbreaks that have a pattern of infection that hasn't been observed before. If people are talking about these new disasters in new ways, you want to ensure that you aren't missing these items and that the items get human labels as quickly as possible.

Finally, you might want to start incorporating new sources of data. If some of the new sources are US English instead of UK English, if they use different slang, or if they are not in English, your model will not be accurate on those new sources of information. You want to make sure that your model can adapt to these new data sources and their stylistic differences as quickly as possible, as it is adapting to new types of information in the text itself.

It is important to reduce bias at every step. If you use your model predictions to find more examples of floods, but your existing model only has data from floods in Australia, you might get more examples from floods in Australia for human review and from no other part of the world, so you would never get away from the initial bias in your model. For that reason, most diversity sampling algorithms are independent of the model we are using.

4.1.2 Interpreting neural models for diversity sampling

For some of the sampling strategies in this chapter, we will need new ways to interpret our models. If you access the raw outputs of a linear activation function in your final layer instead of the softmax output, you can more accurately separate true outliers from items that are the result of conflicting information. An activation function that also includes a negative range, like Leaky ReLU, is ideal; otherwise, you might end up with a lot of zeroed-out scores with no way to determine which is the biggest outlier.

You will learn how to access and interpret the different layers of a PyTorch model in section 4.1.3. But you may not have a say about the architecture of the activation function in the final layer. Softmax may the most accurate activation function for predicting labels precisely because it can ignore the absolute values of its inputs. In these cases, you may still be able to persuade your algorithms team to expose other layers for analysis.

What if I don't control my model architecture?

If you don't have a say on the architecture of the predictive algorithm, you might be able to convince your algorithms team to expose the logits or retrain only the final layer of the model with a Leaky ReLU activation function. Retraining the last layer of the model will be orders of magnitude faster than retraining an entire model. This approach should appeal to someone who is worried about the cost of retraining: they are supporting a new use case for not much extra work with a fun parallel architecture. If you are using Transformer models, the same concept applies, but you would train a new Attention head. (Don't worry if you are not familiar with Transformer models; they are not important for this chapter.)

If you encounter resistance to the idea of retraining the last layer, or if there are technical barriers, your next-best option is to use the second-to-last layer of the model. Regardless, it might be interesting to compare outlier sampling methods on different layers of the model to see what works best with your particular data and model architecture. These kinds of model analytics are some of the most exciting areas of machine learning research today and also allow for transfer learning, which is used in techniques in most of the upcoming chapters.

In this chapter, we will limit ourselves to simple but effective ways to interpret your model. The two scenarios in figure 4.2 interpret the last layer or the second-to-last layer.

The second method, using the second-to-last layer, work bests on deeper networks in which the second-to-last layer more closely resembles the last layer and there are fewer neurons in that layer. More neurons will introduce more random variability that can be more difficult to overcome, statistically.

Regardless of which architecture you use, you are left with a set (vector/tensor) of numbers representing a level of activation at/near the output of your predictive model. For simplicity, we'll refer to either vector as z even though z is typically reserved to mean only the logits from the last layer. We will also use n to indicate the size of that vector (the number of neurons), regardless of whether it happens to be the final layer and therefore also the number of labels or a middle layer.

Low activation means that this item is more likely to be an outlier. Mathematically, an outlier could be any unusual vector, atypically high or atypically low. But when we are interpreting model predictions to find outliers, we are concerned only with the low-activation items—the items that the model has little information about today.

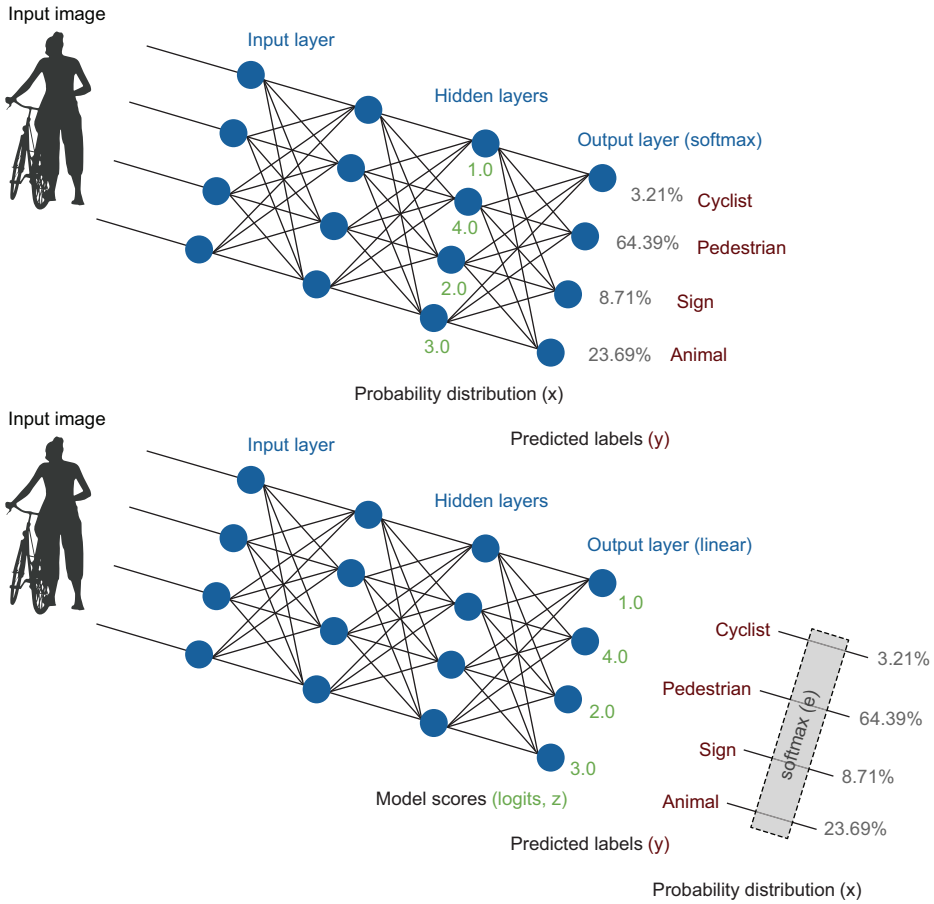


Figure 4.2 Two neural architectures and how you can interpret them for outlier detection. In the top example, you can use the model scores (known as *z* or *logits*), which retain their absolute values before they are normalized via softmax. In the bottom example, you have lost the absolute values in the final layer because of the softmax function, so you can use the activation in the second-to-last layer to determine whether an item is an outlier.

4.1.3 Getting information from hidden layers in PyTorch

To get the z values (logits) of the values from hidden layers in the model, we will need to modify our code so that we can access this information. Fortunately, the code is simple in PyTorch. First, as a reminder, here's the code from chapter 2 that you used for the feed-forward steps in training and for generating the confidences and label predictions in inference:

```
def forward(self, feature_vec):
    # Define how data is passed through the model

    hidden1 = self.linear1(feature_vec).clamp(min=0) # ReLU
    output = self.linear2(hidden1)
    return F.log_softmax(output, dim=1)
```

You can see that the middle layer and outputs are variables (`hidden1` and `output`) that hold the activation from each layer (PyTorch tensors in this case, which will be 1D arrays). So we can simply add a parameter to return all layers, modifying the code accordingly.

Listing 4.1 Allowing our model to return hidden layers in addition to softmax values

```
def forward(self, feature_vec, return_all_layers=False):
    # Define how data is passed through the model and what is returned

    hidden1 = self.linear1(feature_vec).clamp(min=0) # ReLU
    output = self.linear2(hidden1)
    log_softmax = F.log_softmax(output, dim=1)

    if return_all_layers:
        return [hidden1, output, log_softmax]
    else:
        return log_softmax
```

Same as the return function but pulled out into a variable

The only real new line, returning all the layers when return_all_layers=True

That's it! You'll see this modified code in active_learning.py. Now we can use any part of our model to find outliers within the model. Also, we have other ways to query our model's hidden layers.[1] I prefer encoding the option explicitly in the inference function, as with the `forward()` function. We are going to query our model in many ways in future chapters, and this approach makes the simplest code to build on.

Good coding practices for active learning

As a note on good coding practices, you may want to change the `return log_softmax` line in your `forward()` function to also return an array: `return [log_softmax]`. That way, your function is returning the same data type (an array) no matter what parameters are passed to it, which is a better software development practice. The downside is that it's not backward-compatible, so you'll have to change every piece of code that is calling the function. If you're an experienced PyTorch user, you may be accustomed to using a feature in the function that knows when it is in training mode or evaluation mode. This feature can be handy for some common machine learning strategies, such as masking neurons in training but not when predicting. But resist the temptation to use this feature here; it is bad software development in this context because global variables make unit tests harder to write and will make your code harder to read in isolation. Use named parameters like `return_all_layers=True/False`; you want to extend code in the most transparent way possible.

With the addition of code to access all layers of the model in inference, we can use that code to determine outliers. Recall that in chapter 2, you got the log probabilities from your model with this line:

```
log_probs = model(feature_vec)
```

[1] An alternative way to get hidden layers in PyTorch are the `hook()` methods. See the documentation at http://mng.bz/XdzM.

Now you can choose which layer of the model you want to use by calling the function with this line:

```
hidden, logits, log_probs = model(feature_vector, return_all_layers=True)
```

You have the hidden layer, the logits (z), and the log_probabilities of the model for your item.

Recall from chapter 3 and the appendix that our logits (scores from the last layer) lose their absolute values when converted to probability distributions via softmax. Figure 4.3 reproduces some of these examples from the expanded section on softmax in the appendix.

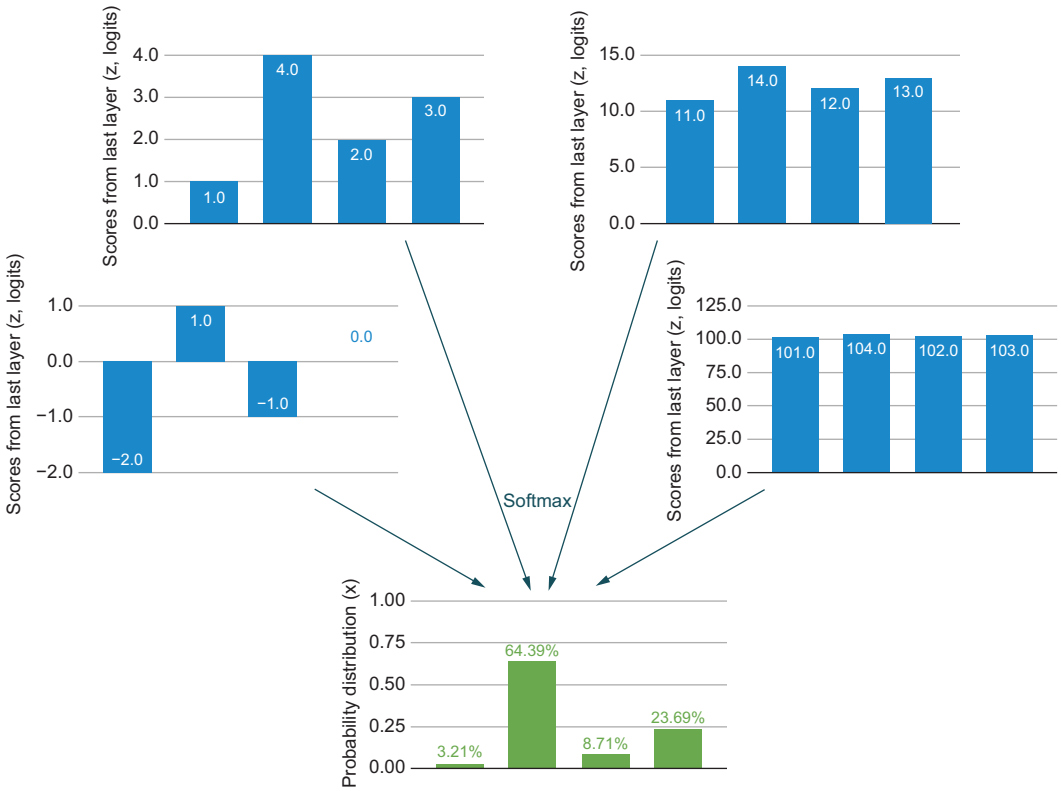


Figure 4.3 Four identical probability distributions that are derived from different inputs via softmax with base e

Our probability distributions, therefore, don't tell us the difference between uncertainty that derives from the lack of information (as in the left example in figure 4.3) and uncertainty due to conflicting but highly confident information (as in the right example in figure 4.3). So it is better to use the logits (scores from the last layer) to differentiate the two types of uncertainty.

Beyond uncertainty, we can find outliers that were certain but wrong. The most valuable unlabeled items to label are those that were incorrectly predicted and far from the decision boundary—that is, items that the current model is predicting confidently but incorrectly. Low activation across all neurons is often a good signal that there is not yet enough training data with the features found in that item.

4.2 Model-based outlier sampling

Now that we can interpret our model, we can query our model to find outliers. A *model outlier* in a neural model is defined as the item with the lowest activation in a given layer. For our final layer, this activation is the logits.

The biggest barrier to choosing the right metric for determining an outlier is knowing the distribution of values from your neurons. You were taught in high school that any data point greater than three standard deviations from the mean is an outlier, but this is true only for normal distributions. Unfortunately, your linear activation functions are not creating normal distributions: they should be bimodally distributed if they are modeling your task accurately. If you've investigated models in the past, you'll also know that some of the neurons might be modeling noise or simple passing through values and can vary even when you train a model twice on identical data. Furthermore, unless you have a simple architecture, you will have different activation functions for different parts of your network, so they will not be directly comparable.

Just like we couldn't trust the absolute values for confidence for uncertainty sampling, we can't trust the absolute values of our neurons to determine outliers. But just like we could trust the ranked order confidence to find the most uncertain predictions, we can trust the ranked order of neuron activation to find the least activated. Rank order is a robust method that lets us avoid determining the actual distribution of activation in every neuron.

Here's a simple example of ranked order for determining how much of an outlier some item is. Let's assume that we have made predictions for 10 items and that these predictions were the result from a neuron, ordered (ranked) from largest to smallest:

```
[2.43, 2.23, 1.74, 1.12, 0.89, 0.44, 0.23, -0.34, -0.36, -0.42]
```

The item with an activation of –0.36 (underlined) is the ninth-lowest of the 10 items, so we can give it an outlier score of 9/10 = 0.9. At either end of the scale, an item with –0.42 activation would have a score of 1.0, and an item with 2.43 activation would have a score of 0. So we can convert this ranked order of activation for each neuron into a scale. The question, then, is what data to use to generate the ranking.

4.2.1 Use validation data to rank activations

We can't use the training data for our rankings, because the model has trained on that data and some neurons will have overfit that data more than others, so we have to use data from the same distribution as our training data. That is, we have to use a validation data set drawn from the same distribution as our training data. This is not a big

difference from an implementation point of view: we simply calculate the rankings on the validation data and then use that ranking to get the outlier score on our unlabeled data, as you'll see in this section.

The main difference is that we will get values from the unlabeled data that are between two values in the rankings. We can use simple linear interpolation to calculate these values. Suppose that our validation data consists of only 10 items, which happen to be the same as in section 4.2:

```
[2.43, 2.23, 1.74, 1.12, 0.89, 0.44, 0.23, -0.34,(-0.35) -0.36, -0.42]
```

Now imagine an unlabeled item with a value of –0.35 (above where it would fall in the ranking). That value is halfway between the eighth- and ninth-lowest items, so we can give the item an outlier score of 8.5/10 = 85%. Similarly, if the unlabeled item has a value of –0.355, which is three-quarters of the distance between the eighth and ninth item, the score would be 87.5%. We treat values above the first item as 1 and values below the last item as 0, giving us a [0–1] range in which the biggest outlier has a value of 100%.

There are different ways to combine the scores across the neurons for each item. It is statistically safest to take the average activation across all the neurons for each item. Especially if you are using activation from one of the hidden layers, you may have some neurons that are essentially spitting out random values and therefore generating a falsely high maximum for what would otherwise be an outlier. Your logits are more likely to be reliable for every value, so you could experiment with the equivalent of least confidence for logits: the lowest maximum score across all neurons. To see the results of model-based outlier sampling, run

```
> python active_learning.py --model_outliers=95
```

As in chapter 2, the code will choose this sampling strategy and select 95 unlabeled items for you to annotate, along with 5 randomly selected items from remaining unlabeled items. As in chapter 2, you always want to include a small number of random items as a safety net. If you don't want to evaluate any random items, you can add a `random=0` option:

```
> python active_learning.py --model_outliers=95 --random_remaining=0
```

You can play around with other numbers to see and/or annotate more or less than 95, too. If you skipped chapter 2, you will first be asked to annotate a purely random sample until you have enough initial training and test options. This time spent annotating will be important for evaluating accuracy and understanding the data, so please do these annotations now if you didn't previously!

The code for calculating the rank model outlier score is broken into four chunks of code. The model outlier function takes the current model, the unlabeled data, and held-out validation data taken from the same distribution as the training data. First, we

create the rankings on our held-out validation data, which you can see within diversity_sampling.py.

Listing 4.2 Get activation rankings using validation data

```
def get_validation_rankings(self, model, validation_data, feature_method):
    """ Get activation rankings using validation data

    Keyword arguments:
        model -- current machine learning model for this task
        validation_data -- held out data drawn from the same distribution as
        ➥ the training data
        feature_method -- the method to create features from the raw text

    An outlier is defined as
    unlabeled_data with the lowest average from rank order of logits
    where rank order is defined by validation data inference

    """

    validation_rankings = [] # 2D array, every neuron by ordered list of
    ➥ output on validation data per neuron

    # Get per-neuron scores from validation data
    if self.verbose:
        print("Getting neuron activation scores from validation data")

    with torch.no_grad():
        v=0
        for item in validation_data:
            textid = item[0]
            text = item[1]

            feature_vector = feature_method(text)
            hidden, logits, log_probs = model(feature_vector,
            ➥ return_all_layers=True)

            neuron_outputs = logits.data.tolist()[0] #logits

            # initialize array if we haven't yet
            if len(validation_rankings) == 0:
                for output in neuron_outputs:
                    validation_rankings.append([0.0] * len(validation_data))

            n=0
            for output in neuron_outputs:
                validation_rankings[n][v] = output
                n += 1

            v += 1

    # Rank-order the validation scores
    v=0
    for validation in validation_rankings:
```

We get the results from all model layers here.

We store the logit score for each validation item and each neuron.

```
        validation.sort()
        validation_rankings[v] = validation
        v += 1

    return validation_rankings
```

Rank-order each neuron according the scores from the held-out validation data.

In the second step, we order each unlabeled data item according to each neuron.

Listing 4.3 Code for model-based outliers in PyTorch

```
def get_model_outliers(self, model, unlabeled_data, validation_data,
➥ feature_method, number=5, limit=10000):
    """Get model outliers from unlabeled data

    Keyword arguments:
        model -- current machine learning model for this task
        unlabeled_data -- data that does not yet have a label
        validation_data -- held out data drawn from the same distribution
        ➥ as the training data
        feature_method -- the method to create features from the raw text
        number -- number of items to sample
        limit -- sample from only this many items for faster sampling
        ➥ (-1 = no limit)

    An outlier is defined as
    unlabeled_data with the lowest average from rank order of logits
    where rank order is defined by validation data inference

    """

    # Get per-neuron scores from validation data
    validation_rankings = self.get_validation_rankings(model,
    ➥ validation_data, feature_method)

    # Iterate over unlabeled items
    if self.verbose:
        print("Getting rankings for unlabeled data")

    outliers = []
    if limit == -1 and len(unlabeled_data) > 10000 and self.verbose:
        # we're drawing from *a lot* of data this will take a while
        print("Get rankings for a large amount of unlabeled data: this
        ➥ might take a while")
    else:
        # only apply the model to a limited number of items
        shuffle(unlabeled_data)
        unlabeled_data = unlabeled_data[:limit]

    with torch.no_grad():
        for item in unlabeled_data:
            text = item[1]

            feature_vector = feature_method(text)
```

Call to get the activation on validation data.

We get the results from all model layers here.

We get the rank order for each unlabeled item.

```
                hidden, logits, log_probs = model(feature_vector,
                ➥ return_all_layers=True)

                neuron_outputs = logits.data.tolist()[0] #logits

                n=0
                ranks = []
                for output in neuron_outputs:
                    rank = self.get_rank(output, validation_rankings[n])
                    ranks.append(rank)
                    n += 1

                item[3] = "logit_rank_outlier"

                item[4] = 1 - (sum(ranks) / len(neuron_outputs)) # average
                ➥ rank

                outliers.append(item)

        outliers.sort(reverse=True, key=lambda x: x[4])
        return outliers[:number:]
```

The ranking function takes the activation value for one unlabeled item for one neuron and the rankings for that neuron that were calculated on the validation data. Use the following code for ordering each unlabeled item according to the validation rankings.

Listing 4.4 Return the rank order of an item in terms of validation activation

```
def get_rank(self, value, rankings):
    """ get the rank of the value in an ordered array as a percentage

    Keyword arguments:
        value -- the value for which we want to return the ranked value
        rankings -- the ordered array in which to determine the value's
        ➥ ranking

    returns linear distance between the indexes where value occurs, in the
    case that there is not an exact match with the ranked values
    """

    index = 0 # default: ranking = 0

    for ranked_number in rankings:
        if value < ranked_number:
            break #NB: this O(N) loop could be optimized to O(log(N))
        index += 1

    if(index >= len(rankings)):
        index = len(rankings) # maximum: ranking = 1

    elif(index > 0):
        # get linear interpolation between the two closest indexes
```

```
            diff = rankings[index] - rankings[index - 1]
            perc = value - rankings[index - 1]
            linear = perc / diff
            index = float(index - 1) + linear

        absolute_ranking = index / len(rankings)

        return(absolute_ranking)
```

This listing is simply the implementation of the ranked-ordering example. Don't worry too much about the linear interpolation part; the code is a little opaque when implemented, but it is not capturing anything more complicated than you saw in the examples.

4.2.2 Which layers should I use to calculate model-based outliers?

You may want to try outlier detection on different layers of your model to see whether they produce better outliers for sampling. In general, the earlier the layer, the closer the neurons will be to the raw data. If you chose the input layer from the model, which is the feature vector, outliers from the input layer are almost identical to the outlier detection method that you implemented in chapter 2. Any hidden layer is going to fall somewhere between representing the raw data (early layers) and representing the predictive task (later layers).

You could also choose to look at multiple layers within the same sample. This approach is used in transfer learning with pretrained models; the model is "flattened" to create one single vector combining all the layers. You could use a flattened model for outlier detection too, but you may want to normalize by the amount of neurons per layer. In our model, the 128 neurons in the hidden layer would become the main contributor to an outlier detection algorithm that also included the 2 neurons from the final layer, so you might want to calculate the outlier ranking for the layers independently and then combine the two results.

Alternatively, you could sample from both, taking half your model-outliers from the logits and half from the hidden layer. Note that the 128 neurons in the hidden layer probably aren't too informative if you still have only 1,000 or so training items. You should expect the hidden layers to be noisy and some neurons to be random until you have many more labeled training items than neurons in your hidden layer—ideally, two or more orders of magnitude more training items than neurons in the layer (more than 10,000 labeled items).

If you are using layers near the input, be careful when your feature values don't represent activation. For our text example, the inputs *do* represent a form of activation, because they represent how often a word occurs. For computer vision, however, a higher input value may simply represent a lighter RGB color. In these cases, the layers toward the output of the model and the logits will be more reliable.

4.2.3 The limitations of model-based outliers

Here is a summary of the main shortcomings of using your model for sampling outliers:

- The method can generate outliers that are similar and therefore lack diversity within an active learning iteration.
- It's hard to escape some statistical biases that are inherent in your model, so you may continually miss some types of outliers.
- You still need a model in place before you start, and this approach gets better with more training data, so model-based outlier sampling is not suited to a cold start.
- We are determining an outlier by using our unlabeled data. It is easy to accidentally sample the opposite of what we want—things that look least like the data that we are trying to adapt to with new labels. For this reason, we use validation data to get our rankings, and you should follow this practice for any other kind of model-based outlier detection.

We will cover some solutions to the first issue in chapter 5, with algorithms that combine outlier detection and transfer learning. The second, third, and fourth issues are harder to overcome. Therefore, if you are sampling model-based outliers, you should consider using other methods of diversity sampling at the same time, including the methods that you can use from a cold start, such as clustering, which we cover next.

4.3 Cluster-based sampling

Clustering can help you target a diverse selection of data from the start. The strategy is fairly straightforward: instead of sampling training data randomly to begin with, we also divide our data into a large number of clusters and sample evenly from each cluster.

The reason why this works should be equally straightforward. By now, you have probably noticed that there are tens of thousands of news articles about local Australian sports teams in the headlines. If we randomly sample the data for human review, we are going to spend a lot of time manually annotating similar headlines about the results of sporting matches. If we precluster our data, however, these headlines are likely to end up together in one cluster, so we will need to annotate only a handful of examples from this sports-related cluster. This approach will save a lot of time, which we can instead spend annotating data from other clusters. Those other clusters may represent rarer types of headlines that are important but so rare that they would have been missed with random sampling. So clustering is saving time and increasing diversity.

Clustering is by far the most common method used for diversity sampling in real-world machine learning. It is the second method discussed in this chapter because it fit the flow of the book better. In practice, you will probably try this method of diversity sampling first.

You have probably encountered unsupervised learning, and you're most likely familiar with k-means, the clustering algorithm that we will be using. The approaches to unsupervised clustering and clustering for active learning are the same, but we will

be using the clusters to sample items for human review for labeling instead of interpreting the clusters or using the clusters themselves in downstream processing.

4.3.1 Cluster members, centroids, and outliers

The item that is closest to the center of a cluster is known as the *centroid*. In fact, some clustering algorithms explicitly measure the distance from the centroid item rather them from the cluster properties as a whole.

You calculated outliers from the entire dataset in chapter 2, and you can also calculate outliers when using clustering. Outliers are the statistical counterpoint of the centroid: they are farthest from the center of any cluster.

Figure 4.4 shows an example with five clusters, with a centroid and outlier for two of the clusters indicated. The majority of items in figure 4.4 are in one cluster: the large one in the middle. So if we sampled randomly instead of by clustering, we would end up spending most of the time labeling similar items. By clustering first and sampling from each cluster, we can ensure more diversity.

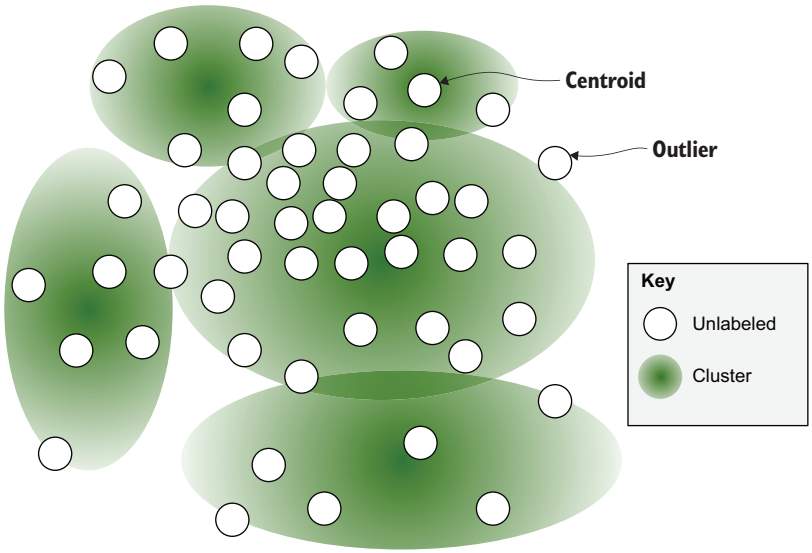


Figure 4.4 An example clustering algorithm applied to the data, splitting it into five separate clusters. For each cluster, the most central item is known as the *centroid*, and the items farthest from the centers are *outliers*.

We will sample from clusters in three ways:

- *Random*—Sampling items at random from each cluster. This strategy is close to random sampling but will spread out our selection across our feature space more evenly than purely random sampling.
- *Centroids*—Sampling the centroids of clusters to represent the core of significant trends within our data.

- *Outliers*—Sampling the outliers from our clustering algorithm to find potentially interesting data that might have been missed in the clusters. Outliers within clustering are sometimes known as *proximity-based* outliers.

Within a single cluster, the ranked centroids are likely to be similar. That is, the item that is the closest to the center is likely to be similar to that item that is second-closest to the center. So we sample randomly within the cluster or chose only the centroid.

Similarly, we probably need to sample only a small number of outliers per cluster. It's possible that the outliers are meaningful trends that the algorithm is missing, but it is more likely that they are genuinely rare: repeated rare words in the case of text or noisy/corrupted images in the case of computer vision. Typically, you need to sample only a small number of outliers, perhaps only one outlier from each cluster if you have a large number of clusters.

To keep the example simple, assume that we are sampling the centroid of each cluster, the single biggest outlier from each cluster, and three additional randomly sampled items within each cluster. To use cluster-based sampling, run

```
> python active_learning.py --cluster_based=95 --verbose
```

This command samples 95 unlabeled items via cluster-based sampling for you to annotate, along with 5 randomly selected items from remaining unlabeled items. I recommend running the code with the `verbose` flag, which prints three random items from each cluster as the code runs. You can get an idea of how well the clusters are capturing meaningful differences by examining whether the items in the cluster seem to be semantically related. In turn, this approach will give you an idea of how many meaningful trends in the data are being surfaced for human annotation.

4.3.2 Any clustering algorithm in the universe

As far as I know, no one has studied in depth whether one clustering algorithm is consistently better than another for active learning. Many pairwise studies look at variations on particular clustering algorithms, but no comprehensive broad study exists, so if you are interested in this topic, this situation would make a great research study.

Some clustering algorithms need only a single pass over the data, and some can be $O(N^3)$ complexity or worse. Although the more compute-intensive algorithms reach more mathematically well-motivated clusters within your data, the distribution of information across clusters won't necessarily be any better or worse for sampling items that need to be labeled.

For the system we will implement here, we don't want to make the people using the system wait a long time for the clustering algorithm to find the best clusters, so we'll choose an efficient clustering algorithm. We are going to use a variation of k-means that uses cosine similarity as the distance measure rather than the more typical Euclidean distance (figure 4.5). We have high-dimensional data, and Euclidean distance doesn't work well in high dimensions. One way to think about this problem is to think

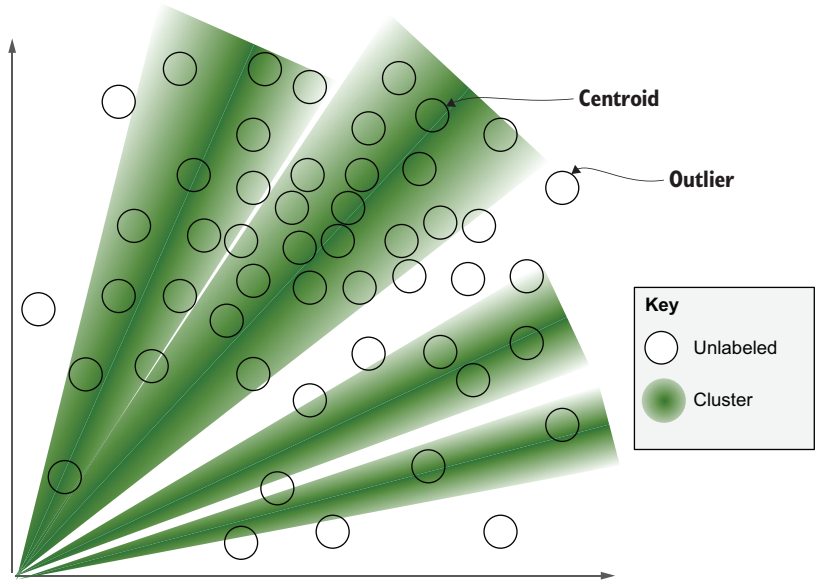


Figure 4.5 An example clustering algorithm using cosine similarity. For each cluster, the center is defined as a vector from 0, and the membership of that cluster is the angle between the vector representing the cluster and the vector representing the item. Note that although this example looks less like real clusters than the spheroidal clusters in figure 4.4, it is limited by being 2D. For higher-dimensional sparse data, which you are more likely to be using with your models, this kind of clustering is often better than the spheroid type shown here.

of many corners in your data. Almost all clustering algorithms are prone to producing unreliable results with high-dimensional data. In figure 4.4, we have examples in two dimensions and only four corners where outliers can hide from the center of the data distributions. If we had features in three dimensions, outliers could occupy eight corners. (Think of the eight corners of a cube.) By the time we get to 300 features, the data has 10^90 corners, and 10^90 is more than the number of atoms in the observable universe. You will certainly have more than 300 features in almost any natural language processing (NLP) task, so outliers can occur in a lot of corners of the space. For data with more than 10 dimensions, more than 99% of the space is in the corners, so if the data is uniformly distributed or even a Gaussian distribution, you will be measuring an artifact of the corners more than the distance, which can be unreliable.

You can think of cosine similarity in terms of looking at stars in the night sky. If you drew a straight line from yourself toward two stars and measured the angle between those lines, that angle would give you the cosine similarity. In the night-sky example, you have only three physical dimensions, but your data has one dimension for each feature. Cosine similarity is not immune to the problems of high dimensionality, but it tends to perform better than Euclidean distance especially for sparse data, such as our text encodings.

Cosine similarity measures whether two vectors are pointing in the same direction but does not measure the distance. There might be a small angle between two stars in the sky, but one happens to be much farther away. Because you are measuring only the angle, you are treating the stars as being equally far away. For this reason, cosine similarity is sometimes called *spherical k-means*, with all data points treated as though they were the same distance from 0 on a multidimensional sphere. This example does bring up an issue: data points can accidentally be in the same direction and therefore erroneously seem to be similar. The chance that this issue will occur in high-dimensional data is low, however, so high dimensions help (and make our calculations simpler). We can calculate a cluster's vector as the sum of all the vectors (features) of the items in that cluster and not worry about normalizing by the number of items, because cosine is not sensitive to the absolute values of the distance function.

4.3.3 K-means clustering with cosine similarity

Given two feature-vectors of the same size, v_1 and v_2, you can calculate the cosine of the angle between those vectors as

$$\phi_{CS}(v_1, v_2) = (v_1 \cdot v_2) / (\|v_1\|_2 \cdot \|v_2\|_2)$$

Cosine similarity is a native function in PyTorch, so we won't go too deeply into the implementation here. The double-line notation indicates the norm of the vector. The intuition of angles between stars in the night sky and the example in figure 4.5 (section 4.3.2) should be enough for you to understand what is going on. (If you are interested in reading more about cosine similarity or looking at other distance functions in PyTorch, you can start with the documentation at http://mng.bz/XdzM.)

Other popular machine learning libraries also have a lot of implementations of clustering algorithms. These other algorithms could work as well as the example that you are implementing here. There is a commonly held belief that clustering algorithms shouldn't be used for datasets with more than 10,000 items, but it isn't true. There have always been clustering algorithms that work reasonably well with a single pass of the data, so you shouldn't think of any limitation according to dataset size unless you are trying to trim your processing time to a few seconds. Even with compute-intensive clustering algorithms, you can often build the clusters in smaller subsets (batches) of the data to build clusters, and using the resulting clusters will be almost as good as using the entire dataset.

The general strategy for k-means is as follows:

1. Select the number of clusters you want by working backward from the number of annotations that you need.
2. Randomly add the data items to one of the initial clusters.
3. Iterate through the items and move them to another cluster if they are closer to that cluster.
4. Repeat step 3 until there are no more items to move or you have reached some predefined limit on the number of epochs through the data.

Cosine similarity and cosine distance are the same thing

You might see cosine similarity referred to as *cosine distance* in the literature. These terms mean the same thing. In general, clustering algorithms are more likely to use the term *distance* than *similarity*, and in the strictest definitions, distance = 1 – similarity. Cosine similarity does not follow the strict definition of the triangle inequality property (Schwarz inequality), however, so cosine similarity does not meet the formal definition of a distance metric—hence the name *similarity*. The terminology is confusing enough in this chapter when we are treating centroids and outliers as complements to get our [0, 1] range for every sampled item, so don't let this add to your confusion!

As noted in step 1, you should work backward and choose the number of clusters that makes the most sense given how many items you want to sample from each cluster. If you want to sample 5 items per cluster (1 centroid, 1 outlier, and 3 randomly chosen), and you want to annotate 100 items in this active learning iteration through this sampling strategy, you would select 20 clusters, as $20 \times 5 = 100$.

For completeness, the full code for k-means clustering with cosine similarity was implemented in the example code for this book, and you can see it at http://mng.bz/MXQm. This k-means strategy is the same regardless of the distance measure. The k-means function takes only two arguments: the data, which can be unlabeled or labeled (in which case the labels are ignored), and the number of clusters you want. You can see the k-means strategy in diversity_sampling.py with the main function in the following listing.

Listing 4.5 Cluster-based sampling in PyTorch

```
def get_cluster_samples(self, data, num_clusters=5, max_epochs=5,
➥ limit=5000):
    """Create clusters using cosine similarity

    Keyword arguments:
        data -- data to be clustered
        num_clusters -- the number of clusters to create
        max_epochs -- maximum number of epochs to create clusters
        limit -- sample only this many items for faster clustering (-1 = no
        ➥ limit)

    Creates clusters by the k-means clustering algorithm,
    using cosine similarity instead of more common euclidean distance

    Creates clusters until converged or max_epochs passes over the data

    """

    if limit > 0:
        shuffle(data)
        data = data[:limit]
```

```
cosine_clusters = CosineClusters(num_clusters)

cosine_clusters.add_random_training_items(data)

for i in range(0, max_epochs):
    print("Epoch "+str(i))
    added = cosine_clusters.add_items_to_best_cluster(data)
    if added == 0:
        break

centroids = cosine_clusters.get_centroids()
outliers = cosine_clusters.get_outliers()
randoms = cosine_clusters.get_randoms(3, verbose)

return centroids + outliers + randoms
```

Initialize clusters with random assignments.

Move each item to the cluster that it is the best fit for, and repeat.

Sample the biggest outlier in each cluster.

Sample the best-fit (centroid) from each cluster.

Sample three random items from each cluster, and pass the verbose parameter to get an idea of what is in each cluster.

You could substitute cosine for any other distance/similarity measure, and it might work equally well. One tactic that you may want to try to speed the process is to create the clusters on a subset of the data and then assign the rest of the data to its clusters. That approach gives you the best of both worlds: creating clusters quickly and sampling from the entire dataset. You may also want to experiment with a different number of clusters and a different number of random selections per cluster.

You'll remember from high school mathematics that cosine(90°) = 0 and cosine(0°) = 1. This makes our goal of a [0,1] range easy, because cosine similarity already returns values in a [0,1] range when calculated only on positive feature values. For our centroids, we can take the cosine similarity directly as our diversity score for each item. For the outliers, we will subtract the values from 1 so that we are consistent in our active learning ranking strategies and always sampling the highest numbers. As we said in chapter 3, consistency is important for downstream tasks.

4.3.4 Reduced feature dimensions via embeddings or PCA

Clustering works better for text than for images. If you come from a computer vision background, you know this already. When you looked at the clusters in your examples in this chapter, you could see semantic relationships between items in each cluster. All the clusters contain news headlines with similar topics, for example. But the same wouldn't be true if cosine similarity were applied to images, because individual pixels are more abstracted from the content of the images than sequences of characters are from the content of the text. If you applied cosine similarity to images, you might get a cluster of images that are landscapes, but that cluster might also erroneously include an image of a green car in front of a blue wall.

The most common method for reducing the dimensionality of data is principal component analysis (PCA). PCA reduces the dimensionality of a dataset by combining highly-correlated features. If you have been doing machine learning for some time, you probably thought that PCA was your first option to reduce the dimensionality of the data. PCA was common for early non-neural machine learning algorithms that

degraded in quality more when there was a high number of dimensions (features) with correlations between features. Neural model-based embeddings are more common in academia today, but PCA is more common in industry.

Implementing PCA is outside the scope of this book. It's a good technique to know in machine learning regardless, so I recommend reading more about it so that you have several tools for dimensionality reduction. PCA is not a native function in PyTorch (although I would not be surprised if it were added fairly soon), but the core operation of PCA is singular value decomposition (SVD), which is covered at https://pytorch.org/docs/stable/torch.html#torch.svd.

As an alternative to PCA, you can use embeddings from your model—that is, use the hidden layers of your model or from another model that has been trained on other data. You can use these layers as representations for modeling directly. Alternatively, you can use model distillation to lower the dimensionality within the clustering process, as follows:

1. Select the number of clusters you want.
2. Cluster the items according to your existing (high-dimensional) feature space.
3. Treat each cluster as a label, and build a model to classify items into each cluster.
4. Using the hidden layer from your new middle as your new feature set, continue the process of reassigning items to the best cluster.

Model design is important here. For text data, your architecture from section 4.2 is probably enough: a single hidden layer with 128 neurons. For image data, you probably want to have more layers and to use a convolutional neural network (CNN) or a similar network to help generalize away from specific pixel locations. In either case, use your intuition from building models for the amount of data you have and the chosen number of clusters (labels).

Note that if you have negative values in your vector, as you would if you are clustering on a hidden layer with LeakyReLU as the activation function, cosine similarity will return values in a [–1,1] range instead of a [0,1] range. For consistency, therefore, you would want to normalize by adding 1 and halving the result of cosine similarity to get a [0,1] range.

For a denser feature vector, whether from a model or from PCA, you might also consider a distance function other than cosine. Cosine similarity is best for large, sparse vectors, such as our word representations. You may not want to treat activations of [0.1, 0.1] the same as activations of [10.1,10.1], as cosine similarity does. PyTorch also has a built-in distance function for pairwise distance, which might be more appropriate in that case. You can see this function commented out where the cosine function now exists in the pytorch_clusters.py file. You can experiment with different distance functions to see whether you get more meaningful clusters. As the code says, you may need to normalize your cluster vectors according to the number of items in that cluster; otherwise, you should be able to sub in other distance functions without making other changes to the code.

As one final point on advanced clustering for computer vision, if you are clustering for diversity sampling, it may not matter if the clusters aren't semantically meaningful. From a sampling point of view, you might get good diversity of images from across your clusters even if the clusters themselves aren't semantically consistent. That is, you might be able to ignore embeddings and PCA, and cluster directly on the pixel values. This approach might give you equal success. Cosine similarity will create identical vectors for RGB = (50,100,100) and RGB = (100,200,200), so lighter, more saturated versions of the same image may be identical, but this may not matter. I'm not aware of any in-depth research on whether pixel-level clustering for images is always worse than using a reduced dimension when sampling for active learning, so this research topic would be a valuable topic for anyone who's interested in pursuing it.

4.3.5 Other clustering algorithms

In addition to other variations of k-means, you may want to experiment with other clustering algorithms and related unsupervised machine learning algorithms. It is beyond the scope of this book to talk about every popular clustering algorithm; many good books have been written on clustering. In this book, however, we will take a high-level look at three algorithms:

- Proximity-based clustering, such as k-nearest neighbors (KNN) and spectral clustering
- Gaussian mixture models (GMM)
- Topic modeling

You are probably familiar with KNN algorithms. KNN forms clusters based on proximity between a small number of items in that cluster (k items, instead of that cluster as a whole. A strength and limitation of k-means is that all clusters have a meaningful center: the mean itself. You can imagine L-shape clusters or other patterns that have no meaningful center; KNN allows you to capture these kinds of clusters. The same is true of spectral clustering, which is a vector-based clustering method that can also discover more complicated cluster shapes by representing the feature space in new vectors.

There is no clear evidence, however, that proximity-based clustering is consistently better than k-means clustering for active learning. You may want to capture data points separately at two different extremes in an L-shape because they are sufficiently different even if there is an unbroken link of items between them. Furthermore, your k-means algorithms will be discovering different kinds of shapes in your features if you build your clusters on hidden layers or PCA-derived vectors, as you learned earlier. Your k-means algorithm will discover simple spheroid clusters only in the vectors that it learns from, but if those vectors are abstracted from a greater number of features, your clusters would be more complicated if they are mapped back into those features. In fact, applying k-means to a vector from a hidden layer is similar using spectral clustering to discover different cluster shapes. So there is no clear advantage for spectral clustering for active learning—at least, no one has yet researched this topic in depth, to the point that one method is clearly better in most active learning use cases.

A GMM allows an item to be a member of multiple clusters at the same time. This algorithm can lead to more mathematically well-motivated clusters compared with k-means, which tries to force a cluster boundary where two clusters naturally overlap. You may see GMMs and related algorithms referred to as soft versus hard clustering or as fuzzy clustering. As with proximity-based clustering, there's no strong evidence that GMMs produce better samples for active learning than k-means. Early in my career, I worked with mixture models and active learning at the same time but never combined the two; I never felt that other active learning techniques fell short in a way that needed GMMs or similar algorithms to overcome them. So from practical experience, I can report that I never found it necessary to try to combine the two, but I haven't tested GMMs for active learning in depth either. This topic is another potentially exciting research area.

Topic modeling is used almost exclusively for text. Topic models explicitly discover sets of related words in a topic and the distributions of those topics across documents. The most popular algorithm is Latent Dirichlet Allocation (LDA), and you might see topic modeling referred to as LDA in the literature. Unlike GMMs, topic modeling is used a lot in practice, and it is especially common in social media monitoring tools. The related words in a single topic are often semantically related, so an expert user can generate topics and then select the most interesting ones for further analysis. This approach is a form of light supervision, an important human-in-the-loop strategy that we will return to in chapter 9. Within diversity sampling, you could generate clusters as topics and sample items from each topic as you would with any other clustering mechanism.

Although any clustering algorithm may not be *better* than k-means for modeling the data, it will be *different*, which will increase diversity. So if you have multiple clustering algorithms producing samples for active learning, you are less likely to have biases resulting from the mathematical assumptions of any one clustering method. If you're already using clustering algorithms on your data for some other reason, try them out as a sampling strategy.

4.4 Representative sampling

Representative sampling refers to explicitly calculating the difference between the training data and the application domain where we are deploying the model. In the model-based outliers and cluster-based sampling methods, we did not explicitly try to model the gap between our model and the data where we are evaluating our model's accuracy. So the natural next step is to try to find items that fit this profile: what unlabeled data looks most like the domain where we are deploying our model? This step can be as useful for you as a data scientist as it is for your model: learning what data looks most like where you are adapting it will give you good intuition about that dataset as a whole and the problems you might face. An example is shown in figure 4.6.

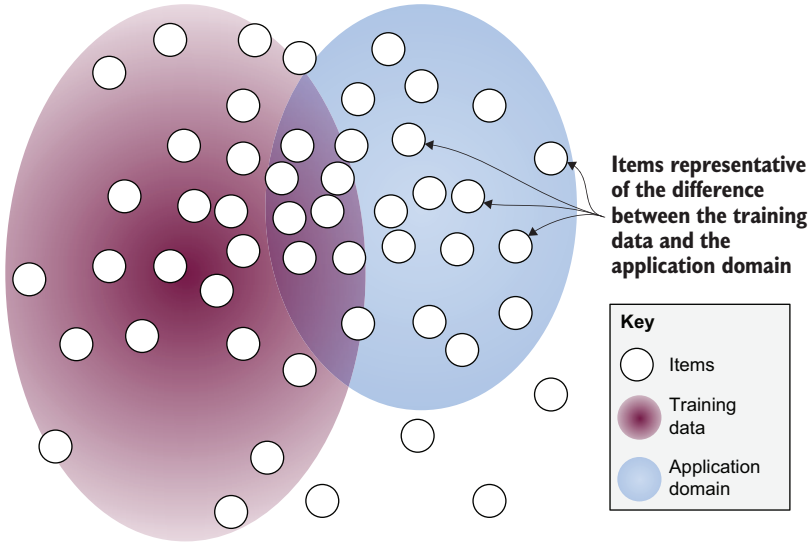


Figure 4.6 An example of representative sampling, showing that the current training data is from a different distribution of the data from the application domain. Representative sampling maximizes sampling of items that look the most like the application domain relative to the current training data.

4.4.1 Representative sampling is rarely used in isolation

It would be understandable if you assumed that representative sampling is the best method for active learning. If we can sample data that looks the most like where we want to deploy our models, doesn't this solve most of our diversity problems? While the general intuition is correct and representative sampling is one of the most powerful active learning strategies, it also one of the most prone to errors and overfitting. So, we will cover some of the limitations before jumping into the implementation.

For one thing, in most real-world scenarios, your unlabeled data is not from the domain where you will deploy your model. If you are deploying a model to identify future news headlines (as in our example) or to help autonomous vehicles navigate on the road at some point in the future, you do not have a sample of data from your target domain; you have a sample from an earlier intermediate time. This fact will be true of most real-world scenarios: you are deploying your model in the future. So if you tune your training data too closely to your unlabeled data, your model will be stuck in the past when it is deployed to future data.

In some deployment scenarios, such as a centralized model processing news headlines, you may be able to adapt to new data in near real time, so you won't have a big problem. In other use cases, such as autonomous vehicles, adapting the model in near real time and deploying it to every vehicle will be impossible. In either case, you still need a greater diversity of training items than only those that look the most like your current unlabeled data.

Representative sampling is the most prone to noise of all the active learning strategies in this book. If you have clean training data, the noise in your unlabeled data is often the most different from that training data. In NLP tasks, this noise would include corrupted text, text from a language that is not part of your target domain, text that came from a list of places names that didn't exist in your training data, and so on. For computer vision, noise would include corrupted image files; photos taken accidentally (while pointing the camera lens at the ground, for example); and artifacts that arise from using different cameras, resolutions, or compression techniques. Chances are that none of these types of noise is interesting for your task, so they will not produce an interesting or diverse range of samples to label.

Finally, representative sampling can do more harm than good if you apply it only during later cycles of the active learning process, especially when you don't have a domain adaptation problem. Suppose that you used uncertainty sampling for a few iterations of active learning and then applied representative sampling for later iterations. You have oversampled items *near* the decision boundary in early iterations, so representative sampling will oversample items *away* from the decision boundary in later iterations. This method will be worse than random sampling if you implement it this way.

For these reasons, representative sampling is rarely used in isolation; it is most often used in algorithms or processes that combine representative sampling with uncertainty sampling. You might use representative sampling only for items that are also near the decision boundary, for example. In some of the foundational academic papers on representative sampling, you might see that what they mean by *representative sampling* is a combination of diversity and uncertainty. We will return to combinations of approaches in chapter 5, which is where we will get the most out of all the sampling techniques. In this chapter, we'll introduce representative sampling in isolation so that you understand the basic principles before learning how to combine it with other methods.

With these caveats, representative sampling can be useful for domain adaptation. In academic research, there is a focus on domain adaptation without any additional labels, which is often called *discrepancy* rather than *representation*. In industry, I have yet to encounter domain adaptation without additional human intervention, so it should become an important tool in your belt.

4.4.2 Simple representative sampling

As with our clustering example in section 4.4.1, we can use many algorithms for representative sampling. We mentioned one in chapter 2, in which a small modification to our outlier detection method calculated whether something was an outlier for the training data but not an outlier for the unlabeled data. Here, we'll step up the sophistication a little and use cosine similarity from our training data to our unlabeled data, as follows:

1. Create one cluster containing the training data.
2. Create a second cluster containing the unlabeled data.
3. Sample items that have the greatest outlier score from the training relative to their outlier score from the unlabeled data.

To try representative sampling, run

```
> python active_learning.py --representative=95
```

This command samples 95 unlabeled items using representative sampling for you to annotate, along with 5 randomly selected items from remaining unlabeled items. The representative sampling function takes the training data and unlabeled data as arguments to find the unlabeled data items that are the most representative of the unlabeled data relative to the training data. Using our existing implementation for clustering, we can see that this is only a few lines of additional code.

Listing 4.6 Representative sampling in PyTorch

```
def get_representative_samples(self, training_data, unlabeled_data,
     number=20, limit=10000):
    """Gets the most representative unlabeled items, compared to training data
    Keyword arguments:
        training_data -- data with a label, that the current model is trained
        ➥ on
        unlabeled_data -- data that does not yet have a label
        number -- number of items to sample
        limit -- sample from only this many items for faster sampling (-1 =
        ➥ no limit)
    Creates one cluster for each data set: training and unlabeled

    """

    if limit > 0:
        shuffle(training_data)
        training_data = training_data[:limit]
        shuffle(unlabeled_data)
        unlabeled_data = unlabeled_data[:limit]

    training_cluster = Cluster()
    for item in training_data:
        training_cluster.add_to_cluster(item)

    unlabeled_cluster = Cluster()
    for item in unlabeled_data:
        unlabeled_cluster.add_to_cluster(item)


    for item in unlabeled_data:
        training_score = training_cluster.cosine_similary(item)
        unlabeled_score = unlabeled_cluster.cosine_similary(item)

        representativeness = unlabeled_score - training_score
```

Create a cluster for the training data.

Create a cluster for the unlabeled data.

For each unlabeled item, calculate how close it is to the unlabeled data relative to the labeled data.

```
            item[3] = "representative"
            item[4] = representativeness

    unlabeled_data.sort(reverse=True, key=lambda x: x[4])
    return unlabeled_data[:number:]
```

As with the clustering code, if you are applying this sampling strategy to images, you might want to use a lower-dimension vector that has abstracted the image away from individual pixels. Nothing in the code needs to change if you are using a different dimensionality of features; you are only plugging that new data vector directly into the algorithm.

4.4.3 Adaptive representative sampling

A small change to our code means that we can make our representative sampling strategy adaptive within each active learning iteration. When we have sampled the most representative item, we know that the item will get a label later, even if we don't know yet what that label will be. So we can add that single item to the hypothetical training data and then run representative sampling again for the next item. This approach will help prevent representative sampling from sampling only similar items. To try adaptive representative sampling, run

```
> python active_learning.py --adaptive_representative=95
```

This command samples 95 unlabeled items using adaptive representative sampling for you to annotate, along with 5 randomly selected items from remaining unlabeled items. The new code is even shorter, taking the same arguments and calling the representative sampling function once for each new item.

Listing 4.7 Adaptive representative sampling in PyTorch

```
def get_adaptive_representative_samples(self, training_data, unlabeled_data,
➥ number=20, limit=5000):
    """Adaptively gets the most representative unlabeled items, compared to
    ➥ training data

    Keyword arguments:
       training_data -- data with a label, that the current model is trained on
        unlabeled_data -- data that does not yet have a label
        number -- number of items to sample
        limit -- sample from only this many items for faster sampling (-1 =
        ➥ no limit)

    Adaptive variant of get_representative_samples() where the training_data
    ➥ is updated
    after each individual selection in order to increase diversity of samples

    """

    samples = []
```

```
    for i in range(0, number):
        print("Epoch "+str(i))
        representative_item = get_representative_samples(training_data,
        ➥ unlabeled_data, 1, limit)[0]
        samples.append(representative_item)
        unlabeled_data.remove(representative_item)

    return samples
```

With our building blocks of clusters and representative sampling, it is a small extension codewise to start implementing more sophisticated active learning strategies. We will cover more of these advanced techniques in detail in chapter 5. The code will be as short in most cases, but it is important to know the building blocks.

Note that this function takes a while to run because it needs to reevaluate the representative score for every unlabeled data point you are sampling. So if you are running this code on a smaller server or personal computer, you may want to lower the `number` of items to sample or the `limit` on items to consider so that you can see the results of this sampling strategy without waiting a long time.

4.5 Sampling for real-world diversity

Strategies for identifying and reducing bias are complicated and could fill a book of their own. In this text, we will concentrate on the data annotation problem: ensuring that training data represents real-world diversity as fairly as possible. As you read in the introduction to this chapter, we expect more from machine learning in some cases than we do from people. We expect many models to include something closer to English's 200,000-word vocabulary than the ~40,000 words known by a typical fluent human, for example. Therefore, this section covers the current best practice for ensuring that models are fair from an active learning point of view, knowing that measuring and reducing real-world bias is a complicated field that is far from solved.

The demographics for real-world diversity can be any real-world division that is meaningful for your data. Here is a (nonexhaustive) list of the kind of demographics that we might care about for our disaster-response examples:

- *Language*—Can we more accurately identify disaster-related content written in certain languages? There is an obvious bias here, as the data is mostly English.
- *Geography*—Can we more accurately identify disaster-related content from/about some countries? There is a high chance of bias here, as some countries will have more media reporting their disasters, and there will also be country-level population biases.
- *Gender*—Can we more accurately identify disaster-related content from/about people of one gender? It is possible that more males are writing the content than other genders, and this could be reflected in the style of writing.
- *Socioeconomics*—Can we more accurately identify disaster-related content from/about people with different incomes? There is often more reporting about wealthy nations, so perhaps this situation leads to bias in the data and models.

- *Race and ethnicity*—Can we more accurately identify disaster-related content from/about people of certain races or ethnicity? The media articles often portray the same type of event, such as a shooting by a lone man, as part of a war of terror for some ethnicities (and therefore disaster-related) but as an individual crime for other ethnicities (and therefore not disaster-related).
- *Date and time*—Can we more accurately identify disaster-related content at certain times of the day, days of the week, or months of the year? Fewer articles are published on weekends, and those articles tend to be more human-interest focused.

The biases may be different in combination, a situation known as *intersectional bias*. A bias toward people of a certain gender might be better, worse, or even inverted for some races and ethnicities, for example.

Depending on where you deploy your model, you may need to conform to local laws. In California, for example, labor laws prevent discrimination in several demographics, including many of the ones in the preceding list, as well as age, immigration status, sexual orientation, and religion. In some cases, solving the problem by encoding the data to change the sampling strategy may not be the right choice; instead, you need to solve the problem while you are collecting the data.

4.5.1 Common problems in training data diversity

Three common problems for fairness in data are summarized in figure 4.7. Each of the three demographics in figure 4.7 shows common problems that you will find when trying to create training data:

- A demographic that is overrepresented in your training data but not from the same distribution as your training data (X)
- A demographic that is from a distribution similar to the overall data distribution but not yet represented in a balanced way in the training data (O)
- A demographic that is underrepresented in the training data in such a way that the resulting model might be worse than if random sampling were used (Z)

Machine learning algorithms themselves are not prone to many inherent biases that are not already in the data, although those biases are possible. Most of the time, when an algorithm shows bias, it is reflecting or amplifying a bias that comes from the training data or the way that the training data is represented as features for the model. Even if bias comes solely from the model itself, you are probably responsible for creating the evaluation data to detect and measure that bias. If the source of data leads to poor results, you are also responsible for identifying that fact when you start to annotate the data. So if you are responsible for annotating the data, you may have more influence on model fairness than anyone else in your organization.

Note that many researchers in AI ethics use a broader definition of *algorithm* than most computer scientists do, including the treatment of the data for the machine learning models and the interpretation of the output. This definition is not inherently better or worse—only different. Be mindful about exactly which parts of an application using machine learning are being referred to when you read about algorithms in the AI ethics literature.

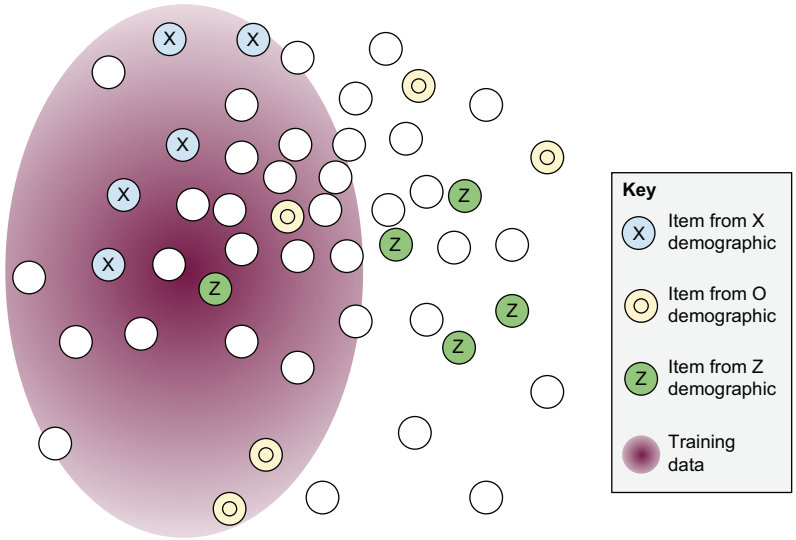


Figure 4.7 An example of the problems that diversity sampling tries to address. Here, we have items mapped to three real-world demographics that we're calling X, O, and Z.

Demographic X looks pretty good. All the examples we have so far are within the current training data. X is not the same distribution as the training data as a whole. This problem isn't typical of neural models, but it could be a problem with simpler models such as Naive Bayes. X is typical of a privileged demographic with a positive bias, such as standard English data within a multilingual dataset.

Demographic O is partially in the training data today and partially outside it. O is distributed across the entire feature range fairly evenly. So if we can collect training data that is representative of the entire feature space, we are least worried about O. O is typical of a demographic with minimal bias (positive or negative), such as time-based demographics, in which each item was collected carefully over a certain period.

By contrast, demographic Z is clustered outside the current training data. Even worse, a Z data point inside the current training data appears to be an outlier for Z. The model may not have information about Z and may actually be modeling Z incorrectly. Z is typical of an underrepresented demographic, such as an underrepresented ethnicity that does not appear in the dataset except when that person happens to share features with a more privileged demographic.

4.5.2 Stratified sampling to ensure diversity of demographics

Without a reference data set of unlabeled items from each demographic, you need to apply active learning strategies that you have applied before, but now in a stratified manner across all your data:

1. Apply least confidence sampling for every demographic, selecting an equal number of items in each demographic where that demographic was the most-confident prediction.
2. Apply margin of confidence sampling for every demographic, selecting an equal number of items in each demographic where that demographic was the most-confident or second-most-confident. Recall that margin of confidence is explicitly looking at the two most confident.)
3. Apply model-based outlier detection for each demographic.
4. Apply cluster-based sampling within each demographic.

Basically, in the same way that we wanted the best possible dataset from our unlabeled data as a whole, we want to do this for each demographic, being careful to stratify the sampling across those demographics.

There is no separate code for this task in this chapter. You should be able to divide up the data according to demographics that you care about and apply the sampling strategies only to data for each demographic.

4.5.3 Represented and representative: Which matters?

There is a subtle but important difference between having data that is representative of a demographic and having the demographic be well-represented in the data. The distinction is especially important depending on the type of model you are using, so we will tease them apart here:

- *Representative demographic data*—Your data is representative of a demographic if it is drawn from the same distribution as that demographic. In statistical terms, your labeled data is representative if it is independent and identically distributed (IDD) from a data that is randomly drawn from that demographic.
- *A demographic that is well-represented*—A demographic is well-represented if there is sufficient data representing that demographic for your model to perform fairly, but the data is not required to be IDD.

If you know that your unlabeled data fairly represents the demographic you care about and it is accurately encoded for that demographic, you can create an additional evaluation data set that draws randomly from within each demographic. If your demographics are not equally frequent, this approach will be faster than creating evaluation data by randomly sampling from across the entire data set. But you can use this dataset only for evaluating your per-demographic accuracy (section 4.5.4).

Remember that your unlabeled data may *not* be representative of each demographic. Data in this chapter that comes from an Australian media organization focuses on news within Australia and countries that are geographically or politically close to Australia. Articles about Uganda, for example, are not going to be representative of actual events in Uganda; the data will be biased toward events that are perceived as more important for Australia. It is not possible to get representative data for Uganda in this case. Instead, you should use clustering to get as diverse a set of articles about Uganda as possible so that at very least, articles about Uganda are well-represented.

If you are using a neural model, you might be OK if you have well-represented data that is *not* representative. Provided that there is enough data, a neural model can be accurate for all items in a given demographic, even if it was trained on data that was imbalanced within that demographic. The Uganda news articles, for example, might be balanced too much toward sports-related articles. Provided there are sufficient examples of the other types of news from Uganda for your model to be accurate on those topics, it won't matter that sports-related news is overrepresented; your model can be equally accurate on all types of news from Uganda.

If you are using generative models, however, especially a simpler one like Naive Bayes, your model is explicitly trying to model the classification task by assuming representative data. In this case, you need to work harder to ensure that your data is representative or to try to encode representativeness in your model by manipulating parameters such as the prior probability of certain data types.

This approach separates sampling for real-world diversity from stratified sampling. In the social sciences, *stratified sampling* is a technique for ensuring that data is as representative as possible and for weighting the results of activities such as surveys to account for demographic imbalances. Depending on the neural model, it might be enough that the data exists in the training data and the bias won't be perpetuated. On the other hand, a model might amplify any bias. So, the situation becomes a little more complicated and needs to be tackled holistically, taking into account the machine learning architecture. If you care about the real-world diversity of your models, the literature on stratified sampling is still a good place to start, knowing that this sampling strategy will not necessarily be the only solution to the problem.

4.5.4 Per-demographic accuracy

If we have real-world demographics in our data, we can calculate a variation of macro accuracy according to those demographics. For each item belonging to a certain demographic, how many of those were predicted correctly for their given labels? Note that each "error" will be both a false positive and a false negative. So unless you are excluding certain labels from your accuracy or are setting a threshold for trusted predictions, you will have identical precision and recall values for per-demographic accuracy (the same situation as for micro precision and recall). Let d indicate membership in each demographic. Precision and recall, therefore, are

$$P_{\text{demographic}} = \frac{\sum_d P_d}{d}$$

$$R_{\text{demographic}} = \frac{\sum_d R_d}{d}$$

I haven't seen this technique used often in industry, but that doesn't mean that it shouldn't be adopted. Most studies of demographic inequality tend to be ad-hoc. For face recognition, for example, there are many examples of popular media organizations selecting a small number of images of people who represent different ethnicities and looking for different levels of accuracy across those ethnicities. In those use cases, the media organizations are testing precision only, and on a small (and possibly nonrepresentative) sample. That approach works for a media story, but it won't work if we are serious about improving the fairness of our models.

If you are responsible for building the model and ensuring that it is as fair as possible, you should look at a broader range of ways to measure accuracy. You may want to refine demographic-based accuracy further, according to your use case. Here are some options:

- *Minimum accuracy*—The lowest precision, recall, and/or F-score of any demographic. If you want to treat your model as being only as strong as its weakest link in terms of fairness across demographics, you should take the minimum accuracy. You could take the minimum F-score from one demographic. For a harsher metric, take the minimum precision and minimum recall, possibly from different labels, and apply the F-score to those.
- *Harmonic accuracy*—The harmonic mean of the per-demographic accuracy, which will be harsher than the average demographic accuracy but not as harsh as taking the minimum (unless there are 0 values). As we take the harmonic mean of precision and recall to get the F-score, instead of the arithmetic mean, we could also take the harmonic mean. The harmonic mean will punish outlier low accuracies more than it rewards outlier high accuracies, but not as much as taking the minimum.

4.5.5 Limitations of sampling for real-world diversity

The biggest shortcoming of sampling for real-world diversity is that you have no way to guarantee that the model will be perfect, but you can measure the bias more accurately and ensure that your models will be much fairer than if you used only random sampling. Sometimes, you won't be able to make up for the bias, simply because not enough unlabeled data is available. I have worked in disaster response in languages such as Haitain Kreyol and Urdu, where there simply wasn't enough available data to cover the same breadth of potential disasters that we have for the English headlines. There is no way to fix this problem with labeling alone. Data collection is outside the scope of this book, but we will return to some other relevant techniques in chapter 9, when we cover methods for creating synthetic data.

4.6 Diversity sampling with different types of models

You can apply diversity sampling to any type of model architecture. Similar to what we learned in chapter 3 for uncertainty sampling, sometimes diversity sampling for other models is the same as for neural models, and sometimes diversity sampling is unique to a given type of model.

4.6.1 Model-based outliers with different types of models

For models that use linear regression, you can calculate model outliers in the same way as for a neural model: which items have the least activation across all labels? Use the prenormalized prediction scores, if you have access to them, as you did with the logits in this chapter.

In the case of Bayesian models, a model-based outlier has the lowest overall probability of each label. As with our neural models here, you could calculate lowest overall as the lowest average or the lowest maximum, depending on what makes the most sense for your use case.

In the case of SVMs, you can look for predictions that are near the hyperplane (decision boundary) but are the maximal distance from the support vectors themselves: the training items that determine the decision boundary. These items will be the equivalent of the model outliers that have high uncertainty in neural models.

4.6.2 Clustering with different types of models

You can use the unsupervised clustering methods in this chapter, such as k-means, to sample for any supervised machine learning algorithm. There is no need to change the k-means approach for different types of supervised machine learning algorithms, so you can start with the ones in this chapter and then think about refining them based on your model and data.

If you want to go deeper into cluster-based sampling, a lot of research was done on diversity sampling in the early 2000s. SVMs were at their peak popularity at the same time, so you will need to brush up on your SVM knowledge to get the most out of the research done at that time.

4.6.3 Representative sampling with different types of models

As mentioned earlier in the chapter, you could use Naive Bayes or Euclidean distance for representative sampling instead of cosine similarity. Any distance function could be as good for your particular data; we used cosine similarity in this book only because of the continuity from section 4.3 on clustering. If you changed the distance function in the clustering algorithm from cosine similarity to the probability of cluster membership, this edit of a few lines of code would be enough to you to try Bayesian clustering.

Decision trees offer a unique type of diversity sampling. You can look at where the number of predictions in different leaves differs from training to evaluation data. Suppose that your decision tree has 10 leaves, and all 10 leaves have an equal number of items when predicting your validation data. Now imagine that when you apply the model to your unlabeled data, 90% of that data ends up in one leaf. That leaf obviously represents the type of data in your target domain much better than your training data so far. So you should sample more items from within the leaf with 90% of the data, knowing that the data is more important for where you will deploy your model.

4.6.4 Sampling for real-world diversity with different types of models

The strategies for improving diversity in your neural models can be applied to other types of machine learning models. You want to ensure that you are optimizing for the same number of labels for each demographic and for equal accuracy for each demographic.

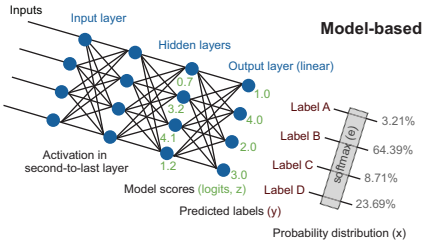
4.7 Diversity sampling cheat sheet

Figure 4.8 is a cheat sheet for the four diversity sampling approaches that you implemented in this chapter. If you are confident about these strategies, keep this cheat sheet handy as a quick reference.

Diversity sampling cheat sheet

Supervised machine learning models are limited by their data. For example, a chat bot will not support diversity if trained only on one variety of English. For many tasks, you need to find data that represents diversity in the data and diversity in the real-world. This is a form of *active learning* known as *diversity sampling*.

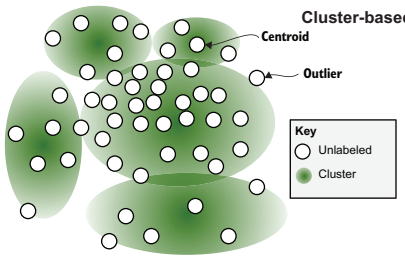
This cheat sheet shares four ways to increase the diversity of your training data.



Model-based outliers: Sampling for low activation in logits and hidden layers

Why? To find items that are confusing to your model because of lack of information. This is different from uncertainty through *conflicting* information, a complementary sampling method.

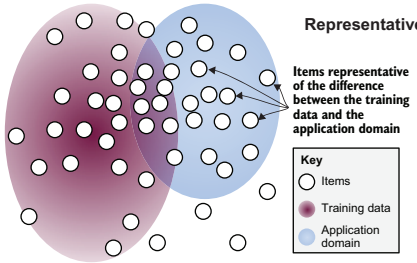
Tip: Experiment with average versus max activation.



Cluster-based sampling: Using unsupervised learning to presegment the data

Why? To ensure that you are sampling data from all the meaningful trends in your data's feature space, not just the trends that contain the most items; also to find outliers that are not part of any trend.

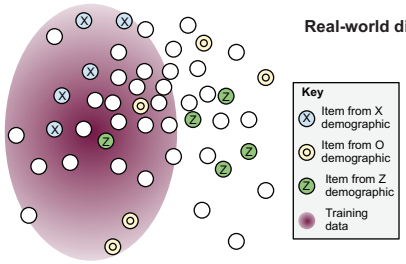
Tip: Try different distance metrics and clustering algorithms.



Representative sampling: Finding items most representative of the target domain

Why? When your target domain is different from your current training data, you want to sample items *most* representative of your target domain to adapt to the domain as fast as possible.

Tip: Extendable to be adaptive within one active learning cycle



Real-world diversity: Increase fairness with data supporting real-world diversity

Why? So as many people can as possible take advantage of your models and you are not amplifying real-world biases. Use all active learning strategies to make your data as fair as possible.

Tip: Your model might not require representative data to be fair.

Robert (Munro) Monarch. *Human-in-the-Loop Machine Learning*, Manning Publications. http://bit.ly/huml_book
See the book for more details on each method and other active learning strategies like uncertainty sampling, with open source implementations in PyTorch. robertmunro.com | @WWRob

Figure 4.8 Cheat sheet for the types of diversity sampling covered in this chapter: model-based outlier sampling, cluster-based sampling, representative sampling, and sampling for real-world diversity. These four strategies ensure diversity and representation in your data—respectively, items that are unknown to your model in its current state; items that are statistically representative of the entire distribution of your data; items that are maximally representative of where you are going to deploy your model; and items that are most representative of real-world demographics.

4.8 Further reading

You will have to go outside of the machine learning literature for many of the most important papers related to diversity sampling. If you are focused on collecting the right data, then the language documentation and archiving literature starting in the early 2000s is the best starting place. If you are focused on stratified sampling within your data, then there is a century of social science literature that is relevant in fields as varied as education and economics. This section limits the further reading to the machine learning literature, so keep in mind that the best papers are building on advances in other fields.

4.8.1 Further reading for model-based outliers

The model-based outlier algorithms are ones that I've personally developed and have not yet published outside this book except in informal presentations and classes. The literature on neural-based methods for determining outliers is growing but tends to focus on statistical outliers not low activation.

The practice of investigating a neural model to determine its knowledge (or lack thereof) is sometimes called *probing*. While there aren't yet papers on probing to discover outliers for active learning, there are no doubt some good techniques in the broader model probing literature that could be adapted for this purpose.

4.8.2 Further reading for cluster-based sampling

For cluster-based sampling, the best starting point is "Active Learning Using Pre-clustering," by Hieu T. Ngyuen and Arnold Smeulders (http://mng.bz/ao6Y). For the most cutting-edge research on cluster-based sampling, look for papers that cited these authors recently and are themselves highly cited.

Note that Ngyuen and Smeulders used an active learning metric that combines clustering with uncertainty sampling. As noted earlier in this chapter, this combination is the most common way to use clustering within active learning. The topics are taught separately in this text so you can understand them in isolation before learning how to combine them. Before jumping into the research, you may want to read chapter 5, in which you combine clustering and uncertainty sampling.

The earliest papers that look at clustering for active learning came from scientists in Russia. The first English version of these papers that I'm aware of is Novosibirk Zagoruiko's "Classification and Recognition" (http://mng.bz/goXn). If you can read Russian, you can find even earlier papers from scientists who were thinking about this problem more than 50 years ago!

4.8.3 Further reading for representative sampling

The principles of representative sampling were first explored in "Employing EM and Pool-Based Active Learning for Text Classification," by Andrew Kachites McCallum and Kamal Nigam (http://mng.bz/e54Z). For the most cutting-edge research on representative sampling, look for papers that cited these authors recently and are themselves highly cited.

4.8.4 Further reading for sampling for real-world diversity

Here are two good papers on machine learning for real-world diversity, one each in computer vision and NLP. Both find that popular models are more accurate for people from wealthier backgrounds and that training data is biased toward the object seen by wealthier people and the languages spoken by wealthier/majority populations:

- "Does Object Recognition Work for Everyone?", by Terrance DeVries, Ishan Misra, Changhan Wang, and Laurens van der Maaten (http://mng.bz/pVG0).
- "Incorporating Dialectal Variability for Socially Equitable Language dentification," by David Jurgens, Yulia Tsvetkov, and Dan Jurafsky (http://mng.bz/OEyO).

For a critical review of bias in the language technology literature, including how inconsistently the term bias is used, I recommend Su Lin Blodgett, Solon Barocas, Hal Daumé III, and Hanna Wallach's paper "Language (Technology) Is Power: A Critical Survey of 'Bias' in NLP" (http://mng.bz/Yq0Q).

Summary

- This chapter covered four common approaches to diversity sampling: model-based outlier sampling, cluster-based sampling, representative sampling, and sampling for real-world diversity. These techniques can help you understand the kinds of "unknown unknowns" in your models.
- Model-based outlier sampling allows you to sample items that are unknown to your model in its current state, helping you expand your model's knowledge where there are currently gaps.
- Cluster-based sampling allows you to sample items that are statistically representative of the entire distribution of your data, helping you expand your model's knowledge to capture all the meaningful trends in your data, including the rarer ones that would likely be missed with random sampling.
- Representative sampling can be used to sample items that are maximally representative of where you are going to deploy your model, helping you adapt your model to domains that are different from your current training data, which is a common problem in real-world machine learning.
- To support real-world diversity, you need to deploy all your techniques from uncertainty sampling and diversity sampling to make your applications more accurate across a diverse set of users and, therefore, more fair.
- Accuracy metrics such as micro and macro F-score can be applied across real-world demographics as one way to measure potential biases in a model.
- Interpreting the layers of a neural model for diversity sampling lets you access as much information as possible for active learning, giving you more options for calculating model outliers and providing a building block for advanced transfer learning techniques.

- The strategies for deciding how many items should be reviewed by humans when implementing diversity sampling is different from uncertainty sampling, because in some cases, they can be adaptive within each iteration of active learning. Adaptive sampling methods allow you to make the human-in-the-loop machine learning feedback loop more efficient because you don't have to wait for your model to retrain.
- Implementing diversity sampling is possible with any supervised machine learning algorithm, including neural models, Bayesian models, SVMs, and decision trees. You can implement active learning with any type of machine learning algorithm that you are currently using; you don't need to switch to the neural models that are the focus of the examples in this book. You might even decide to try some of these additional algorithms for active learning to take advantage of their unique properties.

5 Advanced active learning

This chapter covers

- Combining uncertainty sampling and diversity sampling techniques
- Using active transfer learning to sample the most uncertain and the most representative items
- Implementing adaptive transfer learning within an active learning cycle

In chapters 3 and 4, you learned how to identify where your model is uncertain (what your model knows it doesn't know) and what is missing from your model (what your model doesn't know that it doesn't know). In this chapter, you learn how to combine these techniques into a comprehensive active learning strategy. You also learn how to use transfer learning to adapt your models to predict which items to sample.

5.1 Combining uncertainty sampling and diversity sampling

This section explores ways to combine all the active learning techniques that you have learned up to this point so that you can use them effectively them for your particular use cases. You will also learn one new active learning strategy: expected error reduction, which combines principles of uncertainty sampling and diversity

sampling. Recall from chapter 1 that an ideal strategy for active learning tries to sample items that are near the decision boundary but are distant from one another, as shown in figure 5.1.

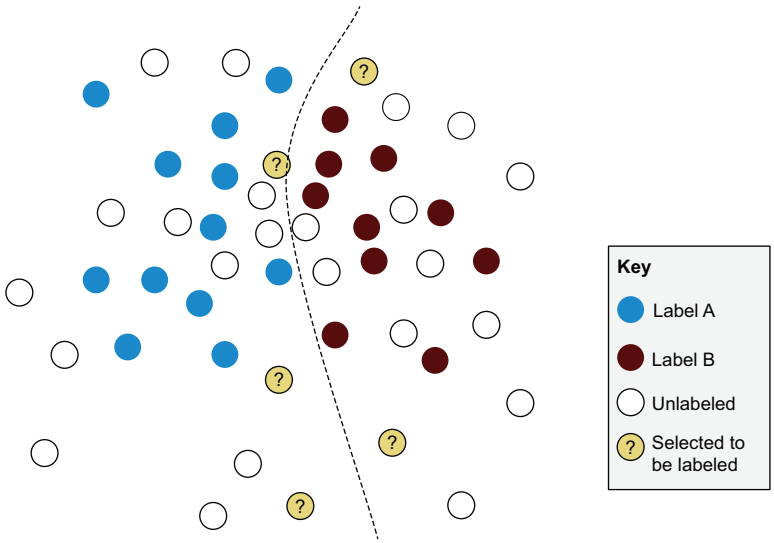


Figure 5.1 One possible result of combining uncertainty sampling and diversity sampling. When these strategies are combined, items near diverse sections of the decision boundary are selected. Therefore, we are optimizing the chance of finding items that are likely to result in a changed decision boundary when they're added to the training data.

You have learned to identify items that are near the decision boundary (uncertainty sampling) and distant from one another (cluster-based sampling and adaptive representative sampling). This chapter shows you how to sample items that are both near the decision boundary and diverse, like those shown in figure 5.1.

5.1.1 Least confidence sampling with cluster-based sampling

The most common way that uncertainty sampling and diversity sampling are combined in industry is takes a large sample from one method and further filter the sample with another method. This technique has no common name, despite its ubiquity, probably because so many companies have invented it independently by necessity.

If you sampled the 50% most uncertain items with least confidence sampling and then applied cluster-based sampling to sample 10% of those items, you could end up with a sample of 5% of your data more or less like those in figure 5.1: a near-optimal combination of uncertainty and diversity. Figure 5.2 represents this result graphically. First, you sample the 50% most uncertain items; then you apply clustering to ensure diversity within that selection, sampling the centroid of each cluster.

With the code you have already learned, you can see that combining least confidence sampling and clustering is a simple extension in advanced_active_learning.py

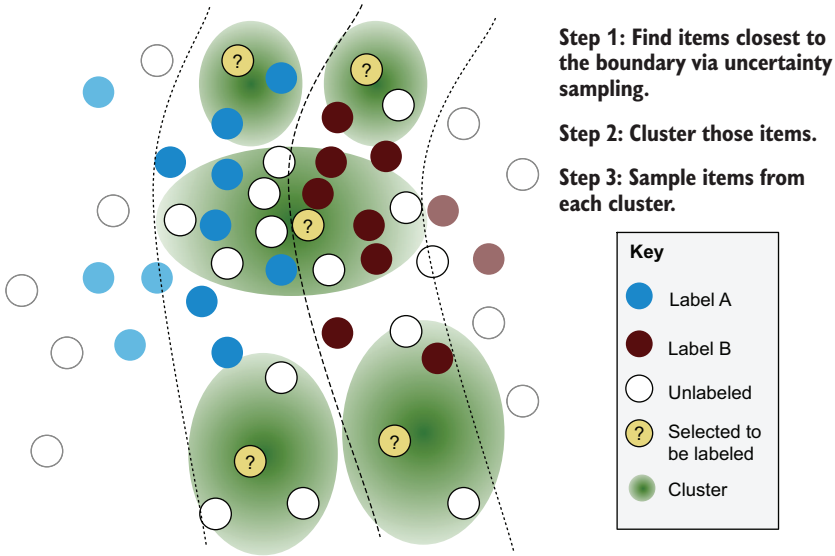


Figure 5.2 An example combining least confidence and clustering-based sampling. First, uncertainty sampling finds items near the decision boundary; then clustering ensures diversity within that selection. In this figure, the centroids from each cluster are sampled. Alternatively, or in addition, you could select random members of outliers.

within the same code repository that we have been using (https://github.com/rmunro/pytorch_active_learning), as shown in the following listing.

Listing 5.1 Combining least confidence sampling and clustering

```
def get_clustered_uncertainty_samples(self, model, unlabeled_data, method,
➥ feature_method, perc_uncertain = 0.1, num_clusters=20, max_epochs=10,
➥ limit=10000):

    if limit > 0:
      shuffle(unlabeled_data)
      unlabeled_data = unlabeled_data[:limit]
    uncertain_count = math.ceil(len(unlabeled_data) * perc_uncertain)

    uncertain_samples = self.uncertainty_sampling.get_samples(model,
    ➥ unlabeled_data,
    ➥ method, feature_method, uncertain_count, limit=limit)
    samples = self.diversity_sampling.get_cluster_samples(uncertain_samples,
    ➥ num_clusters=num_clusters)

    for item in samples:
      item[3] = method.__name__+"_"+item[3] # record the sampling method

    return samples
```

Get a large sample of the most uncertain items.

Within those uncertain items, use clustering to ensure a diverse sample.

Only two new lines of code are needed to combine the two approaches: one to get the most uncertain items and one to cluster them. If you are interested in the disaster-response text classification task, try it with this new command:

```
> python active_learning.py --clustered_uncertainty=10 --verbose
```

You'll immediately see that data tends to fall near the divide of text that may or not be disaster-related and that the items are a diverse selection. You have many options for using uncertainty sampling to find items near the decision boundary and then apply cluster-based sampling to ensure diversity within those items. You can experiment with different types of uncertainty sampling, different thresholds for your uncertainty cutoff, and different parameters for clustering. In many settings, this combination of clustering and uncertainty sampling will be the fastest way to drill down on the highest-value items for active learning and should be one of the first strategies that you try for almost any use case.

The simple methods of combining strategies rarely make it into academic papers; academia favors papers that combine methods into a single algorithm rather than chaining multiple simpler algorithms. This makes sense, because combining the methods is easy, as you have already seen; there is no need to write an academic paper about something that can be implemented in a few lines of code. But as a developer building real-world active learning systems, you should always implement the easy solutions before attempting more experimental algorithms.

Another reason to try simple methods first is that you might need to keep supporting them in your applications for a long time. It will be easier to maintain your code if you can get 99% of the way there without having to invent new techniques. See the following sidebar for a great example of how early decisions matter.

Your early data decisions continue to matter

Expert anecdote by Kieran Snyder

The decisions that you make early in a machine learning project can influence the products that you are building for many years to come. This is especially true for data decisions: your feature-encoding strategies, labeling ontologies, and source data will have long-term impacts.

In my first job out of graduate school, I was responsible for building the infrastructure that allowed Microsoft software to work in dozens of languages around the world. This job included making fundamental decisions such as deciding on the alphabetical order of the characters in a language—something that didn't exist for many languages at the time. When the 2004 tsunami devastated countries around the Indian Ocean, it was an immediate problem for Sinhalese-speaking people in Sri Lanka: there was no easy way to support searching for missing people because Sinhalese didn't yet have standardized encodings. Our timeline for Sinhalese support went from several months to several days so that we could help the missing-persons service, working with native speakers to build solutions as quickly as possible.

(continued)

The encodings that we decided on at that time were adopted by Unicode as the official encodings for the Sinhalese language and now encode that language forever. You won't always be working on such critical timelines, but you should always consider the long-term impact of your product decisions right from the start.

Kieran Snyder is CEO and co-founder of Textio, a widely used augmented writing platform. Kieran previously held product leadership roles at Microsoft and Amazon and has a PhD in linguistics from the University of Pennsylvania.

Don't assume that a complicated solution is necessarily the best; you may find that a simple combination of least confidence and clustering is all you need for your data. As always, you can test different methods to see which results in the biggest change in accuracy against a baseline of random sampling.

5.1.2 Uncertainty sampling with model-based outliers

When you combine uncertainty sampling with model-based outliers, you are maximizing your model's current confusion. You are looking for items near the decision boundary and making sure that their features are relatively unknown to the current model. Figure 5.3 shows the kinds of samples that this approach might generate.

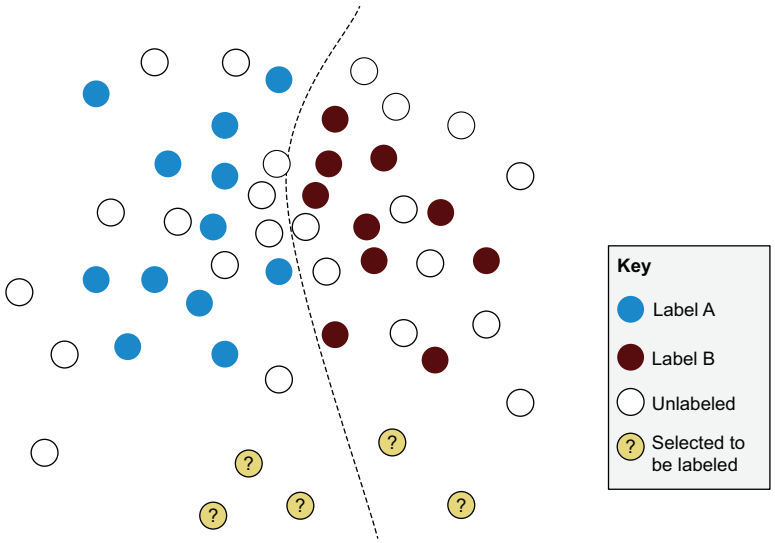


Figure 5.3 This example of combining uncertainty sampling with model-based outliers selects items that are near the decision boundary but that are different from the current training data items and, therefore, different from the model.

Listing 5.2 Combining uncertainty sampling with model-based outliers

```
def get_uncertain_model_outlier_samples(self, model, outlier_model,
➥ unlabeled_data, training_data, validation_data, method, feature_method,
➥ perc_uncertain = 0.1, number=10, limit=10000):

    if limit > 0:
      shuffle(unlabeled_data)
      unlabeled_data = unlabeled_data[:limit]
    uncertain_count = math.ceil(len(unlabeled_data) * perc_uncertain)

    uncertain_samples = self.uncertainty_sampling.get_samples(model,
    ➥ unlabeled_data, method, feature_method, uncertain_count, limit=limit)

    samples = self.diversity_sampling.get_model_outliers(outlier_model,
    ➥ uncertain_samples, validation_data,feature_method,
    ➥ number=number, limit=limit)

    for item in samples:
      item[3] = method.__name__+"_"+item[3]

    return samples
```

Get the most uncertain items.

Apply model-based outlier sampling to those items.

As in the example in listing 5.1, you need only two lines of code here to pull everything together. Although combining uncertainty sampling with model-based outliers is optimal for targeting items that are most likely to increase your model's knowledge and overall accuracy, it can also sample similar items. You can try this technique with this command:

```
> python active_learning.py --uncertain_model_outliers=100 --verbose
```

5.1.3 Uncertainty sampling with model-based outliers and clustering

Because the method in section 5.1.2 might oversample items that are close to one another, you may want to implement this strategy first and then apply clustering to ensure diversity. It takes only one line of code to add clustering to the end of the previous method, so you could implement it easily. Alternatively, if you have quick active learning iterations, this approach ensures more diversity when you combine uncertainty sampling and model-based outliers; you can sample a small number of items in each iteration.

5.1.4 Representative sampling cluster-based sampling

One shortcoming of the representative sampling technique that you learned in chapter 4 is that it treats the training data and target domain as single clusters. In reality, your data will often be multinodal in a way that a single cluster cannot optimally capture.

To capture this complexity, you can combine representative sampling and cluster-based sampling in a slightly more complicated architecture. You can cluster your training data and your unlabeled data independently, identify the clusters that are most representative of your unlabeled data, and oversample from them. This approach gives you a more diverse set of items than representative sampling alone (figure 5.4).

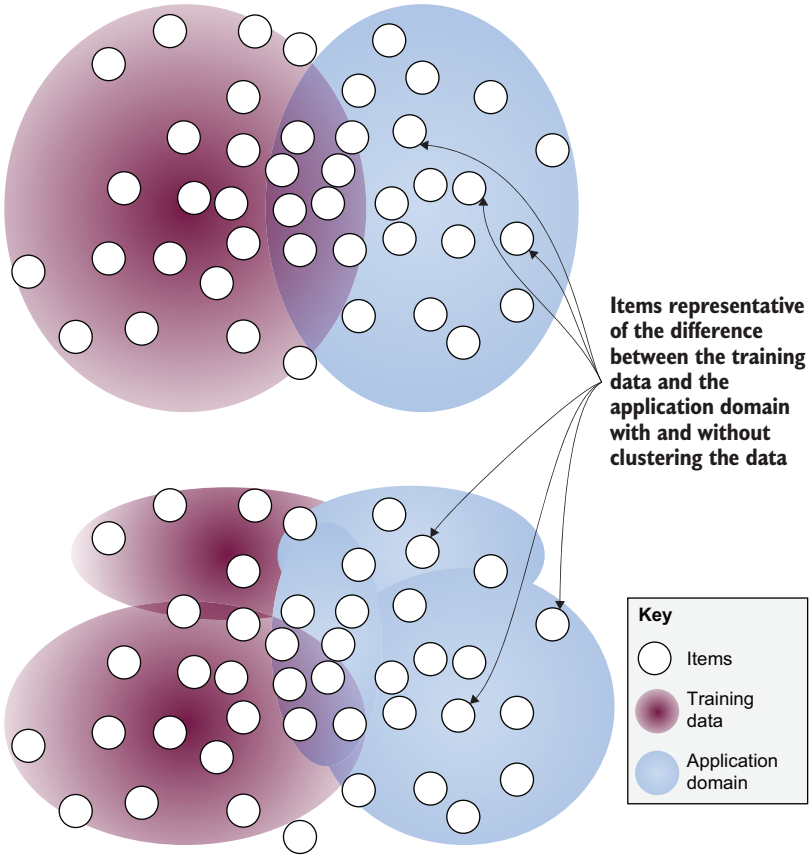


Figure 5.4 An example (bottom) of combining representative sampling and cluster-based sampling. This method samples items that are most like your application domain relative to your current training data and also different from one another. By comparison, the simpler representative sampling method in chapter 4 treats each distribution as a single distribution.

As you can see in figure 5.4, your current training data and target domains may not be uniform distributions within your feature space. Clustering the data first will help you model your feature space more accurately and sample a more diverse set of unlabeled items. First, create the clusters for the training data and unlabeled data from the application domain.

Listing 5.3 Combining representative sampling and clustering

```
def get_representative_cluster_samples(self, training_data, unlabeled_data,
➥ number=10, num_clusters=20, max_epochs=10, limit=10000):
    """Gets the most representative unlabeled items, compared to training data,
    ➥ across multiple clusters
```

```
Keyword arguments:
  training_data -- data with a label, that the current model is trained on
  unlabeled_data -- data that does not yet have a label
  number -- number of items to sample
  limit -- sample from only this many items for faster sampling (-1 =
  ➥ no limit)
  num_clusters -- the number of clusters to create
  max_epochs -- maximum number of epochs to create clusters

"""

if limit > 0:
  shuffle(training_data)
  training_data = training_data[:limit]
  shuffle(unlabeled_data)
  unlabeled_data = unlabeled_data[:limit]

# Create clusters for training data

training_clusters = CosineClusters(num_clusters)
training_clusters.add_random_training_items(training_data)

for i in range(0, max_epochs):
  print("Epoch "+str(i))
  added = training_clusters.add_items_to_best_cluster(training_data)
    if added == 0:
      break

# Create clusters for unlabeled data

unlabeled_clusters = CosineClusters(num_clusters)
unlabeled_clusters.add_random_training_items(training_data)

for i in range(0, max_epochs):
  print("Epoch "+str(i))
  added = unlabeled_clusters.add_items_to_best_cluster(unlabeled_data)
    if added == 0:
      Break
```

Create clusters within the existing training data.

Create clusters within the unlabeled data.

Then iterate each cluster of unlabeled data, and find the item in each cluster that is closest to the centroid of that cluster relative to training data clusters.

Listing 5.4 Combining representative sampling and clustering, continued

```
    most_representative_items = []

    # for each cluster of unlabeled data
    for cluster in unlabeled_clusters.clusters:
        most_representative = None
        representativeness = float("-inf")
```

```
            # find the item in that cluster most like the unlabeled data
            item_keys = list(cluster.members.keys())

            for key in item_keys:
                item = cluster.members[key]

                _, unlabeled_score =
                ➥ unlabeled_clusters.get_best_cluster(item)
                _, training_score = training_clusters.get_best_cluster(item)

                cluster_representativeness = unlabeled_score - training_score

                if cluster_representativeness > representativeness:
                    representativeness = cluster_representativeness
                    most_representative = item

            most_representative[3] = "representative_clusters"
            most_representative[4] = representativeness
            most_representative_items.append(most_representative)

        most_representative_items.sort(reverse=True, key=lambda x: x[4])
        return most_representative_items[:number:]
```

Find the best-fit cluster within the unlabeled data clusters.

Find the best-fit cluster within the training data clusters.

Record the difference between the two as our representative-ness score.

In design, this code is almost identical to the representative sampling method that you implemented in chapter 4, but you are asking the clustering algorithm to create multiple clusters for each distribution instead of only one for training data and one for unlabeled data. You can try this technique with this command:

```
> python active_learning.py --representative_clusters=100 --verbose
```

5.1.5 Sampling from the highest-entropy cluster

If you have high entropy in a certain cluster, a lot of confusion exists about the right labels for items in that cluster. In other words, these clusters have the highest average uncertainty across all the items. These items, therefore, are most likely to change labels and have the most room for changes in label.

The example in figure 5.5 is the opposite of clustering for diversity in some ways, as it deliberately focuses on one part of the problem space. But sometimes, that focus is exactly what you want.

Note that this approach works best when you have data with accurate labels and are confident that the task can be solved with machine learning. If you have data that has a lot of inherent ambiguity, this method will tend to focus in those areas. To solve this problem, see how much of your existing training data falls into your high-entropy clusters. If the cluster is already well represented in your training data, you have good

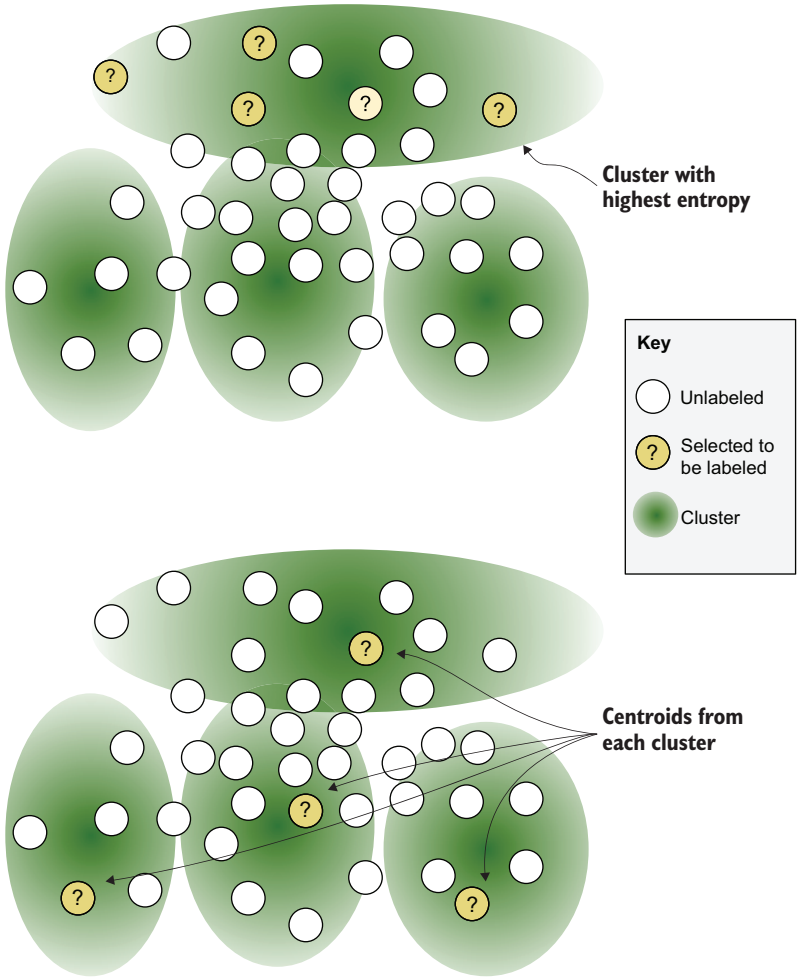


Figure 5.5 This example of combining cluster-based sampling with entropy (bottom) samples items within the cluster that show the most confusion. You might think of this cluster as being the one that straddles the decision boundary most closely. In this example, random items are sampled in the cluster, but you could experiment by sampling the centroid, outliers, and/or oversampling items within the cluster that have the highest entropy. By comparison, simple clustering (top) samples items from every cluster.

evidence that it is an inherently ambiguous part of your feature space and that additional labels will not help. The following listing shows the code for selecting the cluster with the highest average entropy.

Listing 5.5 Sampling from the cluster with the highest entropy

```
def get_high_uncertainty_cluster(self, model, unlabeled_data, method,
➥ feature_method, number=10, num_clusters=20, max_epochs=10, limit=10000):
    """Gets items from the cluster with the highest average uncertainty

    Keyword arguments:
      model -- machine learning model to get predictions from to determine
      ➥ uncertainty
      unlabeled_data -- data that does not yet have a label
      method -- method for uncertainty sampling (eg: least_confidence())
      feature_method -- the method for extracting features from your data
      number -- number of items to sample
      num_clusters -- the number of clusters to create
      max_epochs -- maximum number of epochs to create clusters
      limit -- sample from only this many items for faster sampling
      ➥ (-1 = no limit)
    """

    if limit > 0:
      shuffle(unlabeled_data)
      unlabeled_data = unlabeled_data[:limit]

    unlabeled_clusters = CosineClusters(num_clusters)
    unlabeled_clusters.add_random_training_items(unlabeled_data)

    for i in range(0, max_epochs):      <-- Create the clusters.
      print("Epoch "+str(i))
      added = unlabeled_clusters.add_items_to_best_cluster(unlabeled_data)
      if added == 0:
        break

    # get scores

    most_uncertain_cluster = None
    highest_average_uncertainty = 0.0

    # for each cluster of unlabeled data
    for cluster in unlabeled_clusters.clusters:
      total_uncertainty = 0.0
      count = 0

      item_keys = list(cluster.members.keys())

      for key in item_keys:
        item = cluster.members[key]
        text = item[1] # the text for the message

        feature_vector = feature_method(text)
        hidden, logits, log_probs = model(feature_vector,
        ➥ return_all_layers=True)

        prob_dist = torch.exp(log_probs) # the probability distribution of
        ➥ our prediction
```

```
            score = method(prob_dist.data[0]) # get the specific type of
            ➥ uncertainty sampling

            total_uncertainty += score
            count += 1

        average_uncertainty = total_uncertainty / count
        if average_uncertainty > highest_average_uncertainty:
            highest_average_uncertainty = average_uncertainty
            most_uncertain_cluster = cluster

    samples = most_uncertain_cluster.get_random_members(number)

    return samples
```

Calculate the average uncertainty (using entropy) for the items in each cluster.

In this code example, we are taking the average entropy of all items in a cluster. You can try different aggregate statistics based on your sampling strategy. If you know that you are sampling only the top 100 items, for example, you could calculate the average entropy across the 100 most uncertain items in each cluster rather than across every item in the cluster. You can try this technique with this command:

```
> python active_learning.py --high_uncertainty_cluster=100 --verbose
```

5.1.6 Other combinations of active learning strategies

There are too many possible combinations of active learning techniques to cover in this book, but by this stage, you should have a good idea of how to combine them. Here are some starting points:

- *Combining uncertainty sampling and representative sampling*—You can sample items that are most representative of your target domains and are also uncertain. This approach will be especially helpful in later iterations of active learning. If you used uncertainty sampling for early iterations, your target domain will have items that are disproportionately far from the decision boundary and could be selected erroneously as representative.
- *Combining model-based outliers and representative sampling*—This method is the ultimate method for domain adaptation, targeting items that are unknown to your model today but are also relatively common in your target domain.
- *Combining clustering with itself for hierarchical* clusters—If you have some large clusters or want to sample for diversity within one cluster, you can take the items from one cluster and use them to create a new set of clusters.
- *Combining sampling from the highest-entropy cluster with margin of confidence sampling (or some other uncertainty metrics)*—You can find the cluster with the highest entropy and then sample all the items within it that fall closest to a decision boundary.
- *Combining ensemble methods or dropouts with individual strategies*—You may be building multiple models and decide that a Bayesian model is better for determining

uncertainty, but a neural model is better for determining model-based outliers. You can sample with one model and further refine with another. If you're clustering based on hidden layers, you could adapt the dropout method from uncertainty sampling and randomly ignore some neurons while creating clusters. This approach will prevent the clusters from overfitting to the internal representation of your network.

5.1.7 Combining active learning scores

An alternative to piping the output from one sampling strategy to another is taking the scores from the different sampling strategies and finding the highest average score, which makes mathematical sense for all methods other than clustering. You could average each item's score for margin of confidence, model-based outliers, and representative learning, for example, and then rank all items by that single aggregate score.

Although all the scores should be in a [0–1] range, note that some of them may be clustered in small ranges and therefore not contribute as much to the average. If this is the case with your data, you can try converting all your scores to percentiles (quantiles), effectively turning all the sampling scores into stratified rank orders. You can use built-in functions from your math library of choice to turn any list of numbers into percentiles. Look for functions called `rank()`, `percentile()`, or `percentileofscore()` in various Python libraries. Compared with the other methods that you are using for sampling, converting scores to percentiles is relatively quick, so don't worry about trying to find the most optimal function; choose a function from a library that you are already using.

You could also sample via the union of the methods rather than filtering (which is a combination via intersection). This approach can be used for any methods and might make the most sense when you are combining multiple uncertainty sampling scores. You could sample the items that are in the most 10% uncertain by any of least confidence, margin of confidence, ratio of confidence, or entropy to produce a general "uncertain" set of samples, and then use those samples directly or refine the sampling by combining it with additional methods. There are many ways to combine the building blocks that you have learned, and I encourage you to experiment with them.

5.1.8 Expected error reduction sampling

Expected error reduction is one of a handful of active learning strategies in the literature that aim to combine uncertainty sampling and diversity sampling into a single metric. This algorithm is included here for completeness, with the caveat that I have not seen it implemented in real-world situations. The core metric for expected error reduction sampling is how much the error in the model would be reduced if an unlabeled item were given a label.[1] You could give each unlabeled item the possible labels that it could have, retrain the model with those labels, and then look at how the

[1] "Toward Optimal Active Learning through Sampling Estimation of Error Reduction," by Nicholas Roy and Andrew McCallum (https://dl.acm.org/doi/10.5555/645530.655646).

model accuracy changes. You have two common ways to calculate the change in model accuracy:

- *Overall accuracy*—What is the change in number of items predicted correctly if this item had a label?
- *Overall entropy*—What is the change in aggregate entropy if this item had a label? This method uses the definition of entropy that you learned in the uncertainty sampling chapter in sections 3.2.4 and 3.2.5. It is sensitive to the confidence of the prediction, unlike the first method, which is sensitive only to the predicted label.

The score is weighted across labels by the frequency of each label. You sample the items that are most likely to improve the model overall. This algorithm has some practical problems, however:

- Retraining the model once for every unlabeled item multiplied by every label is prohibitively expensive for most algorithms.
- There can be so much variation when retraining a model that the change from one additional label could be indistinguishable from noise.
- The algorithm can oversample items a long way from the decision boundary, thanks to the high entropy for the labels that are a diminishingly small likelihood.

So there are practical limitations to using this method with neural models. The original authors of this algorithm used incremental Naive Bayes, which can be adapted to new training items by updating the counts of a new item's features, and is deterministic. Given this fact, expected error reduction works for the authors' particular algorithm. The problem of oversampling items away from the decision boundary can be addressed by using the predicted probability of each label rather than the label frequency (prior probability), but you will need accurate confidence predictions from your model, which you may not have, as you learned in chapter 3.

If you do try to implement expected error reduction, you could experiment with different accuracy measures and with uncertainty sampling algorithms other than entropy. Because this method uses entropy, which comes from information theory, you might see it called *information gain* in the literature on variations of this algorithm. Read these papers closely, because *gain* can mean *lower* information. Although the term is mathematically correct, it can seem counterintuitive to say that your model knows more when the predictions have less information.

As stated at the start of this section, no one has (as far as I know) published on whether expected error reduction is better than the simple combination of methods through the intersection and/or union of sampling strategies. You could try implementing expected error reduction and related algorithms to see whether they help in your systems. You may be able to implement them by retraining only the final layer of your model with the new item, which will speed the process.

If you want to sample items with a goal similar to expected error reduction, you can cluster your data and then look at clusters with the highest entropy in the predictions,

like the example in figure 5.4 earlier in this chapter. Expected error reduction has a problem, however, in that it might find items in only one part of the feature space, like the uncertainty sampling algorithms used in isolation. If you extend the example in figure 5.4 to sample items from the *N* highest entropy clusters, not only the single highest entropy cluster, you will have addressed the limitations of expected error reduction in only a few lines of code.

Rather than try to handcraft an algorithm that combines uncertainty sampling and diversity sampling into one algorithm, however, you can let machine learning decide on that combination for you. The original expected error reduction paper was titled "Toward Optimal Active Learning through Sampling Estimation of Error Reduction" and is 20 years old, so this is likely the direction that the authors had in mind. The rest of this chapter builds toward machine learning models for the sampling process itself in active learning.

5.2 Active transfer learning for uncertainty sampling

The most advanced active learning methods use everything that you have learned so far in this book: the sampling strategies for interpreting confusion that you learned in chapter 3, the methods for querying the different layers in your models that you learned in chapter 4, and the combinations of techniques that you learned in the first part of this chapter.

Using all these techniques, you can build a new model with the task of predicting where the greatest uncertainty occurs. First, let's revisit the description of transfer learning from chapter 1, shown here in figure 5.6.

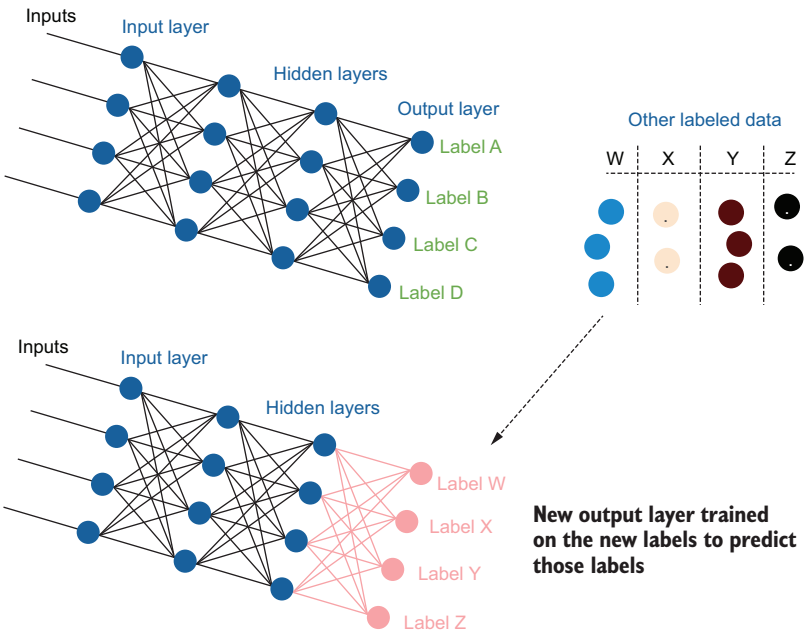


Figure 5.6 We have a model that predicts a label as "A," "B," "C," or "D" and a separate dataset with the labels "W," "X," "Y," and "Z." When we retrain only the last layer of the model, the model is able to predict labels "W," "X," "Y," and "Z," using far fewer human-labeled items than if we were training a model from scratch.

In the example in figure 5.6, you can see how a model can be trained on one set of labels and then retrained on another set of labels by keeping the architecture the same and freezing part of the model, retraining only the last layer in this case. There are many more ways to use transfer learning and contextual models for human-in-the-loop machine learning. The examples in this chapter are variations on the type of transfer learning shown in figure 5.6.

5.2.1 Making your model predict its own errors

The new labels from transfer learning can be any categories that you want, including information about the task itself. This fact is the core insight for active transfer learning: you can use transfer learning to ask your model where it is confused by making it predict its own errors. Figure 5.7 outlines this process.

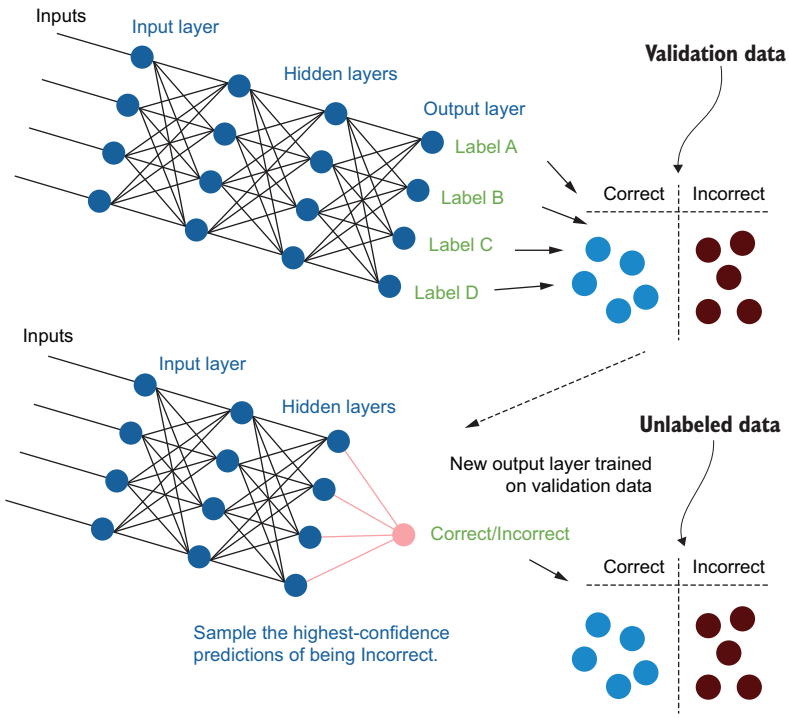


Figure 5.7 Validation items are predicted by the model and bucketed as "Correct" or "Incorrect" according to whether they were classified correctly. Then the last layer of the model is retrained to predict whether items are "Correct" or "Incorrect," effectively turning the two buckets into new labels.

As figure 5.7 shows, this process has several steps:

1. Apply the model to a validation dataset, and capture which validation items were classified correctly and incorrectly. This data is your new training data. Now your validation items have an additional label of "Correct" or "Incorrect."

2 Create a new output layer for the model, and train that new layer on your new training data, predicting your new "Correct" and "Incorrect" labels.
3 Run your unlabeled data items through the new model, and sample the items that are predicted to be "Incorrect" with the highest confidence.

Now you have a sample of items that are predicted by your model as the most likely to be incorrect and therefore will benefit from a human label.

5.2.2 Implementing active transfer learning

The simplest forms of active transfer learning can be built with the building blocks of code that you have already learned. To implement the architecture in figure 5.7, you can create the new layer as its own model and use the final hidden layer as the features for that layer.

Here are the three steps from section 5.2.1, implemented in PyTorch. First, apply the model to a validation dataset, and capture which validation items were classified correctly and incorrectly. This data is your new training data. Your validation items have an additional label of "Correct" or "Incorrect," which is in the (verbosely but transparently named) `get_deep_active_transfer_learning_uncertainty_samples()` method.

Listing 5.6 Active transfer learning

```
correct_predictions = [] # validation items predicted correctly
incorrect_predictions = [] # validation items predicted incorrectly
item_hidden_layers = {} # hidden layer of each item, by id

for item in validation_data:

    id = item[0]
    text = item[1]
    label = item[2]

    feature_vector = feature_method(text)
    hidden, logits, log_probs = model(feature_vector, return_all_layers=True)

    item_hidden_layers[id] = hidden

    prob_dist = torch.exp(log_probs)
    # get confidence that item is disaster-related
    prob_related = math.exp(log_probs.data.tolist()[0][1])

    if item[3] == "seen":
        correct_predictions.append(item)

    elif(label=="1" and prob_related > 0.5) or (label=="0" and prob_related
    ➥ <= 0.5):
        correct_predictions.append(item)
    else:
        incorrect_predictions.append(item)
```

Store the hidden layer for this item to use later for our new model.

The item was correctly predicted, so it gets a "Correct" label in our new model.

The item was incorrectly predicted, so it gets an "Incorrect" label in our new model.

Second, create a new output layer for the model trained on your new training data, predicting your new "Correct" and "Incorrect" labels.

Listing 5.7 Creating a new output layer

```
correct_model = SimpleUncertaintyPredictor(128)
loss_function = nn.NLLLoss()
optimizer = optim.SGD(correct_model.parameters(), lr=0.01)

for epoch in range(epochs):
    if self.verbose:
        print("Epoch: "+str(epoch))
    current = 0

    # make a subset of data to use in this epoch
    # with an equal number of items from each label

    shuffle(correct_predictions) #randomize the order of the validation data
    shuffle(incorrect_predictions) #randomize the order of the validation data

    correct_ids = {}
    for item in correct_predictions:
        correct_ids[item[0]] = True
    epoch_data = correct_predictions[:select_per_epoch]
    epoch_data += incorrect_predictions[:select_per_epoch]
    shuffle(epoch_data)

    # train the final layers model
    for item in epoch_data:
        id = item[0]
        label = 0
        if id in correct_ids:
            label = 1

        correct_model.zero_grad()

        feature_vec = item_hidden_layers[id]
        target = torch.LongTensor([label])

        log_probs = correct_model(feature_vec)

        # compute loss function, do backward pass, and update the gradient
        loss = loss_function(log_probs, target)
        loss.backward(retain_graph=True)
        optimizer.step()
```

The code for training is similar to the other examples in this book.

Here, we use the hidden layer from the original model as our feature vector.

Finally, run your unlabeled data items through the new model, and sample the items that are predicted to be incorrect with the highest confidence.

Listing 5.8 Predicting "Incorrect" labels

```
deep_active_transfer_preds = []

with torch.no_grad():
    v=0
```

The code for evaluation is similar to the others in this book.

```
    for item in unlabeled_data:
        text = item[1]

        # get prediction from main model
        feature_vector = feature_method(text)
        hidden, logits, log_probs = model(feature_vector,
        ➥ return_all_layers=True)

        # use hidden layer from main model as input to model predicting
        ➥ correct/errors
        logits, log_probs = correct_model(hidden, return_all_layers=True)

        # get confidence that item is correct
        prob_correct = 1 - math.exp(log_probs.data.tolist()[0][1])

        if(label == "0"):
            prob_correct = 1 - prob_correct

        item[3] = "predicted_error"
        item[4] = 1 - prob_correct
        deep_active_transfer_preds.append(item)

deep_active_transfer_preds.sort(reverse=True, key=lambda x: x[4])

return deep_active_transfer_preds[:number:]
```

First, we need to get the hidden layer from our original model.

Then we use that hidden layer as the feature vector for our new model.

If you are interested in the disaster-response text classification task, try it with this new method for active transfer learning:

```
> python active_learning.py --transfer_learned_uncertainty=10 --verbose
```

As you can see in this code, we are not altering our original model for predicting whether a message is related to disaster response. Instead of replacing the final layer of that model, we are effectively adding a new output layer over the existing model. As an alternative, you could replace the final layer with the same result.

This architecture is used in this book because it is nondestructive. The old model remains. This architecture prevents unwanted errors when you still want to use the original model, either in production or for other sampling strategies. You also avoid needing the extra memory to have two copies of the full model in parallel. Building a new layer or copying and modifying the model are equivalent, so choose whichever approach is right for your codebase. All this code is in the same file as the methods discussed earlier in this chapter: advanced_active_learning.py.

5.2.3 Active transfer learning with more layers

You don't need to limit active transfer learning to a single new layer or build on only the last hidden layer. As figure 5.8 shows, you can build multiple new layers, and they can connect directly with any hidden layer.

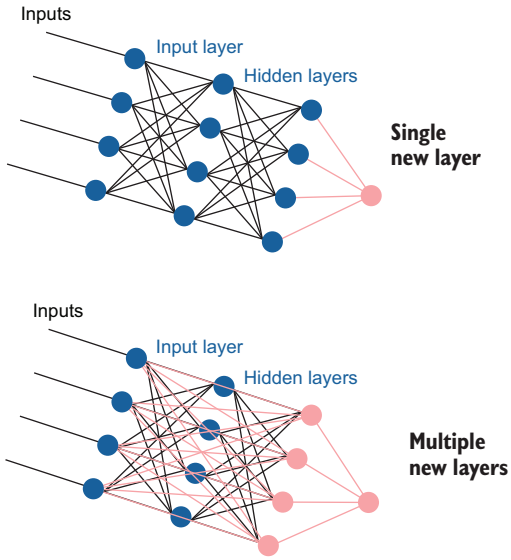


Figure 5.8 More-complicated active transfer learning architectures, using active transfer learning to create a prediction. The top example has a single neuron in the new output layer. The bottom example is a more-complicated architecture, with a new hidden layer that connects with multiple existing hidden layers.

The extension to the more-complicated architecture in figure 5.8 requires only a few lines of extra code. First, the new model to predict "Correct" or "Incorrect" needs a hidden layer. Then that new model will take its features from multiple hidden layers. You can append the vectors from the different layers to one another, and this flattened vector becomes the features for the new model.

If you are familiar with contextual models for natural language processing (NLP) or convolutional models for computer vision, this process is a familiar one; you are extracting the activations of neurons from several parts of your network and flattening into one long feature vector. The resulting vector is often called a *representation* because you are using the neurons from one model to represent your features in another model. We will return to representations in chapter 9, where they are also important for some semi-automated methods for creating training data.

The fact that you *can* build a more complicated model, however, doesn't mean that you *should* build it. If you don't have a lot of validation data, you are more likely to overfit a more-complicated model. It is a lot easier to avoid training errors if you are training only a single new output neuron. Use your instincts about how complicated your model needs to be, based on what you would normally build for that amount of data for a binary prediction task.

5.2.4 The pros and cons of active transfer learning

Active transfer learning has some nice properties that make it suitable for a wide range of problems:

- You are reusing your hidden layers, so you are building models directly based on your model's current information state.

- You don't need too many labeled items for the model to be effective, especially if you are retraining only the last layer (handy if your validation data is not large).
- It is fast to train, especially if you are retraining only the last layer.
- It works with many architectures. You may be predicting labels at document or image level, predicting objects within an image, or generating sequences of text. For all these use cases, you can add a new final layer or layers to predict "Correct" or "Incorrect." (For more on active learning use cases, see chapter 6.)
- You don't need to normalize the different ranges of activation across different neurons, because your model is going to work out that task for you.

The fifth point is especially nice. Recall that with model-based outliers, you need to quantize the activation with the validation data because some of the neurons could be arbitrarily higher or lower in their average activation. It is nice to be able to pass the information to another layer of the neurons and tell that new layer to figure out exactly what weight to apply to the activation of each existing neuron. Active transfer learning also has some drawbacks:

- Like other uncertainty sampling techniques, it can focus too much on one part of the feature space; therefore, it lacks diversity.
- You can overfit your validation data. If there aren't many validation items, your model for predicting uncertainty may not generalize beyond your validation data to your unlabeled data.

The first problem can be partially addressed without additional human labels, as you see later in this chapter in section 5.3.2. This fact is one of the biggest strengths of this approach compared with the other uncertainty sampling algorithms.

The overfitting problem can be diagnosed relatively easily too, because it manifests itself as high confidence that an item is an error. If you have a binary prediction for your main model, and your error-prediction model is 95% confident that an item was classified incorrectly, your main model should have classified that item correctly in the first place.

If you find that you are overfitting and that stopping the training earlier doesn't help, you can try to avoid overfitting by getting multiple predictions, using the ensemble methods from section 3.4 of chapter 3. These methods include training multiple models, using dropouts at inference (Monte Carlo sampling), and drawing from different subsets of the validation items and features.

5.3 Applying active transfer learning to representative sampling

We can apply the same active transfer learning principles to representative sampling. That is, we can adapt our models to predict whether an item is most like the application domain of our model compared with the current training data.

This approach will help with domain adaptation, like the representative sampling methods that you learned in chapter 4. In fact, representative sampling is not too

different. In both chapter 4 and the example in the following sections, you are building a new model to predict whether an item is most representative of the data to which you are trying to adapt your model.

5.3.1 Making your model predict what it doesn't know

In principle, you don't need your existing model to predict whether an item is in your training data or in your unlabeled data. You can build a new model that uses both your training data and your unlabeled data as a binary prediction problem. In practice, it is useful to include features that are important for the machine learning task that you are trying to build.

Figure 5.9 shows the process and architecture for representative active transfer learning, showing how you can retrain your model to predict whether unlabeled items are more like your current training data or more like the application domain for your model.

As figure 5.9 shows, there are few differences from active transfer learning for uncertainty sampling. First, the original model predictions are ignored. The validation and

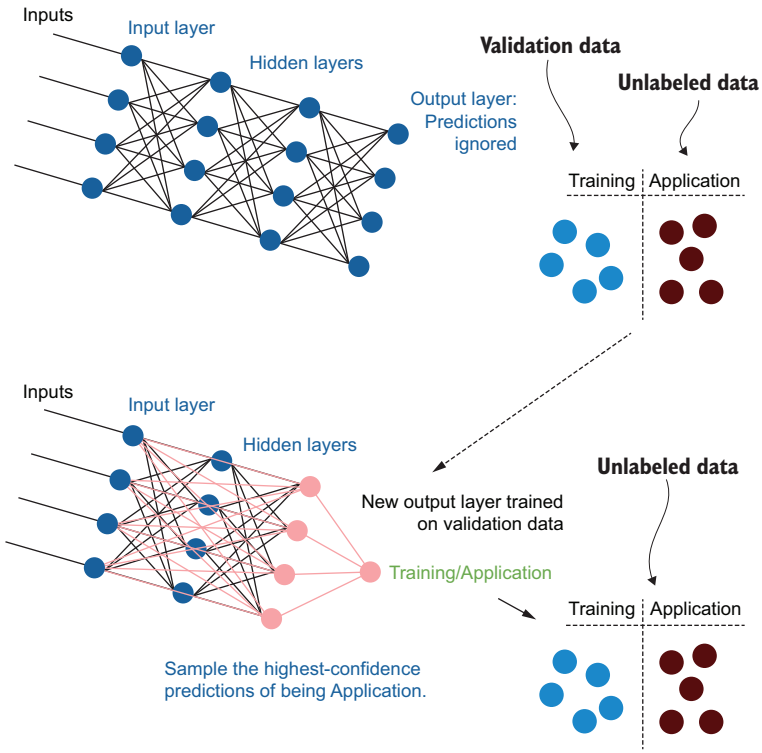


Figure 5.9 We can build a model to sample the items that are most unlike the current training data. To begin, we take validation data from the same distribution as the training data and give it a "Training" label. Then we take unlabeled data from our target domain and give it an "Application" label. We train a new output layer to predict the "Training" and "Application" labels, giving it access to all layers of the model. We apply the new model to the unlabeled data (ignoring the unlabeled items that we trained on), and sample the items that are most confidently predicted as "Application."

unlabeled data can be given labels directly. The validation data is from the same distribution as the training data, so it is given a "Training" label. The unlabeled data from the target domain is given an "Application" label. Then the model is trained on these labels.

Second, the new model should have access to more layers. If you are adapting to a new domain, you may have many features that do not yet exist in your training data. In such a case, the only information that your existing model contains is the fact that these features exist in the input layer as features but have not contributed to any other layer in the previous model. The more-complicated type of architecture will capture this information.

5.3.2 Active transfer learning for adaptive representative sampling

Just like representative sampling (chapter 4) can be adaptive, active transfer learning for representative sampling can be adaptive, meaning that you can have multiple iterations within one active learning cycle, as shown in figure 5.10.

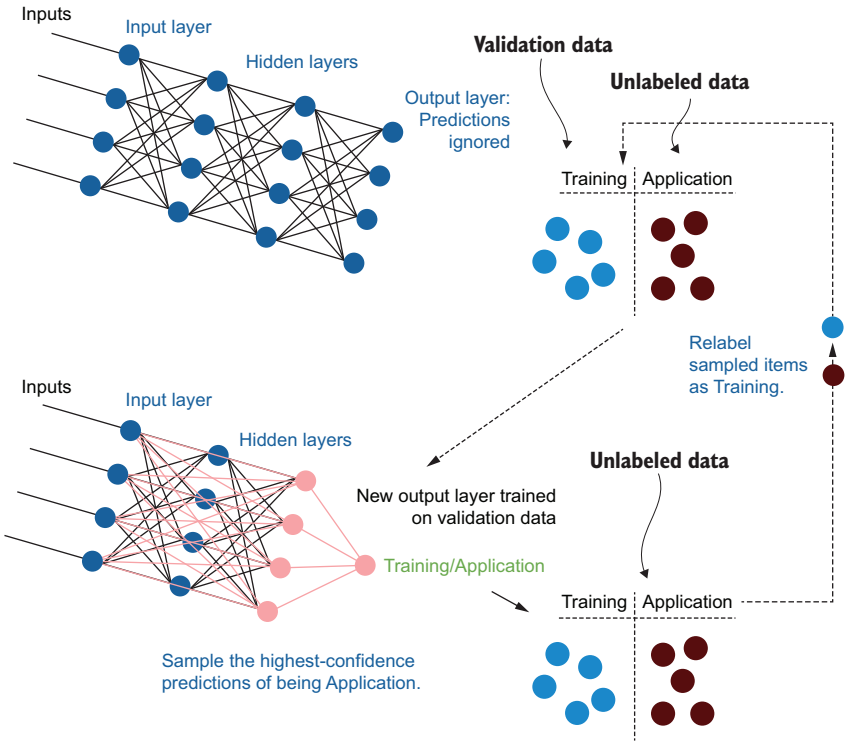


Figure 5.10 Because our sampled items will get a human label later, we can assume that they become part of the training data without needing to know what the label is. To begin, we take validation data from the same distribution as the training data and give it a "Training" label. We take unlabeled data from our target domain and give it an "Application" label. We train a new output layer to predict the "Training" and "Application" labels, giving it access to all layers of the model. We apply the new model to the unlabeled data (ignoring the unlabeled items that we trained on) and sample the items that are most confidently predicted as "Application." We can assume that those items will later get labels and become part of the training data. So we can take those sampled items, change their label from "Application" to "Training," and retrain our final layer(s) on the new dataset.

The process in figure 5.10 starts like the non-adaptive version. We create new output layers to classify whether an item is in the existing training data or in the target domain, sampling the items that are most confidently predicted as "Application." To extend the process to the adaptive strategy, we can assume that the sampled items will later get a label and become part of the training data. So we can take those sampled items, change their label from "Application" to "Training," and retrain our final layer(s) on the new dataset. This process can be repeated until there are no more confident predictions for "Application" domain items, or until you reach the maximum number of items that you want to sample in this iteration of active learning.

5.3.3 The pros and cons of active transfer learning for representative sampling

The pros and cons of active transfer learning for representative sampling are the same as for the simpler representative sampling methods in chapter 4. Compared with those methods, the pros can be more positive because you are using more powerful models, but some of the cons, such as the danger of overfitting, become bigger potential errors.

To summarize those strengths and weaknesses again: representative sampling is effective when you have all the data in a new domain, but if you're adapting to future data that you haven't sampled yet, your model can wind up being stuck in the past. This method is also the most prone to noise of all the active learning strategies in this book. If you have new data that is corrupted text—text from a language that is not part of your target domain, corrupted image files, artifacts that arise from using different cameras, and so on—any of these factors could look different from your current training data, but not in an interesting way. Finally, active transfer learning for representative sampling can do more harm than good if you apply it in iterations after you use uncertainty sampling, because your application domain will have more items away from the decision boundary than your training data. For these reasons, I recommended that you deploy active transfer learning for representative sampling only in combination with other sampling strategies, as you learned in section 5.1.

5.4 Active transfer learning for adaptive sampling

The final algorithm for active learning in this book is also the most powerful; it is a form of uncertainty sampling that can be adaptive within one iteration of active learning. All the uncertainty sampling techniques that you learned in chapter 3 were non-adaptive. Within one active learning cycle, all these techniques risk sampling items from only one small part of the problem space.

Active transfer learning for adaptive sampling (ATLAS) is an exception, allowing adaptive sampling within one iteration without also using clustering to ensure diversity. ATLAS is introduced here with the caveat that it is the least-tested algorithm in this book at the time of publication. I invented ATLAS in late 2019 when I realized that active transfer learning had certain properties that could be exploited to make it adaptive. ATLAS has been successful on the data that I have been experimenting with, but it has not yet been widely deployed in industry or tested under peer review in academia. As you would with any new method, be prepared to experiment to be certain that this algorithm is right for your data.

5.4.1 Making uncertainty sampling adaptive by predicting uncertainty

As you learned in chapter 3, most uncertainty sampling algorithms have the same problem: they can sample from one part of the feature space, meaning that all the samples are similar in one iteration of active learning. You can end up sampling items from only one small part of your feature space if you are not careful.

As you learned in section 5.1.1, you can address this problem by combining clustering and uncertainty sampling. This approach is still the recommended way to think about beginning your active learning strategy; you can try ATLAS after you have that baseline. You can exploit two interesting properties of active transfer learning for uncertainty sampling:

- You are predicting whether the model is correct, not the actual label.
- You can generally expect to predict the labels of your training data items correctly.

Taken together, these two items mean that you can assume that your sampled items will be correct later, even if you don't yet know the labels (figure 5.11).

The process in figure 5.11 starts like the non-adaptive version. We create new output layers to classify whether an item is "Correct" or "Incorrect," sampling the items that are most confidently predicted as "Incorrect." To extend this architecture to the adaptive strategy, we can assume that those sampled items will be labeled later and become part of the training data, and that they will be predicted correctly after they receive a label (whatever that label might be). So we can take those sampled items, change their label from "Incorrect" to "Correct," and retrain our final layer(s) on the new dataset. This process can be repeated until we have no more confidence predictions for "Incorrect" domain items or reach the maximum number of items that we

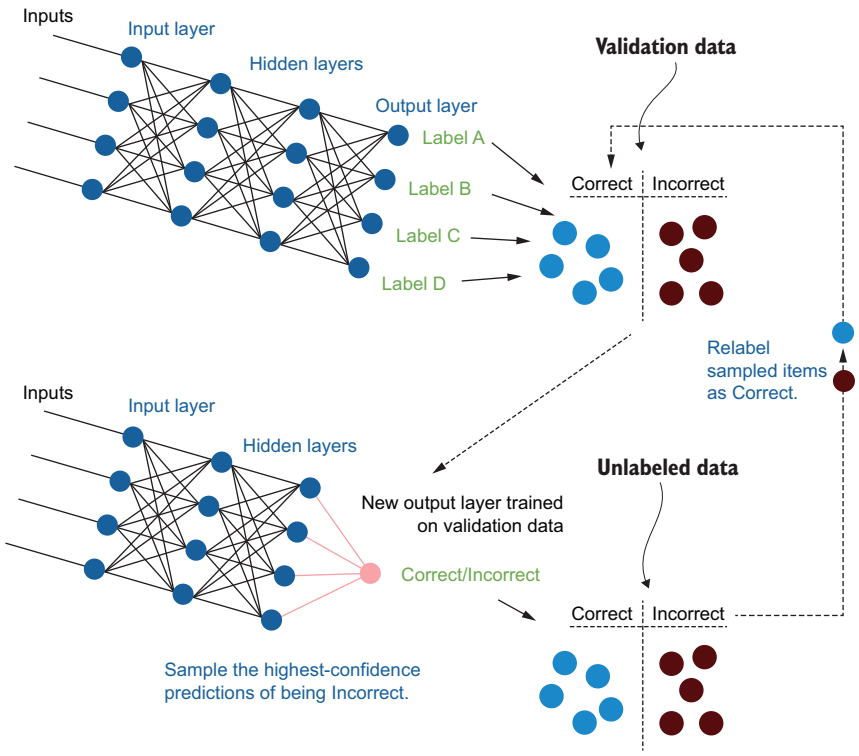


Figure 5.11 Because our sampled items will later get a human label and become part of the training data, we can assume that the model will later predict those items correctly, because models are typically the most accurate on the actual items on which they trained. To begin, validation items are predicted by the model and bucketed as "Correct" or "Incorrect," according to whether they were classified correctly. The last layer of the model is retrained to predict whether items are "Correct" or "Incorrect," effectively turning the two buckets into new labels. We apply the new model to the unlabeled data, predicting whether each item will be "Correct" or "Incorrect." We can sample the most likely to be "Incorrect." Then we can assume that those items will get labels later and become part of the training data, which will be labeled correctly by a model that predicted on that same data. So we can take those sampled items, change their label from "Incorrect" to "Correct," and retrain our final layer(s) on the new dataset.

want to sample in this iteration of active learning. It takes only 10 lines of code to implement ATLAS as a wrapper for active learning for uncertainty sampling.

Listing 5.9 Active transfer learning for adaptive sampling

```
def get_atlas_samples(self, model, unlabeled_data, validation_data,
➥ feature_method, number=100, limit=10000, number_per_iteration=10,
➥ epochs=10, select_per_epoch=100):
"""Uses transfer learning to predict uncertainty within the model

Keyword arguments:
    model -- machine learning model to get predictions from to determine
    ➥ uncertainty
    unlabeled_data -- data that does not yet have a label
    validation_data -- data with a label that is not in the training set, to
    ➥ be used for transfer learning
    feature_method -- the method for extracting features from your data
    number -- number of items to sample
    number_per_iteration -- number of items to sample per iteration
    limit -- sample from only this many items for faster sampling (-1 = no
    ➥ limit)
"""

if(len(unlabeled_data) < number):
    raise Exception('More samples requested than the number of unlabeled
    ➥ items')

atlas_samples = [] # all items sampled by atlas

while(len(atlas_samples) < number):
    samples =
    ➥ self.get_deep_active_transfer_learning_uncertainty_samples(model,
    ➥ unlabeled_data, validation_data, feature_method,
    ➥ number_per_iteration, limit, epochs, select_per_epoch)

    for item in samples:
        atlas_samples.append(item)
        unlabeled_data.remove(item)

        item = copy.deepcopy(item)
        item[3] = "seen" # mark this item as already seen

        validation_data.append(item) # append so that it is in the next
        ➥ iteration

return atlas_samples
```

The key line of code adds a copy of the sampled item to the validation data after each cycle. If you are interested in the disaster-response text classification task, try it with this new method for an implementation of ATLAS:

```
> python active_learning.py --atlas=100 --verbose
```

Because you are selecting 10 items by default (`number_per_iteration=10`) and want 100 total, you should see the model retrain 10 times during the sampling process. Play around with smaller numbers per iteration for a more diverse selection, which will take more time to retrain.

Although ATLAS adds only one step to the active transfer learning for uncertainty sampling architecture that you first learned, it can take a little bit of time to get your head around it. There aren't many cases in machine learning in which you can confidently give a label to an unlabeled item without human review. The trick is that we are not giving our items an actual label; we know that the label will come later.

5.4.2 The pros and cons of ATLAS

The biggest pro of ATLAS is that it addresses both uncertainty sampling and diversity sampling in one method. This method has another interesting advantage over the other methods of uncertainty sampling: it won't get stuck in inherently ambiguous parts of your feature space. If you have data that is inherently ambiguous, that data will continue to have high uncertainty for your model. After you annotate the data in one iteration of active learning, your model might still find the most uncertainty in that data in the next iteration. Here, our model's (false) assumption that it will get this data right later helps us. We need to see only a handful of ambiguous items for ATLAS to start focusing on other parts of our feature space. There aren't many cases in which a model's making a mistake will help, but this case is one of them.

The biggest con is the flip side: sometimes, you won't get enough labels from one part of your feature space. You won't know for certain how many items you need from each part of your feature space until you get the actual labels. This problem is the equivalent of deciding how many items to sample from each cluster when combining clustering and uncertainty sampling. Fortunately, future iterations of active learning will take you back to this part of your feature space if you don't have enough labels. So it is safe to underestimate if you know that you will have more iterations of active learning later.

The other cons largely come from the fact that this method is untested and has the most complicated architecture. You may need a fair amount of hyperparameter tuning to build the most accurate models to predict "Correct" and "Incorrect." If you can't automate that tuning and need to do it manually, this process is not an automated adaptive process. Because the models are a simple binary task, and you are not retraining all the layers, the models shouldn't require much tuning.

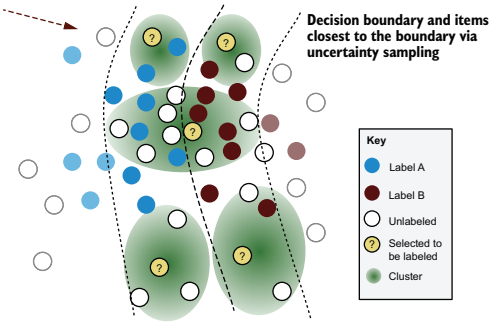
5.5 Advanced active learning cheat sheets

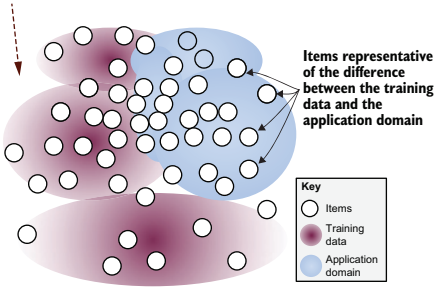
For quick reference, figures 5.12 and 5.13 show cheat sheets for the advanced active learning strategies in section 5.1 and the active transfer learning techniques in sections 5.2, 5.3, and 5.4.

Advanced active learning cheat sheet

Supervised machine learning models have two types of errors that can be fixed with more labeled data: errors that the models know about and errors that the models don't yet know about. *Uncertainty sampling* is the *active learning* strategy to find the known errors, and *diversity sampling* is the strategy to find the unknown errors. This cheat sheet has 10 common methods to combine uncertainty sampling and diversity sampling. See my cheat sheets on each for background: http://bit.ly/uncertainty_sampling | http://bit.ly/diversity_sampling

1. Least confidence sampling with clustering-based sampling: Sample items that are confusing to your model and then cluster those items to ensure a diverse sample.
2. Uncertainty sampling with model-based outliers: Sample items that are confusing to your model and within those find items with low activation in the model.
3. Uncertainty sampling with model-based outliers and clustering: Combine methods 1 and 2.
4. Representative cluster-based sampling: Cluster your data to capture multinodal distributions and sample items that are most like your target domain.





5. Sampling from the highest-entropy cluster: Cluster your unlabeled data and find the cluster with the highest average confusion for your model.
6. Uncertainty sampling and representative sampling: Sample items that are both confusing to your current model and the most like your target domain.
7. Model-based outliers and representative sampling: Sample items that have low activation in your model but are relatively common in your target domain.

8. Clustering with itself for hierarchical clusters: Recursively cluster to maximize the diversity.
9. Sampling from the highest-entropy cluster with margin of confidence sampling: Find the cluster with the most confusion and then sample for the maximum pairwise label confusion within that cluster.
10. Combining ensemble methods and dropouts with individual strategies: Aggregate results that come from multiple models or multiple predictions from one model via Monte Carlo dropouts aka Bayesian deep learning.

Tip: Treat the individual active learning methods as building blocks to be combined:

Uncertainty sampling and diversity sampling work best in combination.

While academic papers about combining uncertainty sampling and diversity sampling focus on single metrics that combine the two, in practice you can simply chain the methods: apply one method to get a large sample and then refine that sample with another method.

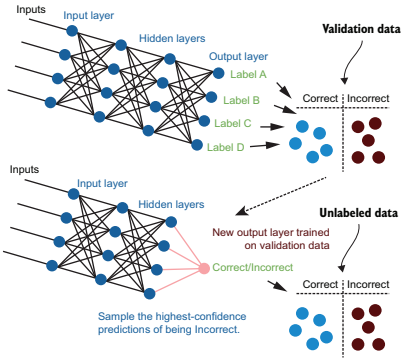
Robert (Munro) Monarch. *Human-in-the-Loop Machine Learning*, Manning Publications. http://bit.ly/huml_book
See the book for more details on active learning building blocks and advanced methods for combining them, with open source implementations in PyTorch. robertmunro.com | @WWRob

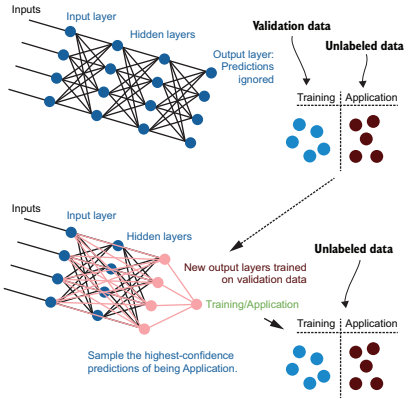
Figure 5.12 Advanced active learning cheat sheet

Active transfer learning cheat sheet

Supervised machine learning models can combine active learning and transfer learning to sample the optimal unlabeled items for human review. Transfer learning tells us whether our model will correctly predict the label of an item and which item looks most like data from our application domain. This cheat sheet builds on principles of uncertainty sampling and diversity sampling: http://bit.ly/uncertainty_sampling | http://bit.ly/diversity_sampling

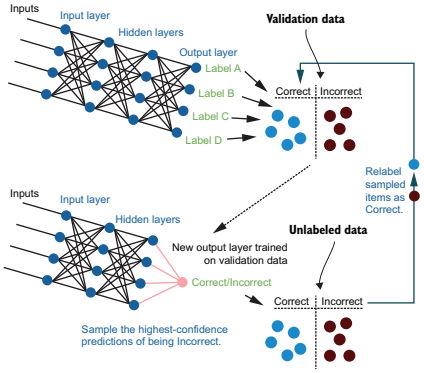
Active transfer learning for uncertainty sampling: Validation items are predicted by the model and relabeled as Correct or Incorrect according to whether they were predicted correctly. The last layer of the model is then retrained to predict whether items are Correct or Incorrect. Unlabeled items can now be predicted by the new model as to whether our initial model will give Correct or Incorrect predictions, and we sample the most likely to be Incorrect.





Active transfer learning for representative sampling: To adapt to new domains, we retrain our model to predict whether an unlabeled item looks more like validation data from the distribution of our current training data or data from our application domain. ***Tip:*** Allow the new model to see all layers to minimize bias from your current model state.

Active transfer learning for adaptive sampling (ATLAS): We can make our models adaptive by assuming that our items will get a human label later, even if we don't yet know what that label will be. We assume that our model will predict items like these correctly after it has been trained on them. So we can continually retrain our model with our samples. ATLAS therefore addresses both uncertainty sampling and diversity sampling in a single adaptive system.



Robert (Munro) Monarch. *Human-in-the-Loop Machine Learning*, Manning Publications. http://bit.ly/huml_book
See the book for more details on active learning advanced active learning, and Active transfer learning, with open source implementations in PyTorch. robertmunro.com | @WWRob

Figure 5.13 Active transfer learning cheat sheet

5.6 Further reading for active transfer learning

As you learned in the chapter, there is little existing work on the advanced active learning techniques in which one method is used to sample a large number of items and a second method is used to refine the sample. Academic papers about combining uncertainty sampling and diversity sampling focus on single metrics that combine the two methods, but in practice, you can simply chain the methods: apply one method to get a large sample and then refine that sample with another method. The academic papers tend to compare the combined metrics to the individual methods in isolation, so they will not give you an idea of whether they are better than chaining the methods together (section 5.1).

The active transfer learning methods in this chapter are more advanced than the methods currently reported in academic or industry-focused papers. I have given talks about the methods before publishing this book, but all the content in those talks appears in this chapter, so there is nowhere else to read about them. I didn't discover the possibility of extending active transfer learning to adaptive learning until late 2019, while I was creating the PyTorch library to accompany this chapter. After this book is published, look for papers that cite ATLAS for the up-to-date research.

If you like the fact that ATLAS turns active learning into a machine learning problem in itself, you can find a long list of interesting research papers. For as long as active learning has existed, people have been thinking about how to apply machine learning to the process of sampling items for human review. One good recent paper that I recommend is "Learning Active Learning from Data," by Ksenia Konyushkova, Sznitman Raphael, and Pascal Fua (http://mng.bz/Gxj8). Look for the most-cited works in this paper and more recent work that cites this paper for approaches to active learning that use machine learning. For a deep dive, look at the PhD dissertation of Ksenia Konyushkova, the first author of the NeurIPS paper, which includes a comprehensive literature review.

For an older paper that looks at ways to combine uncertainty and representative sampling, I recommend "Optimistic Active Learning Using Mutual Information," by Yuhong Guo and Russ Greiner (http://mng.bz/zx9g).

Summary

- You have many ways to combine uncertainty sampling and diversity sampling. These techniques will help you optimize your active learning strategy to sample the items for annotation that will most help your model's accuracy.
- Combining uncertainty sampling and clustering is the most common active learning technique and is relatively easy to implement after everything that you have learned in this book so far, so it is a good starting point for exploring advanced active learning strategies.
- Active transfer learning for uncertainty sampling allows you to build a model to predict whether unlabeled items will be labeled correctly, using your existing

model as the starting point for the uncertainty-predicting model. This approach allows you to use machine learning within the uncertainty sampling process.

- Active transfer learning for representative sampling allows you to build a model to predict whether unlabeled items are more like your target domain than your existing training data. This approach allows you to use machine learning within the representative sampling process.
- ATLAS allows you to extend active transfer learning for uncertainty sampling so that you are not oversampling items from one area of your feature space, combining aspects of uncertainty sampling and diversity sampling into a single machine learning model.

6 Applying active learning to different machine learning tasks

This chapter covers

- Calculating uncertainty and diversity for object detection
- Calculating uncertainty and diversity for semantic segmentation
- Calculating uncertainty and diversity for sequence labeling
- Calculating uncertainty and diversity for language generation
- Calculating uncertainty and diversity for speech, video, and information retrieval
- Choosing the right number of samples for human review

In chapters 3, 4, and 5, the examples and algorithms focused on document-level or image-level labels. In this chapter, you will learn how the same principles of uncertainty sampling and diversity sampling can be applied to more complicated computer vision tasks such as object detection and semantic segmentation (pixel

labeling) and more complicated natural language processing (NLP) tasks such as sequence labeling and natural language generation. The general principles are the same, and in many cases, there is no change at all. The biggest difference is how you sample the items selected by active learning, and that will depend on the real-world problem that you are trying to solve.

Most real-world machine learning systems use tasks that are more complicated than document-level or image-level label predictions. Even problems that sound simple tend to require advanced active learning techniques when you dive into them. Imagine that you are building a computer vision system to help with agriculture. You have smart tractors with cameras that need to distinguish seedlings from weeds so that the tractors can efficiently and accurately apply fertilizer and herbicides. Although weeding fields is one of the most common and repetitive tasks in human history, you need object detection within an image, not image-level labels, to automate this task.

Also, your model has different kinds of confusion. In some cases, your model knows that an object is a plant but can't decide whether that plant is a seedling or a weed. In other cases, your model isn't certain whether some new object is a plant because all kinds of small objects can find their way onto a field. You need uncertainty sampling for the seedling/weed distinction combined with diversity sampling to identify new objects.

Finally, your camera is capturing up to 100 plants in every image, so you have to decide how to resolve the image-level confusion with the object-level confusion. Do you prioritize human review when one object in the image is very confusing or when 100 objects are a little confusing? Do you prioritize the correct label for the type of object or the accuracy of the object outline? Any of these types of error could be the most important for the problem you are addressing, so you need to decide how to map your real-world problem to the right sampling and evaluation strategy. Therefore, even though you are automating one of the most common and repetitive tasks in history, you need advanced active learning techniques to solve the problem.

6.1 Applying active learning to object detection

Until now, we have looked at relatively simple machine learning problems: making predictions about whole images (image labeling) or whole pieces of text (document labeling). For many problems, however, more fine-grained predictions are needed.

You may want to identify only certain objects within an image, for example, so you care more about uncertainty and diversity in the objects than in the backgrounds. Our example at the start of the chapter is like this: you care about identifying weeds more than identifying the field that surrounds them. To the extent that you care about the background, you care only so that you can distinguish the weeds from different backgrounds.

For these examples, you want to employ active learning strategies that also focus on the areas that you care about. Sometimes, you get this focus for free; your models are concentrating on the areas that you care about, so you won't often need to change

anything in the approaches that you learned for image and document labeling. In other cases, you need to crop/mask your data to the areas that you care about and be careful that you are not introducing bias in that process. For the next few sections of this chapter, we will go over some kinds of machine learning problems and see how the active learning strategies that you have already learned can be adapted to them.

Figure 6.1 illustrates a problem for identifying uncertainty and diversity in object detection tasks. Assume that this task uses the same example image from chapter 3, but whereas in chapter 3, we wanted only to predict a label for the image, now we want to identify specific objects within an image and place a bounding box around those images. As figure 6.1 shows, the object we care about—the bicycle—is only a tiny fraction of the pixels in the bounding box that surrounds it.

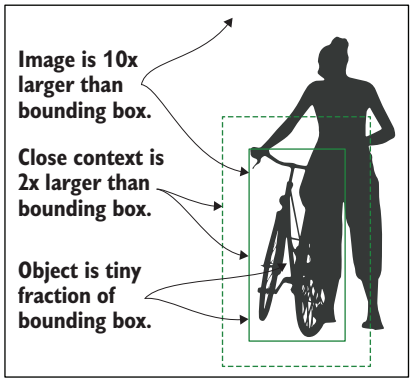


Figure 6.1 An illustration of the problem of identifying uncertainty and diversity in object detection tasks. The object we care about—the bicycle—is a small percentage of the pixels in the bounding box that surrounds it. Even a modest amount of context is twice the number of pixels, and the image as a whole is 10 times the number of pixels of the bounding box. Therefore, if we tried to calculate uncertainty or diversity across the whole image, we would risk focusing on a lot of irrelevant information.

The edge of an object is often where the most information is, but increasing the context by 20% will almost double the total amount of pixels at which we are looking. The image as a whole is 10 times the number of pixels of the bounding box. Therefore, if we tried to calculate uncertainty or diversity across the whole image, we would risk focusing on a lot of irrelevant information. Although we can use the uncertainty sampling and diversity sampling techniques that we learned in chapters 4 and 5, we want to focus that uncertainty and diversity on the areas about which we care most.

The rest of this section will cover how to calculate uncertainty and diversity. You get uncertainty fairly easily from your models; the highest uncertainty will tend to be in your objects, not in the background. For diversity, you want to focus primarily on diversity in areas that are also uncertain.

6.1.1 Accuracy for object detection: Label confidence and localization

You have two tasks here: object detection and object labeling. You should apply different types of uncertainty and diversity to both tasks:

- Labeling each object (bicycle, human, pedestrian, and so on)
- Identifying the boundaries of objects in an image

The confidence for each task is

- Object label confidence (confidence that the label is correct)
- Object localization confidence (confidence that the bounding box is correct)

If you get a confidence score from your object detection algorithm, your confidence score is most likely *only* the object label confidence. The majority of object detection algorithms used today use convolutional neural networks (CNNs) and rely on regression to arrive at the right bounding box. All these algorithms return the label confidence, but few return a score from the regression that arrived at the bounding box itself.

You can determine label accuracy the same way that you determine image-and document-level accuracy: by looking at some variation on F-score or area under the curve (AUC), as you learned in earlier chapters and the appendix. Intersection over union (IoU) is the most common metric for determining localization accuracy. If you've worked in computer vision before, you are aware of IoU already. Figure 6.2 shows an example of IoU, calculating accuracy as the area where the predicted and actual bounding box intersect, divided by the total area covered by those two boxes.

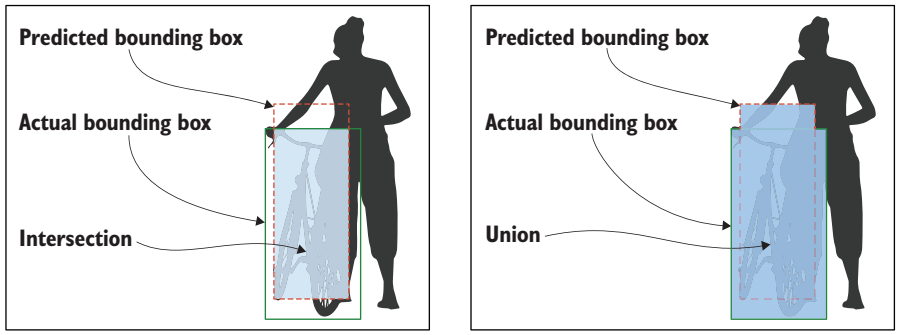


Figure 6.2 An example of IoU for measuring the accuracy of a bounding box. The accuracy is calculated as the area that intersects the predicted bounding box with the actual bounding box, divided by the area that is the union of the two boxes.

IoU is also used in active learning for object detection, so it's important to learn (or refresh your knowledge) before you jump into uncertainty sampling and diversity sampling for object detection. In terms of the accuracy metrics that we've already looked at, IoU is more strict in that it tends to have lower values over the same data. Think of IoU in terms of the amount of area (or pixels) that is correct or incorrectly predicted:

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

$$\text{recall} = \frac{\text{true positives}}{\text{true positives} + \text{false positives}}$$

$$\text{F-score} = \frac{2 \cdot \text{precision} \cdot \text{recall}}{\text{precision} + \text{recall}}$$

$$\text{IoU} = \frac{\text{true positives}}{\text{true positives} + \text{false positives} + \text{false negatives}}$$

Like F-score, IoU combines both types of errors: false positives and false negatives. IoU is always lower than F-score except in the trivial case of 100% accuracy. F-score tends to be more popular in NLP, and IoU is used almost exclusively in computer vision. You'll see AUC in the literature for most machine learning fields, although AUC is not used as often as it should be in NLP and computer vision.

You will also see mean average precision (mAP) in the computer vision literature. mAP is a different kind of curve from AUC, but with a similar idea. For mAP, you rank the items by precision and then plot by recall, creating a precision-recall curve, and the average precision is the area under that curve. This application of mAP requires a threshold at which an object is "correct"—often, an IoU of 0.5 or 0.75. The exact threshold calculation of mAP tends to vary and is often defined specifically for different datasets and use cases. For a highly calibrated task such as autonomous driving, you obviously want much more than 0.50 IoU to call the prediction correct. It is not important to know any of the mAP calculations for this book; it is sufficient to be aware of this other, common accuracy metric that will be task-specific.

For active learning, you generally want to employ a strategy that samples from both localization confidence and label confidence. You need to determine how much you want to focus on each type. Although your label and IoU accuracy will help you determine where you need to focus the most attention, your focus will also depend on the application that you are building.

Suppose that you are deploying our example model to detect pedestrians, cars, bicycles, and other objects on roads. If your application is designed to predict collisions, localization is most important; it doesn't matter whether you get the label wrong as much as it matters whether your object boundaries are off. If your application is meant to identify different traffic volumes, however, the exact boundaries of the objects aren't important, but the labels are important because you want to be precise about knowing exactly how many cars, pedestrians, and other objects are seen.

So you could have the same model deployed in the same place, but depending on the use case, you might focus your active learning and data annotation strategies on either localization or confidence. Determine what is most important for your use case, and focus your active learning strategy accordingly.

6.1.2 Uncertainty sampling for label confidence and localization in object detection

You can use label confidence for uncertainty sampling, as you did for image-level labels in chapter 3. Your object detection model will give a probability distribution, and you can apply least confidence, margin of confidence, ratio of confidence, entropy, or an ensemble model to determine your uncertainty for the label prediction.

For localization confidence, an ensemble model is your best option, combining multiple deterministic predictions into a single one that can be interpreted as confidence. Figure 6.3 shows an example. You can use either of two approaches: a true ensemble or dropouts within one model, both of which you learned in chapter 3.

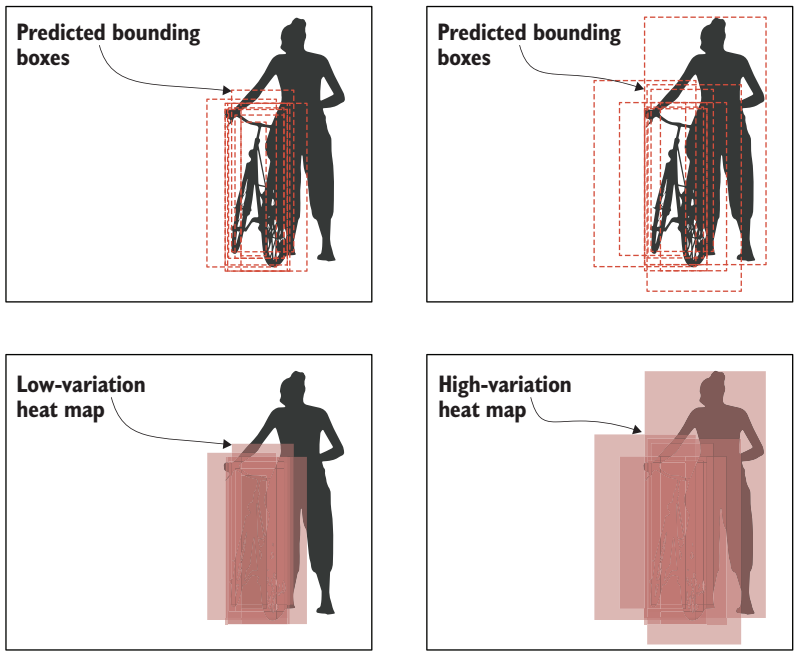


Figure 6.3 An example prediction heat map for an object, showing low variation (left) and high variation (right). The high variation is evidence of more uncertainty in the model; therefore, the right example is a good candidate for human evaluation. You can generate multiple predictions by using ensemble models, getting predictions from multiple models and changing the parameters, using a subset of features or a subset of items, or introducing random variation into your models in some other way. Within a single model, you can generate multiple predictions for a single item by using a dropout over a random selection of neurons for each prediction (known as Monte Carlo dropouts). You can also combine both methods: create an ensemble of models, and use dropouts for multiple predictions per model.

For a true ensemble, you get predictions from multiple models and ensure that those predictions will vary by using different hyperparameters for different models, training on a subset of features for each model, training on a subset of items for each model, or introducing random variation into your training runs in other ways, such as shuffling the order of training items.

For a single model, you can generate multiple predictions by using a dropout over a random selection of neurons for each prediction (aka Monte Carlo dropout). This approach is faster and easier than building multiple models and surprisingly effective for how simple it is. You could also combine both methods: train a handful of models with different parameters and then apply dropouts to each model.

The uncertainty is calculated from the average IoU across all predictions. This calculation naturally gives a [0, 1] range, so there is no need to normalize it. Divide by the number of models, not predictions. Some models may not make a prediction, and this information is important: treat all nonpredictions as IoU=0.

Now that you have an uncertainty score for each bounding box, you can sample the bounding boxes with the greatest uncertainty for human review. If you are using ensemble methods or dropouts for localization, you can use them for label confidence, in place of or in addition to the other uncertainty sampling methods.

6.1.3 Diversity sampling for label confidence and localization in object detection

For diversity sampling, we need to solve the problem that we introduced at the start of this chapter: we care about diversity in the objects more than diversity in the background. The simplest solution is to crop images to the predicted bounding boxes and then apply diversity sampling, but there are more sophisticated variations, which we'll cover in this section. Chapter 4 introduced three types of diversity sampling:

- Model-based outlier sampling
- Cluster-based sampling
- Representative sampling
- Sampling for real-world diversity

For model-based outliers and real-world diversity, you don't necessarily need to do anything beyond what you've already learned for image-level labels:

- You can apply model-based outlier detection to an object detection problem in the same way that you apply it to an image labeling problem.
- You can sample for real-world diversity in an object detection problem in the same way that you sample for an image labeling problem.

For model-based outliers, the hidden layers focus on both the labeling and localization problems, so your neurons will be capturing information primarily about the objects and labels. You can crop the images to the predicted objects and then look for model-based outliers, but the small amount of neurons dedicated to the background might be interesting for diversity, so you could lose something in this case.

For diversity sampling, the principles from chapter 4 also apply. You need to combine all the active learning methods to ensure fair data across real-world demographics. The background can matter in this case too, because you can erroneously model the context of objects rather than the objects themselves if you are not careful. (See the following sidebar.) For object detection, you may want to ensure that your data tries to balance each type of object across factors including the type of camera, zoom, time of day, and weather. Even for highly controlled settings, such as medical imaging, I've seen systems limited by training on data from only a small number of patients and only one type of imaging machine, introducing unwanted real-world bias.

Is your model really ignoring the background?

This book assumes that your model focuses on the objects and not the background. Sometimes, however, your model might be using background information erroneously. If you took photographs of bicycles only in bicycle lanes, for example, your model might be predicting bicycle lanes and be essentially blind to bicycles in other contexts. Or it might rely on bicycle lanes only when the lanes are present, which is still non-ideal, as the model is not generalizing its knowledge of bikes in those contexts to other backgrounds.

An influential recent paper on model interpretability presents another example. The authors created what looked like an accurate model for distinguishing wolves from huskies,[a] but used only photos of wolves in snow and huskies not in snow. They showed that the model was predicting whether snow was in the background, not the actual animals! This problem is a bigger one with image-level labeling, because with object detection, you are also explicitly forcing the model to learn the outline of the object itself, making it hard for your model to focus on the background. But the problem can occur to some degree in any machine learning problem in which context needs to be controlled for.

The solution is better sampling for real-world diversity, ensuring that the contexts are as diverse as possible across all the labels and objects that you care about. If you are worried about this problem with your model, here is how to diagnose it: use a method to find out which pixels are important features for your predictions (such as LIME, the method in the huskies/wolves paper, or the Captum interpretability library, which is in PyTorch as of October 2019) and then measure what percentage of the pixels fall outside the bounding boxes on your validation data. The images with the highest scores are the most likely to be problematic. Look at these images to identify any patterns in what the model is focusing on outside your bounding boxes.

[a] "Why Should I Trust You?": Explaining the Predictions of Any Classifier," by Marco Tulio Ribeiro, Sameer Singh, and Carlos Guestrin (https://www.kdd.org/kdd2016/papers/files/rfp0573-ribeiroA.pdf).

For cluster-based sampling and representative sampling, the focus should be on the objects themselves, not the backgrounds. If your background makes up 90% of your images, such as the example in figure 6.1 (repeated in figure 6.4), it will make up 90% of the influence on what determines a cluster or is representative. Figure 6.1 also contains a relatively large object that takes up half of the height of the frame. But in many cases, the example is more like the second image in figure 6.4, in which the object takes up fewer than 1% of the pixels.

In figure 6.4, the bicycle itself and immediate context are enough to identify the object as a bicycle. Some information outside the box probably can help determine the scale and context in which bicycles appear more often, but not much.

Therefore, the area around each predicted object should be cropped. Because your model isn't 100% accurate, you need to ensure that you are capturing the object.

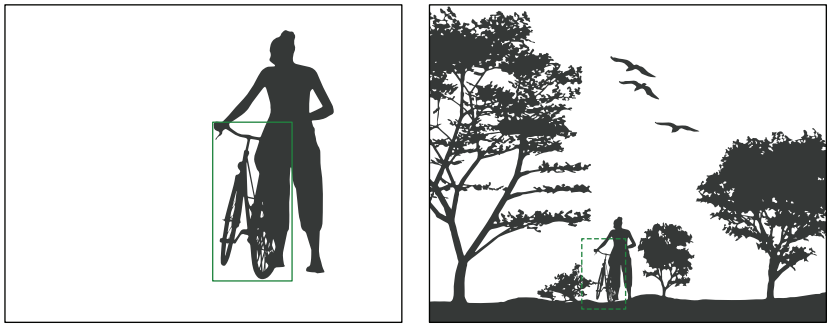


Figure 6.4 An example of an object—a bicycle—in an image in which 99% of the image is not the bicycle. The bicycle and immediate context captured by the dotted line should be enough for identifying that the object is a bicycle. With some strategies, such as representative sampling and clustering, we need to crop or mask the image to target these areas.

Use your method from uncertainty sampling (ensembles or dropout) to make multiple predictions. Then do either of the following:

- *Crop at a given threshold.* You might create the smallest cropping that captures 90% of the predicted bounding boxes for an object, for example.
- *Use every predicted box for the same object, and weight the boxes.* You might apply representative sampling to every predicted box and then average across all the representative sampling in which the weighted average is determined by the average IoU of each box to all others.

As an alternative to cropping the image, you can ignore the pixels outside your contextual box—a process called *masking*. You can think of a mask for a model trained on pixel inputs as being a dropout on the first layer because you are ignoring some input neurons (pixels).

How important is context?

There are a few exceptions in computer vision in which the context *is* important. I've encountered only one of these exceptions on multiple occasions: identifying empty supermarket shelves to help restocking. An empty space (the object) also needed context such as adjacent items and a price tag beneath the empty shelf. Otherwise, it wasn't clear to the model whether the shelf was meant to be empty or whether there were meant to be products on the shelf.

Unless you have a use case like this one, essentially labeling a hole according to context, keep the boxes as tight as possible for clustering and representative sampling. You can capture broader diversity in contexts by using some diversity sampling for entire images.

Depending on your use case, you also want to resize the images. If you've worked in computer vision, you already have your tools of choice for resizing programmatically. It's probably not important that our bicycle is at the bottom of the photo, for example, so you can normalize your data by cropping each prediction to be the entire image and then normalize further by scaling all your sample images to be the same dimensions. As a general rule, make the crop/mask decision based on how you want to encode data for clustering and representative sampling:

- If you are using pixels as features or a separate tool to create features, crop the images, and consider whether you should also resize them.
- If you are using the hidden layer(s) from the *same* model that you are using for object detection, you can mask the images and not move or resize them. Your features can capture the similarities in objects at different locations and scales.

Now you have cropped or masked images that you can use for clustering and representative sampling! With each cropped or masked object in an image, you can apply clustering or representative sampling. You apply cluster-based sampling and representative sampling as you learned in chapter 4.

Make sure that you are sampling images with a different number of objects per image. If you find that you are sampling only images with a small or large number of objects, you are inadvertently introducing bias into your process. In such a case, stratify your sampling. You might sample 100 images that have 1 predicted object, 100 images that have 2 predicted objects, and so on.

6.1.4 Active transfer learning for object detection

You can apply active transfer learning to object detection in the same way that you apply it to image-level labels. You can also apply active transfer learning for adaptive sampling (ATLAS), adapting within one active learning cycle because you can assume that the first objects you sample will be corrected later by human labelers, even if you don't know what those labels are.

Regardless of the type of neural architecture you use for object detection, you can use the hidden layer(s) as the features for a binary "Correct"/"Incorrect" model that you train on validation data. As an interesting extension, instead of a binary "Correct"/"Incorrect" task, you could calculate the IoU of the validation data and create a model that predicts the IoU. That is, you can predict a continuous value instead of the binary "Correct"/"Incorrect." This process could be as simple as making the final layer a regression task instead of a classification task and having that regression task model the IoU of each validation item. This extension could involve changing only one or two lines of code from the ATLAS example in chapter 5.

6.1.5 Setting a low object detection threshold to avoid perpetuating bias

Set your confidence threshold for object detection low, whatever method you use. You don't want to find only objects that are similar to those that already exist in your data, which would perpetuate bias toward those types of objects.

You may find that a low threshold produces too many candidates. You might get 100 predicted images at 50% or greater confidence but 10,000 at 10% confidence, and most of those 10,000 predictions are the background (false positives that are not objects). So you might be tempted to raise the threshold in this case. Don't.

Unless you are confident that you have set the threshold correctly to get near-perfect recall in your predictions, you're still in danger of perpetuating bias in your models. Instead, stratify by confidence and sample from within each:

- Sample 100 predicted images at 10–20% confidence.
- Sample 100 predicted images at 20–30% confidence.
- Sample 100 predicted images at 30–40% confidence, and so on.

Figure 6.5 shows an example of the general strategy for stratifying by confidence.

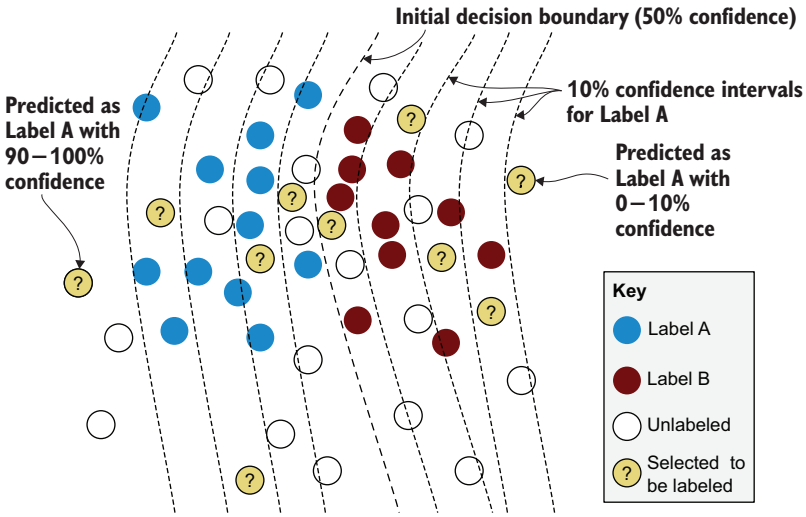


Figure 6.5 Stratifying by confidence: sampling an equal number of items at 0–10% confidence, 10–20% confidence, and so on up to 90–100% confidence. In this example, one item is sampled from each 10% confidence interval for label A. Stratifying by confidence helps most when you have a large number imbalance between labels.

As figure 6.5 shows, you can sample the same number of items at different confidence intervals. This strategy is helpful for a task such as object detection because most of your images will not contain objects that you care about. Using a sample strategy that is stratified by confidence, you'll be spending most of your time sampling high-confidence objects and still have a selection of lower-confidence ones. Note that

although it feels like a waste of time to identify non-objects, that's not the case for your machine learning algorithm. Learning what is not an object but is currently predicted as an object with nontrivial confidence can be as important for your model's accuracy as learning new objects.

This kind of stratification is important for avoiding bias in your data. You can also try combinations of methods as alternatives to random sampling within each confidence:

- Take the 10,000 objects with 10–20% confidence, apply clustering, and sample the centroids to get the 100 most diverse objects within that sample.
- Take the 10,000 objects with 10–20% confidence and apply representative sampling to get the 100 objects most like your target domain.
- Take the 10,000 objects with 10–20% confidence and sample for model-based outliers to get the 100 objects most unlike the current training data.

Note that you can apply this method for stratifying by confidence to any type of task, not only object detection.

6.1.6 Creating training data samples for representative sampling that are similar to your predictions

Because you are cropping or masking your unlabeled images, you should do the same thing with training data if you are implementing representative sampling. If you use only the perfect bounding boxes from your training data but then the imperfect predictions from your unlabeled data, the "representative" samples could end up being the result of different box sizes and cropping strategies instead of the actual objects. Here are four options, in order of preference:

- Cross-validate your training data. You might split your training data into 10 equal datasets. Iteratively train on each group of nine, and predict bounding boxes on each held out dataset. Combine all the predictions, and use the combination as the training data portion of your corpus for representative sampling.
- Use a validation dataset from the same distribution as your training data, get bounding-box predictions over the validation set, and use those validation bounding boxes as the training data portion of your corpus for representative sampling.
- Predict on the training data and then randomly expand or contract the boxes so that they have the same average variation in your predictions.
- Use your actual boxes from the training data and then randomly expand or contract the boxes so that they have the same average variation in your predictions.

Options 1 and 2 are statistically equally good. If you've got a held-out validation set, the process is a little easier than retraining your entire model, but it won't be the exact data in your training set even though it is as close as possible.

For options 3 and 4, although you can increase the sizes of the bounding boxes so that the average is the same, you won't be able to match the kind of errors that you get

in predictions. The predicted bounding-box errors will not be randomly distributed; they will depend on the image itself in ways that are hard to duplicate when you create artificial noise.

6.1.7 Sampling for image-level diversity in object detection

As with any other method, you should always randomly sample some images for review. This sample provides your evaluation data and gives you a baseline for how successful your active learning strategy is.

For a small amount of samples, you can use image-level sampling, which helps diversity in a way that prevents bias more easily than the other methods introduced in this section. If you apply clustering at the whole-image level and find whole clusters with few or no existing training items, you have good evidence that you should get human review for some items in those clusters, because you could be missing something.

If you are introducing new kinds of data to your model (maybe you are using a new camera or collecting from a new location), representative sampling at the image level should help you adapt faster. This strategy also helps you adapt with less bias than trying to implement only object-level active learning when you try to incorporate new data.

If you try object-level methods on different types of data, it will be hard to avoid biasing toward objects you have already seen, as some of those objects might still slip below the threshold you use. Confidence thresholds tend to be least reliable for out-of-domain data.

6.1.8 Considering tighter masks when using polygons

If you are using polygons rather than bounding boxes, as shown in figure 6.6, all the methods still apply. You have one additional option: instead of masking outside the bounding box, you can mask at a certain distance from the nearest polygon edge. In our bicycle example, this approach more closely captures the bicycle itself and not so much of the empty space.

For the same reason, error detection can be more accurate, especially for irregularly shaped objects. You can imagine that a bicycle like the one in figure 6.6 could have a handlebar sticking out in many photos. The bounding-box extension to capture only that handlebar could easily be almost half the box's area, creating a lot of room for error over pixels that aren't part of the object. The next level of complexity in image recognition, after bounding boxes and polygons, is semantic segmentation.

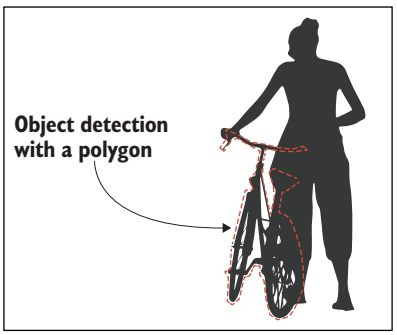


Figure 6.6 An example of object detection using a polygon rather than a bounding box. You can use the same active learning methods for both bounding boxes and polygons, with the extra option to have a closer mask for a polygon.

6.2 Applying active learning to semantic segmentation

Semantic segmentation occurs when the entire image receives a label, with accurate polygon boundaries around all objects. Because this technique labels every pixel in the image, it is also referred to as *pixel labeling*. Figure 6.7 shows an example.

Figure 6.7 An example of semantic segmentation in which every pixel is labeled. This kind of colored photograph is what a lot of semantic segmentation tools look like: a coloring-in exercise. We'll cover those tools later in the book, especially in chapter 10. If you're looking at this image in black and white, the contrasting shades of gray should give you a good idea of what it would look like in color. If the objects receive a label (the four trees are labeled separately, for example), the task is known as *instance segmentation*.

If you are trying to estimate objects that extend behind some other object (occlusion), it is more common to use the bounding-box-type object detection you learned about in section 6.1. It is also more common to paint all objects as a single type with semantic segmentation rather than identify each object separately. Every tree in figure 6.7 is the same color, for example, but the image is not distinguishing one tree from another. These commonalities are not set in stone, however: there are cases in which bounding boxes that ignore occlusion are used, semantic segmentation tries to capture occlusion, and semantic segmentation distinguishes objects (called *instance segmentation*). If a model combines all these methods, it is sometimes known as *panoptic segmentation*, identifying objects and background pixels. All the methods in this chapter should be generic enough that they can apply to any variation on bounding boxes or semantic segmentation.

The methods can apply to other kinds of sensor data, such as the 2D and 3D images that from lidar, radar, or sonar, which are all common for autonomous vehicles. It is also common to collect data outside the range of human vision in the infrared and ultraviolet bands and then shift those results into visible colors for human annotation, which is common in agriculture. Search for "infrared forest" or "ultraviolet flower" photos, and you'll see why: a lot of useful information falls outside human-viewable range! The principles in this section should still apply if additional dimensions and sensor information are involved.

6.2.1 Accuracy for semantic segmentation

Accuracy for semantic segmentation is calculated on a per pixel-level. How many of the pixels are classified correctly, relative to a held-out dataset? You can use all the accuracy metrics that you have learned so far: precision, recall, F-score, AUC, IoU, and micro and macro scores. The right choice for machine learning accuracy depends on your use case.

For determining uncertainty, the macro F-score or macro IoU is often most useful. As with our bounding-box examples, we often have a lot of space in semantic segmentation that we don't care about much, such as the sky and background. You can get into trouble if you have many discontinuous regions. For instance, figure 6.7 probably has more than 100 separate regions of sky between the leaves. By overall size and total number, those sky regions would dominate a micro score on a per-pixel or per-region basis, and tree-leaf confusion would dominate our uncertainty sampling strategies. So assuming that you care about all the labels equally, but not about how much of an image that object happens to take up, use a macro score: the average IoU of each region per label or the average F-score of each pixel per label.

You might also decide to ignore some labels. Perhaps you care only about people and bicycles, so you can choose a macro accuracy value that looks only at those labels. You will still get inputs from errors distinguishing people and bicycles from the background, the ground, and the sky, but not errors between those irrelevant labels. Note that exactly what matters will be specific to your use case. If your task is identifying the amount of forest coverage, the regions between leaves and the sky will matter most!

Use your deployed machine learning model accuracy as your guide for calculating uncertainty. You should make this calculation in either of two ways, depending on whether you weight your labels for your accuracy calculation:

- If you don't weight your labels (you either 100% care or don't care if each label equals absolute weightings), use the same metric that you use for model accuracy to determine where to sample. If you care about confusion for only two labels in your model accuracy, sample only predictions with confusion involving one or both of those labels for active learning.
- If you have an accuracy metric that is weighted, do *not* use the same metric as for model accuracy. Instead, use the stratified sampling methods that you learned in chapter 3. Figure 6.8 shows an example.

As figure 6.8 shows, stratified sampling by label helps focus your active learning strategy on the pixels that matter most. Although you can use stratified sampling for any machine learning problem, semantic segmentation is one of the clearest examples of where it can help.

Note that stratified sampling may diverge from your strategy for evaluating model accuracy. Suppose that you care about Label A nine times as much as you care about Label B. Calculate your model accuracy as 90% × Label A's F-score + 10% × Label B's F-score (a weighted macro F-score). This strategy is fine for model accuracy, but unfortunately, you can't apply weights in a similar way to uncertainty scores, because your weighting will almost certainty rank only Label A items highest, moving them exclusively to the top of your rankings. Instead, use those weights as a ratio of how many to sample. You might sample the 90 most uncertain Label A items and the 10 most uncertain Label B items, for example. This technique is simpler to implement than trying to create a weighted sampling strategy across labels and much more effective. If

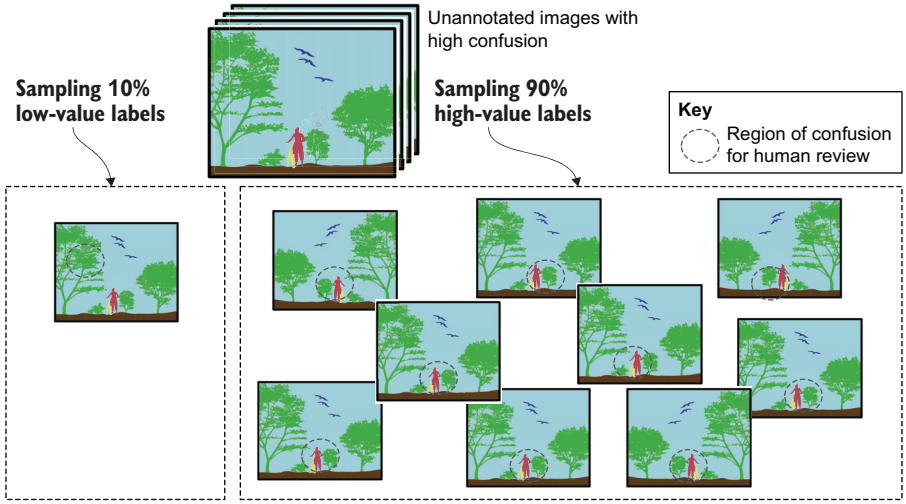


Figure 6.8 An example of stratified sampling by label applied to segmentation. For this example task, assume that we care about errors related to human and bicycle pixels more than we care about errors related to tree and sky pixels. Our active learning sample will consist of a 90:10 split: 90% will be the most confusing samples for the labels that we care about most, and 10% will be about the labels that we don't care about. Note that the pixels bordering the sky and trees vastly outnumber the pixels bordering humans and bicycles, so stratified sampling helps us focus on the errors that we care about most. Therefore, your sampling strategy might diverge from your accuracy evaluation strategy, in which you can simply apply relative weights of 90% and 10% to high- or low-value errors. The uncertainty sampling metrics don't lend themselves as easily to this kind of weighting, so unless you are confident enough in your statistics skills to tune your weighting strategy, use this stratification method.

there are labels that you don't care about, still consider sampling a small number, especially using model-based outliers and representative sampling, because they may be false negatives for the labels you *do* care about.

6.2.2 Uncertainty sampling for semantic segmentation

Most semantic segmentation algorithms are built on variations of CNNs, using softmax to generate a probability distribution across the possible labels for every pixel. So you can calculate the uncertainty on a per-pixel basis, using the methods you learned in chapter 3. Your model is unlikely to make a prediction for every pixel, which would be inefficient, but instead predict regions and select only small (maybe pixel-sized) regions when forced. Know exactly where your predicted confidences are coming from.

As with bounding boxes, the confidence you get from your models probably reflects your label confidence, not the confidence on the borders of your objects. If so, you can derive the localization confidence from the pixel confidences: you know which pixels are next to pixels from a different label, so the aggregate confidence from all boundary pixels is the localization confidence. You may be OK with errors of a few pixels; if so, use this margin of error to determine where you calculate confidence. If you are measuring

the accuracy of your machine learning model by forgiving all errors of less than 3 pixels, for example, do the same for uncertainty, measuring the average uncertainty of the pixels that are 3 pixels away from a border.

For some reason, you may be using a model that does not give a probability distribution for a given label. In that case, you can use ensemble methods and/or dropouts to create multiple predictions and calculate uncertainty as the amount of label agreement across your predictions.

Now that you are sampling only the pixels that you care about and have an uncertainty score for each pixel, you can apply any of the uncertainty sampling algorithms. The simplest way to calculate uncertainty for the entire image is to take the average uncertainty from each of the pixels you care about. If you care mainly about the borders, you can sample the items only within a few pixels of another label.

Depending on your task, you can try metrics other than average if (for example) you want to give the image an uncertainty score that is the maximum uncertainty for any one region. Your ability to focus only on regions within the images will partially depend on your annotation setup. Will someone need to annotate the entire image, or can they annotate only the labels you care about? These considerations are addressed from the annotation point of view in chapter 9.

6.2.3 Diversity sampling for semantic segmentation

You cannot sample for model-based outliers straight from your model for diversity sampling, as you can for object recognition. This approach works with object recognition because you are already forcing the model to focus on the areas that you care about, but a semantic segmentation algorithm is forced to classify every pixel. So you should mask or crop the images to contain only the predicted labels you care about, as outlined in section 6.1, and then apply model-based outliers.

The same is true for clustering and representative sampling: crop or mask the image to the areas you care about and then apply clustering and/or representative sampling. For real-world diversity, the strategy is the same as for bounding boxes: use all the techniques that you know in active learning to sample for diversity across and within the demographics that you care about. See section 6.1 on object detection for more about these methods.

6.2.4 Active transfer learning for semantic segmentation

You can apply active transfer learning to semantic segmentation in the same way that you applied it to image-level labels, but you should use the adaptive method: ATLAS. If you don't use the adaptive version of this algorithm, you could sample confusion exclusively in the areas that you don't care about, such as the division between leaves and the sky when you care mainly about objects on the ground. Note that ATLAS won't completely solve this problem; it might initially sample types of confusion that you don't care about. But it will adapt quickly to assume that those types of confusion are solved and therefore also cover the areas that you care about. If you think about

how many pairwise sets of labels exist in your data and what percentage of those pairs you actually care about, you will have some idea of how successful ATLAS will be out-of-the-box.

To get the most out of ATLAS for semantic segmentation, you can be strategic about how you set up your validation data for transfer learning. If you don't care about errors between leaves and the sky, for example, you can ignore those errors when you run the validation data through the original model to generate your "Correct"/"Incorrect" labels. That way, your model is now predicting errors for only the types of labels you care about.

6.2.5 Sampling for image-level diversity in semantic segmentation

As with object detection, you may want to sample a small number of items from the whole image (especially if you are introducing data from new locations, camera types, and so on) so that you can adapt quickly and find false negatives for the labels that you care about. You could also experiment with loosening the restriction to crop or mask if you are combining methods. You could use representative sampling on the entire image to find images most representative of a new domain or type of image and then sample for the most representative images, apply the mask/crop to those samples, and cluster those samples for diversity. This technique gives you the most diverse selection of items you care about from whole images that are representative of the domain you care about.

6.3 Applying active learning to sequence labeling

Sequence labeling is machine learning applied to labeling spans within a sequence and is one of the most common tasks in NLP. Suppose that you have this sentence (sequence):

"The E-Coli outbreak was first seen in a San Francisco supermarket"

If you are implementing a model to track outbreaks from text reports, you may want to extract information from the sentence, such as the name of the disease, any locations in the data, and the important keywords, as shown in table 6.1.

Table 6.1 An example sequence labels: keyword detection and two types of named entities, diseases, and locations. Label B (beginning) is applied to the beginning of the span, and Label I (inside) is applied to the other words within the span, allowing us to unambiguously distinguish spans that are next to each other, such as "San Francisco" and "supermarket". This process is called *IOB tagging*, in which O (outside) is the nonlabel. (O is omitted from this table for readability.)

	The	E-Coli	outbreak	was	first	seen	in	a	San	Francisco	supermarket
Keywords		B	I						B	I	B
Diseases		B									
Locations									B	I	

In the literature, you most commonly see IOB tagging for spans, as in table 6.1. Notice that you might define spans in different ways for different types of labels. The named entity "E-Coli" is one word, but when we extract the keywords, it's the phrase "E-Coli outbreak." And although "San Francisco" is both an entity (location) and a keyword, the common noun "supermarket" is a keyword but not an entity. Strictly, this process is called *IOB2 tagging*, and IOB uses the B only when there are multiple tokens in a single span. IOB2 is the most common approach that you'll encounter in the literature, sometimes called IOB for short.

Other encodings mark the end of the span rather than the start. This type of encoding is common for whole-sentence segmentation tasks, such as tagging the end of every word and subword span and tagging the end of every sentence. For sentences, the end is tagged because it's a little easier to identify the end of a sentence (typically with punctuation) than the start. The methods in this chapter work for any type of sequence encoding, so the chapter will stick with the IOB2 examples and assume that it is easily adapted if your encoding system is different.

You might also treat some labels as being naturally part of the same task. I have worked a lot in named entity recognition (NER), which considers identifying "Location" and "Disease" as part of the same task, but treats keyword identification as a different task. Even within one task, there can be a lot of variation in how labels are defined. Some popular NER datasets have only four types of entities: "People," "Locations," "Organizations," and "Miscellaneous." By contrast, I once helped build an entity recognition system for a car company that had thousands of entity types; every kind of engine, door, and even headrest had multiple types and names.

Although you might perform a large variety of sequence labeling tasks in NLP, all of them come down to identifying spans of text in a sequence. These kinds of sequence labeling tasks are called *information extraction* in the literature and are often the building blocks for more complicated multifield information extraction tasks. If you have a sentence with one disease and multiple locations, you would also determine the locations at which the disease was detected, if any. In this chapter, we will stick to the example of identifying individual spans and assume that you can extend them to more complicated information extraction tasks.

6.3.1 Accuracy for sequence labeling

The accuracy metric for sequence labeling depends on the task. For named entities, it is typically F-score on the entire span. So predicting "San Francisco" as a location would count 100% toward accuracy, but predicting "Francisco" or "San Francisco supermarket" would count 0% toward accuracy.

In some cases, this strict form of accuracy may be relaxed or reported along with more forgiving metrics, such as per-word accuracy (called *per-token* because not all tokens are exact words). In other cases, the accuracy might be reported for entity versus nonentity, with the type of entity (such as Disease or Location) reported separately.

You will most likely not care about the "O" label for your sequence tasks. F-score will capture the confusion between other labels and "O," which might be enough. As in object detection and semantic segmentation, you care about some parts of each data item more than others. Focusing on these parts of the data for active learning will lead to better samples.

As with the computer vision examples, you should use a metric for active learning sampling that is consistent with how you are measuring accuracy for your NLP model. For many NLP tasks, the context is more important than for object detection. You know that "San Francisco" is a location and not an organization with "San Francisco" in the name because of the context of the full sentence. So it is safer to have a broader context around the predicted sequences and often desirable, as the context can be an important predictor.

6.3.2 Uncertainty sampling for sequence labeling

Almost all sequence-labeling algorithms give you a probability distribution for your labels, most often using softmax, so that you can calculate the per-token uncertainty directly. In place of (or in addition to) the softmax confidences, you can use ensemble models and/or dropouts to produce multiple predictions and calculate the uncertainty as the level of agreement or entropy across those predictions. This approach is like the computer vision example for object detection.

Also as in the computer vision example, your confidences will be about the label confidence for each token, not the span as a whole or the boundary of the spans. But if you are using IOB2 tagging, your "B" tag will jointly predict the label and start boundary.

You can decide the best way to calculate the uncertainty for the entire span. The product of all the confidences is the (mathematically) most correct joint probability. You will also have to normalize for the number of tokens, however, which can be complicated. So the average or minimum confidence over all the tokens in a span might be easier to work with than the product.

Uncertainty can be important for tokens immediately outside the span. If we wrongly predicted that "Francisco" was not a location, we would want to take into account the fact that it might have been. Table 6.2 shows an example.

Table 6.2 An example of location identification and confidence associated with each label. The table shows an error in that only "San" from "San Francisco" is a location, but "Francisco" was reasonably high confidence. So we want to make sure that we take into account information outside the predicted spans when calculating confidence.

	The	E-Coli	outbreak	was	first	seen	in	a	San	Francisco	supermarket
Locations									B		
Confidence	0.01	0.32	0.02	0	0.01	0.03	0	0	0.81	0.46	0.12

Table 6.2 shows an error, in which only "San" from "San Francisco" is predicted to be a location. Even though "Francisco" was a false negative, however, it had reasonably high confidence (0.46). For this reason, we want to calculate uncertainty from more than the predicted span; we want to make sure the boundary is correct too.

In Table 6.2, we can treat "Francisco" as 1 – 0.46 = 0.54, which will lower our confidence for the span boundary. By contrast, at the start of the prediction, the "a" has zero confidence. So 1 – 0 = 1, which will increase our confidence. The "B" tag also helps with the confidence of the initial boundary.

6.3.3 Diversity sampling for sequence labeling

For your machine learning models, you are almost certainly using an architecture and/or feature representation that captures a wide window of context. In some models, you are encoding this representation directly. If you're using transformer-based methods, that context (attention) is discovered as part of the model itself, and you are probably providing only a maximum size. To help determine the context to use in active learning, choose a window for sampling that is consistent with the context your predictive model is using. Chapter 4 covered four types of diversity sampling:

- Model-based outlier sampling
- Cluster-based sampling
- Representative sampling
- Sampling for real-world diversity

We'll start with the first and last approaches, which are easiest, as we did for object detection:

- You can apply model-based outlier detection to a sequence labeling problem in the same way that you apply it to a document labeling problem.
- You can sample for real-world diversity in a sequence labeling problem in the same way that you sample for a document labeling problem.

For model-based outliers, the hidden layers focus on the spans you care about. That is, your neurons will be capturing information primarily to distinguish your spans from the nonspans ("B" and "I" from "O") and from the different labels of your spans. So you can apply model-based outliers directly without truncating the sentences to the immediate context of every predicted span.

You can see different feature representations in figure 6.9: one-hot, noncontextual embeddings (such as word2vec) and contextual embeddings (such as BERT). If you have worked in NLP, you probably used these common feature representations. In all three cases, we want to extract the predicted span of text and create a single feature vector to represent that span. The main differences are that we sum the one-hot encodings instead of using max (although max would probably work) and that there is no need to sample beyond the predicted span when using contextual embeddings because the context is already captured in the vectors. Calculate the contextual embeddings before you extract the phrase. For the other methods, it doesn't matter

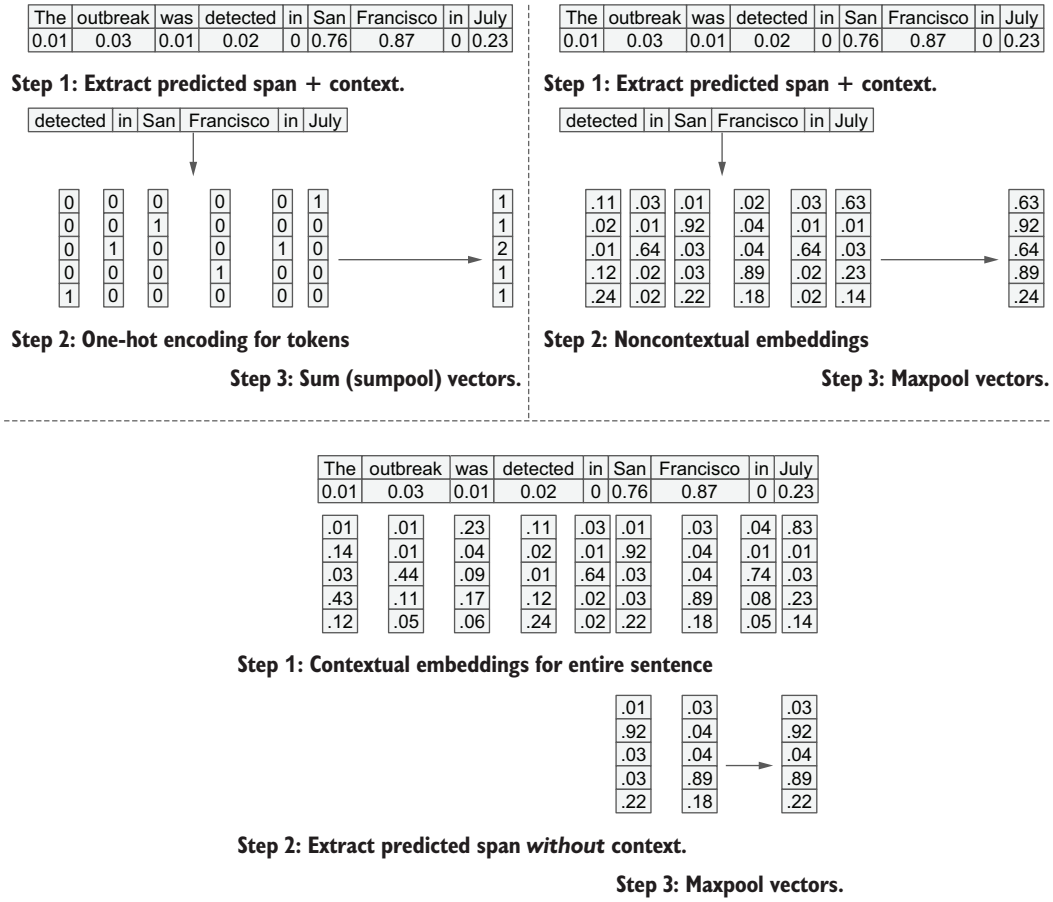


Figure 6.9 Three ways to encode predicted spans for active learning: using one-hot encoding to encode each token as its own feature (top left); using a noncontextual vector (embedding) such as word2vec (top right); and using a contextual embedding such as BERT (bottom). You can also experiment with average pooling (avepool) in place of or in addition to maximum pooling (maxpool).

whether you extract the phrase before or after you calculate the vector. For diversity sampling, the principles from chapter 4 also apply: you need to combine all the active learning methods to ensure fairer data across real-world demographics.

So far, you can see that diversity sampling for sequence labeling has many similarities to diversity sampling for object detection. You care about context of the objects/spans, but you don't necessarily need to worry about them for model-based outliers, as your model will already be focusing most of the neurons on the parts of the image/text you care about most.

For cluster-based sampling and representative sampling, we want to focus our models on the spans themselves, not too far into the context on either side. If you are

using contextual vector-representations of the tokens, you may not need extra context; that context is already captured in your vectors.

The preceding and following text should be cropped at meaningful distances and at word or sentence boundaries (or at phrasal boundaries, if you have this information). Because your model isn't 100% accurate, you need to ensure that you are capturing the full span:

- Crop at a given threshold. If the span is a Location, expand the selection to words before or after the prediction where Location is predicted with at least some low (say, 10%) confidence.
- Crop a wide threshold, maybe the whole sentence, and weight each word or subword sequence by the probability of each word's being part of the span.

Not all algorithms allow you to weight your features in a meaningful way. If you can't do this, use the same strategy as in object detection: generate multiple spans from an ensemble or dropout. Then try representative sampling for each of those predictions, weighted by their average overlap with the other predicted spans. You can use the words and subwords in each span directly for clustering and representative sampling, as you did in chapter 5.

If you are cropping the text and using the hidden layers of the model for cluster-based sampling, model-based outliers, or representative sampling, get those hidden layers before you crop the text. The full sentence context will be necessary for getting accurate contextual representations for each word in the span. When you have a vector of neuron activations for every word or subword in the sentence, you can crop the selection to the span.

The final problem that you'll need to solve is how to combine the vectors for each word or subword. If all your spans are the same length, you can concatenate them. If not, you'll need to combine them—a process known as *pooling* for neural vectors. The vectors tend to be sparse, so maxpooling is probably best (taking the maximum value in each vector index for each word or subword), but you could try averaging or some other pooling method to see the difference.

Whether you are using the words or subwords or a vector representation, you can apply cluster-based sampling and representative sampling as you learned in chapter 4. You can sample centroids, outliers, and random cluster members, and you can sample the most representative items from your target domain.

6.3.4 Active transfer learning for sequence labeling

You can apply active transfer learning to sequence labeling in the same way that you applied it to document-level labels. You can also apply ATLAS, adapting within one active learning cycle because you can assume that the first sequences you sample will be corrected later by human labelers, even if you don't know what those labels are.

Regardless of the type of neural architecture you use for sequence labeling, you can use the hidden layer(s) as the features for a binary "Correct"/"Incorrect" model

that you train on validation data. You will need to decide what counts as "Correct" and "Incorrect" in your validation data. If you care about some sequences more than others, you may want to count only errors with those sequences as "Incorrect" in your new model, focusing on the types of errors you care most about. You also need to decide whether to calculate errors in terms of the per-token error or for the sequence as a whole. As a starting point, it makes sense to calculate errors by using the same method that you use for calculating accuracy in your machine learning model, but you may want to experiment with other methods.

6.3.5 Stratified sampling by confidence and tokens

Set your threshold for predicting spans low, whatever method you use. You don't want to find only spans that are similar to those that already exist in your data, which will perpetuate the bias. You can use the same stratified sampling by confidence method (section 6.1.5) for object detection, perhaps sampling an equal number of spans at 0%–10% confidence, 10%–20% confidence, and so on.

Also, you can stratify the sample according to the tokens themselves. You might cap the sample of spans that are "San Francisco" (or any other sequence) to sample a maximum of 5 or 10 instances, thereby sampling a greater diversity of tokens overall.

6.3.6 Create training data samples for representative sampling that are similar to your predictions

If you are cropping your unlabeled text for representative sampling, you should do the same with the training data. If you use only the perfect span annotations from your training data, but then the imperfect predictions from your unlabeled data, the "representative" samples could end up being the result of different cropping strategies instead of the actual span differences.

Section 6.1.6 covered some strategies for cropping the training data and unlabeled data to reduce bias. These strategies apply to spans too, so look at those methods if you are applying representative sampling to your spans.

As with object detection, you should consider using some sampling methods on uncropped text. You might do more of this here, because the context for a span typically is contextually relevant pieces of that language optimized to encode information; while the backgrounds for object detection are more likely to be random junk in the world.

Some simple approaches to representative sampling can be effective, and you may not need to build a model. You might even choose to focus only on predicted spans that don't yet occur in your training data.

6.3.7 Full-sequence labeling

For a handful of tasks in NLP, you want to label every item in the text. An example is part-of-speech (POS) tagging, as shown in table 6.3.

Table 6.3 An example of a full-sequence parse, showing POS tags (labels) such as nouns, verbs, adverbs (Adv), proper nouns (PRP), and so on

	The	E-Coli	outbreak	was	first	seen	in	a	San	Francisco	supermarket
POS	Dt	PRP	Noun	Aux	Adv	Verb	PR	Dt	PRP	PRP	Noun
Confidence	0.01	0.32	0.02	0	0.01	0.03	0	0	0.81	0.46	0.12

You can treat this task the same as tagging sequences within the text, but it is simplified in that you have to worry less about cropping the text or ignoring "O" labels. Stratification by labels is likely to help for cases such as those in table 6.3, taking the 100 most uncertain nouns, the 100 most uncertain verbs, the 100 most uncertain adverbs, and so on. You can use this sampling method along with macro F-score to evaluate the model accuracy.

6.3.8 Sampling for document-level diversity in sequence labeling

As with any other method, you should always randomly sample text for review. This practice provides your evaluation data and gives you a baseline for how successful your active learning is. If you apply clustering at the whole-document level and find whole clusters with little or no existing training items, you have good evidence that you should get human review for some items in those clusters because you could be missing something.

There is a good chance that there are real-world diversity considerations at the document level, too: the genre of the text, the fluency of the person who created it, the language(s), and so on. Stratified sampling for real-world diversity may be more effective at document level than at sequence level for these cases.

6.4 Applying active learning to language generation

For some NLP tasks, the machine learning algorithm is producing sequences like natural language. The most common use case is text generation, which are the example use cases in this section. Most language generation for signed and spoken languages starts with text generation and then generates the signs or speech as a separate task. The machine learning models are typically general sequence generation architectures that can be applied to other types of sequences, like genes and music, but these are also less common than text.

Even then, it is only on the back of recent advances in transfer learning that full text generation systems are at a level of accuracy that enables us to start using them in real-world applications.

The most obvious exception is machine translation, which has been popular in academic and industry settings for some time. Machine translation is a well-defined problem, taking a sentence in one language and producing one in a new language. Historically, machine translation has a lot of existing training data to draw from in the form of books, articles, and web pages that have been translated between languages manually.

Question-answering, in which a full sentence is given in response to a question, is growing in popularity as an example of text generation. Another example is a dialogue system such as a chatbot, producing sentences in response to interactions. Summarization is yet another example, producing a smaller number of sentences from a larger text. Not all of these use cases necessarily use full text generation, however. Many question-answering systems, chatbots, and summarization algorithms use templated outputs to create what seem like real communications after extracting the important sequences from the inputs. In these cases, they are using document-level labels and sequence labels, so the active learning strategies that you have already learned for document-labeling and sequence labeling will be sufficient.

6.4.1 Calculating accuracy for language generation systems

One complicating factor for language generation is that there is rarely one correct answer. This situation is often addressed by having multiple correct responses in the evaluation data and allowing the best match to be the score that is used. In translation tasks, the evaluation data often contains several correct translations, and accuracy is calculated by the best match of a translation against any of them.

In the past few years, major advances in neural machine translation have been full sentence-to-sentence generation; the machine learning takes in examples of the same sentences in two languages and then trains a model that can translate directly from one sentence to another. This feature is incredibly powerful. Previously, machine translation system had multiple steps to parse in and out of different languages and align the two sentences. Each step used a machine learning system of its own, and the steps were often put together with a meta- machine learning system to combine them. The newer neural machine translation systems that need only parallel text and can take care of the entire pipeline use only about 1% of the code of the earlier cobbled-together systems and are much more accurate. The only step back is that neural machine translation systems are less interpretable today than their non-neural predecessors, so it is harder to identify confusion in the models.

6.4.2 Uncertainty sampling for language generation

For uncertainty sampling, you can look at the variation across multiple predictions, as you did for sequence labeling and the computer vision tasks, but this area is much less studied. If you are building models for text generation, you are probably using an algorithm that generates multiple candidates. It may be possible to look at the variation in these candidates to measure uncertainty. But the neural machine translation models typically generate a small number of candidates by using a method called beam search (around 5), which isn't enough to measure the variation accurately. Recent research shows that widening the search can reduce overall model accuracy, which you obviously want to avoid.[1]

[1] "Analyzing Uncertainty in Neural Machine Translation," by Myle Ott, Michael Auli, David Grangier, and Marc'Aurelio Ranzato (https://arxiv.org/abs/1803.00047).

You could try to model uncertainty with ensemble models or dropouts from a single model. Measuring the level of agreement across ensemble models is a long-standing practice in machine translation for determining uncertainty, but the models are expensive to train (often taking days or weeks), so training multiple models simply to sample for uncertainty could be prohibitively expensive.

Using dropouts during sentence generation can help generate uncertainty scores by getting multiple sentences from a single model. I experimented with this approach for the first time in a paper presented during the writing of this book.[2] Initially, I was going to include this study, which focused on bias detection in language models, as an example in the final chapter in this book. Given that the content is already in that paper and that the disaster-response examples in this book became even more relevant with the COVID-19 pandemic that began while I was writing it, however, I replaced that example in chapter 12 with the example task tracking potential foodborne outbreaks.

6.4.3 Diversity sampling for language generation

Diversity sampling for language generation is more straightforward than uncertainty sampling. If your input is text, you can implement diversity sampling exactly as you did in chapter 4 for document-level labeling. You can use clustering to ensure that you have a diverse set of inputs, representative sampling to adapt to new domains, and model-based outliers to sample what is confusing to your current model. You can also stratify your samples by any real-world demographics.

Diversity sampling has typically been a major focus of machine translation. Most machine translation systems are general-purpose, so the training data needs to cover as many words as possible in your language pairs, with each word in as many contexts as possible, especially if that word has multiple translations depending on context.

For domain-specific machine translation systems, representative sampling is often used to ensure that any new words or phrases that are important to a domain have translations. When you are adapting a machine translation system to a new technical domain, for example, it is good strategy to oversample the technical terms for that domain, as they are important to get correct and are unlikely to be known by a more general machine translation system.

One of the most exciting applications for diversity sampling for text generation is creating new data for other tasks. One long-standing method is *back translation*. If you have a segment of English text labeled as negative sentiment, you could use machine translation to translate that sentence into many other languages and then back into English. The text itself might change, but the label of negative sentiment is probably still correct. This kind of generative approach to training data, known as *data augmentation*, includes some exciting recent advances in human-in-the-loop machine learning that we will cover in chapter 9.

---

[2] "Detecting Independent Pronoun Bias with Partially-Synthetic Data Generation," by Robert (Munro) Monarch and Alex (Carmen) Morrison (https://www.aclweb.org/anthology/2020.emnlp-main.157.pdf).

6.4.4 Active transfer learning for language generation

You can apply active transfer learning to language generation in a similar way to the other use cases in this chapter. You can also apply ATLAS, adapting within one active learning cycle because you can assume that the first sequences you sample will be corrected by human labelers later, even if you don't know what those labels are.

You need to carefully define what counts as a "Correct" or "Incorrect" prediction in your validation data, however. Typically, this task involves setting some threshold of accuracy at which a sentence is considered to be correct or incorrect. If you can calculate accuracy on a per-token basis, you have the option of aggregating the accuracy across all tokens to create a numerical accuracy value. You can predict a continuous value instead of the binary "Correct"/"Incorrect," as in the IoU example for object detection in section 6.1.1.

6.5 Applying active learning to other machine learning tasks

The active learning principles in chapters 3, 4, and 5 can be applied to almost any machine learning task. This section covers a few more at a high level. This section doesn't go into the level of implementation detail that you learned for the computer vision and NLP examples, but it will give you an idea about how the same principles apply to different data types.

For some use cases, it isn't possible to collect new unlabeled data at all and you will need to find ways to measure your accuracy by other means. See the following sidebar for more about one such method: synthetic controls.

Synthetic controls: Evaluating your model without evaluation data

Expert anecdote by Dr. Elena Grewal

How can you measure your model's success if you are deploying an application where you can't run A/B tests? Synthetic control methods are a technique that you can use in this case: you find existing data that is closest in features to where you are deploying the model and use that data as your control group.

I first learned about synthetic controls when studying education policy analysis. When a school tries some new method to improve their students' learning environment, they can't be expected to improve only half the students' lives so that the other half can be a statistical control group. Instead, education researchers might create a synthetic control group of schools that are most similar in terms of the student demographics and performance. I took this strategy, and we applied it at Airbnb when I was leading data science there. When Airbnb was rolling out a product or policy change in a new city/market and could not run an experiment, we would create a synthetic control group of the most similar cities/markets. We could then measure the impact of our models compared with the synthetic controls for metrics such as engagement, revenue, user ratings, and search relevance. Synthetic controls allowed us to take a data-driven approach to measuring the impact of our models, even where we didn't have evaluation data.

(continued)
Elena Grewal is founder and CEO of Data 2 the People, a consultancy that uses data science to support political candidates. Elena previously led Airbnb's data science team and has a PhD in education from Stanford University.

6.5.1 Active learning for information retrieval

Information retrieval is the set of algorithms that drives search engines and recommendation systems. Multiple metrics can be used for calculating the accuracy of retrieval systems that return multiple results from a query. The most common of those metrics today is discounted cumulative gain (DCG), in which rel_i is the graded relevance of the result at a ranked position p:

$$\mathrm{DCG_p} = \sum_{i=1}^{p} \frac{2^{rel_i} - 1}{\log_2(i+1)}$$

The `log()` is used to deweight the lower entries. Perhaps you want the first search result to be the most accurate; you care slightly less about the second search result; slightly less again about the third search result; and so on. The use of a log was a fairly arbitrary weighting when first introduced, but some relatively recent theory suggests that it has mathematical validity.[3]

Real-world search systems are some of the most sophisticated uses cases for human-in-the-loop machine learning today. Think about a simple search in an online store. The store is using one form of machine learning to retrieve the search results. It uses a second form of machine learning to identify keywords and entities in your search string. It uses a third form of machine learning to extract the relevant summary text from each product in the results. The products are categorized into the type of product (electronics, books, and so on) to help with search relevance in a fourth kind of machine learning. The store might also be using a fifth form of machine learning to decide what is an ideal display image (plain background or in context). Many modern search engines also try to maximize diversity, returning different product types rather than 10 versions of the same product. So six or more different machine learning systems may be contributing to your search results, even before any models try to personalize the results for your experience. Each of these machine learning systems needs its own training data. Some of that data can be derived from what people click, but much of it is from offline annotators who provide that feedback.

You may not realize that you're using cutting-edge machine learning when you shop online, but a lot is going on behind the scenes. In fact, this use case is why the best-known crowdsourcing platform, Amazon's Mechanical Turk, was invented: to clean up catalog information for products in the online store.

[3] "A Theoretical Analysis of NDCG Type Ranking Measures," by Yining Wang, Liwei Wang, Yuanzhi Li, Di He, Wei Chen, and Tie-Yan Liu (https://arxiv.org/abs/1304.6480).

Information retrieval also tends to use more real-world accuracy metrics than other machine learning applications. Although DCG is popular for offline evaluation of search relevance, the results for people using the system are often optimized for business-oriented metrics: the number of purchases a person makes, the number of clicks/seconds between a search and a purchase is made, the value of the customer over the next six months, and so on. Because these metrics are about use of the model, they are sometimes called *online metrics*, as opposed to F-score and IoU, which are offline metrics. These metrics are different from F-score and IoU, and much more human-centric, so other use cases can learn a lot from the information retrieval community.

6.5.2 Active learning for video

Most of the solutions for still images also apply to object detection and/or semantic segmentation in videos. Focus on the regions you care about the most in the video, and use them for your samples. If your model focuses only on objects or labels you care about, you can implement uncertainty sampling and model-based outliers without necessarily cropping or masking your videos to the objects you care about. If you are applying diversity sampling, you almost certainly want to crop or mask to those objects first.

The biggest difference between videos and still images is that you have many frames of data from the same video with almost identical images. The obvious solution is the best one: if you have multiple frames from what your model thinks is the same object, sample the frame with the highest uncertainty. The iterative process of retraining on that new object is likely to give you some or all the other frames with high confidence afterward.

Diversity sampling should already reduce the number of times the same object is selected in different frames because the object will look the same across frames. If the object changes form, you probably want to sample it in different forms, so this situation works out. An example is sign language. You are not tracking an object so much as trying to interpret a stream of information, so your active learning strategy might look more like text and speech than object detection.

Note that if you don't use diversity sampling for object detection in videos, you might find that your most uncertain samples are the same object in successive frames. Most companies I've seen sample every *N*th frame and/or sample an exact number of frames per video, typically the first, last, and some number of intermediary ones. There is nothing wrong with this approach to stratified sampling, but sampling diversity through clustering and adaptive representative sampling in addition generally leads to much richer samples. You might also need to oversample some videos to get more frames containing certain rarer labels to improve real-world diversity. You have a lot of individual images if you take every frame of every video, so you can also try large-scale clustering on the whole images first and use the total number of videos as a guide:

- If you have fewer clusters than the total number videos, combine similar videos into one cluster to have targeted diversity.
- If you have more clusters than the total number of videos, you should end up with some videos split into multiple clusters, ideally the videos that have more diverse content.

This approach gives you a lot of scope to combine the active learning methods covered in this book to annotate a video as quickly as possible.

6.5.3 Active learning for speech

Like text or signed language, speech can be a labeling task, a sequence task, or a language generation task. You approach each use case differently, as you do for text or images.

If you are labeling speech at the level of entire speech acts (called *intent* if you are labeling commands spoken to a smart device or similar object), your model is already focused on the phenomena you care about, as with object detection and sequence labeling. So the uncertainty sampling and model-based outliers should work on your speech data without cropping.

If you are transcribing speech into text, or performing some other task that looks at error across the whole recording, this process is more similar to text generation, in which you want to focus on diversity to sample as many speech acts as possible. In pretty much every language of the world, the writing system is more standardized than the spoken language. So diversity becomes more important when you're trying to capture every possible accent and language variation compared with working with text.

Speech falls between text and images in terms of how much data collection technology matters. The quality of the microphone, ambient noise, recording device, file format, and compression techniques can all produce artifacts that your model could learn erroneously instead of the actual information.

More than any other data type covered here, speech differs most between its perceived structure and actual physical structure. You perceive gaps between words, for example, but that perception is an illusion, because real speech almost always runs words together. Almost every sound changes in immediate context, too. The English plural is *s* or *z* depending on the previous phoneme (*cats* and *dogz*), but you might have assumed that the plural suffix was only one sound. When you sample speech data, be careful not to rely only on text transcriptions of that speech.

6.6 Choosing the right number of items for human review

For advanced active learning techniques, the principles that you have already learned apply. You can make some of the active learning strategies, such as representative sampling, adaptive within an active learning iteration, but most combinations of techniques still produce the most benefit when you retrain the model with the newly annotated data.

You probably need to sample a minimum number of items as a result of drawing from a certain number of clusters or stratification to real-world demographics. Your maximum number of items per iteration will vary depending on data type. You might be able to annotate locations in 1,000 short text messages per hour, but only complete semantic segmentation on 1 image over the same time period. So a big factor in your decision will be your data types and the annotation strategies that you are deploying—something that we'll cover in chapters 7–12.

6.6.1 Active labeling for fully or partially annotated data

If your machine learning models can learn from partially annotated data, you are going to make your systems a lot more efficient. To continue the example we've used throughout this book, imagine that you are implementing an object detection model for city streets. Your model might be accurate enough to identify cars and pedestrians, but not accurate enough to identify bicycles and animals.

You might have thousands of images of bicycles and animals, but each image has dozens of cars and pedestrians on average too. Ideally, you'd like to be able to annotate only the bicycles and animals in those images and not spend more than 10 times the resources to make sure that all the cars and pedestrians are also labeled in those same images. A lot of machine learning architectures don't allow you to partially annotate data, however; they need to have every object annotated, because otherwise, those objects will erroneously count toward the background.

You might sample the 100 bicycles and animals that maximize confusion and diversity, but then spend most of your resources annotating 1,000 cars and pedestrians around them for relatively little extra gain. There is no shortcut: if you sample only images without many cars or pedestrians, you are biasing your data toward certain environments that are not representative of your entire dataset. If you are stuck with systems that need full annotation for every image or document, you want to be extra-careful to ensure that you are sampling the highest-value items every time.

Increasingly, it is easier to combine different models or have heterogeneous training data. You might be able to train separate models for pedestrians and cars, and then have a model that combines them via transfer learning.

6.6.2 Combining machine learning with annotation

You should take these options into account when designing your annotation and model strategies, because you might find that a slightly less accurate machine learning architecture will end up producing much more accurate models when you are not constrained to annotating images entirely or not at all.

The best solution to the problem of needing to annotate only a few objects/spans in a large image/document is to incorporate machine learning into the annotation process. It might take an hour to annotate an entire image for semantic segmentation, but only 30 seconds to accept/reject every annotation. The danger when combining predictions and human annotation is that the people might be primed to trust a

prediction that was not correct, therefore perpetuating an existing bias. This situation is a complicated human–computer interaction problem. Chapters 9, 10, and 11 cover the problem of combining model predictions and human annotations in the most effective ways possible.

6.7 Further reading

For more information about calculating confidence for sequence labeling and sequence generation, see "Modeling Confidence in Sequence-to-Sequence Models," by Jan Niehues and Ngoc-Quan Pham (http://mng.bz/9Mqo). The authors look at speech recognition and also extend the machine translation problem in an interesting way by calculating confidence (uncertainty) on the source text tokens, not only the predicted tokens.

For an overview of active learning techniques for machine translation, see "Empirical Evaluation of Active Learning Techniques for Neural MT," by Xiangkai Zeng, Sarthak Garg, Rajen Chatterjee, Udhyakumar Nallasamy, and Matthias Paulik (http://mng.bz/j4Np). Many of the techniques in this paper can be applied to other sequence generation tasks.

Summary

- In many use cases, you want to identify or extract information within an image or document rather than label the entire image or document. The same active learning strategies can be applied to these use cases. Understanding the right strategy helps you understand the kinds of problems to which you can apply active learning and how to build the right strategy for your use case.
- You need to crop or mask your images and documents to get the most out of some active learning strategies. The right cropping or masking strategy produces better samples for human review, and understanding when you need to crop or mask your items helps you select the right method for your use cases.
- Active learning can be applied to many tasks beyond computer vision and NLP, including information retrieval, speech recognition, and videos. Understanding the broader landscape of active learning application areas will help you adapt any machine learning problem.
- The number of items to select for human review in each iteration of advanced active learning is highly specific to your data. Understanding the right strategy for your data is important for deploying the most efficient human-in-the-loop machine learning systems for your problems.

Part 3

Annotation

Annotation puts the human in human-in-the-loop machine learning. Creating datasets with accurate and representative labels for machine learning is often the most underestimated component of a machine learning application.

Chapter 7 covers how to find and manage the right people to annotate data. Chapter 8 covers the basics of quality control for annotation, introducing the most common ways to calculate the overall accuracy and agreement for an entire dataset and between annotators, labels, and on a per-task basis. Unlike with machine learning accuracy, we typically need to adjust for random chance accuracy and agreement for human annotators, which means that the evaluation metrics are more complicated when evaluating human performance.

Chapter 9 covers advanced strategies for annotation quality control, starting with techniques to elicit subjective annotations and then expanding to machine learning models for quality control. The chapter also covers a wide range of methods to semi-automate annotation with rule-based systems, search-based systems, transfer learning, semi-supervised learning, self-supervised learning, and synthetic data creation. These methods are among the most exciting research areas on the machine learning side of human–computer interaction today.

Chapter 10 starts by examining how often wisdom of the crowds applies to data annotation with an example of continuous value annotation (hint: less often than many people realize). The chapter covers how annotation quality control techniques can be applied to different kinds of machine learning tasks, including object detection, semantic segmentation, sequence labeling, and language generation. This information will allow you to develop annotation quality control strategies for any machine learning problem and think about how to break a complicated annotation task into simpler subtasks.

7 Working with the people annotating your data

This chapter covers

- Understanding in-house, contracted, and pay-per-task annotation workforces
- Motivating different workforces using three key principles
- Evaluating workforces when compensation is nonmonetary
- Evaluating your annotation volume requirements
- Understanding the training and/or expertise that annotators need for a given task

In the first two parts of this book, you learned how to select the right data for human review. The chapters in this part cover how to optimize that human interaction, starting with how to find and manage the right people to provide human feedback. Machine learning models often require thousands (and sometimes millions) of instances of human feedback to get the training data necessary to be accurate.

The type of workforce you need will depend on your task, scale, and urgency. If you have a simple task, such as identifying whether a social media posting is positive or negative sentiment, and you need millions of human annotations as soon as possible, your ideal workforce doesn't need specialized skills. But ideally, that workforce can scale to thousands of people in parallel, and each person can be employed for short amounts of time.

If, however, you have a complicated task, such as identifying evidence of fraud in financial documents that are dense with financial terms, you probably want annotators who are experienced in the financial domain or who can be trained to understand it. If the documents are in a language that you don't speak yourself, finding and evaluating the right people to annotate the data is going to be even more complicated.

Often, you want a data annotation strategy that combines different kinds of workforces. Imagine that you work at a large financial company, and you are building a system to monitor financial news articles that might indicate a change in the value of the companies. This system will be part of a widely used application that people use to make decisions about buying and selling shares of companies listed on the stock market. You need to annotate the data with two types of labels: which company each article is about and whether the information in each article implies a change in share price.

For the first type of label—company identification—you can easily employ non-expert annotators. You don't need to understand financial news to identify company names. However, it is complicated to understand which factors can change the share price. In some cases, general fluency in the language is enough if the content is explicit (such as "Shares are expected to tumble"). In other cases, the context is not so obvious. Is the sentence "Acme Corporation meets adjusted Q3 projections," for example, positive or negative for a company's chances of meeting an adjusted quarterly projection? You need to understand the context of the adjustment. For more complicated language with financial acronyms, it is impossible for a person without training in the financial domain to understand.

Therefore, you may decide that you need three kinds of workforces:

- *Crowdsourced workers*, who can scale up and down fastest when news articles are published to identify what companies are being spoken about
- *Contract workers*, who can learn financial terminology to understand changes in share prices
- *In-house experts*, who can label the most difficult edge cases, adjudicate conflicting labels, and provide instructions for other workers

No matter who the right people are, they will do the best work when they are paid fairly, are secure in their jobs, and have transparency about the work that they are completing. In other words, the most ethical way to manage a workforce is also the best for your organization. This chapter covers how to select and manage the right workforces for any annotation task.

7.1 Introduction to annotation

Annotation is the process of creating training data for your models. For almost all machine learning applications that are expected to operate autonomously, you will need more data labels than it is practical for one person to annotate, so you will need to choose the right workforce(s) to annotate your data and the best ways to manage them. The human-in-the-loop diagram in figure 7.1 shows the annotation process, taking unlabeled data and outputting labeled training data.

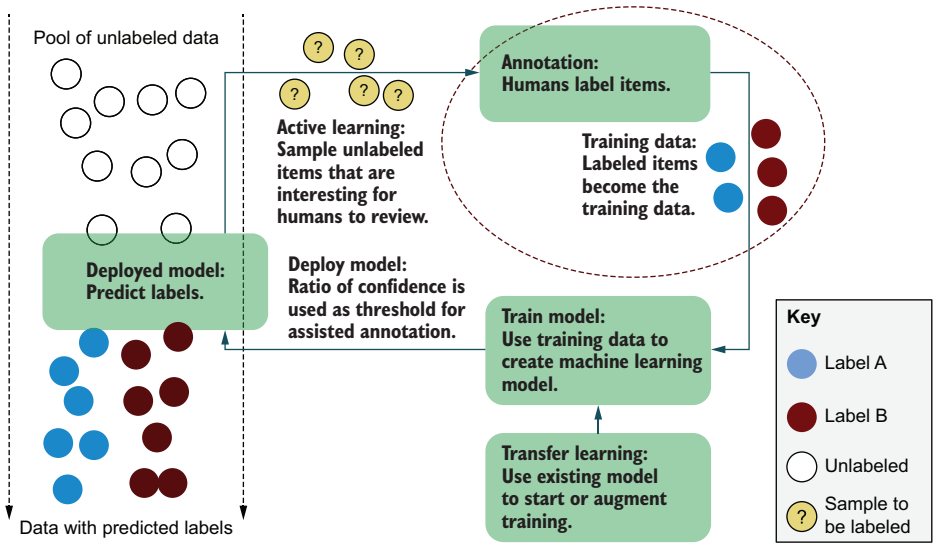


Figure 7.1 ***Data annotation*** **is the process of creating unlabeled data, either by labeling unlabeled data or reviewing labels generated from a model.**

In this chapter and the following ones, we will dive into the annotation component of figure 7.1, showing the subprocesses and algorithms needed to run annotation projects correctly. One piece of advice: start your data strategy with your algorithm strategy. It takes as long to refine your annotation strategy and guidelines as it does to create your algorithm architecture and tune your hyperparameters, and your choice of algorithm and architecture should be informed by the type and volume of annotations that you expect.

7.1.1 Three principles of good data annotation

The more respect you show the people who label your data, the better your data becomes. Whether the person is an in-house subject-matter expert (SME) or an outsourced worker who will contribute only a few minutes to your annotations, these basic principles will ensure that you get the best possible annotations:

- *Salary*—Pay fairly.
- *Security*—Pay regularly.
- *Ownership*—Provide transparency.

The three main types of workforces are summarized in figure 7.2, which shows an uneven amount of work needed over time. The crowdsourced workers are easiest to scale up and down, but the quality of their work is typically lowest. In-house workers are hardest to scale, but they are often SMEs, who provide the highest-quality data. Outsourced workers fall in between: they have some of the flexibility of crowdsourced workers and can be trained to high levels of expertise. These differences should influence your choice of workforce(s). We will cover each workforce in more detail in the following sections, expanding on the principles of salary, security, and ownership for each one.

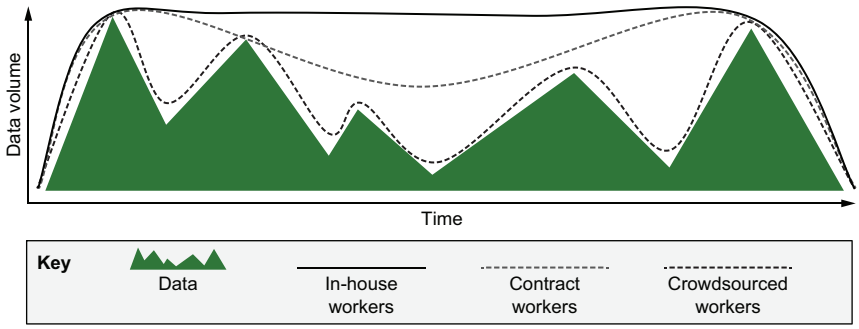


Figure 7.2 An overview of the three main types of workforces you can use to annotate data, in-house workers, contract workers, and crowdsourced workers, where the main trade-off is flexibility versus expertise.

Work in supervised machine learning is people management

If you work with human-annotated data, you work in people management. Most real-world machine learning applications use supervised learning with data that has been annotated for that purpose. There is no way to avoid the responsibility you have for the people who are annotating the data for you: if their work is used in the models you are creating, you have a duty of care for them.

Many data scientists see their work being as pure research. Many senior data scientists in companies do not have to manage other researchers, precisely because people management is seen as getting in the way of doing "real" work in research. Unfortunately for those people, you have no way to outsource your responsibility for the people who annotate the data, even if your organization has a separate data annotation team or has outsourced annotation.

This chapter may seem to be more like management advice than technical advice, but this focus is deliberate, because knowing how to manage distributed teams of people is an important skill for any data scientist. For the reasons outlined in this chapter, good management is also necessary to ensure fair working conditions for every person who contributes to your models.

Your responsibilities include talking to the people who annotate your data. I doubt that you've met a good manager who gave their staff guidelines only at the start of a project and didn't invite feedback. Eliciting feedback can be complicated when there is a power imbalance, so you need to implement communication channels mindfully and with empathy.

7.1.2 Annotating data and reviewing model predictions

In this book, the term *annotation* is used broadly. In some cases, it means annotating raw data; in other cases, it means humans assisted by or interacting with machine learning models.

We will return to user-interfaces and quality control in chapter 11. For now, understand that when you calculate the volume of work required for annotation, you need to take into account the different ways that you might present your data for annotation and the different amounts of work required.

7.1.3 Annotations from machine learning-assisted humans

For many tasks, the goal is to assist human processes. In fact, many models could be used for automation or to assist humans, depending on the application. You could train a collision-detection algorithm to power a fully autonomous vehicle or to alert the driver, for example. Similarly, you could train a medical imaging algorithm to make diagnoses or to inform a medical practitioner's decision-making.

This chapter applies to both types of applications. We will cover the concept of end users as annotators in section 7.5.1. As that section discusses, you may still want to employ annotators other than your end users, even if your application gets many annotations for free as it assists with human tasks.

For machine-learning-assisted humans, one extra point is related to the job security and transparency principles: be clear that your goal is to assist the work of end users, not to train their automated replacements. But if you know that you are getting end user feedback to automate people out of a given task, you must be transparent about that fact so that expectations are realistic, and you should compensate those people accordingly.

7.2 In-house experts

The largest workforce for most machine learning projects is in-house workers—people who work in the same organization as the people who build the algorithms. Despite this fact, quality control and worker management are the least well-studied for in-house annotators compared with outsourced and crowdsourced workers. Most of the academic papers on annotation focus on outsourced and (in particular) crowdsourced pay-per-task workers. You obviously gain a lot by having model builders and annotators within the same organization because they are able to communicate directly.

The advantages of in-house works are domain expertise and protection for sensitive data. If you work on complicated problems, such as analyzing financial reports or diagnosing medical images, the members of your in-house team may be some of the few people in the world who have the skills needed to annotate your data. If your data contains sensitive information, in-house workers also provide the most privacy and security for your data.

For some use cases, you may be restricted to keeping data in-house for regulatory reasons. The data-generation tools that we will cover in chapter 10 can help you in these cases. Even if your synthetic data is not 100% accurate, there's a good chance that your synthetic data will not be as sensitive as your real data. So, you have an opportunity to employ outsourced workforces to filter or edit the synthetic data to your desired level of accuracy when the actual data is too sensitive to share with outsourced workers.

Although in-house workers often have more subject-matter expertise than other workers do, it can be a mistake to think that means that they represent the full range of people who will be using your applications. See the following sidebar for more information about who the best SMEs can be.

Parents are the perfect SMEs

Expert anecdote by Ayanna Howard

Models about people are rarely accurate for people who are not represented in the data. Many demographic biases can lead to people being underrepresented, such as ability, age, ethnicity, and gender. There are often intersectional biases too: if people are underrepresented across multiple demographics, sometimes the intersection of those demographics is more than the sum of the parts. Even if you do have the data, it may be difficult to find annotators with the right experience to annotate it correctly.

When building robots for kids with special needs, I found that there was not sufficient data for detecting emotion in children, detecting emotion in people from underrepresented ethnicities, and detecting emotion in people on the autism spectrum. People without immersive experience tend to be poor at recognizing emotions in these children, which limits who can provide the training data that says when a child is happy or upset. Even some trained child physicians have difficulties accurately annotating data when addressing the intersectionality of ability, age, and/or ethnicity. Fortunately, we found that a child's own parents were the best judges of their emotions, so we created interfaces for parents to quickly accept or reject a model's prediction of the child's mood. This interface allowed us to get as much training data as possible while minimizing the time and technical expertise that parents needed to provide that feedback. Those children's parents turned out to be the perfect SMEs to tune our systems to their child's needs.

Ayanna Howard is dean of the College of Engineering at Ohio State University in Columbus. She was previously chair of the School of Interactive Computing at Georgia Tech University and co-founder of Zyrobotics, which makes therapy and educational products for children with special needs. She formerly worked at NASA and has a PhD from the University of Southern California.

7.2.1 Salary for in-house workers

You probably don't set the salary for the annotators within your company, so this principle comes for free: they are already getting a salary that they have agreed to. If you do set the salary for your in-house annotators, make sure that they are treated with the same respect and fairness as other employees.

7.2.2 Security for in-house workers

An in-house worker already has a job (by definition), so security comes from their ability to keep that job while you have work for them—that is, while you ensure that they can keep that job. If your in-house workers' employment and organizational positions provide less job security because they are temporary or contract workers, use some of the principles for outsourced workers. For contract workers, for example, try to structure the amount of work available to be as consistent as possible and with clear expectations of how long their employment will last. Be transparent about job mobility. Can a person become permanent and/or move into other roles?

7.2.3 Ownership for in-house workers

Transparency is often the most important principle for in-house workers. If workers are already getting paid regardless of whether they are creating annotations, you need to make the task inherently interesting.

The best way to make any repetitive task interesting is to make it clear how important that work is. If your in-house annotators have transparency into how their work is improving your company, this information can be good motivation. In fact, annotation can be one of the most transparent ways to contribute to an organization. If you have daily targets for the number of annotations or can share how the accuracy goes up in your trained models, it is easy to tie the annotation work to company goals. An annotator is much more likely to contribute to an increase in accuracy than a scientist experimenting with new algorithms, so you should share this fact with your annotators.

Beyond having the daily motivation of seeing how annotations help the company quantitatively, in-house annotators should be clear about how their work is contributing to overall company goals. An annotator who spends 400 hours annotating data that powers a new application should feel as much ownership as an engineer who spends 400 hours coding it.

I see companies get this concept wrong all the time, leaving their in-house annotation teams in the dark about the impact that their work has on daily or long-term goals. That failure is disrespectful to the people who do the work and leads to poor motivation, high churn, and low-quality annotations, which doesn't help anyone.

In addition, you have a responsibility to ensure that work is consistently available for in-house workers. Your data might come in bursts for reasons that you can't control. If you are classifying news articles, for example, you will have more data at the times of day and week when news articles are published in certain time zones. This situation lends itself to a crowdsourced annotation workforce, but you may be able to

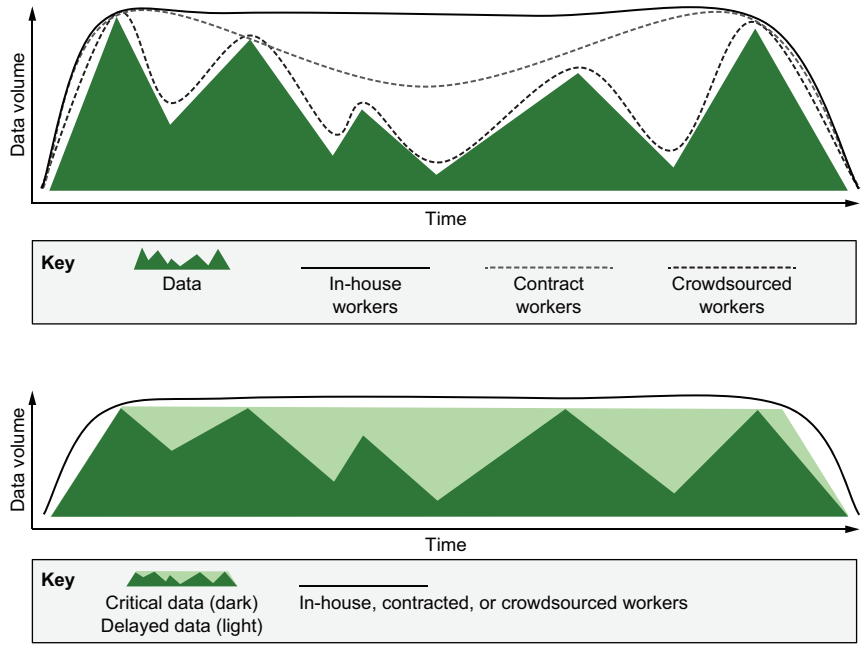


Figure 7.3 Smoothing your work for in-house workers. When data comes in bursts for reasons that you can't control, it can be smoothed out, with the most critical data annotated first and the other data annotated later. The bottom graph shows the annotation workload smoothed out.

decide that some data labeling can be delayed. Figure 7.3 shows an example in which the most critical data is annotated as it comes in and the other data is annotated later.

The more consistently the data needs to be labeled, the easier it will be to manage the annotation processes. When trying to smooth out the workload, you can randomize which data comes first, but you have other options to experiment with. You might cluster all the data first and ensure that the centroids are all annotated first to ensure diversity, for example. Alternatively, or in addition, you may want to apply representative sampling to annotate the newest-looking items first. All these approaches are good ways to smooth out the volume of annotations needed while getting the most out of the data as soon as it arrives.

7.2.4 Tip: Always run in-house annotation sessions

No matter what combination of workforces you use, I recommend running annotation sessions among the most diverse group of internal staff members possible. This approach has several benefits:

- The high-quality annotations that are created by people in-house can become your (human) training examples that form part of your quality control data (see chapter 8).

- Your in-house annotation sessions help surface edge cases early, such as data items that are hard to label because they are not covered by the current annotation guidelines. Understanding these edge cases will help you refine your task definition and your instructions to the people annotate your data.
- This process is a great team-building exercise. If you get people from every department of your organization in one room (with food and drinks if they'll be working for an extended period), the process is a lot of fun, and it allows everybody in your company to contribute to your machine learning applications. Have everyone annotate data for at least one hour while discussing their edge cases as they come across them. For many companies I have worked in, "annotation hour" is many people's favorite time of the week.

This exercise can also be a good way to establish an in-house team of experts to create and update guidelines for your bigger annotation team. Especially if you have data that is changing over time, you will want to update your annotation guidelines regularly and provide current example annotations.

You can also use some outsourced annotators as experts, and occasionally, a great crowdsourced worker may be able to help with this process. A lot of organizations that focus on annotation have internal expertise about the best way to create guidelines and training material. Trust their expertise, and consider inviting people from outsourced organizations to your in-house annotation sessions.

Figure 7.4 shows some examples of integrating expert annotators, from models that ignore expert annotators altogether (recommended only for pilots) to more sophisticated workflows that optimize how experts can help ensure quality control when data is changing over time.

If you implement the first method in figure 7.4 in a pilot, exclude the items that are confusing rather than include them with potentially incorrect labels. If 5% of your items could not be labeled, exclude those 5% of items from your training and evaluation data, and assume that you have 5% additional error.

If you include noisy data resulting from incorrect labels in your training and evaluation data, it will be difficult to measure your accuracy. Do not believe that including noisy training data from hard-to-label items is OK. Many algorithms can remain accurate over noisy training data, but they assume predictable noise (random, uniform, Gaussian, and so on). If items were hard to label, those items are probably not randomly distributed.

The second example in figure 7.4 is the most common in industry, with hard examples being redirected to experts for human review. This method works fine if you have data that is fairly consistent over time.

If your data is changing quickly, the third method in figure 7.4 is recommended. In the third method, the expert annotators are looking at new data before the main annotation team, using active learning strategies to surface as many edge cases as

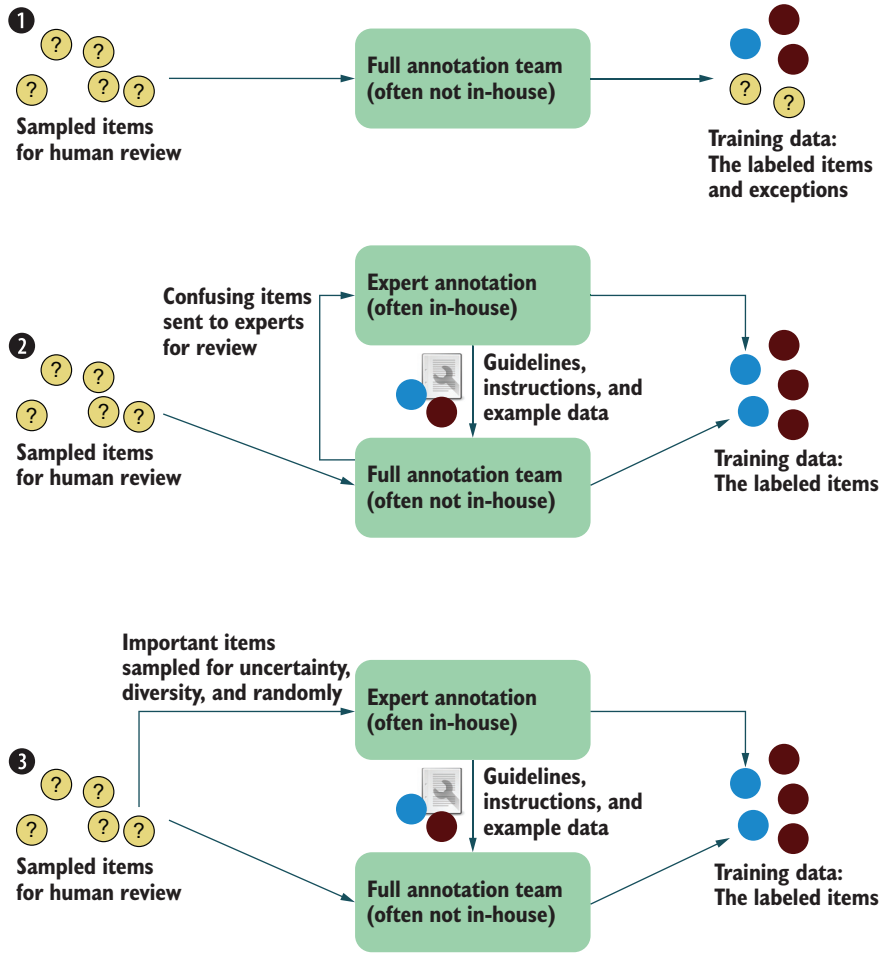


Figure 7.4 Three workflows for in-house annotation. The top workflow is recommended only for pilots; it doesn't use in-house annotators and ignores items that are hard to annotate. The second example is the most common in industry: hard examples are redirected to experts for human review. This approach works fine if you have data that is fairly consistent over time. If your data is changing quickly, the third method is recommended. In this method, the expert annotators are looking for potential new edge cases with diversity sampling and uncertainty sampling before the main annotation process happens; then these examples and updated guidelines are passed to the main annotation team. This method is the only one that ensures that the guidelines are not trailing the actual data. The expert annotators also annotate some randomly selected items for quality-control reasons that we will cover in chapter 8.

possible, to ensure that the guidelines are not trailing the actual data. If you have data coming in regularly, this method allows you to schedule your in-house workers more predictably than asking them to respond to ad-hoc hard examples, as in the second example.

You can combine the second and third methods, trying to get ahead of the new use cases as much as possible but still allowing the hard examples to be redirected for expert human review. You probably need to do this only if your data is particularly difficult to annotate or in the early iterations of annotations, before you have discovered all the major edge cases.

7.3 Outsourced workers

Outsourced workers are the fastest-growing workforce for data annotation. In the past five years, I have seen the amount of work going to outsourcing companies (sometimes called business-process outsourcers) grow at a greater rate than the other types of annotation workforces.

Outsourcing itself is nothing new. There have always been outsourcing companies in the technical industries, with large numbers of employees who can be contracted for different kinds of tasks. The best-known example is call centers. You are probably reaching a call center when you call your bank or a utility company, speaking to someone who is employed by an outsourcing company that is contracted by the company you are calling.

Increasingly, outsourcing companies focus on machine learning. Some focus only on providing a workforce; others also offer some machine learning technology as part of their broader offerings. Most of the time, workers in outsourcing companies are located in parts of the world where the cost of living is relatively low, meaning that salaries are lower too. Cost is often cited as the main reason to outsource; outsourcing is cheaper than hiring people to do the work in-house.

Scalability is another reason to use outsourced workers. It is often easier to scale outsourced workers up and down than it is to scale internal workforces. For machine learning, this flexibility can be especially useful when you don't know whether your application will be successful until you have a large amount of training data. If you are setting the expectations correctly with the outsourcing firm, this approach works better for everyone: you don't have to scale up in-house workers who expect their jobs to last longer, and the outsourcing firm can plan for its staff to switch tasks, which they do regularly and which should be factored into their compensation packages.

Finally, not all outsourced workers should be considered to be low-skilled. If an annotator has been working on annotation for autonomous vehicles for several years, they are a highly skilled individual. If a company is new to autonomous vehicles, outsourced workers can be a valuable source of expertise; they have intuition about what to label and what is important for models from their years of experience.

If you can't smooth out your annotation volume requirements for in-house workers, you might find a middle ground where you can smooth the annotation enough to use outsourced workers, who can cycle on and off faster than in-house workers (but not as fast as crowdsourced workers). Figure 7.5 shows an example.

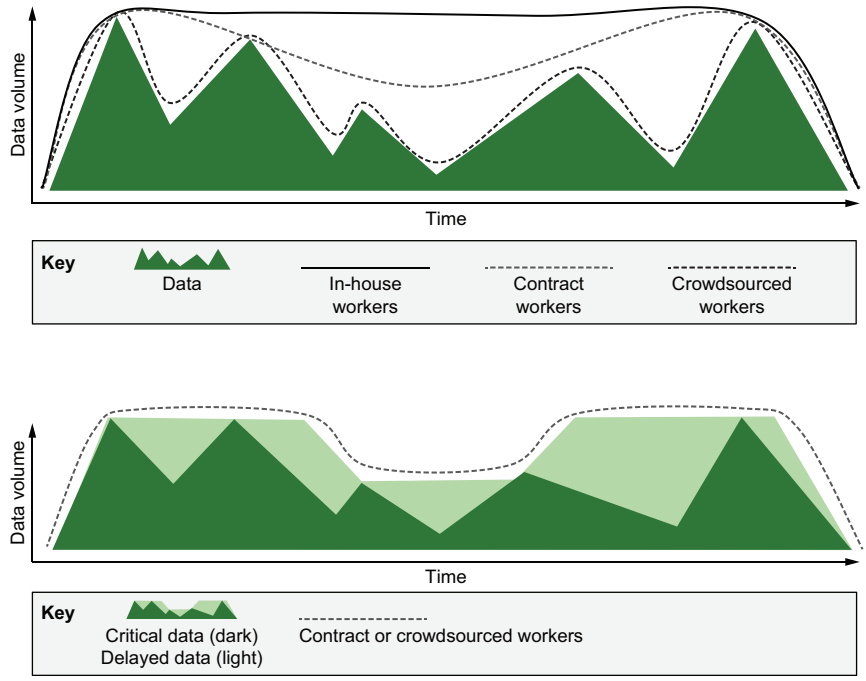


Figure 7.5 Smoothing your work for outsourced workers. If you can't fully smooth out the data volume, you might be able to smooth it out enough to fit the scale at which outsourced workers can scale up and down.

7.3.1 Salary for outsourced workers

Outsourced workers should be offered fair salaries by their employers, but you still have a responsibility to check that the compensation is fair. You have a duty of care for any person who works for you, even if they are contractors provided by another company. This duty of care comes from a power dynamic in which you are in the most powerful position. You don't want an outsourcing firm trying to win your business with lower prices by underpaying its workers. Especially if you are using an outsourcing firm that you are not familiar with, you should ask these questions:

- How much is each worker compensated per hour/day, and how does this pay compare to published numbers for the minimum wage and minimum cost of living at their location?
- Will workers get paid for the time that they are being trained on the task or only when they are annotating?
- Are workers compensated between projects or when your project has less work, or only when they're working directly on projects?

You can drill down into any of these questions. With regard to compensation, for example, you can ask about benefits such as health care, retirement, and paid leave.

Calculate whether compensation is fair for the cost of living where the workers are working, taking into account any reduced or absent pay between projects. Note that you are part of this equation; if you can provide a consistent stream of work for your projects, there will be less downtime for individual workers as they wait for your data to come in or as they switch between projects.

If an outsourcing organization can't give specific answers, it probably isn't a good option. At best, the company might be operating as a layer of management on top of other outsourcing firms or crowdsourced workers, and you don't want to be so many steps removed from your annotators because you are unlikely to get high-quality annotations. At worst, the organization might be covering up the fact that it pays exploitative wages.

Be mindful of cultural and national differences in compensation. Workers based in most countries in Europe have good national health care and may not think about the importance of employer-based health care, for example, and workers based in the United States may not expect a parental-leave benefit. The opposite can also be true: if you're in the United States, you don't need to insist that workers in countries with good national health care options also have employer-based health care.

My advice is to err on the side of asking more questions and apologize afterward if your questions came across as culturally insensitive. It is better to offend someone you pay fairly and then grow from that experience than to contribute to any person being compensated unfairly because you were afraid to ask.

7.3.2 Security for outsourced workers

Job security for outsourced workers comes from their direct employers. In addition to asking about their compensation, you should ask about job security and opportunities for advancement.

Many outsourcing firms have clear paths for promotion within their organizations. The annotators can become line managers, then site managers, and so on. They might also have specializations, such as being trusted with particular difficult annotation tasks and sensitive data, which are compensated at a higher rate.

If there is no opportunity for advancement within an organization for your workers, consider whether they should be compensated more for performing your tasks, knowing that they might need to pay for training and education out of pocket to advance their careers. It is also fine if someone is happy as a professional annotator and does not want to move into management or specialized roles. If the person is working in a positive environment, is paid fairly, and feels ownership of the work they are performing, then it is dignified work.

7.3.3 Ownership for outsourced workers

Outsourced workers are the group most likely to be working as annotators full-time. Therefore, transparency is important so that these workers know they are contributing to your organization. Like in-house workers, outsourced workers will be more

motivated if they have transparency into why they are annotating data. If they are annotating scenes in city parks, for example, they should know whether the use case focuses on pedestrians or plants. Knowing the goals greatly improves the accuracy of the data and gives workers a sense of how they are contributing to an important task.

If possible, let your outsourced workers feel that they are contributing directly to your company. Some companies actively avoid naming their outsourced workers as part of their organizations. This approach might be understandable from a brand perspective (concern that an employee of another company could misrepresent your company), but may be not fair if the goal is to hide the fact that a business process is being outsourced.

Regardless of your organization's policies, someone who works full-time as a contractor is contributing as much on a given day as someone who is employed full-time in-house, and they deserve to feel that way. To the extent possible, let these workers know how they are contributing to your organization, but also let them know if they can't talk about it publicly. There is often a middle ground where you can be transparent with outsourced workers about the value that they are creating, but make it clear that they can talk about it only privately. You probably have in-house people who can't talk about what they work on publicly either.

Outsourced workers are going to have less intuition about your company's goals than your in-house workers do. Your company might be a large multinational or a hot neighborhood startup, but there is no guarantee that an outsourced worker knows this fact, so don't assume too much. It is a win-win situation when an outsourced worker knows more about the context of the work they are completing: they will do higher-quality work that deserves higher pay at a faster rate and feel better about the process. Try to maintain a communication channel with your annotators while you are working with them.

7.3.4 Tip: Talk to your outsourced workers

If you are running a machine learning project, you should have direct communication with the line managers who are managing the annotation day to day. This communication can be emails, forums, or (ideally) something like online chat. Direct communication with the annotators themselves can be even richer, but depending on scale and privacy concerns, that interaction may not be permitted.

As a middle ground, you might have an open channel with the line manager and regular all-hands meetings with the annotators. If you have regular meetings, make it clear to the outsourcing company that this time is billable time for which the annotators should be paid. Questions always come up during annotation work, such as edge cases that you didn't think about and assumptions you made that weren't explicit in your guidelines. Also, it is respectful to give annotators direct communication with the people they are creating data for.

I've commonly seen outsourced workers four or five people removed from the people who build the machine learning models. Data scientists might engage someone

else internally to manage the data; that data manager works with an account manager at the outsourcing company; that account manager works with annotation leadership at the company; the annotation leadership work with the line managers; and finally, the line managers work with the individual annotators. That's five steps to communicate any guidelines or feedback!

Remember that in addition to paying the annotators, you are effectively paying for all the intermediate people. In some industries, 50% of the spend goes to this management overhead. If you're forced to use inefficient management structures, your communication doesn't have to follow that path. Establish direct relationships with the annotators or their immediate managers.

7.4 Crowdsourced workers

Crowdsourced workers who are paid on a per-task basis are the most-talked-about workforce for data annotation, but they are also the smallest group. I worked at the companies that have the two largest marketplaces for crowdsourced annotation work, and even then, I used outsourced workers (who were paid fair hourly wages) more than I used crowdsourced workers.

Online marketplaces for crowdsourced work typically allow you to post your annotation jobs; then people choose to complete that work for a posted price per task. If bonuses or pro-rata hourly wages are offered, these amounts are supplemental to the per-task price. The workers are typically anonymous, and this anonymity is often enforced by the platforms both technically and by their terms and conditions.

Because *crowdsourcing* is a general term that also includes data collection, annotation is also referred to as *microtasking*. Because the work is paid on a per-data-item basis, the process is also called *pay-per-task* or, more generally, considered to be part of the *gig economy*. In all cases, it refers to the greatest flexibility in work but also the greatest vulnerability to exploitation.

The biggest advantage of using crowdsourced workers is how quickly they can scale up and down. If you need only a few minutes of work, but you need it from thousands of people, pay-per-task crowdsourced workers are ideal.

In most companies, it is rare for any large, sustained machine learning project to rely on crowdsourced workers, who are more typically used for quick experiments to validate whether some level of accuracy on a new task is possible. Crowdsourced workers are also used when quick turnaround on annotations is needed on short notice, although some outsourcing companies offer 24/7 workforces that can agree to shorter turnaround times.

Because academic research is more often about quick experiments for different use cases than about sustained increases in accuracy for one use case, crowdsourced workers have been the annotation workforce of choice for many academic departments. The prominence of this work is why crowdsourced work is sometimes wrongly viewed as being a widespread method for annotation in industry. See "Don't apply graduate-student economics to your data labeling strategy" in section 7.4.3 for more

perspective on the relationship between academia and crowdsourcing in terms of how it influences real-world machine learning.

On the worker side, there are valid reasons why a person will choose to be a crowdsourced worker rather than work for an outsourcing company. The biggest reason is that there might not be an outsourcing company that can employ them at their location. You can be a crowdsourced worker from almost anywhere with an internet connection.

For someone who might otherwise face discrimination, crowdsourced work can be an equalizer. The anonymity makes it much less likely that workers are going to be discriminated against based on ethnicity, gender, criminal record, nationality, disability, or any other reason that often limits people's employment. Their work will be evaluated for its inherent merit.

Some people prefer crowdsourced work because they are limited to or prefer pay-per-task work. Perhaps they can contribute only a few minutes at a time due to other commitments, such as caring for their family or working at a full-time job that allows them time to do additional crowdsourced work. This area is the most difficult one in which to ensure fairness. If one worker takes 60 minutes to complete a task that takes most people 15 minutes, it is not fair to pay them for only 15 minutes; they should be paid for 60 minutes. If that pay is not within your budget, exclude those workers from accepting future tasks of yours in a way that doesn't negatively reflect on them in any online reputation system.

Crowdsourced workers are the most easily exploited workforce. It is difficult to predict how long some tasks will take in advance, so pay-per-task compensation can easily underpay someone even if the tasks are set up with good intentions. Also, it is easy for someone to set up tasks *without* good intentions, either misrepresenting the time it will take to complete a task or offering exploitative wages.

I have seen people make the argument that low-paid work (say, around $1 per hour) is better than nothing to someone who has free time and no other source of income. But this argument isn't true. It is ethically wrong to pay someone less than they need to live on. Beyond that, it contributes to that same inequality. If you have a business model that is viable only via exploitative wages, you are driving down the entire industry, and the rest of your industry can remain competitive only by doing the same. Therefore, you are helping create an entire industry that can survive only if it perpetuates an exploitative wage model, which helps no one.

7.4.1 Salary for crowdsourced workers

You should always pay crowdsourced workers fairly. All the major crowdsourcing platforms will tell you how long someone worked on your task, but this figure can be inaccurate because it relies on a browser to track time and may not include the time the worker spent studying the task before they started working on it.

I recommended paying people a fair hourly wage for the work they complete, based on their location in the world and published data on fair pay at that location. Every crowdsourcing marketplace allows you to pay effectively an hourly rate with

bonus structures, even if it is not possible to put hourly pay directly into the payment process. If you can't work out the exact amount of time that someone has spent, you should ask the workers directly rather than risk underpaying them. Software is available to help too.[1]

If you don't think that someone has completed crowdsourced work for you within your budget, every crowdsourcing platform will allow you to exclude them from future work. You should still pay them for the work that they completed. Even if you are 99% sure that they were not genuine in their work, you should pay them so that the 1% doesn't unfairly miss out.

Every major crowdsourcing platform allows you to restrict work to certain workers. This restriction might be implemented as qualifications that you can award or lists of worker IDs, but the result is the same: only those people will be allowed to work on your task. After you find people who can complete your tasks well, you can limit your most important work to them.

Every major crowdsourcing platform also implements some "trusted worker" category that automates how well someone has done in the past, typically based on the amount of work they have completed that was validated. These systems get scammed pretty easily with bots controlled by bad actors, however, so you will likely need to curate your own trusted-worker pool.

Producing good instructions is trickier than for other workforces, because you often can't interact directly. Also, the worker may not speak your language and is using machine translation in their browser to follow your instructions. If people are not being paid to read instructions, they are more likely to skim them and to be annoyed if they have to continually scroll past them, so it is important to have accurate, succinct instructions that make sense when translated into other languages with machine translation. That's not easy, but it is also the respectful thing to do; if you are paying people per task instead of per hour, you should make your interfaces as efficient as possible. I recommend breaking your task into simpler subtasks, not just for your benefit in quality, but so that workers who are being paid per task can be as efficient as possible and therefore earn more per hour.

7.4.2 Security for crowdsourced workers

Job security for crowdsourced workers comes primarily from the marketplace itself. The people who complete the tasks know that other work will be available when yours is complete.

For short-term security, it helps workers if you indicate how much work is available. If a worker knows that they can work on your tasks for many hours, days, or months, they are more likely to start your tasks, but job security may not be obvious to a worker if you break your tasks into smaller jobs. If you have only 100 items that need annotation in a given task, but you will be repeating that same task with millions of items in

[1] One recent example is in the paper "Fair Work: Crowd Work Minimum Wage with One Line of Code," by Mark Whiting, Grant Hugh, and Michael Bernstein (http://mng.bz/WdQw).

total, include that fact in your task description. Your task is more appealing for workers when they know that more work that they are familiar with is coming.

In general, you should factor in the fact that workers receive no benefits for pay-per-task jobs and may spend a lot of their time (often 50% or more) not getting paid while they are looking for projects to work on and reading instructions. Pay people accordingly, and pay extra for short, one-off tasks.

7.4.3 Ownership for crowdsourced workers

Like outsourced workers, crowdsourced workers typically feel more ownership and produce better results when they have as much transparency as possible. Transparency goes both ways: you should always elicit feedback about your task from crowdsourced workers. A simple comments field can be enough.

Even if you can't identify your company for sensitivity reasons, you should share the motivation for your annotation project and the benefits that it can have. Everyone feels better when they know they are creating value.

Don't apply graduate-student economics to your data labeling strategy

Too many data scientists take their data annotation experience from university to industry. In most computer science programs, data annotation is not valued as a science, or at least not as highly valued as algorithm development. At the same time, students are taught not to value their own time; it is fine for students to spend weeks working on a problem that could be outsourced for a few hundred dollars.

A compounding factor is that graduate students typically have little or no budget for human annotation. They might have access to a computer cluster or free credits from a cloud provider, but they might have no easy way to access funds to pay people to annotate new data.

As a result, pay-per-task crowdsourcing platforms are popular with graduate students who are trying to get the most out of their data budgets. They are also willing to spend a lot of time on quality control rather than pay people who have more expertise to ensure data quality, due to the same budget constraints. Because their tasks are often low-volume, they are rarely targeted by spammers, so quality seems to be artificially high.

Because the annotation itself is not part of the science that students are hoping to advance, annotation is often treated as a means to an end. Early in their careers, data scientists too often approach annotation with this same mindset. They want to ignore data as something that is not their problem and spend their own resources curating the right low-paid workers instead of paying better workers a fair salary to annotate the data more accurately.

Be careful that you don't have graduate-student economics wrongly influencing your data annotation strategy. Many of the suggestions in this chapter, such as holding internal data labeling sessions and establishing direct communication with outsourced annotators, will help your company culture and ensure that you are approaching data labeling in a way that benefits everyone.

7.4.4 Tip: Create a path to secure work and career advancement

There's a good chance that you will eventually want some annotators to be available full-time, even if you need only part-time workers to begin with. If there is a path to becoming a full-time worker, you should include this fact in your task description to attract the best workers.

You need to structure the possibility of full-time work based on individual merit, however, not make the situation competitive. If people know that they might miss out in a competitive environment, there is too great a power imbalance for the work to be fair, which could result in people compromising too much for the promise of future work. In plain terms, don't say something like "The 10 best people will get a 3-month contract." Say something like "Anyone who reaches volume X at accuracy Y will get a 3-month contract." Promise nothing if you can't make that commitment so that you don't inadvertently set up an exploitative environment.

If the marketplace offers feedback and review, use it! Anyone who has worked well deserves that recognition, which can help with their future work and career advancement.

7.5 Other workforces

The three workforces that you have seen so far—in-house workers, outsourced workers, and crowdsourced workers—probably cover most machine learning projects that you work on, but other types of workforces may fall between those categories. An outsourcing company might employ subcontractors in structures similar to crowdsourcing, or you might have in-house annotators who are employed as contractors at remote locations. You can apply the right combination of the principles of salary, security, and ownership for any of these configurations to be most respectful to these workforces and ensure that you get the best-quality work.

When running smaller companies, I have found a lot of success when contracting people directly rather than using outsourcing companies. Some online marketplaces for contract annotators allow you transparency into someone's past work, and it is easier to ensure that you are paying someone fairly and communicating openly when you are working with them directly. This approach doesn't always scale, but it can be successful for smaller one-off annotation projects.

You might also engage a handful of other workforces: end users, volunteers, people playing games, and computer-generated annotations. We'll cover them briefly in the following sections.

7.5.1 End users

If you can get your data labels for free from your end users, you have a powerful business model! The ability to get labels from end users might even be an important factor in deciding what products you need to build. If you can get your first working application from data labels that don't cost you anything, you can worry about running annotation projects later. By that point, you'll also have good user data to sample via active learning to focus your annotation efforts.

For many applications, users provide feedback that can power your machine learning models. Many applications that seem to rely on end users for training data, however, still use large numbers of annotators. The most obvious, widespread example is search engines. Whether you are searching for a website, a product, or a location on a map, your choice from the search results helps that search engine become smarter about matching similar queries in the future.

It would be easy to assume that search systems rely only on user feedback, but that isn't the case. Search relevance is the single largest use case for employing annotators. Paid annotators often work on the components. A product page might be indexed for the type of product (electronics, food, and so on), have the keywords extracted from the page, and have the best display images selected automatically, with each task being a separate annotation task. Most systems that can get data from end users spend a lot of time annotating the same data offline.

The biggest shortcoming of training data provided by users is that the users essentially drive your sampling strategy. You learned in chapters 3 and 4 how easy it is to bias your model by annotating the wrong sample of data. If you are sampling only data that seems to be the most interesting to your users on a given day, you run the risk of data that lacks diversity. Chances are that the most popular interactions from your users are not the same as those from a random distribution, or are the most important for your model to learn about, so you could end up with data that is worse than random sampling. Your model might end up accurate for only the most common use cases and be bad for everything else, which can have real-world diversity implications.

If you have a large pool of raw data, the best way to combat bias from end users is to use representative sampling to discover what you are missing from user-provided annotations and then get additional annotations for the items sampled via representative sampling. This approach will mitigate the bias in your training data if it has oversampled what is important to users instead of what was best for the model.

Some of the smartest ways to get user-generated annotations are indirect. CAPTCHAs are examples that you may encounter daily. A *CAPTCHA* (completely automated public Turing test to tell computers and humans apart) is the test that you complete to tell a website or application that you are not a robot. If you completed a CAPTCHA that asked you to transcribe scanned text or identify objects in photographs, there is a good chance that you were creating training data for some company. This use case is a clever one because if machine learning was already good enough to complete the task, that training data wouldn't be needed in the first place. A limited workforce exists for this kind of task, so unless you're in an organization that offers this kind of workforce, it's probably not worth pursuing.

Even if you can't rely on your users for annotations, you should use them for uncertainty sampling. If no data-sensitivity concerns apply, regularly look at examples in which your model was uncertain in its predictions while deployed. This information will enhance your intuition about where your model is falling short, and the sampled items will help your model when they are annotated.

7.5.2 Volunteers

For tasks that have an inherent benefit, you may be able to get people to contribute as volunteer crowdsourced workers. In 2010, I ran the largest use of crowdsourcing for disaster response. An earthquake struck Haiti and killed more than 100,000 people immediately, leaving more than a million homeless. I was responsible for the first step in a disaster-response and reporting system. We set up a free phone number, 4636, that anyone in Haiti could send a text message to, requesting help or reporting on local conditions. Most people in Haiti speak only Haitian Kreyol, and most people from the international disaster response community coming into Haiti only spoke English. So I recruited and managed 2,000 members of the Haitian diaspora from 49 countries to volunteer to help. When a text message was sent to 4636 in Haiti, a volunteer would translate it, categorize the request (food, medicine, etc.), and plot the location on a map. Over the first month following the earthquake, more than 45,000 structured reports were streamed to the English-speaking disaster responders, with a median turnaround of less than 5 minutes.

At the same time, we shared the translations with machine translation teams at Microsoft and Google so that they could use the data to launch machine translation services for Haitian Kreyol that were accurate for disaster-response related data. It was the first time that human-in-the-loop machine learning had been deployed for disaster response. This approach has become more common since, but rarely successfully when volunteers are engaged instead of paid workers.

Other high-profile, volunteer-driven projects that I have seen are in science, such as the gene-folding project Fold It,[2] but these projects tend to be the exception rather than the rule. In general, it is hard to get crowdsourced volunteer projects off the ground. Haiti was a special circumstance, with a large, well-educated group of people who wanted to contribute anything they could from far away.

If you are looking for volunteers, I recommend that you find and manage them via strong social ties. Many people try to launch volunteer crowdsourcing efforts with general callouts on social media, and 99% of them do not get the volume of people required. Worse, volunteers come and leave quickly, so they may not be ramped up to the right level of accuracy by the time they leave and have taken up a lot of resources for training. It is also demoralizing for volunteers who *are* providing substantial volumes of work to see so many people come and go.

When you reach out to people directly and build a community around a smaller number of volunteers, you are more likely to be successful. You will see this same pattern in open-source coding projects and projects like Wikipedia; the majority of work is done by a small number of people.

---

[2] "Building de novo cryo-electron microscopy structures collaboratively with citizen scientists," by Firas Khatib, Ambroise Desfosses, Foldit Players, Brian Koepnick, Jeff Flatten, Zoran Popović, David Baker, Seth Cooper, Irina Gutsche, and Scott Horowitz (http://mng.bz/8NqB).

7.5.3 People playing games

Gamifying work falls somewhere between paid workers and volunteers. Most attempts to get training data from games have failed miserably. You can use this strategy, but I don't recommend it as a way to get annotations.

The greatest success that I have had with games came when I worked in epidemic tracking. During an E-Coli outbreak in Europe, we needed people to annotate German news reports about the number of people affected. We couldn't find enough German speakers on crowdsourcing platforms, and this event predated outsourcing companies that specialized in annotation for machine learning. We ultimately found German speakers in an online game, Farmville, and paid them in virtual currency within the game to annotate the news articles. So people indoors in Germany were being paid in virtual agriculture to help track the real agricultural outbreak happening outside in German fields.

This case was a one-off use case, and it is difficult to see how it might have been exploitative. We paid small amounts of money per task, but the people playing the game were compensated for work that would have taken them 10 times longer within the game.

I have yet to see a game that generates interesting training data except for AI within the game itself or in focused academic studies. People spend an incredible amount of time playing online games, but this potential workforce is largely untapped right now.

Note that I do not recommended that you gamify *paid* work. If you force someone to do paid work within a gamelike environment, that person will get annoyed quickly if the work does not feel like the most efficient way to annotate data. Think about your own work. Would it be more fun if it had the kinds of artificial hurdles that you encounter in games?

There is also a lot of evidence that strategies such as providing a leaderboard are net-negative, motivating only a small number of leaders while demotivating the majority who are not near the top of the leaderboard. If you want to take any one thing from the gaming industry into paid work, use the principle of transparency: let people know their individual progress, but do it in terms of what they are contributing to your organization, not how they compare with their peers.

7.5.4 Model predictions as annotations

If you can get annotations from another machine learning application, you can get a lot of annotations cheaply. This strategy will rarely be your only strategy for getting annotations. If a machine learning algorithm can already produce accurate data, why do you need annotations for a new model? Using high-confidence predictions from your existing model as annotations is a strategy known as semi-supervised machine learning.

Chapter 9 covers using model predictions as annotations in more detail. All automated labeling strategies can perpetuate the existing biases of your model, so they

should be used in combination with human-annotated labels. All the academic papers showing domain-adaptation without additional human labels are in narrow domains.

Figure 7.6 shows an example of how to get started with computer-generated annotations when you want to avoid perpetuating bias and limitations in the past model as much as possible. First, you can autogenerate annotations with an existing model and select only those annotations that have high confidence. As you learned in chapter 3, you can't always trust the confidence alone, especially if you know that you are applying the model to a new domain of data. If the existing model is a neural network, and you can access its logits or hidden layers, also exclude predictions that have low overall activation in the model (model-based outliers), which indicate that they are not similar to the data on which the model was trained. Then use representative sampling to identify the items that you couldn't label automatically and sample those items for human review.

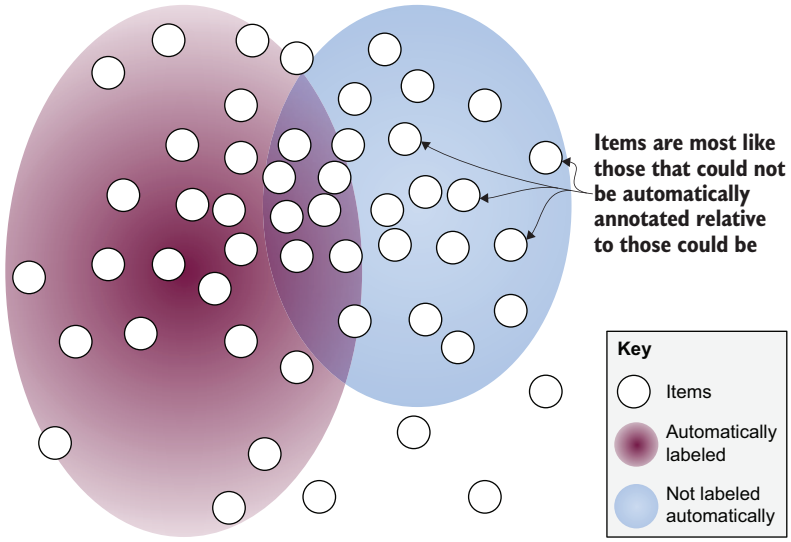


Figure 7.6 Using computer-generated annotations and then augmenting them with representative sampling. If you have a separate model for your task, you can autogenerate labels with that model. It is best to focus on highly confident predictions and (if you can access them) predictions with high activation in your network: model-based outliers. Then use representative sampling to identify the items that you couldn't label automatically and sample those items for human review.

For a slightly more sophisticated approach, you could use adaptive representative sampling to reduce the number of sampled items you need. With a little more sophistication, you could use a combination of clustering and representative sampling, as you learned in chapter 5. Combining clustering and representative sampling is ideal if the problem you are addressing is inherently heterogeneous in the feature space of your dataset.

Using computer-generated annotations can be the biggest kick-start to your model or the biggest rabbit hole, depending on your data and the quality of existing available models. To determine whether this approach is right for you, factor in the whole cost of the human annotation component. If you already need to spend a lot of time refining the right instructions and integrating and training a human workforce, you may not be saving much money by cutting down on how much the workers need to annotate. That is, the advantage might be less than you think.

In some cases, such as machine translation, using an existing model is the best starting point. It is expensive to get human translations for large amounts of data, so it will almost always be more cost-effective to bootstrap a model that starts with a dataset that was machine-translated in the first place.

Another case in which to use computer-generated annotations as a starting point is when you are adapting legacy systems to newer machine learning models. Suppose that you have a legacy system with a lot of hand-coded rules or hand-tuned systems for extracting the right features, and you want to adapt that system to a newer neural machine learning system that doesn't require handcrafted rules or features. You can apply the legacy system to a large amount of raw data and use the resulting predictions as your annotations. It is unlikely that this model will immediately achieve the accuracy you want, but it can be a good starting point, and additional active learning and annotation can build on it. Chapter 9 covers many methods of combining model predictions with human annotations—an exciting and rapidly growing area of research.

7.6 Estimating the volume of annotation needed

Regardless of the workforce you use, you often need to estimate the total amount of time needed to annotate your data. It is useful to break your annotation strategy into four stages as you annotate more data:

- *Meaningful signal*—Better-than-chance accuracy. Your model's accuracy is statistically better than chance, but small changes in parameters or starting conditions produce different models in the accuracy and in which items are classified correctly. At this point, you have enough signal to indicate that more annotations should increase accuracy and that this strategy is worth pursuing.
- *Stable accuracy*—Consistent but low accuracy. Your model's accuracy is still low, but it is stable because small changes in parameters or starting conditions produce models that are similar in terms of accuracy and in which items are classified correctly. You can start to trust the model's confidence and activation at this stage, getting the most out of active learning.
- *Deployed model*—High-enough accuracy for your use case. You have a model that is accurate enough for your use case, and you can start deploying it to your applications. You can start identifying items in your deployed models that are uncertain or that represent new, unseen examples, adapting your model to the changing data it encounters.

- *State-of-the-art model*—Industry-leading accuracy. Your model is the most accurate in your industry. You continue to identify items in your deployed models that are uncertain or that represent new, unseen examples so that you can maintain accuracy in a changing environment.

In every industry I have seen, the state-of-the-art model that won long-term was the winner because of better training data, not because of new algorithms. For this reason, better data is often referred to as a *data moat*: the data is the barrier that prevents your competitors from reaching the same levels of accuracy.

7.6.1 The orders-of-magnitude equation for number of annotations needed

The best way to start thinking about the amount of data needed for your project is in terms of orders of magnitude. In other words, the number of annotations needed to grow exponentially to hit certain milestones in model accuracy.

Suppose that you have a relatively straightforward binary prediction task, such as the example in chapter 2 of this book of predicting disaster-related and non-disaster-related messages. You might get a progression that looks something like this, assuming that N=2 (figure 7.7):

- 100 (10^{N}) annotations—Meaningful signal
- 1,000 (10^{N+1}) annotations—Stable accuracy
- 10,000 (10^{N+2}) annotations—Deployed model
- 100,000 (10^{N+3}) annotations—State-of-the-art model

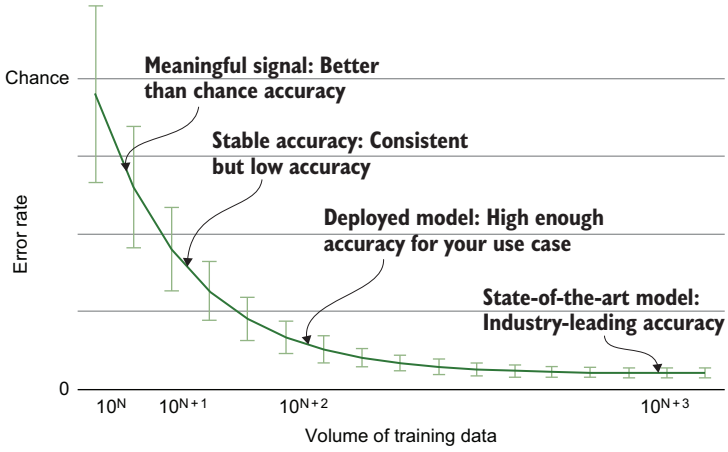


Figure 7.7 The orders-of-magnitude principle for training data. Estimate an order of magnitude more data to move from meaningful signal to stable accuracy to deployed model and to state-of-the-art model. Use this principle as a benchmark to estimate the amount of data you need before you start seeing what your actual increase in accuracy looks like as you start annotating data.

You will be able to reduce the number of items you need to annotate with active learning and transfer learning, but the step function is still approximately exponential with a lower N (say, N=1.2). Similarly, it may take more annotations for tasks that a large number of labels or complicated tasks such as full-text generation (say, N=3), in which case you should assume that the step function is still approximately exponential but with a higher N.

When actual annotations start coming in, you can start plotting your true increase in accuracy and making better estimates of how much data you need. The plotted increase in accuracy (or decrease in error, as in figure 7.7) is called the *learning curve* of your model, but this name doubles up: people often refer to the increase in accuracy as a single model converges as the learning curve too. If your machine learning framework of choice shows you a learning curve, check whether that name refers to an increase in accuracy with more data or an increase in accuracy as the model converges on a constant amount of data. The two cases are not the same.

Even when you have your own data coming in, it is a good idea to keep the diminishing returns of figure 7.7 in mind. It can be exciting when your accuracy moves up quickly with the first 100 or 1,000 annotations but less so when your accuracy improves much more slowly thereafter. This experience is typical. Don't jump too quickly into playing with algorithm architectures and parameters because that is what you are most familiar with. If you can see that the accuracy is improving with more data but the rate is slowing exponentially, this model may be behaving as expected.

7.6.2 Anticipate one to four weeks of annotation training and task refinement

You've got your machine model ready to go, and you've proved that it works with a popular open-source dataset. Now you're ready to turn on the firehose of real annotated data for your application!

If you haven't set up your annotation strategy in parallel, you're in for a surprise: you probably need to wait a few weeks. Waiting is frustrating, but as I recommend at the start of this chapter, you should start your data and algorithm strategies at the same time. If you find that data is too different from the open-source dataset that you first piloted on (maybe some labels are much rarer or the diversity of data is much higher), you will be back to the drawing board for your machine learning architecture in any case. Don't rush your annotation, but if you have to rush for quick results, be prepared to drop those annotations later because they will have too many errors because of the lack of quality control.

It will likely take several iterations with your data labeling leaders to get the instructions correct, to investigate any systematic errors, and to refine your guidelines appropriately before you can confidently turn on the firehose to annotate large amounts of data.

Expect it to take a few weeks to get the annotation process working smoothly, not a few days (although it shouldn't take you many months to get an annotation process working smoothly). If the task is a simple one, such as labeling photographs with a

relatively small number of labels, it will take closer to one week; you will need crisp definitions of what counts for each label, but that should not take too long to refine. If you have a more complicated task with unusual data and labeling requirements, it will take closer to a month to refine your task and for the annotators to ramp up in training, and you will be continually refining your task as more edge cases are discovered.

If you need data right away while you wait for your annotation workforce to get trained, start annotating data yourself. You will learn a lot about your data, which will help with both your models and your annotation guidelines.

7.6.3 Use your pilot annotations and accuracy goal to estimate cost

When you have refined your annotation process to the point where you are confident that your guidelines are comprehensive and your annotators are trained on your task, you can estimate the cost. Take into account your accuracy requirement, using the guideline for orders of magnitude in section 7.6.1 to estimate the total number of annotations required. Do you need state-of-the-art? If so, you can multiply the orders of magnitude required for state-of-the-art results by your per-annotation cost and estimate the total cost. The result might help determine your product strategy. If you don't have the budget to get to state-of-the-art accuracy, as you originally planned, you may still be able to get high-enough accuracy for your use case, which could change your product development strategy. It is important to be honest with yourself and your stakeholders about the accuracy you can achieve. If your model was state-of-the-art on an open-source dataset, but it will not reach that accuracy on your own data due to budget constraints, then you need to set expectations for all the stakeholders in your project.

One variable that we have not covered yet is the number of annotators per item. You will often give the same task to multiple people to find agreement among them and produce training data that is more accurate than any single annotator can create. We will cover that method of quality control in chapter 8. For now, it is enough to understand that you may end up with multiple annotations per item, and that result needs to be part of your budget.

Your budget for labeling might be fixed from the start, of course. In that case, make sure that you carefully implement good active learning strategies so that you get the most out of each annotation.

7.6.4 Combining types of workforces

One common reason that you might want to combine workforces is quality control. Workflows and choice of labeling workforce are common ways to ensure that you get accurate labels for your data (chapter 8). Other common reasons include data sensitivity and complexity, meaning that some data is too sensitive or complex to be outsourced and some is not, resulting in multiple workforces.

When working at larger companies, I typically engaged multiple data labeling companies at the same time to derisk my pipelines, not relying on any vendor to be the only source of data labels. If you end up with multiple workforces, you obviously need

to work out the budget for each workforce and combine the budgets to get your total project spend.

Summary

- There are three main types of workforces for annotation: in-house, outsourced, and crowdsourced. Understanding these workforces will help you choose which workforce or combination is best for your tasks.
- The three key principles of motivating annotators are salary, security, and transparency. Understanding how to apply these principles to different workforces will ensure that you get the best possible work by having the happiest possible workforce.
- You can consider some nonmonetary compensation systems, including application end users, volunteers, and computer-generated data/annotations. You may want to consider these alternative workforces when you are constrained by budget or need special qualifications.
- Don't apply graduate-student economics to your data labeling strategy.
- The orders-of-magnitude principle lets you estimate your total annotation volume requirements. This principle helps you plan your annotation strategy with meaningful early estimates that you can refine as you continue.

8 Quality control for data annotation

This chapter covers

- Calculating the accuracy of an annotator compared with ground truth data
- Calculating the overall agreement and reliability of a dataset
- Generating a confidence score for each training data label
- Incorporating subject-matter experts into annotation workflow
- Breaking a task into simpler subtasks to improve annotation

You have your machine learning model ready to go, and you have people lined up to annotate your data, so you are almost ready to deploy! But you know that your model is going to be only as accurate as the data that it is trained on, so if you can't get high-quality annotations, you won't have an accurate model. You need to give the same task to multiple people and take the majority vote, right?

Unfortunately, your annotation task is probably much harder. I've seen annotation underestimated more often than any other part of the human-in-the-loop machine learning cycle. Even if you have a simple labeling task—such as deciding whether an image contains a pedestrian, an animal, a cyclist, or a sign—how do you decide on the right threshold for majority agreement among annotators when all those annotators have seen different combinations of tasks? How do you know when your overall agreement is so low that you need to change your guidelines or the way you define your task? The statistics to calculate agreement in even the simpler labeling tasks are more advanced than the statistics underlying most neural models, so understanding them takes time and practice.

This chapter and the next two chapters use the concepts of *expected* and *actual* annotation accuracy. If, for example, someone guessed randomly for each annotation, we would expect them to get some percentage correct, so we adjust the actual accuracy to account for a baseline of random chance. The concepts of *expected* and *actual* behavior apply to many types of tasks and annotation scenarios.

8.1 Comparing annotations with ground truth answers

The simplest method for measuring annotation quality is also one of the most powerful: compare the responses from each annotator with a set of known answers, called *ground truth answers*. An annotator might annotate 1,000 items, of which 100 have known answers. If the annotator gets 80 of those known answers correct, you can estimate that they are 80% accurate over the 1,000 items.

You can implement the creation of ground truth data incorrectly in many ways, however, and unfortunately almost all errors make your dataset appear to be more accurate than it is. If you are creating your evaluation data and training data at the same time and don't have good quality controls, you will end up with the same kinds of errors in both your training data and evaluation data. The resulting model may predict the wrong label in some contexts, but the ground truth evaluation data will have the same type of errors, so you may not realize that you have the errors until you deploy your application and it fails.

The most common cause of errors is wrong items sampled for ground truth. Three general sampling strategies identify the items that should become ground truth data:

- *A random sample of data*—You should evaluate the accuracy of your individual annotators on random data. If a random selection isn't possible, or if you know that a random sample is not representative of the population that your application is serving, you should try to get a sample that is as close to representative as possible.
- *A sample of data with the same distribution of features and labels as the batch of data that is being annotated*—If you are using active learning, this sample should be a random sample from your current iteration of active learning, which allows you to calculate the (human) accuracy of each sample of data and, by extension, the accuracy of the dataset as a whole.

- *A sample of data found during the annotation process that is most useful for annotation guidelines*—These guidelines often exemplify important edge cases that are useful for teaching the annotators to be as accurate as possible.

Within our diagram for human-in-the-loop architectures, if we zoom in on the annotation component, we see that the workflow is a little more complicated than the high-level diagram shown in figure 8.1.

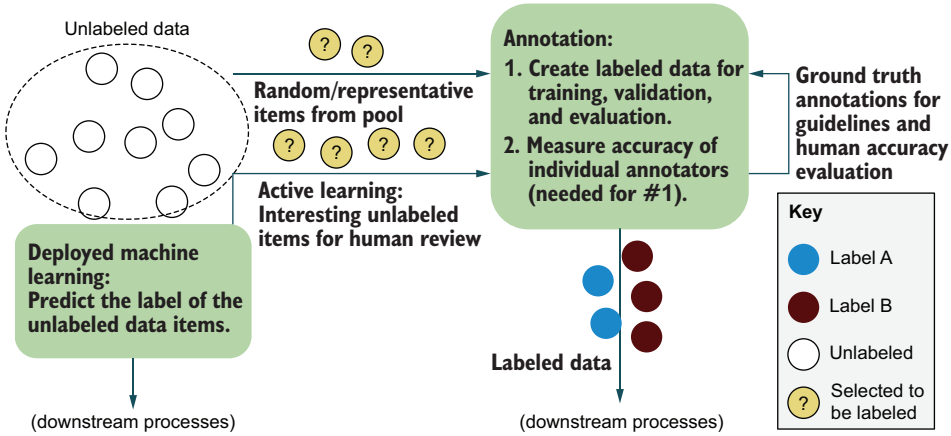


Figure 8.1 The flow of information for annotation. In addition to taking data sampled according to our current active learning strategies, we sample a random or representative set of data and data that some annotators have already seen. Sampling random/representative data lets us calculate the accuracy of annotators in a way that makes it easier to determine their reliability across datasets and whether they are candidates for promotion to experts. Sampling within the current active learning batch allows us to calculate the accuracy for this particular dataset. Sampling during the annotation process finds items that are most useful for annotation guidelines and for adjudication by experts.

To be confident that your ground truth items are as accurate as possible, you need to draw on many of the methods in this chapter and possibly the next two chapters. You must be confident that your ground truth items have few errors; otherwise, you will create misleading guidelines and won't have reliable accuracy metrics, resulting in bad training data. You can't cut corners. If your ground truth items are only items that had the highest agreement, you are likely to have oversampled the easiest items to annotate, which will make your accuracy look better than it is.

When you have a ground truth dataset that you can use to evaluate each annotator, you can calibrate your annotation projects to be higher-quality and more efficient. Using interannotator agreement for quality control also becomes much more effectively with a small but reliable ground truth dataset to support it. As chapter 9 shows, you can still get reliable signals from the least accurate annotator when you know their pattern of errors.

In this chapter and chapter 9, we will use the example data shown in figure 8.2. Although your datasets will have many more items than the 11 rows in figure 8.2, these 11 rows are enough for learning the kinds of quality controls that you might implement.

Annotator/ annotations	Alex	Blake	Cameron	Dancer	Evan
Task 1	Pedestrian	Pedestrian	Pedestrian		
Task 2		Sign	Sign	Sign	
Task 3	Pedestrian	Pedestrian	Cyclist	Cyclist	Pedestrian
Task 4		Cyclist	Cyclist	Cyclist	
Task 5	Pedestrian	Pedestrian	Pedestrian	Pedestrian	
Task 6	Cyclist	Cyclist			Cyclist
Task 7	Pedestrian	Pedestrian		Pedestrian	
Task 8	Animal	Animal		Animal	
Task 9	Sign		Animal	Animal	Animal
Task 10		Sign	Sign		Sign
Task 11		Animal			
...					

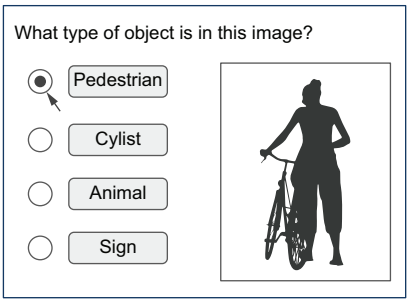


Figure 8.2 Throughout the remainder of this chapter and in chapters 9 and 10, we will use this example data. Five annotators—named Alex, Blake, Cameron, Dancer, and Evan—have annotated an image according to the object in that image. We will assume that the image is the same type that was used in previous chapters, with four labels, "Animal," "Cyclist," "Pedestrian," and "Sign." In this example, Alex has seen seven images (tasks 1, 3, 5, 6, 7, 8, and 9); annotated the first three as "Pedestrian"; and annotated each of the rest as "Cyclist," "Pedestrian," "Animal," or "Sign." The image on the right shows what the annotation interface might look like.

We will use different variations of the correct answer for the data in figure 8.2 throughout this chapter but keep the annotations the same as in the figure. For this section, let's assume that we had ground truth labels for each of these examples.

What should you call an annotator?

Many terms are used for a person who creates training and evaluation data, including *rater, coder, adjudicator, agent, assessor, editor, judge, labeler, oracle, worker,* and *turker* (from the Mechanical Turk platform, sometimes used for other software). In industry, the annotator might go by their job title, such as *analyst*, by the skill they are using, such as *linguist*, or by their employment status, such as *contractor* or *gig-economy worker*. In other cases, an annotator is referred to as a *subject-matter expert,* sometimes truncated to *expert* or to the acronym *SME* (pronounced "smee").

If you are searching for additional reading, make sure to try the different names as search terms. You might find similar papers on *interannotator agreement*, *inter-rater agreement*, and *intercoder agreement*, for example.

This book uses the term *annotator* because it is the least likely to be confused with any other role. If you're working with people who annotate data, use the correct title for that person in your organization. This book also avoids saying *training* annotators (to eliminate confusion with training a model) and uses terms such as *guidelines* and *instructions* instead of *training materials*. Again, use the preferred description in your organization for the process of teaching annotators the instructions for a given task.

8.1.1 Annotator agreement with ground truth data

The basic math for agreement with ground truth data in labeling tasks is simple: the percentage of known answers that an annotator scored correctly. Figure 8.3 gives hypothetical accuracy for each annotator on our example data.

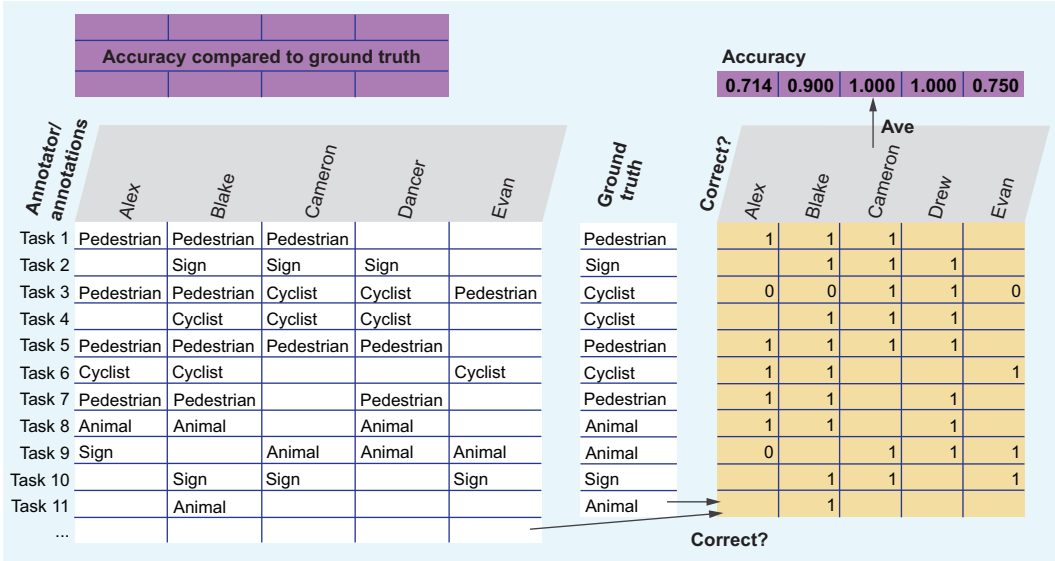


Figure 8.3 An example of annotator accuracy compared with ground truth data. Assume that the Ground Truth column has the known answers for each task (image labels). We calculate each annotator's accuracy as the fraction that they got correct.

You typically want to adjust results like those in figure 8.3 according to a baseline of random chance guessing. We can calculate three baselines for our random-chance labeling. Let's assume that 75% of the images are "Pedestrian," 10% are "Sign," 10% are "Cyclist," and 5% are "Animal." The three baselines are

- *Random*—The annotator guesses one of the four labels. This baseline is 25% in our example data because we have four labels.
- *Most frequent label (mode label)*—The annotator knows that "Pedestrian" is the most frequent label, so they always guess that label. This baseline is 75%.
- *Data frequency*—The annotator guesses according to the frequency of each label. They guess "Pedestrian" 75% of the time, "Sign" 10% of the time, and so on. This baseline can be calculated as the sum of the squares of each probability.

Figure 8.4 shows the calculations.

The adjusted accuracy normalizes the annotator's score so that the baseline from random guessing becomes 0. Let's assume that someone had 90% accuracy overall. Their actual accuracy, adjusted for chance, is shown in figure 8.5.

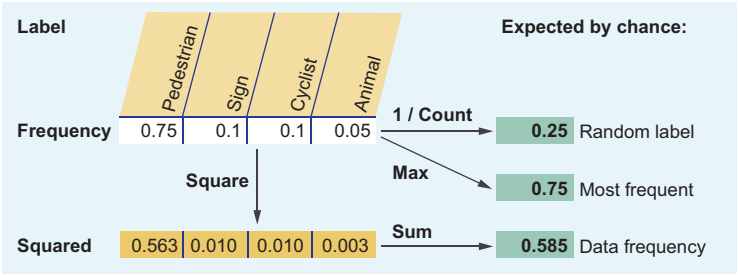


Figure 8.4 The three calculations for different accuracies that would be expected through random chance, showing a wide range of expected accuracy depending on what baseline we use

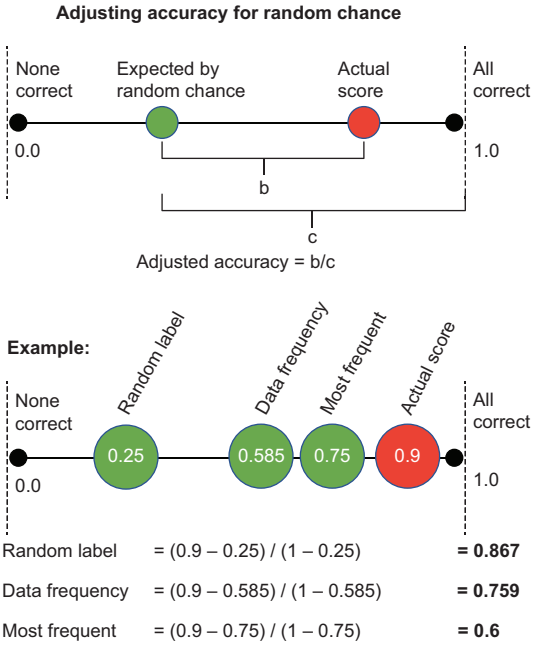


Figure 8.5 Different ways of establishing a baseline expected from random guessing or chance-adjusted accuracy when testing annotators against ground truth data. *Top:* How we normalize the result. If someone was randomly choosing a label, they would sometimes pick the correct one, so we measure accuracy in terms of distance between the random accuracy and 1. *Bottom:* How the different adjusted accuracies look with our example data. Note that the normalized score of 60% accuracy for always guessing "Pedestrian" is different from the 90% raw accuracy score or 86.7% when normalized according to the number of labels. This example highlights why the correct baseline for expected accuracy is so important. There are cases in which each of the three baselines is the better choice, so it is important to know all three.

As figure 8.5 shows, we have different ways to normalize the annotation counts. The most common one used in the statistics community is *data frequency*—a datacentric way to think of expected behavior. It always falls between the random selection and the most frequent, so it has the nice property of being the safe middle option.

Because the expected baseline becomes zero, any result less than zero means that the person guessed worse than random chance. Typically, this result means that the annotator understood the instructions incorrectly or was scamming the system in a simple way, such as always guessing a response that isn't the most frequent. In any of these cases, normalizing the baseline to zero gives us an easy way to set up alerts for

any task. No matter what the task is, a negative score after adjusting for random chance should cause an alert in your annotation process!

If you are familiar with the literature on quality control for annotation, you know that a metric that is normalized according to the expected behavior is often called *chance-corrected* or *chance-adjusted*. In many cases throughout this book, the expected behavior is not random chance, such as when we ask annotators what they expect other annotators to choose (chapter 9). The more general term *expected* is used for those cases, but for objective labeling tasks, *expected* and *chance* mean the same thing.

8.1.2 Which baseline should you use for expected accuracy?

For the three baselines for expected accuracy—random, data frequency, and most frequent—calculating all three metrics will help with your intuition about the data. The right metric to normalize your accuracy will be specific to your task and the experience of the people labeling the data.

When a person is first working on a task, they will not have intuition about which label is more frequent, so they are more likely to be closer to random labeling. But after some time, they realize that one label is much more frequent than the others and may feel safe guessing that label when they are uncertain. For that reason, chapter 11 is devoted entirely to user interfaces for annotation.

My practical recommendation is to wait until an after annotator becomes familiar with a task and then apply the strictest baseline: the most frequent label. You can consider the first few minutes, hours, or days of your task to be a ramp-up period for the annotator to become familiar. When an annotator has a strong intuition about the data, they will be taking the relative frequency of the labels into account. As you will see in section 8.2.3, however, data frequency is more relevant for calculating agreement at the level of the entire dataset. So it's important to understand all the baselines and apply them at the right time.

Good quality control for data annotation can take a lot of resources and should be factored into your budget. See the following sidebar for an example of how quality control led to the engagement of a different set of annotators for a project.

Consider the total cost of annotation projects

Expert anecdote by Matthew Honnibal

It helps to communicate directly with the people who are annotating your data, like anyone else in your organization. Inevitably, some of your instructions won't work in practice, and you will need to work closely with your annotators to refine them. You're also likely to keep refining the instructions and adding annotations long after you go into production. If you don't take the time to factor in refining the instructions and discarding wrongly labeled items, it is easy to end up with an outsourced solution that looked cheap on paper but was expensive in practice.

(continued)

In 2009, I was part of a joint project between the University of Sydney and a major Australian news publisher that required named entity recognition, named entity linking, and event linking. Although academics were increasingly using crowdsourced workers at that time, we instead built a small team of annotators that we contracted directly. This ended up being much cheaper in the long run, especially for the more complicated "entity linking" and "event linking" tasks where crowdsourced workers struggled and our annotators were helped by working and communicating with us directly.

Matthew Honnibal is creator of the spaCy NLP library and co-founder of Explosion. He has been working on NLP research since 2005.

8.2 Interannotator agreement

When data scientists talk about their machine learning models being more accurate than people, they often mean that the models are more accurate than the average person. Speech recognition technologies, for example, are now more accurate than the average English speaker for nontechnical transcription in common accents. How can we evaluate the quality of these speech recognition technologies if humans can't create evaluation data with that level of accuracy?

The "wisdom of the crowd" produces data that is more accurate than any one human. For more than a century, people have studied how to aggregate the judgments of multiple people into a single, more accurate result. In the earliest examples, it was famously shown that when multiple people guess the weight of a cow, the average of all the guesses was close to correct. That result doesn't mean that everyone was less accurate than average: individuals will guess the weight of a cow more accurately than the average, but the average guess was closer to the real weight than *most* people.

So when data scientists brag that their model is more accurate than humans, they often mean that their model is more accurate than the agreement among the annotators, which is called *interannotator agreement*. Model accuracy and annotator agreement are two different numbers that shouldn't be compared directly, so try to avoid making this common mistake.

It is possible, however, to create training data that is more accurate than every individual person who contributed to annotations, and this chapter returns to this topic in section 8.3 after introducing the basics.

8.2.1 Introduction to interannotator agreement

Interannotator agreement is typically calculated on a –1 to 1 scale, where 1 is perfect agreement, –1 is perfect disagreement, and 0 is random-chance labeling. We calculate the agreement by asking how much better our agreement is than expected, similar to our earlier individual annotator accuracy score, but in this case for agreement. Figure 8.6 shows an example.

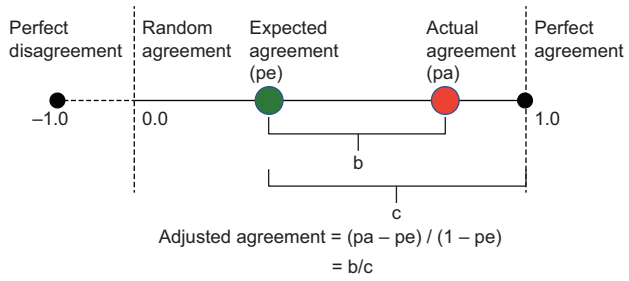


Figure 8.6 How agreement metrics are calculated. Agreement is typically on a –1 to 1 scale, where 1 is perfect agreement, –1 is perfect disagreement, and 0 is random distribution. The resulting agreement is variously known as *actual agreement*, *adjusted agreement*, or *agreement adjusted for random chance*.

Figure 8.6 shows how we calculate agreement that takes random chance agreement into account. This adjustment is similar to adjusting accuracy according to ground truth answers, but in this case, it compares annotators.

We cover different types of interannotator agreement in this book, including overall agreement at the level of the entire dataset, individual agreement between annotators, agreement between labels, and agreement on a per-task basis. The concepts are fairly simple, and we will start with the simple naive agreement algorithm in figure 8.7. This algorithm is so simple that you shouldn't use it, but it is a useful starting point for understanding the equations in this chapter and the next two chapters.

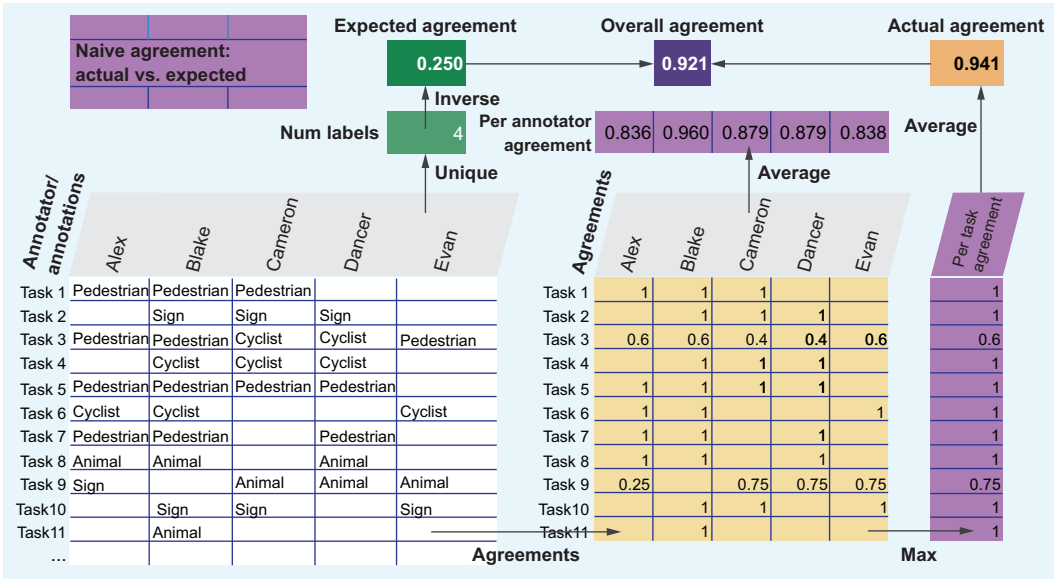


Figure 8.7 A naive way to find agreement per annotator, agreement per task, and overall agreement for the entire set of annotations. We calculate the expected agreement in terms of randomly selecting one of four labels. We calculate the agreements for each task in the large middle table. We derive the per-person and per-task agreements from the Agreements table. We derive the overall agreement by using the combination of the expected and the average task-level agreement. Although you shouldn't use this method for your actual data, because it is too simple, the diagram is useful for highlighting the concepts.

Figure 8.7 shows the basic idea behind three types of agreements. Although all of these calculations are sensible, they fall short a little. Here are some shortcomings of figure 8.7, highlighting the complications in calculating agreement:

- The overall expected agreement is based on the number of labels, but some labels are more frequent than others. If a fifth label was never chosen, it would seem strange to reduce the overall expected agreement as a result.
- The person agreement seems to unfairly penalize people for errors that other people make on the same task. Evan, for example, always agrees with the majority vote for the label but has the second-lowest agreement score.
- The task agreement scores seem to be overly optimistic because they do not take into account the accuracy of individual annotators.
- The actual agreement averages the task agreement, but it would be much lower if we decided to calculate it by averaging the person agreement. What is the right way to aggregate the individual agreements to produce a more correct overall observed actual agreement?
- Task 11 has only one response, so it seems wrong to calculate the response as being 100% in agreement; there is nothing for that one response to agree with.
- We are not tracking agreement for labels. Is "Pedestrian" more likely to be confused than "Sign," for example?
- We are not taking the overall number of annotations into account. Especially with a relatively small number of annotations, there might be artifacts of the data size (although this is less relevant for typical training datasets with thousands of items).

You can play around with this implementation as a spreadsheet at http:// mng.bz/E2qj. This spreadsheet also contains some of the other equations in this chapter.

Sections 8.2.2 through 8.2.7 are dedicated to the best ways to address these issues. Although the math gets more complicated than anything you've seen so far in this book, keep in mind that it is solving one simple question:

> *How can we fairly calculate the agreement between annotators to evaluate the accuracy of our dataset, individual tasks, individual labels, or individual annotators?*

8.2.2 Benefits from calculating interannotator agreement

You can use interannotator agreement as part of your human-in-the-loop machine learning strategy in multiple ways:

- *The reliability of your dataset*—Do the annotators agree with one another often enough that you can rely on the labels that have been created? If not, you may need to redesign your instructions or the task as a whole.
- *The least reliable annotators*—Do any individual annotators disagree with the others too often? They may have misunderstood the task or may not be qualified to keep taking part. Either way, you may want to ignore their past annotations and potentially get new judgments. Alternatively, an unreliable annotator may in

fact have valid but underrepresented annotations, especially for subjective tasks (see "Measuring natural variation" later in this list).

- *The most reliable annotators*—The annotators with high agreement are likely to be the most accurate for your task, so identifying these people for potential reward and promotion is helpful.
- *Collaboration between annotators*—Do any annotators agree nearly perfectly? They might be sharing notes innocently because they sit near one another, in which case you need to remove those responses from any calculations of agreement that assumes independence. On the other hand, this result may be evidence that a bot is duplicating one person's work so that the person wrongly gets paid twice. Regardless of the underlying cause, it is helpful to know when two sets of answers are only one set that has been repeated.
- *An annotator's consistency over time*—If you give the same task to the same person at different times, do they give the same result? This metric, known as *intra-annotator agreement*, can be evidence that an annotator is not paying attention, that your task has ordering effects, and/or that the task is inherently subjective. Also, the annotator may be genuinely changing their mind as they see more data, which is known as *concept evolution*.
- *Creating examples for the instructions*—You can assume that items with high agreement among a large number of annotators are correct and let these items become examples in the guidelines for new annotators. Because you run two risks with this strategy—some errors will still get through and propagate, and only easier tasks will get through with higher agreement—you should not use it as your only strategy for creating ground truth data.
- *Evaluating the inherent difficulty of a machine learning problem*—In general, if the task is hard for humans, it will be hard for your model. This information is especially helpful for adapting to new domains. If your data historically has 90% agreement, but data from a new source has only 70% agreement, this result tells you to expect your model to be less accurate on data from that new source.
- *Measuring the accuracy of your dataset*—If you know the individual reliability of each annotator and how many people have annotated each item, you can calculate the probability that any given label will be annotated incorrectly. From this result, you can calculate the overall accuracy of your data. Taking individual annotator accuracy into account gives you a better upper boundary for the accuracy of a model that is trained on the data, compared with simple interannotator agreement. Models can be more or less sensitive to noise in the training data, so the limit is not a hard one. The limit *is* a hard limit on how precisely you can measure your model's accuracy, because you can't calculate your model's accuracy to be higher than your dataset's accuracy.
- *Measuring natural variation*—For some datasets, lack of agreement is a good thing because it can indicate that multiple annotation interpretations are valid. If you have a task that is subjective, you may want to ensure that you have a

diverse selection of annotators so that no one set of social, cultural, or linguistic backgrounds is inadvertently resulting in biased data.

- *Escalating difficult tasks to experts*—This example was covered in chapter 7, and we return to it again in section 8.5. Low agreement between less-qualified workers might mean that the task should be routed to an expert automatically for review.

The remainder of section 8.2 contains some of the best current methods for calculating agreement in your data.

Don't use agreement as the only measure of accuracy

You should not rely on interannotator agreement alone to find the correct label for your data; always use interannotator agreement in combination with ground truth data. Many data scientists resist this practice, because it means losing training data. If 5% of the labeled data is set aside for quality control, for example, they have 5% less data for their model to train on. Although no one likes having less training data, in the real world you can have the opposite effect: if you are relying on interannotator agreement alone for your labels, you will use more than 5% more human judgments because you can use ground truth data to calibrate your agreement better.

Looking at agreement alone can also hide cases in which the wrong annotations agree. Without ground truth data, you won't be able to calibrate for these errors.

On the other hand, agreement allows you to extend your accuracy analysis beyond what is practical with ground truth data alone, so you get the biggest benefits when you combine agreement with ground truth data. For example, you can calculate the accuray of each annotator with ground truth data and then use that accuracy as your confidence when aggregating multiple annotations for a task. This chapter and chapter 9 show many examples of combining agreement and ground truth data, depending on the problem you are solving, but they are introduced independently to explain the concepts in isolation.

8.2.3 Dataset-level agreement with Krippendorff's alpha

Krippendorff's alpha is a method that aims to answer a simple question: what is the overall agreement in my dataset? To account for the fact that not every item will be annotated by every annotator, Krippendorff's alpha made considerable advances on existing agreement algorithms that were popular in social sciences when used for tasks such as measuring the level of agreement in surveys and census data.

The simple interpretation of Krippendorff's alpha is that it is a [–1,1] range, which can be read as follows:

- *>0.8*—This range is reliable. If you apply Krippendorff's alpha to your data, and you get a result of 0.8 or higher, you have high agreement and a dataset that you can use to train your model.
- *0.67–0.8*—This range has low reliability. It is likely that some of the labels are highly consistent and others are not.

- *0–0.67*—At less than 0.67, your dataset is considered to have low reliability. Something is probably wrong with your task design or with the annotators.
- *0*—Random distribution.
- *–1*—Perfect disagreement.

Krippendorff's alpha also has the nice property that it can be used for categorical, ordinal, hierarchical, and continuous data. Most of the time in practice, you can use Krippendorff's alpha without knowing how the algorithm works and interpret the output according to the 0.8 and 0.67 thresholds. But in order to understand what is happening under the hood and when it might not be appropriate, it is a good idea to get an intuition for the mathematics. Don't worry if you don't get all the steps on the first go. When I re-derived all the equations in this book, it took me longer to derive Krippendorff's alpha than any of the active learning or machine learning algorithms.

Krippendorff's alpha aims to calculate the same metric as the simple example in figure 8.7 earlier in this chapter: what is our actual agreement relative to our expected agreement? We'll start with a partial implementation of Krippendorff's alpha that works for mutually exclusive labels and then move to a more general version.

The expected agreement for Krippendorff's alpha is the data frequency: the sum of the squares of the frequency of each label for a labeling task. The actual agreement for Krippendorff's alpha comes from the average amount that each annotation agrees with the other annotations for the same task. Krippendorff's alpha makes a slight adjustment to the average, epsilon, to account for the loss of precision given the finite number of annotations.

Krippendorff's alpha is the adjusted agreement of the expected and actual agreement from figure 8.6. We can see Krippendorff's alpha from our example data using a simplified representation in figure 8.8.

The agreement in figure 8.8 is much lower than our "naive agreement" in figure 8.7 (0.803 compared with 0.921), so it shows that we need to be careful in how we calculate agreement and that small changes in our assumptions can result in large differences in our quality control metrics.

Figure 8.8 is a partial implementation of Krippendorff's alpha. The full equation takes into account the fact that you might weight some types of disagreements more severely than others. The full implementation of Krippendorff's alpha is shown in figure 8.9.

Although figure 8.9 shows some complicated processes, the main difference between it and figure 8.8 come from how Krippendorff's alpha incorporates the label weights. The label-weights component allows Krippendorff's alpha to be adapted to different types of problems, such as continuous, ordinal, or other tasks in which multiple labels can be applied to one item.

For more details, look at the implementations in the spreadsheet introduced in section 8.2.1. You can see that the expected agreement and actual agreement need to take in some matrix operations to incorporate the weights in the full Krippendorff's alpha implementation, compared with the partial implementation. Also, the epsilon

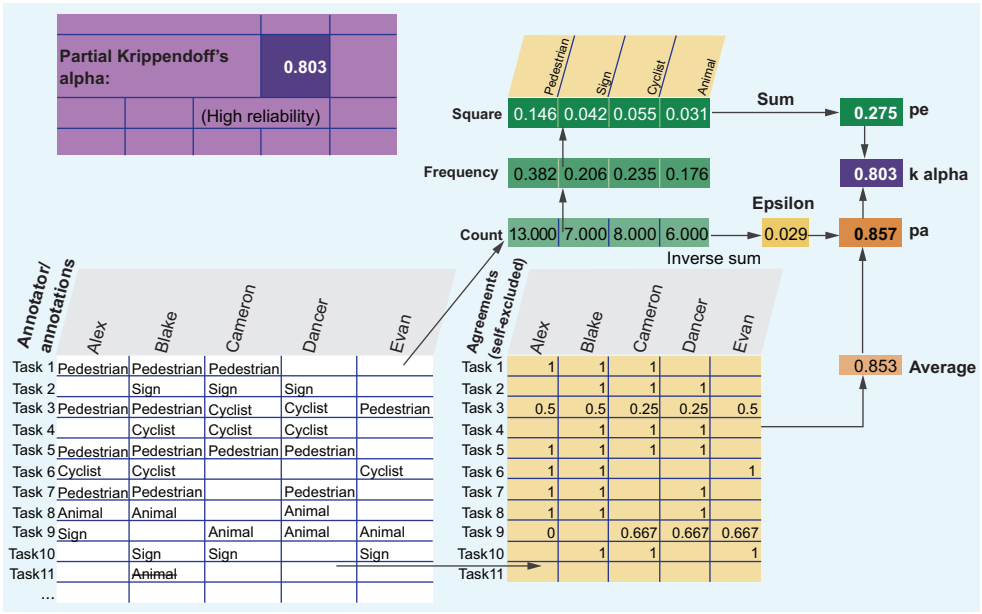


Figure 8.8 **A simplified Krippendorff's alpha that provides an overall score for the reliability of the annotators for our example data. The expected agreement is the sum of squares of the frequency of each label. The actual agreement is the average amount by which each annotation agreed with the other annotations for that task, with a small adjustment (epsilon) made to account for precision in our calculations.**

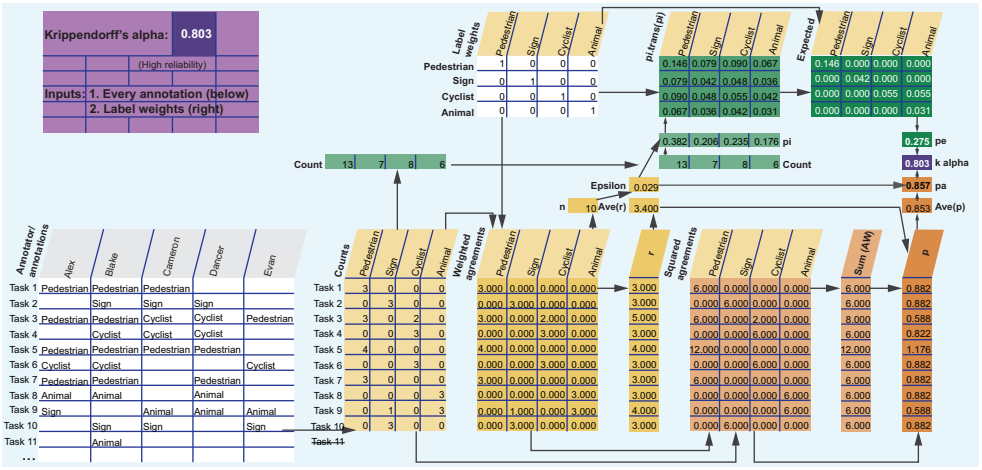


Figure 8.9 **Krippendorff's alpha, calculating the overall level of agreement in a dataset to determine whether it is reliable enough to use for training data. The inputs are the white areas: the annotations (bottom left) and the label weights (top middle). Because we have mutually exclusive labels, this example has each label weighted only with itself. If we had hierarchical, ordinal, or other types of data, we would enter different values as label weights. The top row of calculations contains the expected agreement by random chance, and the bottom row of calculations calculates the actual agreement in the data. The two rows are used to calculate the expected agreement (pe) and actual agreement (pa) for the dataset, from which the adjusted overall agreement alpha is calculated.**

adjustment takes the weights into account and is not simply the inverse of the total count. The general idea behind the simple and full implementations is the same, however: we are calculating an adjusted agreement according to the actual and expected agreements. If you keep that concept in mind and appreciate the fact that all the extra steps in the full implementation of Krippendorff's alpha come from the flexibility needed for different types of annotations, you have the right idea about how to apply it.

When do I need to calculate confidence intervals for Krippendorff's alpha?

This book omits the extensions to Krippendorff's alpha for calculating the confidence intervals because the confidence intervals anticipate the kind of smaller surveys that Krippendorff's alpha was designed for. Most of the time, you won't need confidence intervals for training data because the biggest factor for confidence intervals will be the total number of judgments. Because your training data will likely contain thousands or even millions of examples, the confidence intervals will be tiny.

You need to worry about confidence intervals only if you are going to use Krippendorff's alpha on a small dataset or a small subset of your dataset. Note that if you are using a small amount of data because of a cutting-edge, lightly supervised, few-shot, or data-augmentation technique, you'll need better statistical knowledge to help ensure significance for your smaller datasets. You may have assumed that less data makes the required supporting infrastructure easier to build, but the opposite is true.

Even in these edge cases, I don't recommend relying on confidence intervals alone. If you have a small number of training examples, you should include other types of quality control, including review tasks for experts and incorporating known ground truth examples. Otherwise, your confidence intervals will be so wide that it will be difficult to trust a model built on the data.

Alternatives to Krippendorff's alpha

You may encounter alternatives to Krippendorff's alpha in the literature, such as Cohen's kappa and Fleiss's kappa. Krippendorff's alpha is generally seen as being a refinement of those earlier metrics. The differences are details such as whether all errors should be punished equally, the correct way to calculate the expected prior, the treatment of missing values, and how to aggregate the overall agreement (aggregating per annotation, like Krippendorff's alpha, or per task/annotator, like Cohen's kappa). The additional reading in section 8.6 has some examples.

You may also encounter Krippendorff's alpha expressed in terms of disagreement, instead of agreement, including in Krippendorff's own publications. The techniques are mathematically equivalent and produce the same alpha value. Agreement is more widely used than disagreement in other metrics and is arguably more intuitive, which is why agreement is used here. Assume that disagreement is the complement of agreement: $D = (1 - P)$. Keep this assumption in mind when you look at the literature and libraries, which may have versions of Krippendorff's alpha that are calculated by using disagreement.

8.2.4 Calculating Krippendorff's alpha beyond labeling

Here are some examples of how Krippendorff's alpha can be used for tasks that are more complicated than mutually exclusive labeling tasks. Figure 8.10 shows how we can change the label weights in the Krippendorff's alpha equation to capture ordinal and rotational data.

Labeling (mutually exclusive)

Label Weights	Pedestrian	Sign	Cyclist	Animal
Pedestrian	1	0	0	0
Sign	0	1	0	0
Cyclist	0	0	1	0
Animal	0	0	0	1

Ordinal categories

Label Weights	Excellent	Good	Neutral	Bad
Excellent	1	0.5	0.25	0
Good	0.5	1	0.5	0.25
Neutral	0.25	0.5	1	0.5
Bad	0	0.25	0.5	1

Rotational categories

Label Weights	North	East	South	West
North	1	0.5	0	0.5
East	0.5	1	0.5	0
South	0	0.5	1	0.5
West	0.5	0	0.5	1

Figure 8.10 An example of three types of classification tasks and how the label weights from Krippendorff's alpha can be used for those tasks. The first example repeats the label weights from figure 8.9, showing the mutually exclusive labeling tasks that have been used as an example throughout this chapter. The second example shows an ordinal scale from "Bad" to "Excellent," where we want to give partial credit to adjacent annotations such as "Good" and "Excellent." The third example shows rotational categories—in this case, the compass points. In this case, we give a partial score to anything that is off by 90 degrees, such as "North" and "West," but a zero score to anything that is off by 180 degrees, such as "North" and "South."

The rest of this chapter sticks to mutually exclusive labeling. We'll cover to other types of machine learning problems in chapter 9.

Krippendorff's alpha has some shortcomings when it's used for training data, because it was originally derived for use cases such as when a school is randomly distributing exam papers across multiple graders (annotators). It doesn't capture the fact that some annotators will have a different expected agreement based on what they have seen. When creating training data, we have many good reasons to distribute annotations nonrandomly, such as giving a hard example to additional people to adjudicate. Sections 8.2.5 through 8.2.7 differ from Krippendorff's alpha in key ways for calculating agreement at annotator, label, and task levels.

8.2.5 Individual annotator agreement

Agreement at individual annotator level can be useful in multiple ways. For one thing, it can tell you how reliable each annotator is. You can calculate agreement at the macro level, calculating an annotator's reliability across every response they made, or you may want to see whether they have higher or lower agreement for certain labels or segments of the data. This result might tell you that the annotator is more or less accurate or may highlight a diverse set of valid annotations.

The simplest metric for agreement between annotators is to calculate how often each annotator agrees with the majority of people for a given task. Figure 8.11 shows an example.

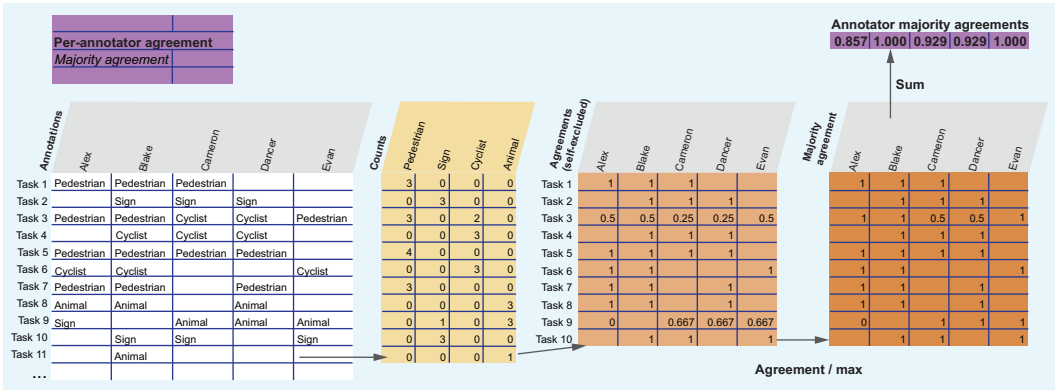


Figure 8.11 The per-annotator agreement with the most common annotation for each task (majority agreement). This example shows that two annotators, Blake and Evan, always agreed with the majority. This method is the simplest way to calculate agreement between annotators; it can be effective when you have a large number of annotators per task but is rarely used for creating training data due to budget constraints. This method can provide insight into your data but should not be your sole means of determining data quality.

Majority agreement, as shown in figure 8.11, looks at the number of times a person agrees with the most commonly annotated label for each task. This result can also be calculated as a count of the fraction of times that a person agrees with the majority, but it is a little more accurate when normalized for agreement on a per-annotation basis. In figure 8.11 and other example data in this chapter, Cameron and Dancer agree that task 3 is "Cyclist," even though most people think task 3 is "Pedestrian." By contrast, Alex is the only one who thinks that task 9 is "Sign." So in our Majority Agreement table in figure 8.11, Cameron and Dancer get 0.5 for task 3, and Alex gets 0 for task 9.

Majority agreement can provide a good quick check on whether your annotators have seen easier or harder examples. In the naive agreement example in figure 8.6 earlier in this chapter, Evan has the next-to-lowest agreement (0.836), but they have the equal highest agreement in figure 8.11 (1.0). In other words, Evan had low agreement on average with other people but always agreed with the majority. This result tells you that Evan saw tasks with lower overall agreement than other people. A good agreement metric, therefore, should take into account the fact that Evan saw harder tasks.

Expected agreement is the biggest piece missing from figure 8.11. Figure 8.12 shows one way to calculate expected agreement, which shows that Evan has the lowest expected agreement if they always chose "Pedestrian."

The first thing to notice in figure 8.12 is that we are using the most frequent label (mode label) to calculate our baseline. Recall that Krippendorff's alpha uses the same number of labels in the data, as though they were assigned randomly. In our example,

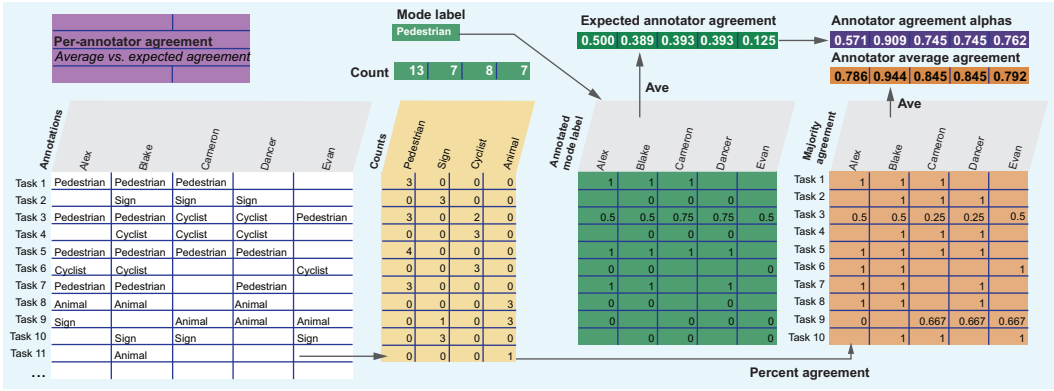


Figure 8.12 The per-annotator agreement is calculated in terms of the actual agreements (bottom right), with the expected agreement calculated on a per-annotator basis (top middle). Note that Evan has an expected agreement of only 0.15. In other words, if Evan guessed the most common label, "Pedestrian," every time, they would agree with about 15% of the other annotations on their tasks. By contrast, Alex could have guessed "Pedestrian" every time and got about 51% agreement. This method takes into account the fact that Evan saw tasks with lower agreement that were presumably more difficult.

someone might randomly assign 13 "Pedestrian" labels, 7 "Sign" labels, and so on. While this example is the (statistical) definition of an expected distribution, it is unlikely that a human annotator will have that probability for each label in mind when annotating. The more likely scenario is that an annotator will have an intuition about the most frequent label (mode label). This result is common in data labeling. Often, one label is obviously more frequent than all the others and feels like a safe default option. There are ways to mitigate the problem of bad labels because a person feels pressured to label a default option when they are uncertain, which we'll cover in chapter 9. Here, we'll treat this most common label as our expected baseline.

The second difference between figure 8.12 and standard Krippendorff's alpha calculation is that figure 8.12 calculates agreement per task, whereas Krippendorff's alpha calculates agreement per annotation. If you have the same number of annotations per task, the numbers would be identical. In our example data, task 3 has five annotations, so it effectively has a larger weight than the other tasks in Krippendorff's alpha. Krippendorff's alpha, however, gives task 3 the same weight as every other task when calculating individual agreement.

You don't want to give different weights to different tasks for data annotation for many reasons. You might deliberately give the same task to more annotators to resolve disagreements, for example, or you might give easier tasks to fewer people based on the label or external information. In both cases, Krippendorff's alpha would be biased toward the more difficult tasks, giving you an artificially low score. If you truly have a random distribution of annotators across tasks, and it is arbitrary that some tasks end up with more annotations, the standard Krippendorff's alpha approach is fine.

Don't p-hack Krippendorff's alpha by iteratively removing annotators with the lowest agreement

Often, you want to ignore the annotations made by your least-accurate annotators. You can improve the overall agreement and accuracy of your training data by removing the worst performers and giving their tasks to other annotators.

You would make a mistake, however, if you iteratively removed the worst performers until your dataset reaches the magic k-alpha=0.8 number, which indicates high agreement. Using the threshold of significance itself as a threshold for removing people is what Regina Nuzzo called *p-hacking* in *Nature* in 2014 (http://mng.bz/8NZP).

Instead of relying on Krippendorff's alpha, you should remove people by one of the following criteria, in order of preference:

- *Use a different criterion from Krippendorff's alpha to decide who is a good or bad performer.* Ideally, you should use the annotator's agreement with known ground truth answers. Then you can use that criterion to remove the worst performers. You can set a threshold level of accuracy on the known answers or decide that you will remove some percentage of annotators (such as the worst 5%). You should make the decision about a threshold or percentage without taking Krippendorff's alpha into account.
- *Remove low performers who are statistical outliers in terms of how badly they performed.* Use this technique if you are confident about your mathematical skills. If you can calculate that all the agreement scores fall in a normal distribution, for example, you can remove any annotator with agreement that is three standard deviations below the average agreement. If you are not confident in your ability to identify the type of distribution and the appropriate outlier metric, stick to the first option and create additional questions with known answers if necessary.
- *Decide in advance what your expected percent of low-performing annotators will be, and remove only those annotators.* If you typically find that 5% perform poorly, remove the bottom 5%, but do not keep going if you are not yet at your target agreement. This approach could contain a little bias, because you are still using Krippendorff's alpha to calculate the lowest 5%. The bias is probably minor, however, and in any case, you shouldn't use this approach if you can use the first two options.

What happens if you p-hack Krippendorff's alpha? You may get bad instructions or an impossible task, but you will never learn that result. You may end up removing everyone except annotators who happened to be sitting next to one another and sharing notes.

If you have established that an annotator is not reliable enough to trust, you should remove that annotator's judgments from your calculation of agreement. Figure 8.13 shows this result with our example data, assuming that we removed the first person.

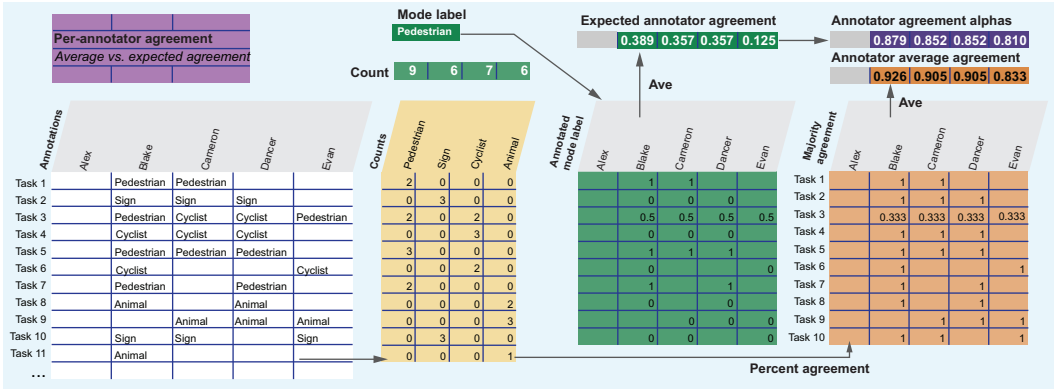


Figure 8.13 Recalculating agreement for our annotators after the first annotator has been removed. Note that three of the four scores went up compared with figure 8.12, but Blake's agreement dropped slightly, and Evan went from second-highest to lowest agreement.

As figure 8.13 shows when compared with figure 8.12, you generally expect overall agreement to go up when the least-accurate person is removed, but some individual agreement scores may still go down (as in the case of Blake), and the rankings may change considerably, as is the case with Evan. Evan has the highest agreement when we calculate agreement with the majority in figure 8.11 but has the lowest agreement when we calculate for chance-adjusted agreement after Alex is removed in figure 8.13. This figure is a good example of why you need to be careful about using agreement as the only way to calculate accuracy: your choices can produce different results for individuals.

8.2.6 Per-label and per-demographic agreement

Ideally, you have some ground truth labels for your dataset, so you can use these labels to plot the errors in a confusion matrix. This confusion matrix is identical to the kind that you use for machine learning models, except that it is the pattern of human errors in place of model errors.

You can also use a confusion matrix for agreement, plotting which annotations occur with others. Figure 8.14 shows the matrices for our example data.

Predicted / Actual	Pedestrian	Sign	Cyclist	Animal
Pedestrian	10	0	0	0
Sign	0	6	0	0
Cyclist	3	0	8	0
Animal	0	1	0	7

Counts	Pedestrian	Sign	Cyclist	Animal
Pedestrian	30	0	6	0
Sign	0	12	0	3
Cyclist	6	0	14	0
Animal	0	3	0	12

Figure 8.14 Annotation confusion matrices: compared with ground truth data in our example data (top) and compared with every pairwise agreement or disagreement (bottom)

This second type of confusion matrix doesn't tell you what the errors are—only where the agreement or disagreement occurs. With either type of matrix, you can see where the greatest pairwise confusion occurs in your annotations, and this information should help you refine your instructions for annotators, as well as indicate which labels may be hardest for your model to predict.

8.2.7 Extending accuracy with agreement for real-world diversity

It can be especially useful to use agreement as an extension of accuracy when you want to track a large number of fine-grained demographics. If you want to track the intersection of demographics, you may have too many combinations of demographic categories for which you are able to collect enough ground truth data.

Consider the example where we suspected that images taken at night are more difficult to annotate than images taken during the day. Now suppose that you also want to track the accuracy of annotation across 1,000 locations. You are unlikely to have a large volume of ground truth labels for every one of these 24,000 time/place combinations, because it would be expensive to create so much ground truth data.

Therefore, looking at agreement for each of the 24,000 time/place combinations is your best window into the difficulty of each demographic intersection. There won't always be a perfect correlation between agreement and accuracy, but this approach can reveal some areas of high agreement that you can review and potentially target for more ground truth data.

8.3 Aggregating multiple annotations to create training data

Task-level confidence is the most important quality control metric for many annotation projects, because it allows us to aggregate the (potentially conflicting) annotations of each annotator and create the label that will become the training and evaluation data.

Therefore, it is important to understand how to combine multiple annotations to create the single label that will become the actual label. Aggregating multiple annotations in a task builds on the other types of quality control metrics that you have seen in this chapter: we want to take our confidence in each annotator into account when calculating the overall agreement for a given task, and ideally, we want to know whether this particular task is inherently easier or more difficult.

8.3.1 Aggregating annotations when everyone agrees

It can be easiest to think about agreement in terms of the chance of error instead of the chance of being correct. Suppose that we have three annotators and they are each 90% accurate. The chance that any one annotator makes an error is 10%. The chance that a second annotator made an error on the same task is 10%, so combined, there is a 1% ($0.1 \times 0.1 = 0.01$) chance that two people made an error on the same item. With three annotators, that chance becomes a 0.1% chance ($0.1 \times 0.1 \times 0.1$). In other words, there is a 1-in-1,000 chance of being incorrect and a 0.999 chance of being correct. If three annotators are 90% accurate, and all three agree, we can assume with

99.9% confidence that the label is correct. Letting the accuracy of the ith annotator be a_i, the overall confidence that the label is correct is

$$1 - \prod_{i=1} (1 - a_i)$$

Unfortunately, this method has limitations because it assumes that the errors are independent. If the first annotator makes an error, does the second annotator still have only a 10% chance of error, or do the errors tend to cluster or diverge?

It is easy to imagine scenarios in which the patterns of errors are nonrandom. Most obviously, some tasks tend to be harder than others. If 10% of all tasks lead to people choosing the wrong label, perhaps that task is where all three annotators made mistakes. If you have a task with a large number of labels, this problem is less common, because people are less likely to choose the same wrong label. You often want to reduce your tasks to as few annotations as possible to make them more efficient, so there is a trade-off between accuracy and cost.

The ground truth data allows you to calculate the following: for each incorrect annotation, what % of annotations for the task are also incorrect? Let's work through an example. Assume that in our example data, every item's actual label is the one shown in figure 8.3 earlier in the chapter. The following table shows two tasks, 3 and 9, with the errors in bold:

Task 3	**Pedestrian**	**Pedestrian**	Cyclist	Cyclist	**Pedestrian**
Task 9	**Sign**		Animal	Animal	Animal

In task 3, each of the three wrong "Pedestrian" annotations agrees with the two other "Pedestrian" annotations, giving us six total agreements for the incorrect label. Note this number is in the column sum(AW) from Krippendorff's alpha. In task 9, the "Sign" error was alone, so there are no agreeing errors. For the correct answers, we have two agreements in task 3 (the two "Cyclists" annotations agreeing with each other) and each of three "Animals" annotations agreeing with each other. So in total, there are eight cases of annotators agreeing with one another when they are correct and six cases of annotators agreeing with one another when they are incorrect. To calculate how often incorrect annotations agree, we calculate

Correlation of errors = 6 / (8 + 6) = 0.429

Therefore, although our overall error rate is 10%, the likelihood that errors in annotation will co-occur is 42.9%—more than four times higher! After the first error, we should assume that errors co-occur at this rate. With agreement from three annotators, the overall confidence in our label would be

$1 - (0.1 \times 0.429 \times 0.429) = 0.982$

So instead of having 99.9% confidence, we are have 98.2% confidence in our label when all three annotators agree, going from an error in every 1,000 items to an error in about every 55 items.

The opposite pattern can also occur, in which the pattern of errors diverges. Let's assume that the three annotators are still 90% accurate individually, but they make different errors. One annotator makes most of their errors identifying "Sign," whereas another annotator might make most of their errors identifying "Animal." They might make errors on different images, so the chance that the errors will co-occur is 2%:

$$1 - (0.1 \times 0.02 \times 0.02) = 0.99996$$

In this case, where your annotators have complementary skills, you can be 99.996% confident that agreement between annotators means that your annotation is correct, and so an error occurs once in every 25,000 items.

8.3.2 The mathematical case for diverse annotators and low agreement

There is a big difference in how errors pattern across annotators, as the example in section 8.3.1 showed. We can expand on this example as the mathematical proof that having a diverse set of annotators will result in more accurate data.

Given the same overall error rate on a per-annotation basis, the data with the highest accuracy will have the lowest agreement, because the errors are spread out and create more opportunities for disagreement. Therefore, this condition has the lowest Krippendorff's alpha score, showing why we don't want to rely on Krippendorff's alpha score alone because it can penalize diversity unfairly. You can see this result in our example data with a Krippendorff's alpha score of 0.803. If we spread out the disagreements so that there is no more than one disagreement per task, however, we get a Krippendorff's alpha score of 0.685. So even though our data has the same frequency for each label and the majority is much more reliable, our dataset looks less reliable.

It is easy to imagine scenarios in which the agreement is clustered: some examples are harder than others or annotators have subjective but similar judgments. It is also easy to imagine scenarios in which agreement is divergent: annotators are diverse and bring different but legitimate perspectives to the data.

It is difficult, however, to imagine real-world scenarios in which annotators are making errors completely independently (except perhaps because of fatigue). Yet almost all agreement metrics make the assumption of independence, which is why they should be used with caution. As this section and section 8.3.1 show, our ground truth data allows us to calibrate to the correct numbers for a given dataset. The advanced methods in chapter 9 go into more detail on data-driven agreement metrics.

8.3.3 Aggregating annotations when annotators disagree

When annotators disagree, you are essentially converging a probability distribution across all the potential labels. Let's expand our example from task 3 and assume that everyone is 90% accurate on average (figure 8.15).

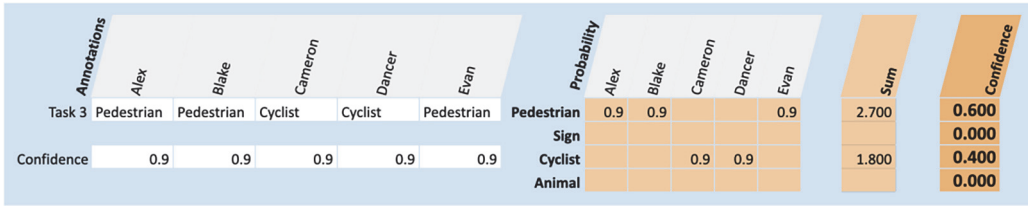


Figure 8.15 Using per-annotator accuracy as probabilities for per-task agreement

In figure 8.15, we have three annotators who labeled the image in this task as "Pedestrian" and two who labeled it "Cyclist." The simplest way to calculate confidence when not all annotators agree is to treat the confidences as a weighted vote. Let's assume that we're calculating confidence for task 3 and that we have 90% confidence in every annotator:

Pedestrian = 3 * 0.9 = 2.7
Cyclist = 2 * 0.9 = 1.8
Confidence in Pedestrian = 2.7 / (2.7 + 1.8) = 0.6
Confidence in Cyclist = 1.8 / (2.7 + 1.8) = 0.4

Another way to think of this calculation is that because we're equally confident in everyone in this example, three-fifths of the annotators agree, so we are 3/5 = 60% confident.

One problem with this method is that it does not leave any confidence for other labels. Recall that when we had perfect agreement, there was still a small chance that it was wrong and that therefore, the correct label was one not annotated by anyone. We can incorporate the possibility that a non-annotated label might be correct by treating the confidence as a probability distribution and assume that all other labels get the weight divided among them, as shown in figure 8.16.

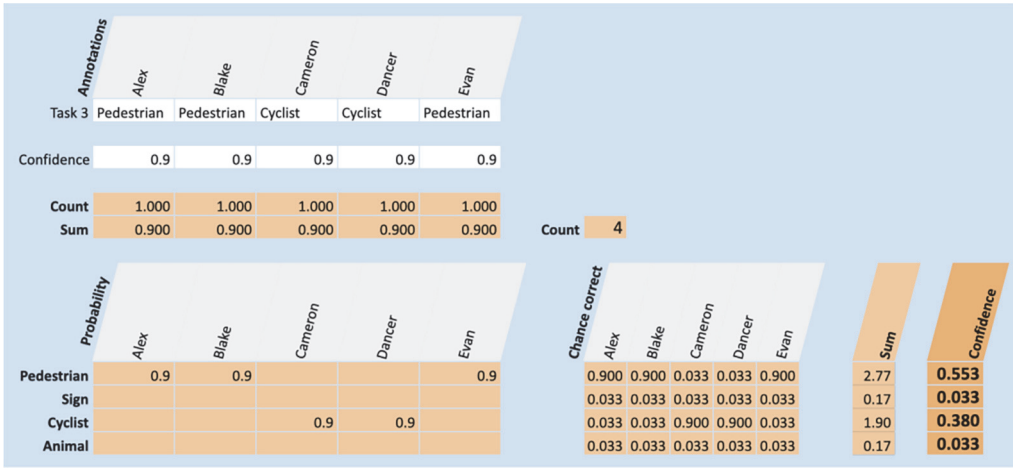


Figure 8.16 Expanding all the confidences in annotators to give some weight to all labels. We have 0.9 confidence for each annotator, so we distribute the remaining 0.1 across the other labels.

This example gives a conservative estimate to confidence, with a large amount of weight going to unseen answers. Note also this method is not the one we used when there was perfect agreement. There are several ways to get a more accurate probability distribution for the annotations, most of which involve a regression or machine learning model because they can't be computed with simple heuristics like those used here. Chapter 9 covers these advanced approaches. This example is enough to build on for the remainder of this chapter.

8.3.4 Annotator-reported confidences

Annotators often have good intuitions about their own errors and which tasks are inherently harder than others. As part of the annotation process, you can ask when annotators are less than 100% confident on a certain task. An example with our data might look like figure 8.17.

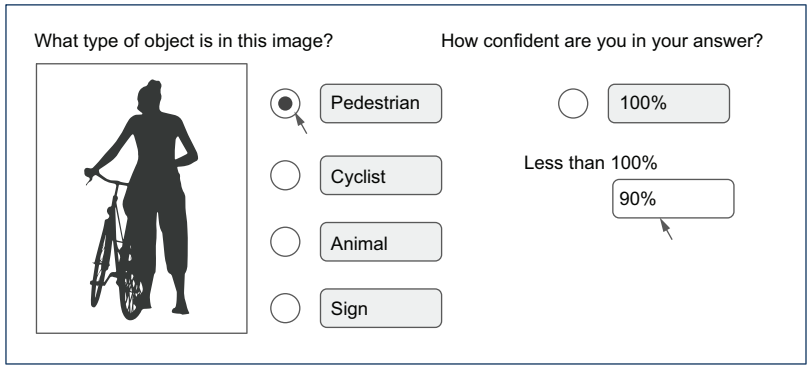


Figure 8.17 Requesting confidence explicitly from the annotator is an alternative (or addition) to calculating the confidence in their response from their accuracy and/or agreement.

You could also request the entire probability distribution, as shown in figure 8.18.

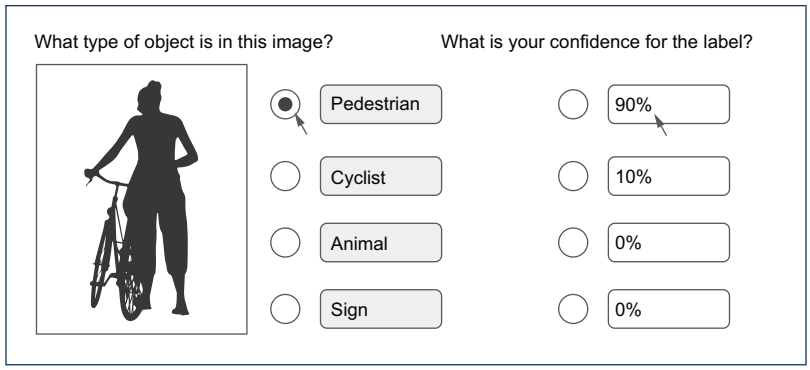


Figure 8.18 Requesting the annotator confidence for every label as an alternative to dividing their remaining confidence across the other labels programmatically

With the approach shown in figure 8.18, you can treat the entered amount as the probability for this label for this annotator, or you might choose to ignore all annotations when the annotator is less than 100% confident. This kind of interface can be extended to ask annotators how other annotators might respond to the question, which has some nice statistical outcomes that help with accuracy and diversity, especially for subjective tasks. These extensions are covered in chapter 9.

Entering this information can greatly increase the annotation time for a simple labeling task like our example, so you will have to weigh the cost of capturing this information against the value that it adds.

8.3.5 Deciding which labels to trust: Annotation uncertainty

When you have the probability distribution for your labels for a given task, you need to set a threshold for when not to trust a label and decide what to do if you don't trust the label. You have three options when you don't trust the label:

- Assign the task to an additional annotator, and recalculate the confidence to see whether the confidence is high enough.
- Assign the task to an expert annotator to adjudicate on the correct label (more on this topic in section 8.4).
- Exclude this item from the dataset so that a potential error won't produce errors in the model.

Generally, you want to avoid the third scenario because you are wasting the effort put into that task. You are also risking introducing bias into your data because the harder tasks are unlikely to be random. Budget or staffing constraints might prevent you from giving the same task to many people, however.

Before you can make a decision about whether you trust your label, you need to work out how to calculate overall confidence in your label. Let's assume that our probability distribution is taken from the example we have been using in this chapter:

Pedestrian = 0.553
Sign = 0.033
Cyclist = 0.380
Animal = 0.033

We have different ways to calculate our overall confidence uncertainty: look only at the 0.553 confidence for "Pedestrian," take into account the next-most-confident label ("Cyclist"), or take all the potential labels into account.

If you recall from chapter 3, this scenario is the same one that we had with uncertainty sampling for active learning. You have different ways to measure your uncertainty for annotation agreement, and each method makes a different assumption about what you care about. Using PyTorch, the example can be expressed as a tensor:

```
prob = torch.tensor([0.533, 0.033, 0.380, 0.033])
```

Reproducing the equations from chapter 3, we can calculate different uncertainty scores, as shown in figure 8.19.

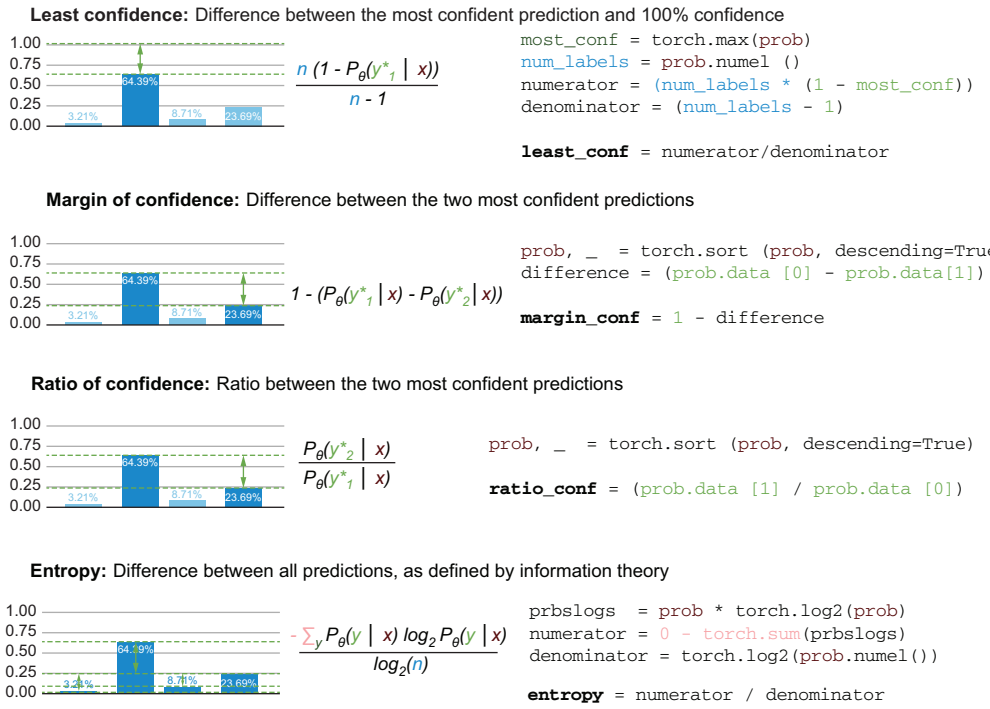


Figure 8.19 Different methods of calculating an uncertainty score for a probability distribution. These methods are the same one that are used in active learning to calculate uncertainty (or confidence) from a model's prediction, used here to calculate uncertainty for agreement among annotators.

For our example, we get these uncertainty scores (remember that 1.0 is the most uncertain):

- Least confidence = 0.6227
- Margin of confidence = 0.8470
- Ratio of confidence = 0.7129
- Entropy = 0.6696

To get our overall confidence, instead of uncertainty, we subtract one of these metrics from 1.

After you have your uncertainty scores, you can plot your overall annotation accuracy at different scores on your ground truth data. Then you can use this plot to calculate the accuracy threshold that will give you the desired accuracy for your data (figure 8.20).

You plot a curve like the one shown in figure 8.20 for each of the uncertainty metrics as one way to decide which is the best for your data: which uncertainty sampling method selects the most items at the right threshold?

Figure 8.20 Calculating the threshold at which you can trust your annotations. In this example, the desired annotation accuracy of ~0.96, calculated on ground truth data, will be achieved if items with agreement uncertainty below ~0.65 are trusted.

The rank order of the different uncertainty scores is identical for binary data, so if you have broken your task into binary problems, you can choose any one of these metrics and not worry about deciding which is best for your data.

As an alternative to calculating the threshold on your ground truth data, as in figure 8.20, you can find the best threshold for your machine learning model's accuracy when trained on data at different thresholds. Try different thresholds for which items to ignore and then observe the downstream accuracy of your model with each threshold. Your model's sensitivity to errors in the training data will probably change with the total number of training items, so you may want to keep revisiting past training data and reevaluate the threshold with each new addition to the training data.

8.4 Quality control by expert review

One of the most common methods of quality control is to engage SMEs to label the most important data points. Generally, experts are rarer and/or more expensive than other workers, so you typically give some tasks only to experts, often for one of these reasons:

- To annotate a subset of items to become ground truth examples for guidelines and quality control
- To adjudicate examples that have low agreement among nonexpert annotators
- To annotate a subset of items to become machine learning evaluation items, for which human label accuracy is more important
- To annotate items that are known to be important for external reasons. If you are annotating data from your customers, for example, you may want expert annotators to focus on examples from the customers who generate the most revenue for you

Figure 8.21 copies a figure from chapter 7 about using experts for review. It illustrates the first two examples in the preceding list: creating ground truth examples for guidelines and quality control, and adjudicating examples that have low agreement (confusing items).

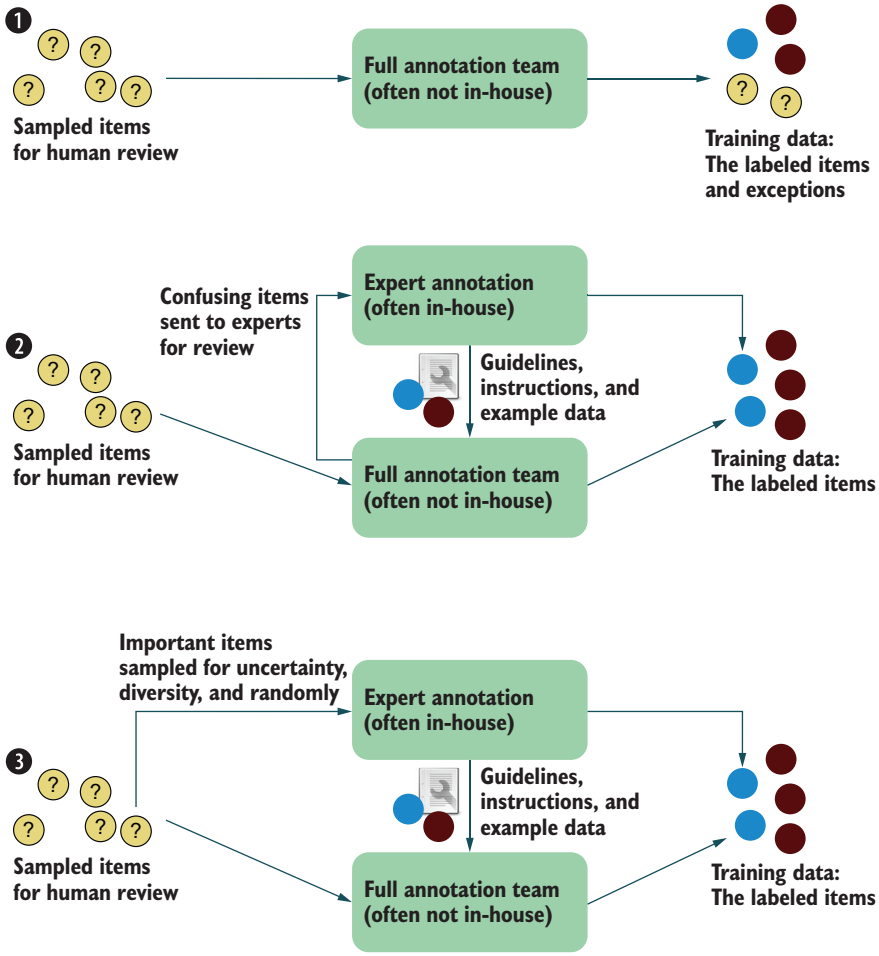


Figure 8.21 Three workflows for expert in-house annotation, repeated from chapter 7. The bottom two workflows show different ways that experts might be incorporated: adjudicating items that were difficult for annotators and creating guidelines for annotators. Both workflows might exist in the same task, and there might be many more steps for more complicated workflows.

To aggregate annotations after expert review, you can treat that expert as being one additional annotator, or you can ignore the previous annotations and calculate confidence in terms of the confidence in the expert(s). Choose the latter option if you know that your experts are much more reliable than most of your workforce.

8.4.1 Recruiting and training qualified people

As we discussed in chapter 7, it is common to have SMEs in-house, but you can often outsource this expertise. An annotator who has been working on annotation for autonomous vehicles for several years, for example, is highly skilled. See chapter 7 for more information about choosing the right workforce for your tasks, including experts.

8.4.2 Training people to become experts

You can take a data-driven approach to identifying experts within your nonexpert annotator pool. Keeping track of individual annotator accuracy, not overall dataset accuracy alone, will allow you to discover experts and promote them to that role.

As a stepping stone to making some annotators expert adjudicators, you might allow those annotators to review but not adjudicate the work of others. This approach will let those people get intuition about the common errors that people are making.

You should track the demographics of your experts, as you track the demographics of your annotators, to ensure diversity (except when tracking violates their privacy). An annotator's age, country of residence, education level, gender, language fluency, and many other factors may be important for a task. If you don't track the demographics of your annotators and use agreement as one metric for determining the best annotators, you run the risk of taking biases from your annotator pool into your expert annotator pool. For this reason, you should ideally identify experts from representative data, not a random sample.

8.4.3 Machine-learning-assisted experts

A common use case for SMEs is for their daily tasks to be augmented by machine learning. If you recall from chapter 1, human-in-the-loop machine learning can have two distinct goals: making a machine learning application more accurate with human input, and improving a human task with the aid of machine learning.

Search engines are a great example. You may be a domain expert in some scientific field searching for a particular research paper. The search engine helps you find this paper after you type the right search terms and learns from what you clicked to become more accurate.

Another common use case is e-discovery. Like search, but often with a more sophisticated interface, e-discovery is used in contexts such as audits where expert analysts are trying to find certain information in a large amount of text. Suppose the audit was for a legal case to detect fraud. An expert analyst in fraud detection might use a tool to find relevant documents and communications for that legal case, and that tool might adapt to what the analyst has found, surfacing all the similar documents and communications that have been tagged as relevant in the case so far. E-discovery was a $10 billion industry in 2020. Although you may not have heard about it in machine learning circles, it is one of the single largest use cases for machine learning.

You can deploy the same quality control measures in these cases: look for agreement between experts, employ adjudication by higher-level experts, evaluate against known answers, and so on. The expert, however, will likely be using an interface that supports their day-to-day tasks, not the annotation process itself. The interface may not be optimized for collecting training data, and their work process might introduce ordering effects that you can't control. So the user interface implications for quality control in chapter 11 will be important in these contexts.

8.5 Multistep workflows and review tasks

One of the most effective ways to get higher-quality labels is to break a complicated task into smaller subtasks. You can get several benefits from breaking your task into simpler subtasks:

- People generally work faster and more accurately on simpler tasks.
- It is easier to perform quality control on simpler tasks.
- You can engage different workforces for different subtasks.

The main downside is the overhead of managing the more complicated workflows. You will end up with a lot of custom code to route data based on certain conditions, and that code may not be reusable for other work. I have never seen an annotation platform that solved these problems with plug-and-play or drop-down options: there are almost always complicated combinations of conditions that require coding or a coding-like environment to be implemented fully.

Figure 8.22 shows how we might break an object labeling task into multiple steps, the last one being a review task for the preceding step.

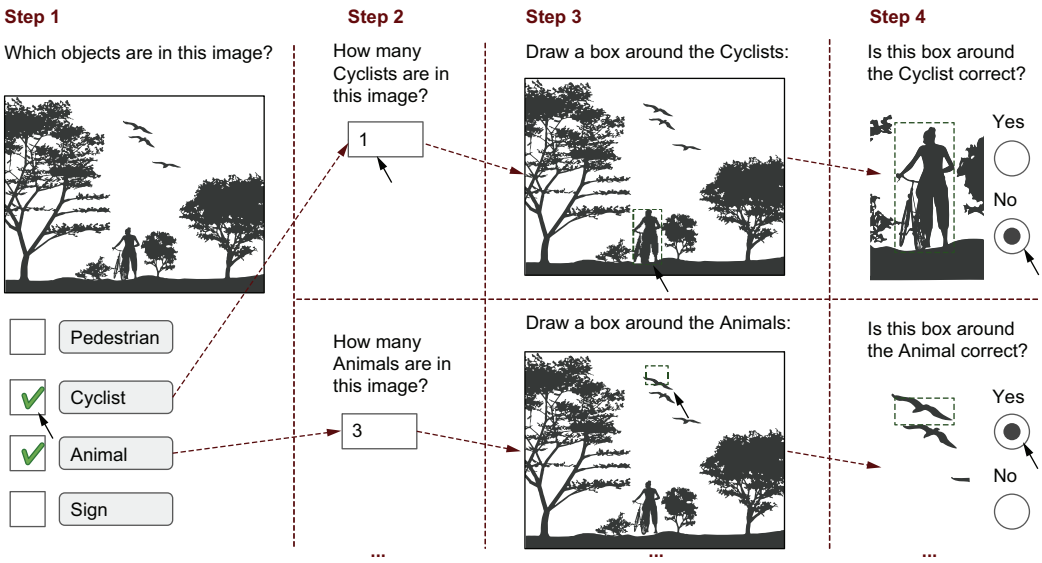


Figure 8.22 An example of a multistep workflow. If we divide steps 2–4 among the four object types, we have 13 total tasks. The individual responses in step 1 and the evaluation in step 4 are binary tasks. Therefore, although our goal is to create a bounding box that requires the advanced quality control metrics from chapter 9, we can use the simpler label-based quality control metrics for this chapter. Compared with a single task that captures every bounding box in one go, we can expect higher throughput and accuracy because annotators are concentrating on one task at a time; easier budgeting if we're paying per task, because there will be less variability in the time needed per task; and easier division of tasks among workforces if only some annotators are trusted for the most complicated tasks.

The most complicated workflow I have seen had about 40 tasks. This workflow, for a computer vision task for autonomous vehicles, had several steps for each type of object that was being tracked in addition to semantic segmentation.

Simpler tasks have some user experience trade-offs. Generally, people appreciate the efficiency, but the tasks feel more repetitive, which can lead to fatigue. Also, some people, especially in-house SMEs, might be offended that a complicated task that they performed in the past has been broken down into simpler tasks; they may interpret this situation as implying that they are not sophisticated enough to solve all the steps in one interface. We will return to the topic of user experience in chapter 11. In these cases, you can clarify that the workflow choice was made due to limitations related to getting good training data for machine learning, not because of annotator expertise.

8.6 Further reading

Quality control for annotation is a fast-changing field, and many of the problems we face are unsolved. A good high-level overview is "Truth Is a Lie: Crowd Truth and the Seven Myths of Human Annotation," by Lora Aroyo and Chris Welty (http://mng.bz/NYq7).

For a recent overview specific to the problems related to agreement, I recommend "Let's Agree to Disagree: Fixing Agreement Measures for Crowdsourcing," by Alessandro Checco, Kevin Roitero, Eddy Maddalena, Stefano Mizzaro, and Gianluca Demartini (http://mng.bz/DRqa).

Klaus Krippendorff has published Krippendorff's alpha in several papers and books since it was developed in the 1970s. I recommend "Computing Krippendorff's Alpha-Reliability," which was most recently updated in 2011, but note that it calculates in terms of disagreement, not agreement, as in this book (http://mng.bz/l1lB).

A good recent paper about workflows that back off to experts, with advice about how annotators can explain their decision process effectively to experts, is "Revolt: Collaborative Crowdsourcing for Labeling Machine Learning Datasets," by Joseph Chee Chang, Saleema Amershi, and Ece Semiha Kamar (http://mng.bz/BRqr).

For a good recent study of annotator bias, see "Are We Modeling the Task or the Annotator? An Investigation of Annotator Bias in Natural Language Understanding Datasets," by Mor Geva, Yoav Goldberg, and Jonathan Berant (http://mng.bz/d4Kv).

For a paper showing how diversity among annotators improves accuracy but lowers agreement, see "Broad Twitter Corpus: A Diverse Named Entity Recognition Resource," by Leon Derczynski, Kalina Bontcheva, and Ian Roberts (http://mng.bz/ry4e).

Although not free, *Handbook of Linguistic Annotation*, edited by Nancy Ide and James Pustejovsky, is a comprehensive book that covers a lot of NLP tasks and has a good diversity of use cases. If you don't want to purchase the book, consider emailing the authors of the chapters that are interesting to you; they might share their contributions.

Summary

- Ground truth examples are tasks that have known answers. By creating ground truth examples for the dataset, you can evaluate the accuracy of annotators, create guidelines for those annotators, and better calibrate other quality control techniques.
- You have many ways to calculate agreement in a dataset, including overall agreement, agreement between annotators, agreement between labels, and agreement at task level. Understanding each type of agreement will help you calculate the accuracy of your training and evaluation data and better manage your annotators.
- For any evaluation metric, you should calculate an expected result that would occur by random chance as a baseline. This approach allows you to normalize your accuracy/agreement metric to a score adjusted for random chance, which makes the score more easily comparable across different tasks.
- You will get the best results when using both ground truth data and interannotator agreement, because ground truth agreement allows you to better calibrate your agreement metrics, and agreement metrics can be applied to more annotations than is practical with ground truth alone.
- You can aggregate multiple annotations to create a single label for each task. This approach allows you to create the training data for your machine learning models and calculate the likelihood that each label is correct.
- Quality control by expert review is one common method of resolving disagreements between annotators. Because experts tend to be rare and/or expensive, they can focus mostly on the tough edge cases and the cases that will become part of the guidelines for other annotators.
- Multistep workflows allow you to break an annotation task into simpler tasks that flow into one another. This approach can create annotations faster and more accurately and allow easier-to-implement quality control strategies.

9 Advanced data annotation and augmentation

This chapter covers

- Evaluating annotation quality for subjective tasks
- Optimizing annotation quality control with machine learning
- Treating model predictions as annotations
- Combining embeddings/contextual representations with annotations
- Using search and rule-based systems for data annotation
- Bootstrapping models with lightly supervised machine learning
- Expanding datasets with synthetic data, data creation, and data augmentation
- Incorporating annotation information into machine learning models

For many tasks, simple quality control metrics aren't enough. Imagine that you need to annotate images for labels like "Cyclist" and "Pedestrian." Some images, such as a person pushing a bicycle, are inherently subjective, and an annotator should not be punished for having a valid but minority interpretation. Some annotators will be

more or less familiar with different data items, depending on their familiarity with the locations in the images and whether they themselves are cyclists. Machine learning can help estimate which annotator is expected to be more or less accurate on a given data point. Machine learning can also automate some of the annotation processes by presenting candidate annotations for faster human review. If there are some contexts with few or no cyclists, you might create new data items synthetically to fill the gaps. Knowing that perfect annotation is rare across an entire dataset, you may want to remove some items from the data before building a model on that data or incorporate the uncertainty into the downstream models. You may also want to perform exploratory data analysis on the dataset without necessarily wanting to build a downstream model. This chapter covers methods for addressing all these advanced problems.

9.1 Annotation quality for subjective tasks

There is not always one single, correct annotation for a given task. You may have a task that is inherently subjective; therefore, you expect different responses. We can use our example data from chapter 8, reproduced here in figure 9.1, showing an item that may have multiple correct annotations.

Annotator/ annotations	Alex	Blake	Cameron	Dancer	Evan
Task 1	Pedestrian	Pedestrian	Pedestrian		
Task 2		Sign	Sign	Sign	
Task 3	Pedestrian	Pedestrian	Cyclist	Cyclist	Pedestrian
Task 4		Cyclist	Cyclist	Cyclist	
Task 5	Pedestrian	Pedestrian	Pedestrian	Pedestrian	
Task 6	Cyclist	Cyclist			Cyclist
Task 7	Pedestrian	Pedestrian		Pedestrian	
Task 8	Animal	Animal		Animal	
Task 9	Sign		Animal	Animal	Animal
Task 10		Sign	Sign		Sign
Task 11		Animal			
...					

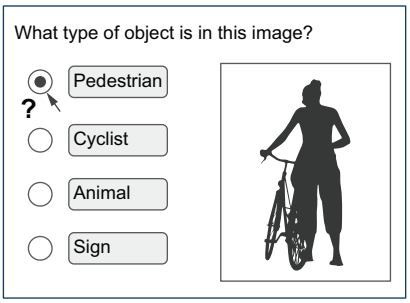


Figure 9.1 A copy of an image from chapter 8, showing how task 3 might have multiple valid interpretations because of the ambiguity between "Pedestrian" and "Cyclist."

There could be multiple reasons why "Pedestrian" or "Cyclist" is favored by one annotator over another, including

- *Actual context*—The person is currently on the road or this image is part of a video in which the person gets on or off the bike.
- *Implied context*—The person looks as though they are getting on or off the bike.

- *Socially influenced variation*—It is likely that local laws treat a person on or off a bicycle differently in different parts of the world. Different laws specify whether bicycles are allowed on a footpath, a road, or a dedicated bike path, and whether people can push a bicycle in any of those places instead of riding it. The laws or common practices with which each annotator is familiar could influence their interpretation.
- *Personal experience*—*We* might expect people who are themselves cyclists to give different answers from people who are not.
- *Personal variation*—Irrespective of social influences and personal experience, two people may have different opinions about the difference between a pedestrian and a cyclist.
- *Linguistic variation*—A cyclist could be strictly interpreted as "anyone who cycles" instead of "someone who is currently cycling," especially if the annotators don't speak English as a first language (common among crowdsourced and outsourced annotators) and the translation of *cyclist* into their first language(s) is not the same definition as in English.
- *Ordering effects*—A person might be primed to interpret this image as a cyclist or pedestrian based on having seen more of one type or the other in the previous annotations.
- *Desire to conform to normality*—A person might themselves think that this image is a cyclist but also think that most other people would call it a pedestrian. They might choose the answer that they don't believe in for fear of being penalized afterward.
- *Perceived power imbalances*—A person who thinks that you are collecting this data to help with safety for cyclists might favor "Cyclist" because they think that you prefer this answer. This kind of accommodation and power imbalance between the annotator and the person who created the task can be important for tasks with obvious negative answers, such as sentiment analysis.
- *Genuine ambiguity*—The photo may be low-resolution or out of focus and not clear.

It may be possible to have detailed guidelines for how our example image should be interpreted, which will mean that there is one objective correct answer. This will not be the case with all datasets, however, and it is often hard to anticipate all the edge cases in advance. So we often want to capture subjective judgments in the best way possible to ensure that we collect the full diversity of possible responses.

In our example in this chapter, we will assume that there is a set of correct answers. For open-ended tasks, this assumption is much harder, and expert review is much more important in these cases. See the following expert anecdote for an example of what can go wrong when you don't take subjectivity into account for open-ended tasks.

In our example dataset, one thing we know from our example image is that "Animal" and "Sign" are not correct answers, so we want an approach to subjective quality control that identifies "Pedestrian" and "Cyclist" as valid answers, but not "Animal" and "Sign."

Annotation bias is no joke

Expert anecdote by Lisa Braden-Harder

Data scientists usually underestimate the effort needed to collect high-quality, highly subjective data. Human agreement for relevance tasks is not easy when you are trying to annotate data without solid ground truth data, and engaging human annotators is successful only with strongly communicated goals, guidelines, and quality control measures, especially important when working across languages and cultures.

I once had a request for Korean knock-knock jokes from a US personal-assistant company expanding into South Korea. The conversation wasn't to explain to the product manager why that wouldn't work and to find culturally appropriate content for their application; it unraveled a lot of assumed knowledge. Even among Korean speakers, the annotators creating and evaluating the jokes needed to be from the same demographics as the intended customers. This case is one example of why the strategies to mitigate bias will touch every part of your data pipeline, from guidelines to compensation strategies that target the most appropriate annotation workforce. Annotation bias is no joke!

Lisa Braden-Harder is a mentor at the Global Social Benefit Institute at Santa Clara University. She was founder and CEO of the Butler Hill Group, one of the largest and most successful annotation companies; and prior to that, she worked as a programmer for IBM and completed computer science degrees at Purdue and NYU.

9.1.1 Requesting annotator expectations

When multiple correct answers exist, the easiest way to understand the possible answers is to ask the annotators directly, and the best way to frame the task is to ask the annotators how they think other annotators might respond. Figure 9.2 shows an example.

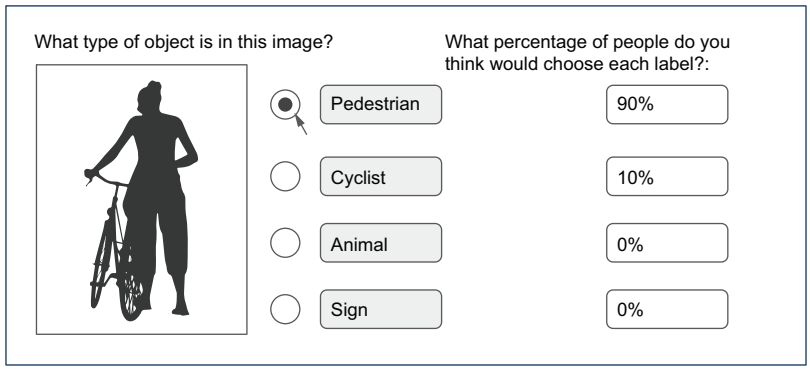


Figure 9.2 Asking people what they expect other annotators to choose for answers. Here, the annotator has indicated that they think that the image is a pedestrian and that 90% of annotators will agree with them, but 10% will think that it is a cyclist. This approach motivates people to give honest responses and provides data to help you decide when multiple responses are valid. In turn, we can capture more diversity in correct answers than the ones offered by any one annotator.

The interface is similar to the example in chapter 8 in which we asked annotators to give their own confidence for each label, but here, we are asking them about other annotators. This relatively simple change has several desirable properties:

- The task design explicitly gives people permission to give an answer that they don't think is the majority answer, which encourages diverse responses and reduces the pressure to conform.
- You can overcome some limitations in annotator diversity. It may not be possible to have an annotator from every demographic that you care about to look at every single item. With this method, you need only annotators who have the right intuition about the full diversity of responses, even if they do not share every interpretation.
- Problems with perceived power dynamics are reduced because you are asking what other annotators think, which makes it easier to report negative responses. This strategy can be a good one when you think that power dynamics or personal biases are influencing responses. Ask what most people would respond instead of what that annotator thinks.
- You can create data that separates valid from nonvalid answers. If we score every person's actual answer as 100% for observed and know that they will divide their expected numbers across multiple labels, they will give less than a 100% score for their expected response for their actual response. So if the actual scores for a label exceed the expected scores, we can trust that label, even if it has low overall percentages of actual and expected.

The last is a lesser-known principle of Bayesian reasoning: people tend to undervalue the probability of their own response. For this reason, we'll look at a popular method called Bayesian Truth Serum in section 9.4.1.

9.1.2 Assessing viable labels for subjective tasks

To start our analysis of viable labels, we can calculate the likelihood that we would have seen each of the labels among the actual annotations, given the number of annotators who have worked on the task. This information will help us decide which labels are valid. If a valid label is expected to occur in only 10% of annotations for a task, but we have only one or two annotators, we wouldn't expect to see an actual annotation for that label.

We calculate the probability that we should have seen each label by using the product of the expected probabilities. Just like when we calculate agreement, we use the complement of the expected annotation percentages. The complement of the expected percentage is calculating the probability that no one annotated a given label, and the probability that at least one person chose that annotation is the complement. Figure 9.3 shows the calculations for our example data.

Figure 9.3 shows that for this task, annotators have selected two labels as being the most likely: "Pedestrian" and "Cyclist" (the same as for our example data) and that people variously believe that "Sign" and "Animal" will be selected by 0% or 5% of people.

	Alex	Blake	Cameron	Dancer	Evan	
Actual annotations	Pedestrian	Pedestrian	Cyclist	Cyclist	Pedestrian	
Expected annotations				**Exists?**		**Actually seen?**
Pedestrian	0.8	0.9	0.35	0.2	0.6	1.000
Sign	0	0	0	0.05	0	0.000
Cyclist	0.2	0.05	0.65	0.7	0.35	1.000
Animal	0	0.05	0	0.05	0.05	0.000
Complement (1 - expected)						**Chance seen (1 - product)**
Pedestrian	0.2	0.1	0.65	0.8	0.4	0.996
Sign	1	1	1	0.95	1	0.050
Cyclist	0.8	0.95	0.35	0.3	0.65	0.948
Animal	1	0.95	1	0.95	0.95	0.143

Figure 9.3 Testing whether a subjective label is viable. Here, the five annotators reported their annotation for the label and what percentage of people they think would choose each label. Blake thinks that the label is "Pedestrian", that 90% of people would choose "Pedestrian," and that 5% each would choose "Cyclist" and "Animal." Taking the product of the complement gives us the probability that we should have encountered this label with this number of annotations, which we can compare with whether we have seen the label.

You can find a copy of the spreadsheet in figure 9.3 and all the other examples in this chapter at http://mng.bz/Vd4W.

First, let's imagine that no one chose "Pedestrian" as the actual annotation, but people still gave some weight to "Pedestrian" in their expected score. Here are the calculations from figure 9.3:

Expected: [0.8, 0.9, 0.35, 0.2, 0.6]
Not Expected: [0.2, 0.1, 0.65, 0.8, 0.4]
Product of Not Expected = 0.004
Probability Seen = 1 – 0.004 = 0.996

With those expected scores, we are 99.6% certain that we should have seen at least one actual "Pedestrian." So we could be fairly certain that this result was an error in the annotators' perception. When there is a high probability that a label will be seen, according to the expected annotations, but has not been seen, we can more confidently rule it out as a viable label.

Now let's look at one of the less expected labels in figure 9.3: "Animal." Although three annotators believe that some people will annotate the image as "Animal," there is a 14.3% chance that one of the five annotators will have chosen "Animal." The fact that no one has chosen "Animal" yet doesn't necessarily rule it out. We wouldn't have expected to see someone choose "Animal" with only five annotators, if we trust these numbers, and wouldn't expect to see one until about 20 annotators have seen this item. We can take several approaches can to discover whether "Animal" is a viable label, each with increasing complexity:

- Add more annotators until "Animal" is seen or the probability seen is so high that we can rule out "Animal" as a viable label.
- Trust an expert annotator to decide whether "Animal" is a viable label when that expert annotator is experienced at putting personal biases aside.
- Find annotators who correctly annotated items as "Animal" in the ground truth data when that annotation was rare but correct and give this task to them (a programmatic way to find the best nonexpert).

Although the first option is easiest to implement, it works only if you are confident in the diversity of your annotators. There may be people who would correctly choose "Animal," but they are not among your annotators, so that situation never arises. On the other hand, it might be objectively incorrect to choose "Animal," but this example is a tough one, and 5% of people would be expected to get it wrong. You probably don't want to select "Animal" in this case.

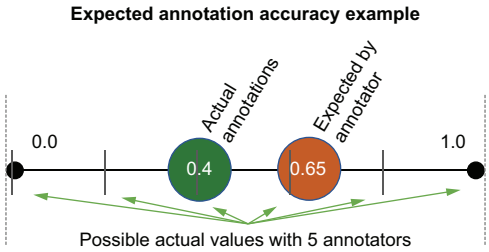
Therefore, when there is ambiguity about whether a label is valid for a subjective task, you will want to find another annotator (possibly an expert) who can be trusted to understand the diversity of possible responses.

9.1.3 Trusting an annotator to understand diverse responses

We can calculate our trust in an individual annotator's expected annotations by looking at the difference between their expected annotations and the actual annotations calculated across all annotators. The basic concept is simple. If an annotator expected a 50:50 split of annotations between two labels and was correct that there was a 50:50 split, that annotator should get a score of 100% for that task.

If there were an odd number of annotators, a 50:50 split wouldn't be possible, so we need to take into account the possible precision given the finite number of annotators. Figure 9.4 illustrates a slightly more complicated example.

In figure 9.4, the annotator overestimated the number of annotators by 0.25. Every value between 0.15 and 0.65 is closer to the actual number of 0.4, and 0.65 – 0.15 = 0.5. So 50% of the possible expected values are closer to 0.4. Given enough annotators, however, the true actual value would be higher than 0.4, so we adjust by the minimum precision of 0.2, which gives us 0.5 * (1 – 0.2) + 0.2 = 0.6. The annotator's accuracy score is 60%.



Annotator accuracy: The fraction of values that are further from 0.4 than 0.65, taking into account the annotator didn't know that only 0.2 increments were possible because there were 5 annotators

Figure 9.4 The accuracy of one annotator to estimate the range of responses across all annotators by comparing the actual fraction of annotations for a given label with the number of annotations expected by an annotator. For our example data, this would correspond to Cameron's expectation that 65% of people would choose "Cyclist" for this task, compared with the 40% of people who actually chose it.

Figure 9.5 gives the calculation for every estimate by every annotator in our example data. To get the overall accuracy for an annotator, you average their accuracy across every subjective task in the dataset.

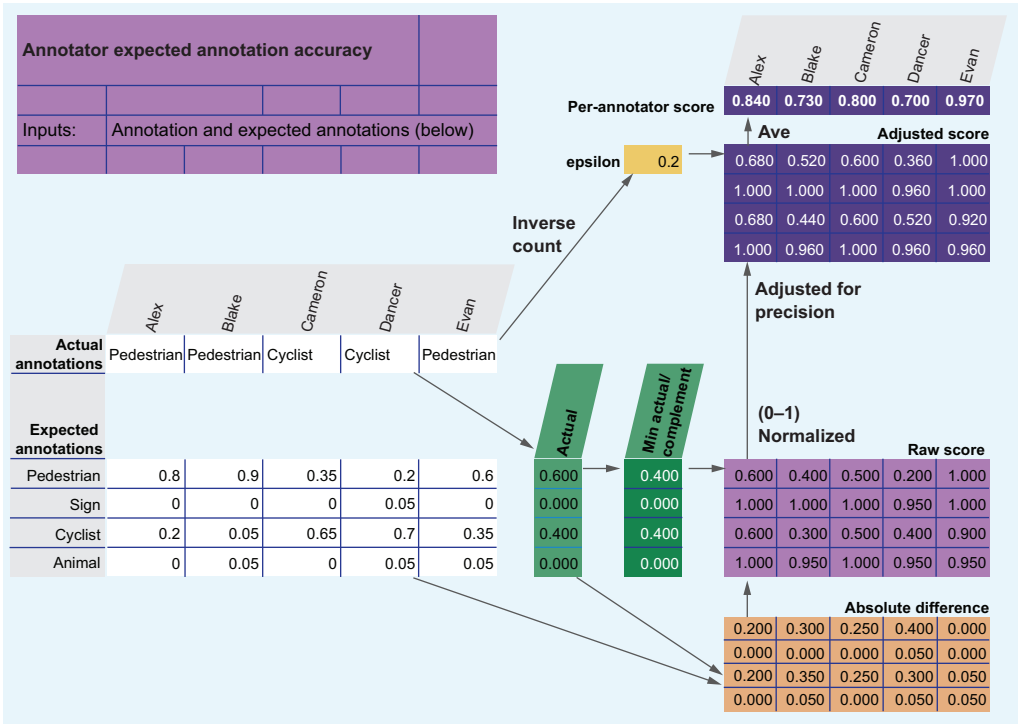


Figure 9.5 Calculating the accuracy of each annotator's estimate as the adjusted score and then averaging those scores to get a per-annotator score for this task. Cameron is 80% accurate in estimating how close the expected distribution was to the actual distribution. Evan is the most accurate, with a score of 97%, and Blake is the least accurate, with a score of 73%.

In figure 9.5, epsilon is the same epsilon used in Krippendorff's alpha in chapter 8. It wasn't important then, because Krippendorff's alpha calculated epsilon over the total number of annotations in the dataset. Here, we are calculating epsilon over the annotations within a single task. You can see by comparing the raw score and adjusted score that epsilon makes a big difference, adjusting the results by 20%.

You can use several variations and extensions if it is especially important to know how accurately your annotators can estimate the actual distributions. A 0 score won't be possible for some tasks because the distribution of each annotator's expected annotations has to add up to 1; therefore, they can't always provide the worst estimate for every label. (In figure 9.5, the worst possible score is 0.44 if an annotator expected that only "Animal" or "Sign" would be chosen.) You could normalize for this baseline, as for ground truth accuracy and agreement in chapter 8.

Cross-entropy is another way to calculate the difference between the expected and the actual distributions. Although cross-entropy is a common way to compare probability distributions in machine learning, I have never seen it used to compare actual and expected annotations for training data. This technique would be an interesting area of research.

9.1.4 Bayesian Truth Serum for subjective judgments

The method in section 9.1.3 focused on how accurately each annotator predicted the frequency of different subjective judgments, but the scores did not take into account the actual annotation from each annotator—only their expected scores. Bayesian Truth Serum (BTS) is a method that combines the two approaches. BTS was created by Dražen Prelec at MIT (see the *Science* paper in section 9.9.1) and was the first metric to combine the actual and expected annotations into a single score.

BTS calculates the score from an information-theoretic point of view. This score does not allow you to interpret the accuracy of an annotator or label directly. Therefore, BTS looks for responses that are more common than collectively predicted by the same annotators, which will not necessarily be the most frequent responses. Figure 9.6 shows an example.

In figure 9.6, Cameron has the highest score from a BTS point of view, primarily because there is high information from choosing "Cyclist" as the actual annotation. That is, the actual annotation frequency for "Cyclist" was higher than the expected frequency compared with "Pedestrian." Blake has the lowest score, primarily because of the prediction that 0.9 of annotations would be "Pedestrian" when only 0.6 were—the largest error of all the predictions. So our dataset in this section is a good example of a case in which the less-frequent subjective label provided more information than the more-frequent label. In some cases, however, the highest-frequency actual label can provide the most information.

Figure 9.6 is also a good example of how information differs from accuracy. Recall that in figure 9.5, Evan had the highest score because Evan's expected annotation frequencies were closest to the actual annotation frequencies. For BTS, Cameron ended

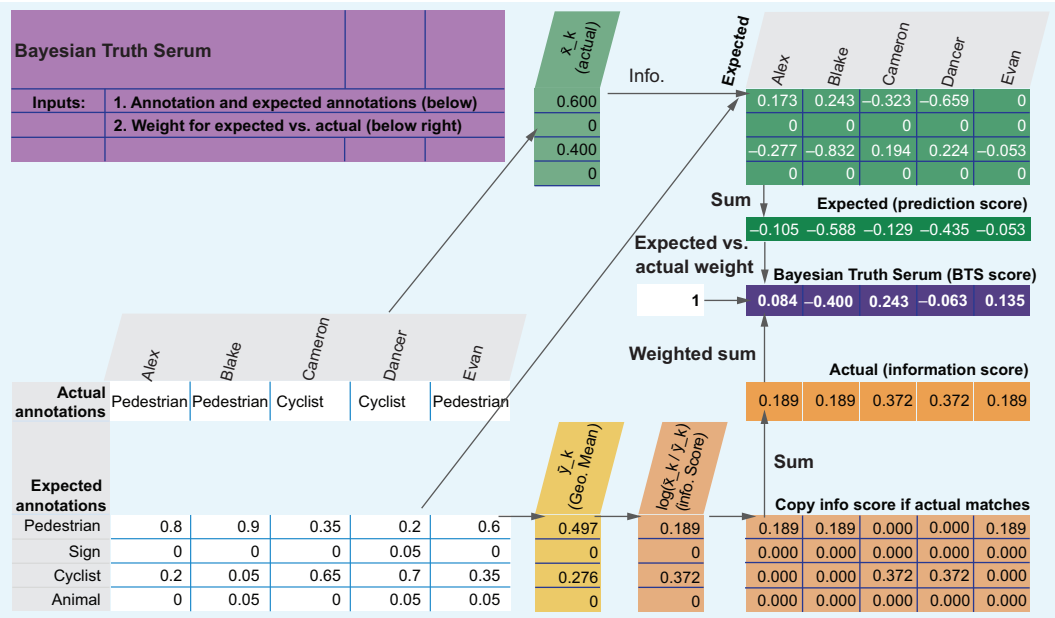


Figure 9.6 BTS combines the person's actual annotation with the predictions for expected annotations into a single score. Info is the information theoretic score (Expected × log(Actual / Expected)). The scores for each annotator show Cameron with the highest score. For both the expected and the actual annotations, the scores are based on information theory. The score is not only about the accuracy of each annotator; it is also about how much information each annotator is providing.

up with the highest score because even though Cameron was less accurate than Evan, there was more value in Cameron's predictions about "Cyclist," the less-frequent label that might have been overlooked.

If you consistently find that an annotator with the highest information score via BTS is not the annotator with the highest accuracy in predicting expected annotation frequencies, this finding can be evidence of a lack of diversity in your annotators. Check whether the annotator with the highest BTS score is typically selecting the less-frequent label; if so, you have evidence that your annotator pool is choosing the most-frequent label more often than a random or representative population would.

In an interesting extension to BTS, the inventors observed that when the actual percentage of annotations exceeds the average expected percentage for a label, this finding is good evidence that the surprisingly popular label is the correct one, even if it is not the majority. But this result relies on having enough annotators that at least one annotator gets chooses that label, which is unlikely for rare but valid labels when you have only a few annotators per task.

Note that we are not adjusting the BTS score in figure 9.6 for the fact that there are only five annotators, so only multiples of 0.2 were possible (epsilon in figure 9.5). The example in this section is the original calculation for BTS, so for educational purposes, it is taught here as it appears in the literature. It would be fine to add this

adjustment, but note that BTS has a nice symmetry that you will lose in that case; if the weight of expected and actual scores is set to 1, as in our example (equal weights), the BTS scores always add to 0. This will not be the case if you adjust for precision, so you won't be able to take advantage of this symmetry with this modification. See section 9.9 for more information about extensions to BTS.

9.1.5 Embedding simple tasks in more complicated ones

If none of the previous techniques for subjective data works, one simple solution is to create an additional question for your task that is not subjective and assume that if an annotator gets that response correct, their subjective label is also valid. Figure 9.7 shows an example.

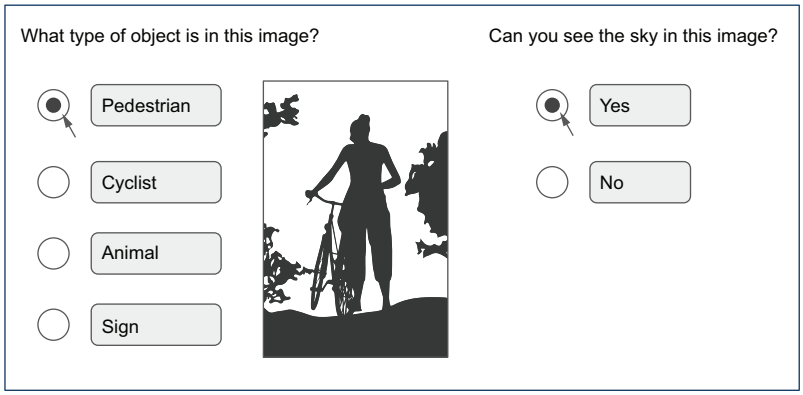


Figure 9.7 A subjective task with an additional question that is objective. This example allows easier quality control by assuming that if a person gets the objective question correct, their subjective judgment is also correct and not an error.

In figure 9.7, we are asking an additional question about whether the sky can be seen in the message. Unlike the object type, this question should be unambiguous and objective: the sky is either visible or not. Therefore, we can easily test whether people are getting the product question correct by embedding known answers for some questions and/or by looking for agreement between annotators, using the techniques discussed in this chapter. Then we assume that the people are equally accurate for the subjective task.

When using this method, we rely on the assumption that accuracy for the simpler objective task will strongly correlate with accuracy for the subjective task, which will be more or less true depending on your data. As a general principle, the closer the question is to the relevant content, the closer this correlation should be. In our example, we are asking about the context of the object, so the accuracy should be highly correlated.

This approach is most effective when the actual task is time-consuming. If you were asking someone to type a summary of a large passage, which typically would take many minutes, there is little additional annotation cost to ask an additional objective question about the passage.

9.2 Machine learning for annotation quality control

Because most quality control strategies for data annotations are statistically driven decision processes, machine learning can be used for the quality control process itself. In fact, most of the heuristics in this chapter and chapter 8 can be modeled as machine learning problems that are trained on held-out data. Four types of machine learning-driven quality controls are introduced here, all of which use the annotator's performance on ground truth data and/or agreement as training data:

- Treating the model predictions as an optimization task. Using the annotator's performance on the ground truth data, find a probability distribution for the actual label that optimizes a loss function.
- Creating a model that predicts whether a single annotation by an annotator is correct or incorrect.
- Creating a model that predicts whether a single annotation by an annotator is likely to be in agreement with other annotators.
- Predicting whether an annotator is actually a bot.

Some methods can be used independently or in combination. The following sections cover these methods in turn.

9.2.1 Calculating annotation confidence as an optimization task

In chapter 8, you learned that you can take the average confidence across all labels. If the confidence in one annotator's annotation was less than 100%, the remaining confidence was spread across the labels that the annotator didn't choose. We can build on this approach by looking at all the annotators' annotation patterns on the ground truth data and then treat our confidence as an optimization problem. Figure 9.8 shows an example.

Converge label confidence

Inputs: 1. Actual Annotation
2. Fraction in Ground-Truth for Annotation

	Alex	*Blake*	*Cameron*	*Dancer*	*Evan*
Actual Annotations	Pedestrian	Pedestrian	Cyclist	Cyclist	Pedestrian
Ground-Truth Fraction					
Pedestrian	0.91	0.93	0.28	0.72	0.58
Sign	0.01	0	0.04	0.07	0.01
Cyclist	0.04	0.05	0.67	0.21	0.39
Animal	0.04	0.02	0.01	0	0.02

Figure 9.8 Using performance on the ground truth data to calculate model confidence as an optimization task. On the ground truth data, when Alex annotated items as "Pedestrian," they were actually "Pedestrians" 91% of the time, "Signs" 1% of the time, "Cyclists" 4% of the time, and "Animals" 4% of the time. When we see that Alex has annotated some new item as "Pedestrian," we can assume the same probability distribution. When Dancer annotates an item as "Cyclist," we know that it is actually a "Pedestrian" 72% of the time, showing confusion about these categories.

Figure 9.8 shows the actual distribution of the annotations in the ground truth data. If you have a small amount of ground truth data, you might consider smoothing this number with a simple smoothing method such as adding a constant (Laplace smoothing).

A nice property of this approach, compared with the methods in chapter 8, is that you might not have to discard all annotations from a low accuracy annotator. Dancer is wrong most of the time in figure 9.8 because they are only correct 21% of the time when they annotate an item as "Cyclist." There is useful information, however, in the fact that "Pedestrian" was the correct answer 72% of the time. So instead of removing Dancer from our annotations because of poor accuracy, we can keep their annotations and let them contribute to our overall confidence by modeling their accuracy.

To calculate the overall confidence, you can take the average of these numbers, which would give 68.4% confidence in "Pedestrian," 2.6% confidence in "Sign," 27.2% confidence in "Cyclist," and 1.8% confidence in "Animal." The average is only one way to calculate the overall confidence, however. You can also treat this task as an optimization task and find the probability distribution that minimizes a distance function, such as mean absolute error, mean squared error, or cross-entropy. If you come from a machine learning background, you will recognize these methods as loss functions, and you can think of this problem as a machine learning problem: you are optimizing for the least loss by finding a probability distribution that best matches the data.

If you try different loss functions on our example data, you will find that they don't differ much from the average. The biggest benefits from making this problem a machine learning problem is that you can incorporate information other than the annotations themselves into your confidence prediction.

9.2.2 Converging on label confidence when annotators disagree

Building on the treatment of aggregation as a machine learning problem, we can use the ground truth data as training data. That is, instead of optimizing the probability distributions that are taken from the ground truth data, we can build a model that uses the ground truth data as labels. Figure 9.9 shows how the ground truth data example from chapter 8 can be expanded to show the feature representation for each ground truth item.

If we build a model with the data in figure 9.9, our model will learn to trust our annotators relative to their overall accuracy on the ground truth data. We don't explicitly tell the model that the annotations have the same values as the labels; the model discovers the correlations itself.

The biggest shortcoming of this method is that people who have annotated more ground truth data will be weighted higher because their features (annotations) have appeared in more training data. You can avoid this result by having most of the ground truth data annotated early in the annotation process (a good idea in any case to determine accuracy and to fine-tune other processes) and by sampling an equal number of annotations per annotator in each training epoch when you build your

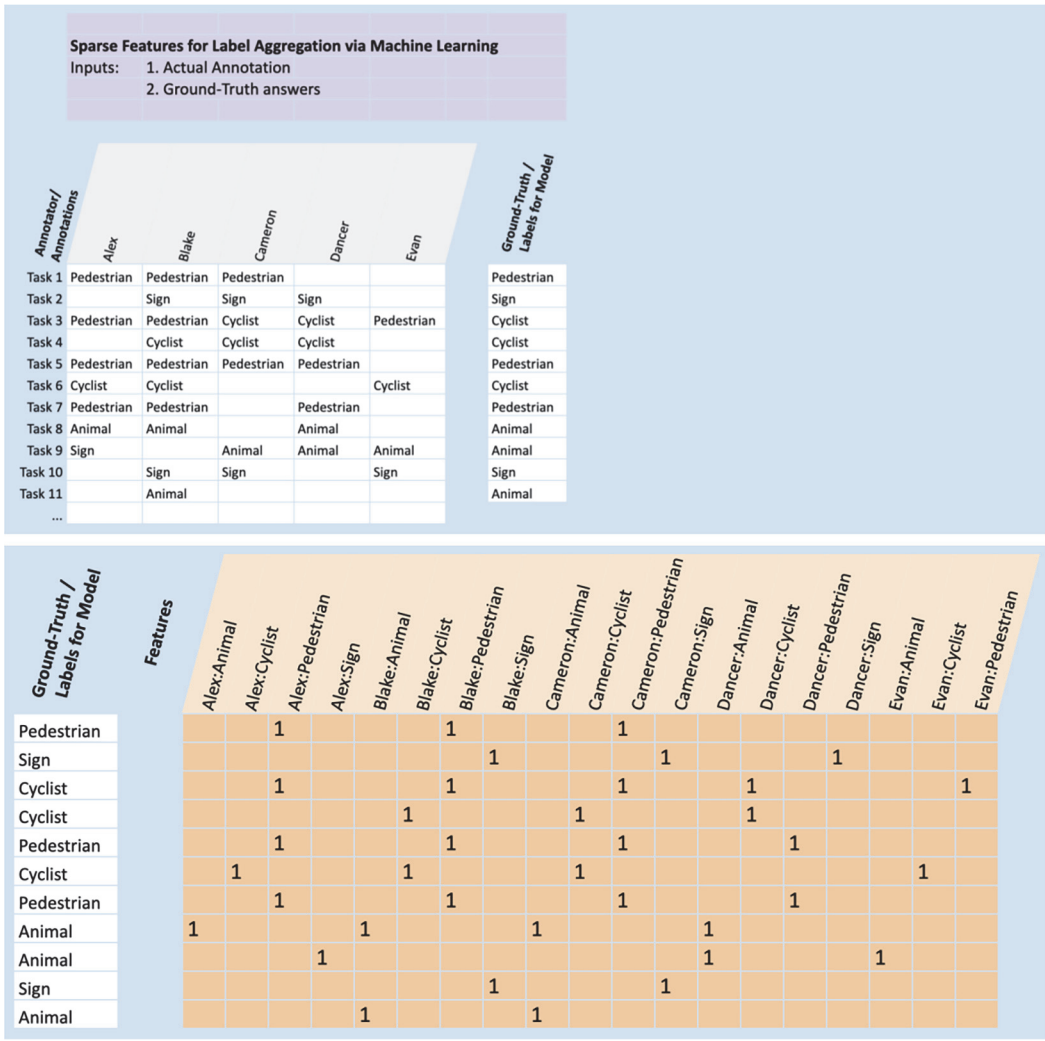


Figure 9.9 Sparse feature representation, with ground truth data as training data. We can take every annotation on the ground truth dataset, and use the actual annotations as features and the ground truth labels as the labels for a machine learning model. Then we have a model that can predict the correct label and give a confidence associated with that prediction.

model. You can also overcome this shortcoming by aggregating the number of labels but ignoring who did the annotations, as shown in figure 9.10.

You may need to normalize the entries in figure 9.10 if your model expects a [0–1] range in feature values. For both the sparse representation in figure 9.9 and the aggregated information in figure 9.10, you can experiment with using your confidence in each prediction instead of counting each annotation as 1. This confidence score could be the self-reported confidence of the annotator, as demonstrated in chapter 8, or an expected distribution, as in the subjective judgments in section 9.1. You might

Ground-Truth / Labels for Model	Features: Animal	Cyclist	Pedestrian	Sign
Pedestrian	0	0	3	0
Sign	0	0	0	3
Cyclist	0	2	3	0
Cyclist	0	3	0	0
Pedestrian	0	0	4	0
Cyclist	0	3	0	0
Pedestrian	0	0	3	0
Animal	3	0	0	0
Animal	3	0	0	1
Sign	0	0	0	3
Animal	1	0	0	0

Figure 9.10 Dense (aggregate) feature representation, with ground truth data as training data. The features are the count of each label, thereby ignoring the identity of the annotator. We can take every annotation on the ground truth dataset, count each annotation as the features, and use the ground truth labels as the labels for a machine learning model. This example is more robust than the one shown in figure 9.9 when you don't have many ground truth labels for many of your annotators.

also have a confidence metric for each annotator that is derived from their past work. Whatever number you experiment with, make sure that it's not derived from the same ground truth data that you are about to train on, which would overfit your quality prediction model.

As with the sparse example, a single neuron or linear model should be enough to give reliable results for figure 9.10 and not overfit the data for the dense representation. You should start with a simpler model before experimenting with anything more complicated in any case.

At this point, you may be wondering why you can't include both the sparse and aggregate information as features in a model. You can do this! You can create a model that uses these features plus any others that may be relevant for calculating how confidently we can aggregate multiple annotations. But even if you decide to take the "throw everything into a model" kitchen-sink approach to aggregation, you should use the feature representations in figures 9.9 and 9.10 as a baseline before you start experimenting with more complicated models and hyperparameter tuning.

To evaluate how accurate this model is, you need to split your ground truth data into training and evaluation data so that you can evaluate the confidence on held-out data. If you are using anything more complicated than a linear model or single neuron, such that you are doing hyperparameter tuning, you also need a further split to create a validation set that you can use for tuning. Both the sparse and aggregate representations are compatible with using a model's predictions as though it were an annotator. For the aggregate representation, you might think about whether you want to aggregate model predictions separately from human annotations.

9.2.3 Predicting whether a single annotation is correct

The most flexible way to use machine learning for quality control in annotation is as a binary classifier to predict whether an individual annotation was correct. The advantage of a simple classification binary task is that you can train a model on relatively little data. If you are training on ground truth data, you are unlikely to have much data to train on, so this approach allows you to get the most out of the limited data you have.

This method is especially useful if you have a small number of annotators per item. You may have the budget for only a single annotator to look at most of your items, especially if annotator is a subject-matter expert (SME) who is reliable most of the time. In this context, you want to identify the small number of cases in which the SME might be wrong, but you don't have agreement information to help with this identification because you have only one annotation most of the time.

The simplest implementation to start with would include the annotator's identity and their annotation as the features, like in figure 9.9. Therefore, this model will tell you which annotators are the strongest or weakest on particular labels in the ground truth data.

You can think about the additional features that might provide additional context as to whether an annotator might make an error. The features that you might try in the model, in addition to annotator identity and annotation, could include

- The number or percentage of annotators who agree with that annotation (where they exist)
- Metadata about the item being annotated (time, place, and other categories) and the annotator (relevant demographics, qualifications, experience on this task, and so on)
- Embeddings from the predictive model or other models

The metadata features can help your model identify areas where there might be biases or meaningful trends in annotation quality. If a metadata feature captured the time of day when a photo was taken, your model might be able to learn that photos taken at night are generally harder to annotate accurately. The same is true for your annotators. If your annotators are themselves cyclists, they might have biases about images that contain cyclists, and the model can learn about this bias.

This approach works with subjective data too. If you have subjective data with multiple correct answers, each of those correct ones could be correct for the binary model. This technique is fairly flexible; it also works for many types of machine learning problems, as covered in chapter 10.

Showing the correct ground truth answers to annotators

You have the option of showing the correct answer to an annotator when they get it wrong. This review should improve that annotator's performance, but it will also make it harder to evaluate that annotator's accuracy. There is a trade-off in design: do you tell the annotator every time they make an error, making that annotator more accurate, or do you keep some or all ground truth items anonymous so that you can perform better quality control on that annotator's performance? You may need to strike a balance.

(continued)
For models built on ground truth data, be careful using items for which the annotator has learned the correct answer. For example, an annotator might have made errors with ground truth data items that had a person pushing a bike. If the annotator was told about that error and given the correct answer, however, that annotator is less likely to make that same error later. Therefore, your quality control model might erroneously predict errors for that annotator on types of items for which they are now highly accurate.

9.2.4 Predicting whether a single annotation is in agreement

As an alternative to predicting whether the annotator is correct, you could predict whether the annotator agrees with other annotators. This approach can increase the number of training items, because you can train a model to predict agreement on all the items that have been annotated by multiple people, not only the ones in the ground truth data. This model is likely to be more powerful.

Predicting agreement can be useful to surface items for which disagreement was expected but didn't occur. Perhaps it was by random chance that a small number of annotators agreed with one another. If you can predict confidently that disagreement should have occurred, even by annotators who didn't work on that task, that finding can be evidence that additional annotation may be needed for that item.

You can try both approaches: build one model that predicts when an annotator is correct, and build a separate model that predicts when an annotator will agree with other annotators. Then you can review the task or elicit additional annotations when an annotation is predicted to be an error or predicted to disagree with other annotations.

9.2.5 Predicting whether an annotator is a bot

If you are working with anonymous annotators and discover that one annotator was really a bot that was scamming your work, you can make a binary classification task to identify other bots. If we discovered that Dancer in our annotation data is a bot, we might suspect that the same bot is posing as other human annotators.

If you are certain that some subset of annotators are human, their annotations can become human training data for your model. This approach effectively allows you to train a model to ask an annotator, "Are we human, or are we Dancer?"

Sometimes, a bot is a good addition to the annotation team. Machine learning models can annotate the data or create data autonomously or in combination with humans. The rest of this chapter is devoted to methods that automate or semi-automate data annotation.

9.3 Model predictions as annotations

The simplest approach to semi-automating annotation is to treat the model's predictions as though the model was an annotator. This process is often called *semi-supervised learning*, although that term has been applied to pretty much any combination of supervised and unsupervised learning.

You can trust a model's predictions or incorporate the model's predictions as one annotator among many. The two approaches have different implications for how you should treat model confidence and the workflows that you might implement to review the model's output, so they are explored separately. You can also use your model predictions to look for potential errors in noisy data, which is covered in section 9.3.3.

Will we replace human annotators?

Every few years since the 1990s, someone has claimed to have solved automated labeling. Thirty years later, however, we still need to label data for more than 99% of supervised machine learning problems.

There are two common problems with many academic papers about automated labeling, whether it is using model confidence, a rule-based system, or some other method. First, they almost always compare the auto-labeling methods with random sampling. As you saw in chapter 2, even a simple active learning system can quickly improve the accuracy of your model, so it can be difficult to evaluate the benefit compared with active learning from these papers. Second, the papers typically assume that the evaluation data already exists, which is true for academic datasets. In the real world, however, you still need to set up annotation processes to create your evaluation data, manage the annotators, create the annotation guidelines, and implement quality control on the annotations. If you are doing all this for your evaluation data, why not put in the extra effort in the annotation component to create training data too?

The reality is rarely an all-or-nothing solution. Although we can't remove human annotators from the majority of supervised machine learning systems, we have some exciting ways to improve our models and annotation strategies, such as using model predictions as labels, embeddings and contextual representations, rule-based systems, semi-supervised machine learning, lightly supervised machine learning, and synthetic data. All these techniques have interesting human-in-the-loop implications and are introduced in this chapter.

9.3.1 Trusting annotations from confident model predictions

The simplest way to use a model as an annotator is to trust the model predictions as labels, trusting predictions beyond a certain confidence threshold as labels. Figure 9.11 shows an example.

Figure 9.11 shows how items are labeled automatically by a predictive model. We can bootstrap our model from that starting point. This approach is suitable if you have an existing model but don't have access to the data that the model is trained on. This situation is common in machine translation. Google released the first major machine translation system, and every major machine translation system since then has used translated data from Google's engine. Although this approach is less accurate than annotating data directly, it can be effective for getting a quick start cheaply.

This kind of semi-supervised learning, sometimes known as *bootstrapped semi-supervised learning*, rarely works in isolation when adapting an existing model to new types of data. If you can confidently classify something correctly, your model gains little extra information with the additional items that it is already confident about, and you run the risk of amplifying bias. If something is truly novel, the model is probably

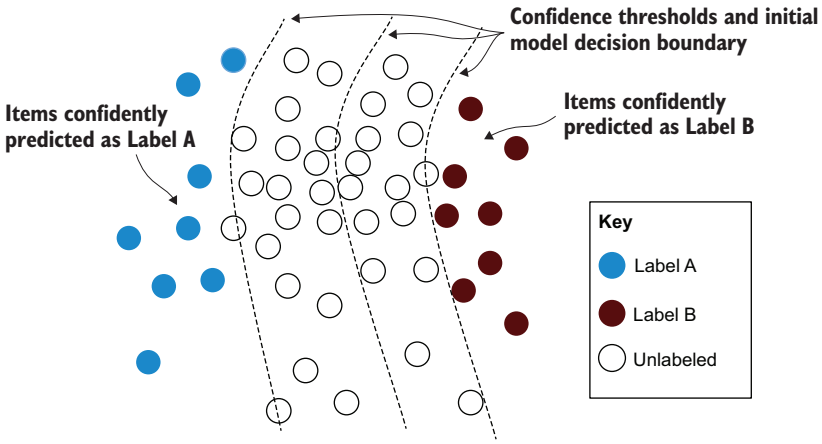


Figure 9.11 Treating the most confident predictions as labels. A model predicts the items as being Label A or Label B, and the most confidently predicted items are treated as the correct label. This example allows us to build a model quickly but has a shortcoming: the model is built from items *away* from the decision boundary, which leaves a large amount of error for where that boundary might be.

not classifying it confidently or (worse) could be misclassifying it. This approach, however, can be effective when used in combination with active learning techniques to ensure that there is enough representative data. Figure 9.12 shows a typical workflow for trusting model predictions as annotations.

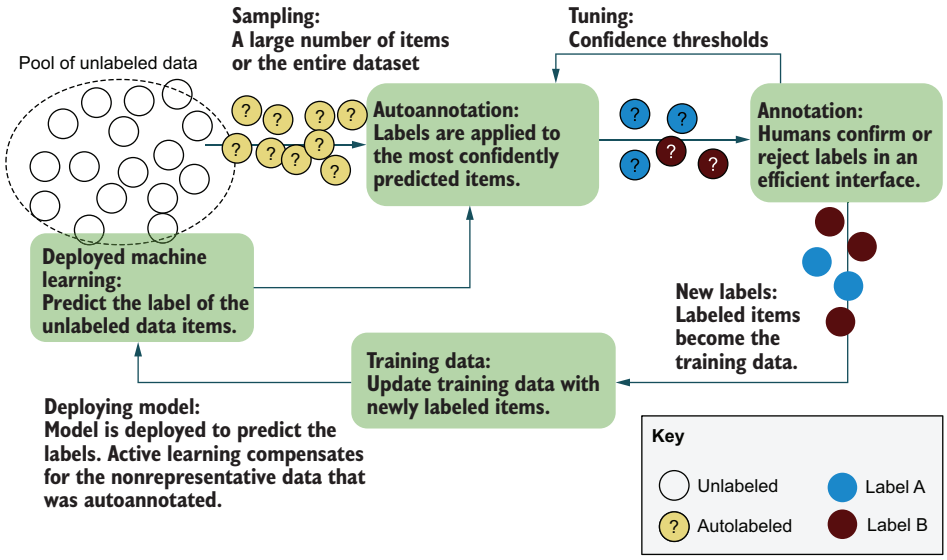


Figure 9.12 A workflow for using confident predictions as annotations. The model is used to predict the labels of a large number of unlabeled items (potentially all of them). Human annotators review some of the labels, and the accepted labels become annotations for the training data. The human annotators also use this process to tune the threshold at which the labels can be confidently turned into annotations.

Here are some tips for using confident model predictions to create annotations:

- Margin-of-Confidence and Ratio-of-Confidence are likely to be the best measure of confidence because you want the highest confidence relative to the other labels. So these metrics are good starting points, but you can test the other uncertainty sampling metrics to see what is best for your data.
- Set a confidence threshold on a per-label basis, or sample the top *N* predictions for each label, rather than try to set one confidence threshold for all labels. Otherwise, your most confident predictions are likely to be from a small number of easy-to-predict labels.
- Keep two models trained in each iteration: one trained on all annotations and one trained only on the annotations that a human has seen. Do not trust the predictions when confidence is high for the first model but low for the second.
- Keep track of human-labeled and auto-labeled items, and ensure that a certain number of training epochs use only the human-labeled items, to keep your model from straying too far. (This strategy is often called *pseudo-labeling*.)
- Use uncertainty sampling in your next iteration of active learning to focus on your new decision boundary.
- Use representative sampling to find data that was different from what the prior model was trained on (See section 7.5.4 for more about using representative sampling when combining human and machine labels.)

Using model predictions to generate candidates for human review, instead of trusting them fully, can be effective if the annotation task is time-consuming. If we had a classification task with hundreds of labels, it would be much faster for an annotator to accept or reject the predicted label as a binary classification task than to choose among the hundreds of labels manually. This scenario tends to be more true of other types of machine learning, such as sequence labeling and semantic segmentation, than of labeling. Chapter 10 goes into more detail on using model predictions for these use cases.

Review workflows like in figure 9.12 can lead to bias where humans trust the model too much, perpetuating and sometimes amplifying the errors. We will cover ways to mitigate these errors in chapter 11 when we discuss user experience and annotation interfaces.

9.3.2 Treating model predictions as a single annotator

A second way to incorporate machine learning into the annotation process is to include the predictions from your downstream model as though they were annotations by one annotator. Suppose that the annotator Evan in our examples is not human; it's our downstream machine learning model. Looking at figure 9.13, we can see that Evan is reasonably accurate, getting every label correct except for task 3, where Evan incorrectly predicted "Cyclist" to be "Pedestrian." Therefore, if we add Evan's predictions as though Evan were a human annotator, we can apply the exact same methods to converge on the right agreement.

	Alex	Blake	Cameron	Dancer	Model Predictions (Evan)
Actual Annotations	Pedestrian	Pedestrian	Cyclist	Cyclist	Pedestrian
Ground-Truth Fraction					
Pedestrian	0.91	0.93	0.28	0.72	0.58
Sign	0.01	0	0.04	0.07	0.01
Cyclist	0.04	0.05	0.67	0.21	0.39
Animal	0.04	0.02	0.01	0	0.02

Figure 9.13 Incorporating predictions from a model as though they were annotations. From our example data, we can assume that Evan was in fact a predictive model, not a human annotator. For any of our methods that take into account the accuracy of each annotator, it is generally fine to incorporate model predictions as human annotations in this part of the workflow.

You can incorporate a model's prediction as you would the annotations of any other annotator. By applying the techniques from section 9.2.1, where we took the annotator's accuracy into account when calculating our final probability distribution, we are using the accuracy of the model on the ground truth data.

You might consider different workflows, depending on how an item was sampled to be annotated. If you consider that Evan was trained by past human interactions and acted on that knowledge, Evan will be shaped by the past interactions and training data and will mirror those human behaviors unless Evan becomes adversarial to humans.

Therefore, if an item was sampled that is similar to past training data and was confidently classified by Evan, you might ask one more annotator to confirm that annotation instead of the minimum number of annotators that you would otherwise use. This approach falls between our strategies of trusting confident predictions and treating the model as an annotator.

9.3.3 Cross-validating to find mislabeled data

If you have an existing annotated dataset and are not certain that all the labels are correct, you can use the model to find candidates for human review. When your model predicts a different label from the ones that have been annotated, you have good evidence that the label might be wrong and that a human annotator should review that label.

If you are looking at your existing dataset, however, your model should not be trained on the same data that it is evaluating, because your model will overfit that data and likely miss many cases. If you cross-validate, such as splitting your data into 10 partitions with 90% of the data as training data and 10% as evaluation data, you can train and predict on different data.

Although there is a large body of literature on training models on noisy data, most of it assumes that it is not possible for humans to review or correct the wrongly labeled data. At the same time, the literature assumes that it is possible to spend a lot of time tuning models to automatically identify and account for noisy data (see graduate-student economics from chapter 7). In almost all real-world use cases, you should be able to annotate more data. If you know that your data is noisy, you should at least set up an annotation process for your evaluation data so that you know your actual accuracy.

There are some legitimate reasons why you might have noisy data that you can't avoid. The data might be inherently ambiguous, you might get a large amount of free but noisy labels, or you might have an annotation interface that sacrifices a little accuracy for much greater throughput. We'll return to ways to account for noisy data later, with the caveat that it is better to have accurate training data in almost all use cases.

9.4 Embeddings and contextual representations

Much current machine learning research focuses on transfer learning: adapting a model from one task to another. This technique opens some interesting possibilities for annotation strategies. If your annotation task is especially time-consuming, such as semantic segmentation, you may be able to annotate orders of magnitude more data in some other way and then use that data in a model that is adapted to the semantic segmentation task. We'll return to this specific example later in this section.

Because transfer learning is a popular research area right now, there is a lot of changing terminology. If a model is specifically built to be adapted to new tasks, it is often called a *pretrained* model, and the information in that model is referred to as an *embedding* or *contextual representation*. Figure 9.14 shows a general architecture for using contextual embeddings.

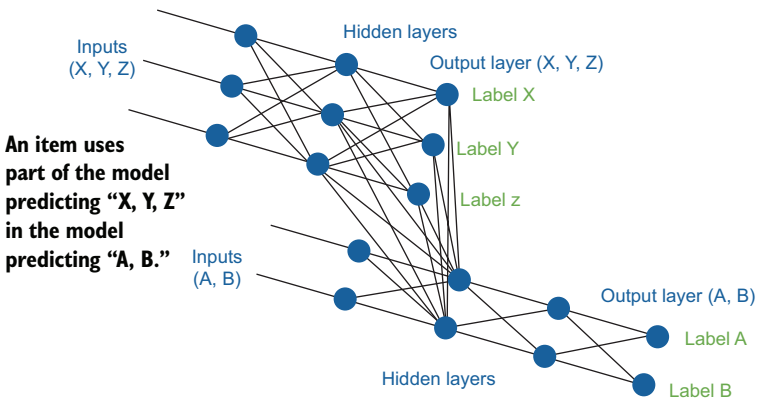


Figure 9.14 An example of transfer learning. We have the task of predicting whether an item is "A" or "B," and we think that our existing model that predicts "X," "Y," or "Z" will have useful information because of similarities between the two tasks. So we can use the neurons from the "X," "Y," or "Z" model as features (a *representation*) in our model predicting "A" or "B." This example is similar to the examples earlier in the book that use the hidden layers as features for clustering and use transfer learning to adapt an existing model to a new task. In some cases, we might ignore the inputs "A" and "B," using only the pretrained model as a representation for our new model.